# Module - 2

Syllabus :-

Brute Force Approaches : Exhaustive Search (Travelling Salesman problem and Knapsack problem)

Decrease and Conquer : Insertion sort, Topological sorting

Divide and Conquer : Merge sort, Quick sort, Binary Tree Traversals, Multiplication of large integers and Strassen's Matrix multiplication

## Divide and Conquer :-

- It is one of the best known method of solving a problem
- It is a top-down technique for designing algorithm which consist of dividing the problem into smaller sub problems hoping that the solution of the sub problem are easier to find. Solution of all the smaller problem are then combined to get a solution for the original problem.
- Divide and conquer technique of solving a problem involves 3 steps at each level.

1) Divide :- Problem is divided into number of sub problems

2) Conquer :- If the sub problems are smaller in size, problem can be solved using straight forward method. If the sub problems are larger in size they are divided into number of sub problem of same type and size.

3) Combine :- Solution of sub-problem are combined to get the solution for the larger problem.

- Divide and conquer technique is shown in figure which depicts the case of dividing a problem into 2 smaller subproblems, by far the most widely occuring case.

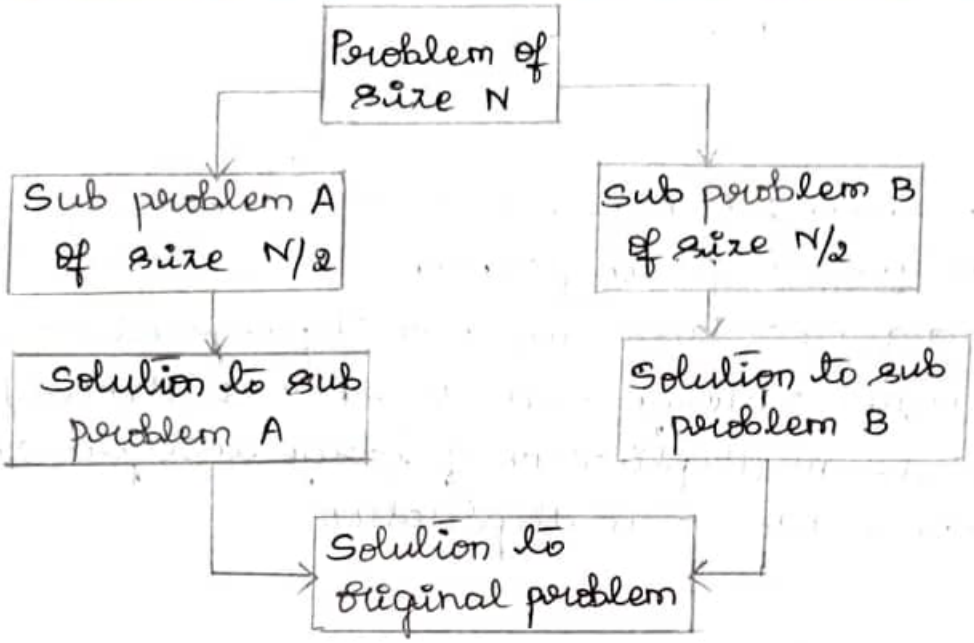



Fig :- Divide and conquer technique

- In the most typical case of divide and conquer, a problem's instance of size 'n' is divided into 2 instances of size $n/2$.
- An instance of size 'n' can be divided into 'b' instances of size $n/b$, with 'a' of them needing to be solved. Here 'a' and 'b' are constants such that $a \geq 1$ and $b > 1$
- Assuming that size 'n' is a power of 'b', to simplify our analysis, we get the following recurrence for the running time $T(n)$:

$$T(n) = a\,T(n/b) + f(n)$$

where,

$f(n)$ → function that accounts for the time spent on dividing the problem into smaller instances and on combining their solutions.

$a$ → Number of times problem is divided

- Above recurrence is called the general method for divide and conquer recurrence.
- Order of growth of the function of its solution $T(n)$ depends on the values of the constants 'a and b' & the order of growth of the function $f(n)$.
- The efficiency analysis of many divide and conquer algorithms is simplified using Master Theorem.

## Master Theorem :-

If $f(n) \in \theta(n^d)$ with $d \geq 0$ in recurrence equation

$T(n) = a\,T(n/b) + f(n)$, then

$$T(n) \in \begin{cases} \theta(n^d) & \text{if } a < b^d \\ \theta(n^d \log n) & \text{if } a = b \\ \theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

Where, d is the power of n in $f(n)$.

## Merge Sort :-

- Merge sort is based on the principle of "Divide and conquer" and it works very well on large set of data.
- Key operation in merge sort is combining the sorted left part and sorted right part into a single sorted array.
- Process of merging of 2 sorted vectors into a single sorted vector is called simple merge.
- The only necessary condition for this problem is that both arrays should be sorted.

## Design of merge sort :-

Consider an array 'a' consisting of integer number 40, 80, 10, 50, 30, 20, 70, 60.

Index low → specifies the position of 1st element in array

Index high → specifies the position of last element in array

Sorting can be done as shown

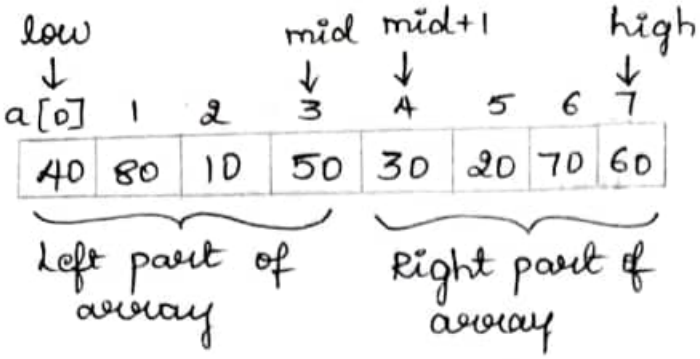
**Step-1 :-** If (low > high)

It indicates that no elements are present in array. So, simply return the control to calling function as

if (low > high) return 0

**Step-2 :-** Divide the given array of elements into 2 equal parts as

mid = (low + high) / 2



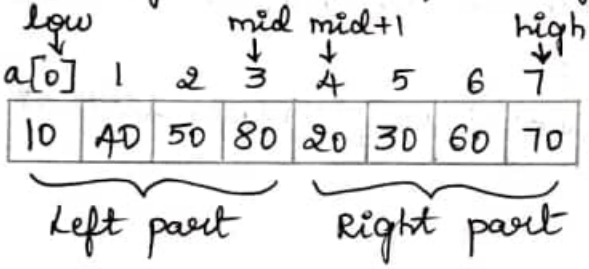

Step-3 :- Left part of array 'a' starts from index low to mid. Now, left part is sorted recursively using the statement

Mergesort (a, low, mid)

Step-4 :- Right part of array 'a' starts from index mid+1 to high. Right part is sorted recursively using the statement.

Mergesort (a, mid+1, high)

- After sorting contents of array can be written as




- Left part is sorted from low to mid and right part is also sorted from mid+1 to high. But the entire contents of the array from low to high is not sorted.

Step-5 :- To sort the array, it is necessary to merge the left part and right part of the array using the statement

Simple Merge (a, low, mid, high)

Algorithm :-

Algorithm Mergesort (a, low, high)
// sort the elements of the array between lower bound and upper bound.
// Input :- A is an unsorted array with low and high as lower bound and upper bound.

```
// output :- A is a sorted array.
if (low > high) return;
mid ← (low+high)/2
Mergesort (a, low, mid)
Mergesort (a, mid+1, high)
Simple Merge (a, low, mid, high)
```

Design of simple merge algorithm :-

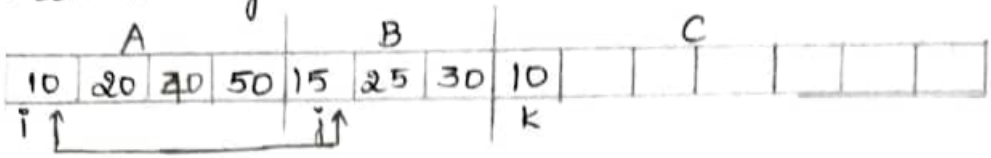
- We have 2 sorted vectors A and B with 'm' and 'n' elements. Following procedure is used to merge these 2 sorted vectors
- Compare $i^{th}$ element of vector A with $j^{th}$ element of vector B and copy the lesser element into $K^{th}$ position of resultant vector C.

```
if (A[i] < B[j]) then
    C[K] ← A[i]
  K ← K+1 , i ← i+1
else
    C[K] ← B[j]
  K ← K+1, j ← j+1
end if
```

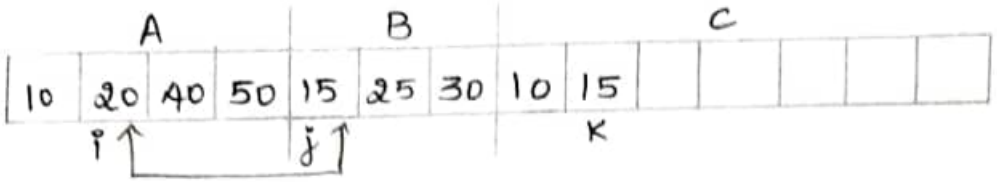
- While updating i, the index i should not exceed 'm' and while updating j, the index j should not exceed 'n'. The above statements can be written as

```
while (i<m and j<n)
if (A[i] < B[j]) then
    C[k] ← A[i]
  K ← K+1, i ← i+1
else
    C[K] ← B[j]
  K ← K+1, j ← j+1
end if
end while
```

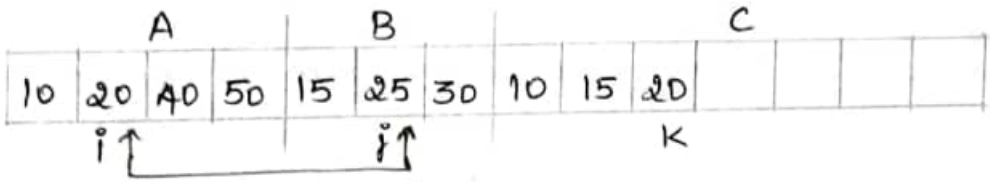
Consider the 2 sorted vectors A containing 4 elements & B containing 3 elements.



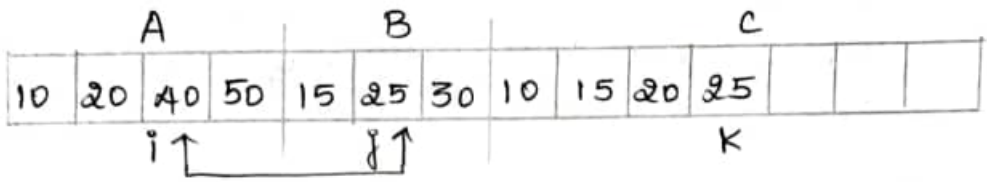

10 < 15, 10 is copied to C. Update i & K



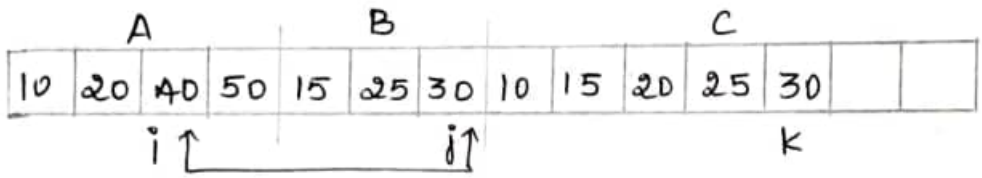

15 < 20, 15 is copied to C. update j & K




20 < 25, 20 is copied to C. Update i & K

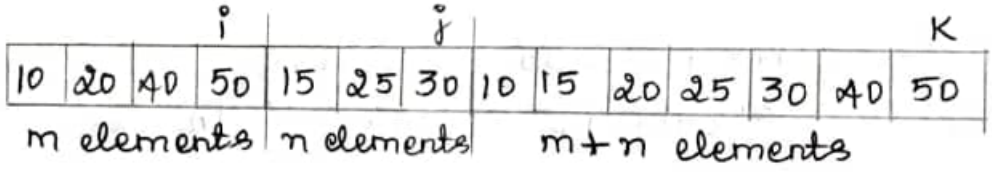



25 < 40, 25 is copied to C. update j & K




30 < 40, 30 is copied to C. Update j & K

Since there are no elements in B to compare, the remaining elements of A can be copied to C. Now 10, 15, 20, 25, 30, 40, 50 are sorted in vector C.



