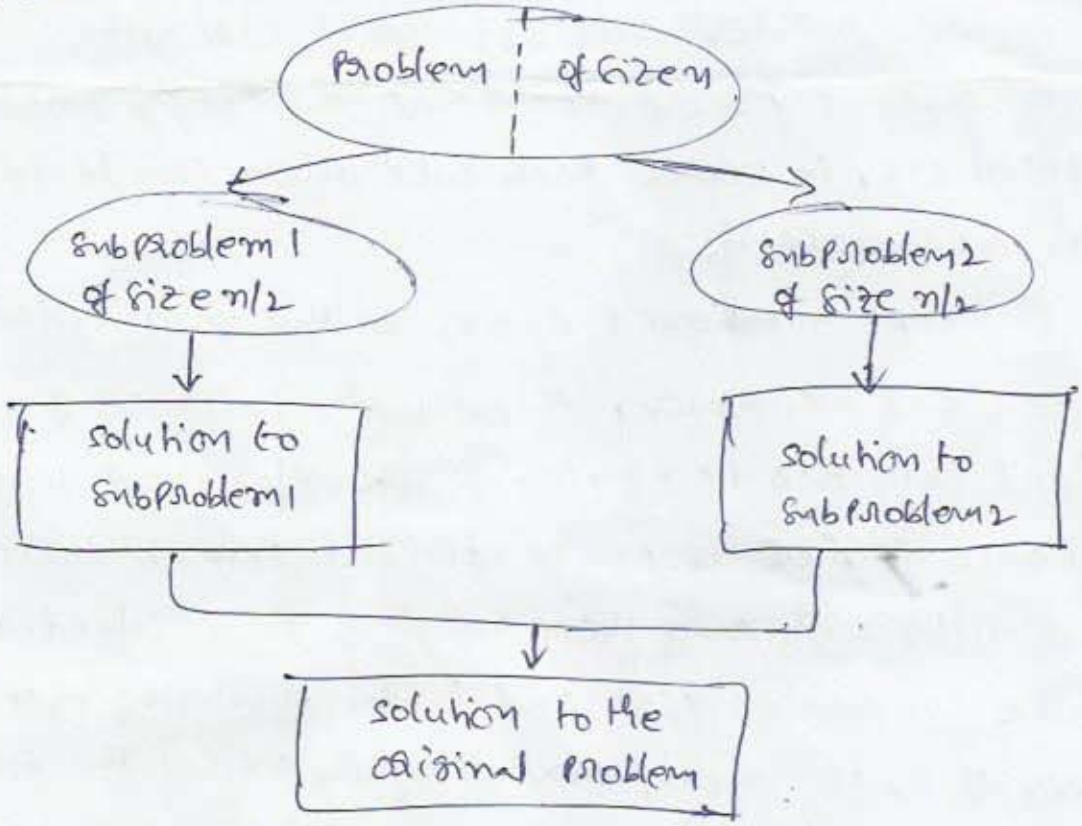
UNIT- II

DIVIDE - AND - CONQUER

Divide - and - conquer algorithms works according to the following general plan:

1. A problem's instance is divided into several smaller instances of the same problem, ideally of about the same size.
2. The smaller instances are solved.
3. If necessary, the solutions obtained for the smaller instances are combined to get a solution to the original instance.



Divide - and - conquer technique.

As an example, let us consider the problem of computing the sum of n numbers $a_0, \cdots, a_{n-1}$. If $n > 1$, we can divide the problem into two instances of the same problem:

to compute the sum of the first $\lfloor n/2 \rfloor$ numbers and to compute the sum of the remaining $\lceil n/2 \rceil$ numbers. (Of course, if $n = 1$, we simply return a_0 as the answer.) Once each of these two sums is computed, we can add their values to get the sum in question:

$$a_0 + \ldots + a_{n-1} = (a_0 + \ldots + a_{\lfloor n/2 \rfloor - 1}) + (a_{\lfloor n/2 \rfloor} + \ldots + a_{n-1})$$

Is this an efficient way to compute the sum of n numbers? A moment of reflection, a small example of summing, say, four numbers by this algorithm, a formal analysis, and common sense all lead to a negative answer to this question.

Thus, not every divide-and-conquer algorithm is necessarily more efficient than even a brute-force solution. Though we consider only sequential algorithms here, the divide-and-conquer technique is ideally suited for parallel computations, in which each subproblem can be solved simultaneously by its own processor.

As mentioned above, in the most typical case of divide-and-conquer, a problem's instance of size n is divided into two instances of size $n/2$. More generally, an instance of size n can be divided into b instances of size n/b, with a of them needing to be solved. (Here, a and b are constants; $a \geq 1$ and $b > 1$.) Assuming that size n is a power of b, to simplify our analysis, we get the following recurrence for the running time $T(n)$:

$$T(n) = a\,T(n/b) + f(n), \longrightarrow (1)$$

where $f(n)$ is a function that accounts for the time spent on the dividing the problem into smaller ones and on

combining their solutions. (For the summation example, $a = b = 2$ and $f(n) = 1$.) Recurrence (1) is called the general divide-and-conquer recurrence. Obviously, the order of growth of its solution $T(n)$ depends on the values of the constants a and b and the order of growth of the function $f(n)$. The efficiency analysis of many divide-and-conquer algorithms is greatly simplified by the following theorem (master theorem)

Theorem (master theorem) If $f(n) \in \Theta(n^d)$ with $d \geq 0$ in recurrence equation (1), then.

$$T(n) \in \begin{cases} \Theta(n^d) & \text{case 1} \quad \text{if } a < b^d \\ \Theta(n^d \log n) & \text{case 2} \quad \text{if } a = b^d \\ \Theta(n^{\log_b a}) & \text{case 3} \quad \text{if } a > b^d \end{cases}$$

(Analogous results hold for the O & Ω notations, too).

For ex, the recurrence equation for the number of additions $A(n)$ made by the divide-and-conquer sum-computation algorithm on inputs of size $n = 2^k$ is

