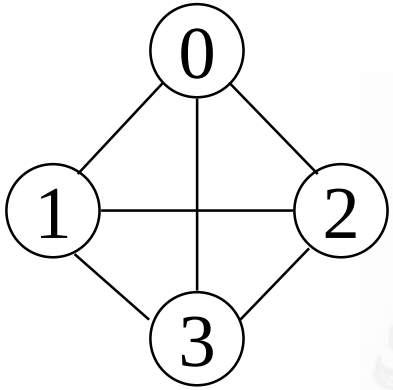
CHAPTERS: GRAPHS, SEARCHING AND SORTING

Definitions

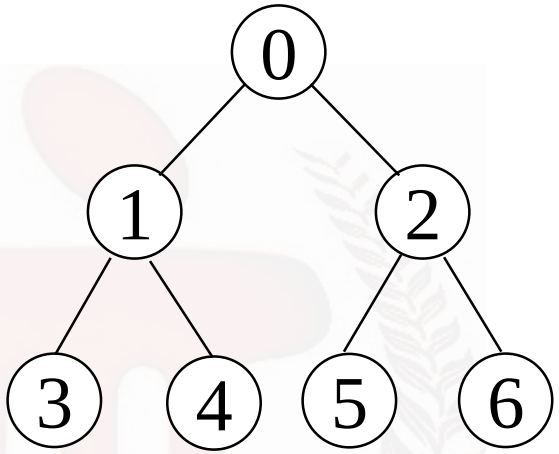
- A graph, $G=(V, E)$, consists of two sets:
  - a finite set of ***vertices(V)***, and
  - a finite, possibly empty set of edges***(E)***
  - *V(G)* and *E(G)* represent the sets of vertices and edges of *G*, respectively
- Undirected graph
  - The pairs of vertices representing any edges is ***unordered***
  - e.g., (v_0, v_1) and (v_1, v_0) represent the same edge (v0, v1) = (v1,v0)
- Directed graph
  - Each edge as a directed pair of vertices <v0, v1> != <v1,v0>
  - e.g. $<v_0, v_1>$ represents an edge, v_0 is the tail and v_1 is the head

Examples for Graph



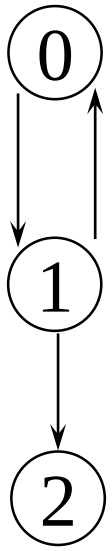
G_1

complete graph



G_2

incomplete graph



G_3

$V(G_1)=\{0,1,2,3\}$
$V(G_2)=\{0,1,2,3,4,5,6\}$
$V(G_3)=\{0,1,2\}$

$E(G_1)=\{(0,1),(0,2),(0,3),(1,2),(1,3),(2,3)\}$
$E(G_2)=\{(0,1),(0,2),(1,3),(1,4),(2,5),(2,6)\}$
$E(G_3)=\{<0,1>,<1,0>,<1,2>\}$

complete undirected graph: n(n-1)/2 edges
complete directed graph: n(n-1) edges

Complete Graph

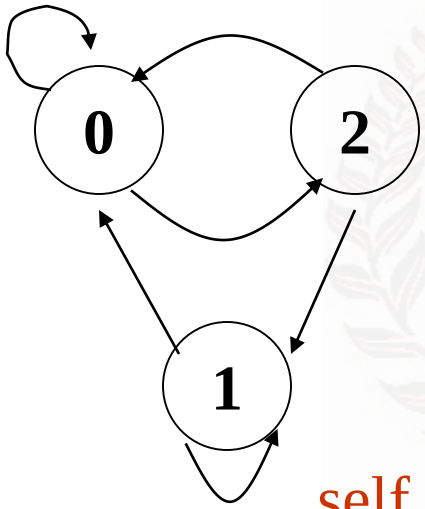
A complete graph is a graph that has the maximum number of edges

- for undirected graph with n vertices, the maximum number of edges is n(n-1)/2
- for directed graph with n vertices, the maximum number of edges is n(n-1)
- example: G1 (previous slide) is a complete graph

Adjacent and Incident

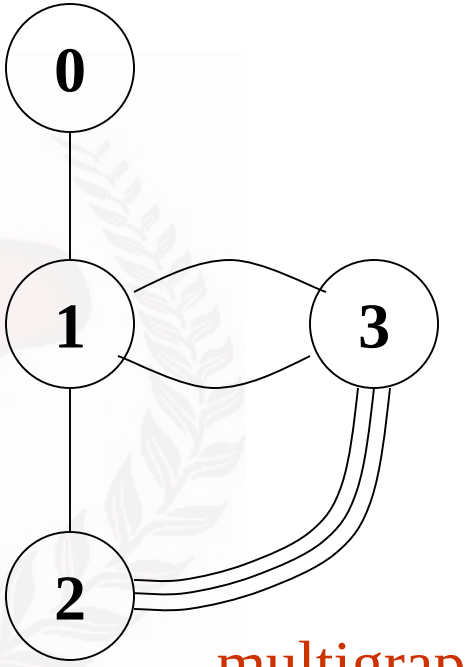
- If (v_0, v_1) is an edge in an undirected graph,
  - v_0 and v_1 are adjacent
  - The edge (v_0, v_1) is incident on vertices v_0 and v_1
- If $<v_0, v_1>$ is an edge in a directed graph
  - v_0 is adjacent to v_1, and v_1 is adjacent from v_0
  - The edge $<v_0, v_1>$ is incident on v_0 and v_1

***Figure 6.3:**Example of a graph with feedback loops and a multigraph (p.260)



(a)

self edge



(b)

multigraph:
multiple occurrences
of the same edge

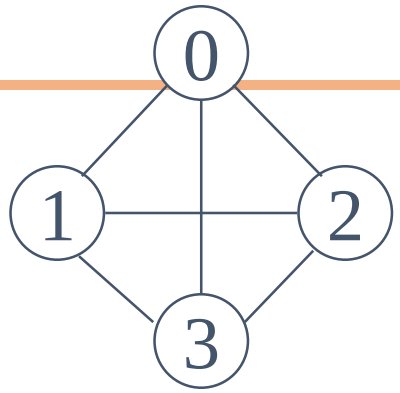
Subgraph and Path

- A subgraph of G is a graph G' such that V(G') is a subset of V(G) and E(G') is a subset of E(G)

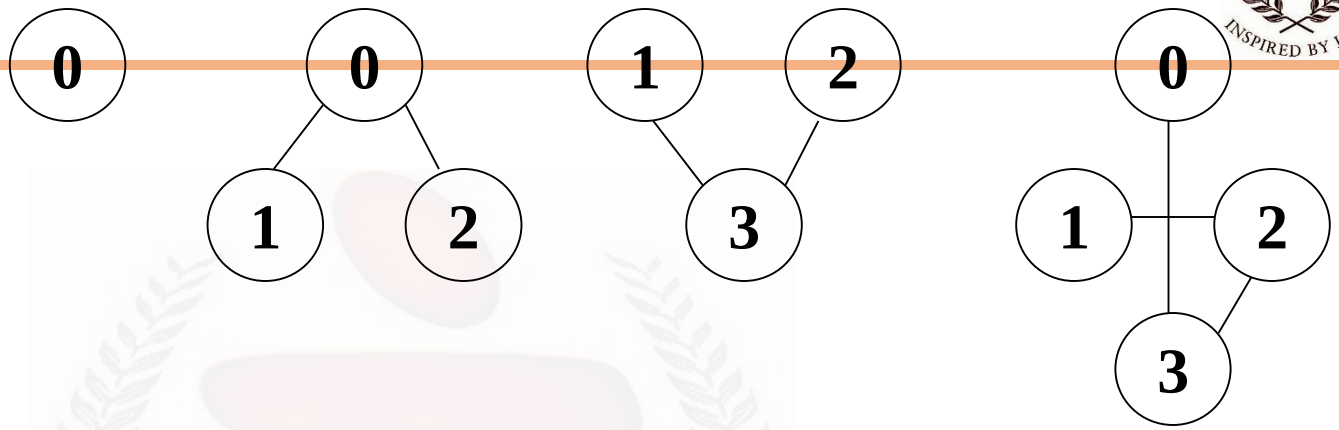
- A path from vertex v_p to vertex v_q in a graph G, is a sequence of vertices, v_p, v_{i1}, v_{i2}, ..., v_{in}, v_q, such that (v_p, v_{i1}), (v_{i1}, v_{i2}), ..., (v_{in}, v_q) are edges in an undirected graph

- The length of a path is the number of edges on it

Figure 6.4: subgraphs of G_1 and G_3 (p.261)



