# Decrease - And - Conquer

The approach decrease and conquer includes the following steps:

a) Decrease: Reduce problem instance to smaller instance of the same problem and extend solution.

b) Conquer the problem by solving a smaller instance of the problem.

c) Extend solution of smaller instance to obtain solution to original problem.

Basic idea of 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. This approach is also known as incremental or inductive approach.

This technique is used when its easier to solve a smaller version of the problem, and the solution to the smaller problem can be solved used to find the solution to the original problem.

This approach can be either implemented as top-down or bottom-up.

Top-down approach: It always leads to the recursive implementation of the problem.

**Bottom-up approach :-** It is usually implemented in iterative way. Starting with a solution to the smallest instance of the problem.

**Control abstraction :-**

```
Algorithm DecreaseAndConquer(A[0...n-1])
// A is the given problem of size n
{
  if (small()) return G(); // Return the
                           // solution of
                           // original problem
  n = n-1;  // decrease by one
  Return DecreaseAndConquer(A);
}
```

Examples which uses decrease-and-conquer technique :

a) Insertion sort
b) Depth First and Breadth First Search
c) Topological Sorting
d) Generating Permutations
e) Generating Subsets

There are three major variations of decrease & conquer :

1. Decrease by a constant
2. Decrease by a constant factor
3. Variable size decreases

## Insertion Sort :-

It is a simple sorting algorithm that works similar to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at 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.

## working :-

Consider an example :

arr[] = {10, 12, 30, 10, 8, 15, 20, 11, 5, 2}

P.T.O

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| [70] | 12 | 30 | 10 | 8 | 15 | 20 | 11 | 5 | 2 |
| [12 | 70] | 30 | 10 | 8 | 15 | 20 | 11 | 5 | 2 |
| [12 | 30 | 70] | 10 | 8 | 15 | 20 | 11 | 5 | 2 |
| [12 | 30 | 70] | 8 | 15 | 20 | 11 | 5 | 2 | |
| [8 | 10 | 12 | 30 | 70] | 15 | 20 | 11 | 5 | 2 |
| [8 | 10 | 12 | 15 | 30 | 70] | 20 | 11 | 5 | 2 |
| [8 | 10 | 12 | 15 | 20 | 30 | 70] | 11 | 5 | 2 |
| [8 | 10 | 11 | 12 | 15 | 20 | 30 | 70] | 5 | 2 |
| [5 | 8 | 10 | 11 | 12 | 15 | 20 | 30 | 70] | 2 |
| [2 | 5 | 8 | 10 | 11 | 12 | 15 | 20 | 30 | 70] |

Design an algorithm for insertion sort and analyse its time complexity.

```
Algorithm InsertionSort (A[0...n-1], n)
// A[0...n-1] is the input array to be sorted
// K is the key element to be inserted
{
  for j = 1 to n-1 do
  {
      K = A[j];
      P = j-1;
      while P ≥ 0 and K < A[P] do
      {
          A[P+1] = A[P];
          P = P-1;
      }
      A[P+1] = K;
  }
}
```

Analysis :-

Worst Case :-

The basic operation of insertion sort is comparison. There are two important points to note in the worst case :

a) The outer loop executes for $j = 1$ to $n-1$

b) The inner loop executes for $P = j-1, j-2, \ldots 0$

$$C_{wc}(n) = \sum_{j=1}^{n-1} \sum_{P=0}^{j-1} 1 = \sum_{j=1}^{n-1} j = \frac{n(n-1)}{2} = O(n^2)$$

**Best Case :** The best case occurs when the array is already sorted and therefore the statement $K < A[P]$ will be executed only once. However, the outer loop must be executed from 1 to $n-1$. Hence,

$$C_{BC}(n) = \sum_{j=1}^{n-1} 1 = (n-1) = \underline{\Omega(n)}$$

**Average case :-** Avg. Case complexity is same as worst-case complexity.

$$C_{AC}(n) = \underline{\theta(n^2)}$$

## Topological Sorting :-

What is topological sorting? What are the various methods using which topological sequence can be obtained?