$$A(n) = 2A(n/2) + 1.$$

Thus, for this example, $a = 2, b = 2$, and $d = 0$; hence, since $a > b^d$,

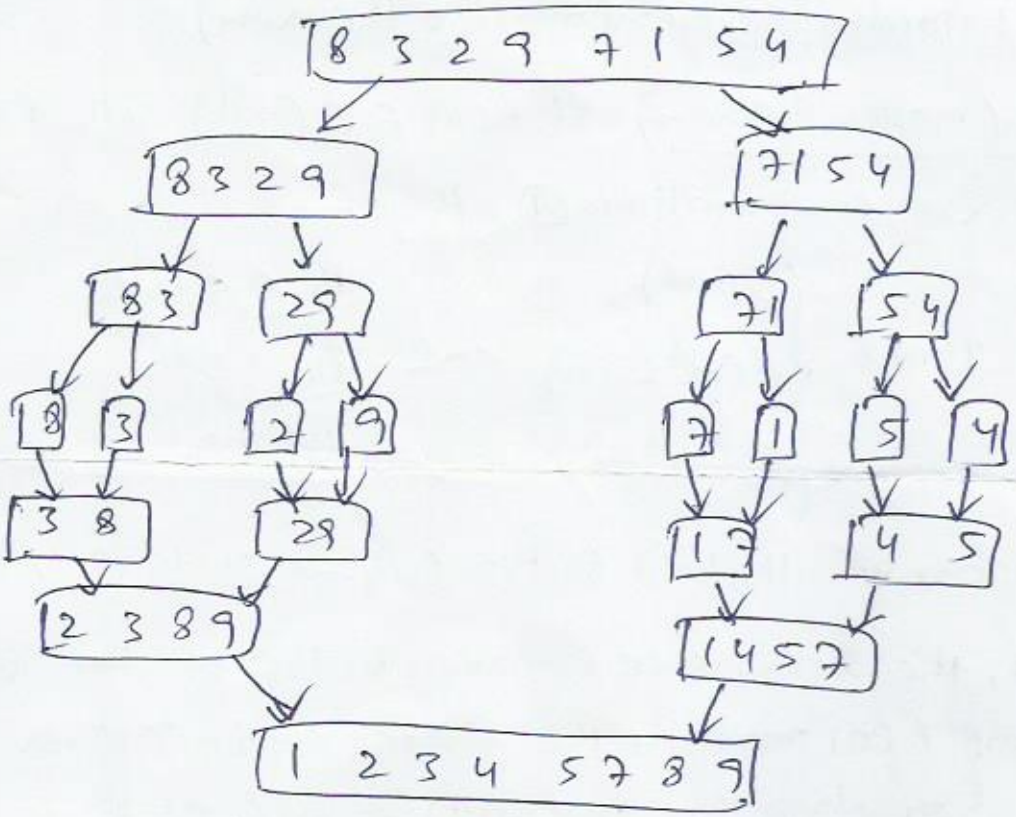
$$A(n) \in \Theta(n^{\log_b a}) = \Theta(n^{\log_2 2}) = \Theta(n).$$

Ex1: Find the order of growth for solutions of the following recurrences.

1. $T(n) = 4T(n/2) + n$, $T(1) = 1$. $d = 1 \xrightarrow{\text{case 3}} \Theta(n^{\log_2 4}) = \Theta(n^2)$
2. $T(n) = 4T(n/2) + n^2$, $T(1) = 1$. $d = 2 \xrightarrow{\text{case 2}} \Theta(n^2 \log n) = \Theta(n^2 \log n)$
3. $T(n) = 4T(n/2) + n^3$, $T(1) = 1$. $d = 3 \xrightarrow{\text{case 1}} \Theta(n^d) = \Theta(n^3)$.

Mergesort

Mergesort is a perfect example of a successful application of the divide-and-conquer technique. It sorts a given array $A[0 \cdots n-1]$ by dividing it into two halves $A[0 \cdots \lfloor n/2 \rfloor - 1]$ and $A[\lfloor n/2 \rfloor \cdots n-1]$, sorting each of them recursively, and then merging the two smaller sorted arrays into a single sorted one.



Example of mergesort operation.

Algorithm mergesort($A[0 \cdots n-1]$)
// Sorts array $A[0 \cdots n-1]$ by recursive mergesort.
// Input: An array $A[0 \cdots n-1]$ of orderable elements.
// output: Array $A[0 \cdots n-1]$ sorted in nondecreasing order.
if $n > 1$
    copy $A[0 \cdots \lfloor n/2 \rfloor - 1]$ to $B[0 \cdots \lfloor n/2 \rfloor - 1]$
    copy $A[\lfloor n/2 \rfloor \cdots n-1]$ to $C[0 \cdots \lceil n/2 \rceil - 1]$
    mergesort($B[0 \cdots \lfloor n/2 \rfloor - 1]$)
    mergesort($C[0 \cdots \lceil n/2 \rceil - 1]$)
    merge(B, C, A).

The merging of two sorted arrays can be done as follows: Two pointers (array indices) are initialized to point to the first elements of the arrays being merged. The elements pointed to are compared, and the smaller of them is added to a new array being constructed; after that, the index of the smaller element is incremented to point to its immediate successor in the array it was copied from. This operation is repeated until one of the two given arrays is exhausted, and then the remaining elements of the array are copied to the end of the new array.

Algorithm merge($B[0 \cdots p-1]$, $C[0 \cdots q-1]$, $A[0 \cdots p+q-1]$)
// Merges two sorted arrays into one sorted array.
// Input: Arrays $B[0 \cdots p-1]$ and $C[0 \cdots q-1]$ both sorted.
// Output: Sorted array $A[0 \cdots p+q-1]$ of the elements of B and C.

```
i ← 0; j ← 0; k ← 0
while i < p and j < q do
    if B[i] ≤ C[j]
        A[k] ← B[i]; i ← i+1.
    else
        A[k] ← C[j]; j ← j+1
    k ← k+1
if i == p
    copy C[j ... q-1] to A[k ... p+q-1].
else
    copy B[i ... p-1] to A[k ... p+q-1].
```

How efficient is mergesort? Assuming for simplicity that n is a power of 2, the recurrence relation for the

number of key comparisons $C(n)$ is

$$C(n) = 2C(n/2) + C_{merge}(n) \quad \text{for } n > 1, \; C(1) = 0.$$

let us analyze $C_{merge}(n)$, the number of key comparisons performed during the merge stage. At each step, exactly one comparison is made, after which the total number of elements in the two arrays still needed to be processed is reduced by one. In the worst case, neither of the two arrays becomes empty before the other one contains just one element (ex., smaller elements may come from the alternating arrays). Therefore, for the worst case, $C_{merge}(n) = n-1$, and we have the recurrence.

$$C_{worst}(n) = 2C_{worst}(n/2) + n - 1 \quad \text{for } n > 1, \; C_{worst}(1) = 0.$$

Hence, according to the master theorem, $C_{worst}(n) \in O(n \log n)$.

Ex: Apply merge sort to sort the list E, X, A, M, P, L, E in alphabetical order.

Quicksort It is based on the divide-and-conquer approach. unlike mergesort, which divides its input's elements according to their position in the array, quicksort divides them according to their value. quicksort rearranges elements of a given array $A[0 \dots n-1]$ to achieve its partition. partition is a situation where all the elements ~~of a given array~~ $A[0], \dots, A[n-1]$ before some position s are smaller than or equal to $A[s]$ and all the elements after position s are greater than or equal to $A[s]$. i.e.,

$$\underbrace{A[0] \dots A[s-1]}_{\text{all are} \le A[s]} \quad A[s] \quad \underbrace{A[s+1] \dots A[n-1]}_{\text{all are} \ge A[s]}$$

Obviously, after a partition has been achieved, $A[s]$ will be in its final position in the sorted array, and we can continue sorting the two subarrays of the elements preceding & following $A[s]$ independently. (ex, by the same method)

ALGORITHM quicksort($A[l \dots r]$)
// sorts a subarray by quicksort.
// Input: A subarray $A[l \dots r]$ of $A[0 \dots n-1]$, defined by its left and right indices l and r.
// Output: subarray $A[l \dots r]$ sorted in nondecreasing order

```
if l < r
    s ← partition(A[l...r])   // s is a split position.
    quicksort(A[l...s-1])
    quicksort(A[s+1...r])
```

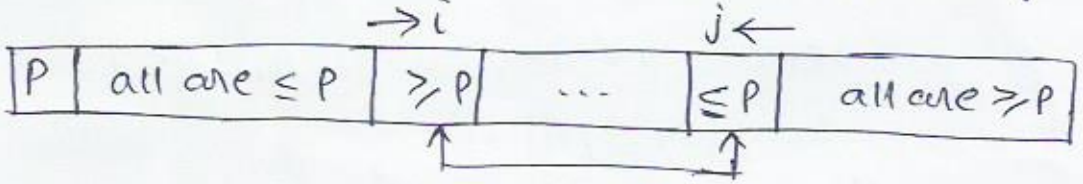
A partition of $A[0 \dots n-1]$ and its subarray $A[l \dots r]$ $(0 \le l < r \le n-1)$ can be achieved by the following algorithm.

