

Structure

1.0	Introduction	1
1.1	Objectives	3
1.2	Some examples to understand Greedy Techniques	3
1.3	Formalization of Greedy Techniques	4
1.4	An overview of Local and Global Optima	6
1.5	Fractional Knapsack Problem	8
1.6	A Task Scheduling Algorithm	13
1.7	Huffman Codes	17
1.8	Summary	22
1.9	Answer to Check Your progress	23
1.10	Further Reading	26

1.0 Introduction

In Greedy algorithm, the solution that looks best at a moment is selected with the hope that it will lead to the optimal solution. This is one of the approaches to solve optimization problems. What is an optimization problem? An optimization problem is a problem which needs the best (or optimal) solution among all the possible solutions of a problem. For example, suppose we are planning to travel from location 'A' to location 'B' and there are different modes like bike, car, train, or plane, to complete this journey. The problem is to determine which mode is the best to complete this journey. So, this is an optimization problem as we want the best mode among all the possible modes to complete this journey and the best mode represents the optimal solution for this problem. Let say, if we need to cover this journey with minimum cost, then this optimization problem will be termed as a minimization problem. Further, we need to complete this journey within 12 hours due to some urgency. This will act as a constraint for our problem. So, the optimization problem can be framed as a minimization problem in which an optimal solution is the mode which incurs minimum cost and does not take more than 12 hours to complete this journey. Suppose, it is given that the time taken to complete this journey on bike and car is more than 12 hours while it takes less than 12 hours on a train and a plane. Thus, train and plane are the possible modes (or solutions) to complete this journey. Therefore, these both solutions are termed as feasible solutions as they are satisfying the constraint of our problem. Formally, a solution which satisfies all constraints of the problem is termed as feasible solution. A solution which is best among all feasible solutions is termed as optimal solution. Assuming train incurs the minimum cost to complete this journey, then train will be the optimal solution for our problem. In case of maximization optimization problem, the optimal solution to the problem will be a maximum solution. One more example to understand greedy approach, consider the scenario of a job recruitment process. During

job interview, there are different rounds like, written, technical, and HR, to select the best candidate. Now, one possible approach to do so is to allow every candidate to appear in each round and select the best candidate. However, this approach would be time consuming. Another possible approach would be to filter out some candidates at each round based on some criteria and then, select the one who clears the last round. This would be less time consuming. So, the idea behind greedy approach is to select the currently best solutions among the given set of solutions based on some criteria with the hope that it will lead to overall best solution.

However, the solution returned by greedy approach may not be always optimal and depends upon the criteria. Moreover, greedy approach may not be appropriate for some optimization problems where other approaches can be used for solving them like, Dynamic programming and Branch and Bound. Therefore, if a problem requires an optimal solution which can be either minimum or maximum, then that problem is termed as optimization problem. In an optimization problem, problem definition is termed as cost function ($f(x)$) and the maximization/minimization of the cost function corresponds to the objective function of the optimization problem. For example, the general mathematical formulation of a maximization optimization problem is defined as follows:

$$\text{Maximize}_{\{x \in S\}} f(x)$$

$$\text{such that: } a_i(x) \geq 0,$$

$$b_j(x) = 0, \text{ and}$$

$$c_k(x) \leq 0.$$

where, x is the considered solution from the set of solutions (S) while $a_i(x)$, $b_j(x)$, and $c_k(x)$ correspond to the possible set of constraints on an optimization problem. For solving such problems, greedy approach is widely popular and applicable.

Generally an optimization problem has n inputs (call this set as **input domain** or **Candidate set**, A), we are required to obtain a subset of A (call it **solution set**, S) that satisfies the given constraints or conditions. Any subset S , which satisfies the given constraints, is called a **feasible** solution. We need to find a feasible solution that maximizes or minimizes a given objective function. The feasible solution that does this is called an **optimal solution**.

A greedy algorithm proceeds step-by-step, by considering one input at a time. At each stage, the decision is made regarding whether a particular input (say x) chosen gives an optimal solution or not. Our choice of selecting input x is being guided by the selection function (say **select**). If the inclusion of x gives an optimal solution, then this input x is added into the partial solution set. On the other hand, if the inclusion of that input x results in an infeasible solution, then this input x is not added to the partial solution. The input we tried and rejected is never considered again. When a greedy algorithm works correctly,

the first solution found in this way is always optimal. In brief, at each stage, the following activities are performed in greedy method:

1. First we select an element, say, from input domain A.
2. Then we check whether the solution set S is feasible or not. That is we check whether x can be included into the solution set S or not. If yes, then solution set. If no, then this input x is discarded and not added to the partial solution set S. Initially S is set to empty.
3. Continue until S is filled up (i.e. optimal solution found) or A is exhausted whichever is earlier.

(Note: From the set of feasible solutions, particular solution that satisfies or nearly satisfies the objective of the function (either maximize or minimize, as the case may be), is called **optimal solution**.)

In this unit, we will discuss those problems for which greedy algorithm gives an optimal solution such as Knapsack problem, Minimum cost spanning tree (MCST) problem and Single source shortest path problem.]

1.1 Objective

After going through this unit, you will be able to :

- Formulate a problem as an optimization problem.
- Define the basic concept of greedy technique.
- Write a general form for greedy technique.
- Formulate fractional Knapsack, task scheduling algorithm and Huffman code problems
- Write algorithms for fractional Knapsack problem. Task scheduling and Huffman code.
- Calculate time complexities of algorithms.

1.2 Some examples to understand Greedy Techniques

Example 1: Suppose there is a list of tasks along with the time taken by each task. However, you are given with limited time only. The problem is which set of tasks will you be doing so that you can complete maximum number of tasks in the given amount of time.

Solution: The intuitive approach would be greedy approach and the task selection criteria would be to select the task with the slowest amount of time. So, the first task will be that task which takes minimum time. The next task would be the one that takes minimum time among the remaining set of tasks and so on.

Example 2: Consider a bus which can travel up to 40 kilometers (Km) with full tank. We need to travel from location 'A' to location 'B' which has distance of 95 Km as depicted in Figure 1.

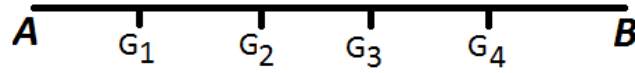


Figure 1: Representing the scenario of Example 2.

In between 'A' and 'B', there are four gas stations, G_1 , G_2 , G_3 , and G_4 , which are at distance of 20 KM, 37.5 KM, 55 KM, and 75 KM, from location 'A' respectively. The problem is to determine the minimum number of refills needed to reach the location 'B' from location 'A'.

Solution: Suppose, the tank refill decision criteria is considered to refill at the gas station which is nearest when the tank is about to get empty. According to the criteria, if we start from location 'A', the tank will be refilled at the second gas station (G_2) as it is 37.5 KM from 'A'. After this, we need to refill at the fourth gas station (G_4) which is at a distance of 37.5 KM from G_2 . From G_4 , we can easily reach the location 'B' as it is 20 KM. Therefore, the number of refills required is two as represented in the Figure 2.

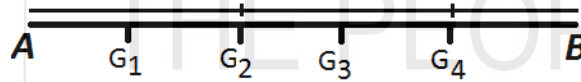


Figure 2: Solution to the scenario of Example 2. Here, the marks over G_2 and G_4 represent the refilling gas stations

1.3 Formalization of Greedy Technique

For a given problem with 'n' input values, the greedy approach will decide criteria to select values, termed as selection criteria, and will run for 'n' times. In each run, following steps will be performed:

1. It will perform selection based upon the selection criteria. This returns a value from the considered input values and also removes the selected value from the input values.
2. It will check the feasibility of the selected value.
3. If the solution is feasible then the selected value is added to the solution set. Else step 1 is repeated.

Based on the above discussion, we develop a template for writing a greedy algorithm

Greedy_Algo_Template(A, n)

/* **Input:** (a) A input domain (or Candidate set) A of size n, from which solution is to be obtained. */

(b) Criteria to select input values

/* function **select** (A: candidate_set) return an element (or candidate). */

/* function **solution** (S: solution_set) return Boolean */

/* function **feasible** (S: solution_set) return Boolean */

/* **Output:** An optimal solution set S, where S, which maximize or minimize the selection criteria w. r. t. given constraints */

```
{
    S ← {}          /*Initially a solution set S is empty */
    While ( not solution(S) and A ≠  $\phi$  )
    {
        x ← select (A)      /* A “best” element x is selected from A which
                               maximize or minimize the selection
                               criteria. */
        A ← A – {x}          /* once x is selected , it is removed from A */
        if ( feasible (S U {x}) then /* x is now checked for feasibility */
            S ← S U {x}
        }
        If (solution (S)) /* if a solution is found */
            return S;
        else
            return “ No Solution”
    } /* end of while
```

Example- 3: To illustrate the same, consider a list of jobs as given in Table 1:

Table 1: List of the jobs.

Jobs (J)	J1	J2	J3	J4	J5
Time Required(T)	3 (T1)	7 (T2)	1 (T3)	2 (T4)	8 (T5)

In this problem, second row represents the time taken to accomplish the corresponding job. The problem is to count maximum number of jobs that are possible in the given time limit, i.e., $T = 6$.

Solution:

Input:

- a. $A = \{J1, J2, J3, J4, J5\}$ and $n = 5$.
- b. Criteria: Select the task that has minimum time.

Step1: Select the task 'J3' as it has minimum time of 1.

Step2: This solution is feasible according to the problem definition

Step3: Add this solution to the set S ; $S = \{J3\}$. Remaining time limit = 5.

Step4: Removing this task(i.e J3) from 'A'. Now' $A = \{J1, J2, J4, J5\}$.

Step5: Select the task 'J4' as it has minimum time of 2.

Step6: This solution is feasible according to the problem definition.

Step7: Add this solution to the set S ; $S = \{J3, J4\}$. Remaining time limit = 3.

Step8: Remove this task from 'A'. Now' $A = \{J1, J2, J5\}$.

Step9: Select the task 'J1' as it has minimum time of 3.

Step10: This solution is feasible according to the problem definition

Step11: Add this solution to the set S ; $S = \{J3, J4, J1\}$. Remaining time limit = 0.

Step12: Removing this task from 'A'. Now', $A = \{J2, J5\}$.

Step13: Select the task 'J2' as it has minimum time of 7.

Step14: This solution is infeasible according to the problem as remaining time is 0.

Step15: Removing this task from 'A'. Now, $A = \{J5\}$.

Step16: Select the task 'J5' as it has minimum time of 8.

Step17: This solution is infeasible according to the problem as remaining time is 0.

Step18: Removing this task from 'A'. Now', $A = \{\}$.

Therefore, maximum number of jobs that can be done within the given time limit is three.

1.4 An overview of local and global optima

Local optima or local optimal solutions correspond to the set of all the feasible solutions that are best locally for an optimization problem. Global optimal solution corresponds to the best solution of the optimization problem. In an optimization problem, there may be many local optima but there is only one global optimal solution. To understand these concepts, consider the 2D- plot illustrated in Figure 3 which represents the solution space of some function for the minimization problem. Here, x-axis represents the range of x variable while y-axis corresponds to the different feasible solutions attained by the considered function $f(x)$ over different values of x . This type of curve is generally termed as a non-convex curve. Here, the global optimal solution is only one as no other solution is better than this objective function value in terms of minimization. However, there are many local optimal solutions as they have

better (or minimum) objective function values than other solutions in the nearby regions.

Greedy Technique

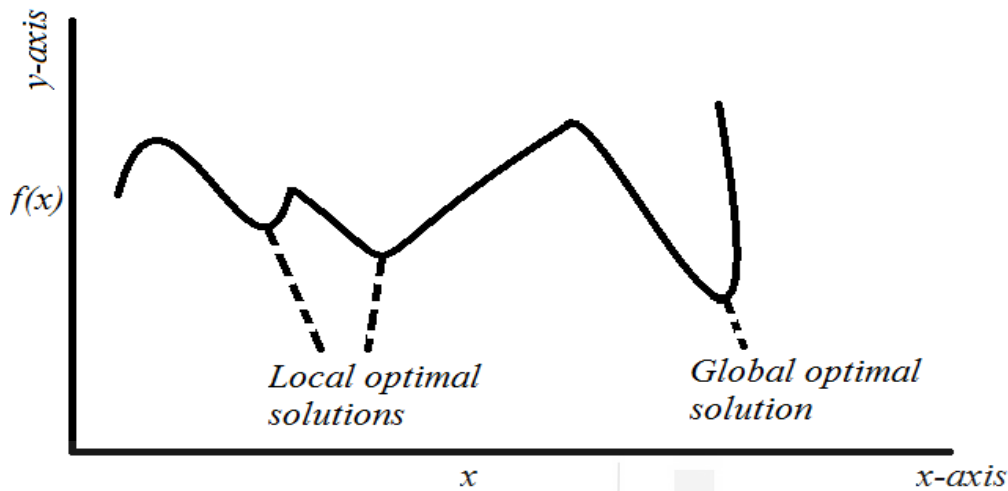


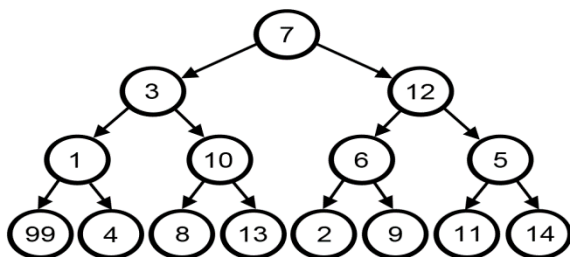
Figure 3: A solution space of some function for the minimization problem.

In Example 2, the solution corresponding to the number of refills as two is the global optimal solution, while other solution where the number of refills is four that solution corresponds to the local optimal solution. It is worthy to note that global optimal solution will not be equal to any local optima.

Check Your Progress-1

Question 1: What is the main step of greedy approach that effects the obtained solution?

Question 2: In the graph below, a greedy algorithm is trying to find the longest path through the graph (the number inside each node contributes to a total length). To do this, it selects the largest number at each step of the algorithm. With a quick visual inspection of the graph, it is clear that this algorithm will not arrive at the correct solution. What is the correct solution? Why is a greedy algorithm ill-suited for this problem?



1.5 Fractional Knapsack problem

This is an optimization problem which we want to solve it through greedy technique. In this problem, a Knapsack (or bag) of some capacity is considered which is to be filled with objects. We are given some objects with their weights and associated profits. The problem is to fill the given Knapsack with objects such that the sum of the profit associated with the objects that are included in the Knapsack is maximum. The constraint in this problem is that the sum of the weight of the objects that are included in the Knapsack should be less than or equal to the capacity of the Knapsack. However, the objects can be included in fractions to the Knapsack because of which this problem is termed as Fractional Knapsack problem. There is another kind of Knapsack problem, termed as 0-1 knapsack problem, in which the objects are not be considered in fraction i.e., the whole object is included in the Knapsack. That why, it is termed as 0-1 knapsack problem. Here we focus on a Fractional Knapsack Problem because Greedy technique works for fractional problem only.

Formulation of a problem

The fractional knapsack problem is defined as:

- Given a list of n objects say $\{O_1, O_2, \dots, O_n\}$ and a Knapsack (or bag).
- Capacity of Knapsack is W
- Each object O_i has a w_i weight and a profit of p_i
- If a fraction x_i (where $x_i \in \{0, \dots, 1\}$) of an object O_i is placed into a knapsack then a profit of $p_i x_i$ is earned.

The **problem** (or Objective) is to fill a knapsack (up to its maximum capacity M) which maximizes the total profit earned.

Mathematically:

Maximumize (the profit) $\sum_{i=1}^n p_i x_i$; subjected to the constraints

$$\sum_{i=1}^n w_i x_i \leq W \text{ and } x_i \in \{0, \dots, 1\}, 1 \leq i \leq n$$

Note that the value of x_i will be any value between 0 and 1 (inclusive). If any object O_i is completely placed into a knapsack then its value is 1 (i.e. $x_i = 1$), if we do not pick (or select) that object to fill into a knapsack then its value is 0 (i.e. $x_i = 0$). Otherwise if we take a fraction of any object then its value will be any value between 0 and 1.

To understand this problem, consider the following instance of a knapsack problem:

Greedy Technique

number of objects; $n = 3$

Capacity of Knapsack; $W = 20$

$(p_1, p_2, p_3) = (25, 24, 15)$

$(w_1, w_2, w_3) = (18, 15, 10)$

To solve this problem, Greedy method may apply any one of the following strategies:

- From the remaining objects, select the object with maximum profit that fit into the knapsack.
- From the remaining objects, select the object that has minimum weight and also fits into knapsack.
- From the remaining objects, select the object with maximum p_i/w_i that fits into the knapsack.

Let us apply all above 3 approaches on the given knapsack instance:

Approach	(x_1, x_2, x_3)	$\sum_{i=1}^3 w_i x_i$	$\sum_{i=1}^3 p_i x_i$
1	$(1, \frac{2}{15}, 0)$	$18+2+0=20$	28.2
2	$(0, \frac{2}{3}, 1)$	$0+10+10=20$	31.0
3	$(0, 1, \frac{1}{2})$	$0+15+5=20$	31.5

Approach 1: (selection of object in decreasing order of profit):

In this approach, we select those object first which has maximum profit, then next maximum profit and so on. Thus we select 1st object (since its profit is 25, which is maximum among all profits) first to fill into a knapsack, now after filling this object ($w_1=18$) (into knapsack remaining capacity is now 2 (i.e. $20-18=2$). Next we select the 2nd object, but its weight $w_2 = 15$, so we take a fraction of this object (i.e. $x_2 = \frac{2}{15}$). Now knapsack is full (i.e. $\sum_{i=1}^3 w_i x_i = 20$) so 3rd object is not selected. Hence we get total profit $\sum_{i=1}^3 p_i x_i = 28$ units and the solution set $(x_1, x_2, x_3) = (1, \frac{2}{15}, 0)$

Approach 2: (Selection of object in increasing order of weights).

In this approach, we select those object first which has minimum weight, then next minimum weight and so on. Thus we select objects in the sequence 2nd then 3rd then 1st. In this approach we have total profit $\sum_{i=1}^3 p_i x_i = 31.0$ units and the solution set $(x_1, x_2, x_3) = (0, \frac{2}{3}, 1)$.

Approach 3: (Selection of object in decreasing order of the ratio p_i/w_i). In this approach, we select those object first which has maximum value of p_i/w_i , that is we select those object first which has maximum profit per unit weight. Since $(p_1/w_1, p_2/w_2, p_3/w_3) = (1.3, 1.6, 1.5)$. Thus we select 2nd object first, then 3rd object then 1st object. In this approach we have total profit $\sum_{i=1}^3 p_i x_i = 31.5$ units and the solution set $(x_1, x_2, x_3) = (0, 1, \frac{1}{2})$.

Thus from above all 3 approaches, it may be noticed that

- Greedy approaches **do not always yield an optimal solution**. In such cases the greedy method is frequently the basis of a heuristic approach.
- Approach3 (Selection of object in decreasing order of the ratio p_i/w_i) gives a optimal solution for knapsack problem.

A pseudo-code for solving knapsack problem using greedy approach is:

Greedy Fractional-Knapsack ($p[1..n]$, $w[1..n]$, $X[1..n]$, W)

Input: a) List of weights and profits of n objects $\{O_1, O_2, \dots, O_n\}$ stored in $w[1..n]$ and $p[1..n]$ respectively.

b) Maximum capacity of the Knapsack.: W

Assume profits and weights are sorted by $\frac{p_i}{w_i}$

$X[1..n]$ is a solution set

Output: Optimal solution

```

{
1:   for i ← 1 to n do
2:       X[i] ← 0
3:       profit ← 0 /*Total profit of items packed in Knapsack
4:       weight ← 0 /* Total weight of items packed in Knapsack
5:       i ← 1
6:       while (weight < W) // W is the Knapsack Capacity
7:       {
8:           if (weight + w[i] ≤ W)
9:               X[i] = 1
10:              weight = weight + w[i]
11:           else

```

```

11:      X[i] = (W-weight)/w[i]
12:      weight = M
13:      profit = profit + p [i]*X[i]
14:
    } end of while
  } //end of Algorithm

```

Complexity for Fractional Knapsack problem using Greedy algorithm :

Sorting of n items (or objects) in decreasing order of the ratio p_i/w_i takes $O(n \log n)$ time. Since this is the lower bound for any comparison based sorting algorithm. Line 6 of **Greedy Fractional-Knapsack** takes $O(n)$ time. Therefore, the total time including sort is $O(n \log n)$.

Example: 1: Find an optimal solution for the knapsack instance $n=7$ and $M=15$,

$(p_1, p_2, \dots, p_7) = (10, 5, 15, 7, 6, 18, 3)$

$(w_1, w_2, \dots, w_7) = (2, 3, 5, 7, 1, 4, 1)$

Solution:

Greedy algorithm gives an optimal solution for knapsack problem if you select the object in decreasing order of the ratio p_i/w_i . That is we select those objects first which have the maximum value of the ratio p_i/w_i ; for all $i = 1, \dots, 7$. This ratio is also called profit per unit weight.

Since $\left(\frac{p_1}{w_1}, \frac{p_2}{w_2}, \dots, \frac{p_7}{w_7}\right) = (5, 1.67, 3, 1.6, 4, 5, 3)$. Thus we select 5th object first, then 1st object, then 3rd (or 7th) object, and so on.

