Sunbeam Institute of Information Technology
Pune and Karad

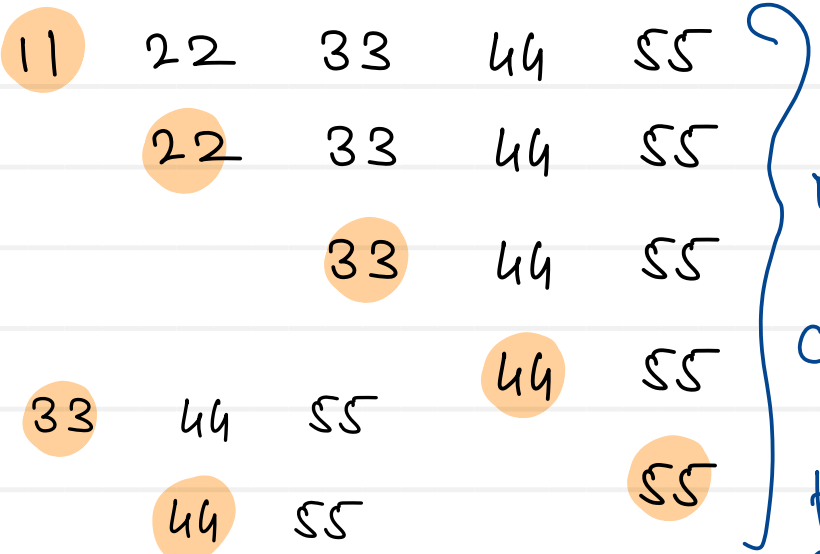
Algorithms and Data structures

Trainer - Devendra Dhande

Email – devendra.dhande@sunbeaminfo.com

Quick sort

1. Select pivot/axis/reference element from array
2. Arrange lesser elements on left side of pivot
3. Arrange greater elements on right side of pivot
4. Sort left and right side of pivot again (by quick sort)



