Erick Vargas

Interview notes

Contents

1 A little bit of theory

1.1 Data structures

1.1.1 ¿What is a data structure?

A simple form to describe this is: a data structure is a container to store data in a specific layout. This layout allow us to make efficient our data structure in some operations but inefficient in others.

1.1.2 ¿Why do we need data strucures?

Depending of different scenarios, data needs to be stored in a specific format, to solve this we need to know the different types of data structures.

1.1.3 Basic data structures

Nowadays there are a lot of types of data structures but here we have a little list of the commonly used data structures.

- Arrays
- Stacks
- Queues
- Linked list
- Trees
- Graphs
- Tries
- Hash tables

Arrays

An array is the simplest used data structure and other data structures can be made using arrays, like stacks and queues. Most of the programming languages index the first element in zero.

We have two types of arrays:

- One-dimensional arrays
- Multi-dimensional arrays

Basic operations on arrays

1. Insert: Inserts an element at given index
2. Get: returns the element at given index
3. Delete: Deletes an element at given index
4. Size: Get the total number of elements in an array

Commonly asked array interview questions

1. Find the second minimum element of an array
2. First non-repeating integers in the array
3. Merge two sorted arrays
4. Rearrage positive and negative values in an array

Stacks

This data structure has a lot of applications, for example when we use the undo command. A stack works like a pile of something for example books. In this example each book are placed in a vertical order. If we have a stack of books we can not get a book placed in the middle, first ypu need to remove all the books on the top to get it. This behaviour is know as LIFO (Last In First Out)

Basic operations

1. Push: inserts an element of the top
2. Pop: returns the top element, after removing from the stack
3. isEmpty: returns true if the stack is is empty
4. Top: returns the top element without removing from the stack

Commonly asked stack interview questions

1. Evalue postfix expression using a stack
2. Sort values in a stack
3. Check balanced parentheses in an expression

Queues

Similar to stack, queue is another data structure that stores the element in a sequential manner. The difference is that instead of use LIFO method, queue implements FIFO method (First In First Out)

An example of a queue in real life is a line of people waiting to buy a ticket. If a new person comes, he or she will join the line from the end, not from the start and the person first person (start) will be the first to buy the ticket and then he or she leaves the line.

Basic operations

1. Enqueue: inserts an element at the end of the queue
2. Dequeue: removes an element from the start of the queue
3. isEmpty: returns true if the queue is empty
4. top: returns the first element of the queue

Commonly asked queue interview questions

1. Implement a stack using a queue
2. Reverse first k elements of a queue
3. Generate binary numbers from 1 to n using a queue

Linked list

A linked list is another important data structure that is very similar to an array but differs in memory allocation, internal structure and how basic operations of insertion and deletion are carried out.

A linked list is like a chain of nodes, where each node contains information like data and a ponter to the succeding node in the chain. There's a head pointer, which points to the first element of the linked list, and if the list is empty then it simply points to null or nothing.

Linked lists are used to implement file systems, hash tables, and adjacency lists.

Also exists different types of linked lists:

- Singly linked lists
- Doubly linked lists

Basic operations

1. InsertAtEnd: inserts given element at the end of the linked list
2. InsertAtHead: inserts given element at the start/head of the linked list
3. Delete: deletes given element from the linked list
4. DeleteAtHead: deletes first element of the linked list
5. Search: returns the given element from a linked list
6. isEmpty: returns true if the linked list is empty

Commonly asked linked list interview questions

1. Reverse a linked list
2. Detect loop in a linked list
3. Return N-th node from the end in a linked list
4. Remove duplicates from a linked list

Graphs

A graph is a set of nodes that are connected to each other in the form of a network. Nodes are also called vertices. A pair(x, y) is called an edge, which indicates that vertex x is conected to vertex y. An edge may contain weight/cost, showing how much cost is required to traverse from vertex x to y.

Also we have different types of graphs:

- Undirected graphs $\longrightarrow$
- Directed graphs $\longleftrightarrow$

We can represent graphs in two forms:

- Adjacency matrix
- Adjacency list

Finally we have two common graph traversing algorithms:

- Breadth First Search (BFS)
- Depth First Search (DFS)

Commonly asked graph interview questions

1. Implement Breath and Depth First Search
2. Check if a graph is tree or not
3. Count number of edges in a graph
4. Find the shortest path between two vertices

Trees

A tree is a special type of graph the difference is that in a tree a cycle cannot exist. This data structure also use vertices (nodes) and edges that connect them.

We have different types of trees:

- N-ary tree
- Balanced tree
- Binary tree
- Binary search tree
- AVL tree
- Red-Black tree
- 2-3 tree

But the most used trees are binary tree and binary search tree.

Commonly asked tree interview questions

1. Find the heigh of a binary tree
2. Find the k-th maximum value in a binary search tree
3. Find nodes at ”k” distance from the root
4. Find ancestor of a given node in a binary tree

Trie

Trie is also known as ”prefix trees”, is a data structure very similar to a tree which proves to be quite efficient for solving problems releted to strings. It is commonly used to search words in a dictionary, providing auto suggestions in a search engine, and even for IP routing

Commonly asked trie interview questions

1. Count total number of words in Trie
2. Print all words stored in Trie
3. Sort elements of an array using trie
4. Form words from a dictionary using trie
5. Build a T9 dictionary

Hash table

Hashing is a process used to uniquely identify objects and store each object at some pre-calculated unique index called its ”key”. So, the object is stored in the form of a ”key-value” pair, and the collection of such items called a ”dictionary”. Each object can be searched using that key. There are a lot of data structures based on hashing, but the commonly used data structure is hash table.

Hash tables are commonly implemented using arrays and the performance of hashing data structure depends of three factors:

- Hash function
- Size of the hash table
- Collision handling method

Commonly asked hash table interview questions

1. Find symetric pairs in an array
2. Trace complete path of a journey
3. Find if an array is a subset of another array
4. Check if given arrays are disjoint

2 Basic data structures

2.1 Arrays

We already known what is an array, and now we need to know how to solve some problmes using this data structure. First of all we need to know how to use arrays in a programming language, in my case I used to use C++ language because is very easy to understand and is faster than a lot of programming languages.

