

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.

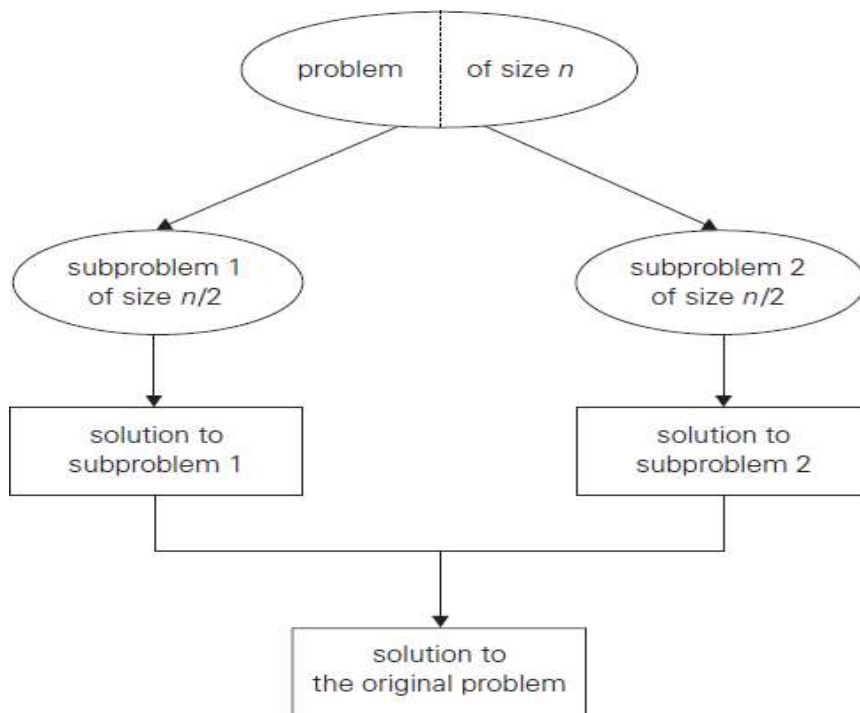


Figure: Divide-and-conquer technique (typical case)

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  $k \geq 1$ ;
8.       Apply DandC to each of these k 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 and if 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 n_1, n_2, \dots, n_k respectively, then the computing time of DandC is described by the recurrence relation:

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

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 taken for dividing P into sub-problems and combining the solutions of sub-problems.

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

$$T(n) = \begin{cases} c, & \text{if } n = 1 \\ a T\left(\frac{n}{b}\right) + f(n), & \text{if } n > 1 \end{cases}$$

where a , b and c are constants.

We assume that n is a power of b (i.e., $n = b^k$, for some $k \geq 1$).

Eg :-1. Solve the following recurrence relation using substitution method:

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

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2[2T(n/4) + n/2] + n = 4T(n/4) + 2n \\ &= 4[2T(n/8) + n/4] + 2n = 8T(n/8) + 3n = 2^3T(n/2^3) + 3n \end{aligned}$$

After applying this substitution k times,

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

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

$$\text{So, } T(n) = nT(1) + n \log_2 n$$

$$\begin{aligned} T(n) &= n + n \log n \\ &= O(n \log n) \end{aligned}$$

Note:

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

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

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

$$(4) \quad 1 \cdot 2 + 2 \cdot 2^2 + 3 \cdot 2^3 + \dots + k \cdot 2^k = (k - 1)2^{k+1} + 2$$

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

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

2)

Q) solve

sol:

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

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

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

$$= 2^3 \log \frac{n}{8} + 2^2 \log \frac{n}{4} + 2 \log \frac{n}{2} + \log n$$

After k substitutions,

$$T(n) = 2^k T\left(\frac{n}{2^k}\right) + \log n + 2 \log \frac{n}{2} + 2^2 \log \frac{n}{2^2} + \dots + 2^{k-1} \log \frac{n}{2^{k-1}}$$

$$= 2^k T\left(\frac{n}{2^k}\right) + \log n + 2(\log n - \log 2) + 2^2(\log n - \log 2^2) + \dots + 2^{k-1}(\log n - \log 2^{k-1})$$

$$= 2^k T\left(\frac{n}{2^k}\right) + \log n + 2 \log n - 2 \cdot 1 + 2^2 \log n - 2 \cdot 2^2 + \dots + 2^{k-1} \log n - (k-1) \cdot 2^{k-1}$$

$$= 2^k T\left(\frac{n}{2^k}\right) + \log n (1 + 2 + 2^2 + \dots + 2^{k-1}) - (1 \cdot 2 + 2 \cdot 2^2 + \dots + (k-1) \cdot 2^{k-1})$$

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

Assuming $n = 2^k \Rightarrow k = \log n$

$$\Rightarrow T(n) = 1 \cdot n T(1) + \log n (n-1) - ((\log n - 2)n + 2)$$

$$= n + \log n - \log n - 2 \log n + 2n - 2$$

$$T(n) = cn + 2n - \log n - 2$$

$$\Rightarrow T(n) = O(n)$$

Solving Recurrence Relations Using Recursion Tree Method: -

(Reference: <https://www.gatevidyalay.com/recursion-tree-solving-recurrence-relations/>)

Problem-1:

Solve the following recurrence relation using recursion tree method:

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

Solution-

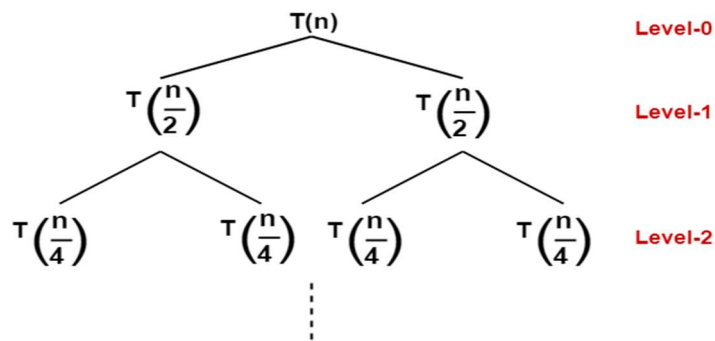
Step-01:

Draw a recursion tree based on the given recurrence relation.

The given recurrence relation shows-

- A problem of size n will get divided into 2 sub-problems of size $n/2$.
- Then, each sub-problem of size $n/2$ will get divided into 2 sub-problems of size $n/4$ and so on.
- At the bottom most level, the size of sub-problems will reduce to 1.