No. of levels ≈ log n

No. of comps per level ≈ n

Total comps ≈ n log n

Time ∝ comps

Time ∝ n log n

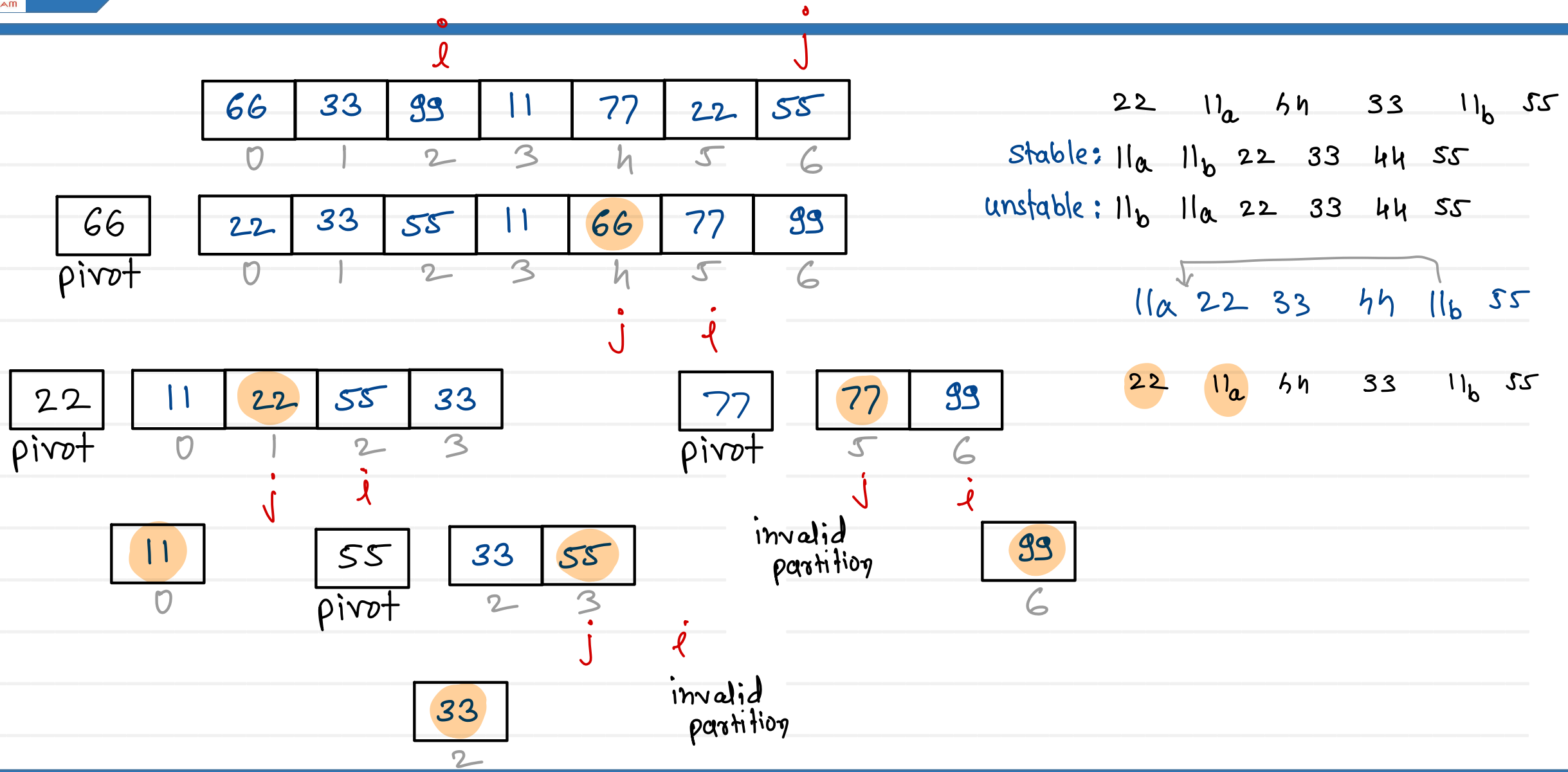
Best Avg $T(n) = O(n \log n)$

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

$S(n) = O(1)$

- time complexity of quick sort is dependent on selection of pivot.
- selection of pivot is done by any one out of below
  - i extreme left
  - ii extreme right
  - iii middle
  - iv. median of three
  - v. median of five
  - vi. dual pivot.

Quick sort



22 11_a 44 33 11_b 55

Stable: 11_a 11_b 22 33 44 55

unstable : 11_b 11_a 22 33 44 55

11_a 22 33 44 11_b 55

22 11_a 44 33 11_b 55

	space	Time Best	Time Avg	Time Worst
selection sort	O(1)	$O(n^2)$	$O(n^2)$	$O(n^2)$
bubble sort	O(1)	$O(n)$	$O(n^2)$	$O(n^2)$
insertion sort	O(1)	$O(n)$	$O(n^2)$	$O(n^2)$
Heap sort	O(1)	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quick sort	O(1)	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
Merge sort	O(n)	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

in place sorting algorithm

Sliding window Technique

- involve moving a fixed or variable-size window through a data structure, to solve problems efficiently.
- This technique is used to find subarrays or substrings according to a given set of conditions.
- This method used to efficiently solve problems that involve defining a **window** or **range** in the input data and then moving that window across the data to perform some operation within the window.
- This technique is commonly used in algorithms like
  - **finding subarrays** with a specific sum
  - **finding the longest substring** with unique characters
  - solving problems that require a fixed-size window to process elements efficiently.

- There are two types of sliding window
  - **Fixed size sliding window**
    - Find the size of the window required
    - Compute the result for 1st window
    - Then use a loop to slide the window by 1 and keep computing the result
  - **Variable size sliding window**
    - increase right pointer one by one till our condition is true.
    - At any step if condition does not match, shrink the size of window by increasing left pointer.
    - Again, when condition satisfies, start increasing the right pointer
    - follow these steps until reach to the end of the array

Maximum Average Subarray

You are given an integer array nums consisting of n elements, and an integer k.

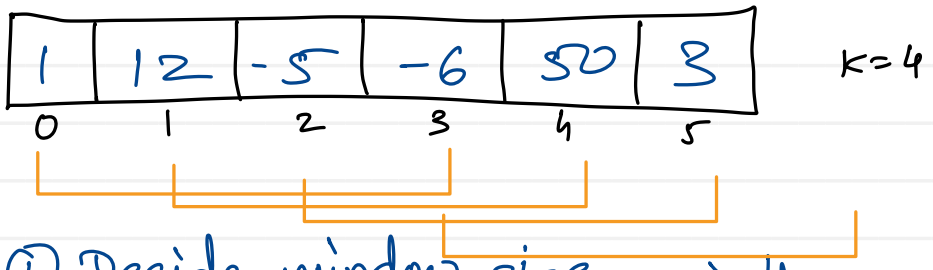
Find a contiguous subarray whose length is equal to k that has the maximum average value and return this value. Any answer with a calculation error less than 10-5 will be accepted.

