Algorithm and Analysis

Introduction and Sorting

Introduction to Algorithms:

An algorithm is:

- a set of rules for carrying out calculation either by hand or on a machine
- a finite step-by-step procedure to achieve a required result
- a sequence of computational steps that transform the input into the output
- a sequence of operations performed on data that have to be organized in data structures
- an abstraction of a program to be executed on a physical machine (model of computation)

All algorithms must satisfy the following criteria:

- **Input:** Zero or more quantities that are externally supplied.
- **Output:** At least one quantity is produced.
- **Definiteness:** Each instruction is clear and unambiguous.
- **Finiteness:** If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.
- **Effectiveness:** Every instruction must be very basic, so that it can be carried out in principal by a person using only pencil and paper.

The Classification of Algorithms:

Dynamic programming algorithms: Remembers older results and attempts to use these to speed the process of finding new results.

Greedy algorithms: Greedy algorithms attempt not only to find a solution but also find the ideal solution to any given problem.

Brute force algorithms: The brute force approach starts at some random point and iterates through every possibility until it finds the solution.

Randomized algorithms: Includes any algorithm that uses a random number at any point during its process.

Branch and bound algorithms: Branch and bound algorithms form a tree of sub problems to the primary problem, following each branch until it is either solved or lumped in with another branch.

Simple recursive algorithms: This type of algorithm goes for a direct solution immediately and then backtracks to find a simpler solution.

Backtracking algorithms: Backtracking algorithms test for a solution, if a solution is found the algorithm has solved, if not it recurs once and tests again, continuing until a solution is found.

Divide and conquer algorithms: Divide and conquer algorithm is similar to a branch and bound algorithm, except that it uses backtracking method of recurring in tandem with dividing a problem into sub problems.

Analysis of Algorithms:

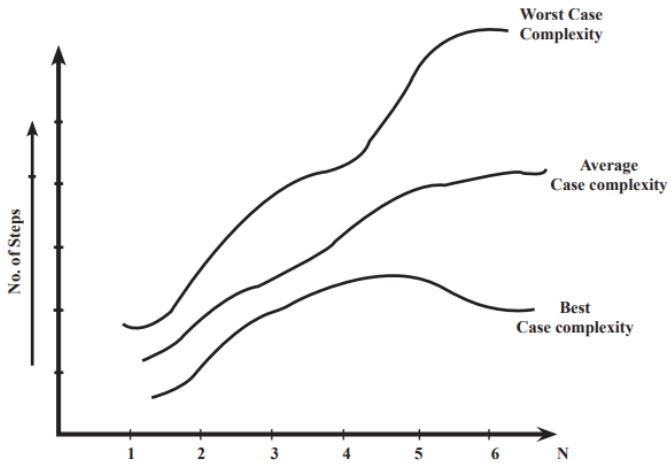
- Analysis of algorithms is the process of finding the computational complexity of algorithms – the amount of time, storage, or other resources needed to execute them.
- The resource usage of algorithms can be divided into:
    i. **Memory (space)**
    Space Complexity: The space complexity of an algorithm is the amount of memory it needs to run to completion.
    ii. **Time**
    Time Complexity: The time complexity of an algorithm is the amount of computer time it needs to run to completion.
- Performance evaluation:
    Performance measurement or a posteriori testing: Implementing the algorithm in a machine and then calculating the time taken by the system to execute the program successfully.
    Performance Analysis or a priori estimates: Before implementing the algorithm in a system. This is done as follows:
        1. How long the algorithm takes will be represented as a function of the size of the input. f (n) →how long it takes if 'n' is the size of input.
        2. How fast the function that characterizes the running time grows with the input size. "Rate of growth of running time". The algorithm with less rate of growth of running time is considered better.
- We perform the following types of analysis:

    i. **Worst-case** – Maximum number of steps taken on any instance of size **a**.

    ii. **Best-case** – Minimum number of steps taken on any instance of size **a**.

    iii. **Average case** – Average number of steps taken on any instance of size **a**.

    iv. **Amortized** – A sequence of operations applied to the input of size **an** averaged over time.



Growth of Functions (Asymptotic notations):

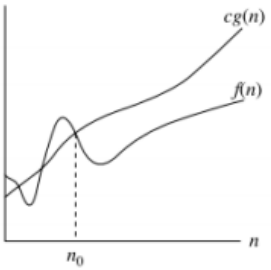
Asymptotic notation

- It is a way to describe the characteristics of a function in the limit.
- It describes the rate of growth of functions.
- Focus on what's important by abstracting away low-order terms and constant factors.
- It is a way to compare "sizes" of functions:
  i. O≈ ≤
  ii. Ω≈ ≥
  iii. Θ ≈ =
  iv. o ≈ <
  v. ω ≈ >

Order of growth: It is described by the highest degree term of the formula for running time. (Drop lower-order terms. Ignore the constant coefficient in the leading term.)

***O*-notation**

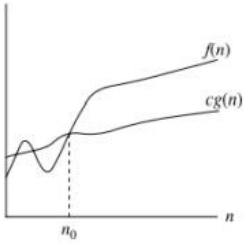
$O(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $0 \le f(n) \le cg(n)$ for all $n \ge n_0\}$.



$g(n)$ is an ***asymptotic upper bound*** for $f(n)$.

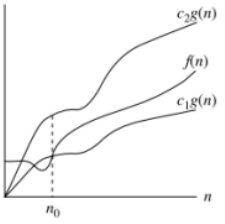
Ω-notation

$\Omega(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $0 \le cg(n) \le f(n)$ for all $n \ge n_0\}$.



Θ-notation

$\Theta(g(n)) = \{f(n)$: there exist positive constants c_1, c_2, and n_0 such that $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0\}$.



***o*-notation**

$o(g(n)) = \{f(n)$: for all constants $c > 0$, there exists a constant $n_0 > 0$ such that $0 \le f(n) < cg(n)$ for all $n \ge n_0\}$

Another view, probably easier to use: $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

***ω*-notation**

$\omega(g(n)) = \{f(n)$: for all constants $c > 0$, there exists a constant $n_0 > 0$ such that $0 \le cg(n) < f(n)$ for all $n \ge n_0\}$.

Another view, again, probably easier to use: $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$.

- O(1) to mean a computing time that is a constant.

```
main()
{
        int n;                  1 unit of time
        scanf("%d", &n);        1 unit of time
        printf("n=%d", n);      1  unit of time

}
```

// total unit of time is 1 + 1+ 1 => 3 units

Regardless of the constant value, the time complexity is O(1)

- O(n) is called linear

```
main()
{
        int i, n;                       1 unit of time
        scanf("%d", &n);        1  unit of time
        for(i=0; i<n; i++)
        //statement;                n   units of time
}
// total unit of time is   2+n
//Therefore time complexity is O(n)
```

- $O(n^2)$ is called quadratic.

```
main()
{
        int i, a[20], n;                1 unit of time
        scanf("%d", &n); 1 unit of time
        for(i=0; i<n; i++)
          //statement;     n units of time
        for(i=0; i<n; i++)
          for(j=0; j<n; j++)
                  //statements      (n*n) units of time
}
//  total units of time is   n2 + n+ 2
//Therefore time complexity is O(n2)
```

- $O(n^3)$ is called cubic

```
for(int i = 0; i < n; i++)
   for(int j = 0; j < n; j++)
      for(int k = 0; k < n; k++)
          print "hello";     (n*n*n) units of time
Time complexity is O(n3)
```

- $O(2^n)$ is called exponential.
- O(log n) is called logarithmic.

```
for(i=1 to n)
        {i=i*2}
```

for loop will run from 1 to n with step 2*i
1, 2, 4, 8, 16, 32….
2^0, 2^1,2^2, 2^3….2^n
x = 2^n (x is last term)
log x = log 2^n
log x = n*log2 (log 2 base 2 = 1)
n = log x
for function of n replace x with n,
Time complexity = O(log n)

- O(nlogn) is called linearithmic time. For k> 1, $O(n\log^k n)$ is called lo-linear time.

Sorting Algorithms:

MERGE SORT:

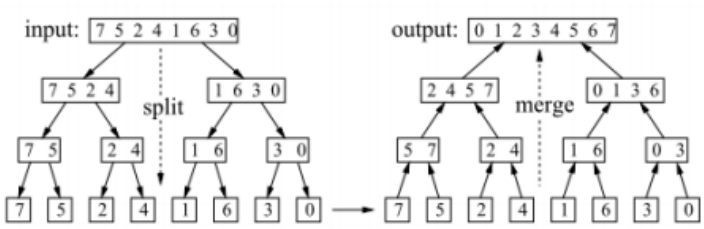
It is one of the well-known divide-and-conquer algorithms. This is a simple and very efficient algorithm for sorting a list of numbers. The major elements of the merge sort algorithm:

Divide: Split A down the middle into two sub-sequences, each of size roughly n/2 .

Conquer: Sort each subsequence (by calling MergeSort recursively on each).

Combine: Merge the two sorted sub-sequences into a single sorted list.

The dividing process ends when we have split the sub-sequences down to a single item.



Merge Sort

Algorithm: Merge-Sort (numbers[], p, r)

```
If p<r

{
   q → ( p+ r)/2   // the middle point to divide the array into two halves
   MERGE-SORT (A, p, q)  // call mergeSort for first half
   MERGE-SORT ( A, q+1,r)  // call mergeSort for second half:
   MERGE ( A, p, q, r)  // Merge the two sorted halves
}
```

Function: Merge (numbers[], p, q, r)

```
n = q – p + 1
n = r – q
declare leftnums[1…n1 + 1] and rightnums[1…n2 + 1] temporary arrays
for  i = 1 to n1
   leftnums[i] = numbers[p + i - 1]
for  j = 1 to n2
    rightnums[j] = numbers[q+ j]
leftnums[n1 + 1] = ∞
rightnums[n + 1] = ∞
i = 1
j = 1
```

```
for k = p to r
   if leftnums[i] ≤ rightnums[j]
      numbers[k] = leftnums[i]
      i = i + 1
   else
      numbers[k] = rightnums[j]
      j = j + 1
```

Analysis:

Let us consider, the running time of Merge-Sort as T(n). Hence

$$T(n) = \begin{cases} c & if\ n \leqslant 1 \\ 2\,x\,T(\frac{n}{2}) + d\,x\,n & otherwise \end{cases}$$ where *c* and *d* are constants

Therefore, using this recurrence relation:

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

As, $i = log\,n,\ T(n) = 2^{log\,n} T(\frac{n}{2^{log\,n}}) + log\,n.d.n$

$$= c.n + d.n.log\,n$$

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

Auxiliary Space: O(n)

Best Case Complexity: The merge sort algorithm has a best-case time complexity of **O(n*log n)** for the already sorted array.

Average Case Complexity: The average-case time complexity for the merge sort algorithm is **O(n*log n)**, which happens when 2 or more elements are jumbled, i.e., neither in the ascending order nor in the descending order.

Worst Case Complexity: The worst-case time complexity is also **O(n*log n)**, which occurs when we sort the descending order of an array into the ascending order.

QUICKSORT

Description of quicksort:

Quicksort is based on the three-step process of divide-and-conquer. It picks an element as pivot and partitions the given array around the picked pivot. The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x.

To sort the subarrayA[p . . r]

Divide: Partition A[p . . r], into two (possibly empty) subarrays A[p . . q − 1] and A[q + 1 . . r], such that each element in the Þrstsubarray A[p . . q − 1] is ≤ A[q] and A[q] is ≤ each element in the second subarray A[q + 1 . . r].

Conquer: Sort the two subarrays by recursive calls to QUICKSORT.

Combine: No work is needed to combine the subarrays, because they are sorted in place.

Perform the divide step by a procedure PARTITION, which returns the index q that marks the position separating the subarrays.

