UNIT-II

DIVIDE-AND-CONQUER

GENERAL METHOD :-

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

1) **Divide:**
A problem is divided into a number of subproblems (which are smaller instances of the given problem) of the same type, ideally of about equal size.
2) **Conquer:**
The subproblems are solved (typically recursively). However, if the subproblems are small enough, they are solved in a straightforward manner.
3) **Combine:**
If necessary, the solutions to the subproblems are combined to get a solution to the original problem.

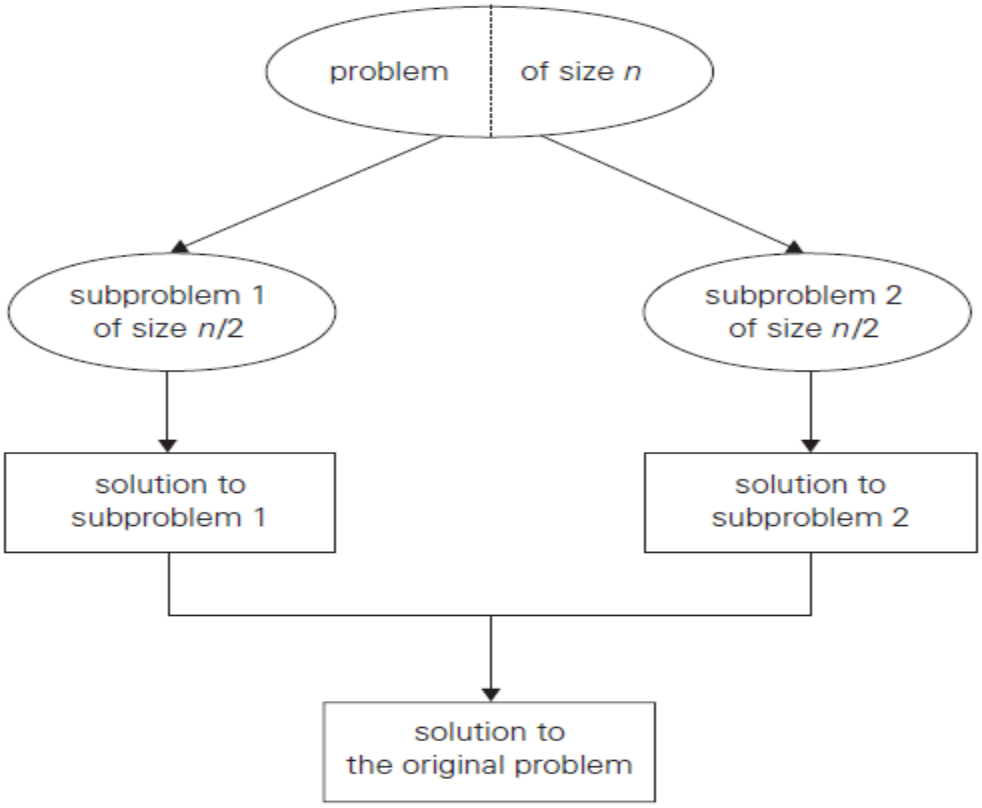


Figure: Divide-and-conquer technique (typical case)

Control abstraction for divide and conquer:-

```
1. Algorithm DandC(P)
2. {
3.    if Small(P) then
4.        return S(P);
5.    else
6.    {
7.        Divide P into smaller instances P1, P2, ..., Pk, where k ≥1;
8.        Apply DandC to each of these sub-problems;
9.        return Combine(DandC(P1), DandC(P2), ..., DandC(Pk));
10.   }
11.}
```

→ Small(P) is a Boolean-valued function that determines whether the input size is small enough to compute without splitting.
If this is so, the function S(P) is invoked. Otherwise the problem P is divided into smaller sub-problems.

→ Combine( ) is a function that determines the solution to P using the solutions to the k sub-problems P1, P2, …, Pk.

Computing the time complexity of DandC:-

→ If the size of the problem P is *n* and the sizes of the *k* sub-problems are *n1*, *n2*, ..,.*nk* respectively, then the computing time of DandC is described by the recurrence relation:

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

where,

- *T(n)* is the time for DandC on any input of size *n*
- *g(n)* is the time to compute the answer directly for small inputs
- *f(n)* is the time for dividing P into sub-problems and combining the solutions to sub-problems.

→ The time 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 and c are constants.

We assume that *n* is a power of b (i.e., $n=b^k$).

Eg :-1. Solve the following recurrence relation:

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

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

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

$= 4[2T(n/8) + n/4] + 2n = 8T(n/8) + 3n = 2^3T(n/2^3) + 3n$

After applying this substitution *k* times,

$T(n) = 2^kT(n/2^k) + kn$

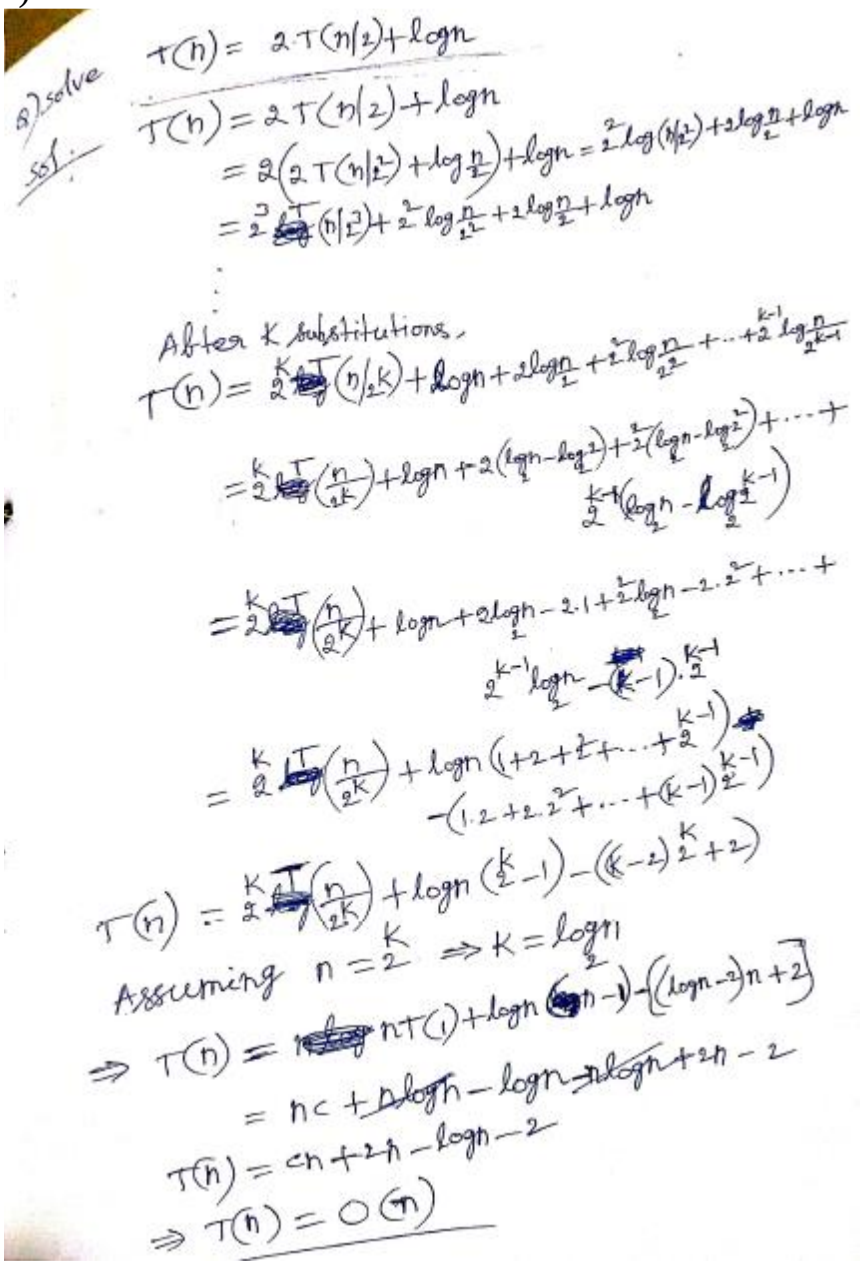
To terminate this substitution process, we switch to the closed form $T(1)=1$, which happens when $2^k = n$ which implies $k=log_2\ n$

So, $T(n) = nT(1) + nlog_2\ n$

$T(n) = n + nlog\ n$

$= O(nlogn)$

2)



Note:

(1) $1+2+3+\ldots+n = \frac{n(n+1)}{2}$

(2) $1+r+r^2+\ \ldots\ +r^k = \frac{r^k-1}{r-1}$

(3) $1+\frac{1}{r}+\frac{1}{r^2}+\ldots+\frac{1}{r^{k-1}} = \frac{1-r^k}{1-r}$

(4) $1.2+2.2^2+3.2^3+\ldots+k.2^k = (k-1)2^{k+1}+2$

(5) $1^2+2^2+3^2+\ldots+n^2 = \frac{n(n+1)(2n+1)}{6}$

(6) $1^3+2^3+3^3+\ldots+n^3 = \frac{n^2(n+1)^2}{4}$

Master Theorem for solving recurrence relations of divide–and-conquer: -

Let $T(n) = aT(n/b)+f(n)$, then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a-\epsilon})$, for some constant $\epsilon>0$ *(i.e.,* if $f(n) < n^{\log_b a})$, then $T(n)=\theta(n^{\log_b a})$.