Example 1:
Input: nums = [1,12,-5,-6,50,3], k = 4
Output: 12.75000
Explanation: Maximum average is (12 - 5 - 6 + 50) / 4 = 51 / 4 = 12.75

Example 2:
Input: nums = [5], k = 1
Output: 5.00000

maxSum 2 51

windowSum 2 51 42



① Decide window size => 4

② Compute result of first window

```
int windowSum = 0;
for(i=0; i<k; i++)
    windowSum += nums[i];
```

③ move window by one & recalculate result and update according to condition

```
int maxSum = windowSum;
for(i=k; i<arr.length; i++){
    windowSum = windowSum - nums[i-k] + nums[i];
    if(windowSum > maxSum)
        maxSum = windowSum);
}
```

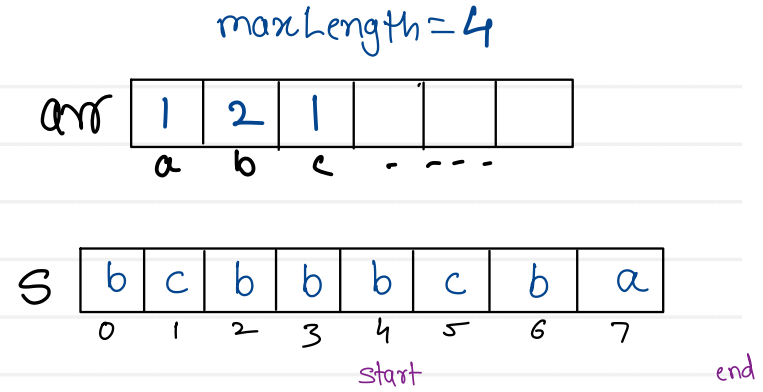
④ find avg = (double) maxSum / k;

Maximum Length Substring With Two Occurrences

Given a string s, return the maximum length of a substring such that it contains at most two occurrences of each character.

Example 1:
Input: s = "bcbbbcba"
Output: 4
Explanation: The following substring has a length of 4 and contains at most two occurrences of each character: "bcbbbcba".

Example 2:
Input: s = "aaaa"
Output: 2
Explanation: The following substring has a length of 2 and contains at most two occurrences of each character: "aaaa".




```
int maxLength = 0;
int start=0, end=0;
int[] arr = new int[26];
for( ; end < s.length() ; end++){
    arr[s.charAt(end) - 'a']++;
    while(arr[s.charAt(end) - 'a']==3){
        arr[s.charAt(start) - 'a']--;
        start++;
    }
    maxLength = Math.max(maxLength, end-start+1);
}
return maxLength;
}
```

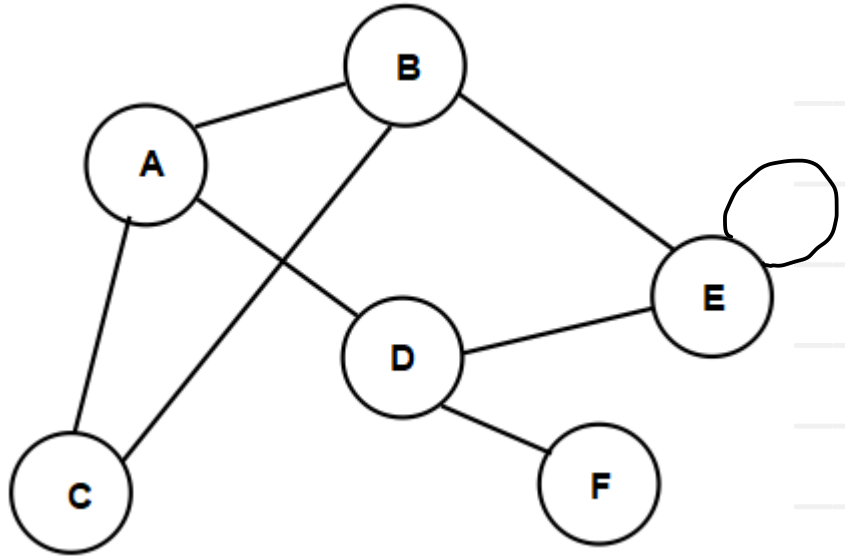
Graph : Terminologies

- **Graph** is a non linear data structure having set of vertices (nodes) and set of edges (arcs).
  - G = {V, E}

    Where V is a set of vertices and E is a set of edges
  - **Vertex (node)** is an element in the graph

    V = {A, B, C, D, E, F}
  - **Edge (arc)** is a line connecting two vertices

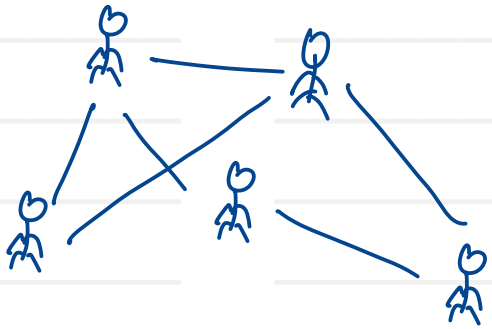
    E = {(A,B), (A,C), (B,C), (B,E), (D, E), (D,F),(A,D)}