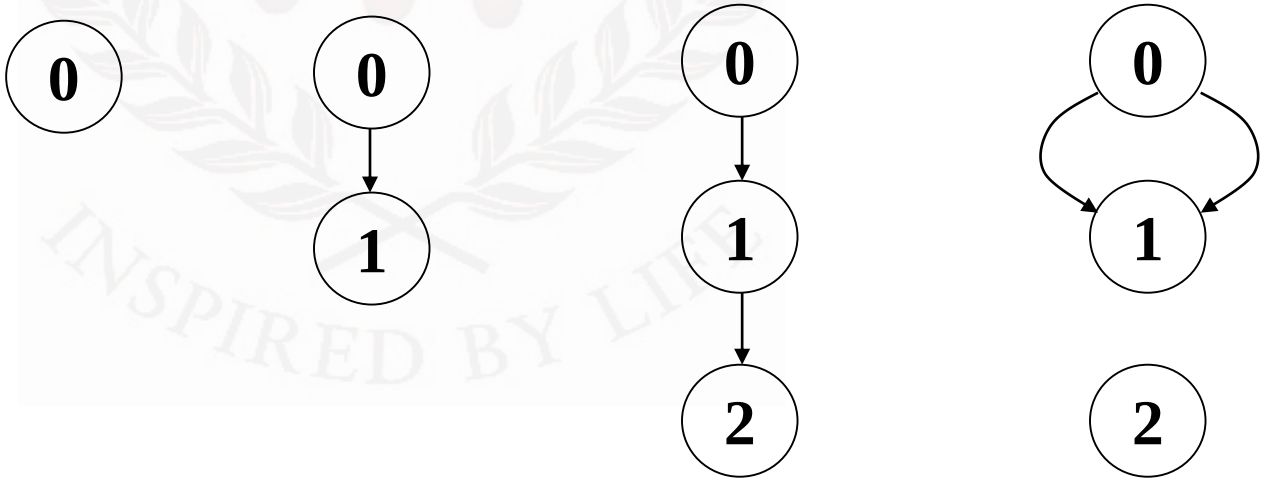
G_1



(i) (ii) (iii) (iv)

(a) Some of the subgraph of G_1

G_3



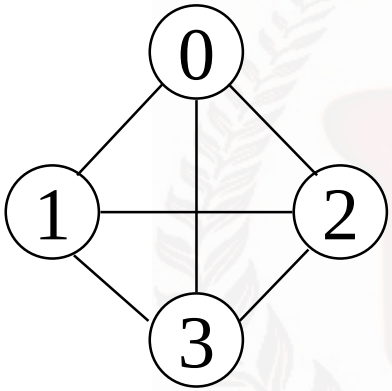
(i) (ii) (iii) (iv)

(b) Some of the subgraph of G_3

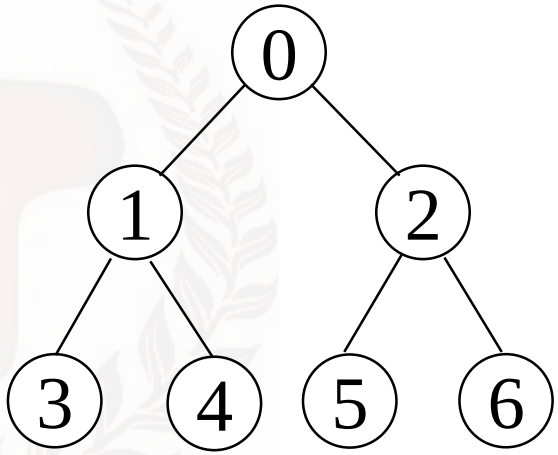
Simple Path and Style

- A simple path is a path in which all vertices, except possibly the first and the last, are distinct
- A cycle is a simple path in which the first and the last vertices are the same
- In an undirected graph G, ***two vertices***, v_0 and v_1, are connected if there is a path in G from v_0 to v_1
- An undirected ***graph*** is connected if, for every pair of distinct vertices v_i, vj, there is a path from vi to v_j

connected



G_1



G_2

tree (acyclic graph)

Degree

- The degree of a vertex is the number of edges incident to that vertex

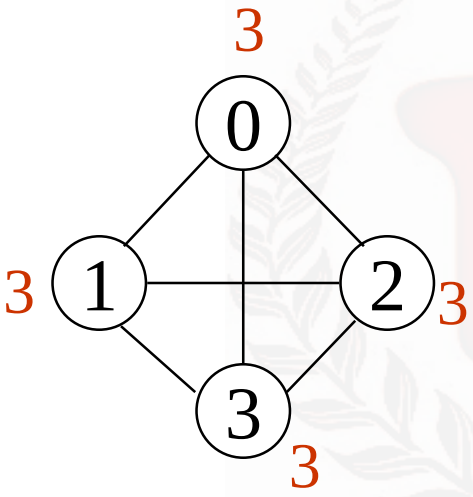
- For directed graph,
  - the in-degree of a vertex *v* is the number of edges that have *v* as the head
  - the out-degree of a vertex *v* is the number of edges that have *v* as the tail

  - if d_i is the degree of a vertex *i* in a graph *G* with *n* vertices and *e* edges, the number of edges is