2. (a) If $f(n) = \theta(n^{\log_b a})$, then $T(n)=\theta(n^{\log_b a} \lg n)$.
   (b) If $f(n) = \theta(n^{\log_b a} lg^k n)$, where k ≥ 0, then $T(n) = \theta(n^{\log_b a} lg^{k+1} n)$.

3. If $f(n) = \Omega(n^{\log_b a+\epsilon})$ for some constant $\epsilon > 0$ *(i.e.,* if $f(n) > n^{\log_b a}$ *)*, and if $af(n/b) \le cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n)= \theta(f(n))$.

Examples:

(1).

$T(n) = 9T(n/3) + n$.

For this recurrence, we have $a = 9$, $b = 3$, $f(n) = n$, and thus we have that $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$. Since $f(n) = O(n^{\log_3 9-\epsilon})$, where $\epsilon = 1$, we can apply case 1 of the master theorem and conclude that the solution is $T(n) = \Theta(n^2)$.

(2).

$T(n) = T(2n/3) + 1$,

in which $a = 1$, $b = 3/2$, $f(n) = 1$, and $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$. Case 2a applies, since $f(n) = \Theta(n^{\log_b a}) = \Theta(1)$, and thus the solution to the recurrence is $T(n) = \Theta(\lg n)$.

(3)

$T(n) = 3T(n/4) + n \lg n$,

we have $a = 3$, $b = 4$, $f(n) = n \lg n$, and $n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$. Since $f(n) = \Omega(n^{\log_4 3+\epsilon})$, where $\epsilon \approx 0.2$, case 3 applies if we can show that the regularity condition holds for $f(n)$. For sufficiently large n, we have that $af(n/b) = 3(n/4)\lg(n/4) \le (3/4)n \lg n = cf(n)$ for $c = 3/4$. Consequently, by case 3, the solution to the recurrence is $T(n) = \Theta(n \lg n)$.

(4)

$T(n) = 2T(n/2) + n \lg n$

even though it appears to have the proper form: $a = 2$, $b = 2$, $f(n) = n \lg n$, and $n^{\log_b a} = n$. You might mistakenly think that case 3 should apply, since $f(n) = n \lg n$ is asymptotically larger than $n^{\log_b a} = n$. The problem is that it is not *polynomially* larger. The ratio $f(n)/n^{\log_b a} = (n \lg n)/n = \lg n$ is asymptotically less than n^{ϵ} for any positive constant ϵ.

Apply case 2b of master theorem to solve the above problem.

Binary search: -

→ Let a[1:n] be a list of elements that are sorted in ascending order.
→ We have to determine whether a given element x is present in the list or not.
→ If x is present, we have to return its position , else return 0.

Iterative algorithm: -

```
Algorithm BinSearch(a,n,x)
// Given an array a[l:n] of elements in nondecreasing
// 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;
}
```

Recursive algorithm: -

```
Algorithm RBinSearch(a,low,high,x)
{
// Given an array a[low:high] of elements in nondecreasing
// order, 1<low<high, determine whether x is present, and
//if so, return j such that x =a[j]; else return0.
    if (low>high) then return 0;
    else
    {
        // Reduce the problem into a smaller subproblem.
        mid:= ⌊(low + high)/2⌋;
        if (x = a[mid]) then
            return mid;
        else if (x <a[mid]) then
            return RBinSearch(a,low,mid-1,x);
        else
            return RBinSearch(a,mid+1,high,x);
    }
}
```

→ This algorithm is initially invoked as RBinsearch(a,1,n,x).

Time Complexity:

→ The main problem is divided into one sub-problem in constant time.

→ The answer to the new sub-problem is also the answer to the original problem. So, there is no need for combining.

→ The running-time of this algorithm can be characterized as follows:

$$T(n) = \begin{cases} c, & if\ n = 1 \\ T(n/2) + c, & otherwise \end{cases}$$

Solving by using substitution method,

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

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

$= [T(n/8) + c] + 2c = T(n/8) + 3c = T(n/2^3) + 3c$

After applying this substitution k times,

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

To terminate this substitution process, we switch to the closed form $T(1)=c$, which happens when $2^k = n$ which implies $k=\log_2 n$

So, $T(n) = T(1) + c\log_2 n$

$T(n) = c + c\log_2 n$

$= O(\log n)$

Merge sort: -

→ Given a sequence of n elements a[1:n], we divide the given set of elements into two subsets a[1:n/2] and a[n/2+1:n].

→ Each subset is individually sorted, and the resulting sorted sequences are merged to produce a single sorted sequence of n elements.

Recursive algorithm for merge sort:

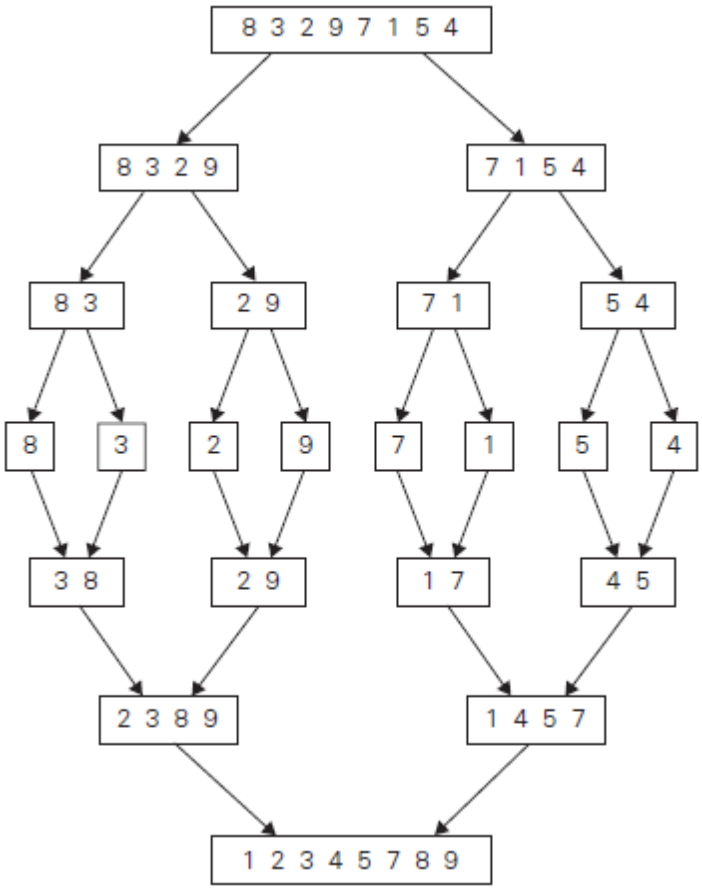
```
Algorithm MergeSort(low,high)
{
    // a[low : high] is a global array to be sorted.
    if (low<high)  then  //If there are more than one element
    {
        mid:= ⌊(low + high)/2⌋;
        MergeSort(low,mid);    // Solve sub-problem1
        MergeSort(mid+1,high);  //Solve sub-problem2
        Merge(low,mid,high);  //Combine the solutions of sub-problems
    }
}
Algorithm Merge(low,mid,high)
{
// a[low :high] is a global array containing two sorted subsets in a[low :mid] and in
// a[mid+1:high]. The goal is to merge these two sets into a single set residing in
//a[low :high].
// b[ ] is an auxiliary(temporary) global array.
      i:=low;
      h: =low;   // h is the indexing variable for the first part of the array a[low :high].
      j: =mid+1;  // j is the indexing variable for the second part of the array a[low :high].
      while ((h≤mid) and (j≤high)) do
      {
          if (a[h]≤a[j]) then
          {
              b[i] :=a[h];  // copy the elements of a into b.
              h:= h+1;
          }
          else
          {
              b[i]:= a[j];  // copy the elements of a into b.

              j:= j+1;
          }
```

```
            i:=i+1;
        }//end of while
        if (h>mid) then
        {
            for k := j to high do
            {
                b[i]:=a[k];  // copy the remaining elements of second part of a into b.
                i:=i+1;
            }
        }
        else
        {
             for k:=h to mid do
             {
                b[i] := a[k]; // copy the remaining elements of first part of a into b.
                i:=i+1;
            }
        }
        for k: = low to high do
            a[k]:=b[k];
    } // end of algorithm.
```

→ MergeSort algorithm is initially invoked as MergeSort(1,n).

Example: Trace the MergeSort on the data: 8,3,2,9,7,1,5,4.



Time Complexity:

→ The merge-sort problem of size *n* is divided into two sub-problems of size *n/2* each.
This division is done in constant time by the statement *mid:*= $\lfloor (low + high)/2 \rfloor$;

→ The solutions to the sub-problems are merged in linear time.