Algorithm:

Quick-Sort (A, p, r)

if p < r then

q ← Partition (A, p, r)

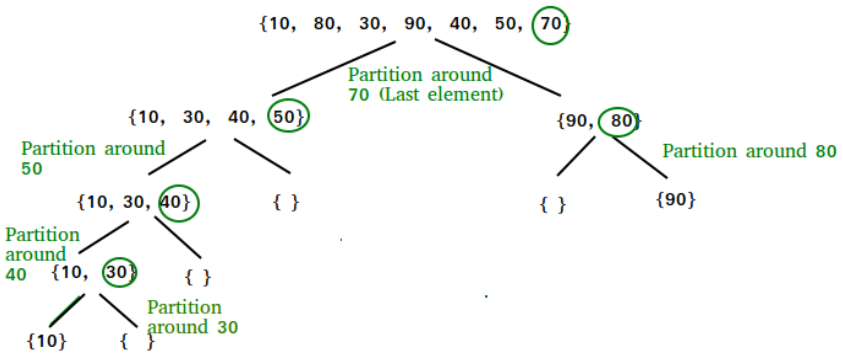
Quick-Sort (A, p, q)

Quick-Sort (A, q + r, r)

Function: Partition (A, p, r)

```
x ← A[p]
i ← p-1
j ← r+1
while TRUE do
    Repeat j ← j - 1
    until A[j] ≤ x
    Repeat i← i+1
    until A[i] ≥ x
    if i < j then
        exchange A[i] ↔ A[j]
    else
        return j
```

Partition always selects the last element A[r] in the subarrayA[p . . r].



Analysis:

T (n) =T(1)+T(n-1)+n **..... (1)**

T (1) is time taken by pivot element.

T (n-1) is time taken by remaining element except for pivot element.

N: the number of comparisons required to identify the exact position of it (every element)

Put T(n-1) in Eqn (1):

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

Put T(n-2) in Eqn (1):

T(n-2) = T(1)+T(n-3)+(n-2)

Put T(n-3) in Eqn (1):

T(n-3) = T(1)+T(n-4)+(n-3)

=>T(n) = (n-1)T(1) + T(1) + n(n+1)/2 – 1

Stopping condition: T(1) = 0

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

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

Worst Case Time Complexity [Big-O]: **$O(n^2)$**

Best Case Time Complexity [Big-omega]: **O(n*log n)**

Average Time Complexity [Big-theta]: **O(n*log n)**

Space Complexity: **O(n*log n)**

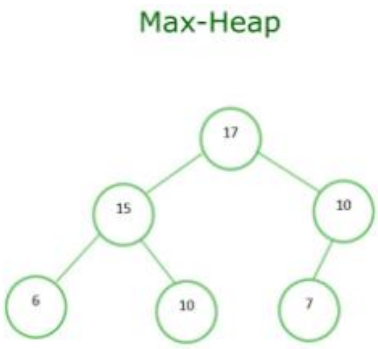
HEAP SORT:

Heap sort is a comparison based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the maximum element and place the maximum element at the end. We repeat the same process for the remaining elements.

Binary Heap:

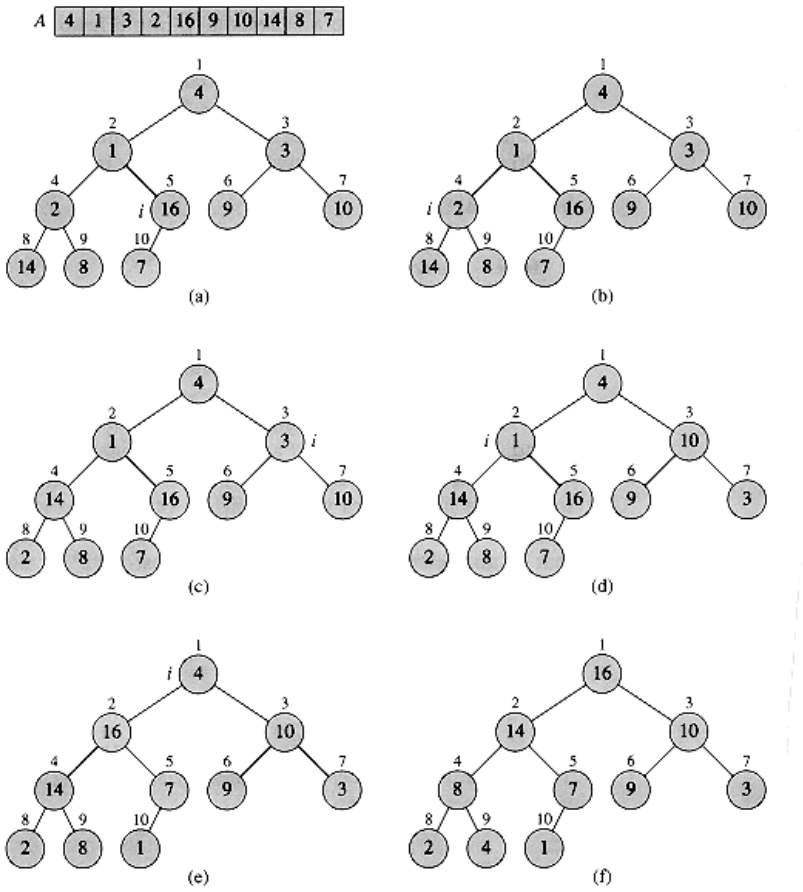
A Binary Heap is a Complete Binary Tree where items are stored in a special order such that value in a parent node is greater(or smaller) than the values in its two children nodes. The former is called as max heap and the latter is called min-heap. The heap can be represented by a binary tree or array.

Max-heap:



In a Max-Heap the key present at the root node must be greater than or equal among the keys present at all of its children.

A-Max Heap is a Complete Binary Tree. A-Max heap is typically represented as an array. The root element will be at Arr[0]. Below table shows indexes of other nodes for the ith node, i.e., Arr[i]:
Arr[(i-1)/2] Returns the parent node.
Arr[(2*i)+1] Returns the left child node.
Arr[(2*i)+2] Returns the right child node.



Algorithm:

MAX-HEAPIFY (A, i)

l ← left [i]

r ← right [i]

if l≤ heap-size [A] and A[l] > A [i]

then largest ← l

Else largest ← i

If r≤ heap-size [A] and A [r] > A[largest]

Then largest ← r

If largest ≠ i

Then exchange A [i] A [largest]

MAX-HEAPIFY (A, largest)

BUILDHEAP (array A, int n)

for i ← n/2 down to 1

do

HEAPIFY (A, i, n)

HEAP-SORT (A)

BUILD-MAX-HEAP (A)

For I ← length[A] down to Z

Do exchange A [1] ←→ A [i]

Heap-size [A] ← heap-size [A]-1

MAX-HEAPIFY (A,1)

Analysis:

The running time of MAX-HEAPIFY by the recurrence can be described as:

T (n) < = T (2n/3) + O (1)

Using Iteration method:

T (n) < = T (4n/9) + 1+1

T (n) < = T (8n/27) + 1+1+1

$T(n) <= T((2/3)^k n)+(k).1$

The solution to this recurrence is **T(n)=O(n log n).**

Worst Case Time Complexity: **O(n*log n)**

Best Case Time Complexity: **O(n*log n)**

Average Time Complexity: **O(n*log n)**

Space Complexity : **O(1)**

RADIXSORT:

Radix Sort is an efficient non-comparison based sorting algorithm. It uses another algorithm namely Counting Sort as a subroutine. Radix Sort takes advantage of the following ideas: Number of digits in an Integer is determined by:

- its base
- very less compared to the data

Numbers increase linearly but number of digits increase logarithmically.

Radix sort was developed to **sort large integers**. It considers integer as a string of digits so we can use Radix Sort to sort strings as well.

Algorithm:

```
Radix-Sort(A, d)
for j = 1 to d do
   int count[10] = {0};
   for i = 0 to n do
      count[key of(A[i]) in pass j]++
   for k = 1 to 10 do
      count[k] = count[k] + count[k-1]
   for i = n-1 downto 0 do
      result[ count[key of(A[i])] ] = A[j]
      count[key of(A[i])]--
   for i=0 to n do
      A[i] = result[i]
end for(j)
end func
```

Analysis:

Best case time complexity: Ω(nk)

Average case time complexity: Θ(nk)

Worst case time complexity: O(nk)

Space complexity: O(n+k)

where n is the number of input data and k is the maximum element in the input data.

Let there be d digits in input integers. Radix Sort takes *O(d(n+b))** time where b is the base for representing numbers, for example, for decimal system, b is 10. What is the value of d? If k is the maximum possible value, then d would be O(logb(k)). So overall time complexity is O((n+b) * logb(k)).

BUBBLESORT:

Bubble Sort, also known as Exchange Sort, is a simple sorting algorithm. It works by repeatedly stepping throughout the list to be sorted, comparing two items at a time and swapping them if they are in the wrong order. The pass through the list is duplicated until no swaps are desired, which means the list is sorted.

1) The bubble sort starts with the very first index and makes it a bubble element. Then it compares the bubble element, which is currently our first index element, with the next element. If the bubble element is greater and the second element is smaller, then both of them will swap.
2) After swapping, the second element will become the bubble element. Now we will compare the second element with the third as we did in the earlier step and swap them if required. The same process is followed until the last element.

Unsorted list:	5	2	1	4	3	7	6
1st iteration:							
5 > 2 swap	2	5	1	4	3	7	6
5 > 1 swap	2	1	5	4	3	7	6
5 > 4 swap	2	1	4	5	3	7	6
5 > 3 swap	2	1	4	3	5	7	6
5 < 7 no swap	2	1	4	3	5	7	6
7 > 6 swap	2	1	4	3	5	6	7

2nd iteration:

2 > 1 swap	1	2	4	3	5	6	7
2 < 4 no swap	1	2	4	3	5	6	7
4 > 3 swap	1	2	3	4	5	6	7
4 < 5 no swap	1	2	3	4	5	6	7
5 < 6 no swap	1	2	3	4	5	6	7

There is no change in 3rd, 4th, 5th and 6th iteration.

Finally,

the sorted list is	1	2	3	4	5	6	7

Algorithm: Sequential-Bubble-Sort (A)

for i← 1 to length [A] do

for j ← length [A] down-to i +1 do

if A[A] < A[j - 1] then

Exchange A[j] ↔ A[j-1]

voidbubbleSort(int numbers[], intarray_size) {

inti, j, temp;

for (i = (array_size - 1); i >= 0; i--)

for (j = 1; j <= i; j++)

if (numbers[j - 1] > numbers[j]) {

temp = numbers[j-1];

numbers[j - 1] = numbers[j];

numbers[j] = temp;

}

}

Analysis:

Input: Given n input elements.

Output: Number of steps incurred to sort a list.

Logic: If we are given n elements, then in the first pass, it will do n-1 comparisons; in the second pass, it will do n-2; in the third pass, it will do n-3 and so on. Thus, the total number of comparisons can be found by:

Output;

(n-1) + (n-2) + (n-3) + (n-4) + ... + 1

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

i.e., $O(n^2)$

Therefore, the bubble sort algorithm encompasses a time complexity of **$O(n^2)$** and a space complexity of **O(1)** because it necessitates some extra memory space for temp variable for swapping.

- **Best Case Complexity**: The bubble sort algorithm has a best-case time complexity of **O(n)** for the already sorted array.
- **Average Case Complexity**: The average-case time complexity for the bubble sort algorithm is **$O(n^2)$**, which happens when 2 or more elements are in jumbled, i.e., neither in the ascending order nor in the descending order.
- **Worst Case Complexity**: The worst-case time complexity is also **$O(n^2)$**, which occurs when we sort the descending order of an array into the ascending order.