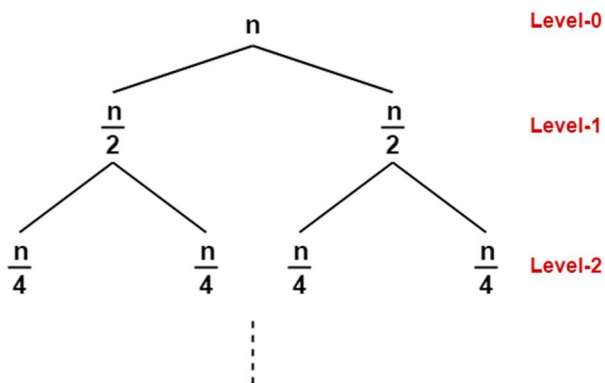
This is illustrated through following recursion tree-



The given recurrence relation shows-

- The cost of dividing a problem of size n into its two sub-problems and then combining their solutions is n .
- The cost of dividing a problem of size $n/2$ into its two sub-problems and then combining their solutions is $n/2$ and so on.

This is illustrated through following recursion tree where each node represents the cost of the corresponding sub-problem:



Step-02:

Determine cost of each level:

- Cost of level-0 = n
- Cost of level-1 = $n/2 + n/2 = n$
- Cost of level-2 = $n/4 + n/4 + n/4 + n/4 = n$, and so on.

Step-03:

Determine total number of levels in the recursion tree:

- Size of problem at level-0 = $n/2^0$
- Size of each sub-problem at level-1 = $n/2^1$
- Size of each sub-problem at level-2 = $n/2^2$

Continuing in similar manner, we have:

Size of each sub-problem at level- i = $n/2^i$

Suppose at level- x (last level), size of each sub-problem becomes 1. Then

$$n / 2^x = 1$$

$$2^x = n$$

Taking log on both sides, we get,

$$x = \log_2 n$$

$$\therefore \text{Total number of levels in the recursion tree} = x+1 = \log_2 n + 1$$

Step-04:

Determine number of nodes in the last level:

- Level-0 has 2^0 nodes i.e., 1 node
- Level-1 has 2^1 nodes i.e., 2 nodes
- Level-2 has 2^2 nodes i.e., 4 nodes

Continuing in similar manner, we have,

Last level (i.e., Level - $\log_2 n$) has $2^{\log_2 n} = n^{\log_2 2} = n$ nodes.

Step-05:

Determine cost of last level:

Cost of last level = $n * T(1) = cn = \theta(n)$.

Step-06:

Add costs of all the levels of the recursion tree and simplify the expression so obtained in terms of asymptotic notation:

$$T(n) = \underbrace{\{ n + n + n + \dots \}}_{\text{For } \log_2 n \text{ levels}} + \theta(n)$$

$$= n \log_2 n + \theta(n)$$

$$= \theta(n \log n)$$

Problem-2:

Solve the following recurrence relation using recursion tree method:

$$T(n) = T(n/5) + T(4n/5) + n$$

Solution-

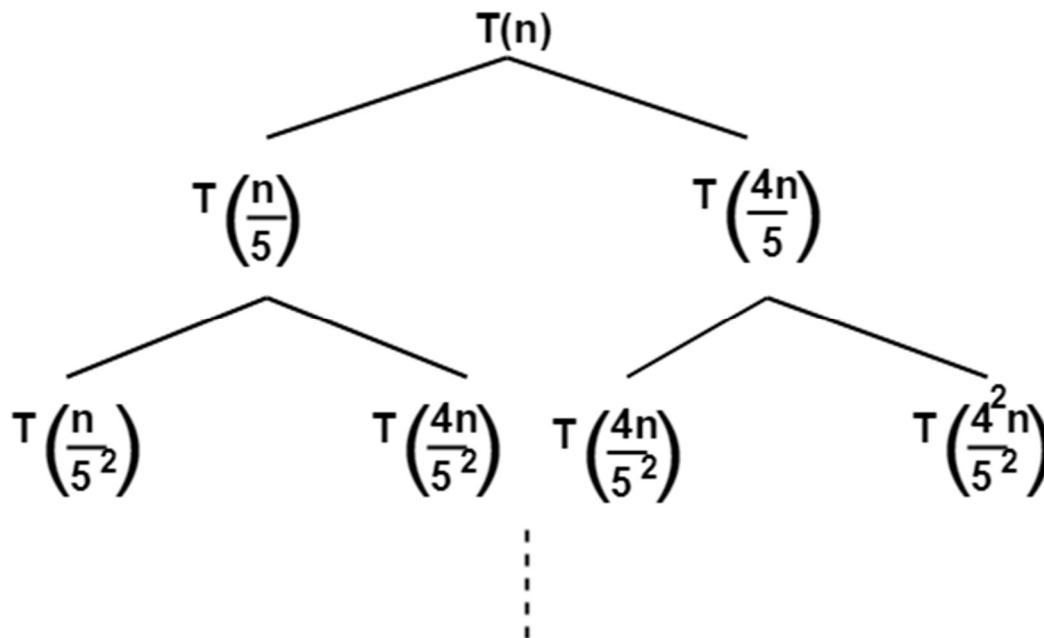
Step-01:

Draw a recursion tree based on the given recurrence relation:

The given recurrence relation shows-

- A problem of size n will get divided into two sub-problems- one of size $n/5$ and another of size $4n/5$.
- Then, sub-problem of size $n/5$ will get divided into two sub-problems- one of size $n/5^2$ and another of size $4n/5^2$.
- On the other side, sub-problem of size $4n/5$ will get divided into two sub-problems- one of size $4n/5^2$ and another of size $4^2n/5^2$, and so on.
- At the bottom most level, the size of sub-problems will reduce to 1.

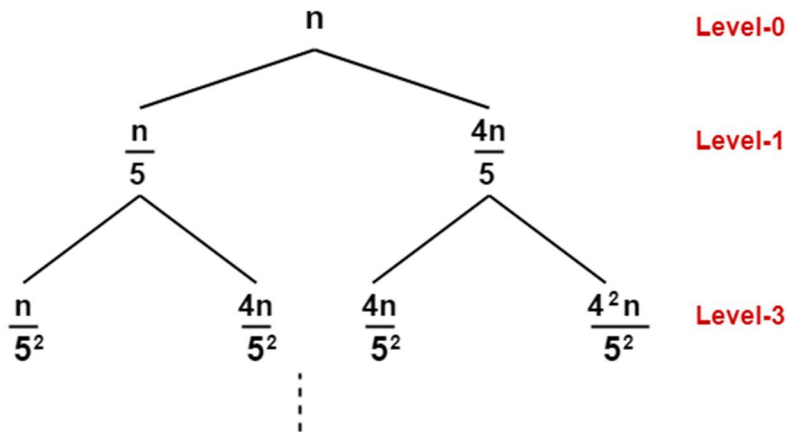
This is illustrated through following recursion tree:



The given recurrence relation shows-

- The cost of dividing a problem of size n into its 2 sub-problems and then combining its solution is n .
- The cost of dividing a problem of size $n/5$ into its 2 sub-problems and then combining its solution is $n/5$.
- The cost of dividing a problem of size $4n/5$ into its 2 sub-problems and then combining its solution is $4n/5$ and so on.

