Divide and Conquer.

General Method:-

If a given problem is small then directly solve that problem.

If a given problem is large then apply divide and conquer approach to it.

First, a large problem can be divided into small subproblems that are possible.

Next, these subproblems are individually solved to obtain subsolutions to them.

Then these sub-solutions are combined to get the solution for the original large problem

```
Algorithm DandC(P)
{
    if (Small(P)) then return S(P);
    else
    {
        divide P into smaller instances
        $P_1, P_2, \ldots P_k$ ; $k \geq 1$;
        Apply DandC to each of these substances
        return combine(DandC(P1), DandC(P2)...
                              ...DandC(Pk));
    }
}
```

Control abstraction for divide-and-conquer.

Small(P) is a boolean valued function that determines whether a problem is small or not.

If it is small then solve it without using dividing method.

If this is true then function S is invoked to give solution to that problem.

otherwise, the problem is divided into smaller subproblems.

These subproblems $P_1, P_2 \cdots P_K$ are solved by recursive applications of DandC procedure.

Combine is a function that determines the solution to P using the ~~sub~~ solutions to the K subproblems.

If the size of the problem P is n and the size of the K subproblems are $n_1, n_2 \cdots n_K$ respectively then the computing is described by the recurrence relation.

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

Here T(n) is the time for DandC procedure on any of size n.

g(n) is the time for computing the answer directly for small inputs.

The function f(n) is the time for dividing P and combining the solutions to subproblems.

Each subproblem is same as the original problem so we use recursion to solve these problems.

The time complexity of divide and conquer algorithms take the form.

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

where a and b are constants.
We assume that $T(1)$ is known value. and n is a power of b ($\therefore n = b^k$).

Binary search

In the binary search method, all elements are sorted in non-decreasing order i.e., ascending order

If we want to search for the key element x then first divide the list at its middle position so that two sublists are created.

If key is greater than middle element then key is searched in the right sublist. otherwise key key is searched in the left sublist.

· Iterative Approach

In this method, binary search process can be done by looping statements.

```
Algorithm BinSearch(a, n, x)
// Given an array a[1:n] of elements in
// non-decreasing order, n>=0
// Determine whether x is present and
// if so, return j such that x = a[j] else return 0
{ low=1; high=n;
  while (low<=high) do
  { mid:= (low+high)/2;
```

```
    if (x< a[mid] then high = mid-1;
    else if (x > a[mid] then low = mid+1;
    else return mid;
  }
  return 0;
}
```

Ex:- a

1	2	3	4	5	6	7	8	9	10
10	20	30	40	50	60	70	80	90	100

$x = 40$

low = 1, high = 10 low ≤ high ⇒ 1 ≤ 10 . True

mid = (low+high)/2 = (1+10)/2 = 11/2 = 5

Here x < a[mid] i.e., 40 < a[5] ⇒

⇒ 40 < 50 True

So high = mid-1 = 5-1 = 4

low = 1, high = 4 low ≤ high ⇒ 1 ≤ 4 True

mid = (1+4)/2 = 5/2 = 2

Here x > a[mid] i.e., 40 > a[2] ⇒ 40 > 2 True.

So low = mid+1 = 2+1 = 3

high = 4 low = 3

low ≤ high ⇒ 3 ≤ 4 True.

mid = (3+4)/2 = 7/2 = 3

Here x > a[mid] ⇒ 40 > 30 True

low = mid+1 = 3+1 = 4

low = 4, high = 4

low ≤ high ⇒ 4 ≤ 4

mid = (4+4)/2 = 4

Here x = a[mid] ⇒ 40 = 40 True.

return 4

So 40 is found at the index 4.

Recursive Approach

In this method, the binary process can be done by using recursion.

```
Algorithm BinSearch(a, low, high, x)
// Given array a[low:high] of elements in
// non-decreasing order, 1 ≤ low ≤ high.
// Determine whether x is present and if so
// return j such that x = a[j]; else return 0
{
    if (low == high) then // Problem is small
    {
        if (x == a[low]) then return high;
        else return 0;
    }
    else
    {
        // Reduce problem into subproblems.
        mid := (low+high)/2;
        if (x == a[mid]) then return mid;
        else if (x < a[mid]) then
        return BinSearch(a, low, mid-1, x);
        else return BinSearch(a, mid+1, high, x);
    }
}
```

Ex:- a

1	2	3	4	5	6	7	8	9	10
10	20	30	40	50	60	70	80	90	100

$x = 40$

low = 1 high = 10

low == high

$1 \neq 10$ True

mid = (low+high)/2 = (1+10)/2 = 11/2 = 5

$x <$ a[mid] => 40 < a[5] => 40 < 50 True.

high = mid - 1 = 5 - 1 = 4 , low = 1

~~return~~ Binsearch(a, low, mid-1, x);

Binsearch(a, 1, 4, 40);

$1 \neq 4$ True.

So mid = (1+4)/2 = 5/2 = 2

$x >$ a[mid] => $x >$ a[2] => 40 > 20 True

low = mid+1 = 2+1 = 3

high = 4

Binsearch(a, mid+1, high, x);

Binsearch(a , 3, 4, 40)

$3 \neq 4$ True

mid = (3+4)/2 = 7/2 = 3

$x >$ a[3] => ~~4 x~~ 40 > 30 True

low = mid + 1 = 3 + 1 = 4

low = 4, high = 4 = Binsearch (a, 4, 4, 40)

low == high => 4 == 4 True

$x ==$ a[4] then return 4
(40 == 40)

So key element 40 is found at the index 4

The time complexity of Binary search can be expressed by the recurrence relation.

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

In each step, we can divide the original list into two sublists of size $n/2$.

we consider only one sublist to search the given key element. $1 \times T(n/2) = T(n/2)$.

In every step, we need to compare the key element with middle element.

Assume that this comparision time is 1 unit of time (constant time).

$\therefore T(n) = T(n/2)+1$

we replace n with $n/2$ repeatedly

$$= \left[T\left(\frac{\frac{n}{2}}{2}\right)+1\right]+1$$

$$= T\left(\frac{n}{2}\times\frac{1}{2}\right)+1]+1$$

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

$$= \left[T\left(\frac{n}{2^3}\right)+1\right]+2$$

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

$\vdots$

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

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

$$= 1+\log n.$$

$$O(1+\log n) = \underline{O(\log n)}$$

	Best case	Average case	Worst case
successful	O(1)	O(log n)	O(log n)
unsuccessful	O(log n)	O(log n)	O(log n)

The time complexities of binary search.