INSERTIONSORT:

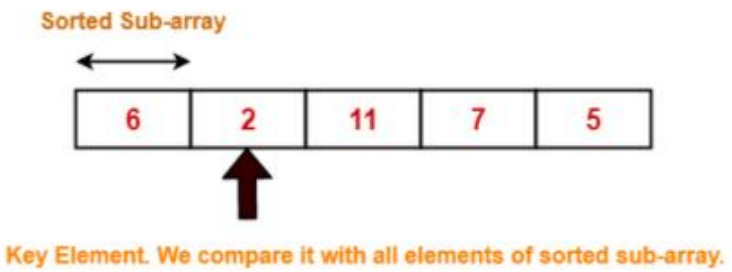
- Insertion sort is an in-place sorting algorithm.
- It uses no auxiliary data structures while sorting.
- It is inspired from the way in which we sort playing cards.

Consider the following elements are to be sorted in ascending order-

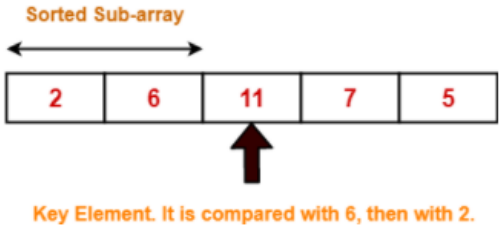
6, 2, 11, 7, 5

The above insertion sort algorithm works as illustrated below-

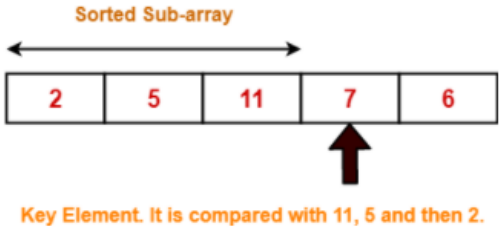
Step-01: For i = 1



Step-02: For i = 2

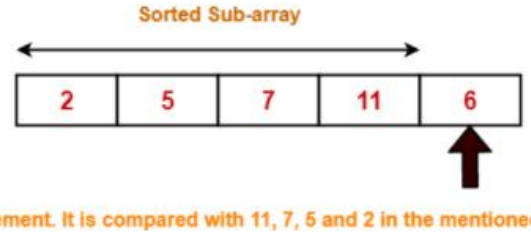


Step-03: For i = 3



Working of inner loop when i = 3

Step-04: For i = 4



Loop gets terminated as 'i' becomes 5. The state of array after the loops are finished-

With each loop cycle,

- One element is placed at the correct location in the sorted sub-array until array A is completely sorted.

Algorithm:

```
for j = 2 to A.length
key = A[j]
 // Insert A[j] into the sorted sequence A[1.. j - 1]
 i = j - 1
 while i > 0 and A[i] > key
    A[i + 1] = A[i]
    i = i -1
 A[i + 1] = key
```

Analysis:

T(n) = (n-1) + (n-2) + (n-3) + (n-4) + ... + 1

Time Complexity: O(n*2)

Auxiliary Space: O(1)

SELECTIONSORT:

Selection sort works as:

1. It finds the first smallest element.
2. It swaps it with the first element of the unordered list.
3. It finds the second smallest element.
4. It swaps it with the second element of the unordered list.
5. Similarly, it continues to sort the given elements.

Consider the following elements are to be sorted in ascending order-

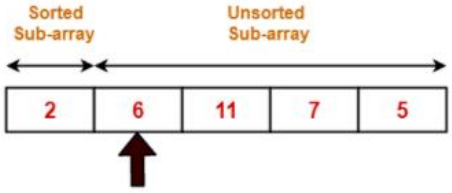
6, 2, 11, 7, 5

The above selection sort algorithm works as illustrated below-

Step-01: For i = 0

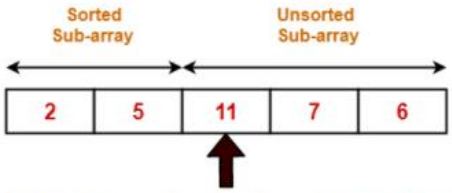
We start here, find the minimum element and swap it with the 1st element of array

Step-02: For i = 1



We start here, find the minimum element and swap it with the 2nd element of array

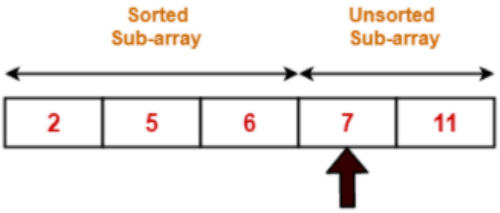
Step-03: For i = 2



We start here, find the minimum element and swap it with the 3rd element of array

---

Step-04: For i = 3

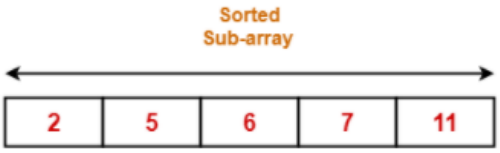


We start here, find the minimum element but there is no need to swap
(4th element is itself the minimum)

Step-05: For i = 4

Loop gets terminated as 'i' becomes 4.

The state of array after the loops are finished is as shown-



With each loop cycle,

- The minimum element in unsorted sub-array is selected.
- It is then placed at the correct location in the sorted sub-array until array A is completely sorted.

Algorithm:

```
Algorithm: Selection-Sort (A)
for  i ← 1 to n-1 do
  min j ← i;
  min x ← A[i]
  for j ←i + 1 to n do
    if A[j] < min x then
      min j ← j
      min x ← A[j]
  A[min j] ← A [i]
  A[i] ← min x
```

Analysis:

Input: Given n input elements.
Output: Number of steps incurred to sort a list.
Logic: If we are given n elements, then in the first pass, it will do n-1 comparisons; in the second pass, it will do n-2; in the third pass, it will do n-3 and so on. Thus, the total number of comparisons can be found by:

Output;

$(n-1) + (n-2) + (n-3) + (n-4) + \ldots\ldots + 1$

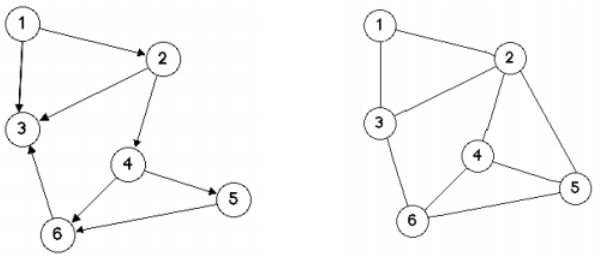
$$Sum = \frac{n(n-1)}{2}$$

i.e., $O(n^2)$

- **Best Case Complexity:** The selection sort algorithm has a best-case time complexity of **$O(n^2)$** for the already sorted array.
- **Average Case Complexity:** The average-case time complexity for the selection sort algorithm is **$O(n^2)$**, in which the existing elements are in jumbled ordered, i.e., neither in the ascending order nor in the descending order.
- **Worst Case Complexity:** The worst-case time complexity is also **$O(n^2)$**, which occurs when we sort the descending order of an array into the ascending order.

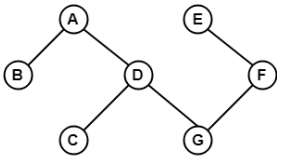
GRAPHS:

A simple graph G consists of a set of vertices V and a set of edges E. The elements of E are defined as pairs of elements of V, ek = (u, v) such that u is not equal to v and (u, v) an element of E implies that (v, u) is also an element of E. (In other words (u, v) and (v, u) represent the same edge). Graphs can be represented pictorially by nodes and lines as shown below:

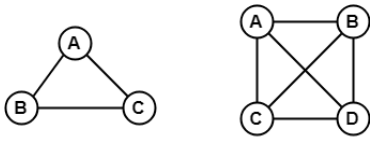


TYPES OF GRAPHS:

- **Multigraphs:** Allow multiple edges between the same pair or vertices and edges from and to the same vertex. The edges of a directed graph are called arcs and have a direction as indicated by an arrow. Unlike graphs, an arc (u,v) in a directed graph does not imply that the arc (v,u) is also in the directed graph.
- **Acyclic graph**: An acyclic graph is a graph with no cycles. That is, there is no path along edges in the graph (or along arcs in a directed graph) that leads from a vertex back to the same vertex.

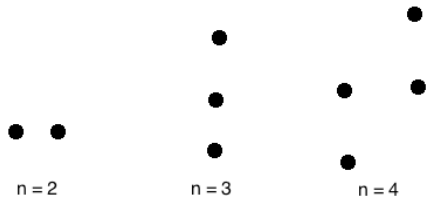


- **Complete Graph:** A complete graph is one in which there is an edge between every pair of vertices.



- **Null Graph**

  A **null graph** is a graph in which there are no edges between its vertices. A null graph is also called empty graph.



  A null graph with n vertices is denoted by Nn.

- **Trivial Graph:** A **trivial graph** is the graph which has only one vertex.

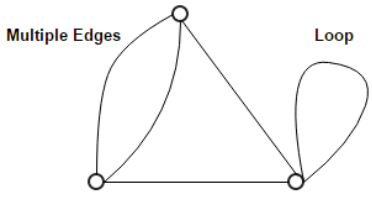
  Example

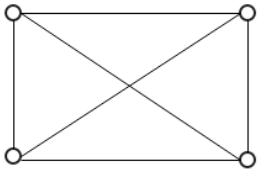
- **Simple Graph:**

  A **simple graph** is the undirected graph with **no parallel edges** and **no loops**. A simple graph which has n vertices, the degree of every vertex is at most n -1.

  Example

  Not a Simple Graph

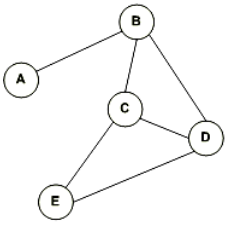
  

  Simple Graph

- **Undirected Graph:**

  An **undirected graph** is a graph whose edges are **not directed**.

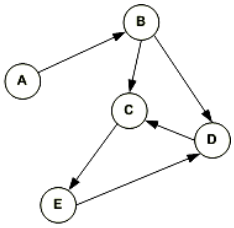
  Example

- **Directed Graph:**

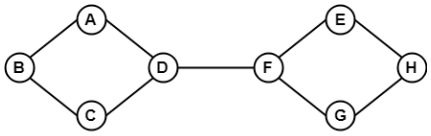
  A **directed graph** is a graph in which the **edges are directed** by arrows. Directed graph is also known as **digraphs**.

  Example

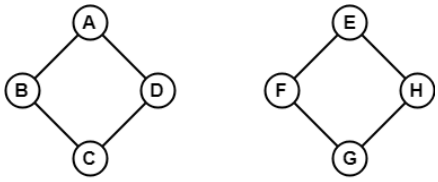
- **Connected Graph:**

  A **connected graph** is a graph in which we can visit from any one vertex to any other vertex. In a connected graph, at least one edge or path exists between every pair of vertices.



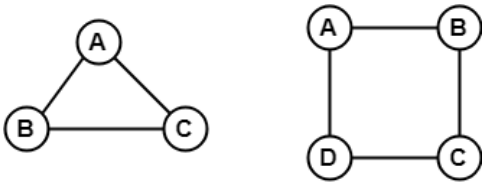
- **Disconnected Graph:**

  A **disconnected graph** is a graph in which any path does not exist between every pair of vertices.



- **Regular Graph:**

  A **Regular graph** is a graph in which degree of all the vertices is same. If the degree of all the vertices is k, then it is called k-regular graph.



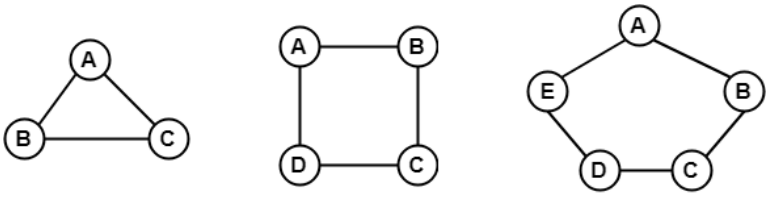
- **Cyclic graph:**

  A graph with 'n' vertices (where, n>=3) and 'n' edges forming a cycle of 'n' with all its edges is known as **cycle graph**.