First, we select an element w.r.to whose value we are going to divide the subarray. We call this element as the pivot. There are several different strategies for selecting a pivot; we will return to this issue when we analyze the algorithm's efficiency. For now, we use the simplest strategy of selecting the subarray's first element: $p = A[l]$.

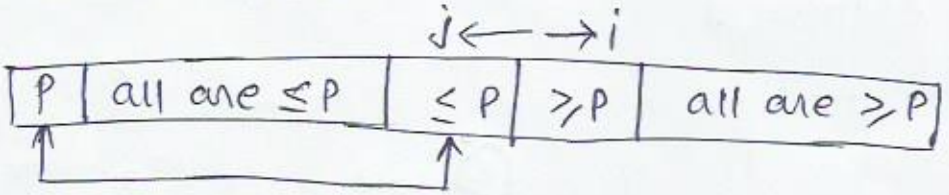
There are also several alternative procedures for rearranging elements to achieve a partition. Here we use an efficient method based on two scans of the subarray: one is left-to-right and the other right-to-left, each comparing the subarray's elements with the pivot. The left-to-right scan, denoted below by index i, starts with the second element. Since we want elements smaller than the pivot to be in the first part of the subarray, this scan skips over elements that are smaller than the pivot and stops on encountering the first element greater than or equal to the pivot. The right-to-left scan, denoted below by index j, starts with the last element of the subarray. Since we want elements larger than the pivot to be in the second part of the subarray, this scan skips over elements that are larger than the pivot and stops on encountering the first element smaller than or equal to the pivot.

After both scans stop, three situations may arise, depending on whether or not the scanning indices

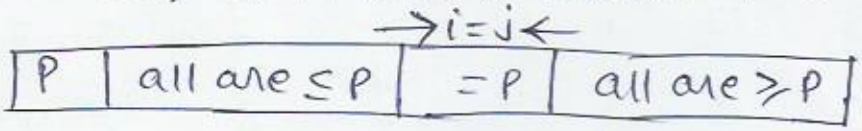
have crossed. If scanning indices i and j have not crossed, i.e., $i < j$, exchange $A[i]$ & $A[j]$ and resume the scans by incrementing i and decrementing j, respectively:



If the scanning indices have crossed over, i.e., $i > j$, you will have partitioned the array after exchanging the pivot with $A[j]$:



Finally, if the scanning indices stop while pointing to the same element, i.e., $i = j$, the value they are pointing to must be equal to p (why?). Thus, you have the array partitioned, with the split position $s = i = j$:



You can combine the last case with the case of crossed-over indices ($i > j$) by exchanging the pivot with $A[j]$ whenever $i \geq j$.

Pseudocode implementing this partitioning procedure.

```
Algorithm Partition( A[l...r] )..
// Partitions a subarray by using its first element as a pivot
// Input: A subarray A[l..r] of A[0...n-1], defined by
   its left and right indices l and r (l < r).
// Output: A partition of A[l...r], with the split position
   returned as this function's value.
```

```
P ← A[l]
i ← l;  j ← r+1.
repeat
    repeat i ← i+1 until A[i] ≥ P.
    repeat j ← j-1 until A[j] ≤ P.
    swap(A[i], A[j]).
until i ≥ j.
swap(A[i], A[j]) // undo last swap when i ≥ j
swap(A[l], A[j])
return j
```

Ex:

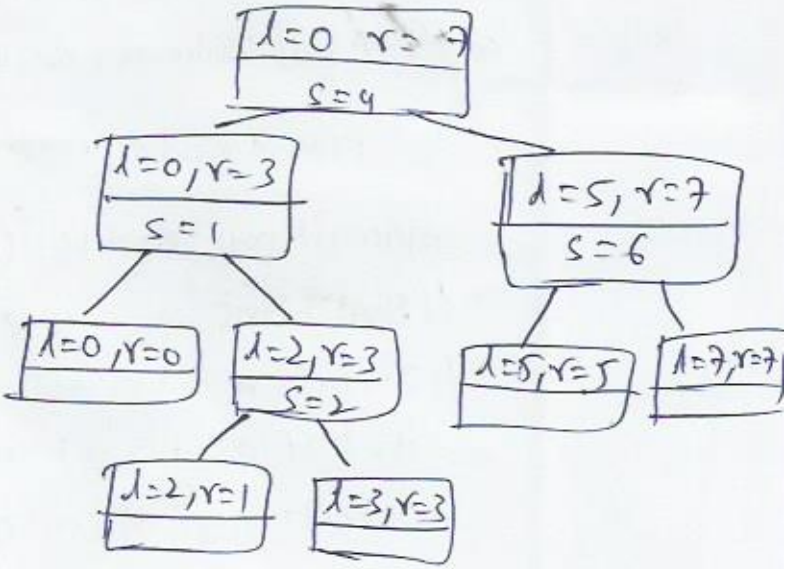
	0	1	2	3	4	5	6	7	
									l=0 r=7
Pivot	5	3	1	9	8	2	4	7	
	5	3	1	4	8	2	9	7	after swapping A[i] & A[j]
	5	3	1	4	8	2	9	7	
	5	3	1	4	2	8	9	7	after swapping A[i] & A[j]
	5	3	1	4	2	8	9	7	
	2	3	1	4	5	8	9	7	after swapping A[l] & A[j]

Subarray

Pivot 2	3	1	4	
	3	1	4	
	1	3	4	swap A[i] & A[j]
	1	3	4	
	1	2	3	swap A[l] & A[j]

Subarray

3	4	
3	4	
	4	



```
        i   j
8   9   7
        i   j
8   7   9
        j   i
8   7   9
7   8   9      swap A[j] & A[l]
7

        9
```

Example of quicksort

Quicksort efficiency analysis: If all the splits happen in the middle of corresponding subarrays, we will have the best case. The no. of key comparisons in the best case will satisfy the recurrence

$$C_{best}(n) = 2 \cdot C_{best}(n/2) + n \quad \text{for } n > 1, \quad C_{best}(1) = 0.$$

According to the master theorem, $C_{best}(n) \in \Theta(n \log_2 n)$; solving it exactly for $n = 2^k$ yields $C_{best}(n) = n \log_2 n$.

```
 0   1   2   3   4   5   6   7
15   12  13  11  20  18  22  14
          (7)  ↓split
```

apply recursion to left subarray

```
14   12  13  11      15      18  22  20
     (3) ↓split              5   ↓split (2)
     1   2   3              6   7
     11  12  13  14     18   22  20
         ↓split
(2)  0   1   2
     11  12  13
         ↓split
         1   2
(1)      12  13
```

apply recursion to right subarray

```
↓split
(1)  6   7
     20  22
```

Total no. of comparisons = 7+3+2+2+1+1 = 16.

no. of comparisons in each split is $n-1$ i.e., except pivot, no. of comparisons for remaining elements.

