Module 2

3) Brute Force Approaches:

3.4) Exhaustive Search (Travelling Salesman problem and Knapsack problem)

Exhaustive search is simply a brute force approach to combinatorial problems (such as Permutations, combinations subsets of a given set). It suggests generating each and every element of the problem domain, selecting those of them that satisfy all the constraints, and then finding a desired element (Ex: the one that optimizes some objective function).

Exhaustive search is a algorithm that systematically enumerates all possible solutions to a problem and checks each one to see if it is a valid solution. This algorithm is typically used for problems that have a small and well defined search space, where it is feasible to check all possible solutions.

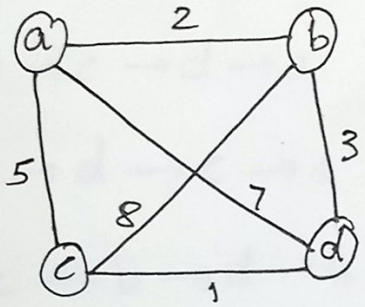
Travelling Salesman Problem : (TSP) $O(n!)$

The problem asks to find the shortest tour through a given set of 'n' cities that visits each city exactly once before returning to the city where it started.

The problem can be conveniently modeled by a weighted graph, with the graph's vertices representing the cities & the edge weights specifying the distances. Then the problem can be stated as the problem of finding the shortest Hamiltonian circuit of the graph.

(A Hamiltonian circuit is defined as a cycle that pass through all the vertices of the graph exactly once.)

Eg:

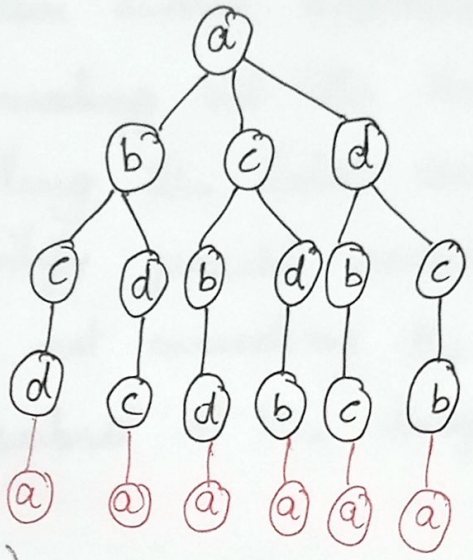


- Undirected weighted graph
- $V = \{a, b, c, d\}$ (4)
- $E = \{a \to b,\ b \to a,\ a \to c,\ c \to a,\ a \to d,\ d \to a,\ b \to c,\ c \to b,\ b \to d,\ d \to b,\ c \to d,\ d \to c\}$ (12)

	a	b	c	d
a	0	2	5	7
b	2	0	8	3
c	5	8	0	1
d	7	3	1	0

← distance matrix
(distance travelled from one node/city to another node)

All the circuits start and end at one particular vertex. Thus, we can get all the tours by generating all the permutations of n-1 intermediate cities, compute the tour lengths, and find the shortest among them.



o(n)

· Tree representation of all possible tours

o(n!)

Tour	length
$a \to b \to c \to d \to a$	$l = 2+8+1+7 = 18$
✓ $a \to b \to d \to c \to a$	$l = 2+3+1+5 = (11)$ — optimal
$a \to c \to b \to d \to a$	$l = 5+8+3+7 = 23$
✓ $a \to c \to d \to b \to a$	$l = 5+1+3+2 = (11)$ — optimal
$a \to d \to b \to c \to a$	$l = 7+3+8+5 = 23$
$a \to d \to c \to b \to a$	$l = 7+1+8+2 = 18$

① Knapsack Problem :

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.

The exhaustive search approach to knapsack 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:

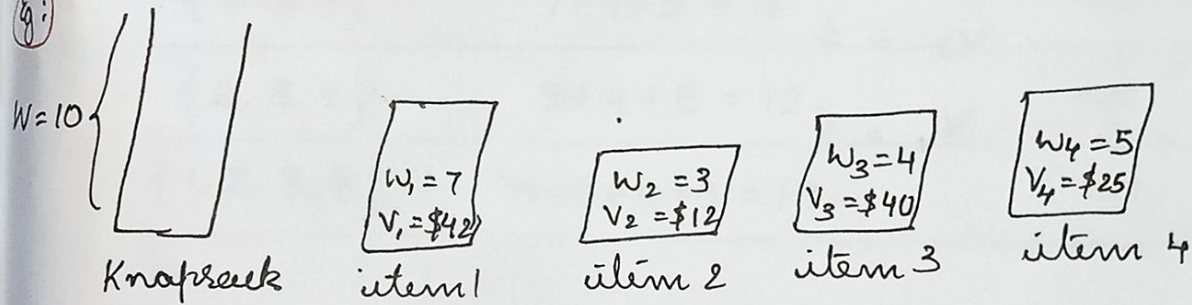


Fig : Instance of the knapsack problem.

Subset	Total weight	Total value
ϕ	0	\$0
{1}	7	\$42
{2}	3	\$12
{3}	4	\$40
{4}	5	\$25

Subset	Total weight	Total value
{1, 2}	7+3 = 10	42+12 = $54
{1, 3}	7+4 = 11	not feasible
{1, 4}	7+5 = 12	not feasible
{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
{1, 2, 4}	7+3+5 = 15	not feasible
{1, 3, 4}	7+4+5 = 16	not feasible
{2, 3, 4}	3+4+5 = 12	not feasible
{1, 2, 3, 4}	7+3+4+5 = 19	not feasible

Subset of the largest value

The number of subsets of an n-element set is 2^n, the exhaustive search leads to a $\Omega(2^n)$ algorithm, no matter how efficiently subsets are generated.

Thus for both TSP and Knapsack problem, exhaustive search leads to algorithm that are extremely inefficient on every input. These two problems are the best-known examples of so called NP-hard problems.

4. Decrease-and-Conquer :

The Decrease-and-conquer technique is based on exploiting the relationship between a solution to a given instance of a problem and a solution to its smaller instance.

Once such relationship is established, it can be exploited either top down or bottom up.

Top-down approach leads to a recursive implementation. Bottom-up approach leads to iterative implementation starting with a solution to the smallest instance of the problem; it is called sometimes the incremental approach.

There are three major variations of decrease-and-conquer.

1. decrease by a constant
2. decrease by a constant factor
3. variable size decrease

1) decrease by a constant- variation : the size of an instance is reduced by the same constant on each iteration of the algorithm.

Typically, this constant is equal to one, although the constant size reductions do happen occasionally.

Eg: Insertion sort, DFS, BFS, Topological sorting

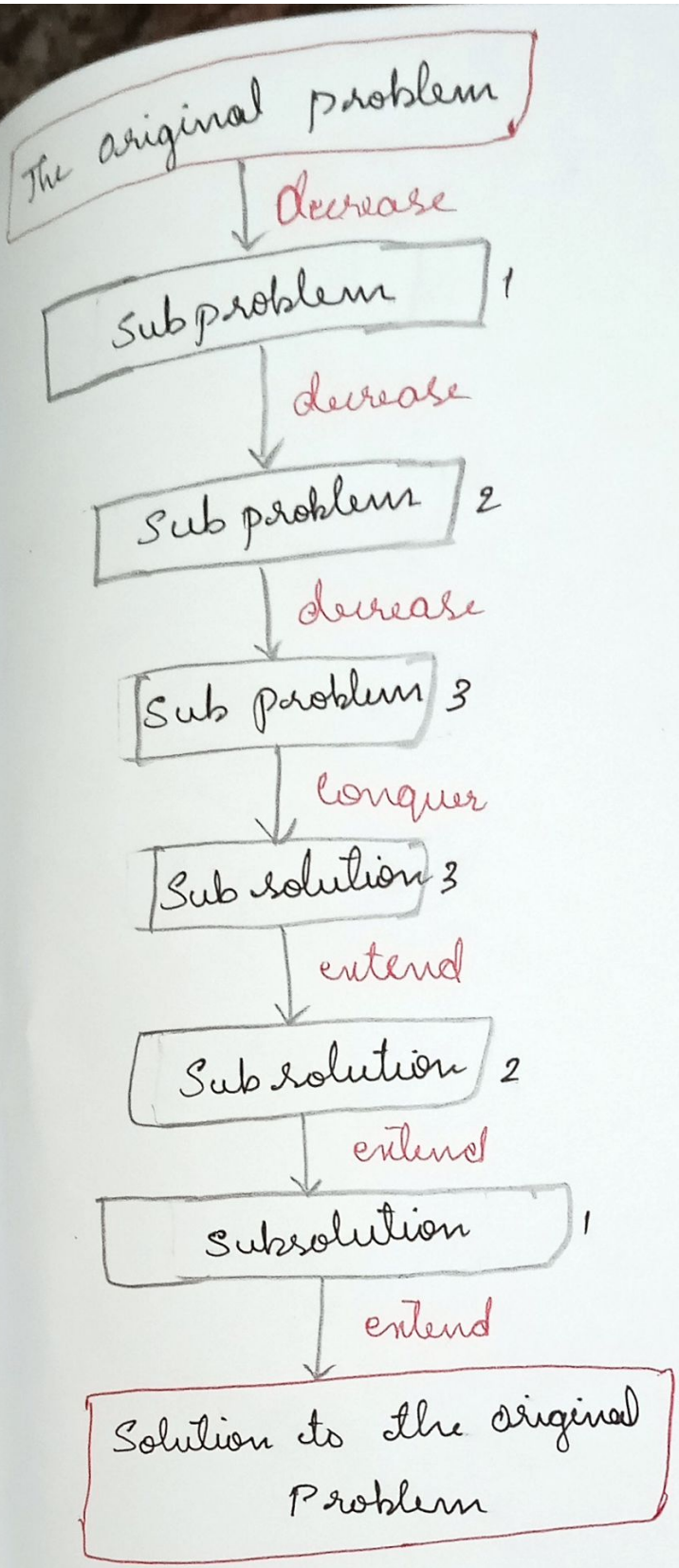
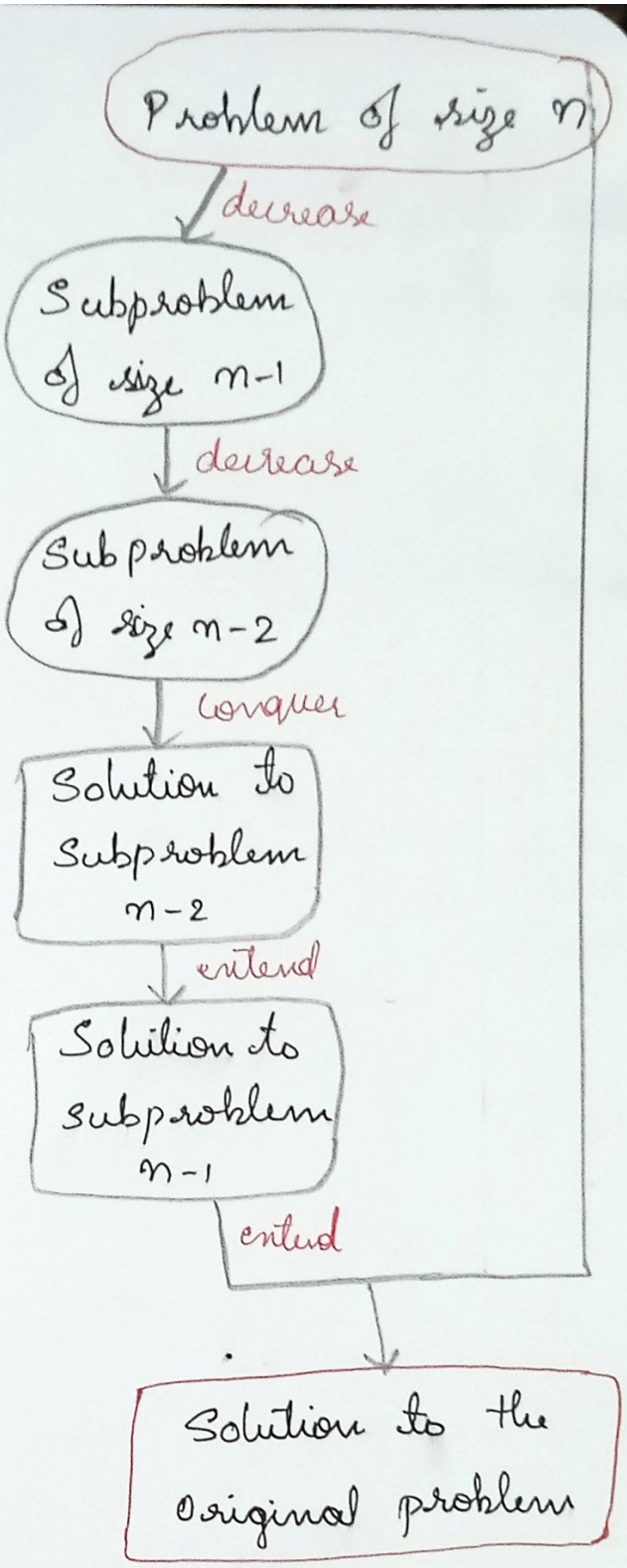


