Graphs:

DFS (**Depth First Search**) and **BFS** (**Breadth First Search**) are search algorithms used for graphs and trees.

> **Breadth-first search** (**BFS**) : starts at the tree root and explores the neighbor nodes first, before moving to the next level neighbors.

- Start from node *s*.
- Visit all neighbors of node *s*.
- Then visit all of their neighbors, if not already visited
- Continue until all nodes visited

> **Depth-first search** (**DFS**) : the idea is to travel as deep as possible from neighbor to neighbor before backtracking.

- First visit all nodes reachable from node *s* (ie visit neighbors of *s* and their neighbors)
- Then visit all (unvisited) nodes that are neighbors of *s*
- Repeat until all nodes are visited

A directed graph G, is a pair (V, E) where V is a set of vertices (nodes) and E is a set of directed edges. A directed graph is often called a **digraph.**

An undirected graph is also a pair (V,E) where V is the set of vertices and E is the set of edges. Edges are bi-directional. A to B then B to A.

A directed edge is an edge from one vertex to another. Sometimes called arcs.

A symmetric diagraph is a directed graph in which every directed edge has a reserve edge: a to b then b to a.

Adjacent: Two vertices a and b are adjacent if there is an edge connecting them.

Neighborhood $N_G(v)$, of a vertex v in G is the set of vertices that are adjacent to it.
Vertices *u* and *v* are **neighbors** if they are adjacent. So we say *v* is a neighbor of *u* and *u* is a neighbor of *v*. Every vertex have (n-1) neighbors.
Two vertices u and v are **neighbors** if they are adjacent.

Incident: Edges connecting a vertex v to its neighbors are said to be **incident** on v.

Degree of a vertex v is the number of edges incident on v . (the sum of degree = 2 * (no of edges).
The minimum degree of a vertex in a graph G is called the minimum degree of G, denoted $\delta(G)$.
The maximum degree of G is the maximum among all the vertex degrees in G. It is denoted $\Delta(G)$.

Planar graph: a graph without edge crossing(K_4)
Smallest non-planar with 5 vert

Any non-Planar graph contains K_5 or $K_{3,3}$

A subgraph of G is a graph G'= (V', E') where V' is a subset of V and E' is a subset of E.
it is zero or more vertex/edges deletion.

A proper subgraph of G is a subgraph G' of G such that: G' ≠ G and G' different from empty set.

An induced subgraph: its only obtained by vertex deletion , if an edge exist in G the its must exist in the induced subgraph

Handshaking Lemma: The sum of degrees in a graph is twice the number of edges.
number of edges is n(n-1)/2 by handshaking

A complete graph is an undirected graph such that any two vertices are connected (adjacent).The complete graph on n vertices is denoted by Kn.

Clique: a complete subgraph

Path: graph (or subgraph) whose vertices can be ordered so that two vertices are adjacent if and only if they are consecutive in the list.Where all nodes in subgraph are connected to every other node in that subgraph. We often denote a path of length *n* by *Pn*

Cycles: Cn: equal number of vertices and edges. Cycle in a diagraph is a non-empty path form one vertex back to itself.

Acyclic: Graph with no cycle.

Reachable: a vertex is reachable from another vertex if there is a path between them.

Connected components: Maximal connected subgraphs of a graph are the connected components of the graph

Strongly connected: A diagraph is strongly connected if there's is a path between every pair of vertices

Tree: Connected ,not cycle.

Forest: not connected, not cycle

Topological order: is a sequence of vertices that can be visited when traversing a DAG(Directed acyclic graph.
If a digraph has a cycle, it has no topological order

An adjacency list of a vertex v ∈Vis the list Adj[v]of vertices adjacent to v

A **shortest path** from u to v is a path of minimum weight from u to v. The **shortest-path weight** from u to v is defined as δ .

→A subpath of a **shortest path** is a shortest path.
→If a graph *G* contains a negative-weight cycle, then some **shortest paths** may not exist.
→ If all edge weights *w*(*u*, *v*)are *nonnegative*, all shortest-path weights must exist.

The Bellman–Ford algorithm is an **algorithm** that computes shortest paths from a single source vertex to all of the other vertices in a weighted digraph.