So,

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

Solving by using substitution method,

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

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

$= 4[2T(n/8) + cn/4] + 2cn = 8T(n/8) + 3cn = 2^3T(n/2^3) + 3c$

After applying this substitution *k* times,

$T(n) = 2^kT(n/2^k) + kcn$

To terminate this substitution process, we switch to the closed form $T(1)=c$, which happens when $2^k = n$ which implies $k=log_2 n$

So, $T(n) = nT(1) + c.log_2 n.n$

$T(n) = cn + cnlog_2 n$

$T(n) = O(nlogn)$

This is the best-case, worst-case and average-case time complexity.

Quick sort: -

→ In quicksort, the division of a[1:n] into two subarrays is made so that, the sorted sub-arrays do not need to be merged later.

→ This is accomplished by picking some element of a[ ], say *t*, which is called *pivot(or, portioning element)*, and then reordering the other elements such that all the elements which are less than or equal to *t* are placed before *t* and all the elements which are greater than *t* are placed after *t* in the array a[1:n].
This rearrangement is referred to as *partitioning.*

→ Although there are many choices for the pivot, here we assume that the *first element in the array acts as pivot*, for convenience.

→ Partition of a set of elements S about the pivot *t* produces two disjoint subsets S_1 and S_2, where

$S_1=\{x \in S-\{t\} \mid x \leq t\}$ and

$S_2=\{x \in S-\{t\} \mid x > t\}$.

→ Obviously, after a partition is achieved, pivot element *t* will be in its final position in the sorted array, and we can continue sorting the two subarrays to the left and to the right of *t* independently.

→ **Note:** The difference between mergesort and quicksort is as follows:
In mergesort, the division of the problem into two subproblems is immediate and the entire work happens in combining their solutions; but in quicksort , the entire work happens in the division stage, with no work required to combine the solutions to the subproblems.

Recursive algorithm for quick sort:

```
Algorithm QuickSort(low, high)
{
        // Sorts the elements a[low], ..., a[high] which reside in the global
        // array a[1 :n] into ascending order.
        // a[n+1] is considered which must be greater than or equal to all elements in a[1:n].

        if (low<high)  then    //If there are more than one element
        {
                j:= Partition(a,low,high+1);  // j is the final position of the partitioning element.
                QuickSort(low, j - 1);  // Solve subproblem1
                QuickSort(j+1, high);  //Solve subproblem2
                 // There is no need for combining solutions.
        }
}

Algorithm Partition(a, l, h)
{
        //a[l] is considered pivot
        t:=a[l];   lp:=l+1;    rp:=h-1;
        while(lp≤rp) do
        {
              while(a[lp]≤t) do lp:=lp+1;
              while(a[rp]>t) do rp:=rp-1;
              if(lp<rp) then
              {    // swap a[lp] and a[rp]
                   temp:= a[lp];
                   a[lp]:=a[rp];
                   a[rp]:=temp;
              }
        }
        a[l]:=a[rp];
        a[rp]:=t;
        return rp;
}
```

→Initially, QuickSort is invoked as QuickSort(1,n).

Example: Trace the QuickSort algorithm on the following data:

5, 3, 1, 9, 8, 2, 4, 7.

Index:	1	2	3	4	5	6	7	8
		lp						*rp*
Elements:	**5**	3	1	9	8	2	4	7
				lp			*rp*	
	5	3	1	9	8	2	4	7
				lp			*rp*	
	5	3	1	4	8	2	9	7
					lp	*rp*		
	5	3	1	4	8	2	9	7
					lp	*rp*		
	5	3	1	4	2	8	9	7
					rp	*lp*		
	5	3	1	4	2	8	9	7
	2	3	1	4	**5**	8	9	7
		lp		*rp*				
	2	3	1	4				
		lp	*rp*					
	2	3	1	4				
		lp	*rp*					
	2	1	3	4				
		rp	*lp*					
	2	1	3	4				
	1	**2**	3	4				
	1							
				lp, rp				
			3	4				
			rp	lp				
			3	4				
				4				
							lp	rp
						8	9	7
							lp	rp
						8	7	9
							rp	lp
						8	7	9
						7	**8**	9
						7		
								9

So, the sorted data is: 1, 2, 3, 4, 5, 7, 8, 9.

Time Complexity:

Quick Sort partitions the given set of elements S into two subsets S1 and S2 around pivot element *t*. That means, S=S1 U {*t*} U S2.
If |S| = n and |S1| = i, then |S2| = n-i-1.
So, $T(n) = T(i) + T(n-i-1) + f(n)$
$T(n) = T(i) + T(n-i-1) + cn$; since the partitioning is done in linear time.

(i) **Worst-Case Analysis:**
This occurs when the elements are in either ascending order or descending order.
→Consider that the elements are in ascending order. Then,
|S1| = 0 and |S2| = n-1.
So,
$T(n) = T(0) + T(n-1) + cn$
$T(n) = c + T(n-1) + cn$

Neglecting the insignificant constant term,
$T(n) = T(n-1) + cn$

Solving using substitution method,
$T(n) = T(n-2) + cn + c(n-1)$
$T(n) = T(n-3) + cn + c(n-1) + c(n-2)$

After k substitutions,
$T(n) = T(n-k) + cn + c(n-1) + c(n-2) + \ldots + c(n-(k-1))$
$= T(n-k) + c(n+(n-1)+(n-2)+\ldots+(n-(k-1))$

Assume n=k,
$T(n) = T(0) + c(k+(k-1)+(k-2)+\ldots+1)$
$= c + c(1+2+\ldots+k)$
$= c + c(k(k+1)/2)$
$= c+ c(n(n+1)/2)$
$T(n) = O(n^2)$.

(ii) **Best-case Analysis:**
This occurs when the pivot element occupies middle position after partitioning.
In this case, |S1| = n/2 and |S2| = n/2.
So,
$T(n) = 2T(n/2) + cn$

Solving this using substation method,

$T(n) = 2T(n/2) + cn$
$= 2[2T(n/4) + cn/2] + cn = 4T(n/4) + cn + cn = 4T(n/4) + 2cn$
$= 4[2T(n/8) + cn/4] + 2cn = 8T(n/8) + 3cn = 2^3T(n/2^3) + 3c$

After applying this substitution *k* times,
$T(n) = 2^kT(n/2^k) + kcn$

To terminate this substitution process, we switch to the closed form $T(1)=c$, which happens when $2^k = n$ which implies $k=log_2 n$.

So,
$T(n) = nT(1) + c.log_2 n.n$
$T(n) = cn + cnlog_2 n$
$T(n) = O(nlogn)$.

(iii) **<u>Average-Case Analysis:</u> (Using Probability Analysis)**
→After partitioning is performed, the pivot element may occupy any position from 1 to *n* in the array. So, S_1 can have any size from 0 to n-1 with equal probability of 1/n. Similarly S_2 also can have any size from 0 to n-1 with equal probability of 1/n.

And we know that $T(n) = T(i) + T(n-i-1) + cn$.
→So, the expected or average value of $T(i)$ is $\frac{\sum_{j=0}^{n-1} T(j)}{n}$.
Similarly, the expected or average value of $T(n-i-1)$ is also $\frac{\sum_{j=0}^{n-1} T(j)}{n}$.

Now, the equation $T(n) = T(i) + T(n-i-1) + cn$, becomes

$$T(n) = \frac{2\sum_{j=0}^{n-1} T(j)}{n} + cn \quad \text{--------} \quad (1)$$

If equation (1) is multiplied by *n*, it becomes

$$nT(n) = 2\sum_{j=0}^{n-1} T(j) + cn^2 \quad \text{--------} \quad (2)$$

From (2) we can get

$$(n-1)T(n-1) = 2\sum_{j=0}^{n-2} T(j) + c(n-1)^2 \quad \text{--------} \quad (3)$$

Subtracting (3) from (2),
$nT(n)-(n-1)T(n-1) = 2T(n-1) + c(n^2-(n-1)^2)$

$nT(n) = (n+1)\,T(n-1) + 2nc-c$

Dropping insignificant term –c, we get
$nT(n) = (n+1)\,T(n-1) + 2nc$ -------- (4)

Dividing (4) by $n(n+1)$, we get
$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2c}{n+1}$ -------- (5)

Solving the above equation using substitution method,
$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2c}{n+1}$

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

$\frac{T(n)}{n+1} = \frac{T(n-3)}{n-2} + \frac{2c}{n-1} + \frac{2c}{n} + \frac{2c}{n+1}$

After k substitutions,
$\frac{T(n)}{n+1} = \frac{T(n-k)}{n-k+1} + 2c\left(\frac{1}{n-k+2} + \frac{1}{n-k+3} + \ldots + \frac{1}{n+1}\right)$

When n=k,
$\frac{T(n)}{n+1} = \frac{T(0)}{1} + 2c\left(\frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n+1}\right)$

$\frac{T(n)}{n+1} = T(0) + 2c\left(1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n+1} - 1\right)$

Since $\sum_{i=1}^{n} \frac{1}{i} \approx \log_e n + \gamma$, where $\gamma = 0.577$ (Euler's constant),

$\frac{T(n)}{n+1} = c + 2c(\log_e(n+1) + \gamma - 1)$
$= c + 2c\log_e(n+1) + 2c(\gamma - 1)$
$= c + 2c\log_e(n+1)$
$T(n) = (n+1)(c + 2c\log_e(n+1))$
$= c(n+1) + 2c(n+1)\log_e(n+1)$
$= c(n+1) + 2cn\log_e(n+1) + 2c\log_e(n+1)$
$T(n)=O(n\log n)$

Matrix multiplication: -

Suppose we are given two square matrices A and B of order *n*x*n* each, and we wish to compute their product C = A*B, which is also an *n*x*n* matrix whose [i,j]th element is found by taking the elements in the ith row of A and jth column of B and multiplying them. So, we get,

$$C[i,j] = \sum_{1 \le k \le n} A[i,k] * B[k,j]$$

To compute any element C[i,j] using this formula, we need *n* multiplications. As the matrix C has n^2 elements, the time for the resulting matrix multiplication algorithm is $O(n^3)$.

```
Algorithm MatrixMul(A, B, n)
{
    for i:=1 to n do
    {
        for j:=1 to n do
        {
            C[i,j]:=0;
            for k:=1 to n do
            {
                C[i,j]:= C[i,j]+ A[i,k]*B[k,j];
            }
        }
    }
}
```

Matrix multiplication using divide & conquer strategy: -

Suppose we want to perform $A_{nxn} * B_{nxn} = C_{nxn}$.
We assume that n is a power of 2.