Fig: Decrease-and-conquer



Decrease-(by one)-and-conquer technique

- Decrease or reduce problem instance into smaller instance of the same problem and extend solution.
Conquer the problem by solving smaller instance of the Problem. Extend solution of smaller instance to obtain

solution to original problem.

2. decrease-by-a-constant-factor technique suggests reducing a problem instance by the same constant factor on each iteration of the algorithm.

Typically, this constant factor is equal to two. A reduction by a factor other than two is especially rare.

Eg: Binary Search, Fake coin problems.

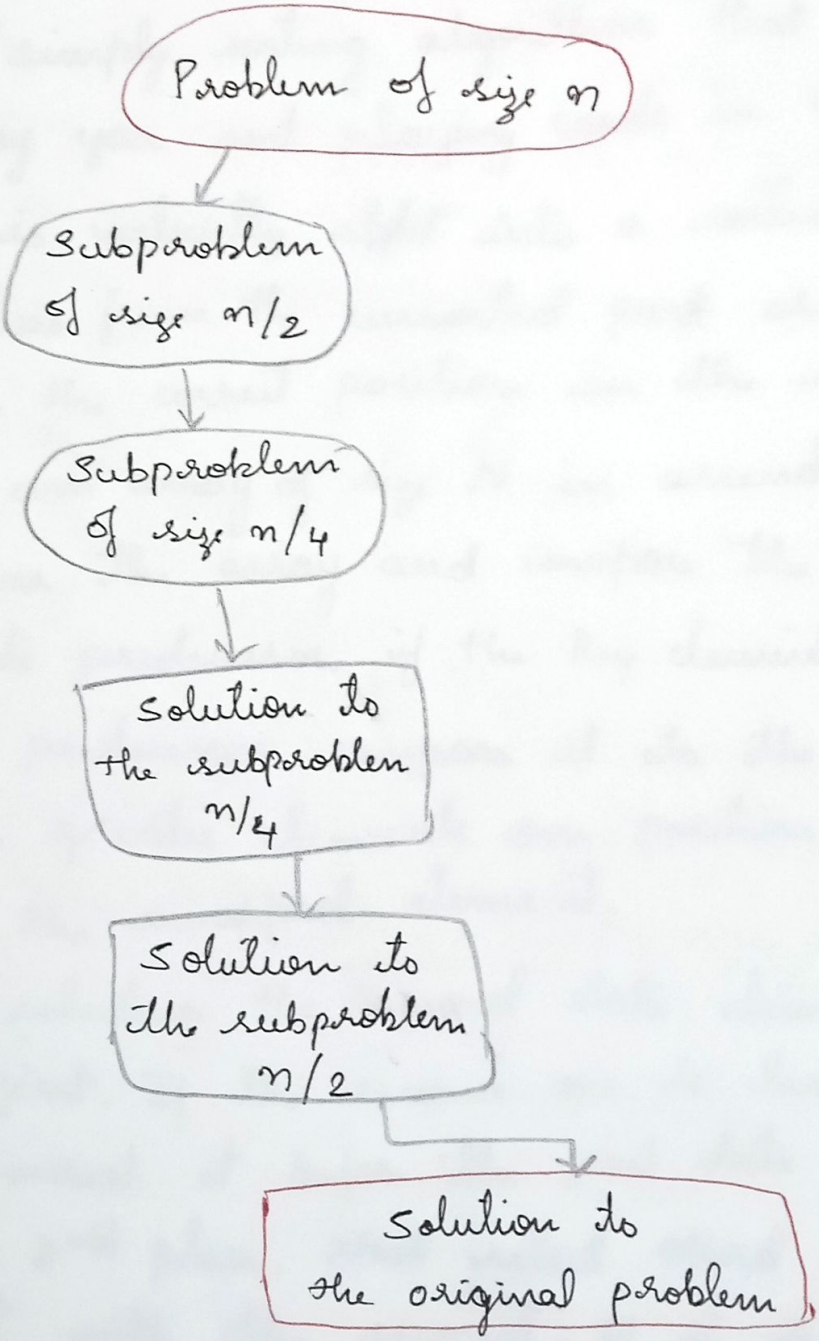


Fig: decrease-(by half)-and-conquer technique

3. Variable-size-decrease, the size-reduction pattern varies from one iteration of an algorithm to another.

Ex. • Euclid's algorithm, $gcd(m, n) == gcd(n, m \bmod n)$
• Computing median & selection problem
• Interpolation Search

4.1 Insertion sort :

It is a simply sorting algorithm that works similarly to the way you sort playing cards in your hands. The array is virtually split into a sorted & unsorted part. Values from the unsorted part are picked and placed in the correct position in the sorted part.

To sort an array of size N in ascending order iterate over the array and compare the current element (Key) to its predecessor, if the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.

• Start by selecting the second data item & compare it with the first. If the second one is less than the first one then insert it before the first data item. Otherwise insert in 2nd place. Next select third data item and compare it with the second. If it is less than the

second then compare it with the first. If it is less than the first one insert it before the first data item. otherwise insert it in between first and second data item. Repeat the steps N-1 times. The entire list get sorted within $(N-1)^{th}$ pass.

Algorithm InsertionSort(A[0..n-1])

// Sorts a given array by insertion sort

// Input : An array A[0..n-1] of n orderable elements

// Output : Array A[0..n-1] sorted in nondecreasing order

for i ← 1 to n-1 do

    v ← A[i]

    j ← i-1

    while j ≥ 0 and A[j] > v do

        A[j+1] ← A[j]

        j ← j-1

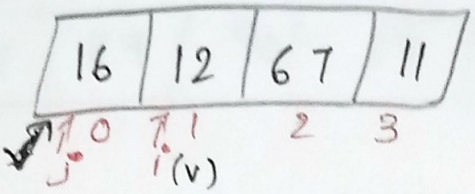
    A[j+1] ← v

- $A[0] \leq \ldots \leq A[j] < A[j+1] \leq \ldots \leq A[i-1] \mid A[i] \ldots A[n-1]$

  Smaller than or equal to A[i] | greater than A[i]

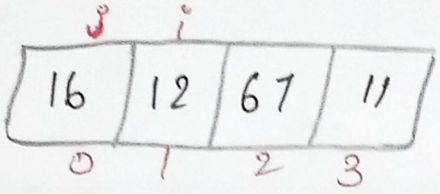
- A[i] is inserted in its proper position among the preceding elements previously sorted.

(a) Working



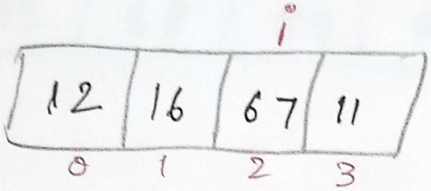
A[j] > V ? A[j+1] ← A[j]
j ← j - 1
A[j+1] ← V

V = 12



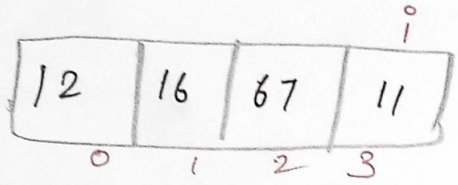
16 > 12 ? Yes
insert 12 in 1st place

V = 67

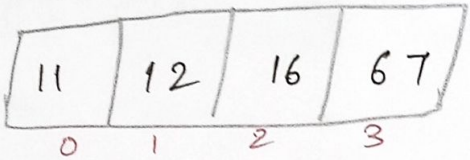


16 67
16 > 67 ? No

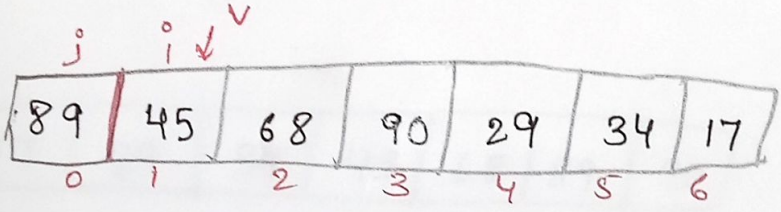
V = 11



67 > 11 ? Yes
16 > 11 ? Yes
12 > 11 ? Yes
insert 11 in 1st place

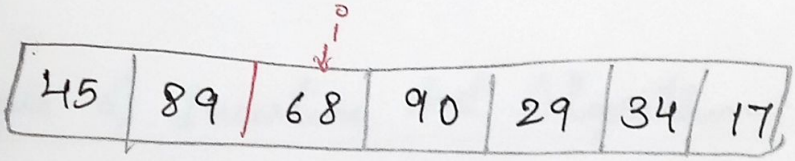


2)