Approach	$(x_1, x_2, x_3, x_4, x_5, x_6, x_7)$	$\sum_{i=1}^7 w_i x_i$	$\sum_{i=1}^7 p_i x_i$
Selection of object in decreasing order of the ratio p_i/w_i	$\left(1, \frac{2}{3}, 1, 0, 1, 1, 1\right)$	$1+2+4+5+1+2=15$	$6+10+18+15+3+3.33=55.33$

Example 2 -Suppose there is a knapsack of capacity 5 Kg, we have 7 items with weight and profit as mentioned in the following table:

Objects	1	2	3	4	5	6	7
Weight	2	3	5	7	1	4	1
Profit	10	5	15	7	6	18	3

The problem is to fill the given knapsack with objects such that profit is maximized. However, objects can be included in fractions. So, this is a maximization optimization problem with a constraint that the total weight of objects that are included in the Knapsack should be less than or equal to 5 Kg.

Solution: Greedy approach may provide multiple solutions to this problem depending upon the selection criteria for including objects in the Knapsack. One criteria may be to include all the objects in the Knapsack, but it will not return a feasible solution as the constraint would not be satisfied. The other criteria can be to consider objects according to associated profit i.e., high profit objects are considered first. Alternatively, we can define the criteria of considering objects with smaller weights irrespective of the associated profit as this will allow more objects to be included in Knapsack which will yield more profit. All discussed criteria are correct but will not return optimal solution. The appropriate criteria will be to include objects based on profit per weight as it will calculate profit by considering per unit of weight of the object. So, the selection criteria for filling the Knapsack will be to select the object with highest profit per weight. To apply this criteria, let us first calculate the profit per weight for each object which is mentioned in Table 3.

Table 3: Profit per weight for each object.

Objects	1	2	3	4	5	6	7
Weight	2	3	5	7	1	4	1
Profit	10	5	15	7	6	18	3
Profit per weight	5	1.3	3	1	6	4.5	3

According to highest profit per weight, object 5 is selected as weight of this object is less than the capacity of Knapsack. So, the remaining capacity of Knapsack is 4 Kg. Next, object 1 is included as available capacity of Knapsack is larger than the total weight of object 1. The Knapsack is left with the capacity of 2 Kg. Now, the next object with highest profit per weight is object 6. But the remaining capacity of Knapsack is 2 Kg. So, we will take a fraction of object 6, i.e. $\frac{2}{4}$ portion of the object 6, in the Knapsack. Now the capacity of Knapsack is 0 after the inclusion of 2 Kg of object 6. So, no further object will be included in the Knapsack.

The total weight of the included objects: $1 \times 2 + 0 \times 3 + 0 \times 5 + 0 \times 7 + 1 \times 1 + (2/4) \times 4 + 0 \times 1 = 5$

The total profit on included objects: $1 \times 10 + 0 \times 5 + 0 \times 15 + 0 \times 7 + 1 \times 6 + (2/4) \times 18 + 0 \times 3 = 24$

1.6 A Task Scheduling Algorithm

A task scheduling problem is formulated as an optimization problem in which we need to determine the set of tasks from the given tasks that can be accomplished within their deadlines along with their order of scheduling such that the profit is maximum. So, this is a maximization optimization problem with a constraint that tasks must be completed within their specified deadlines.

This problem is related to scheduling tasks on a single machine for their processing. Suppose you are given a set of n tasks for p on a single machine. Each task is assigned a profit p_i and a deadline d_i and takes only one unit of time on a machine to get completed. There is availability of a single machine for processing of all the tasks. A brute force approach would be to generate all subsets S of a given set of tasks, examine completion of each job in every subset by its deadline, calculate the profit among all feasible subsets and find out the feasible solution. To calculate S , i.e. the feasible solution p_i of each task i in S needs to be summed up i.e. $\sum_{i \in S} p_i$. There could be multiple subsets of tasks is S . An optimal solution for this problem is one which gives the maximum profit value. Trying out all possible permutation of tasks in S and selecting whether the tasks in S can be processed in any of these permutation without violating the deadline, will require n permutation of S , which is a time consuming.

The pseudo-code of the task scheduling problem is as follows:

- Input:**
- 1) List of t tasks $\{T_1, T_2, \dots, T_t\}$ and
 - 2) List of deadlines, defined with each task.
 - 3) List of Profit, associated with each task.

Output: Optimal solution

1. Sort the given set of tasks according to the profit associated with each task.
2. Make n unfilled time slots where n corresponds to the highest deadline.
3. Perform following steps for each task by considering tasks in decreasing order:
4. {
5. Find an empty time slot which is closest and not missing the deadline of the considered task.
6. Mark the identified slot as filled.

7. If no such slot is found, then ignore the task.
8. }

To understand the above pseudo-code, consider the following example.

Example 1 Let us consider an example to understand. Suppose, there are 5 tasks and each task has associated profit in rupees. The amount of time required to complete each task is one hour. However, the deadline of each task is given in specified in Table 4.

Table 4: Details of tasks which needs to be scheduled on a machine.

Task	T ₁	T ₂	T ₃	T ₄	T ₅
Profit	18	22	5	7	6
Deadline	2	2	3	1	3

Solution: To solve this problem, consider the highest deadline and prepare that many number of slots of unit time to schedule the tasks. This will help in scheduling the tasks easily as there will be no task to be completed after the highest deadline. According to the given problem, the highest deadline is 3. So, there will be three slots as illustrated in Figure 4, each slot of unit time.



Figure 4: List of slots to schedule a task.

Now, greedy approach has to select the set of three tasks and schedule them in such a way that the total profit is maximum on their completion. For this, the selection criteria will be to select the task with highest profit. According to this criteria, task 'T₂' is selected and place it in the slot closest to its deadline i.e., from 1 to 2. Next, task 'T₁' is selected from the remaining set of tasks according to selection criteria. The deadline for task 'T₁' is 2, but we have already scheduled 'T₂' in the slot 1-2. So, look for an empty slot before this slot. The 0-1 slot is empty. So, task 'T₁' is placed in slot 0-1. Now, the next task with highest profit is 'T₄'. The appropriate slot for 'T₄' is 0-1 according to its deadline. However, this slot is already filled and there is no other slot before 0-1. So, this task is not scheduled. Next, task 'T₅' is selected and placed in the 2-3 slot as it has deadline of 3. Now, the remaining task i.e., 'T₃', is having deadline as 3 but there is no empty slot available. So, this task is also discarded.

The schedule of given tasks is presented in Figure 5:

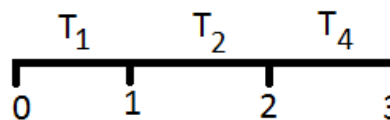


Figure 5: Schedule of selected tasks.

Therefore, the sequence of selected tasks is as follows: {T₁, T₂, T₄}

The total profit is: $18+22+7= 47$

Greedy Technique

Complexity for task scheduling problem using Greedy algorithm:

The time complexity of the task scheduling is $O(n^2)$, where n corresponds to the number of tasks to be scheduled.

Check Your Progress-2

Question 1: A thief enters a house for robbing it. He can carry a maximal weight of 60 kg into his bag. There are 5 items in the house with the following weights and values. What items should thief take if he can even take the fraction of any item with him?

Item	Weight	Value
1	5	30
2	10	40
3	15	45
4	22	77
5	25	90

Question 2: Let us consider that the capacity of the knapsack $W = 60$ and the list of provided items are shown in the following table –

Item	A	B	C	D
Profit	280	100	120	120
Weight	40	10	20	24
Ratio (p_i/w_i)	7	10	6	5

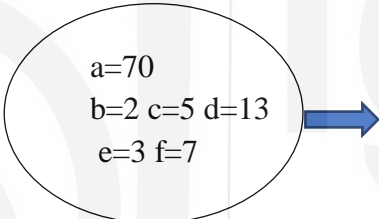
Find the optimal solution for gaining maximum profit.

1.7 Huffman Codes

Huffman coding is a greedy algorithm that is used to compress the data. Data can be sequence of characters and each character can occur multiple times in a data called as frequency of that character. Data transmitted through transmission media that can be digital communication or analog communication. That means we need to represent data in the form of bits. If there is more number of bits to represent the data, it will take more time to

transmit from one source to other source. Thus, Huffman compression algorithm is applied to represent the data in compressed form called as Huffman codes. Basically compression is a technique to reduce the size of the data. Huffman coding compress data by 70-80%. Huffman algorithm checks the frequency of each data, represent those data in the form of Huffman tree and build an optimal solution to represent each character as a binary string. Huffman coding also known as variable length coding.

Fixed Length Encoding: Suppose we have 100-characters data file, each character having different frequency shown in the Figure 1. An order of a character can be in any form. To transfer the data, each character will be encoded into ASCII representation without using any compression algorithm. ASCII code takes 7-bits to represent the data in a binary form. Therefore, in a message each character takes 7-bits to represents the data and there are 100 characters in the message. Thus, a total 700 bits needed to transfer a message from one source to other source without any compression algorithm as shown in a Figure 6. This is also called as fixed length encoding because each character has fixed length encoding of 7-bits



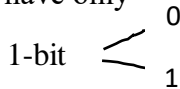
The diagram shows a circle containing the character frequencies: a=70, b=2, c=5, d=13, e=3, f=7. A blue arrow points from this circle to the right side of the table.

character	ASCII Code	Binary Form	Total bit/character
a	97	1100001	7*70= 490
b	98	1100010	7*2=14
c	99	1100011	7*5=35
d	100	1100100	7*13=91
e	101	1100101	7*3=21
f	102	1100110	7*7=49
Total =			700 bit

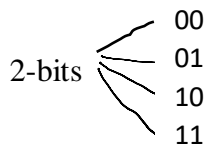
Figure 6: Shows the message and corresponding ASCII code binary representation

In case the above message is sent without applying any compression algorithm, a total number of bits it takes 700bit to transfer the message. In the above example we are not using all English alphabets and symbols. We are using only 6 alphabets in the message. Do we really need 7-bits to represent the data? Not really. only few bits are sufficient to represent the data. So can we use less bits to represent the data? Yes, then how many bits are needed to represent the data? As we know we can represent 128 characters with 7 bits. We have only 6 characters, we need definitely lesser than 7-bits, but how many bits are really needed?

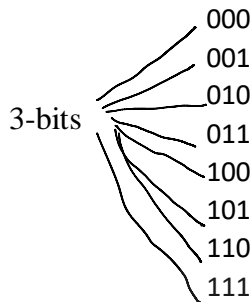
Let's see if we have only



Then only two character can be represented.



Then only four characters can be represented



Then only 8 characters can be represented. We have 6 characters in the message. We need only 6 out of 8 combination to represent the message. Therefore, we need only 3-bit to represent the message.

Table 1: Shows the message and corresponding 3-bit binary representation

character	Frequency	Code	Total bit/character
a	70	000	$3 \times 70 = 210$
b	3	001	$3 \times 2 = 6$
c	5	010	$3 \times 5 = 15$
d	13	011	$3 \times 13 = 39$
e	3	100	$3 \times 3 = 9$
f	7	101	$3 \times 7 = 21$
Total =			300 bits

The above method requires 3-bits only to encode each character of the message having 100 characters and take 300 bits to encode the entire message. Can we do better? Let us look at the appropriate algorithm.

Huffman Coding Algorithm: Variable Length Encoding

A variable-length coding can do considerably better than a fixed-length code using Huffman encoding. Huffman encoding says, we don't need to take fixed size code for each character. In a message some characters may be appearing less number of times and some may be appearing more number of times. So if you give a small size of code for the more frequent occurrence of characters then, the size of the entire message will be definitely reduced. Huffman invented a greedy algorithm that finds an optimal prefix code called a Huffman code. If there is an encoding of message at sender end, there must be decoding at receiver end to read the message. In Huffman coding prefix codes (which is one type of variable length encoding scheme) are assigned to each character in such a way that no code for one character constitute the beginning of another code. For example if 10 is a code for a then 101 cannot be

a code for b. Therefore, while decoding the encoded the message, Huffman coding ensures that there must be no ambiguity in prefix code assigned to each character. The main advantage of prefix code it that it simplifies decoding of the message.

Some more examples of Prefix Code: Set of binary sequence P, such that no sequence in P is a prefix of any other sequence in P.

- $P = \{01, 010, 10\}$ are binary sequences/code. In the code we can see that 01 is a pre-fix of 010 sequence. Therefore, this particular set is not a prefix code.
- $P1 = \{01, 100, 101\}$ are binary sequences. In the code we can see that 01 is not prefix of 100 and 101 sequence. Just remember that 01 is present in 101 sequence but it is not pre-fix, it is a postfix. Similarly 100 is not a pre-fix in any other code and 101 is not a pre-fix in any other code as well. Therefore, this particular set of sequences is a prefix code and each prefix code can be represented as a tree.

Huffman coding works in two steps:

1. Build a Huffman tree.
2. Find Huffman code for each character.

Steps for Building Huffman Tree:

1. Input is a set of M characters and each character $m \in M$ has a frequency m frequency
2. Store all the characters in min-priority queue using frequencies as their values (Priority queue is a data structure that allows the operation of search min (or max), insert, delete min (or max, respectively). If heap is used to implement priority queue, it will take $O(\log n)$ time.
3. Extract two minimum nodes x, y from min-priority queue PQ .
4. Replacing x, y in the queue with a new-node z representing their merger. The frequency of z is computed as sum of the frequencies of x and y . The node z has x as its left child and y as its right child.
5. Repeat steps 3 and 4 until one node left in the queue, which is the root of the Huffman tree.
6. Return the root of the tree.

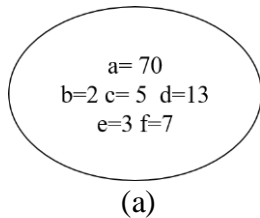
Steps to find Huffman code for each character.

1. Traverse the tree starting from the root node.
2. An edge connecting to the left child node is labeled as 0 and 1 if it is connecting to the right child node.

3. Traverse the complete tree through the left and right child based on the assigned value.
4. Prefix code/ Huffman code for a letter is the sequence of labels on the edge connecting the root to the leaf for that letter.

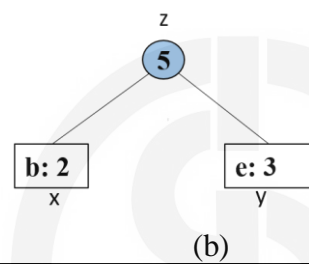
Greedy Technique

Let us understand the algorithm with an example:



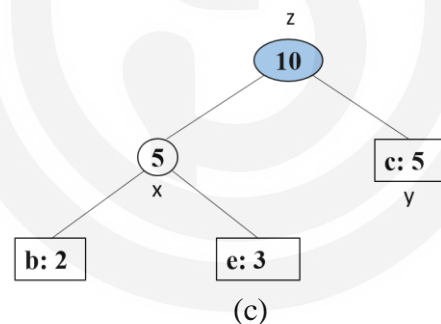
Build a min-priority queue based on the frequency. Initially PQ contains 6 nodes (6 letters) and each node represents the root of tree with single node and requires five merges to build a tree.

b: 2 e: 3 c: 5 f: 7 d: 13 a: 70



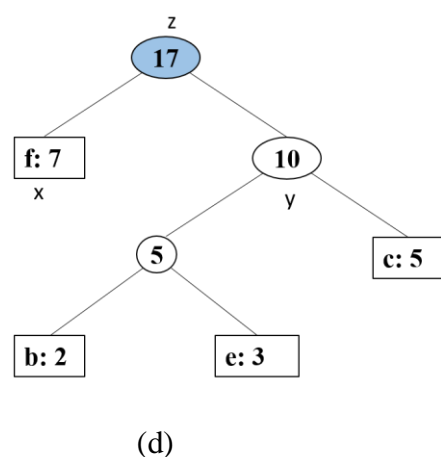
z: 5 c: 5 f: 7 d: 13 a: 70

Extract two minimum node x, y from PQ. Add internal z having with frequency $2+3 = 5$ in the PQ.



z: 10 f: 7 d: 13 a: 70

Extract two minimum node x, y from PQ. Add internal z having with frequency $5+5 = 10$ in the PQ. In this tree node $z=5$ and $c=5$ have same frequency. Same frequency character become the right child in the newly created node z.



z: 17 d: 13 a: 70

Extract two minimum node x, y from PQ. Add internal z having with frequency $7+10 = 17$ in the PQ.

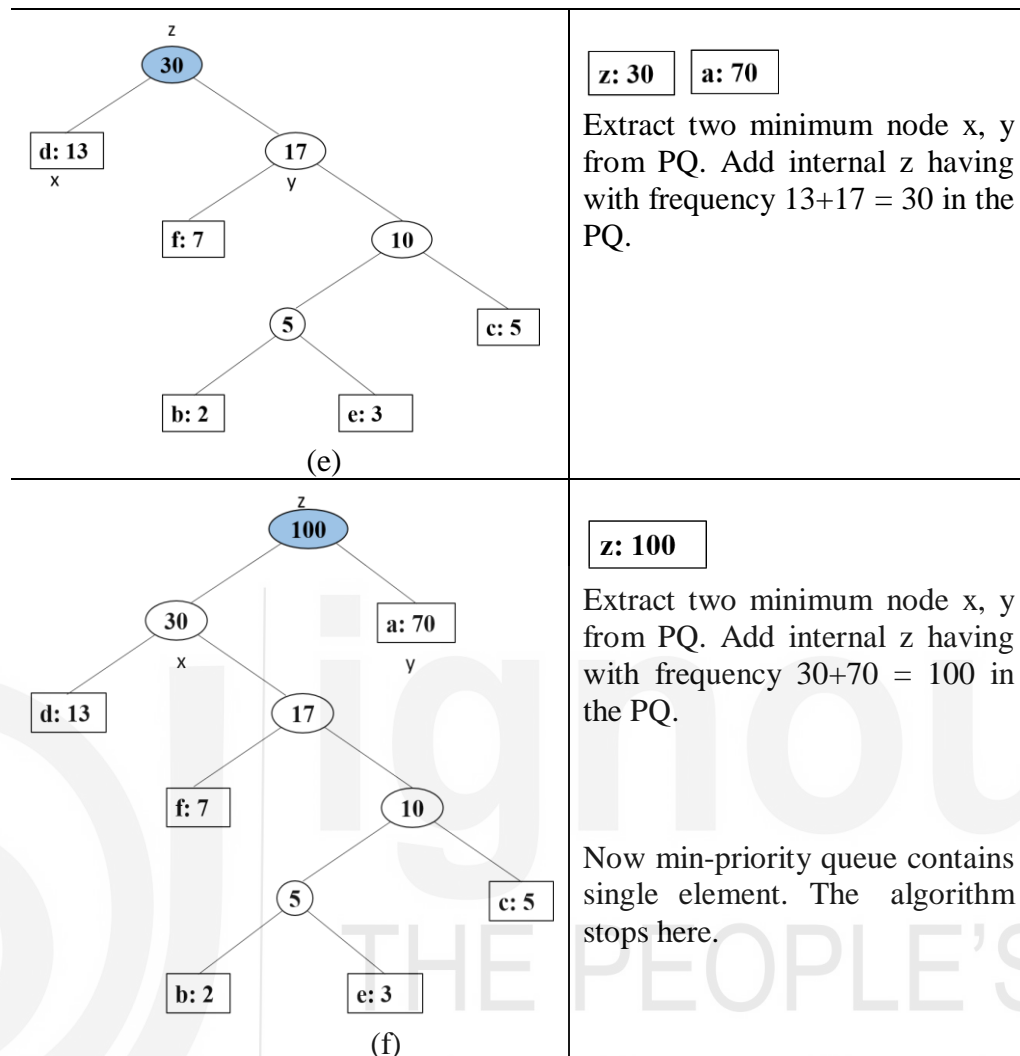


Figure 7: The steps of Huffman's algorithm for the frequencies given in (a). Each part shows the content of min-priority queue sorted into increasing order by frequency. At each step, the two trees with lowest frequencies are merged. Leaves are shown as a rectangle containing a character and its frequency. Internal nodes are shown as a circles containing the sum of the frequencies of their children. (b)-(e) intermediate stages. The final Huffman tree is shown in (f)

Character	Frequency	Code	Total bit/character
A	70	1	$1*70=70$
B	3	01100	$5*2=10$
C	5	0111	$4*5=20$
D	13	00	$2*13=26$
E	3	01101	$5*3=15$
F	7	010	$3*7=21$
Total bit =			162 bits

Table 2: The message and corresponding Huffman code

Now in Figure 7, traverse the tree starting from root node. Take an auxiliary array, while moving to the left child write 0 in the array, while moving to the right child write 1 in array. Print array when a leaf node is encountered. The Huffman code for a character is the sequence of labels on the edges connecting the root to the leaf node for that character.

In the table 2 we can see variable size codes are used. Some character have 1 bit code some have 2 bits codes and some characters have 5 bits codes. Thus, to encode the entire 100 character message we need only 162 bits and average 1.62 bits/character. These Huffman code can be used to encode and decode the message. We need to pass the entire tree or entire table for decoding of message.