Quick sort

A is an array of elements and pivot is any one of the element in that array.
The low value is indicated with i.
The high value is indicated with j.
i will be incremented when A[i] ≤ pivot
j will be decremented when A[j] > pivot.
otherwise there is no change.
If i and j are stopped then swap A[i] and A[j].
Then continue the same process again.
If i and j cross each other then A[j] and pivot will be swapped.
This process will be repeated until all elements are sorted.

After swapping of the A[j] and pivot, the pivot element is placed at its right position in the array.
i.e., pivot element divides the array into two sublists. (left sublist and right sublist)
But these sublists elements are not in sorted order. we have to sort the elements in both sublists to arrange all elements in sorted order.

Example of Quick Sort

```
   50   30   10   90   80   20   40   70
p↑ i                                  ↑j

   50   30   10   90   80   20   40   70
p↑      i↑                       ↑j

   50   30   10   90   80   20   40   70
p↑           i                   j

   50   30   10   90   80   20   40   70
p↑                i↑  swap       ↑j
                      a[i]and a[j]

   50   30   10   40   80   20   90   70
p↑                i              j

   50   30   10   40   80   20   90   70
p↑                     i↑ swap ↑j
                       a[i]&a[j]

   50   30   10   40   20   80   90   70
p↑                     i    j

   50   30   10   40   20   80   90   70
p↑                          i
 ↑swap P and a[j]     ↑j

   20   30   10   40   50   80   90   70
p↑ i              j

   20   30   10   40   50   80   90   70
p↑      swap
        i↑ ↑j

   20   10   30   40   50   80   90   70
p↑      j    i
 ↑swap ↑

   10   20   30   40   50   80   90   70
```

```
10  20  30  40  50  80  90  70
                    p↑i      j

10  20  30  40  50  80  90  70
                    p↑   i   j
                         ↑swap↑

10  20  30  40  50  80  70  90
                    p↑   i   j

10  20  30  40  50  80  70  90
                    p↑   j   i
                     ↑swap↑

10  20  30  40  50  70  80  90
```

```
Algorithm Quicksort(l, h)
{
        if(l<h)
        {
            j := partition(A, l, h);
            Quicksort(l, j-1);
            Quicksort(j+1, h);
        }
}

Algorithm partition(A, l, h);
{
      pivot := A[l];
      i:= l; j:=h;
      while(i<j)
      {
        do
        { i++;
        } while(A[i] ≤ pivot);
        do
        {
          j--;
        } while( A[j]>pivot);
        if(i<j)
        swap(A[i],A[j]);
      }
      swap(A[l],A[j]);
      return j;
}
```

Time complexity for Best and Average cases

In best case, pivot element divides the entire list into two exact ~~halves~~ equal halves.

In an average case, pivot element divides the entire list into two nearly equal halves.

For both cases, the list can be divided into two parts of size $n/2$.

In the quick sort, we require n number of comparisions to sort n elements.

The time complexity can be expressed by the recurrence relation.

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

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

Substitute n with $\frac{n}{2}$ repeatedly.

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

$= 4T(\frac{n}{2^2}) + n + n$

$= 4T(\frac{n}{2^2}) + 2n$

$= 4\left[2\cancel{4}T(\frac{n}{2^3}) + \frac{n}{4}\right] + 2n$

$= 8T(\frac{n}{2^3}) + n + 2n$

$= 8T(\frac{n}{2^3}) + 3n$

$= 2^3 T(\frac{n}{2^3}) + 3n$

$\vdots$

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

Assume $\frac{n}{2^k} = 1$ Then $n = 2^k \Rightarrow k = \log_2 n$

$T(n) = n \cdot T(\frac{n}{n}) + \log n(n)$

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

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

So $O(n + n\log n) = O(n \log n)$

$\therefore$ Time complexity $T(n) = \underline{O(n \log n)}$

Time complexity for worst case

In the worst case, the pivot element is placed at either first or last position in the list. ex:- $\underset{\text{pivot}}{5}\ \underbrace{4\ 3\ 2\ 1}_{\xleftrightarrow{n-1}\ \text{elements}}$

So, the time required for sorting the elements is equal to time required to solve $n-1$ elements that are present in either left or right part plus the time required for comparing the n elements with pivot. one sublist is empty. Another sublist contains $n-1$ elements.

The time complexity can be expressed by the recurrence relation.

$$T(n) = \begin{cases} 1 & \text{if } n=0 \\ T(n-1)+n & \text{if } n>0 \end{cases}$$

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

Let us assume $n-k=0 \Rightarrow n=k$

$$T(n) = T(n-k]+(n-(k-1))+(n-(k-2))+ \cdots +(n-1)+n$$
$$= T(n-k)+(n-k+1)+(n-k+2)+\cdots +(n-1)+n.$$

Substitute $k=n$ in above equation.

we will get $T(n)=T(n-n)+(n-n+1)+(n-n+2)+\cdots +(n-1)+n.$

$$T(n)=T(0)+1+2+\cdots+(n-1)+n.$$
$$= 1+\left[\frac{n(n+1)}{2}\right] = 1+\left[\frac{n^2+n}{2}\right]$$
$$= \frac{2+n^2+n}{2} \cdot O\left(\frac{2+n^2+n}{2}\right)=O(n^2).$$

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

Merge sort

In the merge sort, the given array of elements are divided into two sublists by calculating the middle value. middle value = $\frac{(low+high)}{2}$.

The entire list ~~[illegible]~~ can be divided at its middle position.

This will create two sublists called left sublist and right sublist.

left sublist ranges from low to mid

right sublist ranges from mid+1 to high.

These two sublists are again divided into two small sublists repeatedly.