→We divide each matrix A and B into four quadrants (or, submatrices) of order *n/2* x *n/2* each.

→Then each matrix A and B can be viewed as a 2x2 matrix, whose elements are *n/2* x *n/2* matrices. Then the product A*B can be computed like the product of two 2x2 matrices. So, we can rewrite the equation C = A*B as:

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

where, each A_{ij}, B_{ij} and C_{ij} is an *n/2* x *n/2* matrix.

Now, the submatrices of C can be computed as follows:

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

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

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

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

→ Each of these four equations specifies two multiplications of *n/2* x *n/2* matrices and an addition of their *n/2* x *n/2* products. So, total 8 multiplications and 4 additions are required.

→We can use these equations to create a straightforward, recursive, divide-and-conquer algorithm. This algorithm will perform 8 recursive calls and will continue applying itself to smaller-sized submatrices until *n* becomes suitably small (i.e., *n*=2) in which case the product is computed directly (in constant time).

→Since two *n/2* x *n/2* matrices can be added in cn^2 time for some constant *c*, 4 additions of this similar kind can also be performed in cn^2 time.

→So, the overall computing time T(n) of the resulting divide-and-conquer algorithm is given by the recurrence:

$$T(n) = \begin{cases} c, & if\ n \le 2 \\ 8T\left(\frac{n}{2}\right) + cn^2, & if\ n > 2 \end{cases}$$

Solving the recurrence relation using substitution method,

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

$= 8[8T(n/2^2) + c(n/2)^2] + cn^2 = 8^2T(n/2^2)+2cn^2+cn^2 = 8^2T(n/2^2)+3cn^2$

$= 8^2[8T(n/2^3)+c(n/2^2)^2]+3cn^2 = 8^3T(n/2^3)+4cn^2+3cn^2 = 8^3T(n/2^3)+7cn^2$

$= 8^3T(n/2^3)+(2^3-1)cn^2$

After k substitutions,

$T(n) = 8^kT(n/2^k)+(2^k-1)cn^2$

$= (2^3)^kT(n/2^k)+(2^k-1)cn^2$

$= (2^k)^3T(n/2^k)+(2^k-1)cn^2$

Assume $n=2^k$ which implies $k=log_2 n$

Now,

$T(n) = n^3T(1)+(n-1)cn^2$

$= cn^3+cn^3-cn^2$

$= O(n^3)$

→Even though we used divide and conquer strategy, no improvement has been made in the time complexity over the straightforward matrix multiplication algorithm.
→Since, matrix multiplications are more expensive than matrix additions ($O(n^3)$ versus $O(n^2)$), we can reformulate the equations for C_{ij} in order to have fewer multiplications and more additions. Volker Strassen has discovered a way to compute the C_{ij}'s using only 7 multiplications and 18 additions or subtractions.

Strassen's matrix multiplication: -

→ Given two matrices A and B of order *n*x*n* each, we need compute their product C = A*B, which is also an *n*x*n* matrix. Each of the matrices A and B is divided into four quadrants (or, submatrices) of order *n/2* x *n/2* each.

So, we can rewrite the equation C = A*B as:

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

In this method, the following 7 equations are used to obtain 7 matrices of order n*/2* x *n/2* each, as follows:

$P = (A_{11} + A_{22})*(B_{11} + B_{22})$
$Q = (A_{21} + A_{22})*B_{11}$
$R = A_{11}*(B_{12} - B_{22})$
$S = A_{22}*(B_{21} - B_{11})$
$T = (A_{11} + A_{12})*B_{22}$
$U = (A_{21} - A_{11})*(B_{11}+B_{12})$
$V = (A_{12} - A_{22})*(B_{21}+B_{22})$

Now, the submatrices of C matrix can be obtained as follows:

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

→As can be seen above, the 7 matrices P, Q, R, S, T, U, and V can be computed using total 7 matrix multiplications and 10 matrix additions or subtractions. The C_{ij}'s require an additional 8 additions or subtractions.

→So, the Strassen's matrix multiplication uses total 7 multiplications and 18 additions or subtractions. The 18 additions can be performed in cn^2 time.

The resulting recurrence relation for T(n) is:

$$T(n) = \begin{cases} c, & if\ n \le 2 \\ 7T\left(\frac{n}{2}\right) + cn^2, & if\ n > 2 \end{cases}$$

Solving the recurrence relation using substitution method,

$T(n) = 7T(n/2) + cn^2$

$= 7[7T(n/2^2) + c(n/2)^2] + cn^2 = 7^2T(n/2^2)+(\frac{7}{4})cn^2+cn^2$

$= 7^2[7T(n/2^3)+c(n/2^2)^2]+3cn^2 = 7^3T(n/2^3)+(\frac{7}{4})^2cn^2+(\frac{7}{4})cn^2+cn^2$

After k substitutions,

$T(n) = 7^kT(n/2^k)+cn^2(1+(\frac{7}{4})+(\frac{7}{4})^2+\ldots+(\frac{7}{4})^{k-1})$

$$= 7^kT(n/2^k)+cn^2\left(\frac{\left(\frac{7}{4}\right)^k-1}{\left(\frac{7}{4}\right)-1}\right)$$

Assume $n=2^k$ which implies $k=log_2 n$

Now,

$$T(n) = 7^{log_2 n}T(1)+cn^2\left(\frac{\left(\frac{7}{4}\right)^{log_2 n}-1}{\left(\frac{7}{4}\right)-1}\right)$$

$$= n^{log_2 7}T(1)+cn^2\left(\frac{\left(\frac{7^{log_2 n}}{4^{log_2 n}}\right)-1}{\left(\frac{7}{4}\right)-1}\right)$$

$$= cn^{log_2 7}+cn^2\left(\frac{\left(\frac{n^{log_2 7}}{n^{log_2 4}}\right)-1}{\left(\frac{7}{4}\right)-1}\right)$$

$$= cn^{log_2 7}+cn^2\left(\frac{\left(\frac{n^{log_2 7}}{n^2}\right)-1}{\left(\frac{7}{4}\right)-1}\right)$$

$$= cn^{log_2 7}+c\left(\frac{n^{log_2 7}-n^2}{\left(\frac{7}{4}\right)-1}\right)$$

$= cn^{log_2 7}+c(n^{log_2 7}-n^2)$

$= cn^{log_2 7}+cn^{log_2 7}-cn^2)$

$T(n) = O(n^{log_2 7}) = O(n^{2.81})$

Example: Multiply the following matrices using Strassen's algorithm:

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

Sol:

We want to perform C = A * B.

Divide the given matrices into n/2 x n/2 submatrices each.

So, each of the given 4x4 matrices is divided into four 2x2 submatrices as follows:

$A_{11}=\begin{bmatrix}1 & 2\\ 2 & 1\end{bmatrix}$ $A_{12}=\begin{bmatrix}2 & 1\\ 2 & 1\end{bmatrix}$ $A_{21}=\begin{bmatrix}1 & 2\\ 1 & 1\end{bmatrix}$ $A_{22}=\begin{bmatrix}1 & 2\\ 2 & 2\end{bmatrix}$

$B_{11}=\begin{bmatrix}2 & 1\\ 1 & 1\end{bmatrix}$ $B_{12}=\begin{bmatrix}2 & 2\\ 1 & 2\end{bmatrix}$ $B_{21}=\begin{bmatrix}2 & 1\\ 1 & 2\end{bmatrix}$ $B_{22}=\begin{bmatrix}1 & 1\\ 1 & 2\end{bmatrix}$

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

$=\begin{bmatrix}1 & 2\\ 2 & 1\end{bmatrix}+\begin{bmatrix}1 & 2\\ 2 & 2\end{bmatrix}*\begin{bmatrix}2 & 1\\ 1 & 1\end{bmatrix}+\begin{bmatrix}1 & 1\\ 1 & 2\end{bmatrix}$

$=\begin{bmatrix}2 & 4\\ 4 & 3\end{bmatrix}*\begin{bmatrix}3 & 2\\ 2 & 3\end{bmatrix}$