The 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 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 depends on B and B depends on A, then it is cyclic. A graph which is cyclic does not have topological sequence.

The topological sorting can be done using following two methods:

a. DFS method

b. Source removal method

# Topological sort using DFS Method

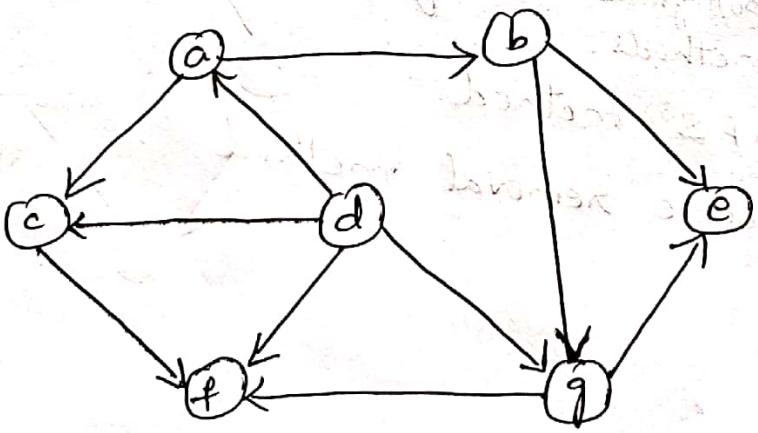
How to get the topological order using DFS method?

The topological order using DFS method can be obtained as shown below:

1. Select any arbitrary vertex
2. When a vertex is visited for the first time, it is pushed on to the stack.
3. When a vertex becomes a dead end, it is removed from the stack.
4. Repeat step 2 to 3 for all the vertices in the graph.
5. Reverse the order of deleted items to get the topological sequence.

Example:-

1. Apply the DFS based algorithm to solve the topological sorting problem for the following graph:



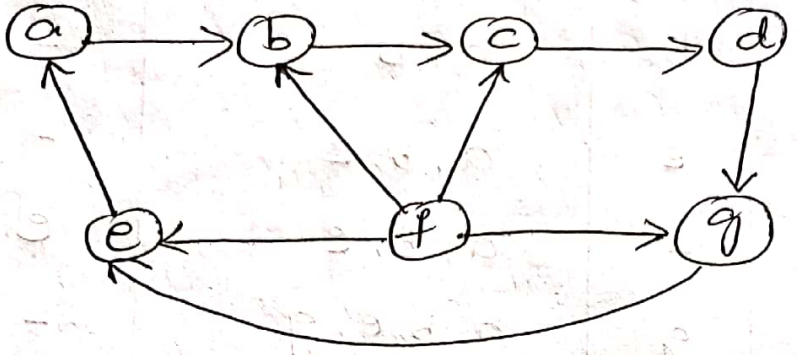
Sol:-

| Stack | adj(s.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 |
| Stack is empty. So, take the next vertex in sequence which is not visited and push it onto stack and add to S. | | | |
| d | - | a, b, e, g, f, c, d | d |

