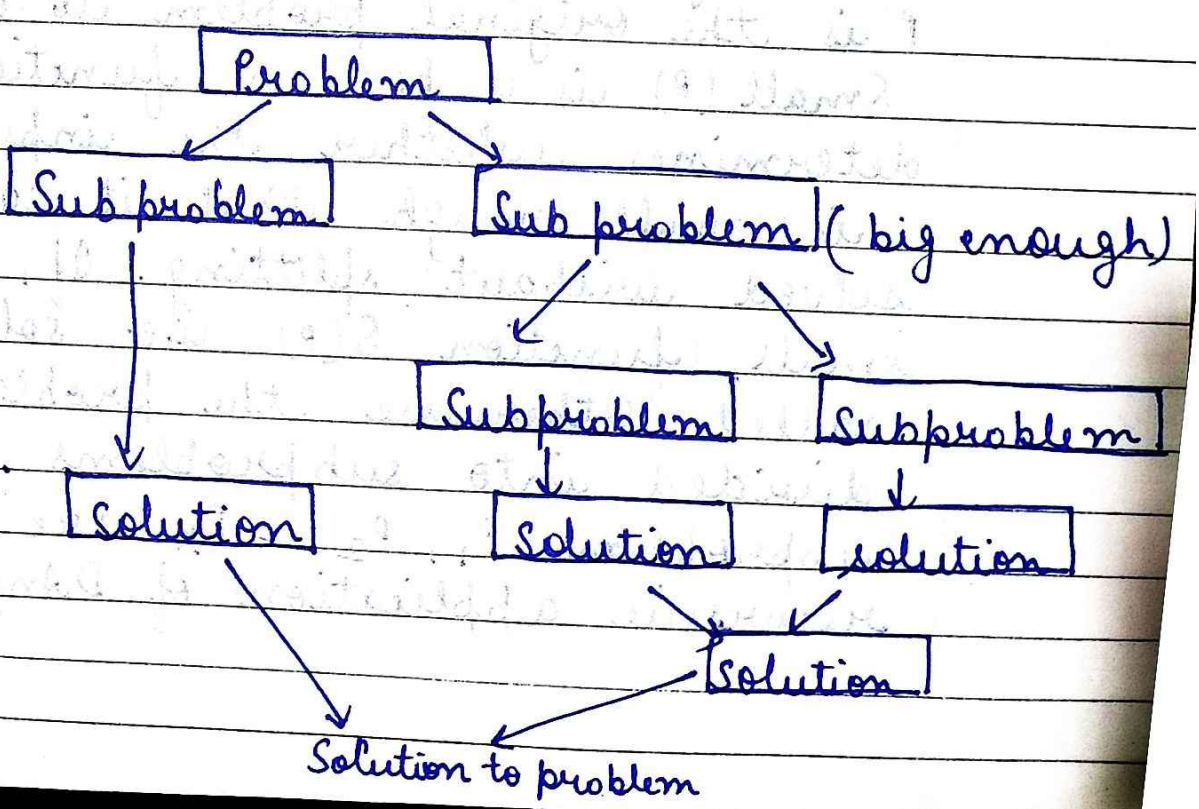
Date

Divide & conquer:

General method:

Given a function to compute n-inputs the divide & conquer strategy says splitting the inputs into K distinct subsets, $1 < K \leq n$, results into K sub-problems. Now, these subproblems are solved separately & then a method must be found to combine subproblems into a solution. If the subproblems are large, then the divide and conquer can be reapplied. The subproblems resulted from big problems are of same type as the original problem. The small & small subproblems are generated until the small subproblems are small enough to be solved without splitted are produced.



Divide & conquer principle is generally implemented using recursive algorithm.

```
Algorithm DAndC(P)
{
    if small(P)
    {
        return S(P);
    }
    else
    {
        divide P into subproblems P1, P2,
            P3 .. PK  K≥1;
        Apply DAndC to each subproblem;
        return (combine (DAndC(P1), DAndC(P2),
                ... DAndC(PK));
    }
}
```

Initially, DAndC(P) is invoked, where P is the original problem to be solved. Small(P) is a boolean function that determines whether the input coming is small enough that it can be solved without splitting. If, input is small, function S(P) i.e. solution is called. Otherwise the problem P is divided into subproblems. These subproblems $P_1, P_2 \ldots P_K$ are solved by recursive application of DAndC. Each subproblem is solved separately & if needed they are again divided into more subproblems. After that, Combine is a function that determines the solution to P using the solutions to K subproblems. If the size of Problem P is n, size of each subproblem is $n_1, n_2, n_3 \ldots n_K$. DAndC is called for each subproblem & then solution is found & then combined into one solution.

Time complexity for DAndC algorithm :

$$T(n) = \begin{cases} g(n) \\ T(n_1) + T(n_2) + T(n_3) \ldots T(n_K) + f(n) \end{cases}$$

where $T(n) \rightarrow$ Time for DAndC algorithm for input size 'n'.

$g(n) \rightarrow$ Time to compute answer directly from subproblems.

$T(n_1), T(n_2) \ldots T(n_K) \rightarrow$ Time for subproblems.

$f(n) \rightarrow$ It is the time for dividing P & combining the solutions to subproblems.

↳ If the size of two subproblems are same, then time complexity,

$$T(n) = \begin{cases} g(n) \\ 2T(n/2) + f(n) \end{cases}$$

Now, if problem is divided into 'a' subproblems & size of each subproblem after dividing 'n' is n/b^k

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

where a & b are constants

e.g. 3.1 $a=2, b=2, f(n)=n, T(1)=2$

$$T(n) = 2(T(n/2)) + n$$
$$= 2(2T(n/4) + n/2) + n$$
$$= 2(2T(n/4)) + 2n$$
$$= 2(2(2T(n/8) + n/4) + 2n$$
$$= 8T(n/8) + 4 \cdot n/4 + 2n$$
$$T(n) = 8T(n/8) + 3n.$$

So, $T(n) = 2^i T(n/2^i) + in.$
where $\log_2 n \geq i \geq 1.$

$$T(n) = 2^{\log_2 n} T(n/2^{\log_2 n}) + n\log_2 n.$$
If $i = \log_2 n.$

$$T(n) = n \cdot T(1) + n\log_2 n$$
$$= n \cdot 2 + n\log_2 n$$
$$= 2n + \log_2 n * n.$$

Binary search :

Let a_i, $1 \leq i \leq n$, be a list of elements that are sorted in non-decreasing order. Consider the problem of determining whether a given element x is present in list. If x is present, return j where x is present in a_i, otherwise j is set to zero.

$P = (n, a_i, \ldots a_l, x)$
where $n \rightarrow$ number of elements
$a_i, a \ldots a_l \rightarrow$ list of elements
$x \rightarrow$ element to be searched.

Divide & conquer can be used to solve this problem. Let Small (P) be true if $n=1$. S(P) will take value i if $x = a_i$, otherwise it will take value 0.