$=\begin{bmatrix}14 & 16\\ 18 & 17\end{bmatrix}$

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

$=\begin{bmatrix}1 & 2\\ 1 & 1\end{bmatrix}+\begin{bmatrix}1 & 2\\ 2 & 2\end{bmatrix}*\begin{bmatrix}2 & 1\\ 1 & 1\end{bmatrix}$

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

$=\begin{bmatrix}8 & 6\\ 9 & 6\end{bmatrix}$

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

$=\begin{bmatrix}1 & 2\\ 2 & 1\end{bmatrix}*\begin{bmatrix}2 & 2\\ 1 & 2\end{bmatrix}-\begin{bmatrix}1 & 1\\ 1 & 2\end{bmatrix}$

$=\begin{bmatrix}1 & 2\\ 2 & 1\end{bmatrix}*\begin{bmatrix}1 & 1\\ 0 & 0\end{bmatrix}=\begin{bmatrix}1 & 1\\ 2 & 2\end{bmatrix}$

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

$$= \begin{bmatrix}1 & 2\\2 & 2\end{bmatrix} * \begin{bmatrix}2 & 1\\1 & 2\end{bmatrix} - \begin{bmatrix}2 & 1\\1 & 1\end{bmatrix}$$

$$= \begin{bmatrix}1 & 2\\2 & 2\end{bmatrix} * \begin{bmatrix}0 & 0\\0 & 1\end{bmatrix}$$

$$= \begin{bmatrix}0 & 2\\0 & 2\end{bmatrix}$$

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

$$= \begin{bmatrix}1 & 2\\2 & 1\end{bmatrix} + \begin{bmatrix}2 & 1\\2 & 1\end{bmatrix} * \begin{bmatrix}1 & 1\\1 & 2\end{bmatrix}$$

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

$$= \begin{bmatrix}6 & 9\\6 & 8\end{bmatrix}$$

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

$$= \begin{bmatrix}1 & 2\\1 & 1\end{bmatrix} - \begin{bmatrix}1 & 2\\2 & 1\end{bmatrix} * \begin{bmatrix}2 & 1\\1 & 1\end{bmatrix} + \begin{bmatrix}2 & 2\\1 & 2\end{bmatrix}$$

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

$$= \begin{bmatrix}0 & 0\\-4 & -3\end{bmatrix}$$

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

$$= \begin{bmatrix}2 & 1\\2 & 1\end{bmatrix} - \begin{bmatrix}1 & 2\\2 & 2\end{bmatrix} * \begin{bmatrix}2 & 1\\1 & 2\end{bmatrix} + \begin{bmatrix}1 & 1\\1 & 2\end{bmatrix}$$

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

$$= \begin{bmatrix}1 & -2\\-2 & -4\end{bmatrix}$$

Now calculating the submatrices of resultant matrix C:

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

$$\begin{bmatrix}14 & 16\\18 & 17\end{bmatrix} + \begin{bmatrix}0 & 2\\0 & 2\end{bmatrix} - \begin{bmatrix}6 & 9\\6 & 8\end{bmatrix} + \begin{bmatrix}1 & -2\\-2 & -4\end{bmatrix}$$

$= \begin{bmatrix} 9 & 7 \\ 10 & 7 \end{bmatrix}$

C_{12}=R+T

$= \begin{bmatrix} 1 & 1 \\ 2 & 2 \end{bmatrix} + \begin{bmatrix} 6 & 9 \\ 6 & 8 \end{bmatrix}$

$= \begin{bmatrix} 7 & 10 \\ 8 & 10 \end{bmatrix}$

C_{21}=Q+S

$= \begin{bmatrix} 8 & 6 \\ 9 & 6 \end{bmatrix} + \begin{bmatrix} 0 & 2 \\ 0 & 2 \end{bmatrix}$

$= \begin{bmatrix} 8 & 8 \\ 9 & 8 \end{bmatrix}$

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

$= \begin{bmatrix} 14 & 16 \\ 18 & 17 \end{bmatrix} + \begin{bmatrix} 1 & 1 \\ 2 & 2 \end{bmatrix} - \begin{bmatrix} 8 & 6 \\ 9 & 6 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ -4 & -3 \end{bmatrix}$

$= \begin{bmatrix} 7 & 11 \\ 7 & 11 \end{bmatrix}$

The resultant matrix, C= $\begin{bmatrix} C11 & C12 \\ C21 & C22 \end{bmatrix} = \begin{bmatrix} 9 & 7 & 7 & 10 \\ 10 & 7 & 8 & 10 \\ 8 & 8 & 7 & 11 \\ 9 & 8 & 7 & 10 \end{bmatrix}$

THE GREEDY METHOD

→The Greedy method is used to solve many of the optimization (i.e., minimization or maximization) problems.

→Some optimization problems also involve some constraints. A solution that satisfies these constraints is called feasible solution. A feasible solution which optimizes (i.e., minimizes or maximizes) a given objective function is called optimal solution.

→ The greedy approach suggests constructing a solution through a sequence of steps, each step expanding a partially constructed solution obtained so far until a complete solution to the problem is reached.

→On each step (and this is the central point of this technique) the choice made must be:

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

→These requirements explain the technique's name: at each step, it suggests a "greedy" grab of the best alternative available in the hope that a sequence of locally optimal choices will yield a (globally) optimal solution to the entire problem.

→There are problems for which a sequence of locally optimal choices yields an optimal solution for every instance of the given problem.

→However, there are some other problems for which this is not the case (that means every time we may not get optimal solution). For such problems, a greedy algorithm can still be of value if we are interested in or have to be satisfied with an approximate (or, near optimal) solution.

→Before Greedy algorithm begins, we set up a selection criterion (called greedy criterion) which decides the order of selection of inputs.

CONTROL ABSTRACTION FOR GREEDY METHOD:

```
Algorithm Greedy(a,n)
// a[l : n] contains the n inputs.
{
      solution:=Φ ;   // Initialize the solution.
      for i:= 1 to n do
      {
           x := Select(a); // Select next i/p from 'a' based on greedy
                           // criterion and assign its value to x
          if (Feasible(solution, x)) then  // Check whether the inclusion of
                                           // x into partially constructed
                                           // solution results in feasible
                                           // solution
                solution := Union(solution, x) ;
      }
 return solution;
}
```

FRACTIONAL KNAPSACK PROBLEM:

→We are given a set of 'n' items (or, objects), such that each item i has a weight 'w_i' and a profit 'p_i'. We wish to pack a knapsack whose capacity is 'M' with a subset of items such that total profit is maximized.

→Further, we are allowed to break each item into fractions arbitrarily. That means, for an item i we can take an amount $w_i x_i$ (which gives a profit of $p_i x_i$) such that $0 \leq x_i \leq 1$, and $\sum_{i=1}^{n} w_i\, x_i \leq$ M.

→Formally the fractional knapsack problem can be stated as follows:

Maximize $\sum_{i=1}^{n} p_i\, x_i$

Subject to the following constraints:

$\sum_{i=1}^{n} w_i\, x_i \leq$ M

and

$0 \leq x_i \leq 1$, $(1 \leq i \leq n)$.

→The solution to this problem is represented as the vector $(x_1, x_2, \ldots, x_n)$.

Note: If the sum of all weights is less than or equal to M (i.e., $\sum_{i=1}^{n} w_i x_i \leq M$), then all the items can be placed in the knapsack which results in the solution x_i=1, 1 ≤ i ≤ n.

POSSIBLE GREEDY STRATEGIES (OPTIMIZATION METHODS):

There are greedy several strategies possible. In each of these strategies the knapsack is packed in several stages.

In each stage, one item is selected for inclusion into the knapsack using the chosen strategy.

1. TO BE GREEDY ON PROFIT: (Don't use it)

The selection criterion is "From the remaining objects, select the object with maximum profit that fits into the knapsack".

Using this criterion, the object with the largest profit is packed first (provided enough capacity is available), then the one with next largest, and so on.

That means the objects are selected in the decreasing order of their profit values.

This strategy does not always guarantee an optimal solution, but only a suboptimal solution.

Example: Consider the following instance of the knapsack problem:
n = 3, M = 20, (p_1, p_2, p_3) = (24, 25, 15), and (w_1, w_2, w_3) = (15, 18, 10).
Solution: Objects are selected and placed into the knapsack in decreasing order of their profits.

i	1	2	3
p_i	24	25	15
w_i	15	18	10
Order of Selection (or) Rank)	2	1	3
x_i	2/15	1	0

So when we are greedy on profit, we obtain the solution as
$(x_1, x_2, x_3) = (\frac{2}{15}, 1, 0)$
which gives a profit of $(24 \times \frac{2}{15}) + (25 \times 1) + (15 \times 0) = 3.2 + 25 + 0 = 28.2$

It is not an optimal solution, as there is a superior solution to this, i.e., $(x_1, x_2, x_3) = (1, 0, \frac{1}{2})$ which gives a profit of 24+15*/2 = 24 + 7.5 = 31.5.

2. TO BE GREEDY ON WEIGHT: (Don't use it)

The selection criterion is "From the remaining objects select the one with minimum weight that fits into knapsack."

That means, the objects are selected in increasing order of their weights.