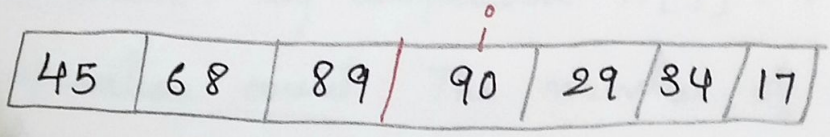
V = 45

→ 89 > 45 ? Yes
Insert 45 in 1st place



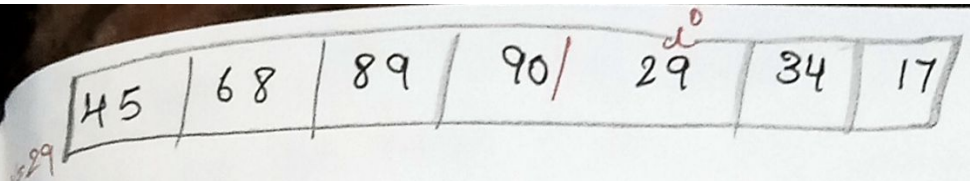
V = 68

→ 89 > 68 ? Yes
45 > 68 ? No
Insert 68 inbetween 45 & 89

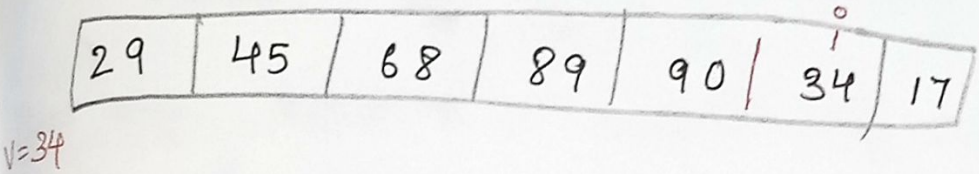


V = 90

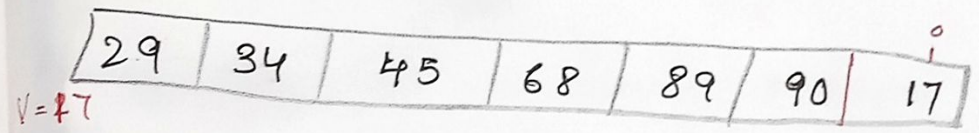
→ 89 > 90 ? No



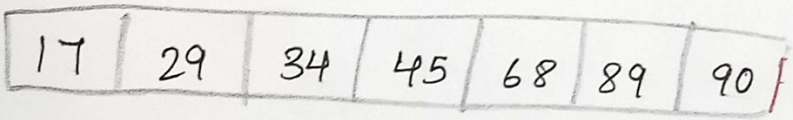
→ 90 > 29? Yes
89 > 29? Yes
68 > 29? Yes
45 > 29? Yes
Insert 29 in 1st position



→ 90 > 34? Yes
89 > 34? Yes
68 > 34? Yes
45 > 34? Yes
29 > 34? No
Insert 34 inbetween 29 & 45



→ 90 > 17? Yes
89 > 17? Yes
68 > 17? Yes
45 > 17? Yes
34 > 17? Yes
29 > 17? Yes
Insert 17 in 1st place



- **Analysis of Insertion Sort Algorithm:**

Input: The input size is given by the number of elements n.

Basic operation: Key comparison A[j] > V

Basic operation count: The number of key comparisons (basic operation in the algorithm doesn't only depend on the size of the input, it also depends on nature of the input.

Worst case : $A[j] > V$ is executed the largest number of times, i.e for every $j = i-1, \dots 0$ (inner loop). Since $V = A[i]$. it happens iff $A[j] > A[i]$ for $j = i-1, \dots 0$ (outer loop)

$$C_{worst}(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} \in \boxed{\Theta(n^2)}$$

Best case : $A[j] > V$ is executed only once on every iteration of the outer loop. It happens iff $A[i-1] \leq A[i]$ for every $i = 1, \dots n-1$, i.e if the input array is already sorted in non decreasing order.

$$C_{best}(n) = \sum_{i=1}^{n-1} 1 = n-1 \in \boxed{\Theta(n)}$$

Average case : on randomly ordered arrays, insertion sort makes an average half as many comparisons as on decreasing arrays, i.e

$$C_{avg}(n) \approx \frac{n^2}{4} \in \boxed{\Theta(n^2)}$$

4.2 Topological Sorting:

Few Basic facts:

A directed graph or digraph, is a graph with directions specified for all its edges. The adjacency matrix and adjacency lists are two principal means of representing a diagraph.

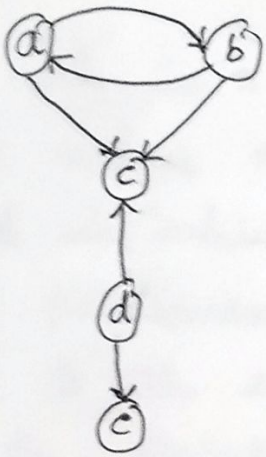


fig (a)
Digraph

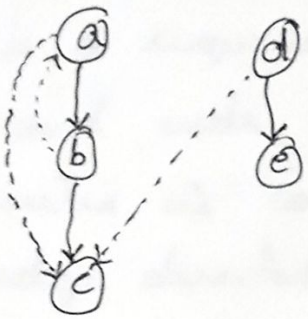


fig (b)
DFS forest of the digraph for the DFS traversal started at a.

- Depth-first search (DFS) & Breadth-first search (BFS) are principle traversal algorithms for traversing diographs.
- Forest: an undirected, disconnected, acyclic graph. A disjoint collection of trees is known as forest. Each component of a forest is tree.
- Tree: an undirected, connected and acyclic graph. A connected graph that doesn't contain even a single cycle is called a tree.
- The DFS forest (fig (b)) exhibits all four types of edges possible in a DFS forest of a directed graph:

- tree edges : (ab, bc, de)
- back edges : (ba) from vertices to their ancestors
- forward edges : (ac) from vertices to their descendants in the tree other than their children
- Cross edges (dc) : it is an edge that connect two nodes such that they do not have any ancestor & a descedant relation between them.

- A directed cycle in a digraph is a sequence of three or more of its vertices that starts and ends with the same vertex and in which every vertex is connected to its immediate predecessor by an edge directed from the predecessor to the successor. For ex, a, b, a is a directed cycle in the digraph in fig(a).
- If a DFS forest of a digraph has no back edges, the digraph is a directed acyclic graph (dag).
- Topological Sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge (u, v), vertex u appears before v in the ordering.
- Topological Sorting 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.
- Topological sorting is not possible for cyclic graph.

Q) Why Topological sort is not possible for graphs with undirected edges?

A. This is due to the fact that undirected edge between two vertices u & v means, there is an edge from u to v as well as from v to u. Because of this, both the nodes u and v depend upon each other & none of them can appear before the other in the topological ordering or linear ordering without creating a contradiction.

Q. Why Topological sort is not possible for graphs having cycles?

A. If we try to topologically sort the cyclic graph starting from any vertex, it will always create a contradiction to the definition of topological sort. All the vertices in a cycle are indirectly dependent on each other, hence topological sorting fails.

- Topological sorting can be done using two methods:
  1) DFS method
  2) Source removal method (Vertex deletion method)

① Topological Sort using DFS method:

- Steps involved:
  1. Create a graph with n vertices and m directed edges.
  2. (Initialize a Stack and select any arbitrary vertex)
  3. For each unvisited vertex in the graph, do:
     - Call the DFS function with the vertex as the parameter

- In the DFS function, mark the vertex as visited and recursively call the DFS function for all unvisited neighbors of the vertex.
- Once all the neighbors have been visited, push the vertex onto the stack.

(When a vertex is visited for the first time it is pushed onto the stack)

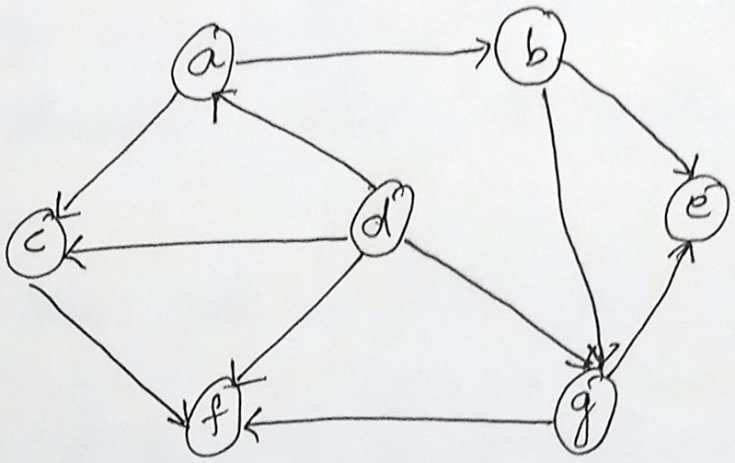
4. After all vertices have been visited, pop elements from the stack & append them to the output list until the stack is empty.

(When a vertex becomes a dead end, it is removed (Popped) from the stack) & backtrack.

(Reverse the order of deleted items (Popped) to get the topological sequence)

5. The resulting list is the topologically sorted order of the graph.

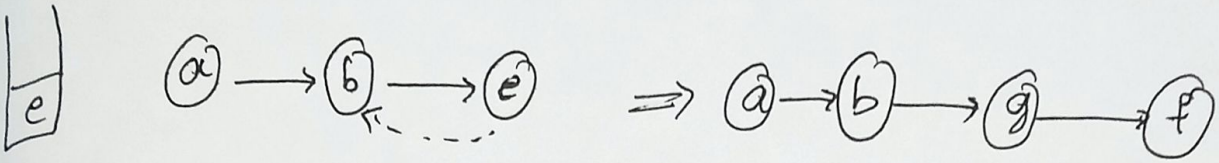
working eg:



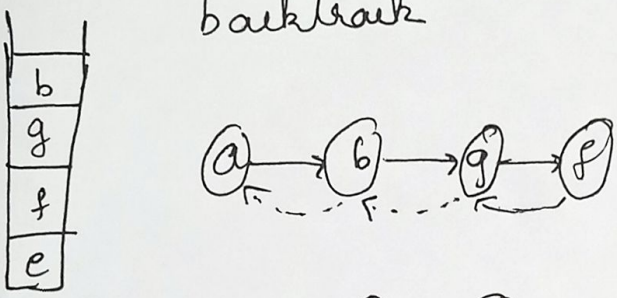
DFS method ①, when we reach dead end, push vertex onto stack & backtrack & so on.

Lets start from (a), DFS(a)

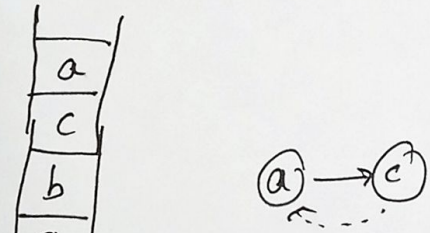
(a) → (b) → (e) - we can't go further, push (e) onto stack and backtrack



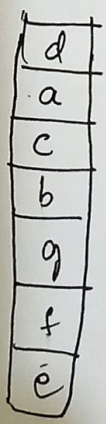
- we can't go further, push (f) onto stack & backtrack



- Next (a) → (c), we can't go further, (f) is already visited, push (c) onto stack & backtrack



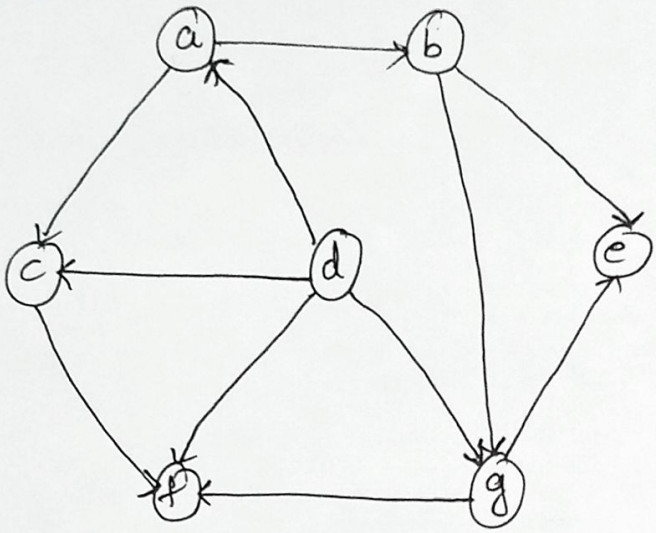
- Next vertex remaining in (d), from (d) we can't go any further because nodes (a), (c), (f), (g) are already visited, so push (d) onto stack.