Recursive binary search :

```
Algorithm BinSearch ( a, i, l, x)
{
   if (i== l ) then {
      if( a[i] ==x)
         return i;
      else
         return 0;
   }
   else {
      mid =[i+ l]/2;
    if (x == a[mid])
       return i;
```

```
    else if (a[mid] > x)
        return BinSearch (a, i, mid-1, x);
    else
        return BinSearch (a, mid+1, l, x);
  }
}
```

Iterative Binary Search :-

```
Algorithm BinSearch (a, n, x)
{
    low = 0;
    high = n-1;
    do {
    mid = (low + high)/2;
    if (x < a[mid]) then
        end = mid - 1;
    else if (x > a[mid]) then
        start = mid + 1;
    else
        return mid;
    } while (low ≤ high)
    return 0; // if element is not found,
}                 it will return 0.
```

E.g :-

0	1	2	3	4	5	6	7	8
-15	-6	0	7	9	10	12	16	18

Let x = 12

low = 0, high = 8

$$mid = \frac{0+8}{2} = \frac{8}{2} = 4$$

Now, x > a[mid]

12 > a[4]

12 > 9

then, high = high

start = mid + 1 = 4 + 1 = 5

5	6	7	8
10	12	16	18

high > low, 8 > 5.

start = 5 , end ≠ high = 8.

$$mid = \frac{5+8}{2} = \frac{13}{2} = 6.5 \simeq 7$$

Now, x < a[mid]

12 < 16

high = mid - 1 = 7 - 1 = 6.

6 > 5

$$mid = \frac{6+5}{2} = \frac{11}{2} = 5.5 = 6.$$

Now, x == a[mid],

12 == a[6],

12 == 12

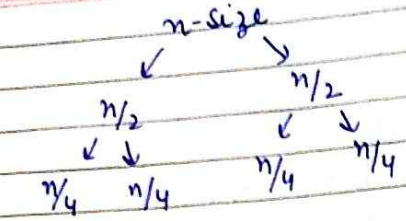
return mid.

So, element found at index 6.

Time complexity :-

Best-case : $\Omega(1)$

Worst-case : $O(\log n)$.



So, every time array is divided into $= \frac{n}{2^k}$

$2^k = n$

$k \log_2 2 = \log_2 n$

$\boxed{k = \log_2 n}$

Average-case: $\theta(\log n)$

Find the maximum & minimum :

The problem is to find the minimum & maximum items in set of n elements.

StraightMaxMin requires $2(n-1)$ elements comparisons in the best, average & worst case.

```
Algorithm StraightMaxMin( a,n,max, min)
  {
    max = min = a[1] ;
   for i = 2 to n do
    {
      if (a[i] > max) then
          max = a[i];
      if (a[i] < min) then
          min = a[i] ;
    }
  }
```

But, there is a problem, a[i] < min is only required when a[i] > max is false. So, we can replace if with elseif.

```
if (a[i] > max) then
      max = a[i]
else if (a[i] < min) then
      min = a[i] .
```

The best case occurs when elements are in in increasing order. The number of element comparisons is n-1. The worst case occurs when elements are in decreasing order. then number of comparisons are $2(n-1)$.

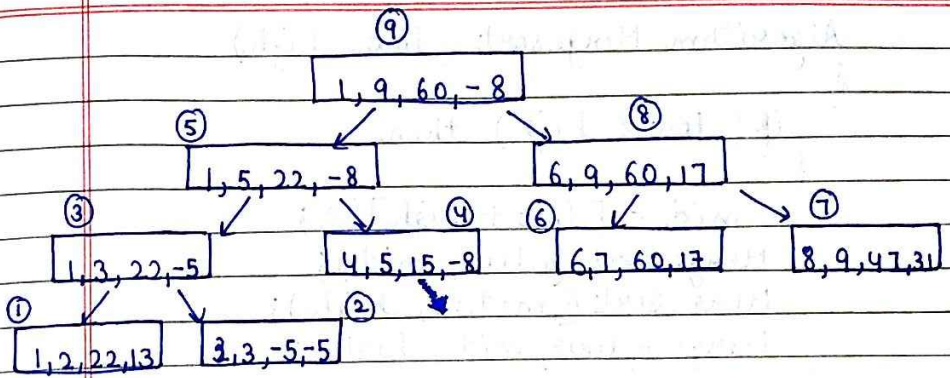
This problem can be implemented using recursion i.e. divide & conquer rule. Let P = (n, a[i] --- a[j]) where n is number of elements in list a[i] --- a[j]. Let Small (P) be true when $n \leq 2$. If n=1, a[i] is the maximum & minimum. If n=2, it means, problem can be solved by making one comparison.

If list has more than 2 elements, P has to be divided into smaller instances. For example, when P is divided into two parts like $P_1 = (n/2, a[1] \text{ --- } a[n/2])$ &

$P_2 = (n - n/2, a[n/2+1], \ldots a[n])$.

We calculate MAX(P_1) & MAX(P_2) & similarly MIN(P_1), MIN(P_2). Now maximum of P_1 is compared with maximum of P_2 & the largest element is largest element from P.

```
Algorithm MaxMin (i, j, max, min)
{
   if (i==j) then
      max = min = a[i];
   else if ( i=j-1) then
    {
      if a[i] < a[j] then {
         max = a[j];
         min= a[i];}
      else {
         max= a[i];
         min = a[j];
      }
   }
else {
   mid =[(i+j)]/2;
   MaxMin (i, mid, Max, Min);
   MaxMin (mid+1, j, max1, min1);
   if (max <max1) then {
      max = max1;
   }
   if ( min > min1) then
   {
      min= min1; }}
}
```



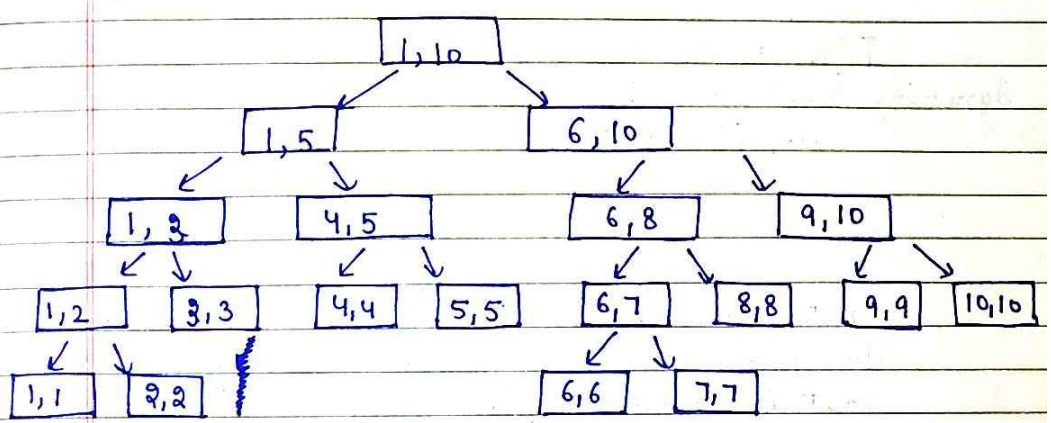
$$T(n) = \begin{cases} T(n/2) + T(n/2) + 2 & n > 2. \\ 1 & n = 2 \\ 0 & n = 1 \end{cases}$$

Merge Sort :-

The Merge Sort algorithm is a sorting algorithm that is based on divide & conquer paradigm. In this algorithim, array is divided into two halves & they are combined into sorted manner.