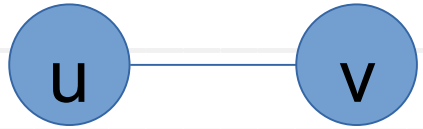


- Vertex A is set be **adjacent** to B, if and only if there is an edge from A to B.
- **Degree of vertex :-** Number of vertices adjacent to given vertex
- **Path :-** Set of edges connecting any two vertices is called as path between those two vertices.
  - Path between A to D = {(A, B), (B, E), (E, D)}
- **Cycle :-** Set of edges connecting to a node itself is called as cycle.
  - {(A, B), (B, E), (E, D), (D, A)}
- **Loop :-** An edge connecting a node to itself is called as loop. Loop is smallest cycle.

Graph : Types

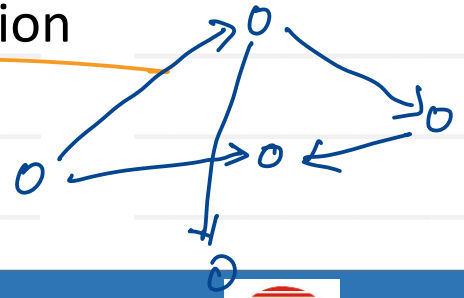
- **Undirected graph.**
  - If we can represent any edge either (u,v) OR (v,u) then it is referred as **unordered pair of vertices i.e. undirected edge.**
  - **graph which contains undirected edges referred as undirected graph.**

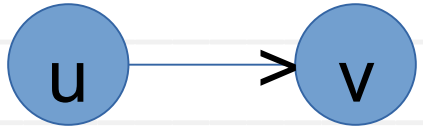




(u, v) == (v, u)

- **Directed Graph (Di-graph)**
  - If we cannot represent any edge either (u,v) OR (v,u) then it is referred as an **ordered pair of vertices** i.e. directed edge.
  - **graph which contains set of directed edges referred as directed graph (di-graph).**
  - graph in which each edge has some direction