This is illustrated through following recursion tree where each node represents the cost of the corresponding sub-problem:



Step-02:

Determine cost of each level-

- Cost of level-0 = n
- Cost of level-1 = $n/5 + 4n/5 = n$
- Cost of level-2 = $n/5^2 + 4n/5^2 + 4n/5^2 + 4^2n/5^2 = n$

Step-03:

Determine total number of levels in the recursion tree:

We will consider the rightmost subtree (i.e., largest subproblem) as it goes down to the deepest level,

- Size of problem at level-0 = $(4/5)^0n$
- Size of larger sub-problem at level-1 = $(4/5)^1n$
- Size of larger sub-problem at level-2 = $(4/5)^2n$

Continuing in similar manner, we have,

Size of larger sub-problem at level- i = $(4/5)^in$

Suppose at level- x (last level), size of larger sub-problem becomes 1. Then,

$$(4/5)^xn = 1$$

$$n = (5/4)^x$$

Taking log on both sides, we get,

$$x = \log_{5/4}n$$

$$\therefore \text{Total number of levels in the recursion tree} = x+1 = \log_{5/4}n + 1$$

Step-04:

Determine number of nodes in the last level:

- Level-0 has 2^0 nodes i.e., 1 node
- Level-1 has 2^1 nodes i.e., 2 nodes
- Level-2 has 2^2 nodes i.e., 4 nodes

Continuing in similar manner, we have,

Last level (i.e., Level- x =Level - $\log_{5/4}n$) has $2^{\log_{5/4}n}$ nodes = $n^{\log_{5/4}2}$.

Cost of each node in the last level is $T(1)$.

Step-05:

Determine cost of last level:

$$\text{Cost of last level} = n^{\log_{5/4}2} * T(1) = \theta(n^{\log_{5/4}2})$$

Step-06:

Add costs of all the levels of the recursion tree and simplify the expression so obtained in terms of asymptotic notation:

$$T(n) = \underbrace{\{ n + n + n + \dots \}}_{\text{For } \log_{5/4}n \text{ levels}} + \theta(n^{\log_{5/4}2})$$

For $\log_{5/4}n$ levels

$$= n \log_{5/4}n + \theta(n^{\log_{5/4}2})$$

$$= \theta(n \log_{5/4}n).$$

Problem-3:

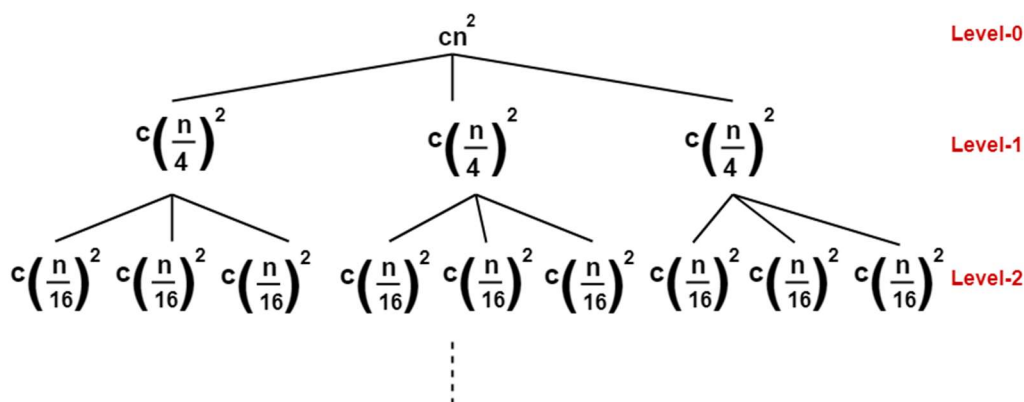
Solve the following recurrence relation using recursion tree method:

$$T(n) = 3T(n/4) + cn^2$$

Solution-

Step-01:

Draw a recursion tree based on the given recurrence relation:



(Here, we have directly drawn a recursion tree representing the cost of sub problems)

Step-02:

Determine cost of each level:

- Cost of level-0 = cn^2
- Cost of level-1 = $c(n/4)^2 + c(n/4)^2 + c(n/4)^2 = (3/16)cn^2$
- Cost of level-2 = $c(n/16)^2 \times 9 = (9/16^2)cn^2$

Step-03:

Determine total number of levels in the recursion tree:

- Size of problem at level-0 = $n/4^0$
- Size of sub-problem at level-1 = $n/4^1$
- Size of sub-problem at level-2 = $n/4^2$

Continuing in similar manner, we have,

Size of sub-problem at level-i = $n/4^i$

Suppose at level-x (last level), size of sub-problem becomes 1. Then,

$$n/4^x = 1$$

$$4^x = n$$

$$x = \log_4 n$$

$$\therefore \text{Total number of levels in the recursion tree} = \log_4 n + 1$$

Step-04:

Determine number of nodes in the last level:

- Level-0 has 3^0 nodes i.e., 1 node
- Level-1 has 3^1 nodes i.e., 3 nodes
- Level-2 has 3^2 nodes i.e., 9 nodes

Continuing in similar manner, we have,

Last level (i.e., level - $\log_4 n$) has $3^{\log_4 n} = n^{\log_4 3}$ nodes.

Step-05:

Determine cost of last level:

$$\text{Cost of last level} = n^{\log_4 3} * T(1) = \theta(n^{\log_4 3})$$

Step-06:

Add costs of all the levels of the recursion tree and simplify the expression so obtained in terms of asymptotic notation:

$$T(n) = \underbrace{\left\{ cn^2 + \frac{3}{16}cn^2 + \frac{9}{(16)^2}cn^2 + \dots \right\}}_{\text{For } \log_4 n \text{ levels}} + \theta(n^{\log_4 3})$$

$$= cn^2 \{ 1 + (3/16) + (3/16)^2 + \dots \} + \theta(n^{\log_4 3})$$

Now, $\{ 1 + (3/16) + (3/16)^2 + \dots + (3/16)^{i-1} \}$ forms Geometric progression, where $i = \log_4 n$.