It does not always yield optimal solution.

Example: Consider the following instance of the knapsack problem:
n = 3, M = 20, $(p_1, p_2, p_3) = (24, 25, 15)$, and $(w_1, w_2, w_3) = (15, 18, 10)$.
Solution: Objects are selected and placed into the knapsack in increasing order of their weights.

i	1	2	3
p_i	24	25	15
w_i	15	18	10
Order of Selection (or) Rank)	2	3	1
x_i	$\frac{10}{15} = \frac{2}{3}$	0	1

So when we are greedy on weights, we obtain the solution as

$(x_1, x_2, x_3) = (\frac{2}{3}, 0, 1)$

which yields a profit of $(24\times\frac{2}{3}) + (25x0) + (15\times1) = 16 + 0 + 15 = 31$.

It is not an optimal solution, as there is a superior solution to this, i.e., $(x_1, x_2, x_3) = (1, 0, \frac{1}{2})$ which gives a profit of 24+15*/2 = 24 + 7.5 = 31.5.

3. TO BE GREEDY ON PROFIT DENSITY (PROFIT PER UNIT WEIGHT):

The selection criterion is "From the remaining objects, select the one with maximum p_i/w_i ratio that fits into the knapsack."

That means, the objects are selected in decreasing order of their p_i/w_i ratios.

This strategy always produces an optimal solution to the fractional knapsack problem.

Example: Consider the following instance of the knapsack problem:
n = 3, M = 20, (p_1, p_2, p_3) = (24, 25, 15), and (w_1, w_2, w_3) = (15, 18, 10).
Solution:

i	1	2	3
p_i	24	25	15
w_i	15	18	10
p_i/w_i	$\frac{24}{15}$ = 1.6	$\frac{25}{18}$ = 1.38	$\frac{15}{10}$ = 1.5
Order of Selection (or) Rank)	1	3	2
x_i	1	0	$\frac{5}{10} = \frac{1}{2}$

The optimal solution is: (x_1, x_2, x_3) = (1, 0, $\frac{1}{2}$)

which yields a profit of 24*1 + 25*0 + 15*$\frac{1}{2}$ = 24+ 0 + 7.5 = 31.5.

Algorithm for knapsack problem using greedy method:

```
Algorithm Greedy_knapsack (M, n)
// p [l: n] and w [1: n] contain the profits and weights respectively of n
//objects which are ordered such that p[i]/w[i] ≥ p[i+1]/w[i+1] .
// M is the knapsack capacity and x[1:n] is the solution vector.
{
        for i := 1 to n do
            x[i] := 0.0;     //Initialization of solution vector
        RC := M;  // Remaining capacity knapsack
        for i := 1 to n do
        {
            if (w[i] > RC) then break;
            else
            {
                 x[i] = 1.0;
                RC= RC- w[i];
            }
         }
   if (i ≤ n) then x[i] = RC/w[i];

}
```

Ignoring the time taken initially sort the objects according to their profit densities, the time complexity of this algorithm O(n).

Exercises:

1) n=7, M = 15, (p1,p2,p3,p4,p5,p6,p7) = (10, 5, 15, 7, 6, 18, 7), and (w1,w2,w3,w4,w5,w6,w7) = (2,3,5,7,1,4,1).

2) n=7, M = 15, (p1,p2,p3,p4,p5,p6,p7) = (10, 5, 15, 7, 6, 18, 3), and (w1,w2,w3,w4,w5,w6,w7) = (2,3,5,7,1,4,1).

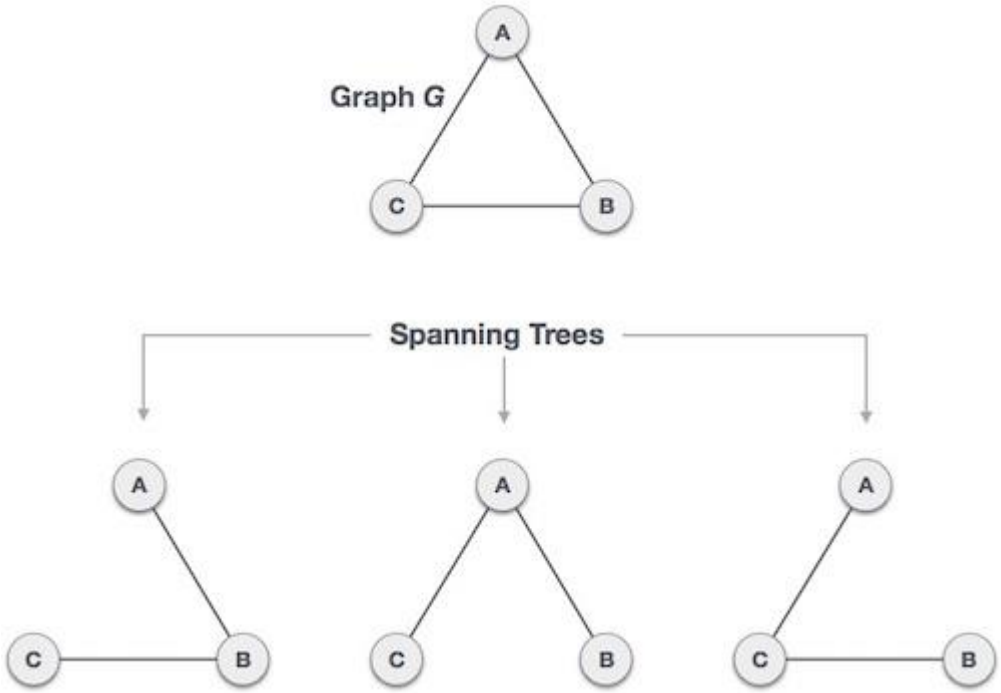
SPANNING TREE:

→Let G = (V, E) be an undirected connected graph. A subgraph G' = (V, E') of G (where E' is subset of E) is called a spanning tree of G iff G' is a tree.

→The spanning tree of a given connected graph with *n* vertices is a connected subgraph with *n* vertices but without any cycles. That means it will have *n*-1 edges.

→Every connected and undirected Graph G has at least one spanning tree.
A disconnected graph does not have any spanning tree.

Example: A complete graph with three nodes together with all of its spanning trees.



General Properties of Spanning Tree:

- A connected graph G can have more than one spanning tree.
- 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 cycle, i.e. the spanning tree is **maximally acyclic**.

Mathematical Properties of Spanning Tree:

- Spanning tree has **n-1** edges, where **n** is the number of nodes (vertices).
- From a complete graph, by removing maximum **e - n + 1** edges, we can construct a spanning tree.
- A complete graph can have maximum $\mathbf{n^{n-2}}$ number of spanning trees.

MINIMUM SPANNING TREE (or, MINIMUM COST SPANNING TREE):

→If a graph G has weights (costs) assigned to its edges, a ***minimum spanning tree(***MST) is its spanning tree with the *smallest sum of the weights on all its edges*.

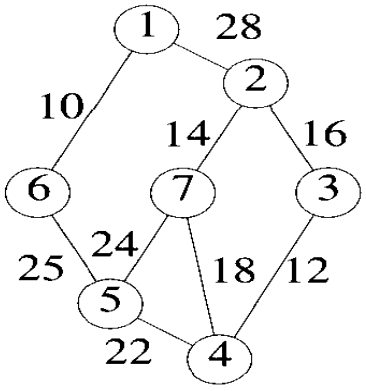
→The ***minimum cost spanning tree problem*** is the problem of finding a minimum spanning tree for a given weighted connected graph.

→There are two algorithms available to find an MST for a given graph:
1. Prims' Algorithm
2. Kriskal's algorithm

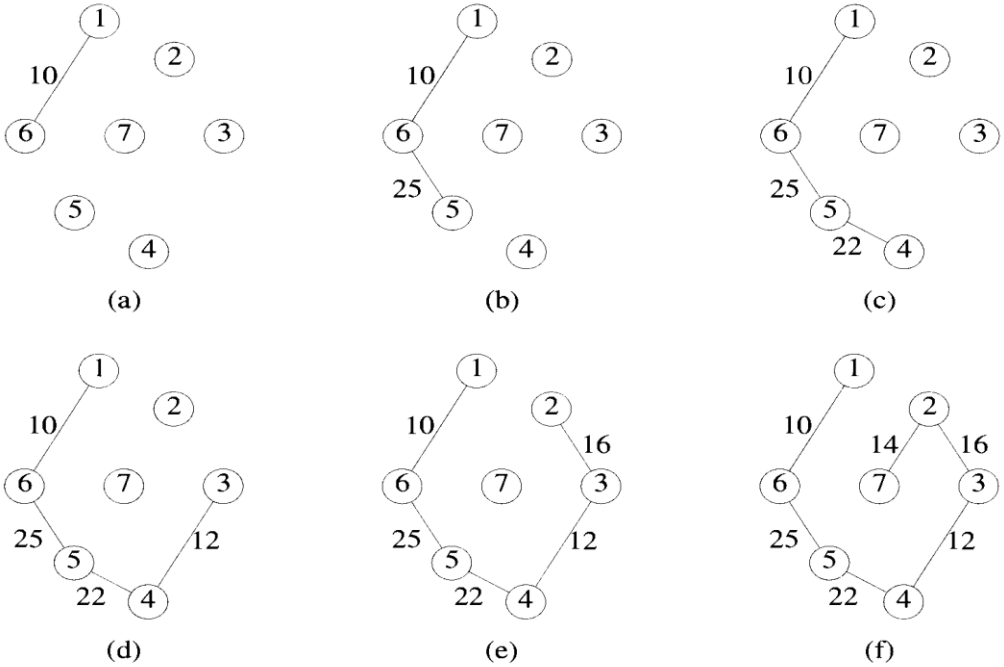
PRIM'S ALGORITHM:

→It is a greedy algorithm to obtain a minimal cost spanning tree.
→It builds MST edge by edge.
→The next edge to include is chosen according to the following selection criterion:
"Choose an edge that results in a minimum increase in the sum of the costs of the edges so far included provided that the inclusion of it will form a tree."
→That means, among the edges which are incident on vertices that are selected so far, select the one with minimum cost, provided that the inclusion of it will not result in a cycle.
→ This algorithm guarantees that a tree is formed at each intermediate stage.

