

DECREASE-AND-CONQUER

As the name suggests, the decrease-and-conquer methodology suggests to decrease the size of the given problem. Here we try to find a relationship between the solution of original problem and that of smaller instance of the same problem. After finding this relationship, we will solve it either by top-down (recursive) approach or by bottom-up (iterative) approach.

For this technique, decreasing the size of the problem may be any one of these types -

- * decrease by a constant
- * decrease by a constant fraction
- * Variable size decrease.

1. Decrease by a constant :- for this technique, the size of an instance is reduced by the same constant on each iteration. Usually, this size will be 1. Consider an example of finding the value of a^n . The relationship between a solution to a problem of size n and of size $(n-1)$ is given by -
$$a^n = a^{n-1} \cdot a$$

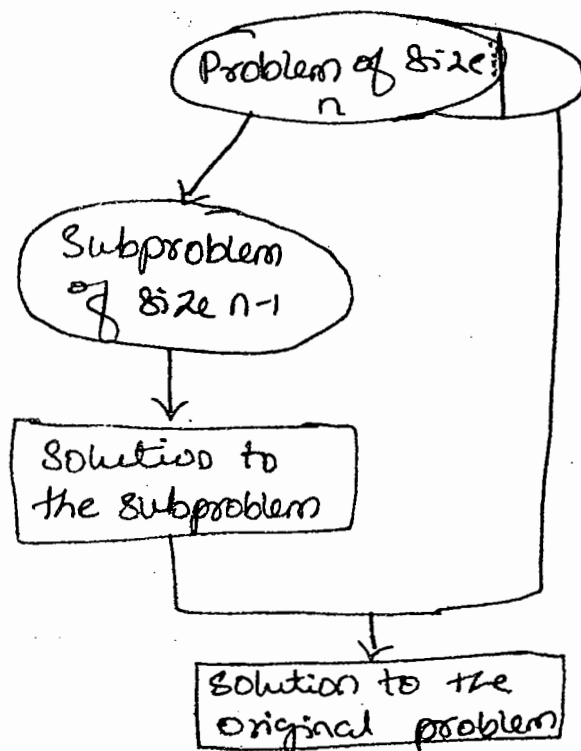
Thus, the function $t(n) = a^n$ can be solved by either top-down approach using

the relation -

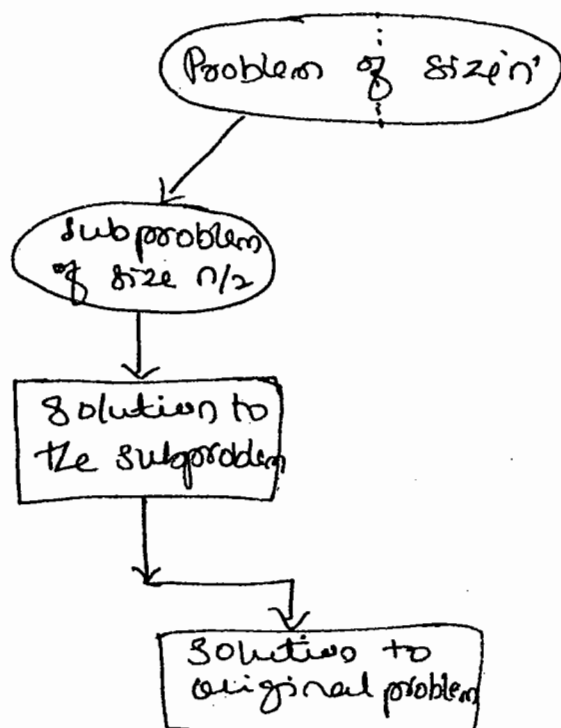
$$t(n) = \begin{cases} t(n-1) + a & , \text{ if } n > 1 \\ a & , \text{ if } n = 1 \end{cases}$$

Or by ~~adding~~ multiplying a by itself $(n-1)$ times, which is bottom-up approach.