- Now pop elements from stack and append.

d, a, c, b, g, f, e Topological order

The topological ordering is non-unique.



	d	a	b	c	g
in(a) =	1	0			
in(b) =	1	0	0		
in(c) =	2	1	0	0	
in(d) =	0				
in(e) =	2	2	2	1	1
in(f) =	3	2	2	2	1
in(g) =	2	1	0	0	0

Remove (d)

Remove (a)

Remove (b)

Remove (c)

Remove (g)

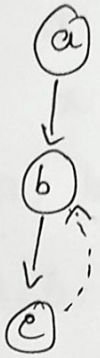
Remove (e)

Remove (f)

d a b c g e f

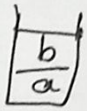
Eg Working Topological ordering using DFS method (2).
as & when vertex are visited, push them onto stack, when deadend is reached, pop them and reverse the popped order to get the topological ordering.

- Lets start from (a)

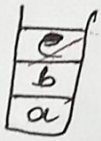


Push (a) onto stack

Push (b) onto stack

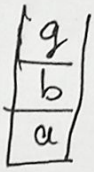


Push (c) onto stack
Can't go further, pop (c)
& backtrack

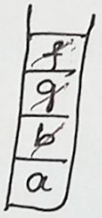


c

Push (g) onto stack

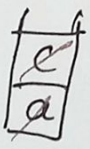


Push (f) onto stack
Can't go further, pop (f)
backtrack, pop (g),
Pop (b)



f
g
b

Push (c) onto stack
Can't go further, (f) is already visited, pop (c)
Pop (a)



c
a

(d)

Push (d) onto stack,
Pop (d)

d

Popped order : e, f, g, b, c, a, d

Now reverse the order : d, a, c, b, g, f, e topological order

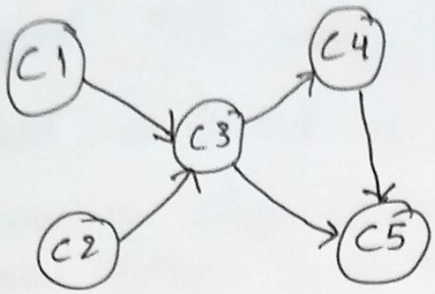
This can also be represented as below :

Stack(Push)	top	nodes visited	Pop
a	-	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, g, f	f
a, b, g	-	a, b, e, g, f	g
a, b	-	a, b, e, g f	b
a	c	a, b, e, g, f, c	-
a, c	-	a, b, e, g, f, c	c
a	-	a, b, e, g, f, c	a
d	-	a, b, e, g, f, c, d	d

Popped order : e, f, g, b, c, a, d

Reverse order : d, a, c, b, g, f, e topological order

Eg: 2

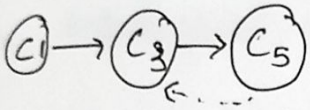


- Topological sorting using DFS method.

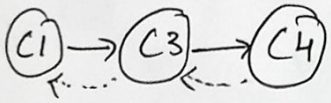
DFS method (1)

deadend, push vertex onto stack after all, pop

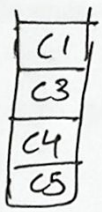
- Start with (C1)



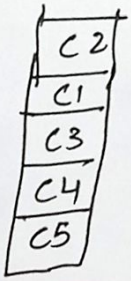
After (C5) Backtrack
Push (C5)



After (C4), push (C4)
Backtrack, push (C3)
Backtrack, push (C1)



(C2) push (C2)



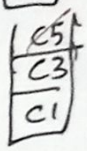
Pop → C2, C1, C3, C4, C5

DFS method (2)

visit vertex & push onto stack, reach deadend & pop & reverse

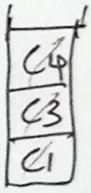
- Start with (C1)

Push (C1)
Push (C3)
Push (C5)



Pop (C5) Backtrack C5

Push (C4), Backtrack



Pop (C4), Backtrack C4
Pop (C3), Backtrack C3
Pop (C1) C1

Push (C2)
Pop (C2) C2

Popped order: C5, C4, C3, C1, C2
Reverse order: C2, C1, C3, C4, C5

(2.) Topological Sort using Source Removal method

. This method is based on removing a vertex which does not have incoming edges (zero indegree) - called as source vertex.

Note down indegree of all the nodes/vertex.

1) Pick a vertex (source vertex) with indegree 0.

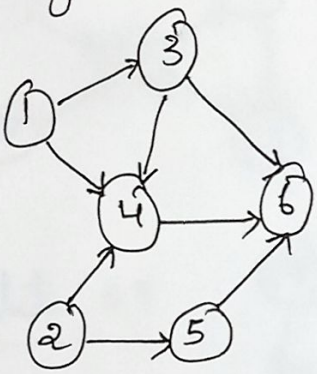
2) Remove source vertex & all its outgoing edges. & decrement indegree of its dependant nodes.

3) Repeat until graph is empty.

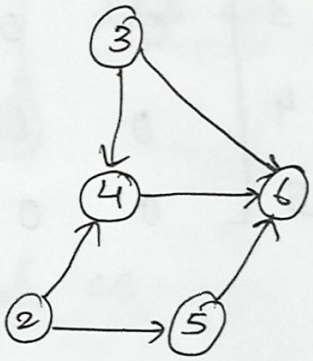
The sequence generated by this removal process will form the topological ordering.

Eg. Working

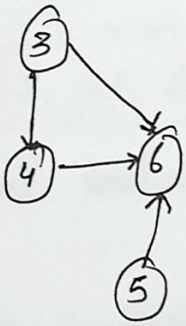
1.



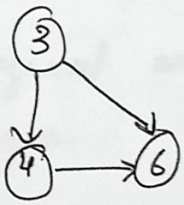
Remove (1) & its outgoing edges



Remove (2)



Remove (5)



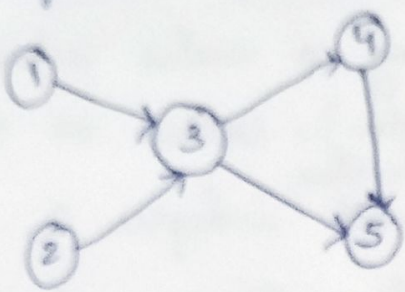
Remove (3)

(4) → (6) Remove (4) (6) Remove (6)

Sequence of source vertex removal method is:

1, 2, 5, 3, 4, 6 Topological order

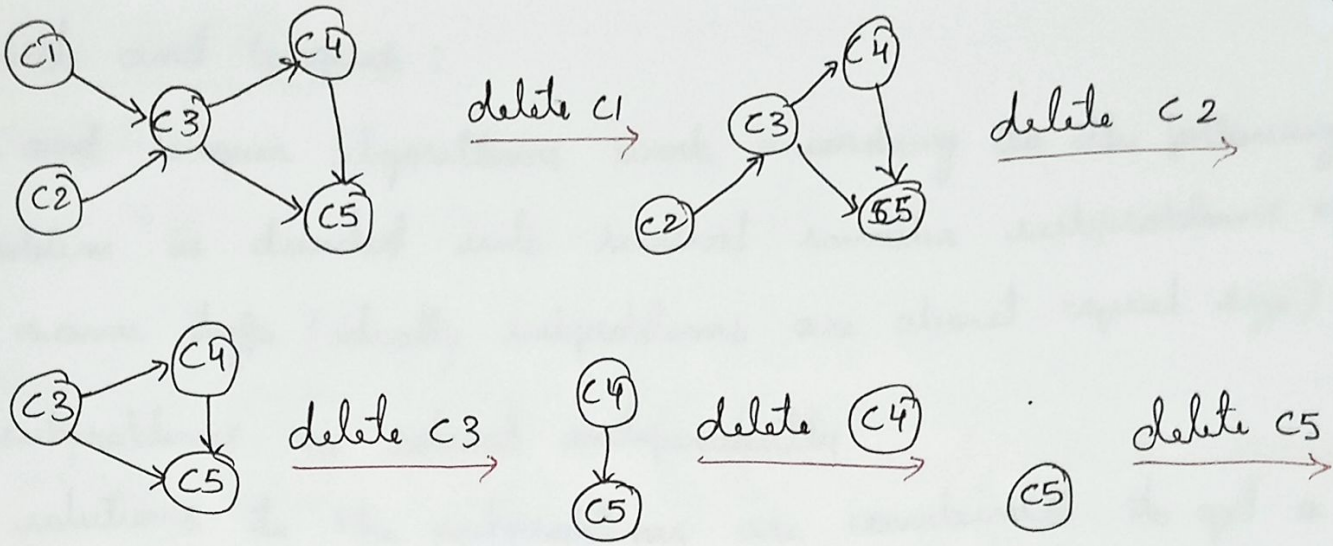
Eg: Topological Sorting / ordering:



Adjacency matrix representation:

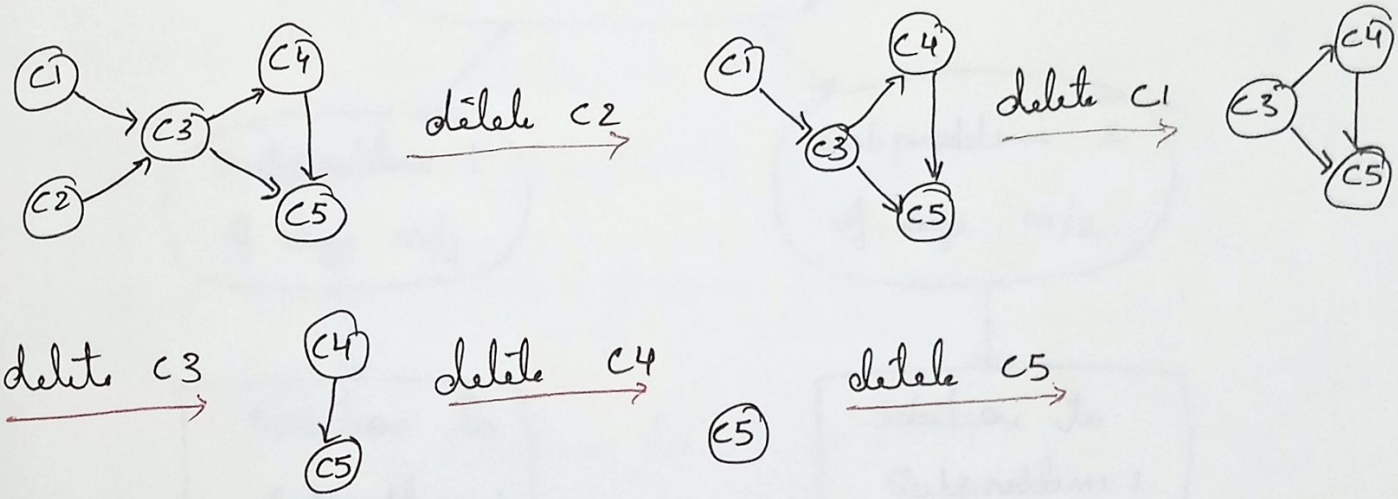
	1	2	3	4	5
1	0	0	1	0	0
2	0	0	1	0	0
3	0	0	0	1	1
4	0	0	0	0	1
5	0	0	0	0	0

Eg 2



Sequence : C1, C2, C3, C4, C5 (Topological order)

OR



- Sequence : C2, C1, C3, C4, C5 (Topological order)
- Hence topogical ordering is non-unique.