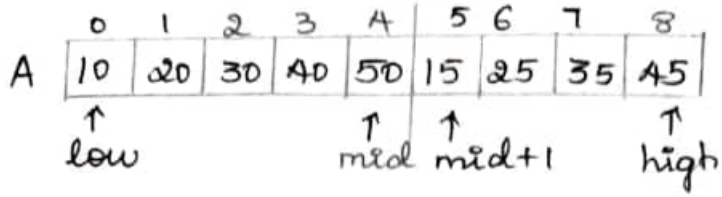

copy remaining elements from A to C.

In general, when either i or j exceeds its limits, the elements remaining in other vector can be copied into vector C using the following statements.

```
while (i<m)
C[k] ← A[i]
  K← K+1, i←i+1
end while
```

```
while (j<n)
C[k]← B[j]
  K← K+1, j←j+1
end while.
```

Consider a situation of a single vector which is divided into 2 sorted parts as shown




In the above array,

- Elements from low to mid are sorted
- Elements from mid+1 to high are also sorted
- But, if we take elements from low to high the elements are not sorted.
- Algorithm to merge 2 sorted vectors is as shown.

Algorithm :-

```
Algorithm Simple Merge (a, low, mid, high)
// Merge 2 sorted arrays where 1st array stands from low to
   mid and 2nd array starts from mid+1 to high.
// Input :- A is sorted from position low to mid
            A is sorted from position mid+1 to high
// Output :- A is sorted from position low to high.
    i ← low
    j ← mid+1
    k ← low
  while (i <= mid and j <= high)
   if (A[i] < A[j]) then
     C[k] ← A[i]
     i ← i+1,  k ← k+1
   else
     C[k] ← A[j]
     j ← j+1,  k ← k+1
   end if
  end while
```

```
while (i<=mid)
  C[k] ← A[i]
  k ← k+1, i ← i+1
end while
while (j<=high)
  C[k] ← A[j]
  k ← k+1, j ← j+1
end while
for i = low to high
  A[i] ← C[i]
end for
```

Analysis :-

- Problem instance is divided into 2 equal parts. Assume that number of elements in the array 'a' are integral multiplies of 2. Recurrence relation can be written as

$$T(n) = \begin{cases} 0 & \text{if } (n=1) \\ T(n/2) + T(n/2) + n & \text{otherwise} \end{cases}$$

$T(n/2)$ : Time required to sort left part

$T(n/2)$ : Time required to sort right part

$n$ : Time required to divide & merge solution with 'n' elements

Above relation can be written as

$$T(n) = \begin{cases} 0 & \text{if } (n=1) \\ 2T(n/2) + Cn & \text{otherwise} \end{cases}$$

Time complexity using master theorem :-

General recurrence relation for divide and conquer method is $T(n) = aT(n/b) + f(n)$

whereas recurrence relation for merge sort is

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

where $a=2$, $b=2$ and $f(n) = Cn = Cn^1 = Cn^d$ $\therefore d=1$

substituting these values in recurrence relation

$$T(n) = \begin{cases} \theta(n^d) & \text{if } a < b^d \\ \theta(n^d \log_b n) & \text{if } a = b^d \\ \theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

We get, $a = 2$ and $b^d = 2^1 = 2$.

$\therefore a = b^d$

$T(n) = \theta(n^d \log_b n)$

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

Time complexity using substitution method :-

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

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

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

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

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

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

⋮

In general,

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

To get the initial condition $T(1) = 0$, let $2^i = n$ — ①

$= n.T\left(\frac{n}{n}\right) + i.n$

$= n.T(1) + i.n$

$= n.0 + i.n$

$T(n) = i.n$ — ②

$T(n) = \log_2 n . n$

$T(n) = n \log_2 n \Rightarrow T(n) \in \theta(n \log_2 n)$

From eqn ①

$2^i = n$, take log on both sides

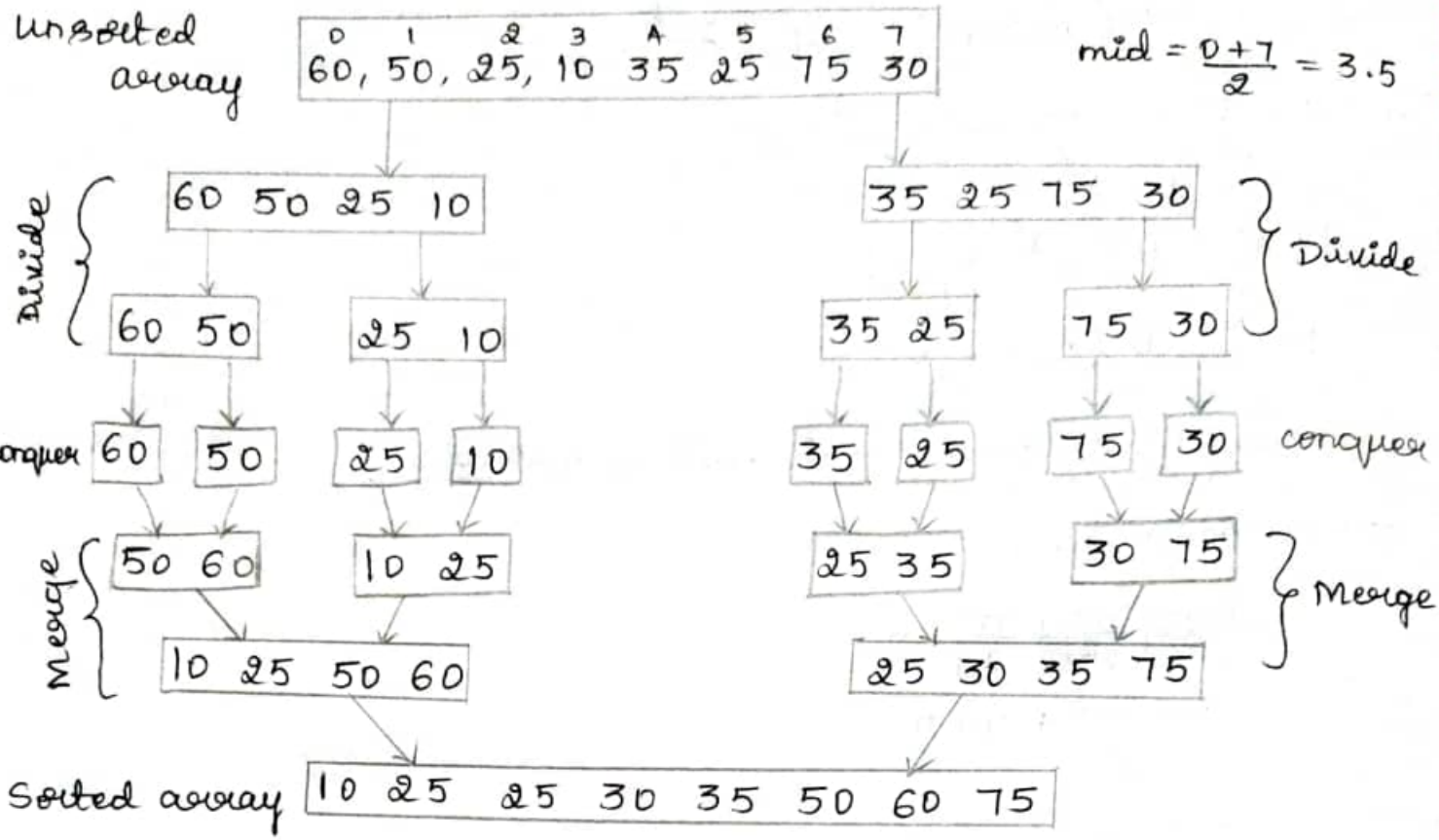
$\log 2^i = \log n$

$i \log_2 2 = \log_2 n$

$i = \log_2 n \rightarrow$ substitute in eqn ②

Eg :- Sort the following elements using merge sort

60, 50, 25, 10, 35, 25, 75, 30




## Quick Sort :-

- Quick sort is based on the principle of divide and conquer, and it works very well on large set of data.
- Quick sort algorithm picks an element as a pivot (key) and partitions the given array around the picked pivot (key) by placing the pivot (key) in its correct position in the sorted array.
- Key process in quicksort is a partition. The target of partition is to place the pivot at its correct position in the sorted array and put all smaller elements to the left of the pivot and all greater elements to the right of the pivot.
- Partition is done recursively on each side of the pivot after the pivot is placed in its correct position and this finally sorts the array.
- Always pick the first element as a pivot (key).

## Design of Quick sort :-

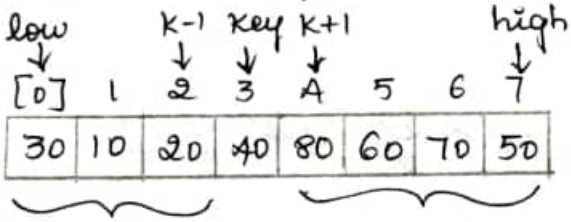
- Consider an array 'a' consisting of integer numbers
  40, 80, 10, 50, 30, 20, 70, 60
- Index low gives the position of 1st element in the array
- Index high gives the position of last element in the array

Sorting can be done as shown.

**Step-1 :-** If (low > high) indicates that there are no elements present, so simply return the control to the calling function

```
if (low > high) return
```

**Step-2 :-** Divide the given array of elements into 2 parts such that elements towards left of key element are less than key element and elements towards right of key element are greater than key element as shown




Element towards left of key are less than key

Element towards right of key are greater than key

- Assume there is a function partition which accepts array 'a', low and high as the parameters and divides the array into 2 parts and return the position of key element. So, the array is partitioned into 2 parts using the code

```
K ← partition (a, low, high)
```

**Step-3 :-** Left part of array 'a' starts from index low to K-1. Sort the left part recursively using the statement

```
Quicksort (a, low, K-1)
```

Step-4 :- Right part of array 'a' starts from index K+1 to high. Sort the right part recursively using the statement

Quicksort (a, K+1, high)

Algorithm :-

Algorithm Quicksort (a, low, high)
// Sort the elements of the array using quick sort
// Input :- Low - position of 1st element of array 'a'
High - position of the last element of array 'a'
a → It is an array consisting of unsorted elements
//Output :- Array consisting of sorted elements.

```
if (low > high) return
K ← partition (a, low, high)
Quicksort (a, low, K-1)
Quicksort (a, K+1, high)
```

Partition the array into 2 parts for quick sort :-

Design :-

- How to divide the array into parts such that element towards left of key element are less than key element and elements towards right of key element are greater than key element.
- Assume that a very large value is stored at the end of the array. This is achieved by storing ∞ in a[n]. Apart from a low and high the other variables that are used are

i → Initial value of index 'i' is low i.e i = low
j → Initial value of index 'j' is one more than high
i.e, j = high+1

Key → a[low] is treated as the key element.

General procedure to partition the array

- Keep incrementing index 'i' as long as key ≥ a[i], using the statement

  do i ← i+1 while (key >= a[i])
- Once the above condition fails, keep decrementing index 'j' as long as key ≤ a[j]

  do j ← j-1 while (key < a[j])
- Once the above condition fails, if 'i' is less than 'j' then exchange a[i] with a[j] and repeat all the above process as long as i <= j
- Finally, exchange a[low] with a[j] and final value of 'j' gives the position of the key element.