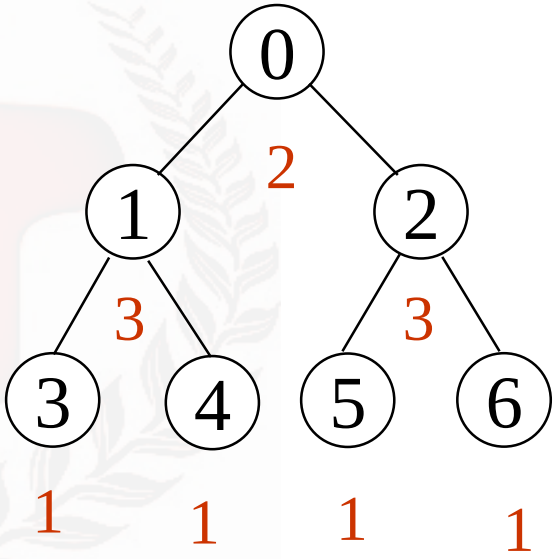
$$e = (\sum_{0}^{n-1} d_i) / 2$$

undirected graph

degree



G_1

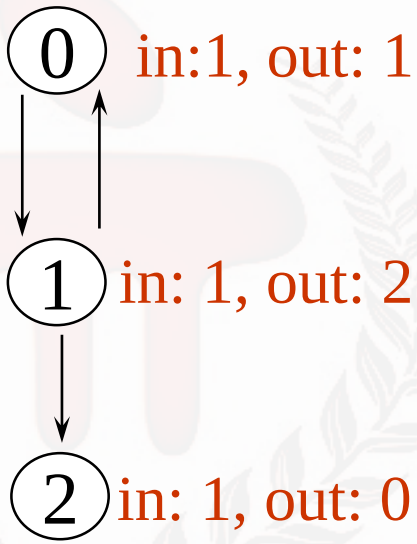


G_2

directed graph

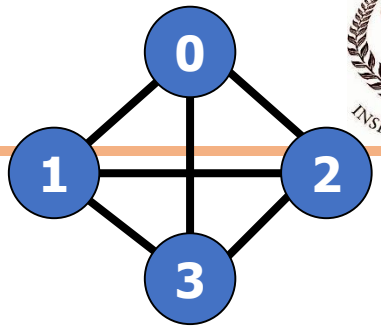
in-degree

out-degree

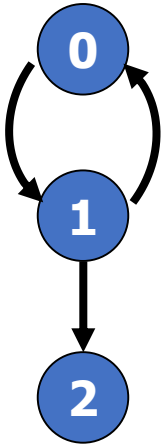


G_3

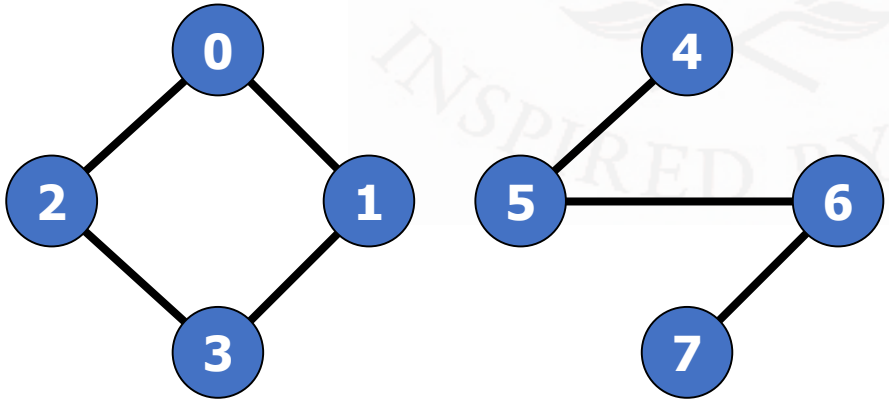
- Adjacency matrices
- Adjacency lists



G1

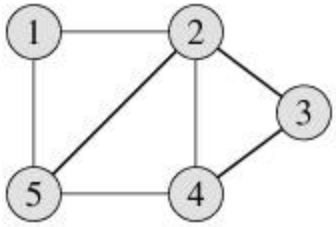


G3

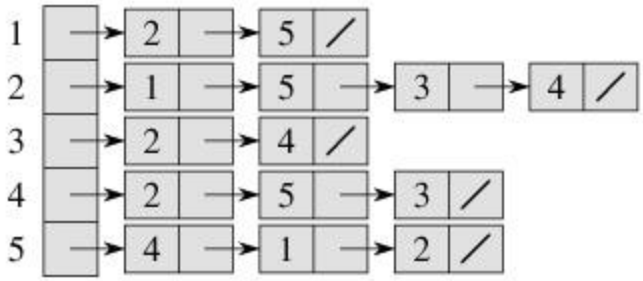


G4

Graph representation – undirected



(a)



(b)

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

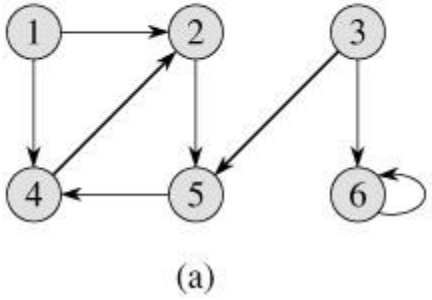
(c)

graph

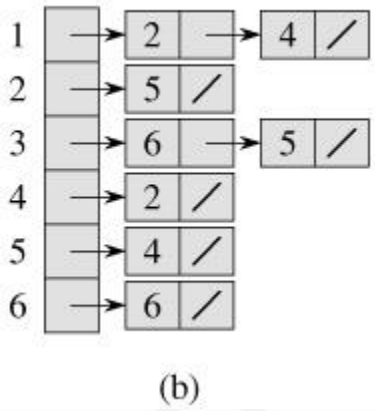
Adjacency list

Adjacency matrix

Graph representation – directed



graph



Adjacency list

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

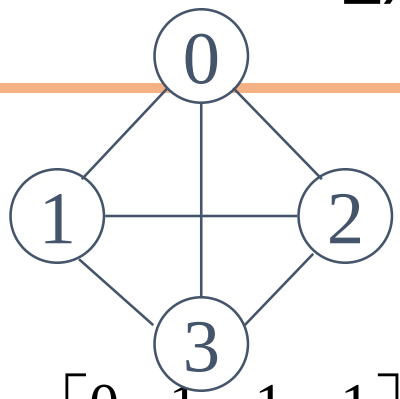
(c)

Adjacency matrix

Adjacency Matrix

- Let G=(V,E) be a graph with n vertices.
- The adjacency matrix of G is a two-dimensional n by n array, say adj_mat
- If the edge (vi, vj) is in E(G), adj_mat[i][j]=1
- If there is no such edge in E(G), adj_mat[i][j]=0
- The adjacency matrix for an undirected graph is symmetric; the adjacency matrix for a digraph need not be symmetric

Examples for Adjacency Matrix



$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

G_1

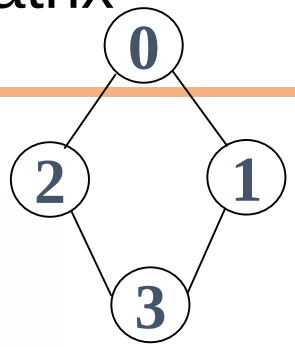
$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

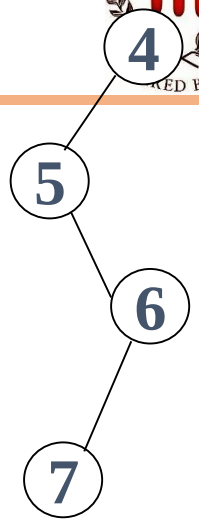
G_2

symmetric

undirected: $n^2/2$

directed: n^2

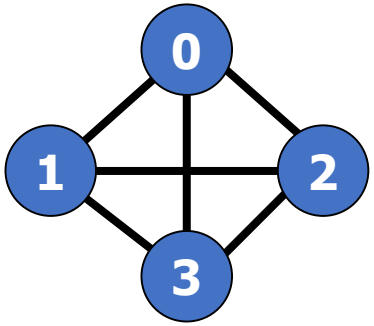




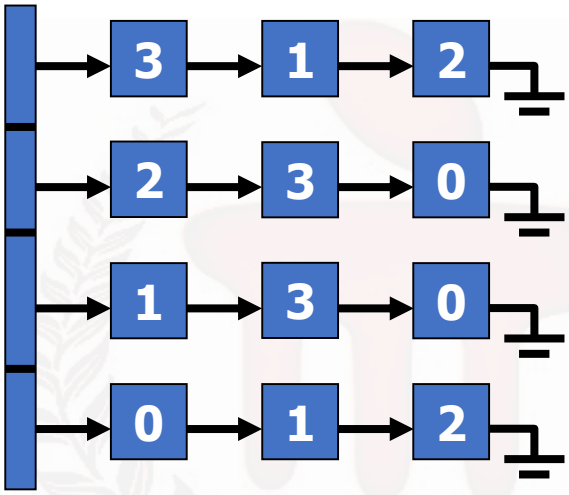
$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