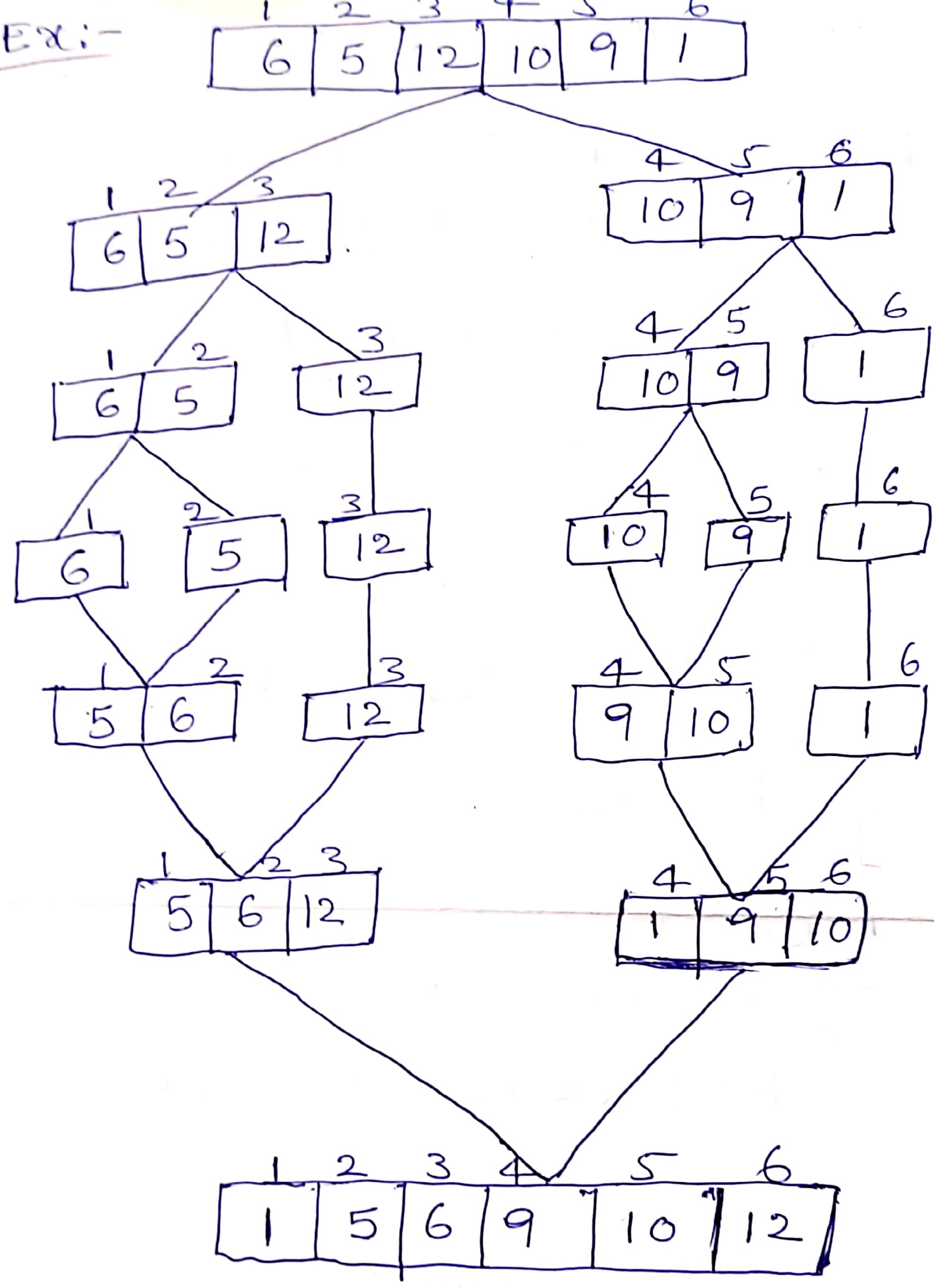
we have to repeat the same process until each sublist can contain only a single element.

Then each sublist is compared to its adjacent sublist and merged to sort the elements in both ~~[illegible]~~ sublists.

~~This process will be repeated until each~~ ~~sublist contains only a single element.~~

This process will be repeated until all elements are arranged in sorted order.

Ex:-



Merge Sort Process

```
Algorithm Merge Sort ( low, high)
{
    if( low < high) // List contain more
                       than one element
    {
        mid := ( low + high)/2 ; // Find where
                                    to split the list
        Merge Sort( low, mid);
        Merge Sort (mid+1, high);
        Merge( low, mid, high);
    }
}
```

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

P

Time complexity of Merge Sort

The time complexity of merge sort is same for all cases (Best, Average and worst cases).

In any case, we have to divide the given array into the two sublists of ~~[illegible]~~ equal size.

Here n number of comparisions are required to sort all n elements.

Time complexity can be expressed by the recurrence relation.

$$T(n) = \begin{cases} a & \text{if } n=1. \\ 2T(\frac{n}{2})+cn & \text{if } n>1 \end{cases}$$

a and c are constants.

$$T(n) = 2T(\frac{n}{2})+cn$$
$$= 2\left[2T(\frac{n}{4})+\frac{cn}{2}\right]+n$$
$$=4T(\frac{n}{4})+2cn$$
$$= 4\left[2T(\frac{n}{8})+\frac{cn}{4}\right]+2cn$$
$$= 8T(\frac{n}{8})+cn+2cn$$
$$=8T(\frac{n}{8})+3cn$$
$$=2^3T(\frac{n}{2^3})+3cn$$

~~[illegible]~~

$\vdots$

$$=2^kT(\frac{n}{2^k})+kcn$$

Assume $n=2^k \Rightarrow k=\log_2 n$

$$T(n)=n\cdot T(\frac{n}{n}) + \log n\,(c\cdot n)$$
$$= n\cdot T(1)+cn\log n$$
$$=a\cdot n + cn\log n$$
$$T(n) = O(a\cdot n+cn\log n) = O(n\log n)$$

GREEDY METHOD

// The general Method:-

Suppose there are n ~~p~~ inputs and we want to obtain a subset that satisfies some constraints

Any subset that satisfies these constraints is called a feasible solution.

We need to find a feasible solution that either maximizes or minimizes the objective function A feasible solution that performs this task is called an optimal solution

In greedy method, we can design an algorithm that works in different stages.
At each stage we can take only one input.
In every stage, we take decision about whether a particular input is an optimal solution
This can be done by taking the inputs in some specific order.
If the selected input gives the feasible solution then this input is added to the solution otherwise this input is not added to the solution.
This selection procedure is based on the some optimization measure or objective function.

```
Algorithm Greedy(a, n)
// a[1:n] Contains the n inputs
{
    Solution := ϕ; // Initialize the Solution.
    for i := 1 to n do
    {
        x := Select(a);
        if Feasible(Solution, x) then
            Solution := Union(Solution, x);
    }
    return Solution;
}
```

Control abstraction for Greedy method.

The function Select selects an input from a[] and removes it.
The selected input's value is assigned to x.
Feasible is a Boolean valued function ~~that~~ that determines whether x can be included to the Solution vector.

The function Union combines x with the Solution and updates the objective function.

~~Greedy algorithm~~

//

Knapsack problem

There are n number of objects and a Knapsack (bag).