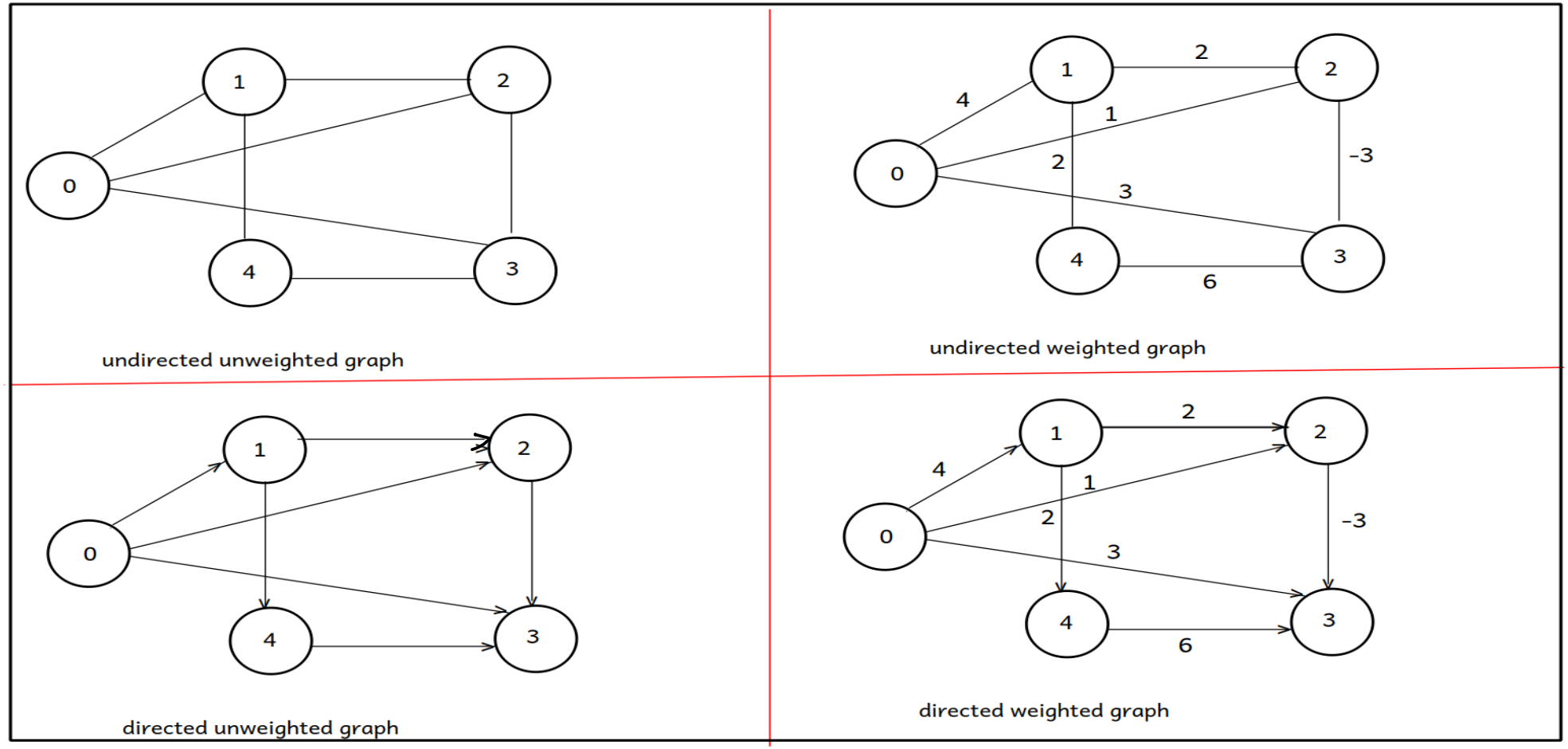




(u, v) != (v, u)

Graph : Types

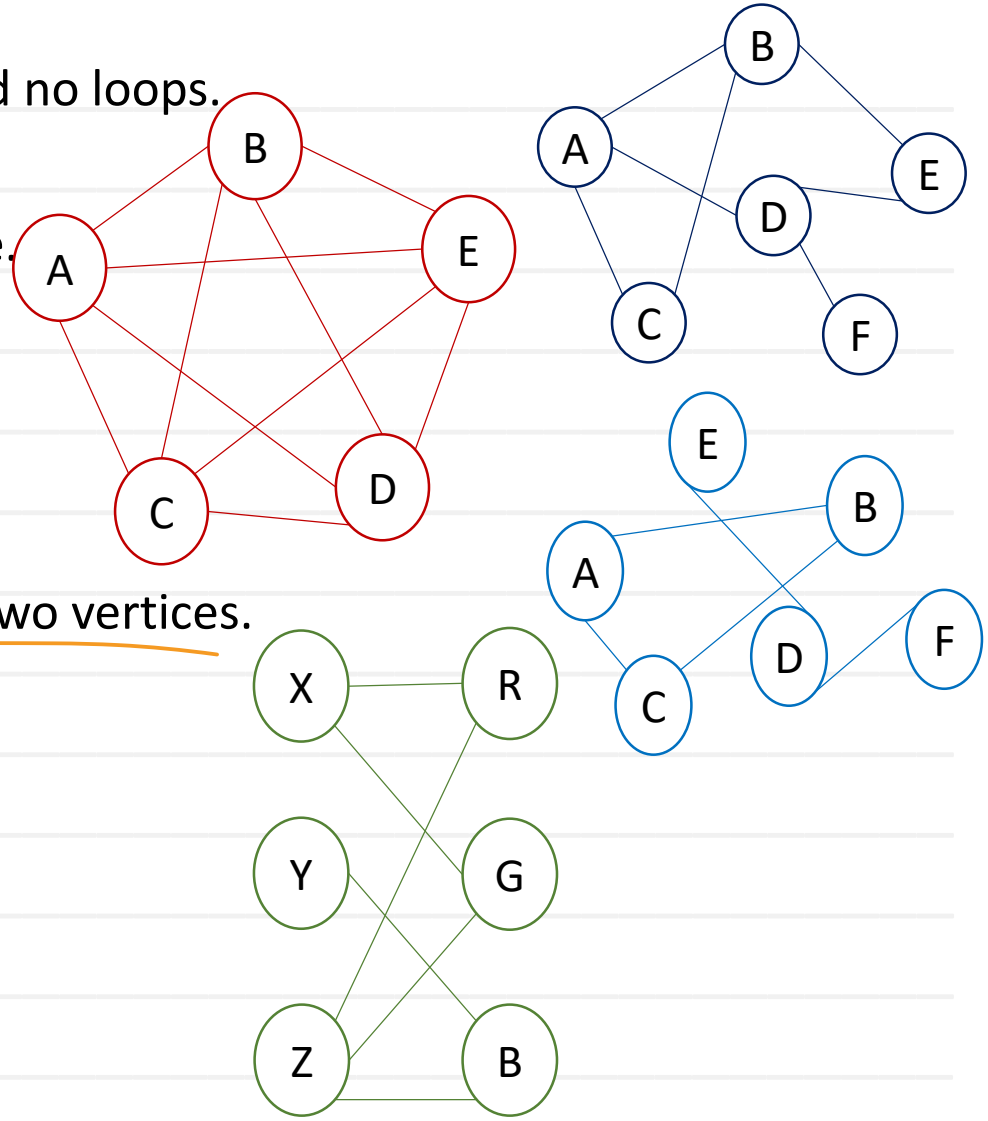
- **Weighted Graph**
  - A graph in which edge is associated with a number (ie weight)