In C++ we can declarate an array using the next piece of code.

```
data_type name_of_our_array[ size_of_our_array ]
```

Imagine that you need an array of integer numbers with length ten, we need to write the next in out C++ programm.

```
#include<bits/stdc++.h>

using namespace std;

int main(){
    //As you can see this is the form of create an array of
        integers with length ten
    int array[10];
    return 0;
}
```

If we need to set a value in a element of our array we just need to put the index of our element, but consider that arrays indexes starts with 0

```
#include <bits/stdc++.h>

using namespace std;

int main(){

    int array[10];
    array[0] = 1;
    array[1] = 2;
```

```
        array[3] = 3;
        .
        .
        .
        array[9] = 10;
        return 0;
    }
```

If we want to get the value of an element of our array we need just to indicate the index of our element, something like this:

```
    #include <bits/stdc++.h>

    using namespace std;

    int main(){
        int array[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
        cout << array[2] << endl;
        return 0;
    }

    Output: 3
```

2.1.1 Some problems

Problem 01

Find the second minimum element of an array For this problem maybe we think that sort the elements of the array can solve this problem efficiently. The complexity of do this is nlog(n) if we sort the elements using merge sort, and the code to do this (if we do not implement the merge sort) is too short, something like this:

```
    #include <bits/stdc++.h>

    using namespace std;

    int main(){

        int n, min_01, min_02 = INT_MAX;
        vector<int> numbers;
        cin >> n;
        numbers.resize(n, 0);
        for(int i = 0; i < n; i++)
            cin >> numbers[i];

        sort( numbers.begin(), numbers.end() );

        min_01 = numbers[n - 1];

        for(int i = n - 2; i >= 0; i--){
            //Searching an element bigger than the minimum (it
                will be different that the minimum)
            if(min_01 != numbers[i]){
                min_02 = numbers[i];
                break;
```

```
            }
        }

        cout << ( min_02 == INT_MAX ? "none" : to_string(min_02)
             ) << endl;
        return 0;
    }
```

But, can we made something more efficient? And the aswer is yes, and is very simple. We only need two variables one to know the minimum element and other to know the second minimum element (if exists). The solution is move inside the array searching by the minimum element, if we find an element smaller than the actual minimum we find a new candidate to minimum and the last value of the minimum now is the second minimum element of the array. It works but what happen if all the elements are equal? If we do this we never find a solution for this problem so we need to compare if the i-th element is different of the minimum current element and if this element is bigger than the second minimum element.

```
#include <bits/stdc++.h>

using namespace std;

int main(){

    int n;
    int min_01 = INT_MAX, min_02 = INT_MAX;
    vector<int> numbers;
    cin >> n;
    numbers.resize(n, 0);
    for(int i = 0; i < n; i++)
        cin >> numbers[i];

    for(int i = 0; i < n; i++){
        if(numbers[i] < min_01){
            min_02 = min_01;
            min_01 = numbers[i];
        }
        //If the elements are in ascending order and the i-
            th number is different of the minimum
        else if (numbers[i] < min_02 && numbers[i] != min_01
            )
            min_02 = numbers[i];
    }

    cout << (min_02 == INT_MAX ? "none" : to_string(min_02))
         << endl;
    return 0;
}
```

Problem 02

First non-repeating integers in the array To solve this problem we can compare each element with the rest of the array looking for the same number, if we do not

find another number we find the solution, but made this is $O(n^2)$ something like this:

```
#include<bits/stdc++.h>

using namespace std;

int main(){
    int n;

    vector<int> numbers;
    cin >> n;
    numbers.resize(n, 0);
    for(int i = 0; i < n; i++)
        cin >> numbers[i];

    for(int i = 0; i < n; i++) {
        for(int j = i + 1; j < n; j++) {
            if(numbers[i] == numbers[j])
                break;
            if(j == n - 1){
                cout << numbers[j];
                exit(0);
            }
        }
    }
    return 0;
}
```

We can made the same more efficiently if we use something else for example hasing the elements inside our array. To make this easier we use a map< key, value > which is a data structure of C++, if you want to see more about this I recommend show the reference of this data structure. For this problem our key is an integer, this integer is the i-th value of our array called numbers, and the value is the number of appearances of this number. Finally if this value is equal to one it means that we find our answer. You can see the solution in the next code:

```
#include <bits/stdc++.h>

using namespace std;

int main(){
    int n;
    vector<int> numbers;
    map<int, int> repeated;
    cin >> n;
    numbers.resize(n, 0);
    for(int i = 0; i < n; i++)
        cin >> numbers[i];

    for(int i = 0; i < n; i++)
        repeated[ numbers[i] ]++;


    for(auto e : repeated ){
        if(e.second == 1){
```

```
                cout << e.first << endl;
                break;
            }
        }

        return 0;
    }
```

Problem 03

Merge two sorted arrays Solve this problem is not difficult just you need to check which element between our two arrays is smaller than the other, and we will do this until one of our indexes is in the limit. Finally we need to check if one of our arrays was not checked completely.

```
#include <bits/stdc++.h>

using namespace std;

int main(){

    int m, n;
    vector<int> M, N, ans;

    cin >> m >> n;
    M.resize(m, 0);
    N.resize(n, 0);

    for(int i = 0; i < m; i++)
        cin >> M[i];

    for(int i = 0; i < n; i++)
        cin >> N[i];

    int i = 0, j = 0;
    while( i < m && j < n ){
        if(M[i] < N[j]){
            ans.push_back( M[i++] );
        } else {
            ans.push_back( N[j++] );
        }
    }

    for ( ; i < m; i++)
        ans.push_back( M[i] );

    for( ; j < n; j++)
        ans.push_back( N[j] );

    for( auto e : ans )
        cout << e << " ";

    cout << endl;
    return 0;
}
```

Problem 04

Rearrage positive and negative values in an array

2.2 Stacks

We already know what is a stack, but how can we made a stack using code? First of all remember that we already defined the basic operations of a Stack, so we have some clues about what we need to code.

For me is a little bit easier to programm a data structure if a have an idea or I imagine the "form" of it. In case of Stacks I imagine something like this:

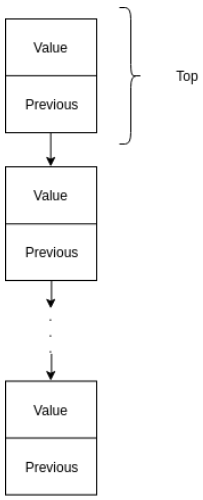


Figure 2.1: Diagram of a Stack

As you can see our Stack have n elements, where the last element is called top. Also each element of our stack (lets call each element as node) has two things inside, the first one is value and the second is previous where previous is a pointer to the element below of the current node.

To our stack we have two elements, as I mentioned before, we have a pointer called Top, this pointer has the last element of our Stack. The second element is size, it counts the number of elements inside our stack.

