Uttar Pradesh Rajarshi Tondon
open University

Master of Computer Science

MCS-111
Design and Analysis of Algorithm

Design and Analysis of Algorithm

Uttar Pradesh Rajarshi Tondon open University

Master of Computer Science

MCS-111
Design and Analysis of Algorithm

Block 1

Introduction and Design Strategies - I

ISBN :

Reprinted and Published by Vinay Kumar, Registrar Uttar Pradesh Rajarshi Tondon Open University, 2024

Printed By: Chandrakala Universal Pvt. Ltd. 42/7 Jawahar Lal Neharu Road, Prayagraj. 211002.

Block-I Introduction and Design Strategies-I

This is the first block on introduction and design strategies. In this block we mainly focused on introduction of algorithm, divide and conquer and sorting in linear time.

So we will begin the first unit on introduction to **algorithm**. An algorithm is a set of instructions designed to perform a specific task. Examples of image processing algorithms include cropping, resizing, sharpening, blurring, red-eye reduction, and color enhancement. In many cases, there are multiple ways to perform a specific operation within a software program.

In the first unit we also describe **psuedo code** for expressing algorithms. If we talk about pseudo code, it is a term which is often used in programming and algorithm based fields. It is a methodology that allows the programmer to represent the implementation of an algorithm. It has no syntax like any of the programming language and thus can't be compiled or interpreted by the computer.

First unit also describe about performance analysis-space complexity, time complexity, growth of functions: asymptotic notation, recurrences: substitution method, master method.

Second unit begins with Divide and Conquer. In this unit you will know all about the concept of **Divide and Conquer**. A **divide and conquer algorithm** is a strategy of solving a large problem by breaking the problem into smaller sub-problems, solving the sub-problems, and combining them to get the desired output. In this unit we have also described General method, Applications-Binary search, finding the maximum and minimum, quick sort, Heap sort and Strassen's Matrix Multiplication.

In the third unit, we provide another important topic i.e. **sorting in Linear Time**. This is the last unit of this block. In the unit we will describe about Lower bounds for sorting, Counting sort, Radix sort, Bucket sort, Medians and Order Statistics and Minimum and maximum.

As you study the material, you will find that figures, tables are properly used and these will help to understand the concept. There are many sections in the units to easily understand the topic. Every unit has summary and review questions in the end of the unit which will help you to review yourself.

In your study, you will find that every unit has different equal length and your study time will vary for each unit.

We hope you enjoy studying the material and once again wish you all the best for your success.

UNIT-1 Introduction

Structure

1.0 Introduction

This is the first unit of this block. In this block, you will study about basics of design strategies. In the Section 1.1 you will learn about Algorithm. Section 1.2 define Psuedo code for expressing algorithms. In the section 1.3 you will know about Performance Analysis and Space complexity, and Time complexity. In the section 1.4 Growth of functions like Asymptotic Notation has been defined. In the section 1.5 you will learn about Recurrences methods. Substitution method, master method has also been explained in this section. Section 1.6 has summary and section 1.7 define Review questions.

Objective

After studying this unit, you should be able to define:

- Algorithm.
- Psuedo code for expressing algorithms
- Performance Analysis-Space complexity
- Time complexity
- Growth of functions: Asymptotic Notation
- Recurrences: substitution method, master method.

1.1 Algorithm

The word Algorithm means "a process or set of rules to be followed in calculations or other problem-solving operations". Therefore, Algorithm refers to a set of rules/instructions that step-by-step define how a work is to be executed upon in order to get the expected results.

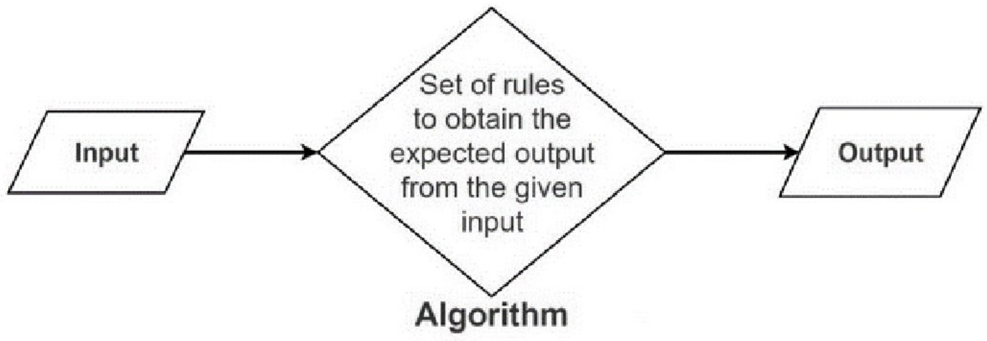

Input
Set of rules to obtain the expected output from the given input
Output
Algorithm

Figure 1.1: Algorithm

It can be understood by taking an example of cooking a new recipe. To cook a new recipe, one reads the instructions and steps and execute them one by one, in the given sequence. The result thus obtained is the new dish cooked perfectly. Similarly, algorithms help to do a task in programming to get the expected output.

The Algorithm designed are language-independent, i.e. they are just plain instructions that can be implemented in any language, and yet the output will be the same, as expected.

Characteristics of an Algorithm

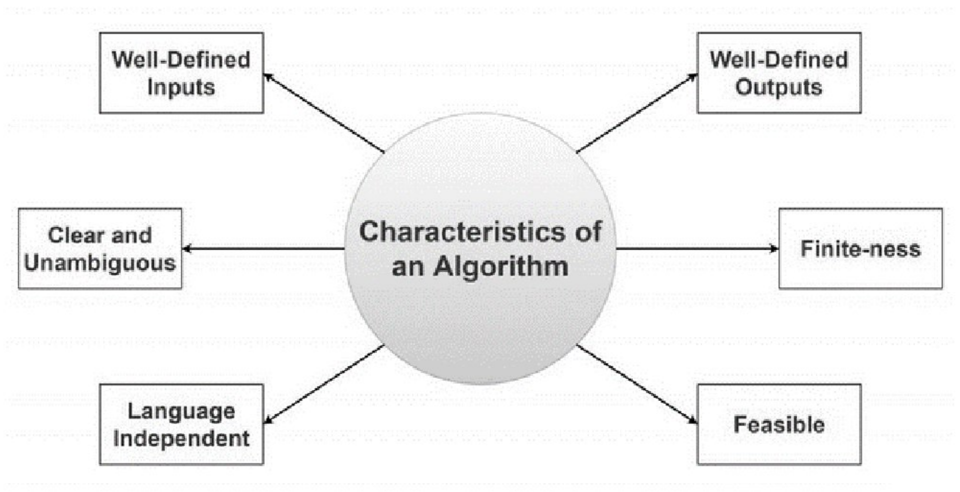


Figure 1.2: Algorithm Characteristics

As one would not follow any written instructions to cook the recipe, but only the standard one. Similarly, not all written instructions for programming is an algorithm. In order for some instructions to be an algorithm, it must have the following characteristics:

- Clear and Unambiguous: Algorithm should be clear and unambiguous. Each of its steps should be clear in all aspects and must lead to only one meaning.
- Well-Defined Inputs: If an algorithm is likely to take inputs, it should be well-defined inputs.
- Well-Defined Outputs: The algorithm must clearly define what output will be yielded and it should be well-defined as well.
- Finite-ness: The algorithm must be finite, i.e. it should not end up in an infinite loops or similar.

- Feasible: The algorithm must be simple, generic and practical, such that it can be executed upon will the available resources. It must not contain some future technology, or anything.
- Language Independent: The Algorithm designed must be language-independent, i.e. it must be just plain instructions that can be implemented in any language, and yet the output will be same, as expected.

Advantages of Algorithms:

- It is easy to understand.
- Algorithm is a step-wise representation of a solution to a given problem.
- In Algorithm the problem is broken down into smaller pieces or steps hence, it is easier for the programmer to convert it into an actual program.

Disadvantages of Algorithms:

- Writing an algorithm takes a long time so it is time-consuming.
- Branching and looping statements are difficult to show in Algorithms.

How to Design an Algorithm?

In order to write an algorithm, following things are needed as a pre-requisite:

- The problem that is to be solved by this algorithm.
- The constraints of the problem that must be considered while solving the problem.
- The input to be taken to solve the problem.
- The output to be expected when the problem is solved.
- The solution to this problem, in the given constraints.

Then the algorithm is written with the help of above parameters such that it solves the problem.

Example: Consider the example to add three numbers and print the sum.

Step 1: Fulfilling the pre-requisites

As discussed above, in order to write an algorithm, its pre-requisites must be fulfilled.

1. The problem that is to be solved by this algorithm: Add 3 numbers and print their sum.
2. The constraints of the problem that must be considered while solving the problem: The numbers must contain only digits and no other characters.
3. The input to be taken to solve the problem: The three numbers to be added.
4. The output to be expected when the problem the is solved: The sum of the three numbers taken as the input.
5. The solution to this problem, in the given constraints: The solution consists of adding the 3 numbers. It can be done with the help of '+' operator, or bit-wise, or any other method.

Step 2: Designing the algorithm

Now let's design the algorithm with the help of above pre-requisites:

Algorithm to add 3 numbers and print their sum:

```
1. START
2. Declare 3 integer variables num1, num2 and num3.
3. Take the three numbers, to be added, as inputs in variables num1,
   num2, and num3 respectively.
4. Declare an integer variable sum to store the resultant sum of the 3
   numbers.
5. Add the 3 numbers and store the result in the variable sum.
6. Print the value of variable sum
7. END
```

Step 3: Testing the algorithm by implementing it.

In order to test the algorithm, let's implement it in C language.

```
// C program to add three numbers

// with the help of above designed algorithm

#include <stdio.h>

int main()
```

```
{

   // Variables to take the input of the 3 numbers
   int num1, num2, num3;
    // Variable to store the resultant sum
   int sum;
    // Take the 3 numbers as input
   printf("Enter the 1st number: ");
   scanf("%d", &num1);
   printf("%d\n", num1);
   printf("Enter the 2nd number: ");
   scanf("%d", &num2);
   printf("%d\n", num2);
    printf("Enter the 3rd number: ");
   scanf("%d", &num3);
   printf("%d\n", num3);
    // Calculate the sum using + operator
   // and store it in variable sum
   sum = num1 + num2 + num3;
    // Print the sum
   printf("\nSum of the 3 numbers is: %d", sum);
    return 0;
```

Output

```
Enter the 1st number: 0

Enter the 2nd number: 0

Enter the 3rd number: -1577141152

Sum of the 3 numbers is: -1577141152
```

1.2 Psuedo code for expressing algorithms

Pseudocode refers to an informal high-level description of the operating principle of a computer program or other algorithm. It uses structural conventions of a standard programming language intended for human reading rather than the machine reading.

Advantages of Pseudocode

- Since it is similar to a programming language, it can be quickly transformed into the actual programming language than a flowchart.
- The layman can easily understand it.
- Easily modifiable as compared to the flowcharts.
- Its implementation is beneficial for structured, designed elements.
- It can easily detect an error before transforming it into a code.

Disadvantages of Pseudocode

- Since it does not incorporate any standardized style or format, it can vary from one company to another.
- Error possibility is higher while transforming into a code.
- It may require a tool for extracting out the Pseudocode and facilitate drawing flowcharts.
- It does not depict the design.

Difference between Algorithm and the Pseudocode

Basis for comparison	Algorithm	Pseudocode
Comprehensibility	Quite hard to understand	Easy to interpret

Uses	Complicated programming language	Combination of programming language and natural language
Debugging	Moderate	Simpler
Ease of construction	Complex	Easier

An algorithm is simply a problem-solving process, which is used not only in computer science to write a program but also in our day to day life. It is nothing but a series of instructions to solve a problem or get to the problem's solution. It not only helps in simplifying the problem but also to have a better understanding of it.

However, Pseudocode is a way of writing an algorithm. Programmers can use informal, simple language to write pseudocode without following any strict syntax. It encompasses semi-mathematical statements.

Example: Suppose there are 60 students in the class. How will you calculate the number of absentees in the class?

Pseudo Approach:

```
1. Initialize a variable called as Count to zero, absent to zero, total to 60
2. FOR EACH Student PRESENT DO the following:
3. Increase the Count by One
4. Then Subtract Count from total and store the result in absent
5. Display the number of absent students
```

Algorithmic Approach:

```
1. Count <- 0, absent <- 0, total <- 60
2. REPEAT till all students counted
3. Count <- Count + 1
4. absent <- total - Count
5. Print "Number absent is:" , absent
```

Q1. What is an algorithm? What is the need for an algorithm?

Q2. What is the Complexity of Algorithm?

1.3 Performance Analysis

Performance analysis of an algorithm is the process of calculating space and time required by that algorithm. Performance analysis of an algorithm is performed by using the following measures... Space required to complete the task of that algorithm (Space Complexity). It includes program space and data space.
In theoretical analysis of algorithms, it is common to estimate their complexity in the asymptotic sense, i.e., to estimate the complexity function for arbitrarily large input. The term "analysis of algorithms" was coined by Donald Knuth.

Algorithm analysis is an important part of computational complexity theory, which provides theoretical estimation for the required resources of an algorithm to solve a specific computational problem. Most algorithms are designed to work with inputs of arbitrary length. Analysis of algorithms is the determination of the amount of time and space resources required to execute it.

Generally, the efficiency or execution time of an algorithm is expressed as a function that relates the length of the input to the number of steps (called time complexity) or the amount of memory (called space complexity).

The Need for Analysis

In this chapter, we will discuss the need for analysis of algorithms and how to choose a better algorithm for a particular problem as one computational problem can be solved by different algorithms.

By considering an algorithm for a specific problem, we can begin to develop pattern recognition so that similar types of problems can be solved by the help of this algorithm.

Algorithms are often quite different from one another, though the objective of these algorithms are the same. For example, we know that a set of numbers can be sorted using different algorithms. Number of comparisons performed by one algorithm may vary with others for the same input. Hence, time complexity of those algorithms may differ. At the same time, we need to calculate the memory space required by each algorithm.

Analysis of algorithm is the process of analysing the problem-solving capability of the algorithm in terms of the time and size required (the size of memory for storage while implementation). However, the main concern of analysis of algorithms is the required time or performance. Generally, we perform the following types of analysis –

- Worst-case − the maximum number of steps taken on any instance of size a.
- Best-case − the minimum number of steps taken on any instance of size a.
- Average case − an average number of steps taken on any instance of size a.
- Amortized − A sequence of operations applied to the input of size a averaged over time.

To solve a problem, we need to consider time as well as space complexity as the program may run on a system where memory is limited but adequate space is available or may be vice-versa.

Space Complexity:

Space Complexity of an algorithm denotes the total space used or needed by the algorithm for its working, for various input sizes. For example:

```
vector<int> myVec(n);

for(int i = 0; i < n; i++)

   cin >> myVec[i];
```

n the above example, we are creating a vector of size n. So the space complexity of the above code is in the order of "n" i.e. if n will increase, the space requirement will also increase accordingly.

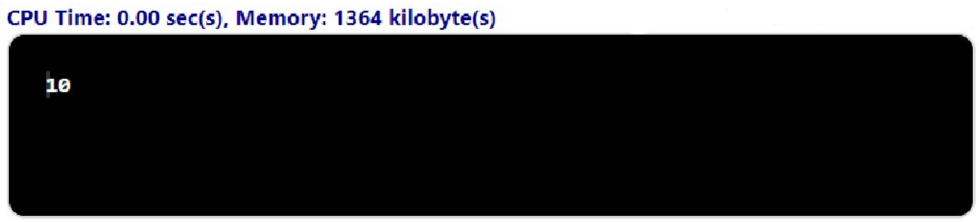
Even when you are creating a variable then you need some space for your algorithm to run. All the space required for the algorithm is collectively called the Space Complexity of the algorithm.

NOTE: In normal programming, you will be allowed to use 256MB of space for a particular problem. So, you can't create an array of size more 10^8 because you will be allowed to use only 256MB. Also, you can't create an array of size more than 10^6 in a function because the maximum space allotted to a function is 4MB. So, to use an array of more size, you can create a global array.

Example

```
#include<stdio.h>

int main()

{

  int a = 5, b = 5, c;

  c = a + b;

  printf("%d", c);

}
```

Output:



```
CPU Time: 0.00 sec(s), Memory: 1364 kilobyte(s)

10
```

Explanation: Do not misunderstand space-complexity to be 1364 Kilobytes as shown in the output image. The method to calculate the actual space complexity is shown below.

In the above program, 3 integer variables are used. The size of the integer data type is 2 or 4 bytes which depends on the compiler. Now, let's assume the size as 4 bytes. So, the total space occupied by the above-given program is 4 * 3 = 12 bytes. Since no additional variables are used, no extra space is required.

Hence, space complexity for the above-given program is O(1), or constant.

Time Complexity

The time complexity is the number of operations an algorithm performs to complete its task with respect to input size (considering that each operation takes the same amount of time). The algorithm that performs the task in the smallest number of operations is considered the most efficient one.

The time taken by an algorithm also depends on the computing speed of the system that you are using, but we ignore those external factors and we are only concerned on the number of times a particular statement is being executed with respect to the input size. Let's say, for executing one statement, the time taken is 1sec, then what is the time taken for executing n statements, it will take n seconds.

Suppose you are having one problem and you wrote three algorithms for the same problem. Now, you need to choose one out of those three algorithms. How will you do that?

- One thing that you can do is just run all the three algorithms on three different computers, provide same input and find the time taken by all the three algorithms and choose the one that is taking the least amount of time. Is it ok? No, all the systems might be using some different processors. So, the processing speed might vary. So, we can't use this approach to find the most efficient algorithm.
- Another thing that you can do is run all the three algorithms on the same computer and try to find the time taken by the algorithm and choose the best. But here also, you might get wrong results because, at the time of execution of a program, there are other things that are executing along with your program, so you might get the wrong time.

NOTE: One thing that is to be noted here is that we are finding the time taken by different algorithms for the same input because if we change the input then the efficient algorithm might take more time as compared to the less efficient one because the input size is different for both algorithms.

So, we have seen that we can't judge an algorithm by calculating the time taken during its execution in a particular system. We need some standard notation to analyse the algorithm. We use asymptotic notation to analyse any algorithm and based on that we find the most efficient algorithm. Here in Asymptotic notation, we do not consider the system configuration, rather we consider the order of growth of the input. We try to find how the time or the space taken by the algorithm will increase/decrease after increasing/decreasing the input size.

1.4 Growth of functions

There are three asymptotic notations that are used to represent the time complexity of an algorithm. They are:

- Θ Notation (theta)
- Big O Notation
- Ω Notation

Before learning about these three asymptotic notation, we should learn about the best, average, and the worst case of an algorithm.

Best case, Average case, and Worst case

An algorithm can have different time for different inputs. It may take 1 second for some input and 10 seconds for some other input.

For example: We have one array named "arr" and an integer "k". we need to find if that integer "k" is present in the array "arr" or not? If the integer is there, then return 1 other return 0. Try to make an algorithm for this question.

The following information can be extracted from the above question:

- Input: Here our input is an integer array of size "n" and we have one integer "k" that we need to search for in that array.
- Output: If the element "k" is found in the array, then we have return 1, otherwise we have to return 0.

Now, one possible solution for the above problem can be linear search i.e. we will traverse each and every element of the array and compare that element with "k". If it is equal to "k" then return 1, otherwise, keep on comparing for more elements in the array and if you reach at the end of the array and you did not find any element, then return 0.

```
/*

* @type of arr: integer array

* @type of n: integer (size of integer array)

* @type of k: integer (integer to be searched)

*/

int searchK(int arr[], int n, int k)

{

  // for-loop to iterate with each element in the array

  for (int i = 0; i < n; ++i)

  {

    // check if ith element is equal to "k" or not

    if (arr[i] == k)

      return 1; // return 1, if you find "k"
  }
  return 0; // return 0, if you didn't find "k"
}
/*
* [Explanation]
* i = 0 -----------> will be executed once
* i < n -----------> will be executed n+1 times
* i++ -------------> will be executed n times
* if(arr[i] == k) --> will be executed n times
* return 1 ---------> will be executed once(if "k" is there in the array)
* return 0 ---------> will be executed once(if "k" is not there in the array)
*/
```

Each statement in code takes constant time, let's say "C", where "C" is some constant. So, whenever you declare an integer then it takes constant time when you change the value of some integer or other variables then it takes constant time, when you compare two variables then it takes constant time. So, if a statement is taking "C" amount of time and it is executed "N" times, then it will take C*N amount of time. Now, think of the following inputs to the above algorithm that we have just written:

NOTE: Here we assume that each statement is taking 1sec of time to execute.

- If the input array is [1, 2, 3, 4, 5] and you want to find if "1" is present in the array or not, then the if-condition of the code will be executed 1 time and it will find that the element 1 is there in the array. So, the if-condition will take 1 second here.
- If the input array is [1, 2, 3, 4, 5] and you want to find if "3" is present in the array or not, then the if-condition of the code will be executed 3 times and it will find that the element 3 is there in the array. So, the if-condition will take 3 seconds here.
- If the input array is [1, 2, 3, 4, 5] and you want to find if "6" is present in the array or not, then the if-condition of the code will be executed 5 times and it will find that the element 6 is not there in the array and the

algorithm will return 0 in this case. So, the if-condition will take 5 seconds here.

As you can see that for the same input array, we have different time for different values of "k". So, this can be divided into three cases:

- Best case: This is the lower bound on running time of an algorithm. We must know the case that causes the minimum number of operations to be executed. In the above example, our array was [1, 2, 3, 4, 5] and we are finding if "1" is present in the array or not. So here, after only one comparison, you will get that your element is present in the array. So, this is the best case of your algorithm.
- Average case: We calculate the running time for all possible inputs, sum all the calculated values and divide the sum by the total number of inputs. We must know (or predict) distribution of cases.
- Worst case: This is the upper bound on running time of an algorithm. We must know the case that causes the maximum number of operations to be executed. In our example, the worst case can be if the given array is [1, 2, 3, 4, 5] and we try to find if element "6" is present in the array or not. Here, the if-condition of our loop will be executed 5 times and then the algorithm will give "0" as output.

So, we learned about the best, average, and worst case of an algorithm. Now, let's get back to the asymptotic notation where we saw that we use three asymptotic notations to represent the complexity of an algorithm i.e. Θ Notation (theta), Ω Notation, Big O Notation.

Θ Notation (theta)

The Θ Notation is used to find the average bound of an algorithm i.e. it defines an upper bound and a lower bound, and your algorithm will lie in between these levels. So, if a function is g(n), then the theta representation is shown as Θ(g(n)) and the relation is shown as:

$$\Theta(g(n)) = \{ f(n): \text{there exist positive constants } c1, c2 \text{ and } n0$$
$$\text{such that } 0 \le c1g(n) \le f(n) \le c2g(n) \text{ for all } n \ge n0 \}$$

The above expression can be read as theta of g(n) is defined as set of all the functions f(n) for which there exists some positive constants c1, c2, and n0 such that c1*g(n) is less than or equal to f(n) and f(n) is less than or equal to c2*g(n) for all n that is greater than or equal to n0.

For example:

if $f(n) = 2n^2 + 3n + 1$

and $g(n) = n^2$

then for c1 = 2, c2 = 6, and n0 = 1, we can say that $f(n) = \Theta(n^2)$

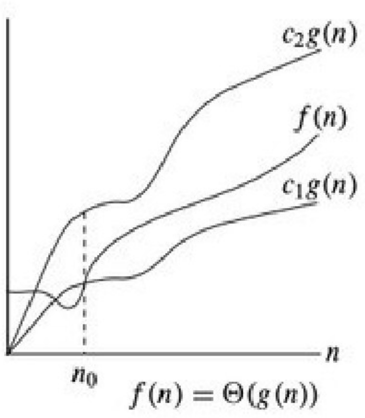


Figure 1.3: Θ Notation (theta)

Ω Notation

The Ω notation denotes the lower bound of an algorithm i.e. the time taken by the algorithm can't be lower than this. In other words, this is the fastest time in which the algorithm will return a result. It's the time taken by the algorithm when provided with its best-case input. So, if a function is g(n), then the omega representation is shown as Ω(g(n)) and the relation is shown as:

$\Omega(g(n)) = \{f(n)$: there exist positive constants c and n0

such that $0 \le cg(n) \le f(n)$ for all $n \ge n0\}$

The above expression can be read as omega of g(n) is defined as set of all the functions f(n) for which there exist some constants c and n0 such that c*g(n) is less than or equal to f(n), for all n greater than or equal to n0.

if $f(n) = 2n^2 + 3n + 1$

and $g(n) = n^2$

then for c = 2 and n0 = 1, we can say that $f(n) = \Omega(n^2)$

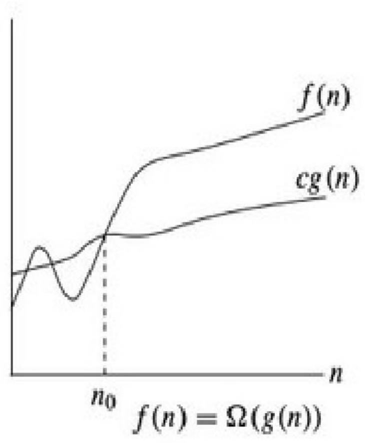


Figure 1.4: Ω Notation

Big O Notation

The Big O notation defines the upper bound of any algorithm i.e. your algorithm can't take more time than this time. In other words, we can say that the big O notation denotes the maximum time taken by an algorithm or the worst-case time complexity of an algorithm. So, big O notation is the most used notation for the time complexity of an algorithm. So, if a function is g(n), then the big O representation of g(n) is shown as O(g(n)) and the relation is shown as:

O(g(n)) = {f(n): there exist positive constants c and n0

such that $0 \le f(n) \le cg(n)$ for all $n \ge n0$}

The above expression can be read as Big O of g(n) is defined as a set of functions f(n) for which there exist some constants c and n0 such that f(n) is greater than or equal to 0 and f(n) is smaller than or equal to c*g(n) for all n greater than or equal to n0.

if $f(n) = 2n^2 + 3n + 1$

and $g(n) = n^2$

then for $c = 6$ and $n0 = 1$, we can say that $f(n) = O(n^2)$

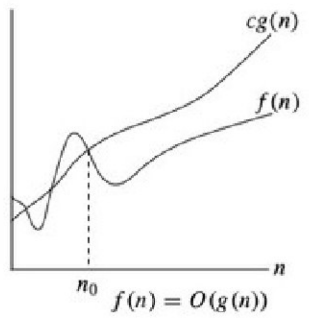


Figure 1.5: Big O Notation

Big O notation example of Algorithms

Big O notation is the most used notation to express the time complexity of an algorithm. In this section of the blog, we will find the big O notation of various algorithms.

Example 1: Finding the sum of the first n numbers.

In this example, we have to find the sum of first n numbers. For example, if n = 4, then our output should be 1 + 2 + 3 + 4 = 10. If n = 5, then the output should be 1 + 2 + 3 + 4 + 5 = 15. Let's try various solutions to this code and try to compare all those codes.

O(1) solution

```
int findSum(int n)

{

   return n * (n+1) / 2; // this will take some constant time c1

}
```

In the above code, there is only one statement and we know that a statement takes constant time for its execution. The basic idea is that if the statement is taking constant time, then it will take the same amount of time for all the input size and we denote this as O(1).

O(n) solution

In this solution, we will run a loop from 1 to n and we will add these values to a variable named "sum".

```
//function taking input "n"

int findSum(int n)

{

   int sum = 0; //----------------> it takes some constant time "c1"

   for(int i = 1; i <= n; ++i) //--> here the comparison and increment will
take place n times(c2*n) and the creation of i takes place with some constant
time

      sum = sum + i; //----------> this statement will be executed n times i.e.
c3*n

   return sum; //-----------------> it takes some constant time "c4"

}

/*

* Total time taken = time taken by all the statements to execute

* here in our example we have 3 constant time taking statements i.e. "sum =
0", "i = 0", and "return sum", so we can add all the constants and replace with
some new constant "c"

* apart from this, we have two statements running n-times i.e. "i < n(in real
n+1)" and "sum = sum  + i" i.e. c2*n + c3*n = c0*n

* Total time taken = c0*n + c

*/
```

The big O notation of the above code is O(c0*n) + O(c), where c and c0 are constants. So, the overall time complexity can be written as O(n).

O(n^2) solution

In this solution, we will increment the value of sum variable "i" times i.e. for i = 1, the sum variable will be incremented once i.e. sum = 1. For i = 2, the sum variable will be incremented twice. So, let's see the solution.

```
//function taking input "n"

int findSum(int n)

{
    int sum = 0; //--------------------> constant time

    for(int i = 1; i <= n; ++i)

        for(int j = 1; j <= i; ++j)

            sum++; //------------------> it will run [n * (n + 1) / 2]

    return sum; //---------------------> constant time

}

/*

* Total time taken = time taken by all the statements to execute

* the statement that is being executed most of the time is "sum++" i.e. n * (n +
1) / 2

* So, total complexity will be: c1*n² + c2*n + c3 [c1 is for the constant terms
of n², c2 is for the constant terms of n, and c3 is for rest of the constant time]

*/
```

The big O notation of the above algorithm is $O(c1*n^2) + O(c2*n) + O(c3)$. Since we take the higher order of growth in big O. So, our expression will be reduced to $O(n^2)$.

Q1. What are the Asymptotic Notations?

Q2. Write pseudo code that reads two numbers and multiplies them together and print out their product.

So, until now, we saw 3 solutions for the same problem. Now, which algorithm will you prefer to use when you are finding the sum of first "n" numbers? If your answer is O(1) solution, then we have one bonus section for you at the end of this blog. We would prefer the O(1) solution because the time taken by the algorithm will be constant irrespective of the input size.

Check your progress

Q1.What is the difference between an algorithm and Psuedo code?
Q2.How is space complexity calculated?
Q3.What is the best time complexity? Explain with example.

1.5 Recurrences

A recurrence is an equation or inequality that describes a function in terms of its values on smaller inputs. To solve a Recurrence Relation means to obtain a function defined on the natural numbers that satisfy the recurrence.

For Example, the Worst Case Running Time T(n) of the MERGE SORT Procedures is described by the recurrence.

$T(n) = \theta(1)$ *if* $n=1$
$2T(\frac{n}{2}) + \theta(n)$ *if* $n>1$

Substitution Method:
The substitution method for solving recurrences involves guessing the form of the solution and then using mathematical induction to find the constants and show that the solution works. The name comes from the substitution of the guessed answer for the function when the inductive hypothesis is applied to smaller values. This method is powerful, but it obviously can be applied only in cases when it is easy to guess the form of the answer.

The Substitution Method consists of two main steps:

- Guess the Solution.
- Use the mathematical induction to find the boundary condition and shows that the guess is correct.

Example

Solve the equation by Substitution Method.

$$T(n) = T(\frac{n}{2}) + n$$

We have to show that it is asymptotically bound by $O(\log n)$.

Solution:

$$\text{For } T(n) = O(\log n)$$

We have to show that for some constant c

$$T(n) \leq c \log_n.$$

Put this in given Recurrence Equation.

$T(n) \leq c\, log(\frac{n}{2}) + 1$

$\leq c\, log(\frac{n}{2}) + 1 = c\, logn - clog_2 2 + 1$

$\leq c\, log_n \; for\; c \geq 1$

Thus T (n) =O $\log_n$.

Master Method

The Master Method is used for solving the following types of recurrence
T (n) = a T(n/b)+ f (n) with a≥1 and b≥1 be constant & f(n) be a function and (n/b) be interpreted as follows:

Let T (n) is defined on non-negative integers by the recurrence.

$$T(n) = a\, T(n/b) + f(n)$$

In the function to the analysis of a recursive algorithm, the constants and function take on the following significance:

- *n* is the size of the problem.
- *a* is the number of sub problems in the recursion.
- *n/b* is the size of each sub problem. (Here it is assumed that all sub problems are essentially the same size.)
- *f(n)* is the sum of the work done outside the recursive calls, which includes the sum of dividing the problem and the sum of combining the solutions to the sub problems.
- It is not possible always bound the function according to the requirement, so we make three cases which will tell us what kind of bound we can apply on the function.

Master Theorem:

It is possible to complete an asymptotic tight bound in these three cases:

$$T(n) = \begin{cases} \Theta\left(n^{\log_b a}\right) & f(n) = O\left(n^{\log_b a - \varepsilon}\right) \\ \Theta\left(n^{\log_b a} \log n\right) & f(n) = \Theta\left(n^{\log_b a}\right) \\ \Theta\left(f(n)\right) & f(n) = \Omega\left(n^{\log_b a + \varepsilon}\right) \text{AND} \\ & af(n/b) < cf(n) \text{ forlarge} n \end{cases} \left.\vphantom{\begin{matrix}a\\b\\c\\d\end{matrix}}\right\} \begin{matrix} \varepsilon > 0 \\ c < 1 \end{matrix}$$

Case1: If f (n) = $O(n^{\log_b a-\epsilon})$for some constant ε >0, then it follows that:

$$T(n) = \Theta(n^{\log_b a})$$

Example:

***T (n) = 8 T (n/2) +1000 n²* apply master theorem on it.**

Solution:

Compare T (n) = $8\ T(n/2) + 1000\ n^2$ with

$T(n)= a\ T(n/b) + f(n)$ *with* $a \geq 1$ *and* $b \geq 1$

$a=8,\ b=2,\ f(n) = 1000\ n^2,\ \log_b a = \log_2 8 = 3$

Put all the values in : $f(n) = O(n^{\log_b a-\varepsilon})$

$1000\ n^2 = o\ (n^{3-\varepsilon})$

If we choose ε = 1, we get: $1000\ n^2 = o\ (n^{3-1}) = o(n^2)$

Since this equation holds, the first case of the master theorem applies to the given recurrence relation, thus resulting in the conclusion:

$T(n) = \Theta(n^{\log_b a})$

Therefore: $T\ (n) = \Theta\ (n^3)$

Case 2: If it is true, for some constant k ≥ 0 that:

$F\ (n) = \Theta\ (n^{\log_b a} \log^k n)$ then it follows that $T(n) = \Theta\ (n^{\log_b a} \log^{k+1} n)$

Example:

$T\ (n) = 2\ T(n/2) + 10n$, solve the recurrence by using the master method.

As compare the given problem with $T\ (n) = a\ T(n/b) + f(n)$ *with* $a \geq 1$ *and* $b \geq 1$

$a=2,\ b=2,\ k=0,\ f(n) = 10n,\ \log_b a = \log_2 2 = 1$

Put all the values in:$f(n) = O\ (n^{\log_b a} \log^k n)$, *we will get*

$10n = \Theta\ (n^1) = \Theta\ (n)$ which is true.

Therefore: T (n) = Θ $(n^{\log_b a} \log^{k+1} n)$

$= \Theta\ (n \log n)$

Case 3: If it is true $f(n) = \Omega\ (n^{\log_b a + \varepsilon)}$ *for* some constant $\varepsilon > 0$ and it also true that: a $f(n/b) \leq cf(n)$for some constant $c<1$ for large value of n, then:

$T\ (n) = \Theta\ ((f\ (n))$

Example: Solve the recurrence relation:

$T(n) = 2\,T(n/2) + n^2$

Solution:

Compare the given problem with $T(n) = a\,T(n/b) + f(n)$ *with* $a \geq 1$ *and* $b \geq 1$

$a = 2$, $b = 2$, $f(n) = n^2$, $\log_b a = \log_2 2 = 1$

Put all the values in $f(n) = \Omega(n^{\log_b a + \varepsilon})$ ………….. (Eq. 1)

If we insert all the value in (Eq.1), we will get

$n^2 = \Omega(n^{1+\varepsilon})$ *put* $\varepsilon = 1$, then the equality will hold.

$n^2 = \Omega(n^{1+1}) = \Omega(n^2)$

Now we will also check the second condition:

$2(n/2)^2 \leq cn^2 \Rightarrow 1/2\, n^2 \leq cn^2$

If we will choose c =1/2, it is true:

$1/2\, n^2 \leq 1/2\, n^2 \;\forall n \geq 1$

So it follows: $T(n) = \Theta((f(n))$

$T(n) = \Theta(n^2)$

1.6 Summary

In this unit you have learnt about Algorithm, Psuedo code for expressing algorithms. You have also studied about performance Analysis with the help of Space complexity and Time complexity. We have explained thegrowth of functions like Asymptotic Notation and in the end we have provided the concept of Recurrences with the help of substitution method and master method.

- An algorithm can be defined as a finite set of steps, which has to be followed while carrying out a particular problem. It is nothing but a process of executing actions step by step.
- An algorithm is a distinct computational procedure that takes input as a set of values and results in the output as a set of values by solving the problem. More precisely, an algorithm is correct, if, for each input instance, it gets the correct output and gets terminated.

- An algorithm unravels the computational problems to output the desired result. An algorithm can be described by incorporating a natural language such as English, Computer language, or a hardware language.

1.7 Review Questions

Q1. What is the space complexity of following code?

```
int a = 0, b = 0;
for (i = 0; i < N; i++) {
   a = a + rand();
}
for (j = 0; j < M; j++) {
   b = b + rand();
}
```

Q2. What is the time complexity of following code?

```
int i, j, k = 0;
for (i = n / 2; i <= n; i++) {
   for (j = 2; j <= n; j = j * 2) {
      k = k + n / 2;
   }
}
```

Q3. What does it mean when we say that an algorithm X is asymptotically more efficient than Y?

Q4. Consider the Recurrence

T (n) = 2T(n/2)+ n n>1
Find an Asymptotic bound on T.

Q5. Explain why the statement, "The running time of algorithm A is at least O(n2)," is content-free.

UNIT-2 Divide and Conquer

Structure

2.0 Introduction

This is the second unit of this block. In this unit, there are eight sections. In the section 2.1 you will know about general method of divide and conquer. In the section 2.2 you will learn about binary search applications. Section 2.3 define about finding the maximum and minimum. Section 2.4 explain about quick sort and section 2.5 define heap sort. In the section 2.6, you will study about Strassen's Matrix Multiplication. Last two sections define summary and review questions.

Objective

After studying this unit, you should be able to define:

- General method of divide and conquer.
- Applications of Binary search
- Finding the maximum and minimum
- Quick sort
- Heap sort
- Strassen's Matrix Multiplication

2.1 Divide and Conquer

Divide and Conquer is an algorithmic pattern. In algorithmic methods, the design is to take a dispute on a huge input, break the input into minor pieces, decide the problem on each of the small pieces, and then merge the piecewise solutions into a global solution. This mechanism of solving the problem is called the Divide & Conquer Strategy.

Divide and Conquer algorithm consists of a dispute using the following three steps.

1. Divide the original problem into a set of subproblems.
2. Conquer: Solve every subproblem individually, recursively.
3. Combine: Put together the solutions of the subproblems to get the solution to the whole problem.

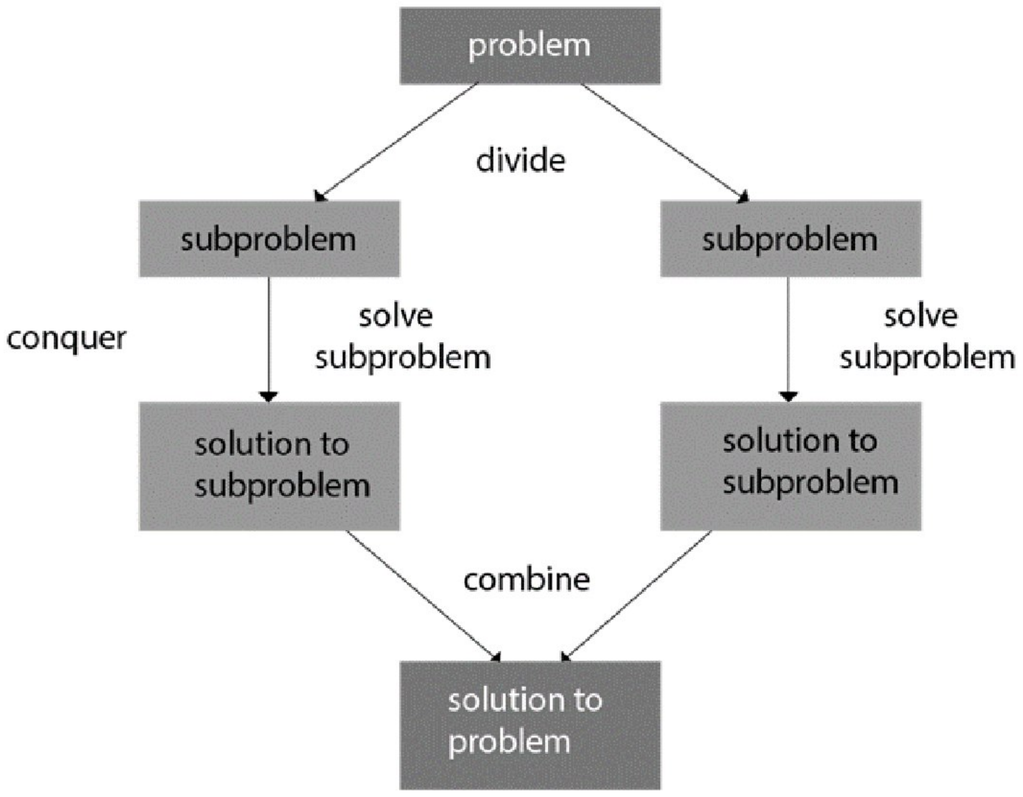


Figure 2.1: Divide and Conquer

Generally, we can follow the divide-and-conquer approach in a three-step process.

Examples: The specific computer algorithms are based on the Divide & Conquer approach:

1. Maximum and Minimum Problem
2. Binary Search
3. Sorting (merge sort, quick sort)
4. Tower of Hanoi.

Fundamental of Divide & Conquer Strategy:

There are two fundamental of Divide & Conquer Strategy:

1. Relational Formula
2. Stopping Condition

1. Relational Formula: It is the formula that we generate from the given technique. After generation of Formula we apply D&C Strategy, i.e. we break the problem recursively & solve the broken subproblems.

2. Stopping Condition: When we break the problem using Divide & Conquer Strategy, then we need to know that for how much time, we need to apply divide & Conquer. So the condition where the need to stop our recursion steps of D&C is called as Stopping Condition.

2.2Applications-Binary search

Following algorithms are based on the concept of the Divide and Conquer Technique:

1. Binary Search: The binary search algorithm is a searching algorithm, which is also called a half-interval search or logarithmic search. It works by comparing the target value with the middle element existing in a sorted array. After making the comparison, if the value differs, then the half that cannot contain the target will eventually eliminate, followed by continuing the search on the other half. We will again consider the middle element and compare it with the target value. The process keeps on repeating until the target value is met. If we found the other half to be empty after ending the search, then it can be concluded that the target is not present in the array.
2. Quicksort: It is the most efficient sorting algorithm, which is also known as partition-exchange sort. It starts by selecting a pivot value from an array followed by dividing the rest of the array elements into two sub-arrays. The partition is made by comparing each of the elements with the pivot value. It compares whether the element holds a greater value or lesser value than the pivot and then sort the arrays recursively.

3. Merge Sort: It is a sorting algorithm that sorts an array by making comparisons. It starts by dividing an array into sub-array and then recursively sorts each of them. After the sorting is done, it merges them back.
4. Closest Pair of Points: It is a problem of computational geometry. This algorithm emphasizes finding out the closest pair of points in a metric space, given n points, such that the distance between the pair of points should be minimal.
5. Strassen's Algorithm: It is an algorithm for matrix multiplication, which is named after Volker Strassen. It has proven to be much faster than the traditional algorithm when works on large matrices with a better asymptotic complexity, although the naive algorithm is often better for smaller matrices.
6. Cooley-Tukey Fast Fourier Transform (FFT) algorithm: The Fast Fourier Transform algorithm is named after J. W. Cooley and John Turkey. It follows the Divide and Conquer Approach and imposes a complexity of O(nlogn).
7. Karatsuba algorithm for fast multiplication: It is one of the fastest multiplication algorithms of the traditional time, invented by Anatoly Karatsuba in late 1960 and got published in 1962. It multiplies two n-digit numbers in such a way by reducing it to at most single-digit.

2.3 Finding the maximum and minimum

Problem: Analyse the algorithm to find the maximum and minimum element from an array.

```
Algorithm: Max ?Min Element (a [])
Max:  a [i]
Min:   a [i]
For i= 2 to n do
If a[i]> max then
max = a[i]
if a[i] < min then
min: a[i]
return (max, min)
```

Analysis:

Method 1: if we apply the general approach to the array of size n, the number of comparisons required are 2n-2.

Method-2: In another approach, we will divide the problem into sub-problems and find the max and min of each group, now max. Of each group will compare with the only max of another group and min with min.

Let n = is the size of items in an array

Let T (n) = time required to apply the algorithm on an array of size n. Here we divide the terms as T(n/2).

2 here tends to the comparison of the minimum with minimum and maximum with maximum as in above example.

$T(n) = \left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + 2$

$T(n) = 2T\left(\frac{n}{2}\right) + 2$ -Eq (i)

T (2) = 1, time required to compare two elements/items. (Time is measured in units of the number of comparisons)

$T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{2^2}\right) + 2$ -Eq (ii)

Put Eq (ii) in Eq (i)

$T(n) = 2\left[2T\left(\frac{n}{2^2}\right) + 2\right]$

$= 2^2T\left(\frac{n}{2^2}\right) + 2^2 + 2$

Similarly, apply the same procedure recursively on each sub problem or anatomy

{Use recursion means, we will use some stopping condition to stop the algorithm}

$T(n) = 2^iT\left(\frac{n}{2^i}\right) + 2^i + 2^{i-1} + \cdots \ldots\ldots\ldots. +2$ ...... Eq(iii)

Recursion will stop, when $\frac{n}{2^i} = 2 \Rightarrow n = 2^{i+!}$ ...Eq. (iv)

Put the Eq.(iv) into Eq.(iii)

$Tn = 2^iT(2) + 2^i + 2^{i-1} + \ldots\ldots\ldots. +2$

$= 2^i.1 + 2^i + 2^{i-1} + \cdots \ldots\ldots.. +2$

$= 2^i + \frac{2(2^i-1)}{2-1}$

$= 2^{i+1} + 2^i - 2$

$= n + \frac{n}{2} - 2$
$= \frac{3n}{2} - 2$

Number of comparisons requires applying the divide and conquering algorithm on n elements/items = $\frac{3n}{2} - 2$

Number of comparisons requires applying general approach on n elements = (n-1) + (n-1) = 2n-2

From this example, we can analyse, that how to reduce the number of comparisons by using this technique.

Analysis: suppose we have the array of size 8 elements.

Method1: requires (2n-2), (2x8)-2=14 comparisons

Method2: requires$\frac{3x8}{2} - 2 = 10\ comparisions$

It is evident; we can reduce the number of comparisons (complexity) by using a proper technique.

Check your progress

Q1. What are the steps in divide and conquer strategy?

Q2. What is the time complexity of MIN () and MAX () method?

2.4 Quick Sort

It is an algorithm of Divide & Conquer type.

Divide: Rearrange the elements and split arrays into two sub-arrays and an element in between search that each element in left sub array is less than or equal to the average element and each element in the right sub-array is larger than the middle element.

Conquer: Recursively, sort two sub arrays.

Combine: Combine the already sorted array.

Algorithm:

QUICKSORT (array A, int m, int n)

```
1 if (n > m)
2 then
3 i ← a random index from [m,n]
4 swap A [i] with A[m]
5 o ← PARTITION (A, m, n)
6 QUICKSORT (A, m, o - 1)
7 QUICKSORT (A, o + 1, n)
```

Partition Algorithm:

Partition algorithm rearranges the sub arrays in a place.

```
PARTITION (array A, int m, int n)
1 x ← A[m]
2 o ← m
3 for p ← m + 1 to n
4 do if (A[p] < x)
5 then o ← o + 1
6 swap A[o] with A[p]
7 swap A[m] with A[o]
8 return o
```

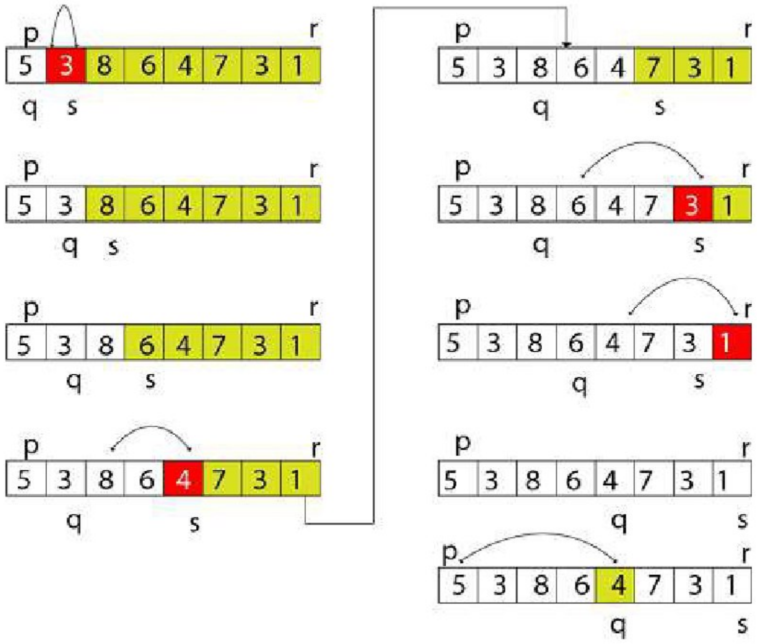


Figure 2.2: shows the execution trace partition algorithm

Example of Quick Sort:

44 33 11 55 77 90 40 60 99 22 88

Let 44 be the Pivot element and scanning done from right to left

Comparing 44 to the right-side elements, and if right-side elements are smaller than 44, then swap it. As 22 is smaller than 44 so swap them.

22 33 11 55 77 90 40 60 99 44 88

Now comparing 44 to the left side element and the element must be greater than 44 then swap them. As 55 are greater than 44 so swap them

22 33 11 44 77 90 40 60 99 55 88

Recursively, repeating steps 1 & steps 2 until we get two lists one left from pivot element 44 & one right from pivot element.

22 33 11 40 77 90 44 60 99 55 88

Swap with 77:

22 33 11 40 44 90 77 60 99 55 88

Now, the element on the right side and left side are greater than and smaller than 44 respectively.

Now we get two sorted lists:

22 33 11 40 **44** 90 77 66 99 55 88

Sublist1 **Sublist2**

And these sublists are sorted under the same process as above done.

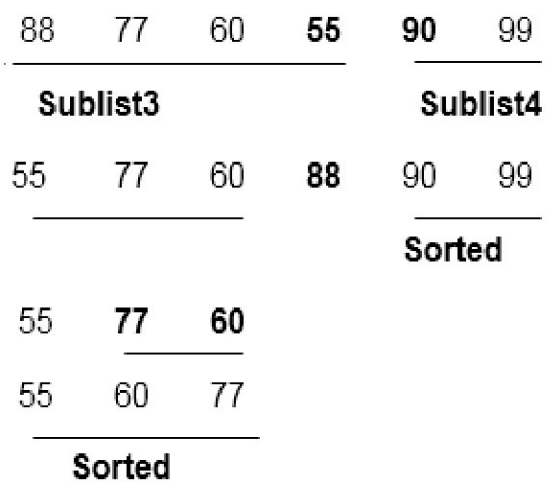
These two sorted sublists side by side.