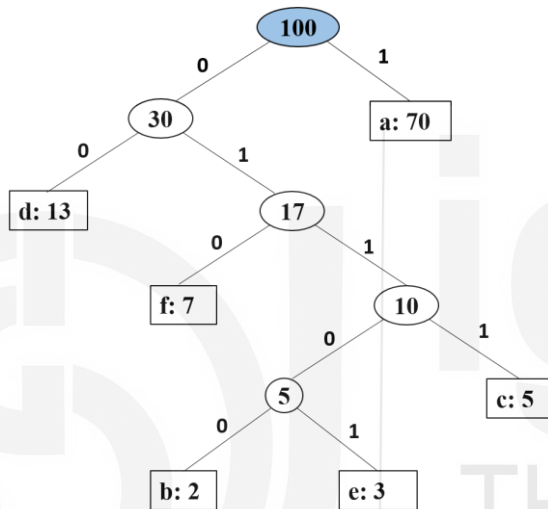


Figure 8: Huffman tree

Example

- Encode the message $M = \text{aabefdaac} \dots$

Corresponding encoding of message will be: 11011000110101000110111.....

a	a	b	e	f	d	a	a	c
1	1	01100	01101	010	00	1	1	0111

- Decode the message $M = 01100110111000110101101010$

Corresponding decoding of message will be: baacdeef

01100	1	1	0111	00	01101	01101	010
b	a	a	c	d	e	e	f

Time Complexity of Huffman Algorithm

Priority queue is implemented using binary-min heap. Building min heap procedure takes $O(n)$ time in step 2. Steps 3 and 4 run exactly $n-1$ times. Since in each step a new node z is added in the heap. When new node added it

take $O(\log n)$ time to heapify. Thus, total running time of Huffman algorithm on a set of n characters is $O(n \log n)$.

Check Your Progress-3

Question 1: What is an optimal Huffman code for the following set of frequencies, based on the first 8 Fibonacci numbers?

a:1 b:1 c:2 d:3 e:5 f:8 g:13 h:21

Can you generalize your answer to find the optimal code when the frequencies, based on the first n Fibonacci numbers?

Question 2: What is an optimal Huffman tree and Huffman code for the following set of frequencies?

M1: .45 M2: .18 M3: .002 M4: .11 M5: .24

Decode the encoding: 111011001101101111100

Question 3: What is an optimal Huffman tree and Huffman code for the following set of frequencies? Find out average number of bit required per character.

A:15 B:25 C:5 D:7 E:10 F:13 G:9

1.8 Summary

- A typically optimization problem is solved by applying Greedy approach.
- An optimization problem is a problem in which the best (or optimal) solution among all the possible solutions of a problem is needed which maximizes/minimizes the objective function under some constraints or conditions.
- For an optimization problem, the input domain defines the candidate set and the subset of candidate set which satisfy all the constraints is termed as feasible set. The feasible solution which is best in terms of maximizing/minimizing the given objective function is called as optimal solution.
- Local optimal solutions correspond to the set of all the feasible solutions that are best locally for an optimization problem. Global optimal solution corresponds to the best solution of the optimization problem.
- Greedy approach does not guarantee to return global optimal solution, but it does on many problems.
- In Fractional Knapsack problem, a Knapsack (or bag) of some capacity is to be filled with objects which can be included in fractions. Each object is given with a weight and associated profit. The problem is to

fill the given Knapsack with objects such that the sum of the profit associated with the objects that are included in the Knapsack is maximum. The constraint in this problem is that the sum of the weight of the objects in the Knapsack should be less than or equal to the capacity of the Knapsack. Greedy approach is a good approach to attain global solution for this problem.

- A task scheduling problem is an optimization problem in which a set of tasks is to be determined from the given tasks. Each task takes one unit of time for completion and has some associated profit. However, there is a deadline to complete every task. The objective is to accomplish a set of tasks within their deadlines such that the profit is maximum.
- Huffman coding is a greedy algorithm that is used to compress the data. As data is transmitted through transmission media, data is represented in the form of bits. If there is more number of bits to represent the data, it will take more time to transmit. Thus, apply Huffman compression algorithm to compress the data in form, termed as Huffman codes.

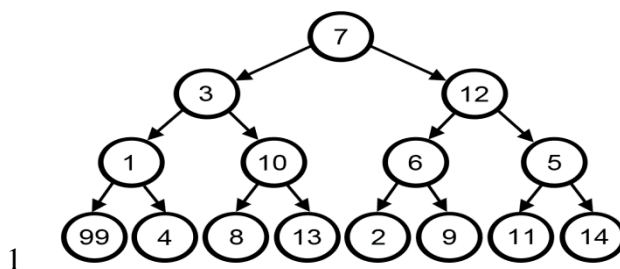
1.9 Answer to Check Your progress

Check Your Progress-1

Question 1: What is the main step of greedy approach that effects the obtained solution?

Solution: Selection criteria.

Question 2: In the graph below, a greedy algorithm is trying to find the longest path through the graph (the number inside each node contributes to a total length). To do this, it selects the largest number at each step of the algorithm. With a quick visual inspection of the graph, it is clear that this algorithm will not arrive at the correct solution. What is the correct solution? Why is a greedy algorithm ill-suited for this problem?



Solution: The correct solution for the longest path through the graph is 7, 3, 1, 99. This is clear because we can see that no other combination of nodes will come close to a sum of 99, so whatever path we choose, we know it should have 99 in the path. There is only one option that includes 99: 7, 3, 1, 99. However, greedy algorithm fails to solve this problem because it makes decisions purely based on what the best answer at the time is: at each step it did choose the largest number. However, since there could be

some huge number that the algorithm hasn't seen yet, it could end up selecting a path that does not include the huge number. The solutions to the sub problems for finding the largest sum or longest path do not necessarily appear in the solution to the total problem. The optimal substructure and greedy choice properties don't hold in this type of problem.

Check Your progress-2

Question 1: A thief enters a house for robbing it. He can carry a maximal weight of 60 kg into his bag. There are 5 items in the house with the following weights and values. What items should thief take if he can even take the fraction of any item with him?

Item	Weight	Value
1	5	30
2	10	40
3	15	45
4	22	77
5	25	90

Solution:

Step 1: Compute the value / weight ratio for each item-

Items	Weight	Value	Ratio
1	5	30	6
2	10	40	4
3	15	45	3
4	22	77	3.5
5	25	90	3.6

Step 2: Sort all the items in decreasing order of their value / weight ratio-

I1	I2	I5	I4	I3
(6)	(4)	(3.6)	(3.5)	(3)

Step 3: Start filling the knapsack by putting the items into it one by one.

Knapsack Weight	Items in Knapsack	Cost
60	\emptyset	0
55	I1	30
45	I1, I2	70
20	I1, I2, I5	160

Now,

- Knapsack weight left to be filled is 20 kg but item-4 has a weight of 22 kg.
- Since in fractional knapsack problem, even the fraction of any item can be taken.
- So, knapsack will contain the following items-
 $\langle I1, I2, I5, (20/22) I4 \rangle$

Total cost of the knapsack = $160 + (20/22) \times 77 = 160 + 57 = 217$ units

Question 2: Let us consider that the capacity of the knapsack $W = 60$ and the list of provided items are shown in the following table –

Item	A	B	C	D
Profit	280	100	120	120
Weight	40	10	20	24
Ratio (p_i/w_i)	7	10	6	5

Find the optimal solution for gaining maximum profit.

Solution: After sorting all the items according to p_i/w_i . First all of **B** is chosen as weight of **B** is less than the capacity of the knapsack. Next, item **A** is chosen, as the available capacity of the knapsack is greater than the weight of **A**. Now, **C** is chosen as the next item. However, the whole item cannot be chosen as the remaining capacity of the knapsack is less than the weight of **C**.

Hence, fraction of **C** (i.e. $(60 - 50)/20$) is chosen.

Now, the capacity of the Knapsack is equal to the selected items. Hence, no more item can be selected.

The total weight of the selected items is $10 + 40 + 20 * (10/20) = 60$

And the total profit is $100 + 280 + 120 * (10/20) = 380 + 60 = 440$

This is the optimal solution. We cannot gain more profit selecting any different combination of items.

Check Your Progress-3

Question 1: What is an optimal Huffman code for the following set of frequencies, based on the first 8 Fibonacci numbers?

a:1 b:1 c:2 d:3 e:5 f:8 g:13 h:21

Can you generalize your answer to find the optimal code when the frequencies, based on the first n Fibonacci numbers?

Solution: a:1111110; b:1111111; c:111110; d:11110; e:1110; f:110; g:10; h:0.

Yes, we can generalize for n numbers.

Question 2: What is an optimal Huffman tree and Huffman code for the following set of frequencies?

M1: .45 M2: .18 M3: .002 M4: .11 M5: .24

Decode the encoding: 111011001101101111100

Solution: M2M1M3M4M5M2M3

Question 3: What is an optimal Huffman tree and Huffman code for the following set of frequencies? Find out average number of bits required per character.

A:15 B:25 C:5 D:7 E:10 F:13 G:9

Solution: 2.67 bits/char

1.10 Further Readings

1. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, Introduction to Algorithms, MIT Press, 3rd Edition, 2009.

Unit 2 Divide and Conquer Technique

Divide & Conquer Technique

2.1	Introduction	1
2.2	Objectives	2
2.3	Recurrence Relation Formulation in Divide and Conquer Technique	2
2.4	Binary Search Algorithm	5
2.5	Sorting Algorithms	9
2.5.1	Merge Sort	9
2.5.2	Quick Sort	17
2.6	Integer Multiplication	25
2.7	Matrix Multiplication Algorithm	28
2.7.1	Straight forward method	28
2.7.2	Divide & Conquer Strategy for multiplication	29
2.8	Summary	36
2.9	Solution to Check Your Progress	37
2.10	Further Readings	38

2.0 Introduction

In Divide and conquer approach, the original problem is divided into two or more sub-problems recursively, till it is small enough to be solved easily. Each sub-problem is some fraction of the original problem. Next, the solutions of the sub-problems are combined together to generate the solution of the original problem. Figure 1 illustrates the working strategy of Divide and Conquer approach.

Many useful algorithms are recursive in nature such as Merge Sort, Quick Sort, Binary Search etc. All these algorithm will be examined in this unit. A large Integer multiplication and Strassen's matrix multiplication algorithm are also formulated as divide and conquer problems

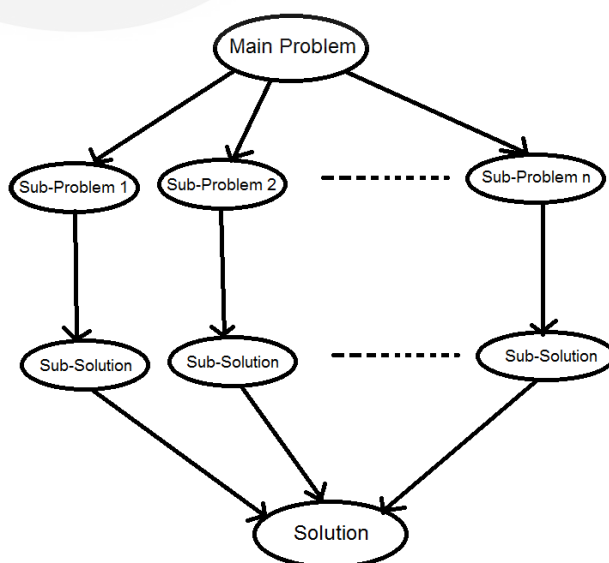


Figure 1: Working strategy of Divide and Conquer approach.

In Divide and Conquer approach, there are mainly three steps:

Step 1(Divide): The original problem is divided into one or more sub-problems in such a way that each sub-problem is equivalent to the original problem but its size is smaller than the original one. Further sub-division of each sub-problem is done till either it is directly solvable or it is impossible to perform sub-division which indicates that there is a direct solution of the sub-problem.

Step 2(Conquer): In this step, each smallest sub-problem is solved independently.

Step 3(Combine): The solution of each sub-problem is combined recursively to generate the solution of the original problem.

The unit is structured as follows. In section 2.2, we define a recurrence relation of divide & conquer approach in general and take Merge Sort and closest pair problems as examples to derive their recurrence relations. From section 2.3 to 2.6 we present several problems, derive their recurrence relations and provide solutions. Summary & solution to check your progress are given in 2.7 & 2.8 respectively.

2.1 Objectives

After reading this Unit, you will be able to:

- Define concepts of divide-and-conquer approach.
- Formulate the recurrence relation of problems that are solvable by Divide-and-Conquer approach
- Apply how divide-and-conquer approach is used to solve problems like Binary search, Quick-sort, Merge-sort, Integer Multiplication and Matrix multiplication.

2.2 Recurrence Relations Formulations in Divide and Conquer Approach

As many algorithms are recursive in nature, the computational complexity of such algorithms is defined by Divide and Conquer approach. In Divide and Conquer approach, the running time is equated as a recurrence relation which is based upon three steps:

- 1) Divide: Dividing the given problem into sub-problems.
- 2) Conquer: Each sub-problem solves itself by calling itself recursively. (Base case: If size of sub-problem is small enough, that solve it directly)
- 3) Combine: Each solution of the sub-problem is combined to obtain the original solution.

In general, the recurrence relation of a Divide and Conquer approach is presented as

Greedy Technique

$$T(n) = \begin{cases} \theta(1) & \text{if } n \leq C \\ a T(n/b) + f(n) & \end{cases}$$

where,

- $T(n)$ = Time required to solve a problem of size 'n'.
- a corresponds to the number of partitions made to a problem.
- $T(n/b)$ corresponds to the running time of solving each subproblem of size (n/b) .
- $f(n) = D(n) + C(n)$ – time required to divide the problem and combine the solutions respectively. If the problem size is small enough say, $n \leq C$ for some constant C , we have a best case which can be directly solved in a constant time: $\theta(1)$, otherwise, divide a problem of a size n into subproblems, each of $\left(\frac{1}{b}\right)$ size.

Example: Merge Sort algorithm closely follows the Divide-and-Conquer approach. which divides the array of n elements which requires to be sorted into two subarrays of $n/2$ elements each. Then, it sorts the two subarrays recursively. Finally it combines (merges) the two sorted subarrays to produce the result. The following procedure MERGE_SORT (A, p, r) sorts the elements in the subarray $A[p, \dots, r]$, where p and r are starting and end indexes of an array A . If $p \geq r$, the subarray has at most one element and is therefore already sorted. Otherwise, the divide step simply computes an index q that partitions $A[p, \dots, r]$ into two sub-arrays: $A[p, \dots, q]$ containing $\lfloor (n/2) \rfloor$ elements, and $A[q+1, \dots, r]$ containing $\lfloor (n/2) \rfloor$ elements. The pseudocode of the Merge Sort is given below in figure 2.

```

MERGE_SORT ( $A, p, r$ )
1. if ( $p < r$ )
2.   then  $q \leftarrow \lfloor (p+r)/2 \rfloor$       /*Divide
3.   MERGE_SORT ( $A, p, q$ ) /*Conquer
4.   MERGE_SORT ( $A, q+1, r$ ) /*Conquer
5.   MERGE ( $A, p, q, r$ )      /*Combine

```

Figure 2: Steps in merge sort algorithms

Problem1 To set up a recurrence $T(n)$ for MERGE SORT algorithm, we can note down the following points:

- Base Case: if $n=1$ (only one element in the array), the time will be $\theta(1)$.
- When there are $n>1$ elements, a running time can be derived as follows:
 - (1) Divide: Compute q as the middle of p and r , which takes constant time.
 - (2) Conquer: Two subproblems are solved recursively each of size $n/2$, which contributes $2T\left(\frac{n}{2}\right)$ to the running time.

(3) Combine: Merging of two sorted subarrays (for which we use MERGE (A, p, r) of an n element array) takes time $\theta(n)$, so $C(n) = \theta(n)$.

Thus $f(n) = D(n) + c(n) = \theta(1) + \theta(n) = \theta(n)$ which is a linear function of n.

Thus from all the above 3 steps, a recurrence relation for MERGE_SORT (A, l, P, r) in the worst case can be written as:

$$T(n) = \begin{cases} \theta(1) & \text{if } n \geq 1 \\ 2T\left(\frac{n}{2}\right) + \theta(n) & n > 1 \end{cases}$$

This algorithms will be explained in detailed in section 2.4.2

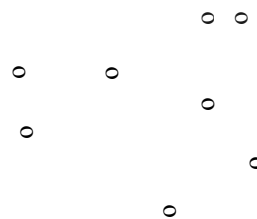
Once the recurrence relation is derived for a problem following divide & conquer technique, the next step would be solve it through any method such as Recursive tree or Substitution method or Master Method. We have $T(n) = \theta(n \log n)$.

Problem 2: Closest Pair Problem

Problem Definition- Given a set of n points in a plane. Find a pair of points which are closest together. If p1 and p2 are two points in the plane, then the Euclidean distance between p1 and p2 is $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$. We need to find the closest pair of points. In case the two points are having the same locations points, that pair is the closest with distance 0. What will be the total number of comparisons of pair of points? If there are n points, there will be $\frac{n(n-1)}{2}$ pair of points for comparisons.

The brute force method will take $O(n^2)$ time complexity. Since this approach requires exhaustive search, let us explore divide and conquer technique to reduce its time complexity.

The following figure illustrates a set of n points in P in a plane.



Assume all the points in P are sorted by x coordinates, it is possible to divide the points set P into two halves: the left half of P and the right half of P by drawing an imaginary vertical line as shown below in a figure

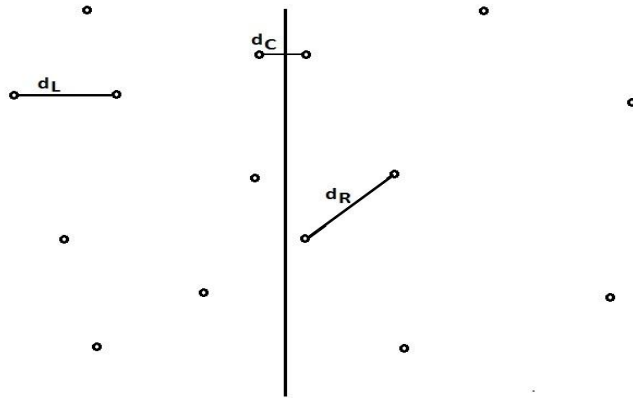


Figure 4: P is partitioned into two halves.

Shortest distances are shown as d_L and d_R in the left half and right half respectively.

The recurrence relation of the closest pair problem is similar to Merge Sort Problem i.e.,

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + \theta(n) \\ &= O(n \log n) \end{aligned}$$

2.3 Binary Search

Binary search is a procedure of finding the location of an element in a sorted array. The iterative version of the algorithm is given in figure . The divide and conquer version of the algorithm proceeds as follows:

- 1) If the key is found at the middle position of the array, the algorithm terminates, otherwise
- 2) **Divide** the array into two sub-arrays recursively. If the key is smaller than the middle value, select the left part of the sub-arrays, otherwise (if it is larger) select the right part of :the sub-array. The process continues until the search interval is not empty or it cannot be broken further
- 3) **Conquer**(solve) the sub-array by determining whether the key is located in that sub-array.
- 4) Obtain the result

The recursive version of the algorithm is given in figure

Iterative Binary Search Algorithm

Input: A sorted array 'A' of size 'n' (in ascending order) and a element 'x' to be searched.

Output: Index of 'x' in 'A' if found, else return 0.

//**lowerIndex:** Index of the first element in the array, considered to search for 'x'.

//**upperIndex:** Index of the last element in the array, considered to search for 'x'.

//**middleIndex:** Index of the middle element in the array under consideration.