Each object i has a weight w_i.

Knapsack capacity is m.

If a fraction x_i ($0 \leq x_i \leq 1$) of object i is placed into the Knapsack then a profit of $p_i x_i$ is obtained.

The objective is to fill the Knapsack so that it maximizes the total profit obtained.

The total weight of all selected objects is atmost m.

So the Knapsack problem can be stated as follows.

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

$$\text{Subject to} \sum_{1\leq i\leq n} w_i x_i \leq m$$

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

The profits (p_i's) and weights (w_i's) are positive numbers.

A feasible solution is any set $(x_1, x_2, \dots x_n)$ satisfying $\sum_{1\leq i\leq n} w_i x_i \leq m$

An optimal solution is a feasible solution that maximizes $\sum_{1\leq i\leq n} p_i x_i$.

Ex:- Find the optimal solution for $n=3$ and $m=20$, profits $P_1, P_2, P_3 = 25, 24, 15$, and weights $W_1, W_2, W_3 = 18, 15, 10$

Given that number of objects $n=3$

For each item, profits are $P_1=25, P_2=24, P_3=1$

For each item, weights are $W_1=18, W_2=15,$

consider the $\frac{\text{Profit}}{\text{Weight}}$ ratio $\frac{P_i}{W_i}$ for $1 \le i \le n$.

$\frac{P_1}{W_1} = \frac{25}{18} = 1.38$ $\frac{P_2}{W_2} = \frac{24}{15} = 1.6$ $\frac{P_3}{W_3} = \frac{15}{10} = 1.5$

objects O_i	1	2	3
Profits P_i	25	24	15
Weights W_i	18	15	10
$\frac{\text{Profit}}{\text{Weights}}\left(\frac{P_i}{W_i}\right)$	1.38	1.6	1.5

Now the solution is x_1, x_2, x_3 where x_i is inbetween 0 and 1 i.e., $0 \le x_i \le 1$.
The bag capacity $m = 20$ ⟸
According to $\frac{\text{profit}}{\text{weight}}$ ratio method consider the maximum element first.
Now add 15 kg to the knapsack.
So $x_2 = 1$, the remaining bag capacity is $m - w_2 = 20 - 15 = 5$
Now add object 3 to the knapsack.
but $w_3 = 10$ kg. only remaining capacity of the knapsack is 5 kg ~~on~~
we can only add fraction of w_3 to the knapsack. This fraction can be obtained as the ratio between knapsack remaining capacity and the object weight.
It is $\frac{5}{10} = 1/2$. So $x_3 = 1/2$ Add 5 kg to knapsack.
Next we can add object 1 according to profit/weight ratio.
But bag already contains 20 kg.
So we cannot add object 1 to the knapsack
$\therefore x_1 = 0$.

Scanned with OKEN Scanner

$\sum p_i x_i = p_1 \times x_1 + p_2 \times x_2 + p_3 \times x_3$

$= 25 \times 0 + 24 \times 1 + 15 \times 0.5$

$= 0 + 24 + 7.5 = 31.5$

So the optimal solution is (x_1, x_2, x_3) $= (0, 1, 1/2)$ and the maximum profit is 31.5.

$\sum w_i x_i \le m$.

$w_1 \times x_1 + w_2 \times x_2 + w_3 \times x_3$ $= 0 + 15 + 5 \le 20$

$= 18 \times 0 + 15 \times 1 + 10(1/2)$

$\Rightarrow 20 \le 20$. Here $x_1 = 0$, $x_2 = 1$ and $x_3 = 1/2$.

$0 \le x_i \le 1$

If we don't follow the profit/weight ratio then we can obtain multiple number of feasible solutions. The optimal solution is one of the feasible solution with maximum profit.

Following are different feasible solutions $(x_1\ x_2\ x_3)$	$\sum w_i x_i$	$\sum p_i x_i$
1. $(1/2, 1/3, 1/4)$	16.5	24.25
2. $(1, 2/15, 0)$	20	31
3. $(0, 2/3, 1)$	20	31
4. $(0, 1, 1/2)$	20	31.5

out of these four feasible solutions the optimal solution is $(0, 1, 1/2)$ because it has the maximum profit 31.5. If the objects are sorted according to their profit/weight ratio then maximum profit will be achived per ~~a~~ ~~[illegible]~~ each unit of capacity used.

```
Algorithm GreedyKnapsack(m,n)
// P[1:n] and w[1:n] contain the profit and weights
// of the n objects ordered such that
// P[i]/w[i] ≥ P[i+1]/w[i+1].
// m is the knapsack size and x[1:n] is the
// solution vector.
{ for i:=1 to n do x[i]:=0; //Initialize x
    U:=m;
    for i:=1 to n do
    {
        if(w[i]>U) then break;
        x[i]:=1; U:=U-w[i];
    }
    if (i≤n) then x[i]:=U/w[i];
}
```

Algorithm for greedy knapsack problem.

The time complexity of the greedy knapsac
is O(n).

Ex:2 Find out the optimal solution for the knapsa
capacity $m=15$ and $n=7$
profits: 10, 5, 15, 7, 6, 18, 3
weights: 2, 3, 5, 7, 1, 4, 1

Given that number of objects $n=7$
$P_1=10$ $P_2=5$ $P_3=15$ $P_4=7$ $P_5=6$, $P_6=18$ $P_7=3$
$W_1=2$ $W_2=3$ $W_3=5$ $W_4=7$ $W_5=1$ $W_6=4$ $W_7=$

consider profit by weight ratio method, P_i/W

$\frac{P_1}{W_1}=\frac{10}{2}=5$ $\frac{P_2}{W_2}=\frac{5}{3}=1.6$ $\frac{P_3}{W_3}=\frac{15}{5}=3$ $\frac{P_4}{W_4}=\frac{7}{7}=1$ $\frac{P_5}{W_5}=\frac{6}{1}=6$