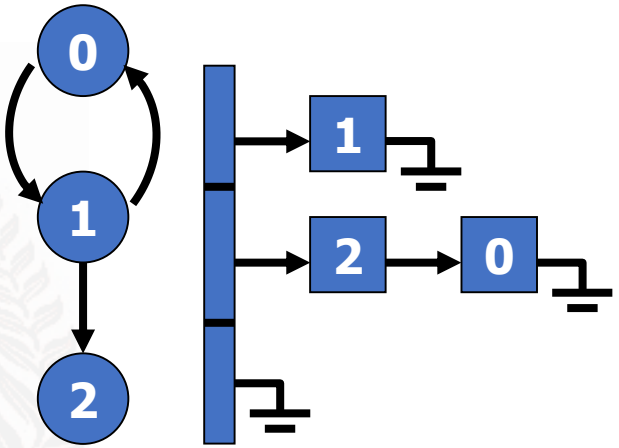
G_4

Adjacency lists

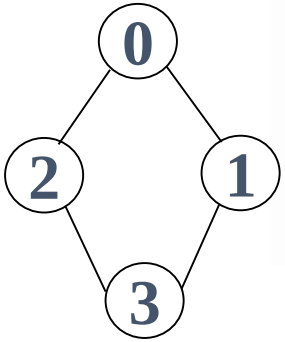


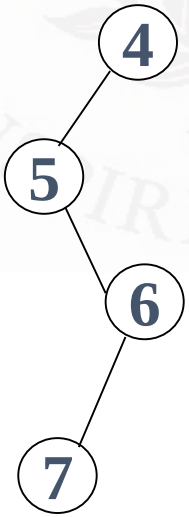
G1

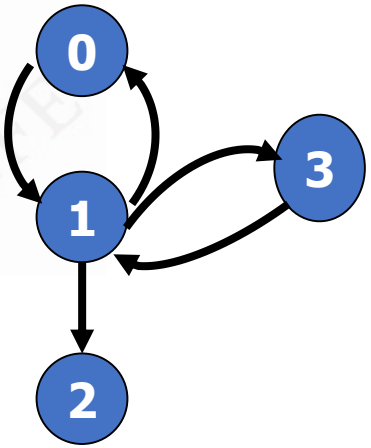




G3

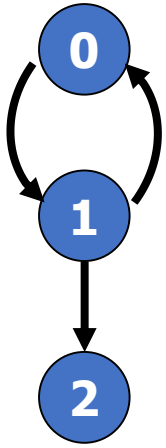






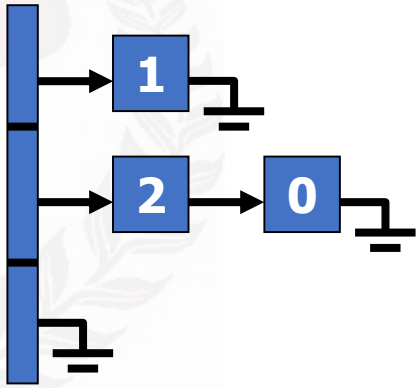
G3

Adjacency lists



G3

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$



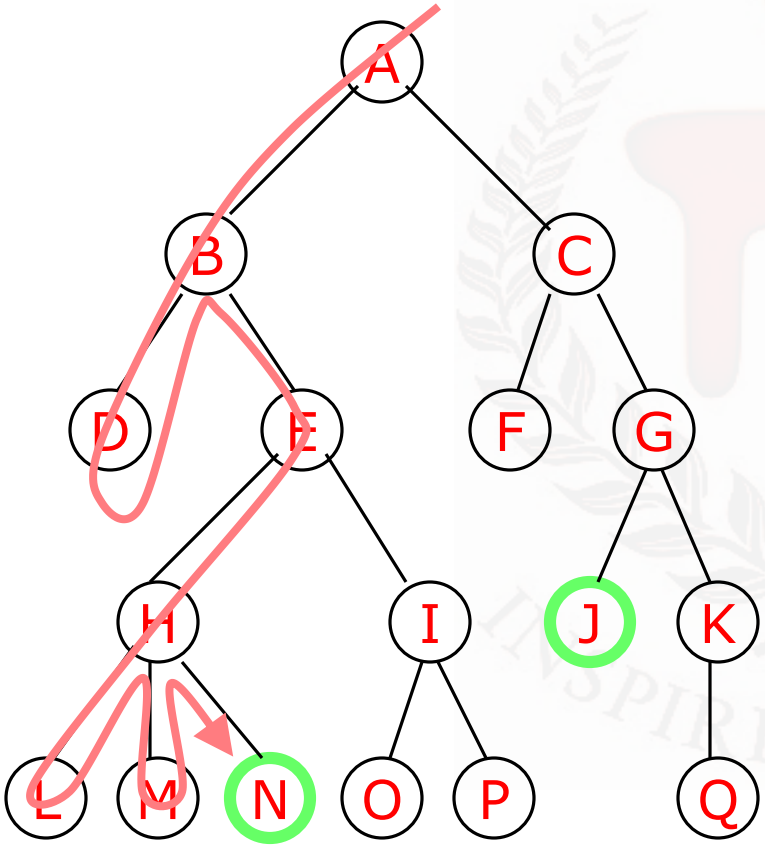
Some Graph Operations

- Traversal

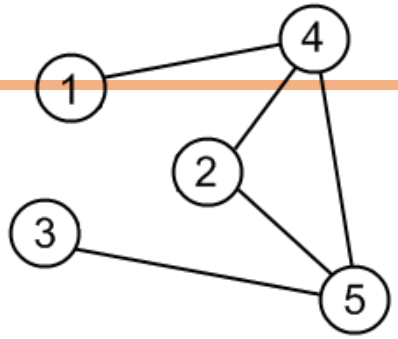
Given G=(V,E) and vertex v, find all w∈V, such that w connects v.

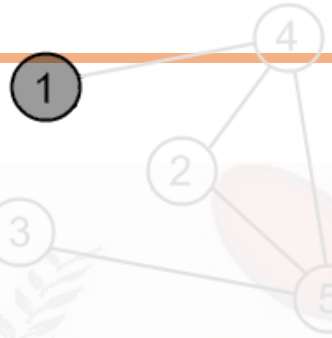
- Depth First Search (DFS)
  preorder tree traversal
- Breadth First Search (BFS)
  level order tree traversal

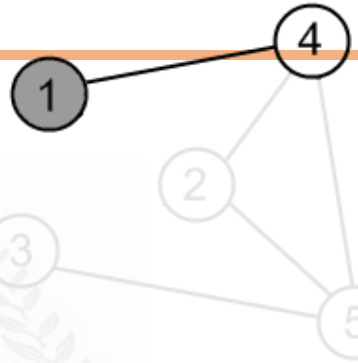
Depth-first search



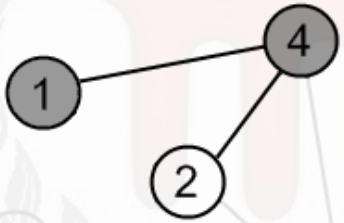
- A depth-first search (DFS) explores a path all the way to a leaf before backtracking and exploring another path
- For example, after searching A, then B, then D, the search backtracks and tries another path from B
- Node are explored in the order A B D E H L M N I O P C F G J K Q