Algorithm :-

```
Algorithm partition (a, low, high)
// Divide the array into 2 parts
// Input :- low ← position of 1st element of array
            high ← position of last element of array
            a ← array consisting of unsorted elements
// Output :- Partitioned array such that elements towards left of
             key are ≤ key element and elements towards right of
             key are ≥ key element
 key ← a[low]
 i ← low
 j ← high+1
 while (i<=j)
 do i ← i+1 while (key >= a[i])
 do j ← j+1 while (key < a[j])
  if (i<j) Exchange (a[i], a[j])
 end while
 Exchange (a[low], a[j])
 return j
```

## Analysis :-

1) Best case time efficiency :-

- Occurs, if the key element is present exactly at the center dividing the array into 2 equal parts. The recurrence relation can be written as

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ T(n/2) + T(n/2) + n & \text{otherwise} \end{cases}$$

- $T(n/2)$: Time required to sort left part of array
- $T(n/2)$: Time required to sort right part of array
- $n$: Time required to partition 'n' elements

Using master theorem :-

General recurrence relation for divide and conquer method is

$$T(n) = aT(n/b) + f(n) \quad \text{---} \quad (1)$$

whereas the recurrence relation for quicksort is

$$T(n) = 2T(n/2) + n \quad \text{---} \quad (2)$$

Comparing (1) & (2) we have

$a = 2, \; b = 2 \; d = 1 \qquad f(n) = n = n^1 = n^d$

$a = 2$ and $b^d = 2^1 = 2$

$\therefore a = b^d$

$T(n) = \theta(n^d \log_b n)$

$T(n) = \theta(n \log_b n)$

$T(n) = \theta(n^d)$ if $a < b^d$

$T(n) = \theta(n^d \log_b n)$ if $a = b^d$

$T(n) = \theta(n^{\log_b a})$ if $a > b^d$

Using substitution method :-

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

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

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

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

$$\vdots$$

In general,

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

To get the initial condition $T(1) = 0$ let $2^i = n$ — (1)

$$T(n) = n . T\left(\frac{n}{n}\right) + i.n$$

$$= n . T(1) + i.n$$

$$T(n) = i.n \quad \text{— (2)}$$

From eqn (1), $2^i = n$ take log on both sides

$$\log 2^i = \log n$$

$$i \log_2 2 = \log_2 n$$

$$i = \log_2 n \Rightarrow \text{Substitute value of } i \text{ in eqn (2)}$$

$$T(n) = i.n$$

$$= \log_2 n . n$$

$$T(n) = \Theta(n \log_2 n)$$

② Worst-case time efficiency:-

- The worst case occurs, when the current array is partitioned into 2 subarrays with one of them being empty.
- Also, occurs when the array is already sorted.
- This situation occurs if all the elements are arranged in ascending order.

Eg :- The array [22 33 44 55] yield the following sequence of partitions

22 [33, 44 55]

22 33 [44 55]

22 33 44 55

- To start with, 22 is the key element. It is clear from the partition algorithm that after partitioning, there are no element towards left of 22 and 3 elements are there towards right of 22.

In general, if an array has 'n' elements after partitioning in the worst case, n-1 elements will be there towards right and zero elements will be there towards left of the key element.

Recurrence relation for this case is

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

- $T(0)$: There are no elements towards left & it can be equated to zero
- $T(n-1)$: Time required to sort remaining n-1 elements
- $n$: Time required to partition the array into 2 subarrays

Time efficiency can be calculated as:

$$T(n) = T(0) + T(n-1) + n$$
$$= T(n-1) + n$$
$$= T(n-2) + (n-1) + n$$
$$= T(n-3) + (n-2) + (n-1) + n$$
$$\vdots$$

From,

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

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

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

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

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

In general,

$$= T(n-n) + 1 + 2 + \cdots + (n-2) + (n-1) + n$$
$$= T(0) + 1 + 2 + \cdots + (n-2) + (n-1) + n$$
$$= \frac{n(n+1)}{2} = \frac{n^2+n}{2} \approx n^2$$

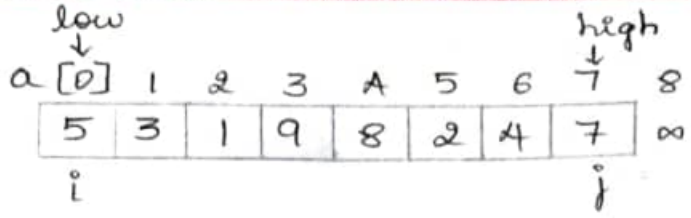
Neglect the lower order terms

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

Eg :-

Apply quick sort for the following elements

5 3 1 9 8 2 4 7




Key = a [low] = 5

i = low = 0

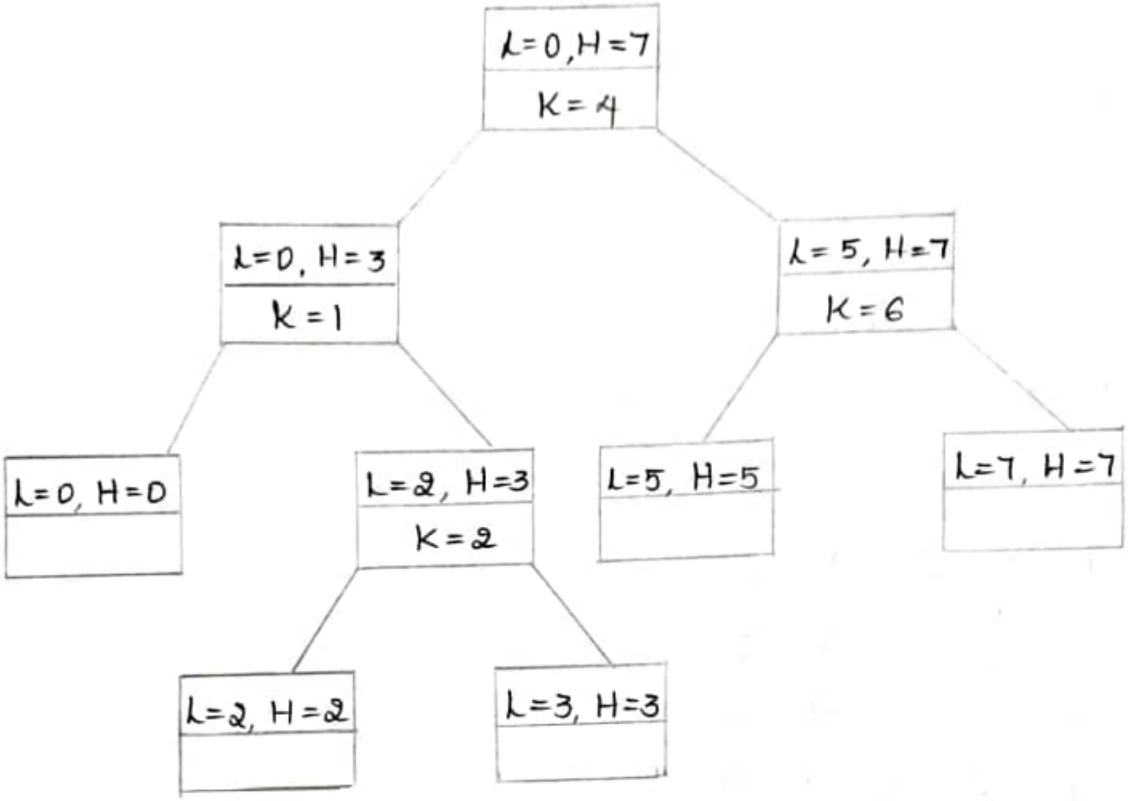
j = high+1 = 8

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| (5) | 3 | 1 | 9 | 8 | 2 | 4 | 7 |
| (5) | 3 | 1 | 9 | 8 | 2 | 4 | 7 |
| (5) | 3 | 1 | 4 | 8 | 2 | 9 | 7 |
| (5) | 3 | 1 | 4 | 8 | 2 | 9 | 7 |
| (5) | 3 | 1 | 4 | 2 | 8 | 9 | 7 |
| (5) | 3 | 1 | 4 | 2 | 8 | 9 | 7 |
| 2 | 3 | 1 | 4 | (5) | 8 | 9 | 7 |
| (2) | 3 | 1 | 4 | | | | |
| (2) | 3 | 1 | 4 | | | | |
| (2) | 1 | 3 | 4 | | | | |
| (2) | 1 | 3 | 4 | | | | |
| 1 | (2) | 3 | 4 | | | | |
| 1 | | | | | | | |

| | | |
|---|---|---|
| | (3) | 4 |
| | (3) | 4 |
| | | 4 |

| | | |
|---|---|---|
| (8) | 9 | 7 |
| (8) | 7 | 9 |
| (8) | 7 | 9 |
| 7 | (8) | 9 |
| 7 | | 9 |

Tree representing the recursive calls :-




## Multiplication of large integers :-

How very long numbers can be multiplied using divide & conquer method ?

- If we use brute-force approach for multiplying 2 n-digit integers each of the 'n' digits of the 1st number is multiplied by each of the 'n' digits of the 2nd number for the total of $n^2$ digit multiplication.
- i.e, In brute force approach, if 'n' denotes the size of 2 integers a & b. The number of multiplication done is $n^2$. Hence, time complexity in this approach is $n^2$. So, to reduce the time complexity of multiplying large number, we use divide and conquer method.

Let a and b be 2 integers of size 'n' denoted with

$a = a_1 a_0$ and $b = b_1 b_0$ where $a_1, a_0, b_1, b_0$ are the digits of size $n/2$.

Product of 2 integers a & b can be calculated as

$$C = C_2 10^n + C_1 10^{n/2} + C_0$$

where,

$C_0 = a_0 \times b_0$ → product of 2nd digit

$C_2 = a_1 \times b_1$ → product of 1st digit

$C_1 = (a_0 + a_1) \times (b_0 + b_1) - (C_2 + C_0)$ → product of sum of a's digit and sum of b's digit minus the sum of $C_2$ and $C_0$

Eg :-

1) Let a = 26 and b = 45. Multiply using divide & conquer methd

Soln :- n = 2

$a_1 = 2 \quad b_1 = 4$

$a_0 = 6 \quad b_0 = 5$

$C_0 = a_0 \times b_0 = 6 \times 5 = 30$

$C_2 = a_1 \times b_1 = 2 \times 4 = 8$

$C_1 = (a_0 + a_1) \times (b_1 + b_0) - (C_2 + C_0)$

$= (6+2) \times (4+5) - (8+30)$

$= 8 \times 9 - 38 \quad = 72 - 38$

$C_1 = 34$

$\therefore C = C_2 10^n + C_1 10^{n/2} + C_0$

$= 8 \cdot 10^2 + 34 \cdot 10 + 30$

$= 800 + 340 + 30$

$C = 1170$

2) a = 1212 b = 322

Soln :- Size of a = 4 & Size of b = 3. In order to make the size equal add 'zero' to b in the prefix position

$\therefore a = 1212 \quad b = 0322 \quad \therefore n = 4$

$a_1 = 12 \quad b_1 = 03$

$a_0 = 12 \quad b_0 = 22$

$C_0 = a_0 \times b_0 = 12 \times 22 = 264$

$C_2 = a_1 \times b_1 = 12 \times 3 = 36$

$C_1 = (a_0 + a_1) \times (b_0 + b_1) - (C_2 + C_0)$

$= (12 + 12) \times (3 + 22) - (36 + 264)$

$= 24 \times 25 - 300 = 600 - 300$

$C_1 = 300$

$\therefore C = C_2 10^n + C_1 10^{n/2} + C_0$

$= 36.10^4 + 300.10^2 + 264$

$= 360000 + 30000 + 264$

$C = 390264$

Analysis :-

- It is clear that, 2 n-digit numbers requires 3 multiplication (i.e, $C_2, C_0, C_1$) of $n/2$ digit numbers and the recurrence relation is

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 3T(n/2) & \text{otherwise} \end{cases}$$

Consider the relation

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

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

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

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

$\vdots$

In general,

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

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

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

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

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

To get the initial condition $T(1) = 1$

let $2^i = n$ — (1)

$T(n) = 3^i . T(1)$

$T(n) = 3^i$ —— (2)

From eqn (1), we have $2^i = n$. Take log on both sides.