Graph : Types

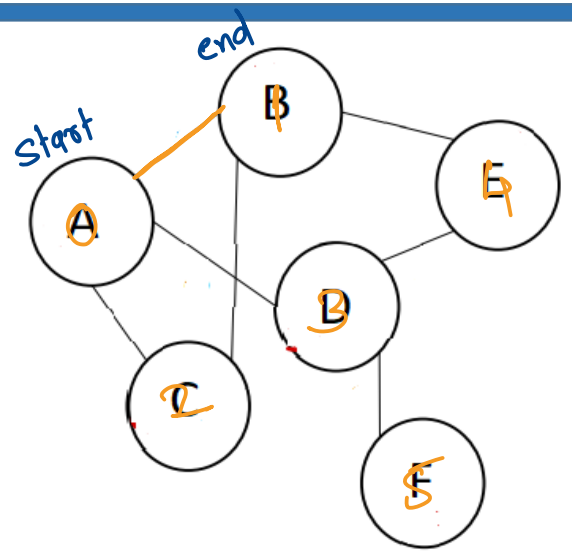
- **Simple Graph**
  - Graph not having multiple edges between adjacent nodes and no loops.
- **Complete Graph**
  - Simple graph in which node is adjacent with every other node.
  - Un-Directed graph: Number of Edges = n (n -1) / 2

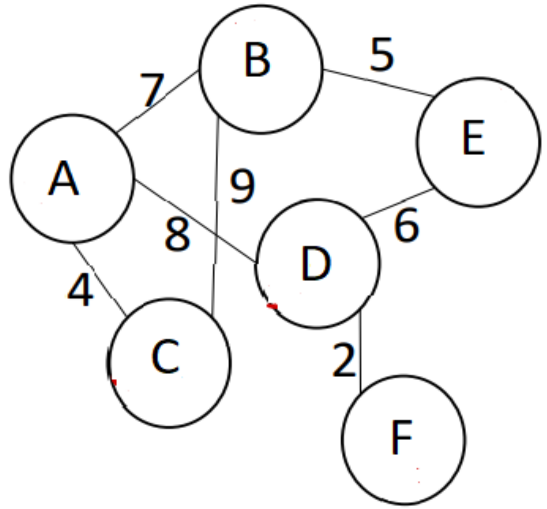
    where, n – number of vertices
  - Directed graph: Number of edges = n (n-1)
- **Connected Graph**
  - Simple graph in which there is some path exist between any two vertices.
  - Can traverse the entire graph starting from any vertex.
- **Bi-partite graph**
  - Vertices can be divided in two disjoint sets.
  - Vertices in first set are connected to vertices in second set.
  - Vertices in a set are not directly connected to each other.



Graph Implementation – Adjacency Matrix

- If graph have V vertices, a V x V matrix can be formed to store edges of the graph.
- Each matrix element represent presence or absence of the edge between vertices.
- For non-weighted graph, 1 indicate edge and 0 indicate no edge.
- For weighted graph, weight value indicate the edge and infinity sign ∞ represent no edge.
- For un-directed graph, adjacency matrix is always symmetric across the diagonal.
- Space complexity of this implementation is O(V^2).