// **A[middleIndex]:** Value at the middle index of the array 'A'.

```

{
    lowerIndex = 0
    upperIndex = n
    while(lowerIndex <= upperIndex){
        middleIndex = ((lowerIndex + upperIndex)/2)
        if(A[middleIndex] == x) // if 'x' is found
            return middleIndex; // Index of 'x' in 'A'
        else if (x < A[middleIndex])
            upperIndex = middleIndex - 1;
        else
            lowerIndex = middleIndex + 1;
    }
    return -1 // if 'x' is not found
}

```

Figure 5: iterative_binary search Algorithm

Recursive Binary Search Algorithm

```

Recursive_Binary_Search( lowerIndex, upperIndex, A [ ], x )
{
    while (lowerIndex ≤ upperIndex)
        middleIndex = (lowerIndex + upperIndex)/2 ;
        if A[ middleIndex] = x
            return middleIndex ; /* terminate the program*/
        else if ( x < A[ middleIndex])
            Recursive_Binary_Search(lowerIndex,middleIndex-1,A[ ],x)
        else
            Recursive_Binary_Search(middleIndex+1, upperIndex, A [ ], x)
}

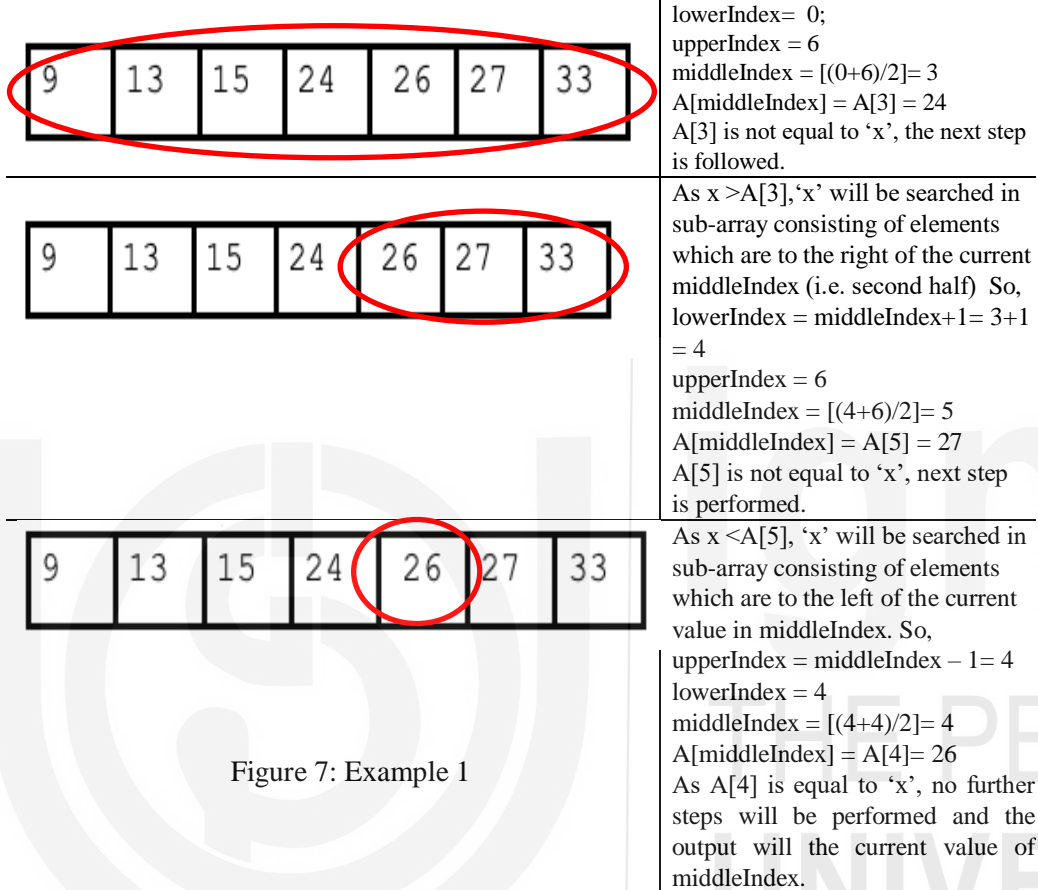
```

Figure 6: Recursive Binary Search

Let us apply the algorithm to the following problems in example 1 (Figure 7) & example 2 (Figure 8)

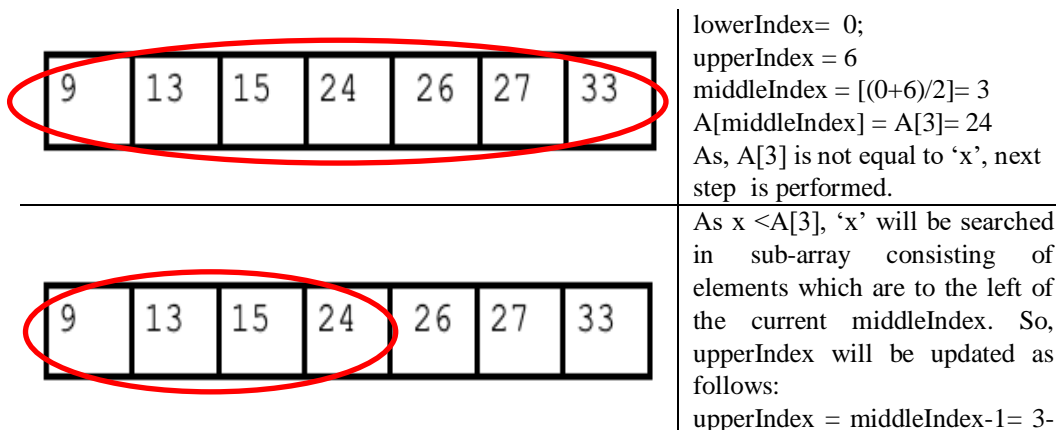
Example 1: Consider a sorted array $A=[9,13,15,24,26,27,33]$ and the element to be searched is '26'.

Solution: For the given array 'A', consider 'x' as 26. Following steps will be performed by binary search. Here, the circle denotes the elements considered in a particular iteration.



Example 2: Consider a sorted array $A = [9,13,15,24,26,27,33]$ and the element to be searched is '17'.

Solution: For the given array 'A', consider 'x' as 17. Following steps will be performed by binary search. Here, red circle denotes the elements considered in a particular iteration.



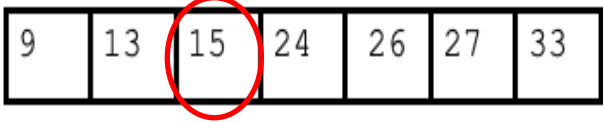

	<div>1 = 2 lowerIndex = 0 Now, middleIndex = [(0+2)/2]= 1 A[middleIndex] = A[1]= 13 As A[1] is not equal to 'x', next step is performed.</div>
<div></div>	<div>As $x > A[1]$, 'x' will be searched in sub-array consisting of elements which are to the right of the current value in middleIndex. So, lowerIndex will be updated as follows: lowerIndex = middleIndex + 1 = 2 upperIndex = 2 Now, middleIndex = [(2 + 2)/2]= 2 A[middleIndex] = A[2]= 15 As A[2] is not equal to 'x', next step is performed.</div>
<div></div>	<div>As $x > A[2]$, 'x' will be searched in sub-array consisting of elements which are to the right of the current value in middleIndex. So, lowerIndex will be updated as follows: lowerIndex = middleIndex + 1 = 3 upperIndex = 2 As lowerIndex > upperIndex, no further steps is performed. The output will be -1 which indicates 'x' is not in the array 'A'.</div>

Figure 8: Example 2

2.3.2 Analysis of Binary Search:

Method 1: Let us assume for the moment that the size of the array is a power of 2, say 2^k . Each time in the while loop, when we examine the middle element, we cut the size of the sub-array into half. So Before the 1st iteration size of the array is 2^k .

After the 1st iteration size of the sub-array of our interest is: 2^{k-1}

After the 2nd iteration size of the sub-array of our interest is: 2^{k-2}

.....

.....

After the kth iteration size of the sub-array of our interest is: $2^{k-k}=1$

So we stop after the next iteration. Thus we have at most $(k+1) = (\log n + 1)$ iterations.

Since with each iteration, we perform a constant amount of work: computing a mid point and few comparisons. So overall, for an array of size n, we perform $C \cdot (\log n + 1) = O(\log n)$ comparisons. Thus $T(n) = O(\log n)$

2.3.3 Recurrence Relation of Binary Search

Greedy Technique

The complexity of divide and conquer approach is defined by recurrence relation as follows:

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

As binary search follows divide and conquer approach and performs dividing the array but searching on only one sub-array in an iteration, the computational complexity of binary search technique can be defined as recurrence relation in the following form:

$$T(n) = T(n/2) + k$$

where, a, b, and f(n) are replaced with values 1, 2, k (constant less than n), respectively. K is constant value for divide and conquer operation on solving this recurrence relation by substitution method, the computational complexity for binary search is $O(\log n)$.

Check Your Progress 1

Question 1: What will be minimum number of comparisons required to find the minimum and the maximum of 100 numbers?

Question 2: Given an array $A = [45, 77, 89, 90, 94, 99, 100]$ and key = 100. What are the mid values of an array (corresponding array elements) generated in the first and second iterations?

2.4 Sorting Algorithms

It is process of rearranging the elements of a list in either ascending or descending order. For example, consider a list of elements $\langle a_1, a_2, a_3, \dots, a_n \rangle$ as input which needs to be sorted in ascending order, then the output of the sorting algorithm will be rearranging the list such that $\langle a_1 \leq a_2 \leq a_3 \leq \dots \leq a_n \rangle$. Generally, there are number of sorting algorithms like bubble sort, merge sort, radix sort, quick sort and many more. In this section, we will cover Merge-Sort and Quick-Sort sorting algorithms as they are based on divide and conquer approach.

2.4.1 Merge-Sort

As discussed in the above section, Merge-Sort algorithm is a divide-and-conquer based sorting algorithm. It follows a divide and conquer approach to perform sorting. The algorithm in divide step repeatedly partitions an array into several sub arrays until each sub array consists of a single element. Then, it merges sub-problem solutions together according to following steps:

- **Divide** the array into two equal sub-arrays Compare the sub-array's first elements
- **Conquer**(solve) each sub-array by performing sorting operation. Apply the recursion unless the array is sufficiently small.
- **Combine**(Merge) the solutions into a single array

The following example illustrates the above steps:

Suppose the array has following numbers:

37 20 23 30 18 15 27 17

In divide step , the array is divided into two subarrays

37 20 23 30 and 18 15 27 17

In conquer step , we sort each subarray

20 23 30 37 and 15 17 18 27

In combine(merge) ,all the subarrays are merged in sorted order

15 17 18 20 23 27 30 37

Pseudo-code of the Merge-Sort

Algorithm 2: Merge-Sort (X, m, n)

```

if ( $m < n$ ) {
     $i = (m+n)/2$ ;           //Divide
    Merge-Sort ( $X, m, i$ );   //Conquer
    Merge-Sort ( $X, i+1, n$ ); //Conquer
    Merge ( $X, m, i, n$ );     //Combine
}
else {
    Already sorted
}
    
```

Figure 7: Merge Sort Algorithm

In figure 7, X is the given array of size $m \times n$. It first checks for the number of elements in X . If $m \Rightarrow n$, then the given array is sorted already as there will be at most one element only. Alternatively, it will first perform the Divide step by computing the middle index 'i' of X and then partitions it into two sub-arrays and calls the same algorithm on each sub-array recursively. Finally, the merge operation performs the Combine step. The pseudo-code to perform Merge operation is detailed in figure 8.

The following algorithm merges the two sorted subarrays $S1[]$ and $S2[]$ into $S[]$

Pseudo-code for Merge Operation

Problem Definition- Merge two sorted arrays $S1[]$ and $S2[]$ into one sorted array $S[]$

Inputs – positive integers m and n , sorted arrays $S1[]$ and $S2[]$, indexed from $1..m$ and $1..n$ respectively

Output- an array S indexed from 1 to $m+ n$ containing sorted values from $S1$ and $S2$

Merge($S1[], S2[], S[] \ m, n$)

{

int i, j, k

/* initialization of i, j, k */

```

i=1; j = 1; k=1
while ( i ≤ m && j ≤ n) /* scanning S1 and S2*/
if S1[i] < S2[j]
{
S[k] = S1[i];
i++;
}
else
{
S[k] = S2[j];
j++;
}
k++
}
if(i > m)
copy S2[j] through S2[n] to S[k] through S[m+n]
else
copy S1[i] through S[m] to S[k] through S[m+n]
}

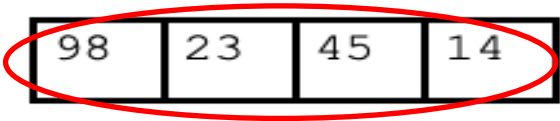
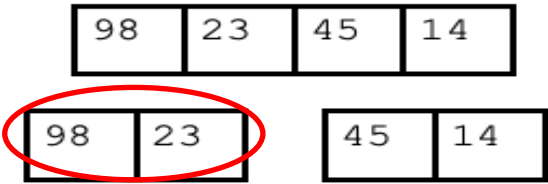
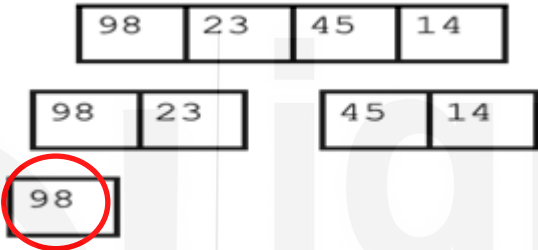
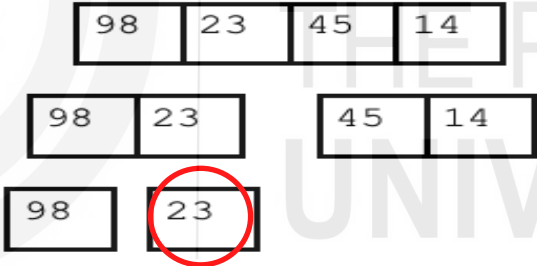
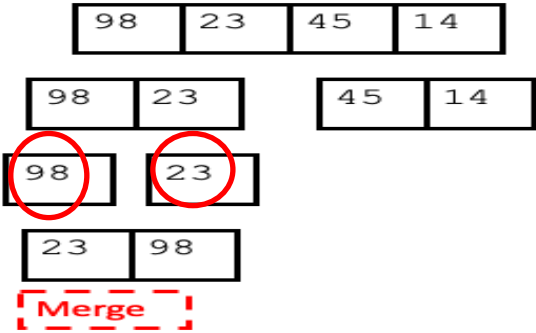
```

Figure 8: Merge Algorithm

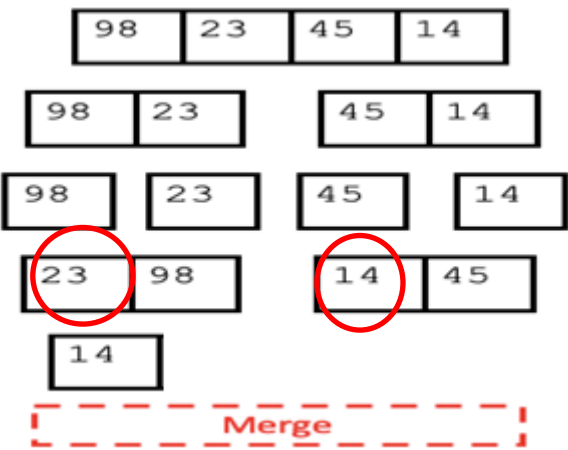
Note that the merge algorithm which requires an amount of *additional storage* equal to the amount of storage for all of the original data .

Example 3: Consider an array 'A' of size '4' as [98, 23, 45, 14]. Perform Merge-Sort on 'A'.

Solution: Merge-Sort follows partitioning of an array into two sub-arrays from the middle index. This partition is performed recursively on each sub-array till individual elements are obtained. Then, merge operation is performed on individual elements in a sorted order recursively. Following are the steps of Merge-Sort to sort array 'A'. Here, red circle denotes the considered elements at a particular step.

	<p>m = 0; n = 3 As m < n, if condition is TRUE. So, calculate 'mid' as: mid = $\lfloor [(0 + 3)/2] \rfloor = 1$; Now, recursive call to Merge-Sort(A, 0, 1);</p>
	<p>m = 0; n = 1; As m < n, if condition is TRUE. So, calculate 'mid' as: mid = $\lfloor [(0 + 1)/2] \rfloor = 0$ Now, recursive call to Merge-Sort(A, 0, 0);</p>
	<p>m = 0; n = 0; As 'm' is not less than 'n', if condition is False. So, <i>else</i> condition is executed. This completes recursive call of Merge-Sort(A, 0, 0). Now, recursive call to Merge-Sort(A, 1, 1);</p>
	<p>m = 1; n = 1; As 'm' is not less than 'n', if condition is False. So, <i>else</i> condition is executed. This completes recursive call of Merge-Sort(A, 1, 1). Now, call Merge(A, 0, 0, 1);</p>
	<p>In <i>Merge</i>, each value in the two arrays are compared individually. Then, they are stored in sorted order in a new array. As there are single values in the two arrays, both values are compared and stored accordingly in a new array termed as 'X'. i.e., 98 > 23, 23 is placed first in 'X' while 98 is placed after 23 in 'X'. This completes call of Merge(A, 0, 0, 1). Now, recursive call to Merge-Sort(A, 2, 3);</p>

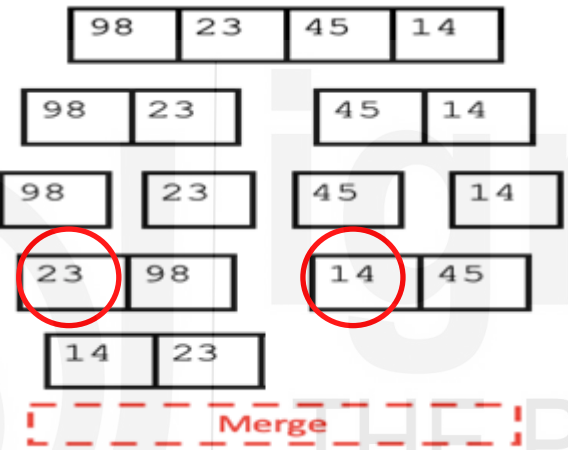
	<p> $m = 2;$ $n = 3;$ As $m < n$, if condition is TRUE. So, calculate 'mid' as: $\text{mid} = \text{floor}[(2+3)/2] = 2$ Now, recursive call to <i>Merge-Sort</i>(A, 2, 2); </p>
	<p> $m = 2;$ $n = 2;$ As 'm' is not less than 'n', if condition is False. So, <i>else</i> condition is executed. This completes recursive call of <i>Merge-Sort</i>(A,2,2). Now, recursive call to <i>Merge-Sort</i>(A, 3, 3); </p>
	<p> $m = 3;$ $n = 3;$ As 'm' is not less than 'n', if condition is False. So, <i>else</i> condition is executed. This completes recursive call of <i>Merge-Sort</i>(A,3,3). Now, call <i>Merge</i>(A, 2, 2, 3); </p>
	<p> In <i>Merge</i>, each value in the two arrays are compared individually. Then, they are stored in sorted order in a new array. As there are single values in the two arrays, both values are compared and stored accordingly in a new array termed as 'X'. i.e., $45 > 14$, 14 is placed first in 'X' while 45 is placed after 14 in 'X'. This completes call of <i>Merge</i>(A, 2, 2, 3). Now, call to <i>Merge</i> (A, 0, 1, 3); </p>



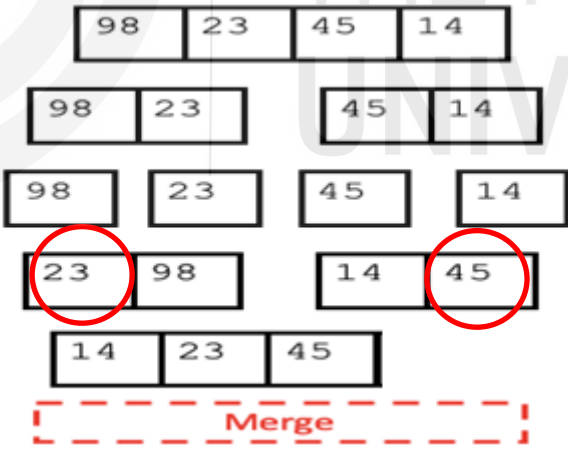
In *Merge*, each value in the two arrays are compared individually. Then, they are stored in sorted order in a new array.

In case of multiple values in any of the two arrays, each value of two arrays are considered one by one and compared to store in a new array termed as 'X' accordingly.

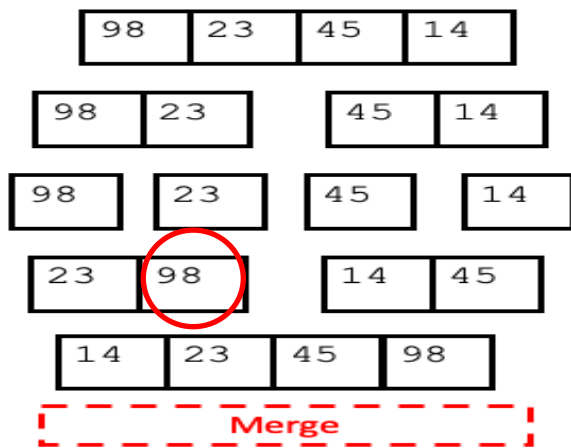
As 23 is greater than 14, 14 is placed first in 'X'. The value 23 will not be placed after 14 in 'X' but will be compared with next value in the other array.



As 23 is less than 45, 23 is placed after 14 in 'X'. The value 45 will not be placed after 23 in 'X' but will be compared with next value in the other array.



As 45 is less than 98, 45 is placed after 23 in 'X'. The value 98 will not be placed after 45 in 'X' but will be compared with next value in the other array.



To compare 98, there is no element in the other array. So, the value 98 will be placed after 45 in 'X'.

Finally, the sorted array is: [14, 23, 45, 98]

Figure 9: Example 3: Merge Sort

Analysis of MERGE-SORT Algorithm

- For simplicity, assume that n is a power of $\Rightarrow 2$ each divide step yields two sub-problem, both of size exactly $n/2$.
- The base case occurs when $n = 1$.
- When $n \geq 2$, then

Divide: Just compute q as the average of p and $r \Rightarrow D(n) = O(1)$

Conquer: Recursively solve sub-problems, each of size $\frac{n}{2} \Rightarrow 2T\left(\frac{n}{2}\right)$

Combine: MERGE an n -element subarray takes $O(n)$ time $\Rightarrow C(n) = O(n)$

❖ $D(n) = O(1)$ and $C(n) = O(n)$

❖ $F(n) = D(n) + C(n) = O(n)$, which is a linear function in n .