The pictorial representation of decrease-and-conquer by one is as below -



2. Decrease by a constant factor :- Here, the size of a problem is reduced by same constant factor on each iteration. Usually, this constant factor will be 2. This can be illustrated as -



For example, if we consider the same problem of finding a^n , it is given by -

$$a^n = \begin{cases} (a^{n/2})^2 & \text{if } n \text{ is even \& tve} \\ [a^{(n-1)/2}]^2 \cdot a & \text{if } n \text{ is odd \& } \geq 1. \\ a & \text{if } n = 1. \end{cases}$$

3. Decrease by a variable size :: Here, on each iteration, the size of original problem is reduced by different values.

For example, if we consider a problem of finding gcd of two numbers -

$$\text{gcd}(m, n) = \text{gcd}(n, m \bmod n)$$

Here, each time n is replaced by $(m \bmod n)$ and obviously it is a ~~variable~~ varying pattern on each step.

Insertion Sort :-

for this sorting technique, we will compare n^{th} element with all other previous elements and if that element finds its proper position, it is placed there, and rest all the elements which are greater, ~~than~~ are pushed one position ahead. ~~In~~ this technique, we follow the idea that, at each step, the elements before n^{th} element are sorted. So, each time, the array size is reduced by one.

The iterative algorithm is given as-

ALGORITHM InsertSort($A[0..n-1]$)

// Sorts given array by insertion sort

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

// Output: A sorted array $A[0..n-1]$.

for $i \leftarrow 1$ to $n-1$ do

$v \leftarrow A[i]$

$j \leftarrow i-1$

 while $j \geq 0$ and $A[j] > v$ do

$A[j+1] \leftarrow A[j]$

$j \leftarrow j-1$

$A[j+1] \leftarrow v$

Analysis:

1. The parameter n is input size n .
2. The basic operation is comparison.
3. The time complexity not only depends on the input size, but also on the fact that how the elements are arranged. So, let us discuss various complexities.

Worst case:

If the comparison $(A[j] > v)$ executes for each value of i and j , then there will be maximum number of comparisons. This will happen if the given array is strictly decreasing. Thus, the worst case happens for an array sorted in decreasing order.

Thus,

$$\begin{aligned}
 C_{\text{worst}}(n) &= \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 \\
 &= \sum_{i=1}^{n-1} (i-1-0+1) = \sum_{i=1}^{n-1} i \\
 &= 1+2+\dots+(n-1) \\
 &= \frac{(n-1)(n)}{2} \approx n^2
 \end{aligned}$$

$$\therefore C_{\text{worst}}(n) \in \Theta(n^2)$$

Best Case:

Best case occurs if the given array is already sorted. In such case, for each value of i , there will be only one comparison. So,

$$\begin{aligned}C_{\text{Best}}^{(n)} &= \sum_{i=1}^{n-1} 1 \\&= (n-1-1+1) \\&= n-1\end{aligned}$$

$$\therefore C_{\text{Best}}^{(n)} \in \Theta(n).$$

Average Case:

Suppose that, we are considering i^{th} element of the array for its checking its proper position. Then j will be at $(i-1)^{\text{th}}$ position. Now, there are several possibilities about the position of i^{th} element—

~~If j^{th} element—~~

If $A[j] < A[i]$, then the position of $A[i]$ is fixed, and only one comparison is made.

If $A[j] > A[i]$, but $A[j-1] < A[i]$, then only two comparisons happen.