If we code the stack with all the operations that I mentioned in the section "Basic data strcutures" we have the next as a result:

```
struct Node{
    Node *previous;
    int value;

    Node(int _value){
        value = _value;
        previous = NULL;
```

```
        }
    };

    struct Stack {

        Node *top;
        int size;

        Stack(){
            top = NULL;
            size = 0;
        }

        void Push(int value){
            Node *node = new Node( value );
            if( size == 0 )
                top = node;
            else {
                node -> previous = top;
                top = node;
            }
            size++;
        }

        int Pop(){
            int v_return = INT_MIN;
            if( size > 0){

                v_return = top -> value;
                top = top -> previous;
                size--;

            } else {
                cout << "\nStack is empty \n";
            }

            return v_return;
        }

        bool IsEmpty(){
            return (size == 0);
        }

        int Top(){
            return top -> value;
        }

        int GetSize(){
            return size;
        }

    };
```

In case of competitive programming we can not program a Stack and then use it, beacuse it takes a lot of time, so C++ has a implementation of this data structure in ”stack” library, and we can use it using the word Stack and defining the data type of our Stack, something like stack < data_type > or show the example

below:

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

int main(){
    //Creation of an Stack of int
    stack<int> Stack;
    //Adding an element to our stack
    Stack.push(0);
    //Gets the number of elements of our Stack
    Stack.size();
    //Gets the value of the top element
    Stack.top();
    //The same as top but also removes the top element
    Stack.pop();
    return 0:
}
```

2.2.1 Some problems

Problem 01

Evalue postfix expression using a stack

```
#include <bits/stdc++.h>

using namespace std;

int main(){
    string s;
    int number_01 = 0, number_02 = 0;
    stack< int > Stack;
    cin >> s;

    for( auto c : s ){
        if( isdigit(c) ){
            Stack.push( c - '0' );
        } else {

            number_01 = Stack.top();
            Stack.pop();
            number_02 = Stack.top();
            Stack.pop();

            if ( c == '+')
                Stack.push( number_02 + number_01 );
            else if ( c == '-' )
                Stack.push( number_02 - number_01 );
            else if ( c == '*' )
                Stack.push( number_02 * number_01 );
            else if ( c == '/' )
                Stack.push( number_02 / number_01 );
        }

    }
    cout << Stack.top() << endl;
```

```
    return 0;
}
```

Problem 02

Sort values in a stack

```
#include <bits/stdc++.h>

using namespace std;

stack< int > StackSort(stack<int> &Stack){

    stack<int> AStack;

    while( !Stack.empty() ){
        int aux = Stack.top();
        Stack.pop();

        while( !AStack.empty() && AStack.top() > aux ){

            Stack.push( AStack.top() );
            AStack.pop();

        }

        AStack.push( aux );
    }
    return AStack;
}

int main(){
    stack<int> Stack;
    int n, v;
    cin >> n;
    while(n--){
        cin >> v;
        Stack.push(v);
    }

    Stack = StackSort( Stack );

    while( !Stack.empty() ){
        cout << Stack.top() << endl;
        Stack.pop();
    }

    return 0;
}
```

Problem 03

Check balanced parentheses in an expression

```
#include <bits/stdc++.h>
```

```cpp
using namespace std;

int main(){
    stack< char > Stack;
    string s;
    cin >> s;

    for( auto c : s ){
        if(c == '('){
            Stack.push(c);
        } else {
            Stack.pop();
        }
    }

    cout << ( Stack.size() == 0 ? "Balanced" : "Not balanced" )
        << endl;

    return 0;
}
```

2.3 Queues

As the Stack we already know the basic operations of a Queue, so we have an idea about what we go to program, but what is the structure of a Queue? As I mentioned before maybe show a draw or schema is better to understand what we need to program so I have maked a little draw of how I imagine a Queue.

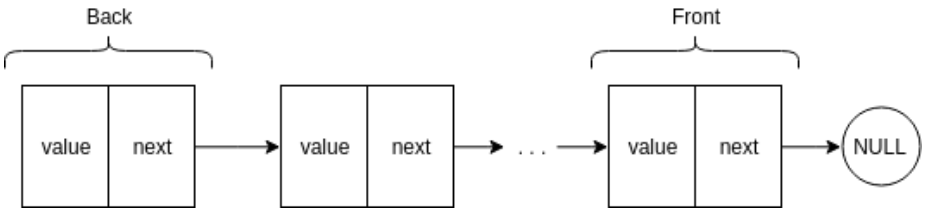


Figure 2.2: Diagram of a Queue

As you can see a stack has a ”back” and ”front” element where ”back” is the last element of the queue and ”front” is the first element. Each element (lets call them as nodes) has a value and a pointer to the next element, if we wrote this using code we have the next result:

```cpp
    #include <bits/stdc++.h>

    using namespace std;

    struct Node{
        int value;
        Node *next;
```

```
        Node( int _value ){
            value = _value;
            next = NULL;
        }
    };

    struct Queue{

        Node *back;
        Node *front;
        int size;

        Queue(){
            back = front = NULL;
            size = 0;
        }

        void Enqueue( int value ){
            Node *node = new Node( value );
            if( size == 0 )
                back = front = node;
            else {
                node -> next = back;
                back = node;
            }

            size++;
        }

        void Dequeue(){
            Node *aux = back;
            if( size > 1 ){

                while( aux -> next != NULL )
                    aux = aux -> next;

                aux -> next = NULL;
                front = aux;

            } else if( size == 1 )
                front = back = NULL;
            else
                return;
            size--;
        }

        bool IsEmpty(){
            return (size == 0);
        }

        int Top(){
            return (front -> value);
        }

        int GetSize(){
            return size;
        }
```

```
        void print(){
            cout << endl;
            Node *aux = back;
            while( aux != NULL ){
                cout << aux -> value << " ";
                aux = aux -> next;
            }
            cout << endl;
        }
    };
```

2.3.1 Problems

Problem 01

Implement a stack using a queue To solve this problem we need to know what is a stack, and I already explain this data structure, so we only need to remember this behaviour to solve this problem.
First of all I have created a struct with all the basic operations of a Stack, but instead of nodes this struct contains a Queue. Our operation of pop does not have any problem but oir operation of push needs to be changed, so invert the elements inside our Queue is a goos solution for this problem, and we can do this using two queues. To solve the problem of push an element as I mentioned before we need two queues, the first one contains the elements of our stack, and the second we go to use it when we need to add a new element, the procedure is the following: First add the new element to our second queue then make a dequeue of our first queue until this queue be empty, finally we made the second queue as our first queue. In code we have the next as a result:

```
    #include <bits/stdc++.h>

    using namespace std;

    struct Stack{

        int size;
        queue<int> Queue;

        Stack(){
            size = 0;
        }

        void Push( int val ){
            queue<int> result;
            result.push( val );
            while( !Queue.empty() ){
                result.push( Queue.front() );
                Queue.pop();
            }
            Queue = result;
            size++;
        }
```

```
        void Pop(){
            if( size > 0 ){
                Queue.pop();
                size--;
            } else {
                cout << "\nStack doesn't have elements\n";
            }
        }

        int Top(){
            return Queue.front();
        }

        int GetSize(){
            return size;
        }

        bool IsEmpty(){
            return ( size == 0 );
        }
    };
```

Problem 02

Reverse first k elements of a queue First of all you need to know that if you are in an interview you need to ask if you can use another data structure, and other constraints, in this case imagine that you can use another data structure apart of queues, so if we need to reverse elements we have a data structure that can help us to do this easily and it is a stack. We need to pop the first k elements of our queue and store this elements in a stack, then make a pop of our stack and store the elements in our queue, and this elements now are reversed because the behaviour of a stack help us to do this. Finally we need to "nove" n - k times the elements inside our queue and thats all, in code we have the next as a result:

```
    #include <bits/stdc++.h>

    using namespace std;

    int main(){
        queue<int> Queue;
        stack<int> Stack;
        int n, v, k;
        cin >> n;
        while(n--){
            cin >> v;
            Queue.push(v);
        }

        cin >> k;

        for (int i = 0; i < k; i++){
            Stack.push( Queue.front() );
            Queue.pop();
        }
```

```
        while( !Stack.empty() ){
            Queue.push( Stack.top() );
            Stack.pop();
        }

        for(int i = 0; i < Queue.size() - k; i++){
            Queue.push( Queue.front() );
            Queue.pop();
        }

        while( !Queue.empty() ){
            cout << Queue.front() << " ";
            Queue.pop();
        }

        return 0;
    }
```

Problem 03

Generate binary numbers from 1 to n using a queue This problem is very interesting and I consider that is better to make an analysis using some draws, imagine that you want to generate binary numbers from 1 to 5, if we do this first we need to add a '1' to our queue (our queue needs to be of strings) then we have two options add a '1' to this string or '0' it generates a new binary string, for example: Let's imagine that we want generate binary strings from 1 to 7, we have a procees like the next:

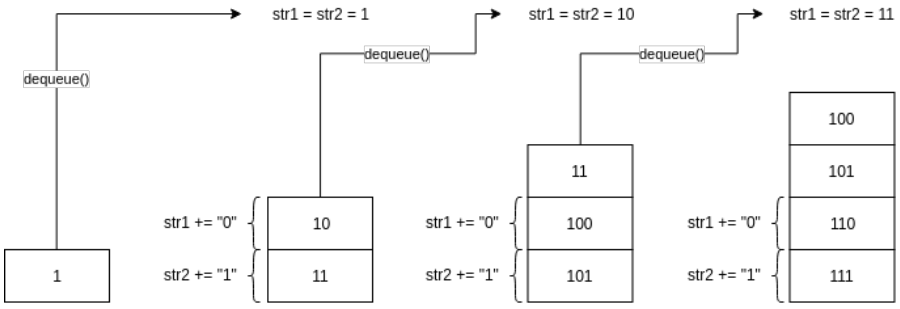


Figure 2.3: Problem 03 diagram

As you can see we only need to to enqueue a new string using the top of our queue and concatenating an extra character first a zero and then a one, and we made the same as many times as we need. If we code this we have the next as a result:

```
#include <bits/stdc++.h>

using namespace std;
```

```
int main(){

    queue<string> Queue;
    string str1, str2;
    int n;
    cin >> n;
    Queue.push("1");
    for(int i = 0; i < n; i++){
        str1 = Queue.front();
        str2 = Queue.front();
        Queue.pop();
        cout << str1 << endl;
        str1 += "0";
        str2 += "1";
        Queue.push(str1);
        Queue.push(str2);
    }
    return 0;
}
```

2.4 Singly linked lists

We already know the behaviour of a linked list, and if we are progrogramming a Singly linked list it means that we need just information about the next node and the value of the current node, so we need to use a struct called Node with a pointer to the next element, and a int value (the value of the current node), if we code this (and knowing the basic operations, explained in the section Basic Structures) we have the next as a result:

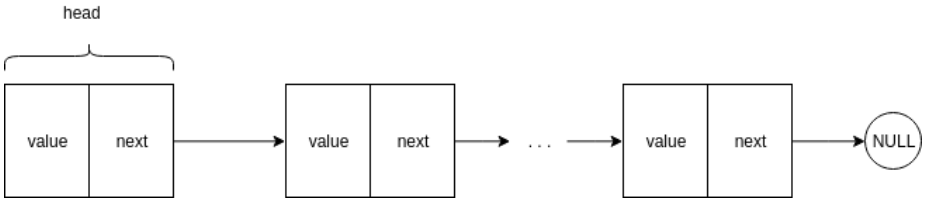


Figure 2.4: Diagram of a singly linked list

```
struct Node{

    Node *next;
    int value;

    Node( int _value ){
        value = _value;
        next = NULL;
    }

};
```

```
typedef struct LinkedList{

    Node *tail;
    Node *head;
    int size;

    LinkedList(){
        tail = head = NULL;
        size = 0;
    }

    void insertAtTail(int value){
        Node *node = new Node( value );
        if( size > 0 ){
            tail -> next = node;
            tail = node;
        } else {
            tail = head = node;
        }
        size++;
    }

    void insertAtHead(int value){
        Node *node = new Node( value );
        if( size > 0 ){
            node -> next = head;
            head = node;
        } else {
            head = tail = node;
        }
        size++;
    }

    void Delete(int i){
        if( i < size && i >= 0 ){
            Node *aux = head;
            //If we want to delete the first element
            if( i == 0 )
                head = head -> next;
            //If we want to delete the last element
            else if ( i == size ){

                for( int j = 0; j <= size - 1; j++ )
                    aux = aux -> next;

                aux -> next = NULL;
                tail = aux;

            } else {
                for( int j = 0; j < i - 1; j++)
                    aux = aux -> next;
                aux -> next = aux -> next -> next;
            }
            size--;
        }
    }
```

```
        void DeleteAtHead(){
            if( size == 1 )
                head = tail = head -> next;
            else
                head = head -> next;
            size--;
        }

        int Search(int i){

            if( i < size ){
                Node *aux = head;
                for(int j = 0; j < i; j++)
                    aux = aux -> next;
                return aux -> value;
            }

            return INT_MIN;
        }

        bool isEmpty(){
            return ( size == 0 );
        }

        int getSize(){
            return size;
        }

        void print(){
            Node *aux = head;
            while( aux != NULL ){
                cout << aux -> value << "\t";
                aux = aux -> next;
            }
            cout << endl;
        }
    } LinkedList;
```

2.4.1 Problems

Problem 01

Reverse a linked list To solve this problem you need to remember that in a singly linked list each node just knows information about the next element, but no more, so we need to move in the linked list using three variables, the first one to know information about the previous element, the second to know the current element and the third to know the next element of the current element (before to recobnect current -> next) so if we do this we can reconnect the current -> next with the previous element, and now the previous has a new value as the current element, the current element now is the next element and the next element now is the current -> next. I think that is easier to understand this if we code this problem, but just imagine that you need to complete a function, if you want to show if you

code works you can try to solve **this problem** on HackeRank

```
#include <bits/stdc++.h>

using namespace std;

struct Node{
    int value;
    Node *next;
};