  A graph containing at least one cycle in it is known as a **cyclic graph**.

  In the cycle graph, degree of each vertex is 2.

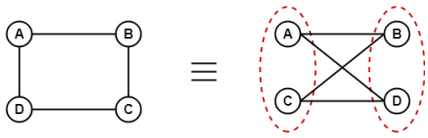
  The cycle graph which has n vertices is denoted by Cn.



Bipartite Graph:

A **bipartite graph** is a graph in which the vertex set can be partitioned into two sets such that edges only go between sets, not within them. A graph G (V, E) is called bipartite graph if its vertex-set V(G) can be decomposed into two non-empty disjoint subsets V1(G) and V2(G) in such a way that each edge e ∈ E(G) has its one last joint in V1(G) and other last point in V2(G).

The partition V = V1 ∪ V2 is known as bipartition of G.

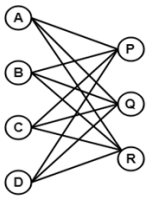


Complete Bipartite Graph

A **complete bipartite graph** is a bipartite graph in which each vertex in the first set is joined to each vertex in the second set by exactly one edge.

A complete bipartite graph is a bipartite graph which is complete.

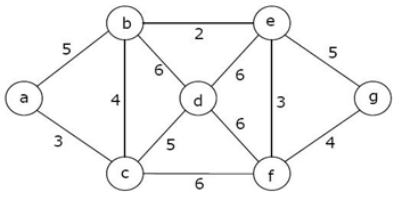
Complete Bipartite graph = Bipartite graph + Complete graph



The above graph is known as $K_{4,3}$.

Weighted Graph

A weighted graph is a graph whose edges have been labeled with some weights or numbers. The length of a path in a weighted graph is the sum of the weights of all the edges in the path.



In the above graph, if path is a -> b -> c -> d -> e -> g then the length of the path is 5 + 4 + 5 + 6 + 5 = 25.

- Two vertices u, v in a graph are said to be adjacent if there is an edge e (or arc) connecting u to v. The vertices u and v are called the endpoints of e.
- The degree of a vertex v is given as deg (v) and is the number of edges incident with v. That is, the number of edges for which v is an endpoint.

GRAPH TRAVERSALS

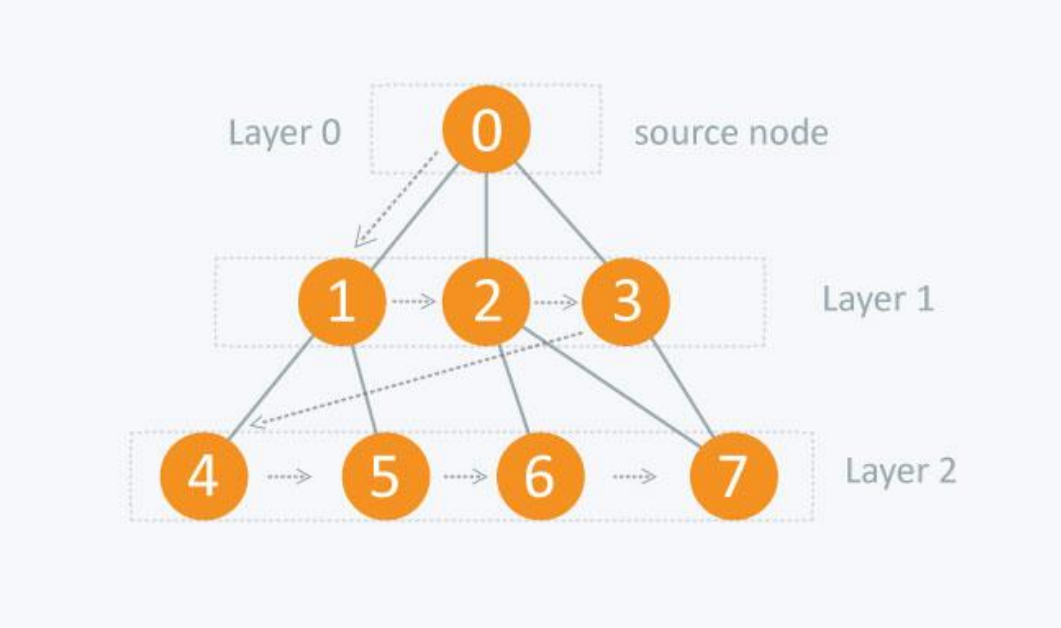
Graph traversal means visiting every vertex and edge exactly once in a well-defined order. While using certain graph algorithms, you must ensure that each vertex of the graph is visited exactly once.

Breadth First Search (BFS):

BFS is a traversing algorithm where you should start traversing from a selected node (source or starting node) and traverse the graph layerwise thus exploring the neighbour nodes (nodes which are directly connected to source node). You must then move towards the next-level neighbour nodes. uses a queue to remember to get the next vertex to start a search.

As the name BFS suggests, you are required to traverse the graph breadthwise as follows:

1. First move horizontally and visit all the nodes of the current layer
2. Move to the next layer



Algorithm:

Step 1: SET STATUS = 1 (ready state)
for each node in G

Step 2: Enqueue the starting node A
and set its STATUS = 2
(waiting state)

Step 3: Repeat Steps 4 and 5 until
QUEUE is empty

Step 4: Dequeue a node N. Process it
and set its STATUS = 3
(processed state).

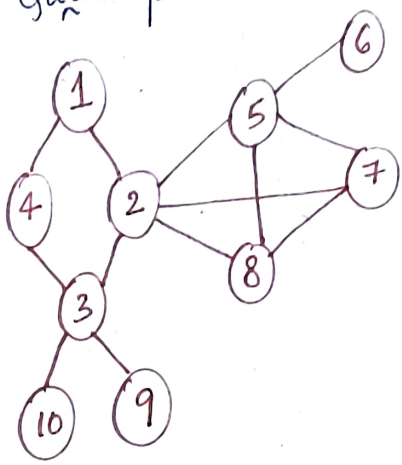
Step 5: Enqueue all the neighbours of
N that are in the ready state
(whose STATUS = 1) and set
their STATUS = 2
(waiting state)
[END OF LOOP]

Step 6: EXIT

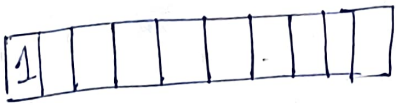
BFS (Breadth First Search)

* Visit Node
* Explore the Node
* Uses Queue for traversal.

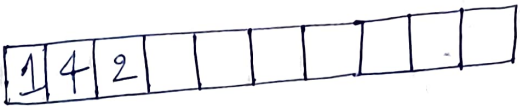
Graph:

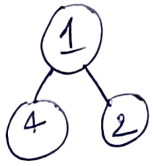


Step 1: Start with Vertex 1. Store in Queue.

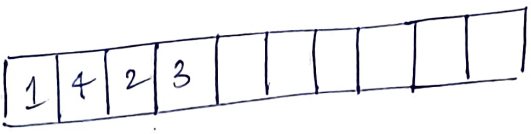


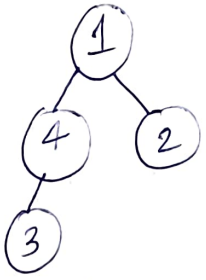
Step 2: Explore 1. 4 and 2 Discovered. Store in Queue.



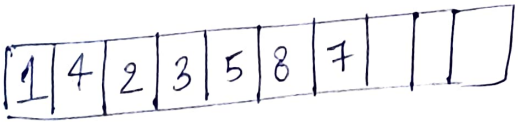


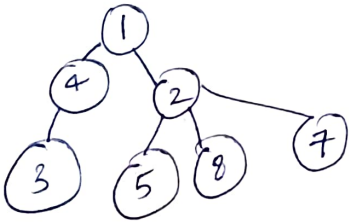
Step 3: Vertex 1 visited. Move to node that appears next in Queue i.e. 4. Explore 4. Vertex 3 Discovered. Store in Queue.





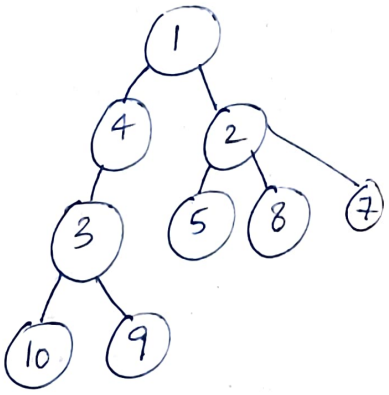
Step 4: Vertex 4 explored. Move to 2. Vertices 5, 8, 7 discovered. Store in Queue.





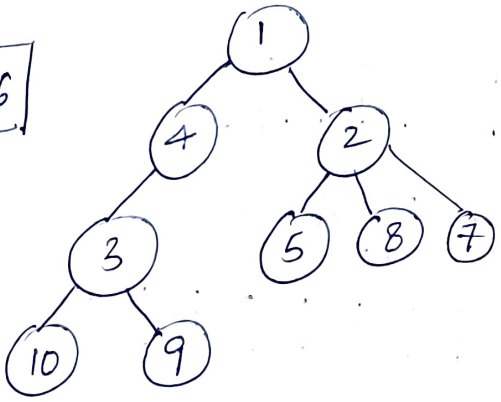
Step 5: Vertex 2 Visited. Move to 3. Vertices 10 and 9 discovered. Store in Queue.

1	4	2	3	5	8	7	10	9	



Step 6: Vertex 3 explored. Move to 5. 6 is Discovered. Store in Queue.

1	4	2	3	5	8	7	10	9	6



All Vertices Visited.

BFS ORDER: 1, 4, 2, 3, 5, 8, 7, 10, 9, 6

Time complexity of BFS (Breadth First Search):

Time Complexity of BFS = O(V+E) where V is vertices and E is edges.

DFS (Depth First Search):

Depth First Search (DFS) algorithm traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

Algorithm:

procedure DFS-iterative(G,v):

```
let S be a stack
S.push(v)
While S is not empty
        v ← S.pop()
        if v is not labeled as discovered:
                label v as discovered
                for all edges from v to winG.adjacentEdges(v) do
                        S.push(w)
```

Applications:

1. Finding connected components.
2. Topological sorting.
3. Finding the bridges of a graph.
4. Finding strongly connected components.
5. Finding bi-connectivity in graphs.

Analysis:

The time complexity of DFS if the entire tree is traversed is O(V) where V is the number of nodes.

If the graph is represented as adjacency list:

Here, each node maintains a list of all its adjacent edges. Let's assume that there are V number of nodes and E number of edges in the graph.

For each node, we discover all its neighbors by traversing its adjacency list just once in linear time.

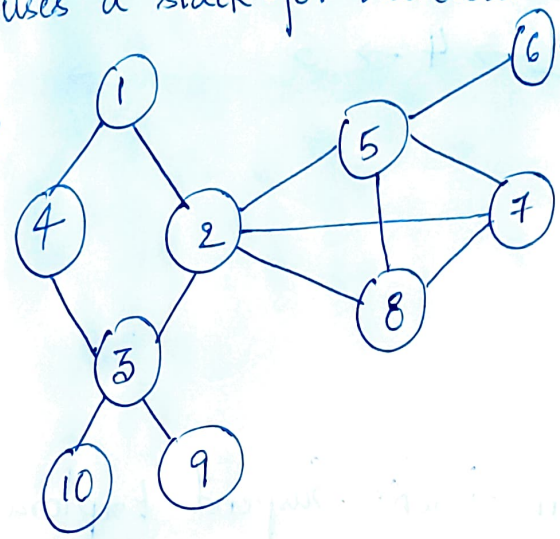
For a directed graph, the sum of the sizes of the adjacency lists of all the nodes is E. So, the time complexity in this case is O(V) + O(E) = O(V + E).

