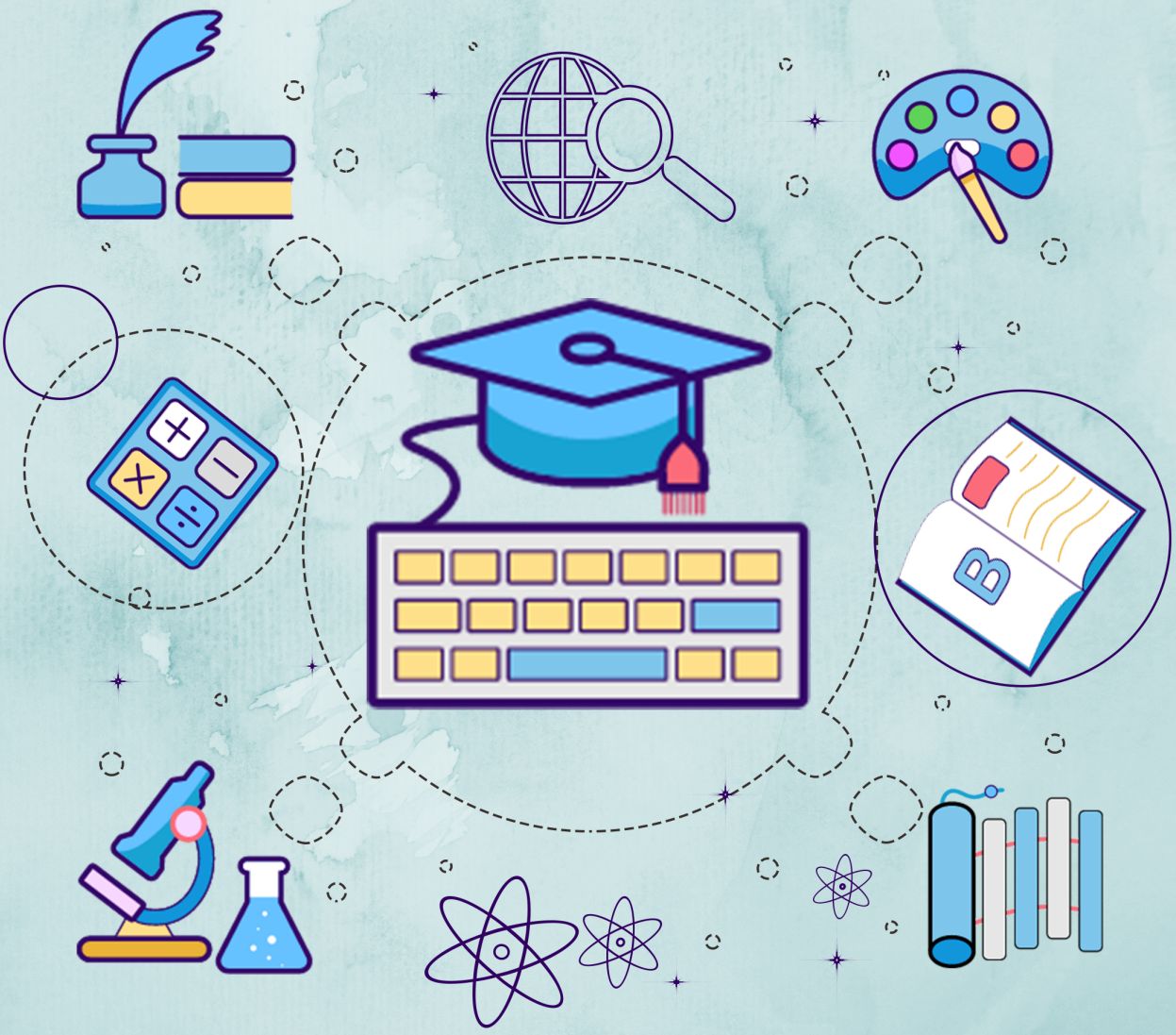
APJ ABDUL KALAM TECHNOLOGICAL UNIVERSITY

KeralaNotes



SYLLABUS | STUDY MATERIALS | TEXTBOOK

PDF | SOLVED QUESTION PAPERS

www.keralanotes.com

KeralaNotes

KTU STUDY MATERIALS

ALGORITHM ANALYSIS AND DESIGN

CST 306

Module 3

Related Link :

- KTU S6 CSE NOTES 2019 SCHEME
- KTU S6 SYLLABUS CSE COMPUTER SCIENCE
- KTU PREVIOUS QUESTION BANK S6 CSE SOLVED
- KTU CSE TEXTBOOKS S6 B.TECH PDF DOWNLOAD
- KTU S6 CSE NOTES | SYLLABUS | QBANK | TEXTBOOKS DOWNLOAD

AAD

ALGORITHM ANALYSIS AND DESIGN - CST306

Module 3

Neethu Mathew , CSE Dept. EKCTC

Module–3 (Divide & Conquer and Greedy Strategy)

The Control Abstraction of Divide and Conquer- 2-way Merge sort, Strassen's Algorithm for Matrix Multiplication-Analysis. The Control Abstraction of Greedy Strategy- Fractional Knapsack Problem, Minimum Cost Spanning Tree Computation- Kruskal's Algorithms - Analysis, Single Source Shortest Path Algorithm - Dijkstra's Algorithm-Analysis.

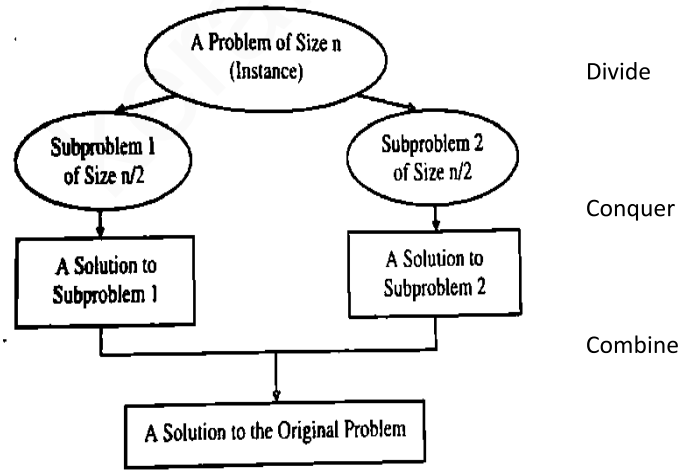
Divide and Conquer

- Divide-and-conquer is a top-down technique for designing algorithms
- A problem is divided into smaller subproblems and solutions of these subproblems are combined to get a final solution
- **3 parts** of Divide and conquer(DAndC) strategy

1. Divide : Divide the problem into a number of subproblems that are smaller instances of the same problem.

2. Conquer :Conquer means solve the subproblems recursively.

3. Combine : Combine the solutions to the subproblems into the solution for the original problem.



- Ex: binary search ,
  quick sort ,
  merge sort ,
  Strassen's matrix multiplication

Control Abstraction of Divide and Conquer

- A control abstraction is a procedure
  - ✓ whose flow of control is clear
  - ✓ whose primary operations are specified by other procedures
  - ✓ whose precise meanings are left undefined.

- The control abstraction for divide and conquer technique is DAndC(P), where P is the problem to be solved.

```
1   Algorithm DAndC(P)
2   {
3       if Small(P) then return S(P);
4       else
5       {
6           divide P into smaller instances P1, P2, ..., Pk, k ≥ 1;
7           Apply DAndC to each of these subproblems;
8           return Combine(DAndC(P1),DAndC(P2),...,DAndC(Pk));
9       }
10  }
```


- Small(P) determines whether the i/p size is small enough so that the answer can be computed without splitting. If this is so function 'S' is invoked (Solution). otherwise, the problem 'P' into smaller sub problems. These sub problems $p_1, p_2 \ldots \ldots p_k$ are solved by recursive application of DAndC
- Combine is a function that determines the solution to P using the solutions to the k subproblems

Complexity Analysis of Divide and Conquer

- If the size of problem P is n and size of k subproblems are $n_1, n_2 \ldots \ldots n_k$, then the computing time of DAndC is described by the recurrence relation

$$T(n) = \begin{cases} g(n) & n \text{ small} \\ T(n_1) + T(n_2) + \cdots + T(n_k) \quad + \quad f(n) & \text{otherwise} \end{cases}$$