Node *Reverse(Node *head){

    if(head -> next == NULL || head == NULL)
        return head;

    node *previous = NULL;
    Node *current = head;
    Node *next = current -> next;

    while( next != NULL ){
        current -> next = previous;
        previous = current;
        current = next;
        next = current -> next;
    }

    current -> next = previous;

    return current;
}
```

Problem 02

Detect a loop in a linked list To solve this problem we can use the history of the hare and the turtle, as we know the hare is too fast and the turtle too slow, let's say that the hare is two times faster than the turtle, if we have a cycle both will meet at some point, if we code this the turtle is a ponter to head and moves using turtle = turtle -> next and the hare is hare = head -> next -> next, and moves hare = hare -> next -> next

Problem 03

Return N-th node from the end in a linked list To solve this problem we can notice than we can make a reverse of the linked list and move node to node n times and we get the element, just you need the smallest value can n can be. Also you can solve this problem using a stack, but I think that is better if we remember how to reverse a linked list. Here you have the solution:

```
#include <bits/stdc++.h>

using namespace std;

struct Node{
```

```
        Node *next;
        int value;

        Node(int _value){
            value = _value;
            next = NULL;
        }
    };

    Node *ReverseLinkedList(Node *head){
        Node *previous = NULL;
        Node *current = head;
        Node *next = current -> next;
        while( next != NULL ){

            current -> next = previous;
            previous = current;
            current = next;
            next = current -> next;

        }
        current -> next = previous;
        return current;
    }

    int main(){

        int n, v;
        Node *head, *aux;
        cin >> n;

        cin >> v;
        head = new Node(v);
        aux = head;
        n--;
        while(n--){
            cin >> v;
            aux -> next = new Node(v);
            aux = aux -> next;
        }
        cin >> n;

        //Reverse a linked list
        aux = ReverseLinkedList( head );

        for(int i = 0; i < n; i++){
            aux = aux -> next;
        }

        cout << aux -> value << endl;

        return 0;
    }
```

Problem 04

Remove duplicates from a linked list To solve this problem we need something to help us, maybe we can use a bucket, but maybe we do not know the biggest number of an element inside the linked list, and also it makes to use a lot of memory so we can use hash to make this efficient in time and memory, in this case I gonna use an unordered set, it is a contianter that uses a hash to store efficientlly data, if you need more information about this you can show the **CPP reference** and here is the solution for this problem.

```
#include <bits/stdc++.h>

using namespace std;

struct Node{
    Node *next;
    int value;
    Node(){
        next = NULL;
    }
};

Node *RemoveDuplicates( Node *head ){
    unordered_set<int> duplicates;
    Node *current = head;
    Node *previous = NULL;
    while(current -> next != NULL){

        if( duplicates.find( current -> value ) !=
            duplicates.end() ){
            previous -> next = current -> next;
        } else {
            duplicates.insert( current -> value );
            previous = current;
        }
        current = current -> next;
    }

    return head;
}

void printList( Node *head ){
    Node *aux = head;
    cout << endl;
    while( aux -> next != NULL ){
        cout << aux -> value << " ";
        aux = aux -> next;
    }
    cout << endl;
}

int main(){
    int n, v;
    Node *head = new Node();
    Node *aux = head;
```

```
        cin >> n;
        while(n--){
            cin >> aux -> value;
            aux -> next = new Node();
            aux = aux -> next;
        }

        printList( head );

        aux = RemoveDuplicates( head );

        printList( aux );

        return 0;
    }
```

2.5 Doubly linked lists

The doubly linked list is the same than singly linked list but as the name tell us we have another pointer to the previous element, something like this:

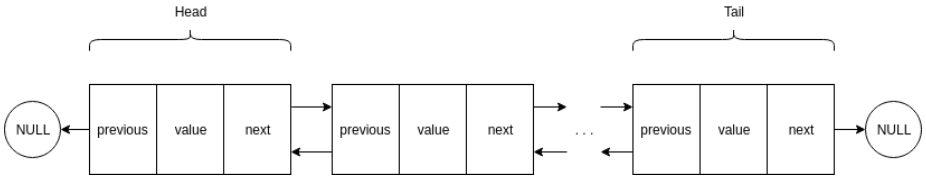


Figure 2.5: Diagram of a doubly linked list

And here you are my own implementation of this data structure, you can nottice that is very similar to a singly linked list:

```
#include <bits/stdc++.h>

using namespace std;

struct Node{

    Node *previous;
    Node *next;
    int value;

    Node(int _value){
        previous = next = NULL;
        value = _value;
    }

};

struct LinkedList{
```

```
        int size;
        Node *head;
        Node *tail;

        LinkedList(){
            size = 0;
            head = tail = NULL;
        }

        void InsertAtTail(int value){
            Node *node = new Node( value );
            if(size == 0)
                head = tail = node;
            else {
                tail -> next = node;
                node -> previous = tail;
                tail = node;
            }
            size++;
        }

        void InsertAtHead(int value){
            Node *node = new Node( value );
            if(size == 0)
                head = tail = node;
            else {
                node -> next = head;
                head -> previous = node;
                head = node;
            }
            size++;
        }

        void Delete(int index){
            Node *aux = head;
            if( index < size && index >= 0){
                int middle = size >> 1;
                for(int i = 0; i < index; i++)
                    aux = aux -> next;

                if(index == 0){
                    head = head -> next;
                    head -> previous = NULL;
                } else if ( index == size - 1){
                    tail = tail -> previous;
                    tail -> next = NULL;
                } else {
                    aux -> previous -> next = aux -> next;
                    aux -> next -> previous = aux -> previous;
                    aux = NULL;
                }
            } else {
                cout << "\nInvalid index" << endl;
                return;
            }
            size--;
        }
```

```
        void DeleteAtHead(){
            head = head -> next;
            head -> previous = NULL;
        }

        void DeleteAtTail(){
            tail = tail -> previous;
            tail -> next = NULL;
        }

        int Search(int index){
            Node *aux = head;
            if(index < size){

                for(int i = 0; i < index; i++)
                    aux = aux -> next;

                return aux -> value;
            } else {
                cout << "\nIndex doesn't exists\n";
            }

            return INT_MIN;
        }

        int getSize(){
            return size;
        }

        bool IsEmpty(){
            return ( size == 0 );
        }

        void print(){
            Node *aux = head;
            cout << endl;
            while( aux != NULL ){
                cout << aux -> value << " ";
                aux = aux -> next;
            }
            cout << endl;
        }