For an undirected graph, each edge appears twice. Once in the adjacency list of either end of the edge. The time complexity for this case will be O(V) + O (2E) ~ O(V + E)

DEPTH FIRST SEARCH (DFS):

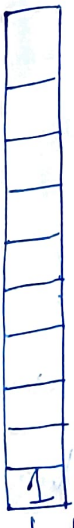
* It uses a stack for traversal.

GRAPH:



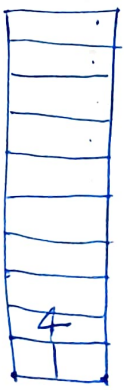
TRAVERSAL STEPS:

Step 1: Start with vertex 1. Vertex 1 visited. Store in stack. Visit any adjacent vertex of 1. 4 is discovered.



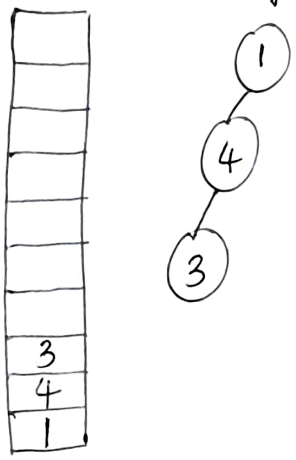
1

Step 2: Store 4 in stack. Visit any adjacent vertex of 4. 3 is discovered.



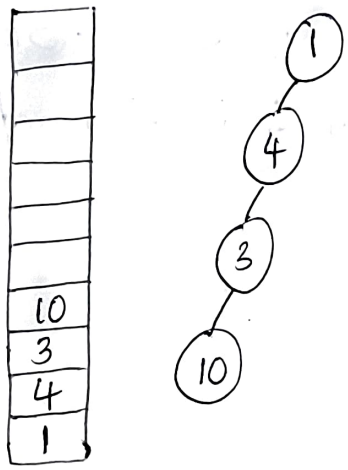
1 → 4

Step 3: Store 3 in stack. Suspend exploration of 4. Explore 3. Vertex 10 is discovered.



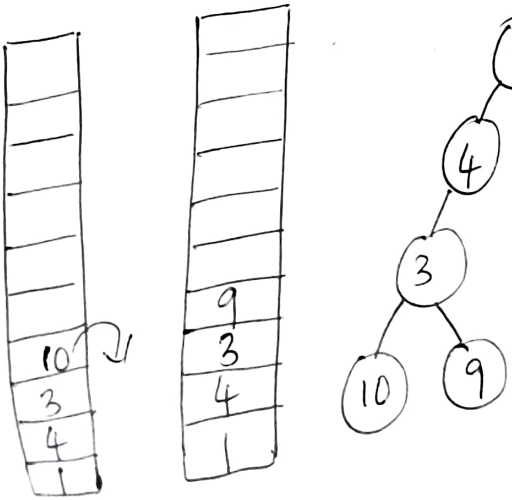
1 → 4 → 3

Step 4: Store 10 in Stack. Suspend Exploration of 3. Explore 10. No adjacent vertices discovered.



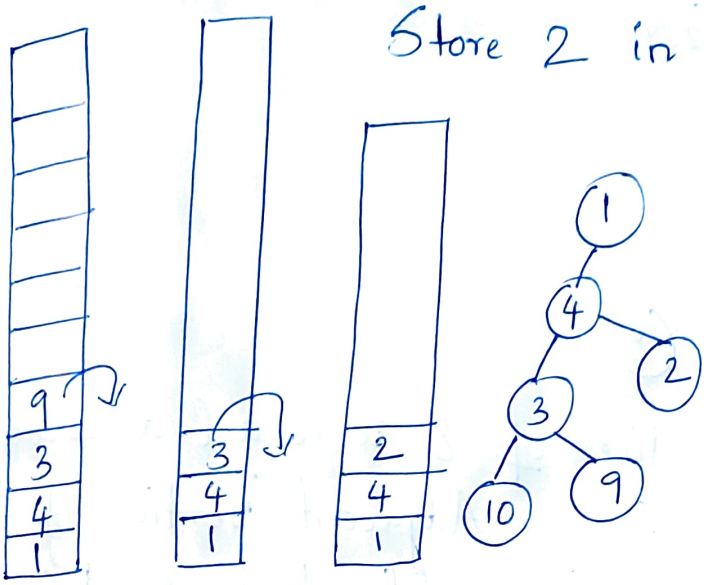
1 → 4 → 3 → 10.

Step 5: Since 10 has no adjacent vertices, we back-track. We pop 10 out of the stack and backtrack to vertex 3. Explore 3. Vertex 9 is discovered. Store 9.



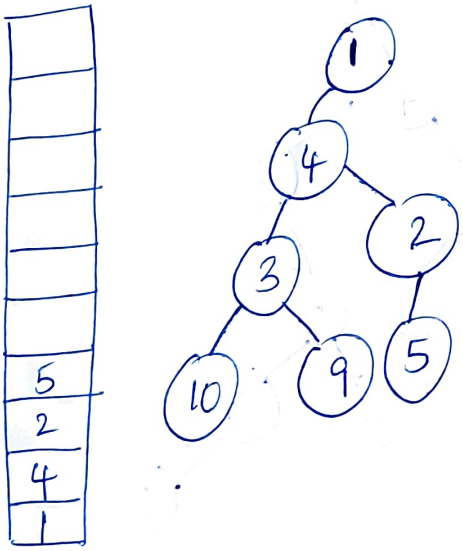
1 → 4 → 3 → 10 → 9

Step 6: Since 9 has no adjacent vertices, we pop 9 and backtrack to 3. Now 3 has no adjacent vertices and we pop it from stack & backtract to 4. 2 is discovered. Store 2 in stack.



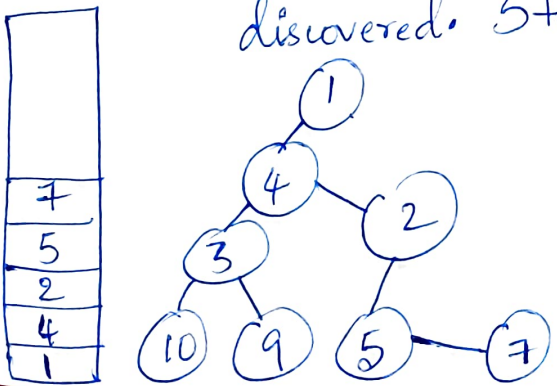
1 → 4 → 3 → 10 → 9 → 2

Step 7: Suspend Exploration of 4. Explore 2. 5 is discovered. Store 5 in stack.

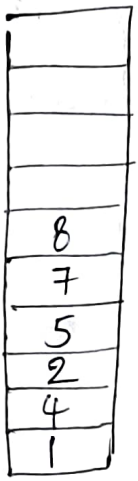


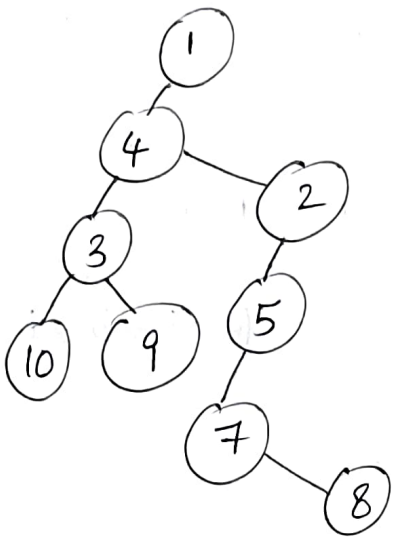
1 → 4 → 3 → 10 → 9 → 2 → 5

Step 8: Suspend exploration of 2. Explore 5. 7 is discovered. Store 7 in stack.



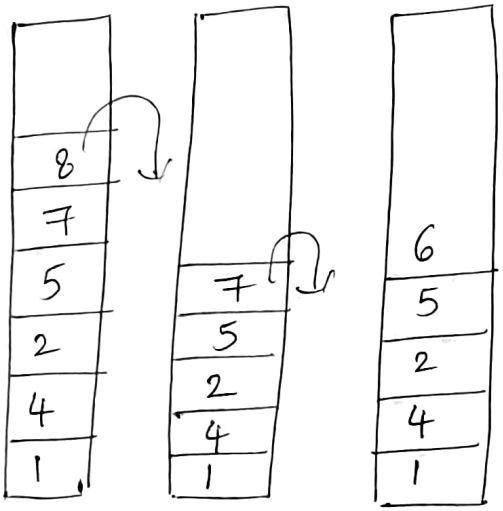
Step 9: Suspend exploration of 5. Explore 7. 8 is discovered. Store in stack.

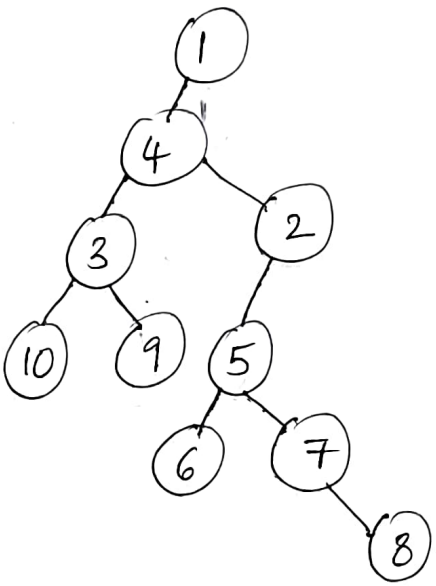




1 → 4 → 3 → 10 → 9 → 2 → 5 → 7
↓
8

Step 10: Adjacent vertices of 8 were all discovered. So backtrack to 7. Adjacent vertices of 7 all discovered to backtrack to 5. We discover 6. Store 6 in stack.





1 → 4 → 3 → 10 → 9 → 2 → 5 → 7 → 8 → 6

All Vertices visited.

Dijkstra's Algorithm:

It is a greedy algorithm that solves the single-source shortest path problem for a directed graph G = (V, E) with nonnegative edge weights.

Algorithm:

```
function Dijkstra(Graph, source):
    dist[source]  := 0                  // Distance from source to source is set to 0
    for each vertex v in Graph:         // Initializations
        if v ≠ source
            dist[v]  := infinity        // Unknown distance function from source to each node set to infinity
        add v to Q                      // All nodes initially in Q
    while Q is not empty:               // The main loop
        v := vertex in Q with min dist[v]  // In the first run-through, this vertex is the source node
        remove v from Q
    for each neighbor u of v:           // where neighbor u has not yet been removed from Q.
        alt := dist[v] + length(v, u)
        if alt < dist[u]:               // A shorter path to u has been found
            dist[u]  := alt             // Update distance of u
    return dist[]
end function
```

Analysis:

The running time of Dijkstra's algorithm on a graph with edges E and vertices V can be expressed as a function of |E| and |V| using the Big - O notation. The simplest implementation of the Dijkstra's algorithm stores vertices of set Q in an ordinary linked list or array, and operation Extract - Min (Q) is simply a linear search through all vertices in Q. In this case, the running time is O $(|V^2|+|E|=O(V^2)$.