This algorithm continously splits the array in half until it can't be further divided. This means that if the array is empty or has one element only left, the dividing will stop. If the elements are more than one, recursively merge sort is invoked on both of halves. Finally, when arrays are sorted, the merge operation is applied. Merge operation is the process of taking two smaller sorted arrays & combine them to large one.

```
Algorithm Mergesort (a, low, high)
{
  if (low < high) then
  {
    mid = [low + high]/2;
    Merge Sort (a, low, mid);
    Merge Sort (a, mid+1, high);
    Merge (a, low, mid, high);
  }
}

Merge (a, low, mid, high)
{
  n1 = mid - low + 1;
  n2 = high - mid;
  // Declare two arrays arr1 & arr2
   of size n1 and n2.
   for i=0 to n1 do
     arr1[i] = a[low+i];
   for j=0 to n2 do
     arr2[j] = a[1+mid+j];
i=j=0;
K=low; while i<n1 and j<n2 do
     if arr1[i] < arr2[j] then
       a[K] = arr1[i]; i++;
     else
       a[K] = arr2[j]; j++;
   k++;
while i<n1 do
   a[K] = arr1[i];
   i++;
   k++;
while j<n2 do
   a[K] = arr2[j];
   k++;
   j++;
}
```



Time complexity :-

$$T(n) = \begin{cases} a & n=1,\ a \text{ is constant} \\ 2T(n/2) + cn & n>1,\ c \text{ is constant} \end{cases}$$

When n is power of 2, $n = 2^k$.

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

$$= 2[2T(n/4) + cn/2] + cn$$

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

$$= 4[2T(n/8) + cn/4] + 2cn$$

$$= 8T(n/8) + 4cn/4 + 2cn$$

$$= 8T(n/8) + 3cn$$

$\rightarrow 2^K T(n/n) + KCn$

$\rightarrow 2^K T(1) + KCn.$ $\quad [2^K = n,\ \log_2 n = K]$

$[2^K = n,\ T(1) = a]$

$= an + Cn\log n$

$T(n) = an + Cn\log n.$

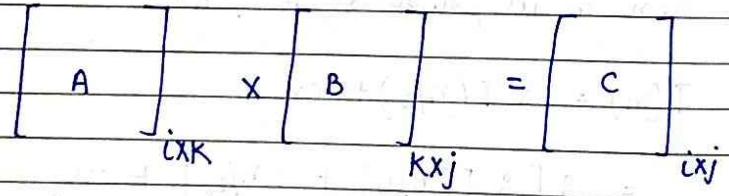
Ignoring constants, $T(n) = n + n\log n.$

So, $T(n) = O(n\log n)$

Strassen's matrix multiplication :-

Let A and B be two $n \times n$ matrices. The product matrix $C = AB$, is also $n \times n$ matrix whose i,jth element is formed by taking elements in the ith row & the jth column of B & multiplying them to get,

$$C(i,j) = \sum_{K=n}^{K=1} A(i,K) \cdot B(K,j)$$



No. of rows of B = No. of columns of A.

Algorithm for multiplication :

```
for i=0 to n do.
    for j=0 to n do
        C[i,j] = 0;
    for K=0 to n do
        C[i,j] += A[i,K] * B[K,j];
    end of for loop.
  end of for loop.
end for loop.
```

To compute C[i,j] we need 'n' multiplications. As the matrix C has n^2 elements, the time complexity for resulting algorithm is $O(n^3)$.

The divide & conquer strategy suggests another way to compute the product of two $n \times n$ matrices. We assume that n is a power 2, i.e. $n = 2^K$ where K is non-negative integer. In case, if n is not a power of 2, then enough rows & columns of zeros can be added to both A & B so that resulting dimensions are of power of two.

Suppose $n = 2^1$, then A and B are each partitioned into four square submatrices, each submatrix having dimensions $n/2 \times n/2$. AB can be computed by using formula $\sum A(i,K)\,B(K,j)$ for product of 2×2 matrices.

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

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

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

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

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

If $n = 2$, the above 4 formulas are computed using multiplication operation for elements of A and B.

If $n > 2$, elements of C can be calculated using matrix multiplication & addition operations applied to matrix size of $n/2$ & $n/2$.

Since n is power of 2, the matrix products can be recursively computed by the same algorithim. This algo. will continously applying to itself until n becomes relatively small.

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

$$A_{11} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \quad A_{12} = \begin{bmatrix} a_{13} & a_{14} \\ a_{23} & a_{24} \end{bmatrix}$$

$$A_{21} = \begin{bmatrix} a_{31} & a_{32} \\ a_{41} & a_{42} \end{bmatrix} \quad A_{22} = \begin{bmatrix} a_{33} & a_{34} \\ a_{43} & a_{44} \end{bmatrix}$$

$$B_{11} = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} \quad B_{12} = \begin{bmatrix} b_{13} & b_{14} \\ b_{23} & b_{24} \end{bmatrix}$$

$$B_{21} = \begin{bmatrix} b_{31} & b_{32} \\ b_{41} & b_{42} \end{bmatrix} \quad B_{22} = \begin{bmatrix} b_{33} & b_{34} \\ b_{43} & b_{44} \end{bmatrix}$$

```
Algorithm SMM(A, B, n)
{
  if (n ≤ 2)
  {
    C11 = A11 B11 + A12 B21
    C12 = a11 b12 + a12 b22
    C21 = a21 b11 + a22 b21
    C22 = a21 b12 + a22 b22
  }
else
  { mid = n/2.
   SMM(A11, B11, n/2) + SMM(A12, B21, n/2)
   SMM(A11, B12, n/2) + SMM(A12, B22, n/2)
   SMM(A21, B11, n/2) + SMM(A22, B21, n/2)
   SMM(A21, B12, n/2) + SMM(A22, B22, n/2)
  }
}
```

Recurrence relation for above algorithm,

$$T(n) = \begin{cases} b & n \le 2 \\ 8\,T(n/2) + cn^2 & n > 2 \end{cases}$$

[8 Matrix multiplication call for $n/2$
Addition if $n = 2$, requires 4 additi-ons i.e. n^2.]

Strassen's has discovered a way to compute C(i,j) using 7 multiplications & 18 additions or subtractions.

1.) Firstly, compute the seven $n/2 \times n/2$ matrices, P, Q, R, S, T, U and V.
2.) Then compute C(i,j).

P, Q, R, S, T, U and V is calculated using 7 matrix multiplications and 10 matrix additions or substractions.

Then, C_{ij} requires additional 8 additions or subtractions.

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

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

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

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

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

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

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

$$C_{11} = P + S - T + V$$
$$C_{12} = R + T$$
$$C_{21} = Q + S$$
$$C_{22} = P + R - Q + U$$

So, resulting recurrence relation,

$$T(n) = \begin{cases} b & n \le 2 \\ 7T(n/2) + an^2 & n > 2 \end{cases}$$

where a & b are constants.