Continuing in this way, the possible number of comparisons for the i^{th} element are

1, 2, 3, ..., p . Thus, the average number of comparisons for each element is given by,

$$C_{avg}(n) = \sum_{i=1}^{n-1} \frac{1}{p} \cdot \sum_{j=1}^p j$$
$$= \sum_{i=1}^{n-1} \left\{ \frac{1}{p} \cdot (1+2+\dots+p) \right\}$$

$$= \sum_{i=1}^{n-1} \left\{ \frac{1}{p} \cdot \frac{p(p+1)}{2} \right\}$$

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

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

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

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

$$= \frac{(n-1)(p+2)}{4}$$

$$\approx \frac{n^2}{4}$$

$$\therefore C_{avg}(n) \in \Theta(n^2)$$

Depth First Search (DFS) :-

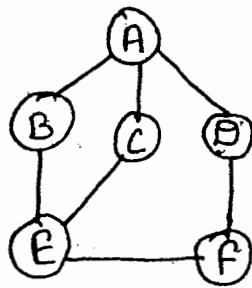
DFS is one of the techniques to process the vertices or edges of a graph in a systematic order. The procedure is as below -

1. Visit a vertex of a graph arbitrarily & mark it as being visited.
2. Visit ~~the~~ any ^{unvisited} vertex which is adjacent to the previous one. ~~to~~
3. Repeat the step 2 until no adjacent vertex ^{is} left ~~with~~ ^{out} for a particular vertex.
4. Go back to one vertex before the dead-end vertex and try to find any unvisited vertex and repeat the process if so.
5. The algorithm halts when the starting vertex becomes the dead-end and by this time, all the ~~same~~ vertices in that connected component of a graph are visited.
6. If the graph contains still unvisited vertices, reapply the above procedure by choosing any arbitrary vertex.

To implement the operations of DFS, we will use stack. When a vertex is reached for the first time, it is pushed into the stack. When it becomes the dead-end, we pop-up it.

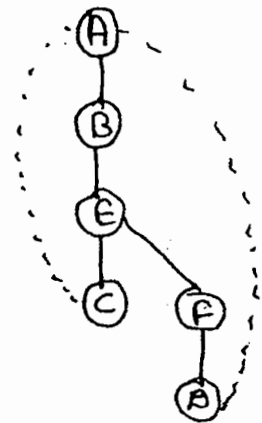
Sometimes, the dfs forest is used for illustrating DFS traversal. In DFS forest, the starting vertex of DFS traversal acts as a root of first tree. Whenever a new unvisited node is encountered, it is put as a child to the vertex from which it is being searched. The edge connecting two such nodes is called as tree edge. In this process, we may get an edge from a particular node to a previously visited node, which is not its parent. Such an edge is called as back-edge.

Consider the following graph.



The stack representation of above graph is given below. Here, the first subscript indicates the order in which the node is visited and the second subscript indicates the order in which the node is deleted from the stack, i.e. when it becomes dead-end.

A _{1, 6}	D _{5, 3}	F _{5, 3}
B _{2, 5}	F _{6, 2}	D _{6, 2}
E _{3, 4}		
C _{4, 1}		



The DFS forest is also given here. Dotted lines show back-edges.

ALGORITHM DFS(G)

// Implementation of DFS traversal of graph G .
// Input: Graph $G = (V, E)$
// Output: Graph G with its vertices marked
// with consecutive integers in the
// order they visited.

mark each vertex in V with 0 as of
being unvisited.

count $\leftarrow 0$

for each vertex v in V do

if v is marked with 0

dfs(v)

dfs(v)

// visits all unvisited vertices connected to v
// and assigns them the ~~ord~~ numbers in the
// order they visited using 'count'.

count $\leftarrow$ count + 1

mark v with count

for each vertex w in V adjacent to v do

if w is marked with 0

dfs(w)

We can create adjacency linked list and adjacency matrix for the given graph based on the DFS algorithm. The efficiency of DFS depends on the fact that which data structure (i.e.,

adjacency matrix or adjacency linked list) we will use for its implementation. If we use adjacency matrix, then the time complexity of DFS is $\Theta(|V|^2)$ and if we use adjacency linked list, the complexity will be $\Theta(|V| + |E|)$. Here, $|V|$ and $|E|$ denotes number of vertices & edges of the graph, respectively.

Applications of DFS are as below-

- * To check the connectivity of a graph: When DFS algorithm halts, if all the vertices have been visited, then the graph is connected, otherwise not.
- * To check acyclicity of a graph: In a DFS forest, if any ~~edge~~^{tree} contains a back-edge, then the graph is cyclic, otherwise, acyclic.