Hence Recurrence for Merge Sort algorithm can be written as:

$$T(n) = \begin{cases} \theta(1) & \text{if } n = 1 \\ 2T\left(\frac{n}{2}\right) + \theta(n) & \text{if } n \geq 2 \end{cases} \quad (1)$$

This Recurrence 1 can be solved by any of two methods:

(1) Master method or

(2) by Recursion tree method:

1) Master Method:

$$T(n) = 2T\left(\frac{n}{2}\right) + c \cdot n \quad \text{-- (1)}$$

By comparing this recurrence with $T(n) = 2T\left(\frac{n}{2}\right) + f(n)$

We have: $a = 2$

$$b = 2$$

$$f(n) = n$$

$$n^{\log_b a} = n^{\log_2 2} = n; \text{ Now compare } f(n) \text{ with } n^{\log_2 2} \text{ i.e. } (n^{\log_2 2} = n)$$

Since $f(n) = n = O(n^{\log_2 2}) \Rightarrow$ Case 2 of Master Method

$$\Rightarrow T(n) = \theta(n^{\log_b a} \cdot \log n) \\ = \theta(n \cdot \log n)$$

$$\text{Total} = C \cdot n + C \cdot n + \dots + (\log_2 n + 1) \text{ terms}$$

$$= C \cdot n (\log n + 1)$$

$$= \theta(n \log n)$$

Recursion tree:

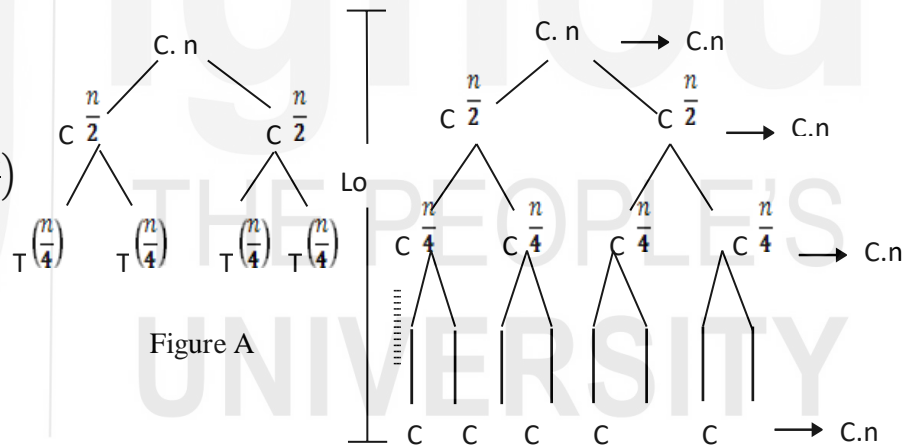
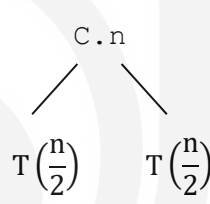


Figure A

Figure B

Figure 10 : Recursion Tree of Merge Sort

On solving these recurrence relations, $T(n)$ will be $O(n \log n)$.

Note:

- Merge-Sort is not an in-place algorithm. However, it is an out-place algorithm as it needs $O(n)$ extra space to perform merge operation.
- It is not suitable for array of small size.
- It is a stable algorithm which means that the order of repeated elements in the given is not changed in the sorted array.

2.4.2 Quick-Sort

Quick-Sort is another sorting algorithm that works on the principle of divide-and-conquer approach. It works by arranging the elements in an array by identifying their correct place (or index). Moreover, it sorts elements within the list without requiring any additional space as compared to Merge-Sort which requires additional space to perform sorting. Due to which, Quick-Sort provides advantage of performing in-place sorting. The arrangement of elements in the input array (which is to be sorted) effects the running time of Quick-Sort.

Generally, it is the fastest among all sorting algorithms in practice and that's why, it is named as Quick-Sort. The running time of quick-sort is $O(n \log n)$ in average scenario. However, if the elements are sorted already, then it is the worst-case scenario for quick-sort and it's running time is $O(n^2)$. Quick sort algorithm is different from Merge Sort algorithm is that in Quick Sort, the array is partitioned around the pivot value. Smaller elements than the pivot value are placed at the left side and larger elements at the right side (best case). This is the central idea of partitioning algorithm. Quick-sort algorithm performs following steps to sort the elements of a given array 'A' of size $p \times r$:

- 1) **Divide:** In this, an element of the given array is considered as pivot element whose correct index 'q' in the array is determined by rearranging the array elements. Then, the given array $A[m \dots n]$ is partitioned into two sub-arrays, $A[p..q]$ and $A[q+1..r]$, in such a way that all the elements in $A[p..q]$ are smaller than $A[q]$ and all the elements in $A[q+1..r]$ are greater than $A[q]$. Generally, the procedure which is used to perform this step is termed as Partition. The output of this procedure is the index 'q' which divides the given array 'A'.
- 2) **Conquer:** The partitioned sub-arrays, $A[p..q]$ and $A[q+1..r]$, recursively call the step (1) to perform sorting of the elements.
- 3) **Combine:** As all the elements are sorted in their respective places, there is no need to perform combine step and the array is sorted.

The basic concept behind Quick-Sort is as follows: Suppose we have an unsorted input data $A[p \dots r]$ to sort. Here **Partition** procedures always select a last element $A[r]$ as a Pivot element and set the position of this $A[r]$ as given below:

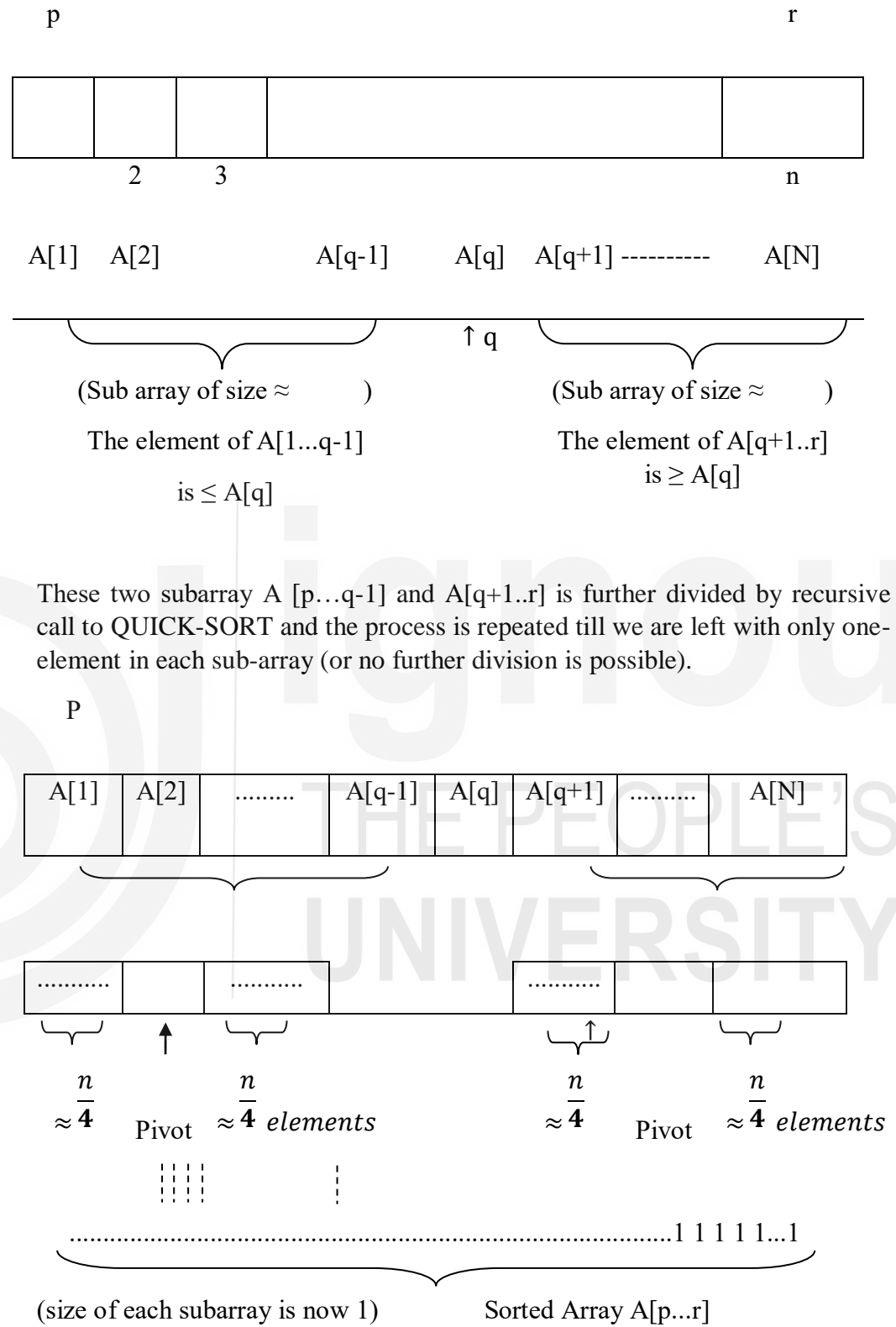


Figure 11: Graphic Representation of partitioning procedure in Quick Sort

Quick sort algorithm. is presented below:

Algorithm 4: QUICK-SORT(A, p, r)

Greedy Technique

```

if (p < r)
{
    q = PARTITION(A, p, r); // Divide
    QUICKSORT(A, p, q);      // Conquer
    QUICKSORT(A, q+1, r);    // Conquer
}

```

Figure 11: Quick Sort Algorithm

A brief description of the algorithm.

- To sort an array A with n -elements, a initial call to QuickSort in QUICKSORT ($A, 1, n$)
- QUICKSORT (A, p, r) uses a procedure Partition (), which always select a last element $A[r]$, and set this $A[r]$ as a Pivot element at some index (say q) in the array $A[p..r]$.

The PARTITION () always return some index (or value), say q , where the array $A[p..r]$ partitioned into two subarray $A[p..q-1]$ and $A[q+1..r]$ such that $A[p..q-1] \leq A[q]$ and $A[q] \leq A[q+1..r]$. The partition algorithm is presented in figure 12.

Algorithm 5: PARTITION(A, m, n)

```

PARTITION (A, p, r)
{
1:  x ← A[r]          /* select last element
2:  i ← p - 1          /* i is pointing one position
                        before than p, initially
3:  for j ← p to r - 1 do
4:      {
5:          if A[j] ≤ x
6:              {
5:                  i ← i + 1
6:                  Exchange (A[i] ↔ A[j])
6:              }
4:      } /* end for
7:  Exchange (A [i + 1] and A[r])
8:  return ( i+ 1)
}

```

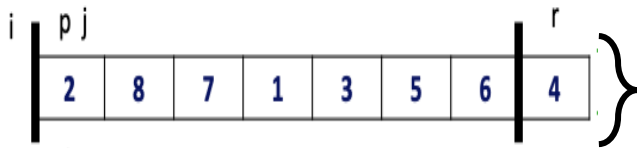
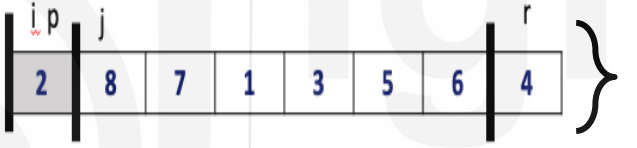
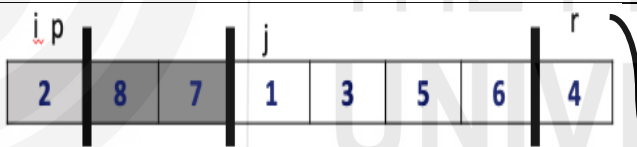
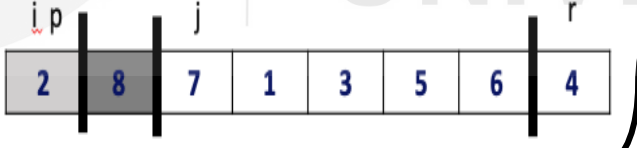
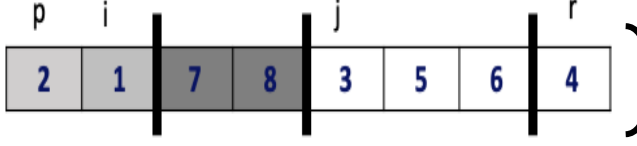
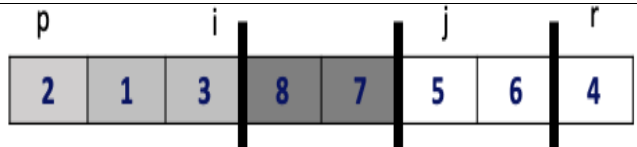
Figure 12: Parition Algorithm

The running time of PARTITION procedure is $\theta(n)$, since for an array $A[p..r]$, the loop at line 3 is running $O(n)$ time and other lines of code take constant time i.e. $O(1)$ so overall time is $O(n)$.

Let us consider one Quick Sort example based on the algorithm presented above.

Example 4: Consider an array ‘A’ of size ‘8’ as [2, 8, 7, 1, 3, 5, 6, 4]. Perform Quick-Sort on ‘A’.

Solution: The sorting of the given array ‘A’ is illustrated below. We have considered four temporary variables i.e., i, p, j, and r, to perform Quick-Sort. Here, ‘r’ represents the chosen pivot element. As Quick-Sort partitions array into two sub-blocks, the elements of the first block are highlighted with light shade while elements of the second block are highlighted with dark shade.

	The initial array of 8 elements and variable settings. None of the elements have been placed in either of the first two partitions.
	The value 2 is “swapped with itself” and put in the partition of smaller values.
 	The values 8 and 7 are added to the partition of larger values
	The values 1 and 8 are swapped, and the smaller partition grows.
	The values 3 and 7 are swapped, and the smaller partition grows.

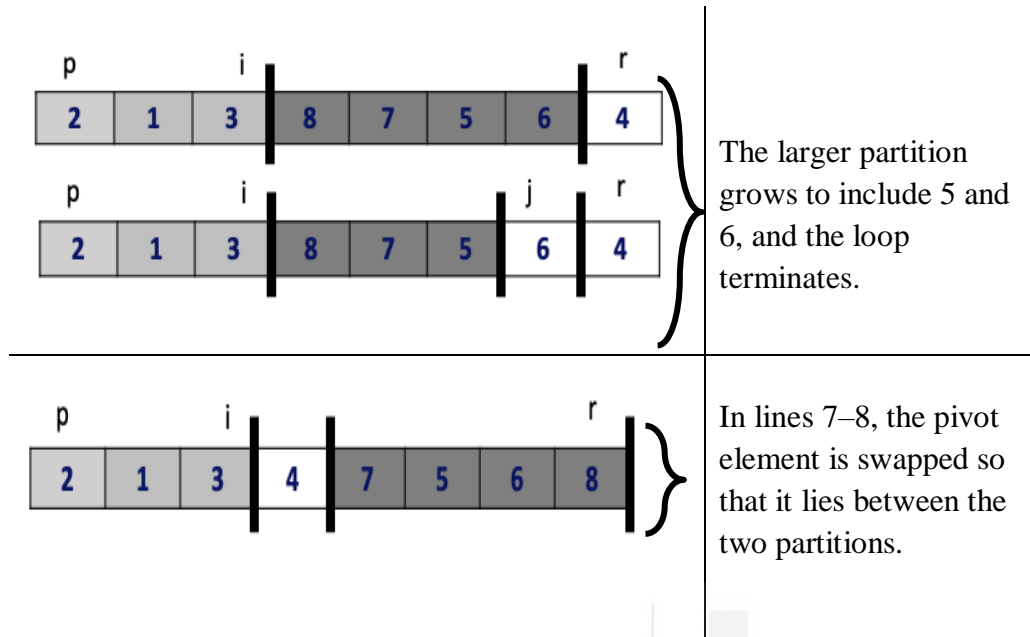


Figure 12: An example of Partition procedure

Analysis of Quick-Sort:

The computation complexity of Quick-Sort is based on the arrangement of array elements. Arrangement of elements effects the partitioning of an array. If partitioning is unbalanced, partitioning procedure will perform more number of times in comparison to balanced partition. So, more number of calls to partitioning procedure means high computational complexity. Therefore, the Quick-Sort has different complexity in different scenarios:

Partitioning of the subarrays depends on the input data we receive for sorting.

Best Case: If the input data is not sorted, then the partitioning of subarray is balanced; in this case the algorithm runs asymptotically as fast as merge-sort (i.e. $O(n \log n)$).

Worst Case: If the given input array is already sorted or almost sorted, then the partitioning of the subarray is unbalancing in this case the algorithm runs asymptotically as slow as Insertion sort (i.e. $\theta(n^2)$).

Average Case: Except best case or worst case. The figure (a to c) shows the recursion depth of Quick-sort for Best, worst and average cases:

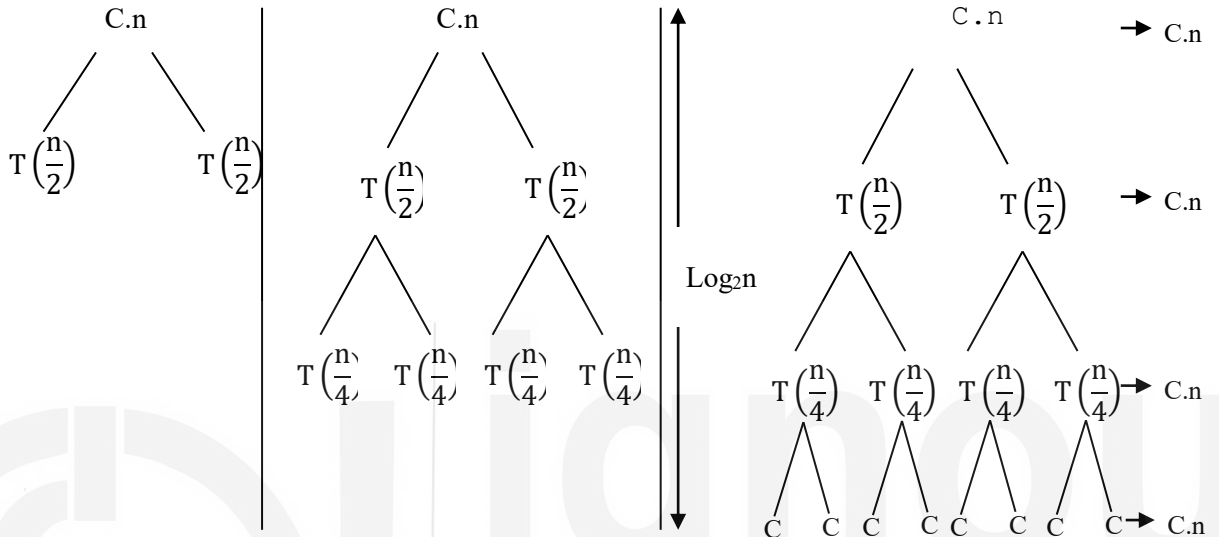
Best Case (Input array is not sorted)

The best case behaviour of Quicksort algorithm occurs when the partitioning

procedure produces two regions of size $\approx \frac{n}{2}$ elements.

In this case, Recurrence can be written as: $T(n) = 2T\left(\frac{n}{2}\right) + \theta(n)$

Method 1: Using Master Method; we have $a=2$; $b=2$, $f(n)=n$ and $n^{\log_b a} = n^{\log_2 2} = n$



❖ $F(n) = n = O(n^{\log_2 2}) \rightarrow$ Case 2 of master method
 $T(n) = \theta(n \log n)$

Figure 12: Method 2: Recursion Tree:

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

$$\begin{aligned} \text{Total} &= C \cdot n + C \cdot n + \dots + \log_2 n \text{ terms } (c) \\ &= C \cdot n \log_2 n \\ &= \theta(n \log n) \end{aligned}$$

Worst Case:

[When input array is already sorted]

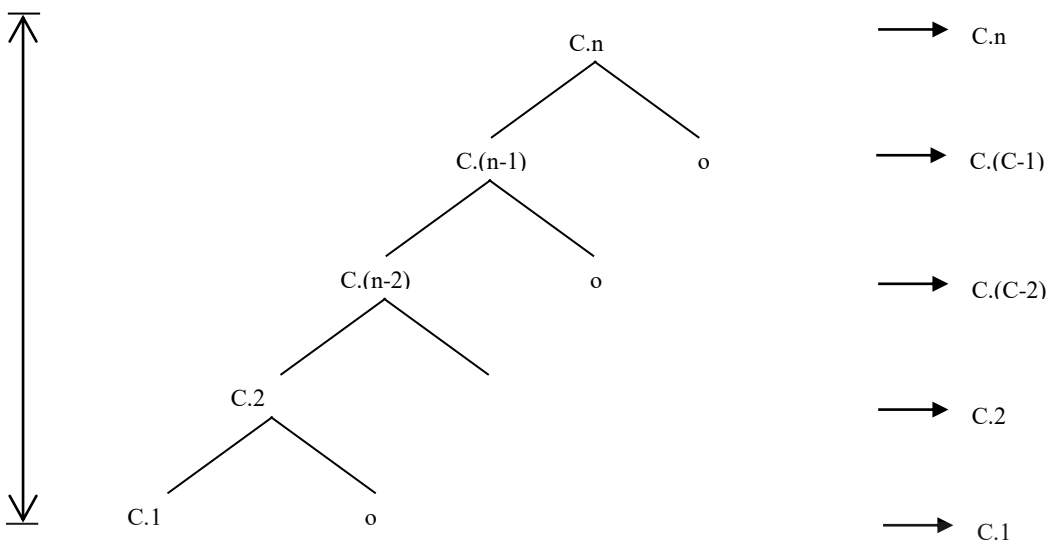
The worst case behaviour for QuickSort occurs, when the partitioning procedures one region with $(n-1)$ elements and one with 0-elements \rightarrow completely unbalanced partition.

In this case:

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \theta(n) \\ &= T(n-1) + 0 + C \cdot n \end{aligned}$$

Recursion Tree:

Greedy Technique



$$\text{Total} = C(n + (n-1) + (n-2) + \dots + 2 + 1)$$

$$= C \cdot \left(\frac{n(n+1)}{2} \right) = O(n^2)$$

Figure 13: Unbalanced Recursion tree

The worst case scenario in Quick Sort can be avoided with the following consideration:

- Pick up any element as a pivot randomly instead of the last element of an array.