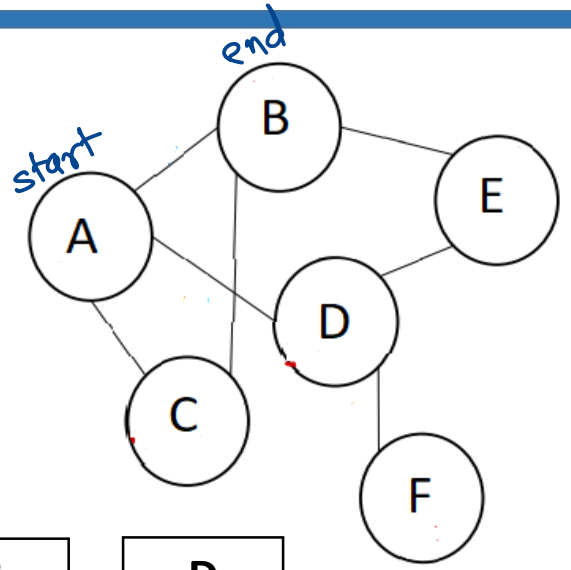


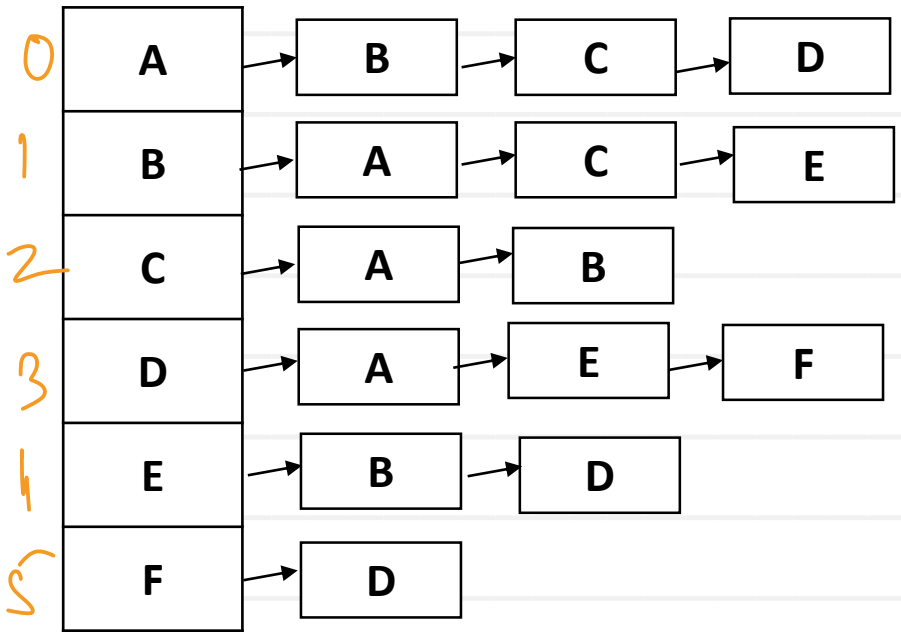
	A	B	C	D	E	F
A	0	1	1	1	0	0
B	1	0	1	0	1	0
C	1	1	0	0	0	0
D	1	0	0	0	1	1
E	0	1	0	1	0	0
F	0	0	0	1	0	0

	A	B	C	D	E	F
A	∞	7	4	8	∞	∞
B	7	∞	9	∞	5	∞
C	4	9	∞	∞	∞	∞
D	8	∞	∞	∞	6	2
E	∞	5	∞	6	∞	∞
F	∞	∞	∞	2	∞	∞

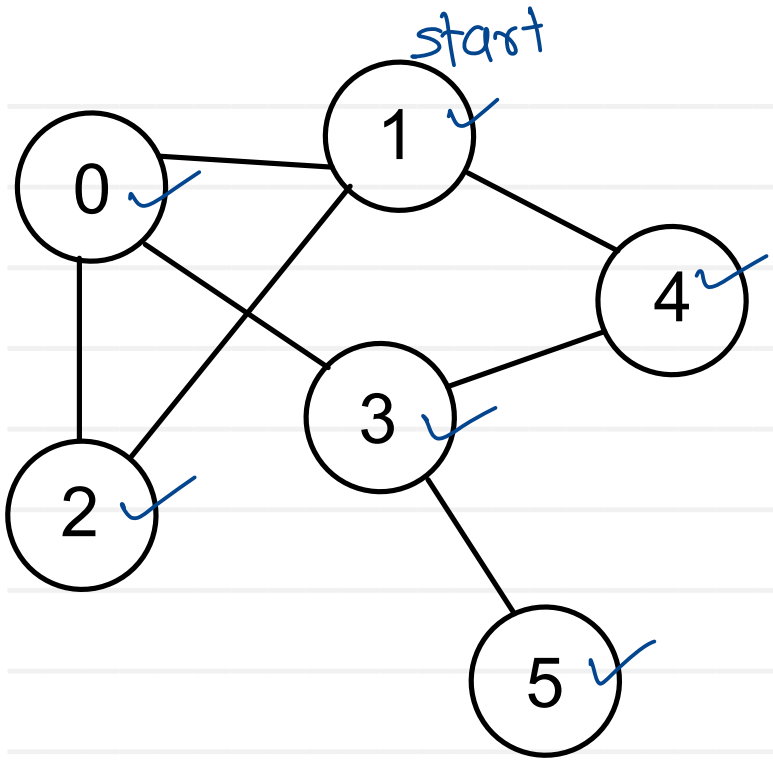
Graph Implementation – Adjacency List

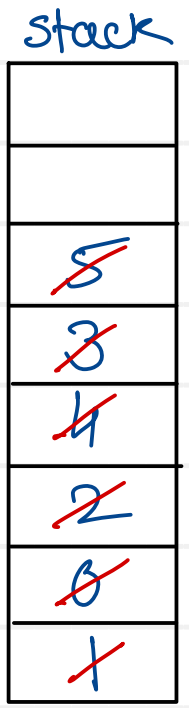
- Each vertex holds list of its adjacent vertices.
- For non-weighted graphs only, neighbor vertices are stored.
- For weighted graph, neighbor vertices and weights of connecting edges are stored.
- Space complexity of this implementation is O(V+E).
- If graph is sparse graph (with fewer number of edges), this implementation is more efficient (as compared to adjacency matrix method).