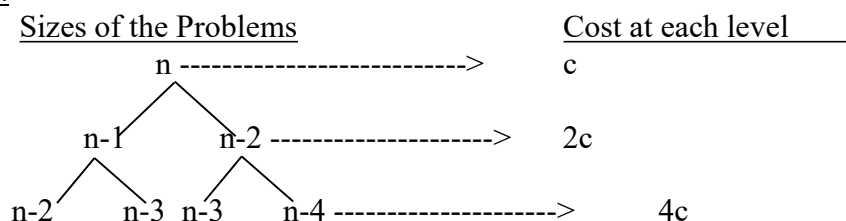
On solving, we get,

$$\begin{aligned}
 &= (16/13)cn^2 (1 - (3/16)^{\log_4 n}) + \theta(n^{\log_4 3}) \\
 &= (16/13)cn^2 - (16/13)cn^2 (3/16)^{\log_4 n} + \theta(n^{\log_4 3}) \\
 &= (16/13)cn^2 - (16/13)cn^2 ((3)^{\log_4 n} / (16)^{\log_4 n}) + \theta(n^{\log_4 3}) \\
 &= (16/13)cn^2 - (16/13)cn^2 (n^{\log_4 3} / n^{\log_4 16}) + \theta(n^{\log_4 3}) \\
 &= (16/13)cn^2 - (16/13)cn^2 (n^{\log_4 3} / n^2) + \theta(n^{\log_4 3}) \\
 &= (16/13)cn^2 - (16/13)c (n^{\log_4 3}) + \theta(n^{\log_4 3}) \\
 &= \mathbf{O(n^2)}
 \end{aligned}$$

Problem-4:

$$T(n) = T(n-1) + T(n-2) + c$$

Solution:



The size of larger subproblem at last level (level x) is $n-x=1$. So, $x=n-1$.

The last level has 2^x nodes. So, the cost of last level is $2^x c$.

$$\begin{aligned}
 \text{The total cost of all levels} &= (c + 2c + 2^2 c + \dots + 2^{x-1} c) + 2^x c \\
 &= c(1 + 2 + 2^2 + \dots + 2^x) \\
 &= c(2^{x+1} - 1) \\
 &= O(2^{x+1}) = O(2^n).
 \end{aligned}$$

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 $f(n) = \theta(n^{\log_b a} \lg n)$, then $T(n) = \theta(n^{\log_b a} \lg n)$.
 (b) If $f(n) = \theta(n^{\log_b a} \lg^k n)$, where $k \geq 0$, 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 $af(n/b) \leq cf(n)$ 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 $af(n/b) = 3(n/4) \lg(n/4) \leq (3/4)n \lg n = cf(n)$ 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.

$$T(n) = \theta(n \lg^2 n).$$

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[1: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 //If there is at least one element in the given part of array
    {
        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 return 0.

    if (low > high) then //If there are no elements in the given part of array
        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:

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

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 $T(l)=c$, which happens when $2^k = n$ which implies $k=\log_2 n$

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

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

$= O(\log n)$ --- Worst-case time complexity

The best-case time complexity is $O(1)$.

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 in the given part of array*

 $\{$
$$\text{mid} := \lfloor (\text{low} + \text{high}) / 2 \rfloor;$$

```
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;
        }
    }
}