The following code fragment illustrates the concept:

```
Randomized_Partition ( A, p, r)
```

```
{
```

```
PivotIndex = random( p, r)
```

```
exchange( A[PivotIndex], A[r])
```

```
partition( A,p,r)
```

```
}
```

Now let us modify the QuickSort()

```
QuickSort(A,p,r)
```

```
if (p < r)
```

```
{
```

```
PivotIndex = Randomized_Partition( A, p,r)
```

```
QuickSort( A, p, PivotIndex-1)
```

$$\}$$

Check Your Progress 2

Greedy Technique

Question 1: Consider the following elements and sort in ascending order using merge sort: 6, 2, 11, 7, 5, 4.

Question 2: What are properties of a Merge-Sort.

Question 3: Consider first element of the array 'A' = [7, 11, 14, 6, 9, 4, 3, 12] as the pivot element. What will be the sequence of elements on applying Quick-Sort on this, after the correct position of the considered pivot element.

Question 4: A machine needs a minimum of 200 sec. to sort 1000 elements by Quick-Sort. Approximately what will be the minimum time needed to sort 200 elements ?

2.5 Integer Multiplication

The brute force algorithm for multiplying two large integer numbers which everyone of us uses by hand, takes quadratic time i.e., $O(n^2)$. Because each digit of one number is multiplied by each digit in another number. Let us explore the better algorithm, which applies divide and conquer technique for integer multiplication for large numbers.

Assume X and Y are two n digits number. Divide X and Y into two halves of approximately $n/2$ digits each. The following examples illustrate the division process:

- (i) $657,138 = 657 * 10^3 + 138$
- (ii) $6578,381 = 6578 * 10^3 + 381$

Let us generalize the number representation. If Z is an n-digit number, it would be divided, into two halves, the first half with Ceiling with $\lceil n/2 \rceil$ and the second half with Floor $\lfloor n/2 \rfloor$ as shown below:

$$Z(\text{ n digit number}) = X_L * 10^m + X_R$$

Suppose are given two n- digit numbers:

$$Z1 = X_L * 10^m + X_R$$

$$Z2 = Y_L * 10^m + Y_R$$

$$\begin{aligned} Z1 * Z2 &= (X_L * 10^m + X_R)(Y_L * 10^m + Y_R) \\ &= X_L * Y_L * 10^{2m} + (X_L * Y_R + Y_L * X_R) 10^m + X_R Y_R \quad (i) \end{aligned}$$

It is visible from the operations that the product of Z1 and Z2 require four operations, each of which is half in size. There are some additional plus operation which increases to $O(n)$ extra work

If we do these four multiplications recursively, applying the algorithm and terminating at the base conditions, the recurrence relation of integer multiplication can be formulated as:

$$T(n) = 4T(n/2) + O(n)$$

Applying the master method, $T(n) = O(n^2)$.

There is no improvement in time complexity. In 1962, A.A. Karatsuba discovered an asymptotically faster algorithm $O(n^{1.59})$ for multiplying two n -digit numbers using divide & conquer approach.

Karatsuba method (using Divide and Conquer)

In 1962, A.A. Karatsuba discovered a method to compute $X.Y$ (as in Equation(1)) in only 3 multiplications, at the cost of few extra additions; as follows:

Let $U=(a+b).(c+d)$

$V=a.c$

$W=b.d$

Now

$$X.Y = V.10^{2^{[n/2]}} + (U - V - W).10^{[n/2]} + W \dots \dots \dots (2)$$

Now, here, $X.Y$ (as computed in equation (2)) requires only 3 multiplications of size $n/2$, which satisfy the following recurrence:

If $n=1$

$$T(n) = \begin{cases} O(1) \\ 3T\left(\frac{n}{2}\right) + O(n) \end{cases}$$

Otherwise

Where $O(n)$ is the cost of addition, subtraction and digit shift (multiplications by power of 10's), all these take time proportional to 'n'.

Method 1 (Master method)

$$T(n) = 3T\left(\frac{n}{2}\right) + O(n)$$

$a=3$

$b=2$

$f(n)=n$

$$n^{\log_b a} = n^{\log_2 3}$$

$$f(n) = n = O(n^{\log_2 3 - \epsilon}) \Rightarrow \text{case 1 of Master Method}$$

$$\Rightarrow T(n) = \Theta(n^{\log_2 3});$$

$$\Rightarrow \Theta(n^{1.59})$$

Method 2 (Substitution Method)

$$T(n) = 3T\left(\frac{n}{2}\right) + O(n)$$

$$= 3T\left(\frac{n}{2}\right) + c.n$$

Now

$$T(n) = c.n + 3T\left(\frac{n}{2}\right)$$

$$= c.n + 3\left\{c.\frac{n}{2} + 3T\left(\frac{n}{4}\right)\right\}$$

$$= c.n + \frac{3}{2}c.n + 3^2.T\left(\frac{n}{4}\right)$$

$$= c.n + \frac{3}{2}c.n + 3^2\left\{c.\frac{n}{4} + 3T\left(\frac{n}{8}\right)\right\}$$

$$= c.n + \frac{3}{2}c.n + \left(\frac{3}{2}\right)^2 c.n + 3^3T\left(\frac{n}{8}\right)$$

$$= c.n + \frac{3}{2}c.n + \left(\frac{3}{2}\right)^2 c.n + 3^3\left\{c.\frac{n}{8} + 3T\left(\frac{n}{16}\right)\right\}$$

$$= c.n + \frac{3}{2}c.n + \left(\frac{3}{2}\right)^2 c.n + \left(\frac{3}{2}\right)^3 c.n + \dots + \left(\frac{3}{2}\right)^k .T\left(\frac{n}{2^k}\right)$$

$$= c.n + \frac{3}{2}c.n + \left(\frac{3}{2}\right)^2 c.n + \left(\frac{3}{2}\right)^3 c.n + \dots + \left(\frac{3}{2}\right)^{\log n} .c$$

$$= c.n \left(1 + \frac{3}{2}c.n + \left(\frac{3}{2}\right)^2 c.n + \left(\frac{3}{2}\right)^3 c.n + \dots + \left(\frac{3}{2}\right)^{\log n}\right)$$

$$= c.n \left[\frac{1 \left[\left(\frac{3}{2}\right)^{\log n + 1} - 1 \right]}{\frac{3}{2} - 1} \right]$$

$$= 2c.n \left[\left(\frac{3}{2}\right)^{\log n + 1} - 1 \right]$$

$$= 2c.n \left[\left(\frac{3}{2}\right)^1 \left(\frac{3}{2}\right)^{\log n} - 1 \right]$$

$$= 2c.n \left[\frac{3}{2} n^{\log \frac{3}{2}} - 1 \right]$$

$$= 2c.n \left[\frac{3}{2} n^{\log_2 3 - \log_2 2} - 1 \right]$$

$$\begin{aligned}
&= 2 c . n \left[\frac{3}{2} n^{\log_2 3 - 1} - 1 \right] \\
&= 2 c . n \left[\frac{3 n^{\log_2 3}}{2 n} - 1 \right] \\
&= 3 c . n^{\log_2 3} - 2 c . n \\
&= O(n^{\log_2 3})
\end{aligned}$$

2.6 Matrix Multiplication

- Matrix multiplication is a binary operation of multiplying two or more matrices one by one that are conformable for multiplication. For example two matrices A, B having the dimensions of $p \times q$ and $s \times t$ respectively; would be conformable for $A \times B$ multiplication only if $q=s$ and for $B \times A$ multiplication only if $t=p$.
- Matrix multiplication is associative in the sense that if A, B, and C are three matrices of order $m \times n$, $n \times p$ and $p \times q$ then the matrices $(AB)C$ and $A(BC)$ are defined as $(AB)C = A(BC)$ and the product is an $m \times q$ matrix.
- Matrix multiplication is not commutative. For example two matrices A and B having dimensions $m \times n$ and $n \times p$ then the matrix $AB = BA$ can't be defined. Because BA is not conformable for multiplication, even if AB are conformable for matrix multiplication.
- For three or more matrices, matrix multiplication is associative, yet the number of scalar multiplications may vary significantly depending upon how we pair the matrices and their product matrices to get the final product.

2.6.1 Straight forward method

Let's suppose, we have taken two matrices A and B of size $m \times n = 2 \times 2$ and want to multiply $A \times B$ and store the multiplication in C.

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$C_{ij} = \sum_{k=1}^n A_{ik} \times B_{kj}$$

$C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$, where, i and j represents the number of rows and columns.

For multiplying both the matrices A and B simple algorithm will be:

```
for (int i = 0; i < N; i++)
```

```

{
    for (int j = 0; j < N; j++)
    {
        C[i][j] = 0;
        for (int k = 0; k < N; k++)
        {
            C[i][j] += A[i][k]*B[k][j];
        }
    }
}

```

In the above algorithm accessing the element from 2D matrix definitely required two for loops. And then finding the multiplication for each element of C matrix required one more **for loop**. Thus, in simple approach multiplying any square matrix needed three for loops and complexity will be $O(N^3)$.

2.6.2 Divide & Conquer Strategy for multiplication

In divide and conquer strategy we say that if the problem is large, we break the problem into small problems called sub problems and solve those sub problems and combined solution of sub problems to get the solution of main problem. Now what will be the size of sub- problem? If we have 2×2 matrices we can multiply that without applying divide and conquer but if we have larger size matrices then we need to divide those matrices and solve the smaller matrices to get the final solution.

Now look at how to find the multiplication in C matrix after multiplying A and B.

$$C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

$$C_{11} = A_{11} \times B_{11} + A_{12} \times B_{21}$$

$$C_{12} = A_{11} \times B_{12} + A_{12} \times B_{22}$$

$$C_{21} = A_{21} \times B_{11} + A_{22} \times B_{21}$$

$$C_{22} = A_{21} \times B_{12} + A_{22} \times B_{22}$$

To multiply the 2×2 matrices A and B, we don't need to use the three for loops. We can perform multiplication using above formula that takes 8 multiplication and 4 additions. That is a constant time computation and this is defined as a small problem. Now what if matrix size is greater than 2 then we need to divide the matrix and solve the small matrices to get the final multiplication. We assume that matrices having dimension of power of 2 only like 4×4 , 8×8 , 16×16 and so on. Powers of 2 matrices are easy to divide into smaller sizes of 2×2 and if any matrix is not of power of 2 then fill it with zero's to make it power of 2. If we want to multiply the product of two $(n \times n)$ matrices and if n is an exact power of 2 (i.e. $n=2^k$), we divide each of A, B and C Where the result will be stored into $\left[\frac{n}{2} \times \frac{n}{2}\right]$ matrices.

Divide and Conquer Method:

Consider two matrices A and B with 4×4 dimension each as shown below,

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \quad B = \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix}$$

The matrix multiplication of the above two matrices A and B is Matrix C,

$$C = \begin{bmatrix} c_{11} & c_{12} & c_{13} & c_{14} \\ c_{21} & c_{22} & c_{23} & c_{24} \\ c_{31} & c_{32} & c_{33} & c_{34} \\ c_{41} & c_{42} & c_{43} & c_{44} \end{bmatrix}$$

where,

$$c_{11} = a_{11} * b_{11} + a_{12} * b_{21} + a_{13} * b_{31} + a_{14} * b_{41}$$

$$c_{12} = a_{11} * b_{12} + a_{12} * b_{22} + a_{13} * b_{32} + a_{14} * b_{42}$$

$$c_{21} = a_{21} * b_{11} + a_{22} * b_{21} + a_{23} * b_{31} + a_{24} * b_{41}$$

$$c_{22} = a_{21} * b_{12} + a_{22} * b_{22} + a_{23} * b_{32} + a_{24} * b_{42}$$

Now, let's look at the Divide and Conquer approach to multiply two matrices. Take two sub-matrices from the above two matrices A and B each as (A11 & A12) and (B11 & B21) as shown below,

$$\begin{array}{c}
 \mathbf{A} \qquad \qquad \mathbf{B} \\
 \left[\begin{array}{cc|cc} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ \hline a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{array} \right] \quad \left[\begin{array}{cc|cc} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ \hline b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{array} \right]
 \end{array}$$

And the matrix multiplication of the two 2x2 matrices A11 and B11 is,

$$\begin{array}{c}
 \left[\begin{array}{cc} a_{11} & a_{12} \\ a_{21} & a_{22} \end{array} \right] * \left[\begin{array}{cc} b_{11} & b_{12} \\ b_{21} & b_{22} \end{array} \right] = \left[\begin{array}{cc} a_{11} * b_{11} + a_{12} * b_{21} & a_{11} * b_{12} + a_{12} * b_{22} \\ a_{21} * b_{11} + a_{22} * b_{21} & a_{21} * b_{12} + a_{22} * b_{22} \end{array} \right] \\
 \mathbf{A11} \qquad \mathbf{B11}
 \end{array}$$

Also, the matrix multiplication of two 2x2 matrices A12 and B21 is as follows,

$$\begin{array}{c}
 \left[\begin{array}{cc} a_{13} & a_{14} \\ a_{23} & a_{24} \end{array} \right] * \left[\begin{array}{cc} b_{31} & b_{32} \\ b_{41} & b_{42} \end{array} \right] = \left[\begin{array}{cc} a_{13} * b_{31} + a_{14} * b_{41} & a_{13} * b_{32} + a_{14} * b_{42} \\ a_{23} * b_{31} + a_{24} * b_{41} & a_{23} * b_{32} + a_{24} * b_{42} \end{array} \right] \\
 \mathbf{A12} \qquad \mathbf{B21}
 \end{array}$$

So if you observe, I can conclude the following,

$$\mathbf{A11 * B11 + A12 * B21} = \begin{bmatrix} c_{11} & c_{12} & \cdot & \cdot \\ c_{21} & c_{22} & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{bmatrix}$$

Where, '+' is Matrix Addition, and c_{11} , c_{12} , c_{21} and c_{22} are equal to equations 1, 2, 3 and 4 respectively.

So the idea is to recursively divide $n \times n$ matrices into $n/2 \times n/2$ matrices until they are small enough to be multiplied in the naive way, more specifically into 8 multiplications and 4 matrix additions. Now let us complete the pseudocode of the algorithm.

Pseudocode of the Matrix Multiplication Algorithm

1. $n \leftarrow$ no. Of rows of A
2. If $n=1$ then return (a11 b11)
3. Else
4. Let A_{ij} , B_{ij} (for $i,j = 1,2$ be $\left(\frac{n}{2} * \frac{n}{2}\right)$ submatrices)

$$\text{S.t } \mathbf{A} = \left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array} \right) \text{ and } \mathbf{B} = \left(\begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{21} & B_{22} \end{array} \right)$$

5. Recursively compute $A_{11}B_{11}$, $A_{12}B_{21}$, $A_{11}B_{12}$,..... $A_{22}B_{22}$

6. Compute

$$C_{11} = A_{11}B_{11} + A_{12} + B_{21}$$

$$C_{12} = A_{11}B_{12} + A_{12} + B_{22}$$

$$C_{21} = A_{21}B_{11} + A_{22} + B_{21}$$

$$C_{22} = A_{21}B_{12} + A_{22} + B_{22}$$

7. Return $\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$

Analysis of Divide and Conquer based Matrix Multiplication

Let $T(n)$ be the no. Of arithmetic operations performed by D&C-MATMUL.

- Line 1,2,3,4,7 require $\Theta(1)$ arithmetic operations.
- Line 5, requires $8T(n/2)$ arithmetic operations.

(i.e. In order to compute AB using e.g. (2), we need 8- multiplications of $\left(\frac{n}{2} \times \frac{n}{2}\right)$ matrices).

- Line 6 requires $4(n/2) = \theta(n^2)$
(i.e. 4 additions of $\left(\frac{n}{2} \times \frac{n}{2}\right)$ matrices)

So the overall computing time, $T(n)$, for the resulting Divide and conquer Matrix Multiplication

$$T(n) = 8T\left(\frac{n}{2}\right) + \theta(n^2)$$

Using Master method, $a=8$, $b=2$ and $f(n)=n^2$; since

$$f(n) = n^2 = O(n^{\log_2 8}) \Rightarrow \text{case 1 of master method}$$

$$\Rightarrow T(n) = Q\left(n^{\log_2 8}\right) = Q(n^3)$$

For multiplying two matrices of size $n \times n$, we make 8 recursive calls above, each on a matrix/sub-problem with size $n/2 \times n/2$. Each of these recursive calls multiplies two $n/2 \times n/2$ matrices, which are then added together. For the addition, we add two matrices of size $\frac{n^2}{4}$, so each addition takes $\Theta\left(\frac{n^2}{4}\right)$ time.

Thus, recurrence relation will be:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 8T(n/2) + \Theta(n^2) & \text{if } n > 1. \end{cases}$$

From the Case 1 of Master's Theorem, the time complexity of the above approach is $O(n \log 28)$ or $O(n^3)$ which is the same as the naive method of matrix multiplication.

Strassen's Matrix Multiplication Algorithm:

Greedy Technique

Strassen's algorithm makes use of the same divide and conquer approach as above, but instead uses only 7 recursive calls rather than 8 as shown in the equations below. Here we save one recursive call, but have several new additions and subtractions of $n/2 \times n/2$ matrices.

1. Divide the input matrices A and B into $n/2 \times n/2$ sub-matrices, which takes $\Theta(1)$ time.
2. Now calculate the 7 sub-matrices M1-M7 by using below formulas:

$$M1 = (A11 + A22) (B11 + B22)$$

$$M2 = (A21 + A22) B11$$

$$M3 = A11 (B12 - B22)$$

$$M4 = A22 (B21 - B11)$$

$$M5 = (A11 + A12) B22$$

$$M6 = (A21 - A11) (B11 + B12)$$

$$M7 = (A12 - A22) (B21 + B22)$$

3. To get the desired sub-matrices C11, C12, C21, and C22 of the result matrix C by adding and subtracting various combinations of the M_i sub-matrices. These four sub-matrices can be computed in $\Theta(n^2)$ time.

$$C11 = M1 + M4 - M5 + M7$$

$$C12 = M3 + M5$$

$$C21 = M2 + M4$$

$$C22 = M1 - M2 + M3 + M6$$

Using the above steps, we get the recurrence of the following format:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 7T(n/2) + \Theta(n^2) & \text{if } n > 1. \end{cases}$$

Using master method: $a=7$, $b=2$ and $n^{\log b^a} = n^{\log 2^7} = n^{2.81}$ $f(n) = n^2$

$f(n) = n^2 = O(n^{\log 2 - \epsilon}) \Rightarrow$ case 1 of master method

$\Rightarrow T(n) = Q(n^{\log b^a}) = Q(n^{\log 2^7}) = Q(n^{2.81})$.

Ex:- To perform the multiplication of A and B

$$AB = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 6 & 0 & 3 \\ 4 & 1 & 1 & 2 \\ 0 & 3 & 5 & 0 \end{bmatrix} \begin{bmatrix} 1 & 4 & 2 & 7 \\ 3 & 1 & 3 & 5 \\ 2 & 0 & 1 & 3 \\ 1 & 4 & 5 & 1 \end{bmatrix}$$

We define the following eight $n/2$ by $n/2$ matrices:

$$\begin{aligned} A_{11} &= \begin{bmatrix} 1 & 2 \\ 0 & 6 \end{bmatrix} & A_{12} &= \begin{bmatrix} 3 & 4 \\ 0 & 3 \end{bmatrix} & B_{11} &= \begin{bmatrix} 1 & 4 \\ 3 & 1 \end{bmatrix} & B_{12} &= \begin{bmatrix} 2 & 7 \\ 3 & 5 \end{bmatrix} \\ A_{21} &= \begin{bmatrix} 4 & 1 \\ 0 & 3 \end{bmatrix} & A_{22} &= \begin{bmatrix} 1 & 2 \\ 5 & 0 \end{bmatrix} & B_{21} &= \begin{bmatrix} 2 & 0 \\ 1 & 4 \end{bmatrix} & B_{22} &= \begin{bmatrix} 1 & 3 \\ 5 & 1 \end{bmatrix} \end{aligned}$$

Strassen showed how the matrix C can be computed using only 7 block multiplications and 18 block additions or subtractions (12 additions and 6 subtractions):

$$p_1 = (A_{11} + A_{22})(B_{11} + B_{22})$$

$$p_2 = (A_{21} + A_{22})B_{11}$$

$$p_3 = A_{11}(B_{12} - B_{22})$$

$$p_4 = A_{22}(B_{21} - B_{11})$$

$$p_4 = A_{22}(B_{21} - B_{11})$$

$$p_5 = (A_{11} + A_{12}) \cdot B_{22}$$

$$p_6 = (A_{21} - A_{11})(B_{11} + B_{12})$$

$$p_7 = (A_{12} - A_{22})(B_{21} + B_{22})$$

$$C_{11} = P_1 + P_4 - P_5 + P_7$$

$$C_{12} = P_3 + P_5$$

$$C_{21} = P_2 + P_4$$

$$C_{22} = P_1 + P_3 - P_2 + P_6$$

The correctness of the above equations is easily verified by substitution.

$$p_1 = (A_{11} + A_{22}) * (B_{11} + B_{22})$$

$$= \begin{bmatrix} 1 & 2 \\ 0 & 6 \end{bmatrix} + \begin{bmatrix} 1 & 2 \\ 5 & 0 \end{bmatrix} * \begin{bmatrix} 1 & 4 \\ 3 & 1 \end{bmatrix} + \begin{bmatrix} 1 & 3 \\ 5 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} 2 & 4 \\ 5 & 6 \end{bmatrix} * \begin{bmatrix} 2 & 7 \\ 8 & 2 \end{bmatrix} = \begin{bmatrix} 36 & 22 \\ 58 & 47 \end{bmatrix}$$

$$P_2 = (A_{21} + A_{22}) * B_{11} = \begin{bmatrix} 14 & 23 \\ 14 & 23 \end{bmatrix}$$

$$P_3 = A_{11} * (B_{12} - B_{22}) = \begin{bmatrix} -3 & 12 \\ -12 & 24 \end{bmatrix}$$

$$P_4 = A_{22} * (B_{21} - B_{11}) = \begin{bmatrix} -3 & 2 \\ 5 & -20 \end{bmatrix}$$

$$P_5 = (A_{11} * A_{12}) * (B_{11} + B_{12}) = \begin{bmatrix} 34 & 18 \\ 45 & 9 \end{bmatrix}$$

$$P_6 = (A_{21} - A_{11}) * (B_{11} + B_{12}) = \begin{bmatrix} 3 & 27 \\ -18 & -18 \end{bmatrix}$$

$$P_7 = (A_{12} - A_{11}) * (B_{21} + B_{22}) = \begin{bmatrix} 18 & 16 \\ 3 & 0 \end{bmatrix}$$

$$C_{11} = P_{21} + P_4 - P_5 + P_7 = \begin{bmatrix} 17 & 22 \\ 21 & 18 \end{bmatrix}$$

$$C_{12} = P_3 + P_5 = \begin{bmatrix} 31 & 30 \\ 33 & 33 \end{bmatrix}$$

$$C_{21} = P_2 + P_4 = \begin{bmatrix} 11 & 25 \\ 14 & 3 \end{bmatrix}$$

$$C_{22} = P_1 + P_3 - P_2 + P_6 = \begin{bmatrix} 22 & 38 \\ 14 & 30 \end{bmatrix}$$

$$C = \begin{bmatrix} 17 & 22 & 31 & 30 \\ 21 & 18 & 33 & 33 \\ 11 & 25 & 22 & 38 \\ 19 & 3 & 14 & 30 \end{bmatrix}$$

The overall time complexity of Strassen's Method can be written as:

$$T(n) = \begin{cases} 0(1) & \text{if } n = 1 \\ 7T\left(\frac{n}{2}\right) + 0(n^2) \end{cases}$$

Otherwise

$$a = 7; b = 2; f(n) = n^2$$

$$n^{\log_a b} = n^{\log_2 7} = n^{2.81}$$

$$f(n)n^2 = 0\left(n^{\log_2 7 - \epsilon}\right) \Rightarrow \text{case 1 of master method}$$

$$\Rightarrow T(n) = Q\left(n^{\log_2 7}\right)$$

$$= Q(n^{2.81})$$

$$\text{The solution of this recurrence is } T(n) = O\left(n^{\log_2 7}\right) = O(n^{2.81})$$

Generally Strassen's Method is not preferred for practical applications for following reasons:

- 1) The constants used in Strassen's method are high and for a typical application Naive method works better.
- 2) For Sparse matrices, there are better methods especially designed for them.