T(n) --- time for DAndC on any i/p of size n

g(n) --- time to compute the answer directly for small inputs

f(n) --- time for dividing P and combining the solutions to subproblems

- **The complexity of many divide and conquer algorithms is given by recurrences of the form :**

$$T(n) = \begin{cases} T(1) & n = 1 \\ aT(n/b) + f(n) & n > 1 \end{cases}$$

Where a ,b are known constants

We assume T(1) is known and n is a power of b (ie, n=b^k)

2- WAY MERGE SORT

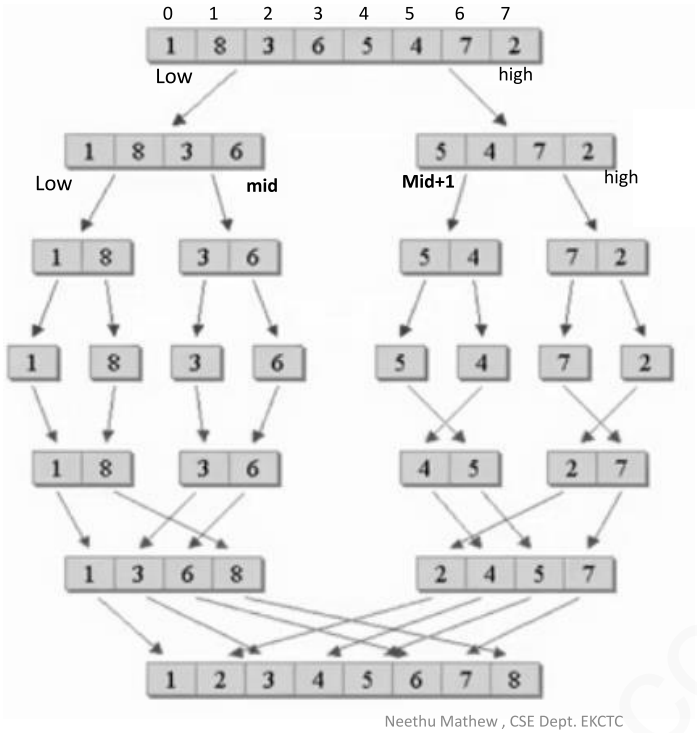
- Given a sequence of n elements a[1],……a[n]. Split this array into 2 sets : a[1]…..,a[n/2] and a[(n/2)+1],….a[n]. Each set is individually sorted and the resulting sorted sequences are merged to produce a single sorted sequence of n elements
- To sort

1. Divide : Divide n element sequence to be sorted into 2 subsequences of n/2 element each

2. Conquer : Sort the 2 subsequences recursively

3. Combine : Merge the 2 sorted subsequences to produce the sorted answer

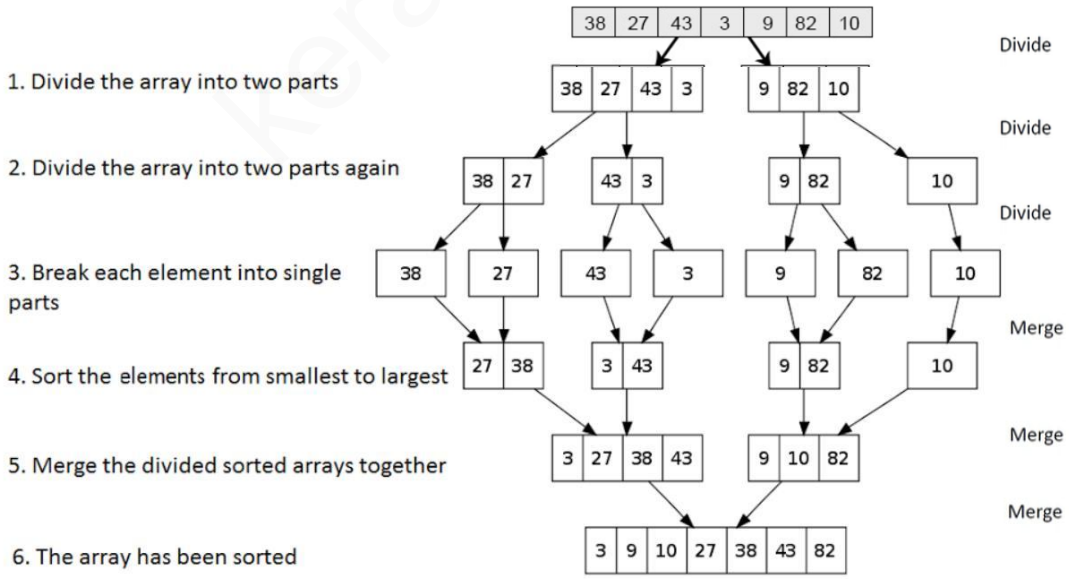
Example 1



Mid =(low + high)/2
=(0+7)/2=3

Mid+1=4

Example 2



Algorithm MergeSort

```
Algorithm MergeSort (low, high)
                                    // a (low : high) is a global array to be sorted.
  {                                 // Small(P) is true if there is only one element to sort. In this case the list is already sorted

    if (low < high)   then          // if there are more than 1 element
     {                              // divide P into subproblems
      mid = (low + high)/2 ;        //finds where to split the set
      MergeSort(low, mid) ;         //sort one subset
      MergeSort(mid+1, high);       //sort the other subset
      Merge(low, mid, high);        // combine the solutions
     }
  }
```



```
Algorithm Merge(low, mid, high)
{     i= low; x= low;  y= mid + 1;
      while((x ≤ mid) and (y ≤ high)) do
      {  if ( a[x] ≤ a[y] ) then
         {      b[i] = a[x];
                x = x+1;
         }
         else
         {      b[i] = a[y];
                y = y+1;
         }
         i=i+1;
      }
      if( x ≤ mid) then
      {  for k=x to mid do
         {      b[i] = a[k];
                i =i+1;
         }
      }
      else
      {  for k=y to high do
         {      b[i] = a[k];
                i =i+1;
         }
      }
      for k= low to high do
         a[k] = b[k];
}
```

Time complexity - 2 way merge sort

$$T(n) = \begin{cases} a & \text{if } n=1 \\ 2\,T(n/2) + cn & \text{Otherwise} \end{cases}$$

a is the time to sort an array of size 1

cn is the time to merge two sub-arrays

2 T(n/2) is the complexity of two recursion calls

Above recurrence can be Solved by iteration method

$$\begin{aligned} T(n) &= 2\,T(n/2) + c\,n \\ &= 2(2\,T(n/4)+c(n/2)) + c\,n \\ &= 2^2T(n/2^2) + 2\,c\,n \\ &= 2^3T(n/2^3) + 3\,c\,n \\ &\ldots\ldots\ldots\ldots \\ &= 2^kT(n/2^k) + k\,c\,n \qquad [\text{Assume that } n/2^k = 1,\ k=\log n] \\ &= n\,T(1) + c\,n\log n \\ &= a\,n + c\,n\log n \\ &= \mathbf{O(n \log n)} \end{aligned}$$

Best Case, Average Case and Worst Case Complexity of Merge Sort
= O(n log n)

Strassen's Matrix Multiplication

Basic Matrix Multiplication

- We can multiply 2 matrices A and B iff no. of columns of A = no. of rows of B
- Consider a square matrix of size n (n is power of 2) . If n is not power of 2 , we can make it so by adding zeros

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}$$

A 2x2 B 2x2 C 2x2

Where C = A*B, The Matrix C can be calculated as,

$$c_{11} = a_{11}b_{11} + a_{12}b_{21}$$
$$c_{12} = a_{11}b_{12} + a_{12}b_{22}$$
$$c_{21} = a_{21}b_{11} + a_{22}b_{21}$$
$$c_{22} = a_{21}b_{12} + a_{22}b_{22}$$

Matrix Multiplication - Divide and Conquer

Example Using 4×4

Since it is a larger problem therefore we must divide it into smaller problems and then solve those sub-problem and combine the solutions.

Following is the simple divide and conquer method to multiply two 4×4 square matrices,

a) Divide both matrices in 4 sub-matrices of size 2×2 (i.e. n/2 x n/2 where n=4)

b) Then calculate the values recursively.

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

A_{11} A_{12} A_{21} A_{22} — 4/2 x 4/2

B_{11} B_{12} B_{21} B_{22} — 4/2 x 4/2

$$C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}$$
$$C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22}$$
$$C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21}$$
$$C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22}$$

Time Complexity

- In the above method, we do **8 multiplications** for matrices of size n/2 x n/2 and **4 additions**. Addition of two matrices takes **O(n^2)** time.
- So the time complexity can be written as

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

From Master's Theorem, **time complexity of above method is O(n^3)** which is unfortunately same as the above naive method.