    };
```

2.6 Graphs

For me it is a ”new” topic, I know a little bit about it but I dont know as much as I want so a decided to get a deeper knowledge about graphs.

2.6.1 Graph theory

Is the mathematical theory of the properties and applications of graphs (networks)

To understand a little bit more this imagine that you have a set of accesories to wear. Things to put in your head, in ypur body and in your legs. So we need to solve the next question, How many different sets of clothing can I make by choosing an article from each category? Using mathematics we can solve this problem but a graph can help us to solve this visually.

Other common problem is how many friends does the person X have? or how many degrees of separation are there between person X and Y?

2.6.2 Types of graphs

To solve particular problems is necessarilyto know the type of graph that we need to code, and we have different types of graphs each can help us to solve something in particular.

Undirected graph

It's the simplest type of graph. An undirected graph is a graph which edges have no orientation. The edge (u, v) is identical to the edge (v, u). Maybe is better if we have a draw:

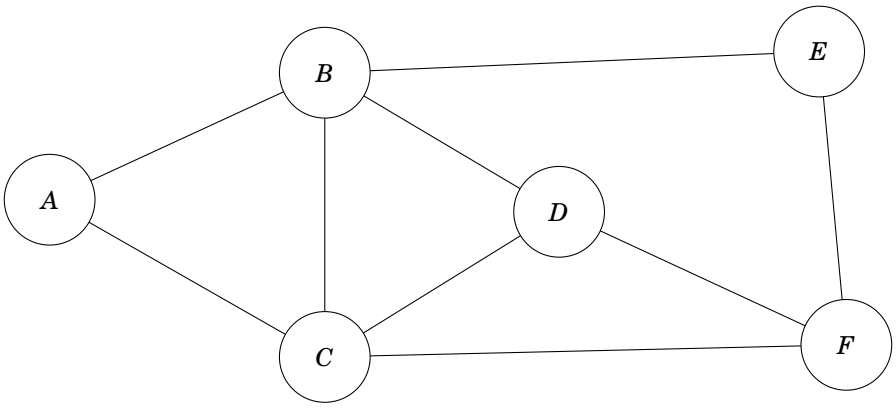


Figure 2.6: Undirected graph diagram

,

Directed graph

A directed graph or digraph is a graph in which edges have orientations. For example, the edge (u, v) is the edge from node u to v.

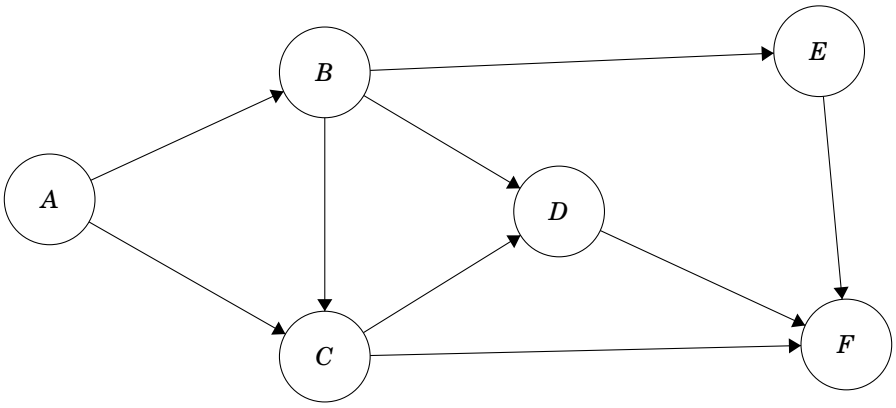


Figure 2.7: Directed graph diagram

2.6.3 Weighted graphs

Many graphs can have edges that contain a certain weigh to represent an arbitrary value such a cost, distance quantity, etc... Something important to know is that an edge will be represented as a triplet (u, v, w) and specify whether the graph is directed or undirected.

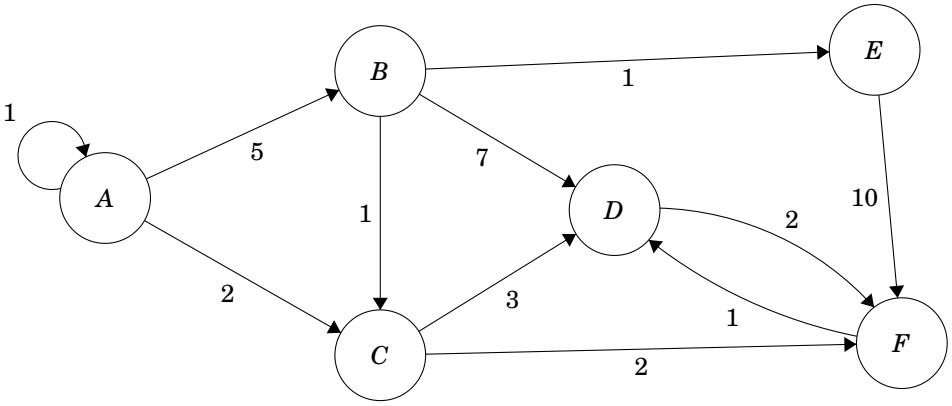


Figure 2.8: Weighted graph diagram

2.6.4 Directed acyclic graphs (DAGs)