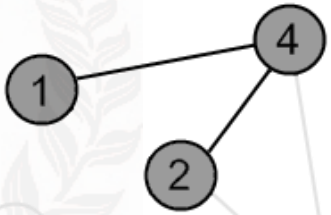

1
4
2
3
5

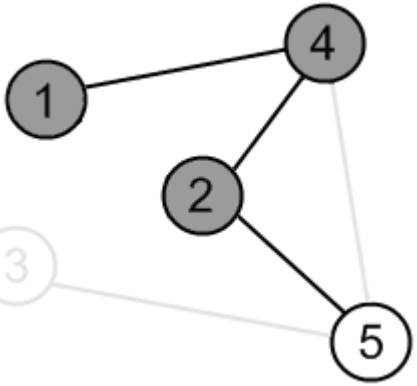

1

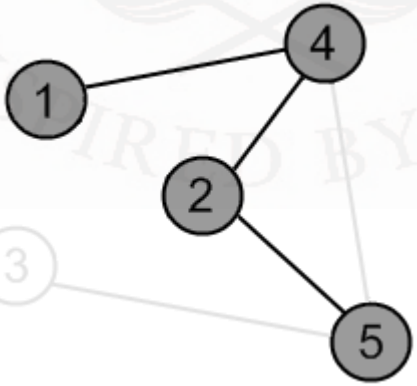

1
4

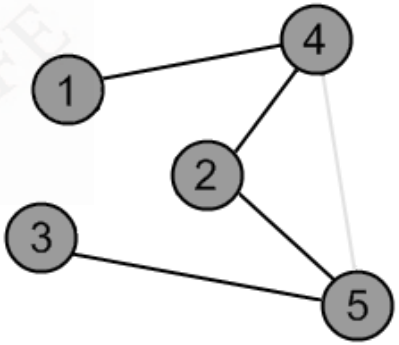

1
4

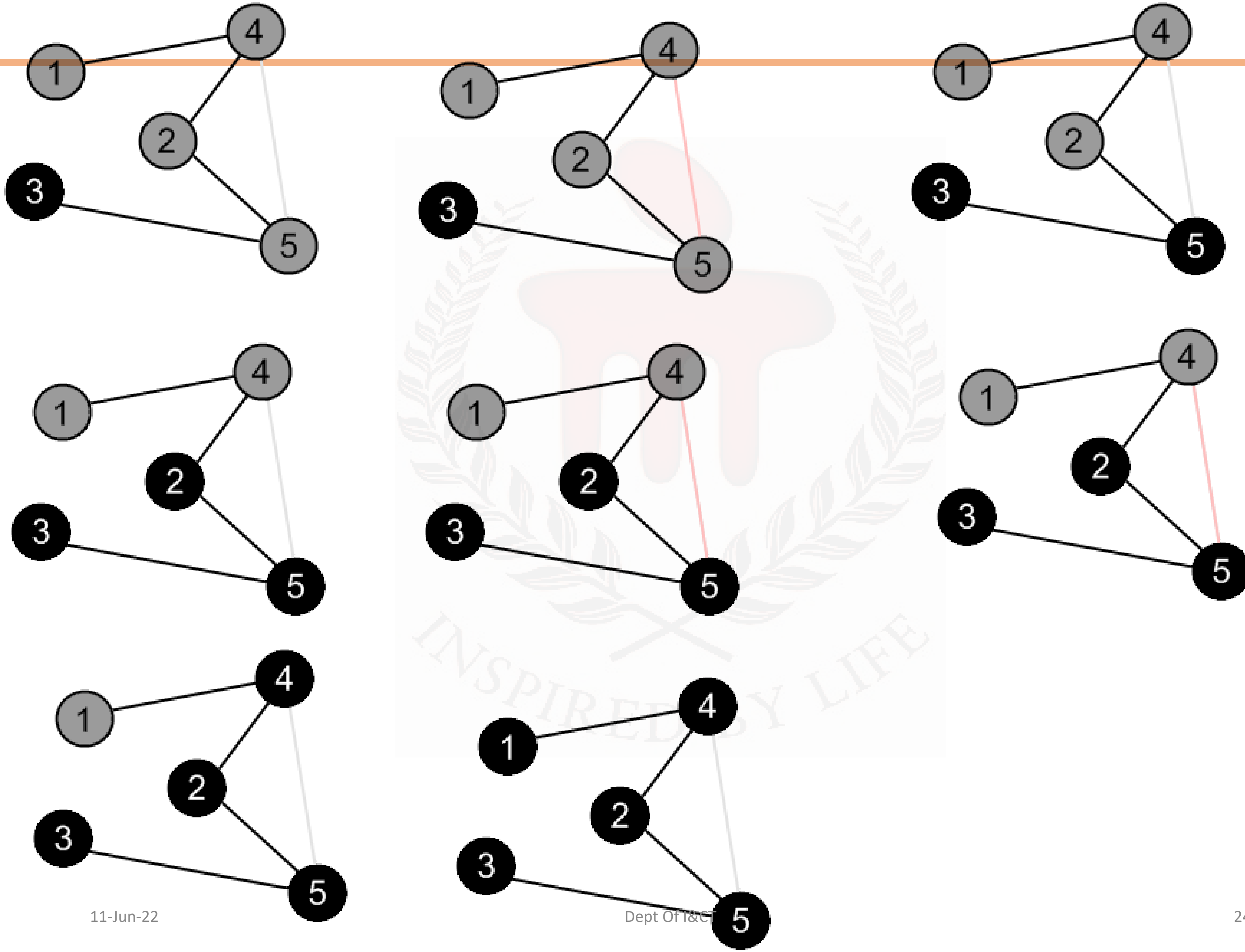

1
4
2


1
4
2


1
4
2
5


1
4
2
5


1
4
2
3
5


1
2
3
4
5

DFS Algorithm

Mark all the n vertices as not visited.

insert source into stack and mark it visited

while(Stack is not empty)

{

delete Stack element into variable u

place all the adjacent (not visited) vertices of u into Stack and also mark them visited

print u

}

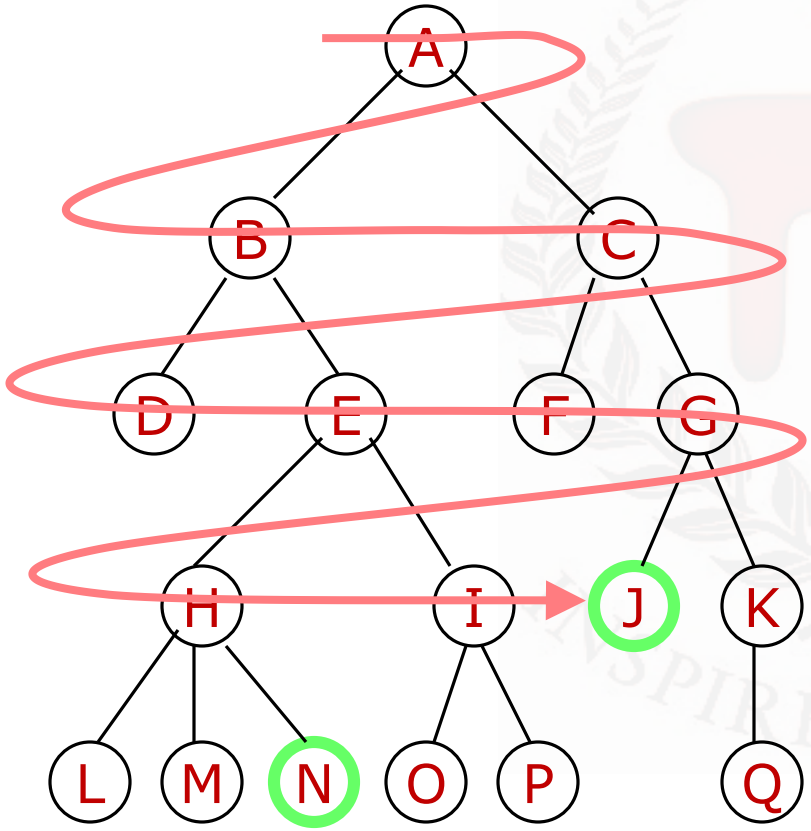
```
void dfs(int a[20][20],int n,int source)
{  int visited[10],u,v,i;
   for(i=1;i<=n;i++)       visited[i]=0;
   int S[20],top=-1;
   S[++top]=source;
   visited[source]=1;
   while(top>=0)
   {  u=S[top--];
      for(v=1;v<=n;v++)
      {   if(a[u][v]==1 && visited[v]==0)
          {
             visited[v]=1;         S[++top]=v;
          }
      }
      cout<<u<<" ";
   }
```

Breadth first search

It is so named because

It discovers all vertices at distance k from s before discovering vertices at distance k+1.

Breadth-first search



- A breadth-first search (BFS) explores nodes nearest the root before exploring nodes further away
- For example, after searching A, then B, then C, the search proceeds with D, E, F, G
- Node are explored in the order A B C D E F G H I J K L M N O P Q

Algorithm BFS

Mark all the n vertices as not visited.

insert source into Q and mark it visited

while(Q is not empty)

{

delete Q element into variable u

place all the adjacent (not visited) vertices of u into Q and also mark them visited

print u

}

```
void bfs(int a[20][20],int n,int source)
{
int visited[10],u,v,i;

  for(i=1;i<=n;i++)        visited[i]=0;
     int Q[20],f=-1,r=-1;
    Q[++r]=source;    visited[source]=1;
    while(f<r)
    {
       u=Q[++f];
       for(v=1;v<=n;v++)
       {   if(a[u][v]==1 && visited[v]==0)
           {
              visited[v]=1;
              Q[++r]=v;
           }
       }
       cout<<u<<" ";
    }
}
```

```
#include<iostream.h>
void bfs(int a[20][20],int n,int source);
void dfs(int a[20][20],int n,int source);
int main()
{
   int a[20][20],source, n,i,j;
   cout<<"Enter the no of vertices: ";    cin>>n;
   cout<<"Enter the adjacency matrix: ";
   for(i=1;i<=n;i++)       for(j=1;j<=n;j++)          cin>>a[i][j];
   cout<<"Enter the source: ";
   cin>>source;
   cout<<"\n BFS: ";    bfs(a,n,source);
   cout<<"\n DFS: ";    dfs(a,n,source);
   return 1;
}
```

Complexity

- Time complexity:
- Time complexity **represents the number of times a statement is executed**.
  - The time complexity of an algorithm is NOT the actual time required to execute a particular code, since that depends on other factors like programming language, operating software, processing power, etc
  - It is the amount of time the algorithm takes for each statement to complete. As a result, it is highly dependent on the size of the processed data.
  - In other words, the time complexity is how long a program takes to process a given input.
- Space Complexity:
- Referred to as the amount of memory consumed by the algorithm.
- Quantifies the amount of space or memory taken by an algorithm to run as a function of the length of the input.

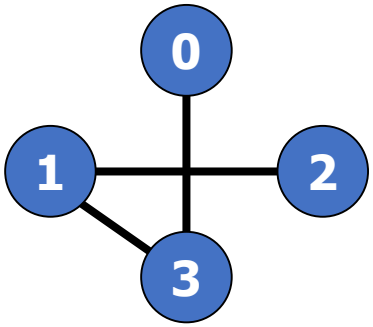
Complexity

- **DFS**
- The time complexity of DFS is *O(V + E)* where *V* is the number of vertices and *E* is the number of edges. This is because the algorithm explores each vertex and edge exactly once.
- The space complexity of DFS is *O(V)*. This is because in the worst case, the stack will be filled with all the vertices in the graph (Example: if the graph is a linear chain).

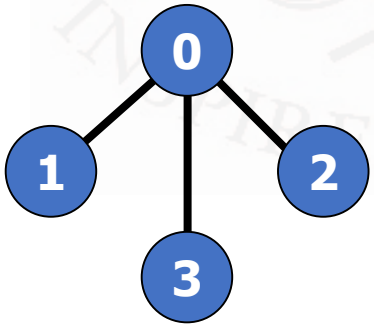
- **BFS**
- The time complexity of BFS is *O(V + E)* where *V* is the number of vertices and *E* is the number of edges. This is because in the worst case, the algorithm explores each vertex and edge exactly once.
  The space complexity of BFS is *O(V)* in the worst case. (Example: Star graph)

Spanning Tree (ST)

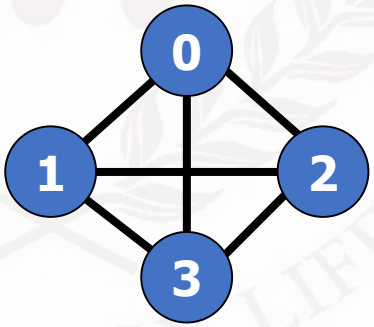
- A spanning tree is a minimal subgraph G', such that V(G')=V(G) and G' is connected. Spanning Tree is always acyclic.