Strassen's matrix multiplications (multiplying two square matrices)

$$A = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix}_{2\times2} \qquad B = \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}_{2\times2}$$

$$C = A * B$$

$$= \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} * \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}$$

$$= \begin{bmatrix} a_{00}\cdot b_{00} + a_{01}\cdot b_{10} & a_{00}\cdot b_{01} + a_{01}\cdot b_{11} \\ a_{10}\cdot b_{00} + a_{10}\cdot b_{10} & a_{10}\cdot b_{01} + a_{11}\cdot b_{11} \end{bmatrix}$$

Brute-force algorithm need 8 multiplications & 4 additions. Reduce no. of multiplications to 7 by increasing additions/subtractions (proposed by V. Strassen).

$$= \begin{bmatrix} m_1 + m_2 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}.$$

where,

$m_1 = (a_{00}+a_{11}) * (b_{00}+b_{11})$ (p)

$m_2 = (a_{10}+a_{11}) * b_{00}$ (q)

$m_3 = a_{00} * (b_{01}-b_{11})$ (r)

$m_4 = a_{11} * (b_{10}-b_{00})$ (s)

$m_5 = (a_{00}+a_{01}) * b_{11}$ (t)

$m_6 = (a_{10}-a_{00}) * (b_{00}+b_{01})$ (u)

$m_7 = (a_{01}-a_{11}) * (b_{10}+b_{11})$ (v)

$c_{00} = m_1 + m_4 - m_5 + m_7$ & $p+s-t+v$

$c_{01} = m_3 + m_5$ & $r+t$

$c_{10} = m_2 + m_4$ & $q+s$

$c_{11} = m_1 + m_3 - m_2 + m_6$ & $p+r-q+u$

Apply Strassen's algorithm to compute:

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

Exiting the recursion when $n=2$, compute the product of 2-by-2 matrices by brute-force algorithm.

let A and B be two n-by-n matrices where n is a power of 2. we can divide A, B, and their product C into four n/2-by-n/2 submatrices each as follows

$$C = A * B$$

$$\begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} * \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix}$$

$$= \begin{bmatrix} A_{00}\cdot B_{00} + A_{01}\cdot B_{10} & A_{00}\cdot B_{01} + A_{01}\cdot B_{11} \\ A_{10}\cdot B_{00} + A_{11}\cdot B_{10} & A_{10}\cdot B_{01} + A_{11}\cdot B_{11} \end{bmatrix}$$

C_{00} can be computed as either $A_{00}\cdot B_{00} + A_{01}\cdot B_{10}$ or as $m_1 + m_4 - m_5 + m_7$ where m_1, m_4, m_5, & m_7 are found by strassen's formulas. If the seven products of n/2-by-n/2 matrices are computed recursively by the same method, we have strassen's algorithm for matrix multiplication.

asymptotic efficiency — If $M(n)$ is the no. of multiplications made by strassen's algorithm in multiplying two n-by-n matrices (where n is a power of 2), we get the following recurrence relation for it:

$$M(n) = 7\,M(n/2) \quad \text{for } n>1,$$
$$M(1) = 1 \quad n=1.$$

$M(n/2) = 7\cdot M(n/4)$ $M(n/4) = 7\cdot M(n/8)$ $M(n/8) = 7\cdot M(n/16)$.

. . . . . .

Apply Backward substitutions, we get.

$$M(n) = 7\,M(n/2)$$
$$= 7\cdot 7\,M(n/4)$$
$$= 7^2\cdot 7\,M(n/8)$$
$$\vdots$$
$$= 7^k\,M(n/2^k)$$

let $n = 2^k$ $\therefore k = \log_2 n$.

$$M(n) = 7^{\log_2 n}\,M(1)$$

$a=7\ b=2\ n=c$ $\left[\because a^{\log_b c} = c^{\log_b a}\right]$

$$\therefore M(n) = n^{\log_2 7} \simeq n^{2.807}.$$

Brute-force algorithm takes n^3

UNIT-3

The Greedy Method: Introduction, Huffman Trees and codes, Minimum Coin Change problem, Knapsack problem, Job sequencing with deadlines, Minimum Cost Spanning Trees, Single Source Shortest paths.

Q) Define the following terms.

i. Feasible solution ii. Objective function iii. Optimal solution

Feasible Solution: Any subset that satisfies the given constraints is called feasible solution.

Objective Function: Any feasible solution needs either maximize or minimize a given function which is called objective function.

Optimal Solution: Any feasible solution that maximizes or minimizes the given objective function is called an optimal solution.

Q) Describe Greedy technique with an example.

Greedy method constructs a solution to an optimization problem piece by piece through a sequence of choices that are:

- feasible, i.e. satisfying the constraints.
- locally optimal, i.e., it has to be the best local choice among all feasible choices available on that step.
- irrevocable, i.e., once made, it cannot be changed on subsequent steps of the algorithm.

For some problems, it yields a globally optimal solution for every instance.

The following is the general greedy approach for control abstraction of subset paradigm.

```
Algorithm Greedy(a,n)
//a[1:n] contains n inputs
{
  solution := Φ // initializes to empty
  for i:=1 to n do
  {
    x := select(a);
    if Feasible(solution x) then
       solution := union(solution, x)
  }
  return solution;
}
```

Eg. Minimum Coin Change:

Given unlimited amounts of coins of denominations d1 > … > dm , give change for amount n with the least number of coins.

here, d1 = 25c, d2 =10c, d3 = 5c, d4 = 1c and n = 48c

Greedy approach: At each step we take a maximum denomination coin which is less than or equal to remaining amount required.

Step 1: 48 – 25 = 23
Step 2: 23 – 10 = 13
Step 3: 13 – 10 = 03
Step 4: 03 – 01 = 02
Step 5: 02 – 01 = 01
Step 6: 01 – 01 = 00

Solution: <1, 2, 0, 3> i.e; d1 – 1coin, d2 – 2 coins, d3 – 0 coin and d4 – 3 coins.
Greedy solution is optimal for any amount and "normal" set of denominations.

Q) Explain Huffman tree and Huffman code with suitable example.

Huffman tree is any binary tree with edges labeled with 0's and 1's yields a prefix-free code of characters assigned to its leaves.

Huffman coding or prefix coding is a lossless data compression algorithm. The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters.

Algorithm to build Huffman tree:
// Input is an array of unique characters along with their frequency of occurrences and output is Huffman Tree.

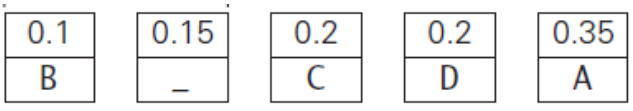
1. Create a leaf node for each unique character and build a min heap of all leaf nodes.
2. Extract two nodes with the minimum frequency from the min heap.
3. Create a new internal node with a frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.
4. Repeat step2 and step3 until the heap contains only one node. The remaining node is the root node and the tree is complete.

Time complexity: O(nlogn) where n is the number of unique characters. If there are n nodes, extractMin() is called 2*(n – 1) times. extractMin() takes O(logn) time as it calles minHeapify(). So, overall complexity is O(nlogn).
Eg.

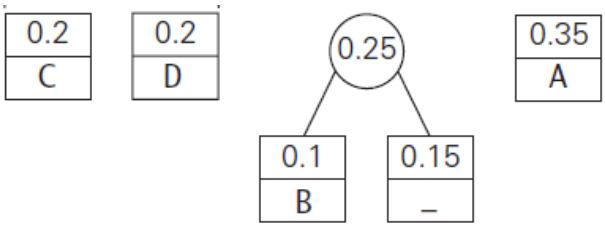
character	A	B	C	D	_
frequency	0.35	0.1	0.2	0.2	0.15

The code word for the character will be 001, 010, 011, 100 and 101 (fixed length encoding) without using Huffman coding, i.e; on an average we need 3 bits to represent a character.