(NOTE: A vertex of a connected graph is said to be articulation point of the graph if its removal with all edges incident to it breaks the graph into disjoint pieces.)

- * To find articulation point of a graph.

Breadth-First Search (BFS): -

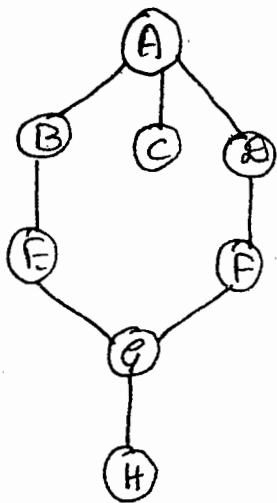
BFS is also a method for processing the vertices or edges of a graph in a systematic order. The procedure is as below -

1. Visit a vertex of a graph arbitrarily & mark it as visited.
2. All the vertices which are adjacent to previously visited vertex are visited sequentially.
3. Repeat the above step for next level and continue till all the vertices are visited in that connected component.
4. If some vertices are still remaining repeat the steps from step 1.

We will use the Queue data structure to trace the operations of BFS. The first vertex of the traversal becomes the initial value of queue, i.e. the element at the front. On each iteration, the algorithm finds the vertices which are adjacent to the front vertex. These vertices are inserted into the queue and the front vertex is deleted from the queue.

As in DFS, for BFS also we will construct a BFS forest. The starting vertex of the traversal will be the root of one tree in BFS forest. Whenever a new node is visited, it must be attached to a previous node as a child. Such an edge is known as 'tree-edge'. Whenever a node finds an edge to already visited vertex other than its parent, then it is treated as 'cross-edge'.

Consider a graph-

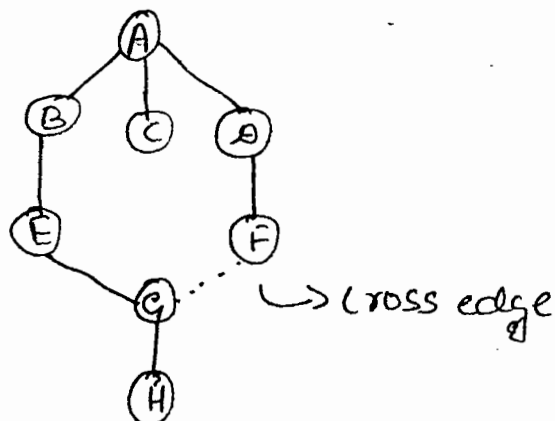


The queue sequence is given by-

$A_1, B_2, C_3, D_4, E_5, F_6, G_7, H_8$

The subscripts denote the order of their insertion into queue.

The BFS forest for this graph will be-



ALGORITHM BFS(G)

// Implementing BFS traversal for a graph G .

// Input: $G = (V, E)$.

// Output: Graph G with its vertices marked

// with consecutive integers in the
// order they visited.

mark each vertex in V with 0 as unvisited.

count $\leftarrow 0$

for each vertex v in V do

if v is marked with 0

bfs(v)

bfs(v)

// visits all unvisited vertices connected to v .

// and assigns them the numbers in the order

// of their visit.

count $\leftarrow$ count + 1

mark v with count and initialize queue
with v

while the queue is not empty do

for each vertex w in V adjacent to
front vertex v do

if w is marked with 0

count $\leftarrow$ count + 1

~~add w to the~~

mark w with count

add w to the queue

remove v from queue.

The analysis of BFS traversal depends on the data structure used for implementation. If we use adjacency matrix, then the time complexity will be $\Theta(V^2)$ and if we use adjacency linked list, the time complexity will be $\Theta(V + E)$.

Applications:

- * Finding connectivity of a graph.
- * Finding acyclicity of a graph.
- * To find the shortest path between any two given vertices.