Example: Find the minimum cost spanning tree of the following graph using Prim's algorithm, showing different stages in constructing the tree.



Stages in Prim's algorithm:



Cost of minimum spanning tree is = 10 + 25 + 22 + 12 + 16 + 14 = 99.

ALGORITHM:

```
Algorithm Prim(E, cost, n, t)
// E is the set of edges in G.
// cost[l : 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,l],t[i,2]) is an i^th 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[l, l] := k; t[1, 2] := l;
    for i := 1 to n do   // Initialize near[ ] array.
        if (cost[i,l] < cost[i,k]) // If l is nearer to i than 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[ ] array
            if (near [k] ≠ 0) and ((cost [k, near [k]) > (cost [k, j])) then
                near [k] :=j;
    }
 return mincost
 }
```

NOTE: To efficiently determine the next edge (i,j) to be added in the spanning tree, we use near [ ] array.

→The running time of Prim's algorithm is $O(n^2)$.

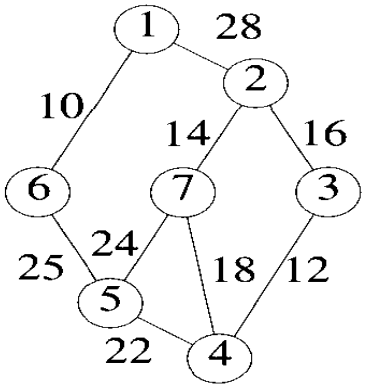
Kruskal's Algorithm:

→It is also a greedy method to obtain a minimal cost spanning tree.
→It also builds MST edge by edge.
→The next edge to include is chosen according to the following selection criterion:
"Choose an edge with smallest weight provided that the inclusion of it will not result in a cycle."

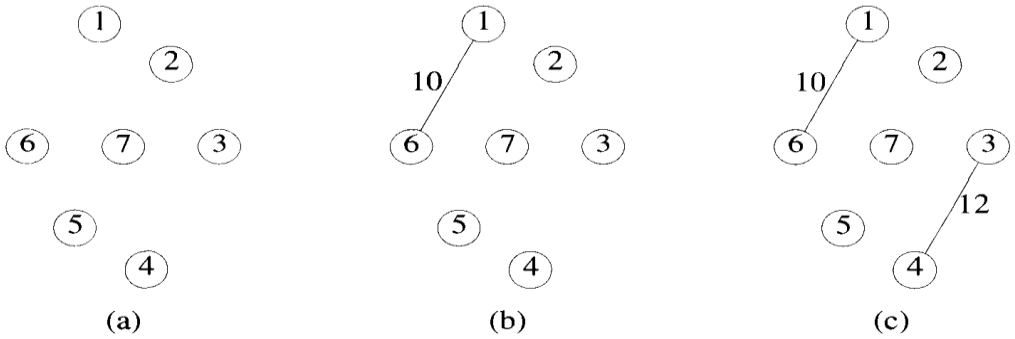
→In other words, the edges are selected in increasing order of weights provided cycle will not be created at any stage.

Note: Unlike in Prim's algorithm, the set of edges so far included need not form a tree at all stages in Kruskal's algorithm, i.e., a forest of trees may be generated.

Example: Find the minimum cost spanning tree of the following graph using Kruskal's algorithm, showing different stages in constructing the tree.



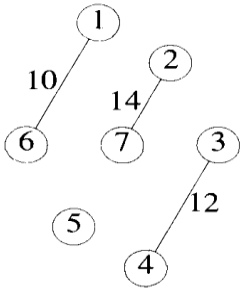
Stages in Kruskal's algorithm:



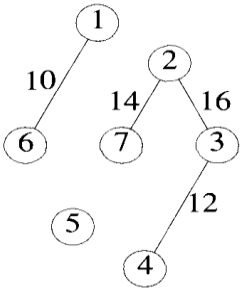
(a)

(b)

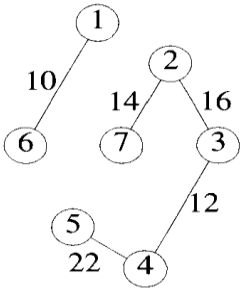
(c)



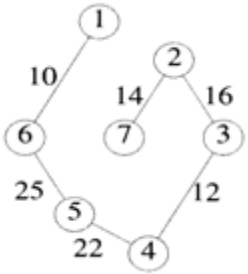
(d)



(e)



(f)



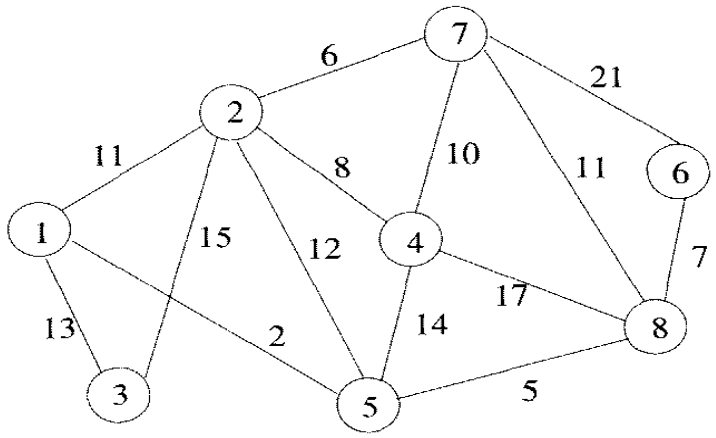
(g)

Cost of minimum spanning tree = 10 + 12 + 14 + 16 + 22 + 25 = 99.

Simple version of Kruskal's algorithm:

t = Φ;
while ((t has less than *n* - *1* edges) **and** (E ≠ Φ)) **do**
{
choose an edge (v, w) of lowest weight from E;
delete (v, w) from E;
if (v, w) does not create a cycle in t **then**
add (v, w) to t;
else discard (v, w);
}

Exercise: Compute MST for the following graph using Prim's and Kruskal's algorithms:



SINGLE SOURCE SHORTEST PATHS PROBLEM: (DIJKSTRA'S ALGORITHM)

PROBLEM DESCRIPTION:

We are given a directed graph (digraph) G = (V,E) with the property that each edge has a non negative weight. We must find the shortest paths from a given source vertex v_0 to all the remaining vertices (called destinations) to which there is a path.

Length of the path is the sum of the weights of the edges on the path.

A GREEDY SOLUTION:

We can solve the shortest paths problem using a greedy algorithm, developed by Dijkstra, that generates the shortest paths in stages.

In each stage, a shortest path to a new destination vertex is generted.

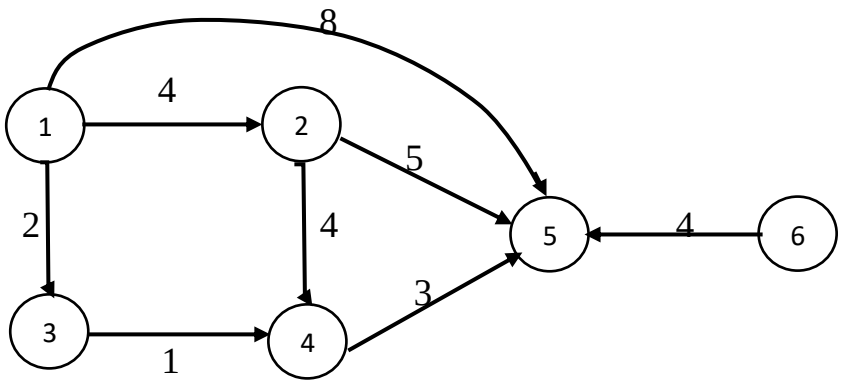
The destination vertex for the next shortest path is selected using the following greedy criterion:

" From the set of vertices, to which a shortest path has not been generated, select one that results in the least path length (or, one that is closest to the source vertex)." .

In other words, Dijkstra's method generates the shortest paths in increasing order of path lengths.

→We represent an *n* vertex graph by an *n*×*n* cost adjacency matrix *cost*, with *cost[i, j]* being the weight of the edge <i, j>. In case the edge <i, j> is not present in the graph, its cost *cost[i, j]* is set to some large number (∞). For i = j, *cost[i, j]* can be set to any non negitive number such as 0.