- 3) The sub matrices in recursion take extra space.
- 4) Because of the limited precision of computer arithmetic on non-integer values, larger errors accumulate in Strassen's algorithm than in Naive Method.

Check your Progress 3:

Question 1: Which designing approach is used in Strassen's matrix multiplication algorithm?

Question 2: What is the time taken by Strassen's algorithm to perform matrix multiplication?

Question 3: In Strassen's matrix multiplication algorithm ($C = AB$), the matrix C can be computed using only 7 block multiplications and 18 block additions or subtractions. How many additions and how many subtractions are there out of 18?

Question 4: Use Strassen's matrix multiplication algorithm to multiply the following two matrices:

$$P = \begin{bmatrix} 5 & 3 \\ 6 & 7 \end{bmatrix}$$

$$Q = \begin{bmatrix} 1 & 4 \\ 5 & 9 \end{bmatrix}$$

2.7 Summary

- **Divide and Conquer** approach follows a recursive approach which making a recursive call to itself until a base (or boundary) condition of a problem is not reached but with reduced problem size.
- **Divide and Conquer** is a top-down approach, which consists of three steps:

Divide: the given problem is break down into smaller parts.

Conquer: Solve each sub-problem by recursively calling them.

Combine: each sub-solution is combined to generate solution to the original problem.

- Divide-and-conquer approach is widely applicable on problems like Binary search, Quick-Sort, Merge-Sort, Matrix multiplication.
- Binary search is the procedure of finding the location of an element in a sorted array by dividing the array into two halves recursively.
- Merge-Sort algorithm is a divide-and-conquer based sorting algorithm which recursively partitions an array into several sub-arrays until each sub-array consists of single element. Then, each sub-array is merged

- ## 2.8 Solution to Check Your Progress

Solution 1: 147.1 to 148.1

Check Your Progress 2

Solution 1: 2, 4, 5, 6, 7

Solution 2:

- Solution 3:** Considering pivot element as 7, following sequence of steps will be followed.

- Therefore, the output is: 6 3 4 7 9 14 11 12

Solution 4: 31.11 sec Approximately. The Quick sort requires $O(n \log n)$ comparisons in best case, where 'n' is size of input array. So, $1000 * \log 1000 \approx 9000$ comparisons are required to sort 1000 elements, which takes 200 sec. To sort 200 elements minimum of $200 * \log 200 \approx 1400$ comparisons are required. This will take $200 * 1400 / 9000 \approx 31.11$ sec.

Check Your Progress 3

Solution 1: Divide and Conquer

Solution 2: $O(n^{2.81})$

Solution 3: 12 and 6

Solution 4: $R = \begin{bmatrix} 23 & 52 \\ 48 & 93 \end{bmatrix}$

2.9 Further Readings

1. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, Introduction to Algorithms, MIT Press, 3rd Edition, 2009.
2. Steven Skiena, The Algorithm Design Manual, Springer; 2nd edition, 2008.
3. Knuth, The art of Computer Programming Volume 1, Fundamental Algorithms, Addison-Wesley Professional; 3 edition, 1997.
4. Horowitz and Sahni, Fundamentals of Computer Algorithms, Computer Science Press, 2008.
5. Sedgewick, Algorithms in C, 3rd edition. Addison Wesley, 2002.

Structure

3.0 Introduction	1
3.1 Objectives	1
3.2 Basic definition and Terminologies	2
3.3 Graph Representation Schemes	5
3.3.1 Adjacency Matrix	6
3.3.2 Adjacency List	7
3.4 Graph Traversal Schemes	8
3.4.1 Depth First Search	8
3.4.2 Breadth First search	9
3.5 Directed Acyclic Graph and Topological Ordering	11
3.6 Strongly Connected Components	16
3.7 Summary	17
3.8 Solution to Check Your Progress	17
3.9 Further Readings	18

3.0 Introduction

Graphs are most widely used mathematical structure. It is widely used in finding shortest path routes, shortest path between every pair of vertices, in computing maximum flow problem which has applications in a large range of problems related to airlines scheduling, maximum bipartite matching and image segmentation.

A graph can be used to model a social network which comprises millions of users or interest groups which can be represented as nodes. There are interdependencies among nodes through mutual interests and common friends. Many algorithms used in social networks are based on graph algorithms like Facebook's friends suggestion algorithm, Google's page ranking algorithm, Linkdn's suggestion to join a group etc.

3.1 Objectives:

After successful completion of this unit, the students will able to:

- Define different types of graphs
- Represent a graph through an adjacency matrix and an adjacency list and calculate complexities of each
- Write pseudo-codes for graph traversal techniques like breadth first search and depth first search and measure their time complexities
- Define directed acyclic graph , topological ordering and connected components
- Write pseudo codes for topological ordering and connected components

Graph:

A graph $G = (V, E)$ is a data structure comprising two set of objects, $V = \{v_1, v_2, \dots\}$ called vertices, and an another set $E = \{e_1, e_2, \dots\}$ called the edges.

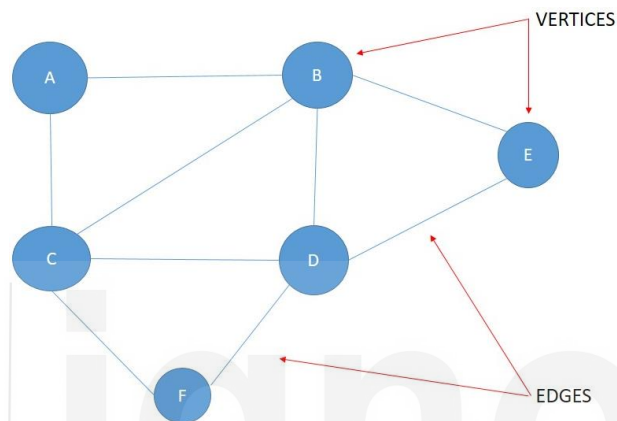


Figure 1: A graph

In the above graph set of vertices $V = \{A, B, C, D, E, F\}$ and set of edges $E = \{A-B, A-C, B-E, B-D, B-C, C-D, C-F, D-F, D-E, E-F\}$. Vertices are unordered set of nodes V . **Edge** = An edge is identified with an unordered pair of vertices (v_i, v_j) , where v_i and v_j are the end vertices of the edge e_k .

Graph Types:

Though a graph contains only vertices and edges, but there are many variations in them. Most of the variations are due to edges (directed / undirected, no of edges). In the following such graph types are listed with examples.

1. **Simple Graph:** A simple graph (figure 2) is a graph in which each edge is connected with two different vertices and no two edges are connected with same vertices. In this there is no self-loop and no parallel edges in a graph. A simple graph may be connected or disconnected. Basically the un-weighted graphs are simple graphs. A simple graph with multiple edges is called multigraph (figure 3).

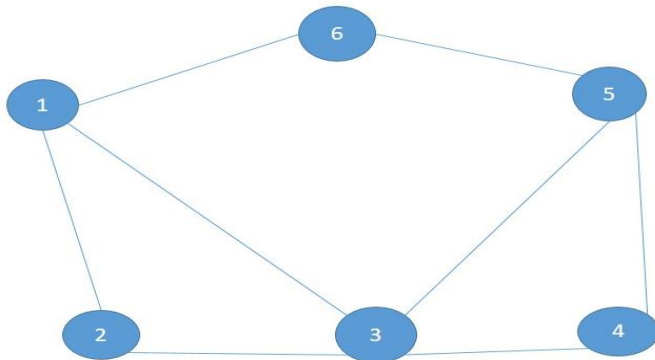
Example:

Figure 2: Simple Graph

The above graph is a simple graph, since no vertex has a self-loop and no two vertices have more than one edge connecting them.

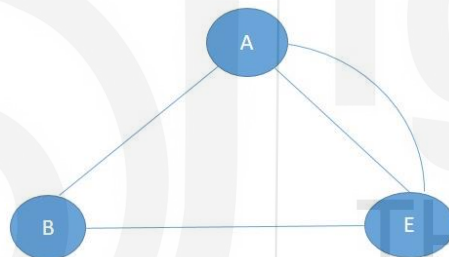
Example: Multi graph

Figure 3: Multi Graph

1. **Undirected Graph:** A graph in which the edges do not have any direction and all the edges are in bi-direction (figure 4). In undirected graph if there is an edge from u to v then we can move from node u to node v and as well as from node v to node u . In this nodes are unordered pairs. We can transform the undirected into directed graph by separating the edges between two edges.

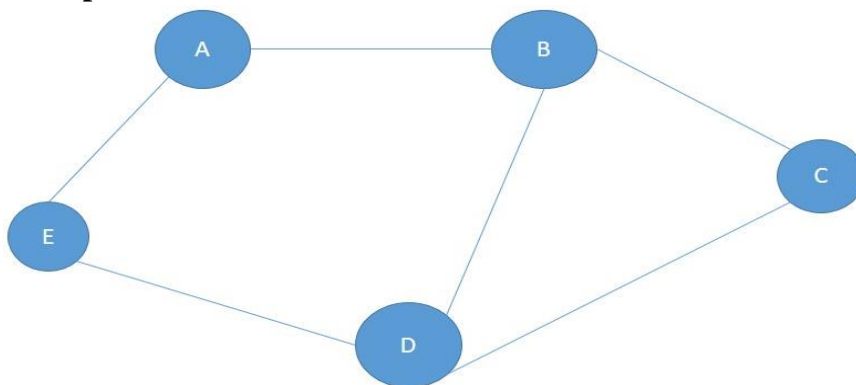
Example:

Figure 4: Undirected Graph

In the above undirected graph we can traverse through following paths:

From A: A – E, A – B
From B: B – A, B – D, B – C
From C: C – B, C – D
From D: D – E, D – B, D – C
From E: E – A, E – D

- 1. Directed Graph:** A graph in which the edges have direction (figure 5). It is also called digraph. This is usually indicated with an arrow on the edge. In a directed graph if there is an edge from u to v then we can move from a node u to a node v only. With the help of directed graph we can represent asymmetrical relationships between nodes, roads network, hyperlinks connecting web pages etc.

Example:

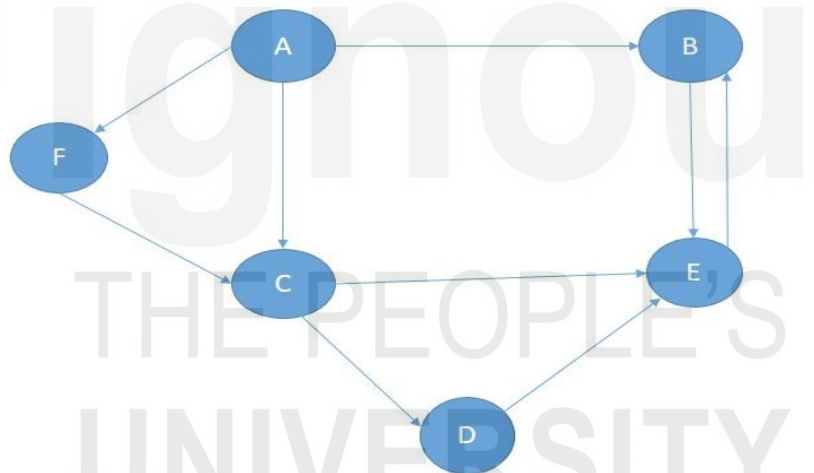


Figure 5: Directed Graph

In the above directed graph we can traverse through following paths:

From A: A – F, A – C, A – B
From B: B – E
From C: C – E, C – D
From D: D – E
From E: E – B
From F: F – C

- 2. Subgraph:** A graph whose vertices and edges are subsets of a another graph. It is not necessary that a subgraph will have all the edges of graph.

Example:

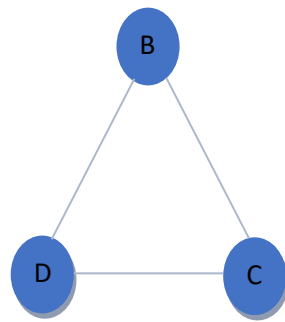


Figure 6: A sub graph of a figure-4

This is a subgraph of the graph which has the nodes A, B, C, D. In this there are C, B, D nodes.

1. **Connected Graph:** A directed graph in which there is a path between each pair of vertices in a subset. An undirected graph is said to be connected if for every pair of two different vertices v_i , v_j , there is a path between these two vertices. The graph G is connected whereas G_2 is not connected

Example:

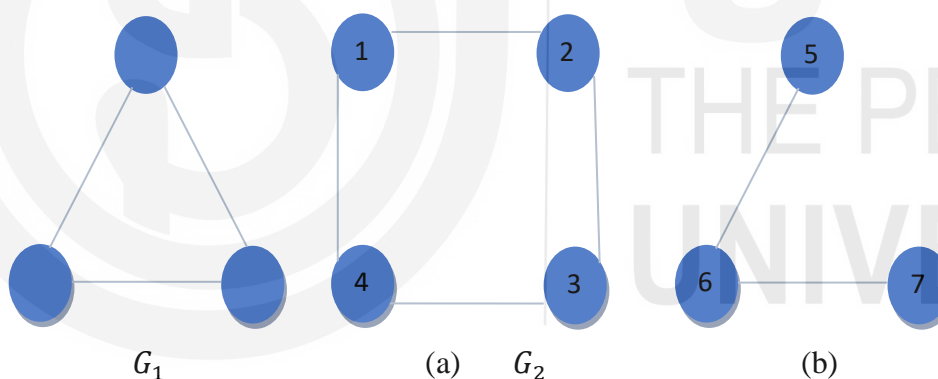


Figure 7: Connected graph

3.3 Graph Representation

The purpose of graph representation is typically to search a graph most systematically such that the edges of the graphs can be used to effectively visit all the vertices of it. In order to have an effective search algorithm, the logical representation of the graph plays a very critical role. When it comes to representations of graphs, two most standard and common computational representations are in practice such as Adjacency Matrix and Adjacency List. Apart from these other additional representations such as Linked lists, contiguous lists and combinations are also used in fewer cases. The selection of a particular

representation depends on applications and functions one wants to perform on these graphs.

In case the graph is sparse for which the number of edges also written as $|E|$ is quite less than $|V|^2$, adjacency list is preferred but if the graph is dense where $|E|$ is close to $|V|^2$ an adjacency matrix is selected.

3.3.1 Adjacency Matrix:

Adjacency matrix representation is typically used to represent both directed and undirected graphs, where the concentration of nodes in a graph is dense in nature. Usually in such graphs $|E|$ is much close to $|V|^2$. These adjacency matrices are typically drawn for Directed Acyclic Graph (DAG), where we have no loops or double directions.

- Adjacency matrix A is a square matrix which is used to represent a finite graph. The matrix elements show whether pairs of vertices are adjacent (connected) or not in the graph. It is a 2-dimensional array, which has the size $V \times V$, where V are the numbers of vertices present in the graph. Otherwise it
 $A[i, j] = 1$ if there is an edge between A_i & A_j
 $A[i, j] = 0$, otherwise

Adjacency Matrix of the following graph () is given below.

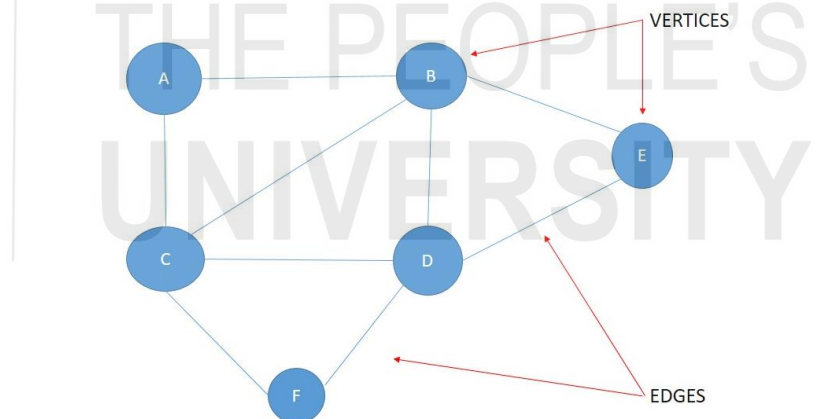


Figure 8: Example graph for matrix representation

In the given graph, if there is a link between the vertices we mark as '1' in the adjacency matrix, else we mark as '0'.

	A	B	C	D	E	F
A	0	1	1	1	0	0
B	1	0	1	1	1	0
C	1	1	0	1	0	1
D	0	1	1	0	1	1
E	0	1	0	1	0	0
F	0	0	1	1	0	0

Figure 9: Adjacency matrix representation of a graph given at figure 8

It is easy to determine if there is an edge between two vertices in a graph if it is represented through adjacency matrix. To find out how many edges are in a graph, it will require at least $O(v^2)$ time as there are v^2 entries of matrix have to be examined except the diagonal element which are zeroes. But when the most of the entries are O_s (in case of a sparse graph) then it might take $O(e + v)$ where $e \ll v^2$ which is significantly less time.

Adjacency matrix for a directed graph may or may not be symmetric but in case of undirected graph. It is always symmetric

Space complexity = $O(v^2)$ time complexity. Where V is the number of vertices is independent of a number of edges.

3.3.2 Adjacency List

Adjacency list representation is typically used to represent graphs, where the number edges $|E|$ is much less than $|V|^2$. Adjacency list is represented as an array of $|V|$ linked lists. There is one linked list for every vertex node in a linked list. Each Node in this linked list is a reference to the other vertices which share an edge with the current vertex. The following figure, (b) shows adjacency list of a given graph.

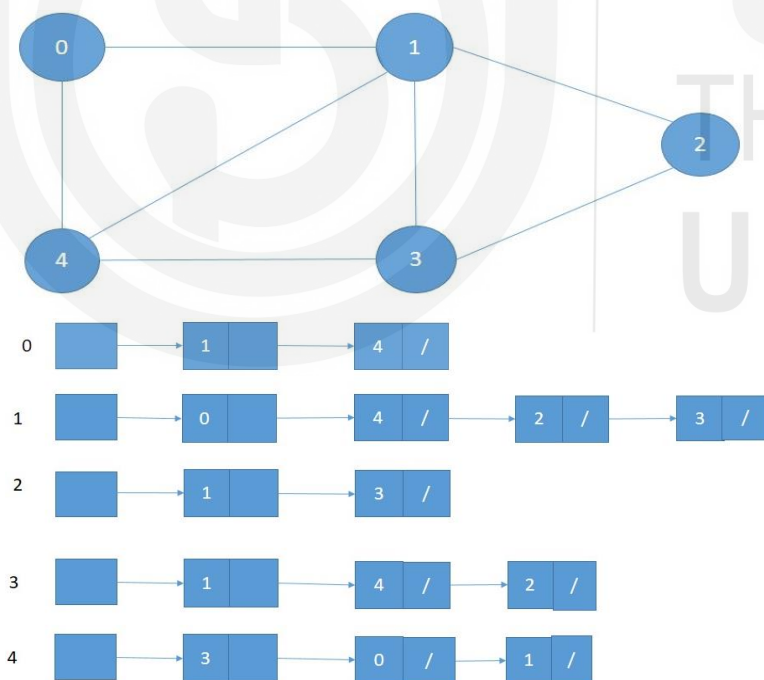


Figure 10 : An adjacency list of the above graph

If there is undirected with V vertices and E edges adjacency list representation requires $|V|$ head nodes and $2|E|$ adjacency list nodes. In an undirected edges (i, j) , i appear in j 's adjacency list and j appears in i 's adjacency list. In terms

of memory requires it is $O(V + E)$ for both types of graphs representations: adjacency matrix and adjacency list.