Comparison of DFS & BFS

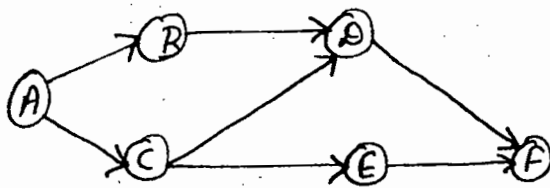
	<u>DFS</u>	<u>BFS</u>
Data structure	Stack	Queue
No. of vertex orderings	two	one
Edge types	tree edge back edge	tree edge cross edge
Applications	Connectivity Acyclicity Articulation point	Connectivity Acyclicity Shortest path
Efficiency for Adj. Matrix	$\Theta(V^2)$	$\Theta(V^2)$
Efficiency for Adj. Linked List	$\Theta(V + E)$	$\Theta(V + E)$

Topological Sorting

Consider a problem of completing a project which can be divided into several parts. Usually, we have to estimate the time required for completing it. For doing that, obviously, we estimate the time required for each of the subparts keeping in mind that some of the subparts can be carried out parallelly, some of them cannot be started until some others are completed etc. That means, there will be some pre conditions and constraints for carrying out these subparts. If we consider each subtask as a vertex, we can draw a directed graph where the direction of each edge indicates the order in which they must be completed. One thing we can note down here is, if such a digraph contains cycles, then such a problem cannot be solved as it will be a deadlock. Thus, if the problem possesses a directed acyclic graph (dag), then we can solve it.

After creating a dag, our intension is to find a sequence of subtasks which says the order of their completion by satisfying all the constraints & prerequisites. The problem of finding such a sequence is called as topological sorting.

For illustration, consider a project which is divided into 6 subtasks viz. A, B, C, D, E & F. The constraints are as below- To start B and C, A must be completed. D will be carried out only after B and C. E can be carried out after the completion of C. F can be done after D and E. The digraph can be given for this problem as-



It can be observed that the graph is acyclic, i.e. it is a dag. So, we can solve it using topological sorting technique.

There are two methods for solving topological problem.

One method is using DFS traversal. Here, we will perform a DFS traversal for the dag and

noted down the order ~~of~~ in which the elements becomes dead-ends. i.e. the order of popping up the vertices from traversal stack. The reverse of this order will give required solution. Thus, for the above problem, DFS traversal would be-

$A_{1,6}$ $B_{2,3}$ $D_{3,2}$ $F_{4,1}$ $C_{5,5}$ $E_{6,4}$

$\therefore$ The order of popping up from stack is.

F, D, B, E, C, A.

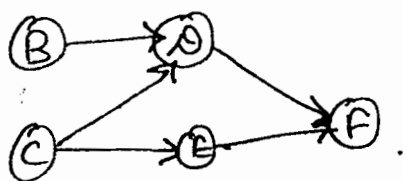
The required order will be-

A, C, E, B, D, F.

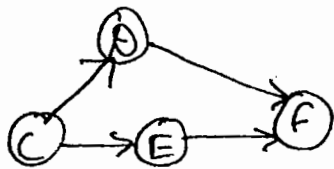
The second method is based on implementation of decrease-by-one-and-conquer strategy. Here, in each step, identify a vertex with no incoming edges. Then delete it with all outgoing edges from it.

The order in which the vertices are deleted will be the required solution for a problem.

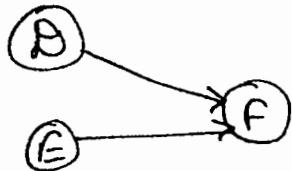
For the above example, first delete A. So,



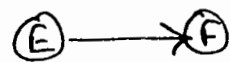
Then delete either B or C, arbitrarily.



Then, deleting C -



Now, deleting D -



Finally, deleting E -



So, required order is -

A B C D E F.

NOTE: The DFS method and BAK method may yield different solutions. Because topological sorting problem will be having several solutions.