```

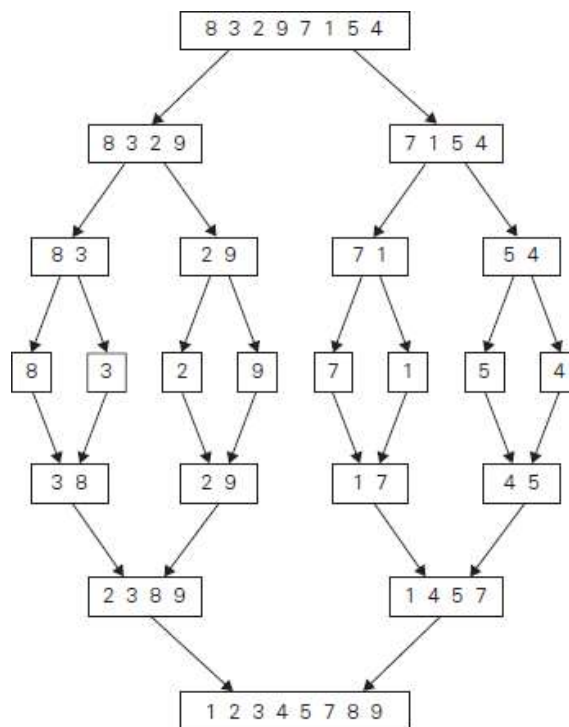
```

    }
}
for k: = low to high do
    a[k]:=b[k]; // copy the elements of b into a.
} // 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,

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

Solving by using substitution method,

$$\begin{aligned}
T(n) &= 2T(n/2) + cn \\
&= 2[2T(n/4) + cn/2] + cn = 4T(n/4) + cn + cn = 4T(n/4) + 2cn \\
&= 4[2T(n/8) + cn/4] + 2cn = 8T(n/8) + 3cn = 2^3T(n/2^3) + 3c
\end{aligned}$$

After applying this substitution k times,

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

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,

$$T(n) = nT(1) + c \cdot \log_2 n \cdot n$$

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

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

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

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

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

→ 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:

65, 70, 75, 80, 60, 55, 50, 45.

| | | | | | | | | | |
|-----------|-----------|-----------------|----|----|----|----|----|-----------------|---|
| Index: | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| Elements: | <u>65</u> | <i>lp</i> 70 | 75 | 80 | 60 | 55 | 50 | <i>rp</i> 45 | ∞ |
| | <u>65</u> | <i>lp</i> 45 | 75 | 80 | 60 | 55 | 50 | <i>rp</i> 70 | ∞ |

| | | | | | | | | |
|------------------|---------------------|-----------------|-----------------|-----------------|-------------------------------|---------------------|-----------------|-----------------------|
| <u>65</u> | 45 | <i>lp</i> 75 | 80 | 60 | 55 | <i>rp</i> 50 | 70 | ∞ |
| <u>65</u> | 45 | <i>lp</i> 50 | 80 | 60 | 55 | <i>rp</i> 75 | 70 | ∞ |
| <u>65</u> | 45 | 50 | <i>lp</i> 80 | 60 | <i>rp</i> 55 | 75 | 70 | ∞ |
| <u>65</u> | 45 | 50 | <i>lp</i> 55 | 60 | <i>rp</i> 80 | 75 | 70 | ∞ |
| <u>65</u> | 45 | 50 | 55 | <i>rp</i> 60 | <i>lp</i> 80 | 75 | 70 | ∞ |
| <u>60</u> | 45 | 50 | 55 | 65 | 80 | 75 | 70 | ∞ |
| <u>60</u> | <i>lp</i> 45 | 50 | <i>rp</i> 55 | | | | | ∞ |
| <u>60</u> | 45 | 50 | <i>rp</i> 55 | <i>lp</i> | | | | ∞ |
| <u>55</u> | 45 | 50 | 60 | | | | | ∞ |
| <u>55</u> | <i>lp</i> 45 | <i>rp</i> 50 | | | | | | ∞ |
| <u>55</u> | 45 | <i>rp</i> 50 | <i>lp</i> | | | | | ∞ |
| <u>50</u> | 45 | 55 | | | | | | ∞ |
| <u>50</u> | <i>lp, rp</i> 45 | | | | | | | ∞ |
| <u>50</u> | <i>rp</i> 45 | <i>lp</i> | | | | | | ∞ |
| 45 | 50 | | | | | | | ∞ |
| | | | | | <u>80</u> | <i>lp</i> 75 | <i>rp</i> 70 | ∞ |
| | | | | | <u>80</u> | 75 | <i>rp</i> 70 | <i>lp</i> ∞ |
| | | | | | <u>70</u> | 75 | 80 | ∞ |
| | | | | | <u>70</u> | <i>lp, rp</i> 75 | | ∞ |
| | | | | | <i>rp</i> <u>70</u> | <i>lp</i> 75 | | ∞ |

| | | | | | | | |
|----|----|----|----|----|----|----|----------|
| | | | | | 70 | 75 | ∞ |
| 45 | 50 | 55 | 60 | 65 | 70 | 75 | 80 |

So, the sorted data is: 45, 50, 55, 60, 65, 70, 75, 80.

Time Complexity:

Quick Sort partitions the given set of elements S into two subsets S1 and S2 around pivot element t . That means, $S = S1 \cup \{t\} \cup S2$.

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,

$$T(n) = T(n-k) + cn + c(n-1) + c(n-2) + \dots + c(n-(k-1))$$

$$= T(n-k) + c(n + (n-1) + (n-2) + \dots + (n-(k-1)))$$

Assume $n=k$,

$$T(n) = T(0) + c(k + (k-1) + (k-2) + \dots + 1)$$

$$= c + c(1 + 2 + \dots + k)$$

$$= c + c(k(k+1)/2)$$

$$= c + c(n(n+1)/2) \quad T(n) = O(n^2).$$

(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 substitution method,

$$\begin{aligned}
T(n) &= 2T(n/2) + cn \\
&= 2[2T(n/4) + cn/2] + cn = 4T(n/4) + cn + cn = 4T(n/4) + 2cn \\
&= 4[2T(n/8) + cn/4] + 2cn = 8T(n/8) + 3cn = 2^3T(n/2^3) + 3c
\end{aligned}$$

After applying this substitution k times,

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

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,

$$T(n) = nT(1) + c \cdot \log_2 n \cdot n$$

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

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

(iii) Average-Case Analysis: (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

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

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

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

From (2) we can get

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

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

$$nT(n) = (n+1) T(n-1) + 2nc \text{ ----- (4)}$$

Dividing (4) by $n(n+1)$, we get

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2c}{n+1} \text{ ----- (5)}$$

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,

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

When $n=k$,

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

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

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