22 33 11 40 **44** **90** 77 60 99 55 88

11 33 **22** 40 **44** 88 77 60 99 55 **90**

11 **22** 33 40 **44** 88 77 60 **90** 55 **99**

First sorted list



Merging Sublists:

11	**22**	**33**	**40**	**44**	**55**	**60**	**77**	**88**	**90**	**99**

Figure 2.3: Sorted Lists

Worst Case Analysis: It is the case when items are already in sorted form and we try to sort them again. This will take lots of time and space.

Equation:

$T\ (n) = T(1) + T(n-1) + n$

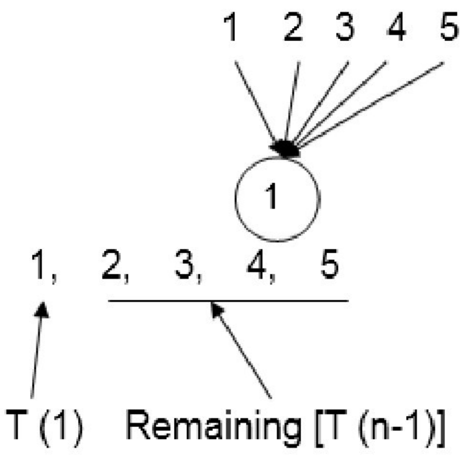
$T\ (1)$ is time taken by pivot element.

$T\ (n-1)$ is time taken by remaining element except for pivot element.

N: the number of comparisons required to identify the exact position of itself (every element)

If we compare first element pivot with other, then there will be 5 comparisons.

It means there will be n comparisons if there are n items.



Relational Formula for Worst Case:

T (n) =T (1) + T (n-1) + n………………… (1)

T (n-1) =T (1) +T (n-1-1) + (n-1)

By putting (n-1) in place of n in equation1

Put T (n-1) in equation1

T(n)=T(1)+T(1)+(T(n-2)+(n-1)+n…………………(ii)

T (n) =2T (1) +T (n-2) + (n-1) +n

T (n-2) =T (1) +T (n-3) + (n-2)

By putting (n-2) in place of n in equation1

Put T (n-2) in equation (ii)

T(n)=2T(1)+T(1)+T(n-3)+(n-2)+(n-1)+n

T(n)=3T(1)+T(n-3)+)+(n-2)+(n-1)+n

T (n-3) =T (1) +T (n-4) +n-3

By putting (n-3) in place of n in equation1

T(n)=3T(1)+T(1)+T(n-4)+(n-3)+(n-2)+(n-1)+n

=4T(1)+T(n-4))+(n-3)+(n-2)+(n-1)+n………………………(iii)

Note: for making T (n-4) as T (1) we will put (n-1) in place of '4' and if we put (n-1) in place of 4 then we have to put (n-2) in place of 3 and (n-3) In place of 2 and so on.

T(n)=(n-1) T (1) + T(n-(n-1)) +(n-(n-2)) +(n-(n-3)) +(n-(n-4)) +n

T (n) = (n-1) T (1) + T (1) + 2 + 3 + 4+…………n

T (n) = (n-1) T (1) +T (1) +2+3+4+………..+n+1-1

[Adding 1 and subtracting 1 for making AP series]

Stopping Condition: T (1) =0

Because at last there is only one element left and no comparison is required.

$$T(n) = (n-1)(0) + 0 + \frac{n(n+1)}{2} - 1$$

$$T(n) = \frac{n^2 + n - 2}{2}$$

Avoid all the terms expect higher terms n^2

$$T(n) = O(n^2)$$

Worst Case Complexity of Quick Sort is T (n) =O (n^2)

Randomized Quick Sort [Average Case]:

Generally, we assume the first element of the list as the pivot element. In an average Case, the number of chances to get a pivot element is equal to the number of items.

Let total time taken =T (n)

For eg: In a given list

p 1, p 2, p 3, p 4............pn

If p 1 is the pivot list then we have 2 lists.

I.e. T (0) and T (n-1)

If p2 is the pivot list then we have 2 lists.

I.e. T (1) and T (n-2)

p 1, p 2, p 3, p 4............pn

If p3 is the pivot list then we have 2 lists.

I.e. T (2) and T (n-3)

p 1, p 2, p 3, p 4............p n

So in general if we take the Kth element to be the pivot element.

Then,

$$T\ (n) = \sum_{K=1}^{n} \quad T(K-1) + T(n-k)$$

Pivot element will do n comparison and we are doing average case so,

$$T\ (n) = \underbrace{n+1}_{\text{N comparisons}} + \underbrace{\frac{1}{n}\ (\sum_{K=1}^{n} \quad T(K-1) + T(n-k))}_{\text{Average of n elements}}$$

So Relational Formula for Randomized Quick Sort is:

$$T\ (n) = n+1 + \frac{1}{n}\ ((\sum_{K=1}^{n} \quad T(K-1) + T(n-k))$$

= n+1 + 1/n (T(0)+T(1)+T(2)+...T(n-1)+T(n-2)+T(n-3)+...T(0))

= n+1 +1/n (T(0)+T(1)+T(2)+...T(n-2)+T(n-1))

n T (n) = n (n+1) +2 (T(0)+T(1)+T(2)+...T(n-1)........eq 1

Put n=n-1 in eq 1

(n -1) T (n-1) = (n-1) n+2 (T(0)+T(1)+T(2)+...T(n-2)......eq2

From eq1 and eq 2

n T (n) - (n-1) T (n-1)= n(n+1)-n(n-1)+2 (T(0)+T(1)+T(2)+?T(n-2)+T(n-1))-2(T(0)+T(1)+T(2)+...T(n-2))

n T(n)- (n-1) T(n-1)= n[n+1-n+1]+2T(n-1)

n T(n)=[2+(n-1)]T(n-1)+2n

n T(n)= n+1 T(n-1)+2n

$$\frac{n}{n+1}T(n) = \frac{2n}{n+1} + T(n-1) \qquad \text{[Divide by n+1]}$$

$$\frac{1}{n+1}\,T(n) = \frac{2}{n+1} + \frac{T(n-1)}{n} \qquad \text{[Divide by n]} \ldots\ldots\ldots\ldots\ldots\ldots\text{eq 3}$$

Put n=n-1 in eq 3

$$\frac{1}{n}T\,(n\text{-}1) = \frac{2}{n} + \frac{T(n-2)}{n-1}\ldots\ldots\ldots\ldots\text{eq4}$$

Put 4 eq in 3 eq

$$\frac{T(n)}{n+1} = \frac{2}{n+1} + \frac{2}{n} + \frac{T(n-2)}{n-1}\ldots\ldots\ldots\ldots\text{eq5}$$

Put n=n-2 in eq 3

$$\frac{T(n-2)}{n-1} = \frac{2}{n-1} + \frac{2}{n} + \frac{T(n-3)}{n-2}\ldots\ldots\ldots\ldots\text{eq6}$$

Put 6 eq in 5 eq

$$\frac{T(n)}{n+1} = \frac{2}{n+1} + \frac{2}{n} + \frac{2}{n-1} + \frac{T(n-3)}{n-2}\ldots\ldots\ldots\text{eq7}$$

Put n=n-3 in eq 3

$$\frac{T(n-3)}{n-2} = \frac{2}{n-2} + \frac{T(n-4)}{n-3}\ldots\ldots\ldots\ldots\ldots\text{eq8}$$

Put 8 eq in 7 eq

$$\frac{T(n)}{n+1} = \frac{2}{n+1} + \frac{2}{n} + \frac{2}{n-1} + \frac{2}{n-2} + \frac{T(n-4)}{n-3}\ldots\ldots\ldots\ldots\text{eq9}$$

2 terms of $\frac{2}{n+1}+\frac{2}{n}=T(n-2)$

3 terms of $\frac{2}{n+1}+\frac{2}{n}+\frac{2}{n-1}=T(n-3)$

4 terms of $\frac{2}{n+1}+\frac{2}{n}+\frac{2}{n-1}+\frac{2}{n-2}$ =T(n-4)

From 3eq, 5eq, 7eq, 9 eq we get

$$\frac{T(n)}{n+1}=\frac{2}{n+1}+\frac{2}{n}+\frac{2}{n-1}+\dots\dots\dots+\frac{2}{3}+\frac{T(n-(n-1))}{n-(n-2)}\dots\dots\dots \text{eq10}$$

From 3 eq $\frac{T(n)}{n+1}=\frac{2}{n+1}+\frac{T(n-1)}{n}$

Put n=1

$$\frac{T(1)}{2}=\frac{2}{2}+\frac{T(0)}{1}$$

$$\frac{T(1)}{2}=1$$

From 10 eq

$$\frac{T(n)}{n+1}=\frac{2}{n+1}+\frac{2}{n}+\frac{2}{n-1}+\dots\dots\dots\dots+\frac{2}{3}+\frac{T(1)}{2}$$

$$=\frac{2}{n+1}+\frac{2}{n}+\frac{2}{n-1}+\dots\dots\dots\dots+\frac{2}{3}+1$$

Multiply and divide the last term by 2

$$=\frac{2}{n+1}+\frac{2}{n}+\frac{2}{n-1}+\ldots\ldots\ldots\ldots\ldots+\frac{2}{3}+2\times\frac{1}{2}$$

$$=2\left[\frac{1}{2}+\frac{1}{3}+\frac{1}{4}\ldots\ldots+\frac{1}{n}+\frac{1}{n+1}\right]$$

$$=2\sum_{2\le k\le n+1}^{n}\frac{1}{k}=2\int_{2}^{n+1}\frac{1}{k}$$

Multiply & divide k by n

$$=2\int_{2}^{n+1}\frac{1}{\frac{k.n}{n}}$$

Put $\frac{k}{n}=x$ and $\frac{1}{n}=dx$

$$\frac{T(n)}{n+1}=2\int_{2}^{n+1}\frac{1}{x}\,dx$$

$$=2\log x\int_{2}^{n+1}$$

NOTE: $\int\frac{1}{x}dx=\log x$

=2[log (n+1)-log2]

T (n) =2(n+1) [log (n+1)-log2]

Ignoring Constant we get

T (n) =n log n

$$\boxed{\textbf{T (n) =O (n log n)}}$$

Is the average case complexity of quick sort for sorting n elements.

3. Quick Sort [Best Case]: In any sorting, best case is the only case in which we don't make any comparison between elements that is only done when we have only one element to sort.

Method Name	Equation	Stopping Condition	Complexities
1.Quick Sort[Worst Case]	T(n)=T(n-1)+T(0)+n	T(1)=0	T(n)=n^2
2.Quick Sort[Average Case]	T(n)=n+1 + $\frac{1}{n}$ $(\sum_{K=1}^{n} T(k-1)+T(n-k))$		T(n)=nlogn

Table 2.1

2.5 Heap Sort

A heap is really nothing more than a binary tree with some additional rules that it has to follow: first, it must always have a heap structure, where all the levels of the binary tree are filled up, from left to right, and second, it must either be ordered as a max heap or a min heap. We'll use min heap as an example.

A heap sort algorithm is a sorting technique that leans on binary heap data structures. Because we know that heaps must always follow a specific order, we can leverage that property and use that to find the smallest, minimum value element, and sequentially sort elements by selecting the root node of a heap, and adding it to the end of the array.

Here is the heap-sort pseudocode:

```
HeapSort(A[1…n]):
    1. H = buildHeap(A[1…n])
    2. for i = 1 to n do:
    3. A[i] = extract-min(H)
```

To start, we have an unsorted array. The first step is to take that array and turn it into a heap; in our case, we'll want to turn it into a min heap. So, we have to transform and build a min heap out of our unsorted array data. Usually, this is encapsulated by a single function, which might be named something like buildHeap.

Here is the buildHeap pseudocode:

```
buildHeap():
    1. initially H is empty
    2. for i = 1 to n do:
    3. Add(H, A[i])
```

Once we have our array data in a min heap format, we can be sure that the smallest value is at the root node of the heap. Remember that, even though the entire heap won't be sorted, if we have built our min heap correctly and without any mistakes, every single parent node in our heap will be smaller in value than its children. So, we'll move the smallest value — located at the root node — to the end of the heap by swapping it with the last element.

Now, the smallest item in the heap is located at the last node, which is great. We know that it is in its sorted position, so it can be removed from the heap completely. But, there's still one more step: making sure that the new root node element is in the correct place! It's highly unlikely that the item that we swapped into the root node position is in the right location, so we'll move down the root node item down to its correct place, using a function that's usually named something like heapify.

Here is the extract-min and heapify pseudocode:

extract-min(H):

```
1. min = H[1]
2. H[1] = H[H.size()]
3. decrease H.size() by 1
4. heapify(H, 1)
5. return min
```

heapify():

```
1. n = H.size()
2. while (LeftChild(i) <= n and H[i] > H[LeftChild(i)]) or (RightChild(i)
   <= n and H[i] > H[RightChild(i)]) do:
3. if (H[LeftChild(i)] < H[RightChild(i)]) then:
4. j = LeftChild(i)
5. else:
6. j = RightChild(i)
7. swap entries H[i] and H[j]
8. i = j
```

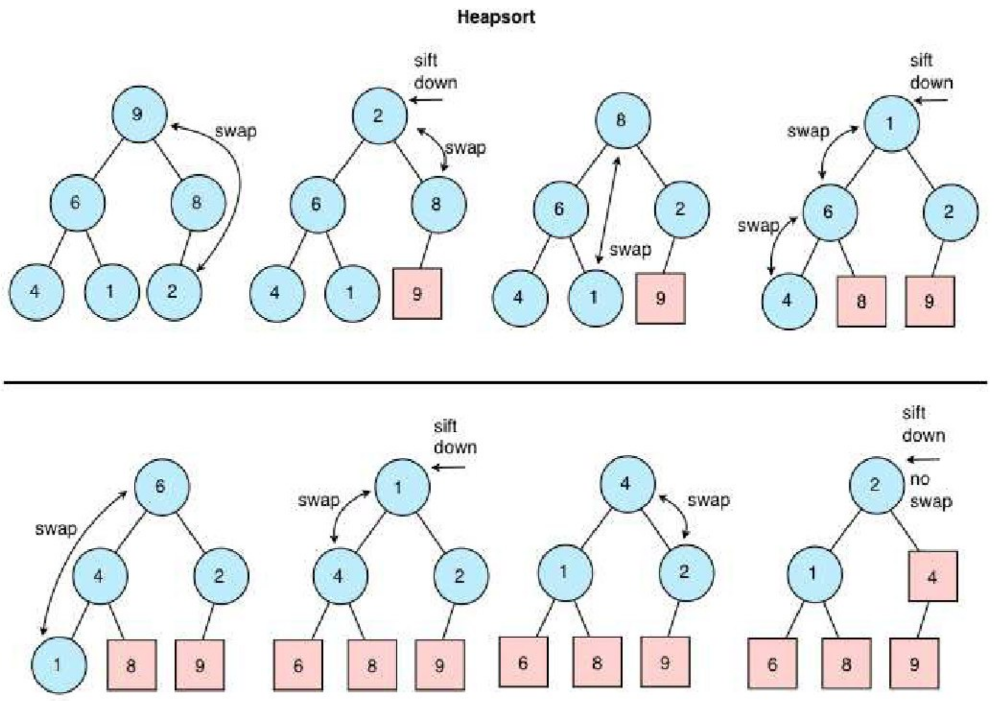


Figure 2.4 Heap sort

And that's basically it! The algorithm continues to repeat these steps until the heap is down to just one single node. At that point, it knows that all the elements in the unsorted array are in their sorted positions and that the last node remaining will end up being the first element in the sorted array. The total running time of this algorithm is O(n log n).

2.6 Strassen's Matrix Multiplication

Strassen in 1969 which gives an overview that how we can find the multiplication of two 2*2-dimension matrix by the brute-force algorithm. But by using divide and conquer technique the overall complexity for multiplication two matrices is reduced. This happens by decreasing the total number if multiplication performed at the expenses of a slight increase in the number of addition.

For multiplying the two 2*2 dimension matrices Strassen's used some formulas in which there are seven multiplications and eighteen additions, subtraction, and in brute force algorithm, there is eight multiplications and four additions. The utility of Strassen's formula is shown by its asymptotic superiority when order n of matrix reaches infinity. Let us consider two matrices A and B, n*n dimension, where n is a power of two. It can be observed that we can contain four n/2*n/2 submatrices from A, B and their product C. C is the resultant matrix of A and B.

Procedure of Strassen matrix multiplication

There are some procedures:

1. Divide a matrix of order of 2*2 recursively till we get the matrix of 2*2.
2. Use the previous set of formulas to carry out 2*2 matrix multiplication.
3. In this eight multiplication and four additions, subtraction are performed.
4. Combine the result of two matrixes to find the final product or final matrix.

Formulas for Stassen's matrix multiplication

In Strassen's matrix multiplication there are seven multiplications and four additions, subtraction in total.

1. $D1 = (a11 + a22)(b11 + b22)$
2. $D2 = (a21 + a22).b11$
3. $D3 = (b12 - b22).a11$
4. $D4 = (b21 - b11).a22$
5. $D5 = (a11 + a12).b22$
6. $D6 = (a21 - a11).(b11 + b12)$
7. $D7 = (a12 - a22).(b21 + b22)$

$C11 = d1 + d4 - d5 + d7$

$C12 = d3 + d5$

$C21 = d2 + d4$

C22 = d1 + d3 – d2 – d6

Algorithm for Strassen's matrix multiplication

Algorithm Strassen (n, a, b, d)

```
begin
        If n = threshold, then compute
                C = a * b is a conventional matrix.
        Else
                Partition a into four sub matrices a11, a12, a21, a22.
                Partition b into four sub matrices b11, b12, b21, b22.
                Strassen ( n/2, a11 + a22, b11 + b22, d1)
                Strassen ( n/2, a21 + a22, b11, d2)
                Strassen ( n/2, a11, b12 – b22, d3)
                Strassen ( n/2, a22, b21 – b11, d4)
                Strassen ( n/2, a11 + a12, b22, d5)
                Strassen (n/2, a21 – a11, b11 + b22, d6)
                Strassen (n/2, a12 – a22, b21 + b22, d7)

                C = d1+d4-d5+d7     d3+d5
                d2+d4          d1+d3-d2-d6

        end if

        return (C)
end.
```

Code for Strassen matrix multiplication

```
#include <stdio.h>

int main()
{
        int a[2][2],b[2][2],c[2][2],i,j;
        int m1,m2,m3,m4,m5,m6,m7;
```

```
printf("Enter the 4 elements of first matrix: ");
for(i=0;i<2;i++)
          for(j=0;j<2;j++)
                    scanf("%d",&a[i][j]);

printf("Enter the 4 elements of second matrix: ");
for(i=0;i<2;i++)
          for(j=0;j<2;j++)
                    scanf("%d",&b[i][j]);

printf("\nThe first matrix is\n");
for(i=0;i<2;i++)
{
          printf("\n");
          for(j=0;j<2;j++)
                    printf("%d\t",a[i][j]);
}

printf("\nThe second matrix is\n");
for(i=0;i<2;i++)
{
          printf("\n");
          for(j=0;j<2;j++)
                    printf("%d\t",b[i][j]);
}

m1= (a[0][0] + a[1][1])*(b[0][0]+b[1][1]);
m2= (a[1][0]+a[1][1])*b[0][0];
m3= a[0][0]*(b[0][1]-b[1][1]);
m4= a[1][1]*(b[1][0]-b[0][0]);
m5= (a[0][0]+a[0][1])*b[1][1];
```

```
        m6= (a[1][0]-a[0][0])*(b[0][0]+b[0][1]);
        m7= (a[0][1]-a[1][1])*(b[1][0]+b[1][1]);

        c[0][0]=m1+m4-m5+m7;
        c[0][1]=m3+m5;
        c[1][0]=m2+m4;
        c[1][1]=m1-m2+m3+m6;

        printf("\nAfter multiplication using \n");
        for(i=0;i<2;i++)
        {
                printf("\n");
                for(j=0;j<2;j++)
                        printf("%d\t",c[i][j]);
        }

        return 0;
}
```

Output:

```
Enter the 4 elements of first matrix:
   5 6 1 7
   Enter the 4 element of second matrix:
   6 2 8 7
   The first matrix is
   5      6
   1      7
   The second matrix is
   6      2
   8      7
   After multiplication
   78       52
```

Complexity: The time complexity is O($N^{2.8074}$).

2.7 Summary

In this unit you have learnt about General method, Applications-Binary search, Finding the maximum and minimum. You have also learnt about Quick sort, Heap sort, and Strassen's Matrix Multiplication

- A Binary Search tree is organized in a Binary Tree. Such a tree can be defined by a linked data structure in which a particular node is an object.
- In addition to a key field, each node contains field left, right, and p that point to the nodes corresponding to its left child, its right child, and its parent, respectively.
- If a child or parent is missing, the appropriate field contains the value NIL. The root node is the only node in the tree whose parent field is Nil.
- Like merge sort, quicksort uses divide-and-conquer, and so it's a recursive algorithm. The way that quicksort uses divide-and-conquer is a little different from how merge sort does. In merge sort, the divide step does hardly anything, and all the real work happens in the combine step.
- Heap sort has the time complexity of a 'divide and conquer' algorithm (such as quick sort), but it does not behave like a divide and conquer algorithm. Because it splits the data into a 'sorted' section and an 'unsorted' section, it is really a kind of selection sort.

2.8 Review Questions

Q1. What are the general method of divide and Conquer? Explain with example.

Q2. How to optimize Quick-Sort so that it takes O (Log n) extra space in worst case?

Q3. Define Binary Search Algorithm? How binary search algorithm works?

Q4. Findingthe maximum and minimum value in an array with suitable example.

Q5. What is running time of Strassen algorithm for matrix multiplication?

UNIT-3 Sorting in Linear Time

Structure

3.0 Introduction

This is the last unit of this block. In this unit you will learn about sorting in linear time. In the section 3.1 you will study about Lower bounds for sorting. In the section 3.2 you will know aboutcounting sort. Section 3.3 defines aboutRadix sort. You will learn about Bucket sort in the section 3.4, Medians and Order Statistics explain in the section 3.5. You will study aboutMinimum and maximum in the section 3.6. Section 3.7 and 3.8 provide summary and review questions respectively.

Objective

After studying this unit, you should be able to define:

- Lower bounds for sorting
- Counting sort
- Radix sort
- Bucket sort
- Medians and Order Statistics
- Minimum and maximu

3.1 Lower bounds for sorting

By now you have seen many analyses for algorithms. These analyses generally define the upper and lower bounds for algorithms in their worst and average cases. For many of the algorithms presented so far, analysis has been easy. This module considers a more difficult task: An analysis for the cost of a problem as opposed to an algorithm. The upper bound for a problem can be defined as the asymptotic cost of the fastest known algorithm. The lower bound defines the best possible cost for any algorithm that solves the problem, including algorithms not yet invented. Once the upper and lower bounds for the problem meet, we know that no future algorithm can possibly be (asymptotically) more efficient.

A simple estimate for a problem's lower bound can be obtained by measuring the size of the input that must be read and the output that must be written. Certainly no algorithm can be more efficient than the problem's I/O time. From this we see that the sorting problem cannot be solved by any algorithm in less than $\Omega(n)$ time because it takes at least n steps to read and write the n values to be sorted. Alternatively, any sorting algorithm must at least look at every input value to recognize whether the input values are in sorted order. So, based on our current knowledge of sorting algorithms and the size of the input, we know that the problem of sorting is bounded by $\Omega(n)$ and $O(n\log n)$.

Computer scientists have spent much time devising efficient general-purpose sorting algorithms, but no one has ever found one that is faster than O(nlogn) in the worst or average cases. Should we keep searching for a faster sorting algorithm? Or can we prove that there is no faster sorting algorithm by finding a tighter lower bound?

This section presents one of the most important and most useful proofs in computer science: No sorting algorithm based on key comparisons can possibly be faster than Ω(nlogn) in the worst case. This proof is important for three reasons. First, knowing that widely used sorting algorithms are asymptotically optimal is reassuring. In particular, it means that you need not bang your head against the wall searching for an O(n) sorting algorithm. (Or at least not one that is in any way based on key comparisons. But it is hard to imagine how to sort without any comparisons. Even Radix Sort is does comparisons, though in quite a different way.) Second, this proof is one of the few non-trivial lower-bounds proofs that we have for any problem; that is, this proof provides one of the relatively few instances where our lower bound is tighter than simply measuring the size of the input and output. As such, it provides a useful model for proving lower bounds on other problems. Finally, knowing a lower bound for sorting gives us a lower bound in turn for other problems whose solution could be made to work as the basis for a sorting algorithm. The process of deriving asymptotic bounds for one problem from the asymptotic bounds of another is called a reduction.

Except for the Radix Sort and Binsort, all of the sorting algorithms we have studied make decisions based on the direct comparison of two key values. For example, Insertion Sort sequentially compares the value to be inserted into the sorted list until a comparison against the next value in the list fails. In contrast, Radix Sort has no direct comparison of key values. All decisions are based on the value of specific digits in the key value, so it is possible to take approaches to sorting that do not involve direct key comparisons. Of course, Radix Sort in the end does not provide a more efficient sorting algorithm than comparison-based sorting. Thus, empirical evidence suggests that comparison-based sorting is a good approach.

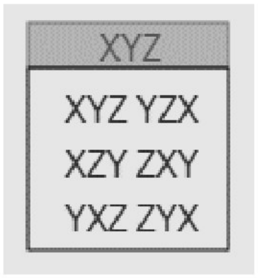
Actually, the truth is stronger than this statement implies. In reality, Radix Sort relies on comparisons as well and so can be modelled by the technique used in this section. The result is an Ω(nlogn) bound in the general case even for algorithms that look like Radix Sort.

The proof that any comparison sort requires Ω(nlogn) comparisons in the worst case is structured as follows. First, comparison-based decisions can be modelled as the branches in a tree. This means that any sorting algorithm based on comparisons between records can be viewed as a binary tree whose nodes correspond to the comparisons, and whose branches correspond to the possible outcomes. Next, the minimum number of leaves in the resulting tree is shown to be the factorial of n. Finally, the minimum depth of a tree with n! leaves is shown to be in Ω(nlogn).

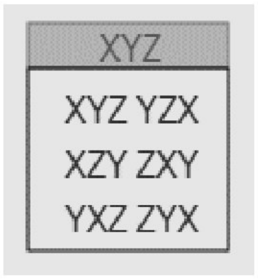
Before presenting the proof of an Ω(nlogn) lower bound for sorting, we first must define the concept of a decision tree. A decision tree is a binary tree that can model the processing for any algorithm that makes binary decisions. Each (binary) decision is represented by a branch in the tree. For the purpose of modelling sorting algorithms, we count all comparisons of key values as decisions. If two keys are compared and the first is less than the second, then this is modelled as a left branch in the decision tree. In the case where the first value is greater than the second, the algorithm takes the right branch.

We will illustrate the sorting lower bound proof by showing the decision tree that models the processing of Insertion sort on an array of 3 element XYZ.

There are 6 possible permutations of the array values XYZ, only one of them represents the sorted array.

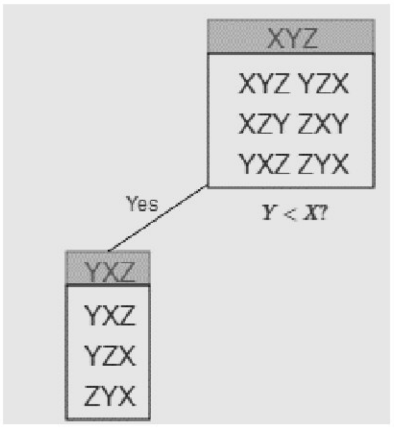


The first step in the insertion sort is to compare the second elements Y with the first element X.

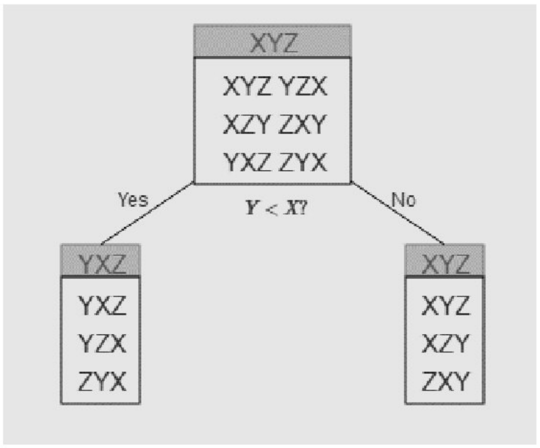


Y<X?

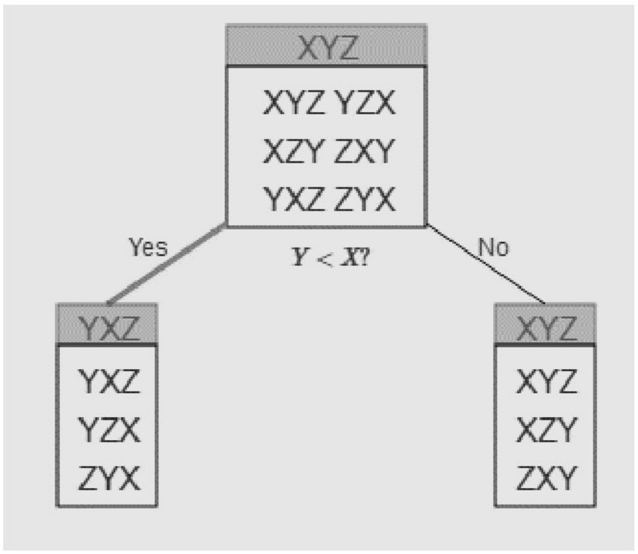
If Y is less than X, the two values are swapped, and then we will end up having only three permutations.



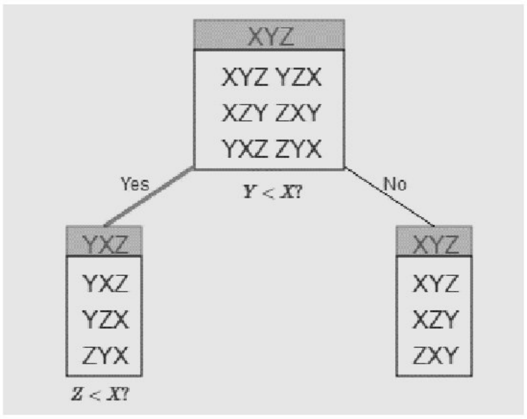
If Y is not less than X, no swap will occur and we will end up having only three permutations.



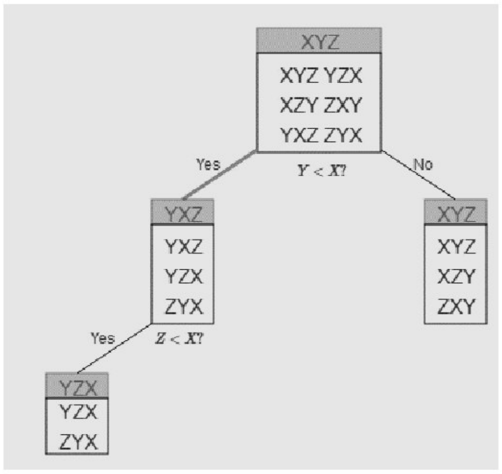
Let us assume for the moment that Y is less than X and so the left branch is taken.



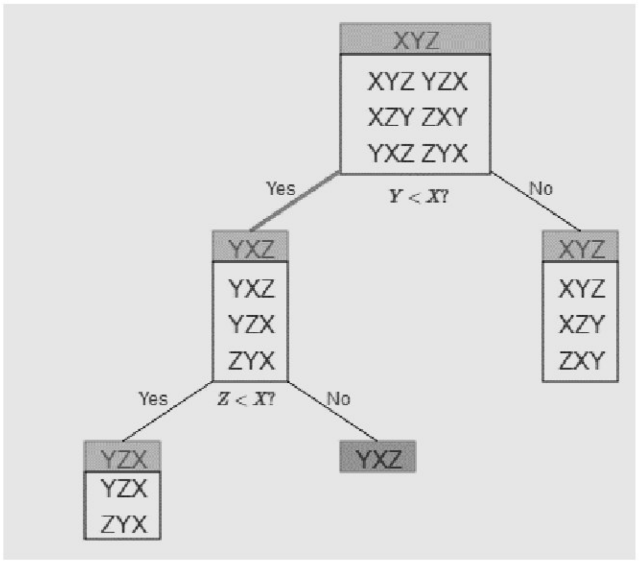
The third element Z is compared to the second element X



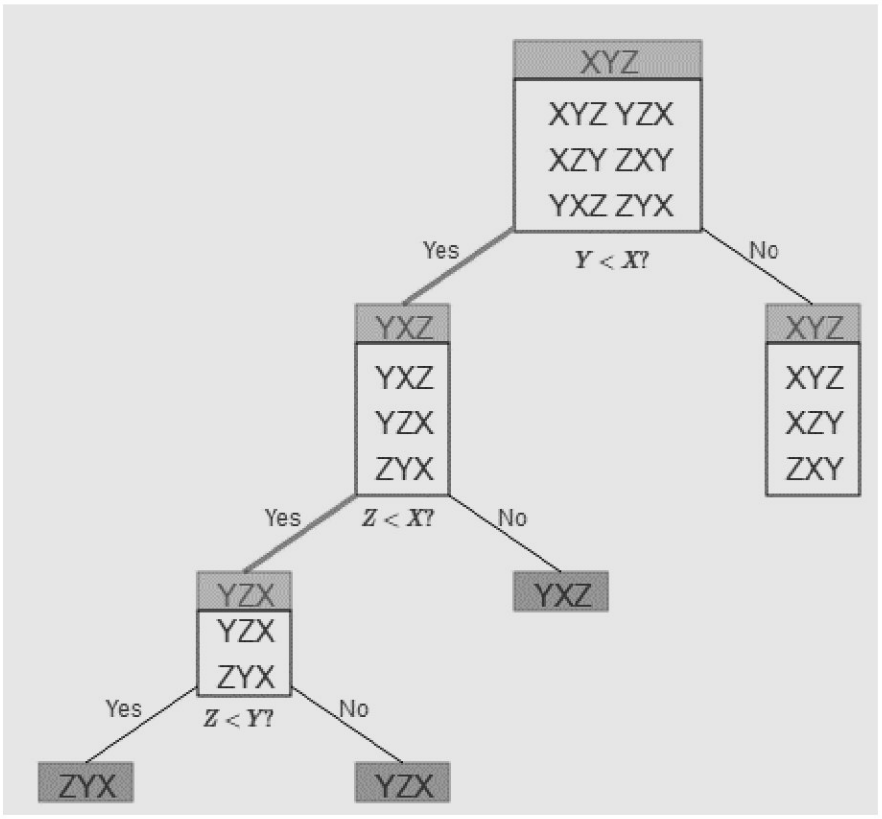
Again there are two possibilities, if Z is less than X then these items should be swapped and we will end up having 2 permutations.



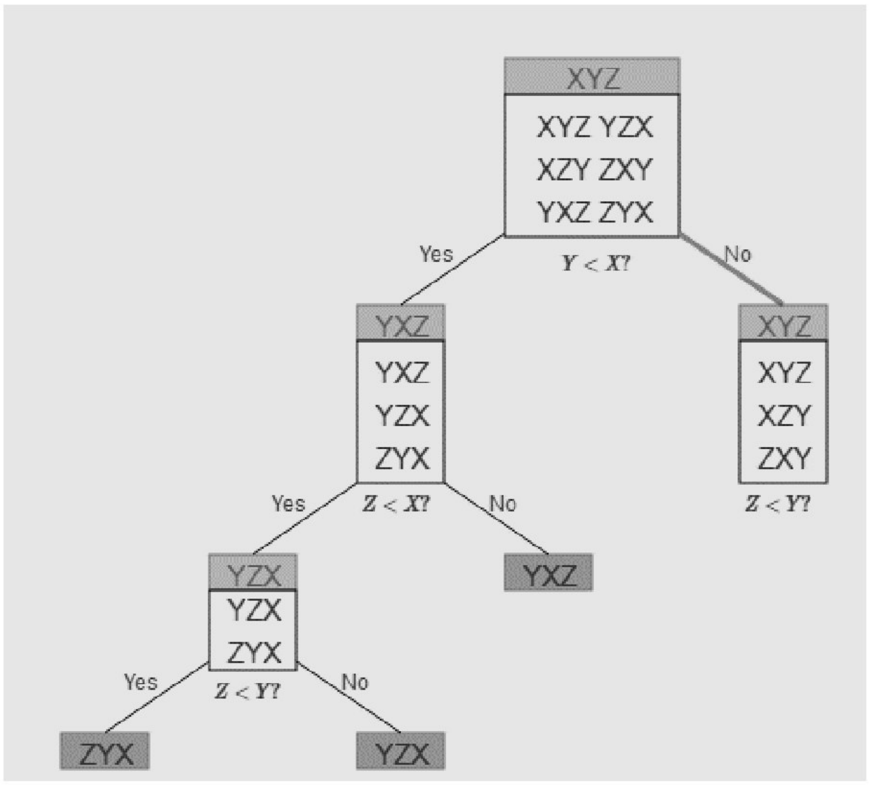
If Z is not less than X, no swap will occur and we will end up only one permutation and insertion sort is complete.



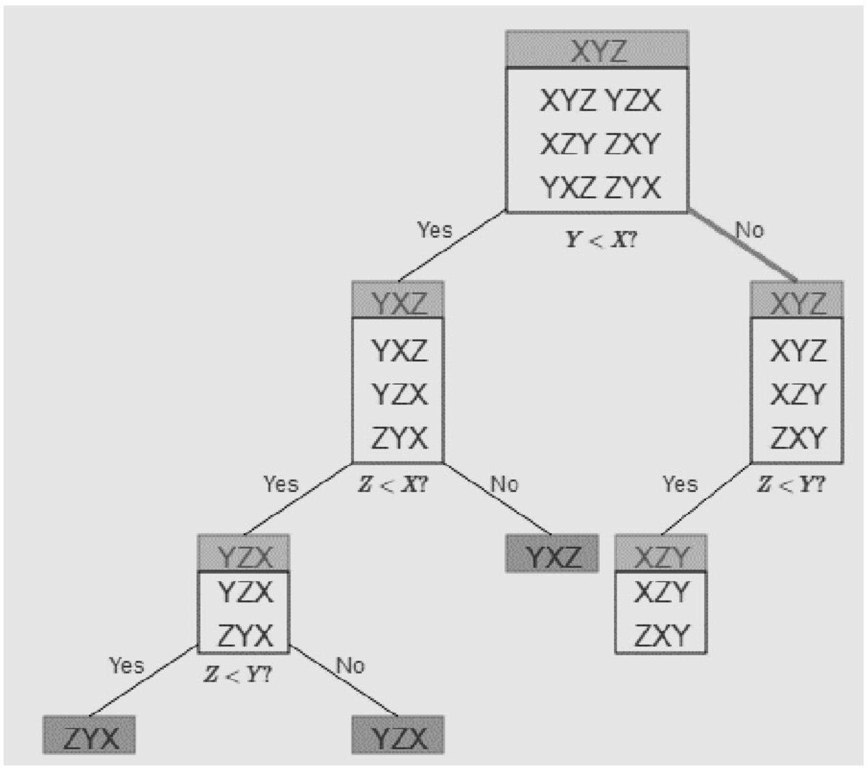
If the left branch was taken, Z is the compared to the Y and insertion sort will be completed regardless of the comparison result.



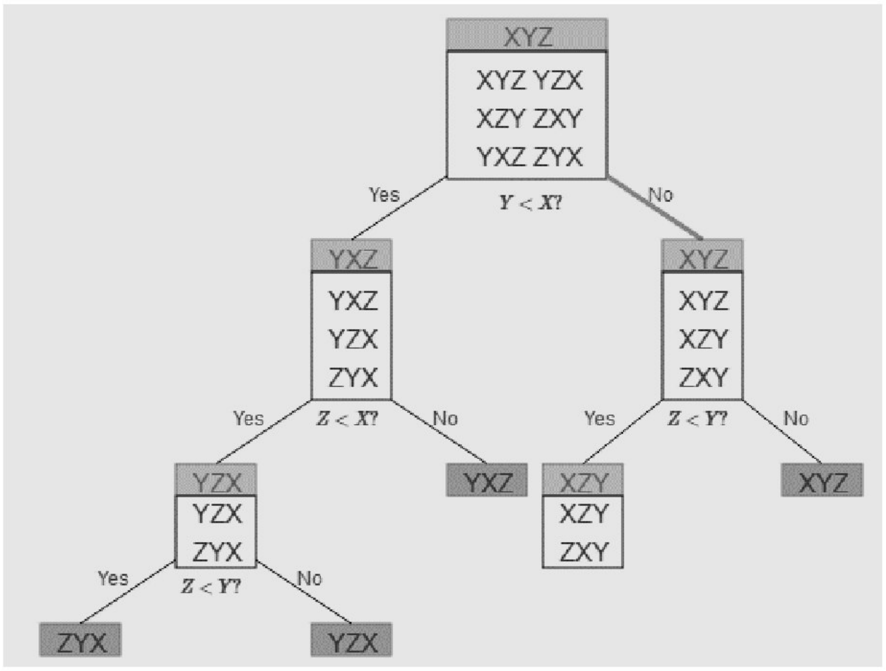
In the first comparison, if the right branch was taken, the third element Z is then compared to the second element Y.



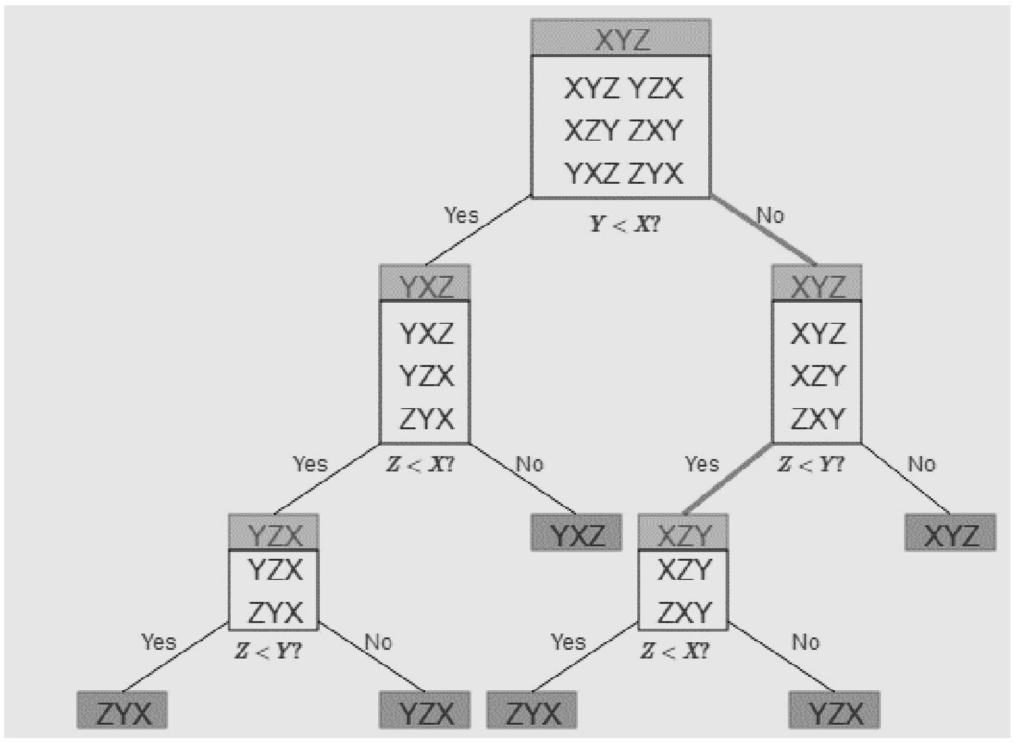
If Z is less than Y, the two values are swapped, and then we will end up having only 2 permutations.



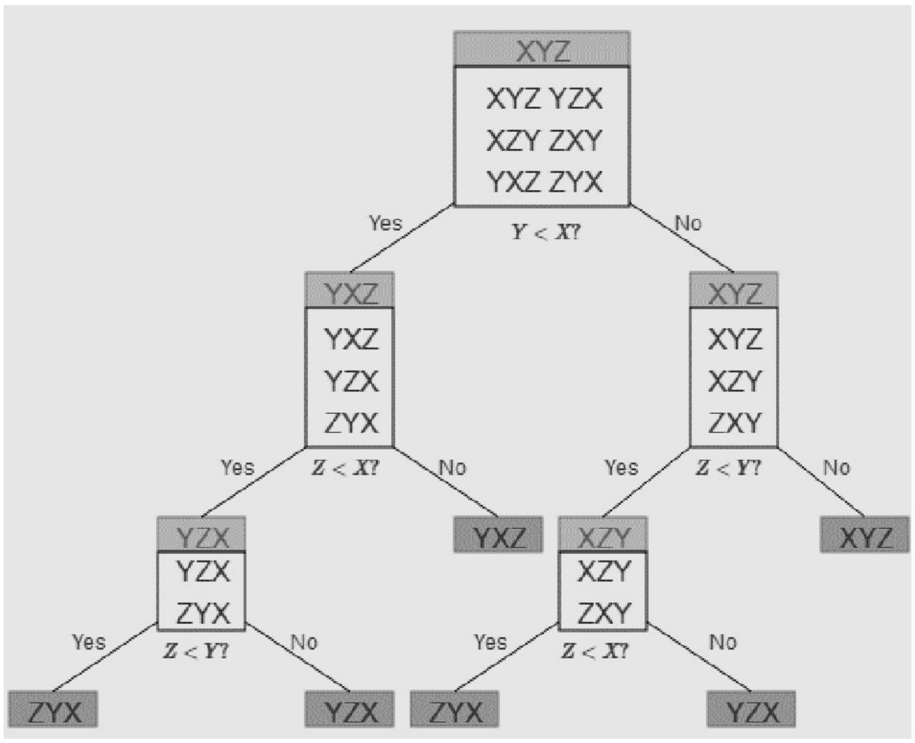
If Z is not less than Y, no swap will occur and we will end up having only 1 permutation and insertion sort is complete.



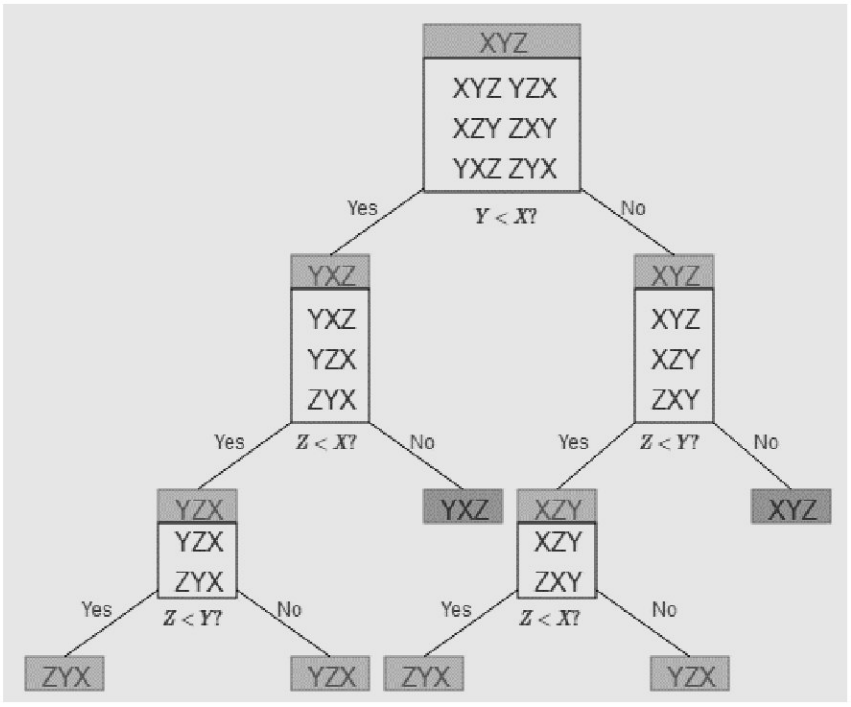
If the left branch was taken, Z is than compared to X and insertion sort will completed regardless of the comparison result.



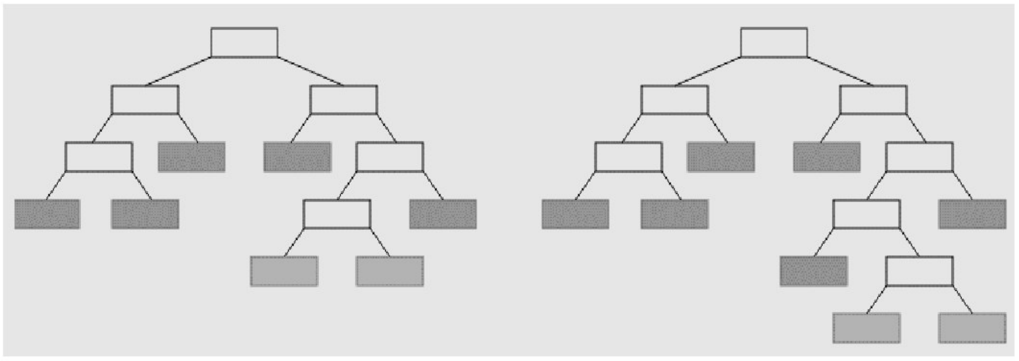
The cost of the algorithm (In worst, best and average cases) is determined by the depth of the nodes indication the number of comparisons required to reach that node.



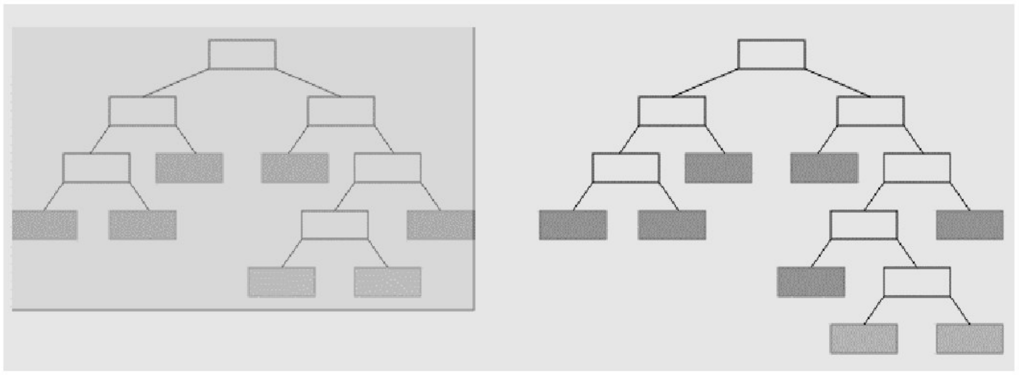
The worst case of the algorithm is determined by the depth of the deepest node(s).