→ **Bellman–Ford** can be used on graphs with negative edge weights, as long as the graph contains no negative cycle reachable from the source vertex *s*.
https://www.youtube.com/watch?v=iTW2yFYd1Nc

A minimum spanning tree: A minimum spanning tree (MST) is the tree subgraph T of G such that it (a) contains all vertices of G and (b) has the minimum sum of edge weights.

→A single graph can have many different spanning trees.
→An algorithm employed to find minimum spanning trees is Prim's algorithm.

A spanning subgraph of a graph G is one that contains all vertices of G, but "may"not contain all edges.

→ the cost of a spanning tree is the sum of the weights of all its edges.

A problem is said to have optimal substructure if an optimal solution can be constructed efficiently from optimal solutions of its subproblems. This property is used to determine the usefulness of **dynamic programming and greedy algorithms** for a problem.

→ **Optimization problem:** Seeking for the best possible solution with the least or best cost

Dynamic Programming:

In one sense, dynamic programming can be thought of as the opposite of divide-and-conquer. Instead of starting from a top-down, divide-and-conquer approach (like recursion), dynamic programming instead solves the smallest subproblems first. It then caches these smallest solutions into a table and uses those solutions to progressively solve the next-larger-set of problems. A dynamic programming is a bottom-up solution method, in contrast to recursion as a top-down solution method.

→ **A dynamic programming** is a method for solving a complex problem by breaking it down into a collection of simpler subproblems, solving each of those subproblems just once

→Dp applies under 2 conditions:

1- Optimal substructure: an optimal solution can be constructed efficiently from the optimal solution of the subproblems.
→ A given problems has Optimal Substructure Property if optimal solution of the given problem can be obtained by using optimal solutions of its subproblems.
For example the shortest path problem has following optimal substructure property: If a node x lies in the shortest path from a source node u to destination node v then the shortest path from u to v is combination of shortest path from u to x and shortest path from x to v.

2- Overlapping subproblems: Revisiting and solving the same problem repeatedly **overlapping subproblems (**problem can be broken down into subproblems which are reused several times)

The longest common subsequence (LCS) problem is the problem of finding the longest subsequence common to all sequences in a set of sequences.
https://www.youtube.com/watch?v=P-mMvhfJhu8
→Code:

```
public static int longestCS(char str1[], char str2[], int length[][]) {
	int m = str1.length;
	int n = str2.length;

	for (int i = 1; i < m; i++)
		for (int j = 1; j < n; j++) {
			if (str1[i] == str2[j])
				length[i][j] = length[i-1][j-1] + 1;
			else if (length[i-1][j] >= length[i][j-1])
				length[i][j] = length[i-1][j];
			else
				length[i][j] = length[i][j-1]; }
		return length[m-1][n-1];} }
```

The longest increasing subsequence (LIS) problem is to find a subsequence of a given sequence in which the subsequence's elements are in sorted order, lowest to highest, and in which the subsequence is as long as possible.
https://www.youtube.com/watch?v=CE2b_-XfVDk
→Code:

```
public static int LongestIS(int A[], int n) {
	int sol[] = new int[n];
	int ans = 0;
	sol[0] = 1;
	for (int i = 1; i < n; i++) {
	sol[i] = 1;
			for (int j = 0; j < i; j++)
				if (A[j] < A[i])
					if (sol[i] < sol[j] + 1)
					sol[i] = sol[j] + 1;
				if (ans < sol[i])
				ans = sol[i]; }
				return ans ;}}
```

Dijkstra's algorithm is an algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, road networks

→From a **dynamic programming** point of view, **Dijkstra's algorithm** is a successive approximation scheme that solves the dynamic programming functional equation for the shortest path problem by the **Reaching** method
https://www.youtube.com/watch?v=WN3Rb9wVYDY

→The pseudo code:

```
while Q ≠∅
   do u ←EXTRACT-MIN(Q)
      S←S∪{u}                                          (SAME AS SLIDES)
         for each  v ∈Adj[u]
            do if d[v] > d[u] + w(u, v)
                  then d[v] ←d[u] + w(u, v)
```

The knapsack problem is a problem in combinatorial optimization(is a topic that consists of finding an optimal object from a finite set of objects) Ex:Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible. (slides)

→ Meaning, given n items of weight wi and value vi find the items that should be taken such that the weight is less than the maximum weight W and the corresponding total value is maximum.

