11 Graph Algorithms

Introduction :

Graphs are mathematical structures that represent pairwise relationships between objects. A graph is a flow structure that represents the relationship between various objects. It can be visualized by using the following two basic components :

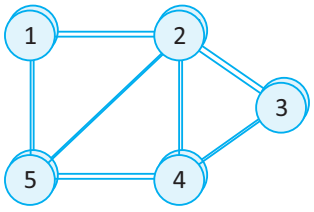
- **Nodes :** These are the most important components in any graph. Nodes are entities whose relationships are expressed using edges. If a graph comprises 2 nodes A and B and an undirected edge between them, then it expresses a bi-directional relationship between the nodes and edge.
- **Edges :** Edges are the components that are used to represent the relationships between various nodes in a graph. An edge between two nodes expresses a one-way or two-way relationship between the nodes.

Some Real Life Applications:

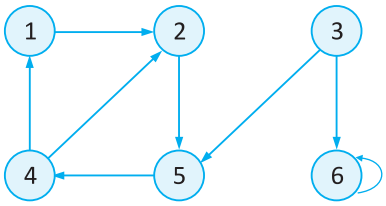
- **Google Maps :** To find a route based on shortest route/Time.
- **Social Networks :** Connecting with friends on social media, where each user is a vertex, and when users connect they create an edge.
- **Web Search :** Google, to search for webpages, where pages on the internet are linked to each other by hyperlinks, each page is a vertex and the link between two pages is an edge.
- **Recommendation System :** On eCommerce websites relationship graphs are used to show recommendations.

Types of Graphs :

- **Undirected :** An undirected graph is a graph in which all the edges are bi-directional i.e. the edges do not point in any specific direction.

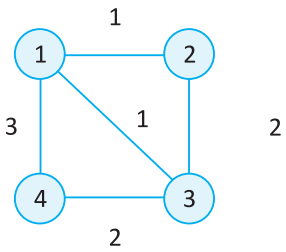


- **Directed :** A directed graph is a graph in which all the edges are uni-directional i.e. the edges point in a single direction.

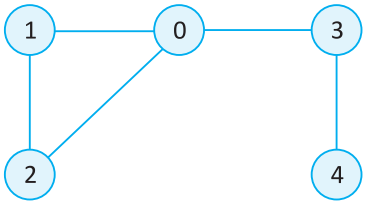


- **Weighted :** In a weighted graph, each edge is assigned a weight or cost. Consider a graph of 4 nodes as in the diagram below. As you can see each edge has a weight/cost assigned to it. If you want to go from vertex 1 to vertex 3, you can take one of the following 3 paths:
  - 1 -> 2 -> 3
  - 1 -> 3
  - 1 -> 4 -> 3

  Therefore the total cost of each path will be as follows: The total cost of 1 -> 2 -> 3 will be (1 + 2) i.e. 3 units - The total cost of 1 -> 3 will be 1 unit - The total cost of 1 -> 4 -> 3 will be (3 + 2) i.e. 5 units



- **Cyclic:** A graph is cyclic if the graph comprises a path that starts from a vertex and ends at the same vertex. That path is called a cycle. An acyclic graph is a graph that has no cycle.

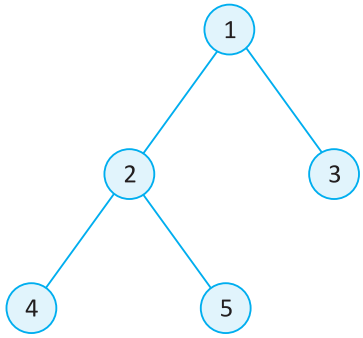


Cycle presents in the above graph (0->1->2)

A tree is an undirected graph in which any two vertices are connected by only one path. A tree is an acyclic graph and has N - 1 edges where N is the number of vertices. Each node in a graph may have one or multiple parent nodes. However, in a tree, each node (except the root node) comprises exactly one parent node.

***Note*:** A root node has no parent.

A tree cannot contain any cycles or self loops, however, the same does not apply to graphs.



(**Tree : There is no any cycle or self loop in this graph**)

Graph Representation

You can represent a graph in many ways. The two most common ways of representing a graph is as follows:

1. Adjacency Matrix

An adjacency matrix is a $V * V$ binary matrix A. Element $A_{i, j}$ is 1 if there is an edge from vertex i to vertex j else $A_{i, j}$ is 0.

***Note*:** A binary matrix is a matrix in which the cells can have only one of two possible values - either a 0 or 1.

The adjacency matrix can also be modified for the weighted graph in which instead of storing 0 or 1 in $A_{i, j}$ the weight or cost of the edge will be stored.

In an undirected graph, if $A_{i, j} = 1$, then $A_{j, i} = 1$.

In a directed graph, if $A_{i, j} = 1$, then may or may not be 1.

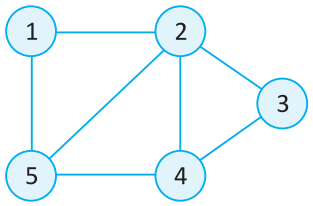
Adjacency matrix provides constant time access (O(1)) to determine if there is an edge between two nodes. Space complexity of the adjacency matrix is $O(V^2)$.

2. Adjacency List

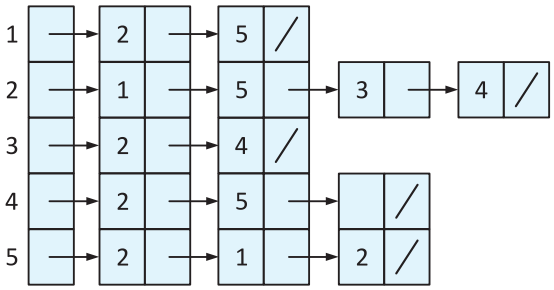
The other way to represent a graph is by using an adjacency list. An adjacency list is an array A of separate lists. Each element of the array A_i is a list, which contains all the vertices that are adjacent to vertex i.

For a weighted graph, the weight or cost of the edge is stored along with the vertex in the list using pairs. In an undirected graph, if vertex j is in list A_i then vertex i will be in list A_j.

The space complexity of adjacency list is O(V + E)because in an adjacency list information is stored only for those edges that actually exist in the graph. In a lot of cases, where a matrix is sparse using an adjacency matrix may not be very useful. This is because using an adjacency matrix will take up a lot of space where most of the elements will be 0, anyway. In such cases, using an adjacency list is better.



(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

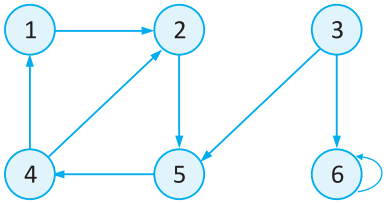
(c)

Two representations of an undirected graph.

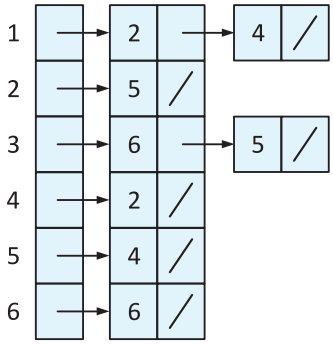
(a) An undirected graph G having five vertices and seven edges

(b) An adjacency-list representation of G

(c) The adjacency-matrix representation of G



(a)



(b)

	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)

Code for Adjacency list representation of a graph :

```
#include<iostream>
#include<list>
using namespace std;
class Graph{
      int V;
      list<int> *l;
public:
      Graph(int v){
            V = v;
            //Array of Linked Lists
            l = new list<int>[V];
      }
      void addEdge(int u,int v,bool bidir=true){
```

```
            l[u].push_back(v);
            if(bidir){
                  l[v].push_back(u);
            }
      }
      void  printAdjList(){
            for(int  i=0;i<V;i++){
                        cout<<i<<“->”;
                        //l[i] is a linked list
                        for(int  vertex:  l[i]){
                              cout<<vertex<<“,”;
                        }
                        cout<<endl;
            }
      }
};
int  main(){
      // Graph has 5 vertices number from 0 to 4
      Graph  g(5);
      g.addEdge(0,1);
      g.addEdge(0,4);
      g.addEdge(4,3);
      g.addEdge(1,4);
      g.addEdge(1,2);
      g.addEdge(2,3);
      g.addEdge(1,3);
      g.printAdjList();
return  0;
}
```

Graph Traversal :

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. The order in which the vertices are visited are important and may depend upon the algorithm or question that you are solving.

During a traversal, it is important that you track which vertices have been visited. The most common way of tracking vertices is to mark them.

Breadth First Search (BFS) :

There are many ways to traverse graphs. BFS is the most commonly used approach.

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.

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

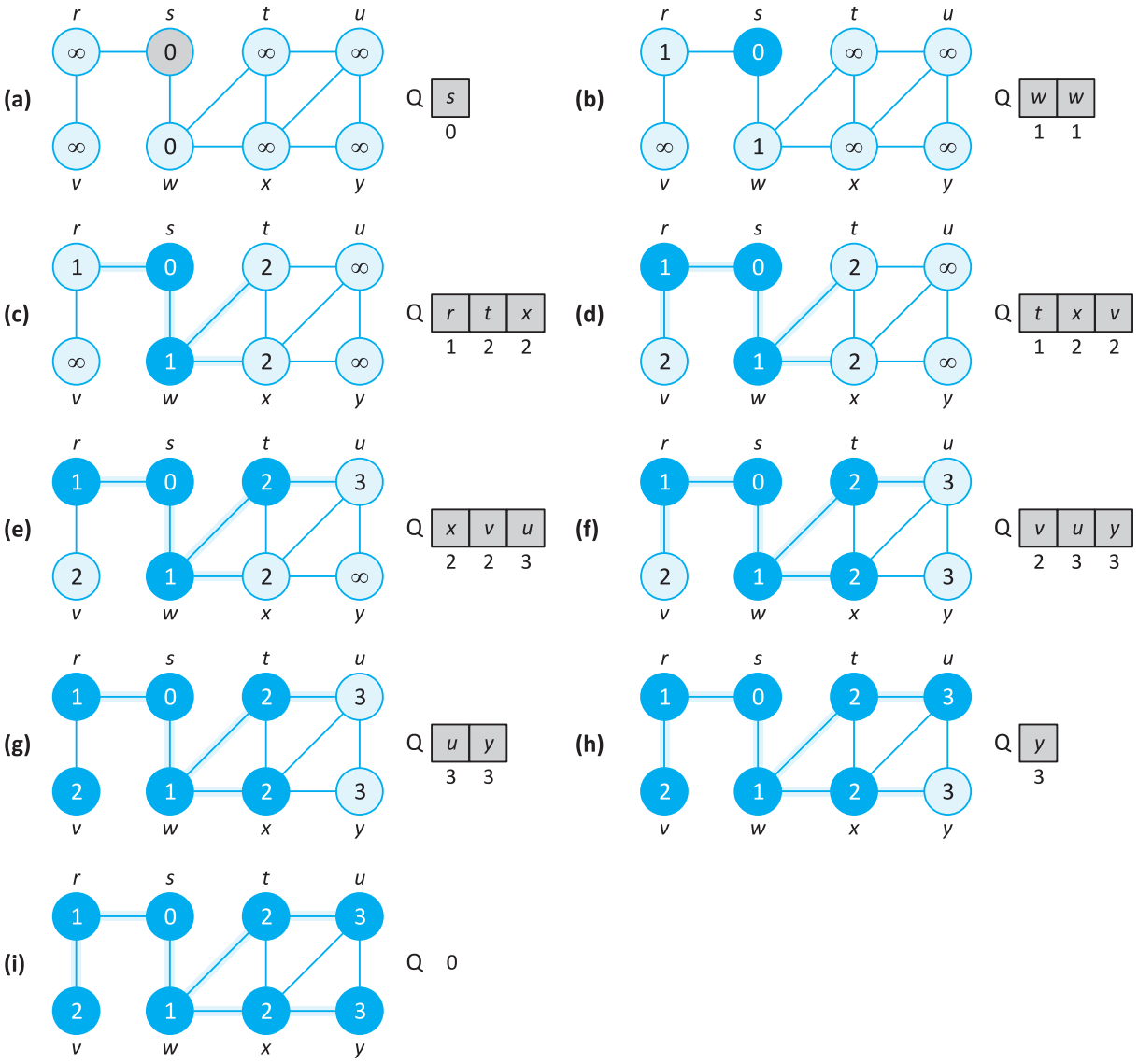
Breadth-first search colors each vertex white, gray, or black. All vertices start out white and may later become gray and then black. A vertex is *discovered* the first time it is encountered during the search, at which time it becomes nonwhite. Gray and black vertices, therefore, have been discovered, but breadth-first search distinguishes between them to ensure that the search proceeds in a breadth-first manner. If (*u*, v) " *E* and vertex *u* is black, then vertex v is either gray or black; that is, all vertices adjacent to black vertices have been discovered. Gray vertices may have some adjacent white vertices; they represent the frontier between discovered and undiscovered vertices.

Breadth-first search constructs a breadth-first tree, initially containing only its root, which is the source vertex *s*. Whenever a white vertex v is discovered in the course of scanning the adjacency list of an already discovered vertex *u*, the vertex v and the edge (*u*, v) are added to the tree. We say that *u* is the *predecessor* or *parent* of v in the breadth-first tree.

Algorithm :

```
BFS(G, s)
    for each vertex u ∈ V [G] - {s}
        do color[u] ← WHITE
           d[u] ← ∞
           π[u] ← NIL
    color[s] ← GRAY
    d[s] ← 0
    π[s] ← NIL
    Q ← 0
    ENQUEUE(Q, s)
    while Q ≠ 0
       do u ← DEQUEUE(Q)
          for each ∈ Adj[u]
              do if color[v] = WHITE
                 then color[v] ← GRAY
```

```
                    d[v] ← d[u] + 1
                    π[v] ← u
                    ENQUEUE(Q, v)
    color[u] ← BLACK
```



The operation of BFS on an undirected graph. Tree edges are shown shaded as they are prduced by BFS. Within each vertex vertex u is shown d[u]. The queue Q is shown at the beginning of each iteration of the while loop of lines 10-18. Vertex distances are shown next to vertices in the queue.

Code :

```
#include<iostream>
#include<map>
#include<list>
#include<queue>
using namespace std;

template<typename T>
class Graph{

    map<T,list<T> > adjList;

public:
    Graph(){

    }
    void addEdge(T u, T v,bool bidir=true){

        adjList[u].push_back(v);
        if(bidir){
            adjList[v].push_back(u);
        }
    }

    void print(){
```

Time Complexity :

Time complexity and space complexity of above algorithm is O(V + E) and O(V).

Application of BFS

1. **Shortest Path and Minimum Spanning Tree for unweighted graph** In unweighted graph, the shortest path is the path with least number of edges. With Breadth First, we always reach a vertex from given source using minimum number of edges. Also, in case of unweighted graphs, any spanning tree is Minimum Spanning Tree and we can use either Depth or Breadth first traversal for finding a spanning tree.

   Code for finding shortest path using BFS :

```
#include<iostream>
#include<map>
#include<list>
```

```
#include<queue>
using namespace std;
template<typename T>
class Graph{
            map<T,list<T> > adjList;
public:
            Graph(){
 }
            void addEdge(T u, T v,bool bidir=true){
                  adjList[u].push_back(v);
            if(bidir){
                  adjList[v].push_back(u);
            }
      }

      void print(){
            //Iterate over the map
            for(auto i:adjList){
                  cout<<i.first<<"->";
            //i.second is LL
            for(T entry:i.second){
                  cout<<entry<<",";
            }
            cout<<endl;
      }
}
voidbfs(T src){
      queue<T> q;
      map<T,int>dist;
      map<T,T> parent;
      for(auto i:adjList){
            dist[i.first] = INT_MAX;
      }
      q.push(src);
```

```
                dist[src] = 0;
                parent[src] = src;
                while(!q.empty()){
                        T node = q.front();
                        cout<<node<<" ";
                        q.pop();
                        // For the neigbours of the current node, find out the nodes which
                are not visited
                        for(intneigbour :adjList[node]){
                                if(dist[neigbour]==INT_MAX){
                                        q.push(neigbour);
                                        dist[neigbour] = dist[node]  + 1;
                                        parent[neigbour] = node;
                                }
                        }
                }
                //Print the distance to all the nodes
                for(auto i:adjList){
                        T node = i.first;
                        cout<<"Dist of "<<node<<" from "<<src<<" is "<<dist[node]<<endl;
                }
        }
};
intmain(){
        Graph<int> g;
        g.addEdge(0,1);
        g.addEdge(1,2);
        g.addEdge(0,4);
        g.addEdge(2,4);
        g.addEdge(2,3);
        g.addEdge(3,5);
        g.addEdge(3,4);
        g.bfs(0);
}
```

2. **Peer to Peer Networks :** In Peer to Peer Networks like BitTorrent, Breadth First Search is used to find all neighbor nodes.
3. **Crawlers in Search Engines :** Crawlers build index using Breadth First. The idea is to start from source page and follow all links from source and keep doing same. Depth First Traversal can also be used for crawlers, but the advantage with Breadth First Traversal is, depth or levels of built tree can be limited.
4. **Social Networking Websites :** In social networks, we can find people within a given distance 'k' from a person using Breadth First Search till 'k' levels.
5. **GPS Navigation systems :** Breadth First Search is used to find all neighboring locations.
6. **Broadcasting in Network :** In networks, a broadcasted packet follows Breadth First Search to reach all nodes.
7. **In Garbage Collection :** Breadth First Search is used in copying garbage collection using Cheney's algorithm. Refer this and for details. Breadth First Search is preferred over Depth First Search because of better locality of reference:
8. **Cycle detection in undirected graph :** In undirected graphs, either Breadth First Search or Depth First Search can be used to detect cycle. In directed graph, only depth first search can be used.
9. **Ford–Fulkerson algorithm :** In Ford-Fulkerson algorithm, we can either use Breadth First or Depth First Traversal to find the maximum flow. Breadth First Traversal is preferred as it reduces worst case time complexity to O(VE2).
10. **To test if a graph is Bipartite :** We can either use Breadth First or Depth First Traversal.
11. **Path Finding :** We can either use Breadth First or Depth First Traversal to find if there is a path between two vertices.
12. **Finding all nodes within one connected component :** We can either use Breadth First or Depth First Traversal to find all nodes reachable from a given node.