Strassen's Matrix Multiplication

- Strassen algorithm is a recursive method for matrix multiplication where we divide the matrix into 4 sub-matrices of dimensions n/2 x n/2 in each recursive step.
- In **divide and conquer method**, the main component for high time complexity is **8 recursive calls**.
- The idea of **Strassen's method** is to reduce the number of **recursive calls to 7**.

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

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

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

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

$T = (A_{11} + A_{12}) B_{22}$

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

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

Using these 7 formula ,we get

$C_{11} = P + S - T + V$

$C_{12} = R + T$

$C_{21} = Q + S$

$C_{22} = P + R - Q + U.$

- Strassen's algorithm uses **7 multiplications** which reduces the time complexity of the matrix multiplication algorithm a little bit.
- Addition and Subtraction operation takes less time compared to multiplication process. In Strassen's matrix multiplication algorithm the number of multiplication reduced but the number of addition and subtraction increased.
- Strassen's method is similar to divide and conquer method in the sense that this method also divide matrices to sub-matrices of size n/2 x n/2 , but in Strassen's method, the four sub-matrices of result are calculated using above formula:

Time Complexity of Strassen's Method

- Addition and Subtraction of two matrices takes **O(n^2)** time.

  So time complexity can be written as

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

From Master's Theorem, time complexity of above method is **O(n^{log7})** which is approximately **O($n^{2.8074}$)**

The O($n^{2.8074}$) is slightly lesser than **O(n^3)**

Ques 8) Let consider $A=\begin{bmatrix}1 & 2\\ 0 & 2\end{bmatrix}$ **and** $B=\begin{bmatrix}0 & 1\\ 3 & 2\end{bmatrix}$ **Then. Find AB using Strassen's Algorithm for Matrix Multiplication.**

Ans:

$A = \begin{bmatrix}1 & 2\\ 0 & 2\end{bmatrix}$ $B = \begin{bmatrix}0 & 1\\ 3 & 2\end{bmatrix}$

$A_{11} = 1$ $B_{11} = 0$

$A_{12} = 2$ $B_{12} = 1$

$A_{21} = 0$ $B_{21} = 3$

$A_{22} = 2$ $B_{22} = 2$.

$P = (A_{11} + A_{22})(B_{11} + B_{22}) = (1+2)(0+2) = 6$

$Q = (A_{21} + A_{22})B_{11} = (0+2)0 = 0$

$R = A_{11}(B_{12} - B_{22}) = 1(1-2) = -1$

$S = A_{22}(B_{21} - B_{11}) = 2(3-0) = 6$

$T = (A_{11} + A_{12})B_{22} = (1+2)2 = 6$

$U = (A_{21} - A_{11})(B_{11} + B_{12}) = (0-1)(0+1) = -1$

$V = (A_{12} - A_{22})(B_{21} + B_{22}) = (2-2)(3+2) = 0$

$C_{11} \rightarrow P + S - T + V = 6+6-6+0 = 6$

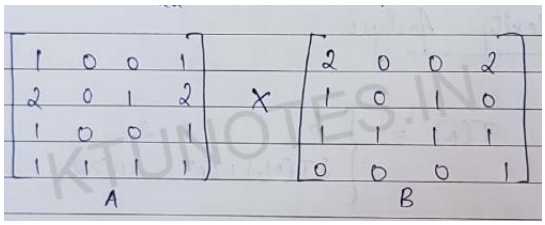
$C_{12} \rightarrow R + T = (-1)+6 = 5$

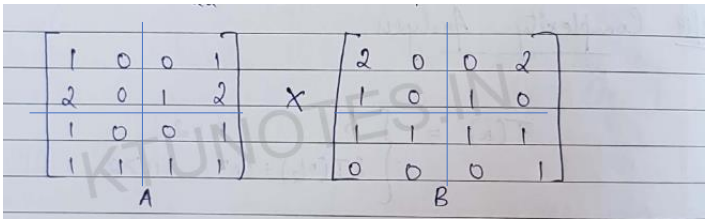
$C_{21} \rightarrow Q + S = 0+6 = 6$

$C_{22} \rightarrow P + R - Q + U = 6+(-1)-0+(-1) = 4$

$$AB = \begin{bmatrix}C_{11} & C_{12}\\ C_{21} & C_{22}\end{bmatrix} = \begin{bmatrix}6 & 5\\ 6 & 4\end{bmatrix}$$

Example:





Ans:

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

$Q = (A_{21} + A_{22})\, B_{11}$

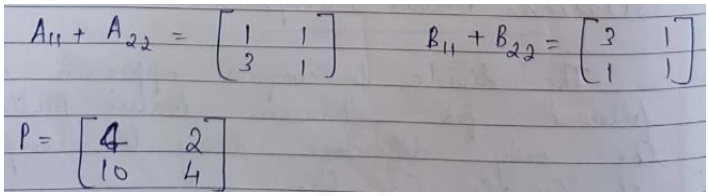
$R = A_{11}\, (B_{12} - B_{22})$

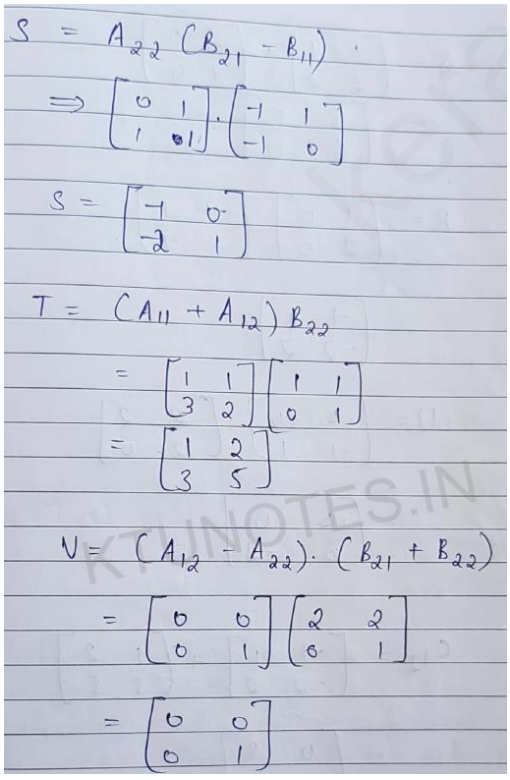
$S = A_{22}\, (B_{21} - B_{11})$

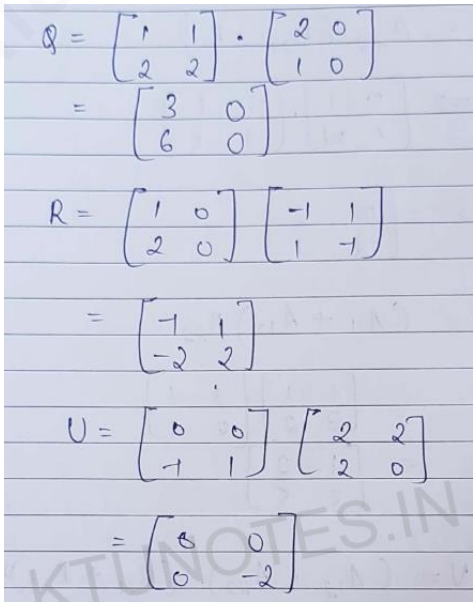
$T = (A_{11} + A_{12})\, B_{22}$

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

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





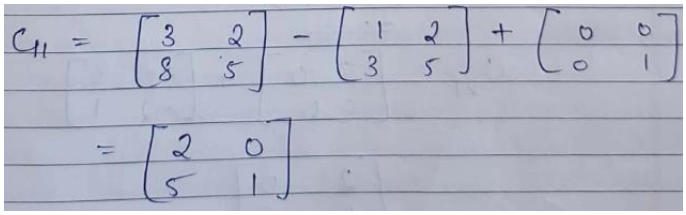


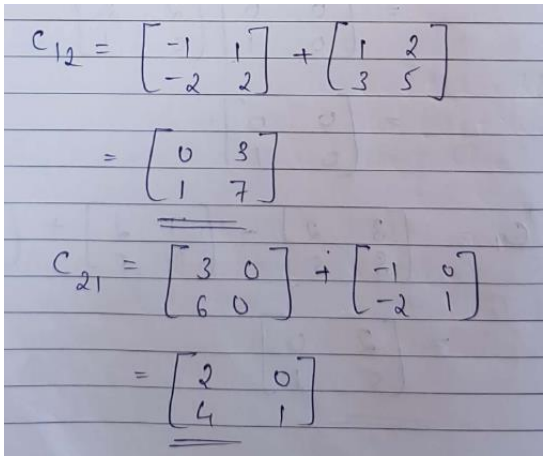
$C_{11} = P + S - T + V$

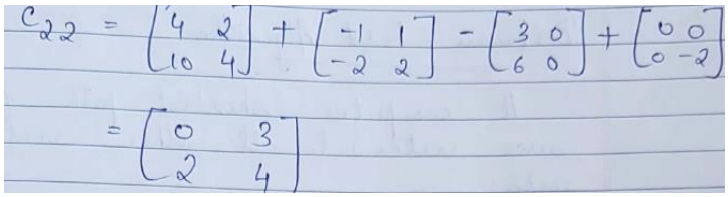
$C_{12} = R + T$

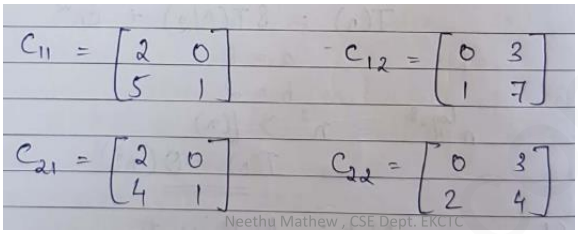
$C_{21} = Q + S$

$C_{22} = P + R - Q + U.$









Greedy Strategy

- **Used to solve optimization problems**
- **Greedy Algorithm works by taking a sequence of choices or decisions to solve a given problem**
- **It takes the choice that seems best at the particular moment**
- **The decisions are taken by some selection criteria , which will hopefully given an optimal solution**
- **This approach does not always leads to an optimal solution**

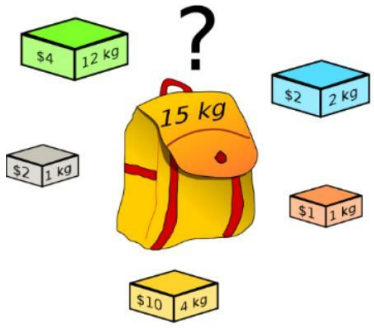
✓ Control Abstraction for Greedy algorithm

```
Algorithm Greedy ( a , n )
{
      // a[1:n] contains n inputs
      solution= ϕ          //initialize the solution to empty
      for i = 1 to n   do
      {
          x = Select(a);
          if Feasible(solution, x) then
              solution = Union(solution, {x})
      }
return solution
}
```


- The array a[1…n] stores the n inputs
- The solution set is initialized as null
- With the help of some selection criteria ,an input x is chosen from the array.
- The function **Select** selects an input from 'a' whose value is assign to x.
- **Feasible** is a boolean valued function that checks whether adding x to the solution set will still keep the solution feasible. If yes, the input x will be added to the solution set
- The function **Union** combines x with the solution, and update the objective function.
- The function **Greedy** describes the essential way that a greedy algorithm will look, once a particular problem is chosen and the functions Select, Feasible and Union are properly implemented.

Problem Scenario (knapsack problem)

- A thief is robbing a store and can carry a maximal weight into his knapsack. There are n items available in the store and each with a weight and a profit. What items should the thief take?



- We need to put these items in a knapsack according to its capacity to get maximum profit.
- In this context, the items should be selected in such a way that the thief will carry those items for which he will gain maximum profit. Hence, the objective of the thief is to maximize the profit.

- Knapsack problems are categorized as
  - ✓ Fractional Knapsack problem
    : items are divisible (fraction of items can be taken)
    (We can even put the fraction of any item into the knapsack if taking the complete item is not possible.)
    : it can be solved by greedy method
  - ✓ 0-1 Knapsack problem
    : items are indivisible (either take an item or not)
    : can be solved by dynamic programming

Fractional Knapsack Problem / Greedy Knapsack Problem

- Given a knapsack of **capacity m** and **n items** to be considered to fill the knapsack with.
- The item i is associated with a **weight** w_i and a **profit** p_i
- If a **fraction** x_i (where $0 \leq x_i \leq 1$) of item is placed into the knapsack , then a profit of $p_i x_i$ is earned.
- Since the total capacity is m ,the total weight of chosen items should be ≤ m
- The **objective** of the knapsack problem is to **maximize the profit** within the capacity limit, subject to the following conditions…..

Maximize $$\sum_{1 \leq i \leq n} p_i x_i$$

subject to the constraint,

$$\sum_{1 \leq i \leq n} w_i x_i \leq m \quad , \quad 0 \leq x_i \leq 1 \quad \text{and} \quad 1 \leq i \leq n$$

Greedy algorithm gives the optimal solution for the Fractional Knapsack problem

Steps

Step 1: Compute the profit / weight ratio for each item

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

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

Algorithm

```
Algorithm GreedyKnapsack(m,n)
    // p[i:n] andw[1:n] contain the profits and weights respectively
    // if the n-objects ordered such that p[i]/w[i]>=p[i+1]/w[i+1], m is the size of knapsack 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]:=1.0;
        U:=U-w[i];
      }
     If(i<=n) then x[i]:=U/w[i];
    }
```

Time Complexity is O(n logn)

Step 1:- Find profit/weight for each object.

Step 2:- Since we have to select whether we have to completely select the object or partially select it.

So we sort the profit/weight in descending order.

Step 3:-The object with the highest profit/weight is selected first.

Step 4:-Mark the object with 1 if it's completely selected or the fraction part if it is not selected completely.

Step 5:-While we select a particular object, Deduct the knapsack size by its particular object size.

Step 6:-Repeat steps 4 & 5.

Step 7:-Note the final fraction part and count that object in the Knapsack(Remaining weight/Total weight).

Step 8:-Find the total weight (Summation of weights*(Selected objects weight)).

Find total profit earned

Neethu Mathew , CSE Dept. EKCTC

Example

Q)

For the given set of items and knapsack capacity = 60 kg, find the optimal solution for the fractional knapsack problem making use of greedy approach. Consider-

n = 5

w = 60 kg

(w1, w2, w3, w4, w5) = (5, 10, 15, 22, 25)

(p1, p2, p3, p4, p5) = (30, 40, 45, 77, 90)

solution

item	Profit p_i	Weight w_i
i -1	30	5
i -2	40	10
i -3	45	15
i -4	77	22
i -5	90	25

Step 1: Compute the profit / weight ratio for each item

item	Profit p_i	Weight w_i	Profit/weight p_i/w_i
i -1	30	5	6
i -2	40	10	4
i -3	45	15	3
i -4	77	22	3.5
i -5	90	25	3.6

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

item	Profit p_i	Weight w_i	Profit/weight p_i/w_i
i -1	30	5	6
i -2	40	10	4
i -5	90	25	3.6
i -4	77	22	3.5
i -3	45	15	3

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

item	Profit p_i	Weight w_i	Remaining weight (knapsack capacity = 60 kg)	x	$p_i x_i$
i -1	30	5	60–5=55	1	1 x 30=30
i -2	40	10	55-10=45	1	1 x 40=40
i -5	90	25	45-25=20	1	1 x 90=90
i -4	77	22	Weight of item 4 is 22 ,But remaining knapsack capacity is 20. So fraction of it added knapsack	20/22	(20/22)77 =70
i -3	45	15	Capacity =0 So item 3 is not included	0	0

w = 60 kg ie, m=60

x: Full item included-1 , Item not included -0, Or include fraction

$\sum p_i x_i$ = Total profit earned = 30+40+90+70=230

Solution vector x[ ] based on item(i-1 to i-5)

$(x_1, x_2, x_3, x_4, x_5)$= (1 , 1 , 0 , 20/22 , 1)

(w1, w2, w3, w4, w5) = (5, 10, 15, 22, 25)

$(x_1, x_2, x_3, x_4, x_5)$= (1 , 1 , 0 , 20/22 , 1)

$\sum w_i x_i$ =1x5+1x10+0x15+(20/22)x22+1x25=60

$\sum p_i x_i$ = 230

m=60

$constraint : \sum w_i x_i \leq m \qquad 60 \leq 60$

Objective : Maximize $\sum p_i x_i$

True !

Q)

State fractional knapsack problem. Give an algorithm for fractional knapsack problem using greedy strategy.

Q)

Find an optimal solution to the fractional knapsack problem for an instance with number of items 7, Capacity of the sack W=15, profit associated with the items (p1,p2,…,p7)= (10,5,15,7,6,18,3) and weight associated with each item (w1,w2,…,w7)= (2,3,5,7,1,4,1).