Greedy:

A greedy algorithm always makes the choice that looks best at the moment. That ism it makes a locally optimal choice in the hope that his choice will lead to a globally optimal solution. A greedy algorithm obtains an optimal solution to a problem by making a sequence of choices. At each decision point, the algorithm makes choice that seems best at the moment. This decision does not always lead to an optimal solution.

→ A **greedy algorithm** is an algorithm that follows the problem solving heuristic of making the locally optimal choice at each stage with the hope of finding a global optimum. (a heuristic is a technique designed for solving a problem more quickly)
→ **A *greedy algorithm*** always makes the choice that looks best at the moment

Prim's algorithm is a greedy algorithm that finds a **minimum spanning tree** for a weighted undirected graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. The algorithm operates by building this tree one vertex at a time, from an arbitrary starting vertex, at each step adding the cheapest possible connection from the tree to another vertex.
https://www.youtube.com/watch?v=z1L3rMzG1_A

→The pseudo code:

$Q \leftarrow V$
$key[v] \leftarrow \infty$ for all $v \in V$
$key[s] \leftarrow 0$ for some arbitrary $s \in V$
while $Q \neq \emptyset$ (SAME AS SLIDES)
  do $u \leftarrow$ EXTRACT-MIN(Q)
    for each $v \in Adj[u]$
      do if $v \in Q$ and $w(u, v) < key[v]$
        then $key[v] \leftarrow w(u, v)$
          $\pi[v] \leftarrow u$ *degree*

Kruskal's algorithm is a **minimum-spanning-tree algorithm** where the algorithm finds an edge of the least possible weight that connects any two trees in the forest. It is a **greedy algorithm** in graph theory as it finds a minimum spanning tree for a connected weighted graph adding increasing cost arcs at each step.
https://www.youtube.com/watch?v=71UQH7Pr9kU

CHECK THE SLIDES EXAMPLE

Greedy AND Dynamic programming:

In dynamic programming, we make a choice at each step, ***but the choice may depend on solutions to subproblems***. In a greedy algorithm, we make whatever choice seems best at the moment and then solve the subproblems arising after the choice is made. The choice made by a greedy algorithm may depend on choices made so far, but ***it cannot depend on any future choices or on the solutions to subproblems***. Thus, unlike dynamic programming, which solves the subproblems bottom up, a greedy strategy usually progresses in top-down fashion, making one greedy choice after another, iteratively reducing each given problem instance to a smaller one.

→Greedy vs. Dynamic programming:

1- DP follows a buttom-up approach, whereas greedy foloows top down.
2- Dp has overlapping subproblems, and the solutions of the subproblems are used in order to compute the optimal solution of a larger problem. Whereas in greedy solutions doesn't depend on the subproblems since it makes a decision based on the current available choices and picks the optimal.

Class P: Consists of problems that are solvable in polynomial time. O(kn) for some constant K.

Class NP: consists of problems that can be verified in polynomial time, meaning that given a specific solution for a problem. Ex: VC.

Backtracking algorithms are based on a depth-first recursive search

→The classic example of the use of backtracking is the eight queens puzzle, that asks for all arrangements of eight chess queens on a standard chessboard

Vector Cover:

A **vertex cover** of a graph G is a subset of vertices S ⊆ V (G) such that every edge has an endpoint in S

→Pseudo Code:
CHECK CB

A minimal vertex cover :is a vertex cover that contain any other vertex ,where if u try to make it smaller you lose the property(no more vertex cover)
A minimum is minimal with the least number of elements

Queens:

The **queen puzzle** is the problem of placing n chess queens on an n×n chessboard so that no two queens threaten each other. Thus, a solution requires that no two queens share the same row, column, or diagonal.
→Code:

```
boolean Queens(int [] board ,int row){
	if(row ==8)
		return true;
	for(int col = 0 ; col<8;col++){
		if(feasible(board,row,col)){
			board[row]=col; //placing queen
			row++;
			if(Queens(board,row))
				return true;
			row--;}} //backtracking
			return false;}
```

3Color:

→Code in class:

```
boolean 3color(int[][]al,int deg[],int color,int vertex){
int(vertex==deg.length)
return true;

for(int  i = 1 ; i <=3; i++){
      if(feasible(al,deg,col,vertex,i)){

            color[vetex]=i;
if(3color(al,deg,col,vertex+1))
      return true;
col[vertex]=0; }}//backtracking
return false;
}
```