Depth First Search (DFS)

The DFS algorithm is a recursive algorithm that uses the idea of backtracking. It involves exhaustive searches of all the nodes by going ahead, if possible, else by backtracking.

Here, the word backtrack means that when you are moving forward and there are no more nodes along the current path, you move backwards on the same path to find nodes to traverse. All the nodes will be visited on the current path till all the unvisited nodes have been traversed after which the next path will be selected.

This recursive nature of DFS can be implemented using stacks. The basic idea is as follows:

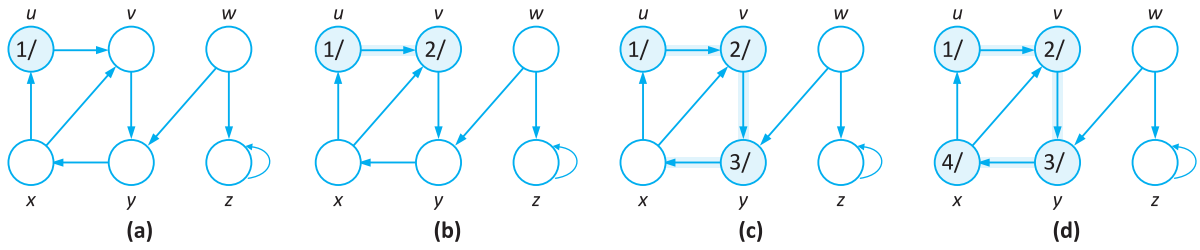
- Pick a starting node and push all its adjacent nodes into a stack.
- Pop a node from stack to select the next node to visit and push all its adjacent nodes into a stack.

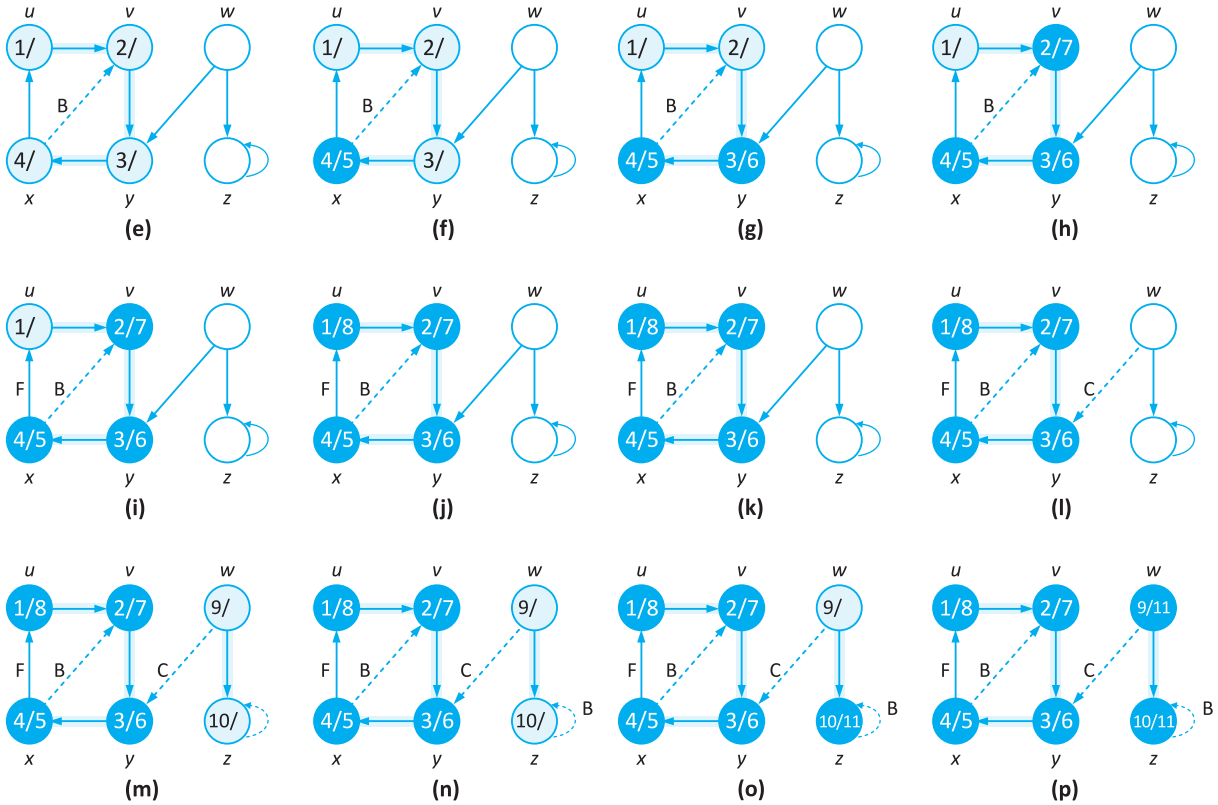
- Repeat this process until the stack is empty. However, ensure that the nodes that are visited are marked. This will prevent you from visiting the same node more than once. If you do not mark the nodes that are visited and you visit the same node more than once, you may end up in an infinite loop.

  As in breadth-first search, vertices are colored during the search to indicate their state. Each vertex is initially white, is grayed when it is ***discovered*** in the search, and is blackened when it is ***finished***, that is, when its adjacency list has been examined completely. This technique guarantees that each vertex ends up in exactly one depth-first tree, so that these trees are disjoint.

Algorithm :

```
DFS(G)
for each vertex u ∈ V[G]
     do color [u] ← WHITE
         p[u] ← NIL
time ← 0
for each vertex u ∈ V[G]
     do if color[u] = WHITE
           then DFS-VISIT(u)
DFS-VISIT(u)
color[u] ← GRAY        White vertex u has just been discovered
time ← time + 1
d[u] ← time
for each v Î Adj[u]           Explore edge (u, v)
     do if color[v] = WHITE
           then π[v] ← u
                DFS-VISIT(v)
color[u] BLACK          Blacken u; it is finished
f[u] ← time ← time + 1
```





The progress of the depth-first-search algorithm DFS on a directed path. As edges are explored by the algorithm, they are shown as either shaded (if they are tree edges) or dashed (otherwise). Nontree edges are labeled B, C or F according to whether they are back, cross, or forward edges. Vertices are timestamped by discovery time/finishing time.

Code :

```
#include<iostream>
#include<map>
#include<list>
using namespace std;

template<typename T>
class Graph{

    map<T,list<T> > adjList;

public:
    Graph(){

    }
    void addEdge(T u, T v,bool bidir=true){
        adjList[u].push_back(v);
```

```
            if(bidir){
                  adjList[v].push_back(u);
            }
      }

      void print(){

            //Iterate over the map
            for(auto i:adjList){
                  cout<<i.first<<"->";

                  //i.second is LL
                  for(T entry:i.second){
                        cout<<entry<<",";
                  }
                  cout<<endl;
            }
      }
      void dfsHelper(T node,map<T,bool> &visited){
            //Whenever to come to a node, mark it visited
            visited[node] = true;
            cout<<node<<" ";
            //Try to find out a node which is neigbour of current node and not
yet visited
            for(T neighbour: adjList[node]){
                  if(!visited[neighbour]){
                        dfsHelper(neighbour,visited);
                  }
            }
      }
      void dfs(T src){
            map<T,bool> visited;
            dfsHelper(src,visited);
      }
};
int main(){

      Graph<int> g;
      g.addEdge(0,1);
      g.addEdge(1,2);
      g.addEdge(0,4);
```

```
    g.addEdge(2,4);
    g.addEdge(2,3);
    g.addEdge(3,4);
    g.addEdge(3,5);
    g.dfs(0);


return 0;
}
```

Time Complexity :

Time complexity of above algorithm is O(V + E).

Application of DFS :

1. For an unweighted graph, DFS traversal of the graph produces the minimum spanning tree and all pair shortest path tree.
2. **Detecting cycle in a graph :** A graph has cycle if and only if we see a back edge during DFS. So we can run DFS for the graph and check for back edge.

Code for detect a cycle in a graph :

```
#include<iostream>
#include<map>
#include<list>
using namespace std;
template<typename T>
class Graph{
    map<T,list<T> > adjList;
public:
    Graph(){
    }
    void addEdge(T u, T v,bool bidir=true){
          adjList[u].push_back(v);
          if(bidir){
                adjList[v].push_back(u);
          }
    }
    bool isCyclicHelper(T node,map<T,bool> &visited,map<T,bool> &inStack){
```

```
                //Processing the current node - Visited, InStack
                visited[node] = true;
                inStack[node] = true;
```

3. **Path Finding :** We can specialize the DFS algorithm to find a path between two given vertices u and z.
   (i) Call DFS(G, u) with u as the start vertex.
   (ii) Use a stack S to keep track of the path between the start vertex and the current vertex.
   (iii) As soon as destination vertex z is encountered, return the path as the contents of the stack.

4. **Topological Sorting :** In the field of computer science, a **topological sort** or **topological ordering** of a directed graph is a linear ordering of its vertices such that for every directed edge *uv* from vertex *u* to vertex *v*, *u* comes before *v* in the ordering. For instance, the vertices of the graph may represent tasks to be performed, and the edges may represent constraints that one task must be performed before another; in this application, a topological ordering is just a valid sequence for the tasks. A topological ordering is possible if and only if the graph has no directed cycles, that is, if it is a directed acyclic graph (DAG). Any DAG has at least one topological ordering.

Code for printing Topological sorting of DAG :

```
#include<iostream>
#include<map>
#include<list>
#include<queue>
using namespace std;
template<typename T>
class Graph{
     map<T,list<T> > adjList;
 public:
     Graph(){
     }
     void addEdge(T u, T v,bool bidir=true){
           adjList[u].push_back(v);
           if(bidir){
                 adjList[v].push_back(u);
           }
     }
     void topologicalSort(){
```

```
queue<T> q;
map<T,bool> visited;
map<T,int> indegree;
for(auto i:adjList){
        //i is pair of node and its list
        T node = i.first;
        visited[node] = false;
        indegree[node] = 0;
}
//Init the indegrees of all nodes
for(auto i:adjList){
    T u = i.first;
    for(T v: adjList[u]){
    indegree[v]++;
    }
}
//Find out all the nodes with 0 indegree
for(auto i:adjList){
    T node = i.first;
    if(indegree[node]==0){
        q.push(node);
    }
}
//Start with algorithm
while(!q.empty()){
    T node = q.front();
    q.pop();
    cout<<node<<"—>";
    for(T neighbour:adjList[node]){
        indegree[neighbour]—;
    if(indegree[neighbour]==0){
        q.push(neighbour);
    }
}
```

```
                }
        }
};

intmain(){
        Graph<string> g;
        g.addEdge("English","ProgrammingLogic",false);
        g.addEdge("Maths","Programming Logic",false);
        g.addEdge("Programming Logic","HTML",false);
        g.addEdge("Programming Logic","Python",false);
        g.addEdge("Programming Logic","Java",false);
        g.addEdge("Programming Logic","JS",false);
        g.addEdge("Python","WebDev",false);
        g.addEdge("HTML","CSS",false);
        g.addEdge("CSS","JS",false);
        g.addEdge("JS","WebDev",false);
        g.addEdge("Java","WebDev",false);
        g.addEdge("Python","WebDev",false);
        g.topologicalSort();

return0;
}
```

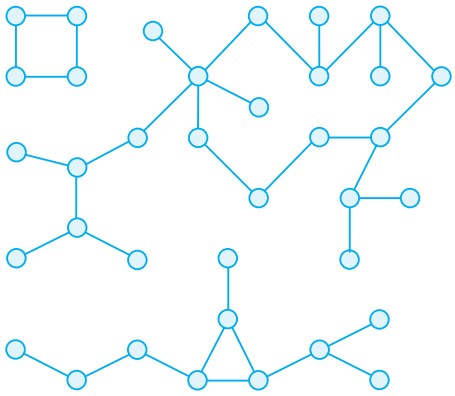
5. **To test if a graph is bipartite :** We can augment either BFS or DFS when we first discover a new vertex, color it opposited its parents, and for each other edge, check it doesn't link two vertices of the same color. The first vertex in any connected component can be red or black.

6. **Finding Strongly Connected Components of a graph :** A directed graph is called strongly connected if there is a path from each vertex in the graph to every other vertex.

7. **Solving puzzles with only one solution**, such as mazes. (DFS can be adapted to find all solutions to a maze by only including nodes on the current path in the visited set.)

8. **For finding Connected Components :** In graph theory, a connected component (or just component) of an undirected graph is a subgraph in which any two vertices are connected to each other by paths, and which is connected to no additional vertices in the supergraph.

Example **:**



In the above picture, there are three connected components.

Code for finding connected components :

```
#include<iostream>
#include<map>
#include<list>
using namespace std;

template<typename T>
class Graph{
     map<T,list<T> > adjList;
public:
     Graph(){
     }
     void addEdge(T u, T v,bool bidir=true){
            adjList[u].push_back(v);
            if(bidir){
                    adjList[v].push_back(u);
            }
      }
       voidprint(){
           //Iterate over the map
           for(auto i:adjList){
                 cout<<i.first<<"->";
                 //i.second is LL
```

```
            for(T entry:i.second){
                  cout<<entry<<“,”;
            }
            cout<<endl;
      }
}
void dfsHelper(T node,map<T,bool> &visited){
      //Whenever to come to a node, mark it visited
      visited[node] = true;
      cout<<node<<“ “;
      //Try to find out a node which is neigbour of current node and not yet visited
      for(T neighbour: adjList[node]){
            if(!visited[neighbour]){
                  dfsHelper(neighbour,visited);
            }
      }
}
void dfs(T src){
      map<T,bool> visited;
      int component = 1;
      dfsHelper(src,visited);
      cout<<endl;
      for(auto i:adjList){
            T city = i.first;
            if(!visited[city]){
                  dfsHelper(city,visited);
                  component++;
            }
      }
cout<<endl;
            cout<<“The current graph had “<<component<<“ components”;
      }
};
int main(){
      Graph<string> g;
```

```
      g.addEdge(“Amritsar”,”Jaipur”);
      g.addEdge(“Amritsar”,”Delhi”);
      g.addEdge(“Delhi”,”Jaipur”);
      g.addEdge(“Mumbai”,”Jaipur”);
      g.addEdge(“Mumbai”,”Bhopal”);
      g.addEdge(“Delhi”,”Bhopal”);
      g.addEdge(“Mumbai”,”Bangalore”);
      g.addEdge(“Agra”,”Delhi”);
      g.addEdge(“Andaman”,”Nicobar”);
      g.dfs(“Amritsar”);
return 0;
}
```

Minimum Spanning Tree

What is a Spanning Tree?

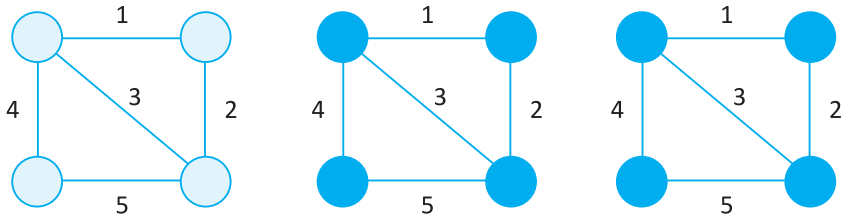
Given an undirected and connected graph G=(V,E) , a spanning tree of the graph G is a tree that spans G (that is, it includes every vertex of G) and is a subgraph of G (every edge in the tree belongs to G)

What is a Minimum Spanning Tree?

The cost of the spanning tree is the sum of the weights of all the edges in the tree. There can be many spanning trees. Minimum spanning tree is the spanning tree where the cost is minimum among all the spanning trees. There also can be many minimum spanning trees.

Minimum spanning tree has direct application in the design of networks. It is used in algorithms approximating the travelling salesman problem, multi-terminal minimum cut problem and minimum-cost weighted perfect matching. Other practical applications are:

1. Cluster Analysis
2. Handwriting recognition
3. Image segmentation



Undirected Graph

Spanning Tree

Cost = 11(= 4 + 5 + 2)

Minimum Spanning Tree

Cost = 7(= 4 + 1 + 2)

There are two famous algorithms for finding the Minimum Spanning Tree:

Kruskal's Algorithm

Kruskal's Algorithm builds the spanning tree by adding edges one by one into a growing spanning tree. Kruskal's algorithm follows greedy approach as in each iteration it finds an edge which has least weight and add it to the growing spanning tree.

Algorithm Steps

- Sort the graph edges with respect to their weights.
- Start adding edges to the MST from the edge with the smallest weight until the edge of the largest weight.
- Only add edges which doesn't form a cycle , edges which connect only disconnected components.