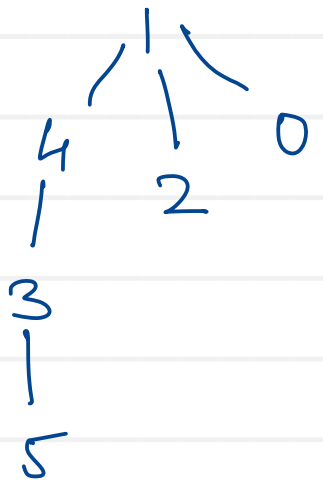
DFS Traversal



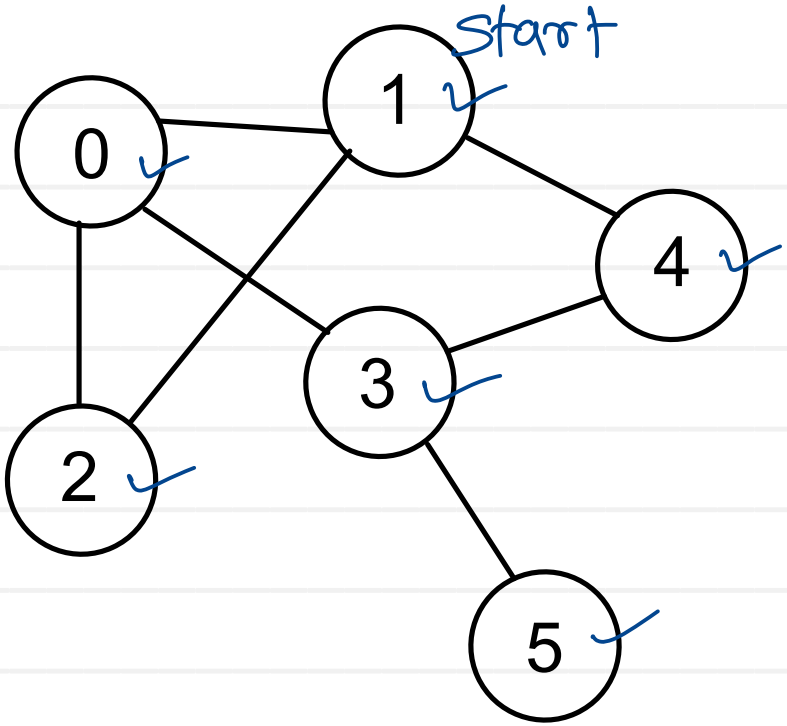


1. Choose a vertex as start vertex.
2. Push start vertex on stack & mark it.
3. Pop vertex from stack.
4. Print the vertex.
5. Put all non-visited neighbours of the vertex on the stack and mark them.
6. Repeat 3-5 until stack is empty.

Traversal : 1, 4, 3, 5, 2, 0



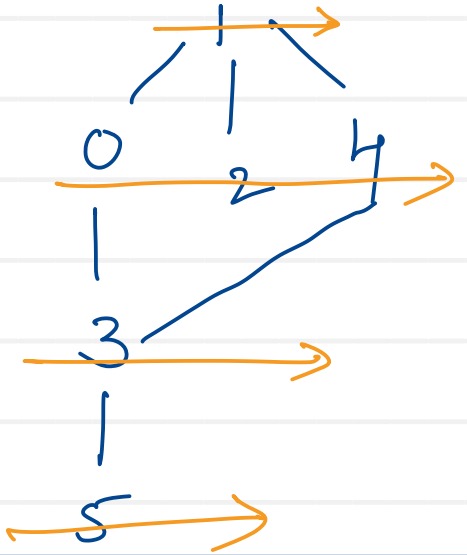
BFS Traversal



Queue | 1 | 0 | 2 | 4 | 3 | 5 | | |

Traversal : 1, 0, 2, 4, 3, 5

1. Choose a vertex as start vertex.
2. Push start vertex on queue & mark it
3. Pop vertex from queue.
4. Print the vertex.
5. Put all non-visited neighbours of the vertex on the queue and mark them.
6. Repeat 3-5 until queue is empty.



Thank you!!!

Devendra Dhande

devendra.dhande@sunbeaminfo.com