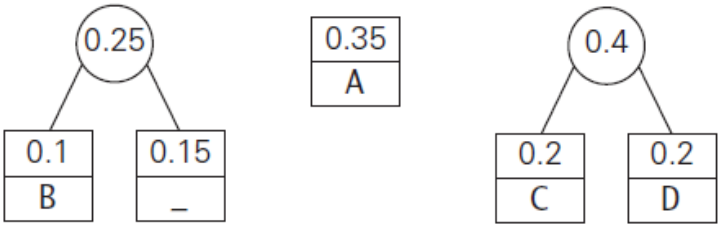
Step1:



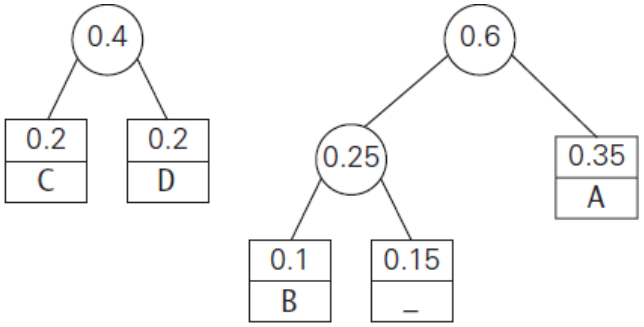
Step2:



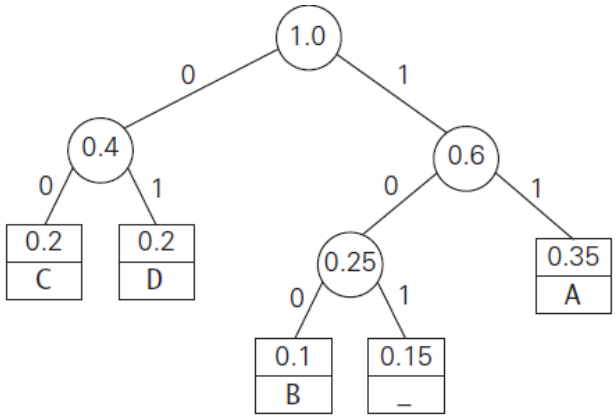
Step3:



Step4:



Step5:



Therefore, the codeword we get after using Huffman coding is

character	A	B	C	D	_
frequency	0.35	0.1	0.2	0.2	0.15
codeword	11	100	00	01	101

Average bits per character using Huffman coding
= 2*0.35 + 3*0.1 + 2*0.2 + 2*0.2 + 3*0.15
= 2.25
Therefore, compression ratio: (3 - 2.25)/3*100% = 25%

Q) Briefly explain about knapsack problem with an example.

Knapsack Problem

Given a set of items, each with a weight and a value, determine a subset of items to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.

Fractional Knapsack

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

According to the problem statement,

- There are **n** items in the store
- Weight of $\mathbf{i^{th}}$ item wi > 0
- Profit for $\mathbf{i^{th}}$ item pi>0 and
- Capacity of the Knapsack is **W**

In this version of Knapsack problem, items can be broken into smaller pieces. So, the thief may take only a fraction x_i of $\mathbf{i^{th}}$ item.

$$0 \le xi \le 1$$

The $\mathbf{i^{th}}$ item contributes the weight xi *wi to the total weight in the knapsack and profit xi.pi to the total profit.

Hence, the objective of this algorithm is to

$$\text{Maximize } \sum_{n=1}^{n} xi.\,pi$$

subject to constraint,

$$\sum_{n=1}^{n} xi.\,wi \le W$$

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

Thus, an optimal solution can be obtained by

$$\sum_{n=1}^{n} xi.\,wi = W$$

Algorithm Greedyknapsack(m,n)

```
//p[1:n] and w[1:n] contain the prfits and weights respectively
//all n objects are ordered p[i]/w[i] ≥ p[i+1]/w[i+1]
//m is the knapsack size and x[1:n] is the solution vector
{
  for i:=1 to n do
    x[i]:=0.0;
  u := m;
  for i:=1 to n do
  {
    if(w[i] > u) then
      break;
    x[i] := 1;
    u := u - w[i];
  }
  if(i ≤ n) then
    x[i]:= u/w[i];
}
```

Analysis

If the provided items are already sorted into a decreasing order of p_i/w_i, then the while loop takes a time in ***O(n)***; Therefore, the total time including the sort is in ***O(n logn)***.

Eg. 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

Step 1: find p/w ratio for each item.

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

Step2:

As the provided items are not sorted based on p_i/w_i. After sorting, the items are as shown in the following table.

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

Step3:

We choose 1st item B as weight of ***B*** is less than the capacity of the knapsack.

Now knapsack contains weight = 60 – 10 = 50

Step4:

item ***A*** is chosen, as the available capacity of the knapsack is greater than the weight of ***A***.

Now knapsack contains weight = 50 – 40 = 10

Step5:

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**

Eg 2. Find an optimal solution to the Knapsack instance $n=7$, $m=18$, $(P_1, P_2, \cdots P_7) = (15, 5, 6, 7, 16, 0, 1)$ & $(w_1, w_2, \cdots w_7) = (7, 4, 8, 2, 1, 4, 1)$

Given knapsack instance is

$n = 7$ (no. of items), $m = 18$ (capacity of Knapsack)

Profits $(P1, P2, \cdots P_7) = (15, 5, 6, 7, 16, 0, 1)$

Weights $(w1, w2, \cdots w_7) = (7, 4, 8, 2, 1, 4, 1)$

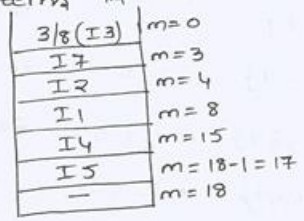
Let us find P_i/w_i for the items

Item, I	I1	I2	I3	I4	I5	I6	I7
Profit, P	15	5	6	7	16	0	1
Weight, w	7	4	8	2	1	4	1
P_i/w_i	2.14	1.25	0.75	3.5	16	0	1

After arranging the given items in non-decreasing order of P_i/w_i, we have

I	I5	I4	I1	I2	I7	I3	I6
P	16	7	15	5	1	6	0
w	1	2	7	4	1	8	4

The way of filling the knapsack with the given items is as shown below



∴ Solution is

$(x1, x2, \cdots x_7) = (1, 1, 3/8, 1, 1, 0, 1)$

Profit $= \sum_{i=1}^{7} P_i x_i$

$= 15(1) + 5(1) + 6(3/8) + 7(1) + 16(1) + 0(0) + 1(1)$

$= 46.25$

Q) Explain job sequencing with deadlines indetail with an example.

We are given a set of n jobs. Associated with job i is an integer deadline di ≥ 0 and a profit $p_i > 0$. For any job i, the profit p_i is earned iff the job is completed by its deadline.

To complete a job one has to process the job on a machine for one unit of time. Only one machine is available for processing jobs.

A feasible solution for this problem is a subset J of jobs such that each job in this subset can be completed by its deadline.

The value of a feasible solution J is the sum of the profits of the jobs in J. i.e; is $\sum_{i=J} Pi$

An optimal solution is a feasible solution with maximum value.

Eg.

Let $n = 4, (p_1, p_2, p_3, p_4) = (100, 10, 15, 27)$ and $(d_1, d_2, d_3, d_4) = (2, 1, 2, 1)$. The feasible solutions and their values are:

	feasible solution	**processing sequence**	**value**
1.	(1, 2)	2, 1	110
2.	(1, 3)	1, 3 or 3, 1	115
3.	(1, 4)	4, 1	127
4.	(2, 3)	2, 3	25
5.	(3, 4)	4, 3	42
6.	(1)	1	100
7.	(2)	2	10
8.	(3)	3	15
9.	(4)	4	27

The above is exhaustive technique in which we check all 1 and 2 jobs feasible possibilities and the optimal is 3rd sequence which is 4,1 sequence.