So now the question is how to check if 2 vertices are connected or not ?

This could be done using DFS which starts from the first vertex, then check if the second vertex is visited or not. But DFS will make time complexity large as it has an order of O(V+E) where V is the number of vertices, E is the number of edges. So the best solution is **"Disjoint Sets"**:

Disjoint sets are sets whose intersection is the empty set so it means that they don't have any element in common.

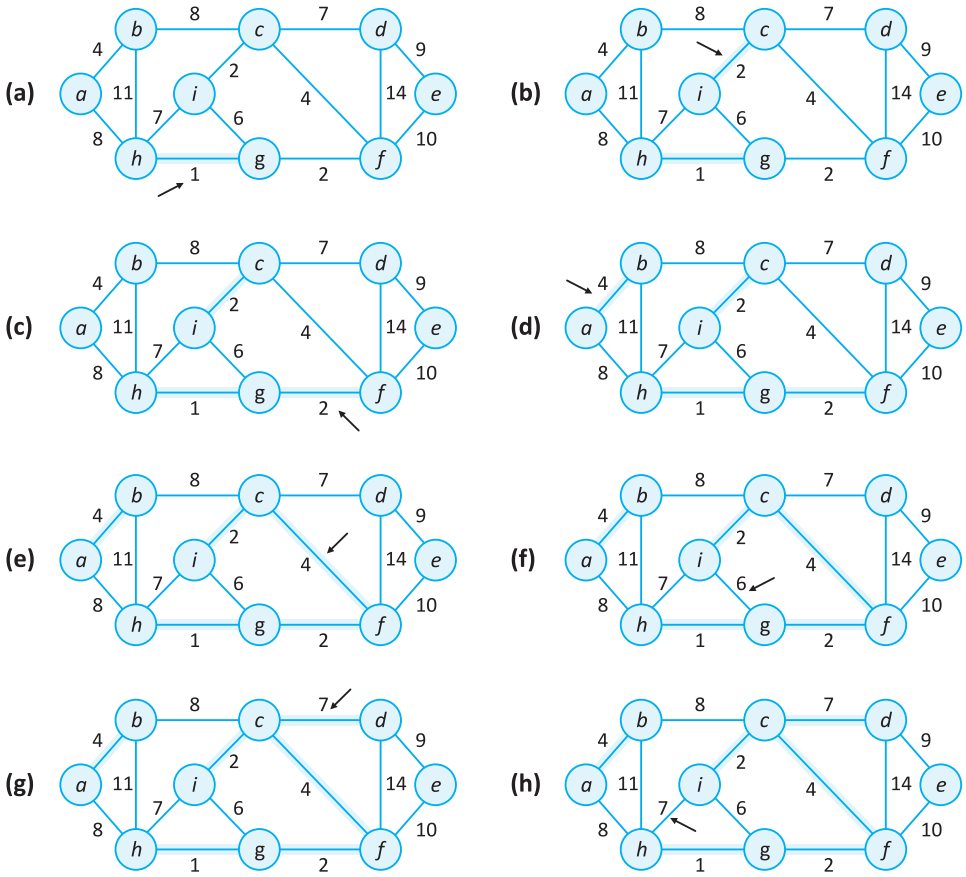
In Kruskal's algorithm, at each iteration we will select the edge with the lowest weight. So, we will start with the lowest weighted edge first.

Note **:** Kruskal's algorithm is a greedy algorithm, because at each step it adds to the forest an edge of least possible weight.

Algorithm :

```
MST-KRUSKAL(G, w)
A ← 0
For each vertex v ∈ V[G]
    Do MAKE-SET(v)
sort the edges of E into nondecreasing order by weight w
for each edge (u, v) ∈ E, taken in nondecreasing order by weight
    do if FIND-SET(u)≠ FIND-SET(v)
          then A ← A ∪ {(u, v)}
               UNION(u, v)
return A
```

Here **A** is the set which contains all the edges of minimum spanning tree.



The execution of Kruskal's algorithm on the graph from figure. Shaded edges belong to the forest A being grown. The edges are considered by the algorithm in sorted order by weight. An arrow points to the edge consideration at each step of the algorithm. If the edge joins two distinct trees in the forest, it is added to the forest, thereby merging the two trees.

Code :

```
#include <iostream>
#include <vector>
#include <utility>
#include <algorithm>

using namespace std;
const int MAX = 1e4 + 5;
int id[MAX], nodes, edges;
pair <long long, pair<int, int> > p[MAX];

void initialize()
{
    for(int i = 0;i < MAX;++i)
```

```
        id[i] = i;
}

int root(int x)
{
    while(id[x] != x)
    {
        id[x] = id[id[x]];
        x = id[x];
    }
    return x;
}

void union1(int x, int y)
{
    int p = root(x);
    int q = root(y);
    id[p] = id[q];
}

long long kruskal(pair<long long, pair<int, int> > p[])
{
    int x, y;
    long long cost, minimumCost = 0;
    for(int i = 0;i < edges;++i)
    {
        // Selecting edges one by one in increasing order from the beginning
        x = p[i].second.first;
        y = p[i].second.second;
        cost = p[i].first;
        // Check if the selected edge is creating a cycle or not
        if(root(x) != root(y))
        {
            minimumCost += cost;
            union1(x, y);
        }
```

```
    }
    return minimumCost;
}

int main()
{
    int x, y;
    long long weight, cost, minimumCost;
    initialize();
    cin >> nodes >> edges;
    for(int i = 0;i < edges;++i)
    {
        cin >> x >> y >> weight;
        p[i] = make_pair(weight, make_pair(x, y));
    }
    // Sort the edges in the ascending order
    sort(p, p + edges);
    minimumCost = kruskal(p);
    cout << minimumCost << endl;
    return 0;
}
```

Time Complexity :

The total running time complexity of kruskal algorithm is O(V log E).

Prim's Algorithm :

Prim's Algorithm also use Greedy approach to find the minimum spanning tree. In Prim's Algorithm we grow the spanning tree from a starting position. Unlike an edge in Kruskal's, we add vertex to the growing spanning tree in Prim's.

Algorithm Steps :

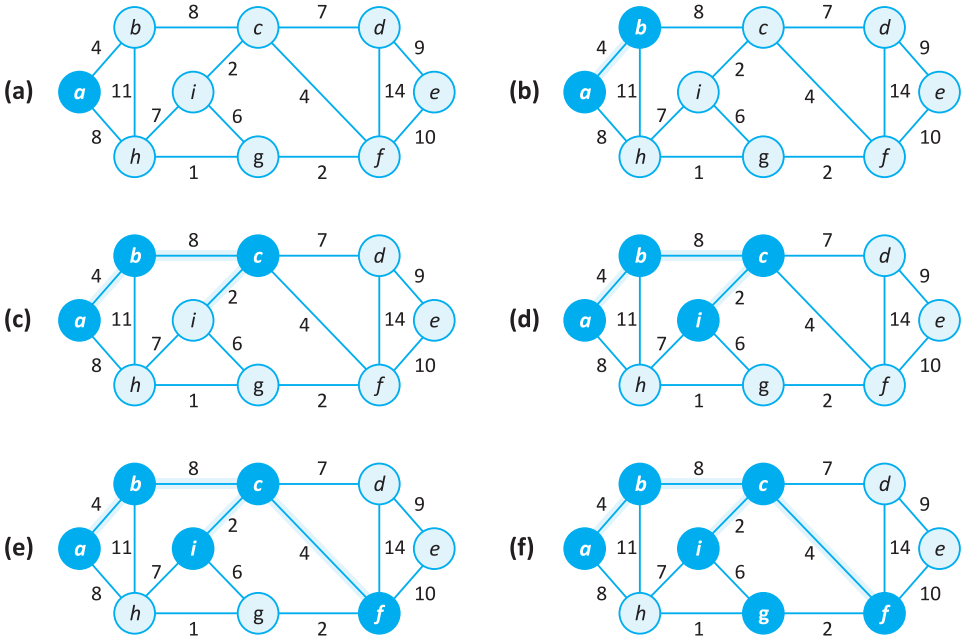
- Maintain two disjoint sets of vertices. One containing vertices that are in the growing spanning tree and other that are not in the growing spanning tree.
- Select the cheapest vertex that is connected to the growing spanning tree and is not in the growing spanning tree and add it into the growing spanning tree. This can be done using Priority Queues. Insert the vertices, that are connected to growing spanning tree, into the Priority Queue.

- Check for cycles. To do that, mark the nodes which have been already selected and insert only those nodes in the Priority Queue that are not marked.

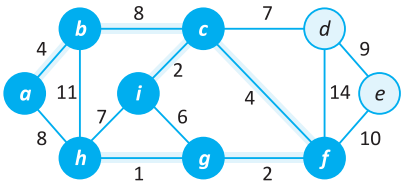
  In Prim's Algorithm, we will start with an arbitrary node (it doesn't matter which one) and mark it. In each iteration we will mark a new vertex that is adjacent to the one that we have already marked. As a greedy algorithm, Prim's algorithm will select the cheapest edge and mark the vertex.

```
MST-PRISM(G, w, r)
   for each u ∈ V[G]
        do key[u] ← ∞
           p[u] ← NIL
   key[r] ← 0
   Q ← V[G]
   while Q ≠ 0
       do u ← EXTRACT-MIN(Q)
          for each v ∈ Adj[u]
              do if v ∈ Q and w(u, v) < key[v]
                    then π[v] ← u
                         key[v] ← w(u, v)
```

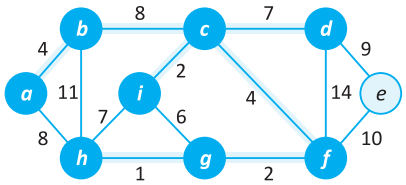
The prims algorithm works as shown in below graph.



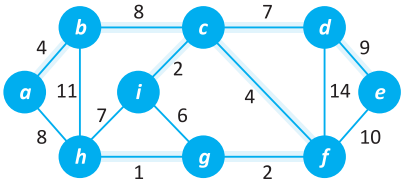
(g)



(h)



(i)



Code :

```
#include <iostream>
#include <vector>
#include <queue>
#include <functional>
#include <utility>
using namespace std;
const int MAX = 1e4 + 5;
typedef pair<long long, int> PII;
bool marked[MAX];
vector <PII> adj[MAX];

long long prim(int x)
{
    priority_queue<PII, vector<PII>, greater<PII> > Q;
   int y;
   long long minimumCost = 0;
   PII p;
   Q.push(make_pair(0, x));
   while(!Q.empty())
   {
        // Select the edge with minimum weight
        p = Q.top();
        Q.pop();
        x = p.second;
        // Checking for cycle
```

```
        if(marked[x] == true)
            continue;
        minimumCost += p.first;
        marked[x] = true;
        for(int i = 0;i < adj[x].size();++i)
       {
            y = adj[x][i].second;
            if(marked[y] == false)
                Q.push(adj[x][i]);
       }
   }
    return minimumCost;
}
int main()
{
   int nodes, edges, x, y;
    long long weight, minimumCost;
   cin >> nodes >> edges;
   for(int i = 0;i < edges;++i)
   {
        cin >> x >> y >> weight;
         adj[x].push_back(make_pair(weight, y));
         adj[y].push_back(make_pair(weight, x));
   }
   // Selecting 1 as the starting node
   minimumCost = prim(1);
   cout << minimumCost << endl;
   return 0;
}
```

Shortest Path Algorithms

The shortest path problem is about finding a path between 2 vertices in a graph such that the total sum of the edges weights is minimum.

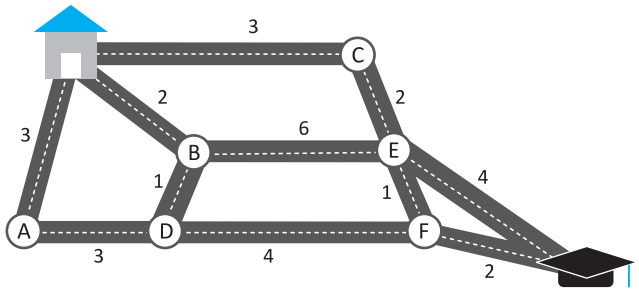
This problem could be solved easily using (BFS) if all edge weights were (1), but here weights can take any value. There are some algorithm discussed below which work to find Shortest path between two vertices.

1. Single Source Shortest Path Algorithm :

In this kind of problem, we need to find the shortest path of single vertices to all other vertices.

Example **:**

Find the shortest path from home to school in the following graph:



A weighted graph representing roads from home to school

The shortest path, which could be fund using Dijkstra's algorithm, is

Home → B → D → F → School

There are two algorithm works to find Single source shortest path from a given graph.

A. Dijkstra's Algorithm

Dijkstra's algorithm solves the single-source shortest-paths problem on a weighted, directed graph $G = (V, E)$ for the case in which all edge weights are nonnegative.

Algorithm Steps:

- Set all vertices distances = infinity except for the source vertex, set the source distance = 0.
- Push the source vertex in a min-priority queue in the form (distance , vertex), as the comparison in the min-priority queue will be according to vertices distances.
- Pop the vertex with the minimum distance from the priority queue (at first the popped vertex = source).
- Update the distances of the connected vertices to the popped vertex in case of "current vertex distance + edge weight < next vertex distance", then push the vertex with the new distance to the priority queue.
- If the popped vertex is visited before, just continue without using it.
- Apply the same algorithm again until the priority queue is empty.

Pseudo Code :

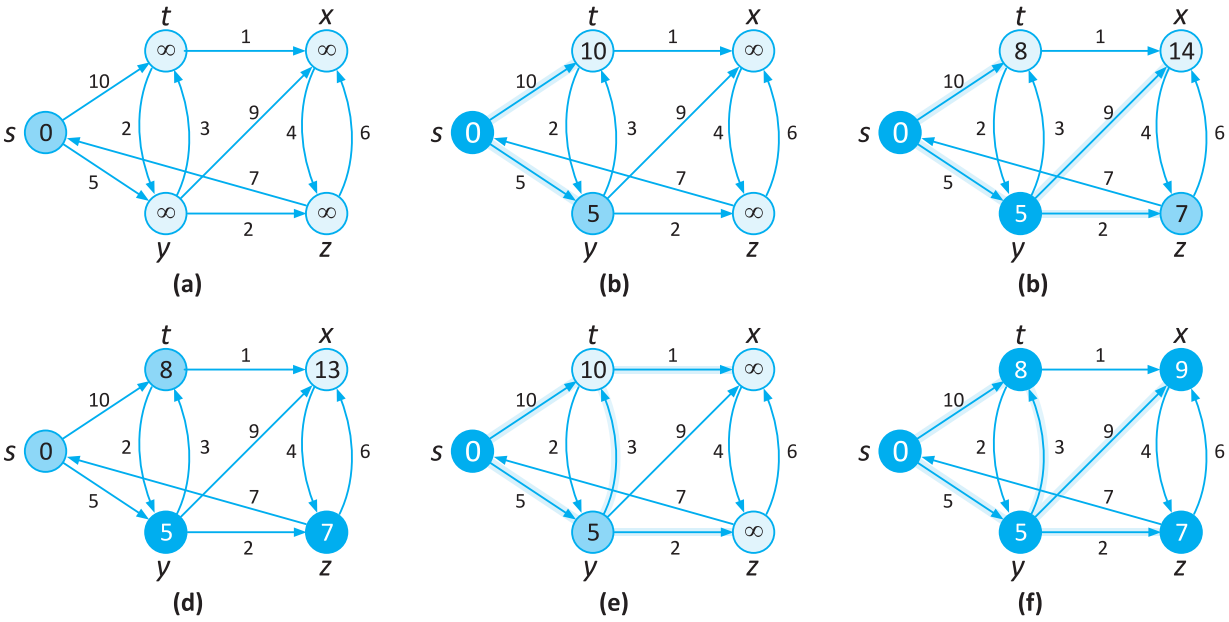
```
DIJKSTRA(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
S ← 0
Q ← V[G]
while Q ≠ 0
```

```
    do u ← EXTRACT-MIN(Q)
       S ← S ∪ {u}
       for each vertex v ∈ Adj[u]
           do RELAX(u, v, w)
```

For relaxing an edge of given graph, Algorithm works likes this :

```
RELAX(u, v, w)
   if d[v] > d[u] + w(u, v)
      then d[v] ← d[u] + w(u, v)
           π[v] ← u
```

Example **:**



The execution of Dijkstra's algorithm. The source *s* is the leftmost vertex. The shortest-path estimates are shown within the vertices, and shaded edges indicate predecessor values. Black vertices are in the set *S*, and white vertices are in the min-priority queue $Q = V - S$. **(a)** The situation just before the first iteration of the **while** loop. The shaded vertex has the minimum *d* value and is chosen as vertex *u* . **(b)–(f)** The situation after each successive iteration of the **while** loop. The shaded vertex in each part is chosen as vertex *u* in line 5 of the next iteration. The *d* and À values shown in part (f) are the final values.