Clique:

the **clique problem** refers to any of the problems related to finding particular complete subgraphs in a graph(LIKE: sets of elements where each pair of elements is connected)

→Pseudo code:

```
boolean clique(graph G, int k )
      pick v ∈ V(G)
      if(deg[v]=n-1)
            return clique(G-v,n-1);
      if(clique(G-v,k))         //v not in clique
            return true;
      H=G[N(v)];   //graph induced by N(v)
      return clique(H,k-1);
```

A divide and conquer algorithm consists of two parts:
Divide the problem into smaller subproblems of the same type, and solve these subproblems recursively
Combine the solutions to the subproblems into a solution to the original problem

Big-O:

Algorithms	Best case	Worst case	Stable	In-place
Heap Sort	O(nlogn)	O(nlogn)	No	**yes**
Merge Sort	O(nlogn)	O(nlogn)	Yes	no
Quick sort	O(nlogn)	$O(n^2)$	No	Yes
Bubble Sort	O(n)	$O(n^2)$	Yes	Yes
Insertion	O(n)	$O(n^2)$	Yes	Yes
Selection	$O(n^2)$	$O(n^2)$	No	Yes
Counting	O(n+k)	O(n+k)	Yes	NO
BST	O(nlogn)/omega(logn)	O(n)	Yes	Yes
Bucket Sort	O(n+k)	$O(n^2)$	Yes	NO
Radix Sort	O(nk)	O(nk)	Yes	NO

Notes:

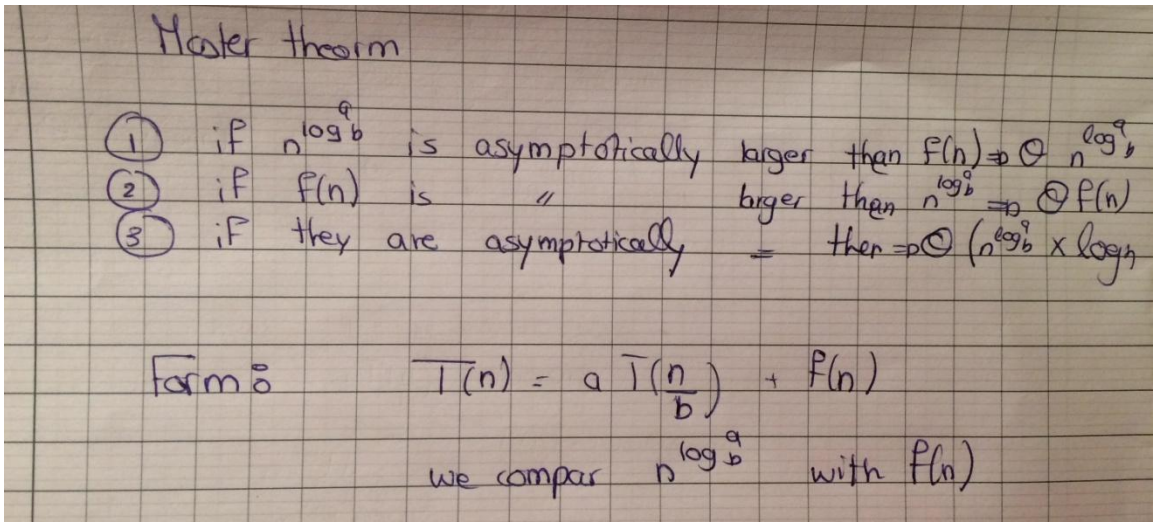
Il asr3 -→ az8ar but Il akbar → abt2

Binary Search in sorted array 0(logn)

Inserting element in BST O(n)

Inserting in AVL O(logn)

Master Theorm:



AVL Trees :

- is a balancing binary search tree. It was the first data structure to be invented.
- is a organized binary tree
- each node has a value
- it includes insertion , searching and deletion
- The balanced factor = height left – height right
- unbalanced binary tree needs $O(n)$ time in the worst-case

To know if the tree is AVL :

```
Boolean  isAVL( node T)
        if(T==null)
        return true ;
if(math.abs(height(T.left)-height(T.right)>1)
        return false;
return isAVL(T.left) && isAVL(T.right);
```

To calculate the height in a binary search tree :

```
height(node T):
  if node == null:
      return -1
  else:
      return max(height (T.left), height(T.right) + 1
```

To calculate the number of nodes in a binary search tree:

```
count(node T):
  if node == null:
      return 0
  else:
      return 1+ count(T.left) + count(T.right)
```