The following algorithm is a high level description of job sequencing:

```
Algorithm GreedyJob(d, J, n)
// J is a set of jobs that can be completed by their deadlines.
{
    J := {1};
    for i := 2 to n do
    {
        if (all jobs in J ∪ {i} can be completed
            by their deadlines) then J := J ∪ {i};
    }
}
```

The following JS is the correct implementation of above algorithm:

```
Algorithm JS(d, j, n)
// d[i] ≥ 1, 1 ≤ i ≤ n are the deadlines, n ≥ 1. The jobs
// are ordered such that p[1] ≥ p[2] ≥ ··· ≥ p[n]. J[i]
// is the ith job in the optimal solution, 1 ≤ i ≤ k.
// Also, at termination d[J[i]] ≤ d[J[i + 1]], 1 ≤ i < k.
{
     d[0] := J[0] := 0; // Initialize.
     J[1] := 1; // Include job 1.
     k := 1;
     for i := 2 to n do
     {
          // Consider jobs in nonincreasing order of p[i]. Find
          // position for i and check feasibility of insertion.
          r := k;
          while ((d[J[r]] > d[i]) and (d[J[r]] ≠ r)) do r := r − 1;
          if ((d[J[r]] ≤ d[i]) and (d[i] > r)) then
          {
               // Insert i into J[ ].
               for q := k to (r + 1) step −1 do J[q + 1] := J[q];
               J[r + 1] := i; k := k + 1;
          }
     }
     return k;
}
```

The above algorithm assumes that the jobs are already sorted such that P1 ≥ p2 ≥ ... ≥ pn. Further it assumes that n>=1 and the deadline d[i] of job i is atleast 1.

For the above algorithm JS there are 2 possible parameters in terms of which its time complexity can be measured.
1. the number of jobs, n
2. the number of jobs included in the solution J, which is s.

The while loop in the above algorithm is iterated atmost k times. Each iteration takes O(1) time.
The body of the conditional operator if require O(k-r) time to insert a job i.
Hence the total time for each iteration of the for loop is O(k). This loop is iterated for n-1 times.

If s is the final value of k, that is, S is the number of jobs in the final solution, then the total time needed by the algorithm is O(sn). Since s ≤ n, in worst case, the time complexity is $O(n^2)$

Eg. 1 Find an optimal solution using greedy method to the following instance of job sequencing with deadlines & profits problem.

$n=7$, $(P_1, P_2, \cdots P_7) = (3, 5, 20, 18, 1, 6, 30)$ and

$(d_1, d_2, \cdots d_7) = (1, 3, 4, 3, 2, 1, 2)$

Given instance of job sequencing with deadlines & profits is

$n=7$, $(P_1, P_2, \cdots P_7) = (3, 5, 20, 8, 1, 6, 30)$

$(d_1, d_2, \cdots d_7) = (1, 3, 4, 3, 2, 1, 2)$

After arranging the jobs in non-increasing order of profits

Jobs (J)	J7	J3	J4	J6	J2	J1	J5
Profits (P)	30	20	18	6	5	3	1
Deadlines (d)	2	4	3	1	3	1	2

- Initially we start with $J = \phi$ & profit value $\sum_{i \in J} P_i = 0$.
- we consider jobs in the above non-inc. order of Profits
- we add job, i to solution J if $J \cup \{i\}$ is feasible otherwise discard it.

S.No.	Job Considered	Action taken	Feasible solution	P	Processing Sequence	Profit value
1.	–	–	$J = \{\phi\}$		–	0
2.	7	added to J slot = (1-2)	$J = \{7\}$		7	30
3.	3	added to J slot = (3-4)	$J = \{3, 7\}$		7, 3	30+20 = 50
4.	4	added to J slot = (2-3)	$J = \{3, 4, 7\}$		7, 4, 3	50+18 = 68
5.	6	added to J slot = (0-1)	$J = \{3, 4, 6, 7\}$		6, 7, 4, 3	68+6 = 74
6	2	discarded	$J = \{3, 4, 6, 7\}$		6, 7, 4, 3	74
7	1	discarded	$J = \{3, 4, 6, 7\}$		6, 7, 4, 3	74
8	5	discarded	$J = \{3, 4, 6, 7\}$		6, 7, 4, 3	74

∴ Job Sequence is $J = \{6, 7, 4, 3\}$ with Profit = 74.

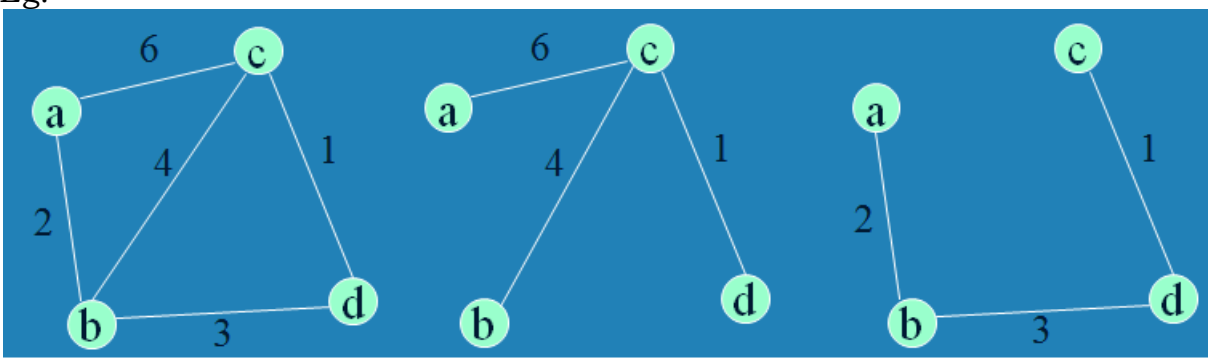
Q) What is minimum spanning tree?
i) Explain Prim's algorithm with an example.
ii) Explain Kruskal's algorithm with an example.

A **spanning tree** of an undirected connected graph is its connected acyclic subgraph (i.e., a tree) that contains all the vertices of the graph. If such a graph has weights assigned to its edges,

A **minimum spanning tree** is its spanning tree of the smallest weight, where the weight of a tree is defined as the sum of the weights on all its edges.

The **minimum spanning tree problem** is the problem of finding a minimum spanning tree for a given weighted connected graph.

Eg.



(a) (b) (c)

In the above image (a) is given graph and (b),(c) are two different spanning trees. Image (c) is the minimum spanning tree as it have less cost compare to (b).

i. Prim's algorithm:

- Start with tree T_1 consisting of one (any) vertex and "grow" tree one vertex at a time to produce MST through a series of expanding subtrees T_1, T_2, …, T_n
- On each iteration, construct T_{i+1} from T_i by adding vertex not in T_i that is closest to those already in T_i (this is a "greedy" step!)
- Stop when all vertices are included.