So, $O(n^{\log_2 7}) = O(n^{2.81}) \simeq O(n^2)$.

Quicksort :-

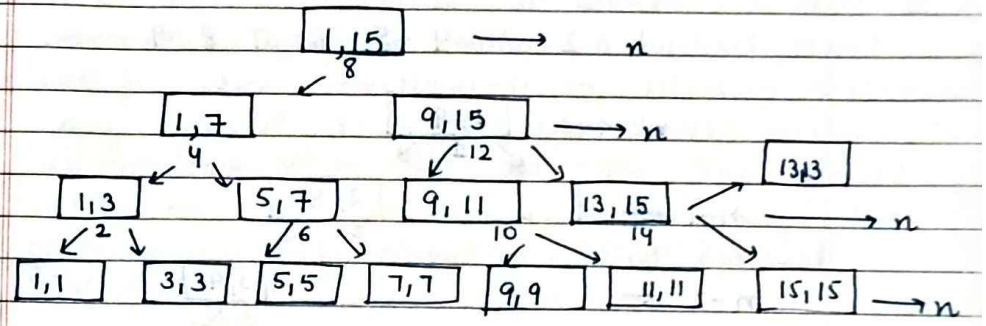
Quicksort is a sorting algorithm which is based on divide & conquer paradigm. In merge sort a[1:n] was divided at its mid point into subarrays which were independently sorted & later merged. In quicksort, the division into two subarrays is made so that the sorted subarrays need not to be merged. The elements in a[1:n] is rearranged such that $a[i] \le a[j]$ for all i between 1 & m and for all j between m+1 & n. So a[1:m] & a[m+1 :n] is sorted independently, thus need not to be merged. The rearrangement is done in such a manner that, we picked one element of a[], say pivot, pivot = a[s], and then reordering the other elements so that all elements before pivot in a[1:n] are less than or equal to pivot & all elements appearing after it are greater than equal to pivot. This rearranging is referred to as partitioning.

```
Algorithm QuickSort (a, p, q)
{
    if (p < q) then
        loc = partition (a, p, q);
        quicksort (a, p, loc-1);
        quicksort (a, loc+1, q);
```

```
    // There is no need to combine solution
  }
}

Algorithm partition (a, p, q)
  {
    pivot = a[p];
    low = p;
high = q;
  repeat
  {
    repeat
    { low++;
    } until (a[low] ≤ a[pivot])

    repeat
     high--;
     until ( a[high] > a[pivot])
  if ( low < high)
    interchange (a, low, high);
} until (low ≤ high);

  Interchange ( a, high, pivot);
    return high;
}

Algorithm interchange (a, i, j){
    p = a[i];
    a[i] = a[j];
    a[j] = p;
}
```

Time complexity :

1. Best case : The partition is always done at the middle.




```
Algorithm quicksort (a,p,q)
{
   if(p<q) then
n  ←  loc= partition (a,p,q);
      quicksort (a,p, loc-1);
      quicksort (a, loc+1, q);
}
```

The partition function does 'n' comparison to divide the array.

The height of tree = $\frac{n}{2}$ [The 'n' is splitted to half everytime]

$\frac{n}{2}$

$\frac{\frac{n}{2}}{2}$

$\vdots$

$\frac{n}{2^k} = 1$ until 1 element left

$n = 2^k$

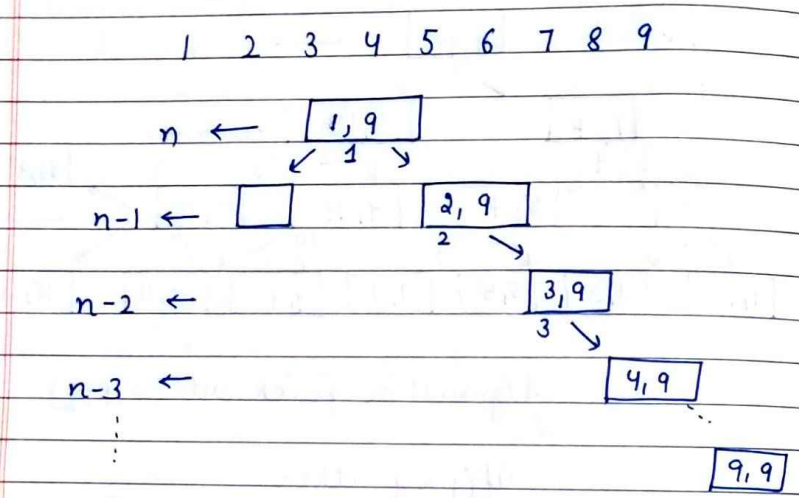
$\log_2 n = k$.

So, the height of tree = $\log n$

Time complexity = $n \log n$

So, best-case complexity is $O(n \log n)$.

2. Worst-Case: It happens when array is already sorted.



$$= n + n-1 + n-2 \cdots + 1$$

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

$$O\left(n^2 + \frac{n}{2}\right) \simeq O(n^2)$$

So, worst-case time complexity is $O(n^2)$.

(i) If the pivot element is the middle element of array, the worst-case can be converted into best-case.

(ii) Select ~~mddd~~ random element as pivot.

Space complexity : $\log n$ to n.

Greedy method :

The general method :

In greedy method, a problem has 'n' inputs & problem is stored in such a manner that we obtain a subset that satisfies some constraints or selection creteria. Any subset that satisfies these constraints is called a feasible solution. The feasible solution that maximizes or minimizes the given objective function is called optimal solution.

The greedy method suggests that, algorithm can divide the algorithm into stages, considering one input at a time. At each stage, a decision is made that a particular input will give a optimal solution or not. This is done by considering the inputs in the order determined by selection procedure. If the inclusion of next input into partially constructed optimal solution will result in an infeas-ible solution, then the input is n't added to partial solution. Otherwise, it is added. The selection procedure is also based on some optimization measure. Different optimi-zation measures are possible for a given solution/problem.

Greedy algorithm satisfies two paradigm i.e. subset & ordering paradigm.

Subset paradigm satisfies that only those inputs should be included in the

set that can satisfies the objective function or lead to feasible solution.

Ordering paradigm is used to select the inputs in the set in such a order that can meet the objective function. Each decision is made using optimization criteria that can be computed using decision already made.

```
Algorithm Greedy (a,n)
  // a[0:n] contain n inputs
  {
     solution = ϕ; // Initialize solution
    for i = 0 to n do
     {
        x = Select (a);
        if Feasible ( solution, x) then
           solution = union (solution, x);
     }
    return solution;
  }
```

The function Select() selects an input from a[] & removes it. The selected input value is assigned to x. Feasible is a Boolean-valued function that determines whether x can be included into the solution vector. If it returns True, then the function Union combines x with the solution & update the solution vector. The function Greedy describes the essential way that a greedy algorithm will look.

Knapsack Problem :

In Knapsack problem, we are given 'n' objects & a knapsack or a bag having capacity 'm'. Object 'i' has a weight w_i & profit p_i. The objective is to obtain a filling the knapsack that maximizes the total profit earned. Since, the knapsack capacity is m, we require that total weight of all chosen objects to be at most m.