$\log 2^i = \log n$

$i \log_2 2 = \log_2 n$

$i = \log_2 n$

Substitute value of i in eqn (2)

$T(n) = 3^i$

$= 3^{\log_2 n}$

$= n^{\log_2 3}$ By $a^{\log_2 b} = b^{\log_2 a}$

$T(n) = n^{1.585}$

$T(n) \in \theta(n^{1.585})$

## Strassen's Matrix Multiplication :-

- Time complexity of an algorithm to multiply 2 matrices using brute force method is $\theta(n^3)$
- Strassen's multiplication is a new method where the number of multiplication can be reduced there by increasing the efficiency. It can be accomplished by using the formula which follows divide and conquer technique.

$$\begin{bmatrix} C_1 & C_2 \\ C_3 & C_4 \end{bmatrix} = \begin{bmatrix} A_1 & A_2 \\ A_3 & A_4 \end{bmatrix} \times \begin{bmatrix} B_1 & B_2 \\ B_3 & B_4 \end{bmatrix} = \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}$$

where, $m_1 = (A_1 + A_4) \times (B_1 + B_4)$

$m_2 = (A_3 + A_4) \times B_1$

$m_3 = A_1 \times (B_2 - B_4)$

$m_4 = A_4 \times (B_3 - B_1)$

$m_5 = (A_1 + A_2) \times B_4$

$m_6 = (A_3 - A_1) \times (B_1 + B_2)$

$m_7 = (A_2 - A_4) \times (B_3 + B_4)$

- From the above relation, we can say Strassen's matrix multiplication algorithm require 7 multiplications.
- If A and B are the matrices of size $n \times n$ then $A_1, A_2, A_3, A_4$ and $B_1, B_2, B_3, B_4$ are the values of size $n/2$.
- Strassen's algorithm is more efficient than the conventional matrix multiplication for large value of $n$, but not for smaller values of $n$.
- If all 7 products are computed recursively then the recurrence relation can be written as

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 7T(n/2) & \text{otherwise} \end{cases}$$

Analysis :-

Time Complexity can be calculated as shown

$$T(n) = 7 \cdot T\left(\frac{n}{2}\right)$$

$$= 7\left[7 \cdot T\left(\frac{n}{2^2}\right)\right] = 7^2 T\left(\frac{n}{2^2}\right)$$

$$= 7^2\left[7 \cdot T\left(\frac{n}{2^3}\right)\right] = 7^3 T\left(\frac{n}{2^3}\right)$$

$$\vdots$$

$$T(n) = 7T\left(\frac{n}{2}\right)$$

$$T\left(\frac{n}{2}\right) = 7T\left(\frac{n}{2^2}\right)$$

$$T\left[\frac{n}{2^2}\right] = 7T\left(\frac{n}{2^3}\right)$$

In general,

$$T(n) = 7^i T\left(\frac{n}{2^i}\right)$$

To get the initial condition $T(1) = 1$ let $2^i = n$ — ①

$$= 7^i T\left(\frac{n}{n}\right)$$

$$= 7^i \cdot T(1)$$

$$T(n) = 7^i \text{ —— ②}$$

From eqn ①, we have $2^i = n$, take log on both sides

$$\log 2^i = \log n$$

$$i \log_2 2 = \log n \quad \because i = \log_2 n$$

Substitute the value of 'i' in eqn (2), we get

$$T(n) = 7^i$$
$$= 7^{\log_2 n}$$
$$= n^{\log_2 7} \qquad \text{By } a^{\log_2 b} = b^{\log_2 a}$$
$$T(n) = n^{2.807}$$
$$T(n) \in \theta(n^{2.807})$$

Note :- Algorithm works fine if the size of the matrix is always a power of 2. If the size of the matrix is not a multiple of 2, add sufficient zeros both in rows and columns. So, that size are always a multiple of 2.

Eg :-

1) Using divide and conquer method, multiply 2 matrices with the help of strassen's matrix multiplication

$$A = \begin{bmatrix} 1 & 2 \\ 5 & 6 \end{bmatrix} \qquad B = \begin{bmatrix} 8 & 7 \\ 1 & 2 \end{bmatrix}$$

Soln :- $n = 2 \times 2 \qquad n/2 = 2/2 = 1 \quad$ i.e, $1 \times 1$

$$A = \begin{bmatrix} A_1 & A_2 \\ A_3 & A_4 \end{bmatrix} \quad \text{So, } A_1 = 1 \quad A_3 = 5 \quad A_2 = 2 \quad A_4 = 6$$

$$B = \begin{bmatrix} B_1 & B_2 \\ B_3 & B_4 \end{bmatrix} \quad \text{So, } B_1 = 8 \quad B_3 = 1 \quad B_2 = 7 \quad B_4 = 2$$

$$m_1 = (A_1 + A_4)(B_1 + B_4) = (1+6)(8+2) = 70$$
$$m_2 = (A_3 + A_4) B_1 = (5+6) 8 = 88$$
$$m_3 = A_1 (B_2 - B_4) = 1 (7-2) = 5$$
$$m_4 = A_4 (B_3 - B_1) = 6 (1-8) = -42$$
$$m_5 = (A_1 + A_2) B_4 = (1+2) 2 = 6$$
$$m_6 = (A_3 - A_1)(B_1 + B_2) = (5-1)(8+7) = 60$$
$$m_7 = (A_2 - A_4)(B_3 + B_4) = (2-6)(1+2) = -12$$

Final matrix is :-

$$C = \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix} = \begin{bmatrix} C_1 & C_2 \\ C_3 & C_4 \end{bmatrix}$$

$$= \begin{bmatrix} 70 - 42 - 6 + 12 & 5 + 6 \\ 88 - 42 & 70 + 5 - 88 + 60 \end{bmatrix}$$

$$C = \begin{bmatrix} 10 & 11 \\ 46 & 47 \end{bmatrix}$$

2) $$A = \begin{bmatrix} 1 & 0 & 2 & 1 \\ 4 & 1 & 1 & 0 \\ 0 & 1 & 3 & 0 \\ 5 & 0 & 2 & 1 \end{bmatrix} \qquad B = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 2 & 1 & 0 & 4 \\ 2 & 0 & 1 & 1 \\ 1 & 3 & 5 & 0 \end{bmatrix}$$

Sol$^n$ :- $n = 4 \times 4$, so $n/2 = 4/2 = 2$ i.e; $2 \times 2$

$$A_1 = \begin{bmatrix} 1 & 0 \\ 4 & 1 \end{bmatrix} \quad A_2 = \begin{bmatrix} 2 & 1 \\ 1 & 0 \end{bmatrix} \quad A_3 = \begin{bmatrix} 0 & 1 \\ 5 & 0 \end{bmatrix} \quad A_4 = \begin{bmatrix} 3 & 0 \\ 2 & 1 \end{bmatrix}$$

$$B_1 = \begin{bmatrix} 0 & 1 \\ 2 & 1 \end{bmatrix} \quad B_2 = \begin{bmatrix} 0 & 1 \\ 0 & 4 \end{bmatrix} \quad B_3 = \begin{bmatrix} 2 & 0 \\ 1 & 3 \end{bmatrix} \quad B_4 = \begin{bmatrix} 1 & 1 \\ 5 & 0 \end{bmatrix}$$

$$m_1 = (A_1 + A_4)(B_1 + B_4) = \begin{bmatrix} 4 & 0 \\ 6 & 1 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ 7 & 1 \end{bmatrix} = \begin{bmatrix} 4 & 8 \\ 20 & 14 \end{bmatrix}$$

$$m_2 = (A_3 + A_4) B_1 = \begin{bmatrix} 3 & 1 \\ 7 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 4 \\ 2 & 8 \end{bmatrix}$$

$$m_3 = A_1 (B_2 - B_4) = \begin{bmatrix} 1 & 0 \\ 4 & 1 \end{bmatrix} \begin{bmatrix} -1 & 0 \\ -5 & 4 \end{bmatrix} = \begin{bmatrix} -1 & 0 \\ -9 & 4 \end{bmatrix}$$

$$m_4 = A_4 (B_3 - B_1) = \begin{bmatrix} 3 & 0 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} 2 & -1 \\ -1 & 2 \end{bmatrix} = \begin{bmatrix} 6 & -3 \\ 3 & 0 \end{bmatrix}$$

$$m_5 = (A_1 + A_2)B_4 = \begin{bmatrix} 3 & 1 \\ 5 & 1 \end{bmatrix}\begin{bmatrix} 1 & 1 \\ 5 & 0 \end{bmatrix} = \begin{bmatrix} 8 & 3 \\ 10 & 5 \end{bmatrix}$$

$$m_6 = (A_3 - A_1)(B_1 + B_2) = \begin{bmatrix} -1 & 1 \\ 1 & -1 \end{bmatrix}\begin{bmatrix} 0 & 2 \\ 2 & 5 \end{bmatrix} = \begin{bmatrix} 2 & 3 \\ -2 & -3 \end{bmatrix}$$

$$m_7 = (A_2 - A_4)(B_3 + B_4) = \begin{bmatrix} -1 & 1 \\ -1 & -1 \end{bmatrix}\begin{bmatrix} 3 & 1 \\ 6 & 3 \end{bmatrix} = \begin{bmatrix} 3 & 2 \\ -9 & -4 \end{bmatrix}$$

Final matrix is given by:

$$C = \begin{bmatrix} C_1 & C_2 \\ C_3 & C_4 \end{bmatrix} = \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}$$

where, $C_1 = \begin{bmatrix} 4 & 8 \\ 20 & 14 \end{bmatrix} + \begin{bmatrix} 6 & -3 \\ 3 & 0 \end{bmatrix} - \begin{bmatrix} 8 & 3 \\ 10 & 5 \end{bmatrix} + \begin{bmatrix} 3 & 2 \\ -9 & -4 \end{bmatrix} = \begin{bmatrix} 5 & 4 \\ 4 & 5 \end{bmatrix}$

$$C_2 = \begin{bmatrix} -1 & 0 \\ -9 & 4 \end{bmatrix} + \begin{bmatrix} 8 & 3 \\ 10 & 5 \end{bmatrix} = \begin{bmatrix} 7 & 3 \\ 1 & 9 \end{bmatrix}$$

$$C_3 = \begin{bmatrix} 2 & 4 \\ 2 & 8 \end{bmatrix} + \begin{bmatrix} 6 & -3 \\ 3 & 0 \end{bmatrix} = \begin{bmatrix} 8 & 1 \\ 5 & 8 \end{bmatrix}$$

$$C_4 = \begin{bmatrix} 4 & 8 \\ 20 & 14 \end{bmatrix} + \begin{bmatrix} -1 & 0 \\ -9 & 4 \end{bmatrix} - \begin{bmatrix} 2 & 4 \\ 2 & 8 \end{bmatrix} + \begin{bmatrix} 2 & 3 \\ -2 & -3 \end{bmatrix} = \begin{bmatrix} 3 & 7 \\ 7 & 7 \end{bmatrix}$$

$$\therefore C = \begin{bmatrix} C_1 & C_2 \\ C_3 & C_4 \end{bmatrix} = \begin{bmatrix} 5 & 4 & 7 & 3 \\ 4 & 5 & 1 & 9 \\ 8 & 1 & 3 & 7 \\ 5 & 8 & 7 & 7 \end{bmatrix}$$

## Decrease and Conquer :-

- It is a method of solving a problem by
  a) changing the problem size from n to smaller size of $n-1$, $n/2$ etc
  b) Solve or conquer the problem of smaller size
  c) Convert the solution of smaller size problem into a solution for larger size problem.
- Using this technique, we can solve a given problem using top-down technique (recursively) or bottom up technique (Iteration)
- Decrease and conquer method is a slight variation of divide and conquer method.
- There are 3 major variation of decrease and conquer method
  1) Decrease by a constant
  2) Decrease by a constant factor
  3) Variable size decrease.