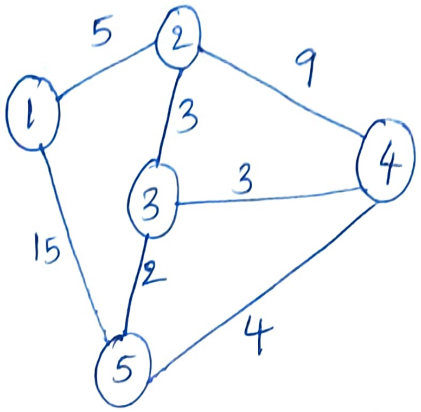
Applications:

- To find the shortest path
- In social networking applications
- In a telephone network
- To find the locations in the map

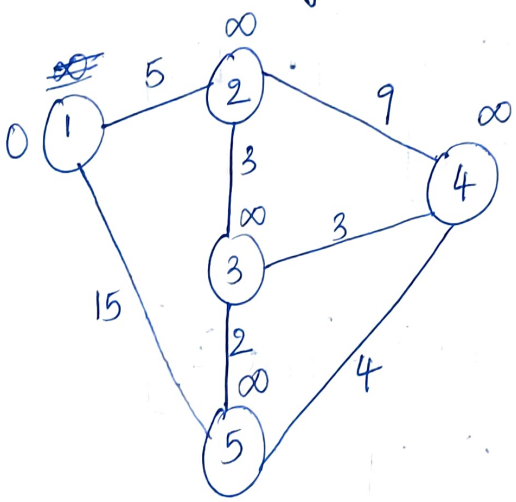
DIJKSTRA'S ALGORITHIM:

(Single Source Shortest path Algorithm)

Given Graph: Source is '1' (undirected graph)



Initially, we set the distance from source '1' to all other vertices as ~~zero~~ infinity. (Source to source distance is zero)



Dijkstra's formula: if $([d(u)+d(u,v)] < d(v))$ then we update the distance to the lesser value.

① $d(1,2) = \min[d(1)+d(1,2), d(2)]$

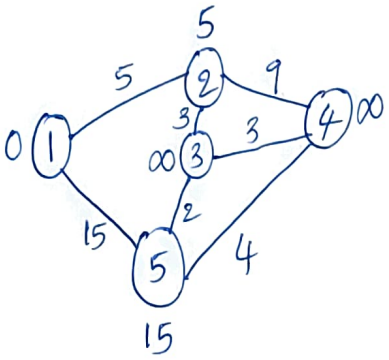
$= \min[0+5, \infty]$

$= 5$

② $d(1,5) = \min\left[[d(1)+d(1,5)], d(5)\right]$

$= \min\left[(0+15), \infty\right]$

$= 15$



Vertex '1' is visited. We choose the next shortest value which is 5. So we move to vertex 2. Vertex 2 has Vertices 3 and 4 as adjacent.

$d(2,3) = \min\left[[d(2)+d(2,3)], d[3]\right]$

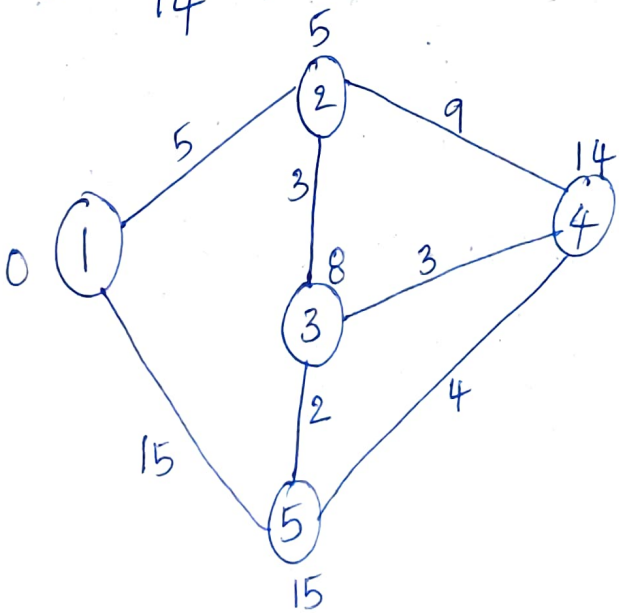
$= \min\left[(5+3), \infty\right]$

$= 8$

$d(2,4) = \min\left[[d(2)+d(2,4)], d[4]\right]$

$= \min\left[(5+9), \infty\right]$

$= 14$



Vertex 2 is visited. We choose next shortest value which is 8. We move to vertex 3. From 3, we can move to 2, 4 and 5. Since 2 is already visited we only consider 4 and 5.

$$d(3,4) = \min\left[\left(d(3)+d(3,4)\right), d(4)\right]$$

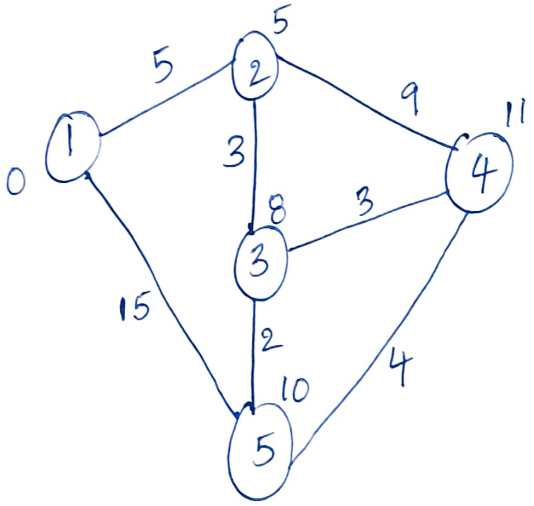
$$= \min\left[(8+3), 14\right]$$

$$= 11$$

$$d(3,5) = \min\left[\left(d(3)+d(3,5)\right), d(5)\right]$$

$$= \min\left[(8+2), 15\right]$$

$$= 10$$

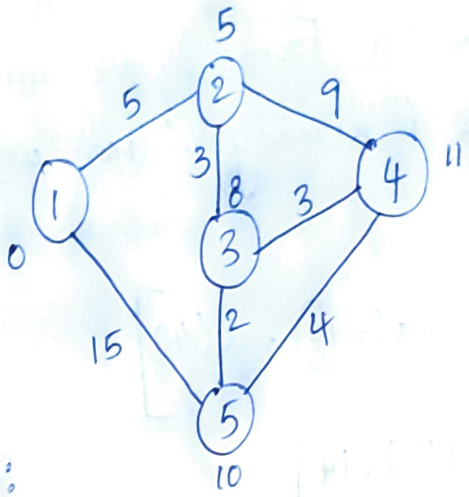


Vertex 3 is visited. We move to next shortest value which is '10'. We move to 5. From '5' we can move to 1, 3 and 4. Since 1 and 3 are already visited, we ~~move to~~ consider 4.

$$d(5,4) = \min\left[\left(d(5)+d(5,4)\right)+d(4)\right]$$

$$= \min\left[(10+4), 11\right]$$

$$= 11$$



Final Costs:

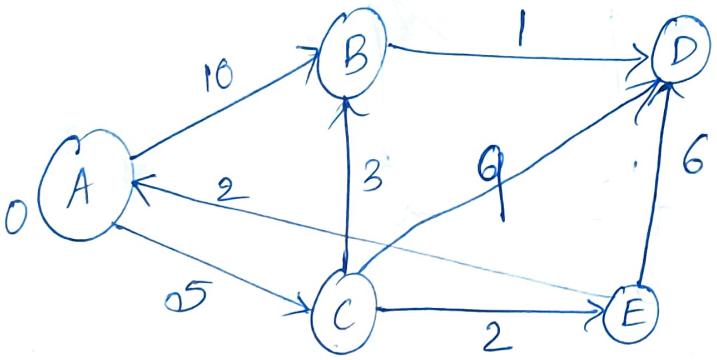
Cost of (1,2) = 5

" " (1,3) = 8

" " (1,4) = 11

" " (1,5) = 10

DIJKSTRA'S ALGORITM
(directed Graph)



* Source Vertex → A.

1) From A we can visit B and C.

$d(A,B) = 10$, $d(A,C) = 5$.

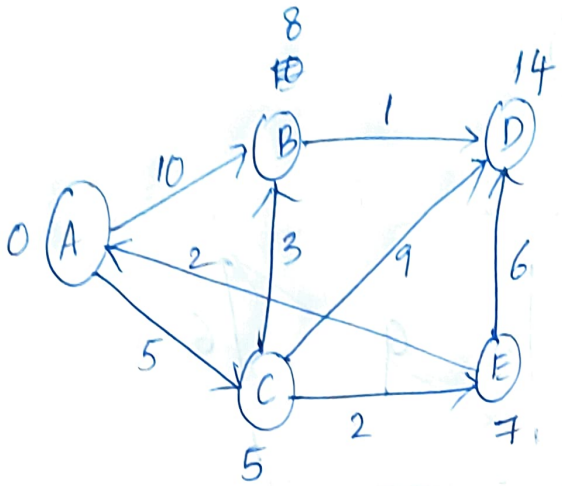
2) We move to C.

3) From 'C' we can visit B, D, E.

$$d(C,B) = \min\left((d(C)+d(C,B)), d(B)\right)$$
$$= \min\left((5+3), 10\right)$$
$$= 8$$

$$d(C,D) = \min\left[(d(C)+d(C,D)), d(D)\right]$$
$$= \min\left[(5+9), \infty\right]$$
$$= 14$$

$$d(C,E) = \min\left[(d(C)+d(C,E)), d(E)\right]$$
$$= \min\left[(5+2), \infty\right]$$
$$= 7$$

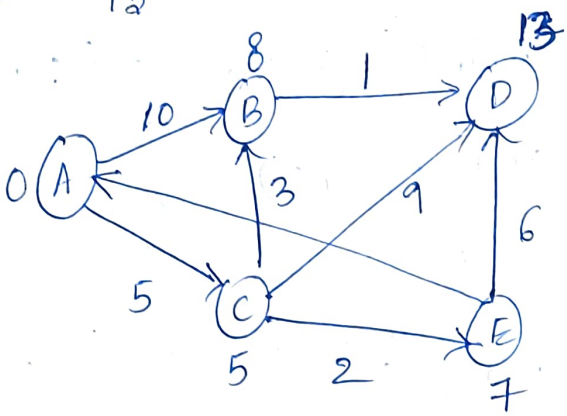


We choose the next shortest value. We move to E. From 'E' we can visit 'A' and 'D'. But 'A' is already visited. We only consider 'D'.

$$d(E,D) = \min\left[\left(d(E)+d(E,D)\right), d(D)\right]$$

$$= \min\left[(7+6), 14\right]$$

$$= 13$$

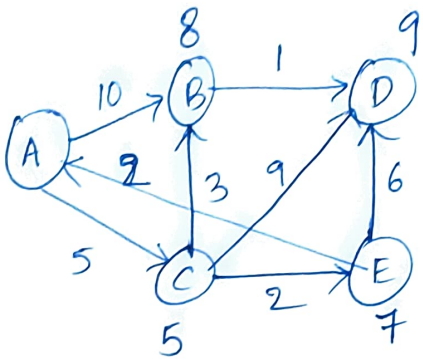


The next shortest value out of 'B' and 'D' is 'B'.

$$d(B,D) = \min\left[\left(d(B)+d(B,D)\right), d(D)\right]$$

$$= \min\left[(8+1), 13\right]$$

$$= 9$$



Final Costs:

From (A, B) = 8

" (A, C) = 5

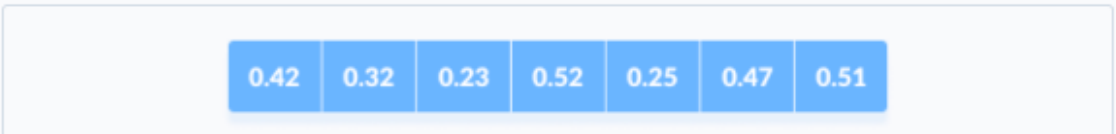
" (A, D) = 9

" (A, E) = 7

Bucket Sort:

- Bucket Sort is a sorting technique that sorts the elements by first dividing the elements into several groups called buckets.
- The process of bucket sort can be understood as **a scatter-gather** approach. The elements are first scattered into buckets then the elements of buckets are sorted. Finally, the elements are gathered in order.