If weight of all given items is less than or equal to m, all are choosen. If the total weight exceeds the capacity, we can't pack them all. In that case, we choose subset of items / object that can fit in knapsack with maximum profit.

So, knapsack problem can be stated as,

$$\text{maximize} \sum_{1 \le i \le n} p_i$$

$$\text{subject to} \sum_{1 \le i \le n} w_i \le m.$$

The profits & the weights are positive numbers.

There are three greedy strategies to obtain feasible solutions,

(i) First, trying filling the knapsack by including object with maximum profit.

ii) Secondly, considering the objects in order of non-decreasing weights w_i.

iii) Last, is to achieve a balance between the rate at which profit increases & the rate at which capacity is used. So, we include object which has maximum profit per unit of capacity used. So, objects are considered in order of p_i/w_i.

Each of above strategy requires $O(n)$ time.

e.g.: $i = 7$

$\{P_1 \dots P_7\} = (10, 5, 15, 7, 6, 18, 3)$

$\{W_1 \dots W_7\} = (2, 3, 5, 7, 1, 4, 1)$

Knapsack capacity $(m) = 15$.

Object	1	2	3	4	5	6	7
P_i	10	5	15	7	6	18	3
W_i	2	3	5	7	1	4	1
P_i/W_i	5	1.67	3	1	6	4.5	3

1. Selecting objects on basis of maximum profit :

Obj	P_i	W_i	Remaining Weight
6	18	4	15-4 = 11
3	15	5	11-5 = 6
1	10	2	6-2 = 4
4	4	4x1=4	4-4 = 0

$\Sigma = 47$

2. Selecting object on basis of increasing weight or with minimum weight :

Obj	P_i	W_i	Remaining weight
5	6	1	15-1 = 14
7	3	1	14-1 = 13
1	10	2	13-2 = 11
2	5	3	11-3 = 8
6	18	4	8-4 = 4
3	12	4	4-4 = 0

$\Sigma = 54$

3. Selecting object on basis of maximum P_i/W_i :

Obj	P_i	W_i	Remaining Wt.
5	6	1	15-1 = 14
1	10	2	14-2 = 12
6	18	4	12-4 = 8
3	15	5	8-5 = 3
7	3	1	3-1 = 2
2	2x1.67	2	2-2 = 0

$\Sigma = 55.34$

So, the maximum profit is taken when objects are included on basis of maximum P_i/W_i.

```
Algorithm MinWeight (wt, pf, n)
{
    // W is capacity of knapsack
    // wt array of decreasing order.
    // pf corresponding profits
    // n number of values in
          array.
// n is initilizated to size of array.

if (n == 0 || W == 0) then
    return 0;
else
  return max((pf[n-1] + knapsack(W - wt[n-1],
                wt, pf, n-1)), knapsack(W,
                wt, pf, n-1))
}

Algorithm MaxRatio (W, wt, pf, n)
  {
    for i = 1 to n do
          x[i] = 0
    weight = 0; profit = 0;
    for i = n to 1 do {
      if weight + w[i] ≤ W then
          x[i] = pf[i] / w[i]
          weight = weight + w[i]
                profit = profit + pf[i]
      else
          x[i] = pf[i] / w[i]
      profit = profit + (W - weight) * x[i]
       break.
  }
return profit;}
```

In above algorithm, arrays are sorted in increasing order of P_i/w_i ratio.

```
Algorithm MaxProfit (W, wt, pf, n)
  {
    weight = 0
    profit = 0.
   for i = n to 1 do
      if weight + w[i] ≤ W then
          weight = weight + w[i]
          profit = profit + pf[i]
      else
          x = pf[i] / w[i].
        profit = profit + (W - weight) * x.
        break.
    return profit
  }.
```

Job sequencing with deadlines :-

In this problem, there are set of n jobs. Associated with job 'i' is an integer deadline $d_i \geq 0$ and a profit $P_i > 0$. For any job 'i', the profit P_i is earned if the job is executed 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 subset J of jobs is a feasible solution such that each job is completed by its deadline. The value of a feasible solution

J is sum of profits of jobs in J. A optimal solution is a feasible solution with maximum value.

Since, the problem involves the identification of a subset, it fits the subset paradigm.

```
Algorithm JS(d, p, n)
{
    dmax = max(d);
    for i=1 to n do
        set K = min(dmax, deadline(i))
        while K ≥ 1 do
            if timeslot[K] == empty then
                timeslot[K] = job[i]
                break
            end if
            Set K = K-1
        end while.
    for i= 1 to dmax then
        profit = profit + p[i].
    return timeslot
}
```

e.g.→

Job	Deadline	profit
A_1	2	60
A_2	1	100
A_3	3	20
A_4	2	40
A_5	1	20

Sorting the array in decreasing order of max. profit,

Job	A_2	A_1	A_4	A_5	A_3
Deadline	1	2	2	1	3
Profit	100	60	40	20	20

(i) $d_{max} = 3$.

$K = \min(3, 1)$

$K = 1$

while $1 \geq 1$ do.

timeslot [1] = empty, true

timeslot [1] = job [1]

$K = 0$

1	2	3
A_2	A_1	A_3

(ii) $K = \min(2, 3)$

$K = 2$

while $2 \geq 1$ do

timeslot [2] = empty, true

timeslot [2] = job [2]

$K = 1$

while $1 \geq 1$ do

timeslot [1] = empty, false

$K = 0$.

(iii) $K = \min(3, 3)$

$K = 3$

while $3 \geq 1$ do

timeslot [3] = empty then

timeslot [3] = job [5]

$K = 2$, timeslot [2] = empty, false

$K = 1$, timeslot [1] = empty, false.

2.

Job	Deadline	Profit
A_1	2	100
A_4	1	27
A_3	2	15
A_2	1	10

1	2
A_4	A_1

(i) $i=1$, $d_{max} = 2$

$K = (2, 2)$

$K = 2$.

while $2 \geq 1$ then

if timeslot[2] = empty then

timeslot[2] = job[1]

~~K=1~~ break.

(ii) $i = 2$

$K = \min(1, 2)$

$K = 1$

while $1 \geq 1$ do

if timeslot[1] = empty, true

timeslot[1] = job[2]

break.

$K = 0$.

Spanning Tree :

A spanning tree of a graph is just a subgraph that contains all the vertices & is tree. That is, a spanning tree of a connected graph G contains all the vertices & has the edges which connect all the ~~edges~~ vertices. So, the number of edges is 1 less than the number of vertices.