$\frac{P_6}{W_6}=\frac{18}{4}=4.5$ $\frac{P_7}{W_7}=\frac{3}{1}=3$

objects O_i	1	2	3	4	5	6	7
Profits P_i	10	5	15	7	6	18	3
Weights W_i	2	3	5	7	1	4	1
profits P_i / weights W_i	5	1.6	3	1	6	4.5	3

ch x_i value is in between 0 and 1
i.e., $0 \leq x_i \leq 1$
The bag capacity $m = 15$
According to $\frac{\text{profit}}{\text{weight}}$ ratio method, we consider maximum element first.
Now add 1 kg to the knapsack.
So $x_5 = 1$, the remaining bag capacity is
$15 - 1 = 14$
Now add 2 kgs to the knapsack
So $x_1 = 1$, the remaining bag capacity is $14 - 2 = 12$
Now add object 6 i.e. 4 kgs to knapsack
So $x_6 = 1$, the remaining bag capacity $m = 12 - 4 = 8$.
Now add object 3 i.e., 5 kgs to the knapsack
So $x_3 = 1$, the remaining capacity $m = 8 - 5 = 3$
Now add object 7 i.e., 1 kg to the knapsack
So $x_7 = 1$, the remaining capacity $m = 3 - 1 = 2$
Now add object 2 i.e., 3 kg to knapsack.
But knapsack remaining capacity is only 2 kgs
Here we have to calculate ratio between knapsack remaining capacity and object weight
$= 2/3$. So $x_2 = 2/3$.
We can not add object 4 because knapsack already contains 15 kgs. (its maximum capacity)
So $x_4 = 0$

$$\sum_{1 \leq i \leq n} w_i x_i = w_1 x_1 + w_2 x_2 + w_3 x_3 + w_4 x_4 + w_5 x_5 + w_6 x_6 + w_7 x_7.$$

$$= 2 \times 1 + 3 \times \frac{2}{3} + 5 \times 1 + 7 \times 0 + 1 \times 1 + 4 \times 1 + 1 \times 1$$

$$= 2 + 2 + 5 + 0 + 1 + 4 + 1 = 15$$

$\therefore \sum w_i x_i \leq m. \Rightarrow 15 \leq 15$

$$\sum_{1\leq i\leq n} p_i x_i = P_1x_1 + P_2x_2 + P_3x_3 + P_4x_4 + P_5x_5 + P_6x_6 + P_7x_7$$

$$= 10\times1 + 5\times\frac{2}{3} + 15\times1 + 7\times0 + 6\times1 + 18\times1 + 3\times1$$

$$= 10 + 3.33 + 15 + 0 + 6 + 18 + 3$$

$$= 55.3$$

So the optimal solution is $(x_1, x_2, x_3, x_4, x_5, x_6, x_7)$ $= (1, 2/3, 1, 0, 1, 1, 1)$ and the maximum profit is 55.3. //

// Minimum cost Spanning Tree

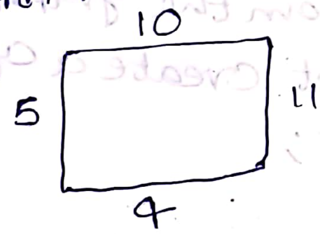
spanning tree :- A spanning tree of a graph G=(V,E) is a subgraph of G that contains all the vertices ~~of~~ contain no circuit.

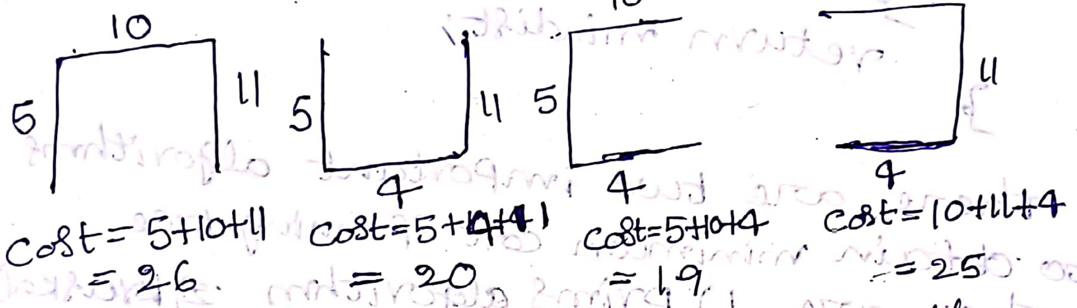
An edge of a spanning tree is called a branch

An edge in the graph that is not in the spanning tree is called a chord.

Spanning trees are represented by using two graph searching algorithms. They are

1. Breadth First Search (BFS)
2. Depth First Search (DFS)





Four Possible Spanning tree with different costs

Minimum Spanning tree

It is one of the possible spanning tree with minimum cost.

In the above example, the minimum spanning tree is the tree that has cost 19. which is less than all other ~~sp~~ costs of other three spanning trees.

```
Algorithm MST(G, W, T).
{
    T[]:=0;
    Mindist :=0;
    put the n nodes in T and no edges
    While T has less than n-1 edges do
    {
        choose a remaining edge e of
    minimum weight;
        Assign the minimum weight to mindist;
        Delete e from the graph.
        if(e does not create a cycle in T) then
        Add e to T;
    }
    return mindist;
}
```

There are two important algorithms to obtain minimum cost spanning tree.
They are 1) Prims algorithm 2) Kruskals algorithm.

Prims algorithm

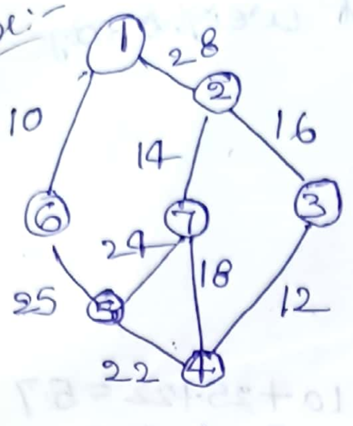
In this method, first select the least cost weighted edge.

Next, we select the unvisited least cost ~~edge~~ adjacent edge.

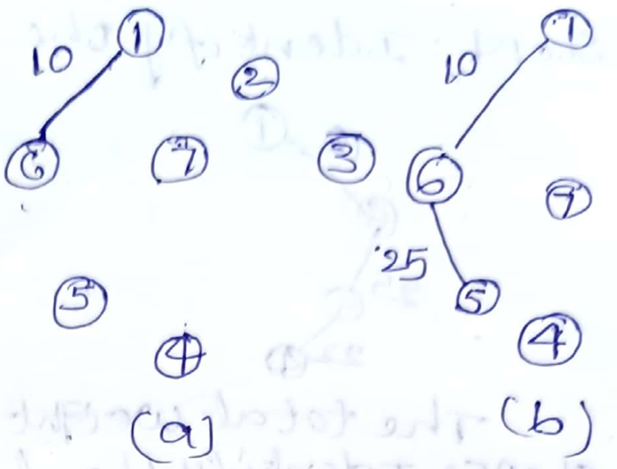
In every step, we need to select the adjacent unvisited least cost edge for nodes that are already selected in previous step.