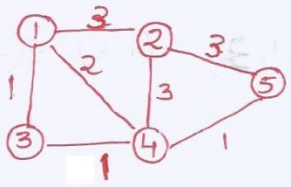
ALGORITHM *Prim(G)*

//Prim's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph $G = \langle V, E \rangle$
//Output: E_T, the set of edges composing a minimum spanning tree of G
$V_T \leftarrow \{v_0\}$ //the set of tree vertices can be initialized with any vertex
$E_T \leftarrow \varnothing$
for $i \leftarrow 1$ **to** $|V| - 1$ **do**
    find a minimum-weight edge $e^* = (v^*, u^*)$ among all the edges (v, u) such that v is in V_T and u is in $V - V_T$
    $V_T \leftarrow V_T \cup \{u^*\}$
    $E_T \leftarrow E_T \cup \{e^*\}$
return E_T

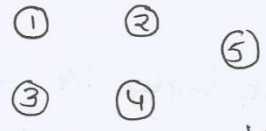
- Needs priority queue for locating closest fringe(not visited) vertex.
- Efficiency:
    i. $O(n^2)$ for weight matrix representation of graph and array implementation of priority queue
    ii. O(m log n) for adjacency lists representation of graph with n vertices and m edges and min-heap implementation of the priority queue

Eg.

Eg 1: Find Minimum spanning tree for the below graph using Prim's alg with starting vertex = {1}



(i) $V_T = \{1\}$, $E_T = \{\phi\}$ & MST at this instance is as follows



Remaining vertices are: 2(1,3), 3(1,1), 4(1,2), 5(-,∞)

(vertex, Parent, cost from Parent to vertex)

(ii) From remaining vertices min. cost vertex is 3(1,1).

∴ add vertex 3 to V_T ⇒ $V_T = \{1, 3\}$ and

$E_T = \{(1,3)\}$

Now, considering $V_T = \{1, 3\}$ the remaining vertices are: 2(1,3), 4(1,2), 5(-,∞), 4(3,1)

↓ is add as 4 is not in V_T

(iii) Now at this instance, MST is

(iii) From remaining vertices min. cost vertex is 4(3,1).

∴ add vertex 4 to V_T ⇒ $V_T = \{1, 3, 4\}$ &

$E_T = \{(1,3), (3,4)\}$

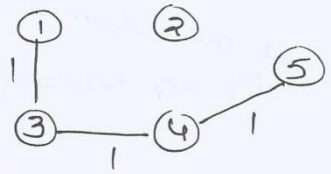
Now, considering $V_T = \{1, 3, 4\}$, the remaining vertices are: 2(1,3), 4(1,2), 5(-,∞), 2(4,3), 5(4,1)

↓ added as 2, 5 not in V_T

(iv) From remaining vertices min. cost vertex is 5(4,1)

$\therefore V_T = \{1, 3, 4, 5\}$ & $E_T = \{(1,3), (3,4), (4,5)\}$

Now, remaining vertices are: 2(1,5), 4(1,2), 5(-,∞) 2(4,3), 2(5,3).



(v) From remaining vertices min. cost vertex is 4(1,2).

But vertex 4 is already in V_T. So reject it.

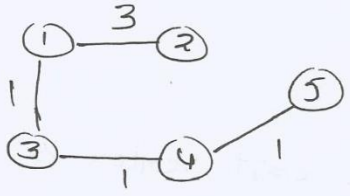
Now, remaining vertices : 2(1,3), 5(-,∞), 2(4,3) & 2(5,3)

(vi) From remaining vertices min. cost vertex is 2(1,3)

$\therefore V_T = \{1, 3, 4, 5, 2\}$ & $E_T = \{(1,3), (3,4), (4,5), (1,2)\}$

as we reach (n-1) edges we stop iterating

Now, min. spaning tree is

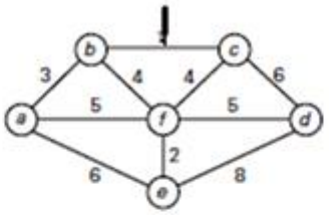


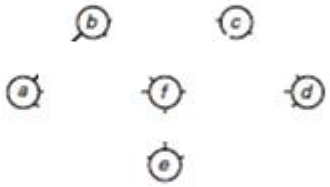
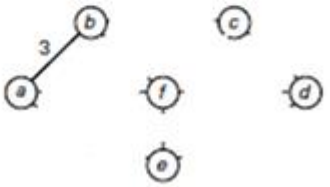
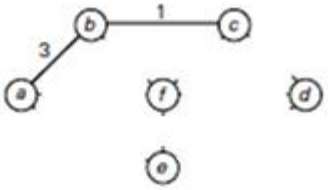
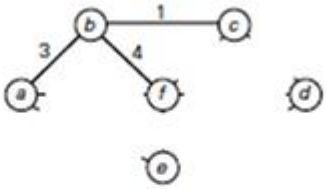
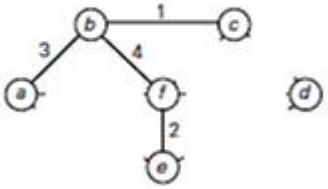
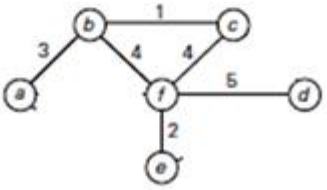
Min. spanning tree cost is

$= 1 + 1 + 1 + 3$

$= 6.$

Eg. 2:



Tree vertices	Remaining vertices	Illustration
a(−, −)	**b(a, 3)** c(−, ∞) d(−, ∞) e(a, 6) f(a, 5)	
b(a, 3)	**c(b, 1)** d(−, ∞) e(a, 6) **f(b, 4)**	
c(b, 1)	d(c, 6) e(a, 6) **f(b, 4)**	
f(b, 4)	d(f, 5) **e(f, 2)**	
e(f, 2)	**d(f, 5)**	
d(f, 5)		

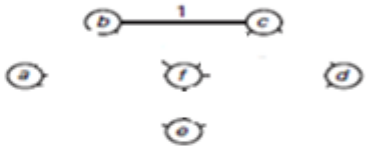
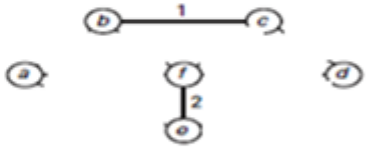
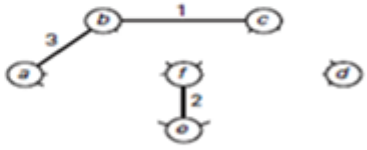
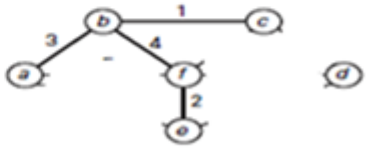
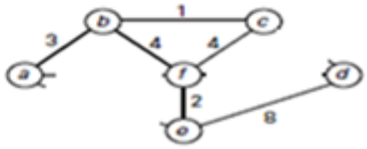
ii. Kruskal's algorithm:

- Sort the edges in nondecreasing order of lengths
- "Grow" tree one edge at a time to produce MST through a series of expanding forests $F_1, F_2, \ldots, F_{n-1}$
- On each iteration, add the next edge on the sorted list unless this would create a cycle. (If it would, skip the edge.)

ALGORITHM *Kruskal(G)*

//Kruskal's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph $G = \langle V, E \rangle$
//Output: E_T, the set of edges composing a minimum spanning tree of G
sort E in nondecreasing order of the edge weights $w(e_{i_1}) \le \cdots \le w(e_{i_{|E|}})$
$E_T \leftarrow \varnothing$; $ecounter \leftarrow 0$ //initialize the set of tree edges and its size
$k \leftarrow 0$ //initialize the number of processed edges
while $ecounter < |V| - 1$ **do**
  $k \leftarrow k + 1$
  if $E_T \cup \{e_{i_k}\}$ is acyclic
    $E_T \leftarrow E_T \cup \{e_{i_k}\}$; $ecounter \leftarrow ecounter + 1$
return E_T

- Algorithm looks easier than Prim's but is harder to implement (checking for cycles!)
- Cycle checking: a cycle is created iff added edge connects vertices in the same connected component
- Runs in *O(m log m)* time, with $m = |E|$. The time is mostly spent on sorting.