Ans) Step 1: Compute the profit / weight ratio for each item

item	Profit p_i	Weight w_i	Profit/weight p_i/w_i
i -1	10	2	5
i -2	5	3	1.67
i -3	15	5	3
i -4	7	7	1
i -5	6	1	6
i -6	18	4	4.5
i -7	3	1	3

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

item	Profit p_i	Weight w_i	Profit/weight p_i/w_i
i -5	6	1	6
i -1	10	2	5
i -6	18	4	4.5
i -3	15	5	3
i -7	3	1	3
i -2	5	3	1.67
i -4	7	7	1

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

item	Profit p_i	Weight w_i	Remaining weight	x	$p_i x_i$
i -5	6	1	15 -1 =14	1	1x6
i -1	10	2	14-2=12	1	1x10
i -6	18	4	12-4=8	1	1x18
i -3	15	5	8-5=3	1	1x15
i -7	3	1	3-1=2	1	1x3
i -2	5	3	Weight of item 2 is 3 ,but remaining knapsack capacity is 2. So fraction of it added knapsack	2/3	5(2/3)
i -4	7	7	Capacity =0, So item 4 is not included	0	0x7

Capacity of the sack W=15
ie, m=15

x: Full item included-1 ,
Item not included -0,
Or include fraction

(w1, w2, w3, w4, w5) = (5, 10, 15, 22, 25) , (p1,p2,.....p7)=(10,5,15,7,6,18,3) , **$(x_1, x_2, x_3, x_4, x_5, x_6, x_7)$= (1 , 2/3 ,1, 0 , 1, 1,1)** and m=15

$\sum w_i x_i$ =1x2+(2/3)x3+1x5+0x7+1x1+1x4+1x1=15

$\sum p_i x_i$ = 1x10+(2/3)5+1x15+0+1x6+1x18+1x3=10+2x1.3+15+6+18+3=55.33

So Total profit = 55.33

constraint : $\sum w_i x_i \leq m$ $15 \leq 15$

Objective : *Max* $\sum p_i x_i$

Solution vector

$(x_1, x_2, x_3, x_4, x_5, x_6, x_7)$= (1 , 2/3 ,1, 0 , 1, 1,1)

Questions

Q) Explain Greedy Approach. Write the general greedy algorithm.

Q) Formulate Fractional Knapsack Problem.

Q) Write Greedy Algorithm for fractional Knapsack Problem.

Q) Find the optimal solution for the following fractional Knapsack problem.

n=4, m = 60, W={40, 10, 20, 24} and P={280, 100, 120, 120}

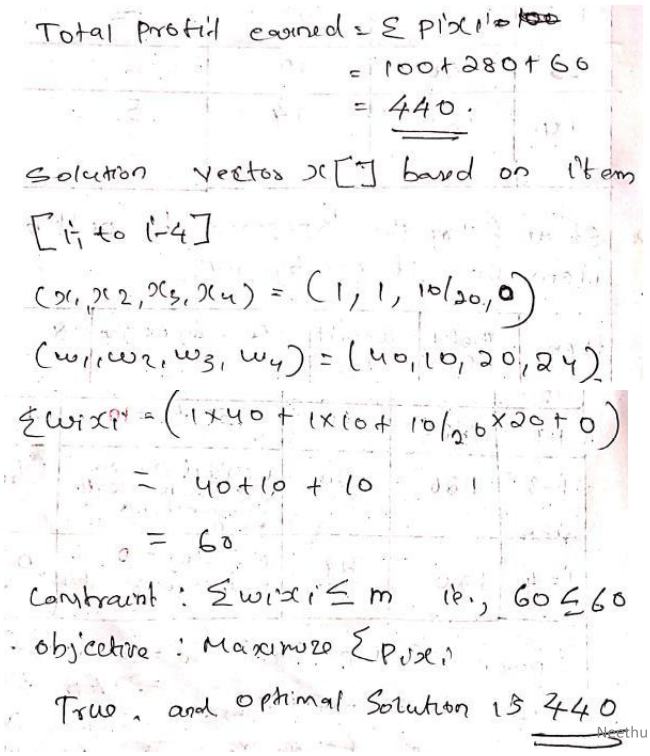
Find the optimal solution for the following fractional Knapsack problem.

n=4, m = 60, W={40, 10, 20, 24} and P={280, 100, 120, 120}

item	Profit	weight	Profit/weight Pi/wi
i-1	280	40	7
i-2	100	10	10
i-3	120	20	6
i-4	120	24	5

item	Pi	wi	Pi/wi
i-2	100	10	10
i-1	280	40	7
i-3	120	20	6
i-4	120	24	5

Item	Profit Pi	weight wi	Remaining weight	x	Pixi
i-2	100	10	60-10 = 50	1	1x100
i-1	280	40	50-40 = 10	1	1x280
i-3	120	20	weight of i3 is 20 but remaining KC = 10 so fraction of i3 is added to it	10/20	10/20 x 120
i=4	120	24	capacity = 0 so i4 is not included	0	0


Total Profit earned = $\sum p_i x_i$
= 100 + 280 + 60
= 440.

Solution vector x[] based on item [1 to 4]

$(x_1, x_2, x_3, x_4) = (1, 1, 10/20, 0)$

$(w_1, w_2, w_3, w_4) = (40, 10, 20, 24)$

$\sum w_i x_i = (1 \times 40 + 1 \times 10 + 10/20 \times 20 + 0)$

= 40 + 10 + 10

= 60

Constraint : $\sum w_i x_i \leq m$ i.e., $60 \leq 60$

objective : Maximize $\sum p_i x_i$

True, and optimal Solution is 440

Applications of greedy method:

- Knapsack Problem
- Minimum Cost Spanning Tree
- Activity Selection Problem
- Huffman Code
- Job Sequencing
- Topological sorting
- Single source shortest paths
- Graph coloring
- TSP

Spanning Tree

- The spanning tree of a graph (G) is a subset of G that covers all of its vertices using the minimum number of edges.

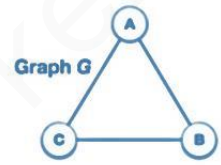
Properties of Spanning Tree

- A connected graph G can have more than one spanning tree.
- If **n** number of nodes in a graph ,spanning tree has **n-1** edges
- All possible spanning trees of graph G, have the same number of edges and vertices.
- The spanning tree does not have any cycle (loops).
- Removing one edge from the spanning tree will make the graph disconnected

  i.e. the spanning tree is **minimally connected**.
- Adding one edge to the spanning tree will create a circuit or loop, i.e. the spanning tree is **maximally acyclic**.
- A complete graph can have maximum $n^{(n-2)}$ number of spanning trees.

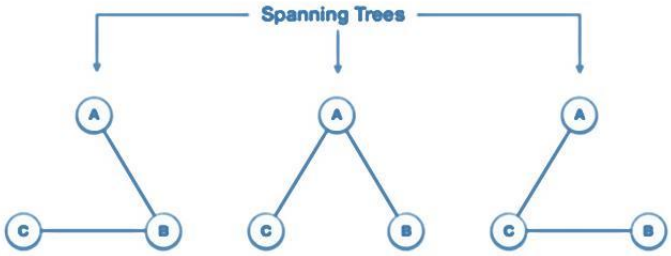
(Connected Graph : a path exists from a vertex to any other vertex
undirected graph : edges do not point to any particular direction)

Example



n=3, n-1=2 edges

n is 3, hence $3^{3-2} = 3$ spanning trees are possible



Qtn) Write the total number of spanning trees possible for a complete graph with 6 vertices

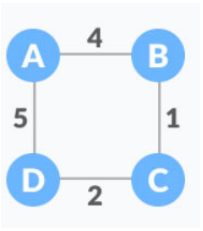
- Total number of spanning trees possible for a complete graph with n vertices = $\mathbf{n^{n-2}}$
- Total number of spanning trees possible for a complete graph with 6 vertices = $\mathbf{6^{6-2} = 6^4 = 1296}$

Minimum Spanning Tree (MST)

- A minimum spanning tree is a spanning tree with minimum edge weight
- Cost of a spanning tree= sum of cost of its edges
- A **minimum cost spanning tree** is a spanning tree that has minimum weight than all other spanning trees of the same graph.
- **Minimum Spanning-Tree Algorithms**

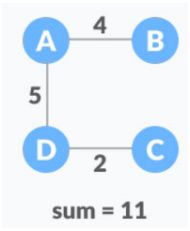
2 basic algorithms for finding minimum cost spanning tree