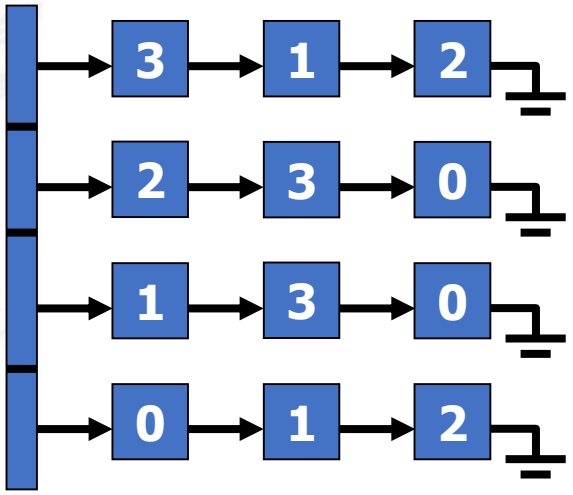
ST1(G)

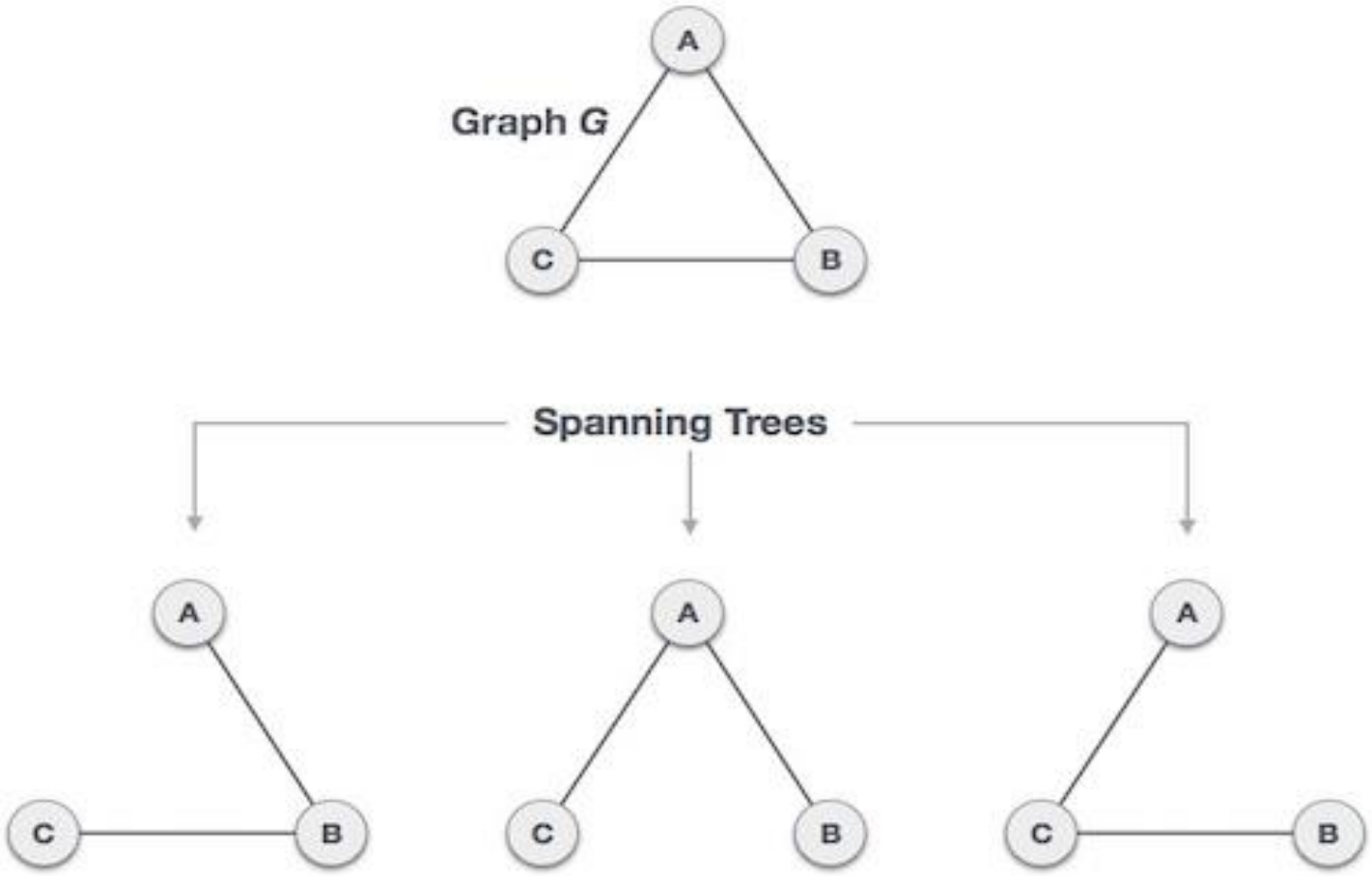


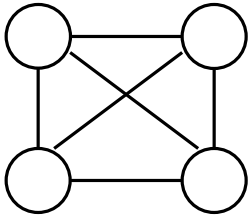
ST2(G)



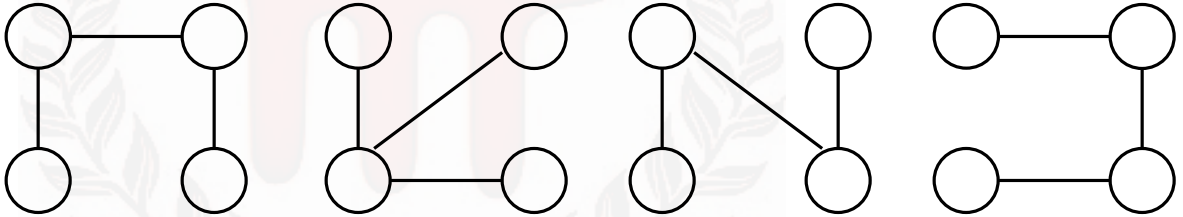
G1




Graph G
A
C
B
Spanning Trees
A
C
B
A
C
B
A
C
B



A connected, undirected graph



Four of the spanning trees of the graph

Important differences between BFS and DFS*

Sr. No.	Key	BFS	DFS
1	Definition	BFS, stands for Breadth First Search.	DFS, stands for Depth First Search.
2	Data structure	BFS uses Queue to find the shortest path.	DFS uses Stack to find the shortest path.
3	Source	BFS is better when target is closer to Source.	DFS is better when target is far from source.
4	Suitablity for decision tree	As BFS considers all neighbour so it is not suitable for decision tree used in puzzle games.	DFS is more suitable for decision tree. As with one decision, we need to traverse further to augment the decision. If we reach the conclusion, we won.
5	Speed	BFS is slower than DFS.	DFS is faster than BFS.
6	Time Complexity	Time Complexity of BFS = O(V+E) where V is vertices and E is edges.	Time Complexity of DFS is also O(V+E) where V is vertices and E is edges.

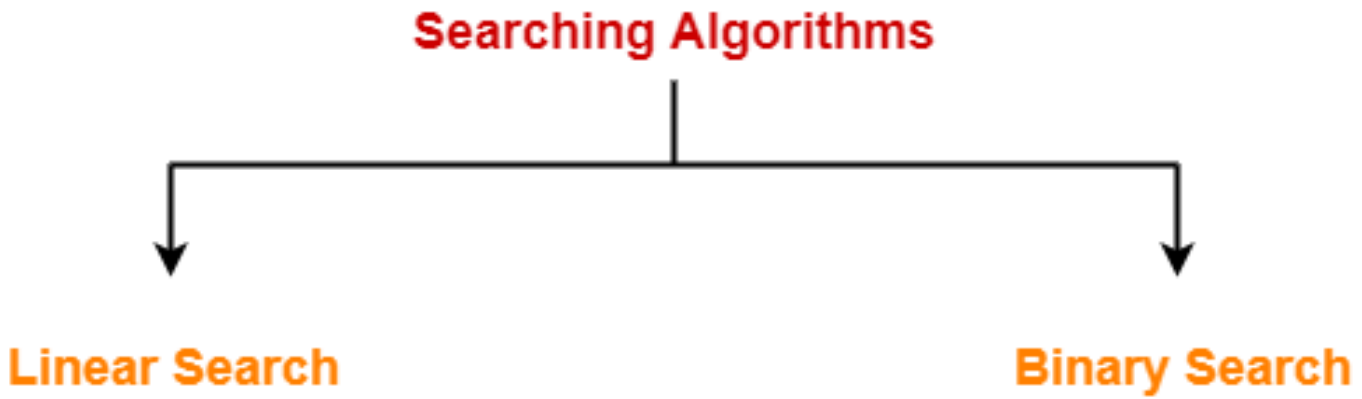
*https://www.tutorialspoint.com/difference-between-bfs-and-dfs

Searching and sorting algorithms

- **Searching-**

- Searching is a process of finding a particular element among several given elements.
- The search is successful if the required element is found.
- Otherwise, the search is unsuccessful.