Following is the algorithm for topological sort using DFS technique -

ALGORITHM DFT ($A[0..n-1][0..n-1]$, n)

// To obtain the sequence of jobs to be executed

// Input: Adjacency matrix A of the graph

// The number of vertices n .

// Output: Sequence in the reverse order that are to be executed.

```

for i ← 0 to n-1 do
    S[i] ← 0 // no vertex has been visited.
j ← 0 // vertexindex which is dead-end.

```

```

for v ← 0 to n-1 do
    if (S[v] = 0)
        DFS(a, n, v)

```

ALGORITHM DFS(a[0..n-1][0..n-1], n, v)

// Input: The vertex v from which traversal starts

S[v] ← 1

```

for u ← 0 to n-1 do
    if (cost[u][u] = 1 and S[u] = 0) then
        DFS(a, n, u)

```

j ← j + 1

res[j] ← v

Analysis:

In two functions, we have two loops. The time complexity depends on these. As there is one comparison for each loop, we have

$$\begin{aligned}
 \sum_{v=0}^{n-1} \sum_{u=0}^{n-1} 1 &= \sum_{v=0}^{n-1} (n-1-0+1) \\
 &= n \sum_{v=0}^{n-1} 1 = n^2
 \end{aligned}$$

∴ Time complexity is $\Theta(n^2)$, where $n=|V|$, i.e. the number of vertices in a graph.

Following is the algorithm for topological sort by source removal method.

ALGORITHM TS($A[0..n-1][0..n-1], n$)

// To get a sequence of vertices to be executed.

// Input: Adjacency matrix A of the graph.

// Number of vertices, n .

// Output: The sequence of vertices, S .

for $j \leftarrow 0$ to $n-1$ do

$sum \leftarrow 0$

 for $i \leftarrow 0$ to $n-1$ do

$sum \leftarrow sum + A[i][j]$

$indegree[j] \leftarrow sum$

for $i \leftarrow 0$ to $n-1$ do

 if ($indegree[i] = 0$)

$top \leftarrow top + 1$

$S[top] \leftarrow i$

while ($top \neq 1$)

$u \leftarrow S[top]$

$top \leftarrow top - 1$

 Add u to solution vector T

 for each vertex v adjacent to u

 decrement $indegree[v]$ by one

 if ($indegree[v] = 0$)

$top \leftarrow top + 1$

$S[top] \leftarrow v$.

Generating Combinatorial Objects

The combinatorial objects like permutations, subset of a set etc. plays a vital role in the field of computer science. Here, we will study few of the algorithms for their generation.

Generating Permutation:-

As we know, if there are n different items, we can arrange them in $n!$ several ways. That is, we can find $n!$ permutations for n elements.

Now, our problem is to generate these $n!$ permutations. For the sake of simplicity, let us consider the numbers $1, 2, \dots, n$. We have to find all $n!$ permutations of this set. The decrease-by-one technique says ~~that~~ to generate $(n-1)!$ permutations first, and then to place n^{th} number in n different positions of all of these permutations. ~~The process~~ Now we get $n \times (n-1)! = n!$ permutations.

The process can be applied recursively to get all permutations.

For illustrations, consider the generation of permutations of $\{1, 2, 3, 4\}$.

Start: 1,

Insert 2: 12 21

Insert 3: $\underbrace{123 \ 132 \ 312}_{\text{right to left}} \quad \underbrace{321 \ 231 \ 213}_{\text{left to right}}$

Insert 4: $\underbrace{1234 \ 1243 \ 1423 \ 4123}_{\text{right to left}} \quad \left. \begin{array}{l} 1324 \ 1342 \ 1432 \ 4132 \\ 3124 \ 3142 \ 3412 \ 41312 \end{array} \right\} \text{right to left}$

$\left. \begin{array}{l} 4321 \ 3421 \ 3241 \ 3214 \\ 41231 \ 2431 \ 2341 \ 2314 \\ 4213 \ 2413 \ 2143 \ 2134 \end{array} \right\} \text{left to right}$

The above method is known as minimal-change technique. Because, each number can be generated by exchanging only two ^{digits} of its previous number.