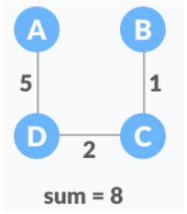
1. Kruskal's Algorithm
2. Prim's Algorithm

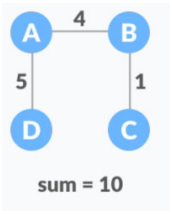


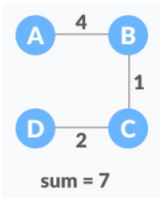
Weighted graph

The possible spanning trees from this graph are:

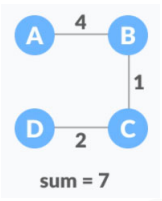


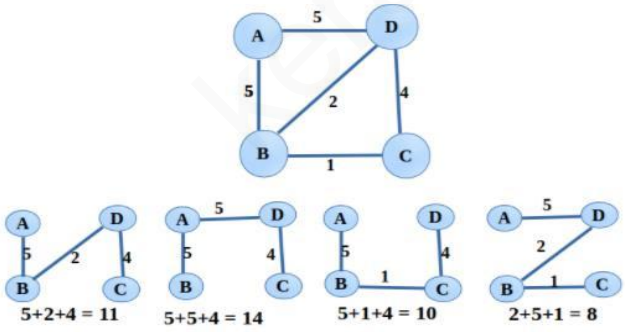




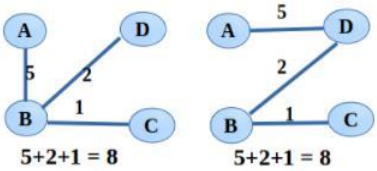


The minimum spanning tree from the above spanning trees is:





Spanning Tree



Minimum Spanning Tree

Kruskal's Algorithm

- Kruskal's algorithm finds a minimum cost spanning tree for a connected weighted graph
- **Steps :**
    1. Sort all the edges in increasing order of their weight.
    2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far.

        If cycle is not formed, include this edge.

        Else, discard it.
    3. Repeat step 2 until there are (V-1) edges in the spanning tree.

Kruskal's algorithm

```
MST-KRUSKAL(G, w)
1  A = ∅
2  for each vertex v ∈ G.V
3       MAKE-SET(v)
4  sort the edges of G.E into nondecreasing order by weight w
5  for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
6       if FIND-SET(u) ≠ FIND-SET(v)
7            A = A ∪ {(u, v)}
8            UNION(u, v)
9  return A
```

Our implementation of Kruskal's algorithm uses a disjoint-set data structure to maintain several disjoint sets of elements. Each set contains the vertices in one tree of the current forest. The operation FIND-SET(u) returns a representative element from the set that contains u. Thus, we can determine whether two vertices u and v belong to the same tree by testing whether FIND-SET(u) equals FIND-SET(v). To combine trees, Kruskal's algorithm calls the UNION procedure.

Analysis :-

Time Complexity: O(E log E) or O(E log V).

Sorting of edges takes O(E Log E) time.

After sorting, we iterate through all edges and apply find-union algorithm.

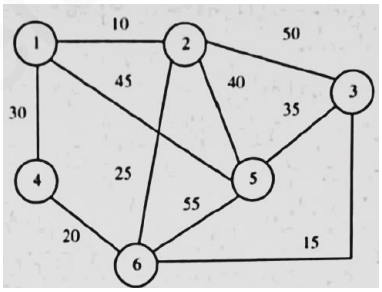
The find and union operations can take at most O(LogV) time.

So overall complexity is O(ELogE + ELogV) time.

The value of E can be at most O(V^2) and log V^2 =2 log V , so O(LogV) are O(LogE) same.

Therefore, overall time complexity is O(ElogE) or O(ElogV)

Q) Construct the minimal spanning tree for the graph shown below

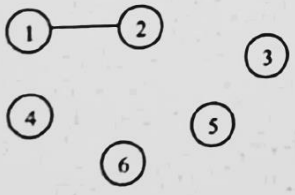
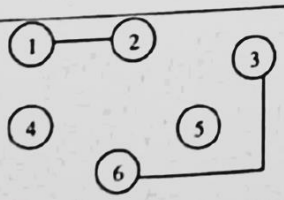
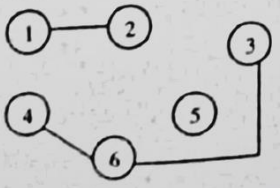
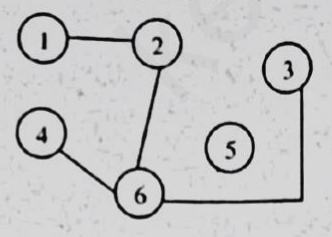
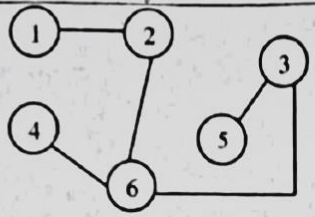


Ans: Arrange all the edges in the increasing order of their costs:

Cost	10	15	20	25	30	35	40	45	50	55
Edge	(1, 2)	(3, 6)	(4, 6)	(2, 6)	(1, 4)	(3, 5)	(2, 5)	(1, 5)	(2, 3)	(5, 6)

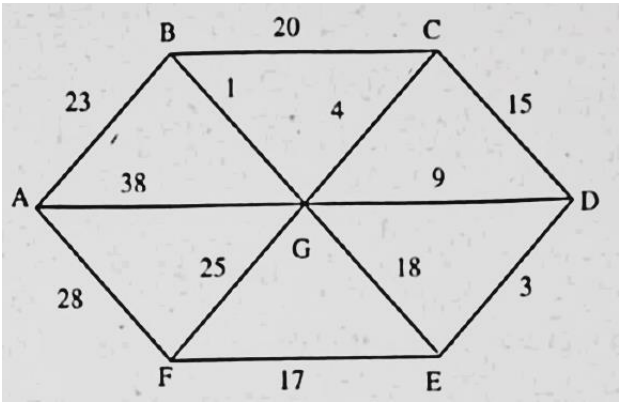
The stages in Kruskal's algorithm for minimal spanning tree are as follows:

Table 5.3

Edge	Cost	Stages in Kruskal's Algorithm	Remarks
(1, 2)	10		The edge between vertices 1 and 2 is the first edge selected. It is included in the spanning tree.
(3, 6)	15		Next, the edge between vertices 3 and 6 is selected and included in the tree.
(4, 6)	20		The edge between vertices 4 and 6 is next included in the tree.
(2, 6)	25		The edge between vertices 2 and 6 considered next and included in the tree.
(1, 4)	30	Reject	The edge between the vertices 1 and 4 is discarded as its inclusion creates a cycle.
(3, 5)	35		Edges (2,5) (1,5) (2,3) (5,6) form cycle, so discard them Finally, the edge between vertices 3 and 5 is considered and included in the tree built. This completes the tree. Weight of the MST (cost) = Sum of all edge weights Cost =10 + 15 + 20 + 25+ 35=105 The cost of the minimal spanning tree is 105.

Neethu Mathew , CSE Dept. EKCTC

Q) Find the minimum spanning tree (MST) of the following graph using Kruskal's Algorithm-

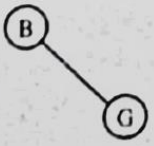


Solution-

Arrange all edges in the increasing order of their costs.

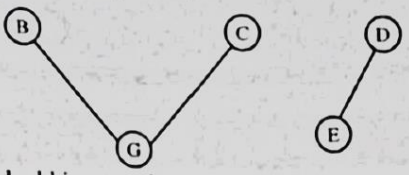
Cost	1	3	4	9	15	17	18	20	23	25	28	38
Edge	(B,G)	(D,E)	(C,G)	(D,G)	(C,D)	(E,F)	(E,G)	(B,C)	(A,B)	(F,G)	(A,F)	(A,G)

The edge between vertices B and G is first edge selected. It is included in spanning tree.

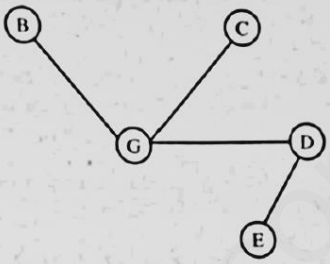


Next the edge between D, E is selected and add in spanning tree.

Next the edge between C and G is selected and added in spanning tree.

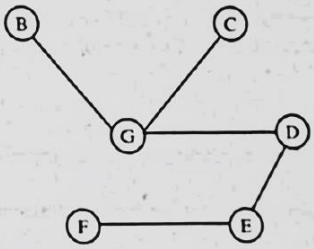


Next the edge between D and G is selected and add in spanning tree.