1) Note down the indegree of all the nodes
2) Place the node/nodes which has indegree of 0.
3) (deleting the vertex) decrement the indegree of those nodes where there was an incoming edge from the node placed in step 2.
4) Repeat step 2 & 3 until all nodes are placed.

5. Divide and Conquer :

Divide and conquer algorithms work according to the following:

1. A problem is divided into several smaller subproblems of the same type (ideally subproblems are about equal size)
2. The subproblems are solved independently.
3. The solutions to the subproblems are combined to get a solution to the original problem.

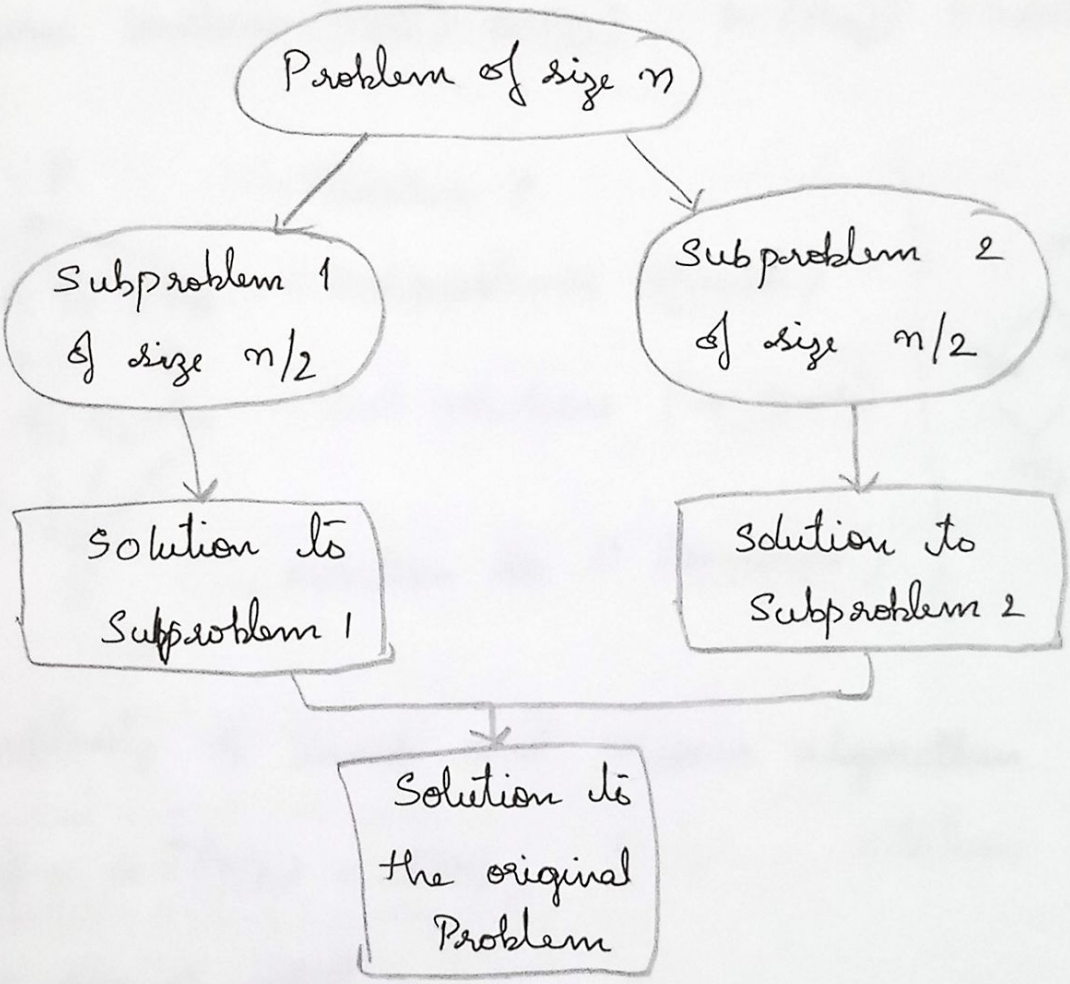
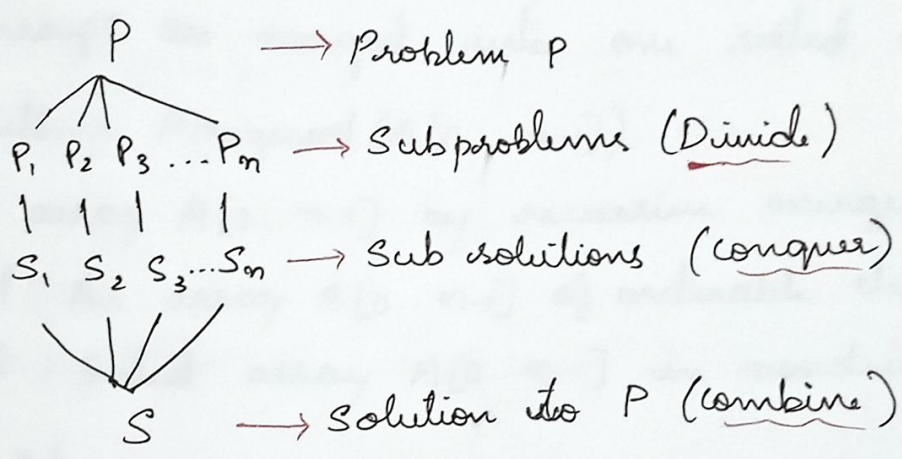


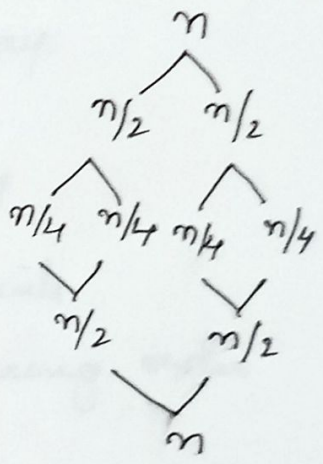
Fig: Divide and conquer technique (typical case)

Applications : Binary search, Quick Sort, Merge sort, Strassen's Matrix multiplication, Binary search Traversals

Algorithm DC(P)

```
if Small(P)        // if P is too small
return S(P)        // return solution of P
else
    divide(P) into P1, P2, .... Pn
    Apply DC(P1), DC(P2) ... DC(Pn)    // Apply DC to subproblems
    Compute DC(P1), DC(P2) ..., DC(Pn)
    return Combine(DC(P1), DC(P2) ... DC(Pn))    // solution to P
```





- Time complexity of Divide and conquer algorithm:

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

where, n = size of input

a = number of subproblems in the recursion

n/b = size of each subproblems

$f(n)$ = cost of the work done outside the recursive call,
(cost of dividing the problem into subproblems,
cost of merging or combining the sub solutions)

5.1 Mergesort

- It sorts a given array $A[0...n-1]$ by dividing it into two halves $A[0..., (n/2)-1]$ and $A[(n/2), ..., n-1]$, sorting each of them recursively, and then merging the two smaller sorted arrays into a single sorted one.
- Merge sort is a recursive algorithm that continuously splits the array in half until it cannot be further divided i.e, the array has only one element left (an array with one element is always sorted). Then the sorted sub arrays are merged into one sorted array.
- Algorithm Mergesort $(A[0...n-1])$

// Sorts array $A[0..n-1]$ by recursive mergesort

// Input: An array $A[0...n-1]$ of orderable elements

// output: Sorted array $A[0..n-1]$ in nondecreasing order

if $n > 1$

Copy $A[0,..,\lfloor n/2 \rfloor - 1]$ to $B[0,....,\lfloor n/2 \rfloor - 1]$ // $\lfloor \rfloor$ Floor value of $n/2$

Copy $A[\lfloor n/2 \rfloor, ..., n-1]$ to $C[0,....,\lceil n/2 \rceil - 1]$

Mergesort $(B[0,.., \lfloor n/2 \rfloor - 1])$

Mergesort $(C[0,..., \lceil n/2 \rceil - 1)$

Merge (B, C, A)

Algorithm Merge ($B[0..p-1]$, $C[0..q-1]$, $A[0..p+q-1]$)

// Merges two sorted arrays into one sorted array

// Input : Arrays $B[0..p-1]$ and $C[0..q-1]$ both sorted

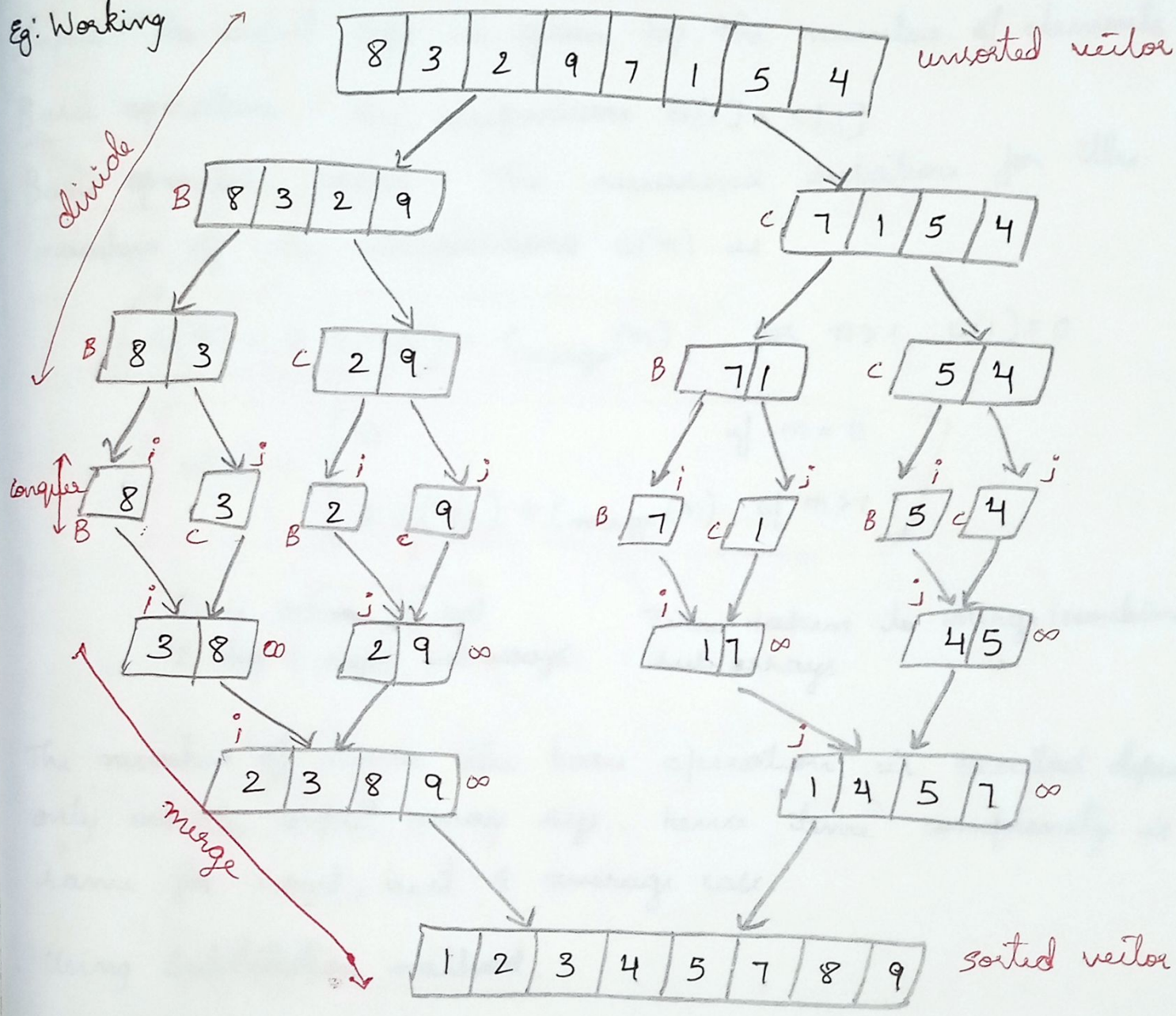
// output : Sorted array $A[0..p+q-1]$ of the elements of B & C

```
i ← 0 ; j ← 0 ; K ← 0
while i < p and j < q do
    if B[i] ≤ C[j]
        A[K] ← B[i];        // copy the lowest element of B to A
        i ← i+1             // point to next element in B
    else
        A[K] ← C[j];        // copy the lowest element of C to A
        j ← j+1             // point to next element in C
    K ← K+1                 // point to next item in A
if i = p
    copy C[j..q-1] to A[K..p+q-1]   // copy the remaining items from part of C to A
else
    copy B[i..p-1] to A[K..p+q-1]   // copy the remaing items from part of B to A
```