There is one more method called Johnson-Trotter algorithm. In this method, we first take a permutation and associate an arrow mark for each digit 'k' of the permutation, to indicate direction. The digit

or component k is said to be 'mobile' in arrow-marked permutation if its arrow points to a smaller number adjacent to it. For example—

$\vec{3} \leftarrow 2 \vec{4} \leftarrow 1$.

Here 3 and 4 are mobiles.

Using this notation, the Johnson-Trotter algorithm is implemented as below—

ALGORITHM JT(n)

// Johnson-Trotter algorithm for generating
// permutations.

// Input: A positive integer n .

// Output: A list of all permutations of $\{1, \dots, n\}$

Initialize the first permutation with
 $\leftarrow 2 \dots n$

While there exists a mobile integer k do
find the largest mobile integer k .

Swap k and the adjacent digit ^{towards} which
 k is having arrow. ~~towards it~~.

reverse the direction of all integers
that are larger than k .

Consider an example of generating permutations of $\{1, 2, 3\}$.

Initially, we write $\uparrow \underline{3} \underline{2}$.

Here, $\underline{3}$ is largest mobile integer. So, next term will be $\uparrow \underline{3} \underline{2}$. Continuously applying the J-T algorithm, we get

$\uparrow \underline{2} \underline{3}$ $\uparrow \underline{3} \underline{2}$ $\underline{3} \uparrow \underline{2}$
 $\underline{3} \underline{2} \uparrow$ $\underline{2} \underline{3} \uparrow$ $\underline{2} \uparrow \underline{3}$

Here, one thing we have to note that, the order of the numbers generated by J-T algorithm is quite unobvious. Because, usually, permutations are written by increasing order i.e. for example-

123, 132, 213, 231, 312, 321

This method is known as lexicographic Order.

The method is as explained below- Suppose, if we want to generate permutations of $a_1 a_2 \dots a_{n-1} a_n$. Then, if $a_{n-1} < a_n$, the next number is obtained by just interchanging a_{n-1} and a_n . If $a_{n-1} > a_n$, we will consider a_{n-2} . If $a_{n-2} < a_{n-1}$ the last three elements i.e. a_{n-2} , a_{n-1} and a_n are rearranged in lexicographic order only & the process is continued.

Generating Subsets :-

for many problems, we need to generate all the subsets of a given set for finding the solution to the problem. Knapsack is one such example.

The decrease-by-one-and-conquer method for finding a power set of a set $\{a_1, \dots, a_n\}$ suggests to group the sets of power set into two parts viz. those containing a_n and those not containing a_n . Then, the second group will obviously be the subsets of $\{a_1, \dots, a_{n-1}\}$ and the first group can be obtained by placing a_n in all the subsets of $\{a_1, \dots, a_{n-1}\}$. Now, the procedure is reapplied on $\{a_1, \dots, a_{n-1}\}$.

The following example illustrates this method.

n	Subsets
0	ϕ
1	ϕ $\{a_1\}$
2	ϕ $\{a_1\}$ $\{a_2\}$ $\{a_1, a_2\}$
3	ϕ $\{a_1\}$ $\{a_2\}$ $\{a_1, a_2\}$ $\{a_3\}$ $\{a_1, a_3\}$ $\{a_2, a_3\}$ $\{a_1, a_2, a_3\}$

One more method for generating the power set of a set is by establishing a one-to-one relationship between the set $\{a_1, \dots, a_n\}$ and all 2^n bit strings of length n . For example, if we take $n=3$ then.

bit strings: 000 001 010 100 101 110 111

Subsets : $\emptyset$ $\{a_3\}$ $\{a_2\}$ $\{a_1\}$ $\{a_1, a_3\}$ $\{a_1, a_2\}$ $\{a_1, a_2, a_3\}$