DAGs are directed graphs with no cycles. These graphs play an important role in representing structures with dependencies. Several efficient algorithms exits to operates on DAGs. **IMPORTANT:** All out-trees are DAGs but not all DAGs are out-trees.

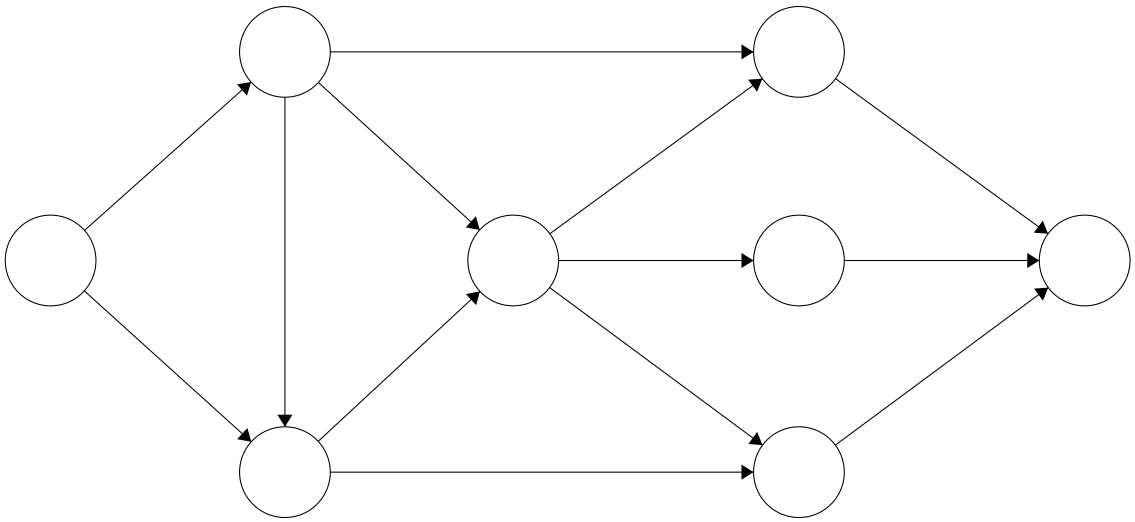


Figure 2.9: Directed Acyclic graph diagram

2.6.5 Bipartite graph

A bipartite graph is one whose vertices can be split into two independent groups U, V such that every edge connects betweens U and V.

Other definitions exist such as: the graph is two colourable or there is no odd lenght cycle.

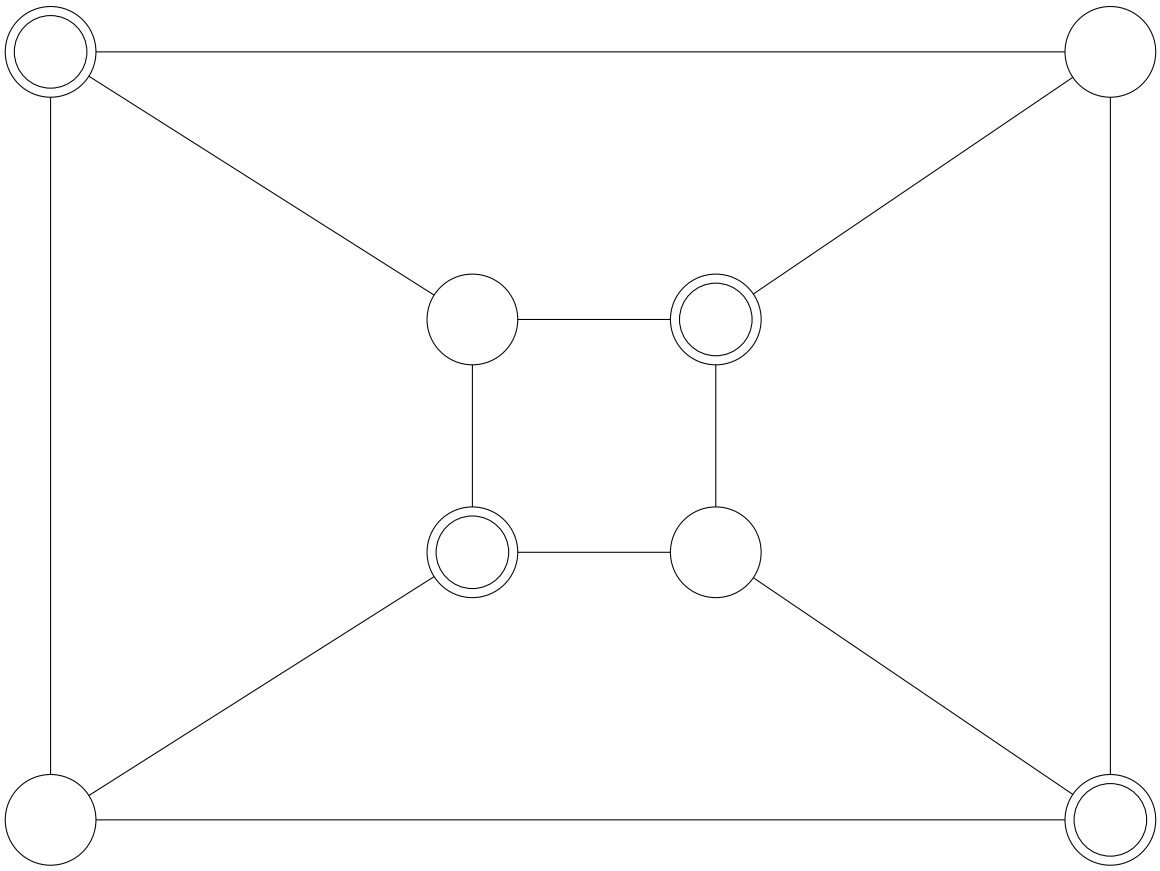


Figure 2.10: Bipartite graph diagram 01

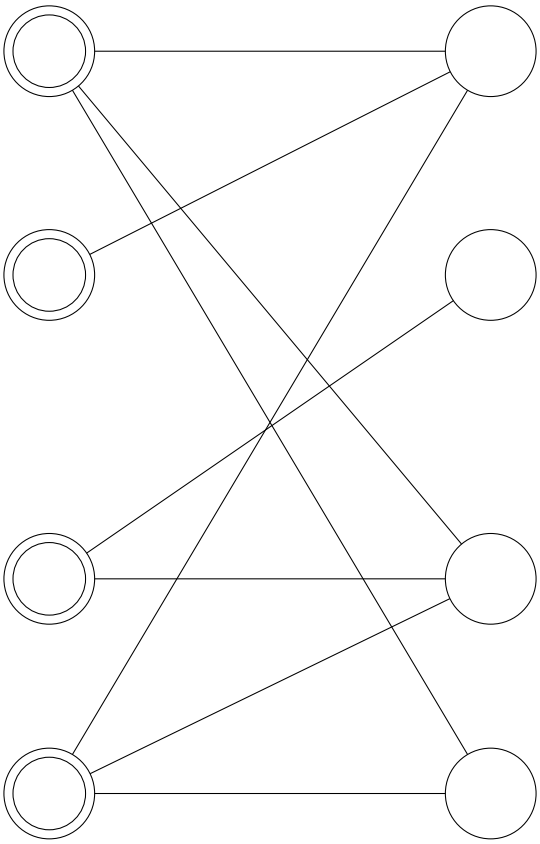


Figure 2.11: Bipartite graph diagram 02

2.6.6 Complete graphs

A complete graph is one where there is a unique edge between every pair of nodes. A complete graph with n vertices is denoted as the graph K_n

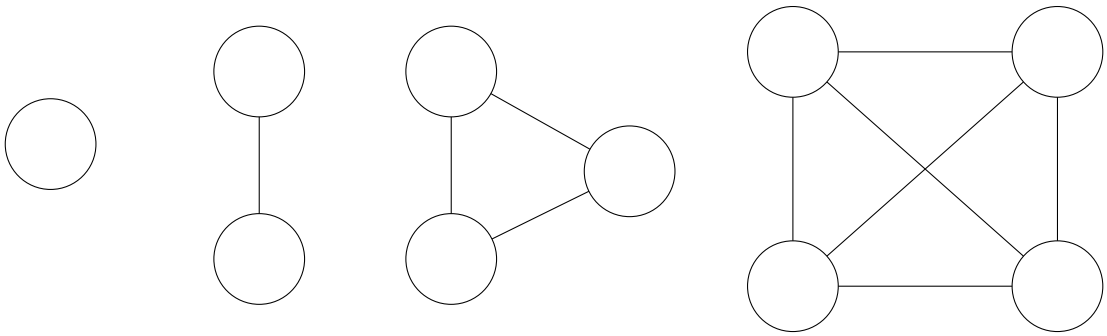
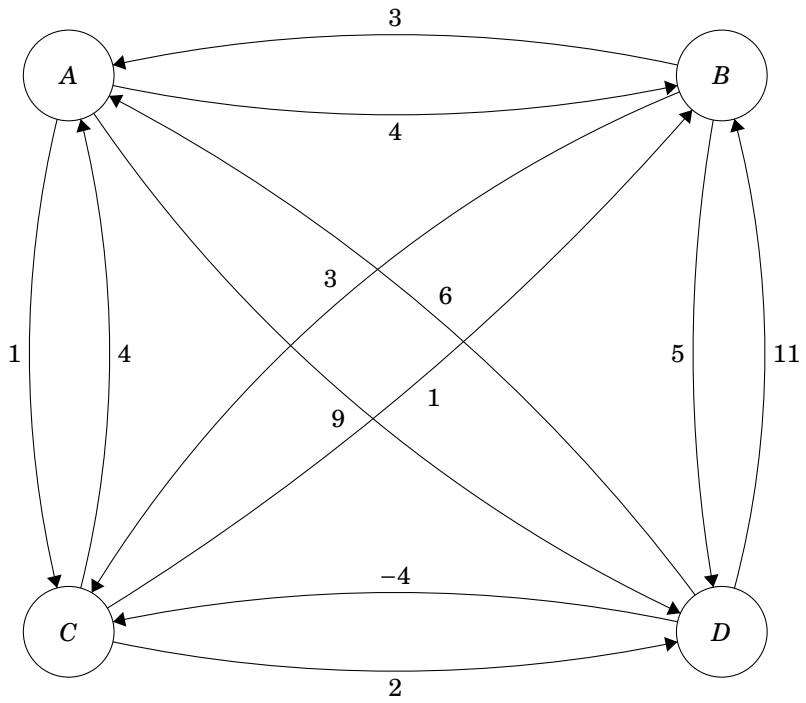


Figure 2.12: Complete graph diagram

2.6.7 ¿How can we represent a graph?

Adjacency Matrix

An adjacency matrix m is a very simple way to represent a graph. The idea is that the cell $m[i][j]$ represents the edge weigh of going from node i to node j. Let's suppose that we have the next graph:



If we build an adjacency matrix using the graph above, we have the next as a result

$$\begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} 0 & 4 & 1 & 9 \\ 3 & 0 & 6 & 11 \\ 4 & 1 & 0 & 2 \\ 6 & 5 & -4 & 0 \end{bmatrix} \end{array}$$

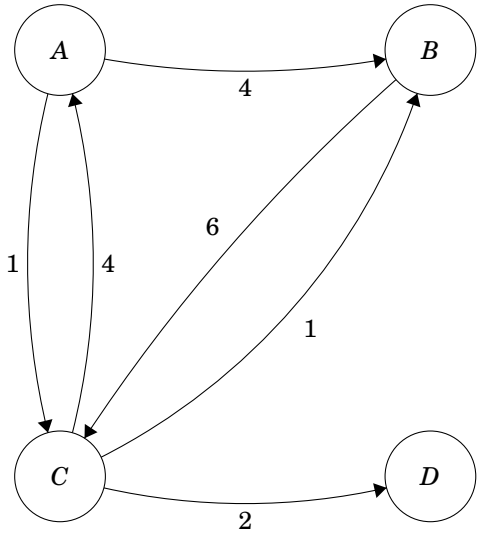
NOTE: It is often assumed that the edge of going from a node to itself has a cost of zero.

Now we need to know why is better or not to use an adjacency matrix

Pros	Cons
Space eifficient for representating dense graphs	Requires $O(V^2)$ space
Edge weight lookup is $O(1)$	Iterating over all edges takes $O(V^2)$ time
Simplest graph representation	

Adjacency list