Code :

```
#include<bits/stdc++.h>
using namespace std;
template<typename T>
class Graph{
    unordered_map<T, list<pair<T,int> > > m;
```

```
public:
    void addEdge(T u,T v,int dist,bool bidir=true){
         m[u].push_back(make_pair(v,dist));
        if(bidir){
             m[v].push_back(make_pair(u,dist));
        }
    }
    void printAdj(){
        //Let try to print the adj list
        //Iterate over all the key value pairs in the map
        for(auto j:m){
             cout<<j.first<<"->";
             //Iterater over the list of cities
             for(auto l: j.second){
                      cout<<"("<<l.first<<","<<l.second<<")";
            }
             cout<<endl;
        }
    }
    void dijsktraSSSP(T src){
         unordered_map<T,int> dist;
        //Set all distance to infinity
        for(auto j:m){
             dist[j.first] = INT_MAX;
        }
       //Make a set to find a out node with the minimum distance
         set<pair<int, T> > s;
       dist[src] = 0;
         s.insert(make_pair(0,src));
        while(!s.empty()){
            //Find the pair at the front.
            auto p =   *(s.begin());
            T node = p.second;
            int nodeDist = p.first;
            s.erase(s.begin());
```

```
            //Iterate over neighbours/children of the current node
             for(auto childPair: m[node]){
                  if(nodeDist + childPair.second < dist[childPair.first]){
                      //In the set updation of a particular is not possible
                       // we have to remove the old pair, and insert the new pair
to   simulation updation
                      T dest = childPair.first;
                       auto f = s.find( make_pair(dist[dest],dest));
                      if(f!=s.end()){
                          s.erase(f);
                      }
                       //Insert the new pair
                        dist[dest] = nodeDist + childPair.second;
                        s.insert(make_pair(dist[dest],dest));
                  }
              }
        }
         //Lets print distance to all other node from src
         for(auto d:dist){
              cout<<d.first<<" is located at distance of  "<<d.second<<endl;
        }
    }
};
int  main(){
    Graph<int> g;
     g.addEdge(1,2,1);
     g.addEdge(1,3,4);
     g.addEdge(2,3,1);
     g.addEdge(3,4,2);
     g.addEdge(1,4,7);
     //g.printAdj();
    // g.dijsktraSSSP(1);
     Graph<string> india;
      india.addEdge("Amritsar","Delhi",1);
      india.addEdge("Amritsar","Jaipur",4);
      india.addEdge("Jaipur","Delhi",2);
```

```
    india.addEdge("Jaipur","Mumbai",8);
    india.addEdge("Bhopal","Agra",2);
    india.addEdge("Mumbai","Bhopal",3);
    india.addEdge("Agra","Delhi",1);
   //india.printAdj();
    india.dijsktraSSSP("Amritsar");


return 0;
}
```

Time Complexity : Time Complexity of Dijkstra's Algorithm is $O(V^2)$ but with min-priority queue it drops down to O(V + E logV).

B. Bellman Ford's Algorithm

Bellman Ford's algorithm is used to find the shortest paths from the source vertex to all other vertices in a weighted graph. It depends on the following concept: Shortest path contains at most n – 1 edges, because the shortest path couldn't have a cycle.

So why shortest path shouldn't have a cycle ?

There is no need to pass a vertex again, because the shortest path to all other vertices could be found without the need for a second visit for any vertices.

Algorithm Steps:

- The outer loop traverses from 0 to n – 1.
- Loop over all edges, check if the next node distance > current node distance + edge weight, in this case update the next node distance to "current node distance + edge weight".

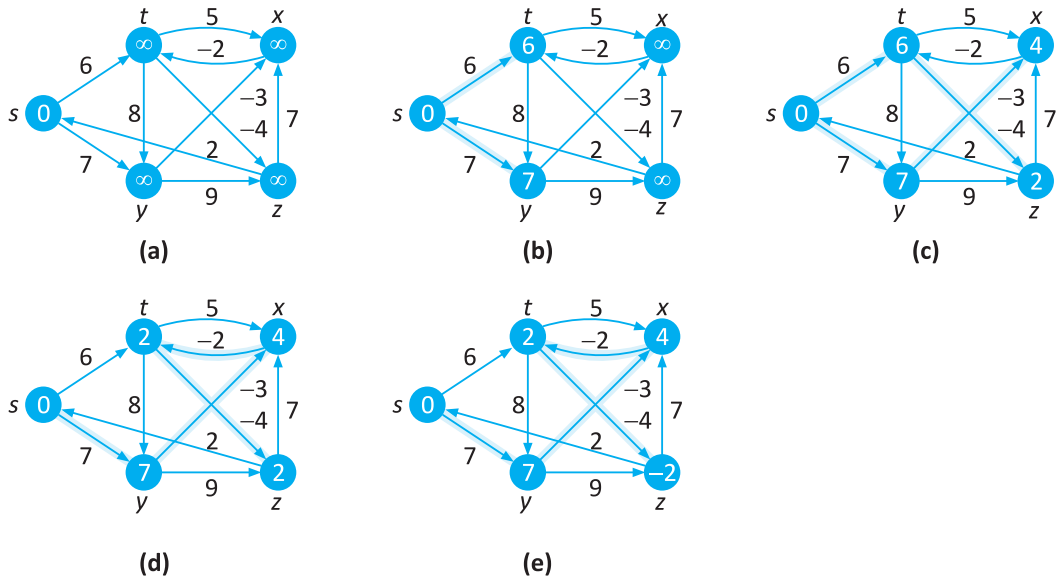
This algorithm depends on the relaxation principle where the shortest distance for all vertices is gradually replaced by more accurate values until eventually reaching the optimum solution. In the beginning all vertices have a distance of "Infinity", but only the distance of the source vertex = 0, then update all the connected vertices with the new distances (source vertex distance + edge weights), then apply the same concept for the new vertices with new distances and so on.

Pseudo Code :

```
BELLMAN-FORD(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
for i ← 1 to |V[G]| - 1
      do for each edge (u, v) ∈ E[G]
             do RELAX(u, v, w)
for each edge (u, v) ∈ E[G]
```

```
      do if d[v] > d[u] + w(u, v)
            then return FALSE
return TRUE
```

Example :



All-Pairs Shortest Paths :

The all-pairs shortest path problem is the determination of the shortest graph distances between every pair of vertices in a given graph. The problem can be solved using n applications of Dijkstra's algorithm at each vertex if graph doesn't contains negative weight. We use two algorithms for finding All-pair shortest path of a graph.

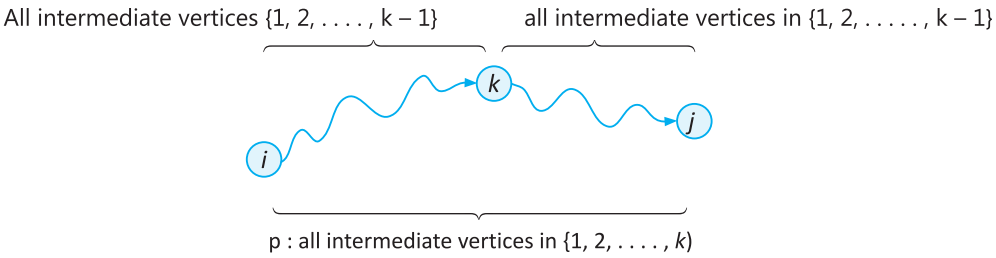
1. Floyd-Warshall's Algorithm
2. Johnson's Algorithm

Floyd-Warshall's Algorithm

Here we use a different dynamic-programming formulation to solve the all-pairs shortest-paths problem on a directed graph $G = (V, E)$. The resulting algorithm, known as the ***Floyd-Warshall algorithm***. Floyd–Warshall's Algorithm is used to find the shortest paths between between all pairs of vertices in a graph, where each edge in the graph has a weight which is positive or negative. The biggest advantage of using this algorithm is that all the shortest distances between any 2 vertices could be calculated in $O(V^3)$, where V is the number of vertices in a graph.

The Floyd-Warshall algorithm is based on the following observation. Under our assumption that the vertices of G are $V = \{1, 2, \ldots, n\}$, let us consider a subset $\{1,2,\ldots,k\}$ of vertices for some k. For any pair of vertices i, $j - V$, consider all paths from i to j whose intermediate vertices are all drawn from $\{1, 2, \ldots, k\}$, and let p be a minimum-weight path from among them. The Floyd- Warshall algorithm exploits a relationship between path p and shortest paths from i to j with all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$. The relationship depends on whether or not k is an intermediate vertex of path p. There would two possibilities :

1. If *k* is not an intermediate vertex of path *p*, then all intermediate vertices of path *p* are in the set {1,2,...,*k* – 1}. Thus, a shortest path from vertex *i* to vertex *j* with all intermediate vertices in the set {1,2,...,*k* –1} is also a shortest path from *i* to *j* with all intermediate vertices in the set {1, 2, . . . , *k*}.
2. If *k* is an intermediate vertex of path *p*, then we break *p* down into *i* -> *k* -> *j* as *p*1 is a shortest path from *i* to *k and p*2 is a shortest path from *k* to *j. Since p*1 is a shortest path from *i* to *k* with all intermediate vertices in the set {1, 2, . . . , *k*}. Because vertex *k* is not an intermediate vertex of path *p*1, we see that *p*1 is a shortest path from *i* to *k* with all intermediate vertices in the set {1, 2, . . . , *k* – 1}. Similarly, *p*2 is a shortest path from vertex *k* to vertex *j* with all intermediate vertices in the set {1,2,...,*k* – 1}.



Path P is a shortest path from vertices *i* to vertex j, and k is the highest=numbered intermediate vertex of p. Path p_1, the portion of path p from vertex i to vertex k, has all intermediate vertices in the set {1, 2, ... , k – 1}. The same holds for path p_2 from vertex k to vertex j.

Let $d_{ij}^{(k)}$ be the weight of shortest path from vertex *i* to vertex *j* for which all intermediate vertices are in the set {1, 2, . . . , *k*}. A recursive definition following the above discussion is given by

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if} \quad k = 0 \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if} \quad k \geq 1 \end{cases}$$

The Algorithm Steps:

For a graph with V vertices:

- Initialize the shortest paths between any 2 vertices with Infinity.
- Find all pair shortest paths that use 0 intermediate vertices, then find the shortest paths that use 1 intermediate vertex and so on.. until using all V vertices as intermediate nodes.
- Minimize the shortest paths between any 2 pairs in the previous operation.
- For any 2 vertices (i , j) , one should actually minimize the distances between this pair using the first K nodes, so the shortest path will be:

  Min (dist[i][k] + dist[k][j] , dist[i][j])

dist[i][k] represents the shortest path that only uses the first K vertices, dist[k][j] represents the shortest path between the pair k , j. As the shortest path will be a concatenation of the shortest path from i to k, then from k to j.

Constructing a shortest path

For construction of the shortest path, we can use the concept of predecessor matrix π to construct the path. We can compute the predecessor matrix π "on-line" just as the Floyd-Warshall algorithm computes the matrices $D(k)$. Specifically, we compute a sequence of matrices $\pi^{(k)}$ where $\pi^{(k)}$ is defined to be the predecessor of vertex j on a shortest path from vertex i with all intermediate vertices in the set {1,2,...,k}. We can give a recursive formulation of $\pi^{(k)}$ as

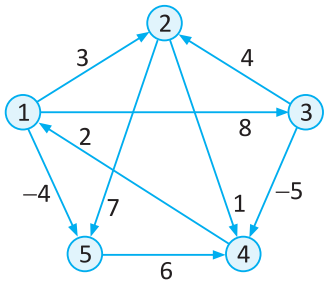
For k = 0 :

$$\pi_{ij}^{(0)} = \begin{cases} \text{NILL} & \text{if} \quad i = j \text{ or } w_{ij} = \infty \\ i & \text{if} \quad i \neq j \text{ and } w_{ij} < \infty \end{cases}$$

And For 1<=k<=V

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{if} \quad d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ i & \text{if} \quad d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases}$$

Consider the below graph and find distance (D) and parents (π) matrix for this graph



Computed distance (D) and parents (π) matrix for the above graph :

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(3)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(2)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(3)} = \begin{pmatrix} NIL & 1 & 1 & 2 & 1 \\ NIL & NIL & NIL & 2 & 2 \\ NIL & 3 & NIL & 2 & 2 \\ 4 & 3 & 4 & NIL & 1 \\ NIL & NIL & NIL & 5 & NIL \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(4)} = \begin{pmatrix} NIL & 1 & 4 & 2 & 1 \\ 4 & NIL & 4 & 2 & 1 \\ 4 & 3 & NIL & 2 & 1 \\ 4 & 3 & 4 & NIL & 1 \\ 4 & 3 & 4 & 5 & NIL \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(5)} = \begin{pmatrix} NIL & 3 & 4 & 5 & 1 \\ 4 & NIL & 4 & 2 & 1 \\ 4 & 3 & NIL & 2 & 1 \\ 4 & 3 & 4 & NIL & 1 \\ 4 & 3 & 4 & 5 & NIL \end{pmatrix}$$

Implemented Code :

```
#include<bits/stdc++.h>
#include<iostream>
#include<unordered_map>
#define INF 99999
#define V 5
using namespace std;
template<typename T>
class Graph
{
   unordered_map<T, list<pair<T, int> > > m;
   int graph[V][V];
   int parent[V][V];

public:
 //Adjacency list representation of the graph
 void addEdge(T u, T v, int dist,bool bidir = false)
 {
      m[u].push_back(make_pair(v,dist));
     /*if(bidir){
```

```
            m[v].push_back(make_pair(u,dist));
        }*/

    }
    //Print Adjacency list
    void printAdj()
    {
        for(auto j:m)
        {
            cout<<j.first<<"->";
            for(auto l: j.second)
            {
                cout<<"("<<l.first<<","<<l.second<<")";
            }
            cout<<endl;
        }
    }
    void matrix_form(int u, int v , int w)
    {
       graph[u-1][v-1] = w;
       parent[u-1][v-1] = u;
       return;
    }

//Adjacency matrix representation of the graph
    void matrix_form2()
    {
        for(int i=0;i<V;i++)
        {
            for(int j=0;j<V;j++)
            {
                if(i==j)
                {
                    graph[i][j]=0;
                    parent[i][j]=0;
                }
```

```
            else
            {
                graph[i][j]=INF;
                parent[i][j]=0;
            }
        }
    }
    return;
}
//Print Adjacency matrix
 void print_matrix()
{
    for(int i=0;i<V;i++)
    {
        for(int j=0;j<V;j++)
        {
            if(graph[i][j]==INF)
            cout<<"INF"<<"   ";
            else
            cout<<graph[i][j]<<"    ";
        }
        cout<<endl;
    }

}

//Print predecessor matrix
 void printParents(int p[][V])
{

    cout<<"Parents Matrix"<<"\n";
  for (int i = 0; i < V; i++)
  {
      for (int j = 0; j < V; j++)
      {
          if(p[i][j]!=0)
```

```
            cout<<p[i][j]<<"   ";
           else
           {
            cout<<"NIL"<<" ";
            //cout<<p[i][j]<<"   ";
           }
       }
         cout<<endl;
   }
 }
 //All pair shortest path matrix i.e D
 void printSolution(int dist[][V])
{

             cout<<"Following matrix shows the shortest distances"
             " between every pair of vertices"<<"\n";
   for (int i = 0; i < V; i++)
   {
        for (int j = 0; j < V; j++)
        {
             if (dist[i][j] == INF)
                 cout<<"INF   ";
            else
                 cout<<dist[i][j]<<"   ";
        }
         cout<<endl;
   }
}
 //Print the shortest path , distance and all intermediate vertex
 void print_path(int p[][V], int d[][V])
{
  // cout<<"Hello"<<"\n";
   for(int i=0;i<V;i++)
   {
        for(int j=0;j<V;j++)
        {
```

```
            // cout<<“Hello1”<<“\n”;
            if(i!=j)
            {
             cout<<“Shortest path from “<<i+1<<“ to “<< j+1<<“ => “;
             cout<<“[Total Distance : “<<d[i][j]<<“ ( Path : “;
             //cout<<“Hello1”<<“\n”;
            int k=j;
            int l=0;
            int a[V];
            a[l++] = j+1;
             while(p[i][k]!=i+1)
            {
                 //cout<<“Hello1”<<“\n”;
                 a[l++]=p[i][k];
                k=p[i][k]-1;
            }
            a[l]=i+1;
             //cout<<“Hello1”<<“\n”;
             for(int r =l;r>0;r–)
            {
                 //cout<<“Hello1”<<“\n”;
              cout<<a[r]<<“ –—> “;
            }
             cout<<a[0]<<“ )”;
             cout<<endl;

            }
       }
   }
}
 //Floyd Warshall Algorithm
void floydWarshall ()
{
   int dist[V][V], i, j, k;
   int parent2[V][V];
   for (i = 0; i < V; i++)
```

```
        for (j = 0; j < V; j++)
             dist[i][j] = graph[i][j];

    for(int i=0;i<V;i++)
    {
         for(int j=0;j<V;j++)
        {
             parent2[i][j] = parent[i][j];
        }
    }

    for (k = 0; k < V; k++)
    {
         for (i = 0; i < V; i++)
        {
             for (j = 0; j < V; j++)
            {
                 if (dist[i][k] + dist[k][j] < dist[i][j])
                {
                     dist[i][j] = dist[i][k] + dist[k][j];
                     parent2[i][j] = parent2[k][j];
                }
            }
        }
    }

    printSolution(dist);
    cout<<"\n\n";
    printParents(parent2);
    cout<<"\n\n";
    cout<<"All pair shortest path is given below :"<<"\n";
    print_path(parent2, dist);
}
};
//Main Function
int main()
```

```
{
   Graph<int> g;
   g.matrix_form2();
  // g.make_parent();
 //Intializing the graph given in above picture
   g.addEdge(1,2,3);
   g.matrix_form(1,2,3);
   g.addEdge(1,3,8);
   g.matrix_form(1,3,8);
   g.addEdge(1,5,-4);
   g.matrix_form(1,5,-4);
   g.addEdge(2,4,1);
   g.matrix_form(2,4,1);
   g.addEdge(2,5,7);
   g.matrix_form(2,5,7);
   g.addEdge(3,2,4);
   g.matrix_form(3,2,4);
   g.addEdge(4,3,-5);
   g.matrix_form(4,3,-5);
   g.addEdge(4,1,2);
   g.matrix_form(4,1,2);
   g.addEdge(5,4,6);
   g.matrix_form(5,4,6);
   cout<<“Graph in the form of adjecency list representation : “<<“\n”;
   g.printAdj();
   cout<<“Graph in the form of matrix representation : “<<“\n”;
   g.print_matrix();
   cout<<“\n\n”;
   g.floydWarshall();

}
```

DO IT YOURSELVES

- ❑ Implement a BFS based algorithm to find the shortest route in **Snakes and Laddders Game**
- ❑ Implement a program to classify each of the graph as **forward edge, back edge** or **cross edge.**
- ❑ Find the shortest path in a weighed graph where each egde is 1 or 2.
- ❑ Read about **Hamiltonian Cycle**. Implement an optimisied **Travelling Salesman Problem** solution using Dynamic Programming. [Refer Tutorial]
- ❑ Read about -
  Articulation Points
  Bridges
  Eulerian Paths and Circuits
- ❑ Read about **Strongly Connected Components** and its algorithms -
  - Kosaraju's algorithm
  - Tarjan's algorithm
- ❑ Solve **Holdiay Accomodation(HOLI**) on Spoj [Refer Tutorial]
- ❑ Implement a **Flood Fill Algorithm**, to color various components of a given pattern. Pattern boundary is represented by '#'
- ❑ Implement an algorithm for '**Splitwise App**', to minimize the cash flow between a group of friends.
- ❑ Solve the Graphs Assignment at **HackerBlocks**

SELF STUDY NOTES

SELF STUDY NOTES

12 Combinatorial Game Theory

Introduction :

Combinatorial games are two-person games with perfect information and no chance moves (no randomization like coin toss is involved that can affect the game). These games have a win-or-lose or tie outcome and determined by a set of positions, including an initial position, and the player whose turn it is to move. Play moves from one position to another, with the players usually alternating moves, until a terminal position is reached. A terminal position is one from which no moves are possible. Then one of the players is declared the winner and the other the loser. Or there is a tie (Depending on the rules of the combinatorial game, the game could end up in a tie. The only thing that can be stated about the combinatorial game is that the game should end at some point and should not be stuck in a loop.