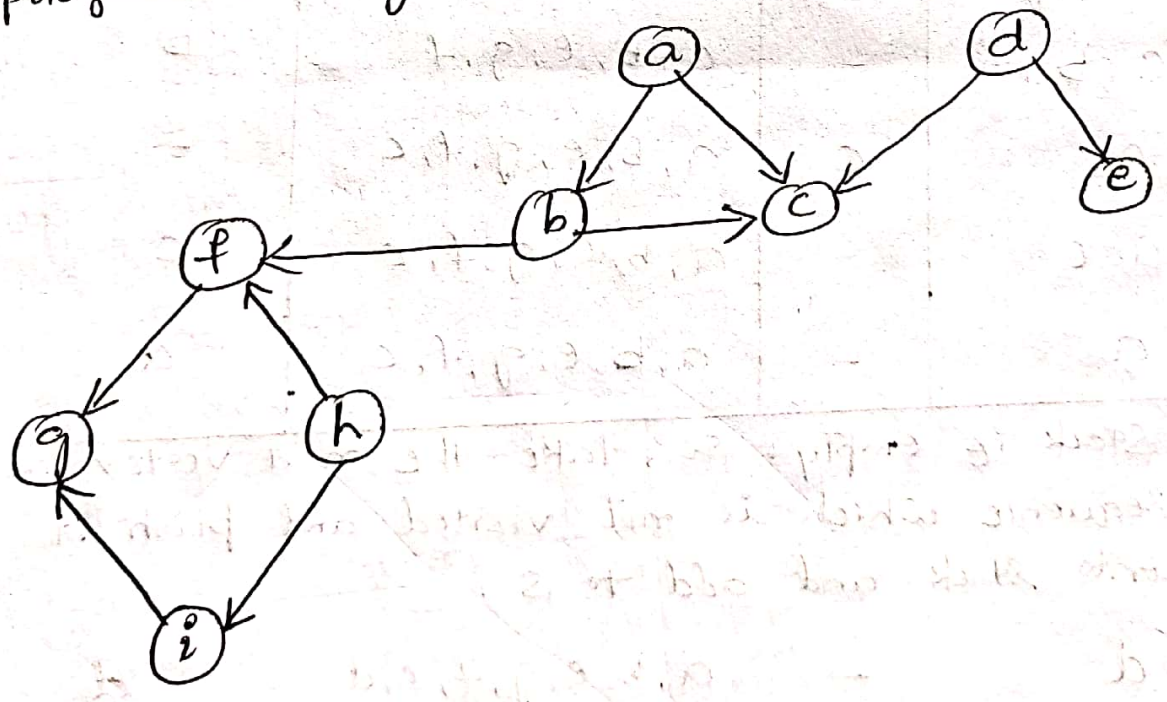
The order in which the vertices are removed from the stack is obtained from the last column. The popped order is : e, f, g, b, c, a, d.

The topological order is obtained by reversing the above popped order : d → a → c → b → g → f → e