### 1) Decrease by a constant :-

- Here, problem size is usually decremented by one in each iteration of the loop.
- Various problem that can be solved using this method are computing $a^n$, insertion sort algorithm, traversing the graph using BFS & DFS method, topological sorting etc...

Eg :- Exponentiation problem of computing $a^n$ for positive integer exponents.

Design :- Value of $a^n$ can be recursively (top-down) defined using decrease by a constant as shown

$$a^n = \begin{cases} a & \text{if } n = 1 \\ a^{n-1} \cdot a & \text{otherwise} \end{cases}$$

- In the given relation, the larger instance of size 'n' is expressed in terms of smaller instance of size n-1
- Recursive relation can also be written as

$$f(a,n) = \begin{cases} a & \text{if } n=1 \\ f(a,n-1).\,a & \text{otherwise} \end{cases}$$

Algorithm :-

```
Algorithm power(a,n)
// To compute a^n
// Input :- a and n are inputs used to compute a^n
// output :- Result of a^n is returned
    if (n=1) return a
    return power(a, n-1) * a
```

- Decrease by a constant is illustrated as shown in figure

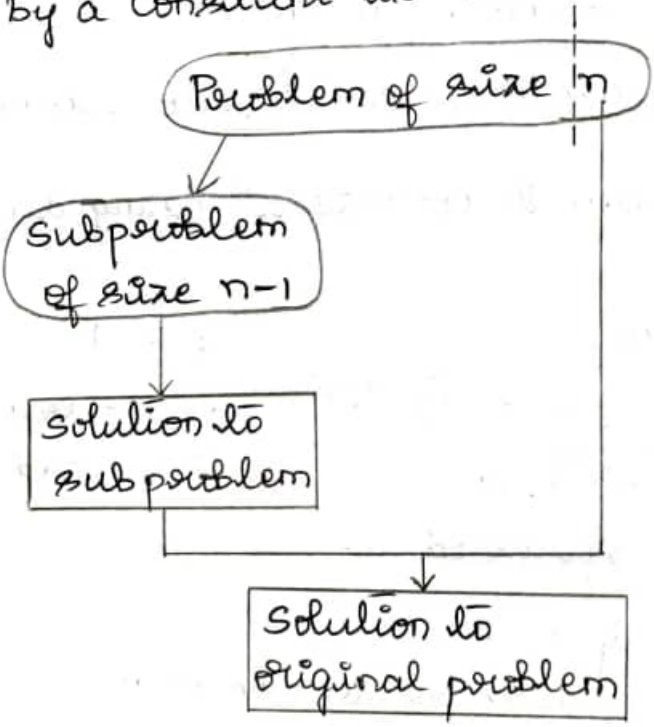



Fig :- Decrease by constant technique

## 2) Decrease by a constant factor :-

- Here, the problem size is reduced by same constant factor (usually by 2) on each iteration of the algorithm.
- Idea of decrease by constant factor is illustrated as shown in figure

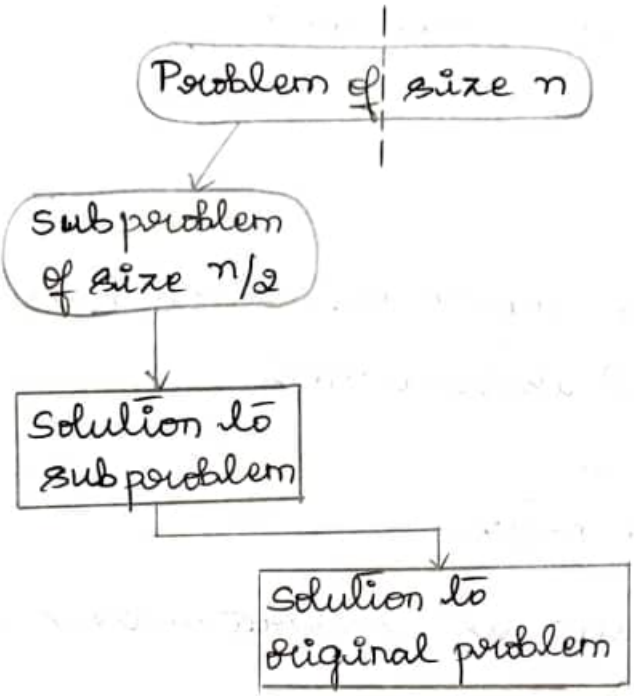



Fig :- Decrease by constant factor technique

Eg :- Design the algorithm to compute $a^n$ using decrease by a constant factor.

we know that, $a^n = a$ if $n=1$

$a^n = a^{n/2} \cdot a^{n/2} = (a^{n/2})^2$ if $n$ = even

$a^n = (a^{n/2})^2 \cdot a$ if $n$ = odd

These relation can be expressed as

$$a^n = \begin{cases} a & \text{if } n=1 \\ (a^{n/2})^2 & \text{if } n \text{ is positive and even} \\ (a^{n/2})^2 \cdot a & \text{if } n \text{ is odd and greater than 1} \end{cases}$$

In above relation, larger instance of size 'n' is expressed in terms of smaller instance of size $n/2$.

Above relation can also be written as

$$f(a,n) = \begin{cases} a & \text{if } n=1 \\ f(f(a,n/2),2) & \text{if } n \text{ is positive \& even} \\ f(f(a,n/2),2)*a & \text{if } n \text{ is odd \& } >1 \end{cases}$$

Algorithm :-

```
Algorithm power (a,n)
// To compute a^n
// Input :- a and n are positive integers
// Output :- a^n is returned
    if (n=1) return a
    if (n>1 & n%2 = 0)  return power (power(a, n/2), 2)
    return power (power (a, n/2), 2) * a
```

3) Variable size decrease :-

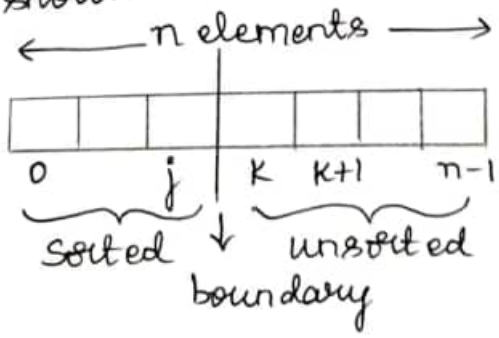
- Here, in each iteration of the loop, the size reduction pattern varies from one iteration of the algorithm to another iteration

Eg :- GCD of 2 numbers m and n using

$$gcd(m,n) = gcd(n, m \bmod n)$$

## Insertion Sort :-

- It is one of the simplest technique to arrange the given elements in ascending order.
- It follows decrease and conquer technique
- Given list will be divided into 2 parts - sorted and unsorted part as shown

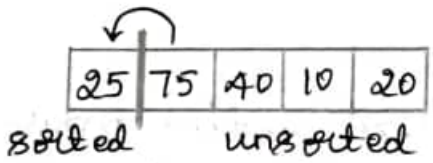


- All the elements from 0 to j are sorted and elements from K to n-1 are not sorted.
- $k^{th}$ element can be inserted into any of the position from 0 to j so that elements towards the left of the boundary is sorted
- As each element is inserted towards the sorted left part, boundary moves to the right decreasing the unsorted list.
- Finally, once the boundary moves to the right most position the the elements towards the left of boundary represent the sorted list.

Eg :- sort the elements 25, 75, 40, 10, 20 using insertion sort

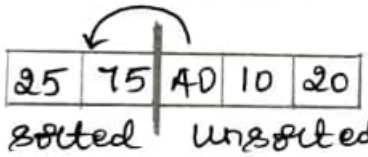
Sol$^n$ :-

Item to be inserted is 75 i.e, a[1] = 75



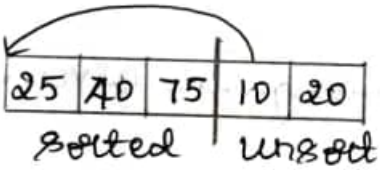

75 is inserted after 25

Item to be inserted is 40 i.e, a[2] = 40



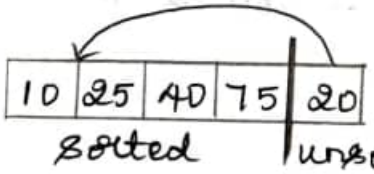

40 is inserted between 25 & 75

Item to be inserted is 10 i.e, a[3] = 10



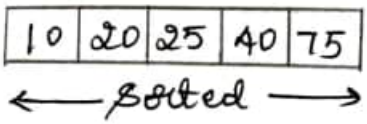

10 is inserted before 25

Item to be inserted is 20 i.e, a[4] = 20




20 is inserted between 10 & 25

output :-



Design :-

- Consider an array of 'n' elements to sort. Elements to be inserted can be accessed as shown

Item = a[1]
Item = a[2]
Item = a[3]
Item = a[4]

i.e, Item = a[i] where i = 1 to 4
i.e, i = 1 to n-1

So, in general item = a[i] for i = 1 to n-1

- Now 'item' has to be compared with a[j] as long as (item < a[j] and j >= 0) with initial value of j = i-1. As long as the above condition is true perform the steps
  - copy a[j] to a[j+1]
  - Decrement j by 1
- Equivalent statements can be written as

```
j = i-1
while ( a[j] > item and j >=0)
  a[j+1] ← a[j]
  j ← j-1
end while
```

- Once control comes out of the above loop insert the item into a[j+1] using the statement.

```
a[j+1] ← item
```

Algorithm :-

```
Algorithm insertion sort (a, n)
// sort the list in ascending order
// Input :- a → list to be sorted
           n → total no of elements in the list
// Output :- a → sorted list
  for i ← 1 to n-1 do
   item ← a[i]
    j ← i-1
```

```
while (j>=0 and a[j] > item)
    a[j+1] ← a[j]
    j ← j-1
end while
    a[j+1] ← item
end for
```

Analysis :-

1) Best-case time efficiency :-

- Best case occurs when the items in the list are partially or nearly sorted
- Whenever for loop is executed, condition 'a[j] > item' is executed and it is the basic operation of this algorithm.
- Since the loop is executed (n-1) times, condition is also executed (n-1) times. So, the time complexity is given by

$$T(n) = \sum_{i=1}^{n-1} 1 \qquad \text{Res} = UB - LB + 1$$

$$= (n-1) - 1 + 1 = n-1$$

$$T(n) = n$$

$$T(n) \in \Omega(n)$$

2) Worst-case time efficiency :-

- Worst case occurs when the condition 'a[j] > item' is executed maximum number of times.
- This situation occurs when the elements of the list are sorted in descending order.
- Time complexity can be obtained as shown

```
for i ← 1 to n-1 do
    item ← a[i]
    j ← i-1
    while (j>=0 and a[j] > item)
```

$$T(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1$$

$$T(n) = \sum_{i=1}^{n-1} \left( \sum_{j=0}^{i-1} 1 \right)$$

$$= \sum_{i=1}^{n-1} (i-1) - 0 + 1$$

$$= \sum_{i=1}^{n-1} i \qquad \text{By } \sum_{i=1}^{n} = \frac{n(n+1)}{2}$$

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

$$T(n) = n^2$$

$$T(n) \in O(n^2)$$

3) Average case time efficiency :-

- Assume that 2 elements are already sorted and we have to insert the 3rd element at the appropriate position. There are 3 possible places where the element can be inserted.
- Consider all possible cases.