Let's see an Example :

Consider a simple game which two players can play. There are N coins in a pile. In each turn, a player can choose to remove one or two coins.

The players keep alternating turns and whoever removes the last coin from the table wins.
If N=1 then?
If N=2 then?
If N=3 then?
If N=4 then?
If N=5 then?
If N=6 then?

Once you start playing this game, you will realise that it isn't much fun. You will soon be able to devise a strategy which will let you win for certain values of N. Eventually you will figure out that if both players play smartly, the winner of the game is decided entirely by the initial value of N.

Strategy :

Continuing in this fashion, we find that the first player is guaranteed a win unless N is a multiple of 3. The winning strategy is to just remove coins to make N a multiple of 3.

Finders Keepers game :

A and B playing a game in which a certain number of coins are placed on a table. Each player picks at least 'a' and atmost 'b' coins in his turn unless there is less than 'a' coins left in which case the player has to pick all those left.

(a) Finders-Winners : In this format, the person who picks the last coin wins. Strategy is to reduce opponent to a state containing $(a + b) \times k$ coins which is a loosing state for opponent.

(b) Keepers-Losers : In this format, the person who picks last coin loses. Strategy is to reduce opponent to a state containing $(a + b) \times k + x$ coins (x lies between 1 and a, both inclusive) which is a loosing state for opponent.

PROBLEMS

1. A and B play game of finder-winners with a = 2 and b = 6. If A starts the game and there are 74 coins on the table initially, how many should A pick?

Sol: If A picks 2, B will be left with 72 and no matter what number B picks, A can always pick (8 – x) and wrap up.

2. In a game of keepers-losers B started the game when there were N coins on the table. If B is confident of winning the game and a = 3, b = 5, which of the following cannot be the value of N?

(a) 94 (b) 92 (c) 76 (d) 66

Sol: (d) In keeper-losers, the motto is to give opponent 8k + 1 coins to win for sure. With 66 coins, no matter what B does, he cannot give 65 coins to A.

3. A and B play Finders-winners with 56 coins, where A plays first anad a = 1, b = 6. what should B pick immediately after A?

(a) 2 (b) 3 (c) 4 (d) Can't be determined

Sol: (d) A will lose in any case. But the number of coins B picks depends on what A picks.

4. In a game of Keepers-losers played with 126 coins where A plays first ans a = 3, b = 6, who is the winner?

Sol: In order to win, A should leave 9k+1, 9k + 2 or 9k + 3 coins on the table, since he can do by picking 6 coins and hence can win the game.

5. In a interesting version of game B gets to choose the number of coins on the table and A gets to choose the format of the game it will be as well as pick coins first. If B chooses 144 and a = 1, b = 5 which format should A choose in order to win?

Sol: A should choose Keepers-losers and pick 5 coins from the table, leaving 139 coins for B

Properties of the above Games :

1. The games are sequential. The players take turns one after the other, and there is no passing.
2. The games are impartial. Given a state of the game, the set of available moves does not depend on whether you are player 1 or player.
3. Chess is a partisan game(moves are not same).
4. Both players have perfect information about the game. There is no secrecy involved.
5. The games are guaranteed to end in a finite number of moves.
6. In the end, the player unable to make a move loses. There are no draws. (This is known as a normal game. If on the other hand the last player to move loses, it is called a misère game)

Impartial games can be solved using Sprague-Grundy theorem which reduces every such game to Game of NIM.

Game of NIM

Given a number of piles in which each pile contains some numbers of stones/coins. In each turn, a player can choose only one pile and remove any number of stones (at least one) from that pile. The player who cannot move is considered to lose the game (i.e., one who take the last stone is the winner).

SOLUTION :

Let n_1, n_2, ...n_k, be the sizes of the piles. It is a losing position for the player whose turn it is if and only if n_1 x or n_2 x or ... x or n_k = 0.

Nim-Sum : The cumulative XOR value of the number of coins/stones in each piles/heaps at any point of the game is called Nim-Sum at that point.

If both A and B play optimally (i.e. they don't make any mistakes), then the player starting first is guaranteed to win if the Nim-Sum at the beginning of the game is non-zero. Otherwise, if the Nim-Sum evaluates to zero, then player A will lose definitely.

Why does it work?

From the losing positions we can move only to the winning ones :

- if x or of the sizes of the piles is 0 then it will be changed after our move (at least one 1 will be changed to 0, so in that column will be odd number of 1s).
  From the winning positions it is possible to move to at least one losing:
- if xor of the sizes of the piles is not 0 we can change it to 0 by finding the left most column where the number of 1s is odd, changing one of them to 0 and then by changing 0s or 1s on the right side of it to gain even number of 1s in every column.

From a balanced state whatever we do, we always end up in unbalanced state. And from an unbalanced state we can always end up in atleast one balanced state.

Now, the pile size is called the nimber/Grundy number of the state.

Sprague-Grundy Theorem

A game composed of K solvable subgames with Grundy numbers G_1, G_2 ... G_K is winnable iff the Nim game composed of Nim piles with sizes G1, G2 ... GK is winnable.

So, to apply the theorem on arbitrary solvable games, we need to find the Grundy number associated with each game state. But before calculating Grundy Numbers, we need to learn about another term- Mex.

What is Mex Function ?

'**Minimum excludant**' a.k.a 'Mex' of a set of numbers is the smallest non-negative number not present in the set.

Eg S = {1, 2, 3, 5}

Mex(S) = 0

S1 = {0, 1, 3, 4, 5}

Mex(S1) = 2

Calculating Grundy Numbers

1. **The game starts with a pile of n stones, and the player to move may take any positive number of stones. Calculate the Grundy Numbers for this game. The last player to move wins. Which player wins the game?**

```
Grundy(0)=?
Grundy(1)=?
Grundy(n) = Mex (0, 1, 2, ....n-1) = n
int calculateMex(set<int> Set)
{
    int Mex = 0;
    while (Set.find(Mex) != Set.end())
        Mex++;
    return (Mex);
}
// A function to Compute Grundy Number of 'n'
```

```
// Only this function varies according to the game
int calculateGrundy(int n)
{
    if (n == 0)
        return (0);
    set<int> Set; // A Hash Table
    for (int i=1; i<=n; i++)
        Set.insert(calculateGrundy(n-i));
    return (calculateMex(Set));
}
```

2. **The game starts with a pile of n stones, and the player to move maytake any positive number of stones upto 3 only. The last player to move wins. Which player wins the game? This game is 1 pile version of Nim.**

```
Grundy(0)=?
Grundy(1)=?
Grundy(2)=?
Grundy(3)=?
Grundy(4)=mex(Grundy(1),Grundy(2),Grundy(3))
```

```
int calculateMex(set<int> Set)
{
    int Mex = 0;
    while (Set.find(Mex) != Set.end())
        Mex++;
    return (Mex);
}

// A function to Compute Grundy Number of 'n'
// Only this function varies according to the game

int calculateGrundy(int n)
{
    if (n == 0)
```

```
        return (0);
    if (n == 1)
        return (1);
    if (n == 2)
        return (2);
    if (n == 3)
        return (3);

    set<int> Set; // A Hash Table
    for (int i=1; i<=3; i++)
        Set.insert(calculateGrundy(n - i));

    return (calculateMex(Set));
}
```

3. **The game starts with a number- 'n' and the player to move divides the number- 'n' with 2, 3 or 6 and then takes the floor. If the integer becomes 0, it is removed. The last player to move wins. Which player wins the game?**

```
int calculateGrundy (int n)
{
if (n == 0)
return (0);
set<int> Set; // A Hash Table
Set.insert(calculateGrundy(n/2));
Set.insert(calculateGrundy(n/3));
Set.insert(calculateGrundy(n/6));
return (calculateMex(Set));
}
```

How to apply Sprague Grundy theorem ?

1. Break the composite game into sub-games.
2. Then for each sub-game, calculate the Grundy Number at that position.
3. Then calculate the XOR of all the calculated Grundy Numbers.
4. If the XOR value is non-zero, then the player who is going to make the turn (First Player) will win else he is destined to lose, no matter what.

PROBLEMS

QCJ3

Tom and Hanks play the following game. On a game board having a line of squares labelled from 0,1,2 ... certain number of coins are placed with possibly more than one coin on a single square. In each turn a player can move exactly one coin to any square to the left i.e, if a player wishes to remove a coin from square i, he can then place it in any square which belongs to the set (0,1, ... i-1) . The game ends when all coins are on square 0 and player that makes the last move wins. Given the description of the squares and also assuming that Tom always makes the first move you have tell who wins the game (Assuming Both play Optimally).

(http://www.spoj.com/problems/QCJ3/)

Hint :

Each stone at position P, corresponds to heap of size P in NIM Now, if there are x stones at position n, then n is XORed x times because each stone corresponds to a heap size of n.

Then we use the property of xor operator

```
#include<iostream>
using namespace std;

int main()
{

     int n;
     cin>>n;

    while(n--)
    {

     int s;
     cin>>s;
     long r=0;
     int x;
     for(int i=1;i<=s;i++)
      {
            cin>>x;
           if(x&1)
                 r=r^i;
```

```
    }
    cout<<(r==0?"Hanks Wins":"Tom Wins")<<endl;

  }
  return 0;
  }
```

Try it yourself !

M&M Game
http://www.spoj.com/problems/MMMGAME/

Hint : This is a Miser Game.

Game of Stones
https://www.hackerrank.com/challenges/game-of-stones-1/problem

Piles Again
(Variation of Nim, HackerBlocks)

Game Theory-1
(Grundy Numbers, HackerBlocks)

Game Theory-2
(Sprague Grundy Thm, HackerBlocks)

Game Theory-3
(Sprague Grundy Thm, HackerBlocks)

MORE QUESTIONS FOR PRACTICE

1. http://www.spoj.com/problems/RESN04
2. http://www.spoj.com/problems/GAME3
3. http://www.spoj.com/problems/GAME31
4. http://www.spoj.com/problems/NGM
5. http://www.spoj.com/problems/NGM2
6. http://www.spoj.com/problems/NUMGAME
7. http://www.spoj.com/problems/NIMGAME

8. http://www.spoj.com/problems/MAIN73
9. http://www.spoj.com/problems/MATGAME
10. http://www.spoj.com/problems/MCOINS
11. http://www.spoj.com/problems/REMGAME
12. http://www.spoj.com/problems/TRIOMINO
13. http://www.spoj.com/problems/BOMBER/
14. http://www.codechef.com/problems/CHEFBRO/
15. http://www.codeforces.com/contest/256/problem/C

SELF STUDY NOTES

SELF STUDY NOTES

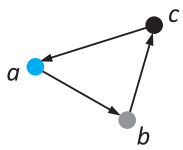
13 Geometric Algorithms

How to check if two line segments intersect?

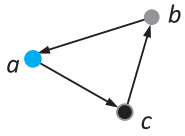
Given two line segments (p1, q1) and (p2, q2), find if the given line segments intersect with each other.

Orientation of an ordered triplet of points in the plane can be

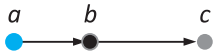
- counterclockwise
- clockwise
- colinear



counterclockwise



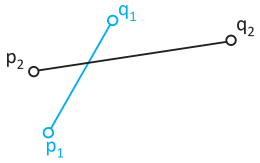
clockwise



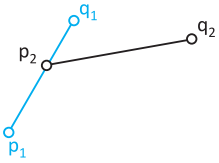
collinear

Two segments (p_1, q_1) and (p_2, q_2) intersect if and only if one of the following two conditions is verified:

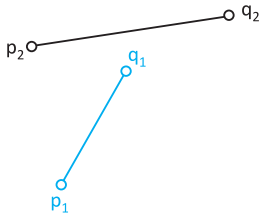
1. (p_1, q_1, p_2) and (p_1, q_1, q_2) have different orientations and (p_2, q_2, p_1) and (p_2, q_2, q_1) have different orientations.



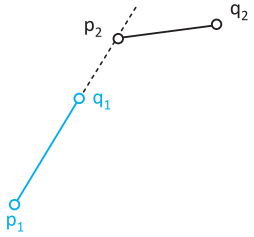
Example 1: Orientations of (p_1, q_1, p_2) and (p_1, q_1, q_2) are different. Orientations of (p_2, q_2, p_1) and (p_2, q_2, p_1) are also different



Example 2: Orientations of (p_1, q_1, p_2) and (p_1, q_1, q_2) are different. Orientations of (p_2, q_2, p_1) and (p_2, q_2, p_1) are also different



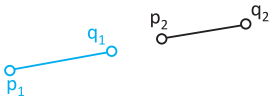
Example 3: Orientations of (p_1, q_1, p_2) and (p_1, q_1, q_2) are different. Orientations of (p_2, q_2, p_1) and (p_2, q_2, p_1) are also different



Example 4: Orientations of (p_1, q_1, p_2) and (p_1, q_1, q_2) are different. Orientations of (p_2, q_2, p_1) and (p_2, q_2, p_1) are also different

2. (p_1, q_1, p_2), (p_1, q_1, q_2), (p_2, q_2, p_1), and (p_2, q_2, q_1) are all collinear and
   - the x-projections of (p_1, q_1) and (p_2, q_2) intersect
   - the y-projections of (p_1, q_1) and (p_2, q_2) intersect

Example 1: All points are collinear. The x-projections of (p_1, q_1) and (p_2, q_2) intersect. The y-projections of (p_1, q_1) and (p_2, q_2) intersect



Example 2: All points are collinear. The x-projections of (p_1, q_1) and (p_2, q_2) do not intersect. The y-projections of (p_1, q_1) and do not (p_2, q_2) intersect

How to compute Orientation?