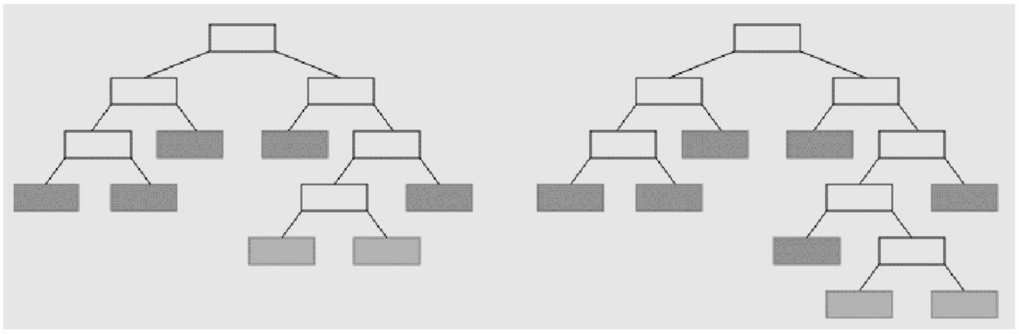
Each sorting algorithm has its own decision tree with different maximum depths.



The best algorithm (in the worst case) will be the one with the shallowest deepest node.



The depth of the shallowest deepest nodes depends in part on the number of nodes in the decision tree.



The maximum number of nodes that must be in the decision tree for any comparison based sorting algorithm for n values should be n! Since the decision tree must have at least n! Leaf nodes, since at least one leaf node will correspond to each input permutation.

A tree with node requires a minimum of logn levels. Because there is at least n! nodes in the tree. We know the tree must have Ω (log n!) levels.

According, the decision tree for any comparison-based sorting algorithm must have at least one node that is Ω(nlogn) levels deep. This deepest node represents the algorithm's worst case. So in the worst case, any such sorting algorithm must require Ω (nlogn) comparisons.

3.2 Counting sort

Counting Sort Algorithm is an efficient sorting algorithm that can be used for sorting elements within a specific range. This sorting technique is based on the frequency/count of each element to be sorted and works using the following algorithm-

- Input: Unsorted array A[] of n elements
- Output: Sorted arrayB[]

Step 1: Consider an input array A having n elements in the range of 0 to k, where n and k are positive integer numbers. These n elements have to be sorted in ascending order using the counting sort technique. Also note that A[] can have distinct or duplicate elements

Step 2: The count/frequency of each distinct element in A is computed and stored in another array, say count, of size k+1. Let u be an element in A such that its frequency is stored at count[u].

Step 3: Update the count array so that element at each index, say i, is equal to –

$$count[i]=\sum count[u] \text{ where } 0 \le u \le i$$

Step 4: The updated count array gives the index of each element of array A in the sorted sequence. Assume that the sorted sequence is stored in an output array, say B, of size n.

Step 5: Add each element from input array A to B as follows:

a) Set i=0 and t = A[i]
b) Add t to B[v] such that v = (count[t]-1).
c) Decrement count[t] by 1
d) Increment i by 1

Repeat steps (a) to (d) till i = n-1

Step 6: Display B since this is the sorted array

Example of Pictorial Representation of Counting Sort:

Let us trace the above algorithm using an example:

Consider the following input array A to be sorted. All the elements are in range 0 to 9

Step 1: Initialize an auxiliary array, say count and store the frequency of every distinct element. Size of count is 10 (k+1, such that range of elements in A is 0 to k)

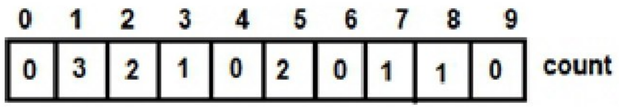


Figure 3.1: count array

Step 2: Using the formula, updated count array is -

$$count[i]=\sum count[u] \text{ where } 0 \le u \le i$$

Figure 3.2: Formula for updating count array

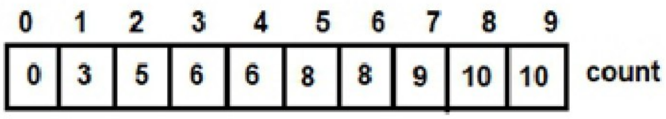


Figure 3.3: Updated count array

Step 3: Add elements of array A to resultant array B using the following steps:

For, i=0, t=1, count[1]=3, v=2. After adding 1 to B[2], count[1]=2 and i=1

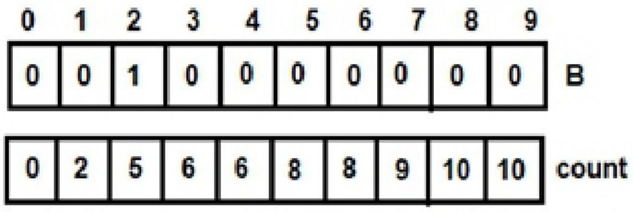


Figure 3.4: For i=0

For i=1, t=3, count[3]=6, v=5. After adding 3 to B[5], count[3]=5 and i=2

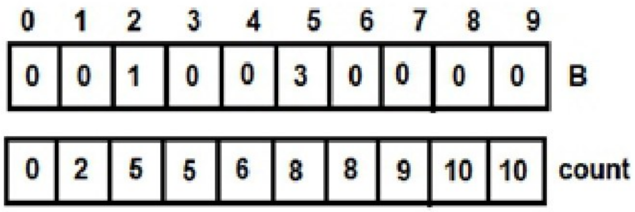


Figure 3.5: For i=1

For i=2, t=2, count[2]= 5, v=4. After adding 2 to B[4], count[2]=4 and i=3

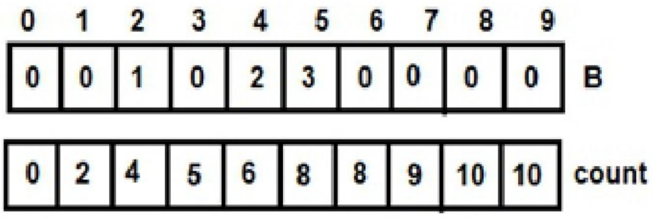


Figure 3,6: For i=2

For i=3, t=8, count[8]= 10, v=9. After adding 8 to B[9], count[8]=9 and i=4

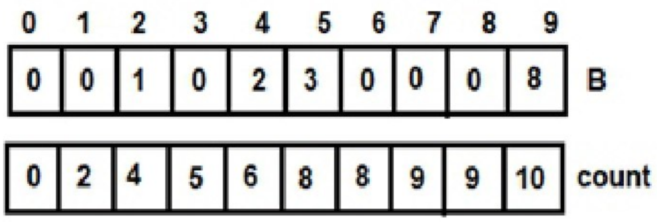


Figure 3.7: For i=3

On similar lines, we have the following:

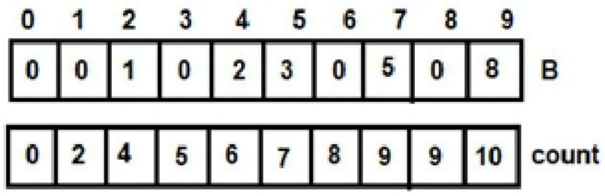


Figure 3.8: For i=4

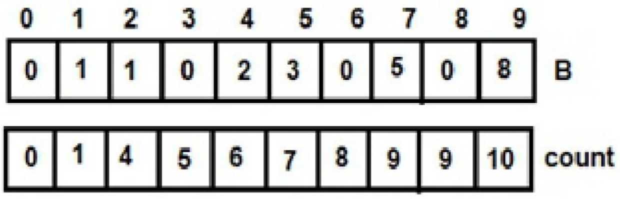


Figure 3.9: For i=5

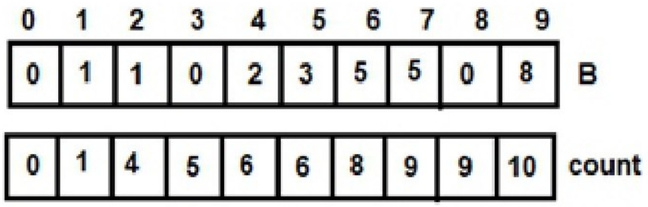


Figure 3.10: For i=6

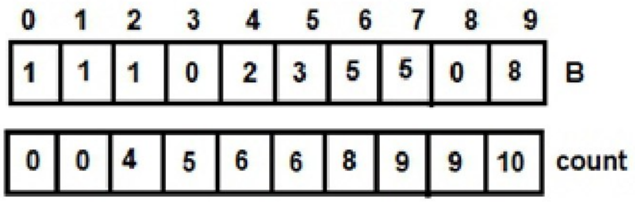


Figure 3.11: For i=7

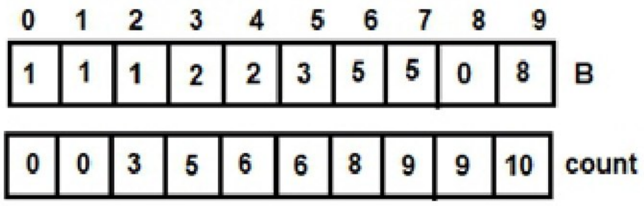


Figure 3.12: For i=8

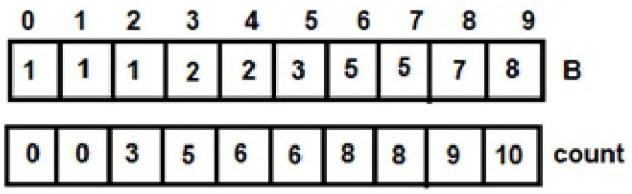


Figure 3.13: For i=9

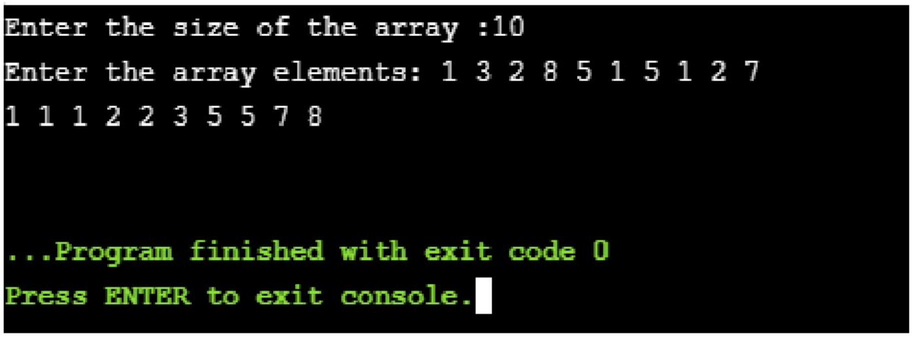
Thus, array B has the sorted list of elements.

Program for Counting Sort Algorithm

```
#include <stdio.h>
void counting_sort(int A[], int k, int n)
{
    int i, j;
    int B[15], C[100];
    for (i = 0; i <= k; i++)
        C[i] = 0;
    for (j = 1; j <= n; j++)
        C[A[j]] = C[A[j]] + 1;
    for (i = 1; i <= k; i++)
        C[i] = C[i] + C[i-1];
    for (j = n; j >= 1; j--)
    {
        B[C[A[j]]] = A[j];
        C[A[j]] = C[A[j]] - 1;
    }
    printf("The Sorted array is : ");
    for (i = 1; i <= n; i++)
        printf("%d ", B[i]);
}
/*  End of counting_sort()  */
/*  The main() begins  */
int main()
{
    int n, k = 0, A[15], i;
    printf("Enter the number of input : ");
    scanf("%d", &n);
    printf("\nEnter the array elements :");
    for (i = 1; i <= n; i++)
    {
        scanf("%d", &A[i]);
        if (A[i] > k) {
            k = A[i];
        }
    }
    counting_sort(A, k, n);
    printf("\n");
    return 0;
}
```

The input array is the same as that used in the example:

Output



```
Enter the size of the array :10
Enter the array elements: 1 3 2 8 5 1 5 1 2 7
1 1 1 2 2 3 5 5 7 8


...Program finished with exit code 0
Press ENTER to exit console.
```

Figure 3.14: Output of Program

Time Complexity Analysis

For scanning the input array elements, the loop iterates n times, thus taking O(n) running time. The sorted array B[] also gets computed in n iterations, thus requiring O(n) running time. The count array also uses k iterations, thus has a running time of O(k). Thus the total running time for counting sort algorithm is O(n+k).

Key Points:

- The above implementation of Counting Sort can also be extended to sort negative input numbers
- Since counting sort is suitable for sorting numbers that belong to a well-defined, finite and small range, it can be used asa subprogram in other sorting algorithms like radix sort which can be used for sorting numbers having a large range
- Counting Sort algorithm is efficient if the range of input data (k) is not much greater than the number of elements in the input array (n). It will not work if we have 5 elements to sort in the range of 0 to 10,000
- It is an integer-based sorting algorithm unlike others which are usually comparison-based. A comparison-based sorting algorithm sorts numbers only by comparing pairs of numbers. Few examples of comparison based sorting algorithms are quick sort, merge sort, bubble sort, selection sort, heap sort, insertion sort, whereas algorithms like radix sort, bucket sort and comparison sort fall into the category of non-comparison based sorting algorithms.

Advantages of Counting Sort:

- It is quite fast
- It is a stable algorithm

Note: For a sorting algorithm to be stable, the order of elements with equal keys (values) in the sorted array should be the same as that of the input array.

Disadvantages of Counting Sort:

- It is not suitable for sorting large data sets
- It is not suitable for sorting string values

Check your progress

Q1.Define the lower bound for sorting with suitable example.
Q2.Define time complexity for counting sort algorithm.

3.3Radix Sort

Radix sort is a sorting technique that sorts the elements by first grouping the individual digits of the same place value. Then, sort the elements according to their increasing/decreasing order.

Suppose, we have an array of 8 elements. First, we will sort elements based on the value of the unit place. Then, we will sort elements based on the value of the tenth place. This process goes on until the last significant place.

Let the initial array be [121, 432, 564, 23, 1, 45, 788]. It is sorted according to radix sort as shown in the figure below.

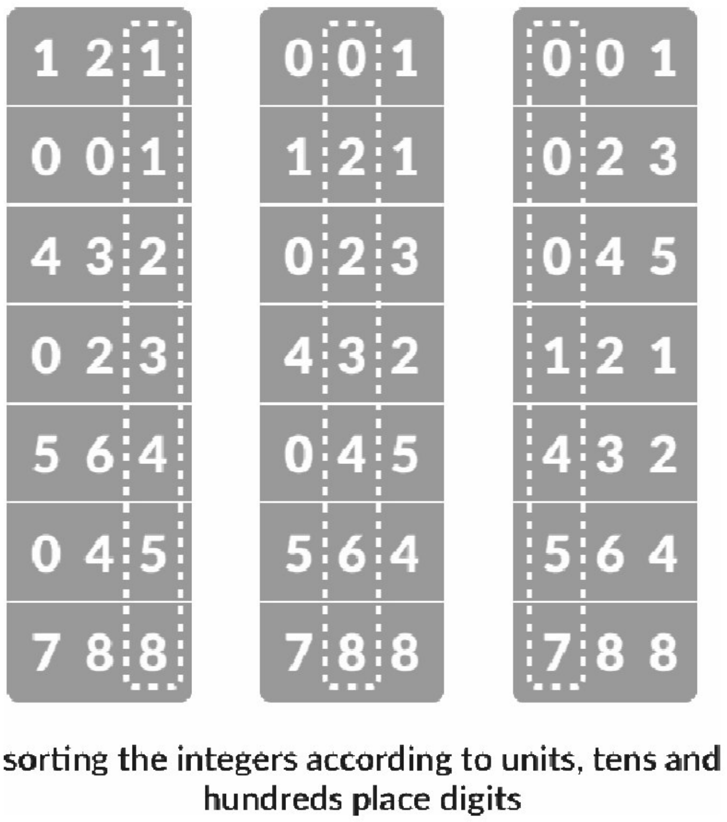


Figure 3.15: Working of Radix Sort

How Radix Sort Works?

Find the largest element in the array, i.e. max. Let X be the number of digits in max. X is calculated because we have to go through all the significant places of all elements.

In this array [121, 432, 564, 23, 1, 45, 788], we have the largest number 788. It has 3 digits. Therefore, the loop should go up to hundreds place (3 times).

Now, go through each significant place one by one.

Use any stable sorting technique to sort the digits at each significant place. We have used counting sort for this.

Sort the elements based on the unit place digits (X=0).

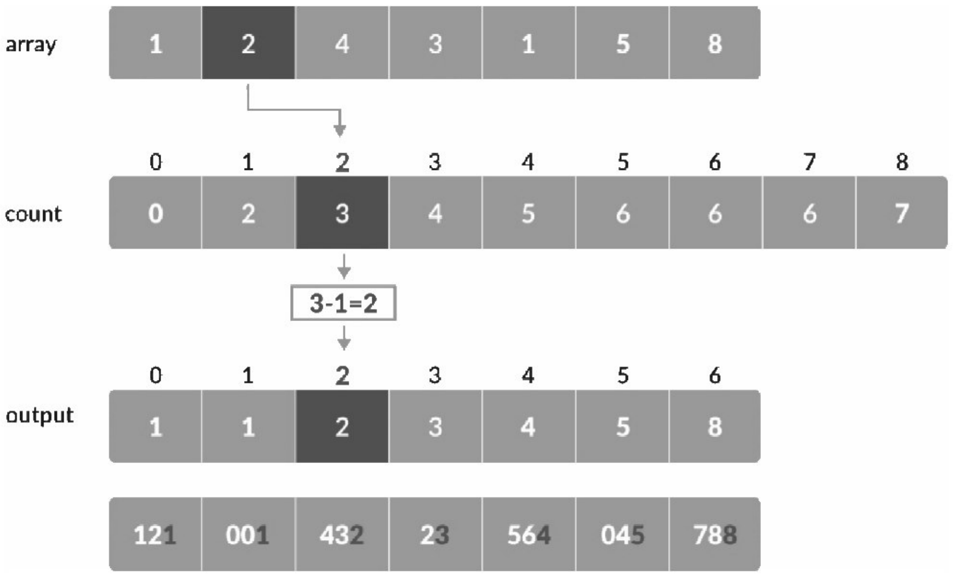


Figure 3.16: Using counting sort to sort elements based on unit place

Now, sort the elements based on digits at tens place.

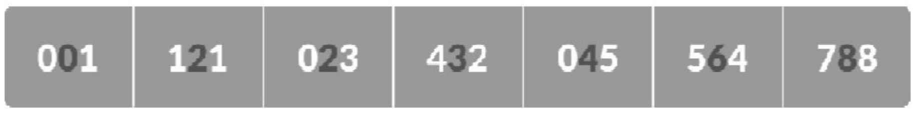


Figure 3.17: Sort elements based on tens place

Finally, sort the elements based on the digits at hundreds place.

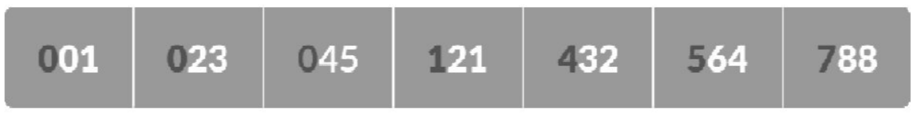


Figure 3.18: Sort elements based on hundreds plac

Radix Sort Algorithm

```
radixSort(array)
  d <- maximum number of digits in the largest element
  create d buckets of size 0-9
  for i <- 0 to d
    sort the elements according to ith place digits using countingSort

countingSort(array, d)
  max <- find largest element among dth place elements
  initialize count array with all zeros
  for j <- 0 to size
    find the total count of each unique digit in dth place of elements and
    store the count at jth index in count array
  for i <- 1 to max
    find the cumulative sum and store it in count array itself
  for j <- size down to 1
    restore the elements to array
    decrease count of each element restored by 1
```

Program for Radix Sort in C Language

```c
// Radix Sort in C Programming

#include <stdio.h>

// Function to get the largest element from an array
int getMax(int array[], int n) {
  int max = array[0];
  for (int i = 1; i < n; i++)
    if (array[i] > max)
      max = array[i];
  return max;
}

// Using counting sort to sort the elements in the basis of significant places
void countingSort(int array[], int size, int place) {
  int output[size + 1];
  int max = (array[0] / place) % 10;

  for (int i = 1; i < size; i++) {
    if (((array[i] / place) % 10) > max)
      max = array[i];
  }
  int count[max + 1];

  for (int i = 0; i < max; ++i)
    count[i] = 0;

  // Calculate count of elements
  for (int i = 0; i < size; i++)
    count[(array[i] / place) % 10]++;

  // Calculate cummulative count
  for (int i = 1; i < 10; i++)
    count[i] += count[i - 1];

  // Place the elements in sorted order
  for (int i = size - 1; i >= 0; i--) {
    output[count[(array[i] / place) % 10] - 1] = array[i];
    count[(array[i] / place) % 10]--;
  }
  for (int i = 0; i < size; i++)
    array[i] = output[i];
}

// Main function to implement radix sort
void radixsort(int array[], int size) {
  // Get maximum element
  int max = getMax(array, size);

  // Apply counting sort to sort elements based on place value.
  for (int place = 1; max / place > 0; place *= 10)
    countingSort(array, size, place);
}

// Print an array
void printArray(int array[], int size) {
  for (int i = 0; i < size; ++i) {
    printf("%d  ", array[i]);
  }
  printf("\n");
}

// Driver code
int main() {
  int array[] = {121, 432, 564, 23, 1, 45, 788};
  int n = sizeof(array) / sizeof(array[0]);
  radixsort(array, n);
  printArray(array, n);
}
```

Complexity

Since radix sort is a non-comparative algorithm, it has advantages over comparative sorting algorithms.

For the radix sort that uses counting sort as an intermediate stable sort, the time complexity is O(d(n+k)).

Here, d is the number cycle and O(n+k) is the time complexity of counting sort.

Thus, radix sort has linear time complexity which is better than O(nlog n) of comparative sorting algorithms.

If we take very large digit numbers or the number of other bases like 32-bit and 64-bit numbers, then it can perform in linear time however the intermediate sort takes large space.

This makes radix sort space inefficient. This is the reason why this sort is not used in software libraries.

Radix Sort Applications

Radix sort is implemented in

- DC3 algorithm (Kärkkäinen-Sanders-Burkhardt) while making a suffix array.
- places where there are numbers in large ranges.

3.4Bucket Sort

Bucket Sort is a sorting technique that sorts the elements by first dividing the elements into several groups called buckets. The elements inside each bucket are sorted using any of the suitable sorting algorithms or recursively calling the same algorithm.

Several buckets are created. Each bucket is filled with a specific range of elements. The elements inside the bucket are sorted using any other algorithm. Finally, the elements of the bucket are gathered to get the sorted array.

The process of bucket sort can be understood as a scatter-gather approach. The elements are first scattered into buckets then the elements of buckets are sorted. Finally, the elements are gathered in order.

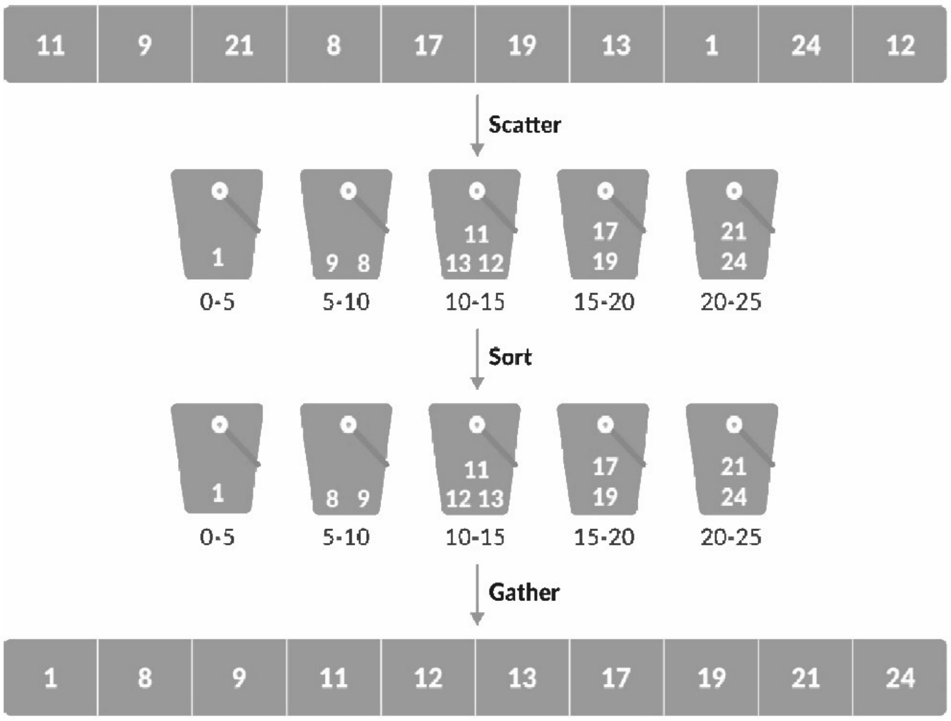


Figure 3.19: Working of Bucket Sort

How Bucket Sort Works?

1. **Suppose, the input array is:**

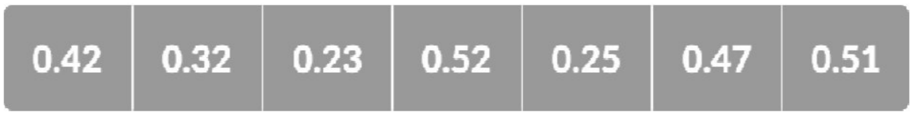


Figure 3.20: Input array

Create an array of size 10. Each slot of this array is used as a bucket for storing elements.

0 0 0 0 0 0 0 0 0 0

Figure 3.21: Array in which each position is a bucket

2. **Insert elements into the buckets from the array. The elements are inserted according to the range of the bucket.**

In our example code, we have buckets each of ranges from 0 to 1, 1 to 2, 2 to 3, ... (n-1) to n.

Suppose, an input element is .23 is taken. It is multiplied by size = 10 (ie. .23*10=2.3). Then, it is converted into an integer (ie. 2.3≈2). Finally, .23 is inserted into bucket-2.

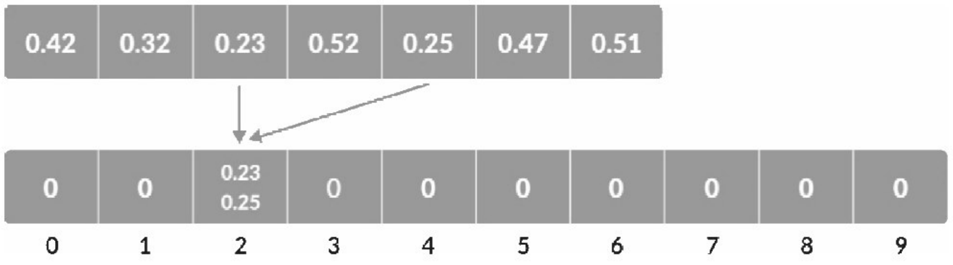


Figure 3.22: Insert elements into the buckets from the array

Similarly, .25 is also inserted into the same bucket. Every time, the floor value of the floating point number is taken.

If we take integer numbers as input, we have to divide it by the interval (10 here) to get the floor value.

Similarly, other elements are inserted into their respective buckets.

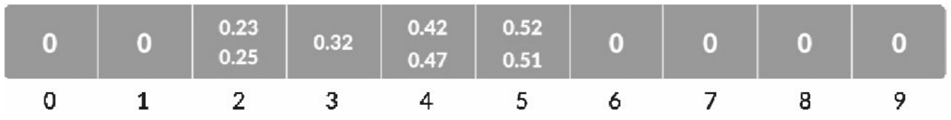


Figure 3.23: Insert all the elements into the buckets from the array

3. **The elements of each bucket are sorted using any of the stable sorting algorithms. Here, we have used quicksort (inbuilt function).**

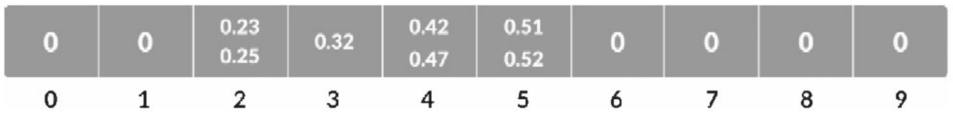


Figure 3.24: Sort the elements in each bucket

4. The elements from each bucket are gathered.

It is done by iterating through the bucket and inserting an individual element into the original array in each cycle. The element from the bucket is erased once it is copied into the original array.

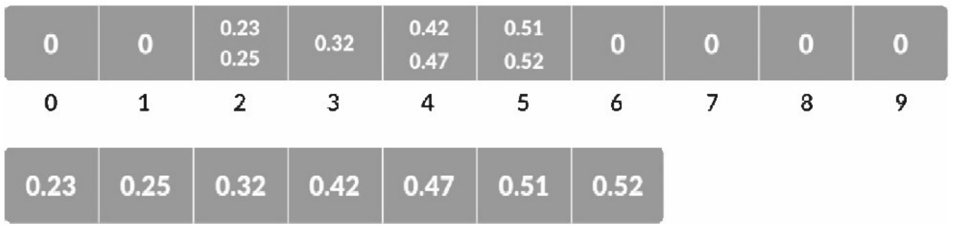


Figure 3.25: Gather elements from each bucket

Bucket Sort Algorithm

```
bucketSort()
  create N buckets each of which can hold a range of values
  for all the buckets
    initialize each bucket with 0 values
  for all the buckets
    put elements into buckets matching the range
  for all the buckets
    sort elements in each bucket
  gather elements from each bucket
end bucketSort
```

Program in C

```
// Bucket sort in C
#include <stdio.h>
#include <stdlib.h>

#define NARRAY 7   // Array size
#define NBUCKET 6  // Number of buckets
```

```
#define INTERVAL 10  // Each bucket capacity

struct Node {
  int data;
  struct Node *next;
};

void BucketSort(int arr[]);
struct Node *InsertionSort(struct Node *list);
void print(int arr[]);
void printBuckets(struct Node *list);
int getBucketIndex(int value);

// Sorting function
void BucketSort(int arr[]) {
  int i, j;
  struct Node **buckets;

  // Create buckets and allocate memory size
  buckets = (struct Node **)malloc(sizeof(struct Node *) * NBUCKET);

  // Initialize empty buckets
  for (i = 0; i < NBUCKET; ++i) {
    buckets[i] = NULL;
  }

  // Fill the buckets with respective elements
  for (i = 0; i < NARRAY; ++i) {
    struct Node *current;
    int pos = getBucketIndex(arr[i]);
    current = (struct Node *)malloc(sizeof(struct Node));
    current->data = arr[i];
```

```
   current->next = buckets[pos];
   buckets[pos] = current;
 }

 // Print the buckets along with their elements
 for (i = 0; i < NBUCKET; i++) {
   printf("Bucket[%d]: ", i);
   printBuckets(buckets[i]);
   printf("\n");
 }

 // Sort the elements of each bucket
 for (i = 0; i < NBUCKET; ++i) {
   buckets[i] = InsertionSort(buckets[i]);
 }

 printf("-------------\n");
 printf("Bucktets after sorting\n");
 for (i = 0; i < NBUCKET; i++) {
   printf("Bucket[%d]: ", i);
   printBuckets(buckets[i]);
   printf("\n");
 }

 // Put sorted elements on arr
 for (j = 0, i = 0; i < NBUCKET; ++i) {
   struct Node *node;
   node = buckets[i];
   while (node) {
     arr[j++] = node->data;
     node = node->next;
   }
```

```
  }

  return;
}

// Function to sort the elements of each bucket
struct Node *InsertionSort(struct Node *list) {
  struct Node *k, *nodeList;
  if (list == 0 || list->next == 0) {
    return list;
  }

  nodeList = list;
  k = list->next;
  nodeList->next = 0;
  while (k != 0) {
    struct Node *ptr;
    if (nodeList->data > k->data) {
      struct Node *tmp;
      tmp = k;
      k = k->next;
      tmp->next = nodeList;
      nodeList = tmp;
      continue;
    }

    for (ptr = nodeList; ptr->next != 0; ptr = ptr->next) {
      if (ptr->next->data > k->data)
        break;
    }
```

[80]

```
    if (ptr->next != 0) {
      struct Node *tmp;
      tmp = k;
      k = k->next;
      tmp->next = ptr->next;
      ptr->next = tmp;
      continue;
    } else {
      ptr->next = k;
      k = k->next;
      ptr->next->next = 0;
      continue;
    }
  }
  return nodeList;
}

int getBucketIndex(int value) {
  return value / INTERVAL;
}

void print(int ar[]) {
  int i;
  for (i = 0; i < NARRAY; ++i) {
    printf("%d ", ar[i]);
  }
  printf("\n");
}

// Print buckets
void printBuckets(struct Node *list) {
  struct Node *cur = list;
  while (cur) {
    printf("%d ", cur->data);
    cur = cur->next;
  }
}

// Driver code
int main(void) {
  int array[NARRAY] = {42, 32, 33, 52, 37, 47, 51};

  printf("Initial array: ");
  print(array);
  printf("------------\n");

  BucketSort(array);
  printf("------------\n");
  printf("Sorted array: ");
  print(array);
  return 0;
}
```

Complexity

- **Worst Case Complexity: O(n2)**

  When there are elements of close range in the array, they are likely to be placed in the same bucket. This may result in some buckets having more number of elements than others.

  It makes the complexity depend on the sorting algorithm used to sort the elements of the bucket.

  The complexity becomes even worse when the elements are in reverse order. If insertion sort is used to sort elements of the bucket, then the time complexity becomes O(n2).

- **Best Case Complexity: O(n+k)**

  It occurs when the elements are uniformly distributed in the buckets with a nearly equal number of elements in each bucket.

The complexity becomes even better if the elements inside the buckets are already sorted.

If insertion sort is used to sort elements of a bucket, then the overall complexity in the best case will be linear ie. O(n+k). O(n) is the complexity for making the buckets and O(k) is the complexity for sorting the elements of the bucket using algorithms having linear time complexity at the best case.

- **Average Case Complexity: O(n)**

  It occurs when the elements are distributed randomly in the array. Even if the elements are not distributed uniformly, bucket sort runs in linear time. It holds true until the sum of the squares of the bucket sizes is linear in the total number of elements.

Bucket Sort Applications

Bucket sort is used when:

- Input is uniformly distributed over a range.
- there are floating point values

3.5 Medians and Order Statistics

The *ith* order statistic of a set of n elements is the ith smallest element. For example, the minimum of a set of elements is the first order statistic (i = 1), and the maximum is the nth order statistic (i = n). A median, informally, is the "halfway point" of the set. When n is odd, the median is unique, occurring at i = (n + 1)/2. When n is even, there are two medians, occurring at i = n/2 and i = n/2 + 1. Thus, regardless of the parity of n, medians occur at $i = \lfloor (n + 1)/2 \rfloor$ and $i = \lceil (n + 1)/2 \rceil$

This chapter addresses the problem of selecting the *ith* order statistic from a set of n distinct numbers. We assume for convenience that the set contains distinct numbers, although virtually everything that we do extends to the situation in which a set contains repeated values. The selection problem can be specified formally as follows:

Input: A set A of n (distinct) numbers and a number i, with 1 ≤ i ≤ n.

Output: The element x A that is larger than exactly i -1 other elements of A.

The selection problem can be solved in O(n log n) time, since we can sort the numbers using heapsort or merge sort and then simply index the *ith* element in the output array. There are faster algorithms, however.

3.6 Minimum and maximum

How many comparisons are necessary to determine the minimum of a set of n elements? We can easily obtain an upper bound of n - 1 comparisons: examine each element of the set in turn and keep track of the smallest element seen so far. In the following procedure, we assume that the set resides in array A, where length [A] = n.

```
1. MINIMUM (A)
2. min  A[1]
3. for i  2 to length[A]
4. do if min > A[i]
       a. then min  A[i]
5. return min
```

Finding the maximum can, of course, be accomplished with n - 1 comparisons as well.

Is this the best we can do? Yes, since we can obtain a lower bound of n - 1 comparisons for the problem of determining the minimum. Think of any algorithm that determines the minimum as a tournament among the elements. Each comparison is a match in the tournament in which the smaller of the two elements wins. The key observation is that every element except the winner must lose at least one match. Hence, n - 1 comparisons are necessary to determine the minimum, and the algorithm MINIMUM is optimal with respect to the number of comparisons performed.

An interesting fine point of the analysis is the determination of the expected number of times that line 4 is executed.

Simultaneous minimum and maximum

In some applications, we must find both the minimum and the maximum of a set of n elements. For example, a graphics program may need to scale a set of (x, y) data to fit onto a rectangular display screen or other graphical output device. To do so, the program must first determine the minimum and maximum of each coordinate.

It is not too difficult to devise an algorithm that can find both the minimum and the maximum of n elements using the asymptotically optimal Ω (n) number of comparisons. Simply find the minimum and maximum independently, using n - 1 comparisons for each, for a total of 2n - 2 comparisons.

In fact, only $3\lceil n/2\rceil$comparisons are necessary to find both the minimum and the maximum. To do this, we maintain the minimum and maximum elements seen thus far. Rather than processing each element of the input by comparing it against the current minimum and maximum, however, at a cost of two comparisons per element, we process elements in pairs. We compare pairs of elements from the input first with each other, and then compare the smaller to the current minimum and the larger to the current maximum, at a cost of three comparisons for every two elements.

3.7 Summary

In this unit you have learnt about the concept of Lower bounds for sorting. You have also learnt about different sorting algorithms like counting sort, radix sort andbucket sort. In the last we have described about medians and order statistics and minimum and maximum.

- A lower bound for a problem is the worst-case running time of the best possible algorithm for that problem. To prove a lower bound of Ω(n lg n) for sorting, we would have to prove that no algorithm, however smart, could possibly be faster, in the worst-case, then n lg n.
- Counting Sort is a linear time sorting algorithm which works faster by not making a comparison. It assumes that the number to be sorted is in range 1 to k where k is small. Basic idea is to determine the "rank" of each number in the final sorted array.
- Radix Sort is a Sorting algorithm that is useful when there is a constant'd' such that all keys are d digit numbers. To execute Radix Sort, for p =1 towards 'd' sort the numbers with respect to the Pth digits from the right using any linear time stable sort.
- Bucket Sort runs in linear time on average. Like Counting Sort, bucket Sort is fast because it considers something about the input. Bucket Sort considers that the input is generated by a random process that distributes elements uniformly over the interval μ=[0,1].

3.8 Review Questions

Q1. What is the lower bound for number of comparisons while sorting? Explain your answer.

Q2. What is the auxiliary space requirement of counting sort? Elaborate your answer.

Q3. Is Radix Sort preferable to Comparison based sorting algorithms like Quick-Sort? Explain in detail.

Q4. Write a program to sort an array with negative numbers using bucket sort.

Q5. How Radix sort operates on seven 3-digits number with suitable example.

Uttar Pradesh Rajarshi Tondon open University

Master of Computer Science

MCS-111 Design and Analysis of Algorithm

Block 2

Algorithm Design Strategies - II

© UPRTOU , Prayagraj
ISBN :

BLOCK 2 INTRODUCTION

We believe effective programmers must combine theory with practice, so they can adapt to ever-changing computing environments. This block does not cover the breadth of topics found in some professional reference books, but it has a number of features that make it useful in the classroom:

- A step-by-step learning approach in which new ideas and concepts build on existing ones.
- Check-up exercises at the end of each section.
- Review questions and programming exercises at the end of each chapter.

Block 2 contains two units i.e. unit 4 and unit 5 intended with Greedy method and Dynamic programming.

Unit 4 introduces the greedy methods; it incorporates introduction to greedy method, general method, Knapsack problem, Job sequencing with deadlines, optimal two-way merge patterns, Huffman codes, Prims and Kruskal's algorithm to find the minimum cost spanning tree, The Bellman-Ford and Dijkstra's algorithm to find the single source shortest path.

Unit 5 covers thesignificance of dynamic programming and it covers general method, applications, capital budgeting problem, multistage graphs, matrix chain multiplication, 0/1 knapsack problem, all pairs shortest path problem, and travelling sales person problem.

Unit 4: Greedy Method

Structure

4.0 INTRODUCTION

This unit basically deals with the Greedy method. Greedy algorithms are supposed be the simplest algorithms when demonstrating the subjects. These are also considered as the most natural approach for any real time problem. Hence they are also called naïve methods. It explains about the significance of greedy method, its applications in various applications like to find out the minimum cost spanning tree, finding the shortest paths etc. As we know that all these applications are very important and widely used, hence this unit is important for the students so that they could understand better the concepts of design and analysis of algoorithms; moreover, these algorithms could be easily applied in various realtime applications. Greedy approach and dynamic programming are simple and widely used in design and analysis of algorithms. Accordingly, it maximizes the usefulness and utility of the entire process. Furthermore, it can alsobe used in computer network protocols viz in packet switching networks etc. But sometimes this algorithm fails i.e it cannot be applied everywhere. Of course, the immediate application of greedy algorithms does not always produce the optimal result.

4.1 OBJECTIVES

At the end of this unit you will come to know about the following:

- o Greedy method
- o Knapsack problem
- o Job sequencing with deadlines
- o optimal two-way merge patterns
- o Huffman codes
- o Minimum cost spanning trees
- o Prims and Kruskal's algorithm
- o The Bellman-Ford algorithm
- o Dijkstra's algorithm

4.2 INTRODUCTION TO GREEDY METHOD

This is the algorithm that gives the solution in parts i.e. piece by piece. And the main thing is that the next piece or part is always better that the previous one. In *Greedy Algorithm* a set of resources are recursively divided based on the maximum, immediate availability of that resource at any given stage of execution.

Usually, to solve a problem based on the greedy approachthere are two stages

1. Scanning the list of items
2. Optimization

To understand the greedy algorithm in right manner, one needs to know about the recursion concept properly, as it uses the concept of recursion and context switching. This helps to understand how to trace the code of the problem. Moreover, one can define the greedy paradigm in terms of own necessary and sufficient statements.

Two conditions define the greedy paradigm basically

1. Each stepwise solution must structure a problem towards its best-accepted solution.
2. It is sufficient if the structuring of the problem can halt in a finite number of greedy steps.

4.2.1 History of Greedy Algorithms

Now let us know about the history of greedy method in subsequent topicsgiven below. Following are some important points:

- Greedy algorithms were conceptualized for many graph walk algorithms in the 1950s.
- Esdger Djikstra conceptualized the algorithm to generate minimal spanning trees. He aimed to shorten the span of routes within the Dutch capital, Amsterdam.
- In the same decade, Prim and Kruskal achieved optimization strategies that were based on minimizing path costs along weighed routes.
- In the 1970s, American researchers, Cormen, Rivest, and Stein proposed a recursive substructuring of greedy solutions in their classical introduction to algorithms book.
- The Greedy search paradigm was registered as a different type of optimization strategy in the NIST records in 2005.

- Till date, protocols that run the web, such as the open-shortest-path-first (OSPF) and many other network packet switching protocols use the greedy strategy to minimize time spent on a network.

4.2.2General Method

A greedy algorithm is any algorithm that follows the problem-solving heuristic of making the locally optimal choice at each stage. In several problems, a greedy strategy does'nt usually produces an optimal solution, but nonetheless, a greedy heuristic may yield locally optimal solutions that approximate a globally optimal solution in a reasonable time. This approach never reconsiders the choices taken previously. This approach is mainly used to solve optimization problems.

Greedy method is easy to implement and quite efficient in most of the cases. Hence, we can say that *Greedy algorithm* is an algorithmic paradigm based on heuristics that follow local optimal choice at each step with the hope of finding global optimal solution. Some issues do'nt have any solution but greedy method can be used to find the near optimal solution for the unsolved issues. Moreover, it is accepted in various situations like packet switching network protocols, finding the shortest path among various nodes etc.

A greedy algorithm works well for the problems specically having following two properties:

1. **Greedy Choice Property-** In short, the optimal solution for a particular problem can be found by creating "greedy" choices.
2. **Optimal Substructure-** Basically optimal solutions contain optimal sub solutions; or in other words we can say answers to the subproblems are optimal solutions.

There are three steps for achieving a Greedy algorithm:

1. **Feasibility-** First of all the feasibility of the problem is checked; and hence at least one solution must be obtained satisfying all possible constraints of the problem.
2. **Local Optimal Choice-** The choice must be optimum and selected from the currently available choices.
3. **Unaltrable-**After selection of the choice, it should not be altered at all.

4.2.3 Chracteristics of Greedy Approach

The important characteristics of a Greedy method are:

- There is an ordered list of resources, with costs or value attributions. These quantify constraints on a system.

- The maximum quantity of resources is taken into consideration in the time a constraint applies.
- For example, in an activity scheduling problem, the resource costs are in hours, and the activities need to be performed in serial order.

4.2.4 Why Use the Greedy Approach?

Here are the reasons for using the greedy approach:

- The greedy approach has a few tradeoffs, which may make it suitable for optimization.
- One prominent reason is to achieve the most feasible solution immediately. In the activity selection problem (which has been explained below), if more activities are done before finishing the current activity, these activities can be performed within the same time.
- Another reason is to divide a problem recursively based on a condition, with no need to combine all the solutions.
- In the activity selection problem, the "recursive division" step is achieved by scanning a list of items only once and considering certain activities.

4.2.5 Architecture of Greedy Approach

Step-1: Scan the list of activity costs, starting with index 0 as the considered Index.

Step-2: When more activities can be finished by the time, the considered activity finishes, start searching for one or more remaining activities.

Step-3: If there are no more remaining activities, the current remaining activity becomes the next considered activity. Repeat step-1 and step-2, with the new considered activity. If there are no remaining activities left, then goto step-4.

Step-4: Return the union of considered indices. These are the activity indices that would be used to maximize throughput.

4.2.6 Components of Greedy Approach

Greedy algorithms have the following five components

1. **A candidate set** – A solution is created from this set.
2. **A selection function** – Used to choose the best candidate to be added to the solution.
3. **A feasibility function** – Used to determine whether a candidate can be used to contribute to the solution.
4. **An objective function** – Used to assign a value to a solution or a partial solution.

5. **A solution function** – Used to indicate whether a complete solution has been reached.

4.2.7 Application Areas

Greedy approach can be used to solve many problems such as

- o To find the optimal solution using e.g. Job Sequencing, Huffman Coding, Fractional Knapsack, Activity Selection.
- o To find the shortest path between two vertices using Dijkstra's algorithm.
- o To find close to the optimal solution for NP-hard problems like Travelling Salesman problem.
- o To find the minimal spanning tree in a graph using Prim's and Kruskal's algorithm.

***Note:**In several problems, Greedy algorithm fails to find an optimal solution, rather it may produce a worst solution. Problems like Travelling Salesman and Knapsack cannot be solved using this approach.*

4.3 KNAPSACK PROBLEM

The *Knapsack Problem* is a famous *Dynamic Programming* problem that falls in the *optimization* category. It arose in 1897, hence it is near about more than 100 years old. It is based on the resource allocation condition. It is helpful in combinatorics. In this problem, the user basically decides to allocate the most valuable item or task or resource to the fixed-size block, or in other words when he has only fixed budget/time. Moreover, it basically refers to the most common real time problem of packing something which is very costly or important item in a container without mixing it with other item/items. This method is useful in various real-time applications where wastage of raw material should be very less or without cutting, the raw material could be properly utilized.

As far as the basic concept is concerened, Knapsack means a bag. A bag of given capacity and one wants to pack ***n*** number of items in the luggage. Let us consider a set of items having weight and value say ***w*** and ***v***, now determine a subset of items to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible. Moreover, this knapsack problem is in combinatorial optimization problem. It appears as a subproblem in many more complex mathematical models of real-world problems. One general approach to difficult problems is to identify the most restrictive constraint, ignore the others, solve a knapsack problem, and somehow adjust the solution to satisfy the ignored constraints.

Let us imagine about a small story of a thief who wants to steal some valuable artifacts like jwellery, idols of gods, manuscrips etc. from a famous museum. And he is having only one bag to keep all these items. Now because the thief is new for the place and not proactively aware about the valuable things of the museum. He was also not knowing that how many bags would be required to carry the items. Now he has to select only valuable items out of the available artifacts but without overloading the bag. He has to run away but with care that the bag should not be too heavy to carry and any of the items should also not be broken. The goal of the thief is to get away with the most valuable objects without overloading the bag until it breaks or becomes too heavy to carry. Now the question arises, how does he choose among the objects to maximize his loot? He might list all the artifacts and their weights to work out the answer by hand. But the more objects there are, the more taxing this calculation becomes for a person—or a computer. Ultimately, it can be said the solution is Kanapsack.

4.3.2 Application of Knapsack problem

In many cases of resource allocation along with some constraint, the problem can be derived in a similar way of *Knapsack problem*. Some related examples are

- Construction and scoring of tests in which the test-takers have a choice as to which questions they answer
- Finding the least wasteful way to cut raw materials
- Selection of investments and portfolios
- Cutting stock problems
- Selection of assets for asset-backed securitization

4.3.3Problem Description

Let us consider a problem in which a thief is robbing a store and can carry a maximal weight of ***W*** into his knapsack. There are ***n*** items available in the store and weight of i^{th} item is w_i and its profit is p_i. What items should the thief take?

In this context, the items should be selected in such a way that the thief will carry those items for which he will gain maximum profit. Hence, the objective of the thief is to maximize the profit.

Based on the nature of the items, Knapsack problems are categorized as

- Fractional Knapsack
- Knapsack

4.3.4 Fractional Knapsack

In this case, items can be broken into smaller pieces, hence the thief can select fractions of items.

Now, as per the problem statement,

- There are ***n*** items in the store
- Weight of **i**th item $w_i>0$
- Profit for **i**th item $p_i>0$ and
- Capacity of the Knapsack is **W**

In this version of Knapsack problem, items can be broken into smaller pieces. So, the thief may take only a fraction x_i of **i**th item.

$$0 \leqslant x_i \leqslant 1$$

The **i**th item contributes the weight $x_i w_i$ to the total weight in the knapsack and profit $x_i p_i$ to the total profit.

Hence, the objective of this algorithm is to

$$maximize \sum_{n=1}^{n}(x_i.pi)$$

subject to constraint,

$$\sum_{n=1}^{n}(x_i.wi) \leqslant W$$

It is clear that an optimal solution must fill the knapsack exactly, otherwise we could add a fraction of one of the remaining items and increase the overall profit.

Thus, an optimal solution can be obtained by

$$\sum_{n=1}^{n}(x_i.wi) = W$$

In this context, first those items need to be sorted according to the value of p_i/w_i, so that $(p_i+1)/(w_i+1) \leq (p_i/w_i)$. Here, ***x*** is an array to store the fraction of items.

Algorithm: Greedy-Fractional-Knapsack (w[1..n], p[1..n], W)

```
for i = 1 to n
  do x[i] = 0
weight = 0
for i = 1 to n
  if weight + w[i] ≤ W then
    x[i] = 1
    weight = weight + w[i]
  else
    x[i] = (W - weight) / w[i]
    weight = W
    break
return x
```

Analysis: If the provided items are already sorted into a decreasing order of p_i/w_i, then the whileloop takes a time in ***O(n)***; therefore, the total time including the sort is in *O(nlogn)*.