$$\begin{aligned} \frac{T(n)}{n+1} &= c + 2c(\log_e(n+1) + \gamma - 1) \\ &= c + 2c\log_e(n+1) + 2c(\gamma - 1) \\ &= c + 2c\log_e(n+1) \end{aligned}$$

$$\begin{aligned} T(n) &= (n+1)(c + 2c\log_e(n+1)) \\ &= c(n+1) + 2c(n+1)\log_e(n+1) \\ &= c(n+1) + 2cn\log_e(n+1) + 2c\log_e(n+1) \end{aligned}$$

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

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.

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

→ 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 an approximate (or, near optimal) solution.

CONTROL ABSTRACTION FOR GREEDY METHOD:

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); *// 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, \dots, 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 $x_i=1, 1 \leq i \leq n$.

POSSIBLE GREEDY STRATEGIES:

There are several greedy 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 | $\frac{2}{15}$ | 1 | 0 |

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

$$(x_1, x_2, x_3) = \left(\frac{2}{15}, 1, 0\right)$$

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 $24 + 15 \cdot \frac{1}{2} = 24 + 7.5 = 31.5$.

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 $(24 \times \frac{2}{3}) + (25 \times 0) + (15 \times 1) = 16 + 0 + 15 = 31$.

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 $24 + 15 \cdot \frac{1}{2} = 24 + 7.5 = 31.5$.

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 [1: n] and w [1: n] contain the profits and weights respectively of n
//objects which are ordered such that  $p[i] \geq \frac{p[i+1]}{w[i]} \cdot \frac{w[i+1]}{w[i]}$ 
// 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 to sort the objects according to their profit densities, the time

complexity of this algorithm $O(n)$.

Exercises:

1) $n=7$, $M = 15$, $(p_1, p_2, p_3, p_4, p_5, p_6, p_7) = (10, 5, 15, 7, 6, 18, 7)$, and $(w_1, w_2, w_3, w_4, w_5, w_6, w_7) = (2, 3, 5, 7, 1, 4, 1)$.

2) $n=7$, $M = 15$, $(p_1, p_2, p_3, p_4, p_5, p_6, p_7) = (10, 5, 15, 7, 6, 18, 3)$, and $(w_1, w_2, w_3, w_4, w_5, w_6, w_7) = (2, 3, 5, 7, 1, 4, 1)$.

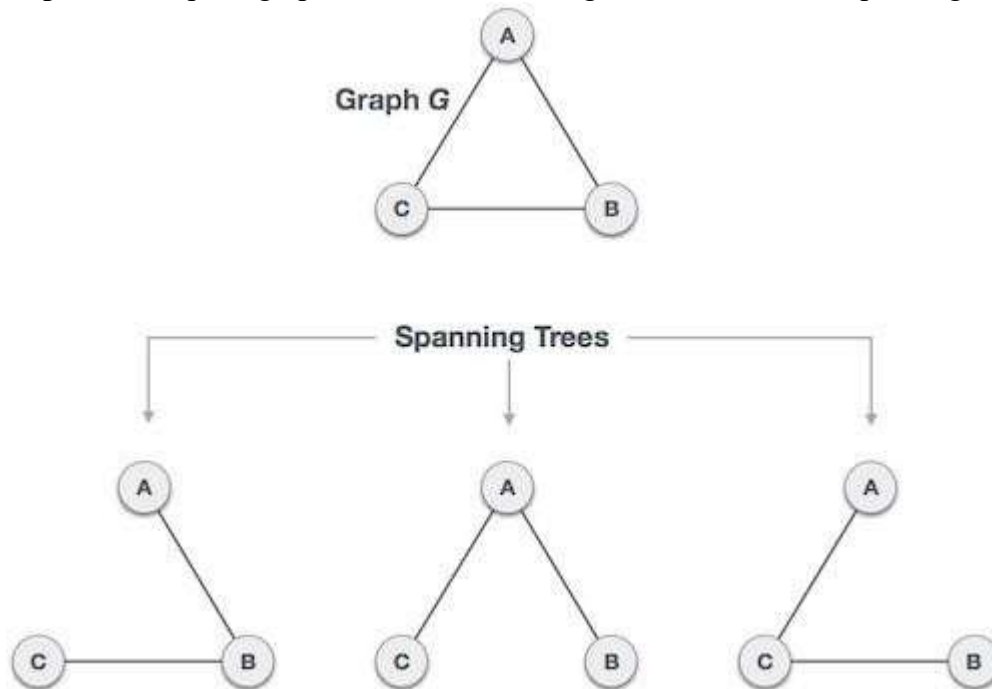
SPANNING TREE:

→ 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 n^{n-2} number of spanning trees.

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

→ Given a weighted connected graph G , 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. Prim's Algorithm
2. Kruskal's algorithm

PRIM'S ALGORITHM:

→ It is a greedy algorithm to obtain a minimal cost spanning tree.

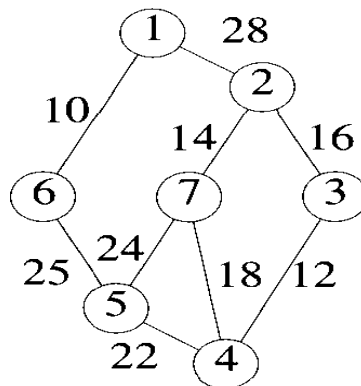
→ It builds MST edge by edge.

→ 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.”

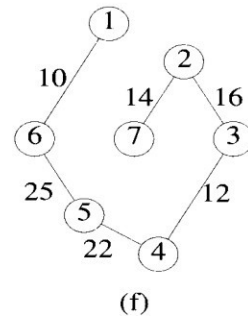
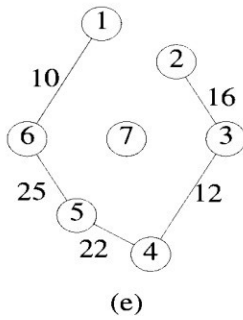
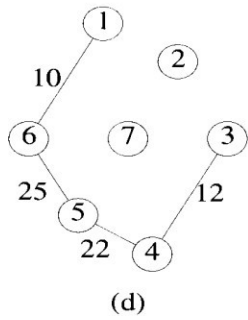
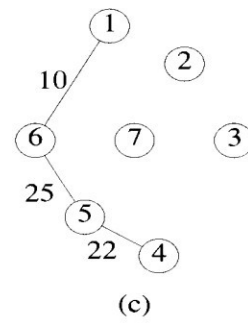
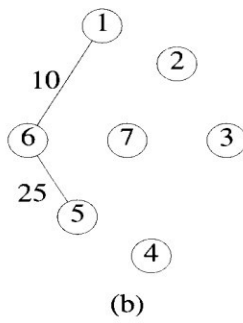
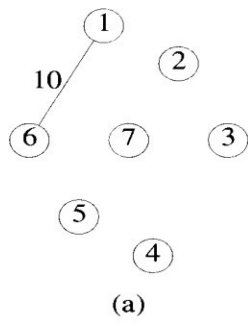
→ That means, among the edges which are incident on vertices that are selected so far, select the one with minimum cost, provided that the inclusion of it will not result in a cycle.

→ 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.

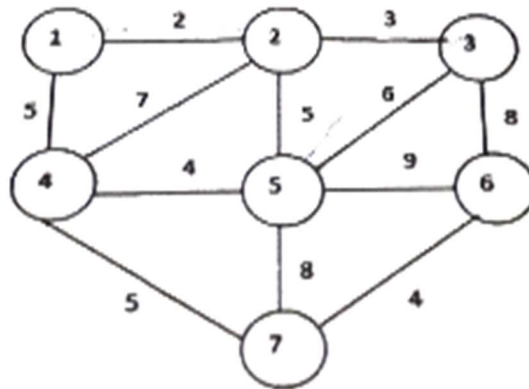


Stages in Prim's algorithm:



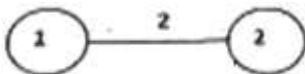
Cost of minimum spanning tree is = $10 + 25 + 22 + 12 + 16 + 14 = 99$.

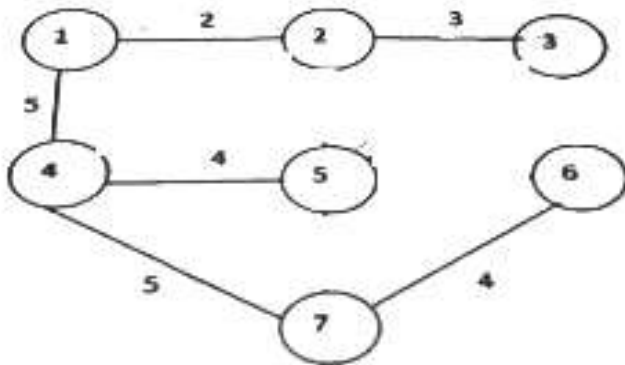
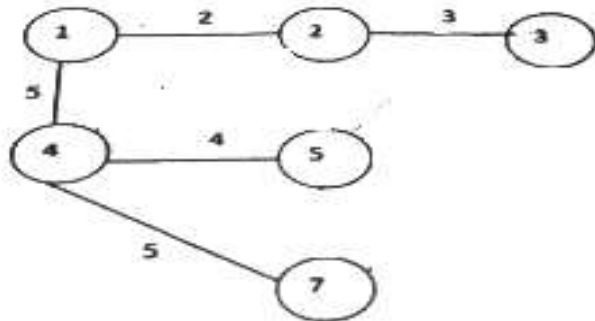
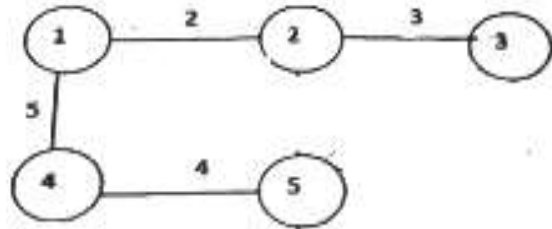
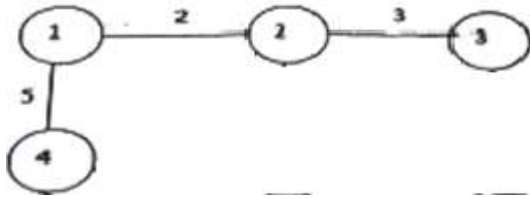
Example: Apply Prim's algorithm on the following graph to find MST.



Solution:

Steps in constructing MST:





The cost of MST = $2+3+5+4+5+4 = 23$.

Note: More than one MST is possible but cost must be same.

ALGORITHM:

Algorithm Prim(E, cost, n, t)

// E is the set of edges in G .

// $\text{cost}[1 : n, 1 : n]$ is the cost adjacency matrix of an n vertex graph such that

// $\text{cost}[i,j]$ 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 ith 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[ ] array.
        if (cost[i,l] < cost[i,k]) then // If l is nearer to i than k
            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.

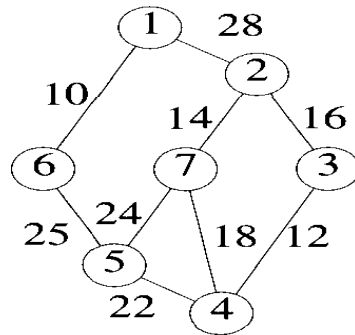
→It also builds MST edge by edge.

→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."

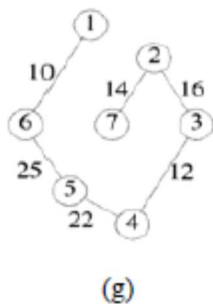
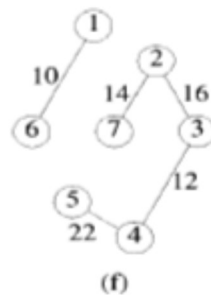
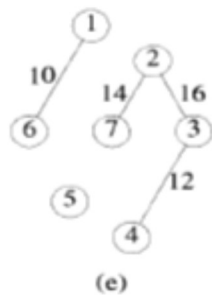
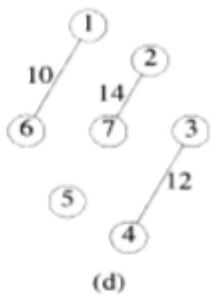
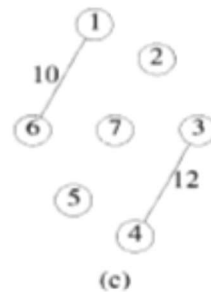
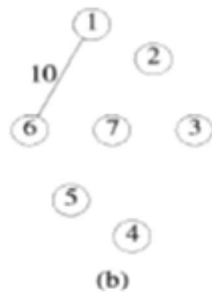
→In other words, the edges are selected in increasing order of weights provided cycle will not be created at any stage.

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. But at the end, a tree will 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:

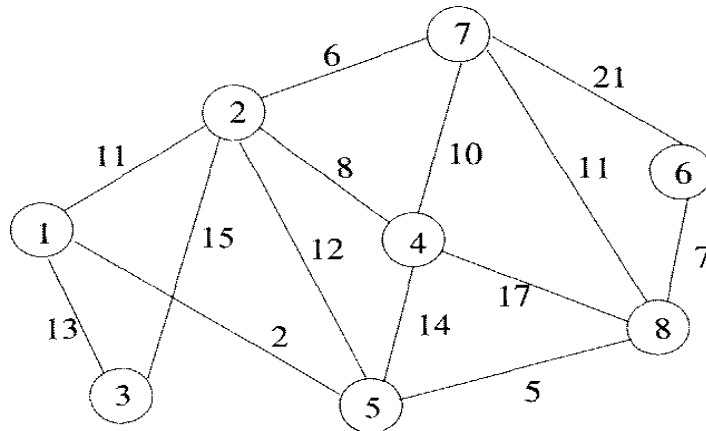


Cost of minimum spanning tree = $10 + 12 + 14 + 16 + 22 + 25 = 99$.

Simple version of Kruskal's algorithm:

```
t =  $\Phi$ ;  
while ((t has less than  $n - 1$  edges) and ( $E \neq \Phi$ )) 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 generated.

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

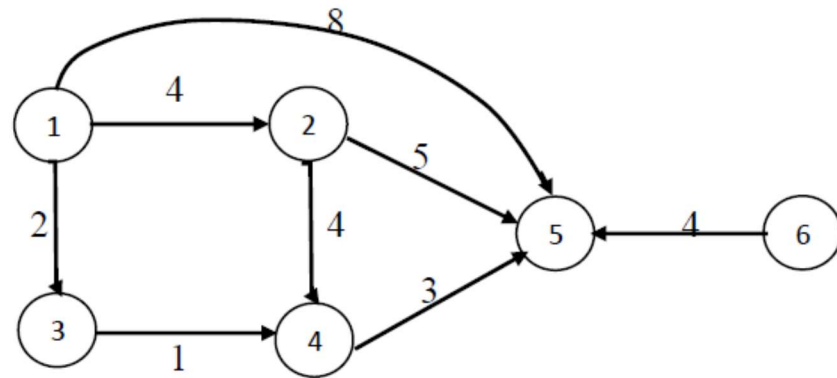
“From the set of vertices, to which a shortest path has not been generated, select one that results in the least path length (or, one that is closest to the source vertex).”

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

→ We represent an n vertex graph by an $n \times n$ cost adjacency matrix $cost$, with

$cost[i, j]$ being the weight of the edge $\langle i, j \rangle$. In case the edge $\langle i, j \rangle$ 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-negative number such as 0.

Example:



Cost adjacency matrix:

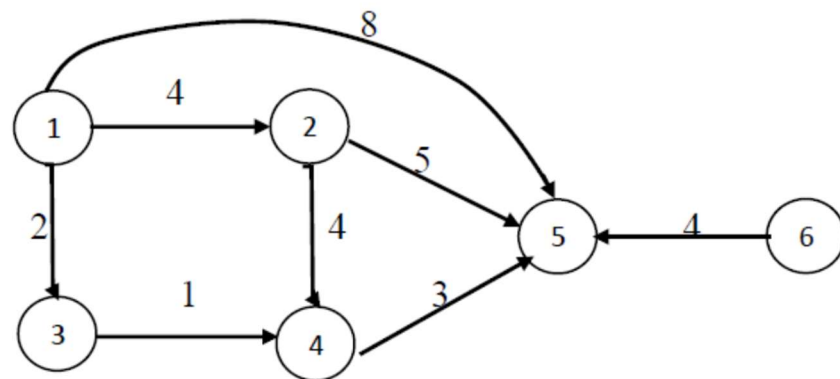
$$\begin{bmatrix} 0 & 4 & 2 & \infty & 8 & \infty \\ \infty & 0 & \infty & 4 & 5 & \infty \\ \infty & \infty & 0 & 1 & \infty & \infty \\ \infty & \infty & \infty & 0 & 3 & \infty \\ \infty & \infty & \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & \infty & 4 & 0 \end{bmatrix}$$