3.4 Graph Traversal Algorithms

Traversal algorithms are used to navigate across a given graph among all the nodes using all possible vertices. These algorithms will help us in finding the nodes, making paths that are shortest or feasible or prioritized in nature. Graph traversal can be carried out using breadth or depth as a criteria. In the data structure course we have learnt to traverse a tree in preorder, inorder and postorder. One encounters a similar problem of traversing a graph: given a graph $G = (V, E)$ one wants to visit all the vertices in G from a given vertex. There are two key graphs traversal Depth First Search (DFS) & Breadth First Search (BFS) algorithms. In the next section we will examine both algorithms.

3.4.1 Depth First Search (DFS)

DFS starts with any arbitrary vertex in a graph and traverses to the deepest node as far as possible and then backtracks. The algorithm proceeds as follows: it starts with selecting any arbitrary vertex as start vertex then traverses the next node w adjacent to the starting vertex v . The process of traversing the unexplored nodes adjacent to the previous node w and continues till it finds that no adjacent node is left to be examined. If the last vertex is u which does not have any adjacent vertices, it backtracks to w and explores the unvisited adjacent nodes.

The traversal process terminates when all the vertices are traversed. Recursive implementation of DFS pseudo-code is given below.

```
Function DFS( V ) // V is a starting vertex of a graph
Input  $G(V, E)$ , visited  $[n]$ , visited $[n]$  // visited $[n] = 0$ 
{
  visited  $[v] = 1$ 
  for each vertex  $w$  adjacent to  $v$  do
  {
    If visited  $[w] = 0$  // if vertex  $w$  is not yet visited
      DFS( $w$ ); // Recursive call to DFS
  }
} - // end of DFS ( )
```

Figure 11: pseudo-code of DFS

Let us apply the pseudo-code of DFS to the following example graph

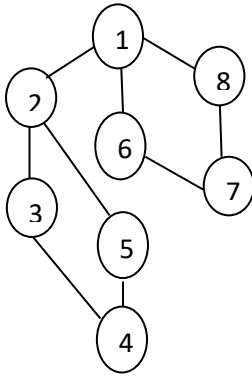


Figure 12 : An example graph for DFS traversal

Output of DFS traversal of a graph at figure 12: 1, 2, 3, 4, 5, 6, 7, 8

What is the running time of DFS ?

Suppose $G(V, E)$ is an undirected graph with v number of vertices and E edges and represented through adjacency list, then a sum of the length of all the adjacency list is $2|E|$.

DFS visits each node only once in the adjacency list. Therefore the time to complete visiting all edges and the associated vertices is $O(V+E)$. If a graph is represented through its adjacency matrix, the time to determine all vertices which are adjacent to the starting vertex v is V . Since at most V vertices are visited, the total running time is $O(V^2)$.

3.4.2 Breadth First Search (BFS)

BFS is a graph searching algorithm which start from any arbitrary vertex as a starting vertex and visits all its adjacent vertices first and then moves at the second level to visit all the unvisited vertices which are adjacent to vertices at the first level vertices and so on. BFS has become a basis for the development of many other graph algorithms such as Dijkstra's single source shortest path algorithm and Prim's minimum cost spanning tree problem. BFS algorithm starts with any arbitrary vertex 1 and then visits all its adjacency vertices 2,6,8(i.e nodes at the first level) first instead of discovering the deepest node as in DFS. At the next level (i.e the second level) it will visit the adjacent vertices of 2,6 and 8 respectively. The adjacent vertices of 2 are 3 and 5, the adjacent vertex of 6 and 8 is 7 respectively. Finally it will visit 4 which is a common adjacent vertex of 3 and 5 vertices.

The final output of BFS traversal of a graph is: 1, 2,6,8,3,5,7,4

The following algorithm describes the implementation details of BFS.

Queue is a main data structure used to implement the algorithm

```

function BFS (v)  // v is a starting node of a graph
1.  let Q be queue data structure .
2.  Q.insert(v)//Insert s in queue until all its neighbour vertices are marked.
3.  Visited[v] =1 // Visited[ ] will mark a vertex '1' if it is visited, otherwise
    it is 0
4.  print Q
4.  while( Q is not empty)
5.  //Remove that vertex from queue,whose neighbour will be visited now
6.  Q.remove()

    //processing all the neighbours of v
7.  for all neighbours w of v in a Graph G
8.  if w is not visited
9.  Q.insert(w)//Stores w in Q to further visit its neighbor and mark as
    visited.
10 Visited[ w] = 1
11 Print Q.

```

Figure 13: BFS Pseudo-code

Step wise explanation of Algorithm:

- Initially visited [] and Q(which represents a queue data structure)are empty because all vertices are unvisited
visited[] =
Q = Nil
- All visited vertices are marked as ' 1'
- BFS is initiated with a starting vertex 1 in the graph (figure) and marked as '1' in visited array and entered in Q.
visited [] =
Q =
- Print the visited vertex '1'(starting vertex) and remove it from Q
visited[] =
print: 1
Q :
- Insert non-visited adjacent vertices 2, 6,8 of the starting vertex 1 in Q and mark them as '1' in visited array
Visited[] =
Q :
- Insert adjacent vertices 3 and 5 of 2 in Q and mark them as '1'in visited[]
visited [] =
Q :
- Print 2 and remove it from Q
visited [] =

Print: 2

Q : 6,8,3,5

1. Insert a common vertex 7 of vertices 6,8 in Q and mark them as '1' in visited [].
visited[]=
Q : 6,8,3,5,7
2. Print 6 and 8 and remove 6 and 8 from Q
Visited[] =
Q : 3,5,7
Print : 6,8
3. Finally we insert 4 in Q which is adjacent to vertices 3 and 5 and mark it as '1' in visited[].
visited[] =
Q : 3,5,7,4
4. Print 3,5,7 and remove it from Q
Visited[] =
Q : 4
Print: 3,5,7
5. All vertices have been visited and vertex 4 does not have any adjacent node. So print 4 and remove it from Q
Visited []=
Q : Nil
Print: 4

The final output of a BFS traversal is 1,2,6,8,3,5,7,4

Complexities:

Time complexity: $O(V + E)$, where $O(V)$ is a total time taken to complete queue operations(insertion and deletion of vertices . Insertion and deletion of a single vertex takes $O(1)$ unit of time . Since there are V number of vertices in the graph, it will take $O(V)$ time. The time taken to traverse each adjacency list only once is $O(E)$. Therefore the total time is $O(V + E)$

Check Your progress -1

- Q1 Which is the most suitable graph representation scheme for a sparse graph?
- Q2 What is a simple graph?
- Q3 How does BFS graph traversal scheme work?

3.5 Directed Acyclic Graph and Topological Ordering

A directed graph without a cycle is called a directed acyclic graph (or a DAG (for short) which is a frequently used graph structure to represent

precedence relation or dependence is a network. The following is an example of a directed acyclic task graph

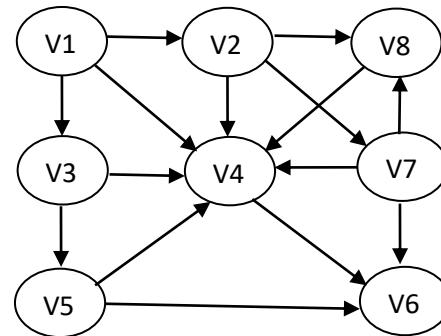


Figure 14: Directed Acyclic Graph

In this graph except vertex V1, all other vertices are dependent upon other vertices.

Any major task can be broken down into several subtasks. The successful completion of the task is possible only when all the subtasks are completed successfully. Dependencies among subtasks are represented through a directed graph. In such representation, subtasks are represented through vertices and an edge between two vertices defines a precedence relation. After showing dependency, the next task is to ordering is to ordering of these subtasks for execution. This is also called topological sorting or topological ordering.

Topological sort or ordering of a DAG $G = (V, E)$ is a linear ordering of its vertices V_1, V_2, \dots, V_n so that if a graph G contains an edge from v_i to v_j then v_i comes before v_j in the ordering. In other words, all edges between vertices representing tasks show forward direction in ordering or sorting. If the graph G contains a cycle then no topological order is possible.

Therefore, if a graph G has a topological sort, then G is a DAG.

Proof- Applying contradiction that G has a topological sorting of all its vertices (tasks): V_1, V_2, \dots, V_n and G is also having a cycle. Let all the vertices be indexed: V_1, V_2, \dots, V_n . Let V_i be the lowest index in topological ordering. Let there be an edge (V_j, V_i) in a cyclic graph G . If $V_j > V_i$ i.e. V_j comes before V_i which contradicts that all the vertices V_1, V_2, \dots, V_n are topologically sorted in G .

The following is a pseudo-code for computing a DAG.
Pseudo-code to compute topological sorting

```

Function topological_sort(G)
Input   $G = (V, E)$  //  $G$  is a DAG.
{
  search for a node  $V$  with zero in-degree (no incoming edges) and order it first in
  topological sorting
  remove  $V$  from  $G$ 
  topological_sort( $G - \{V\}$ ) // recursively compute topological sorting
  append the ordering
}

```

Figure 15: Pseudo-code for topological sorting

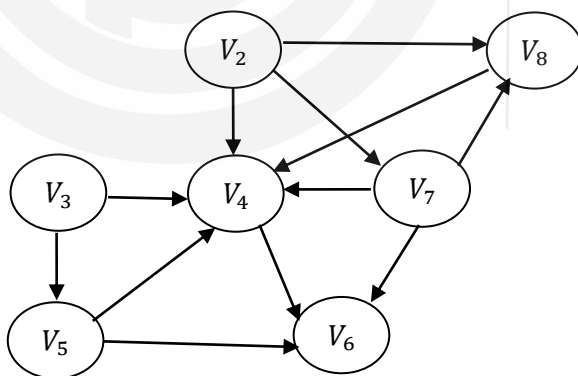
Topological sorting time complexity

Finding out a vertex with no incoming edge, deleting it from a graph and appending it in the linear ordering would take $O(n)$ time. Since there are n number of vertices in G , there will be n times loop, the total time will be $O(n^2)$. But if the graph is sparse where the $|E| \ll n^2$ and the graph is represented through an adjacency list it is possible to have $O(m + n)$ time complexity, where m is $|E|$.

Illustration of an example

The following figure show the application of the algorithm to the example given in the figure 15

Step 1: V_1 - does not have incoming edges so it is deleted first.

Figure 16 (a) : Deletion of V_1

Step 2 : In the graph there are the two nodes V_2 & V_3 with no incoming edges. One can pick up either of the two vertices. Let us remove V_2 and append it after V_1 , i.e., V_1, V_2

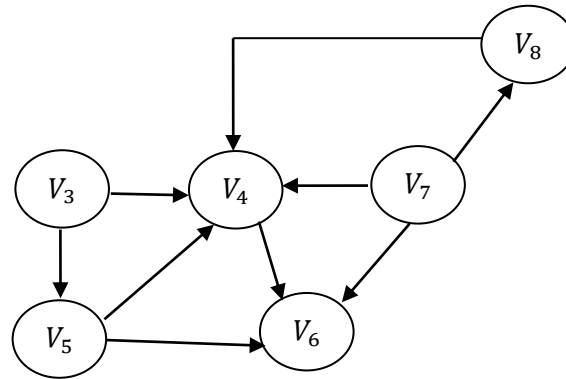


Figure 16(b): Deletion of V_2

Step 3: - V_3 is the only node with no incoming edge. Therefore V_3 will be removed and appended after V_2 , i.e., $V_1 V_2 V_3$

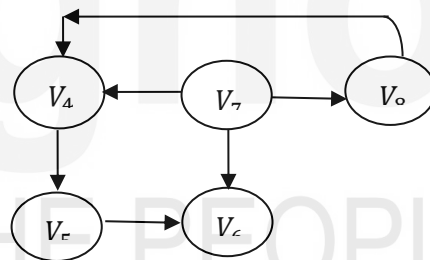


Figure 16(c): Deletion of V_3

Step 4 : V_5 is the only node with no incoming edge. V_5 will be removed and appended to the list ,i.e.,

$V_1 V_2 V_3 V_5$

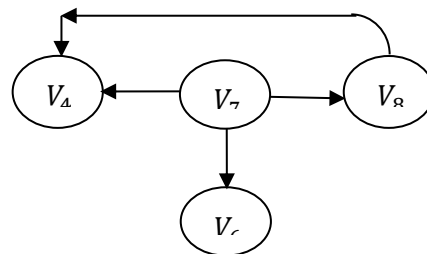
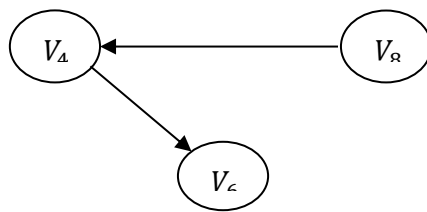


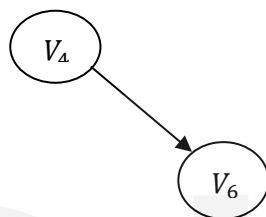
Figure 16(d): Deletion of V_5

Step 5 : V_7 is the only node with no incoming edge. Therefore V_7 will be removed and appended, i.e.,

$V_1 V_2 V_3 V_5 V_7$

Figure 16(e): Deletion of V_7

Step 6: - V_8 will be removed and appended, i.e., $V_1 V_2 V_3 V_5 V_7 V_8$

Figure 16(f) : Deletion of V_8

Step 7: V_4 will be removed and appended as: $V_1, V_2, V_3, V_5, V_7, V_8, V_4$



Step 8: Finally V_6 will be appended: $V_1, V_2, V_3, V_5, V_7, V_8, V_4, V_6$

The linear ordering of vertices is shown in the following figure:

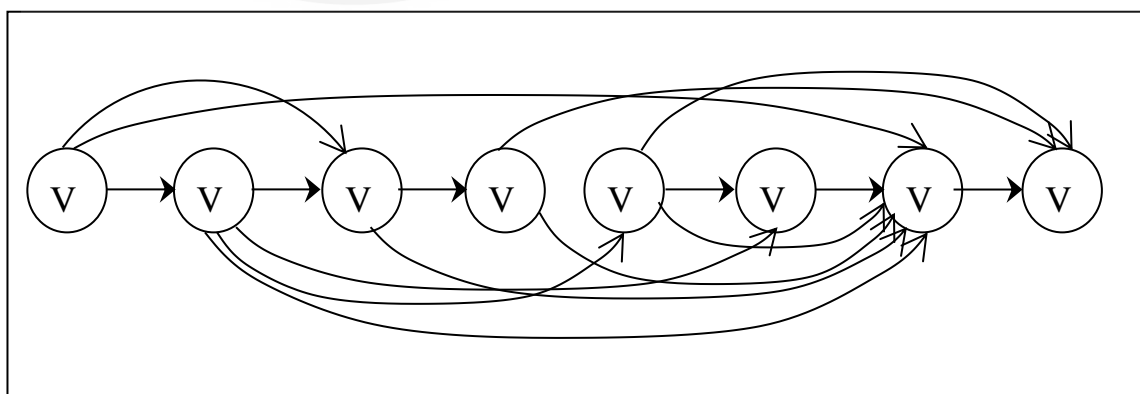


Figure 17: Topological Sorted order of vertices

3.6 Strongly Connected Components (SCC):

In this section we define two terms: strongly connected and strongly connected components of a directed graph then we apply an algorithm to find out whether a given directed graph is strongly connected component or not. A directed graph $G = (V, E)$ is strongly connected if for every two vertices v_i and v_j there is a pair of edges from v_i to v_j and v_j to v_i .

Strongly connected components is a maximal set of vertices $M \subseteq V$ such that every pair of vertices in M are mutually reachable. The following figure is an example of strongly connected components of a graph G .

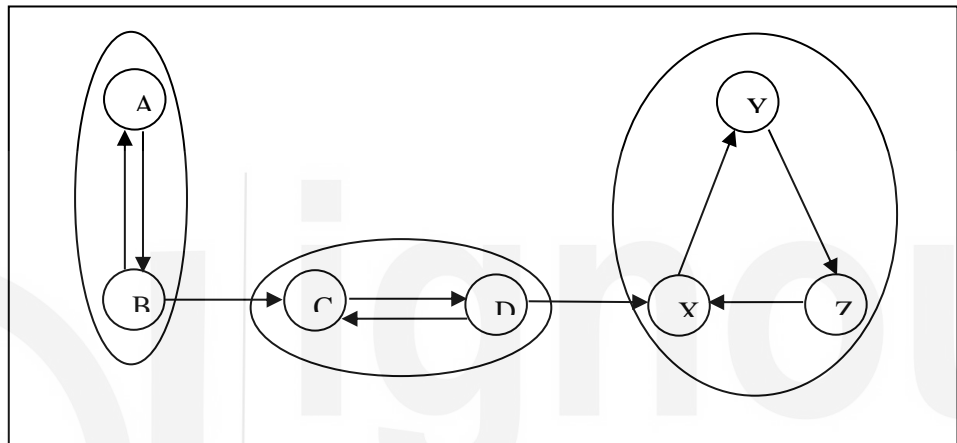


Figure 18:

In the above graph G there are 3 subsets of vertices, i.e., AB and CD and XYZ which are mutually reachable.

Pseudocode of Strongly Connected Components

- Perform DFS (Depth First Search) on the whole graph G
- Transpose of the original G (i.e. G^T)
- Perform DFS (Depth First Search) on $G^T = (V, E^T)$
- Print the final result.

Figure 19 : Strongly connected components

The proposed pseudocode for finding strongly connected components of a $G = (V, E)$ uses the graph traversal scheme DFS first and then performs transpose of a graph G , i.e., $G^T(V, E^T)$ with all the edges of G reversed. E^T consists of all the edges in reverse direction of G .

It is to be observed that a graph G and its transpose G^T have exactly the same strongly connected components.

Time Complexity :- If a graph is represented through adjacency lists, time to create G^T is $O(V + E)$

Greedy Technique

Check your progress -2

Q1. Define topological ordering.

Q2 If a graph G has a topological sort, then G is a *DAG*. Write a proof.

3.7 Summary

A graph is a very frequently used data structure for many basic and significant algorithms. There are two standard approaches to represent a graph: Adjacency matrix and adjacency lists which can be used to represent both directed as well as undirected graph. When the graph is a sparse, adjacency list is preferred because it is a more compact way to represent it. Many graph algorithms are based on the assumption that the graph is represented through an adjacency list. BFS and DFS are two simplest searching algorithms. BFS is a basis of Prim's minimum cost spanning tree problem and Dijkstra's single source shortest path algorithm. The unit also describes two applications of DFS: (i) directed acyclic graph and topological sorting (ii) decomposing a graph into strongly connected components.

3.8 Solution to Check Your Progress

Check Your progress -1

Q1 . Which is the most suitable graph representation scheme for a sparse graph?

Ans. Adjacency list provides a compact way to represent a sparse graph

Q2 What is a simple graph?

Ans. A simple graph is a graph in which each edge is connected with two different vertices and no two edges are connected with same vertices. In this there is no self-loop and no parallel edges in a graph. A simple graph may be connected or disconnected. Basically the un-weighted graphs are simple graphs. A simple graph with multiple edges is called multigraph

Q3 How does BFS graph traversal scheme work?

Ans. BFS is a graph searching algorithm which start from any arbitrary vertex as a starting vertex and visits all its adjacent vertices first and then moves at the second level to visit all the unvisited vertices which are adjacent to vertices at the first level vertices and so on

Check your progress -2

Q1. Define topological ordering.

Topological sort or ordering of a *DAG* $G = (V, E)$ is a liner ordering of its vertices $V_1, V_2 \dots V_n$ so that if a graph G contains an edge *from* v_i *to* v_j

then v_i comes before v_j in the ordering. In other words, all edges between vertices representing tasks show forward direction in ordering or sorting. If the graph G contains a cycle then no topological order is possible. Therefore, if a graph G has a topological sort, then G is a *DAG*.

Q2 If a graph G has a topological sort, then G is a *DAG*. Write a proof.

Proof- Applying contradiction that G has a topological sorting of all its vertices (tasks): $V_1, V_2 \dots V_n$ and G is also having a cycle. Let all the vertices be indexed: $V_1, V_2 \dots V_n$. Let V_i be the lowest index in topological ordering. Let there be an edge (V_j, V_i) in a cyclic graph G . If $V_j > V_i$ i.e. V_j comes before V_i which contradicts that all the vertices $V_1, V_2 \dots V_n$ are topologically sorted in G .

The following is a pseudo-code topological for computing a *DAG*.

3.9 Further Readings

1. Thomas H. Cormen, et al, Introduction to Algorithms, 3rd edition, Prentice Hall of India, 2012
2. John Kleinberg and Eva Tardos, Algorithm Design, Pearson, 2012