4.4 JOB SEQUENCING WITH DEADLINES

In job sequencing problem the main goal is to find the sequence of jobs completed within deadlines which give highest profit. The sequencing of jobs on a single processor with defined deadline constraints is called as Job Sequencing with Deadlines.The problem can be described as:

- In the beginning, a set of jobs are given.
- Each job has a defined deadline and some profit associated with it.
- The profit of a particular job is given only when that job is completed within its deadline.
- Only one processor is available for the processing of all jobs.
- Processor takes one unit of time to complete a job.

4.4.1 Solution

A feasible solution would be a subset of jobs where each job of the subset gets completed within its deadline. Value of the feasible solution would be the sum of profit of all the jobs contained in the subset. An optimal solution of the problem would be a feasible solution which gives the maximum profit.

Let us consider, a set of ***n*** given jobs which are associated with deadlines and earned profit, if a job is completed by its deadline. These jobs need to be ordered in such a way that there should be maximum profit. It may happen that all of the given jobs may not be completed within their deadlines. Now, suppose the deadline of ith job J_i is d_i and the profit received from this job is p_i.

Hence, the optimal solution of this algorithm would be a feasible solution with maximum profit.

Thus, $D(i)>0$ for $1 \leqslant i \leqslant n$

Initially, these jobs are ordered according to profit, i.e. $p_1 \geqslant p_2 \geqslant p_3 \geqslant \ldots \geqslant p_n$

4.4.2Algorithm

Greedy Algorithm is adopted to determine how the next job is selected for an optimal solution.
The greedy algorithm described below always gives an optimal solution to the job sequencing problem. The steps for it are given below

Step-1: Sort all the given jobs in decreasing order of their profit.

Step-2: Check the value of maximum deadline. Draw a Gantt chart where maximum time on Gantt chart is the value of maximum deadline.

Step-3: Pick up the jobs one by one. Put the job on Gantt chart as far as possible from 0 ensuring that the job gets completed before its deadline.

Pseudocode:

```
Job-Sequencing-With-Deadline (D, J, n, k)
D(0) := J(0) := 0
k := 1
J(1) := 1   // means first job is selected
for i = 2 … n do
  r := k
  while D(J(r)) > D(i) and D(J(r)) ≠ r do
    r := r – 1
  if D(J(r)) ≤ D(i) and D(i) > r then
    for l = k … r + 1 by -1 do
      J(l + 1) := J(l)
      J(r + 1) := i
      k := k + 1
```

Analysis: In this algorithm, two loops are used, one is within another. Hence, the complexity of this algorithm is $O(n^2)$

4.4.3 Example

Let us consider a set of given jobs as shown in the table given below. Now find a sequence of jobs, which will be completed within their deadlines and will give maximum profit. Each job is associated with a deadline and profit.

Jobs	J_1	J_2	J_3	J_4	J_5
Deadline	2	1	3	2	1
Profit	60	100	20	40	20

Solution: Firstly, the jobs are arranged in their profit order as given below in table

Jobs	J_2	J_1	J_4	J_3	J_5
Deadline	1	2	2	3	1
Profit	100	60	40	20	20

Moreover,

- Firstly,job **J_2** is selected as it has maximum profit and completed within the deadline.
- Secondly, job **J_1** is selected as it has maximum profit and completed within the deadline in comparison to job **J_4**.
- Thirdly, the job **J_4** cannot be selected as its time is over; rather job **J_3** is selected as it is completed within its deadline.
- Fourth, job **J_5** is discarded as its deadline is also over.

Hence the final solution is the sequence **J_2,J_1,J_3**; because these are excuted within deadlines and having maximum profit.

Finally, the total profit = 100 + 60 +20 = 180

CHECK YOUR PROGRESS

- Define the importance of *Greedy method.*
- What do you understand by a *general geedy method?*
- Write the names of some real world apllications of *Greedy method.*

4.5 OPTIMAL TWO-WAY PATTERNS

Optimal merge pattern is a pattern that relates to the merging of two or more sorted files in a single sorted file.In this way we need to find an optimal solution, where the resultant file would be generated in minimum time. If the number of sorted files are given, there are many ways to merge them into a single sorted file. This merge can be performed pair wise. Hence, this type of merging is called as 2-way merge patterns. As, different pairings require different amounts of time, in this strategy an optimal way of merging many files together is determined. At each step, two shortest sequences are merged to obtain one step.

Now let us consider two sorted files containing ***n*** and ***m*** records respectively then they could be merged to obtain one sorted file in time O(n+m). There are many ways in which pairwise merge can be done to obrain a single sorted file. Different pairings require a different amount of computing time. The main thing is to pairwise merge***n***number of sorted files so that the number of comparisons could be reduced in time.

The formula of external merging cost is:

$$\sum_{i=1}^{n} f(i)d(i)$$

Where, f (i) represents the number of records in each file and d(i) represents the depth.

Algorithm: TREE(n)

```
for i:= 1 to n – 1 do
  declare new node
node.leftchild:= least (list)
node.rightchild:= least (list)
  node.weight) := ((node.leftchild).weight) + ((node.rightchild).weight)
  insert (list, node);
return least (list);
```

Note:*At the end of this algorithm, the weight of the root node represents the optimal cost.*

Example:*Let us consider the given files, f_1, f_2, f_3, f_4 and f_5 with 20, 30, 10, 5 and 30 number of elements respectively.*

Now, if the merge operations are performed according to the provided sequence, then

M_1 = merge f_1 and f_2 => 20 + 30 = 50
M_2 = merge M_1 and f_3 => 50 + 10 = 60
M_3 = merge M_2 and f_4 => 60 + 5 = 65
M_4 = merge M_3 and f_5 => 65 + 30 = 95

Hence, the total number of operations are

50 + 60 + 65 + 95 = 270 --------------- (A)

Now, the question arises; Is there any better solution?

Sorting the numbers according to their size in an ascending order, we get the following sequence −

f_4, f_3, f_1, f_2, f_5

Hence, merge operations can be performed on this sequence

M_1 = merge f_4 and f_3 => 5 + 10 = 15
M_2 = merge M_1 and f_1 => 15 + 20 = 35
M_3 = merge M_2 and f_2 => 35 + 30 = 65
M_4 = merge M_3 and f_5 => 65 + 30 = 95

Therefore, the total number of operations are

15 + 35 + 65 + 95 = 210------------------ (B)

***Note:**Hence solution B is better than solution A.*

Working of algorithm with example: *Consider a pool of file **L** which contain **n** files.*

Step-1: Find first two minimum files.

Step-2: Merge them & add new merge file to the pool ***L***.

Step-3: Repeat step-1 &step-2 until no file remaining in the pool ***L*** for merging. e.g., ***L*** (F1, F2, F3, F4, F5) = (20, 30, 10, 5, 30).

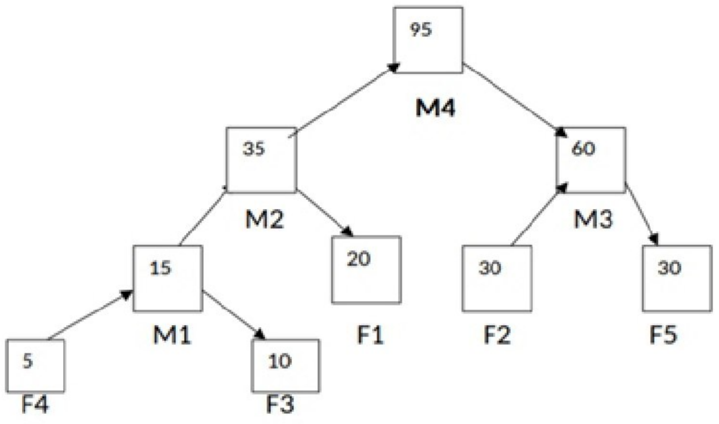


Fig. 4.1 Optimal Merge Pattern

4.6 HUFFMAN CODE

Huffman coding is a lossless data compression algorithm. Data can be encoded using it efficiently. It is the widely used technique for compressing the data. Huffman's greedy algorithm uses a table of the frequencies of occurrences of each character to build up an optimal way of representing each character as a binary string.

In this algorithm, a variable-length code is assigned to input different characters. The code length is related to how frequently characters are used. Most frequent characters have the smallest codes and longer codes for least frequent characters.There are mainly two parts- *First one to create a Huffman tree*, and another one to *traverse the tree to find codes*.

For example, consider some strings ***"YYYZXXYYX"***, the frequency of character ***Y*** is larger than ***X*** and the character ***Z*** has the least frequency. So, the length of the code for ***Y*** is smaller than ***X***, and code for ***X*** will be smaller than Z. The complexity for assigning the code for each character according to their frequency is*O(n log n)*.

4.6.1 Steps of Huffman Coding

Input: A string with different characters.

Output: The codes for each individual characters.

Input is an array of unique characters along with their frequency of occurrences and output is Huffman Tree.

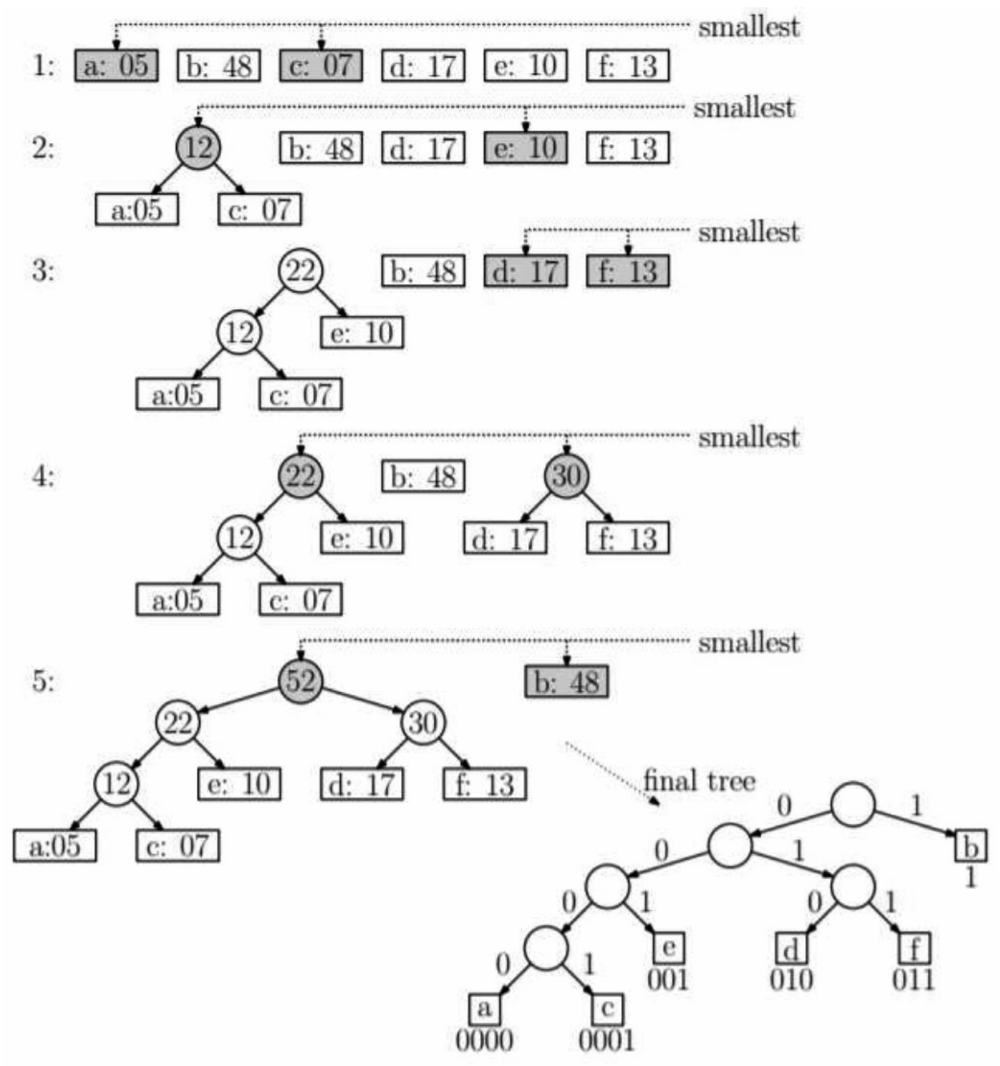
Step-1: Create a leaf node for each unique character and build a min-heap of all leaf nodes (min-heap is used as a priority queue). The value of frequency field is used to compare two nodes in min-heap. Initially, the least frequent character is at root.

Step-2: Extract two nodes with the minimum frequency from the min-heap.

Step-3: Create a new internal node with a frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min-heap.

Step-4: Repeat step-2 and step-3 until the heap contains only one node. The remaining node is the root node and the tree is complete.

Example:*Draw a tree using Huffman Coding given the input values for a = 05, b = 48, c = 07, d = 17, e = 10 and f = 13.*



smallest
1: a: 05 b: 48 c: 07 d: 17 e: 10 f: 13
2: 12 a:05 c: 07 b: 48 d: 17 e: 10 f: 13
3: 22 12 e: 10 a:05 c: 07 b: 48 d: 17 f: 13
4: 22 b: 48 30 d: 17 f: 13
5: 52 22 30 b: 48
final tree
0 1
a 0000
c 0001
e 001
d 010
f 011
b 1

Fig. 4.2 Huffman Coding

(Source:http://www.cs.umd.edu/class/fall2017/cmsc451-0101/Lects/lect06-greedy-huffman.pdf)

4.6.2 Advantage of Huffman Coding

Advantages of Huffman Encoding are given below:

- This encoding scheme results in saving lot of storage space, since the binary codes generated are variable in length.
- It generates shorter binary codes for encoding symbols/characters that appear more frequently in the input string.
- The binary codes generated are prefix-free.

4.6.3 Disadvantage of Huffman Coding

Disadvantages of Huffman Encoding are:

- It is a lossless data encoding scheme, as it achieves a lower compression ratio compared to lossy encoding techniques. Thus, lossless techniques like Huffman encoding are suitable only for encoding text and program files; and are unsuitable for encoding digital images.
- It is relatively slower process since it uses two passes —*one for building the statistical model* and *another for encoding*. Thus, the lossless techniques that use Huffman encoding are considerably slower than others.
- Since length of all the binary codes is different, it becomes difficult for the decoding software to detect whether the encoded data is corrupt. This can result in an incorrect decoding and subsequently, a wrong output.

4.6.4 Real-life Applications of Huffman Encoding

- It is widely used in compression formats like*GZIP, PKZIP and BZIP*.
- Multimedia codecs like *JPEG, PNG*, and *MP3*uses Huffman encoding.
- Huffman encoding still dominates the compression industry because newer arithmetic and range coding schemes are avoided due to their patent issues.

4.7 MINIMUM COST SPANNING TREES

A Minimum Spanning Tree or MST in short, is a subset of edges of a connected weighted undirected graph that connects all the vertices together with the minimum possible total edge weight. To derive an MST, ***Prim's algorithm*** or ***Kruskal's algorithm*** can be used. For a given undirected and connected graph G = (V, E), a spanning tree of the graph G is a tree that spans G (that is, it includes every vertex of G) and is a subgraph of G (every edge in the tree belongs to G).

Moreover, for a given a connected undirected graph, a spanning tree of that graph is a subgraph that is a tree and joined all vertices. A single graph can have many spanning trees. Let us consider an example shown in fig. 4.3.

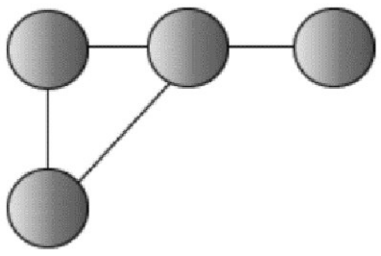


Fig. 4.3 Connected undirected graph

(*source: https://www.javatpoint.com/minimum-spanning-tree-introduction*)

For the connected undirected graph (Fig. 4.3) there may be many spanning trees e.g. Fig. 4.4 (a), (b), (c)

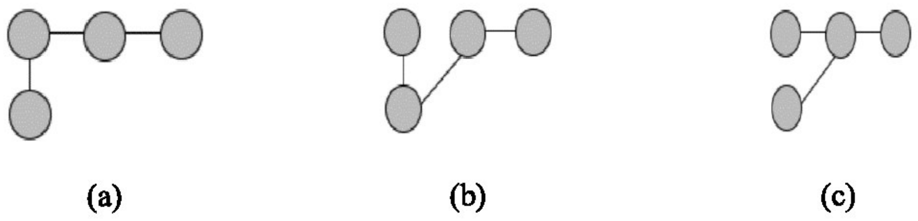


Fig. 4.4 Multiple spanning trees of fig.4.3

4.7.1 Properties of Spanning Trees

- There may be several minimum spanning trees of the same weight having the minimum number of edges.
- If all the edge weights of a given graph are the same, then every spanning tree of that graph is minimum.
- If each edge has a distinct weight, then there will be only one, unique minimum spanning tree.
- A connected graph G can have more than one spanning trees.
- A disconnected graph can't have to span the tree, or it can't span all the vertices.
- Spanning Tree doesn't contain cycles.
- Spanning Tree has (n-1)edges where n is the number of vertices.

4.7.2 Minimum Spanning Trees

Minimum Spanning Tree is a Spanning Tree which has minimum total cost. If we have a linked undirected graph with a weight (or cost) combine with each edge. Then the cost of spanning tree would be the sum of the cost of its edges.

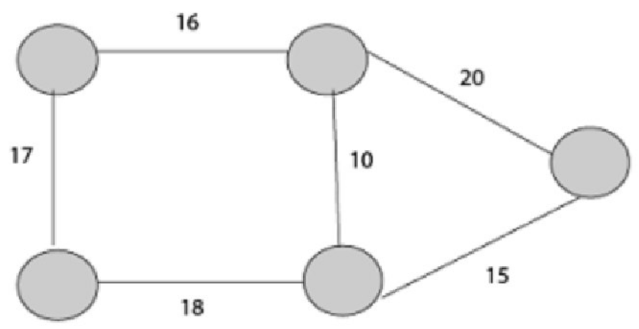


Fig. 4.5 Connected Undirected Graph
(source: https://www.javatpoint.com/minimum-spanning-tree-introduction)

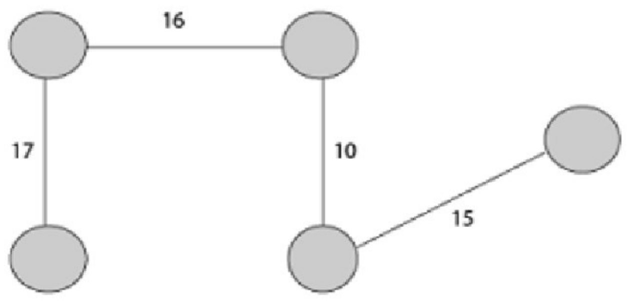


Fig. 4.6 Minimum cost Spanning Tree
(Total cost= 17+16+10+15=58)
(image source: https://www.javatpoint.com/minimum-spanning-tree-introduction)

4.7.3 Real Life Applications

Minimum spanning trees are used for network designs (i.e., telephone or cable networks). They are also used to find approximate solutions for complex mathematical problems like the *Traveling Salesman Problem (TSP)*. Other, diverse applications include:

1. Cluster Analysis
2. Real-time face tracking and verification (i.e., locating human faces in a video stream)
3. Protocols in computer science to avoid network cycles
4. Entropy based image registration
5. Max bottleneck paths

6. Dithering (adding white noise to a digital recording in order to reduce distortion)

4.7.4 Prim's Algorithm

It is a greedy algorithm and starts with an empty spanning tree. The basic concept is to maintain two sets of vertices— a) vertrices that included in MST, b) vertices not included yet.

In Prim's algorithm, first of all the edges are taken into consideration, and at every step minimum weight edge is picked; in second step, the minmum weight edge is picked out of all the remaining edges and so on. Consequently, it is clear that Prim's Algorithm use *Greedy approach* to find the minimum spanning tree. Moreover, the spanning tree from a starting position is grown unlike an edge in Kruskal's. Usually, a vertex is added to the growing spanning tree in Prim's algorithm.

Steps for finding MST using Prim's Algorithm

Step-1: Maintain two disjoint sets of vertices. One containing vertex that are in the growing spanning tree and other that are not in the growing spanning tree.

Step-2: Select the cheapest vertex that is connected to the growing spanning tree and is not in the growing spanning tree and add it into the growing spanning tree. This can be done using Priority Queues. Insert the vertices, that are connected to growing spanning tree, into the Priority Queue.

Step 3: Check for cycles,to do that, mark the nodes which have been already selected and insert only those nodes in the Priority Queue that are not marked.
Example:*Let us understand with the following example given in figure-4.7*

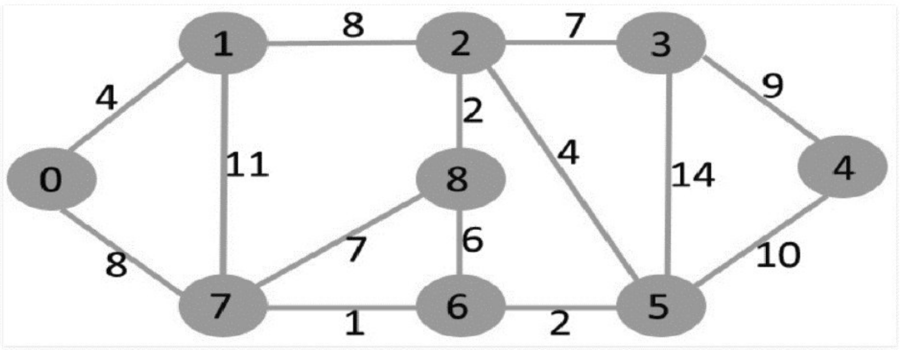


Fig. 4.7 Prim's Algorithm
(image source:https://www.geeksforgeeks.org/prims-mst-for-adjacency-list-representation-greedy-algo-6/)

Pick the vertex with minimum key value and not already included in MST (not in mstSET). The vertex 1 is picked and added to mstSet. So mstSet now becomes {0, 1}. Update the key values of adjacent vertices of 1. The key value of vertex 2 becomes 8.

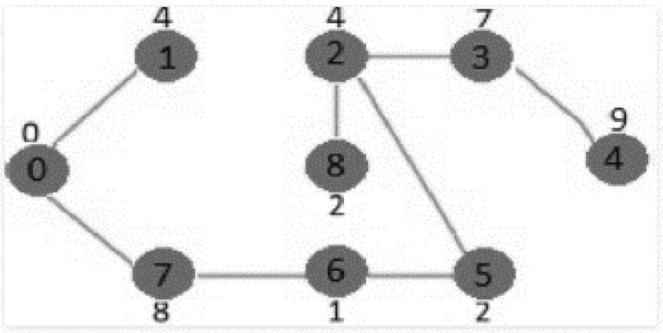


Fig. 4.8 Prim's Algorithm

(image source:https://www.geeksforgeeks.org/prims-mst-for-adjacency-list-representation-greedy-algo-6/)

Time Complexity: The time complexity of the *Prim's Algorithm* is *O((V+E)logV)* because each vertex is inserted in the priority queue only once and insertion in priority queue take logarithmic time.

4.7.5 Kruskal's Algorithm

Kruskal's Algorithm builds the spanning tree by adding edges one by one into a growing spanning tree. Kruskal's algorithm follows greedy approach as in each iteration, it finds an edge which has least weight and add it to the growing spanning tree. It is an algorithm to construct a Minimum Spanning Tree for a connected weighted graph. If the graph is not linked, then it finds a Minimum Spanning Tree.

Steps for finding MST using Kruskal's Algorithm:

1. Arrange the edge of G in order of increasing weight.
2. Starting only with the vertices of G and proceeding sequentially add each edge which does not result in a cycle, until (n - 1) edges are used.
3. Exit.

MST- KRUSKAL (G, w)

```
1. A ←Ø
2. for each vertex v ∈ V [G]
3. do MAKE - SET (v)
4. sort the edges of E into non decreasing order by weight w
5. for each edge (u, v) ∈ E, taken in non decreasing order by        weight
6. do if FIND-SET (μ) ≠ if FIND-SET (v)
```

```
7. then A← A ∪ {(u, v)}
8. UNION(u, v)
9. return A
```

Analysis: E is the number of edges in the graph and V is the number of vertices. Kruskal's Algorithm can be shown to run in O (E log E) time, or simply, O (E log V) time, all with simple data structures. These running times are equivalent because of the following reasons:

- E is at most V^2 and log V^2= 2 x log V is O (log V).
- If we ignore isolated vertices, which will each their components of the minimum spanning tree, $V \leq 2$ E, so log V is O (log E).

Thus the total time isO(ElogE) = O(ElogV)

Example: *Find the Minimum Spanning Tree of the following graph using Kruskal's algorithm.*
(source: https://www.javatpoint.com/kruskals-minimum-spanning-tree-algorithm)

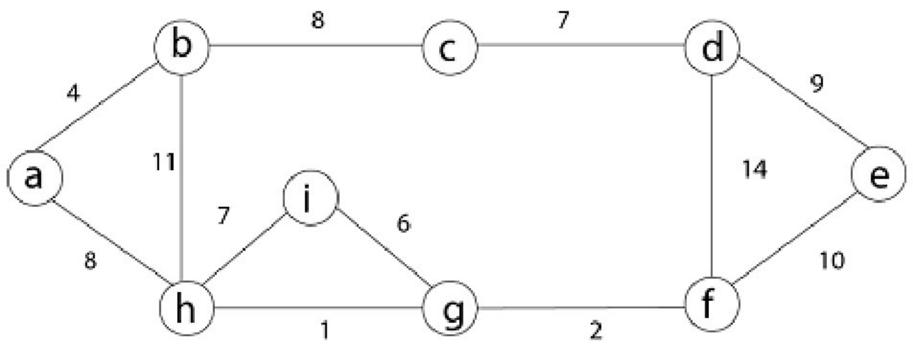


Fig. 4.9 Example

(image source: https://www.javatpoint.com/minimum-spanning-tree-introduction)

Solution: First we initialize the set A to the empty set and create |v| trees, one containing each vertex with MAKE-SET procedure. Then sort the edges in E into order by non-decreasing weight. There are 9 vertices and 12 edges. So MST formed (9-1) = 8 edges

Weight	**Source**	**Destination**
1	H	G
2	G	F

4	A	B
6	I	G
7	H	I
7	C	D
8	B	C
8	A	H
9	D	E
10	E	F
11	B	H
14	D	F

Now, check for each edge (u, v) whether the endpoints u and v belong to the same tree. If they do then the edge (u, v) cannot be supplementary. Otherwise, the two vertices belong to different trees, and the edge (u, v) is added to A, and the vertices in two trees are merged in by union procedure.

Step1: So, first take (h, g) edge

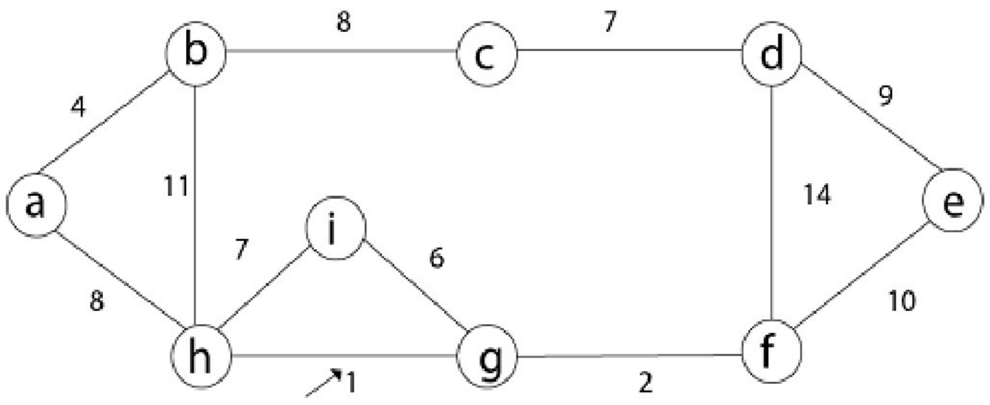


Fig. 4.10 Step-1

(image source: https://www.javatpoint.com/minimum-spanning-tree-introduction)

Step 2: then (g, f) edge.

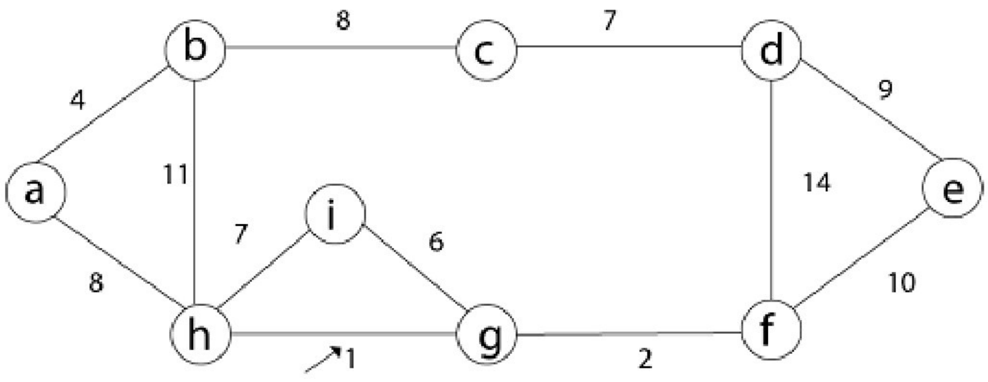


Fig. 4.11 Step-2

(image source: https://www.javatpoint.com/minimum-spanning-tree-introduction)

Step 3: then (a, b) and (i, g) edges are considered, and the forest becomes

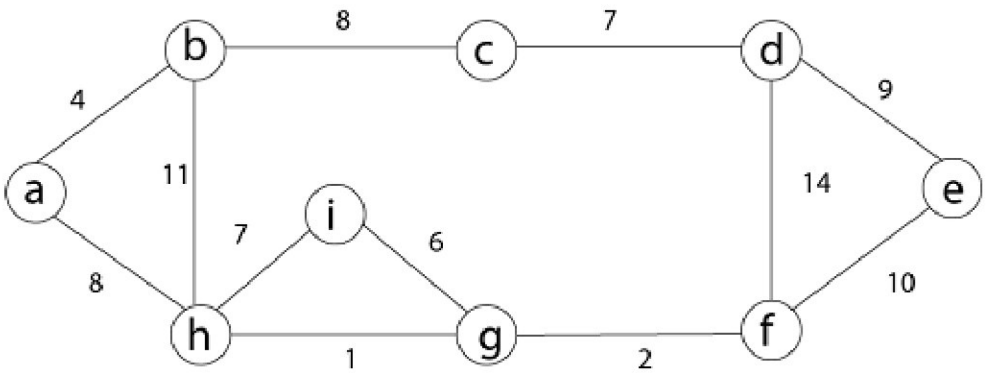


Fig. 4.12Step-3

(image source: https://www.javatpoint.com/minimum-spanning-tree-introduction)

Step-4: Now, edge (h, i). Both h and i vertices are in the same set. Thus it creates a cycle. So this edge is discarded.

Then edge (c, d), (b, c), (a, h), (d, e), (e, f) are considered, and the forest becomes.

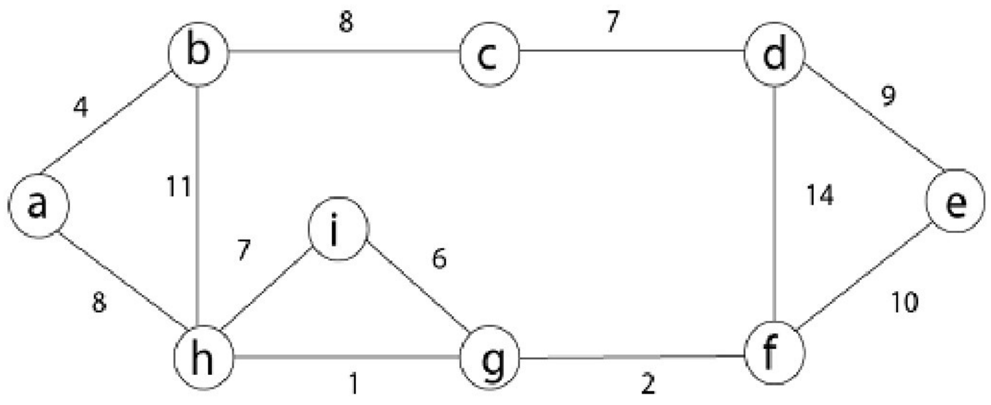


Fig. 4.13Step-4

(image source: https://www.javatpoint.com/minimum-spanning-tree-introduction)

Step-5: In (e, f) edge both endpoints e and f exist in the same tree so discarded this edge. Then (b, h) edge, it also creates a cycle.

Step-6: After that edge (d, f) and the final spanning tree is shown as in dark lines.

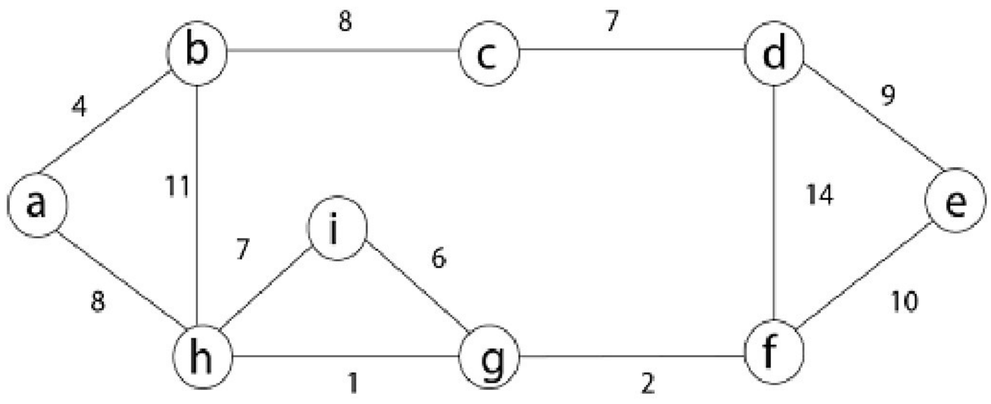


Fig. 4.14Step-6

(image source: https://www.javatpoint.com/minimum-spanning-tree-introduction)

Step-7: This step will be required Minimum Spanning Tree because it contains all the 9 vertices and (9 - 1) = 8 edges

e → f, b → h, d → f [cycle will be formed]

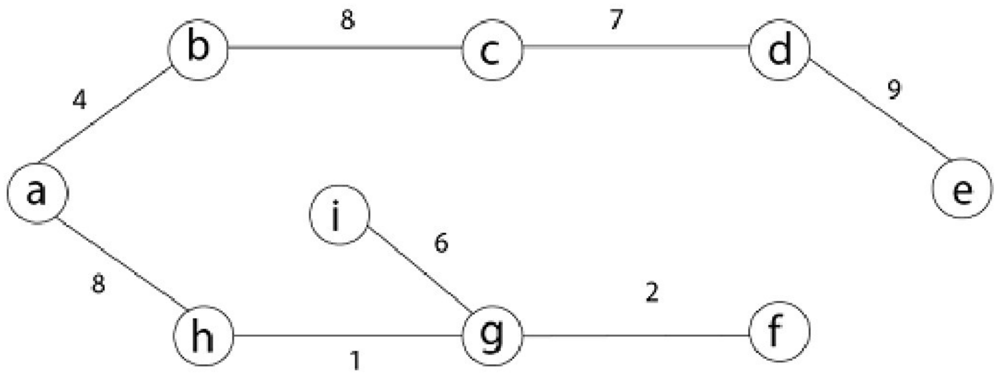


Fig. 4.15 Step-7

(image source: https://www.javatpoint.com/minimum-spanning-tree-introduction)

CHECK YOUR PROGRESS

- Define spanning tree.
- What is the main difference between Prim's and Kruskal's algorithm?
- Write any two applications of Kruskal's algorithm.

4.8 SINGLE SOURCE SHORTEST PATH

In *single-source shortest path problem*, shortest paths is obtained from a source vertex to all other vertices in the graph. On the other hand, in *single-destination shortest path problem,* shortest path is obtained from all vertices in the directed graph to a single destination vertex. This problem is also sometimes known as the single-pair shortest path problem.

Now, to distinguish it the following variations are taken into consideration:

- The *single-source shortest path problem*, in which we have to find shortest paths from a source vertex to all other vertices in the graph.
- The *single-destination shortest path problem*, in which we have to find shortest paths from all vertices in the directed graph to a single destination vertex. This can be reduced to the single-source shortest path problem by reversing the arcs in the directed graph.
- The *all-pairs shortest path problem*, in which we have to find shortest paths between every pair of vertices in the graph.
- Given a connected weighted directed graph G (V, E), associated with each edge ⟨u, v⟩∈E, there is a weight w (u, v). The *single source shortest paths (SSSP)* problem is to find a shortest path from a given source r to

every other vertex v∈V-{r}. The weight (length) of a path p=⟨ v_0, v_1 …, v_k ⟩ is the sum of the weights of its constituent edges:

$$w(P) = \sum_{i=1}^{k} w(v_{i-1}v_i)$$

- The weight of a shortest path from u to v is defined by δ (u, v)=min{w(p): p is a path from u to v}.

4.8.1The Bellman-Ford Algorithm

This algorithm solves the single source shortest path problem of a directed graph *G = (V, E)* in which the edge weights may be negative. Moreover, this algorithm can be applied to find the shortest path, if there is no negative weighted cycle. Moreover, negative edge weights are found in various applications of graphs, hence this algorithm is useful in such situations where negative edge weights are found. If a graph contains a negative cycle (i.e. a cycle whose edges sum to a negative value) that is reachable from the source, then there is no cheapest path; any path that has a point on the negative cycle can be made cheaper by one more walk around the negative cycle. In such a case, the *Bellman-Ford algorithm* can detect and report the negative cycle.

Given a weighted directed graph G = (V, E) with source s and weight function w: E → R, the Bellman-Ford algorithm returns a Boolean value indicating whether or not there is a negative weight cycle that is attainable from the source. If there is such a cycle, the algorithm produces the shortest paths and their weights. The algorithm returns TRUE if and only if a graph contains no negative - weight cycles that are reachable from the source.

Recurrence relation: $dist^k$ [u] = [min[$dist^{k-1}$ [u],min[$dist^{k-1}$ [i]+cost [i,u]]] as i except u.

Where, k → k is the source vertex; u → u is the destination vertex; i → no of edges to be scanned concerning a vertex.

Algorithm: ***Bellman-Ford-Algorithm (G, w, s)***

```
1. INITIALIZE - SINGLE - SOURCE (G, s)
2. for i ← 1 to |V[G]| - 1
3. do for each edge (u, v) ∈ E [G]
4. do RELAX (u, v, w)
5. for each edge (u, v) ∈ E [G]
6. do if d [v] > d [u] + w (u, v)
7. then return FALSE.
8. return TRUE.
```

Example: *Here first we list all the edges and their weights.*
(source: https://www.javatpoint.com/bellman-ford-algorithm)

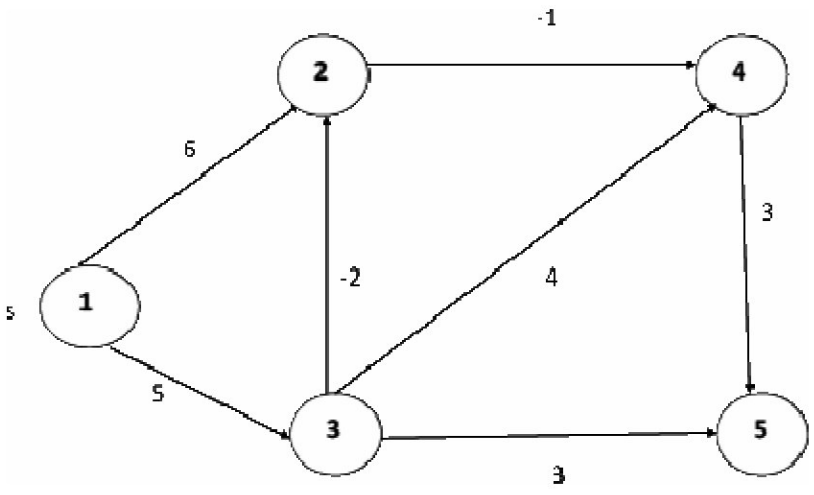


Fig. 4.16 Example of Bellman Ford Algorithm

(image source: https://www.javatpoint.com/bellman-ford-algorithm)

Solution: $dist^k[u] = [\min[dist^{k-1}[u], \min[dist^{k-1}[i]+cost[i,u]]]$ as $i \neq u$.

$\longrightarrow$ Vertices

No of Edges Traversed $\uparrow$

	1	2	3	4	5
1	0	6	5	∞	∞
2	0	3	5	5	8
3	0	3	5	2	8
4	0	3	5	2	5

$dist^2[2]=\min[dist^1[2], \min[dist^1[1]+cost[1,2], dist^1[3]+cost[3,2], dist^1[4]+cost[4,2], dist^1[5]+cost[5,2]]]$

$Min = [6, 0 + 6, 5 + (-2), \infty + \infty, \infty + \infty] = 3$

$dist^2[3]=\min[dist^1[3], \min[dist^1[1]+cost[1,3], dist^1[2]+cost[2,3], dist^1[4]+cost[4,3], dist^1[5]+cost[5,3]]]$

Min = [5, 0 +∞, 6 +∞, ∞ + ∞ , ∞ + ∞] = 5

$dist^2$ [4]=min[$dist^1$ [4],min[$dist^1$ [1]+cost[1,4],$dist^1$ [2]+cost[2,4],$dist^1$ [3]+cos t[3,4],$dist^1$ [5]+cost[5,4]]]

Min = [∞, 0 +∞, 6 + (-1), 5 + 4, ∞ +∞] = 5

$dist^2$ [5]=min[$dist^1$ [5],min[$dist^1$ [1]+cost[1,5],$dist^1$ [2]+cost[2,5],$dist^1$ [3]+cos t[3,5],$dist^1$ [4]+cost[4,5]]]

Min = [∞, 0 + ∞,6 + ∞,5 + 3, ∞ + 3] = 8

$dist^3$ [2]=min[$dist^2$ [2],min[$dist^2$ [1]+cost[1,2],$dist^2$ [3]+cost[3,2],$dist^2$ [4]+cos t[4,2],$dist^2$ [5]+cost[5,2]]]

Min = [3, 0 + 6, 5 + (-2), 5 + ∞ , 8 + ∞] = 3

$dist^3$ [3]=min[$dist^2$ [3],min[$dist^2$ [1]+cost[1,3],$dist^2$ [2]+cost[2,3],$dist^2$ [4]+cos t[4,3],$dist^2$ [5]+cost[5,3]]]

Min = [5, 0 + ∞, 3 + ∞, 5 + ∞,8 + ∞] = 5

$dist^3$ [4]=min[$dist^2$ [4],min[$dist^2$ [1]+cost[1,4],$dist^2$ [2]+cost[2,4],$dist^2$ [3]+cos t[3,4],$dist^2$ [5]+cost[5,4]]]

Min = [5, 0 + ∞, 3 + (-1), 5 + 4, 8 + ∞] = 2

$dist^3$ [5]=min[$dist^2$ [5],min[$dist^2$ [1]+cost[1,5],$dist^2$ [2]+cost[2,5],$dist^2$ [3]+cos t[3,5],$dist^2$ [4]+cost[4,5]]]

Min = [8, 0 + ∞, 3 + ∞, 5 + 3, 5 + 3] = 8

$dist^4$ [2]=min[$dist^3$ [2],min[$dist^3$ [1]+cost[1,2],$dist^3$ [3]+cost[3,2],$dist^3$ [4]+cos t[4,2],$dist^3$ [5]+cost[5,2]]]

Min = [3, 0 + 6, 5 + (-2), 2 + ∞, 8 + ∞] =3

$dist^4$ [3]=min[$dist^3$ [3],min[$dist^3$ [1]+cost[1,3],$dist^3$ [2]+cost[2,3],$dist^3$ [4]+cos t[4,3],$dist^3$ [5]+cost[5,3]]]

Min = 5, $0 + \infty$, $3 + \infty$, $2 + \infty$, $8 + \infty$] =5

$dist^4$ [4]=min[$dist^3$ [4],min[$dist^3$ [1]+cost[1,4],$dist^3$ [2]+cost[2,4],$dist^3$ [3]+cos t[3,4],$dist^3$ [5]+cost[5,4]]]

Min = [2, $0 + \infty$, $3 + (-1)$, $5 + 4$, $8 + \infty$] = 2

$dist^4$ [5]=min[$dist^3$ [5],min[$dist^3$ [1]+cost[1,5],$dist^3$ [2]+cost[2,5],$dist^3$ [3]+cos t[3,5],$dist^3$ [5]+cost[4,5]]]

Min = [8, $0 +\infty$, $3 + \infty$, 8, 5] = 5

***Note:** The first for loop is used for initialization, which runs in O(V) times. The next for loop runs |V - 1| passes over the edges, which takes O(E) times. Hence, Bellman-Ford algorithm runs in O (V, E) time.*

4.8.2 Dijkstra's algorithm

Dijkstra's algorithm is an algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, road networks. It was given by computer scientist Edsger W. Dijkstra in 1956 and published three years later. The algorithm exists in many variants. Dijkstra's original algorithm found the shortest path between two given nodes, but a more common variant fixes a single node as the source node and finds shortest paths from the source to all other nodes in the graph, producing a shortest-path tree.

Following steps can be used for it—

Step-1: Initialize distances according to the algorithm.

Step-2: Pick first node and calculate distances to adjacent nodes.

Step-3: Pick next node with minimal distance; repeat adjacent node distance calculations.

Step-4: Final result of shortest-path tree.

Dijkstra's algorithm solves the *single-source shortest-paths* problem on a directed weighted graph G = (V, E), where all the edges are non-negative (i.e., w (u, v) $\geq$ 0 for each edge (u, v) Є E).In the following algorithm, one function Extract-Min() is used, which extracts the node with the smallest key.

Algorithm: *Dijkstra's-Algorithm (G, w, s)*

```
1. INITIALIZE - SINGLE - SOURCE (G, s)
2. S←∅
3. Q←V [G]
4. while Q ≠ ∅
5. do u ← EXTRACT - MIN (Q)
6. S ← S ∪ {u}
7. for each vertex v ∈ Adj [u]
8. do RELAX (u, v, w)
```

Analysis: The running time of Dijkstra's algorithm on a graph with edges E and vertices V can be expressed as a function of |E| and |V| using the Big - O notation. The simplest implementation of the Dijkstra's algorithm stores vertices of set Q in an ordinary linked list or array, and operation Extract - Min (Q) is simply a linear search through all vertices in Q. In this case, the running time is O ($|V^2|$+|E|=O(V^2).

Example:*Dijkstra's algorithm Source (s)*

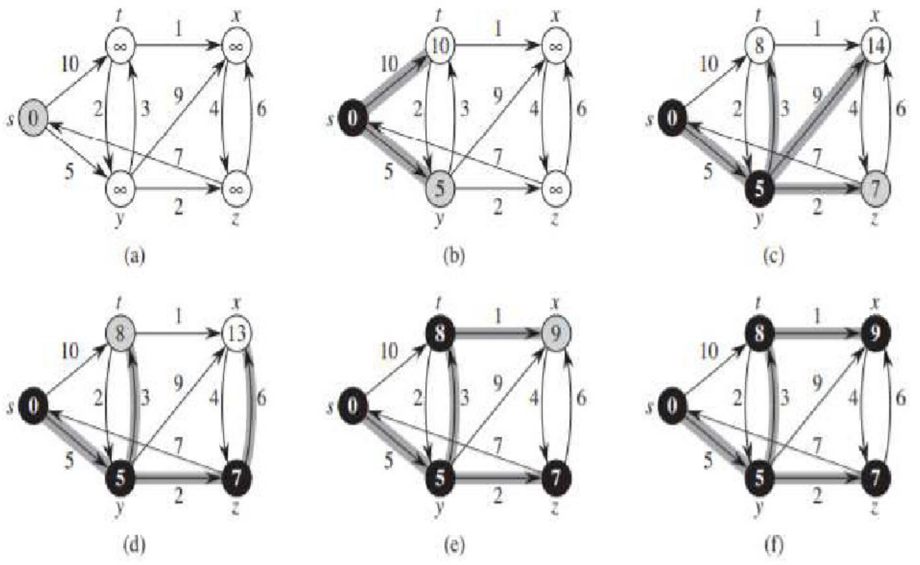


Fig. 4.16 The execution of Dijkstra's algorithm. The source s is the leftmost vertex. The shortest-path estimates appear within the vertices, and shaded edges indicate predecessor values. Black vertices are in the set S, and white vertices are in the
min-priority queue Q = V - S

(image source:https://edutechlearners.com/download/Introduction_to_algorithms-3rd%20Edition.pdf)

Analysis: The complexity of this algorithm is fully dependent on the implementation of Extract-Min function. If extract min function is implemented using linear search, the complexity of this algorithm is *$O(V^2 + E)$*.

Disadvantage of Dijkstra's Algorithm:

- It does a blind search, so wastes a lot of time while processing.
- It can't handle negative edges.
- It leads to the acyclic graph and most often cannot obtain the right shortest path.
- We need to keep track of vertices that have been visited.

CHECK YOUR PROGRESS

- Give the meaning of single source shoretest path.
- Compare Bellman ford and Dijkstra algorithm.
- Write the use of Dijkstra algorithm.

4.9 SUMMARY

- To summarize, the chapter defined the greedy paradigm, showed how greedy optimization and recursion, can help you obtain the best solution up to a point.
- A greedy algorithm is a simple, intuitive algorithm that is used in optimization problems. The algorithm makes the optimal choice at each step as it attempts to find the overall optimal way to solve the entire problem. However, in many problems, a greedy strategy does not produce an optimal solution.
- The *Greedy algorithm* is widely taken into application for problem solving in many languages as Greedy Algorithm*Python, C, C#, PHP, Java*, etc.
- The activity selection of *Greedy algorithm* example was described as a strategic problem that could achieve maximum throughput using the greedy approach. In the end, the demerits of the usage of the greedy approach were explained.
- In Knapsack problem, the user basically decides to allocate the most valuable item or task or resource to the fixed-size block, or in other words when he has only fixed budget/time. Moreover, it basically refers to the most common real time problem of packing something

which is very costly or important item in a container without mixing it with other item/items.