Use Cross Product :)

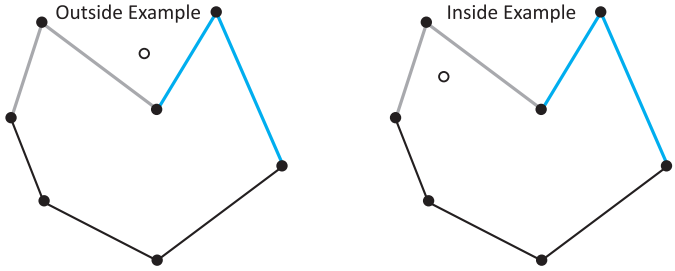
Code:

```
//cross product of OA and OB, this function will tell the orientation of OAB
ll cross(Point o, Point a, Point b) {
    int val = ((a.x - o.x)*(b.y - o.y) - (a.y - o.y)*(b.x - o.x));
    if(val == 0)return 0;
    return (val > 0 ? 2 : 1); //1 - clockwise , 2 - anticlockwise
}
struct Point
{
    int x;
    int y;
};
// Given three colinear points p, q, r, the function checks if
// point q lies on line segment 'pr'
bool onSegment(Point p, Point q, Point r)
{
    if (q.x <= max(p.x, r.x) && q.x >= min(p.x, r.x) &&
        q.y <= max(p.y, r.y) && q.y >= min(p.y, r.y))
        return true;

    return false;
}

bool doIntersect(Point p1, Point q1, Point p2, Point q2)
{
```

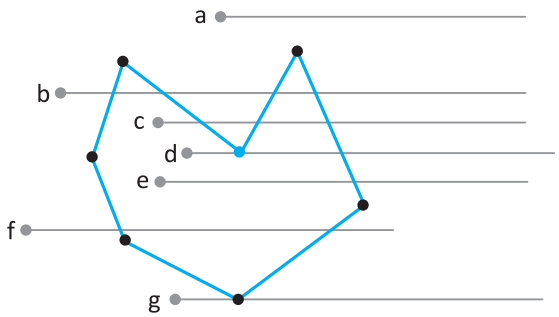
```
    // Find the four orientations needed
    int o1 = orientation(p1, q1, p2);
    int o2 = orientation(p1, q1, q2);
    int o3 = orientation(p2, q2, p1);
    int o4 = orientation(p2, q2, q1);

    // Case 1
    if (o1 != o2 && o3 != o4)
        return true;
    // Case 2
    // p1, q1 and p2 are colinear and p2 lies on segment p1q1
    if (o1 == 0 && onSegment(p1, p2, q1)) return true;

    // p1, q1 and q2 are colinear and q2 lies on segment p1q1

    if (o2 == 0 && onSegment(p1, q2, q1)) return true;

    // p2, q2 and p1 are colinear and p1 lies on segment p2q2

    if (o3 == 0 && onSegment(p2, p1, q2)) return true;

    // p2, q2 and q1 are colinear and q1 lies on segment p2q2
    if (o4 == 0 && onSegment(p2, q1, q2)) return true;

    return false;
}
```

How to check if a given point lies inside or outside a polygon?

Given a polygon and a point 'p', find if 'p' lies inside the polygon or not.
The points lying on the border are considered inside.



1. Draw a horizontal line to the right of each point and extend it to infinity
1. Count the number of times the line intersects with pol ygon edges.
2. A point is inside the polygon if either count of inter sections is odd or point lies on an edge of polygon. If none of the conditions is true, then point lies outside.

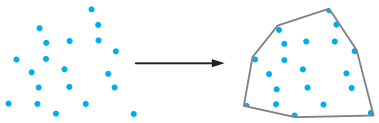



```
bool isInside(Point polygon[], int n, Point p)
{
     // Create a point for line segment from p to infinite
     Point extreme = {INF, p.y};
     // Count intersections of the above line with sides of polygon
     int count = 0, i = 0;
     do
     {
         int next = (i+1)%n;
         // Check if the line segment from 'p' to 'extreme' intersects
    // with the line segment from 'polygon[i]' to 'polygon[next]'
    if (doIntersect(polygon[i], polygon[next], p, extreme))
    {
         // If the point 'p' is colinear with line segment 'i<->next',
```

```
        // then check if it lies on segment. If it lies, return true,
        // otherwise false
        if (orientation(polygon[i], p, polygon[next]) == 0)
           return onSegment(polygon[i], p, polygon[next]);

        count++;
    }
    i = next;
    } while (i != 0);
    // Return true if count is odd, false otherwise
    return count&1; // Same as (count%2 == 1)
}
```

Convex Hull :

Given a set of points in the plane. the convex hull of the set is the smallest convex polygon that contains all the points of it.



Jarvis Algorithm :

The idea of Jarvis's Algorithm is simple, we start from the leftmost point (or point with minimum x coordinate value) and we keep wrapping points in counterclockwise direction. The big question is, given a point p as current point, how to find the next point in output?

The idea is to use orientation() here. Next point is selected as the point that beats all other points at counterclockwise orientation, i.e., next point is q if for any other point r, we have "orientation(p, r, q) = counterclockwise".

```
void convexHull(Point points[], int n)
{
    // There must be at least 3 points
    if (n < 3) return;
    // Initialize Result
    vector<Point> hull;
    // Find the leftmost point
    int l = 0;
```

```
    for (int i = 1; i < n; i++)
        if (points[i].x < points[l].x)
            l = i;
    // Start from leftmost point, keep moving counterclockwise
    // until reach the start point again.
    int p = l, q;
    do
    {
         // Add current point to result
         hull.push_back(points[p]);
         // Search for a point 'q' such that orientation(p, x, q)
         //is counterclockwise for all points 'x'. The idea
         // is to keep track of last visited most counterclock-
         // wise point in q. If any point 'i' is more counterclock-
         // wise than q, then update q.
         q = (p+1)%n;
         for (int i = 0; i < n; i++)
         {
             // If i is more counterclockwise than current q, then
             // update q
             if (orientation(points[p], points[i], points[q]) == 2)
                 q = i;
         }
         // Now q is the most counterclockwise with respect to p
         // Set p as q for next iteration, so that q is added to
         // result 'hull'
         p = q;
    } while (p != l); // While we don't come to first point
}
```

Graham Scan :

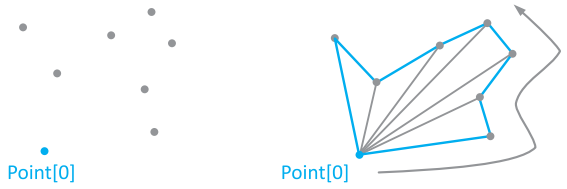
Let points[0…n-1] be the input array.

1. Find the bottom-most point by comparing **y** coordinate of all points. If there are two points with same y value, then the point with smaller x coordinate value is considered. Let the bottommost point be P0. Put P0 at first position in output hull.

2. Consider the remaining n-1 points and sort them by polor angle in counterclockwise order around points[0]. If polor angle of two points is same, then put the nearest point first.
3. After sorting, check if two or more points have same angle. If two more points have same angle, then remove all same angle points except the point farthest from P0. Let the size of new array be m.
4. If m is less than 3, return (Convex Hull not possible)
5. Create an empty stack 'S' and push points[0], points[1] and points[2] to S.
6. Process remaining m-3 points one by one. Do following for every point 'points[i]'
   4.1 Keep removing points from stack while orientation of following 3 points is not counterclockwise (or they don't make a left turn).
   (a) Point next to top in stack **(b)** Point at the top of stack
   (c) points[i]
   4.2 Push points[i] to S
7. Print contents of S.

The above algorithm can be divided in two phases :

Phase 1 (Sort points): We first find the bottom-most point. The idea is to pre-process points be sorting them with respect to the bottommost point. Once the points are sorted, they form a simple closed path.



Given Points

Points considered according to increasing angle with respect to point [0] form a simple closed path

Phase 2 (Accept or Reject Points): Once we have the closed path, the next step is to traverse the path and remove concave points on this path. How to decide which point to remove and which to keep? Again, orientation helps here. The first two points in sorted array are always part of Convex Hull. For remaining points, we keep track of recent three points, and find the angle formed by them. Let the three points be prev - p, curr - c and next - n. If orientation of these points (considering them in same order) is not counterclockwise, we discard c, otherwise we keep it.

```
void convexHull(Point points[], int n)
{
    // Find the bottommost point
    int ymin = points[0].y, min = 0;
    for (int i = 1; i < n; i++)
```

```
{
  int y = points[i].y;
  // Pick the bottom-most or chose the left
  // most point in case of tie
  if ((y < ymin) || (ymin == y &&
      points[i].x < points[min].x))
     ymin = points[i].y, min = i;
  }

  // Place the bottom-most point at first position swap(points[0], points[min]);

  // Sort n-1 points with respect to the first point.
  // A point p1 comes before p2 in sorted ouput if p2
  // has larger polar angle (in counterclockwise
  // direction) than p1
  p0 = points[0];
  sort(&points[1], n-1,compare);

  // If two or more points make same angle with p0,
  // Remove all but the one that is farthest from p0
  // Remember that, in above sorting, our criteria was
  // to keep the farthest point at the end when more than
  // one points have same angle.
  int m = 1; // Initialize size of modified array
  for (int i=1; i<n; i++)
  {
      // Keep removing i while angle of i and i+1 is same
      // with respect to p0
      while (i < n-1 && orientation(p0, points[i],points[i+1]) == 0)
      i++;


    points[m] = points[i];
    m++; // Update size of modified array
}
```

```
    // If modified array of points has less than 3 points,
    // convex hull is not possibles
    if (m < 3) return;

    // Create an empty stack and push first three points
    // to it.
    stack<Point> S;
    S.push(points[0]);
    S.push(points[1]);
    S.push(points[2]);

    // Process remaining n-3 points
    for (int i = 3; i < m; i++)
    {
       // Keep removing top while the angle formed by
       // points next-to-top, top, and points[i] makes
       // a non-left turn
       while (orientation(nextToTop(S), S.top(), points[i]) != 2)
          S.pop();
       S.push(points[i]);
    }
    // Now stack has the output points, print contents of stack
}
```

Andrew's monotone chain convex hull algorithm :

It constructs the convex hull of a set of 2-dimensional points.

It does so by first sorting the points lexicographically (first by xcoordinate, and in case of a tie, by y-coordinate), and then constructing upper and lower hulls of the points.

Input: a list P of points in the plane.

Sort the points of P by x-coordinate (in case of a tie, sort by y-coordinate).

Initialize U and L as empty lists.

The lists will hold the vertices of upper and lower hulls respectively.

for i = 1, 2, ..., n:

while L contains at least two points and the sequence of last two points of L and the point P[i] does not make a counter-clockwise turn:

remove the last point from L
append P[i] to L
for i = n, n-1, ..., 1:

while U contains at least two points and the sequence of last two points of U and the point P[i] does not make a counter-clockwise turn:

remove the last point from U

append P[i] to U

Remove the last point of each list (it's the same as the first point of the other list).

Concatenate L and U to obtain the convex hull of P.

Points in the result will be listed in counter-clockwise order.

How will you check whether a given point lies inside a triangle or not??

Hint : Don't give the answer discussed above for polygon.

Problem - KTHCON

A 1-concave polygon is a simple polygon (its sides don't intersect or touch) which has at least 1 concave interior angle.

There are N points on a plane. Let S be the maximum area of a 1-concave polygon with vertices in those points. Compute 2S. Note that if there is no such (1-concave) polygon, you should print -1.

Constraints:

$1 \leq T \leq 100$
$3 \leq N \leq 105$
No two points coincide (have identical both x and y coordinates).

No three points are collinear.

The sum of N over all test cases won't exceed 5·105.

$|x|, |y| \leq 109$

Input :

2
5
2 2
-2 -2
2 -2

-2 2
0 1
3
0 0
1 0
0 1

Output

28
-1

Code :

```
ll cross(pair < ll, ll > o, pair < ll, ll >a, pair < ll,
ll > b) {
//cross product of OA and OB
return ((a.first - o.first)*(b.second - o.second) - (a.second - o.second)*(b.first
- o.first));
}

void convex_hull(vector < pair < ll, ll > > &pt) {
    //function to calculate the hull
    if (pt.size() < 3)return;
    sort(pt.begin(), pt.end());
    vector < pair < ll, ll > > up, dn;
    int k = 0;
    for (int i = 0; i < pt.size(); ++i) {
        while (up.size() > 1 && cross(up[up.size() - 2],
        up[up.size() - 1], pt[i]) >= 0)up.pop_back();
        while (dn.size() > 1 && cross(dn[dn.size() - 2],
        dn[dn.size() - 1], pt[i]) <= 0)dn.pop_back();
        up.push_back(pt[i]);
        dn.push_back(pt[i]);
    }
    pt = dn;
    for (int i = up.size() - 2; i >= 1; i—) {
        pt.push_back(up[i]);
    }
}
```

```
ll triangle_area(pair < ll, ll > a, pair < ll, ll > b, pa
ir < ll, ll > c) {
    return abs(cross(a, b, c));
}
int main() {

cin >> t;
while (t—) {
    vector < pair < ll, ll > > outer;
    vector < pair < ll, ll > > inner;
    map < pair < ll, ll >, int > mp;
    cin >> n1;
    for (int i = 0; i < n1; ++i) {
        cin >> a >> b;
        outer.push_back(make_pair(a, b));
        mp[make_pair(a, b)] = 1;
    }
    convex_hull(outer);
    for (int i = 0; i < outer.size(); ++i)mp[outer[i]] = 0;
    for (map < pair < ll, ll >, int > ::iterator it = mp.begin(); it !=
    mp.end(); ++it) {
        if ((it->second) == 1) {
           inner.push_back(it->first);
        }
    }
    convex_hull(inner);
    if (inner.size() == 0) {
        cout << “-1\n”;
        continue;
    }
    //now we got the inner and outer HUll
    //calc the outer HUll area
    ll area = 0;
    for (int i = 1; i < outer.size() - 1; ++i) {area += triangle_area(outer[0],
    outer[i], outer[i + 1]);
```

```
    }
    //cout << area << "\n";
    //got the area of convex hull now we have to remove one point from inner hull
    ll minTrianglearea = INT64_MAX;
    int fptr = 0, sptr = 0;
    for (int i = 0; i < outer.size(); ++i) {
        ll currarea = triangle_area(outer[i], outer[(i + 1) % outer.size()],
        inner[sptr]);

        while (true) {
            ll newarea = triangle_area(outer[i], outer[(i + 1) % outer.size()], inner[(sptr
            + 1) % inner.size()]);
            if (newarea < currarea) {
                currarea = newarea;
                sptr = (sptr + 1) % inner.size();
            }
            else {
                break;
            }
        }
        minTrianglearea = min(minTrianglearea, currarea);
    }
    cout << abs(area - minTrianglearea) << "\n";

   }

   return 0;
}
```

14 Fast Fourier Transformation

Step-1 FFT

$$A(x) = a_0x^0 + a_1x^1 + \ldots + a_{n-1}x^{n-1}$$

Divide it into two polynomials, one with even coefficients(0) and the other with the odd coefficients(1):

$$A_0(x) = a_0x^0 + a_2x^1 + \ldots + a_{n-2}x^{n/2-1}$$

$$A_1(x) = a_1x^0 + a_3x^1 + \ldots + a_{n-1}x^{n/2-1}$$

Finally we can write original polynomial as combination of Ao and A1:

$$A(x) = A_0(x^2) + xA_1(x^2)$$

Now writing th original polynomial A(x) in point form for n-th root of unity:

$$y_k = y_k^0 + w_n^k y_k^1, \quad k = 0 \ldots n/2-1$$

$$y_{k+n/2} = A(w_n^{k+n/2}) = A_0(w_n^{2k+n}) + w_n^{k+n/2}A_1(w_n^{2k+n}) = A_0(w_n^{2k}w_n^n)A_1(w_n^{2k}w_n^n)$$

$$= A(w_n^{2k}) - w_n^k A_1(w_n^{2k}) = y_k^0 - w_n^k y_k^1$$

Finally our **point form** will be:

$$y_k = y_k^0 + w_n^k y_k^1, \quad k = 0 \ldots n/2-1,$$

$$y_{k+n/2} = y_k^0 + w_n^k y_k^1, \quad k = 0 \ldots n/2-1$$

FFT()

```
vector<complex<double> > fft(vector<complex<double> > a){
    int n = (int)a.size();
    if(n <= 1)
        return a;
    //Dividing a0 and a1 as even and odd polynomial of degree n/2
    vector<complex<double> > a0(n/2), a1(n/2);
    for(int i = 0;i<n/2;i++){
       a0[i] = a[2*i];
       a1[i] = a[2*i + 1];
    }
```

```
    //Divide step
    //Recursively calling FFT on polynolmial of degree n/2
    a0 = fft(a0);
    a1 = fft(a1);
    double ang = 2*PI/n;
    //defining w1 and wn
    complex<double> w(1) , wn(cos(ang),sin(ang));
    for(int i = 0;i<n/2;i++){
        //for powers of k<= n/2
        a[i] = a0[i] + w*a1[i];
        //powers of k > n/2
        a[i + n/2] = a0[i] - w*a1[i];
        //Updating value of wk
        w *= wn;
    }
    return a;
}
```

Time Complexity: O(nlogn)

Step 2: Convolution

Multiply()

```
void multiply(vector<int> a, vector<int> b){
   vector<complex<double> > fa(a.begin(),a.end()), fb(b.begin(),b.end());
   int n = 1;
   //resizing value of n as power of 2
   while(n < max(a.size(),b.size()))
       n <<= 1;
   n <<= 1;
   //cout<<n<<endl;
   fa.resize(n);
   fb.resize(n);
   //Calling FFT on polynomial A and B
   //fa and fb denotes the point form
   fa = fft(fa);
   fb = fft(fb);
```

```
    //Convolution step
    for(int i = 0;i<n;i++){
       fa[i] = fa[i] * fb[i];
    }
    //Converitng fa from coefficient back to point form
    fa = inv_fft(fa);
    return;
}
```

Time Complexity: O(n) (exluding the time for calculating FFT())

Step 3: Inverse FFT

So, suppose we are given a vector (y_0,y_1,y_2, ...y_{n-1}).Now we need to restore it back to coefficient form.

$$\begin{pmatrix} w_n^0 & w_n^0 & w_n^0 & w_n^0 & \cdots & w_n^0 \\ w_n^0 & w_n^1 & w_n^2 & w_n^3 & \cdots & w_n^{n-1} \\ w_n^0 & w_n^2 & w_n^4 & w_n^6 & \cdots & w_n^{2(n-1)} \\ w_n^0 & w_n^3 & w_n^6 & w_n^9 & \vdots & w_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ w_n^0 & w_n^{n-1} & w_n^{2(n-1)} & w_n^{3(n-1)} & \cdots & w_n^{(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{pmatrix}$$