Case-1 :-

- $i = 2$, position where item has to be inserted and item $= a[2] = 13$

| A | 10 | 12 | 13 |
|---|---|---|---|
| | 0 | 1 | 2 |
| | | j | i |

- Item 13 is compared with 12. Since 13 is greater than 12, control comes out of the while loop & the while loop is executed only once. So, the total number of times the while loop is executed = 1.

Case-2 :-

- $i = 2$, position where the item has to be inserted & item $= a[2] = 11$

| A | 10 | 12 | 11 |
|---|---|---|---|
| | 0 | 1 | 2 |
| | | j | i |

- Item 11 has to be compared with 12 as well as with 10. The item should be inserted between 10 and 12, which results in the while loop to be executed twice.
- So, the total number of times the while loop is executed = 2

Case-3 :-

- $i = 2$, position where the item has to be inserted & item $= a[2] = 9$

| A | 10 | 12 | 9 |
|---|---|---|---|
| | 0 | 1 | 2 |
| | | j | i |

- Item 9 is compared with 12 as well as 10 and should be inserted before 10 which results in while loop to be executed 3 times
- So, the total number of times the while loop is executed = 3
- All the 3 cases have the same probability and the average number of times the while loop is executed is given by

$$\frac{1+2+3}{3}$$

- This result is true, if we are inserting the 3rd element into the array.
- In general, to insert an item 'x' with index 'i' in correct position, the total number of times the while loop is executed is given by

$$\frac{1+2+3+\cdots+i}{i} = \frac{i(i+1)}{2*i} = \frac{i+1}{2}$$

- From algorithm, index 'i' start from 1 to n-1. So, the average number of times the while loop is executed is given by

$$T(n) = \sum_{i=1}^{n-1} \frac{i+1}{2}$$

$$= \frac{1}{2} \sum_{i=1}^{n-1} i+1$$

$$= \frac{1}{2} \left[ \sum_{i=1}^{n-1} i + \sum_{i=1}^{n-1} 1 \right]$$

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

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

$$= \frac{1}{2} \left[ \frac{n^2+n-2}{2} \right] = \frac{n^2}{4} + \frac{n}{4} - \frac{1}{2}$$

By neglecting lower order terms

$$T(n) = n^2$$

$$T(n) \in \Theta(n^2)$$

## Depth First Search :-

- DFS starts visiting vertices of a graph at an arbitrary vertex by marking it as having been visited, on each iteration algorithm proceeds to an unvisited vertex that is adjacent to the one it is currently in.
- This process continues until a dead end i.e, a vertex with no adjacent unvisited vertices is encountered.
- At a dead end, algorithm backs up one edge to the vertex it came from and tries to continue visiting unvisited vertices from there.
- Algorithm eventually halts after backing up the starting vertex, with the latter being a dead end.
- By then, all the vertices in the same connected component as the starting vertex have been visited.
- If any unvisited vertices still remain, DFS must be restarted at any one of them
- It is convenient to use stack to trace the operation of DFS
- We push a vertex onto the stack when the vertex is reached for the 1st time and we pop a vertex off the stack when it becomes a dead end.
- DFS uses 2 types of edges — Tree edge and Back edge

Tree edge :- whenever a new unvisited vertex is reached for the 1st time, it is attached as a child to the vertex from which it is being reached. such an edge is called a tree edge Because the set of all such edges form a forest.

Back edge :- An edge leading to previously visited vertex other than its immediate predecessor i.e, its parent. such an edge is called a back edge. Because it connects a vertex to its ancestor other than the parent.

- Tree edges are represented using solid lines.
- Back edges are represented using dotted lines.

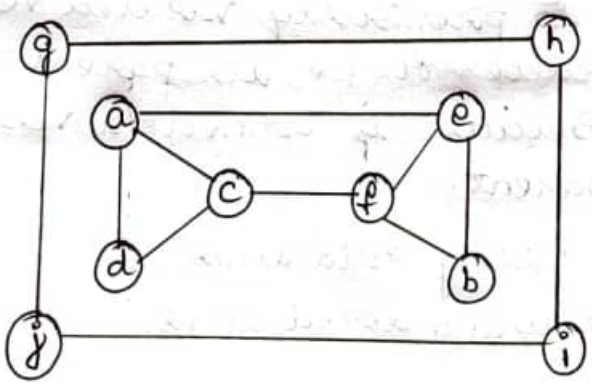
Algorithm :-

Algorithm dfs(a, n, u, s, t)
// Traverse the graph from the given node source in DFS
// Input :- a → adjacency matrix
n → number of nodes in graph
u → from where the traversal is to be started
s → Indicates the vertices that are visited and not visited.
// Output :- (u, v) → nodes v reachable from u are sorted in a vector t

```
s[u] ← 1
for every v adjacent to vertex u
if v is not visited
  t[k][0] ← u
  t[k][1] ← v
  k ← k+1
  dfs(a, n, v, s, t)
end if
end for
```

Eg :-

Traverse the graph using DFS and construct the DFS tree. Give the order in which the vertices were reached for the 1st time (pushed onto stack) and order in which the vertices become dead end (popped off the stack). Mention tree edges and back edges of the graph.



| Step-1 Stack | Step-2 $V = adj(S[top])$ | Step-3 Node Visited(S) | Step-4 pop(stack) | Step-5 Output |
|---|---|---|---|---|
| - | - | a | - | |
| $a_1$ | c | a, c, | - | a-c |
| $a_1 c_2$ | d | a, c, d | - | c-d |
| $a_1 c_2 d_3$ | - | a, c, d | $d_{3,1}$ | - |
| $a_1 c_2$ | f | a, c, d, f | - | c-f |
| $a_1 c_2 f_4$ | b | a, b, c, d, f | - | f-b |
| $a_1 c_2 f_4 b_5$ | e | a, b, c, d, e, f | - | b-e |
| $a_1 c_2 f_4 b_5 e_6$ | - | a, b, c, d, e, f | $e_{6,2}$ | - |
| $a_1 c_2 f_4 b_5$ | - | a, b, c, d, e, f | $b_{5,3}$ | - |
| $a_1 c_2 f_4$ | - | a, b, c, d, e, f | $f_{4,4}$ | - |
| $a_1 c_2$ | - | a, b, c, d, e, f | $c_{2,5}$ | - |
| $a_1$ | - | a, b, c, d, e, f | $a_{1,6}$ | - |
| - | - | a, b, c, d, e, f | - | - |

Note :- Stack is empty. So, take the next unvisited vertex.

| | | | | |
|---|---|---|---|---|
| $g_7$ | h | a, b, c, d, e, f, g | - | g-h |
| $g_7 h_8$ | i | a, b, c, d, e, f, g, h | - | h-i |
| $g_7 h_8 i_9$ | j | a, b, c, d, e, f, g h, i | - | i-j |
| $g_7 h_8 i_9 j_{10}$ | - | a, b, c, d, e, f, g h, i, j | $j_{10,7}$ | - |
| $g_7 h_8 i_9$ | - | a, b, c, d, e, f, g, h i, j | $i_{9,8}$ | - |
| $g_7 h_8$ | - | a, b, c, d, e, f, g, h i, j | $h_{8,9}$ | - |
| $g_7$ | - | a, b, c, d, e, f, g, h, i j | $g_{7,10}$ | - |
| - | - | a, b, c, d, e, f, g, h, i, j | - | - |

Ordering | DFS traversal

- Traversal stack (or) the order in which vertices are pushed and popped out of the stack are obtained from the column values of step-4 as shown.

$d_{3,1}$ $e_{6,2}$ $j_{10,7}$
$c_{2,5}$ $b_{5,5}$ $i_{9,8}$
$a_{1,6}$ $f_{4,4}$ $h_{8,9}$
$g_{7,10}$

- 1st subscript number indicates the order in which a vertex was visited i.e, pushed onto the stack
- 2nd subscript number indicates the order in which a vertex became a dead end i.e, popped out of the stack
- DFS tree and traversal is obtained by looking at the output column of step-5 as shown

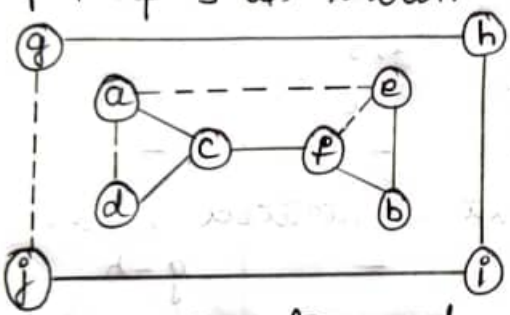



Fig :- DFS traversal

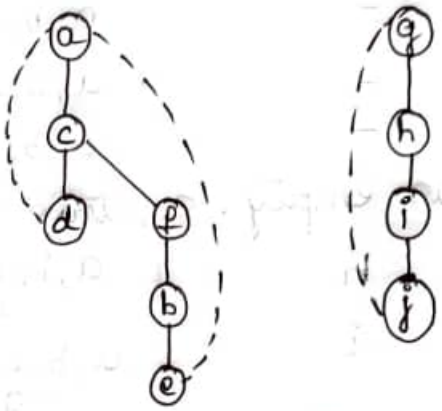






Fig :- DFS tree

Note :-

- After traversing the graph in DFS, if there are no back edges then the graph is acyclic. If there is a back edge from some vertex u to its ancestor v, then the graph has a cycle.

Analysis :-

- Analysis of DFS depends on the number of vertices and whether the adjacency matrix or list representation of a graph is used.
- Time efficiency for adjacency matrix representation is $\theta(|V|^2)$
- Time efficiency of adjacency linked list representation is $\theta(|V|+|E|)$

## Breadth First Search :-

- BFS is a method of traversing the graph by visiting each node of the graph in a systematic order.
- It proceeds in a concentric manner by visiting first all the vertices that are adjacent to a starting vertex then all the unvisited vertices 2 edges apart from it and so on, until all the vertices in the same connected component as the starting vertex are visited
- If there are still remain unvisited vertices, algorithm has to be restarted at an arbitrary vertex of another connected component of the graph.
- It is convenient to use queue to trace the operation of BFS
- when a vertex is reached for the 1st time, it is inserted into rear end of queue
- when we get a dead end i.e, the vertex is already explored we delete a vertex from the front of queue.
- Thus, BFS yields only one ordering of vertices i.e, The order in which the vertices are inserted into queue is same as the order in which vertices are deleted from queue.
- BFS uses 2 types of edges — Tree edge and cross edge.

Tree edge :- whenever a new unvisited vertex is reached for the 1st time the vertex is attached as a child to the vertex it is being reached from with an edge called tree edges which are represented as solid lines.

Cross edge :- If an edge leading to a previously visited vertex other than its immediate predecessor i.e, its parent is encountered, edge is noted as cross edge which is represented as dotted lines.