- In job sequencing problem the main goal is to find the sequence of jobs completed within deadlines which give highest profit. The sequencing of jobs on a single processor with defined deadline constraints is called as Job Sequencing with Deadlines.
- Optimal merge pattern is a pattern that relates to the merging of two or more sorted files in a single sorted file. In this way we need to find an optimal solution, where the resultant file would be generated in minimum time. If the number of sorted files are given, there are many ways to merge them into a single sorted file. This merge can be performed pair wise.
- Huffman coding is a lossless data compression algorithm. It is the widely used technique for compressing the data. Huffman's greedy algorithm uses a table of the frequencies of occurrences of each character to build up an optimal way of representing each character as a binary string.
- A Minimum Spanning Tree (*MST*) is a subset of edges of a connected weighted undirected graph that connects all the vertices together with the minimum possible total edge weight. To derive an MST, ***Prim's algorithm*** or ***Kruskal's algorithm*** can be used.
- Minimum Spanning Tree is a Spanning Tree which has minimum total cost. If we have a linked undirected graph with a weight (or cost) combine with each edge. Then the cost of spanning tree would be the sum of the cost of its edges.
- In Prim's algorithm, first of all the edges are taken into consideration, and at every step minimum weight edge is picked; in second step, the minmum weight edge is picked out of all the remaining edges and so on.
- Kruskal's algorithm follows greedy approach, as in each iteration it finds an edge which has least weight and add it to the growing spanning tree. It is an algorithm to construct a Minimum Spanning Tree for a connected weighted graph.
- The ***single-source shortest path problem***, in which we have to find shortest paths from a source vertex v to all other vertices in the graph.
- Bellman-Ford algorithm solves the single source shortest path problem of a directed graph e.g. $G = (V, E)$, in which the edge weights may be negative. Moreover, this algorithm can be applied to find the shortest path, if there does not exist any negative weighted cycle.
- ***Dijkstra's algorithm*** is an algorithm for finding the shortest paths between nodes in a graph, which may represent, for

example, road networks. It was conceived by computer scientist Edsger W. Dijkstra in 1956 and published three years later.

4.11 TERMINAL QUESTIONS

1. What do you understand by a Greedy Algorithm?
2. Write the significance of using Greedy Algorithms.
3. Compare Dynamic Programming and Greedy Algorithms.
4. What is the difference between Greedy and Heuristic algorithm?
5. Describe Prim's and Kruskal algorithm with suitable example.
6. Explain the single source shortest path using suitable example.
7. Is there any proof to decide if Greedy approach will produce the best solution?
8. Given a directed acyclic weighted graph, how to find the shortest path from a source s to a destination t in O(V+E) time?
9. Describe a greedy algorithm to make change consisting of quarters, dimes, nickels, and pennies. Prove that your algorithm yields an optimal solution.
10. Give a set of coin denominations for which the greedy algorithm does not yield an optimal solution.
11. Suppose a data file contains a sequence of 8-bit characters such that all 256 characters are about as common: the maximum character frequency is less than twice the minimum character frequency. Prove that Huffman coding in this case is no more efficient than using an ordinary 8-bit fixed-length code.
12. Prove that the fractional knapsack problem has the greedy-choice property.
13. Suppose you are given a connected graph G, with edge costs that are all distinct. Prove that G has a unique minimum spanning tree.

Unit 5: Dynamic Programming

Structure

5.0 INTRODUCTION

Dynamic programming is a one of the good design techniques used for optimization problems. As far as Divide and Conquer algorithm is concerned, it solves the problems recursively after partitioning the problem into sub-problems. But Divide and Conquer algorithm is a lengthy process and it uses unnecessary things as it uses recursive approach which is costly and time taking (because it solves same sub-problem many times using recursion). Dynamic Programming, on the other hand uses non-recursive approach to solve the problems. It uses bottom up approach in contrary to the top down approach used in Divide and Conquer.Dynamic programming is a general method, widely applied in operation-research to solve optimization problems and computational biology. It uses Bottom-up approach because— it starts with smallest sub-problem; move from smaller sub-problem to the big sub-problems; finally solves the original problem.

One common thing in both the algorithms i.e. bottom-up and top-down approach, both break the problem into sub-problems. Dynamic programming works on the basis of four components— stages, states, decision, and optimal policy. It can be effectively and efficiently used for solving many real time problems like scheduling the jobs assigned to the CPU hence to increase the CPU usage, flight navigation in air traffic control systems, control systems, robotic applications, design problems of reliability, finding shortest path, optimization problems of Mathematics, knapsack problems, the DNA-related sequencing problems etc.

5.1 OBJECTIVES

At the end of this unit you will come to know about:

- o Introduction of dynamic programming
- o Difference between top down and bottom up approach
- o Applications of dynamic programming
- o Capital budgeting problem
- o Multistage graph
- o Matric chain multiplication
- o Knapsack problem
- o Travelling salesman problem

5.2 INTRODUCTION TO DYNAMIC PROGRAMMING

Like the Divide and Conquer method, Dynamic programming solves problems by combining the solutions to sub-problems. Obtaining solution by dividing the problem in small problem is common in both but divide and conquer uses top down approach while dynamic programming uses bottom up approach. In this context, in dynamic programming, the word "programming" refers to problem-solving using a tabular structure as the result once obtained is memorized and stored in a table for further reference. Divide and conquer algorithms partition the problem into disjoint sub-problems, solve the sub-problems recursively, and then combine their solutions to solve the original problem. In contrast, dynamic programming applies when the sub-problems overlap-that is, when sub-problems share subsub-problems.

In this context, a divide-and-conquer algorithm does more work than necessary, repeatedly solving the common subsub-problems because it uses recursion. A dynamic-programming algorithm on the other hand solves each subsub-problem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time. Dynamic programming uses non-recursive approach as it does not use repetition. Whenever encounter it again, look up the solutionin the table. Accordingly, the time complexitywill be the size of the table times the cost of computing an entry.

In dynamic programming, the divide and conquer principle is carried to an extreme: when we don't know exactly whichsmaller problems to solve, we simply solve them all, then store the answersaway to be used later in solving larger problems. But, there are two main issues with the application of this technique. First, it may not always be possible to combine the solutions of two problems into a larger one. Second, there may be an unacceptablylarge number of small problems to solve. No one has precisely characterizedwhich problems can be effectively solved with dynamic programming.

Dynamic programming is a paradigm design which works on the basis of "principle of optimality" in which the optimization problem is solved by combining the small sub-solutions into bigger sub-solutions until the final solution is obtained for the original optimal problem.

Basic idea states:*1) Break the problems into sub-problems using bottom-up approach.2) Solve smallest sub-problem first, and then use the solution to the sub-problems to solve the next smallest till solve the original problem.* Here, time will be usually dominated by the number of possiblesub-problems.

Note:A bottom-up approach means— 1) start from the smallest sub-problems; 2)obtain the solution of increasing size after combining the solution of step-1; 3) repeat steps-1 & 2 until solution for the original problem is obtained.

5.2.1 Characteristics of Dynamic Programming

Dynamic programming works well when the following features a problem has

- **Optimal Substructure:** If an optimal solution contains optimal sub solutions then a problem exhibits optimal substructure.
- **Overlapping sub-problems:** When a recursive algorithm would visit the same sub-problems repeatedly, then a problem has overlapping sub-problems.

If a problem has optimal substructure, then an optimal solution can be defined recursively. If a problem has overlapping sub-problems, then we can improve on a recursive implementation by computing each sub-problem only once.

On the other hand, if a problem doesn't have optimal substructure, there is no basis for defining a recursive algorithm to find the optimal solutions. If a problem doesn't have overlapping sub problems, we don't have anything to gain by using dynamic programming.

***Note:** If the sub-problems' space is adequate (i.e. polynomial in the size of the input), dynamic programming can be better efficient than recursion.*

5.2.2 Elements of Dynamic Programming

Three elements basically in dynamic programming

1. **Substructure:** Decomposing the problem into smaller problems is called sub structuring, and hence express the solution of original problem in smaller problems.
2. **Table structure:** the result of the solution to the sub-problem is stored in the table so that it can be reused accordingly if same result occurs.
3. **Bottom-up approach:** Using table to combine the solution of smaller problems and later arriving to the final solution of original problem.

5.2.3 Components of Dynamic Programming

1. **Stages:** The problem can be broken into sub-problems called stages. A stage is nothing but a small part of the given problem.
2. **States:** Each stage can have several states associated with it.

3. **Decision:** At each stage, there can be multiple choices out of which one of the best decisions should be taken. The decision taken at every stage should be optimal; this is called a stage decision.
4. **Optimal policy:** A rule which determines the decision at each stage is called optimal policy; a policy is called an optimal policy if it is globally optimal. This is known as Bellman principle of optimality

5.3 GENERAL METHOD & APPLICATIONS

Dynamic programming is typically applied to optimization problems. Such problems can have many possible solutions. Each solution has a value, and we wish tofind a solution with the optimal (minimum or maximum) value. We call such asolution an optimal solution to the problem, as opposed to the optimal solution,since there may be several solutions that achieve the optimal value.

When developing a dynamic-programming algorithm, we follow a sequence offour steps:

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution, typically in a bottom-up fashion.
4. Construct an optimal solution from computed information.

It is to be noted here that steps 1–3 form the basis of a dynamic-programming solution to a problem. If weneed only the value of an optimal solution, and not the solution itself, then step- 4 might be omitted. Moreover, when step-4 is performed, we sometimes maintain the additionalinformation during step-3 too, so that an optimal solution could be easily constructed.

The Dynamic Programming techniquescan be applied to solve the following problems for finding an optimal solution:

- Matrix Chain Multiplication
- Longest Common Subsequence
- Travelling Salesman Problem
- Knapsack
- World Series
- Capital budgeting problem

5.4 MATRIX CHAIN MULTIPLICATION

It is a Method under Dynamic Programming in which previous output is taken as input for next. Here, Chain means one matrix's column is equal to the second matrix's row [always].

In general:

If $A = \lfloor a_{ij} \rfloor$ is a p x q matrix

$B = \lfloor b_{ij} \rfloor$ is a q x r matrix

$C = \lfloor c_{ij} \rfloor$ is a p x r matrix

Then

$$AB = C \text{ if } c_{ij} = \sum_{k=1}^{q} a_{ik}\, b_{kj}$$

Suppose that the six matrices are to be multiplied together, i.e. A with dimension 4x2, B with dimension 2x3, C with dimension 3x1, D with dimension 1x2, E with dimension 2x2, and F with dimension 2x3. Of course, for the multiplications to be valid, the number of columns in one matrix must be the same as the number of rows in the next. But the total number of scalar multiplications involved depends on the order in which the matrices are multiplied.

For example, we could proceed from left to right: multiplying A by B, we get a 4x3 matrix after using 24 scalar multiplications. Multiplying this result by C gives a 4x1 matrix after 12 more scalar multiplications. Multiplying this result by D gives a 4x2 matrix after 8 more scalar multiplications. Continuing in this way, we get a 4x3 result after a grand total of 84 scalar multiplications. But if we proceed from right to left instead, we get the same 4x3 result with only 69 scalar multiplications.

Many other orders are clearly possible. The order of multiplication can be expressed by parenthesization: **for example,** the left-to-right order described above is the ordering (((((A*B)*C)*D)*E)*F), and the right-to-left order is (A*(B*(C*(D*(E*F))))). Any legal parenthesization will lead to the correct answer, but which leads to the fewest scalar multiplications? Very substantial savings can be achieved when large matrices are involved: **for example**, if matrices B, C, and F in the example above were to each have a dimension of 300 where their dimension is 3, then the left-to-right order will require 6024 scalar multiplications but the right-to-left order will use an astronomical 274,200.

Dynamic programming is an algorithm that solves the problem of matrix-chain multiplication. We are given a sequence (chain) $<A_1, A_2, \ldots, A_n>$ of n matrices to be multiplied, and we wish to compute the product

$$A_1A_2\ldots\ldots A_n. \qquad (5.1)$$

We can evaluate the expression (5.1) using the standard algorithm for multiplying pairs of matrices as a subroutine, once we have parenthesized it to resolve all ambiguities in how the matrices are multiplied together. Matrix multiplication is associative, and so all parenthesization yield the same product. A product of matrices is *fully parenthesized* if it is either a single matrix or the product of two fully parenthesized matrix products, surrounded by parentheses. **For example**, if the chain of matrices is $<A_1, A_2, A_3, A_4>$ we can fully parenthesize the product $A_1A_2A_3 A_4$ in five distinct ways:

$(A_1 (A_2 (A_3 A_4)))$,
$(A_1 ((A_2A_3) A_4))$,
$((A_1A_2) (A_3A_4))$,
$((A_1 (A_2 A_3)) A_4)$,
$(((A_1 A_2) A_3) A_4)$.

How we parenthesize a chain of matrices can have a dramatic impact on the cost of evaluating the product. Consider first the cost of multiplying two matrices. The standard algorithm is given by the following pseudocode. The attributes rows and columns are the numbers of rows and columns in a matrix.

Matrix-Multiply (A, B)

```
1 if A.columns ≠ B.rows
2       error "incompatible dimensions"
3 else let C be a new A.rows x B.columns matrix
4       for i= 1 to A.rows
5           for j = 1 to B.columns
6               $c_{ij} = 0$
7               for k = 1 to A.columns
8                   $c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$
9       return C
```

We can multiply two matrices A and B only if they are compatible: the number of columns of A must equal the number of rows of B. If A is a $p\times q$ matrix and B is a $q\times r$ matrix, the resulting matrix C is a $p\times r$ matrix. The time to compute C is dominated by the number of scalar multiplications in line 8, which is pqr.

The matrix-chain multiplication problem can be stated as follows: given a chain $<A_1, A_2, \ldots, A_i, \ldots, A_n>$ of n matrices, where for $i=1,2,3,\ldots,n$, matrix A_i

has dimension $p_{i-1} \times p_i$, fully parenthesize the product $A_1A_2.......A_n$ in a way that minimizes the number of scalar multiplications.

The optimal cost is computed by using a tabular, bottom-up approach in dynamic programming approach through the procedure **MATRIXCHAIN-ORDER** given below. This procedure assumes that matrix Ai has dimensions $p_{i-1} \times p_i$ for i=1,2,3,.....,n. Its input is a sequence p= $<p_0, p_1, ..., p_n>$, where p.*length* = n + 1. The procedure uses an auxiliary table m[1..n, 1..n] for storing the m[i, j] costs and another auxiliary table s[1..n-1,2..n] that records which index of k achieved the optimal cost in computing m[i, j]. The minimum cost of parenthesizing the product $A_iA_{i+1}.....A_j$becomes

m[i, j]= min{ m[i, k] +m[k+1,j]+ $p_{i-1}p_kp_j$} for i<=k<j if i<j otherwise 0.

Matrix-Chain-Order(p)

```
1  n = p.length-1
2  let m[1..n, 1..n] and s[1..n-1,2..n] be new tables
3  for i = 1 to n
4      m [i, i] = 0
5  for l = 2 to n                      // l is the chain length
6      for i = 1 to n-l+1
7          j= i + l - 1
8          m [i, j] = ∞
9          for k = i to j -1
10             q = m [i, k] + m [k+1, j] + $p_{i-1}p_kp_j$
11             if q < m [i, j]
12                 m [i, j] = q
13                 s [i, j]= k
14  return m and s
```

The MATRIX-CHAIN-ORDER determines the optimal number of scalar multiplications needed to compute a matrix-chain product, it does not directly show how to multiply the matrices. The table s[1…n-1, 2..n] gives us the information we need to do so. Each entry s[i,j] records a value of k such that an optimal parenthesization of $A_iA_{i+1}....A_j$splits the product between $A_k$and $A_{k+1.}$ The following recursive procedure prints an optimal parenthesization of $<A_{i,}$ $A_{i+1,}.....,A_j>$ given the s table computed by **Matrix-Chain-Order**and the indices i and j. The initial call **Print-Optimal-Parens**(s, 1, n) prints an optimal parenthesization of $<A_1, A_2,. .,A_i, . . ,A_n>$

Print-Optimal-Parens(s,i,j)

```
1  if i = = j
2    print "A_i"
3  else print "("
4    PRINT-OPTIMAL-PARENS(s, i, s[i, j])
5     PRINT-OPTIMAL-PARENS.s; s[i, j]+1, j )
6     print ")"
```

Example:

Matrix	A_1	A_2	A_3	A_4	A_5	A_6
Dimension	30 x 35	35 x 15	15 x 5	5 x 10	10 x 20	20 x 25

The m table uses only the main diagonal and upper triangle, and the s table uses only the upper triangle. The minimum number of scalar multiplications to multiply the 6 matrices is m[1,6]= 15,125. Of the darker entries, the pairs that have the same shading are taken together in line 10 when computing m[2,5]=min among below entries

1. m[2,2]+ m[3,5] + $p_1p_2p_5$ = 0 + 2500 + 35*15*20 =13,000,
2. m[2,3]+ m[4,5] + $p_1p_2p_5$ = 2625+ 1000 +35*5*20 =7125,
3. m[2,4] + m[5,5]+ $p_1p_4p_5$ = 4375 +0 + 35.10.20= 11,375

= 7125

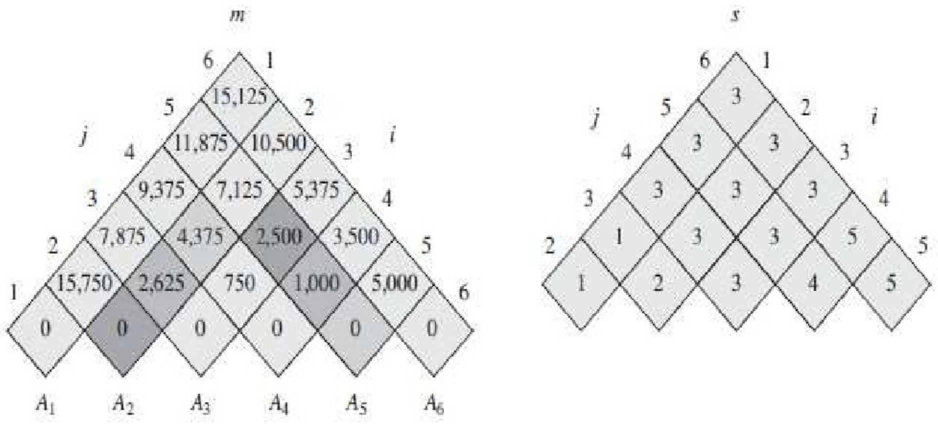


Fig. 5.1 The m and s tables computed by MATRIX-CHAIN-ORDER for n=6

The call **Print-Optimal-Parens**(s, 1, 6) prints the parenthesization ((A_1 (A_2A_3)) ((A_4 A_5) A_6)).

5.5 LONGEST COMMON SEQUENCE

A subsequence of a given sequence is just the given sequence with some elements left out. Given two sequences X and Y, we say that the sequence Z is a common sequence of X and Y if Z is a subsequence of both X and Y. In the longest common subsequence problem, we are given two sequences X = (x_1 x_2....x_m) and Y = (y_1 y_2 y_n) and wish to find a maximum length common subsequence of X and Y. LCS Problem can be solved using dynamic programming. A brute-force approach we find all the subsequences of X and check each subsequence to see if it is also a subsequence of Y, this approach requires exponential time making it impractical for the long sequence.

Given two strings x,y over some common alphabet, the problem is tofind a longest common sub-string (not necessarily consecutivecharacters) of both x and y.

Example: X= A B C B D A B
Y = B D C A B A

Then, BCBA is sub-string of both x and y and is of maximal lengthpossible among all common sub-strings. Note that it is not unique, BDAB is also a longest common sub-string. We are interested in findingone of maximal length. We define LCS(x,y) = a longest sequence ofcharacters that appears left to right in both strings.

This problem is useful in genetics. Given 2 DNA fragments, LCS givesinformation about what they have in common and what is the best way toline them up. Moreover, an equivalent problem is finding the minimumedit-distance (sequence of inserts and deletes) to transform string xinto string y.

Let X = < x_1, x_2, x_3,..., x_m > and Y = < y_1, y_2, y_3,..., y_n > be the sequences. To compute the length of an element the following algorithm is used.

In this procedure, table C[m, n] is computed in row major order and another table B[m,n] is computed to construct optimal solution.

Algorithm: LCS-Length-Table-Formulation (X, Y)

```
m := length(X)
n := length(Y)
for i = 1 to m do
  C[i, 0] := 0
for j = 1 to n do
  C[0, j] := 0
for i = 1 to m do
  for j = 1 to n do
    if $x_i = y_j$
      C[i, j] := C[i - 1, j - 1] + 1
      B[i, j] := 'D'
    else
      if $C[i -1, j] \geq C[i, j -1]$
        C[i, j] := C[i - 1, j] + 1
        B[i, j] := 'U'
      else
      C[i, j] := C[i, j - 1]
      B[i, j] := 'L'
return C and B
```

Algorithm: Print-LCS (B, X, i, j)

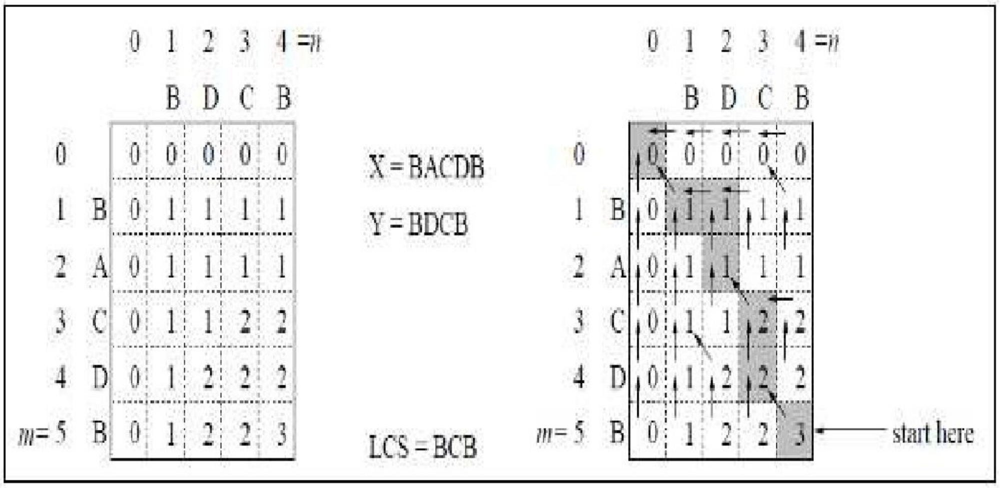
```
if i = 0 and j = 0
  return
if B[i, j] = 'D'
  Print-LCS(B, X, i-1, j-1)
  Print($x_i$)
else if B[i, j] = 'U'
  Print-LCS(B, X, i-1, j)
else
  Print-LCS(B, X, i, j-1)
```

This algorithm will print the longest common subsequence of X and Y.

Example: In this example, we have two strings ***X = BACDB*** and ***Y = BDCB*** to find the longest common subsequence.

Now, Following the algorithm LCS-Length-Table-Formulation (as stated above), we have calculated table C (shown on the left hand side) and table B (shown on the right hand side).

In table B, instead of 'D', 'L' and 'U', we are using the diagonal arrow, left arrow and up arrow, respectively. After generating table B, the LCS is determined by function LCS-Print. The result is BCB.



		0	1	2	3	4 = n
			B	D	C	B
0		0	0	0	0	0
1	B	0	1	1	1	1
2	A	0	1	1	1	1
3	C	0	1	1	2	2
4	D	0	1	2	2	2
m = 5	B	0	1	2	2	3

Fig. 5.2LCS Problem solving using dynamic Programming

(Source:https://www.tutorialspoint.com/design_and_analysis_of_algorithms/design_and_analysis_of_algorithms_dynamic_programming.htm)

CHECK YOUR PROGRESS

- o Write the need of dynamic programming for problem solving.
- o Compare Dynamic Programming with Greedy approach of problem solving.

o Give the features of long common sequence.

5.6 CAPITAL BUDGETING PROBLEM

Capital budgeting is a common area of finance but very useful for certain other areas like engineering and technology. In capital budgeting certain amount of money is to be invested carefully and intelligently so that best profit could be earned. Moreover, in the capital budgeting problem certain capital is available for investment and there are various options available in which it can be invested. But the money should be invested in such a way that we get maximum possible benefit. Now let us consider an example given below intended with this capital budgeting problem.

Example: *Nafisa has Rs.6000 to invest. She can invest in any of the three ventures, A, B and C, available to her. But, she must invest in units of Rs.1000. The potential return from investment in any one venture depends upon the amount invested, according to Table 5.1.*

Table 5.1 Investment and expected return in thousands.

Amount Invested	**Return from Venture**		
	A	**B**	**C**
0	0	0	0
1	0.5	1.5	1.2
2	1.0	2.0	2.4
3	3.0	2.2	2.5
4	3.1	2.3	2.6
5	3.2	2.4	2.7
6	3.3	2.5	2.8

Nafisa wants to invest Rs.6000 in such a way that the total return from the investment is maximized. We will formulate this problem as a dynamic programming problem and find the solution. V, is the initial state where she hasn't made any investment. There are two intermediate stages. In the first stage, Nafisa invests money in venture A. In the second stage, Nafisa invests in venture B. In the finalstage, Nafisa invests money in venture C, i.e. all of Rs.6000 is invested. We also denote by V1, the state where Nafisa invests Rs.i x 1000 in venture A. For example, the state V12 occurs if she invests 2 units(Rs.2000) in venture A. Similarly, V2i occurs if Nafisa invests a total of Rs.i x 1000 in ventures A and B.

For example, V22 will occur for any of the following situations:

1) Nafisa invests no money in venture A and 2 units in venture B.
2) Nafisa invests 1 unit each in venture A and venture B.
3) Nafisa invests 2 units in venture A and no money in venture B.

How are the distances calculated? The distance between any states is the money that Nafisa makes if the two states occur. For example, V11 follows V, if she invests one unit(Rs. 1000) in venture A. In this case, from second column in the second row of Table 5.1 on the facing page, it can be see that the expected return is 0.5 (Rs.500). So, d(V,. Vll) = 0.5. Similarly, suppose she invests one unit in venture 13 after investing one unit in venture A. From the third column in the second row of Table 3 on the preceding page, we see that she will get 1.5 units more in addition to 0.5 she earned for investing 1 unit in venture A. So d (Vll, V22) = 1.5. Ve is the final state where she has invested all the money.

Since there are too many options at each stage, a diagram for this problem will be too complicated. So, we will introduce a tabular method. As earlier, we have calculated f (V2,), 0 5 i < 6 first. What is the value of f (Vzo)? This is the state where Nafisa hasn't made any investment in ventures A and B. She will then invest the whole of Rs.6000 in venture C. From Table 5.1 on the facing page we see that she will get a return of 2.8 units(Rs.2800). So, f (V20) = 2.8. Similarly, in the state Val she would have invested a total of one unit, i.e. Rs.1000, amongst the ventures A and B. She will then invest the remaining Rs.5000 in venture C and will get in return 2.7 units(Rs.2700). We can similarly calculate the other values. Next, we calculate the maximum distances for the states in stage 1. We use the recurrence

$$f(V1i) = \max\{d(V1i, V2k) + f(V2k)\} \qquad for\ i \leq k \leq 6$$

5.7 MULTISTAGE GRAPHS

A multistage graph **G = (V, E)** is a directed graph where vertices are partitioned into **k** (where ***k*** > **1**) number of disjoint subsets ***S*** = $\{s_1, s_2, \ldots, s_k\}$ such that edge *(u, v)* is in E, then $u \in s_i$ and $v \in s_{1+1}$ for some subsets in the partition and $|s_1| = |s_k| = 1$.The vertex $s \in s_1$ is called the **source** and the vertex $t \in s_k$ is called **sink**.

G is usually assumed to be a weighted graph. In this graph, cost of an edge *(i, j)* is represented by *c(i, j)*. Hence, the cost of path from source ***s*** to sink ***t*** is the sum of costs of each edges in this path.

The multistage graph problem is finding the path with minimum cost from source ***s*** to sink ***t***. Dynamic Programming can be used to find the optimal solution for multistage network. This is because in multistage graph problem we obtain the minimum path at each current stage by considering the path length of each vertex obtained in earlier stage.Thus the sequence of decisions is taken by considering overlapping solutions. In dynamic programming, we may get any number of solutions for given problem. From all these solutions we seek for optimal solution, finally solution becomes the solution to given problem.

5.7.1 Using forward approach to find cost of the path

Forward approach can be used to find the cost of path; it can be given as

$$\text{Cost}(i,j) = \min \{ c(j, l) + \text{cost}(i+1, l)\}$$

Where, i-stage number, j-vertices available at particular stage, l- vertices away from the vertex and $l \in V_{i+1}$

```
Algorithm FGraph($G, k, n, p$)
// The input is a $k$-stage graph $G = (V, E)$ with $n$ vertices
// indexed in order of stages. $E$ is a set of edges and $c[i, j]$
// is the cost of $\langle i, j \rangle$. $p[1 : k]$ is a minimum-cost path.
{
    $cost[n] := 0.0$;
    for $j := n - 1$ to 1 step $-1$ do
    { // Compute $cost[j]$.
        Let $r$ be a vertex such that $\langle j, r \rangle$ is an edge
        of $G$ and $c[j, r] + cost[r]$ is minimum;
        $cost[j] := c[j, r] + cost[r]$;
        $d[j] := r$;
    }
    // Find a minimum-cost path.
    $p[1] := 1$; $p[k] := n$;
    for $j := 2$ to $k - 1$ do $p[j] := d[p[j - 1]]$;
}
```

Fig. 5.3 Multistage graph pseudocode corresponding to the forward approach

(Source:http://kraghavarao.weebly.com/uploads/2/2/5/5/22557070/dp-part-1multistagegraph.pdf

Example:*Consider the following example to understand the concept of multistage graph.*

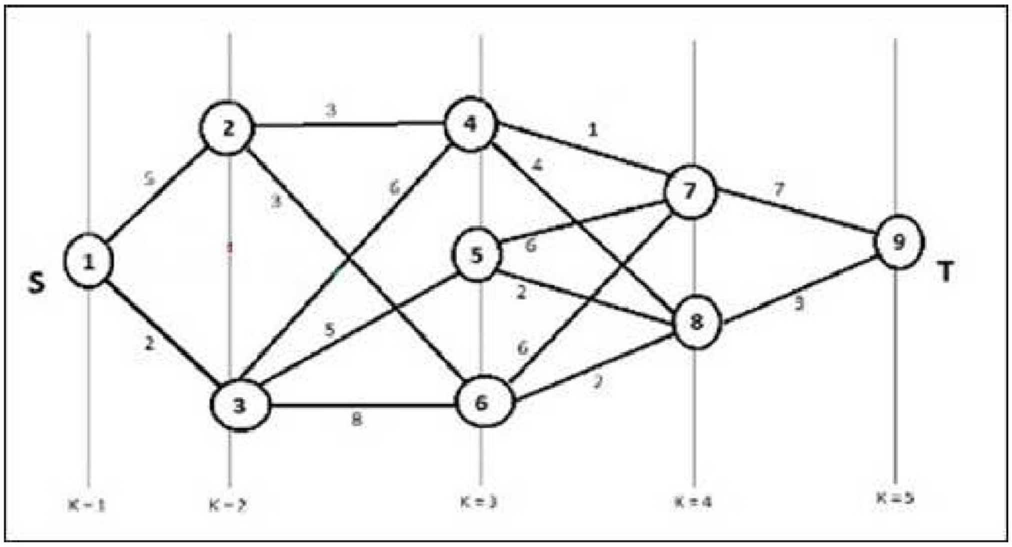


Fig. 5.4 Multistage graph

(Source:https://www.tutorialspoint.com/design_and_analysis_of_algorithms/design_and_analysis_of_algorithms_multistage_graph.htm)

According to the formula, we have to calculate the cost **(i, j)** using the following steps

Step-1: Cost (K-2, j)—In this step, three nodes (node 4, 5. 6) are selected as j. Hence, we have three options to choose the minimum cost at this step.

Cost(3, 4) = min {c(4, 7) + Cost(7, 9), c(4, 8) + Cost(8, 9)} = 7

Cost(3, 5) = min {c(5, 7) + Cost(7, 9), c(5, 8) + Cost(8, 9)} = 5

Cost(3, 6) = min {c(6, 7) + Cost(7, 9), c(6, 8) + Cost(8, 9)} = 5

Step-2: Cost (K-3, j) —Two nodes are selected as j because at stage k - 3 = 2 there are two nodes, 2 and 3. So, the value i = 2 and j = 2 and 3.

Cost(2, 2) = min {c(2, 4) + Cost(4, 8) + Cost(8, 9), c(2, 6) +Cost(6, 8) + Cost(8, 9)} = 8

Cost(2, 3) = {c(3, 4) + Cost(4, 8) + Cost(8, 9), c(3, 5) + Cost(5, 8)+ Cost(8, 9), c(3, 6) + Cost(6, 8) + Cost(8, 9)} = 10

Step-3: Cost (K-4, j)

Cost (1, 1) = {c(1, 2) + Cost(2, 6) + Cost(6, 8) + Cost(8, 9), c(1, 3) + Cost(3, 5) + Cost(5, 8) + Cost(8, 9))} = 12, c(1, 3) + Cost(3, 6) + Cost(6, 8 + Cost(8, 9))} = 13

Hence, the path having the minimum cost is **1→ 3→ 5→ 8→ 9**.

CHECK YOUR PROGRESS

- Define capital budgeting problem.
- Write the basic concept of multistage graph.
- Find an optimal parenthesization of a matrix-chain product whose sequence ofdimensions is <5, 10, 3, 12, 5, 50, 6>.

5.8 KNAPSACK PROBLEM

Knapsack is basically means bag. A bag of given capacity. We want to pack n items in a luggage.Given some items, pack the knapsack to get the maximum total value. Each item has some weight and some value. Total weight that we can carry must not more than some fixed number W. Following points are important in this regard

- Items are indivisible here.
- Fraction of any item can't be taken.
- An item, either should be taken completely or should be left completely.
- Knapsack problem basically uses dynamic programming approach.

Input:

- Maximum weight M and the number of packages n.
- Weight W[i] and their corresponding value V[i].

Output:

- Maximize value and corresponding weight in capacity.
- Which packages the thief will take away?

Problem Description:*A thief is robbing a store and can carry a maximum weight **W** into his knapsack. There are **n** items and weight of i^{th} item is w_i and the profit of selecting this item is p_i. What items should the thief take?*

Methodology Used: *Dynamic-Programming*

For the solution of the given problem, dynamic programming algorithm is used. Hence, let ***i*** be the highest-numbered item in an optimal solution **S** for **W** dollars. Then $S' = S - \{i\}$ is an optimal solution for $W - w_i$ dollars and the value to the solution ***S*** is V_i plus the value of the sub-problem.

We can express this fact in the following formula: define **c[i, w]** to be the solution for items **1,2, … , i** and the maximum weight **w**.

The algorithm takes the following inputs

- The maximum weight **W**
- The number of items **n**
- The two sequences **v = <v_1, v_2, …, v_n>** and **w = <w_1, w_2, …, w_n>**

Dynamic-0-1-knapsack (v, w, n, W):

```
for w = 0 to W do
  c[0, w] = 0
for i = 1 to n do
  c[i, 0] = 0
  for w = 1 to W do
    if $w_i \le w$ then
      if $v_i$ + c[i-1, w-$w_i$] then
        c[i, w] = $v_i$ + c[i-1, w-$w_i$]
      else c[i, w] = c[i-1, w]
    else
      c[i, w] = c[i-1, w]
```

The set of items to take can be deduced from the table, starting at **c[n, w]** and tracing backwards where the optimal values came from. If *c[i, w] = c[i-1, w]*, then item ***i*** is not part of the solution, and we continue tracing with **c[i-1, w]**. Otherwise, item ***i*** is part of the solution, and we continue tracing with **c[i-1, w-W]**.

Analysis: *This algorithm takes θ(n, w) times as table c has (n + 1).(w + 1) entries, where each entry requires θ(1) time to compute.*

Example: *Consider five items with respect weight and values*

I = {I_1, I_2, I_3, I_4, I_5}

w = {5, 10, 20, 30, 40}

v = {30, 20, 100, 90, 160}

The capacity of knapsack W = 60.

Items	**Wi**	**Vi**
I_1	5	30
I_2	10	20
I_3	20	100
I_4	30	90
I_5	40	160

Taking value per weight ratio $p_i = v_i/w_i$

Item	W_i	v_i	$p_i = v_i/w_i$
I_1	5	30	6.0
I_2	10	20	2.0
I_3	20	100	5.0
I_4	30	90	3.0
I_5	40	160	4.0

Now arrange the value of pi decreasing order.

Item	w_i	v_i	$p_i = v_i/w_i$
I_1	5	30	6.0
I_3	20	100	5.0
I_5	40	160	4.0
I_4	30	90	3.0
I_2	10	20	2.0

First, we choose item I_1 whose weight is 5, then choose item I_3 whose weight is 20. Now the total weight in knapsack is 5+20 = 25.

Now, the next item is I_5 and its weight is 40, but we want only 35. So, we choose fractional part i.e. (160/40)*35 = 140

Hence the maximum value is = 30+100+140 = 270

5.9 ALL PAIRS SHORTEST PATH PROBLEM

The *all-pairs shortest path problem* is the determination of the shortest graph distances between every pair of vertices in a given graph. The problem can be solved using. applications of Dijkstra's algorithm or all at once using the Floyd-Warshall algorithm. Given a directed, connected weighted graph G(V,E), for each edge $\langle u,v\rangle \in E$, a weight w(u,v) is associated with the edge. The *all-pairs of shortest paths problem* (APSP) is to find a shortest path from u to v for every pair of vertices u and v in V.

Representation of G:

The input is an n×n matrix $W = (w_{ij})$.

$$w(i,j)=\begin{cases} 0 \; if \; i=j \\ the \; weight \; of \; the \; directed \; edge <i,j> if \; i \neq j \; and <i,j> \in E \\ \infty \; if \; i \neq j \; and <i,j> \notin E \end{cases}$$

The dynamic-programming formulation can be used to solvethe all-pairs shortest-paths problem on a directed graph G = (V,E). The resultingalgorithm, known as the *Floyd-Warshall algorithm*, runs in $\Theta(V^3)$ time. The negative-weight edges may be present, but we assume that there are nonegative-weight cycles.

Floyd-Warshall's algorithm is based upon the observation that a path linking any two vertices u and v may have zero or more intermediate vertices. The algorithm begins by disallowing all intermediate vertices. In this case, the partial solution is simply the initial weights of the graph or infinity if there is no edge.

The algorithm proceeds by allowing an additional intermediate vertex at each step. For each introduction of a new intermediate vertex x, the shortest path between any pair of vertices u and v, x,u,v∈V, is the minimum of the previous best estimate of δ(u,v), or the combination of the paths from u→x and x→v.

$$\delta(u,v) \leftarrow \min(\delta(u,v), \delta(u,x)+\delta(x,v))$$

Let the directed graph be represented by a weighted matrix W.

```
FLOYD-WARSHALL (W)
1 n ← rows [W]
2 D(0) ← W
3 for k ← 1 to n
4 do for i ← 1 to n
5 do for j ← 1 to n
6 do dij (k) ← MIN ( dij (k-1) , dik (k-1) + dkj (k-1) )
7 return D(n)
```

Analysis: The time complexity of the algorithm above is O(n3).

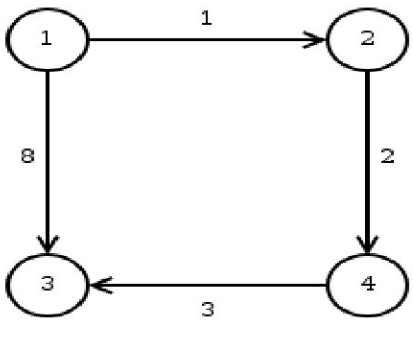


Fig. 5.5 Example Graph for Floyd-Warshall

(Source:http://users.cecs.anu.edu.au/~Alistair.Rendell/Teaching/apac_comp3600/module4/all_pairs_shortest_paths.xhtml#:~:text=The%20algorithm%20is%20based%20on,u%20to%20v%20is%200.)

***Example:** Let us now work through the steps of the FLOYD-WARSHALL algorithm when it is applied on the graph depicted in Fig. 5.5.*

W=(018∞∞0 ∞2 ∞∞0∞ ∞∞30)

1. Allowable intermediate vertices {}: D(0) =W
2. Allowable intermediate vertices {1}: D(1) no change
3. Allowable intermediate vertices {1,2}:

   D(2) =(018 3 ∞0∞2 ∞∞0∞ ∞∞30)

   Where 3= min {∞,1+2}, using paths 1→2 and 2→4.

4. Allowable intermediate vertices {1,2,3}: D(3) no change
5. Allowable intermediate vertices {1,2,3,4}:

   D(4) =(01 6 3 ∞0 5 2 ∞∞0∞ ∞∞30)

   Where 6= min {8,3+3}, using paths 1→4 and 4→3.

   Where 5= min {∞,2+3}, using paths 2→4 and 4→3.

5.10 TRAVELLING SALESMAN PROBLEM

The travelling salesman problem is related with the sales person and the set of cities in which he/she has to visit for some work. Now the salesman has to visit all the cities at least once starting from his hometown and has to return back to home after covering all the cities using shortest path hence shortening his length of the trip.

Problem Description: A traveler needs to visit all the cities from a list, where distances between all the cities are known and each city should be visited just once. What will be the shortest possible route that he visits each city exactly once and returns to the origin city?

Solution: Travelling salesman problem is the most notorious computational problem. We can use *brute-force* approach to evaluate every possible tour and select the best one. For **n** number of vertices in a graph, there are ***(n* - 1)!** number of possibilities.

Instead of *brute-force* using *dynamic programming approach*, the solution can be obtained in lesser time, though there is no polynomial time algorithm.

Let us consider a graph ***G = (V, E)***, where ***V*** is a set of cities and ***E*** is a set of weighted edges. An edge ***e(u, v)*** represents that vertices ***u*** and ***v*** are connected. Distance between vertex ***u*** and ***v*** is ***d(u, v)***, which should be non-negative.

Suppose we have started at city ***1*** and after visiting some cities now we are in city ***j***. Hence, this is a partial tour. We certainly need to know ***j***, since this will determine which cities are most convenient to visit next. We also need to know all the cities visited so far, so that we don't repeat any of them. Hence, this is an appropriate sub-problem.

For a subset of cities ***S* Є *{1, 2, 3, ..., n}*** that includes ***1***, and ***j* Є *S***, let ***C(S, j)*** be the length of the shortest path visiting each node in **S** exactly once, starting at ***1*** and ending at ***j***.

When |***S***| > 1, we define ***C(S, 1)*** = ∝ since the path cannot start and end at **1**.

Now, let express **C(S, j)** in terms of smaller sub-problems. We need to start at ***1*** and end at **j**. We should select the next city in such a way that

$C(S,j)=\min C(S-\{j\},i)+d(i,j)$ where $i \in S$ and $i \neq j$

Algorithm: *Traveling-Salesman-Problem*

```
C ({1}, 1) = 0
for s = 2 to n do
  for all subsets S Є {1, 2, 3, … , n} of size s and containing 1
    C (S, 1) = ∞
  for all j Є S and j ≠ 1
    C (S, j) = min {C (S – {j}, i) + d(i, j) for i Є S and i ≠ j}
Return minj C ({1, 2, 3, …, n}, j) + d(j, i)
```

Analysis: There are at the most $2^n.n$ sub-problems and each one takes linear time to solve. Therefore, the total running time is $O(2^n.n^2)$.

Example: In the following example, we will illustrate the steps to solve the travelling salesman problem.

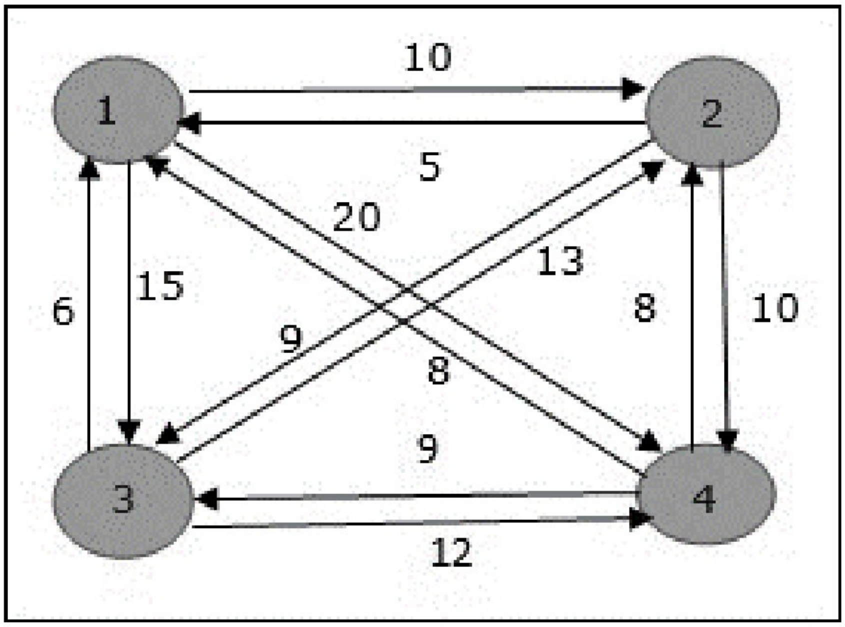


Fig. 5.5 Illustration of Travelling Salesman Problem using Dynamic Programming

(Source:https://www.tutorialspoint.com/design_and_analysis_of_algorithms/design_and_analysis_of_algorithms_travelling_salesman_problem.htm)

From the above graph, the following table is prepared.

	1	2	3	4
1	0	10	15	20
2	5	0	9	10
3	6	13	0	12
4	8	8	9	0

S = Φ

Cost(2,Φ,1)=d(2,1)=5

Cost(3,Φ,1)=d(3,1)=6

Cost(4,Φ,1)=d(4,1)=8

S = 1

Cost(i,s)=min{Cost(j,s–(j))+d[i,j]} for i Є S and i ≠ j

Cost(2,{3},1)=d[2,3]+Cost(3,Φ,1) =9+6 =15

Cost(2,{4},1)=d[2,4]+Cost(4,Φ,1) =10+8 =18

Cost(3,{2},1)=d[3,2]+Cost(2,Φ,1) =13+5 =18

Cost(3,{4},1)=d[3,4]+Cost(4,Φ,1) =12+8 =20

Cost(4,{3},1)=d[4,3]+Cost(3,Φ,1) =9+6 =15

Cost(4,{2},1)=d[4,2]+Cost(2,Φ,1) =8+5 =13

S = 2

$$\text{Cost}(2,\{3,4\},1)= \boldsymbol{min} \begin{cases} \mathbf{d[2,3] + Cost(3,\{4\},1) = 9 + 20 = 29} \\ \mathbf{d[2,4] + Cost(4,\{3\},1) = 10 + 15 = 25} \end{cases} \mathbf{= 25}$$

$$\text{Cost}(3,\{2,4\},1)= \boldsymbol{min} \begin{cases} \mathbf{d[3,2] + Cost(2,\{4\},1) = 13 + 18 = 31} \\ \mathbf{d[3,4] + Cost(4,\{2\},1) = 12 + 13 = 25} \end{cases} \mathbf{= 25}$$

$$\text{Cost}(4,\{2,3\},1)= \boldsymbol{min} \begin{cases} \mathbf{d[4,2] + Cost(2,\{3\},1) = 8 + 15 = 23} \\ \mathbf{d[4,3] + Cost(3,\{2\},1) = 9 + 18 = 27} \end{cases} \mathbf{= 23}$$

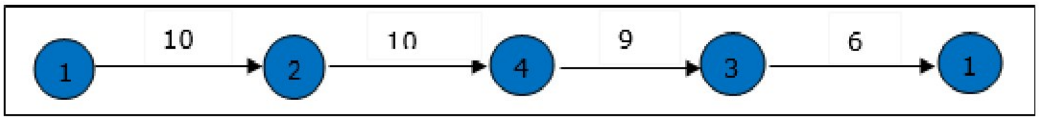
S = 3

$$\text{Cost}(1,\{2,3,4\},1)= \begin{cases} \mathbf{d[1,2] + Cost(2,\{3,4\},1) = 10 + 25 = 35} \\ \mathbf{d[1,3] + Cost(3,\{2,4\},1) = 15 + 25 = 40} \\ \mathbf{d[1,4] + Cost(4,\{2,3\},1) = 20 + 23 = 43} \end{cases} \mathbf{= 35}$$

The minimum cost path is 35.

Start from cost **{1, {2, 3, 4}, 1}**, we get the minimum value for **d [1, 2]**. When **s = 3**, select the path from 1 to 2 (cost is 10) then go backwards. When **s = 2**, we get the minimum value for **d [4, 2]**. Select the path from 2 to 4 (cost is 10) then go backwards.

When **s = 1**, we get the minimum value for **d [4, 3]**. Selecting path 4 to 3 (cost is 9), then we shall go to then go to **s = Φ** step. We get the minimum value for **d [3, 1]** (cost is 6).



CHECK YOUR PROGRESS

o Explore Knapsack problem with an example. Show, how dynamic programming can be used to solve this.

- Write Algorithm for matrix chain multiplication through dynamic programming. Give an example.
- Define all-pair shortest path in brief.
- What do you understand by travelling salesman problem?

5.11 SUMMARY

- Dynamic programming is a one of the good design techniques used for optimization problems. As far as Divide and Conquer algorithm is concerned, it solves the problems recursively after partitioning the problem into sub-problems. But Divide and Conquer algorithm is a lengthy process and it uses unnecessary things as it uses recursive approach which is costly and time taking (because it solves same sub-problem many times using recursion). There are three elements of dynamic programming —*Substructure, Table structure, and Bottom-up approach.*
- Basic idea of dynamic programming states — 1) Break the problems into sub-problems using bottom-up approach. 2) Solve smallest sub-problem first, and then use the solution to the sub-problems to solve the next smallest till solve the original problem. Here, time will be usually dominated by the number of possible sub-problems.
- The Dynamic Programming techniques can be applied to solve the problems for finding an optimal solution — Matrix Chain Multiplication, Longest Common Subsequence, Travelling Salesman Problem, Knapsack, World Series, and Capital budgeting problem.
- Matrix chain multiplication is a Method under Dynamic Programming in which previous output is taken as input for next. Here, Chain means one matrix's column is equal to the second matrix's row [always]. Moreover, matrix Multiplication operation is associative in nature rather commutative. By this, we mean that we have to follow the given matrix order for multiplication but we are free to parenthesize.
- Characteristics of longest common sequence are — *Optimal Substructure of an LCS, Recursive Solution, Computing the length of an LCS.*
- In capital budgeting certain amount of money is to be invested carefully and intelligently so that best profit could be earned. Moreover, in the capital budgeting problem certain capital is available for investment and there are various options available in which it can be invested. But the money should be invested in such a way that we get maximum possible benefit.

- Knapsack is basically means bag. A bag of given capacity. We want to pack n items in a luggage. Given some items, pack the knapsack to get the maximum total value. Each item has some weight and some value. Total weight that we can carry must not more than some fixed number.
- The travelling salesman problem is related with the sales person and the set of cities in which he/she has to visit for some work. Now the salesman has to visit all the cities at least once starting from his hometown and has to return back to home after covering all the cities using shortest path hence shortening his length of the trip.

5.12 TERMINAL QUESTIONS

1. How Dynamic Programming is different from Recursion and Memorization?
2. Describe the main characteristics of Dynamic Programming.
3. Explain the elements and components of Dynamic programming.
4. Differentiate Divide & Conquer and Dynamic Programming Algorithm.
5. Compare Greedy and dynamic programming algorithm.
6. Briefly explain the important applications of Dynamic programming.
7. Explain the process of matrix chain multiplication. Give an example to show the procedure.
8. Explain Longest Common Subsequences problem using dynamic programming approach.
9. Explain capital budgeting problem in detail and solve through dynamic programing.