Then the vector {a_0,a_1,a_2,...a_{n-1}}can be found by multiplying the vector {y_0, y_1, y_2,... y_{n-1}} by an inverse matrix:

$$\begin{pmatrix} w_n^0 & w_n^0 & w_n^0 & w_n^0 & \cdots & w_n^0 \\ w_n^0 & w_n^1 & w_n^2 & w_n^3 & \cdots & w_n^{n-1} \\ w_n^0 & w_n^2 & w_n^4 & w_n^6 & \cdots & w_n^{2(n-1)} \\ w_n^0 & w_n^3 & w_n^6 & w_n^9 & \vdots & w_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ w_n^0 & w_n^{n-1} & w_n^{2(n-1)} & w_n^{3(n-1)} & \cdots & w_n^{(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{pmatrix}$$

Thus we get the formula:

$$a_k = \frac{1}{n}\sum_{j=0}^{n-1} y_j w_n^{-kj}$$

Comparing it with the formula for y_k:

$$y_k = \sum_{j=0}^{n-1} a_j w_n^{kj},$$

Inverse_FFT()

```
vector<complex<double> > inv_fft(vector<complex<double> > y){
    int n = y.size();
    if(n <= 1)
       return y;
    vector<complex<double> > y0(n/2), y1(n/2);
    for(int i = 0;i<n/2;i++){
       y0[i] = y[2*i];
       y1[i] = y[2*i + 1];
    }
    y0 = inv_fft(y0);
    y1 = inv_fft(y1);
    double ang = 2 * PI/n * -1;
    complex<double> w(1), wn(cos(ang), sin(ang));
    for(int i = 0;i<n/2;i++){
       y[i] = y0[i] + w*y1[i];
       y[i + n/2] = y0[i] - w*y1[i];
       y[i] /= 2;
       y[i + n/2] /= 2;
       w *= wn;
    }
    //each element of the result is divided by 2
    //assuming that the division by 2 will take place at each level of recursion
    //then eventually just turns out that all the elements are divided into n.

    return y;
}
```

Time Complexity: O(nlogn)

Ques: Very Fast Multiplicaiton

http://www.spoj.com/problems/VFMUL/

```
#include<iostream>
#include<vector>
#include<complex>
#include<algorithm>

using namespace std;
```

```
#define PI 3.14159265358979323846
vector<complex<double> > fft(vector<complex<double> > a){
    //for(int i = 0;i<a.size();i++)cout<<a[i]<<" ";cout<<endl;
    int n = (int)a.size();
    if(n <= 1)
       return a;
    vector<complex<double> > a0(n/2), a1(n/2);
    for(int i = 0;i<n/2;i++){
        a0[i] = a[2*i];
        a1[i] = a[2*i + 1];
        //cout<<n<<" "<<a0[i]<<" "<<a1[i]<<endl;
    }
    a0 = fft(a0);
    a1 = fft(a1);
    double ang = 2*PI/n;
    complex<double> w(1) , wn(cos(ang),sin(ang));
    for(int i = 0;i<n/2;i++){
        a[i] = a0[i] + w*a1[i];
        a[i + n/2] = a0[i] - w*a1[i];
        w *= wn;
        //cout<<a[i]<<" "<<a[i+n/2]<<endl;
    }
    return a;
}

vector<complex<double> > inv_fft(vector<complex<double>>y){
    int n = y.size();
    if(n <= 1)
         return y;
    vector<complex<double> > y0(n/2), y1(n/2);
    for(int i = 0;i<n/2;i++){
        y0[i] = y[2*i];
        y1[i] = y[2*i + 1];
    }
    y0 = inv_fft(y0);
```

```
    y1 = inv_fft(y1);
    double ang = 2 * PI/n * -1;
    complex<double> w(1), wn(cos(ang), sin(ang));
    for(int i = 0;i<n/2;i++){
        y[i] = y0[i] + w*y1[i];
        y[i + n/2] = y0[i] - w*y1[i];
        y[i] /= 2;
        y[i + n/2] /= 2;
        w *= wn;
    }
    return y;
}
void multiply(vector<int> a, vector<int> b){
    vector<complex<double> > fa(a.begin(),a.end()), fb(b.begin(),b.end());
    int n = 1;
    while(n < max(a.size(),b.size()))
         n <<= 1;
    n <<= 1;
    //cout<<n<<endl;
    fa.resize(n);
    fb.resize(n);
    fa = fft(fa);
    fb = fft(fb);
    for(int i = 0;i<n;i++){
        fa[i] = fa[i] * fb[i];
        //cout<<fa[i]<<endl;
    }
    fa = inv_fft(fa);
    vector<int> res(n);
    for(int i = 0;i<n;i++){
        res[i] = int(fa[i].real() + 0.5);
        //cout<<res[i]<<endl;
    }
    int carry = 0;
    for(int i = 0;i<n;i++){
```

```
        res[i] = res[i] + carry;
        carry = res[i] / 10;
        res[i] %= 10;
    }
    bool flag = 0;
    for(int i = n-1;i>=0;i—){if(res[i] || flag){printf(“%d”,res[i]);flag = 1;}}
        if(!flag)printf(“0”);
        printf(“\n”);
        return;
}
int main(){
    int n;
    scanf(“%d”,&n);
    while(n—){
        string x,y;
        vector<int> a,b;
        cin>>x>>y;
        for(int i = 0;i<x.length();i++){
             a.push_back(int(x[i] - ‘0’));
        }
        for(int i = 0;i<y.length();i++){
            b.push_back(int(y[i] - ‘0’));
        }
        reverse(a.begin(),a.end());
        reverse(b.begin(),b.end());
        multiply(a,b);
    }
    return 0;
}
```

SELF STUDY NOTES

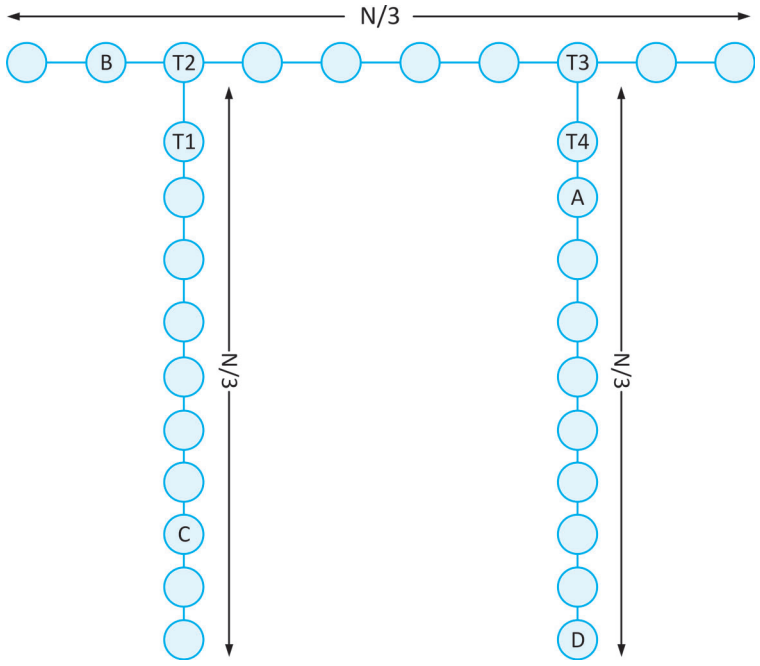
SELF STUDY NOTES

15 Heavy Light Decomposition

Heavy Light Decomposition

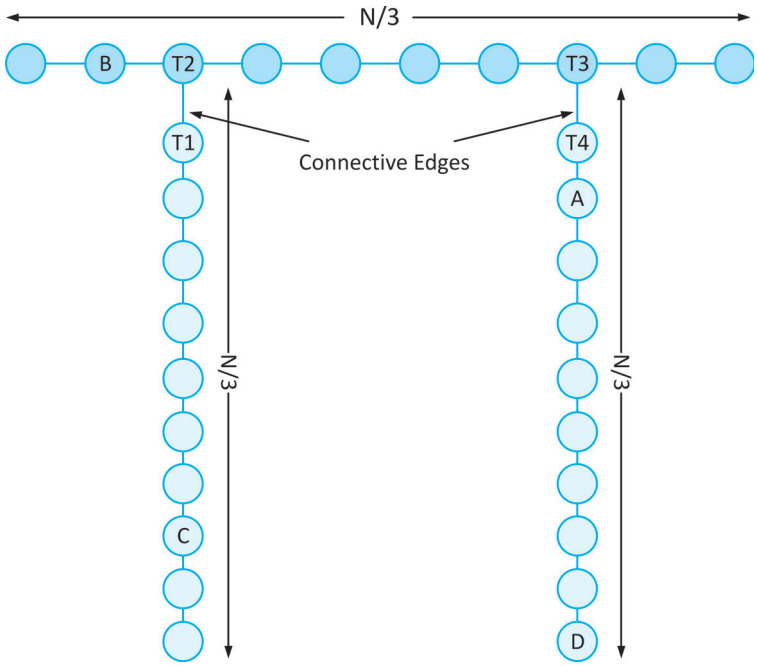
- **Balanced Binary Tree** is good, need to visit at most **2log N** nodes to reach from any node to any other node in the tree.
- If a balanced binary tree with N nodes is given, then many queries can be done with O(log N) complexity. **Distance of a path, Maximum/Minimum in a path, Maximum contiguous sum** etc etc.
- **Chains** are also good. We can do many operations on array of elements with O(log N) complexity using **segment tree / BIT**.
- **Unbalanced Tree** is bad

Unbalanced Trees :



Travelling from one node to other requires **O(n)** time.

What if we broke the tree in to 3 chains?



Basic Idea

We will divide the tree into **vertex-disjoint chains** (Meaning no two chains has a node in common) in such a way that to move from any node in the tree to the root node, we will have to change at most **log N chains**. To put it in another words, the **path** from **any node** to **root** can be broken into pieces such that the each piece belongs to only one chain, then we will have **no more than log N pieces**.
So what?

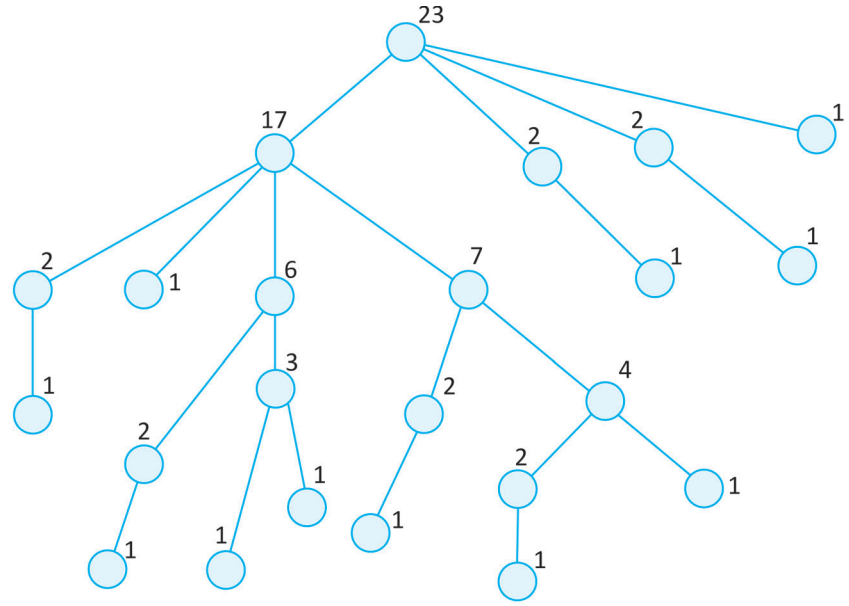
Now the path from any **node A to any node B** can be broken into two paths: **A to LCA(A, B) and B to LCA(A, B)**.

Time Complexity

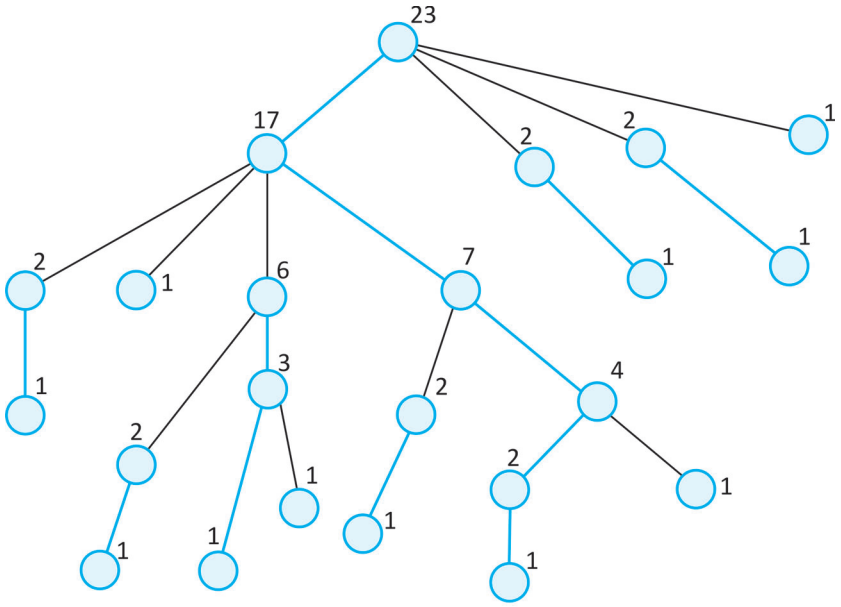
We already know that queries in **each chain** can be answered with **O(log N)** complexity and there are **at most log N chains** we need to consider per path. So on the whole we have **O(log^2 N) complexity** solution.

Special Child : Among all child nodes of a node, the one with **maximum sub-tree size** is considered as Special child. Each non leaf node has exactly one Special child.

Special Edge : For each **non-leaf node**, the edge **connecting the node with its Special child** is considered as Special Edge.



Each node has its sub-tree size written on top.
Each non-leaf node has exactly one special child whose sub-tree size is colored.
Colored child is the one with maximum sub-tree size.



Each chain is represented with different color.
Thin black lines represent the connecting edges. They connect 2 chains.

Algorithm

```
HLD(curNode, Chain):
    Add curNode to curChain
    If curNode is LeafNode: return
    //Nothing left to do
    sc := child node with maximum sub-tree size
    //sc is the special child
    HLD(sc, Chain)
    //Extend current chain to special child
    for each child node cn of curNode:
     //For normal childs
        if cn != sc: HLD(cn, newChain) //As told above, for each normal child,
        a new chain starts
```

dfs()

```
void dfs(int cur, int prev, int level = 0){
    //pa[cur][0] will be its immediate parent
    pa[cur][0] = prev;
    //setting the depth of the current node
    depth[cur] = level;
    //Initially set the subtree size of every node as 1
    //Add the subtree size of every child node of cur node in its size
    subsz[cur] = 1;
    for (int i = 0; i < adj[cur].size(); i++){
        if (adj[cur][i] != prev){
            otherEnd[ind[cur][i]] = adj[cur][i];
            dfs(adj[cur][i], cur, level + 1);
            subsz[cur] += subsz[adj[cur][i]];
        }
    }
}
```

LCA()

```
int LCA(int p, int q){
    if (depth[q] > depth[p])swap(p, q);
    int diff = depth[p] - depth[q];
    for (int i = 0; i < LN; i++){
        if ((diff >> i) & 1)
            p = pa[p][i];
    }
    //Now p and q are at same level
    if (p == q)return p;
    for (int i = LN - 1; i >= 0; i–){
        if (pa[p][i] != -1 && pa[p][i] != pa[q][i]){
            p = pa[p][i];
            q = pa[q][i];
        }
    }
    return pa[p][0];
}
```

HLD()

```
ChainNo –> Chain number of current chain.
chainHead[i] –> Chain head of the chain in which i-th vertex is present.
chainPos[i] –> Position of i-th node in its chain.
chainInd[i] –> Index of chain in which i-th node is present.
chainSize[chainNo] –> Size of chain with chain number chainNo.
baseArray[] –> array on which segment tree will operate,
all the nodes in the same chain will be adjacent in base array
posInBase[i] –> position of i-th node in base array
int chainNo=0,chainHead[N],chainPos[N],chainInd[N],chainSize[N];
void hld(int cur,, int prev) {
    //If we have encountered this chain for the first time
    if(chainHead[chainNo] == -1) chainHead[chainNo]=cur;
    //This node will be present in current chain
    chainInd[cur] = chainNo;
    //Position of current node in chain will be present size of chain
    chainPos[cur] = chainSize[chainNo];
```

```
    //Increment size of chain
    chainSize[chainNo]++;
    // Position of this node in baseArray which we will use in Segtree
    //Initially ptr was 0
    posInBase[curNode] = ptr;
    //Insert the cost/value of node in the base array
    baseArray[ptr++] = cost;

    //Find the maximum sized subtree and its index
    int ind = -1,mx_sz = -1;
    for(int i = 0; i < adj[cur].sz; i++) {
            //make sure we are not calling HLD for the parent node
            if(adj[cur][i] != prev && subsize[ adj[cur][i] ] > mx_sz) {
                    mx_sz = subsize[ adj[cur][i] ];ind = i;
        }
    }
    //continue the HLD for present chain with special child as new new member
    //of current chain
    if(ind >= 0) hld( adj[cur][ind], cur);

    for(int i = 0; i < adj[cur].sz; i++) {
        //If the child is not special, start a new chain
        if(i != ind) {
            chainNo++;
            hld( adj[cur][i], cur );
        }
    }
}
```

query_up()

```
//v is ancestor of u
//query the chain in which 'u' is present till head of that chain, then move to
next chain up
//we do that till u and v are in the same chain and then we query from u to v
in the same chain :D
int query_up(int u, int v){
```

```
    if (u == v)return 0;
    int uchain, vchain = chainInd[v], ans = -1;
    while (1){
        //If both u and v are in same chain, query that chain and we are done
uchain = chainInd[u];
        if (uchain == vchain){
            if (u == v)break;
            ans = max(ans,query_tree(1, 0, sz, posInBase[v] + 1, posInBase[u]));
            break;
        }
        //else, query the u-chain from u to its head and then change the chain
        //by calling query from parent[u] to v
        ans = max(ans,query_tree(1, 0, sz, posInBase[chainHead[uchain]],
        posInBase[u]));
        u = chainHead[uchain];
        u = pa[u][0];//move to parent of u, we are moving a chain up

        //keep on doing this untill u and v are in same chain
    }
    return ans;
}
```

query()

```
int query(int p, int q){
    int lca = LCA(p, q);
    //path from p to q is divided into path from p to lca
    (p,q) and q to lca(p,q)
    int ans1 = query_up(p, lca);
    int ans2 = query_up(q, lca);

    //If max query
    int ans = max(ans1,ans2);

    //If sum query
    int ans = ans1 + ans2 - cost[LCA(p,q)];
    return ans;
}
```