- Two pointers are initialized to point the first elements of the arrays being merged (B & C). The elements pointed to are compared, & the small of them is added to a new array being constructed, after that, the index of smaller element is incremented to point to its immediate successor (next element). This is repeated until one of the array is exhausted, then the remaining elements of other array are

copied to the end of the new array.

Eg: Working



- Compare i & j, put the lesser value in K

B[i] C[j] A[K]

B[i] < C[j]	Yes A[K] ← B[i]	No A[K] ← C[i]	A[ ]
8 < 3 ?		No	3 8
2 < 9 ?	Yes		2 9
7 < 1 ?		No	1 7
5 < 4 ?		No	4 5

• Analysis of Merge Sort Algorithm :

Input : The input size is given by the number of elements n.

Basic operation : Key comparison $B[i] < C[j]$

Basic operation count : The recurrence relation for the number of key comparisons $C(n)$ is

$$C(n) = 2\,C(n/2) + C_{merge}(n) \quad \text{for } n > 1,\ C(1) = 0$$

$$C(n) = \begin{cases} 0 & \text{if } n = 0 \\ 2\,C(n/2) + C_{merge}(n) & \text{if } n > 1 \end{cases}$$

Time taken to sort 2 left & right sub arrays

Time taken to merge / combine sub arrays

The number of times the basic operation is executed depends only on the input array size, hence time complexity is same for worst, best & average case.

• Using Substitution method :

$$C(n) = 2\,C(n/2) + cn \quad \text{---(1)} \qquad cn \Rightarrow C_{merge}(n)$$

$$C(n/2) = 2\,C(n/4) + cn/2 \quad \text{---(2)}$$

$$C(n/4) = 2\,C(n/8) + cn/4 \quad \text{---(3)}$$

By substituting

$$C(n) = 2.(2\,C(n/4) + cn/2) + cn$$

$$= 4\,C(n/4) + 2cn$$

$$= 4(2\,C(n/8) + c\,n/4) + 2cn$$

$$= 8\,C(n/8) + 3cn$$

$$\vdots$$

$= 2^3 c(n/2^3) + 3cn$

Put $2^k = n$

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

$= n\, c(n/n) + \log_2 n$ $\qquad 2^k = n$

$= n\, c(1) + \log_2 n\, cn$ $\qquad k = \log_2 n$

$= n.0 + cn \log_2 n$

$c(n) = \boxed{O(n \log n)}$

Eg: ① 4, 9, 0, -1, 6, 8, 9, 2, 3, 12

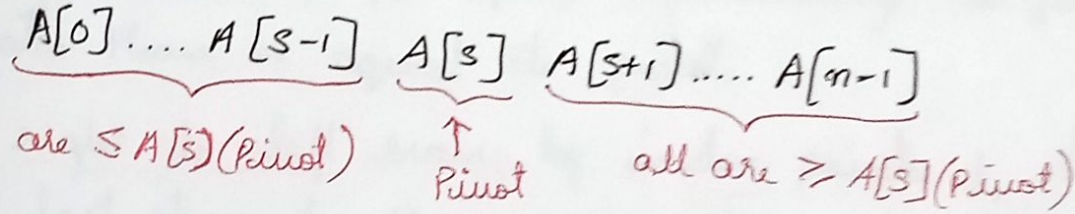
② 67, 90, 12, 56, 23, 34, 45

③ 60, 50, 25, 10, 35, 25, 75, 30

④ E, X, A, M, P, L, E

5.2 Quick Sort

- Quick Sort is a sorting algorithm that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.
- Quick sort divides its input elements according to their value, it partitions its array elements such that all the elements to the left of some element A[s] (Pivot - an element with respect to whose value we are going to divide the subarray) are less than or equal to A[s], and all the elements to the right of A[s] (Pivot) are greater than or equal to it.



After a partition is achieved, A[s] (Pivot) will be in its final position in the sorted array, & continue sorting the two subarrays to the left and right of A[s] (pivot) independently (recursively sorting).

Algorithm Quicksort (A[l...r])
// Sorts a subarray by quicksort
// Input : Subarray of array A[0..n-1], defined by its left and right indices l and r
// Output : Subarray A[l...r] sorted in nondecreasing order

if $l < r$

$s \leftarrow$ Partition $(A[l \ldots r])$ // s is a split position (Pivot)

Quicksort $(A[l \ldots s-1])$

Quicksort $(A[s+1 \ldots r])$

There are several different strategies for selecting a pivot, we use the simplest strategy of selecting the subarray's first element : $P = A[l]$.

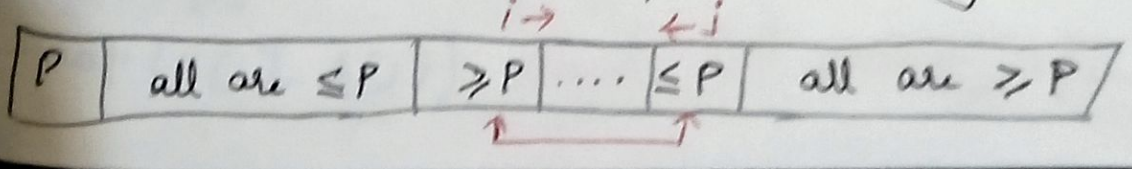
We now scan the subarray from both ends, comparing the subarray's element to the pivot.

The left to right scan, by index pointer i, starts with second element, this scan skips over smaller elements than pivot & stops upon encountering the first element greater than or equal to pivot.

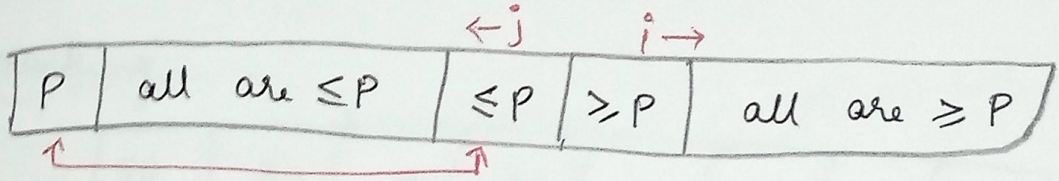
The right to left scan, by index pointer j, starts with the last element, this scan skips over larger elements than pivot & stops on encountering the first element smaller than or equal to pivot.

After both scans stop, three situations may arise, depending on ~~with~~ whether or not the scanning indices have crossed.

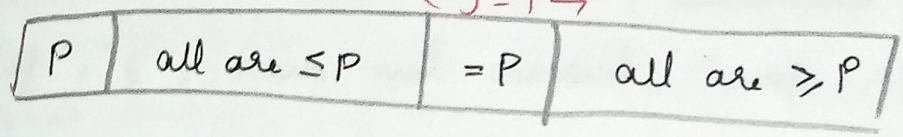
(1) If i and j have not crossed, i.e $i < j$, exchange $A[i]$ & $A[j]$ and resume the scans by incrementing i & decrementing j.



2) If i & j have crossed over, i.e i > j, swap pivot with A[j] and partition the subarray.



3) If i & j points to the same element, i.e i = j, split position s = i = j;



Algorithm HoarePartition (A[l...r])

// Partitions a subarray by Hoare's algorithm, using the first element as a pivot

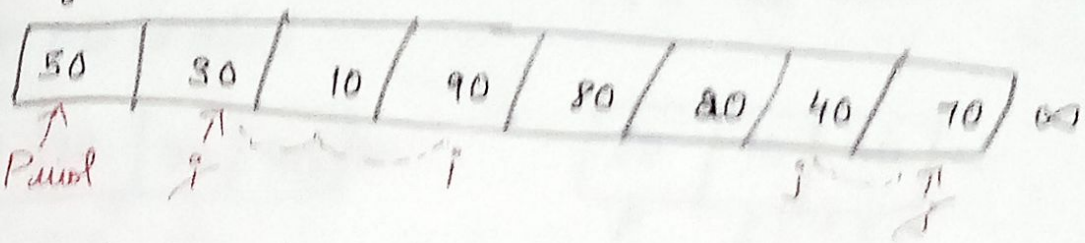
// Input: Subarray of array A[0...n-1], defined by its left- and right indices l and r (l < r)

// Output: Partition of A[l...r], with the split position returned as this function's value

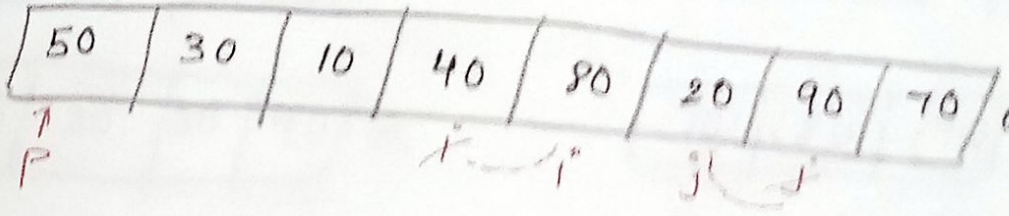
```
P ← A[l]
i ← l ; j ← r+1
repeat
    repeat i ← i+1 until A[i] ≥ P
    repeat j ← j-1 until A[j] ≤ P
    Swap (A[i], A[j])
until i ≥ j
Swap (A[i], A[j])  // undo last swap when i ≥ j
```

Swap (A[l], A[j])
return j

(Eg: Working)