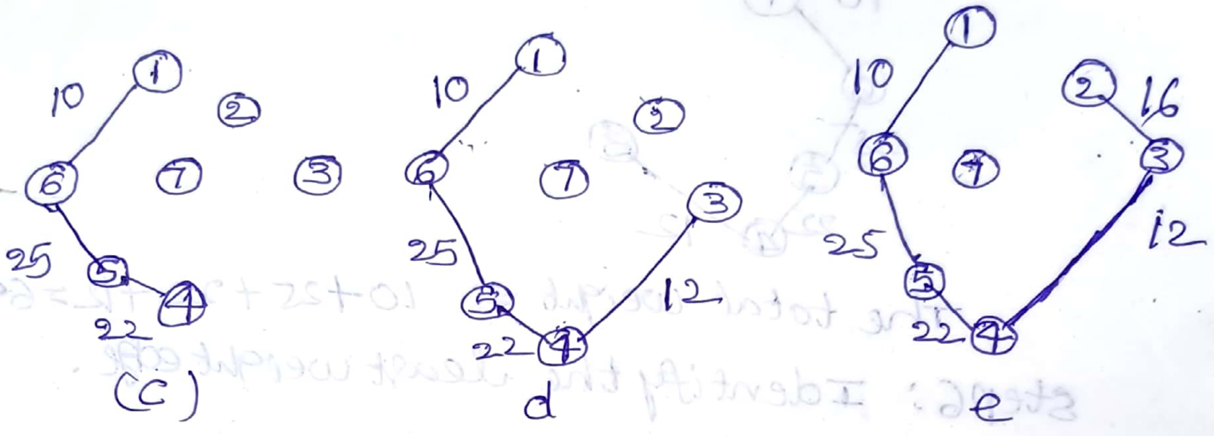
This process will be repeated until the all vertices are included and has no circuit,

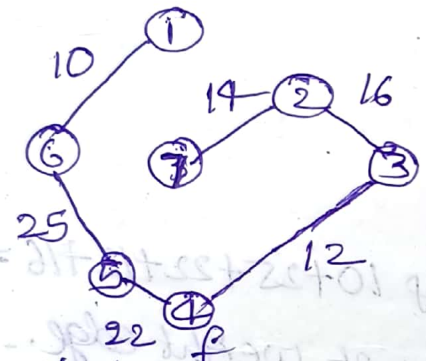
Ex:-



Graph G



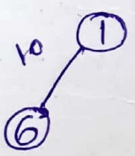




Minimum spanning tree

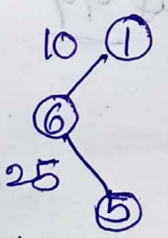
Step1: Initially the total weight or cost is 0.

Step2: Identify the least weight edge

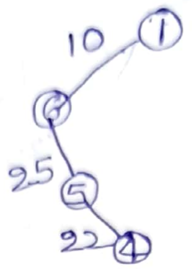


Now the total weight is 0+10 = 10.

Step3: Identify the least weight edge

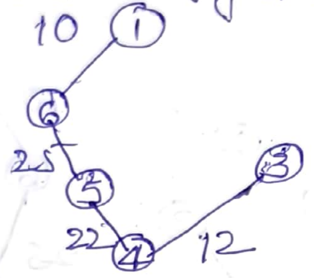


Step4: Identify the least weight edge



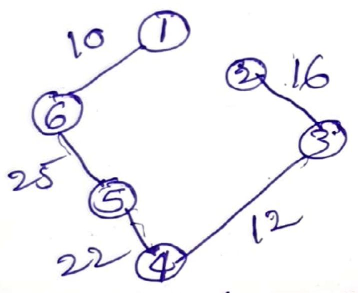
The total weight is 10+25+22 = 57

Step5: Identify the least weight edge



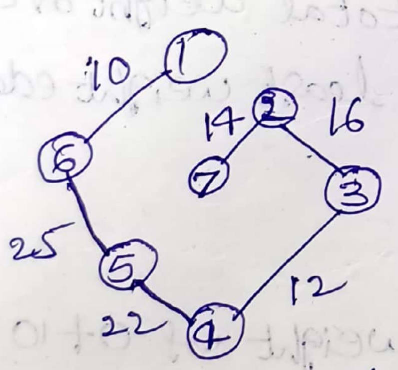
The total weight is 10+25+22+12=69

Step6: Identify the least weight edge.



The total weight is 10+25+22+12+16 = 85

Step 7: Identify the least weight edge.



The total weight is 10+25+22+12+16+14
= ~~85~~ 99

So the minimum cost is 99

```
Algorithm Prim(E, cost, n, t)
// E is the set of edges in G. cost[1:n, 1:n] is
// the cost adjacency matrix of an n vertex graph
// such that cost[i, j] is either a positive real
// number or ∞ if no edge (i, j) exists.
// A minimum spanning tree is computed and stored
// as a set of edges in the array t[1:n-1, 1:2].
// t[i, 1], t[i, 2] is an edge in the minimum-cost
// spanning tree. The final cost is returned.
{
  Let (k, l) be an edge of minimum cost in E;
  mincost := cost[k, l];
  t[1,1] := k;   t[1,2] := l;
  for i:=1 to n do // Initialize near.
  if (cost[i, l] < cost[i, k]) then near[i] := l;
  else near[i] := k;
  near[k] := near[l] := 0;
  for i := 2 to n-1 do
  {
     // Find n-2 additional edges for t.
     Let j be an index such that near[j] ≠ 0
     and cost[j, near[j]] is minimum;
     t[i,1] := j; t[i,2] := near[j];
     mincost := mincost + cost[j, near[j]];
     near[j] := 0;
     for k := 1 to n do // update near[ ];
      if ((near[k] ≠ 0) and cost[k, near[k]]
                         > cost(k, j]))
       then near[k] := j;
  }
  return mincost;
}
```

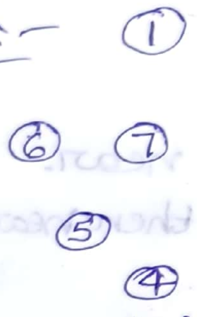
Prim's minimum-cost spanning tree algorithm

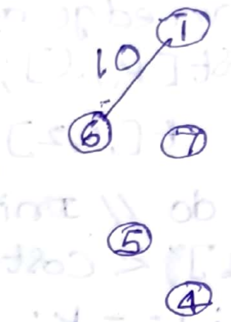
Time complexity of Prim's algorithm is $O(n^2)$.