Time Complexity

Build Time —> **O(n)**

Query Time —> **O(logn * logn)**

Spoj QTREE

http://www.spoj.com/problems/QTREE/

```
#include<bits/stdc++.h>
typedef long long ll;
#define fast std::ios::sync_with_stdio(false);std::cin.ti
e(false)
#define endl "\n"
//#define abs(a) a >= 0 ? a : -a
#define ll long long int
#define mod 1000000007
#define Endl endl
using namespace std;

const int N = 10000 + 10;
const int MAX = 1e6 + 10;
const int LN = 14;
vector<int> adj[N], costs[N], ind[N];
int n, sz, arr[N], tree[N], depth[N], pa[N][20], otherEnd
[N], subsz[N];
int chainNo, chainHead[N], chainInd[N], posInBase[N];
int st[4 * N];

void build(int node, int l, int r){
    if (l == r){
      st[node] = arr[l];
      return;
    }
    int mid = (l + r) >> 1;
    build(2 * node, l, mid);
    build(2 * node + 1, mid + 1, r);
    st[node] = (st[2 * node] > st[2 * node + 1]) ? st[2 * node] : st[2 * node + 1];
}
```

```
void update(int node, int l, int r, int i, int val){
    if (l > r || l > i || r < i)return;
    if (l == i && r == i){
        st[node] = val;
        return;
    }
    int mid = (l + r) >> 1;
    update(2 * node, l, mid, i, val);
    update(2 * node + 1, mid + 1, r, i, val);
    st[node] = (st[2 * node] > st[2 * node + 1]) ? st[2 * node] : st[2 * node + 1];
}

int query_tree(int node, int l, int r, int i, int j){
    if (l > r || l > j || r < i){
        return -1;
    }
    if (l >= i && r <= j){
        return st[node];
    }
    int mid = (l + r) >> 1;
    return max(query_tree(2 * node, l, mid, i, j), query_tree(2 * node + 1, mid +
    1, r, i, j));
}

//v is ancestor of u
//query the chain in which ‘u’ is present till head of th
at chain, then move to next chain up
//we do that till u and v are in the same chain and then
we query from u to v in the same chain :D
int query_up(int u, int v){
    if (u == v)return 0;
    int uchain, vchain = chainInd[v], ans = -1;
    while (1){
        uchain = chainInd[u];
        if (uchain == vchain){
```

```
            if (u == v)break;
            ans = max(ans,query_tree(1, 0, sz, posInBase[v] + 1, posInBase[u]));break;
        }
        ans = max(ans,query_tree(1, 0, sz, posInBase[chainHead[uchain]],
        posInBase[u]));
        u = chainHead[uchain];
        u = pa[u][0];//move to parent of u, we are moving a chain up
    }
    return ans;
}

int LCA(int p, int q){
    if (depth[q] > depth[p])swap(p, q);
    int diff = depth[p] - depth[q];
    for (int i = 0; i < LN; i++){
        if ((diff >> i) & 1)
            p = pa[p][i];
    }
    //Now p and q are at same level
    if (p == q)return p;
    for (int i = LN - 1; i >= 0; i–){
        if (pa[p][i] != -1 && pa[p][i] != pa[q][i]){
            p = pa[p][i];
            q = pa[q][i];
        }
    }
    return pa[p][0];
}

int query(int p, int q){
    int lca = LCA(p, q);
    //path from p to q is divided into path from p to lca (p,q) and q to lca(p,q)
    int ans = query_up(p, lca);
    int temp = query_up(q, lca);
    if (temp > ans)ans = temp;
    return ans;
}
```

```
void change(int i, int val){
     int p = otherEnd[i];
     update(1, 0, sz, posInBase[p], val);
    }
void HLD(int cur, int cost, int prev){
    if (chainHead[chainNo] == -1){
        chainHead[chainNo] = cur;
    }
    chainInd[cur] = chainNo;
    posInBase[cur] = sz;
    arr[sz++] = cost;

    int sc = -1, sc_cost;
    for (int i = 0; i < adj[cur].size(); i++){
         if (adj[cur][i] != prev){
             if (sc == -1 || subsz[sc] < subsz[adj[cur][i]]){
                 sc = adj[cur][i];
                 sc_cost = costs[cur][i];
             }
         }
    }
    if (sc != -1)HLD(sc, sc_cost, cur);

    for (int i = 0; i < adj[cur].size(); i++){
         if (adj[cur][i] != prev && sc != adj[cur][i]){
             //new chain at each normal node :)
             chainNo++;
             HLD(adj[cur][i], costs[cur][i], cur);
         }
    }
}

void dfs(int cur, int prev, int level = 0){
    pa[cur][0] = prev;
    depth[cur] = level;
```

```
    subsz[cur] = 1;
    for (int i = 0; i < adj[cur].size(); i++){
        if (adj[cur][i] != prev){
            otherEnd[ind[cur][i]] = adj[cur][i];
            dfs(adj[cur][i], cur, level + 1);
            subsz[cur] += subsz[adj[cur][i]];
        }
    }
}

int main(){
    int t;
    scanf("%d", &t);
    while (t--){
        //printf("\n");
        sz = 0;
        scanf("%d", &n);
        for (int i = 0; i < n; i++){
            adj[i].clear();
            costs[i].clear();
            ind[i].clear();
            chainHead[i] = -1;
            for (int j = 0; j < LN; j++)pa[i][j] = -1;
        }
        for (int i = 1; i < n; i++){
            int a, b, c;
            scanf("%d %d %d", &a, &b, &c);
            --a;--b;
            adj[a].push_back(b);
            adj[b].push_back(a);
            ind[a].push_back(i - 1);
            ind[b].push_back(i - 1);
            costs[a].push_back(c);
            costs[b].push_back(c);
        }
```

```
        chainNo = 0;
        dfs(0, -1);//setup subtree size, depth and parent for each node;
        //cout << "dfs over !!!" << endl;
        HLD(0, -1, -1);//decompose the tree and create the baseArray
        //cout << "HDL over" << endl;
        build(1, 0, sz);
        //cout << "Build over" << endl;
        //for (int i = 0; i < n; i++)cout << "edge : " << i << " " << otherEnd[i]
        <<" "<<posInBase[i] << endl;

        //bottom up DP code for LCA:
        for (int j = 1; j < LN; j++){
            for (int i = 0; i < n; i++){
                if (pa[i][j - 1] != -1)pa[i][j] = pa[pa[i][j - 1]][j - 1];
            }
        }
        char ch[20];
        while (1){
            scanf("%s", ch);
            if (ch[0] == 'D')break;
            int a, b;
            scanf("%d %d", &a, &b);
            //cout << ch << " " << a << " " << b << Endl;
            if (ch[0] == 'Q')
              printf("%d\n", query(a - 1, b - 1));
            else
              change(a - 1, b);
        }
    }
    return 0;
}
```

Spoj QTREE2

http://www.spoj.com/problems/QTREE2/

```
#include<bits/stdc++.h>
typedef long long ll;
#define fast std::ios::sync_with_stdio(false);std::cin.tie(false)
#define endl "\n"
//#define abs(a) a >= 0 ? a : -a
#define ll long long int
#define mod 1000000007
#define Endl endl
using namespace std;
const int N = 200000 + 10;
const int MAX = 1e6 + 10;
const int LN = 15;
vector<int> adj[N], costs[N], ind[N];
int n, sz, arr[N], tree[N], depth[N], pa[N][20], otherEnd[N], subsz[N];
int chainNo, chainHead[N], chainInd[N], posInBase[N];
int st[4 * N];

void build(int node, int l, int r){
    if (l == r){
        st[node] = arr[l];
        return;
    }
    int mid = (l + r) >> 1;
    build(2 * node, l, mid);
    build(2 * node + 1, mid + 1, r);
    st[node] = st[2 * node] + st[2 * node + 1];
}
int query_tree(int node, int l, int r, int i, int j){
    if (l > r || l > j || r < i){
        return 0;
    }
    if (l >= i && r <= j){
        return st[node];
```

```
    }
    int mid = (l + r) >> 1;
    return (query_tree(2 * node, l, mid, i, j) + query_tree(2 * node + 1, mid + 1,
    r, i, j));
}

int query_up(int u,int v){
    if (u == v)return 0;
    int uchain, vchain = chainInd[v], ans = 0;//uchain and vchain contains the
    chain number
    while (1){
        uchain = chainInd[u];
        if (uchain == vchain){
            if (u == v)break;
            ans += query_tree(1, 0, sz, posInBase[v] + 1,
posInBase[u]);//please note that we are doing +1 because arr[v] contains the
edge length

              //between node ‘v-1’ and ‘v’. So by doing
arr[posInBase[v] + 1] we get the
//first edge between v and v + 1 and so forth and so on upto u.
          break;
       }
       ans += query_tree(1, 0, sz, posInBase[chainHead[uchain]], posInBase[u]);
       u = chainHead[uchain];
       u = pa[u][0];
    }
    return ans;
}

int LCA(int u, int v){
    if (depth[u] < depth[v])swap(u, v);
    int diff = depth[u] - depth[v];
    for (int i = 0; i < LN; i++){
        if ((diff >> i) & 1){
            u = pa[u][i];
```

```
        }
    }
    if (u == v)return u;
    //Now u and v are at the same level
    for (int i = LN - 1; i >= 0; i–){
        if (pa[u][i] != pa[v][i]){
            u = pa[u][i];
            v = pa[v][i];
        }
    }
    return pa[u][0];
}
int query(int u, int v){
    int lca = LCA(u, v);
    int ans = query_up(u, lca);
    ans += query_up(v, lca);
    //cout << “query: “ << u << “ “ << v << “ lca : “ << lca << endl;
    return ans;
}

void HLD(int node, int prev, int cost){
    //cout << “head : “ << chainNo << “ “ << node<<“ “<<chainHead[chainNo] <<
    endl;
        if (chainHead[chainNo] == -1){
            chainHead[chainNo] = node;
        }
        chainInd[node] = chainNo;
        posInBase[node] = sz;
        arr[sz++] = cost;

        int sc = -1, ncost;
        for (int i = 0; i < adj[node].size(); i++){
            if (adj[node][i] != prev){
               if (sc == -1 || subsz[sc] < subsz[adj[node][i]]){
                   sc = adj[node][i];
                   ncost = costs[node][i];
```

```
            }
        }
    }
    if (sc != -1){
        HLD(sc, node, ncost);
    }
    for (int i = 0; i < adj[node].size(); i++){
        if (adj[node][i] != prev && adj[node][i] != sc){chainNo++;
            HLD(adj[node][i], node, costs[node][i]);
        }
    }
    return;
}

void dfs(int node, int prev, int level){
    pa[node][0] = prev;
    depth[node] = level;
    subsz[node] = 1;
    for (int i = 0; i < adj[node].size(); i++){
        if (adj[node][i] != prev){
            otherEnd[ind[node][i]] = adj[node][i];
            dfs(adj[node][i], node, level + 1);
            subsz[node] += subsz[adj[node][i]];
        }
    }
}

int getkth(int p, int q, int k){
    int lca = LCA(p, q), d;
    if (lca == p){
    d = depth[q] - depth[p] + 1;
    swap(p, q);//we want p to be at higher depth.....
so swap p and q if p is at lower depth i.e. it is the lca
        k = d - k + 1;//decide 'k' accordingly i.e. k will now become total
        distance minus k as we have now change our p(which was originally q)
    }
```

```
    else if (q == lca);//do nothing if q is lca
    //case when neither p and q are lca
    else{
        d = depth[p] + depth[q] - 2 * depth[lca] + 1;
        /*
        d denotes the total dist between the nodes p and q. it will be =
        dist(p,lca) + dist(lca,q) - 1
        = (depth[p] - depth[lca] + 1) + (depth[q] - depth [lca] + 1) - 1
        = depth[p] + depth[q] - 2 * depth[lca] + 1
        */
        if (k > depth[p] - depth[lca] + 1){//case when 'k' will be between lca
        and q i.e. dist b/w lca and 'p' is less than k
        k = d - k + 1;//change 'k' accordingly
        swap(p, q);//swap p and q as we want to calculate the dist from 'p'
        only.
        }
    }
    //Now we have set starting node as 'p' and changed k accordingly such that the
    kth node between 'p'
    //and 'q' will always lie between 'p' and lca(p,q) at a dist 'k' from p
    //Also dist(p,lca) > k
    //cout << "p : " << p << " q : " << q << " k : " << k << " lca : " << lca
    << endl;
    k--;//decrement k as k = 1 will indicate p itself.
    for (int i = LN - 1; i >= 0; i--){
        if ((1 << i) <= k){//if k is greater than or equal to 2^i than we can
        move up by that much nodes
            p = pa[p][i];//p will become 2^i th ancestor of p
            k -= (1 << i);//we will move 2^i nodes up and k will be decreased
            by that amount
        }
    }
    return p;
}
```

```
int main(){
    int t, a, b, c, x, y;
    scanf("%d", &t);
    while (t--){
        scanf("%d", &n);
        for (int i = 0; i < n; i++){
            ind[i].clear();
            adj[i].clear();
            costs[i].clear();
            chainHead[i] = -1;
            for (int j = 0; j < LN; j++)pa[i][j] = -1;
        }
        for (int i = 1; i < n; i++){
            scanf("%d %d %d", &a, &b, &c);
            --a; --b;
            adj[a].push_back(b);
            adj[b].push_back(a);
            costs[a].push_back(c);
            costs[b].push_back(c);
            ind[a].push_back(i - 1);
            ind[b].push_back(i - 1);
        }
        //cout << "chaiHead : "; for (int i = 0; i < n; i++)cout << chainHead[i]
        << " "; cout << endl;
        chainNo = 0;
        dfs(0, -1, 0);
        HLD(0, -1, -1);
        //cout<<"arr : ";for (int i = 0; i <= sz; i++)cout << arr[i] << " "; cout
        << endl;
        build(1, 0, sz);
        for (int j = 1; j < LN; j++){
            for (int i = 0; i < n; i++){
                if (pa[i][j - 1] != -1)pa[i][j] = pa[pa[i][j - 1]][j - 1];
            }
        }
```

```
        //cout << "sz : " << sz << endl;
        //for (int i = 0; i <= chainNo; i++)cout << "chain : " << i << " " <<
        chainHead[i] << endl;
        char ch[20];
        while (1){
            scanf("%s", ch);
            if (ch[1] == 'O')break;
            if (ch[0] == 'D'){
                scanf("%d %d", &a, &b);
                printf("%d\n", query(a-1, b-1));
            }
            else if (ch[0] == 'K'){
                scanf("%d %d %d", &a, &b, &c);
                printf("%d\n", getkth(a - 1, b - 1, c) + 1);
            }
        }
    }
    return 0;
}
```

Important Resources

I would suggest all the readers to go through anudeep's bolg as most of the content have been taken from there:

https://blog.anudeep2011.com/heavy-light-decomposition/

Problems to try

SPOJ – QTREE – http://www.spoj.com/problems/QTREE/

SPOJ – QTREE2 – http://www.spoj.com/problems/QTREE2/

SPOJ – QTREE3 – http://www.spoj.com/problems/QTREE3/

SPOJ – QTREE4 – http://www.spoj.com/problems/QTREE4/

SPOJ – QTREE5 – http://www.spoj.com/problems/QTREE5/

SPOJ – COT – http://www.spoj.com/problems/COT/

SPOJ – COT2 – http://www.spoj.com/problems/COT2/

SPOJ – COT3 – http://www.spoj.com/problems/COT3/

SPOJ – GOT – http://www.spoj.com/problems/GOT/

SPOJ – GRASSPLA – http://www.spoj.com/problems/GRASSPLA/

SPOJ – GSS7 – http://www.spoj.com/problems/GSS7/

CODECHEF – RRTREE – http://www.codechef.com/problems/RRTREE

CODECHEF – QUERY – http://www.codechef.com/problems/QUERY

CODECHEF – QTREE – http://www.codechef.com/problems/QTREE

CODECHEF – DGCD – http://www.codechef.com/problems/DGCD

CODECHEF – MONOPLOY –

http://www.codechef.com/problems/MONOPLOY

SELF STUDY NOTES

SELF STUDY NOTES