→ 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: Obtain 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:

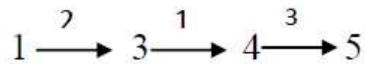
$1 \xrightarrow{2} 3 \xrightarrow{1} 4$

(ii) Finding the next shortest path

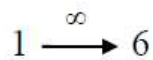
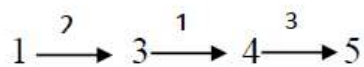
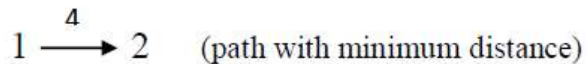
$1 \xrightarrow{4} 2$
 $1 \xrightarrow{2} 3 \xrightarrow{1} 4$ (path with minimum distance)
 $1 \xrightarrow{8} 5$
 $1 \xrightarrow{\infty} 6$

Step 3:

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



(ii) Finding the next shortest path

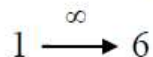
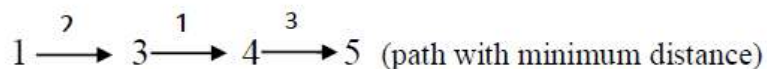


Step 4:

(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

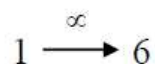


Step 5:

(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

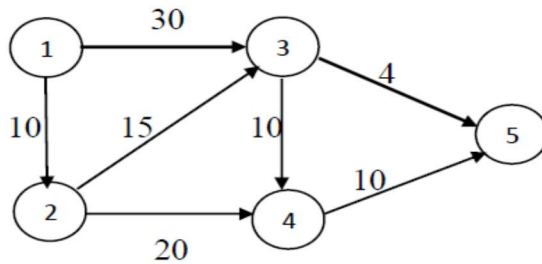


There is no path to node 6.

Shortest paths in increasing order:

| <u>Shortest Paths</u> | <u>Lengths</u> |
|---|----------------|
| 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], $1 \leq i \leq n$, 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];
        }
    }
}
}

```

Algorithm to find out paths also:

Algorithm ShortestPaths(v , cost, dist, predecessor, n)

//graph G with n vertices is represented by its cost adjacency matrix $cost[1:n, 1:n]$.

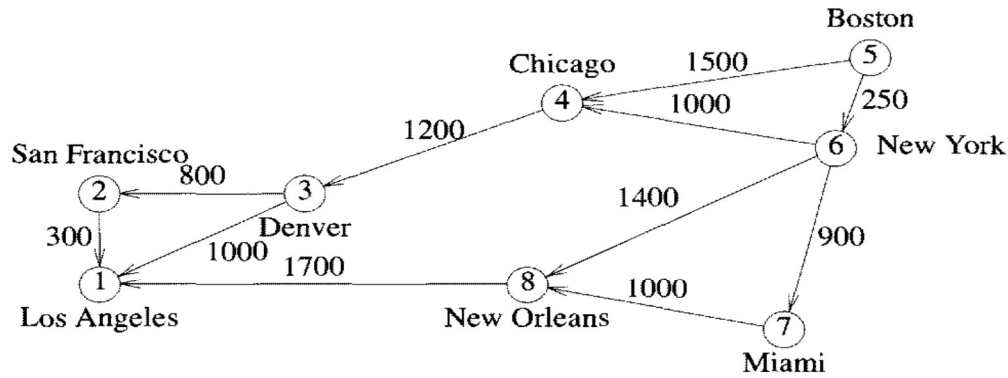
//dist[i], $1 \leq i \leq n$, is set to the length of the shortest path from source vertex v to the vertex i .

```

{
    for i:= 1 to n do
    {
        S[i]:= 0;                // Initialize S[ ] array
        dist[i]:= cost[v, i];    //Initialize dist[ ] array
        if(i is adjacent to v) then
            predecessor[i]:= v;

    }
    S[v]:= 1;                //put v in S
    predecessor[v] :=0;      // source vertex has no predecessor
    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 distance to w.
            if (dist[w] > (dist[u] + cost[u, w])) then
            {
                dist[w]:= dist[u] + cost[u, w];
                predecessor[w]:= u;
            }
        }
    }
}
}

```



(a) Digraph

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|------|-----|------|------|---|-----|-----|------|
| 1 | 0 | | | | | | | |
| 2 | 300 | 0 | | | | | | |
| 3 | 100 | 800 | 0 | | | | | |
| 4 | | | 1200 | 0 | | | | |
| 5 | | | | 1500 | 0 | 250 | | |
| 6 | | | | 1000 | | 0 | 900 | 1400 |
| 7 | | | | | | | 0 | 1000 |
| 8 | 1700 | | | | | | | 0 |

(b) Length-adjacency matrix

Actions for Finding Shortest Paths from vertex 5:

| Iteration | S | Distance to vertices | | | | | | | | Vertex Selected | Predecessor for selected vertex | Path | Path Length |
|-----------|-----------------|----------------------|----------|----------|------|-----|-----|----------|----------|-----------------|---------------------------------|-----------|-------------|
| | | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | | | | |
| 1 | {5} | ∞ | ∞ | ∞ | 1500 | 0 | 250 | ∞ | ∞ | 6 | 5 | 5→6 | 250 |
| 2 | {5,6} | ∞ | ∞ | ∞ | 1250 | - | - | 1150 | 1650 | 7 | 6 | 5→6→7 | 1150 |
| 3 | {5,6,7} | ∞ | ∞ | ∞ | 1250 | - | - | - | 1650 | 4 | 6 | 5→6→4 | 1250 |
| 4 | {5,6,7,4} | ∞ | ∞ | 2450 | - | - | - | - | 1650 | 8 | 6 | 5→6→8 | 1650 |
| 5 | {5,6,7,4,8} | 3350 | ∞ | 2450 | - | - | - | - | - | 3 | 4 | 5→6→4→3 | 2450 |
| 6 | {5,6,7,4,8,3} | 3350 | 3250 | - | - | - | - | - | - | 2 | 3 | 5→6→4→3→2 | 3250 |
| 7 | {5,6,7,4,8,3,2} | 3350 | - | - | - | - | - | - | - | 1 | 8 | 5→6→8→1 | 3350 |