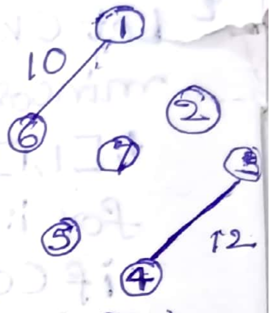
Kruskal's Algorithm

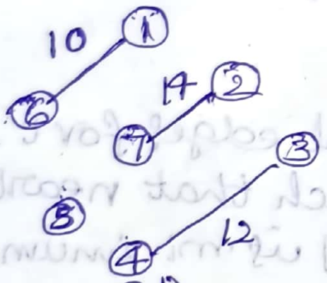
In this method, the edges are arranged in nondecreasing order of their cost.
First, we have select minimum cost edge and inserted in the minimum cost spanning tree.
Next, we need to select minimum cost edge from the remaining edges in the graph.
This process will be repeated until all the vertices are included and has no cycle.

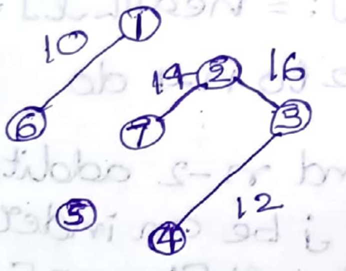
Ex:-

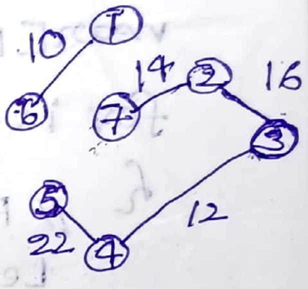

(a)

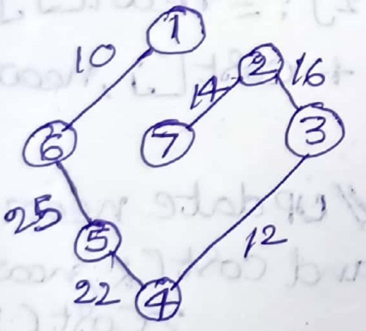

(b)


(c)


(d)


(e)


(f)


minimum cost Spanning tree.
(g)

minmCost = $10+12+14+16+22+25 =$ ~~[illegible]~~ 99.

In this algorithm, we begin with no edges elected. Figure (a) show the current graph with no edges selected.
Next, we consider (1,6) edge that has minimum cost. It is included in the spanning tree

Later we select the edge(3,4) and it is included in the tree figure (c).

The next edge is to be selected is (2,7)
If we include this edge then it does not form a cycle.
so it is included in the tree. (figure d).
Next, edge(2,3) is selected next and included in the tree. This is shown in figure (e).
After that we select (7,4)
If we include this edge then cycle will be formed in the tree. This edge is not selected. and it is discarded.
Next edge is (5,4) is included in the tree
It is shown in the figure(f).
The next edge to be considered is (7,5).
It is discarded. because its inclusion forms a cycle in the graph.
Finally, we select an edge (6,5) and it is included in the spanning tree.
This is a minimum cost spanning tree.
with cost 99

```
Algorithm Kruskal(E, cost, n, t)
// E is the set of edges in G. G has n vertices
// cost [u,v] is the cost of edge (u,v). t is the
// set of edges in the minimum-cost spanning tree
// The final cost is returned.
{
    Construct a heap out of the edge costs using Heapify;
    for i:=1 to n do parent[i]:=-1;
    // Each vertex is in a different set.
    i:=0; mincost:=0;
    while ((i<n-1) and (heap not empty)) do
    {
        Delete a minimum cost edge (u,v) from the heap
        and reheapify using Adjust;
j := Find(u); k := Find(v);
if (j ≠ k) then
{
    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 spanning tree");
else return mincost;
}
```

Kruskals Algorithm:

The time complexity of Kruskals algorithm is $O(E \log E)$ where E is the set of edges present in the graph G.
//

optimal Merge patterns

Suppose we have two sorted files containing m and n records respectively.

We need to merge these two files into one sorted file in time $O(m+n)$.

When more than two sorted files are to be merged together, the merge can be performed by repeatedly merging sorted files in pairs.

If we want to merge four files x_1, x_2, x_3 and x_4 then it can be achived in the following way

First merge x_1 and x_2 to get sorted file y_1

Then we merge y_1 and x_3 to get y_2.

Finally merge y_2 and x_4 to get the desired sorted file.

Here there are many combinations to obtain the sorted file.

Different pairs will give various computation time

We have to find ~~out~~ optimal way to merge various files inorder to obtain the required sorting file

Ex:- The files x_1, x_2 and x_3 are three sorted files of length 30, 20, and 10 records.

Merging x_1 and x_2 requires 50 records moves or comparisions. Merging the result with x_3 requires 60 moves.

So the total number of records required to merge the three files is 110.

Alternatively, we first merge x_2 and x_3 (30 moves) and then x_1 (60 moves).

So the total records required is only 90.

$\therefore$ The second merge pattern is faster than first

A greedy solution for this problem is as follows.

1. All the files are arranged in increasing order their number of records.
2. Then merge two smallest size files together. and place merged file at its correct position in a list.
3. Repeat step 2 until all the files will be merged to get a single sorted file.

This process will generate a binary merge tree

The process of merging two files is called two way merge pattern.

The two way merge pattern can be represented by binary merge trees.

In this tree leaf nodes are drawn as ~~[illegible]~~ squares and represent the files.

These nodes are called as external nodes.

The remaining nodes are drawn as circles.

these are called internal nodes.

The internal nodes has exactly two children,

It represents the file by obtained by merging the files represented by its two children.

The number in each node is the number of records in the file that is represented by that node.

If we don't use greedy approach, then it is very difficult to find the optimal order in which files can be merged.

To merge three files, we require $3C_2 = \frac{3!}{(3-2)!\,2!}$

$= \frac{3!}{1!\,2!} = 3$

To merge four files we require $4C_2 = \frac{4!}{2!\,2!} = 6$

To merge 10 files we require $10C_2$ = This is very difficult.

To avoid this difficulty we can use greedy approach.

Ex:- Let files $x_1, x_2, x_3, x_4, .., x_5$ are to be merged with 20, 30, 10, 5, 30 records respectively. Then apply greedy method to find the optimal merge pattern.