Uttar Pradesh Rajarshi Tondon open University

Master of Computer Science

MCS-111 Design and Analysis of Algorithm

Block 3

Algorithm Design Strategies & Completeness

© UPRTOU , Prayagraj
ISBN :

Block-III Algorithm design strategies & Completeness

This is the third block on algorithm design strategies and completeness. In this block we mainly focused on algorithm design strategies and the completeness of the algorithm. This block has four units. Each unit has defined easily and covered all the topics.

So we will begin the first unit on introduction to **algorithm**. A graph is an abstract notation used to represent the connection between pairs of objects. A graph consists of Vertices (Interconnected objects in a graph are called vertices. Vertices are also known as nodes.) And Edges (Edges are the links that connect the vertices.)

In the first unit we will also describe representation of graphs, Breadth first search, depth first search and topological sort. In the first unit you will also learn about strongly connected component, flow networks, Ford-Fulkerson method.

Second unit begins with Backtracking. In this unit you will know all about the concept of **Backtracking**. Backtracking is a technique based on algorithm to solve problem. It uses recursive calling to find the solution by building a solution step by step increasing values with time. Backtracking algorithm is applied to some specific types of problems, Decision problem used to find a feasible solution of the problem. In this unit you will also learn about 8-queen problem, sum of subsets problem, graph coloring and Hamiltonian cycles.

In the third unit, we provide another important topic i.e. **Branch-And-Bound**. The branch and bound approach is based on the principle that the total set of feasible solutions can be partitioned into smaller subsets of solutions. These smaller subsets can then be evaluated systematically until the best solution is found. In this unit 2 important concept will also cover i.e. travelling salesperson problem, 15 puzzle problem.

In the fourth and last unit of this block we will cover **NP-Hard and NP-Complete problems**. A problem is NP-hard if all problems in NP are polynomial time reducible to it, even though it may not be in NP itself. In this unit you will also study about non deterministic algorithms, NP - Hard and NP Complete classes, satisfiability problem and reducibility.

As you will study the material, you will find that figures, tables are properly used and these will help to understand the concept. There are many sections in the units to easily understand the topic. Every unit has summary and review questions in the end of the unit which will help you to review yourself.

We hope you enjoy studying the material and once again wish you all the best for your success.

UNIT-6 Graph Algorithms

Structure

6.0 Introduction

This is the first unit of this block. In this unit you will learn about graph algorithm. This unit is divided in to ten sections. In the section 1.1, you will know about the basics of graph algorithm. Section 1.2 defines about the representation of graphs. Section 1.3 defines about Breadth first search. In the section 1.4 you will know about depth first search. Topological sort has been defining in the section 1.5. You will study about strongly connected component in the section 1.6. Flow networks explain in the next section i.e. section 1.7. In the section 1.8 you will know about Ford-Fulkerson method. Last two section defines summary and review questions.

Objective

After studying this unit, you should be able to define:

- ☐ Graph algorithm
- ☐ Representation of graphs
- ☐ Breadth first search
- ☐ Depth first search
- ☐ Topological sort
- ☐ Strongly connected component
- ☐ Flow networks
- ☐ Ford Fulkerson method

6.1 Graph algorithm- Introduction

A graph can be thought of as a data structure that is used to describe relationships between entities. An entity can be any item that has a distinctive and independent existence. It could either be an actual physical object or an abstract idea. For example, an entity can be a person, place or an organization about which data can be stored.

In the computing world, graphs have become ubiquitous owing to their ability to not only provide abstractions to real life but also demonstrate complicated relationships with ease. As such, a variety of practical problems can be represented as graphs. For example, a linked structure of websites can be viewed as a graph.

Every graph is a set of points referred to as vertices or nodes which are connected using lines called edges. The vertices represent entities in a graph. Edges, on the other hand, express relationships between entities. Hence, while nodes model entities, edges model relationships in a network graph. A graph G with a set of V vertices together with a set of E edges is represented as G= (V, E). Both vertices and edges can have additional attributes that are used to describe the entities and relationships. Figure 1.1 depicts a simple graph with five nodes and six edges.

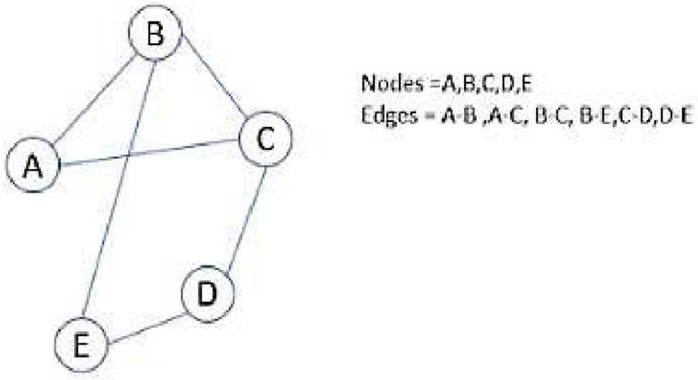


Figure 6.1: A simple Graph

In real-world applications of graphs, an edge might represent professional relationships that exist between people in LinkedIn or a personal relationship on a social media platform such as Facebook or Instagram.

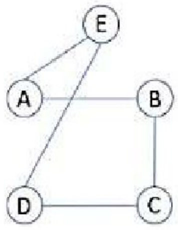


Figure 6.2: Undirected Graph

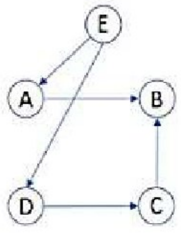


Figure 6.3: Directed Graph

Graphs can broadly be categorized into Undirected (Figure1.2) or Directed (Figure1.3). An undirected graph is directionless. This means that the edges have no directions. In other words, the relationship is mutual. For example, a Facebook or a LinkedIn connection. Contrarily, edges of directed graphs have directions associated with them. An asymmetric relationship between a boss and an employee or a teacher and a student can be represented as a directed graph in data structure. Graphs can also be weighted (Figure1.4) indicating real values associated with the edges. Depending upon the specific use of the graph, edge weights may represent quantities such as distance, cost, similarity etc.

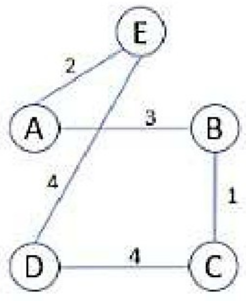


Figure 6.4: Weighted Graph

6.2 Representation of Graph

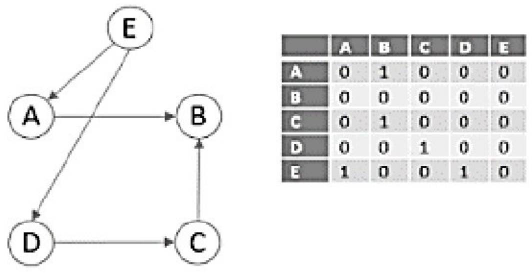
A graph can be represented using 3 data structures- **adjacency matrix, adjacency list and adjacency set.**

An adjacency matrix can be thought of as a table with rows and columns. The row labels and column labels represent the nodes of a graph. An adjacency matrix is a square matrix where the number of rows, columns and nodes are the same. Each cell of the matrix represents an edge or the relationship between two given nodes. For example, adjacency matrix Aij represents the number of links from i to j, given two nodes i and j.

	A	B	C	D	E
A	0	0	0	0	1
B	0	0	1	0	0
C	0	1	0	0	1
D	1	0	0	1	0
E	0	1	1	0	0

Table 6.1: Adjacency Matrix

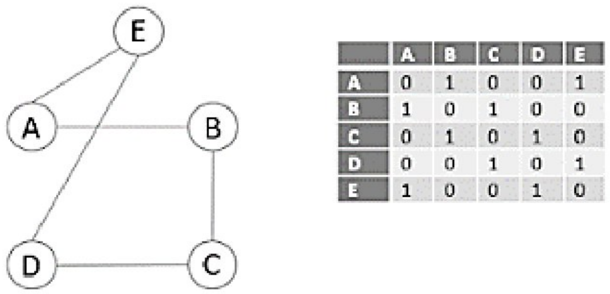
The adjacency matrix for a directed graph is shown in Figure1.5. Observe that it is a square matrix in which the number of rows, columns and nodes remain the same (5 in this case). Each row and column correspond to a node or a vertex of a graph. The cells within the matrix represent the connection that exists between nodes. Since, in the given directed graph, no node is connected to itself, all cells lying on the diagonal of the matrix are marked zero. For the rest of the cells, if there exists a directed edge from a given node to another, then the corresponding cell will be marked one else zero.



	A	B	C	D	E
A	0	1	0	0	0
B	0	0	0	0	0
C	0	1	0	0	0
D	0	0	1	0	0
E	1	0	0	1	0

Figure 6.5: Adjacency Matrix for a directed graph

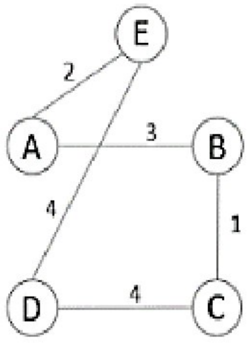
To understand how an undirected graph can be represented using an adjacency matrix, consider a small undirected graph with five vertices (Figure1.6). Here, A is connected to B, but B is connected to A as well. Hence, both the cells i.e., the one with source A Destination B and the other one with source B Destination A are marked one. This suffices the requirement of an undirected edge. Observe that the second entry is at a mirrored location across the main diagonal.

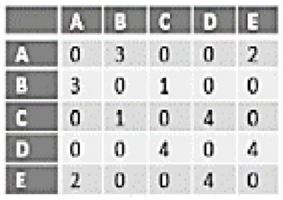


	A	B	C	D	E
A	0	1	0	0	1
B	1	0	1	0	0
C	0	1	0	1	0
D	0	0	1	0	1
E	1	0	0	1	0

Figure 6.6: Adjacency Matrix for a undirected graph

In case of a weighted graph, the cells are marked with edge weights instead of ones. In Figure1.7, the weight assigned to the edge connecting nodes B and D is 3. Hence, the corresponding cells in the adjacency matrix i.e. row B Column D and row D Column B are marked 3.





	A	B	C	D	E
A	0	3	0	0	2
B	3	0	1	0	0
C	0	1	0	4	0
D	0	0	4	0	4
E	2	0	0	4	0

Figure 6.7: Adjacency Matrix for a weighted graph

In adjacency list representation of a graph, every vertex is represented as a node object. The node may either contain data or a reference to a linked list. This linked list provides a list of all nodes that are adjacent to the current node. Consider a graph containing an edge connecting node A and node B. Then, the node A will be available in node B's linked list. Figure1.8 shows a sample graph of 5 nodes and its corresponding adjacency list.

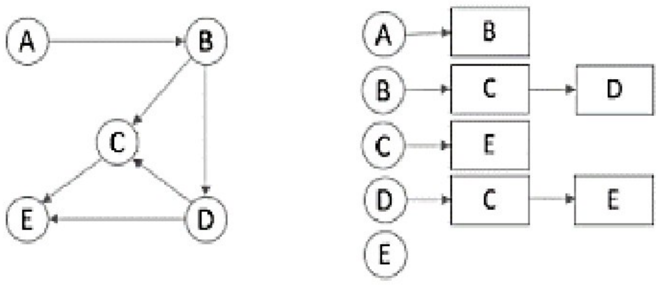


Figure 6.8: Adjacency list for directed graph

Note that the list corresponding to node E is empty while lists corresponding to nodes B and D have 2 entries each.

Similarly, adjacency lists for an undirected graph can also be constructed. Figure1.9 provides an example of an undirected graph along with its adjacency list for better understanding.

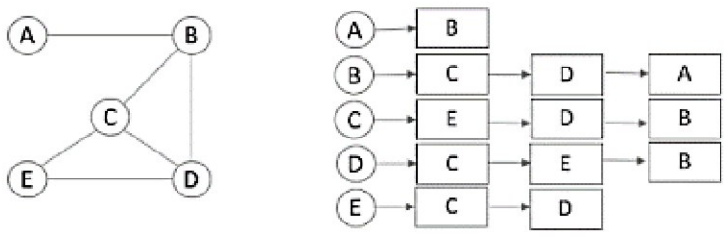


Figure 6.9: Adjacency list for undirected graph

Adjacency list enables faster search process in comparison to adjacency matrix. However, it is not the best representation of graphs especially when it comes to adding or removing nodes. For example, deleting a node would involve looking through all the adjacency lists to remove a particular node from all lists.

The adjacency set mitigates a few of the challenges posed by adjacency list. Adjacency set is quite similar to adjacency list except for the difference that instead of a linked list; a set of adjacent vertices is provided. Adjacency list and set are often used for sparse graphs with few connections between nodes. Contrarily, adjacency matrix works well for well-connected graphs comprising many nodes.

6.3 Breadth First Search

Breadth first search is a graph traversal algorithm that starts traversing the graph from root node and explores all the neighbouring nodes. Then, it selects the nearest node and explore all the unexplored nodes. The algorithm follows the same process for each of the nearest node until it finds the goal.

The algorithm of breadth first search is given below. The algorithm starts with examining the node A and all of its neighbours. In the next step, the neighbours of the nearest node of A are explored and process continues in the further steps. The algorithm explores all neighbours of all the nodes and ensures that each node is visited exactly once and no node is visited twice.

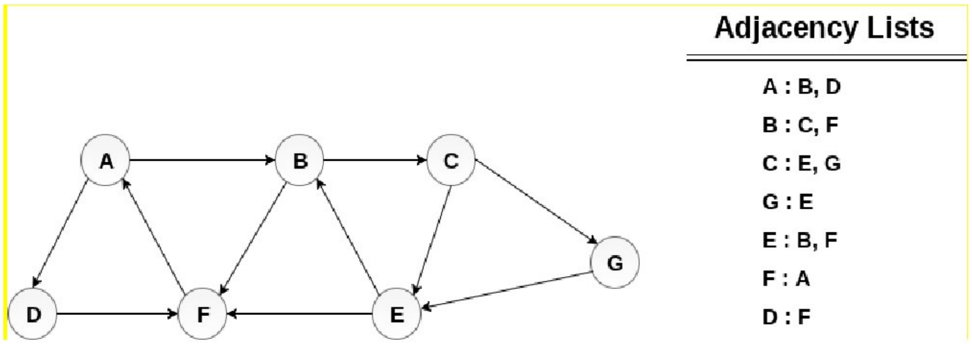
Algorithm

```
Step 1:    SET STATUS = 1 (ready state) for each node in G
Step 2:    Enqueue the starting node A and set its STATUS = 2
Step 3:    Repeat Steps 4 and 5 until QUEUE is empty
Step 4:    Dequeue a node N. Process it and set its STATUS = 3
Step 5:    Enqueue all the neighbours of
```

```
            N that are in the ready state
            (whose STATUS = 1) and set
            their STATUS = 2
            (waiting state)
            [END OF LOOP]
Step 6:     EXIT
```

Example

Consider the graph G shown in the following image, calculate the minimum path p from node A to node E. Given that each edge has a length of 1.



Solution:

Minimum Path P can be found by applying breadth first search algorithm that will begin at node A and will end at E. the algorithm uses two queues, namely QUEUE1 and QUEUE2. QUEUE1 holds all the nodes that are to be processed while QUEUE2 holds all the nodes that are processed and deleted from QUEUE1.

Let's start examining the graph from Node A.

1. Add A to QUEUE1 and NULL to QUEUE2.

   QUEUE1 = {A}

   QUEUE2 = {NULL}

2. Delete the Node A from QUEUE1 and insert all its neighbours. Insert Node A into QUEUE2

   QUEUE1 = {B, D}

   QUEUE2 = {A}

3. Delete the node B from QUEUE1 and insert all its neighbours. Insert node B into QUEUE2.

QUEUE1 = {D, C, F}

QUEUE2 = {A, B}

4. Delete the node D from QUEUE1 and insert all its neighbours. Since F is the only neighbour of it which has been inserted, we will not insert it again. Insert node D into QUEUE2.

QUEUE1 = {C, F}

QUEUE2 = {A, B, D}

5. Delete the node C from QUEUE1 and insert all its neighbours. Add node C to QUEUE2.

QUEUE1 = {F, E, G}

QUEUE2 = {A, B, D, C}

6. Remove F from QUEUE1 and add all its neighbours. Since all of its neighbours has already been added, we will not add them again. Add node F to QUEUE2.

QUEUE1 = {E, G}

QUEUE2 = {A, B, D, C, F}

7. Remove E from QUEUE1, all of E's neighbours has already been added to QUEUE1 therefore we will not add them again. All the nodes are visited and the target node i.e. E is encountered into QUEUE2.

QUEUE1 = {G}

QUEUE2 = {A, B, D, C, F, E}

Now, backtrack from E to A, using the nodes available in QUEUE2.

The minimum path will be A → B → C → E.

Check your progress

Q1. What are graph algorithms used for? Explain your answer
Q2. What is breadth first search in graph?
Q3. Compute the least cost path in a weighted digraph using BFS.
Q4. Find the path between given vertices in a directed graph.

6.4 Depth First Search

Depth first search (DFS) algorithm starts with the initial node of the graph G, and then goes to deeper and deeper until we find the goal node or the node which has no children. The algorithm, then backtracks from the dead end towards the most recent node that is yet to be completely unexplored.

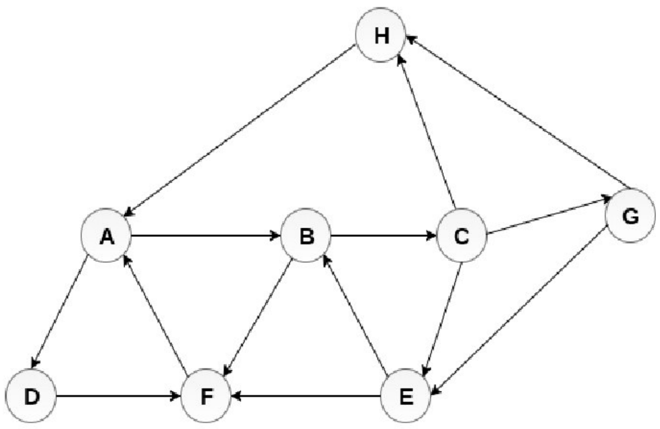
The data structure which is being used in DFS is stack. The process is similar to BFS algorithm. In DFS, the edges that leads to an unvisited node are called discovery edges while the edges that leads to an already visited node are called block edges.

Algorithm

```
Step 1:   SET STATUS = 1 (ready state) for each node in G
Step 2:   Push the starting node A on the stack and set its STATUS = 2
Step 3:   Repeat Steps 4 and 5 until STACK is empty
Step 4:   Pop the top node N. Process it and set its STATUS = 3
Step 5:   Push on the stack all the neighbours of N that are in the ready
      state (whose STATUS = 1) and set their STATUS = 2 (waiting state)
          [END OF LOOP]
Step 6:   EXIT
```

Example:

Consider the graph G along with its adjacency list, given in the figure below. Calculate the order to print all the nodes of the graph starting from node H, by using depth first search (DFS) algorithm.



Adjacency Lists

A :	B, D
B :	C, F
C :	E, G, H
G :	E, H
E :	B, F
F :	A
D :	F
H :	A

Solution:

Push H onto the stack

STACK: H

POP the top element of the stack i.e. H, print it and push all the neighbours of H onto the stack that are is ready state.

Print H

STACK: A

Pop the top element of the stack i.e. A, print it and push all the neighbours of A onto the stack that are in ready state.

Print A

Stack: B, D

Pop the top element of the stack i.e. D, print it and push all the neighbours of D onto the stack that are in ready state.

Print D

Stack: B, F

Pop the top element of the stack i.e. F, print it and push all the neighbours of F onto the stack that are in ready state.

Print F

Stack: B

Pop the top of the stack i.e. B and push all the neighbours

Print B

Stack: C

Pop the top of the stack i.e. C and push all the neighbours.

Print C

Stack: E, G

Pop the top of the stack i.e. G and push all its neighbours.

Print G

Stack: E

Pop the top of the stack i.e. E and push all its neighbours.

Print E

Stack:

Hence, the stack now becomes empty and all the nodes of the graph have been traversed.

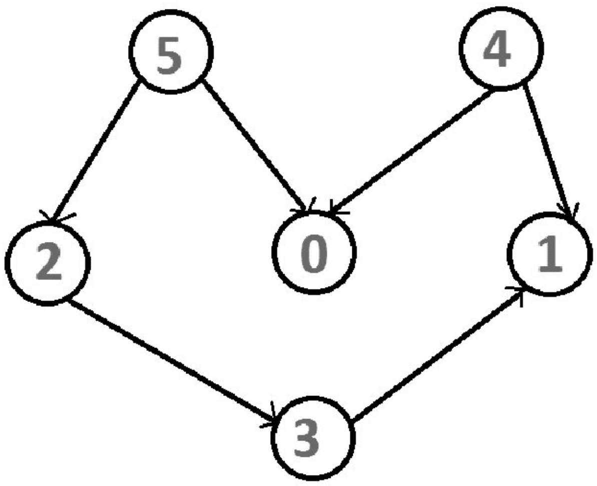
The printing sequence of the graph will be:

H → A → D → F → B → C → G → E

6.5 Topological sort

Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge u v, vertex u comes before v in the ordering. Topological Sorting for a graph is not possible if the graph is not a DAG.

For example, a topological sorting of the following graph is "5 4 2 3 1 0". There can be more than one topological sorting for a graph. For example, another topological sorting of the following graph is "4 5 2 3 1 0". The first vertex in topological sorting is always a vertex with in-degree as 0 (a vertex with no incoming edges).

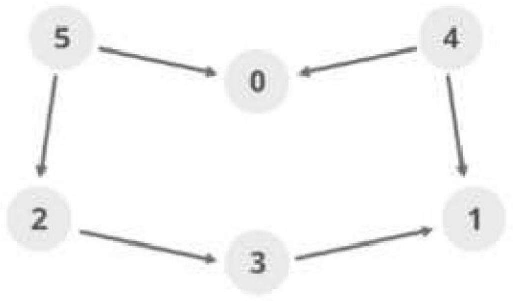


Topological Sorting vs Depth First Traversal (DFS):

In DFS, we print a vertex and then recursively call DFS for its adjacent vertices. In topological sorting, we need to print a vertex before its adjacent vertices. For example, in the given graph, the vertex '5' should be printed before vertex '0', but unlike DFS, the vertex '4' should also be printed before vertex '0'. So topological sorting is different from DFS. For example, a DFS of the shown graph is "5 2 3 1 0 4", but it is not a topological sorting.

Algorithm to find Topological Sorting:

We recommend to first see the implementation of DFS. We can modify DFS to find Topological Sorting of a graph. In DFS, we start from a vertex, we first print it and then recursively call DFS for its adjacent vertices. In topological sorting, we use a temporary stack. We don't print the vertex immediately, we first recursively call topological sorting for all its adjacent vertices, then push it to a stack. Finally, print contents of the stack. Note that a vertex is pushed to stack only when all of its adjacent vertices (and their adjacent vertices and so on) are already in the stack.



```
Adja cent list  (G)   0 →
                      1 →
                      2 → 3
                      3 → 1
                      4 → 0, 1
                      5 → 2, 0
```

	0	1	2	3	4	5
visited	false	false	false	false	false	false

Stack(empty)

Step 1: Topological Sort(0), visited[0] = true

↓

List is empty. No more recursion call.

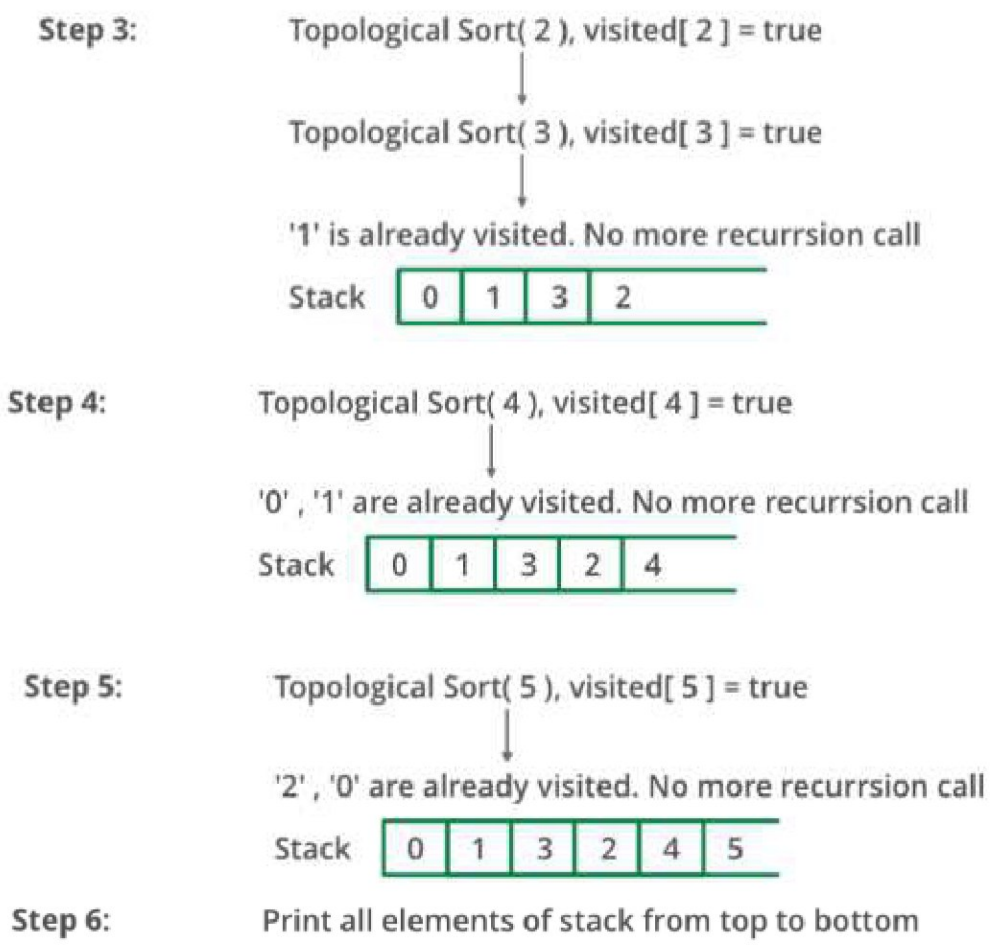
Stack	0

Step 2: Topological Sort(1), visited[1] = true

↓

List is empty. No more recursion call.

Stack	0	1


Step 3:
Topological Sort(2), visited[2] = true
Topological Sort(3), visited[3] = true
'1' is already visited. No more recurrsion call
Stack 0 1 3 2
Step 4:
Topological Sort(4), visited[4] = true
'0' , '1' are already visited. No more recurrsion call
Stack 0 1 3 2 4
Step 5:
Topological Sort(5), visited[5] = true
'2' , '0' are already visited. No more recurrsion call
Stack 0 1 3 2 4 5
Step 6:
Print all elements of stack from top to bottom

Following are the implementations of topological sorting.

```
// A C++ program to print topological
// sorting of a DAG
#include <iostream>
#include <list>
#include <stack>
using namespace std;

// Class to represent a graph
class Graph {
    // No. of vertices'
    int V;

    // Pointer to an array containing adjacency listsList
    list<int>* adj;

    // A function used by topologicalSort
    void topologicalSortUtil(int v, bool visited[],
                                            stack<int>& Stack);
public:
    // Constructor
    Graph(int V);

    // function to add an edge to graph
    void addEdge(int v, int w);

    // prints a Topological Sort of
    // the complete graph
    void topologicalSort();
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    // Add w to v's list.
    adj[v].push_back(w);
}

// A recursive function used by topologicalSort
void Graph::topologicalSortUtil(int v, bool visited[],
                                                            stack<int>&
Stack)
{
    // Mark the current node as visited.
    visited[v] = true;

    // Recur for all the vertices
    // adjacent to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
            if (!visited[*i])
                    topologicalSortUtil(*i, visited, Stack);

    // Push current vertex to stack
    // which stores result
    Stack.push(v);
}

// The function to do Topological Sort.
// It uses recursive topologicalSortUtil()
void Graph::topologicalSort()
{
    stack<int> Stack;

    // Mark all the vertices as not visited
    bool* visited = new bool[V];
    for (int i = 0; i < V; i++)
            visited[i] = false;

    // Call the recursive helper function
    // to store Topological
    // Sort starting from all
    // vertices one by one
    for (int i = 0; i < V; i++)
            if (visited[i] == false)
                    topologicalSortUtil(i, visited, Stack);

    // Print contents of stack
    while (Stack.empty() == false) {
            cout << Stack.top() << " ";
            Stack.pop();
    }
}

// Driver Code
int main()
{
    // Create a graph given in the above diagram
    Graph g(6);
    g.addEdge(5, 2);
    g.addEdge(5, 0);
    g.addEdge(4, 0);
    g.addEdge(4, 1);
    g.addEdge(2, 3);
    g.addEdge(3, 1);

    cout << "Following is a Topological Sort of the given "
                    "graph \n";

    // Function Call
    g.topologicalSort();

    return 0;
}
```

Output

```
Following is a Topological Sort of the given graph
5 4 2 3 1 0
```

Complexity Analysis:

- Time Complexity: O(V+E).
    - The above algorithm is simply DFS with an extra stack. So time complexity is the same as DFS which is.
- Auxiliary space: O(V).
    - The extra space is needed for the stack.

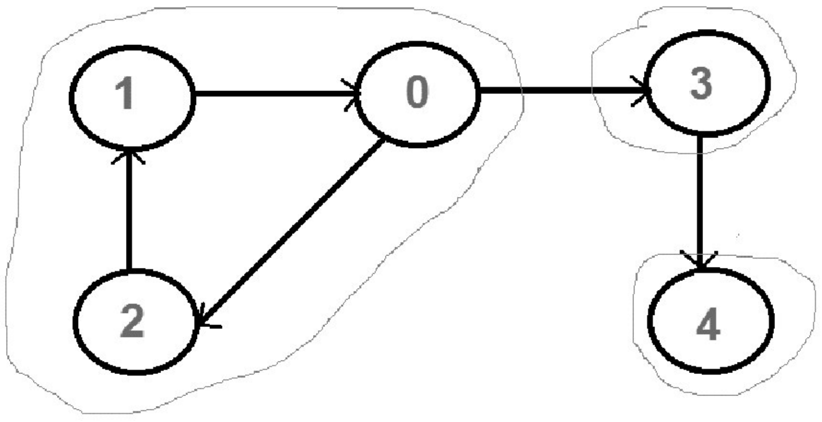
***Note:** Here, we can also use vector instead of the stack. If the vector is used, then print the elements in reverse order to get the topological sorting.*

Applications:

Topological Sorting is mainly used for scheduling jobs from the given dependencies among jobs. In computer science, applications of this type arise in instruction scheduling, ordering of formula cell evaluation when re-computing formula values in spreadsheets, logic synthesis, determining the order of compilation tasks to perform in make files, data serialization, and resolving symbol dependencies in linkers

6.6 Strongly connected components

A directed graph is strongly connected if there is a path between all pairs of vertices. A strongly connected component (SCC) of a directed graph is a maximal strongly connected subgraph. For example, there are 3 SCCs in the following graph.



We can find all strongly connected components in O(V+E) time using Kosaraju's algorithm. Following is detailed Kosaraju's algorithm.

1) Create an empty stack 'S' and do DFS traversal of a graph. In DFS traversal, after calling recursive DFS for adjacent vertices of a vertex, push the vertex to stack. In the above graph, if we start DFS from vertex 0, we get vertices in stack as 1, 2, 4, 3, 0.
2) Reverse directions of all arcs to obtain the transpose graph.
3) One by one pop a vertex from S while S is not empty. Let the popped vertex be 'v'. Take v as source and do DFS (call DFSUtil(v)). The DFS starting from v prints strongly connected component of v. In the above example, we process vertices in order 0, 3, 4, 2, 1 (One by one popped from stack).

How does this work?

The above algorithm is DFS based. It does DFS two times. DFS of a graph produces a single tree if all vertices are reachable from the DFS starting point. Otherwise DFS produces a forest. So DFS of a graph with only one SCC always produces a tree.

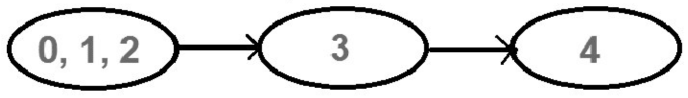
The important point to note is DFS may produce a tree or a forest when there are more than one SCCs depending upon the chosen starting point. For example, in the above diagram, if we start DFS from vertices 0 or 1 or 2, we get a tree as output. And if we start from 3 or 4, we get a forest. To find and print all SCCs, we would want to start DFS from vertex 4 (which is a sink vertex), then move to 3 which is sink in the remaining set (set excluding 4) and finally any of the remaining vertices (0, 1, 2). So how do we find this sequence of picking vertices as starting points of DFS? Unfortunately, there is no direct way for getting this sequence.

However, if we do a DFS of graph and store vertices according to their finish times, we make sure that the finish time of a vertex that connects to other SCCs (other than its own SCC), will always be greater than finish time of vertices in the other SCC (See this for proof). For example, in DFS of above example graph, finish time of 0 is always greater than 3 and 4 (irrespective of the sequence of vertices considered for DFS). And finish time of 3 is always greater than 4. DFS doesn't guarantee about other vertices, for example finish times of 1 and 2 may be smaller or greater than 3 and 4 depending upon the sequence of vertices considered for DFS.

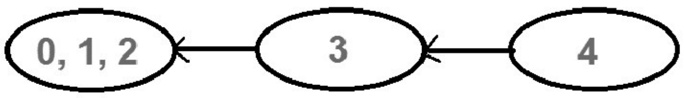
So to use this property, we do DFS traversal of complete graph and push every finished vertex to a stack. In stack, 3 always appears after 4, and 0 appear after both 3 and 4.

In the next step, we reverse the graph. Consider the graph of SCCs. In the reversed graph, the edges that connect two components are reversed. So the SCC {0, 1, 2} becomes sink and the SCC {4} becomes source. As discussed above, in stack, we always have 0 before 3 and 4. So if we do a DFS of the reversed graph using sequence of vertices in stack, we process vertices from sink to source (in reversed graph). That is what we wanted to achieve and that is all needed to print SCCs one by one.

Graph of SCCs



SCCs in reverse graph



Following is C++ implementation of Kosaraju's algorithm.

```
// C++ Implementation of Kosaraju's algorithm to print all SCCs
#include <iostream>
#include <list>
#include <stack>
using namespace std;

class Graph
```

```
{
        int V; // No. of vertices
        list<int> *adj; // An array of adjacency lists

        // Fills Stack with vertices (in increasing order of finishing
        // times). The top element of stack has the maximum finishing
        // time
        void fillOrder(int v, bool visited[], stack<int> &Stack);

        // A recursive function to print DFS starting from v
        void DFSUtil(int v, bool visited[]);
public:
        Graph(int V);
        void addEdge(int v, int w);

        // The main function that finds and prints strongly connected
        // components
        void printSCCs();

        // Function that returns reverse (or transpose) of this graph
        Graph getTranspose();
};

Graph::Graph(int V)
{
        this->V = V;
        adj = new list<int>[V];
}

// A recursive function to print DFS starting from v
```

```
void Graph::DFSUtil(int v, bool visited[])
{
        // Mark the current node as visited and print it
        visited[v] = true;
        cout << v << " ";

        // Recur for all the vertices adjacent to this vertex
        list<int>::iterator i;
        for (i = adj[v].begin(); i != adj[v].end(); ++i)
                if (!visited[*i])
                        DFSUtil(*i, visited);
}

Graph Graph::getTranspose()
{
        Graph g(V);
        for (int v = 0; v < V; v++)
        {
                // Recur for all the vertices adjacent to this vertex
                list<int>::iterator i;
                for(i = adj[v].begin(); i != adj[v].end(); ++i)
                {
                        g.adj[*i].push_back(v);
                }
        }
        return g;
}

void Graph::addEdge(int v, int w)
{
        adj[v].push_back(w); // Add w to v's list.
}

void Graph::fillOrder(int v, bool visited[], stack<int>&Stack)
{
        // Mark the current node as visited and print it
        visited[v] = true;

        // Recur for all the vertices adjacent to this vertex
        list<int>::iterator i;
        for(i = adj[v].begin(); i != adj[v].end(); ++i)
                if(!visited[*i])
                        fillOrder(*i, visited, Stack);

        // All vertices reachable from v are processed by now, push v
        Stack.push(v);
}

// The main function that finds and prints all strongly connected
// components
void Graph::printSCCs()
{
        stack<int> Stack;

        // Mark all the vertices as not visited (For first DFS)
        bool *visited = new bool[V];
        for(int i = 0; i < V; i++)
                visited[i] = false;

        // Fill vertices in stack according to their finishing times
        for(int i = 0; i < V; i++)
            if(visited[i] == false)
                fillOrder(i, visited, Stack);

        // Create a reversed graph
        Graph gr = getTranspose();

        // Mark all the vertices as not visited (For second DFS)
        for(int i = 0; i < V; i++)
            visited[i] = false;

        // Now process all vertices in order defined by Stack
        while (Stack.empty() == false)
        {
            // Pop a vertex from stack
            int v = Stack.top();
            Stack.pop();

            // Print Strongly connected component of the popped vertex
            if (visited[v] == false)
            {
                gr.DFSUtil(v, visited);
                cout << endl;
            }
        }
}

// Driver program to test above functions
int main()
{
    // Create a graph given in the above diagram
    Graph g(5);
    g.addEdge(1, 0);
    g.addEdge(0, 2);
    g.addEdge(2, 1);
    g.addEdge(0, 3);
    g.addEdge(3, 4);

    cout << "Following are strongly connected components in "
            "given graph \n";
    g.printSCCs();

    return 0;
}
```

Output:

```
Following are strongly connected components in given graph
0 1 2
3
4
```

Time Complexity: The above algorithm calls DFS, finds reverse of the graph and again calls DFS. DFS takes O(V+E) for a graph represented using adjacency list. Reversing a graph also takes O(V+E) time. For reversing the graph, we simple traverse all adjacency lists.

The above algorithm is asymptotically best algorithm, but there are other algorithms like Tarjan's algorithm and path-based algorithm which have same time complexity but find SCCs using single DFS. The Tarjan's algorithm is discussed in the following post.

Applications:

SCC algorithms can be used as a first step in many graph algorithms that work only on strongly connected graph.

In social networks, a group of people are generally strongly connected (For example, students of a class or any other common place). Many people in these groups generally like some common pages or play common games. The SCC algorithms can be used to find such groups and suggest the commonly liked pages or games to the people in the group who have not yet liked commonly liked a page or played a game.

6.7 Flow Networks

Flow Network is a directed graph that is used for modelling material Flow. There are two different vertices; one is a source which produces material at some steady rate, and another one is sink which consumes the content at the same constant speed. The flow of the material at any mark in the system is the rate at which the element moves.

Some real-life problems like the flow of liquids through pipes, the current through wires and delivery of goods can be modelled using flow networks.

Definition: A Flow Network is a directed graph G = (V, E) such that

1. For each edge (u, v) ∈ E, we associate a nonnegative weight capacity c (u, v) ≥ 0. If (u, v) ∉ E, we assume that c (u, v) = 0.
2. There are two distinguishing points, the source s, and the sink t;
3. For every vertex v ∈ V, there is a path from s to t containing v.

Let G = (V, E) be a flow network. Let s be the source of the network, and let t be the sink. A flow in G is a real-valued function f: V x V→R such that the following properties hold:

- ☐ Capacity Constraint: For all u, v ∈ V, we need f (u, v) ≤ c (u, v).
- ☐ Skew Symmetry: For all u, v ∈ V, we need f (u, v) = - f (u, v).
- ☐ Flow Conservation: For all u ∈ V-{s, t}, we need

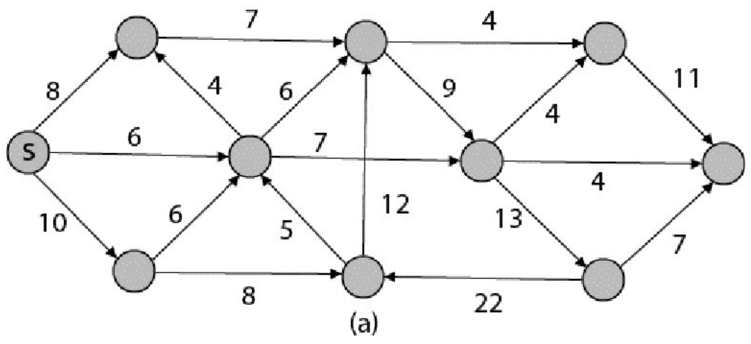
$$\boxed{\sum_{v \in V} f(u,v) = \sum_{u \in V} f(u,v) = 0}$$

The quantity f (u, v), which can be positive or negative, is known as the net flow from vertex u to vertex v. In the maximum-flow problem, we are given a flow network G with source s and sink t, and we wish to find a flow of maximum value from s to t.

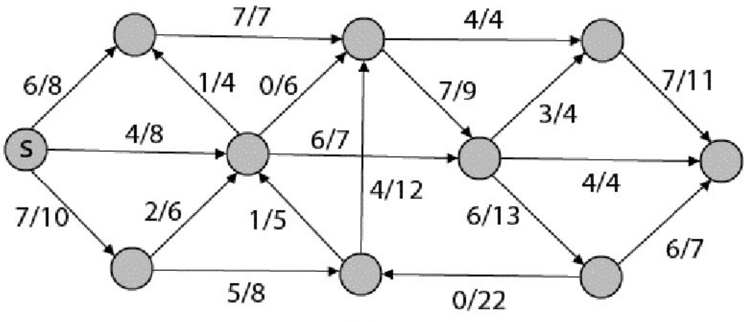
The three properties can be described as follows:

1. Capacity Constraint makes sure that the flow through each edge is not greater than the capacity.
2. Skew Symmetry means that the flow from u to v is the negative of the flow from v to u.
3. The flow-conservation property says that the total net flow out of a vertex other than the source or sink is 0. In other words, the amount of

flow into a v is the same as the amount of flow out of v for every vertex v ∈ V - {s, t}



(a)



(b)

The value of the flow is the net flow from the source,

$$|f| = \sum_{v \in V} f(s, v)$$

The positive net flow entering a vertex v is described by

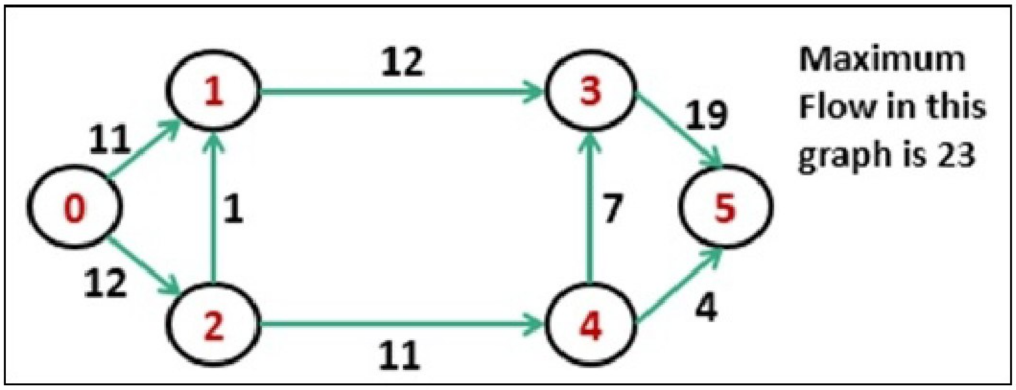
$$\sum_{\{u \in V : f(u,v) > 0\}} f(u, v)$$

The positive net flow leaving a vertex is described symmetrically. One interpretation of the Flow-Conservation Property is that the positive net flow entering a vertex other than the source or sink must equal the positive net flow leaving the vertex.

A flow f is said to be integer-valued if f (u, v) is an integer for all (u, v) ∈ E. Clearly, the value of the flow is an integer is an integer-valued flow.

6.8 Ford-Fulkerson Algorithm

The Ford-Fulkerson algorithm is used to detect maximum flow from start vertex to sink vertex in a given graph. In this graph, every edge has the capacity. Two vertices are provided named Source and Sink. The source vertex has all outward edge, no inward edge, and the sink will have all inward edge no outward edge.



There are some constraints:

- ☐ Flow on an edge doesn't exceed the given capacity of that graph.
- ☐ Incoming flow and outgoing flow will also equal for every edge, except the source and the sink.

Input and Output

Input:

The adjacency matrix:

0	10	0	10	0	0
0	0	4	2	8	0
0	0	0	0	0	10
0	0	0	0	9	0
0	0	6	0	0	10
0	0	0	0	0	0

Output:

Maximum flow is: 19

Algorithm

bfs(vert, start, sink)

Input: The vertices list, the start node, and the sink node.

Output − True when the sink is visited.

```
Begin
  initially mark all nodes as unvisited
  state of start as visited
  predecessor of start node is φ
  insert start into the queue qu
  while qu is not empty, do
    delete element from queue and set to vertex u
    for all vertices i, in the residual graph, do
      if u and i are connected, and i is unvisited, then
        add vertex i into the queue
        predecessor of i is u
        mark i as visited
    done
  done
  return true if state of sink vertex is visited
End
```

fordFulkerson(vert, source, sink)

Input: The vertices list, the source vertex, and the sink vertex.

Output − The maximum flow from start to sink.

```
Begin
  create a residual graph and copy given graph into it
  while bfs(vert, source, sink) is true, do
    pathFlow := ∞
    v := sink vertex
    while v ≠ start vertex, do
      u := predecessor of v
      pathFlow := minimum of pathFlow and residualGraph[u, v]
      v := predecessor of v
    done

    v := sink vertex
    while $v \neq$ start vertex, do
      u := predecessor of v
      residualGraph[u,v] := residualGraph[u,v] – pathFlow
      residualGraph[v,u] := residualGraph[v,u] – pathFlow
      v := predecessor of v
    done

    maFlow := maxFlow + pathFlow
  done
  return maxFlow
End
```

Implementation of fordFulkerson

```
#include<iostream>
#include<queue>
#define NODE 6
using namespace std;

typedef struct node {
  int val;
  int state; //status
  int pred; //predecessor
}node;
```

```
int minimum(int a, int b) {
  return (a<b)?a:b;
}

int resGraph[NODE][NODE];

/* int graph[NODE][NODE] = {
  {0, 16, 13, 0, 0, 0},
  {0, 0, 10, 12, 0, 0},
  {0, 4, 0, 0, 14, 0},
  {0, 0, 9, 0, 0, 20},
  {0, 0, 0, 7, 0, 4},
  {0, 0, 0, 0, 0, 0}
}; */

int graph[NODE][NODE] = {
  {0, 10, 0, 10, 0, 0},
  {0, 0, 4, 2, 8, 0},
  {0, 0, 0, 0, 0, 10},
  {0, 0, 0, 0, 9, 0},
  {0, 0, 6, 0, 0, 10},
  {0, 0, 0, 0, 0, 0}
};

int bfs(node *vert, node start, node sink) {
  node u;
  int i, j;
  queue<node> que;

  for(i = 0; i<NODE; i++) {
    vert[i].state = 0;   //not visited
  }

  vert[start.val].state = 1;  //visited
  vert[start.val].pred = -1;  //no parent node
  que.push(start);  //insert starting node

  while(!que.empty()) {
    //delete from queue and print
    u = que.front();
    que.pop();

    for(i = 0; i<NODE; i++) {
      if(resGraph[u.val][i] > 0 && vert[i].state == 0) {
        que.push(vert[i]);
        vert[i].pred = u.val;
        vert[i].state = 1;
      }
    }
  }
  return (vert[sink.val].state == 1);
}

int fordFulkerson(node *vert, node source, node sink) {
  int maxFlow = 0;
  int u, v;

  for(int i = 0; i<NODE; i++) {
    for(int j = 0; j<NODE; j++) {
      resGraph[i][j] = graph[i][j];   //initially residual graph is main graph
    }
  }

  while(bfs(vert, source, sink)) {    //find augmented path using bfs algorithm
    int pathFlow = 999;//as infinity
    for(v = sink.val; v != source.val; v=vert[v].pred) {
      u = vert[v].pred;
      pathFlow = minimum(pathFlow, resGraph[u][v]);
    }

    for(v = sink.val; v != source.val; v=vert[v].pred) {
      u = vert[v].pred;
      resGraph[u][v] -= pathFlow;   //update residual capacity of edges
      resGraph[v][u] += pathFlow;    //update residual capacity of reverse
edges
    }

    maxFlow += pathFlow;
  }
  return maxFlow;    //the overall max flow
}

int main() {
  node vertices[NODE];
  node source, sink;

  for(int i = 0; i<NODE; i++) {
    vertices[i].val = i;
  }

  source.val = 0;
  sink.val = 5;
  int maxFlow = fordFulkerson(vertices, source, sink);
  cout << "Maximum flow is: " << maxFlow << endl;
}
```

Output

```
Maximum flow is: 19
```

6.9 Summary

In this unit you have learnt about Graph Algorithms introduction, representation of graphs. You have than learnt about breadth first search and depth first search, topological sort and finally you have studied about strongly connected component, flow networks and Ford-Fulkerson method.

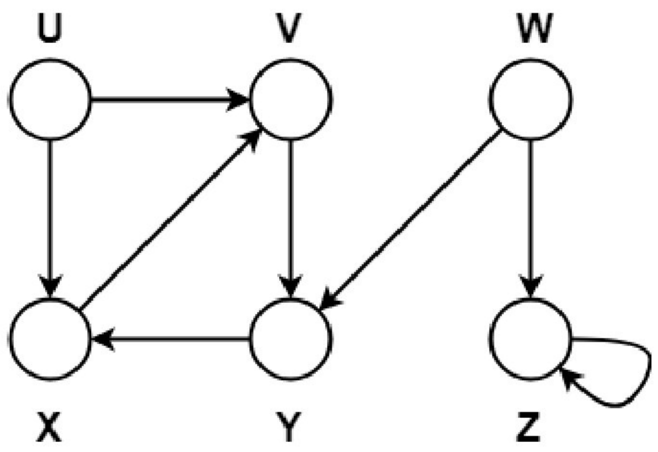
- ☐ A graph is an abstract notation used to represent the connection between pairs of objects. A graph consists of − Vertices − Interconnected objects in a graph are called vertices.
- ☐ Breadth-first search (BFS) is an algorithm that is used to graph data or searching tree or traversing structures. This algorithm selects a single node (initial or source point) in a graph and then visits all the nodes adjacent to the selected node. Remember, BFS accesses these nodes one by one.
- ☐ Depth-first search is used in topological sorting, scheduling problems, cycle detection in graphs, and solving puzzles with only one solution, such as a maze or a Sudoku puzzle. Other applications involve analyzing networks, for example, testing if a graph is bipartite.