Algorithm :-

```
Algorithm BFS (a, n, source, T)
// Traverse the graph from the given source node in BFS
// Input :- a → adjacency matrix of the given graph
            n → number of nodes in the graph.
```

Source → from where the traversal is initiated

// output :- (u, v) → nodes v reachable from u are stored in vector T

```
for i ← 0 to n-1 do
  s[i] ← 0
end for
f ← r ← 0
q[r] ← source
s[source] ← 1
  k ← 0
while (f <= r)
  u ← q[f]
  f ← f+1
```

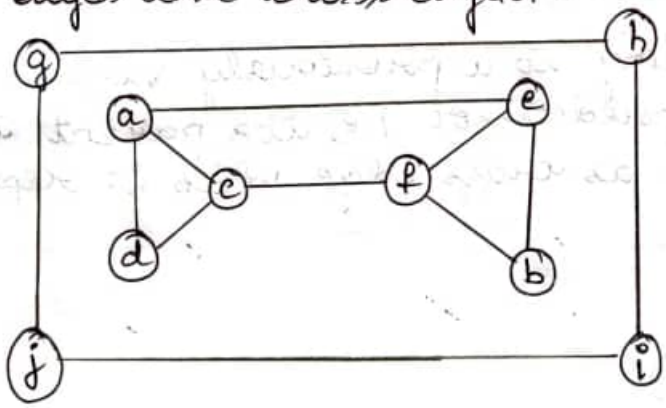
```
for every v adjacent to u do
  if v is not visited
    s[v] ← 1
    r ← r+1
    q[r] ← v
    T[k,1] ← u
    T[k,2] ← v
    k ← k+1
  end if
end for
end while
```

Analysis :-

- Analysis of BFS depends on the number of vertices and whether the adjacency matrix or adjacency list representation of a graph is used.
- Time efficiency for adjacency matrix is $\theta(|V^2|)$
- Time efficiency for adjacency linked list representation is $\theta(|V|+|E|)$

Eg :-

Traverse the graph by BFS and construct the BFS tree. Show tree edges and cross edges.



| Step - 1 | Step - 2 | Step - 3 | | |
|---|---|---|---|---|
| $u = del(Q)$ | $V = adj$ to $u$ | Nodes visited, S | Queue, Q | Output, $T(u,v)$ |
| – | – | a | a | – |
| a | c, d, e | a, c, d, e | c, d, e | a – c<br>a – d<br>a – e |
| c | a, d, f | a, c, d, e, f | d, e, f | c – f |
| d | a, c | a, c, d, e, f | e, f | – |
| e | a, b, f | a, c, d, e, f, b | f, b | e – b |
| f | c, b, e | a, b, c, d, e, f | b | – |
| b | e, f | a, b, c, d, e, f | – | – |

Note :- Queue is empty, but there are some vertices that are not visited. So, for the next subgraph redo the above procedure

| | | | | |
|---|---|---|---|---|
| – | – | a, b, c, d, e, f, g | g | – |
| g | h, j | a, b, c, d, e, f, g, h, j | h, j | g – h<br>g – j |
| h | g, i | a, b, c, d, e, f, g, h, i, j | j, i | h – i |
| j | g, i | a, b, c, d, e, f, g, h, i, j | i | – |
| i | h, j | a, b, c, d, e, f, g, h, i, j | – | – |

ordering

BFS traversal

- Ordering of vertices is obtained by looking at the 1st column where the vertices are deleted from queue.

  a, c d e f b g h j i

- BFS traversal is obtained by looking at the output column.

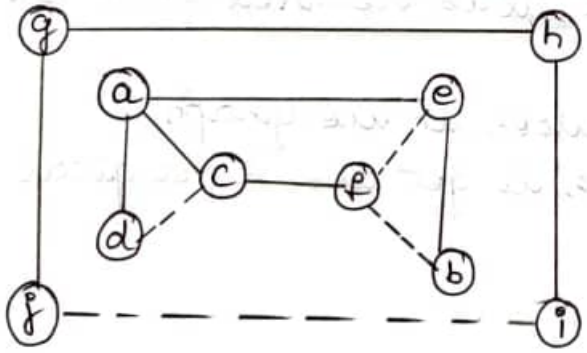



Fig :- BFS traversal

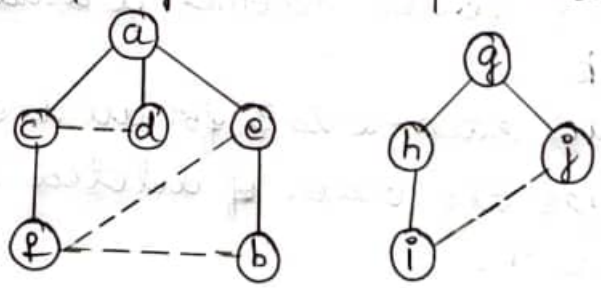



Fig :- BFS tree

# Topological Sorting :-

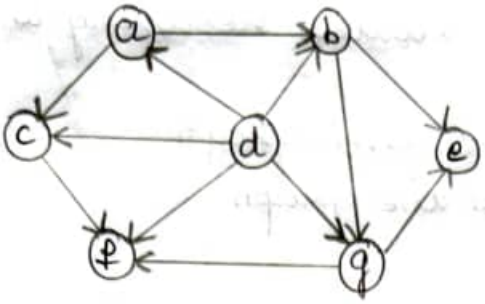
- Topological sort of a Directed Acyclic Graph (DAG) $G=(V,E)$ is a linear ordering of all the vertices such that for every edge $(u,v)$ in a graph G, the vertex u appears before the vertex v in the ordering.
- A topological sort of a graph can be viewed as an ordering of vertices along a horizontal line so that all directed edges go from left to right.
- For a cyclic graph, no linear ordering is possible.
- If a digraph has no cycles, topological sorting problem for it has a solution.
- Topological sorting can be done using following 2 methods
  1) DFS method
  2) Source Removal method
- Topological sorting solution obtained by the source removal method is different from the one obtained by the DFS method Both of them are correct of course, the topological sorting problem may have several alternative solution.

## 1) Topological sort using DFS method :-

Topological sort using DFS method can be obtained as shown in below steps:

a) Select any arbitrary vertex
b) When a vertex is visited for the 1st time, it is pushed onto the stack
c) When a vertex becomes a dead end, it is removed from the stack
d) Repeat step-2 to 3 for all the vertices in the graph
e) Reverse the order of deleted items to get the topological sequence.

Eg :- Apply the DFS based method to solve the topological problem for the following graph.




| Stack | V = adj(s[top]) | Nodes Visited, S | Pop (stack) |
|---|---|---|---|
| – | – | a | – |
| a | b | a, b | – |
| a, b | e | a, b, e | – |
| a, b, e | – | a, b, e | e |
| a, b | g | a, b, e, g | – |
| a, b, g | f | a, b, e, g, f | – |
| a, b, g, f | – | a, b, e, f, g | f |
| a, b, g | – | a, b, e, f, g | g |
| a, b | – | a, b, e, f, g | b |
| a | c | a, b, c, e, f, g | – |
| a, c | – | a, b, c, e, f, g | c |
| a | – | a, b, c, e, f, g | a |
| – | – | a, b, c, e, f, g | – |
| – | – | a, b, c, e, f, g, d | – |
| d | – | a, b, c, e, f, g, d | d |
| – | – | a, b, c, d, e, f, g | – |

- The order in which vertices are removed from the stack is obtained from the last column. The popped order is
  e, f, g, b, c, a, d → popped sequence.
- Topological order is obtained by reversing the above popped sequence which is given by
  d → a → c → b → g → f → e

Algorithm :-

Algorithm topological_order(n, a)
// To obtain the sequence of jobs to be executed resulting in topological order
// Input :- a → adjacency matrix of the given graph
n → number of vertices in the graph
// Global Variables :-
S → to know what are the nodes visited and what are the nodes that are not visited
j → Index variable to store the vertices (nodes which are dead ends)
res → array which holds the order in which the vertices are popped.
// output :- res → indicates the vertices in the reverse order that are to be executed.

```
for i←0 to n-1 do
   S[i]←0
end for
 j←0
 for u←0 to n-1 do
  if (S[u] = 0)
  DFS (u, n, a)
end for
for i←n-1 to 0
  print res[i]
end for
return
```

Analysis :-

1) Adjacency matrix :-

- Running time of DFS function is the time required to execute the statement
- If the graph is represented using adjacency matrix time

efficiency is calculated as shown

$$T(n) = \sum_{v=0}^{n-1} \sum_{v=0}^{n-1} 1$$

$\sum_{v=0}^{n-1}$ → from topological ordering

$\sum_{v=0}^{n-1}$ → from DFS

$$= \sum_{v=0}^{n-1} n-1-0+1$$

$$= n \sum_{v=0}^{n-1} 1 \quad = n(n-1-0+1) = n^2$$

$$T(n) = n^2$$

Time complexity of topological sort is given by $\Theta(n^2)$. For a given graph $G=(V,E)$ where $V \rightarrow$ no of vertices. Hence, $n = |V|$

∴ Time complexity is given by $\Theta|V^2|$

$$T(n) = \Theta|V^2|$$

2) Adjacency list representation :-

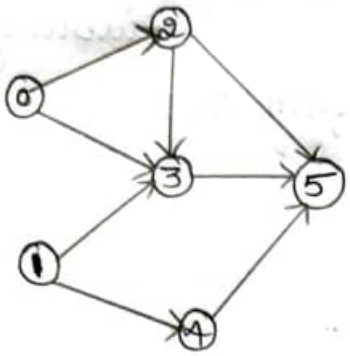
- Let $G=(V,E)$ be a graph where $V \rightarrow$ set of vertices and $E \rightarrow$ set of edges.
- Each vertex is visited each edge is taken from the list. So, time efficiency of accessing each vertex and time efficiency to access the edge is obtained by $|V| + |E|$
- If the given graph is represented using adjacency list representation then the time efficiency to access the vertices and edges is obtained as

$$T(n) = |V| + |E| \quad \text{i.e, } T(n) = \Theta(|V| + |E|)$$

3) Topological sort using source removal method :-

- This method is based on decrease and conquer technique.
- In this method, a vertex with no incoming edges is selected and deleted along with the outgoing edges.
- If there are several vertices with no incoming edges, arbitrarily a vertex is selected
- The order in which the vertices are visited & deleted one by one results in topological sorting.

Eg:- Apply source removal method to solve topological sorting problem for the graph showing tasks to be completed.




Soln:- Corresponding adjacency matrix

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 | 1 | 0 |
| 2 | 0 | 0 | 0 | 1 | 0 | 1 |
| 3 | 0 | 0 | 0 | 0 | 0 | 1 |
| 4 | 0 | 0 | 0 | 0 | 0 | 1 |
| 5 | 0 | 0 | 0 | 0 | 0 | 0 |

| 0 | 0 | 1 | 3 | 1 | 3 |
|---|---|---|---|---|---|

→ Sum of columns

Sum of indegree of each node

- Vertices for which the indegree is 0 are independent task which do not depend on any task and can be computed independently.
- From the matrix, it is clear that
  Indegree [0] = Indegree [1] = 0. Since indegree of jobs 0 & 1 is 0, they are considered independent jobs.
  Indegree [2] = 1, indicates that job 2 depends on 1 job, namely job 0.
  Indegree [3] = 3, indicates that job 3 depends on 3 jobs, namely 0, 1 & 2.
  Indegree [4] = 1, indicates that job 4 depends on only 1 job namely 1
  Indegree [5] = 3, indicates that job 5 depends on 3 jobs, namely 2, 3 and 4.

The various steps involved in the design have to be performed repeatedly till the stack is empty.

- Find the vertices whose indegree is zero and place them on the stack. These vertices denote the jobs which do not depend on any other job and can be performed independently.
- Pop a vertex 'u' and it is the task to be done.