1. Suppose, the input array is:



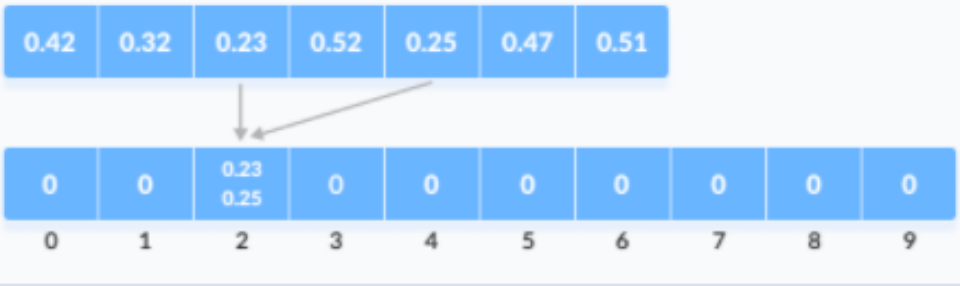
Input array

Create an array of size 10. Each slot of this array is used as a bucket for storing elements.

Array in which each position is a bucket

2. Insert elements into the buckets from the array. The elements are inserted according to the range of the bucket.

In our example code, we have buckets each of ranges from 0 to 1, 1 to 2, 2 to 3,..... (n-1) to n. Suppose, an input element is `.23` is taken. It is multiplied by `size = 10` (ie. `.23*10=2.3`). Then, it is converted into an integer (ie. `2.3=2`). Finally, .23 is inserted into **bucket-2**.

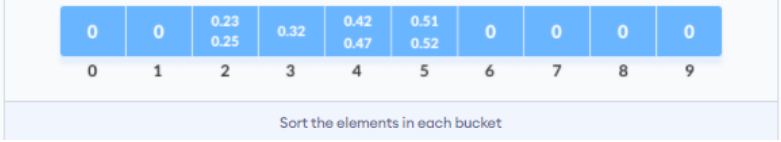


Insert elements into the buckets from the array

Similarly, other elements are inserted into their respective buckets.

Insert all the elements into the buckets from the array

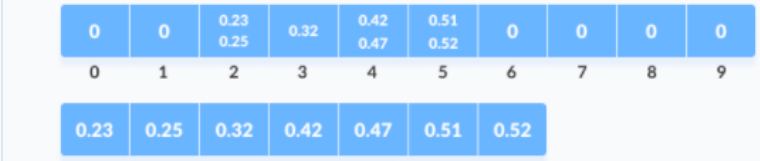
3. The elements of each bucket are sorted using any of the stable sorting algorithms. Here, we have used quicksort (inbuilt function).



Sort the elements in each bucket

4. The elements from each bucket are gathered.

It is done by iterating through the bucket and inserting an individual element into the original array in each cycle. The element from the bucket is erased once it is copied into the original array.



Gather elements from each bucket

Algorithm:

bucketSort()

create N buckets each of which can hold a range of values

for all the buckets

initialize each bucket with 0 values

for all the buckets

put elements into buckets matching the range

for all the buckets

sort elements in each bucket

gather elements from each bucket

end bucketSort

Analysis:

- **Worst Case Complexity: O(n^2)**
  When there are elements of close range in the array, they are likely to be placed in the same bucket. This may result in some buckets having more number of elements than others.
  It makes the complexity depend on the sorting algorithm used to sort the elements of the bucket.
  The complexity becomes even worse when the elements are in reverse order. If insertion sort is used to sort elements of the bucket, then the time complexity becomes O(n^2).

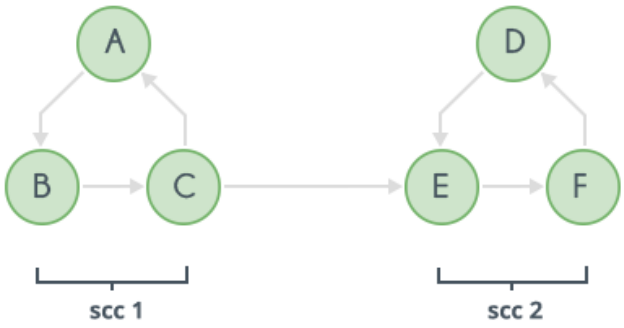
- **Best Case Complexity: O(n+k)**
  It occurs when the elements are uniformly distributed in the buckets with a nearly equal number of elements in each bucket.
  The complexity becomes even better if the elements inside the buckets are already sorted. If insertion sort is used to sort elements of a bucket then the overall complexity in the best case will be linear ie. O(n+k). O(n) is the complexity for making the buckets and O(k) is the complexity for sorting the elements of the bucket using algorithms having linear time complexity at the best case.

- **Average Case Complexity: O(n)**
  It occurs when the elements are distributed randomly in the array. Even if the elements are not distributed uniformly, bucket sort runs in linear time. It holds true until the sum of the squares of the bucket sizes is linear in the total number of elements.

Strongly Connected Graphs:

Connectivity in an undirected graph means that every vertex can reach every other vertex via any path. If the graph is not connected the graph can be broken down into **Connected Components**.

Strong Connectivity applies only to directed graphs. A directed graph is strongly connected if there is a **directed path** from any vertex to every other vertex. This is same as connectivity in an undirected graph, the only difference being strong connectivity applies to directed graphs and there should be directed paths instead of just paths. Similar to connected components, a directed graph can be broken down into **Strongly Connected Components**.



Kosaraju's Linear time algorithm to find Strongly Connected Components:

Algorithm:

1. Do a DFS on the original graph, keeping track of the finish times of each node. This can be done with a **stack**

```
stack STACK
void DFS(int source)
{
    visited[s]=true
    for all neighbours X of source that are not visited:
       DFS(X)
    STACK.push(source)
}
```

2. **Reverse the original graph**, it can be done efficiently if data structure used to store the graph is an adjacency list.

```
CLEAR ADJACENCY_LIST
 for all edges e:
    first = one end point of e
    second = other end point of e
    ADJACENCY_LIST[second].push(first)
```

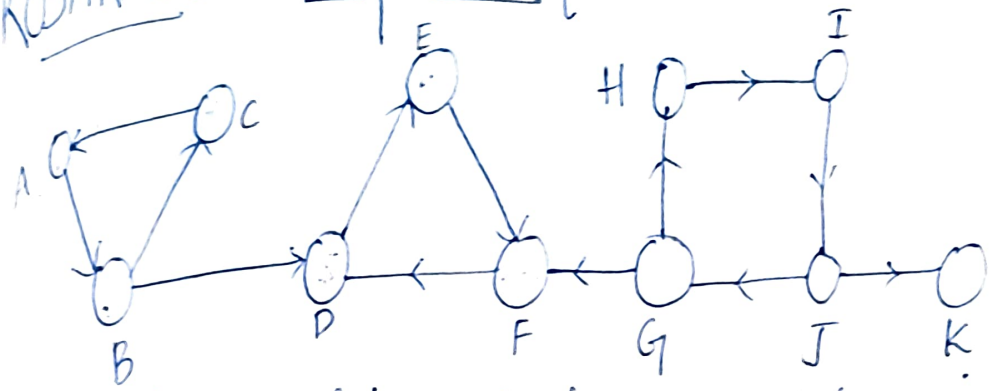
3. Do DFS on the reversed graph, with the source vertex as the vertex on top of the stack. When DFS finishes, all nodes visited will form one Strongly Connected Component. If any more nodes remain unvisited, this means there are more Strongly Connected Component's, so pop vertices from top of the stack until a valid unvisited node is found. This will have the highest finishing time of all currently unvisited nodes. This step is repeated until all nodes are visited.

```
while STACK is not empty:
   source = STACK.top()
   STACK.pop()
   if source is visited :
      continue
   else :
      DFS(source)
```

Analysis:

The above algorithm calls DFS, finds reverse of the graph and again calls DFS. DFS takes O(V+E) for a graph represented using adjacency list. Reversing a graph also takes O(V+E) time. For reversing the graph, we simple traverse all adjacency list.

KOSARAJU ALGORITHM:

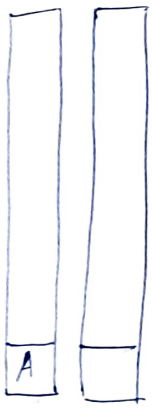


* This algorithm is used to compute strongly connected components.
* It is a DFS based algorithm and hence used a stack.
* This algorithm performs 2 passes over the given graph.
* In the 1st pass, we construct DFS for given graph.
* In 2nd pass, we reverse the graph, pop out the elements of given graph's DFS and reconstruct DFS for reversed graph.

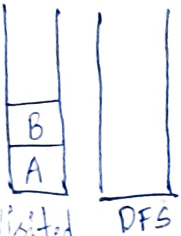
PASS 1: CONSTRUCTION OF STACK (DFS TRAVERSAL)

Two stacks are maintained. One to keep track of the visited nodes and another to construct DFS traversal.

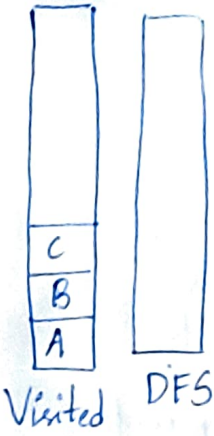
1. We start with vertex A. 'A' is visited and stored in "visited" stack.



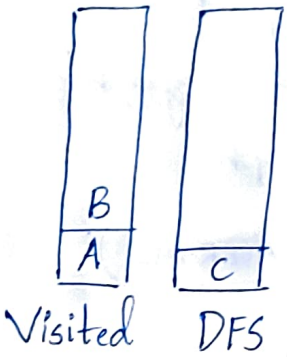
2. We move to B. 'B' is visited. Store in "visited" stack.



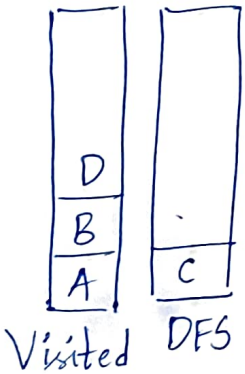
3. We move to 'C'. 'C' is visited. Store 'C' in visited stack.



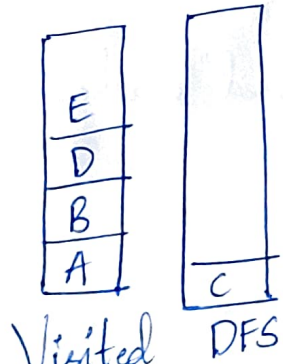
4. From 'C', no unvisited adjacent vertices are present. We pop 'C', store it in 'DFS' stack and backtrack to 'B'.



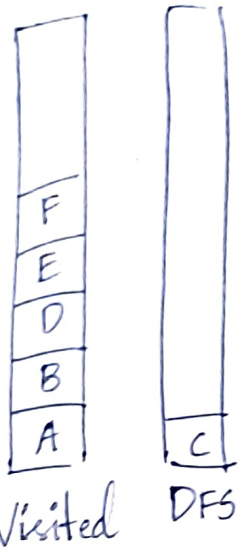
5. From 'B', we have unvisited adjacent vertex 'D'. We move to 'D'. Store it in visited stack.



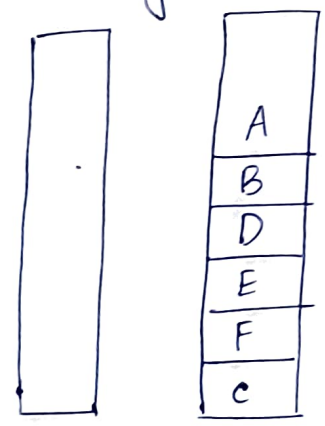
6. From 'D', we have 'E' as adjacent node. We move to 'E'. Store it in ~~adjacent~~ visited stack.