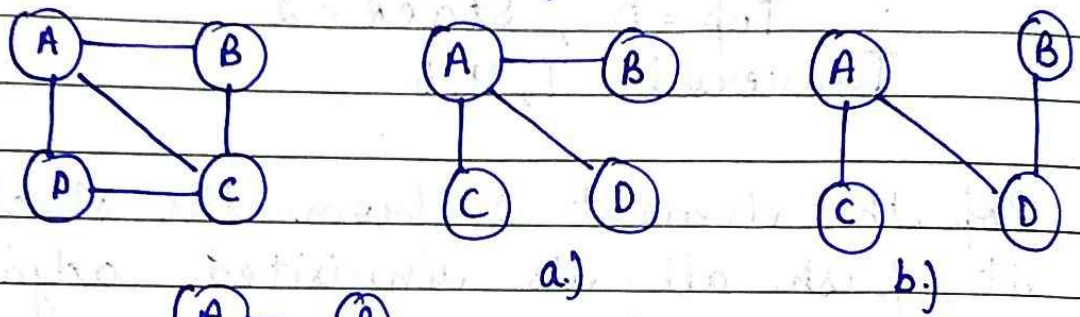
$G(V, E)$

$G'(V', E')$

$V' = V$

$E' \subset E$

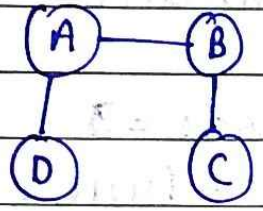
$E' = |V| - 1$



a.)

b.)

c.)



All the trees have edges 1 less than vertices.

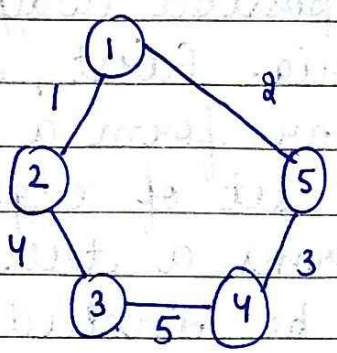
If the graph is not connected, i.e. graph with n-vertices has n-1 edges, spanning tree is not possible.

Page No.
Date

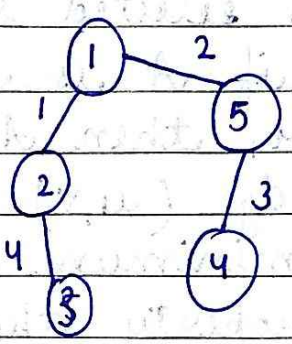
Minimum spanning tree :

When, a connected weighted graph, then to create a spanning tree T for Gr, such that the weight of edges in T is as small as possible. Such a tree is called minimum spanning tree.

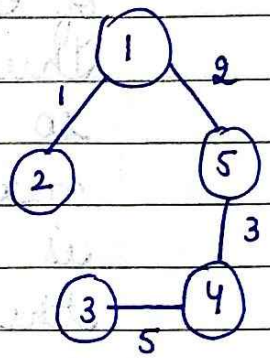
E.g.



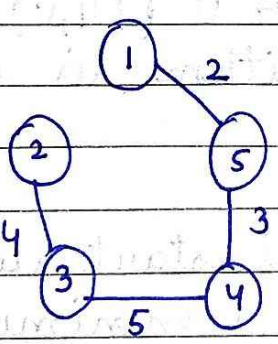
→ possible spanning trees.



Cost = 10



Cost = 11



cost = 14

So, the minimum spanning tree is tree with cost 10.

The minimum spanning tree should follow below rules :

1. If Gr is a graph with vertices V and E. Then, minimum spanning tree Gr' should be, Gr(V', E') such that.
   $V' = V$
   $E' \subset E$ or $E' = |V| - 1$
2. The minimum spanning tree shouldn't create a cycle.
3. The MST should be connected.

1. Prim's Algorithm :-

This method builds the tree edge by edge. The next edge to include is chosen according to some optimization creteria. The simplest creteria is to choose an edge that results in minimum increase in the sum of the costs of edges so far included.

There are two possible ways to interpret this creteria. First, the set of edges selected so far form a tree. Thus, if A is the set of edges selected so far, then A forms a tree. The next edge (u,v) to be included in A is a minimum-cost edge not in A with the property that A ∪ {(u,v)} also forms a tree. This algorithm is called Prim's algorithm.

The algorithm will start with a tree that includes only a minimum-cost edge of G. Then, edges are added to this tree one by one. The next edge (i,j) to be added is such that 'i' is a vertex already included in tree, j is not included ~~included~~ yet (j is adjacent to i) and the cost of cost[i,j] is minimum among all edges (K,l) such that K is in tree & vertex l is not in tree. To determine this edge (i,j) efficiently, we associate 'i' with each vertex j not yet included in tree with value near[j]. The value near[j] is a vertex in a tree such that cost[j, near[j]] is minimum among all choices for near[j].

The near[j] is set to 0 for all vertices j that are included in tree. The next edge to be included is defined by j such that near[j] ≠ 0 & cost[j, near[j]] is minimum.

```
Algorithm Prim(E, cost, n, t)
// E → Set of edges
// cost[1:n, 1:n] is cost adjacency matrix
   of an n vertex graph such that cost[i,j]
   is either positive number or ∞ if no
   edge exists.
// The minimum spanning tree is stored as
   set of edges in the array t[1:n-1, 1:2]
   such that (t[i,1], t[i,2]) is an
   edge in minimum spanning tree.
   The final cost 'mincost' is returned.
{
   let (K,l) be an edge of minimum cost
      in E;
   mincost = cost[K,l];
1. t[1,1] = K; t[1,2] = l;
2. for i = 1 to n do // initialize near
      if cost[i,l] < cost[i,K]
         near[i] = l;
      else
         near[i] = K;
```

```
3.  Set near[K] = near[1] = 0;
4.  for i=2 to n-1 do
    {
      // Find n-2 edges for tree.
      let j be index where near[j] ≠ 0 &
      cost [j, near[j]] is minimum.
      t[i,1] = j ; t[i,2] = near[j];
      near[j] = 0;
mincost=  mincost + cost[j,near[j]];
      for K=1 to n do
          if (near[K] ≠ 0 and cost(K,near[K]) >
                                  cost(K,j))
              then near[K] = j;
    }
5.  return mincost;
}
```

The time complexity of Prim's algorithm is $O(n^2)$. [as, the step 4 has complexity $O(n)$ and nested loop has $O(n)$. So, total complexity is $O(n^2)$.]

Kruskhal's algorithm :-

In Kruskhal's algorithm, edges are considered in non-decreasing order i.e. increasing order of cost. In this algo, the set 't' of edges selected at the end will result into a tree ie. it is possible to complete 't' into a tree. Thus t may not be a tree at all stages of algorithm. It will generally only be a forest & set of edges t can be completed into a tree if there are no cycles in t. This method is Kruskal algorithm.

```
1.  t = ∅ ;
    while ((t has less than n-1 edges) & E ≠ ∅)) do
    {
01. choose an edge (v,w) from E of
      lowest cost;
02. Delete (v,w) from E;
03. if (v,w) doesn't form a cycle in t
          then add (v,w) into t;
      else
          discard (v,w);
    }
```

The time complexity of above method is $O(|V||E|)$.

As $|V| = n$, $|E| = n-1$

So, $O(n(n-1)) = O(n^2) + O(n)$

$= O(n^2)$

If the edges are maintained as a minheap, then the next edge to consider can be obtained in $O(\log|E|)$ time. The construction of heap itself takes $O(|E|)$ time.