- Add the vertex 'u' to the solution vector. This vertex represents the next job to be considered for executing.
- Find the vertices V adjacent to the vertex 'u'. The vertices V represents the jobs which depend on job 'u'
- Decrement indegree [V] by one thereby reducing the number dependencies on V by one. Since, indegree [V] gives the number of jobs to be completed before job V and job u is completed, dependent jobs of V are reduced by one.

| Stack | U = Pop() | Solution T | V = adj(u) | Indegree of jobs |
|---|---|---|---|---|
| Find indegree of each node from cost adjacency matrix → | | | | [0] [1] [2] [3] [4] [5]<br>0 0 1 3 1 3 |
| 0, 1 | 1 | 1 | 3, 4 | 0 0 1 2 0 3 |
| 0, 4 | 4 | 1, 4 | 5 | 0 0 1 2 0 2 |
| 0 | 0 | 1, 4, 0 | 2, 3 | 0 0 0 1 0 2 |
| 2 | 2 | 1, 4, 0, 2 | 3, 5 | 0 0 0 0 0 1 |
| 3 | 3 | 1, 4, 0, 2, 3 | 5 | 0 0 0 0 0 0 |
| 5 | 5 | 1, 4, 0, 2, 3, 5 | – | — |

Topological sequence

∴ Topological sequence obtained is → 1, 4, 0, 2, 3, 5

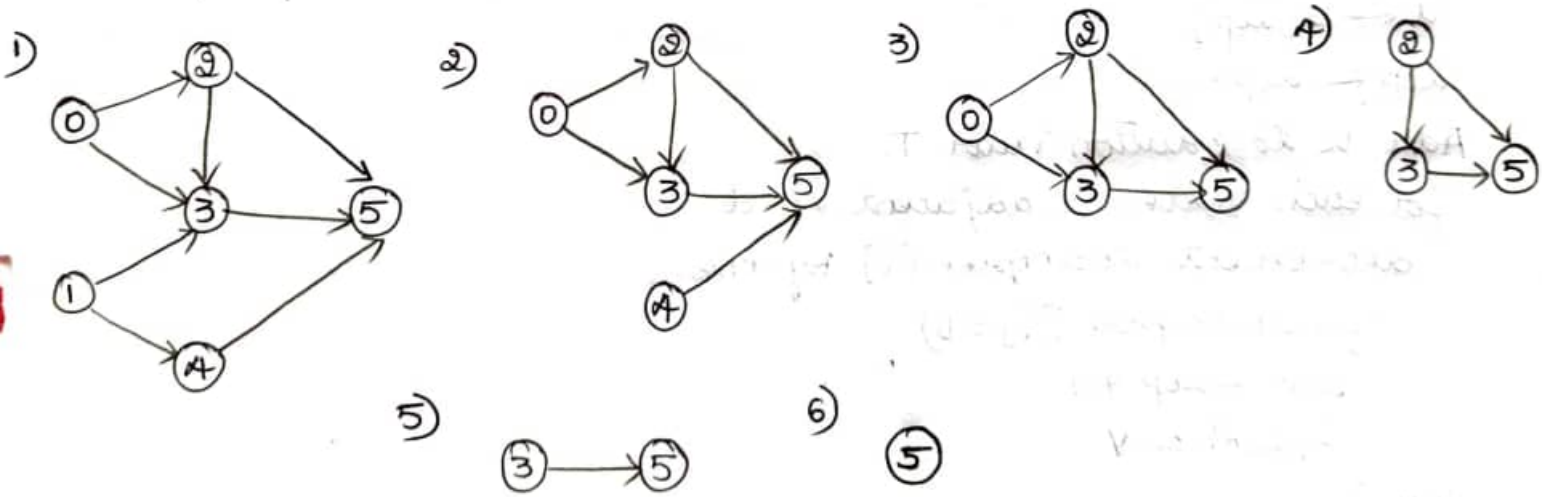



Fig :- Graphs obtained after removing the vertex with indegree zero.

Algorithm :-

```
Algorithm topological sort(a, n, s)
// To obtain the sequence of jobs to be executed resulting in
   topological order
// Input :- a → adjacency matrix
            n → number of vertices.
// Output :- s → indicates the jobs that are to be executed in order
  for j ← 0 to n-1 do
    sum ← 0
    for i ← 0 to n-1 do
      sum ← sum + a[i][j]
    end for
    indegree[j] + sum
  end for
  top ← -1
  for i ← 0 to n-1 do
  if (indegree[i] == 0)
    top ← top + 1
    s[top] ← i
  end if
end for
  while top ≠ 1
    u ← s[top]
    top ← top - 1
    Add u to solution vector T
    for each vertex V adjacent to u
      decrement indegree[v] by one
        if (indegree[v] = 0)
          top ← top + 1
          s[top] ← v
  end if
  end for
end while
  write T
  return
```

## Exhaustive Search :-

- Many important problems require finding an element with a special property in a domain that grows faster with an instant size.
- Typically, such problems arise in situation that involves combinatorial objects such as permutation, combination and subsets of a given set.
- Exhaustive search is simply a brute force approach to combinatorial problems.
- It suggests generating each and every element of the problem's domain selecting those of them that satisfy all the constraints and then finding a desired element.
- Exhaustive search is illustrated by applying it to 2 important problems.
  1) Travelling salesman problem
  2) Knapsack problem.

## Travelling Salesman Problem :- [TSP]

Definition :- Given 'n' cities, a salesperson starts at a specified city (source) visits all n-1 cities only once and returns to the city from where he has started.

- The objective of this problem is to find a route through the cities that minimizes the cost there by maximizing the profit
- This problem can be modeled by an undirected weighted graph as shown, where
- The vertices of the graph represent the various cities.
- Weights associated with edge represent the distances between 2 cities or the cost involved while travelling from one city to other city
- Cost involved from city 'i' to city 'j' is represented using a 2D-array and c[i, j] gives the cost from city 'i' to city 'j'
- Then this problem can be stated as the problem of finding the shortest Hamiltonian circuit of a graph.

- Hamiltonian circuit is defined as the cycle that passes through all the vertices of graph exactly once. All circuit starts and ends at one particular vertex.
- TSP can be solved using 2 methods — Brute force approach and Dynamic programming.

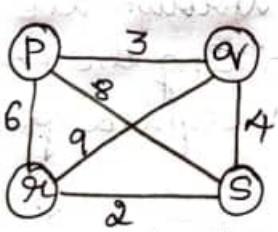
Using brute force approach TSP can be solved as shown below:

1) Get all the routes from one city to another city by taking various permutation
2) Compute the route length for each permutation and select the shortest route among them.

- In this brute force technique, the execution time increases rapidly as the size of the TSP problem increases.
- Easiest way to find an exact solution to TSP is to compute the cost of all tours and determine the tour with the minimum cost. This method is called exhaustive sequential search and uses brute force approach.

Eg :-

Solve the following TSP represented as a graph where the cities are represented as p, q, r & s and the distance between various cities are represented as number in each of the edge.




Sol$^n$ :- cost adjacency matrix is as shown

| | p | q | r | s |
|---|---|---|---|---|
| p | 0 | 3 | 6 | 8 |
| q | 3 | 0 | 9 | 4 |
| r | 6 | 9 | 0 | 2 |
| s | 8 | 4 | 2 | 0 |

- If we assume that the salesperson starts from city 'p', the various routes using which he can visit each and every city exactly once and returns back to the start city 'p' along with the cost incurred for each route is as shown

$p \xrightarrow{3} q \xrightarrow{9} r \xrightarrow{2} s \xrightarrow{8} p$ [cost = 22]

$p \xrightarrow{3} q \xrightarrow{4} s \xrightarrow{2} r \xrightarrow{6} p$ [cost = 15]

$p \xrightarrow{6} r \xrightarrow{9} q \xrightarrow{4} s \xrightarrow{8} p$ [cost = 27]

$p \xrightarrow{6} r \xrightarrow{2} s \xrightarrow{4} q \xrightarrow{3} p$ [cost = 15]

$p \xrightarrow{8} s \xrightarrow{4} q \xrightarrow{9} r \xrightarrow{6} p$ [cost = 27]

$p \xrightarrow{8} s \xrightarrow{2} r \xrightarrow{9} q \xrightarrow{3} p$ [cost = 22]

- Even though we have variety of routes with varying costs, we have to consider the route with minimum cost to get the maximum profit. So, the following routes can be selected by the salesperson.

$p \xrightarrow{3} q \xrightarrow{4} s \xrightarrow{2} r \xrightarrow{6} p$ [cost = 15]

$p \xrightarrow{6} r \xrightarrow{2} s \xrightarrow{4} q \xrightarrow{3} p$ [cost = 15]

Analysis :-

- Let us see the various routes (permutation) that can be used by a travelling salesperson when he visits each and every city exactly once and returns back to the city from where he has started. Let us find out the possible routes taken by the salesperson by assuming he has started from the city 'a'. Let 'n' denote the number of cities to visit.

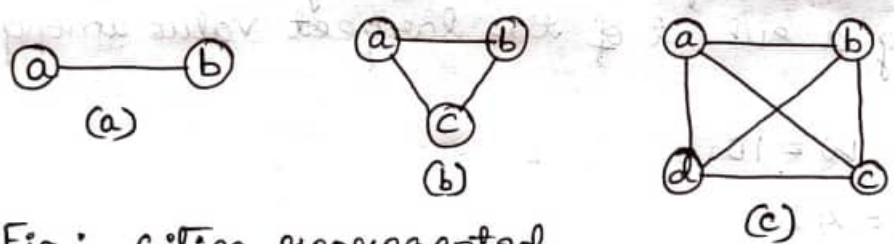



Fig :- Cities represented using graphs.

When n = 2 :-

It is clear from figure that the various routes are

$a \rightarrow b \rightarrow a$ i.e, no of routes = 1 [(2-1)!]

When n = 3 :-

It is clear from figure that the various routes are

$a \to b \to c \to a$
$a \to c \to b \to a$
} 2 routes $[(3-1)!]$

When $n = 4$ :-

It is clear from figure that the various routes are

$a \to b \to c \to d \to a$
$a \to b \to d \to c \to a$
$a \to c \to b \to d \to a$
$a \to d \to b \to c \to a$
$a \to c \to d \to b \to a$
$a \to d \to c \to b \to a$
} 6 routes $[(4-1)!]$

- In general, for 'n' cities number of routes $= (n-1)!$
  i.e, $f(n) = (n-1)!$
- So, time complexity is given by $f(n) \in O(n!)$

## Knapsack Problem :-

Definition :- Given 'n' items of known weights $w_1, w_2 \ldots w_n$ and values $v_1, v_2 \ldots v_n$ and a knapsack of capacity W, find the most valuable subset of the items that fit into the knapsack.

- Exhaustive search approach to this problem leads to generating all the subsets of the set of 'n' items given, computing the total weight of each subset in order to identify feasible subsets [i.e, the ones with the total weight not exceeding the knapsack capacity] and finding a subset of the largest value among them

Eg :- Knapsack capacity, $W = 10$

Item-1 :- $w_1 = 7$ $v_1 = 42$
Item-2 :- $w_2 = 3$ $v_2 = 12$
Item-3 :- $w_3 = 4$ $v_3 = 40$
Item-4 :- $w_4 = 5$ $v_4 = 25$

Sol$^n$ :- Capacity, $W_0 = 10$ i.e, Total weight

| Subset | Total weight | Total value |
|---|---|---|
| {1} | 7 | 42 |
| {2} | 3 | 12 |
| {3} | 4 | 40 |
| {4} | 5 | 25 |
| {1,2} | 7+3 = 10 | 42+12 = 54 |
| {1,3} | 7+4 = 11 | Not feasible, 11>10 |
| {1,4} | 7+5 = 12 | Not feasible, 12>10 |
| {2,3} | 3+4 = 7 | 12+40 = 52 |
| {2,4} | 3+5 = 8 | 12+25 = 37 |
| {3,4} | 4+5 = 9 | 40+25 = 65 |
| {1,2,3} | 7+3+4 = 14 | Not feasible, 14>10 |
| {1,2,4} | 7+3+5 = 15 | Not feasible, 15>10 |
| {1,3,4} | 7+4+5 = 16 | Not feasible, 16>10 |
| {2,3,4} | 3+4+5 = 12 | Not feasible, 12>10 |
| {1,2,3,4} | 7+3+4+5 = 19 | Not feasible, 19>10 |

∴ Subset of the largest value is {3, 4} where the largest value = 65, & weight = 9

Analysis :-

- The number of subsets of an element set is $2^n$, the exhaustive search leads to a time efficiency of

$$f(n) = 2^n \quad \text{i.e, } f(n) \in \Omega(2^n)$$