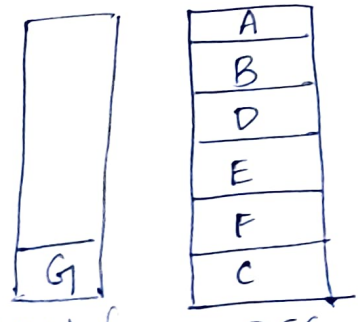
7. From E, we have F as adjacent vertex. We store 'F' in visited stack



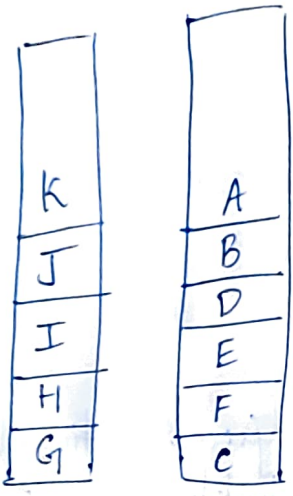
8. From 'F' we have no non-visited adjacent vertices. We pop it and store in DFS stack. We backtrack to 'E'. From 'E' we have no non-visited adjacent vertices and so we pop it & store in DFS stack. From 'D' we have no non-visited adjacent vertices. 'D' is popped and stored in DFS stack Same goes with vertices 'A' and 'B' respectively.



9. Now we move to 'G'. Store it in visited stack.

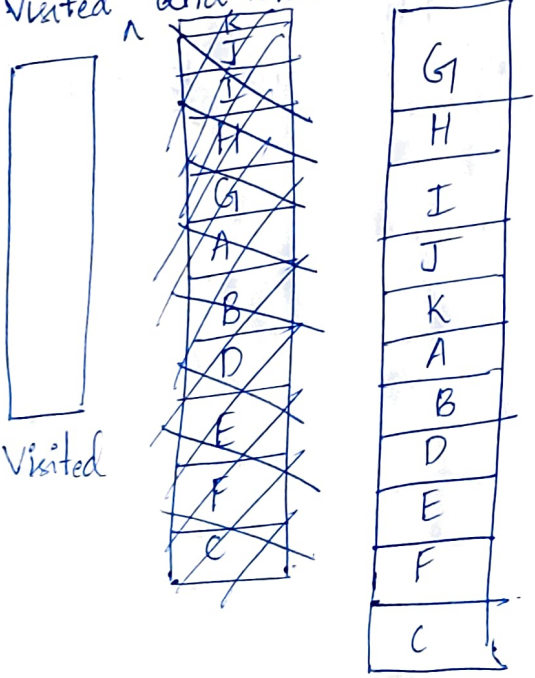


10. We move to 'H'. Store it in visited. From 'H' we move to 'I'. " " " " " " 'I' we move to 'J'. " " " " " " 'J' " " " 'K'. " " " " .



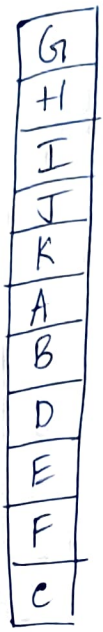
Visited. DFS

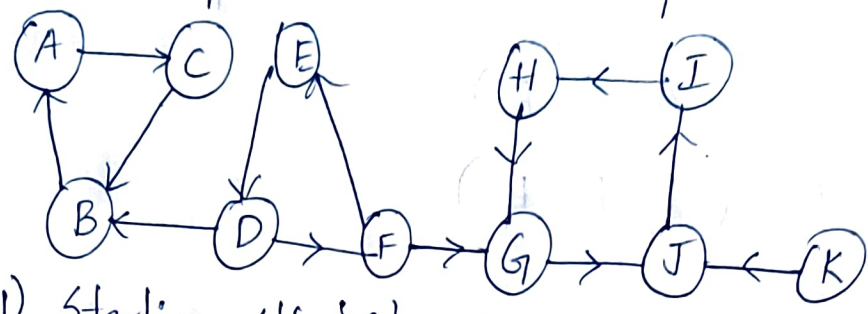
11. All vertices are visited. We pop the elements from visited stack and store it in 'DFS' stack.



DFS

PASS 2 : REVERSING GRAPH AND FINDING IT'S DFS



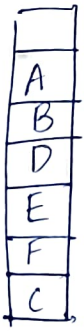


1) Starting with 'G'. We can move from G to J. From J → I. From I → H and from H to G. It forms a cycle. This cycle is a strongly connected component (SCC).

⇒ GHIJ is SCC 1 and the vertices are popped from stack.

2) Moving to 'K'. We can move from K to J but J is already visited. So single node K is a SCC.

⇒ K is SCC2 and it is popped from stack.



3) We move to 'A'. From A we go to C and from C → B. This is a cycle and SCC3.

⇒ ABC is SCC3 and the vertices A & B are popped from stack.

4) We move to 'D'. From D → F and F → E & E → D forms a cycle. This is SCC4.

⇒ DEF is SCC4 and the vertices D, E & F are popped.

5) 'C' is already visited. So we pop it out.

We get 4 strongly connected components:

1. A B C (A C B)
2. D E F (D F E)
3. K
4. G J I H

Topological sort:

Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge u v, vertex u comes before v in the ordering. Topological Sorting for a graph is not possible if the graph is not a DAG.

Algorithm:

In order to perform topological sort for a given graph, we need to find a vertex with in-degree = 0. If no such vertex exists, then the graph is cyclic and we cannot perform the topological sorting on it.

For a graph that has vertices with in-degree = 0, the algorithm using DFS is as follows:

Begin

initially mark all nodes as unvisited

for all nodes v of the graph, do

if v is not visited, then

topoSort (i, visited, stack)

done

pop and print all elements from the stack

End.

Analysis:

Time and Space Complexity

- **Time Complexity:** O (V+E).
  The above algorithm is simply DFS with an extra stack. So time complexity is the same as DFS which is.
- **Auxiliary space:** O (V).
  The extra space is needed for the stack.

Applications:

Few important applications of topological sort are-

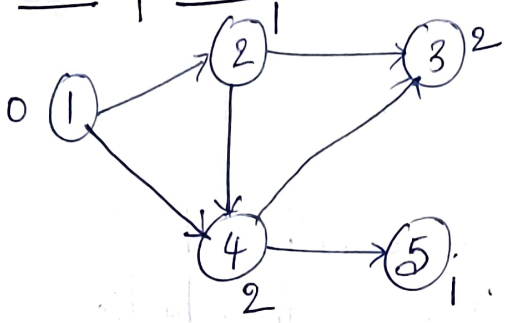
- Scheduling jobs from the given dependencies among jobs
- Instruction Scheduling
- Determining the order of compilation tasks to perform in make files
- Data Serialization

TOPOLOGICAL SORT: (USING DFS)

NOTE:

Topological sort is only possible if graph is a DAG.

GIVEN GRAPH:



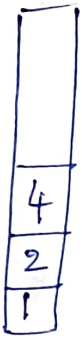
In-degree of 1 = 0
" " " 2 = 1
" " " 3 = 2
" " " 4 = 2
" " " 5 = 1

Step 1: Find a vertex with in-degree = 0.
Vertex 1 has in-degree '0'. So we start with vertex 1.

Step 2: Vertex 1 is stored in stack

Step 3: From 1, we can move either to 4 or 2.
We observe that there is an edge running from 2 to 4. ⇒ 2 appears before 4.
So we move to 2 and store it in stack.

Step 3: Now we move to 4 and store it in stack as there is an edge from 2 to 4.



Step 4: From '4' we can move either to 3 or 5. We move to 3. Store 3 in stack.

Step 5: From '3', No adjacent vertex. We backtr-ack to '4'. From '4' we move to '5'. Store 5 in stack.

Step 6: All the vertices are visited. Now, we pop the elements from the stack and print it in reverse order which will give us the topological sort order.

popped order → 5 → 3 → 4 → 2 → 1

Topological Order → 1 → 2 → 4 → 3 → 5

(or)

1 → 2 → 4 → 5 → 3.

Proof of correctness:

Mathematical Induction:

The simplest and most common form of mathematical induction infers that a statement involving a natural number n holds for all values of n. The proof consists of two steps:

1. The basis (base case): prove that the statement holds for the first natural number n. Usually, n = 0 or n = 1.

2. The inductive step: prove that, if the statement holds for some natural number n, then the statement holds for n + 1.

The hypothesis in the inductive step that the statement holds for some n is called the induction hypothesis (or inductive hypothesis).

Let S_n be a statement concerning the positive integer n. Suppose that:

1. S1 is true;
2. for any positive integer k, $k \leq n$, if S_k is true, then S_{k+1} is also true.

Then S_n is true for every positive integer value of n.

Example 1 - Proving An Equality Statement

Let S_n represent the statement

$$1+2+3+\cdots+n=\frac{n(n+1)}{2}.$$

Prove that S_n is true for every positive integer n.

Solution

Step 1 Show that the statement is true when $n = 1$. If $n = 1$, S_1 becomes

$$1=\frac{1(1+1)}{2}, \quad \text{which is true.}$$

Step 2 Show that S_k implies S_{k+1}, where S_k is the statement

$$1+2+3+\cdots+k=\frac{k(k+1)}{2},$$

and S_{k+1} is the statement

$$1+2+3+\cdots+k+(k+1)=\frac{(k+1)[(k+1)+1]}{2}.$$

Step 2 Start with S_k and assume it is a true statement.

$$1+2+3+\cdots+k=\frac{k(k+1)}{2},$$

Add $k + 1$ to both sides of this equation to obtain S_{k+1}.

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

Step 2

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

$$=(k+1)\left(\frac{k}{2}+1\right)$$ Factor out $k + 1$.

$$=(k+1)\left(\frac{k+2}{2}\right)$$ Add inside the parentheses.

$$=\frac{(k+1)[(k+1)+1]}{2}$$ Multiply; $k + 2 = (k + 1) + 1$.

This final result is the statement for n = k + 1; it has been shown that if S_k is true, then S_{k+1} is also true.

The two steps required for a proof by mathematical induction have been completed, so the statement S_n is true for every positive integer value of n.

Loop invariant method:

We use loop invariants to help us understand why an algorithm is correct. We must show three things about a loop invariant:

1. Initialization: It is true prior to the first iteration of the loop.
2. Maintenance: If it is true before an iteration of the loop, it remains true before the next iteration.
3. Termination: When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

When the first two properties hold, the loop invariant is true prior to every iteration of the loop.

Considering it's similarity to mathematical induction, where to prove that a property holds, we prove a base case and an inductive step. Here, showing that the invariant holds before the first iteration is like the base case, and showing that the invariant holds from iteration to iteration is like the inductive step.

HeapSort loop invariant:

Build – Max – Heap (A)

1. heap-size[A] ← length[A]
2. for i ←⌊ length[A]/2⌋ downto 1

3. do Max-Heapify(A, i)

To show why Build-Max-Heap works correctly, we use the following loop invariant:

At the start of each iteration of the for loop of lines 2– 3, each node i + 1, i + 2, . . . , n is the root of a max-heap.

Initialization: Prior to the first iteration of the loop, i = ⌊ n/2⌋ . Each node ⌊ n/2⌋ +1, ⌊ n/2⌋ +2, . . . , n is a leaf and is thus the root of a trivial max-heap.

Maintenance: To see that each iteration maintains the loop invariant, observe that the children of node i are numbered higher than i. By the loop invariant, therefore, they are both roots of max-heaps. This is precisely the condition required for the call Max-Heapify(A, i) to make node i a max-heap root. Moreover, the Max-Heapify call preserves the property that nodes i + 1, i + 2, . . . , n are all roots of max-heaps. Decrementing i in the for loop update reestablishes the loop invariant for the next iteration.

Termination: At termination, i = 0. By the loop invariant, each node 1, 2, . . . , n is the root of a max-heap. In particular, node 1 is.