Codes: concerning DFS/BFS:

```
public static void Bfs(int[][] al, int[] d) {
             Queue<Integer> q = new LinkedList<Integer>();
             int[] color = new int[d.length];
             int u, v;
             q.offer(0);
             color[0] = 1;

             while (!q.isEmpty()) {
                    u = q.poll();
                    color[u] = 2;
                    System.out.println(u);
                    for (int j = 0; j < d[u]; j++) {
                           v = al[u][j];
                           if (color[v] == 0) {
                                  q.offer(v);
                                  color[v] = 1; }}}} }
```

```
public static void Dfs(int[][] al, int[] d){
    Stack <Integer> q = new Stack<Integer>();
    int[] color = new int[d.length];
    int u, v;
    q.push(0);
    color[0] = 1;

      while(!q.empty())
        {
          u = q.pop();
           color[u] = 2;
           System.out.println(u);
           for(int j = 0; j < d[u]; j++)
             {
                v = al[u][j];
                if(color[v] == 0)
                 {
                   q.push(v);
                   color[v] = 1; } } } }

public static void checkCycle(int al[][], int deg[]) {
             int color[] = new int[deg.length];
             Queue<Integer> q = new LinkedList<Integer>();
             int u, v;

             for (int i = 0; i < color.length; i++) {
                    if (color[i] == 0) {
                           q.offer(i);
                           color[i] = 1;

                           while (!q.isEmpty()) {
                                  u = q.poll();
                                  for (int j = 0; j < deg[u]; j++) {
                                         v = al[u][j];
                                         if (color[v] == 0) {
                                                q.offer(v);
                                                color[v] = 1;
                                         } else {
                                                System.out.print("has cycle");
                                                return; } }
                                                color[u] = 2; } } }
                                         System.out.print("has no cycle"); } }
```

```
public static boolean bipartite(int al[][], int deg[]) {
	int u, v;
	Queue<Integer> q = new LinkedList<Integer>();
	int color[] = new int[deg.length];

	for (int i = 0; i < deg.length; i++) {
		if (color[i] == 0) {

			q.offer(i);
			color[i] = 1;

			while (!q.isEmpty()) {

				u = q.poll();
			for (int j = 0; j < deg[u]; j++) {
					v = al[u][j];
					if (color[v] == color[u])
						return false;
					if (color[v] == 0) {
						q.offer(v);
						color[v] = 3 - color[u];
						} } } } }
						return true; }
```

```
public static boolean BfsConnected(int[][] al, int deg[]) {
int[] color = new int[deg.length];
Queue<Integer> q = new LinkedList<Integer>();

int u, v, count = 0;
for (int i = 0; i < color.length; i++) {
	if (color[i] == 0) {
		count++;
		if (count > 1) {
			return false;
		}
		q.add(i);
		color[i] = 1;
		while (!q.isEmpty()) {
			u = q.poll();
			color[u] = 2;
			for (int k = 0; k < deg[u]; k++) {
				v = al[u][k];
				if (color[v] == 0) {
					q.add(v);
					color[v] = 1; } } } } }
			return true; }
```

Notes:

- **-In case of a planar graph we can use 4 colors**
- **-Vertices can't have same color**
- **-Best algorithm for K coloring is : O *(2^k) (exponential)**
- **-Check from slides: activity selection, running time of prims…etc +codes of recursion(C.B)**

Code for MAX SUM(we did with yassin):

```
public class maxsum {
// i have a seq b2lbo number w bade le2e akbr sum bil array through subseq
      // its dp
      public static void main(String[] args) {
            // TODO Auto-generated method stub
            Scanner scan=new Scanner(System.in);
            int n = scan.nextInt();
            int [] array = new int[n];
            for(int i=0;i<array.length;i++){
                  array[i]=scan.nextInt();
            }
            int [] sol = new int[n+1];
            sol[0]=array[0];
            for(int i = 1 ; i <array.length;i++){
                  if(array[i]>=0){
                        if(array[i-1] >=0){
                              sol[i]+=sol[i-1]+array[i];
                        }
                        else sol[i]=array[i];}

                  else sol[i]=array[i];
            }
            for(int i = 0 ; i <array.length;i++){
                  System.out.println(sol[i]+ " ");
            }
            System.out.println(); } }
```