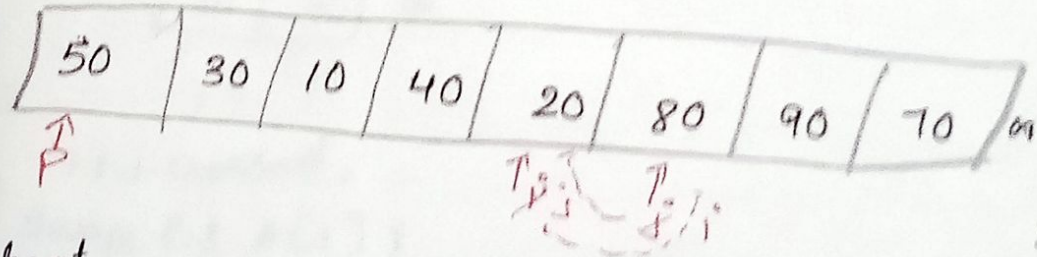


increment i until A[i] > P, decrement j until A[j] < P
if i & j have not crossed, Swap A[i] & A[j]



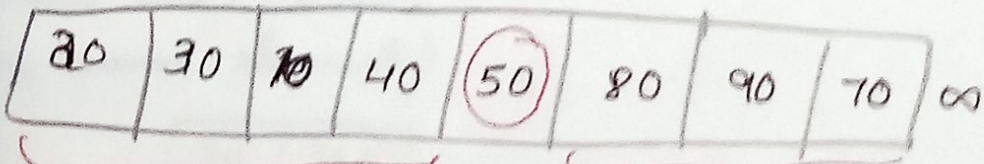
i & j not crossed, Swap A[i] & A[j]

repeat



i & j have crossed
Swap P & A[j] &
make the partition

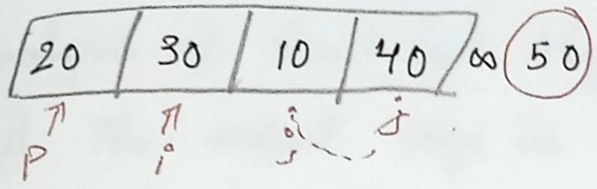
repeat



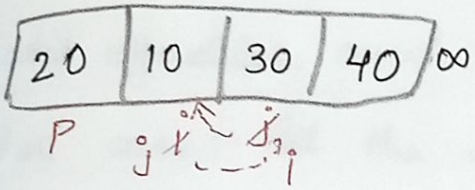
l — Left subarray
all elements less than P

r — Right subarray
all elements greater than P

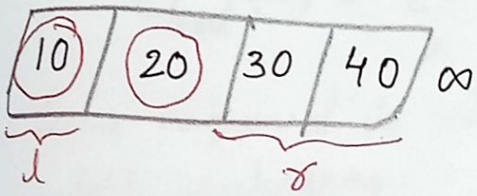
Now apply sorting recursively for left & right subarray independently, until the entire array is sorted. i.e the left & right subarray must contain only one element



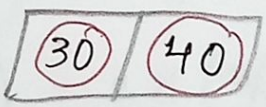
i & j not crossed,
Swap A[i] & A[j]



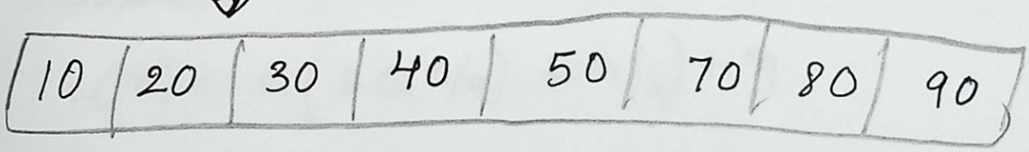
i & j crossed,
Swap P & A[j] &
make the partition



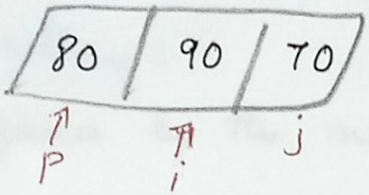
i & j crossed,
Swap P & A[j] &
make the partition
(in this case P & A[j]
are same element)



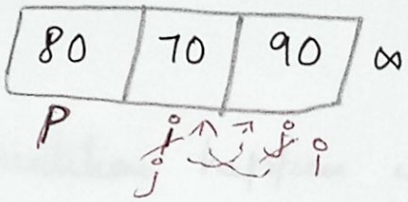
⇓ combining



The sorted array



i & j not crossed, swap A[i] & A[j]



i & j crossed, swap P & A[j]
& make the partition

Analysis of Quick Sort Algorithm:

Input: The input size is given by the number of elements n.

Basic operation: Key comparison $A[i] > P$, $A[j] < P$

Basic operation count:

- **Best case**: All the splits/partition happen in the middle of the corresponding subarrays. The recurrence relation is given by:

$$C_{best}(n) = C(n/2) + C(n/2) + n \quad \text{for } n > 1, \; C(1) = 0$$

$C(n/2)$ ↑ time required to sort left subarray

$C(n/2)$ ↑ time required to sort left subarray

n ↑ time required for partitioning the subarray.

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

- Solving this recurrence relation by substitution method

$$C(n) = 2C(n/2) + n \quad \text{---(1)}$$

$$C(n/2) = 2C(n/4) + n/2 \quad \text{---(2)}$$

$$C(n/4) = 2C(n/8) + n/4 \quad \text{---(3)}$$

- substitute

$$C(n) = 2(2C(n/4) + n/2) + n$$

$$= 4C(n/4) + 2n$$

$$= 4(2C(n/8) + n/4) + 2n$$

$= 8\,c(n/8) + 3n$

$\vdots$

$= 2^3\,c(n/2^3) + 3n$

$= 2^k\,c(n/2^k) + kn$

if $n = 2^k$

$= n.\,c(n/n) + n\log n$

$= n.\,c(1) + n\log n$

$= n.0 + n\log n$

$C(n) = n\log n$

$\therefore$ $C(n) \in O(n\log n)$, Best case (Random input)

- Worst case: all the splits will be skewed to the extreme; one of the two subarrays will be empty & the size of the other will be just 1 less than the size of the subarray being partitioned. This happens when the input array is already sorted. (0 elements to the left or right of the pivot & $(n-1)$ elements on the other side of the pivot)

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

Solve the recurrence relation by substitution method.

$C(n) = C(n-1) + n$ — (1)

$C(n-1) = C(n-2) + (n-1)$ — (2)

$C(n-2) = C(n-3) + (n-2)$ — (3)

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

$$= C(n-3) + (n-2) + (n-1) + n$$

$$\vdots$$

$$= C(n-n) + 1 + 2 + \cdots + (n-2) + (n-1) + n$$

$$= 0 + 1 + 2 + \cdots + (n-2) + (n-1) + n$$

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

$C(n) = O(n^2)$ worst case (Ascending order input / Descending order input)

Average case : split / partition can happen in any position

$$C_{avg}(n) = \begin{cases} \frac{1}{n} \sum_{s=0}^{n-1} [(n+1) + C(s) + C(n-1-s)] & \text{for } n > 1. \\ 0 & \text{for } n = 0 \text{ & } 1 \end{cases}$$

By solving the above we get $C(n) \approx 2n \ln n \approx 1.39 n \log_2 n$

$C_{avg}(n) = n \log n$ Average case (Random input)

5.3 Binary Tree Traversals :

A Binary Tree T is a tree data structure with finite set of nodes, in which each node has at most two children (0, 1, 2), referred to as the left child or left subtree T_L and right child or right subtree T_R.

- The topmost node in the tree is called the root, & the nodes with no children are called leaves.

Each node in the tree consists of a value (or key) and pointers to its children.

- Properties of Binary Trees :

Height : is defined as the length of the longest path from the root to a leaf. An empty tree has a height of -1, & a tree with only the root node has a height of 0.

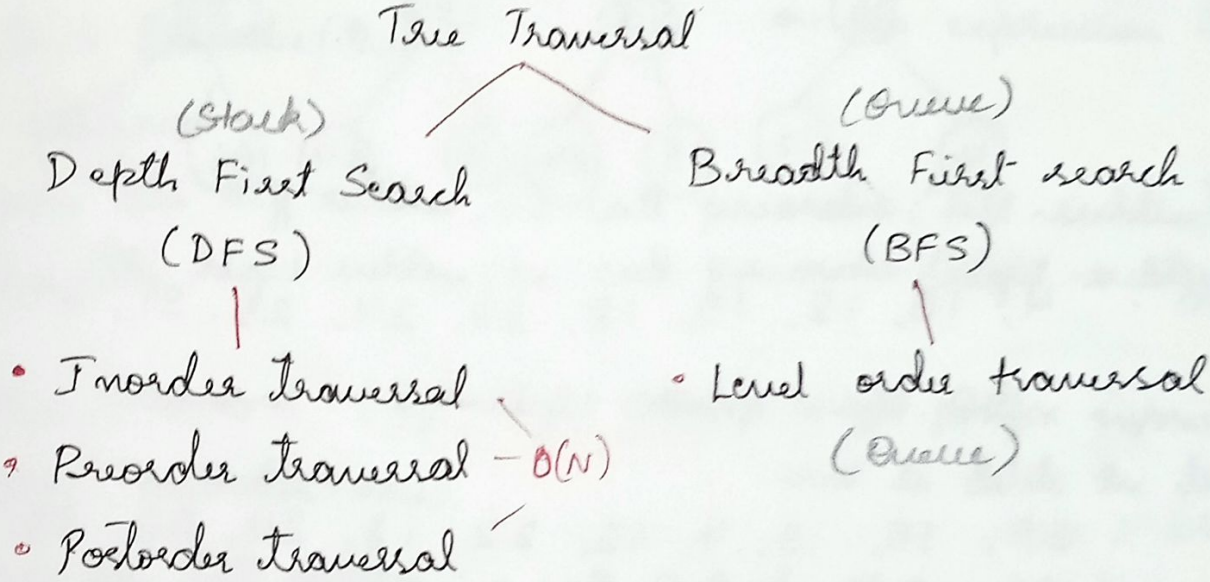
Balanced Binary Tree : A Binary tree is balanced if the height difference between the left & right subtrees of every node is at most 1.

Full Binary Tree : A Binary tree is full if every level of the tree, except possibly the last, is completely filled (0 or 2 children) & all nodes are as left as possible.

Complete Binary Tree : A Binary tree is complete if every level, except the last, is completely filled, and all nodes in the last level are as far left as possible.

- Traversal Techniques :

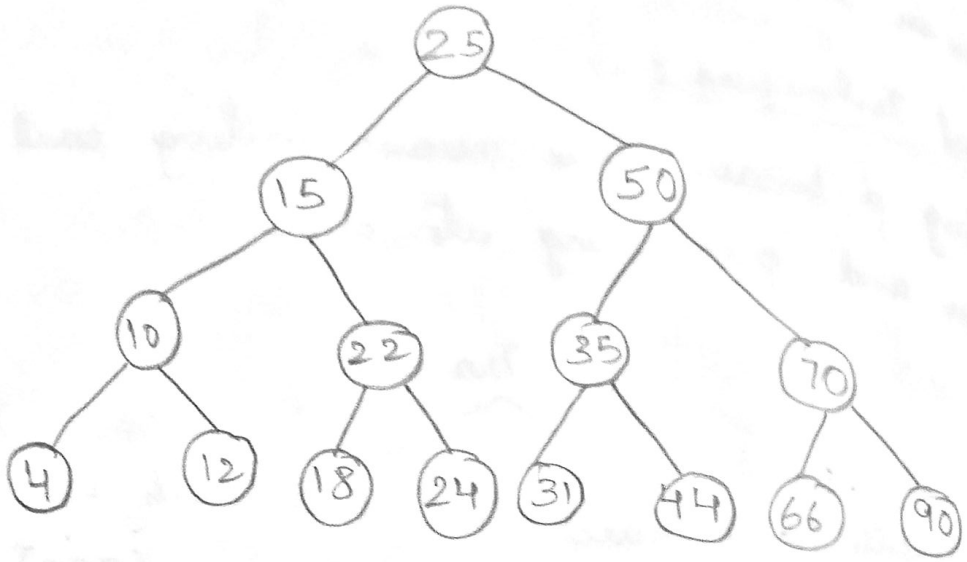
Traversing a binary tree means visiting each node in the tree and processing its data.