2. Apply the DFS based algorithm to solve the topological sorting problem for the following graph :



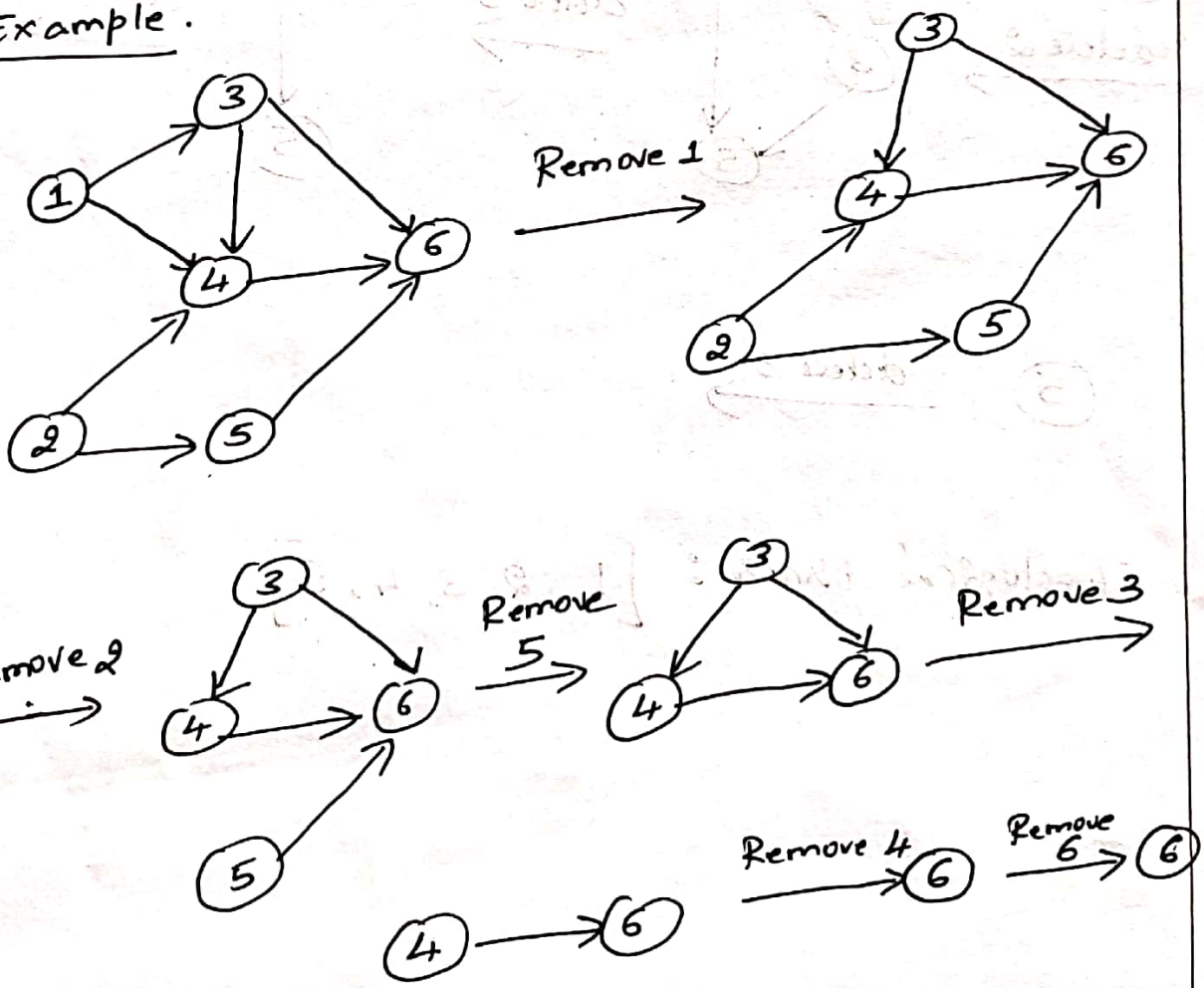

3. Apply the DFS based algorithm to solve the topological sorting problem for the following graph.



## 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. Once we remove this vertex then all its outgoing edges must also be removed. During every stage (iteration), identify such a source vertex and iterate until all vertices are covered. The sequence generated by this removal process will form the topological ordering.