Example:



Cost adjacency matrix:

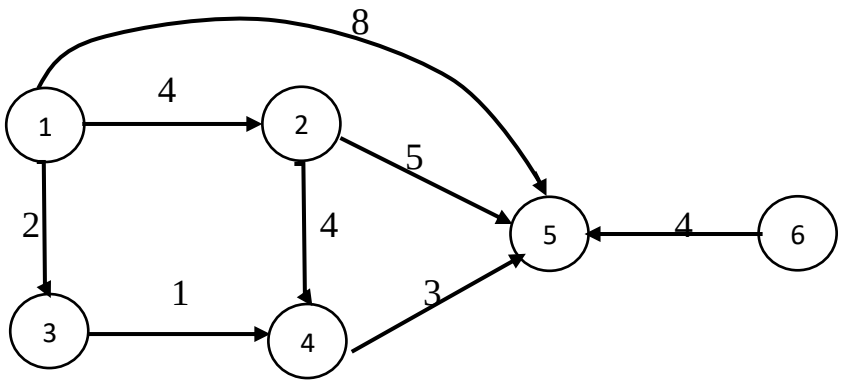
$$\begin{bmatrix} 0 & 4 & 2 & \infty & 8 & \infty \\ \infty & 0 & \infty & 4 & 5 & \infty \\ \infty & \infty & 0 & 1 & \infty & \infty \\ \infty & \infty & \infty & 0 & 3 & \infty \\ \infty & \infty & \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & \infty & 4 & 0 \end{bmatrix}$$

→Let 'S' denote the set of vertices (including source vertex v_0) to which the shortest paths have already been generated.

The set 'S' is maintained as a bit array with S[i] = 0 if vertex 'i' is not in 'S' and S[i] = 1 if vertex 'i' is in 'S'.

→ To store the lengths of the resultant shortest paths from the source vertex to remaining vertices, we use an array *dist[1:n]*.

Example: **O**btain the shortest paths in increasing order of lengths from vertex 1 to all remaining vertices in the following digraph.



Solution:

Step1: Finding the next shortest path

$1 \xrightarrow{4} 2$

$1 \xrightarrow{2} 3$ (path with minimum distance)

$1 \xrightarrow{8} 5$

$1 \xrightarrow{\infty} 4$

$1 \xrightarrow{\infty} 6$

Step2:

(i) Update distances of those nodes adjacent to 3 (if possible) which are not covered:

$1 \xrightarrow{2} 3 \xrightarrow{1} 4$

(ii) Finding the next shortest path

$1 \xrightarrow{4} 2$

$1 \xrightarrow{2} 3 \xrightarrow{1} 4$ (path with minimum distance)

$1 \xrightarrow{8} 5$

$1 \xrightarrow{\infty} 6$

Step 3:

(i) Update distances of those nodes adjacent to 4 (if possible) which are not covered:

$1 \xrightarrow{2} 3 \xrightarrow{1} 4 \xrightarrow{3} 5$

(ii) Finding the next shortest path

$1 \xrightarrow{4} 2$ (path with minimum distance)

$1 \xrightarrow{2} 3 \xrightarrow{1} 4 \xrightarrow{3} 5$

$1 \xrightarrow{\infty} 6$

Step 4:

(i) Update distances of those nodes adjacent to 2 (if possible) which are not covered:

Node 5 is adjacent to node 2. But distance of 5 via node 2 (i.e., 9) will be more than its current distance (i.e., 6). So, its distance is not updated.

(ii) Finding the next shortest path

$1 \xrightarrow{2} 3 \xrightarrow{1} 4 \xrightarrow{3} 5$ (path with minimum distance)

$1 \xrightarrow{\infty} 6$

Step 5:

(i) Update distances of those nodes adjacent to 5 (if possible) which are not covered:

Nothing is adjacent to node 5.

(ii) Finding the next shortest path

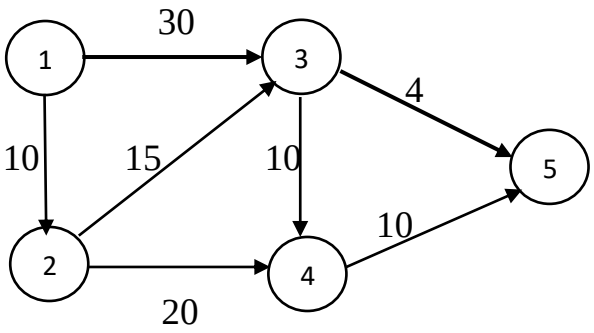
$1 \xrightarrow{\infty} 6$

There is no path to node 6.

Shortest paths in increasing order:

Shotest Paths	**Lengths**
$1 \xrightarrow{2} 3$	2
$1 \xrightarrow{2} 3 \xrightarrow{1} 4$	3
$1 \xrightarrow{4} 2$	4
$1 \xrightarrow{2} 3 \xrightarrow{1} 4 \xrightarrow{3} 5$	6

Exercise: Obtain the shortest paths in increasing order of lengths from vertex 1 to all remaining vertices in the following digraph.



ALGORITHM FOR SHORTEST PATHS:

Algorithm ShortestPaths(*v*, cost, dist, n)
//graph G with n vertices is represented by its cost adjacency matrix cost[1:n,1:n].
//dist [i], 1≤ i ≤ n, is set to the length of the shortest path from source vertex v to the vertex i.

```
{
    for i:= 1 to n do
    {
        S[i] := 0;  // Initialize S[ ] array
        dist[i] := cost[v, i];    //Initialize dist[ ] array
    }
    S[v] := 1;    //put v in S
    dist[v] := 0;
    for j := 2 to n do
    {
        //determine n-1 paths from v
        choose a vertex u from among those vertices not in S, such that dist[u] is minimum;
        S[u] := 1;   // put u in S
        for (each vertex w adjacent to u with S[w] = 0) do
        {
            // Update distances.
            if (dist[w] > (dist[u] +cost[u, w]) ) then
            {
                dist[w] := dist[u] +cost[u, w];
            }
        }
    }
}
```

Job sequencing with deadlines

→We are given a set of *n* jobs where each job *i* has a deadline $d_i > 0$ and a profit $p_i > 0$ associated with it.

→**Each job takes one unit of time** on a machine (or, processor) for its processing (or, execution) and only one machine is available and it can process only one job at a time.

→ For any job *i* the profit p_i is earned if and only if the job is completed within its deadline.

→We have to choose a subset of jobs all of which are processed within their deadline and which results in maximum profit.

Greedy solution:

→The selection of next job is done according the following selection criterion:
"From the remaining jobs, select the one with maximum profit provided there is a time slot available on the machine to process it within its deadline."

→That means, the jobs are selected in the decreasing order of their profits.

Steps for solving the problem:

→Find out the total number of time slots required on the machine, which is equal to the maximum of deadlines of all jobs.

→Create an array *J* whose size is equal to the number of time slots.

→Initialize *J* array with 0's, to mean that no job has been selected,

→From the remaining jobs, select the one with maximum profit and place it in a time slot which is equal to its deadline, provided the time slot is vacant. If the time slot is not vacant, place the job in a time slot less than its deadline but nearest to its deadline.

Example: Solve the following job sequencing problem instance:
n=4, (d1, d2, d3, d4) = (2, 1, 2, 1) and (p1, p2, p3, p4) = (100, 10, 15, 21).

Solution:

Maximum deadline is 2.

So, total number of time slots required on the machine is 2.

Job i	1	2	3	4
d_i	2	1	2	1
p_i	100	10	15	21
Order of Selection (or) Rank)	1	4	3	2

Initialization:

Time slot number:	1	2
Job Array:	0	0

Step 1:

Time slot number:	1	2
Job number:	0	1

Step 2:

Time slot number:	1	2
Job number:	4	1

Optimal solution is: {4, 1} which gives a profit of 21+100 = 121.

That means, the jobs are processed in the order of 4 followed by 2 which can be processed within their deadlines and which give a maximum profit of 121.

Exercise: Solve the following job sequencing problem instance:
n=5, (d1, d2, d3, d4, d5) = (2, 1, 3, 2, 1) and
(p1, p2, p3, p4, p5) = (60, 100, 20, 40, 20).

Algorithm:

```
Algorithm JS(d, J, n)
// d[i]≥1, 1≤i≤n are deadlines of n jobs where n≥1.
// The job are ordered such that p[1] ≥ p[2] ≥ ... ≥ p[n].
// J[i] is the i^th job in the optimal solution, 1≤i≤k.
{
        Let the maximum deadline be k.
        Create array J of size k and initialize it with 0's.
        for i := 1 to k do
        {
            r := d[i];
            if(J[r] = 0) then   // If the time slot equal to the deadline of  i^th job is vacant
                J[r] := i;  // Place job i in array J in a position equal to the deadline of  i^th job.
            else  // If the time slot equal to the deadline of  i^th job is not vacant
            {
                 // Search, for vacancy, the time slots previous to the current slot
                r := r-1;
                while(r > 0) do
                {
                    if(J[r] = 0) then
                        J[r] := i;
                    r := r-1;
                }
            }
        }
}
```