6.10 Review Questions

Q1. What is Graph and its representation? Explain your answer

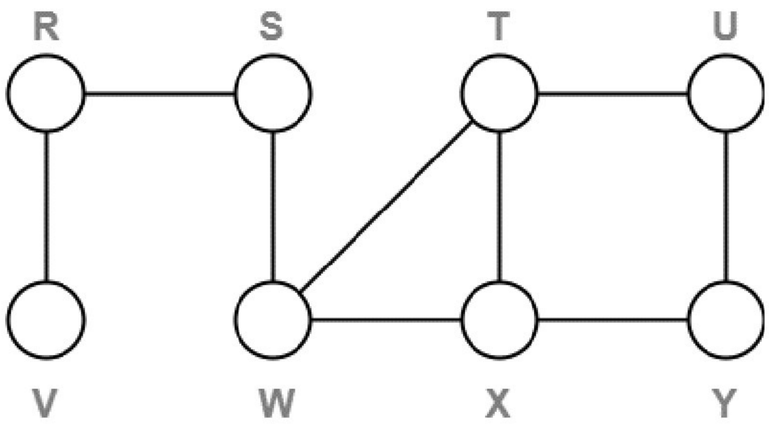
Q2. How do you do a breadth first search? Elaborate your answer.

Q3. Compute the DFS tree for the graph given below-



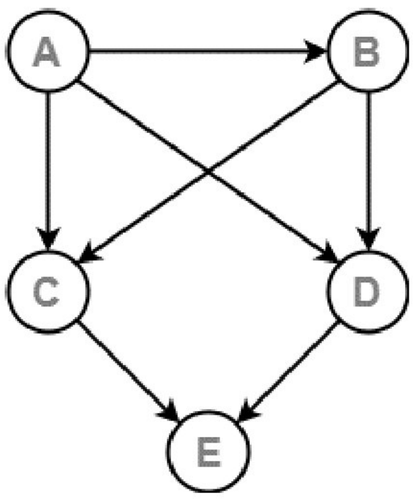
Also, show the discovery and finishing time for each vertex and classify the edges.

Q4. Traverse the following graph using Breadth First Search Technique-



Consider vertex S as the starting vertex.

Q5. Find the number of different topological orderings possible for the given graph-



UNIT-7 Backtracking

Structure

7.0 Introduction

This is the second unit of this block. In this unit you will learn about backtracking. There are eight sections in this unit. Section 2.1 explain about backtracking general method. Next section i.e. section 2.2 define applications of backtracking. Section 2.3 define 8-queen problem. In the section 2.4 you will learn about sum of subsets problem. Section 2.5 describe about graph colouring. In the section 2.6 you will know about Hamiltonian cycles. Last two section provide summary and review questions.

Objective

After studying this unit, you should be able to define:

- ☐ General method of Backtracking
- ☐ Applications of Backtracking
- ☐ 8-queen problem
- ☐ Sum of subsets problem
- ☐ graph coloring
- ☐ Hamiltonian cycles.

7.1 Backtracking-General method

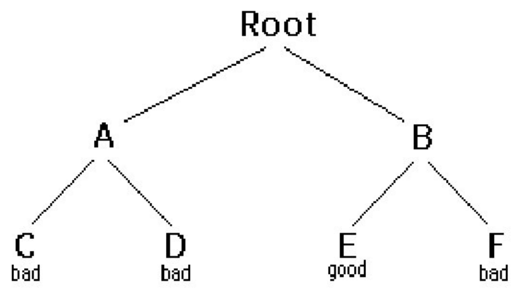
Backtracking is a form of recursion.

The usual scenario is that you are faced with a number of options, and you must choose one of these. After you make your choice you will get a new set of options; just what set of options you get depends on what choice you made. This procedure is repeated over and over until you reach a final state. If you made a good sequence of choices, your final state is a goal state; if you didn't, it isn't.

Conceptually, you start at the root of a tree; the tree probably has some good leaves and some bad leaves, though it may be that the leaves are all good or all bad. You want to get to a good leaf. At each node, beginning with the root, you choose one of its children to move to, and you keep this up until you get to a leaf.

Suppose you get to a bad leaf. You can backtrack to continue the search for a good leaf by revoking your most recent choice, and trying out the next option in that set of options. If you run out of options, revoke the choice that got you here, and try another choice at that node. If you end up at the root with no options left, there are no good leaves to be found.

This needs an example.



1. Starting at Root, your options are A and B. You choose A.
2. At A, your options are C and D. You choose C.
3. C is bad. Go back to A.
4. At A, you have already tried C, and it failed. Try D.
5. D is bad. Go back to A.
6. At A, you have no options left to try. Go back to Root.
7. At Root, you have already tried A. Try B.
8. At B, your options are E and F. Try E.
9. E is good. Congratulations!

In this example we drew a picture of a tree. The tree is an abstract model of the possible sequences of choices we could make. There is also a data structure called a tree, but usually we don't have a data structure to tell us what choices we have. (If we do have an actual tree data structure, backtracking on it is called depth-first tree searching.)

The backtracking algorithm.

Here is the algorithm (in pseudocode) for doing backtracking from a given node n:

```
boolean solve (Node n) {
   if n is a leaf node {
      if the leaf is a goal node, return true
      else return false
   } else {
      for each child c of n {
         if solve(c) succeeds, return true
      }
```

```
        return false
      }
  }
```

Notice that the algorithm is expressed as a Boolean function. This is essential to understanding the algorithm. If solve(n) is true, that means node n is part of a solution--that is, node n is one of the nodes on a path from the root to some goal node. We say that n is solvable. If solve (n) is false, then there is no path that includes n to any goal node.

How does this work?

- ☐ If any child of n is solvable, then n is solvable.
- ☐ If no child of n is solvable, then n is not solvable.

Hence, to decide whether any non-leaf node n is solvable (part of a path to a goal node), all you have to do is test whether any child of n is solvable. This is done recursively, on each child of n. In the above code, this is done by the lines

```
for each child c of n {
        if solve(c) succeeds, return true
      }
      return false
```

Eventually the recursion will "bottom" out at a leaf node. If the leaf node is a goal node, it is solvable; if the leaf node is not a goal node, it is not solvable. This is our base case. In the above code, this is done by the lines

```
if n is a leaf node {
   if the leaf is a goal node, return true
   else return false
}
```

The backtracking algorithm is simple but important. You should understand it thoroughly. Another way of stating it is as follows:

- ☐ To search a tree:

1. If the tree consists of a single leaf, test whether it is a goal node,
2. Otherwise, search the subtrees until you find one containing a goal node, or until you have searched them all unsuccessfully.

Non-recursive backtracking, using a stack

Backtracking is a rather typical recursive algorithm, and any recursive algorithm can be rewritten as a stack algorithm. In fact, that is how your recursive algorithms are translated into machine or assembly language.

```
boolean solve (Node n) {
  put node n on the stack;
  while the stack is not empty {
    if the node at the top of the stack is a leaf {
      if it is a goal node, return true
      else pop it off the stack
    }
    else {
      if the node at the top of the stack has untried children
        push the next untried child onto the stack
      else pop the node off the stack


  }
  return false
}
```

Starting from the root, the only nodes that can be pushed onto the stack are the children of the node currently on the top of the stack, and these are only pushed on one child at a time; hence, the nodes on the stack at all times describe a valid path in the tree. Nodes are removed from the stack only when it is known that they have no goal nodes among their descendants. Therefore, if the root node gets removed (making the stack empty), there must have been no goal nodes at all, and no solution to the problem.

When the stack algorithm terminates successfully, the nodes on the stack form (in reverse order) a path from the root to a goal node.

Similarly, when the recursive algorithm finds a goal node, the path information is embodied (in reverse order) in the sequence of recursive calls. Thus as the recursion unwinds, the path can be recovered one node at a time, by (for instance) printing the node at the current level, or storing it in an array.

Here is the recursive backtracking algorithm, modified slightly to print (in reverse order) the nodes along the successful path:

```
boolean solve (Node n) {
   if n is a leaf node {
      if the leaf is a goal node {
        print n
        return true
      }
      else return false
   } else {
      for each child c of n {
         if solve(c) succeeds {
            print n
            return true
         }
      }
      return false
   }
}
```

Keeping backtracking simple

All of these versions of the backtracking algorithm are pretty simple, but when applied to a real problem, they can get pretty cluttered up with details. Even determining whether the node is a leaf can be complex: for example, if the path represents a series of moves in a chess endgame problem, the leaves are the checkmate and stalemate solutions.

To keep the program clean, therefore, tests like this should be buried in methods. In a chess game, for example, you could test whether a node is a leaf by writing a game Over method (or you could even call it isLeaf). This method would encapsulate all the ugly details of figuring out whether any possible moves remain.

Notice that the backtracking algorithms require us to keep track, for each node on the current path, which of its children have been tried already (so we don't have to try them again). In the above code we made this look simple, by just saying for each child c of n. In reality, it may be difficult to figure out what the possible children are, and there may be no obvious way to step through them. In chess, for example, a node can represent one arrangement of pieces on a chessboard, and each child of that node can represent the arrangement after some piece has made a legal move. How do you find these children, and how do you keep track of which ones you've already examined?

The most straightforward way to keep track of which children of the node have been tried is as follows: Upon initial entry to the node (that is, when you first get there from above), make a list of all its children. As you try each child, take it off the list. When the list is empty, there are no remaining untried children, and you can return "failure." This is a simple approach, but it may require quite a lot of additional work.

There is an easier way to keep track of which children have been tried, if you can define an ordering on the children. If there is an ordering, and you know which child you just tried, you can determine which child to try next.

For example, you might be able to number the children 1 through n, and try them in numerical order. Then, if you have just tried child k, you know that you have already tried children 1 through k-1, and you have not yet tried children k+1 through n. Or, if you are trying to color a map with just four colors, you can always try red first, then yellow, then green, then blue. If child yellow fails, you know to try child green next. If you are searching a maze, you can try choices in the order left, straight, right (or perhaps north, east, south, west).

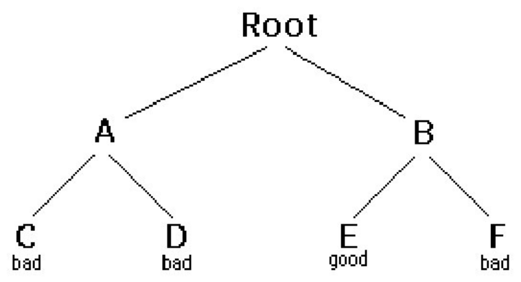
It isn't always easy to find a simple way to order the children of a node. In the chess game example, you might number your pieces (or perhaps the squares of the board) and try them in numerical order; but in addition each piece may also have several moves, and these must also be ordered.

You can probably find some way to order the children of a node. If the ordering scheme is simple enough, you should use it; but if it is too cumbersome, you are better off keeping a list of untried children.

Example: TreeSearch

For starters, let's do the simplest possible example of backtracking, which is searching an actual tree. We will also use the simplest kind of tree, a binary tree.

A binary tree is a data structure composed of nodes. One node is designated as the root node. Each node can reference (point to) zero, one, or two other nodes, which are called its children. The children are referred to as the left child and/or the right child. All nodes are reachable (by one or more steps) from the root node, and there are no cycles. For our purposes, although this is not part of the definition of a binary tree, we will say that a node might or might not be a goal node, and will contain its name. The first example in this paper (which we repeat here) shows a binary tree.



Here's a definition of the BinaryTree class:

```
public class BinaryTree {
   BinaryTree leftChild = null;
   BinaryTree rightChild = null;
   boolean isGoalNode = false;
   String name;

   BinaryTree(String name, BinaryTree left, BinaryTree right, boolean
isGoalNode) {
      this.name = name;
      leftChild = left;
```

```
        rightChild = right;
        this.isGoalNode = isGoalNode;
    }
}
```

Next we will create a TreeSearch class, and in it we will define a method makeTree() which constructs the above binary tree.

```
static BinaryTree makeTree() {
    BinaryTree root, a, b, c, d, e, f;
    c = new BinaryTree("C", null, null, false);
    d = new BinaryTree("D", null, null, false);
    e = new BinaryTree("E", null, null, true);
    f = new BinaryTree("F", null, null, false);
    a = new BinaryTree("A", c, d, false);
    b = new BinaryTree("B", e, f, false);
    root = new BinaryTree("Root", a, b, false);
    return root;
}
```

Here's a main program to create a binary tree and try to solve it:

```
public static void main (String args[]) {
    BinaryTree tree = makeTree();
    System.out.println(solvable(tree));
}
```

And finally, here's the recursive backtracking routine to "solve" the binary tree by finding a goal node.

```
static boolean solvable (BinaryTree node) {
/* 1 */ if (node == null) return false;
/* 2 */ if (node.isGoalNode) return true;
/* 3 */ if (solvable(node.leftChild)) return true;
/* 4 */ if (solvable(node.rightChild)) return true;
/* 5 */ return false;
}
```

Here's what the numbered lines are doing:

1. If we are given a null node, it's not solvable. This statement is so that we can call this method with the children of a node, without first checking whether those children actually exist.
2. If the node we are given is a goal node, return success.
3. See if the left child of node is solvable, and if so, conclude that node is solvable. We will only get to this line if node is non-null and is not a goal node, says to
4. Do the same thing for the right child.
5. Since neither child of node is solvable, node itself is not solvable.

This program runs correctly and produces the unenlightening result true.

Each time we ask for another node, we have to check if it is null. In the above we put that check as the first thing in solvable. An alternative would be to check first whether each child exists, and recur only if they do. Here's that alternative version:

```
static boolean solvable (BinaryTree node) {
   if (node.isGoalNode) return true;
   if (node.leftChild != null && solvable(node.leftChild)) return true;
   if (node.rightChild != null && solvable(node.rightChild)) return true;
   return false;
}
```

What are the children?

One of the things that simplifies the above binary tree search is that, at each choice point, you can ignore all the previous choices. Previous choices don't give you any information about what you should do next; as far as you know, both the left and the right child are possible solutions. In many problems, however, you may be able to eliminate children immediately, without recursion.

Consider, for example, the problem of four-coloring a map. It is a theorem of mathematics that any map on a plane, no matter how convoluted the countries are, can be colored with at most four colors, so that no two countries that share a border are the same color.

To color a map, you choose a color for the first country, then a color for the second country, and so on, until all countries are colored. There are two ways to do this:

- ☐ Method 1. Try each of the four possible colors, and recur. When you run out of countries, check whether you are at a goal node.
- ☐ Method 2. Try only those colors that have not already been used for an adjacent country, and recur. If and when you run out of countries, you have successfully colored the map.

Let's apply each of these two methods to the problem of coloring a checkerboard. This should be easily solvable; after all, a checkerboard only needs two colors.

In both methods, the colors are represented by integers, from RED=1 to BLUE=4. We define the following helper methods. The helper method code isn't displayed here because it's not important for understanding the method that does the backtracking.

boolean mapIsOK()

Used by method 1 to check (at a leaf node) whether the entire map is colored correctly.

boolean okToColor(int row, int column, int color)

Used by method 2 to check, at every node, whether there is an adjacent node already colored with the given color.

int[] nextRowAndColumn(int row, int column)

Used by both methods to find the next "country" (actually, the row and column of the next square on the checkerboard).

Here's the code for method 1:

```
boolean explore1(int row, int column, int color) {
    if (row >= NUM_ROWS) return mapIsOK();
    map[row][column] = color;
    for (int nextColor = RED; nextColor <= BLUE; nextColor++) {
        int[] next = nextRowAndColumn(row, column);
        if (explore1(next [0], next [1], nextColor)) return true;
    }
    return false;
}
```

And here's the code for method 2:

```
boolean explore2(int row, int column, int color) {
    if (row >= NUM_ROWS) return true;
    if (okToColor(row, column, color)) {
        map[row][column] = color;
        for (int nextColor = RED; nextColor <= BLUE; nextColor++) {
            int[] next = nextRowAndColumn(row, column);
            if (explore2(next [0], next [1], nextColor)) return true;
        }
    }
    return false;
}
```

7.2Applications-Backtracking

1. To find all Hamiltonian Paths present in a graph.
2. To solve the N Queen problem.
3. Maze solving problem.
4. The Knight's tour problem.

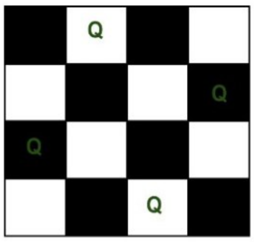
7.2.1 Hamiltonian path problem:

In the mathematical field of Graph Theory, the Hamiltonian path problem and the Hamiltonian cycle problem are problems of determining whether a Hamiltonian path (a path in an undirected or directed graph that visits each vertex exactly once) or a Hamiltonian cycle exists in a given graph (whether directed or undirected). Both problems are NP-complete.

The Hamiltonian cycle problem is a special case of the travelling salesman problem, obtained by setting the distance between two cities to one if they are adjacent and two otherwise, and verifying that the total distance travelled is equal to n (if so, the route is a Hamiltonian circuit; if there is no Hamiltonian circuit then the shortest route will be longer).

7.2.2 N Queen Problem

The N Queen is the problem of placing N chess queens on an N×N chessboard so that no two queens attack each other. For example, following is a solution for 4 Queen problem.



The expected output is a binary matrix which has 1s for the blocks where queens are placed. For example, following is the output matrix for above 4 queen solution.

```
            { 0, 1, 0, 0}
    { 0, 0, 0, 1}
    { 1, 0, 0, 0}
    { 0, 0, 1, 0}
```

7.2.3 Maze Solving Problem

There are a number of different maze solving algorithms, that is, automated methods for the solving of mazes. The random mouse, wall follower, Pledge, and Trémaux's algorithms are designed to be used inside the maze by a traveller with no prior knowledge of the maze, whereas the dead-end filling and shortest path algorithms are designed to be used by a person or computer program that can see the whole maze at once.

Mazes containing no loops are known as "simply connected", or "perfect" mazes, and are equivalent to a tree in graph theory. Thus many maze solving algorithms are closely related to graph theory. Intuitively, if one pulled and stretched out the paths in the maze in the proper way, the result could be made to resemble a tree.

7.2.4 The Knight's tour problem.

A knight's tour is a sequence of moves of a knight on a chessboard such that the knight visits every square exactly once. If the knight ends on a square that is one knight's move from the beginning square (so that it could tour the board again immediately, following the same path), the tour is closed; otherwise, it is open.

The knight's tour problem is the mathematical problem of finding a knight's tour. Creating a program to find a knight's tour is a common problem given to computer science students. Variations of the knight's tour problem involve chessboards of different sizes than the usual 8 × 8, as well as irregular (non-rectangular) boards.

0	59	38	33	30	17	8	63
37	34	31	60	9	62	29	16
58	1	36	39	32	27	18	7
35	48	41	26	61	10	15	28
42	57	2	49	40	23	6	19
47	50	45	54	25	20	11	14
56	43	52	3	22	13	24	5
51	46	55	44	53	4	21	12

Figure: Path-foll0wed-By-Knight-to-cover-all-the-cells

Check your progress

Q1. What do you understand by backtracking algorithm? Explain with example.

Q2. Define all the applications related to backtracking.

7.3 8-queen problem

You are given an 8x8 chessboard, find a way to place 8 queens such that no queen can attack any other queen on the chessboard. A queen can only be attacked if it lies on the same row, or same column, or the same diagonal of any other queen. Print all the possible configurations.

To solve this problem, we will make use of the Backtracking algorithm. The backtracking algorithm, in general checks all possible configurations and test whether the required result is obtained or not. For the given problem, we will explore all possible positions the queens can be relatively placed at. The solution will be correct when the number of placed queens = 8.

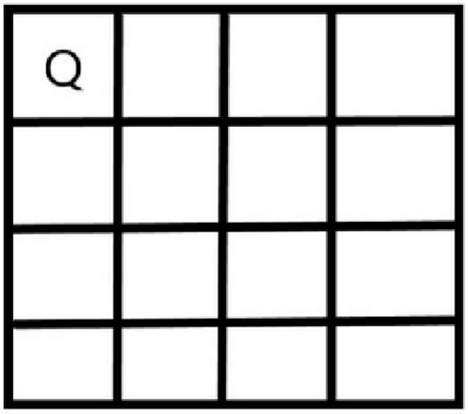
The time complexity of this approach is O(N!).

Input Format - the number 8, which does not need to be read, but we will take an input number for the sake of generalization of the algorithm to an NxN chessboard.

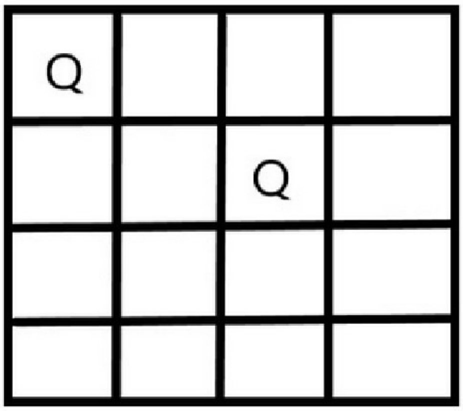
Output Format - all matrices that constitute the possible solutions will contain the numbers 0(for empty cell) and 1(for a cell where queen is placed). Hence, the output is a set of binary matrices.

Visualisation from a 4x4 chessboard solution:

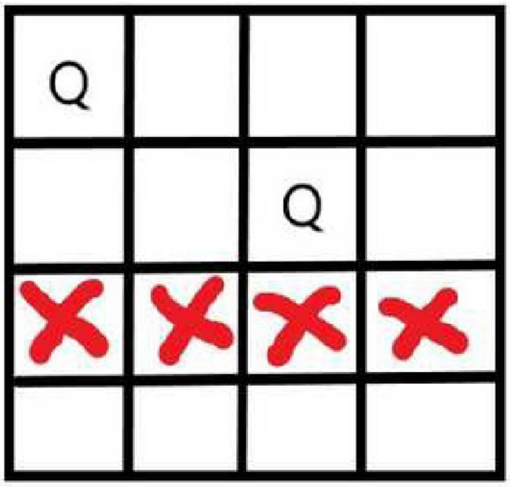
In this configuration, we place 2 queens in the first iteration and see that checking by placing further queens is not required as we will not get a solution in this path. Note that in this configuration, all places in the third rows can be attacked.



Let us first consider an empty chessboard and start by placing the first queen on cell chessboard [0][0].

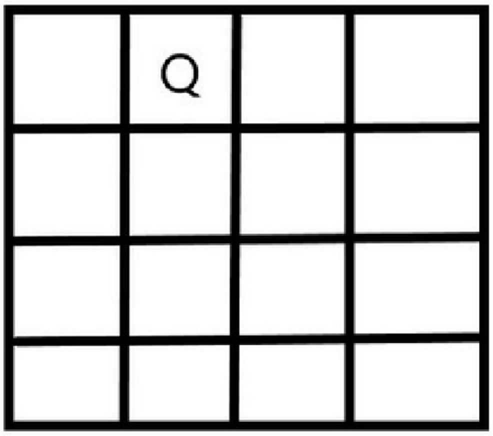


Chessboard [1][2] is the only possible position where the second queen can be placed.

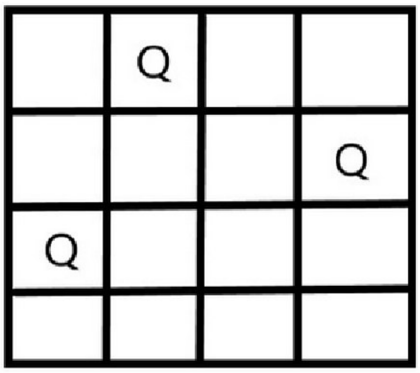


No queen can be placed further as queen1 is in column1, queen2 is diagonally opposite to column2 and 3; and column3 has queen2 in it. Since number of queen is not 4 this is an infeasible solution and will not be printed.

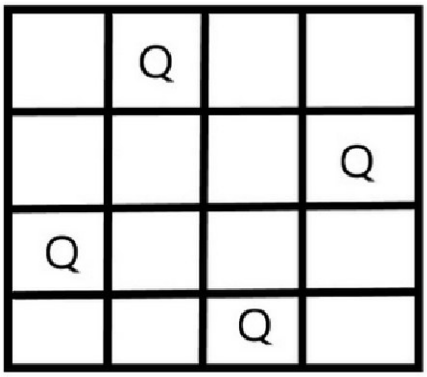
As the above combination was not possible, we will go back and go for the next iteration. This means we will change the position of the second queen.



The next iteration of our algorithm will begin with the second column and start placing the queens again.



Queens2 and 3 are easily placed onto the chessboard without the creating the possibility of attacking one another.



The fourth queen will also be placed accordingly. Since the number of queens =4, this solution will be printed.

nqueen2

In this, we found a solution.

Now let's take a look at the backtracking algorithm and see how it works:

The idea is to place the queens one after the other in columns, and check if previously placed queens cannot attack the current queen we're about to place.

If we find such a row, we return true and put the row and column as part of the solution matrix. If such a column does not exist, we return false and backtrack*

Pseudocode

```
START
1. begin from the leftmost column
2. if all the queens are placed,
   return true/ print configuration
3. check for all rows in the current column
   a) if queen placed safely, mark row and column; and
      recursively check if we approach in the current
      configuration, do we obtain a solution or not
   b) if placing yields, a solution, return true
   c) if placing does not yield a solution, unmark and
```

```
    try other rows
4. if all rows tried and solution not obtained, return
   false and backtrack
END
```

Implementation

Implementation of the above backtracking algorithm:

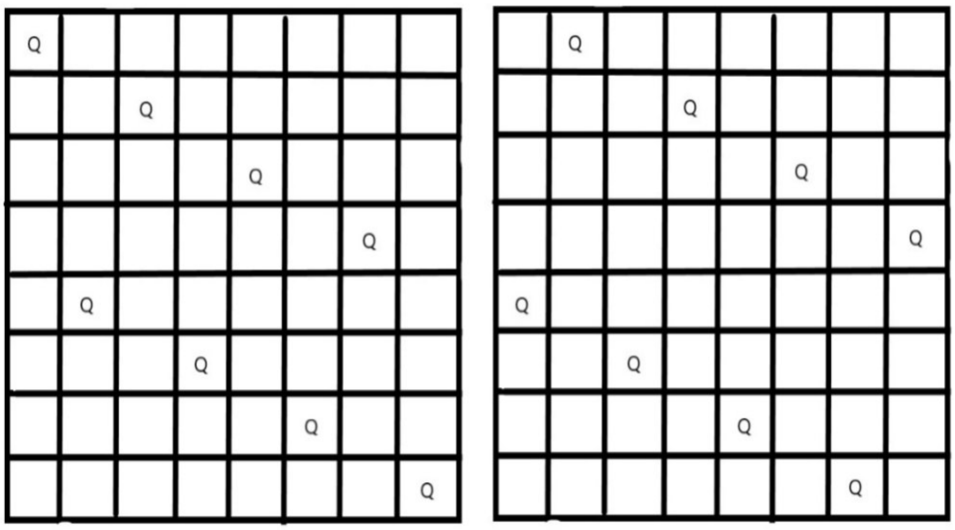
```
#include <bits/stdc++.h>

using namespace std;

int board[8][8]; // you can pick any matrix size you want

bool isPossible(int n,int row,int col){  // check whether
                    // placing queen possible or not

// Same Column
 for(int i=row-1;i>=0;i--){
   if(board[i][col] == 1){
    return false;
   }
 }

//Upper Left Diagonal
 for(int i=row-1,j=col-1;i>=0 && j>=0 ; i--,j--){
   if(board[i][j] ==1){
    return false;
   }
 }

 // Upper Right Diagonal
 for(int i=row-1,j=col+1;i>=0 && j<n ; i--,j++){
   if(board[i][j] == 1){
     return false;
   }
 }


 return true;
}
void nQueenHelper(int n,int row){
 if(row==n){
   // We have reached some solution.
   // Print the board matrix
   // return


   for(int i=0;i<n;i++){
     for(int j=0;j<n;j++){
       cout << board[i][j] << " ";
     }
   }
   cout<<endl;
   return;


 }


 // Place at all possible positions and move to smaller problem
 for(int j=0;j<n;j++){


   if(isPossible(n,row,j)){  // if no attack, proceed
     board[row][j] = 1;      // mark row, column with 1
     nQueenHelper(n,row+1);  // call function to continue
                             // further
   }
```

```
    board[row][j] = 0;     // unmark to backtrack
  }
  return;

}
void placeNQueens(int n){

  memset(board,0,8*8*sizeof(int)); // allocate 8*8 memory
                                   // and initialize all
                                   // cells with zeroes

  nQueenHelper(n,0);     // call the backtracking function
                    // and print solutions
}

int main(){

    int n;
    cin>>n; // could use a default 8 as well

    placeNQueens(n);
    return 0;
}
```

Output (for n = 4): 1 indicates placement of queens

```
 0 0 1 0
 1 0 0 0
 0 0 0 1
 0 1 0 0
```

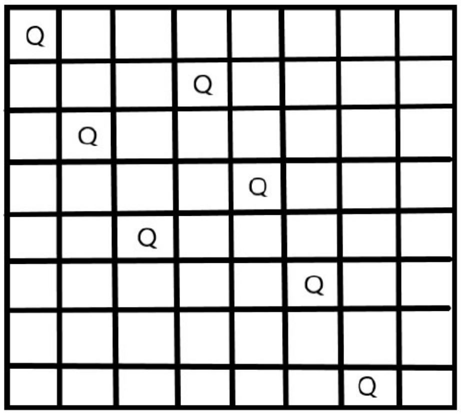
Explanation of the above code solution:



These are two possible solutions from the entire solution set for the 8 queen problem.

```
main()
{
 call placeNQueens(8),

 placeNQueens(){

  call nQueenHelper(8,0){  row = 0

  if(row==n) // won't execute as 0 != 8

  for(int j=0; j<8; j++){
  {
    if(isPossible==true)
    {  board[0][0] = 1  // board[row][0] = 1

      call nQueenHelper(8,row+1)  // recur for all rows further

      print matrix when row = 8 if solution obtained
      and (row==n) condition is met
    }
    board[0][0] = 0  // backtrack and try for
                 // different configurations
  }
}
```

For example, the following configuration won't be displayed



no solution with this configuration, hence we backtrack and try again

Time Complexity Analysis

1. The isPossible method takes O(n) time
2. For each invocation of loop in nQueenHelper, it runs for O(n) time
3. The isPossible condition is present in the loop and also calls nQueenHelper which is recursive

Adding this up, the recurrence relation is:

T(n) = O(n^2) + n * T(n-1)

Solving the above recurrence by iteration or recursion tree,the time complexity of the nQueen problem is = O(N!)

7.4 Sum of subsets problem

Subset sum problem is to find subset of elements that are selected from a given set whose sum adds up to a given number K. We are considering the set contains non-negative values. It is assumed that the input set is unique (no duplicates are presented).

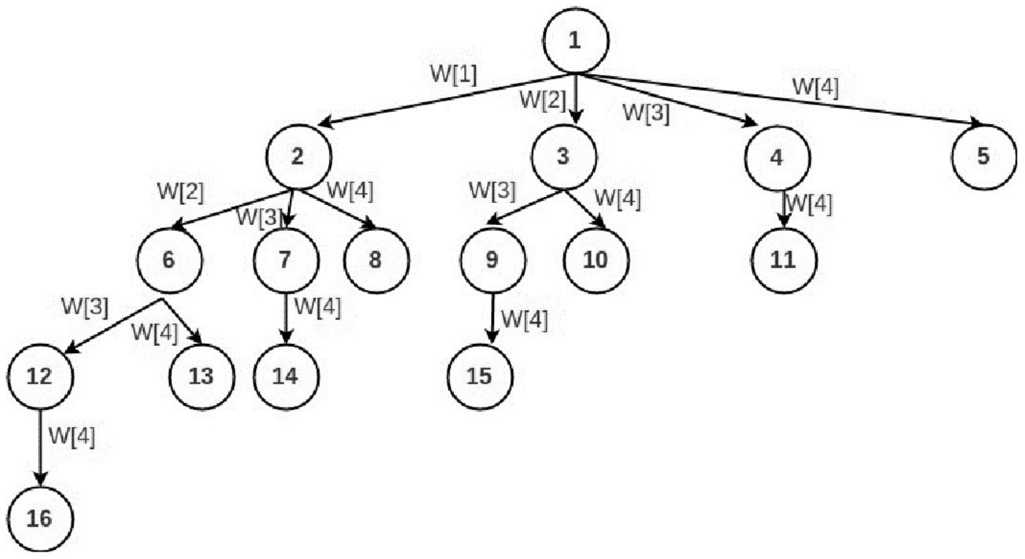
Exhaustive Search Algorithm for Subset Sum

One way to find subsets that sum to K is to consider all possible subsets. A power set contains all those subsets generated from a given set. The size of such a power set is 2N.

Backtracking Algorithm for Subset Sum

Using exhaustive search, we consider all subsets irrespective of whether they satisfy given constraints or not. Backtracking can be used to make a systematic consideration of the elements to be selected.

Assume given set of 4 elements, say w [1] … w[4]. Tree diagrams can be used to design backtracking algorithms. The following tree diagram depicts approach of generating variable sized tuple.



In the above tree, a node represents function call and a branch represents candidate element. The root node contains 4 children. In other words, root considers every element of the set as different branch. The next level sub-trees correspond to the subsets that includes the parent node. The branches at each level represent tuple element to be considered. For example, if we are at level 1, tuple_vector[1] can take any value of four branches generated. If we are at level 2 of left most node, tuple_vector[2] can take any value of three branches generated, and so on…

For example the left most child of root generates all those subsets that include w[1]. Similarly the second child of root generates all those subsets that includes w[2] and excludes w[1].

As we go down along depth of tree we add elements so far, and if the added sum is satisfying explicit constraints, we will continue to generate child nodes further. Whenever the constraints are not met, we stop further generation of sub-trees of that node, and backtrack to previous node to explore the nodes not yet explored. In many scenarios, it saves considerable amount of processing time.

The tree should trigger a clue to implement the backtracking algorithm (try yourself). It prints all those subsets whose sum add up to given number. We need to explore the nodes along the breadth and depth of the tree. Generating nodes along breadth is controlled by loop and nodes along the depth are generated using recursion (post order traversal). Pseudo code given below,

```
if(subset is satisfying the constraint)
   print the subset
   exclude the current element and consider next element
else
   generate the nodes of present level along breadth of tree and
   recur for next levels
```

Following is the implementation of subset sum using variable size tuple vector. Note that the following program explores all possibilities similar to exhaustive search. It is to demonstrate how backtracking can be used. See next code to verify, how we can optimize the backtracking solution.

```
#include <stdio.h>
#include <stdlib.h>

#define ARRAYSIZE(a) (sizeof(a))/(sizeof(a[0]))

static int total_nodes;
// prints subset found
```

```
void printSubset(int A[], int size)
{
        for(int i = 0; i < size; i++)
        {
                printf("%*d", 5, A[i]);
        }

        printf("\n");
}

// inputs
// s            - set vector
// t            - tuplet vector
// s_size       - set size
// t_size       - tuplet size so far
// sum          - sum so far
// ite          - nodes count
// target_sum - sum to be found
void subset_sum(int s[], int t[],
                        int s_size, int t_size,
                        int sum, int ite,
                        int const target_sum)
{
        total_nodes++;
        if( target_sum == sum )
        {
                // We found subset
                printSubset(t, t_size);
                // Exclude previously added item and consider next candidate
                subset_sum(s, t, s_size, t_size-1, sum - s[ite], ite + 1,
target_sum);
```
[210]
```
                return;
```

```
        }
        else
        {
                // generate nodes along the breadth
                for( int i = ite; i < s_size; i++ )
                {
                        t[t_size] = s[i];
                        // consider next level node (along depth)
                        subset_sum(s, t, s_size, t_size + 1, sum + s[i], i + 1,
target_sum);
                }
        }
}

// Wrapper to print subsets that sum to target_sum
// input is weights vector and target_sum
void generateSubsets(int s[], int size, int target_sum)
{
        int *tuplet_vector = (int *)malloc(size * sizeof(int));

        subset_sum(s, tuplet_vector, size, 0, 0, 0, target_sum);

        free(tuplet_vector);
}

int main()
{
        int weights[] = {10, 7, 5, 18, 12, 20, 15};
        int size = ARRAYSIZE(weights);

        generateSubsets(weights, size, 35);
        printf("Nodes generated %d \n", total_nodes);
        return 0;
}
```

The power of backtracking appears when we combine explicit and implicit constraints, and we stop generating nodes when these checks fail. We can improve the above algorithm by strengthening the constraint checks and pre-sorting the data. By sorting the initial array, we need not to consider rest of the array, once the sum so far is greater than target number. We can backtrack and check other possibilities.

Similarly, assume the array is pre-sorted and we found one subset. We can generate next node excluding the present node only when inclusion of next node satisfies the constraints. Given below is optimized implementation (it prunes the sub tree if it is not satisfying constraints).

```
#include <stdio.h>
#include <stdlib.h>

#define ARRAYSIZE(a) (sizeof(a))/(sizeof(a[0]))

static int total_nodes;

// prints subset found
void printSubset(int A[], int size)
{
        for(int i = 0; i < size; i++)
        {
                printf("%*d", 5, A[i]);
        }

        printf("n");
}

// qsort compare function
int comparator (const void *pLhs, const void *pRhs)
```

```
{
          int *lhs = (int *)pLhs;
          int *rhs = (int *)pRhs;

          return *lhs > *rhs;
}

// inputs
// s                    - set vector
// t                    - tuplet vector
// s_size               - set size
// t_size               - tuplet size so far
// sum                  - sum so far
// ite                  - nodes count
// target_sum - sum to be found
void subset_sum(int s[], int t[],
                                        int s_size, int t_size,
                                        int sum, int ite,
                                        int const target_sum)
{
          total_nodes++;

          if( target_sum == sum )
          {
                    // We found sum
                    printSubset(t, t_size);

                    // constraint check
                    if( ite + 1 < s_size && sum - s[ite] + s[ite+1] <= target_sum )
                    {
                              // Exclude previous added item and consider next
candidate
```

```
                    subset_sum(s, t, s_size, t_size-1, sum - s[ite], ite + 1,
target_sum);
            }
            return;
        }
        else
        {
            // constraint check
            if( ite < s_size && sum + s[ite] <= target_sum )
            {
                // generate nodes along the breadth
                for( int i = ite; i < s_size; i++ )
                {
                    t[t_size] = s[i];

                    if( sum + s[i] <= target_sum )
                    {
                        // consider next level node (along depth)
                        subset_sum(s, t, s_size, t_size + 1, sum +
s[i], i + 1, target_sum);
                    }
                }
            }
        }
}

// Wrapper that prints subsets that sum to target_sum
void generateSubsets(int s[], int size, int target_sum)
{
        int *tuplet_vector = (int *)malloc(size * sizeof(int));

[214]   int total = 0;
```

```
    // sort the set
    qsort(s, size, sizeof(int), &comparator);

    for( int i = 0; i < size; i++ )
    {
        total += s[i];
    }

    if( s[0] <= target_sum && total >= target_sum )
    {

        subset_sum(s, tuplet_vector, size, 0, 0, 0, target_sum);

    }

    free(tuplet_vector);
}

int main()
{
    int weights[] = {15, 22, 14, 26, 32, 9, 16, 8};
    int target = 53;

    int size = ARRAYSIZE(weights);

    generateSubsets(weights, size, target);

    printf("Nodes generated %dn", total_nodes);

    return 0;
}
```

Output

8 9 14 22n 8 14 15 16n 15 16 22nNodes generated 68

As another approach, we can generate the tree in fixed size tuple analogues to binary pattern. We will kill the sub-trees when the constraints are not satisfied.

7.5 Graph coloring

Graph coloring is the procedure of assignment of colors to each vertex of a graph G such that no adjacent vertices get same color. The objective is to minimize the number of colors while coloring a graph. The smallest number of colors required to color a graph G is called its chromatic number of that graph. Graph coloring problem is a NP Complete problem.

Method to Color a Graph

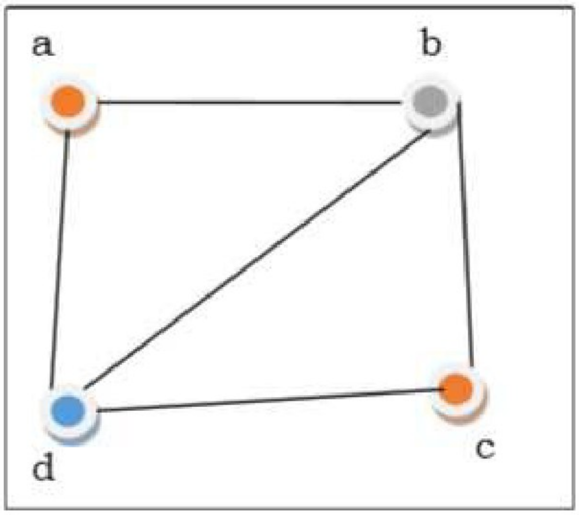
The steps required to color a graph G with n number of vertices are as follows

Step1: Arrange the vertices of the graph in some order.

Step2: Choose the first vertex and color it with the first color.

Step3: Choose the next vertex and color it with the lowest numbered color that has not been colored on any vertices adjacent to it. If all the adjacent vertices are colored with this color, assign a new color to it. Repeat this step until all the vertices are colored.

Example



In the above figure, at first vertex a is colored red. As the adjacent vertices of vertex a are again adjacent, vertex b and vertex d are colored with different color, green and blue respectively. Then vertex c is colored as red as no adjacent vertex of c is colored red. Hence, we could color the graph by 3 colors. Hence, the chromatic number of the graph is 3.

Applications of Graph Coloring:

The graph coloring problem has huge number of applications.

1) Making Schedule or Time Table: Suppose we want to make am exam schedule for a university. We have list different subjects and students enrolled in every subject. Many subjects would have common students (of same batch, some backlog students, etc). How do we schedule the exam so that no two exams with a common student are scheduled at same time? How many minimum time slots are needed to schedule all exams? This problem can be represented as a graph where every vertex is a subject and an edge between two vertices mean there is a common student. So this is a graph coloring problem where minimum number of time slots is equal to the chromatic number of the graph.

2) Mobile Radio Frequency Assignment: When frequencies are assigned to towers, frequencies assigned to all towers at the same location must be different. How to assign frequencies with this constraint? What is the minimum number of frequencies needed? This problem is also an instance of graph coloring problem where every tower represents a vertex and an edge between two towers represents that they are in range of each other.

3) Sudoku: Sudoku is also a variation of Graph coloring problem where every cell represents a vertex. There is an edge between two vertices if they are in same row or same column or same block.

4) Register Allocation: In compiler optimization, register allocation is the process of assigning a large number of target program variables onto a small number of CPU registers. This problem is also a graph coloring problem.

5) Bipartite Graphs: We can check if a graph is Bipartite or not by coloring the graph using two colors. If a given graph is 2-colorable, then it is Bipartite, otherwise not. See this for more details.

6) Map Coloring: Geographical maps of countries or states where no two adjacent cities cannot be assigned same color. Four colors are sufficient to color any map (See Four Color Theorem)

7.6 Hamiltonian cycles

In an undirected graph, the Hamiltonian path is a path, that visits each vertex exactly once, and the Hamiltonian cycle or circuit is a Hamiltonian path, that there is an edge from the last vertex to the first vertex.

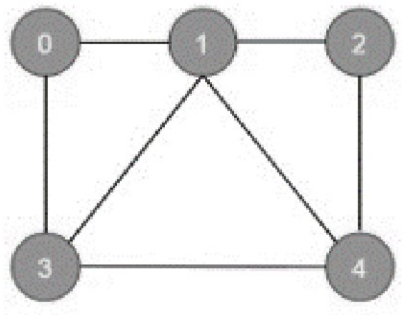
In this problem, we will try to determine whether a graph contains a Hamiltonian cycle or not. And when a Hamiltonian cycle is present, also print the cycle.

Input and Output

Input:

The adjacency matrix of a graph G(V, E).

$$\boxed{\begin{matrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{matrix}}$$



Output:

The algorithm finds the Hamiltonian path of the given graph. For this case it is (0, 1, 2, 4, 3, 0). This graph has some other Hamiltonian paths.

If one graph has no Hamiltonian path, the algorithm should return false.

Algorithm

isValid(v, k)

Input – Vertex v and position k.

Output – Checks whether placing v in the position k is valid or not.

```
Begin
  if there is no edge between node(k-1) to v, then
    return false
  if v is already taken, then
    return false
  return true; //otherwise it is valid
End
```

cycleFound(node k)

Input – node of the graph.

Output – True when there is a Hamiltonian Cycle, otherwise false.

```
Begin
  if all nodes are included, then
    if there is an edge between nodes k and 0, then
      return true
    else
      return false;

  for all vertex v except starting point, do
    if isValid(v, k), then //when v is a valid edge
      add v into the path
      if cycleFound(k+1) is true, then
        return true
      otherwise remove v from the path
  done
  return false
End
```

Example

```
#include<iostream>
#define NODE 5
using namespace std;

int graph[NODE][NODE] = {
  {0, 1, 0, 1, 0},
  {1, 0, 1, 1, 1},
  {0, 1, 0, 0, 1},
  {1, 1, 0, 0, 1},
  {0, 1, 1, 1, 0},
};

/* int graph[NODE][NODE] = {
  {0, 1, 0, 1, 0},
  {1, 0, 1, 1, 1},
  {0, 1, 0, 0, 1},
  {1, 1, 0, 0, 0},
  {0, 1, 1, 0, 0},
}; */

int path[NODE];

void displayCycle() {
  cout<<"Cycle: ";

  for (int i = 0; i < NODE; i++)
    cout << path[i] << " ";
  cout << path[0] << endl;     //print the first vertex again
}

bool isValid(int v, int k) {
  if (graph [path[k-1]][v] == 0)   //if there is no edge
    return false;

  for (int i = 0; i < k; i++)   //if vertex is already taken, skip that
    if (path[i] == v)
      return false;
  return true;
```

```
}

bool cycleFound(int k) {
  if (k == NODE) {           //when all vertices are in the path
    if (graph[path[k-1]][ path[0] ] == 1 )
      return true;
    else
      return false;
  }

  for (int v = 1; v < NODE; v++) {      //for all vertices except starting point
    if (isValid(v,k)) {              //if possible to add v in the path
      path[k] = v;
      if (cycleFound (k+1) == true)
        return true;
      path[k] = -1;             //when k vertex will not in the solution
    }
  }
  return false;
}

bool hamiltonianCycle() {
  for (int i = 0; i < NODE; i++)
    path[i] = -1;
  path[0] = 0; //first vertex as 0

  if ( cycleFound(1) == false ) {
    cout << "Solution does not exist"<<endl;
    return false;
  }

  displayCycle();
  return true;
}
int main() {
  hamiltonianCycle();
}
```

Output

```
Cycle: 0 1 2 4 3 0
```

7.7 Summary

In this unit you have learnt about General method of backtracking, applications of backtracking. You have also learnt about 8-queen problem, sum of subsets problem, graph coloring and Hamiltonian cycles.

- ☐ Backtracking is a technique based on algorithm to solve problem. It uses recursive calling to find the solution by building a solution step by step increasing values with time.
- ☐ Backtracking algorithm is applied to some specific types of problems, Decision problem used to find a feasible solution of the problem.
- ☐ The eight queens puzzle is the problem of placing eight chess queens on an 8×8 chessboard so that no two queens threaten each other; thus, a solution requires that no two queens share the same row, column, or diagonal.
- ☐ The eight queens puzzle is an example of the more general n-queen's problem of placing n non-attacking queens on an n×n chessboard, for which solutions exist for all natural numbers n with the exception of n = 2 and n = 3.
- ☐ Hamiltonian Path in an undirected graph is a path that visits each vertex exactly once. A Hamiltonian cycle (or Hamiltonian circuit) is a Hamiltonian Path such that there is an edge (in the graph) from the last vertex to the first vertex of the Hamiltonian Path. Determine whether a given graph contains Hamiltonian Cycle or not. If it contains, then prints the path.

7.8 Review Questions

Q1.Print all possible solutions to N Queens problem using backtracking.

Q2. Which is the optimal solution for 8-queen's problem? Explain with example.

Q3. What is the difference between recursion and backtracking? Elaborate your answer.

Q4. What is the technique used to solve sum of subsets problem and explain the sum of subsets problem?

Q5. How can we find the number of Hamiltonian cycles in a complete undirected graph?

UNIT-8 Branch-And-Bound

Structure

8.0 Introduction

This is the third unit of this block. In this block you will learn about branch-and-bound. There are five sections in this unit. In the first section i.e. Section 3.1 you will study about method of branch-and-bound. Section 3.2 describe about travelling salesperson problem. In the section 3.3 you will learn about 15 puzzle problem. Section 3.4 and 3.5 define summary and review questions.

Objective

After studying this unit, you should be able to define:

- ☐ Branch-And-Bound method
- ☐ Travelling salesperson problem
- ☐ 15 puzzle problem

8.1 Branch-And-Bound method

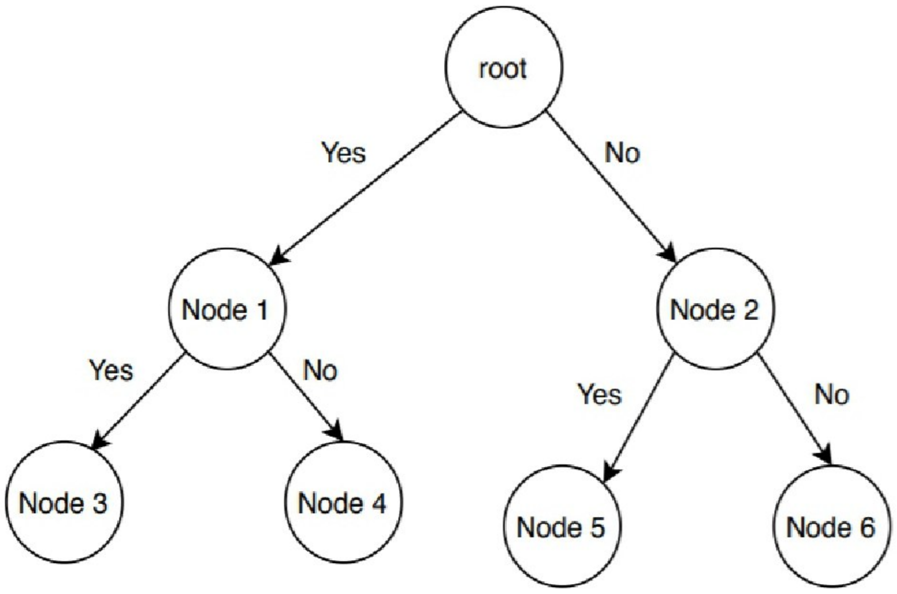
In computer science, there is a large number of optimization problems which has a finite but extensive number of feasible solutions. Among these, some problems like finding the shortest path in a graph or Minimum Spanning Tree can be solved in polynomial time.

A significant number of optimization problems like production planning, crew scheduling can't be solved in polynomial time, and they belong to the NP-Hard class. These problems are the example of NP-Hard combinatorial optimization problem.

Branch and bound (B&B) is an algorithm paradigm widely used for solving such problems.

Branch and bound algorithms are used to find the optimal solution for combinatory, discrete, and general mathematical optimization problems. In general, given an NP-Hard problem, a branch and bound algorithm explores the entire search space of possible solutions and provides an optimal solution.