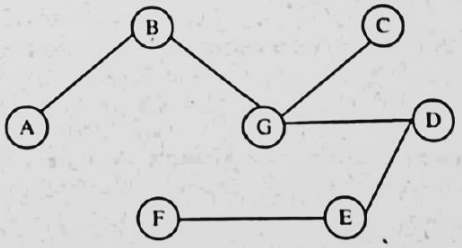


Next the edge between C and D is discarded as its inclusion creates a cycle.

Next the edge between E and F is selected and add in spanning tree.



Next the edges between E and G, and B and C are discarded. Next the edge between A and B is selected and add in spanning tree.



Next the edges between, F and G, and A and F, and A and G are discarded.

Now the tree is complete and the minimum cost is:

$1 + 3 + 4 + 9 + 17 + 23 = 57$

To apply Kruskal's algorithm,

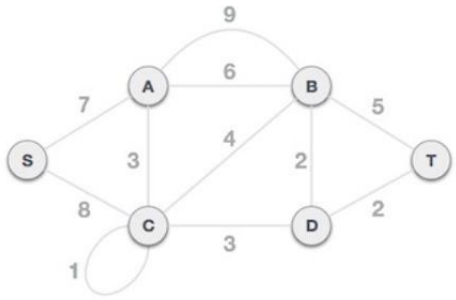
1st Remove all loops and parallel edges If any

Sort/arrange all edges in their increasing order of weight

Add the edge which has the least weightage iff it does not form cycle

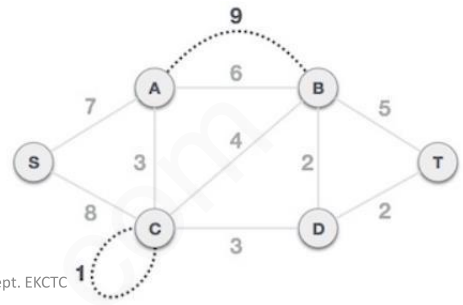
Example :

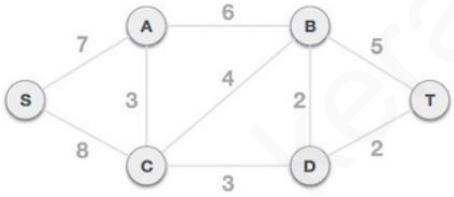
Construct the minimal spanning tree for the graph shown below



Ans) **1st Remove all loops and parallel edges if any**

(In case of parallel edges, keep the one which has the least cost)

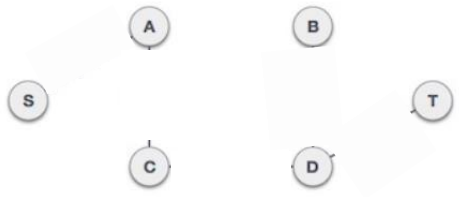


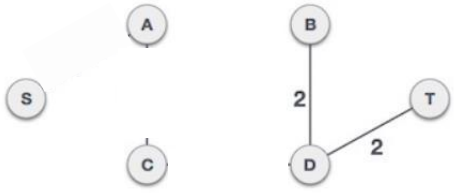


2nd - arrange all edges in their increasing order of weight

B, D	D, T	A, C	C, D	C, B	B, T	A, B	S, A	S, C
2	2	3	3	4	5	6	7	8

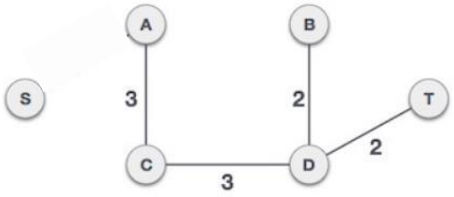
3rd - Add the edge which has the least weightage iff it does not form cycle





least cost(weight) is 2 and edges involved are B,D and D,T. So add the edges BD and then DT

Next least weight is 3, and associated edges are A,C and C,D.
We add them –



Next least weight is 4 .

Adding (CB) it will create a cycle ,so discard it

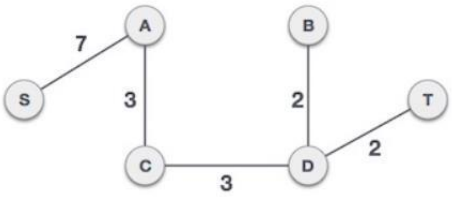
Next least weight is 5

Adding (BT) will create a cycle ,so discard it

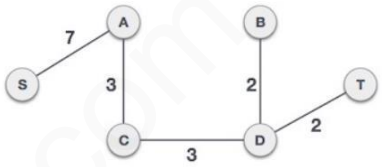
Next least weight is 6 .

Adding (AB) will create a cycle ,so discard it

Next is 7 (S,A)

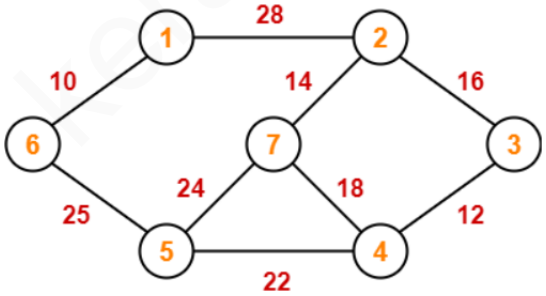


Next is 8 (SC). it forms a cycle, so discard it



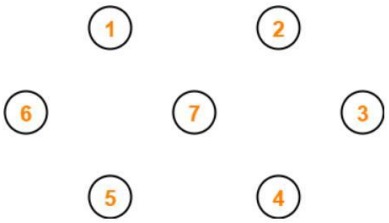
Total Cost Of MST = **Sum of all edge weights** =7 + 3 + 3 + 2+ 2=17

Q) Construct the minimum spanning tree (MST) for the given graph using Kruskal's Algorithm-

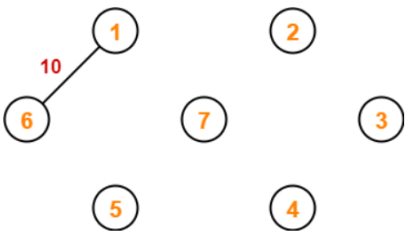


Solution-

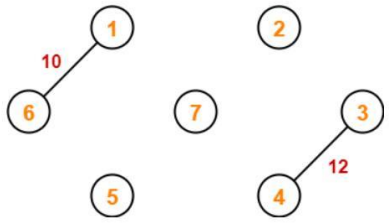
Step-1: draw all the vertices



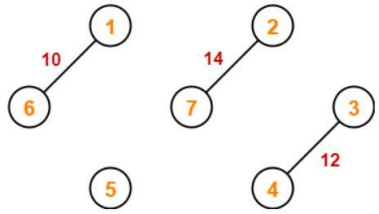
Step-2:
Connect these vertices using edges with minimum weights such that no cycle gets formed.



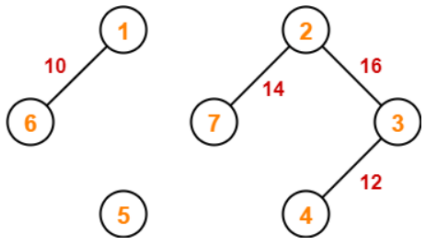
Step-3



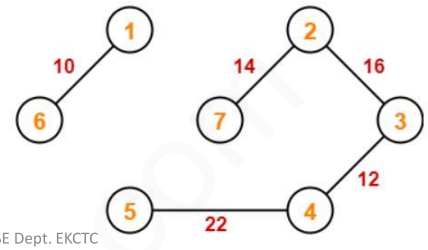
Step-4



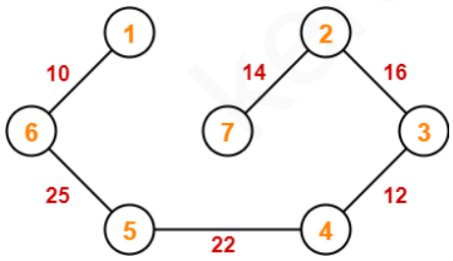
Step-5



Step-6



Step-7



Since all the vertices have been connected / included in the MST, so we stop.

Cost or Weight of the MST = Sum of all edge weights

= 10 + 25 + 22 + 12 + 16 + 14

= 99

Applications of Spanning trees

- Civil Network Planning
- Computer Network Routing Protocol
- Cluster Analysis
- Image segmentation
- Handwriting recognition

Shortest Path Problem

- Shortest path problem is a problem of finding the shortest path(s) between vertices of a given graph.
- Shortest path between two vertices is a path that has the least cost as compared to all other existing paths.