Files	x_1	x_2	x_3	x_4	x_5
records	20	30	10	5	30
	$\boxed{20}$	$\boxed{30}$	$\boxed{10}$	$\boxed{5}$	$\boxed{30}$

Arrange all files in sorted order based on the number of records.

5 10 20 30 30

15

15 20 30 30

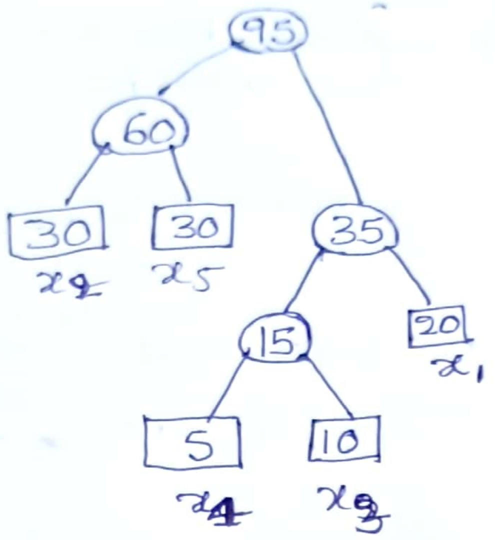
35

30 30 35

60

35 60

95



Total number of merges = Sum of internal nod
weights (file lengths) = 15 + 35 + 60 + 95 = 205

If d_i is the distance from the root to the external node x_i and q_i is the length of x_i [illegible] then the total number of merge operations

$$= \sum_{i=1}^{n} d_i q_i$$

This sum is called the weighted extern path length of the tree.

$$\sum_{i=1}^{5} d_i q_i = (2\times 20)+(2\times 30)+(3\times 10)+(3\times 5)+(2$$

$$= 205$$

```
treenode = record {
    treenode * lchild;
    treenode * rchild;
    integer weight;
};

Algorithm Tree(n)
// list is a global list of n single node
// binary trees as described above.
{
    for i := 1 to n-1 do
    {
        pt := new treenode; // Get a new treenode
        ~~pt~~ (pt → lchild) := Least(list); // Merge
                                            // two trees
        (pt → rchild) := Least(list); // with
                                      // small leng
        (pt → weight) := ((pt → lchild) → weight)
                          + ((pt → rchild) → weigh

        Insert(list, pt);
    }
    return Least(list); // Tree left in list
                        // is the merge tree
}
```

Algorithm to generate a two way merge tree.

In the above algorithm, the for loop is repeated n times nearly.
To insert the n nodes in a list, we need n units of time. So totally time complexi

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

Suppose we maintain a min heap to represent the n nodes: we require log n ti

So totally time complexity $T(P) = \underline{O(n \log}$

Single source shortest path problem.

In this greedy technique, we need to find out the shortest path from single source to many destination nodes.

This can be done by using Dijkstra algorithm.

Dijkstra algorithm is used for finding the shortest path from source node to all other nodes in a graph. This can be applied for both undirected and directed graphs

Shortest path can be detected by relaxing all the nodes except the source node.

The relaxing process can be done repeatedly until each node containing the shortest path from source node.

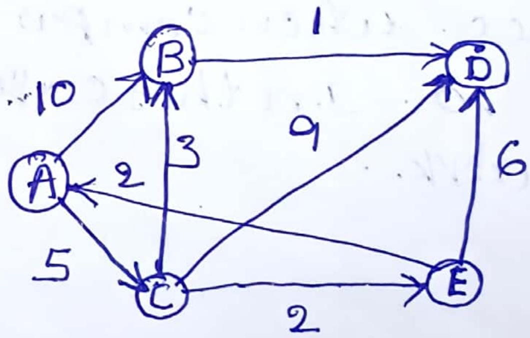
The rel~~ax~~ation formula is

if ($d(u) + c(u,v) < d(v)$) then

$d(v) := d(u) + c(u,v)$;

Suppose $d(u) + c(u,v) \geq d(v)$ then there is no change in $d(v)$ value.

Consider that the source vertex is A.



	A	B	C	D	E
A	$\boxed{0}$	∞	∞	∞	∞
C		10	$\boxed{5}$	∞	∞
E		8		14	$\boxed{7}$
B		$\boxed{8}$		13	
D				$\boxed{9}$	

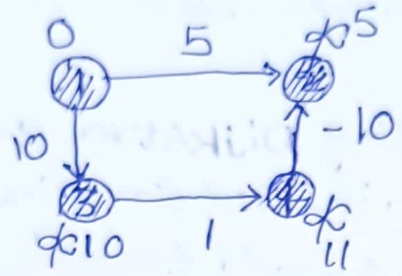
Shortest path from A to B is ACE with distance 8.

Shortest path from A to C is AC with distance 5

Shortest path from A to D is ACB with distance 9

Shortest path from A to E is AC with distance 7.

The dijkstra algorithm has one drawback. i.e., It may not work for negative weighted edges.



After visiting all nodes, we will get the shortest path from source node ~~0~~1

The shortest distance from ~~0~~1 to 2 is 5 after visiting of all vertices.

If we consider the negative weighted edge from 4 to 2 then the shortest path from source node ~~0~~1 to node 2 will become 1 [∵ 11 − 10 = 1]

So the dijkstra algorithm can not work for this case.

When we change this negative edge (4 to 2) to −1, it will work. [∵ 11 − 1 = 10].

The node 2 already has distance 5. It is less distance when compare with distance from 4 to 2 i.e., 10. In this case dijkstra alorithm will work.

```
Algorithm Shortestpaths(v, cost, dist, n)
// dist[j], 1 ≤ j ≤ n, is set to the length of the
// shortest path from vertex v to vertex j in
// a digraph G with n vertices. dist[v] is set to zero
// G is represented by its cost adjacency matrix
// cost[1:n, 1:n].
{
    for i := 1 to n do
    { // Initialize S.
        S[i] := false; dist[i] := cost[v, i];
    }
    S[v] := true; dist[v] := 0; // put v in S
    for num := 2 to n do
    {
        // Determine n-1 paths from v.
        Choose u from among those vertices
        not in S such that dist[u] is minimum;
        S[u] := true; // Put u in S.
        for (each w adjacent to u with
                        S[w] = false) do
        // Update distances.
         if ( dist[w] > dist [u] + cost [u, w] th
          dist [w] := dist [u] + cost [u, w];
    }
}
```

Greedy algorithm to generate shortest path

The time complexity $T(n) = O(n^2)$ if we use adjacency matrix representation.

The time complexity $T(n) = O(n+e) \log n$ if we use adjacency list representation.