- Searching Algorithms are a family of algorithms used for the purpose of searching.
- The searching of an element in the given array may be carried out in the following two ways-



Linear Search

- Linear Search (Array A, Value x)

- Step 1: Set i to 1
- Step 2: if i > n then go to step 7
- Step 3: if A[i] = x then go to step 6
- Step 4: Set i to i + 1
- Step 5: Go to Step 2
- Step 6: Print Element x Found at index i and go to step 8
- Step 7: Print element not found
- Step 8: Exit

- Linear Search is the simplest searching algorithm.
- It traverses the array sequentially to locate the required element.
- It searches for an element by comparing it with each element of the array one by one.
- So, it is also called as **Sequential Search**.

- Linear Search Algorithm is applied when-
- No information is given about the array.
- The given array is unsorted or the elements are unordered.
- The list of data items is smaller.

Linear Search Algorithm

- Consider-
- There is a linear array ‘a’ of size ‘n’.
- Linear search algorithm is being used to search an element ‘item’ in this linear array.
- If search ends in success, it sets loc to the index of the element otherwise it sets loc to -1.
-

- procedure linear_search (list, value)

  for each item in the list
    if match item == value
      return the item's location
    end if
  end for

- end procedure

```
Begin
 for i = 0 to (n - 1) by 1
do
 if (a[i] = item) then
set loc = i
Exit
endif
endfor
set loc = -1
 End
```

Step-01:

- It compares element 15 with the 1st element 92.
- Since 15 ≠ 92, so required element is not found.
- So, it moves to the next element.

Step-02:

- It compares element 15 with the 2nd element 87.
- Since 15 ≠ 87, so required element is not found.
- So, it moves to the next element.

- **Step-03:**
  - It compares element 15 with the 3rd element 53.
  - Since 15 ≠ 53, so required element is not found.
  - So, it moves to the next element.

- **Step-04:**
  - It compares element 15 with the 4th element 10.
  - Since 15 ≠ 10, so required element is not found.
  - So, it moves to the next element.

- **Step-05:**
  - It compares element 15 with the 5th element 15.
  - Since 15 = 15, so required element is found.
  - Now, it stops the comparison and returns index 4 at which element 15 is present.

Time Complexity Analysis-

- **Best case-**
- In the best possible case,
  - The element being searched may be found at the first position.
  - In this case, the search terminates in success with just one comparison.
  - Thus in best case, linear search algorithm takes O(1) operations.

- **Worst Case-**
- In the worst possible case,
  - The element being searched may be present at the last position or not present in the array at all.
  - In the former case, the search terminates in success with n comparisons.
  - In the later case, the search terminates in failure with n comparisons.

Binary Search

- Binary Search is one of the fastest searching algorithms.
- It is used for finding the location of an element in a linear array.
- It works on the principle of divide and conquer technique.

- Binary Search Algorithm can be applied only on **Sorted arrays**.
  - So, the elements must be arranged in-
    - Either ascending order if the elements are numbers.
    - Or dictionary order if the elements are strings.

- To apply binary search on an unsorted array,
  - First, sort the array using some sorting technique.
  - Then, use binary search algorithm.

Binary Search Algorithm

- Consider-
  - There is a linear array ‘a’ of size ‘n’.
  - Binary search algorithm is being used to search an element ‘item’ in this linear array.
  - If search ends in success, it sets loc to the index of the element otherwise it sets loc to -1.
  - Variables beg and end keeps track of the index of the first and last element of the array or sub array in which the element is being searched at that instant.
  - Variable mid keeps track of the index of the middle element of that array or sub array in which the element is being searched at that instant.

```
Begin
Set beg = 0
Set end = n-1
Set mid = (beg + end) / 2
while ( (beg <= end) and (a[mid] ≠ item) ) do
if (item < a[mid]) then
Set end = mid - 1
else
Set beg = mid + 1
endif
Set mid = (beg + end) / 2
endwhile
if (beg > end) then
Set loc = -1
else
Set loc = mid
endif
End
```

- Binary Search Algorithm searches an element by comparing it with the middle most element of the array.
- Then, following three cases are possible-
-
- **Case-01**
-
- If the element being searched is found to be the middle most element, its index is returned.
-

- **Case-02**

- If the element being searched is found to be greater than the middle most element,
- then its search is further continued in the right sub array of the middle most element.
- **Case-03**

- If the element being searched is found to be smaller than the middle most element,
- then its search is further continued in the left sub array of the middle most element.
- This iteration keeps on repeating on the sub arrays until the desired element is found or size of the sub array reduces to zero.

Time Complexity Analysis-

- Binary Search time complexity analysis is done below-
- In each iteration or in each recursive call, the search gets reduced to half of the array.
- So for n elements in the array, there are $\log_2 n$ iterations or recursive calls.

- **Time Complexity of Binary Search Algorithm is $O(\log_2 n)$,** where, n is the number of elements in the sorted linear array.

- Binary Search time complexity remains unchanged irrespective of the element position even if it is not present in the array.

- Consider-
- We are given the following sorted linear array.
- Element 15 has to be searched in it using Binary Search Algorithm.

- **Step-01:**

To begin with, we take beg=0 and end=6.

We compute location of the middle element as-

mid= (beg + end) / 2 = (0 + 6) / 2 = 3

Here, a[mid] = a[3] = 20 ≠ 15 and beg < end.

So, we start next iteration.

-

Step 2

- Since a[mid] = 20 > 15, so we take end = mid – 1 = 3 – 1 = 2 whereas beg remains unchanged.
- We compute location of the middle element as-
- mid= (beg + end) / 2 = (0 + 2) / 2= 1
- Here, a[mid] = a[1] = 10 ≠ 15 and beg < end.
- So, we start next iteration.

Step 3

- Since a[mid] = 10 < 15, so we take beg = mid + 1 = 1 + 1 = 2 whereas end remains unchanged.
- We compute location of the middle element as-
- Mid = (beg + end) / 2 = (2 + 2) / 2 = 2
- Here, a[mid] = a[2] = 15 which matches to the element being searched.
- So, our search terminates in success and index 2 is returned.