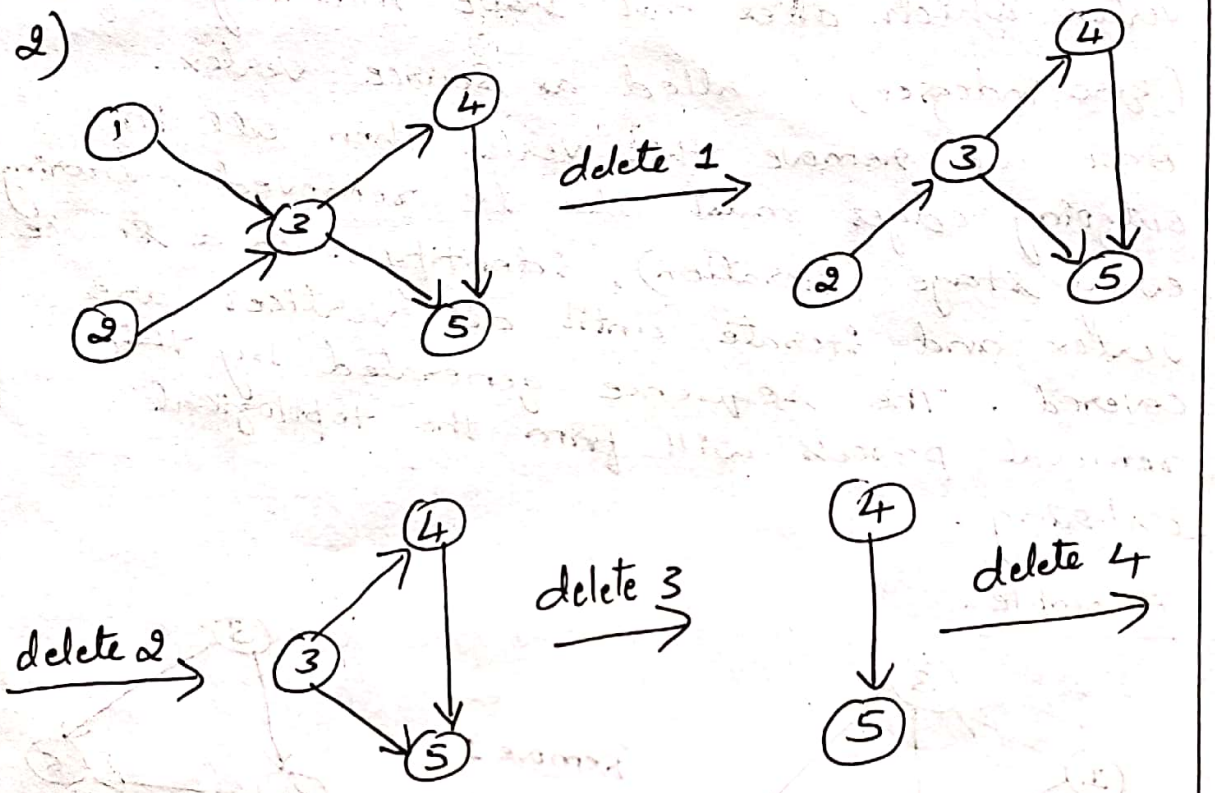
Example.



Sequence of source vertex removal method is :

[1, 2, 5, 3, 4, 6]

2)



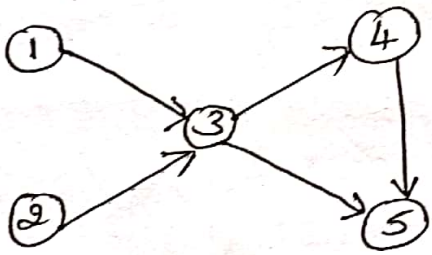

Topological order : [1, 2, 3, 4, 5]

## Topological Sort using Source Removal Method :-

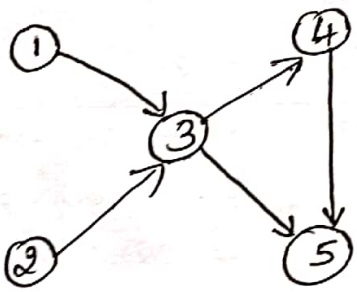
This method is based on decrease-and-conquer technique. In this method, a vertex with no incoming edges is selected and deleted along with the outgoing edges. If there are several vertices with no incoming edges, arbitrarily a vertex is selected. The order in which the vertices are visited and deleted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

Example :-

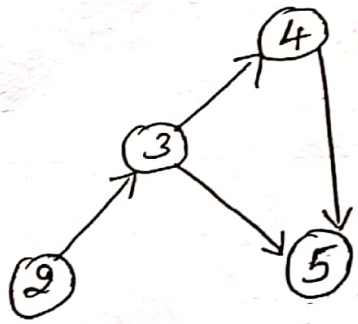
Apply the source removal method to solve the topological sorting problem for the following graph:



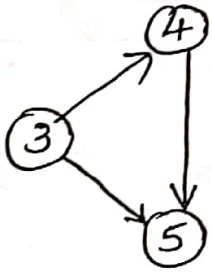

Sol :-



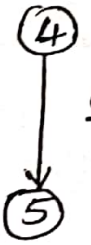

delete 1 →




delete 2 →




delete 3 →




delete 4 → (5) → delete 5 →

∴ 1 → 2 → 3 → 4 → 5 (Topological Order)

Assignment :- Obtain the topological ordering using source removal method.