Breadth First Search or BFS :- A graph traversal algorithm that starts traversing the graph from the root node and explores all the neighboring nodes at the present depth (level) prior to moving on to the nodes at the next depth level.

Depth First Search or DFS :- The algorithm starts at root node and explores as far as possible along each branch before backtracking. (Backtracking to its parent node if no sibling of that node exists)

E.



Im : 4, 10, 12, 15, 18, 22, 24, 25, 31, 35, 44, 50, 66, 70, 90

Pre : 25, 15, 10, 4, 12, 22, 18, 24, 50, 35, 31, 44, 70, 66, 90

Pod : 4, 12, 10, 18, 24, 22, 15, 31, 44, 35, 66, 90, 70, 50, 25

- Inorder Traversal : (left - root - right) To get nodes in non-decreasing order.

Algorithm Inorder(tree)

- Traverse the left-subtree, i.e call Inorder(left-subtree)
- Visit the root
- Traverse the right subtree, i.e call Inorder(right subtree)

- Preorder Traversal : (root - left - right) To get prefix expressions on an expression tree

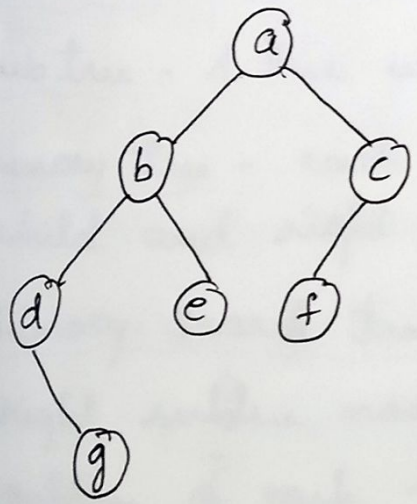
Algorithm Preorder(tree)

- visit the root
- Traverse the left subtree, i.e call preorder(left-subtree)
- Traverse the right subtree, i.e call preorder(right-subtree)

- Postorder Traversal : (left - right - root) - To get postfix expression
  - used to delete the tree

Algorithm Postorder(tree)

- Traverse the left subtree, i.e call Postorder(left-subtree)
- Traverse the right subtree, i.e call Postorder(right-subtree)
- visit the root



Inorder : d, g, b, e, a, f, c

Preorder : a, b, d, g, e, c, f

Postorder : g, d, e, b, f, c, a

Binary tree

- Tree is a hierarchical data structure which stores information. Tree contains nodes (data) and connections (edges) which should not form a cycle.
- Node - A node is a structure which may contain a value or condition, or represent a separate data structure.
- Root - The top node in a tree, the prime ancestor.
- Child - A node directly connected to another node when moving away from the root, an immediate descendant.
- Parent - an immediate ancestor.
- Leaf - A node with no children.
- Internal node - A node with at least one child.
- Edge - The connection between one node and another.
- Depth - The distance between a node and the root.
- Level - The number of edges between a node & the root + 1.
- Height - The number of edges on the longest path between a node and a descendant leaf.
- Breadth - The number of leaves.
- Subtree - A tree which is a child of a node.
- Binary tree - each node has at most two children, left child and right child
- Binary Search tree - left subtree nodes < node's root key right subtree nodes > node's root key. The left & right subtree of each node must also be a binary search tree.

Insertion, Deletion, Search

5.4 Multiplication of Large Integers and Strassen's Matrix Multiplication

1) Multiplication of Large Integers :

The multiplication of n-digit number x, by a single digit is called short multiplication.

The concept of multiplying n-digit x with another n-digit number Y is called long multiplication. Multiplying two 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 multiplications. (If one of the numbers has fewer digits than the other, we can pad the shorter number with leading zeros to equalize their lengths.)

Using divide-and-conquer technique it would be possible to design an algorithm with fewer than n^2 digit multiplications.

To demonstrate the basic idea of the algorithm, Let's take two-digit integers, say 23 and 14. These numbers can be represented as:

$$23 = 2.10^1 + 3.10^0 \quad \text{and} \quad 14 = 1.10^1 + 4.10^0$$

Lets multiply them

$$\begin{aligned} 23 \times 14 &= (2.10^1 + 3.10^0) \times (1.10^1 + 4.10^0) \\ &= (2 \times 1)10^2 + (2 \times 4 + 3 \times 1)10^1 + (3 \times 4)10^0 \\ &= 322 \end{aligned}$$

we can compute the middle term with just one digit

multiplication

$2\times4 + 3\times1 = (2+3) * (1+4) - 2\times1 - 3\times4.$

- For any pair of two digit numbers $a = a_1 a_0$ and $b = b_1 b_0$. their product C can be computed by the formula

$$C = a * b = c_2 10^2 + c_1 10^1 + c_0$$

where, $c_2 = a_1 * b_1$ is the product of their first digits,

$c_0 = a_0 * b_0$ is the product of their second digits,

$c_1 = (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0)$ is the product of the sum of the a's digits and the sum of the b's digits minus the sum of c_2 & c_0.

- Now apply this to multiplying two n-digit integers a and b, where n is a positive even number.

Lets divide both the numbers in the middle (divide-and-conquer technique).

Denote first half of the a's digits by a_1 & second half by a_0

first half of the b's digits by b_1 & second half by b_0

$a = a_1 a_0 \Rightarrow a = a_1 10^{n/2} + a_0$

$b = b_1 b_0 \Rightarrow b = b_1 10^{n/2} + b_0$

$\therefore C = a * b = (a_1 10^{n/2} + a_0) * (b_1 10^{n/2} + b_0)$

$= (a_1 * b_1) 10^n + (a_1 * b_0 + a_0 * b_1) 10^{n/2} + (a_0 * b_0)$

$$C = c_2 10^n + c_1 10^{n/2} + c_0$$

where, $C_2 = a_1 * b_1$ is the product of their first halves,

$C_0 = a_0 * b_0$ is the product of their second halves,

$C_1 = (a_1 + a_0) * (b_1 + b_0) - (C_2 + C_0)$ is the product of the sum of the a's halves and the sum of the b's halves minus the sum of C_2 and C_0.

- If $n/2$ is even, we can apply the same method for computing the products C_2, C_0 and C_1. Thus, if n is a power of 2, we have a recursive algorithm for computing the product of two n-digit integers.
- The recursion is stopped when n becomes 1, it can also be stopped when we deem n small enough to multiply the numbers of that size directly.
- Multiplication of n-digit numbers requires three multiplications of $n/2$ digit numbers, the recurrence for the number of multiplications $M(n)$ is

$$M(n) = 3M(n/2) \quad \text{for } n>1, \quad M(1) = 1$$

- Solving it by backward substitutions for $n = 2^k$ yields

$$\begin{aligned} M(2^k) &= 3M(2^{k-1}) \\ &= 3[3M(2^{k-2})] \\ &= 3^2 M(2^{k-2}) \\ &= \vdots \\ &= 3^i M(2^{k-i}) \\ &= 3^k M(2^{k-1}) \\ &= 3^k \end{aligned}$$

Since $K = \log_2 n$

$$M(n) = 3^{\log_2 n} = n^{\log_2 3} \approx n^{1.585} \qquad \because a^{\log_b c} = c^{\log_b a}$$

- Let $A(n)$ be the number of digit additions & subtractions executed by the algorithm in multiplying two n-digit decimal integers. The recurrence is

$$A(n) = 3A(n/2) + cn \quad \text{for } n > 1,\ A(1) = 1$$

$$A(n) \in O(n^{\log_2 3})$$

- Informal Algorithm:
  - Divide the large digits x & y (a & b) recursively.
  - Express large integers as polynominals.
  - Reuse and combine the terms to compute the product of large integers
  - Return the result.

2) Strassen's Matrix Multiplication :

The principal insight of the algorithm is that we can find the product C of two 2×2 matrices A and B with just 7 multiplications as opposed to the general matrix multiplication algorithm (8).
This is accomplished by using the following formulas:

$$\begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} * \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}$$

$$= \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}$$

where

$$m_1 = (a_{00} + a_{11}) * (b_{00} + b_{11})$$

$$m_2 = (a_{10} + a_{11}) * (b_{00})$$

$$m_3 = a_{00} * (b_{01} - b_{11})$$

$$m_4 = a_{11} * (b_{10} - b_{00})$$

$$m_5 = (a_{00} + a_{01}) * b_{11}$$

$$m_6 = (a_{10} - a_{00}) * (b_{00} + b_{01})$$

$$m_7 = (a_{01} - a_{11}) * (b_{10} + b_{11})$$

Thus, to multiply two 2×2 matrices, Strassen's algorithm makes 7 multiplications & 18 additions/subtractions.

- Let A & B be two $n \times n$ matrices where n is a power of 2. (If n is not a power of 2, matrices can be padded with rows & columns of zeros). We can divide A, B and their product C into 4 $n/2 \times n/2$ submatrices

$$\begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} * \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix}$$

If the 7 products of $n/2 \times n/2$ matrices are computed recursively by the same method, we have Strassen's algorithm for matrix multiplication.

- Let us evaluate the asymptotic efficiency of the algorithm: If $M(n)$ is the number of multiplications made by Strassen's algorithm in multiplying two $n \times n$ matrices (where n is a power of 2), we get the recurrence relation

$$M(n) = 7M(n/2) \quad \text{for } n > 1, \quad M(1) = 1$$

Since $n = 2^k$

$$\begin{aligned} M(2^k) &= 7M(2^{k-1}) \\ &= 7[7M(2^{k-2})] \\ &= 7^2 M(2^{k-2}) \\ &\vdots \\ &= 7^i M(2^{k-i}) \end{aligned}$$

$= 7^k M(2^{k-k})$

$= 7^k M(2^0) = 7^k M(1)$

$= 7^k$

Since $k = \log_2 n$

$$M(n) = 7^{\log_2 n} = n^{\log_2 7} \approx n^{2.807}$$

which is smaller than n^3 required by normal matrix multiplication.

Since this savings in the number of multiplications was achieved at the expense of making extra additions, Let $A(n)$ be the number of additions made, the recurrence relation is

$$A(n) = 7A(n/2) + 18(n/2)^2 \quad \text{for } n > 1, \; A(1) = 0$$

$$A(n) \in O(n^{\log_2 7})$$

the number of additions has the same order of growth as the number of multiplications. i.e Strassen's algorithm is in $O(n^{\log_2 7})$, which is better efficiency class than $O(n^3)$ of normal matrix multiplication (brute force method).