Insertion sort

- Main idea here,
  - that each element in the array is consumed in each iteration to find its right position in the sorted array such that the entire array is sorted at the end of all the iterations.
  - **i.e., it compares the current element with the elements on the left-hand side (sorted array)**
- If the current element is greater than all the elements on its left hand side, then it leaves the element in its place and moves on to the next element.
- Else it finds its correct position and moves it to that position by shifting all the elements, which are larger than the current element, in the sorted array to one position ahead.

```
for i = 1 to n
key = arr[i]
j = i - 1// comparing whether the first element is greater than the second element
// if yes, then store the largest element to the next position
while j >= 0 and arr[j] > key
arr[j + 1] = arr[j]
j = j - 1
end while// storing the smallest element in the correct position
arr[j + 1] = key
end fo
```

```
void insertionSort(int arr[], int n)
{
    int i, key, j;
    for (i = 1; i < n; i++) {
        key = arr[i];
        j = i - 1;

        // Move elements of arr[0..i-1], that are greater than
            //key to one position ahead of their current position
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

INSERTION-SORT(A)	*cost*	*times*
1 **for** $j = 2$ **to** $A.length$	c_1	n
2 $key = A[j]$	c_2	$n-1$
3 // Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
4 $i = j-1$	c_4	$n-1$
5 **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 $A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n}(t_j - 1)$
7 $i = i-1$	c_7	$\sum_{j=2}^{n}(t_j - 1)$
8 $A[i+1] = key$	c_8	$n-1$

Summing up, the total cost for insertion sort is -

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n}(t_j - 1) + c_7 \sum_{j=2}^{n}(t_j - 1) + c_8(n-1) .$$

- Merge Sort is a Divide and Conquer algorithm.
- It divides the input array into two halves, calls itself for the two halves, and then it merges the two sorted halves.
- **The merge() function** is used for merging two halves.
- The merge(arr, l, m, r) is a key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one.

MergeSort(a[], lb, ub)

If ub > lb

1. Find the middle point to divide the array into two halves:
   middle m = lb+ ((ub-lb)/2)
2. Call mergeSort for first half:
   Call mergeSort(a, lb, m)
3. Call mergeSort for second half:
   Call mergeSort(a, m+1, ub)
4. Merge the two halves sorted in step 2 and 3:
   Call merge(a, lb, m, ub)

Alternatively: middle m = (lb+ub)/2

MergeSort(arr[], l, r)

If r > l

1. Find the middle point to divide the array into two halves:

   middle m = l+ (r-l)/2

2. Call mergeSort for first half:

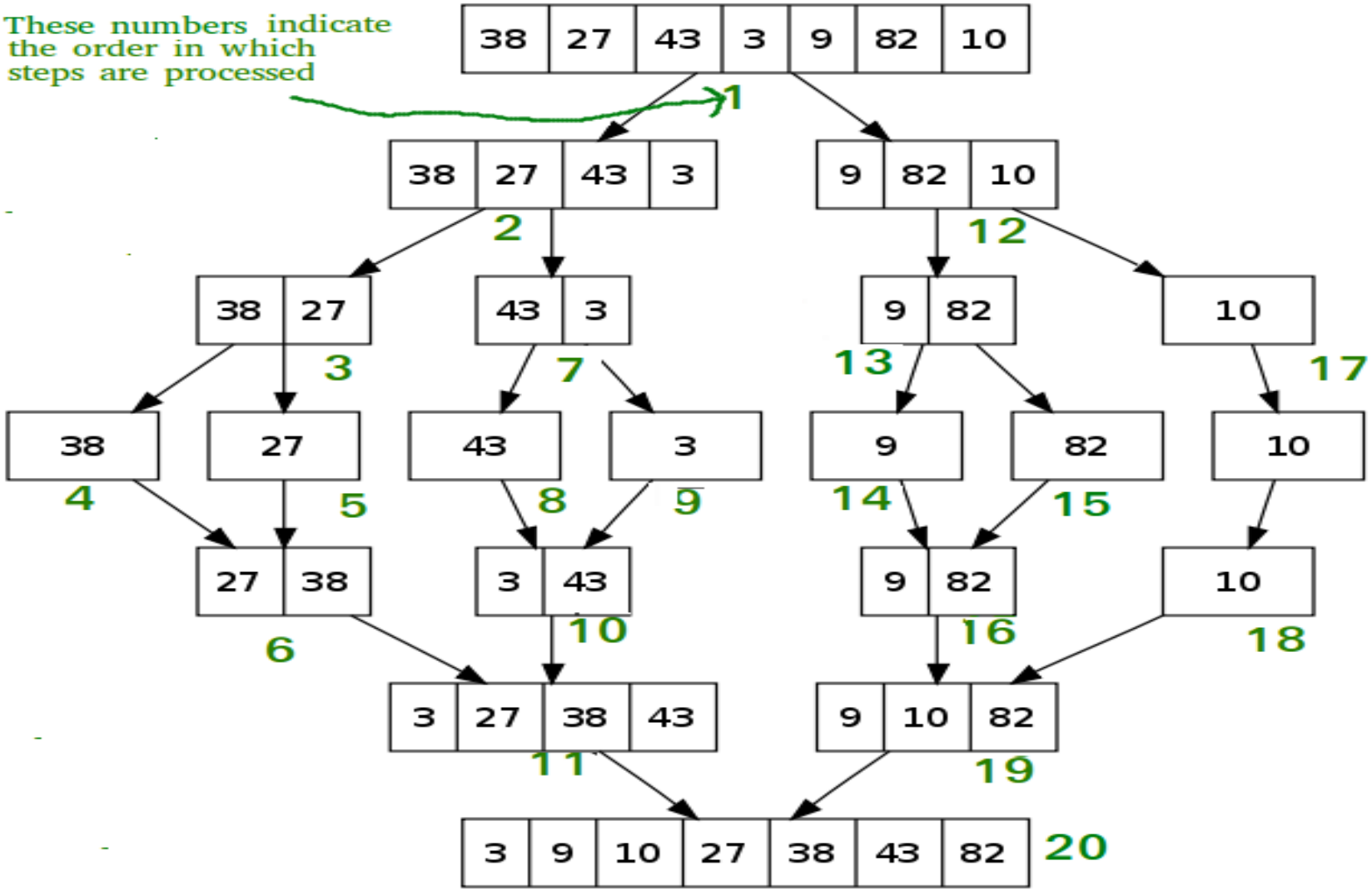
   Call mergeSort(arr, l, m)

3. Call mergeSort for second half:

   Call mergeSort(arr, m+1, r)

4. Merge the two halves sorted in step 2 and 3:

   Call merge(arr, l, m, r)


These numbers indicate the order in which steps are processed
38 27 43 3 9 82 10
1
38 27 43 3
9 82 10
2
12
38 27
43 3
9 82
10
3
7
13
17
38
27
43
3
9
82
10
4
5
8
9
14
15
27 38
3 43
9 82
10
6
10
16
18
3 27 38 43
9 10 82
11
19
3 9 10 27 38 43 82
20

```
void merge(int a[], int n)
{
 int len=1;
 int s[MAX];
 while(len<n)
 {
  merge_pass(a,s,n,len);
  len=len*2;
  merge_pass(s,a,n,len);
  len=len*2;
 }
}
```

```
void merge_pass(int a[], int sort[], int n, int len)
{
 int i, j;
 for(i=0;i<=n-2*len;i=i+2*len)
   mergesort(a,sort,i,i+len-1,i+2*len-1);
 if(i+len<n)
   mergesort(a,sort,i,i+len-1,n-1);
 else
   for(j=i;j<n;j++)
      sort[j]=a[j];
}
```

```
void mergesort(int a[],int sort[], int i, int m, int n)
{
 int j,k,t;
 j=m+1;
 k=i;
 while(i<=m && j<=n) {
  if(a[i]<=a[j])
      sort[k++]=a[i++];
  else
      sort[k++]=a[j++];
 } while(i<=m)
   sort[k++]=a[i++];
 while(j<=n)
   sort[k++]=a[j++];
}
```

Quick Sort

- QuickSort is a Divide and Conquer algorithm.
- 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.
- All this should be done in linear time.

```
void quick(int a[],int low, int high)
{
 int pivot,i,j,temp;
 if(low<high)
 {
  i=low;
  j=high+1;
  pivot=a[low];
  do
  {
   do
   {
    i++;
   }while(a[i]<pivot);
   do  {
    j--;
   }while(a[j]>pivot);
   if(i<j)
   {
    temp=a[i];
    a[i]=a[j];
    a[j]=temp;
   }
  }while(i<j);
  temp=a[low];
  a[low]=a[j];
  a[j]=temp;
  quick(a,low,j-1);
  quick(a,j+1,high);
 }}
```

Heap Sort

```
void heapSort(int arr[], int n)
{

   // Build heap (rearrange array)
   for (int i = n / 2 - 1; i >= 0; i--)
      heapify(arr, n, i);

   // One by one extract an element from heap
   for (int i = n - 1; i > 0; i--) {

      // Move current root to end
      swap(arr[0], arr[i]);

      // call max heapify on the reduced heap
      heapify(arr, i, 0);
   }
}
```

Complexity of Heapsort O (n log n)

```
void heapify(int arr[], int n, int i)
{
    int largest = i;
    int l = 2 * i + 1;
    int r = 2 * i + 2;

    // If left child is larger than root
    if (l < n && arr[l] > arr[largest])
        largest = l;

    // If right child is larger than largest so far
    if (r < n && arr[r] > arr[largest])
        largest = r;

    // If largest is not root
    if (largest != i) {
        swap(arr[i], arr[largest]);

        // Recursively heapify the affected sub-tree
        heapify(arr, n, largest);
    }
}
```

Shell Sort

```
int shellSort(int arr[], int n)
{
 for (int gap = n/2; gap > 0; gap /= 2)
   {
    for (int i = gap; i < n; i += 1)
      {
         int temp = arr[i];

         int j;
         for (j = i; j >= gap && arr[j - gap] > temp; j -= gap)
            arr[j] = arr[j - gap];

        arr[j] = temp;
      }
   }
   return 0;
}
```