Types of Shortest Path Problem-

- **Single-pair shortest path problem** : It is a shortest path problem where the shortest path between a given pair of vertices is computed.
- **All pairs shortest path problem** :It is a shortest path problem where the shortest path between every pair of vertices is computed.
- **Single-source shortest path problem** : It is a shortest path problem where the shortest path from a given source vertex to all other remaining vertices is computed.
- **Single-destination shortest path problem** : It is a shortest path problem where the shortest path from all the vertices to a single destination vertex is computed.

Single Source Shortest Path Algorithms

- ***Single-source shortest-path problem*** **?**

Given a graph *G* = (*V*, *E*), we want to find a shortest path from a given ***source*** vertex *s Ɛ V* to every other vertex *v Ɛ V*.

: find the shortest path from a node (called the "source node") to all other nodes in the graph

- **Single-Source Shortest Path ?**

The shortest path problem is about finding a path between 2 vertices in a graph such that the sum of the weights of its constituent edges is minimized.(the total sum of weight of edges on path is minimum)

- ***Single-source shortest-paths algorithms ?***

1. Dijkstra's Algorithm

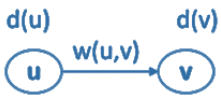
2. Bellman Ford's Algorithm

1. Dijkstra's Algorithm

Algorithm Dijkstra(G,W, S)

1. For each vertex v in G
   1. distance[v] = infinity
   2. previous[v] = Null
2. distance[S] = 0
3. Q = set of vertices of graph G
4. While *Q is not empty*
   1. u = vertex in Q with minimum distance
   2. remove u from Q
   3. for *each neighbor v of u which is still in Q*
      1. alt = distance[u] + W(u,v)
      2. if alt < distance[v]
         1. distance[v] = alt
         2. previous[v] = u
5. Return distance[], previous[]

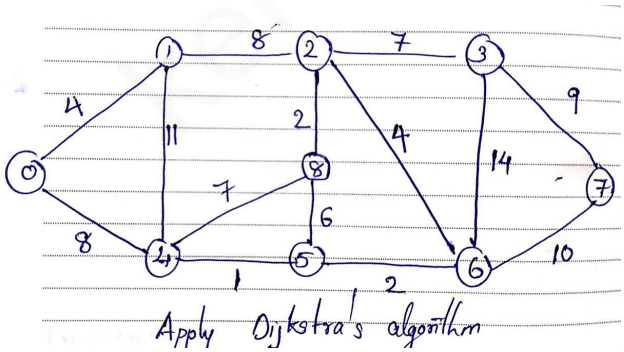
S -source vertex/node



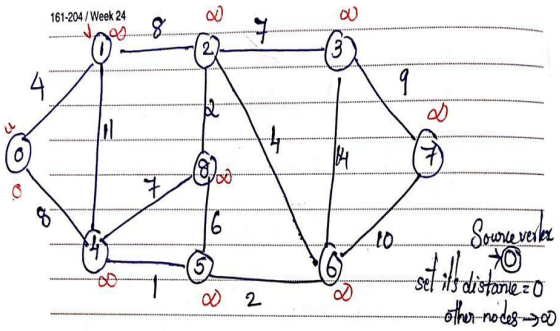
Dijkstra's Algorithm - Complexity

- The complexity mainly depends on the implementation of Q Ueue ,Q
- The simplest version of Dijkstra's algorithm stores the vertex set Q as an ordinary linked list or array, and extract-minimum is simply a linear search through all vertices in Q. In this case, the running time = $\mathbf{O(V^2)}$
- Graph represented using adjacency list can be reduced to **O(E log V)** with the help of binary heap.

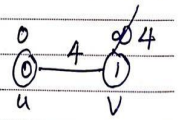
Q)



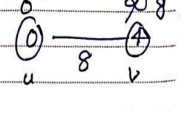
Ans)



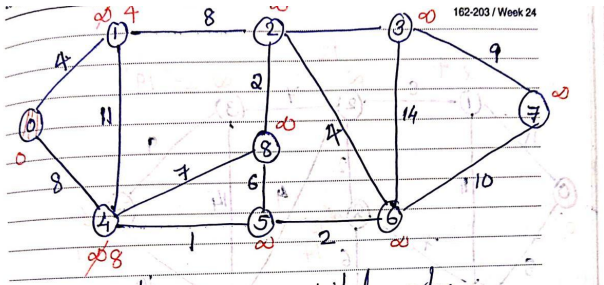
if $d(u) + w(u,v) < d(v)$
$d(v) = d(u) + w(u,v)$



$0 + 4 < \infty$
$4 < \infty$ update



$0 + 8 < \infty$
$8 < \infty$ update

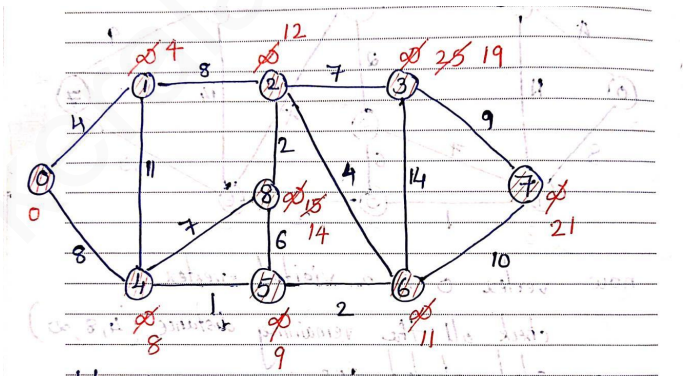


now vertex 0 is a visited vertex
check all the remaining distance (4, 8, ∞)
select shortest one
8 is 4 . ie, vertex (1)
update distance from vertex 1.

from (1) — (2): 4 + 8 < ∞, 12 < ∞ update.

from (1) — (4): 11 + 4 < 8, 15 < 8 ✗
Here 15 > 8, will not update

Vertex (1) visited . check min distance 8 ie, (4) vertex



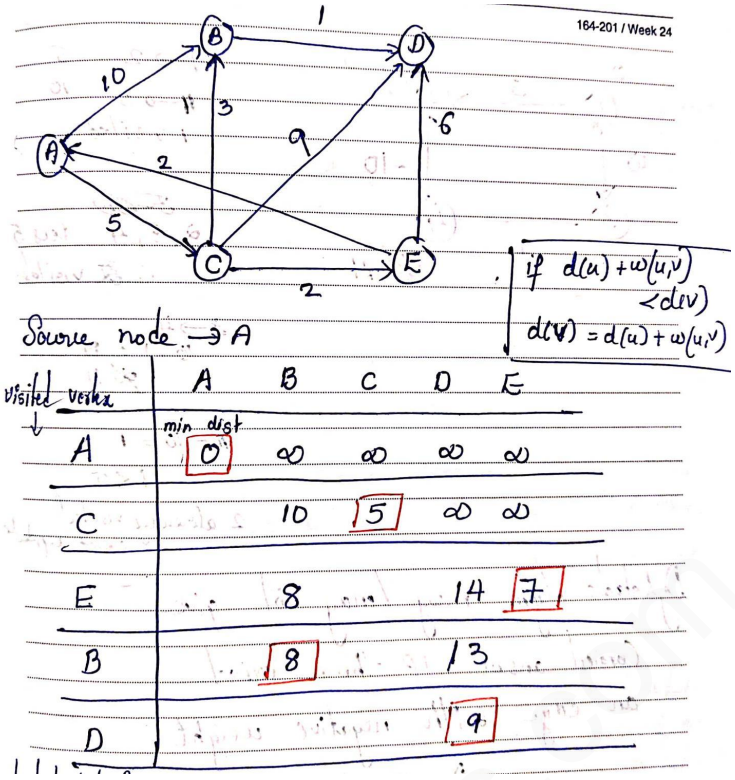
visited vertices → order → 0 1 4 5 6 2 3 7

Shortest dist
0 → 7 : 21
0 → 1 : 4
0 → 2 : 12
0 → 6 : 11
0 → 5 : 9
0 → 3 : 19
0 → 4 : 8
0 → 8 : 14

2019

Q)

Ans)



Advantages of Bellman Ford Algorithm Over Dijkstra's Algorithm

- Dijkstra's algorithm may or may not give correct answer if there exist an edge with negative weight.
- Bellman-Ford algorithm will give correct answer even if there is a –ve weighted edge.