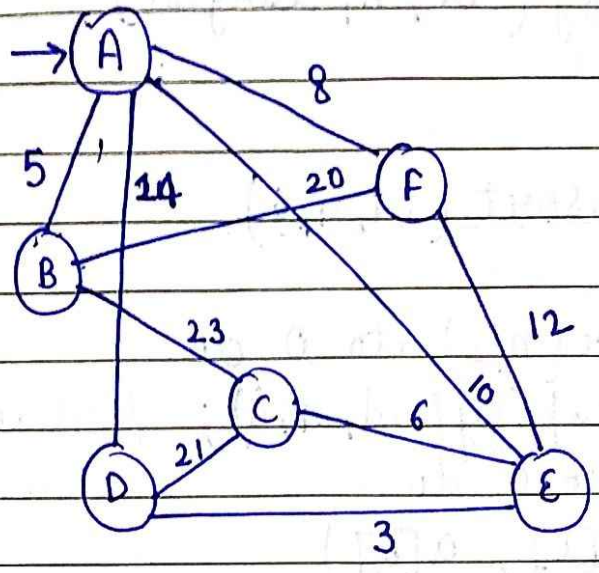
To determine whether an edge inclusion forms a cycle, the vertices in t should be grouped together in such a way that one can easily determine whether the vertices v and w are already connected by earlier selection edges. If they are, then the edge (v,w) will be discarded. If they aren't, the edge

(v,w) is added in t. One possible way of grouping is to place all the vertices in same connected component of it into a set. So, two vertices v & w, if connected are in same set. E.g. → The edge (2,6) is to considered, the sets are {1,2}, {3,4,6} & 5. Vertices 2 & 6 are not in same set, so these sets are combined → {1,2,3,4,6} & {5}. The next edge to be considered is (1,4). Since 1 & 4 are in same set, the edge is rejected. The next edge (3,5) connect vertices in two sets into single set & result in formation of minimum spanning tree.

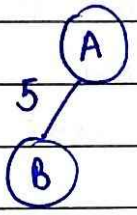
Algorithm Kruskal(E, cost, n, t) {
Construct a heap out of edge costs using heapify.
for i = 1 to n do ; parent[i] = -1;
// Each vertex is in different set.
i = 0; mincost = 0.0;
while (i < n-1 && (heap not empty)) do
{ Delete a minimum cost edge(u,v) from heap & reheapify using Adjust;
j = Find(u); k = Find(v);
if (j ≠ k){
i = i+1;
t[i,1] = u ; t[i,2] = v;
mincost = mincost + cost[u,v];
Union(j,k);
}
}
if (i ≠ n-1) then write "No MST";
else { return mincost; } }

// Examples of Prim's Algo & Kruskal's algo;

1. Prim's algo:

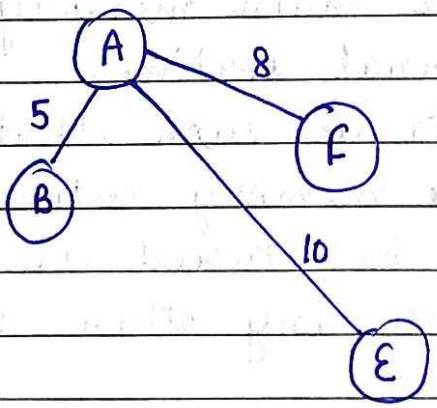


a.) Choose mincost edge.



Neighbours of A → ~~B~~, F, E, D
B → ~~A~~, F, C.
Mincost = 5

b.) A → F cost is minimum among all the choices present.

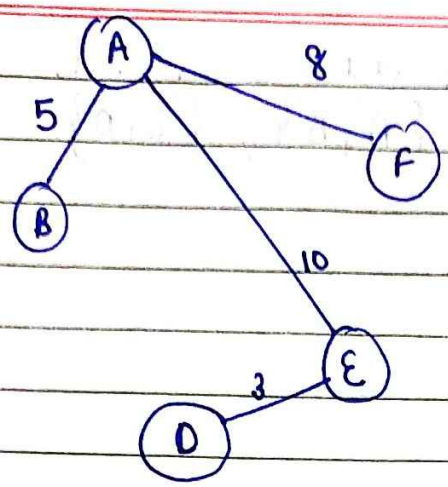


Mincost = 13

Neighbours of F → E, B
(12), (20)

c.) A → E, D
(10), (20)
Minimum cost is Mincost = 23 AE, so it is selected.

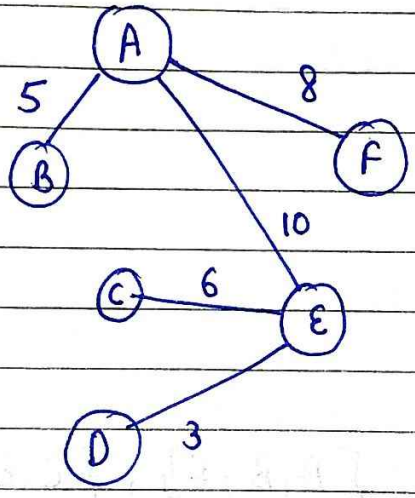
d.) E → C = 6, E → D = 3
So, ED edge is selected.



Mincost = 26

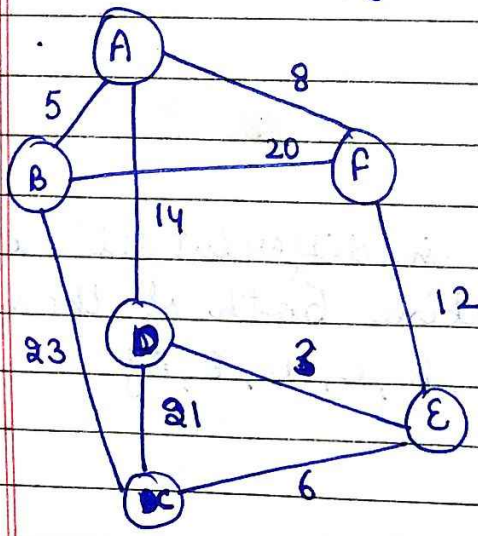
e) $D \to C = 21$, $E \to C = 6$, $B \to C = 23$, $A \to C = \infty$, $F \to C = \infty$

So, edge EC is selected.



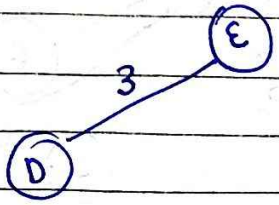
Mincost = 32

Kruskal's algorithm :

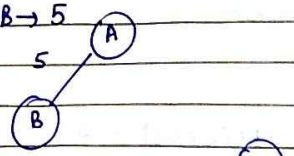


→

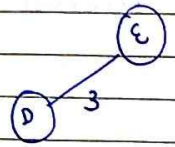
(i) $DE \to 3$



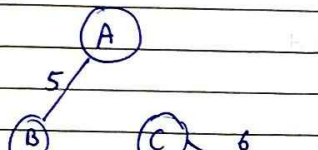
(ii) AB → 5



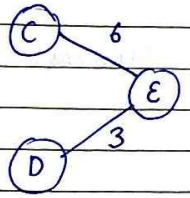
{A, B}, {E, D}



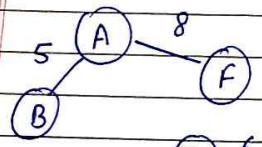
(iii) EC → 6



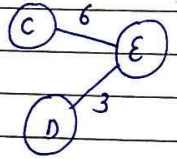
{A, B}, {C, E, D}



(iv) AF → 8

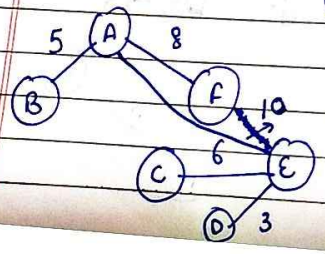


{A, B, F}, {C, E, D}



) EA → 10

[EA are in different set, so we combine both of them]
{A, B, C, D, E, F}



Single source shortest path :

In directed graph, $G = (V, E)$, a weighting function costs for edges of G, vertex v_0 referred as source vertex. The problem is to determine the shortest path from v_0 to all remaining vertices of G called destination. All the weights are positive.