Tree edges	Sorted list of edges	Illustration
—	bc cf ab bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
bc 1	bc cf ab bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
cf 2	bc cf ab bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
ab 3	bc cf ab bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8 cf & af are rejected	
bf 4	bc cf ab bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
df 5		

Q) Explain indetail about single source shortest path problem.

Single Source Shortest Paths Problem: Given a weighted connected (directed) graph G, find shortest paths from source vertex s to each of the other vertices.

Dijkstra's algorithm: Similar to Prim's MST algorithm, with a different way of computing numerical labels: Among vertices not already in the tree, it finds vertex u with the smallest sum

$$d_v + w(v,u)$$

where

v is a vertex for which shortest path has been already found on preceding iterations (such vertices form a tree rooted at s)

d_v is the length of the shortest path from source s to v

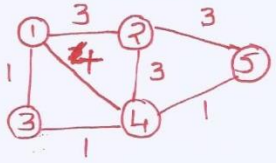
w(v,u) is the length (weight) of edge from v to u.

ALGORITHM *Dijkstra(G, s)*

```
//Dijkstra's algorithm for single-source shortest paths
//Input: A weighted connected graph G = ⟨V, E⟩ with nonnegative weights
//        and its vertex s
//Output: The length d_v of a shortest path from s to v
//          and its penultimate vertex p_v for every vertex v in V
Initialize(Q)  //initialize priority queue to empty
for every vertex v in V
    d_v ← ∞;   p_v ← null
    Insert(Q, v, d_v)  //initialize vertex priority in the priority queue
d_s ← 0;  Decrease(Q, s, d_s)  //update priority of s with d_s
V_T ← ∅
for i ← 0 to |V| − 1 do
    u* ← DeleteMin(Q)  //delete the minimum priority element
    V_T ← V_T ∪ {u*}
    for every vertex u in V − V_T that is adjacent to u* do
        if d_u* + w(u*, u) < d_u
            d_u ← d_u* + w(u*, u);   p_u ← u*
            Decrease(Q, u, d_u)
```

- Doesn't work for graphs with negative weights
- Applicable to both undirected and directed graphs
- Efficiency
  - $O(|V|^2)$ for graphs represented by weight matrix and array implementation of priority queue
  - $O(|E|\log|V|)$ for graphs represented by adj. lists and min-heap implementation of priority queue

Eg 1. Find the shortest path from vertex 1 to remaining vertices in the following graph



(i)

Vertex (V)	1	2	3	4	5
dist. (d)	∞	∞	∞	∞	∞
Parent (P)	-1	-1	-1	-1	-1

(ii) Starting vertex is 1 ⇒

Vertex V	1	2	3	4	5
dist. (d)	0	∞	∞	∞	∞
Parent (P)	0*	-1	-1	-1	-1

$V_T = \{\phi\}$

(iii) u = min. distance vertex = 1

$V_T = \{1\}$

adjacent vertices of 1 = {2, 3, 4}

(a) adjacent vertex, v = 2:
The constraint to update d_v & P_v is

$d_u + w(u,v) < d_v$

$= d_1 + w(1,2) < d_2$

$= 0 + 3 < \infty$, is True

∴ update $d[2] = 0+3 = 3$ & $P[2] = u = 1$.

(b) v = 3.

$d_1 + w(1,3) < d_3 = 0 + 1 < \infty$, is True

∴ $d[3] = 1$ & $P[3] = 1$

(c) v = 4. $d_1 + w(1,4) < d_4 = 0 + 4 < \infty$, is True

∴ $d[4] = 4$, $P[4] = 1$.

V	1	2	3	4	5
d	0	3	1	4	∞
P	0	1	1	1	-1

(iv) $u = 3$ ($\because$ Vertex 3 not in V_T after vertex 1)

$V_T = \{1, 3\}$

$V_i = \{1, 4\}$

$v = 1 \Rightarrow$ 1 is in V_T, don't consider

$v = 4 \Rightarrow d_3 + w(3,4) < d_4 = 1 + 1 < 4$ True

$\therefore$ update of $d[4] = 2$ $P[4] = 3$

V	1	2	3	4	5
d	0	3	1	2	∞
P	0	1	1	3	-1

(v) $u = 4 \Rightarrow V_T = \{1, 3, 4\}$

$V_i = \{1, 2, 3, 5\}$

$v = 1 \Rightarrow$ not considered as 1 in V_T

Similarly $v = 3$ also not considered

$v = 2 \Rightarrow d_4 + w(4,2) < d_2 = 2 + 3 < 3$, not True

$\therefore$ No update of $d[2]$ & $P[2]$.

$v = 5 \Rightarrow d_4 + w(4,5) < d_5 = 2 + 1 < \infty$, True

$\therefore$ update $d[5] = 3$, $P[5] = 4$.

V	1	2	3	4	5
d	0	3	1	2	3
P	0	1	1	3	4

(vi) $u = 2 \Rightarrow V_T = \{1, 3, 4, 2\}$

$V_i = \{1, 4, 5\}$

$v = 1$ & 4 not considered as they are in V_T

$\therefore v = 5 \Rightarrow d_2 + w(2,5) < d_5 = 3 + 3 < 3$, not True

$\therefore$ No update of $d[5] \leftarrow p[5]$.

v	1	2	3	4	5
d	0	3	1	2	3
p	0	1	1	3	4

(vii) $u = 5 \Rightarrow V_T = \{1, 3, 4, 2, 5\}$

$V_i = \{2, 4\}$

$V = 2, 4$ not considered as they are in V_T.

v	1	2	3	4	5
d	0	3	1	2	3
p	0	1	1	3	4

$\therefore$ shortest paths are:

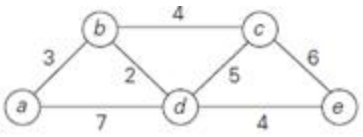
1-2 : 1 — 2, length = 3

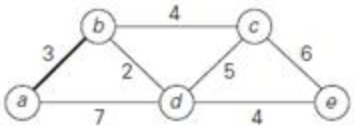
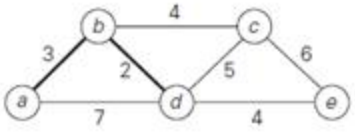
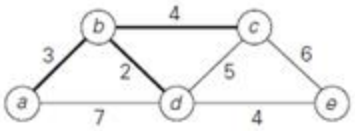
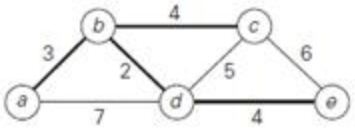
1-3 : 1 — 3, length = 1

1-4 : 1 — 3 — 4, cost = 2

1-5 : 1 — 3 — 4 — 5, cost = 3.

Eg 2.



Tree vertices	Remaining vertices	Illustration
a(−, 0)	**b(a, 3)** c(−, ∞) d(a, 7) e(−, ∞)	
b(a, 3)	c(b, 3 + 4) **d(b, 3 + 2)** e(−, ∞)	
d(b, 5)	**c(b, 7)** e(d, 5 + 4)	
c(b, 7)	**e(d, 9)**	
e(d, 9)		

The shortest paths and their lengths are:
From a to b: a – b of length 3
From a to d: a – b – d of length 5
From a to c: a – b – c of length 7
From a to e: a – b – d – e of length 9