A branch and bound algorithm consist of stepwise enumeration of possible candidate solutions by exploring the entire search space. With all the possible solutions, we first build a rooted decision tree. The root node represents the entire search space:



Here, each child node is a partial solution and part of the solution set. Before constructing the rooted decision tree, we set an upper and lower bound for a given problem based on the optimal solution. At each level, we need to make a decision about which node to include in the solution set. At each level, we explore the node with the best bound. In this way, we can find the best and optimal solution fast.

Now it is crucial to find a good upper and lower bound in such cases. We can find an upper bound by using any local optimization method or by picking any point in the search space. On the other hand, we can obtain a lower bound from convex relaxation or duality.

In general, we want to partition the solution set into smaller subsets of solution. Then we construct a rooted decision tree, and finally, we choose the best possible subset (node) at each level to find the best possible solution set.

Example

In this section, we'll discuss how the job assignment problem can be solved using a branch and bound algorithm.

Problem Statement

Let's first define a job assignment problem. In a standard version of a job assignment problem, there can be N jobs and N workers. To keep it simple, we're taking 3 jobs and 3 workers in our example:

	Job 1	Job 2	Job 3
A	9	3	4
B	7	8	4
C	10	5	2

We can assign any of the available jobs to any worker with the condition that if a job is assigned to a worker, the other workers can't take that particular job. We should also notice that each job has some cost associated with it, and it differs from one worker to another.

Here the main aim is to complete all the jobs by assigning one job to each worker in such a way that the sum of the cost of all the jobs should be minimized.

Branch and Bound Algorithm Pseudocode

Now let's discuss how to solve the job assignment problem using a branch and bound algorithm.

Let's see the pseudocode first:

Algorithm 1: Job assignment problem using branch and bound

```
Data: Input cost matrixM[] []

Result: Assignment of jobs to each worker according to optimal cost

Function MinCost(M[] [])

while True do

        E = LeastCost()

        If E is a leaf node,then
```

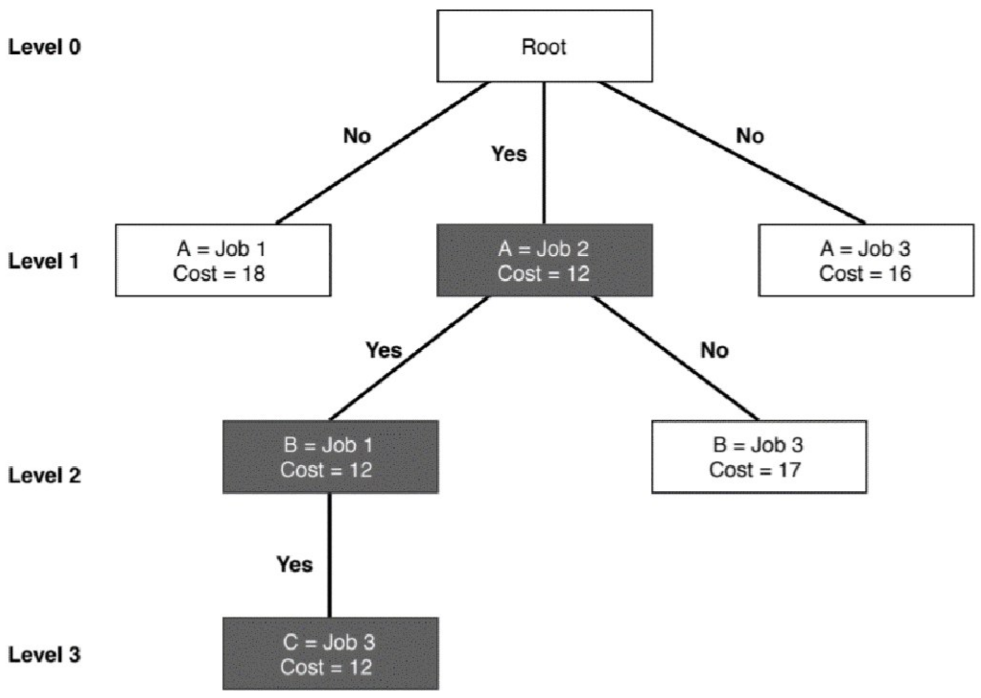
```
            print();
            return;
        end

        foreach child S to E do
            Add(S);
            S → parent = E;

        end

    end
```

Here, M[][] is the input cost matrix that contains information like the number of available jobs, a list of available workers, and the associated cost for each job. The function MinCost() maintains a list of active nodes. The function LeastCost() calculates the minimum cost of the active node at each level of the tree. After finding the node with minimum cost, we remove the node from the list of active nodes and return it.

We're using the Add() function in the pseudocode, which calculates the cost of a particular node and adds it to the list of active nodes.

In the search space tree, each node contains some information, such as cost, a total number of jobs, as well as a total number of workers.

Now let's run the algorithm on the sample example we've created:



Initially, we've 3 jobs available. The worker *A* has the option to take any of the available jobs. So at level 1, we assigned all the available jobs to the worker *A* and calculated the cost. We can see that when we assigned jobs 2 to the worker *A*, it gives the lowest cost in level 1 of the search space tree. So we assign the job 2 to worker *A* and continue the algorithm. "Yes" indicates that this is currently optimal cost.

After assigning the job 2 to worker *A*, we still have two open jobs. Let's consider worker *B* now. We're trying to assign either job 1 or 3 to worker *B* to obtain optimal cost.

Either we can assign the job 1 or 3 to worker *B*. Again we check the cost and assign job 1 to worker *B* as it is the lowest in level 2.

Finally, we assign the job 3 to worker C, and the optimal cost is 12.

8.2 Travelling salesperson problem

Problem Statement

A traveller needs to visit all the cities from a list, where distances between all the cities are known and each city should be visited just once. What is the shortest possible route that he visits each city exactly once and returns to the origin city?

Solution

Travelling salesperson problem is the most notorious computational problem. We can use brute-force approach to evaluate every possible tour and select the best one. For n number of vertices in a graph, there are (n - 1)! number of possibilities.

Instead of brute-force using dynamic programming approach, the solution can be obtained in lesser time, though there is no polynomial time algorithm.

Let us consider a graph G = (V, E), where V is a set of cities and E is a set of weighted edges. An edge e(u, v) represents that vertices u and v are connected. Distance between vertex u and v is d(u, v), which should be non-negative.

Suppose we have started at city 1 and after visiting some cities now we are in city j. Hence, this is a partial tour. We certainly need to know j, since this will determine which cities are most convenient to visit next. We also need to know all the cities visited so far, so that we don't repeat any of them. Hence, this is an appropriate sub-problem.

For a subset of cities $S \in \{1, 2, 3, \dots, n\}$ that includes 1, and $j \in S$, let C(S, j) be the length of the shortest path visiting each node in S exactly once, starting at 1 and ending at j.

When |S| > 1, we define $C(S, 1) = \propto$ since the path cannot start and end at 1.

Now, let express C(S, j) in terms of smaller sub-problems. We need to start at 1 and end at j. We should select the next city in such a way that

$$C(S,j) = \min C(S-\{j\},i) + d(i,j) \text{ where } i \in S \text{ and } i \neq j c(S,j)$$

$$= \min C(s-\{j\},i) + d(i,j) \text{ where } i \in S \text{ and } i \neq j$$

Algorithm: Traveling-Salesperson-Problem

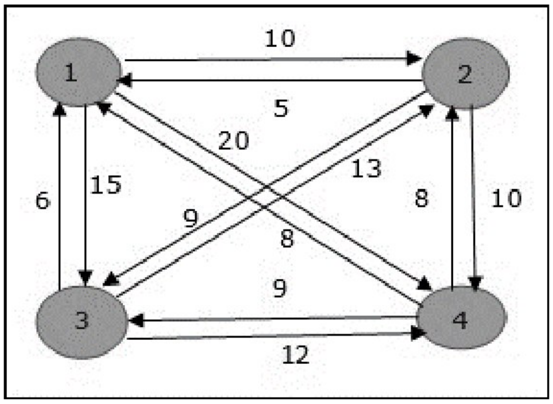
```
$C(\{1\}, 1) = 0$
for $s = 2$ to $n$ do
  for all subsets $S \in \{1, 2, 3, \dots, n\}$ of size $s$ and containing 1
    $C(S, 1) = \infty$
  for all $j \in S$ and $j \neq 1$
    $C(S, j) = \min \{C(S - \{j\}, i) + d(i, j)$ for $i \in S$ and $i \neq j\}$
Return $\min_j C(\{1, 2, 3, \dots, n\}, j) + d(j, i)$
```

Analysis

There are at the most 2nsub-problems and each one takes linear time to solve. Therefore, the total running time is $O(2^n.n^2)$.

Example

In the following example, we will illustrate the steps to solve the travelling salesperson problem.



From the above graph, the following table is prepared.

	1	2	3	4
1	0	10	15	20
2	5	0	9	10
3	6	13	0	12
4	8	8	9	0

S = Φ

$Cost(2,\Phi,1)=d(2,1)=5Cost(2,\Phi,1)=d(2,1)=5$
$Cost(3,\Phi,1)=d(3,1)=6Cost(3,\Phi,1)=d(3,1)=6$
$Cost(4,\Phi,1)=d(4,1)=8Cost(4,\Phi,1)=d(4,1)=8$

S = 1

$Cost(i,s)=min\{Cost(j,s-(j))+d[i,j]\}Cost(i,s)=min\{Cost(j,s)-(j))+d[i,j]\}$

$Cost(2,\{3\},1)=d[2,3]+Cost(3,\Phi,1)=9+6=15cost(2,\{3\},1)=d[2,3]+cost(3,\Phi,1)$

$=9+6=15$

$Cost(2,\{4\},1)=d[2,4]+Cost(4,\Phi,1)=10+8=18cost(2,\{4\},1)=d[2,4]+cost(4,\Phi,1)$

$=10+8=18$

$Cost(3,\{2\},1)=d[3,2]+Cost(2,\Phi,1)=13+5=18cost(3,\{2\},1)=d[3,2]+cost(2,\Phi,1)$

$=13+5=18$

$Cost(3,\{4\},1)=d[3,4]+Cost(4,\Phi,1)=12+8=20cost(3,\{4\},1)=d[3,4]+cost(4,\Phi,1)$

=12+8=20

Cost(4,{3},1)=*d*[4,3]+*Cost*(3,Φ,1)=9+6=15*cost*(4,{3},1)=*d*[4,3]+*cost*(3,Φ,1)

=9+6=15

Cost(4,{2},1)=*d*[4,2]+*Cost*(2,Φ,1)=8+5=13*cost*(4,{2},1)=*d*[4,2]+*cost*(2,Φ,1)

=8+5=13

S = 2

$$Cost(2,\{3,4\},1$$

$$= \begin{cases} \text{d}[2,3] + \text{Cost}(3,\{4\},1) = 9 + 20 = 29 \\ \text{d}[2,4] + \text{Cost}(4,\{3\},1) = 10 + 15 = 25 = 25\text{Cost}(2,\{3,4\},1) = 25 \\ \{d[2,3] + cost(3,\{4\},1) = 9 + 20 = 29d[2,4] + Cost(4,\{3\},1) = 10 + 15 = 25 \end{cases}$$

$$Cost(3,\{2,4\},1)$$

$$= \begin{cases} \text{d}[3,2] + \text{Cost}(2,\{4\},1) = 13 + 18 = 31 \\ \text{d}[3,4] + \text{Cost}(4,\{2\},1) = 12 + 13 = 25 = 25\text{Cost}(3,\{2,4\},1) = 25 \\ \{d[3,2] + cost(2,\{4\},1) = 13 + 18 = 31d[3,4] + Cost(4,\{2\},1) = 12 + 13 = 25 \end{cases}$$

$$Cost(4,\{2,3\},1)$$

$$= \begin{cases} \text{d}[4,2] + \text{Cost}(2,\{3\},1) = 8 + 15 = 23 \\ \text{d}[4,3] + \text{Cost}(3,\{2\},1) = 9 + 18 = 27 = 23\text{Cost}(4,\{2,3\},1) = 23 \\ \{d[4,2] + cost(2,\{3\},1) = 8 + 15 = 23d[4,3] + Cost(3,\{2\},1) = 9 + 18 = 27 \end{cases}$$

S = 3

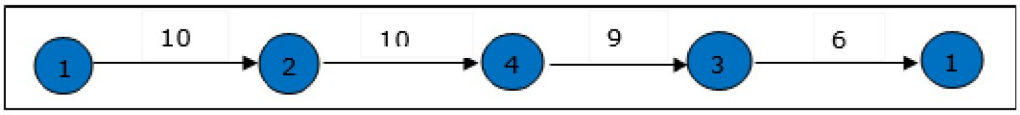
$$Cost(1,\{2,3,4\},1)$$

$$= \begin{cases} \text{d}[1,2] + \text{Cost}(2,\{3,4\},1) = 10 + 25 = 35 \\ \text{d}[1,3] + \text{Cost}(3,\{2,4\},1) = 15 + 25 = 40 \\ \text{d}[1,4] + \text{Cost}(4,\{2,3\},1) = 20 + 23 = 43 = 35\text{cost}(1,\{2,3,4\}),1) \\ \text{d}[1,2] + \text{cost}(2,\{3,4\},1) = 10 + 25 = 35 \\ \text{d}[1,3] + \text{cost}(3,\{2,4\},1) = 15 + 25 = 40 \\ \text{d}[1,4] + \text{cost}(4,\{2,3\},1) = 20 + 23 = 43 = 35 \end{cases}$$

The minimum cost path is 35.

Start from cost {1, {2, 3, 4}, 1}, we get the minimum value for d [1, 2]. When s = 3, select the path from 1 to 2 (cost is 10) then go backwards. When s = 2, we get the minimum value for d [4, 2]. Select the path from 2 to 4 (cost is 10) then go backwards.

When s = 1, we get the minimum value for d [4, 3]. Selecting path 4 to 3 (cost is 9), then we shall go to then go to s = Φ step. We get the minimum value for d [3, 1] (cost is 6).

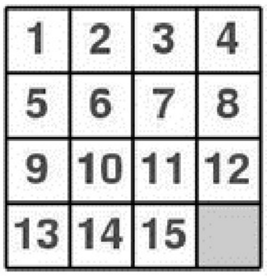


Check your progress

Q1.What is branch and bound method in DAA? Explain with example.
Q2.What is Travelling Salesman Problem?Explain with example.

8.3 15-puzzle problem

The Fifteen Puzzle has been around for over a hundred years, and has been a craze for almost every generation. The basic form is of a 4 by 4 grid usually made with sliding tiles in a tray. There are 15 tiles numbered 1 to 15 and the 16th place is empty as shown. Before play begins the tiles are randomly scrambled by sliding to any starting position. The object of the game is then to unscramble the tiles (by sliding, not lifting them) to get them into consecutive order, with the space in the bottom right again. (This will be referred to as the home position.)

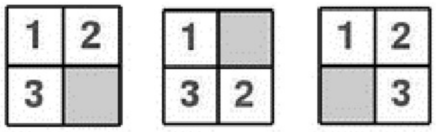


Sam Loyd was the man who invented the 14-15 or Boss puzzle. This was a version where the starting position was very similar to the home position except that the 14 and 15 were inverted. In the early 1870's he offered a $1000 reward to anyone who could solve it, which set off "fifteen fever". However, no-one claimed the prize for the simple reason that it is not solvable! The reason shall have dealt with later.

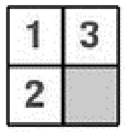
Permutations, Interchanges and Parity

It can be seen that the puzzle has 15 tiles and a space distributed over 16 squares. The number of different arrangements of tiles can be calculated: the first square has 16 possible numbers/space to choose from, the 2nd will have 15 possibilities...etc so there are 16 factorial possible arrangements. This is over 20 billion different arrangements. It shall be seen however that only 16!/2 of these are solvable.

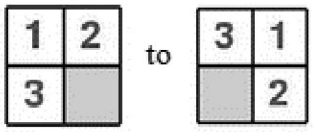
To understand the principles involved, first look at a simpler case of permutations (or arrangements) of a 2 by 2 grid, with 3 tiles; 1,2,3.



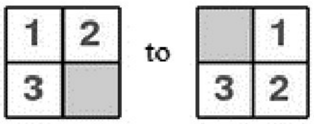
These are 3 of the arrangements of the grid, with the first picture being the home state; 9 other arrangements can be found by cycling the 1 round to a new position and then rearranging the 2 and 3. Of course there are 4! = 24 possible arrangements, but the other 12 cannot be reached by simple sliding moves. They would be found by performing an interchange; i.e. where one tile is moved to the position of another tile or the space, which in turn is moved somewhere else, possibly to the first tile's original position. (i.e. swapping two of the tiles). One can see that the following arrangement is not possible from the home state by following legal moves (i.e. sliding not lifting):



Cycles can be used to describe the permutations of objects through interchanges. Lifting a tile up is not a legal operation, but a cycle of interchanges may describe a set of legal sliding moves in a shorter form. For example, (3 1 2 *) would be the cycle from



Where * is the empty space. Of course one can see that this is the same as (2 * 3 1). In a similar way, (2 * 1) would be the permutation of

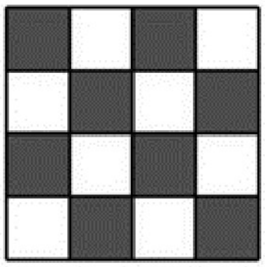


As the 3 is not mentioned, there is an implication that it is left in its original position. Thus the impossible arrangement seen previously would have a cycle from the home state of the form (2 3), which can easily have been seen to be an interchange that is not a legal permutation. It shall be seen later how to tell generally if a particular permutation is legal for a particular starting state.

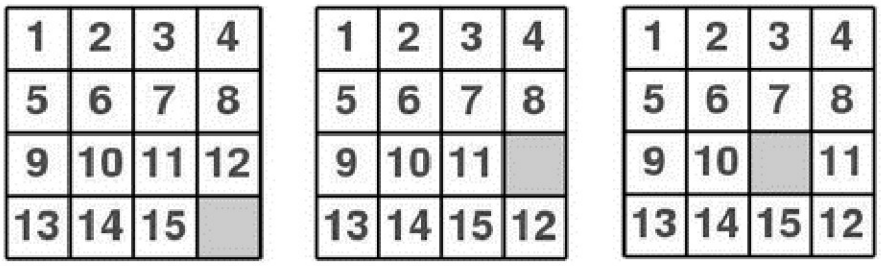
Cycles can be strung together to get more permutations, which will be especially useful for the 4 by 4 grid later on. If for example (2 * 1) is performed to the home state then (1 2 3 *), as long as each individual cycle is legal for the previous state, the combination of them will also be possible. Permutations describe the interchanges between tiles and so combining them will not necessarily produce permutations that can be performed by sliding tiles in those orders; a cycle could represent an interchange that could be possible by another, longer cycle of sliding permutations.

One can differentiate between types of cycles by breaking them up into single cycles. For example (2 1 3 *) can be written (2 1)(2 3)(2 *) which is applied by following each number through the cycles. This called an odd permutation as it has an odd number of products. In the same way (2 * 1)=(2 *)(2 1) is even. Notice that the length of an odd permutation is an even number, and vice versa. Combining two permutations of the same type is even, but combining an odd and even permutation will be odd.

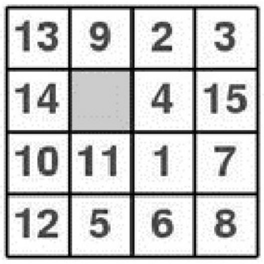
These same rules can be applied to the 4 by 4 grid, by permutations of squares or rectangles of any size. One can tell that the solution will need to be an even permutation if the empty space is left in its home slot. (See following pictures and text.) Indeed, it can be said that if the empty space is originally on any of the green squares, the solution will be an even permutation, but if the space is on a white square, an odd permutation is needed. These can be described by parity; if there is a solution, one says the problem has even parity ie that the starting state has the same parity as the home state. This is why only half the possible arrangements are solvable; half the time the problem has even parity, and half, odd parity.



One can see that this is true by looking at the home state:



By sliding the 12 down one permutation is performed (12 *) which is odd. To get back to the home state therefore the solution will be an odd permutation as the empty space was left on a white square. Similarly, if the starting state is (12 * 11) from the home state, one can tell that the solution is possible with an even permutation. In the same way one can look at any shuffled starting position:



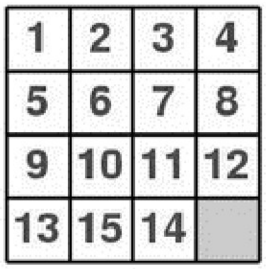
Here the permutations are

(1 11 10 9 2 3 4 7 12 13 8 * 6 15)(5 14)which is a combination of two odd permutations so is even. The empty square is found on a green position so one can say that this problem is solvable.

The 14-15 Puzzle and other problems

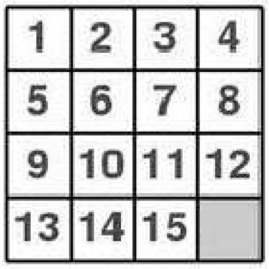
Now Loyd's 14-15 puzzle shall be examined: this is the starting state which looks like this:

Even though this appears to be so close to the final solution, one can tell from the cycle that it is not solvable, because (14 15) is an odd permutation and the space on a green position, i.e. odd parity.



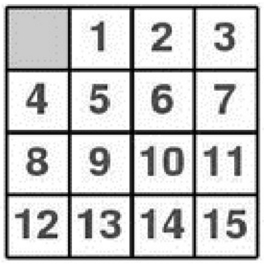
Other starting states that can be looked at are the effect of rotating the grid clockwise 90 degrees. The permutation is

(1 4 * 13)(2 8 15 9)(3 12 14 5)(6 7 11 10)which is a combination of 4 odd permutations which is itself even. The empty space is on a white position so this problem (including the problem of having the tiles on their sides) is unsolvable.



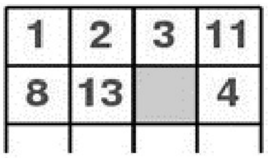
If one takes the problem of having the empty space at the beginning rather than at the end, the permutation cycle is

(1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 *)which is an odd permutation with the space on a green position so it is an unsolvable problem.



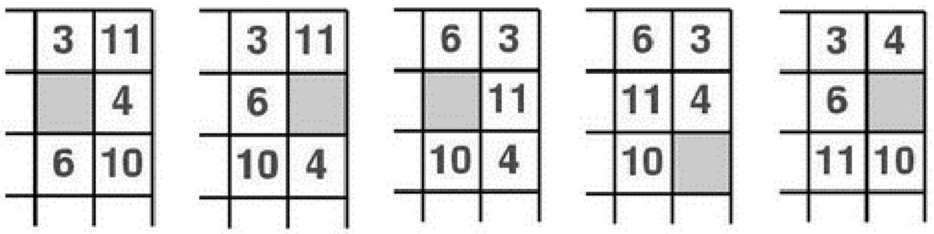
Finding Solutions-Simple

Of course after working out whether a particular problem is solvable, it would then be useful to be able to find a solution. One logical process to follow is to get the tiles in place in order, i.e.by getting the 1 into place, then the 2, 3, 4 etc. However, one may quickly find that by placing the 3 in place, the 4 cannot fit without disturbing the 3 again. From the work before, it is known that a permutation of (11 4) on the top right-hand 2 by 2 grid is not legal by sliding so instead permutations of a 2 by 3 grid are used.



This is the same as a permutation of (4 10 6 *)(6 3 11 *)(11 * 4)(3 6 11 10 * 4)= (11 6 * 4). In a similar way, the 7 and 8 can be placed in the correct positions. The bottom two lines will have similar problems; the 9 and 13 need to be in the right order then cycled round to their positions, and finally one is left with another 3 by 2 grid with the 10 and 14 to be placed first like the 3 and

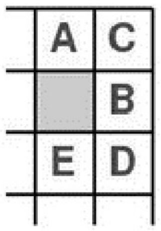
4, then the last three rotated to get home! Of course it makes sense to think about the other numbers and where they should eventually end up, and so move them in those particular directions.



For the general case if a tile, B needs to get into a upper right corner (either of the grid, or previously positioned tiles) in the order A B at the top, the cycle of moves from this position is thus:

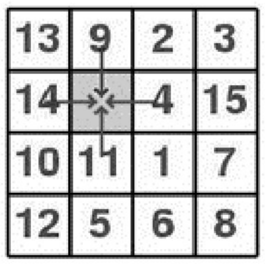
(A C B D E *)(D B C E *)(B A E C*) in terms of the legal slides.

These motions can then be used for any part of the problem in a corner, just mirror the permutation to fit the corner.



Finding Solutions-Fastest

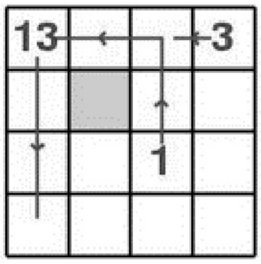
Obviously trying to solve the problem like this is not the fastest (i.e. the one with the least number of moves) as one tends to move some tiles in the wrong direction unnecessarily as the first ones are concentrated on individually. As one can see in arranging the corner piece, it is sometimes necessary to go backwards to be able to go forwards, but these motions ought to be limited where possible



A computer could be used to evaluate all possible solutions and then show the quickest one. It would look at all possible moves at each node- where there is a branching off of possible solutions. For example, in the previous problem (shown again to the left), the starting position has a node of different 4 paths or branches; the 4, 9, 11 or 14 could be moved. If the 9 is chosen to be moved, the new node would have 3 branches, but the 9 would be discounted as it leads back to the original position.

One way of getting the computer to perform the search is to make it evaluate possibilities at each node, to pick one path and follow it until either the home state is reached or a dead end. If a dead end is reached, the computer goes back to the last node and takes a different path. However, one cannot be sure that this is the fastest, as it discards all other possibilities once it has found a solution. Instead the computer could examine all nodes that are one permutation from the starting position. If none of these are the solution they are expanded to nodes 2 single permutations away from the starting state, etc until the home state is found, so this will be the shortest. This however is very complicated and certainly not something a human could do easily. There are many sites on the internet that have programs to find the quickest solution (see sources) so instead the problem of whether a person can solve the fifteen puzzle in the shortest solution will be examined.

One thing a human can do is to work out a heuristic or best fit- an estimate of the number of moves a tile would take to get from a particular position on the grid to the home state. The heuristic really wants to be either optimistic or correct as it will show the minimum number of moves expected to get home. It is worked out by looking at the position of each tile in the starting state and calculating the number of moves it would take to get to its home place if the space were adjacent to it, in the direction of home. In the example, the 1 would need 4 moves to get home (so the heuristic is 4), the 13 would need 3, the 3, 1 move. It is known that the heuristic calculated in this way is optimistic or accurate, because if the empty space were to the left of its position now, the 13 could get to its home place in the heuristic's estimated number of moves. However, as it is not, the heuristic would be larger as the tile cannot be simply lifted up. The heuristic of a particular problem is just the sum of all the individual heuristics.



It would seem sensible to work out the total heuristic for a particular problem and then follow the path of best fit where the value of the heuristic decreases. For the example taken, the heuristic at the start would be 4 + 1 + 1 +...+ 3 + 3 + 3 = 36 for the tiles in chronological order. To decrease the heuristic, the 9 or 14 could be moved, as they are the only two moves that move the tile closer to its home position. Moving one tile in its home direction decreases the value of the heuristic by one. A person can keep track of the moves made and where the corresponding best fit nodes are. When a dead end is reached, the person should go back to the last node and examine the other branches, until a solution is found. However, in practice this does not work out. It was seen from the process of moving corner pieces into position that it is necessary to move tiles in what seems the wrong direction in order to solve the problem.

8.4 Summary

In this unit you have learnt about the concept of the methodof branch-and-bound, travelling salesperson problem, and 15 puzzle problem.

- ☐ Branch and bound is a systematic method for solving optimization problems. B&B is a rather general optimization technique that applies where the greedy method and dynamic programming fail.
- ☐ Travelling Salesman Problem (TSP): Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point. Note the difference between Hamiltonian Cycle and TSP.
- ☐ The 15 puzzle problem is invented by sam loyd in 1878. In this problem there are 15 tiles, which are numbered from 0 – 15. The objective of this problem is to transform the arrangement of tiles from initial arrangement to a goal arrangement.

8.5 Review Questions

Q1. What is difference between backtracking and branch and bound? Elaborate your answer.

Q2. What is the time complexity of travelling salesperson problem? Explain with suitable example.

Q3. How to check if an instance of 15 puzzle is solvable? Elaborate your answer.

Q4. What type of task generation is used for a 15 puzzle problem?

Q5. What is the solution of 15 puzzle problem? Explain with example.

UNIT-9 NP-Hard and NP-Complete problems

Structure

9.0 Introduction

This is the fourth and last unit of this block. This unit define about NP-Hard and NP-Complete problems. In this unit, there are seven sections. In the Section 4.1 you will learn about basic concept of NP-Hard and NP-Complete problems. Section 4.2 describe about non deterministic algorithms. Section 4.3 explain about NP - Hard and NP Complete classes. In the section 4.4 you will learn about satisfiability problem. Last topic defines reducibility in the section 4.5. Summary and Review questions define in the section 4.6 and 4.7 respectively.

Objective

After studying this unit, you should be able to define:

- ☐ Basic conceptsNP-Hard and NP-Complete problems
- ☐ Non-deterministic algorithms
- ☐ NP - Hard and NP Complete classes
- ☐ Satisfiability problem
- ☐ Reducibility

9.1 Basic concepts

The computing times of algorithms fall into two groups.

Group1– consists of problems whose solutions are bounded by the polynomial of small degree.

Example – Binary search o (log n), sorting o(n log n), matrix multiplication 0(n 2.81).

NP –HARD AND NP – COMPLETE PROBLEMS

Group2 – contains problems whose best known algorithms are non-polynomial.

Example –Traveling salesperson problem 0(n22n), knapsack problem 0(2n/2) etc.

There are two classes of non-polynomial time problem

1. *NP- hard*
2. *NP-complete*

A problem which is NP complete will have the property that it can be solved in polynomial time iff all other NP – complete problems can also be solved in polynomial time. The class NP (meaning non-deterministic polynomial time) is the set of problems that might appear in a puzzle magazine: "Nice puzzle".

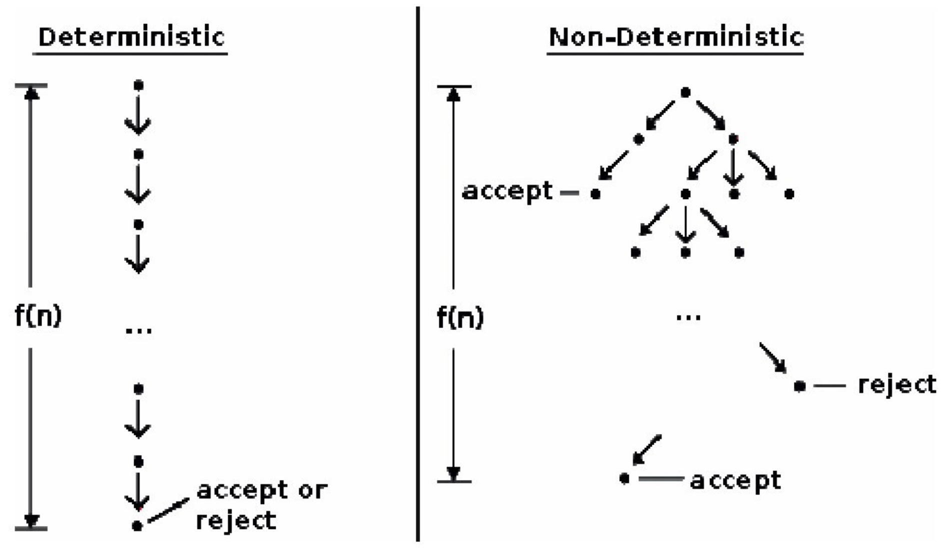
What makes these problems special is that they might be hard to solve, but a short answer canalways be printed in the back, and it is easy to see that the answer is correct once you see it.

Example... Does matrix A have LU decomposition? No guarantee if answer is "no".

Another way of thinking of NP is it is the set of problems that can solved efficiently by a really good guesser. The guesser essentially picks the accepting certificate out of the air (Non-deterministic Polynomial time). It can then convince itself that it is correct using a polynomial time algorithm. (Like a right-brain, left-brain sort of thing.) Clearly this isn't a practically useful characterization: how could we build such a machine?

9.2 Non deterministic algorithms

In computer science, a nondeterministic algorithm is an algorithm that, even for the same input, can exhibit different behaviours on different runs, as opposed to a deterministic algorithm. There are several ways an algorithm may behave differently from run to run. A concurrent algorithm can perform differently on different runs due to a race condition. A probabilistic algorithm's behaviours dependon a random number generator. An algorithm that solves a problem in nondeterministic polynomial time can run in polynomial time or exponential time depending on the choices it makes during execution. The nondeterministic algorithms are often used to find an approximation to a solution, when the exact solution would be too costly to obtain using a deterministic one. The notion was introduced by Robert W. Floyd in 1967.



Often in computational theory, the term "algorithm" refers to a deterministic algorithm. A nondeterministic algorithm is different from its more familiar deterministic counterpart in its ability to arrive at outcomes using various routes. If a deterministic algorithm represents a single path from an input to an outcome, a nondeterministic algorithm represents a single path stemming into many paths, some of which may arrive at the same output and some of which may arrive at unique outputs. This property is captured mathematically in "nondeterministic" models of computation such as the nondeterministic finite automaton. In some scenarios, all possible paths are allowed to run simultaneously.

In algorithm design, nondeterministic algorithms are often used when the problem solved by the algorithm inherently allows multiple outcomes (or when there is a single outcome with multiple paths by which the outcome may be discovered, each equally preferable). Crucially, every outcome the nondeterministic algorithm produces are valid, regardless of which choices the algorithm makes while running.

In computational complexity theory, nondeterministic algorithms are ones that, at every possible step, can allow for multiple continuations (imagine a person walking down a path in a forest and, every time they step further, they must pick which fork in the road they wish to take). These algorithms do not arrive at a solution for every possible computational path; however, they are guaranteed to arrive at a correct solution for some path (i.e., the person walking through the forest may only find their cabin if they pick some combination of "correct" paths). The choices can be interpreted as guesses in a search process.

A large number of problems can be conceptualized through nondeterministic algorithms, including the most famous unresolved question in computing theory,P vs NP.

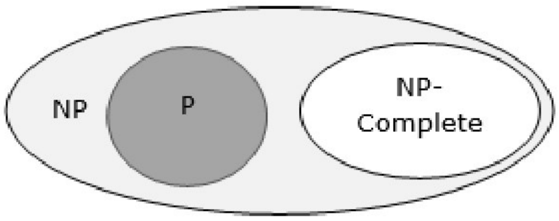
Check your progress

Q1. Define the basic concept of NP hard harder than NP complete with suitable example.

Q2. Why do we need non deterministic algorithm?

9.3NP - Hard and NP Complete classes

A problem is in the class NPC if it is in NP and is as hard as any problem in NP. A problem is NP-hard if all problems in NP are polynomial time reducible to it, even though it may not be in NP itself.



If a polynomial time algorithm exists for any of these problems, all problems in NP would be polynomial time solvable. These problems are called NP-complete. The phenomenon of NP-completeness is important for both theoretical and practical reasons.

Definition of NP-Completeness

A language B is NP-complete if it satisfies two conditions

- ☐ B is in NP
- ☐ Every A in NP is polynomial time reducible to B.

If a language satisfies the second property, but not necessarily the first one, the language B is known as NP-Hard. Informally, a search problem B is NP-Hard if there exists some NP-Complete Problem A that Turing reduces to B.

The problem in NP-Hard cannot be solved in polynomial time, until P = NP. If a problem is proved to be NPC, there is no need to waste time on trying to find an efficient algorithm for it. Instead, we can focus on design approximation algorithm.

NP-Complete Problems

Following are some NP-Complete problems, for which no polynomial time algorithm is known.

- ☐ Determining whether a graph has a Hamiltonian cycle
- ☐ Determining whether a Boolean formula iscapable of being satisfied, etc.

NP-Hard Problems

The following problems are NP-Hard

- ☐ The circuit-satisfiability problem
- ☐ Set Cover
- ☐ Vertex Cover
- ☐ Travelling Salesman Problem

In this context, now we will discuss TSP is NP-Complete

TSP is NP-Complete

The traveling salesman problem consists of a salesman and a set of cities. The salesman has to visit each one of the cities starting from a certain one and returning to the same city. The challenge of the problem is that the traveling salesman wants to minimize the total length of the trip.

Proof

To prove TSP is NP-Complete, first we have to prove that TSP belongs to NP. In TSP, we find a tour and check that the tour contains each vertex once. Then the total cost of the edges of the tour is calculated. Finally, we check if the cost is minimum. This can be completed in polynomial time. Thus TSP belongs to NP.

Secondly, we have to prove that TSP is NP-hard. To prove this, one way is to show that Hamiltonian cycle $\leq_p$ TSP (as we know that the Hamiltonian cycle problem is NP-Complete).

Assume G = (V, E) to be an instance of Hamiltonian cycle.

Hence, an instance of TSP is constructed. We create the complete graph G' = (V, E'), where

$$E=\{(i,j): i,j \in V \text{ and } i \neq j$$

Thus, the cost function is defined as follows –

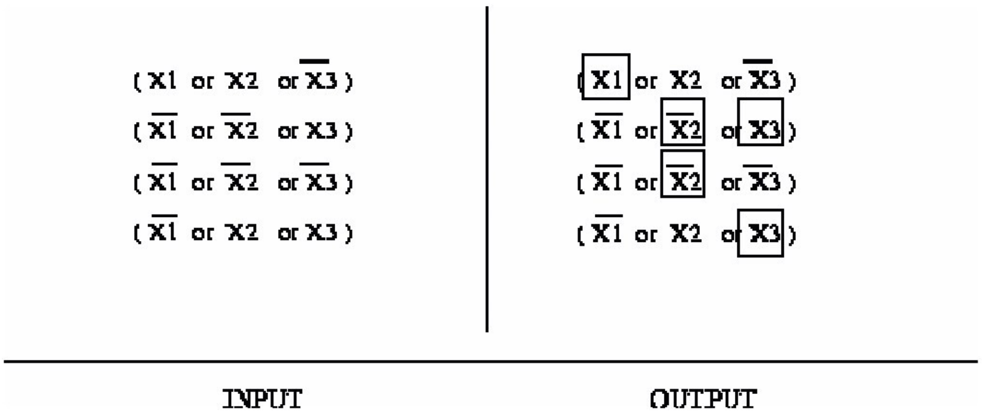
$$t(i,j) = \begin{cases} 0 & if(i,j) \in E \\ 1 & Otherwise \end{cases}$$

Now, suppose that a Hamiltonian cycle h exists in G. It is clear that the cost of each edge in h is 0 in G' as each edge belongs to E. Therefore, h has a cost of 0 in G'. Thus, if graph G has a Hamiltonian cycle, then graph G' has a tour of 0 cost.

Conversely, we assume that G' has a tour h' of cost at most 0. The cost of edges in E' are 0 and 1 by definition. Hence, each edge must have a cost of 0 as the cost of h' is 0. We therefore conclude that h' contains only edges in E.

We have thus proven that G has a Hamiltonian cycle, if and only if G' has a tour of cost at most 0. TSP is NP-complete.

9.4 Satisfiability


(X1 or X2 or $\overline{X3}$)
($\overline{X1}$ or $\overline{X2}$ or X3)
($\overline{X1}$ or $\overline{X2}$ or $\overline{X3}$)
($\overline{X1}$ or X2 or X3)
($\boxed{X1}$ or X2 or $\overline{X3}$)
($\overline{X1}$ or $\boxed{\overline{X2}}$ or $\boxed{X3}$)
($\overline{X1}$ or $\boxed{\overline{X2}}$ or $\overline{X3}$)
($\overline{X1}$ or X2 or $\boxed{X3}$)
INPUT
OUTPUT

Input description: A set of clauses in conjunctive normal form.

Problem description: Is there a truth assignment to the Boolean variables such that every clause is simultaneously satisfied?

Discussion: Satisfiability arises whenever we seek a configuration or object that must be consistent with (i.e. satisfy) a given set of constraints. For example, consider the problem of drawing name labels for cities on a map. For the labels to be legible, we do not want the labels to overlap, but in a densely populated region many labels need to be drawn in a small space. How can we avoid collisions?

For each of the n cities, suppose we identify two possible places to position its label, say right above or right below each city. We can represent this choice by a Boolean variable v_i, which will be true if city c_i's label is above c_i, otherwise v_i=false. Certain pairs of labels may be forbidden, such as when c_i's above label would obscure c_j's below label. This pairing can be forbidden by the two-element clause (v_i or v_j), where$\overline{v}$means "not v". Finding a satisfying truth assignment for the resulting set of clauses yields a mutually legible map labelling if one exists.

Satisfiability is the original NP-complete problem. Despite its applications to constraint satisfaction, logic, and automatic theorem proving, it is perhaps most important theoretically as the root problem from which all other NP-completeness proofs originate.

Is your formula in CNF or DNF? - In satisfiability, the constraints are specified as a logical formula. There are two primary ways of expressing logical formulas, conjunctive normal form (CNF) and disjunctive normal form (DNF). In CNF formulas, we must satisfy all clauses, where each clause is constructed by and-ing or's of literals together, such as

$$(v_1 \; or \; \overline{v_2}) and \; (v_2 \; or \; v_3)$$

In DNF formulas, we must satisfy any one clause, where each clause is constructed by or-ing ands of literals together. The formula above can be written in DNF as

$$(\overline{v_1} \; and \; \overline{v_2} \; and \; v_3) or \; (\overline{v_1} \; and \; v_2 and \; \overline{v_3})$$

$$or$$

$$(\overline{v_1} \; and \; v_2 \; and \; v_3) \; or \; (v_1 \; and \; \overline{v_2} \; and \; v_3)$$

Solving DNF-satisfiability is trivial, since any DNF formula can be satisfied unless every clause contains both a literal and its complement (negation). However, CNF-satisfiability is NP-complete. This seems paradoxical, since we can use De Morgan's laws to convert CNF-formulae into equivalent DNF-formulae and vice versa. The catch is that an exponential number of terms might be constructed in the course of translation, so that the translation itself might not run in polynomial time.

How big are your clauses? - k-SAT is a special case of satisfiability when each clause contains at most k literals. The problem of 1-SAT is trivial, since we must set true any literal appearing in any clause. The problem of 2-SAT is not trivial, but it can still be solved in linear time. This is very interesting, because many problems can be modelled as 2-SAT using a little cleverness. Observe that the map labelling problem described above is an instance of 2-SAT and hence can be solved in time linear in the number of clauses, which might be quadratic in the number of variables.

The good times end as soon as clauses contain three literals each, i.e. 3-SAT, for 3-SAT is NP-complete. Thus in general it will not be helpful to model a problem as satisfiability unless we can do it with two-element clauses.

Does it suffice to satisfy most of the clauses? - Given an instance of general satisfiability, there is not much you can do to solve it except by backtracking algorithms such as the Davis-Putnam procedure. In the worst case, there are 2^m truth assignments to be tested, but fortunately, there are lots of ways to prune the search. Although satisfiability is NP-complete, how hard it is in practice depends on how the instances are generated. Naturally defined "random" instances are often surprisingly easy to solve, and in fact it is nontrivial to generate instances of the problem that are truly hard.

Still, we would likely benefit by relaxing the problem so that the goal is to satisfy as many clauses as possible. Here optimization techniques such as simulated annealing can be put to work to refine random or heuristic solutions. Indeed, any random truth assignment to the variables will satisfy any particular k-SAT clause with probability$\mathbf{1 - (1 - 1/2)^{K}}$, so our first attempt is likely to satisfy most of the clauses. Finishing off the job is the hard part. Finding an assignment that satisfies the maximum number of clauses is NP-complete even for non-capable of being satisfied instances.

9.5 Reducibility

In computability theory and computational complexity theory, a reduction is an algorithm for transforming one problem into another problem. A sufficiently efficient reduction from one problem to another may be used to show that the second problem is at least as difficult as the first.

Intuitively, problem A is reducible to problem B if an algorithm for solving problem B efficiently (if it existed) could also be used as a subroutine to solve problem A efficiently. When this is true, solving A cannot be harder than solving B. "Harder" means a higher estimate of the computing resources required in a given context (for example, compared to single-threaded solutions, higher time complexity, higher memory requirements, Expensive requirements for hardware processor cores, etc.).The existence of a reduction from A to B can be written in the shorthand notation A ≤m B, usually with a subscript on the ≤ to indicate the type of reduction being used (m: mapping reduction, p: polynomial reduction).

The mathematical structure generated on a set of problems by the reductions of a particular type generally forms a pre-order, whose equivalence classes may be used to define degrees of un-solvability and complexity classes.

There are two main situations where we need to use reductions:

- ☐ First, we find ourselves trying to solve a problem that is similar to a problem we've already solved. In these cases, often a quick way of solving the new problem is to transform each instance of the new problem into instances of the old problem, solve these using our existing solution, and then use these to obtain our final solution. This is perhaps the most obvious use of reductions.
- ☐ Second: suppose we have a problem that we've proven is hard to solve, and we have a similar new problem. We might suspect that it is also hard to solve. We argue by contradiction: suppose the new problem is easy to solve. Then, if we can show that every instance of the old problem can be solved easily by transforming it into instances of the new problem and solving those, we have a contradiction. This establishes that the new problem is also hard.

A very simple example of a reduction is from multiplication to squaring. Suppose all we know how to do is to add, subtract, take squares, and divide by two. We can use this knowledge, combined with the following formula, to obtain the product of any two numbers:

$$x * y = \frac{((x+y)^2 - x^2 - y^2)}{2}$$

We also have a reduction in the other direction; obviously, if we can multiply two numbers, we can square a number. This seems to imply that these two problems are equally hard. This kind of reduction corresponds to Turing reduction.

However, the reduction becomes much harder if we add the restriction that we can only use the squaring function one time, and only at the end.In this case, even if we allow all basic arithmetic operations, including multiplication, it will generally not be reduced, because in order to get the desired square result, we must first calculate its square root, and this square root can be an irrational number. For example, √2, you cannot use rational numbers for arithmetic operations. Going in the other direction, however, we can certainly square a number with just one multiplication, only at the end. Using this limited form of reduction, we have shown the unsurprising result that multiplication is harder in general than squaring. This corresponds to many-one reduction.

Properties

A reduction is a pre-ordering, that is a reflexive and transitive relation, on P(N)×P(N), where P(N) is the power set of the natural numbers.

Types and applications of reductions

As described in the example above, there are two main types of reductions used in computational complexity, the many-one reduction and the Turing reduction. Many-one reductions map instances of one problem to instances of another; Turing reductions compute the solution to one problem, assuming the other problem is easy to solve. The many-one reduction is a stronger type of Turing reduction, and is more effective at separating problems into distinct complexity classes. However, the increased restrictions on many-one reductions make them more difficult to find.

A problem is complete for a complexity class if every problem in the class reduces to that problem, and it is also in the class itself. In this sense the problem represents the class, since any solution to it can, in combination with the reductions, be used to solve every problem in the class.

However, in order to be useful, reductions must be easy. For example, by letting the reduction machine solve the problem in exponential time and generate zeros only in the following cases, it is possible to reduce difficult NP-complete problems (such as Boolean satisfiability problems) to trivial problems (such as determining whether a number is zero) only if There is a solution. However, this does not achieve much, because even though we can solve the new problem, performing the reduction is just as hard as solving the old problem. Likewise, a reduction computing a non-computable function can reduce an undecidable problem to a decidable one. As Michael Sipser, an American theoretical computer scientistpoints out in Introduction to the Theory of Computation: "The reduction must be easy, relative to the complexity of typical problems in the class [...] If the reduction itself were difficult to compute, an easy solution to the complete problem wouldn't necessarily yield an easy solution to the problems reducing to it."

Therefore, the appropriate notion of reduction depends on the complexity class being studied. When studying the complexity class NP and harder classes such as the polynomial hierarchy, polynomial-time reductions are used. When studying classes within P such as NC and NL, log-space reductions are used. Reductions are also used in computability theory to show whether problems are or are not solvable by machines at all; in this case, reductions are restricted only to computable functions.

In case of optimization (maximization or minimization) problems, we often think in terms of approximation-preserving reduction. Suppose we have two optimization problems, such that an instance of a problem can be mapped to an instance of another problem, so that the near optimal solution of the last problem instance can be converted back to the near optimal solution of the first problem. This way, if we have an optimization algorithm (or approximation algorithm) that finds near-optimal (or optimal) solutions to instances of problem B, and an efficient approximation-preserving reduction from problem A to problem B, by composition we obtain an optimization algorithm that yields near-optimal solutions to instances of problem A. Approximation preserve reduction is generally used to test the roughness of the approximate result: if for some α, some optimization problem A is difficult to approximate within a factor better than α (under certain complexity assumptions), and there is a problem, problem A to problem B The preservation of approximation β decreases, and we can conclude that problem B is difficult to approximate within the factor α / β.

Examples

- ☐ To show that a decision problem P is undecidable we must find a reduction from a decision problem which is already known to be undecidable to P. That reduction function must be a computable function. In particular, we often show that a problem P is undecidable by showing that the halting problem reduces to P.
- ☐ The complexity classes P, NP and PSPACE are closed under (many-one, "Karp") polynomial-time reductions.
- ☐ The complexity classes L, NL, P, NP and PSPACE are closed under log-space reduction.

9.6 Summary

In this unit you have learnt about the basic concept of the NP-Hard and NP-complete, non-deterministic algorithms. You have also studied about NP - Hard and NP Complete classes in detail. After this you have learnt about satisfiability problem and reducibility.

- ☐ A problem is NP-hard if all problems in NP are polynomial time reducible to it, even though it may not be in NP itself. ... If a polynomial time algorithm exists for any of these problems, all problems in NP would be polynomial time solvable. These problems are called NP-complete.
- ☐ In computer science, a nondeterministic algorithm is an algorithm that, even for the same input, can exhibit different behaviors on different runs, as opposed to a deterministic algorithm. There are several ways an algorithm may behave differently from run to run.
- ☐ In logic and computer science, the Boolean satisfiability problem (sometimes called propositional satisfiability problem and abbreviated SATISFIABILITY, SAT or B-SAT) is the problem of determining if there exists an interpretation that satisfies a given Boolean formula.
- ☐ Reducibility- If we can convert one instance of a problem A into problem B (NP problem) then it means that A is reducible to B. NP-

hard-- Now suppose we found that A is reducible to B, then it means that B is at least as hard as A.

9.7 Review Questions

Q1. Differentiate between NP-Hard and NP-Complete Problems with suitable example.

Q2. If a sub-problem (which is a part of a main problem) is NP-Complete, does it mean that the main problem will be NP-Complete as well?

Q3. Is K means an example of a deterministic algorithm? Elaborate your answer with suitable example.

Q4. Is the satisfiability problem known to be in NP, or only conjectured to be in NP? Explain with example.

Q5. What does Reducibility mean in NP problems and why is it required?

Notes

Notes