The greedy way to generate shortest path from v_0 to remaining vertices is to generate these paths in increasing order. First, a shortest path of nearest vertex is generated. Then, a shortest path to next nearest vertex is generated & so on.

So, we determine

(i) The next vertex to which a shortest path is generated.

(ii) A shortest path to that vertex.

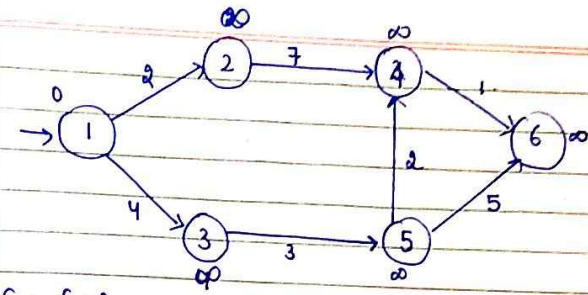
Let S be set of vertices to which shortest path is already determined. For vertex w not in S, let dist[w] be length of shortest path starting from w_0, going through only those vertices that are in S & ending at w.

```
Algorithm ShortestPath( v, cost, dist, n)
{
  // dist[j], 1 ≤ j ≤ n, is set to length to path
     for vertex v i.e. source to j in
     graph G with n vertices. dist[v]
     is set to zero. G is represented by
```

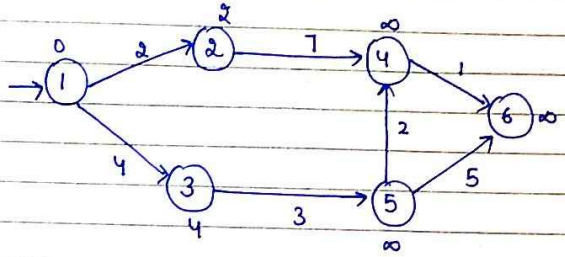
```
cost adjacency matrix cost[1:n, 1:n].
{
  for i=1 to n do{
     Set S[i] = false ;
     dist [i] = cost [v, i];
  }
  S[v] = true ; dist [v] =0.0;
  // As source vertex is included in
     Set S.
  for num 2 to n-1 do
    {
       Choose u from among those
       vertices not in S & dist [u] is
       minimum.
       S[u] = true; // Put u in S.
     for( each w adjacent to u, S[w] = false)
       do
       {
          if ( dist [w] > (dist [u] + cost[u,w]))
             then
                dist [w] = dist [u] + cost [u,w];
             // Update distances.
       }
    }
}
```

Time complexity for single ~~short~~ source shortest path algorithm is $O(n^2)$

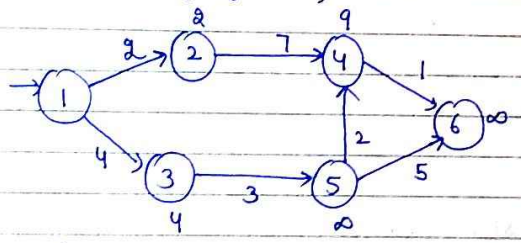
e.g.



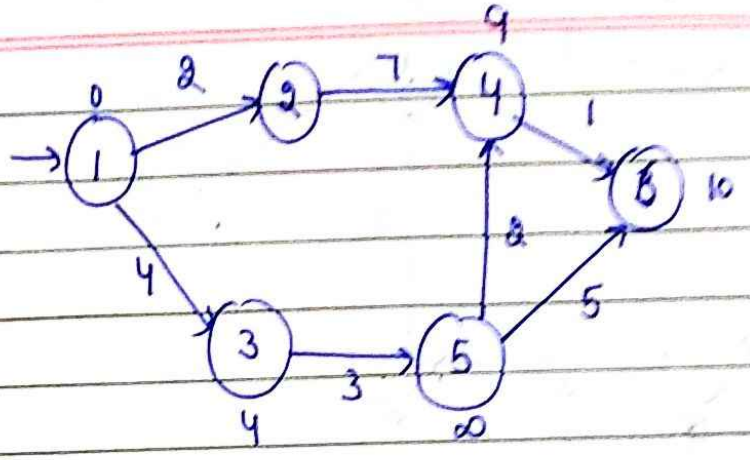
(i) $S = \{1\}$
dist [1] = 0;
Vertex 1 is the source vertex.
So, dist [2] = 2
dist [3] = 4.



(ii) $S = \{1, 2, 3\}$, Let $w = 4$, $u = 2$
if (dist [w] > dist [u] + cost [u, w])
$\infty > 2 + 7$; $\infty > 9$. Yes,
dist [w] = 9;



(iii) $S = \{1, 2, 3, 4\}$
$u = 4$, $w = 6$.
dist [6] > 9+1 ; $\infty > 10$.
So, dist [6] = 10.



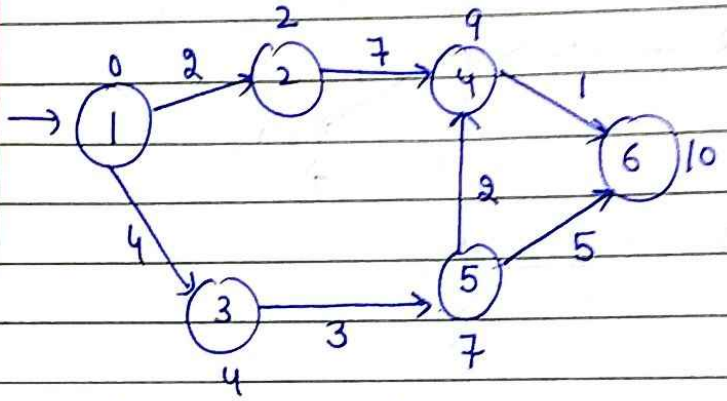
$S = \{1, 2, 3, 4, 6\}$

Let $u = 3$, $w = 5$

dist [5] > dist [3] + cost [3,5]

$\infty > 4 + 3$; $\infty > 7$

So, dist [5] = 7.



Let $u = 5$, $w = 4$.

if dist [4] > dist [5] + cost [5, 4]

$9 > 7 + 2$, No.

Let $u = 5$, $w = 6$.

if dist [6] > dist [5] + cost [5, 6]

$10 > 7 + 5$

$10 > 12$, No.

So, final shortest path.