An adjacency list is a way to represent a graph as a map from nodes to lists of edges



If we made a list we have the next as a result:
A -> [(B, 4), (C,1)]
B -> [(C,6)]
C -> [(A,4),(B,1),(D,2)]
D -> []
Let's take the node C as an example. As we can see the node C can reach with nodes:

- Node A with cost 4
- Node B with cost 1
- Node D with cost 2

As we made with the adjacency matrix we have some cons and pros for this representation:

Pros	Cons
Space efficient for representating sparse graphs	Less space efficient for denser graphs
Iterating over all edges is efficient	Edge weight lookup is $O(E)$
	Slightly more complex graph representation

Pros	Cons
Space efficient for representing sparse graphs	Less space efficient for denser graphs
Iterating over all edges is efficient	Edge weight lookup is $O(E)$
Very simple structure	

Edge list

An edge list is a way to represent a graph simply as an unordered list of edges. Assume the notation for any triplet (u, v, w) means: ”the cost from node u to node v is w”

If we take the same graph as the adjacent list and make a transformation of this graph to an edge list we get the next:

$$[(C,A,4),(A,C,1),(B,C,6),(A,B,4),(C,B,1),(C,D,2)]$$

This representation is seldomly used because of its lack of structure. However, it is conceptually simple and practical in a handful algorithms.

As the last representations we have some cons and pros for the edge list:

2.6.8 Common graph theory problems

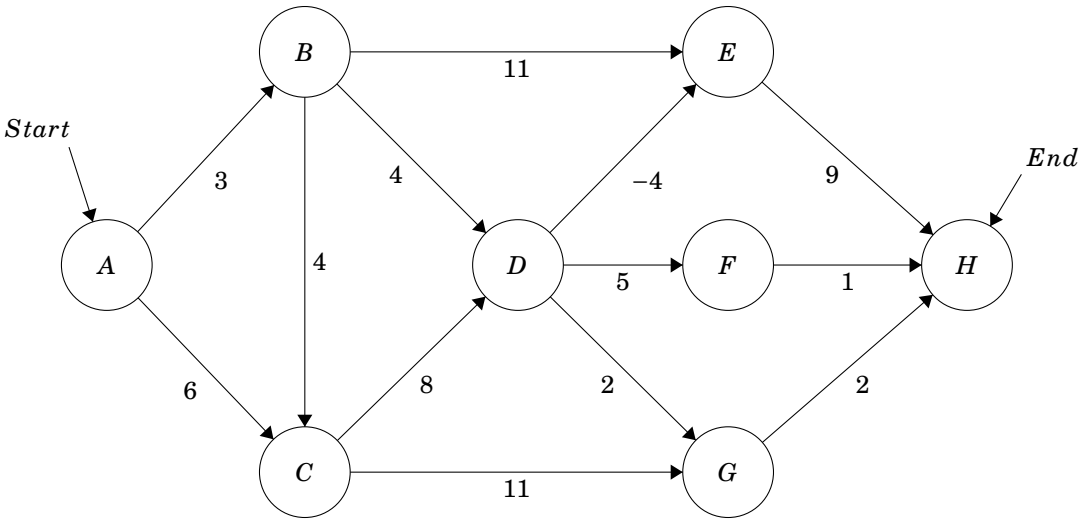
How can we solve graph theory problems?

Is necessarily ask yourself:

1. Is the graph directed or undirected?
2. Are the edges of the graph weighted?
3. Is the graph I will encounter likely to be sparse or dense with edges?
4. Should I use an adjacency matrix, adjacency list, an edge list or other structure to represent the graph efficiently?

Shortest path problem

Given a weighted graph, find the shortest path of edges from node A to node B.

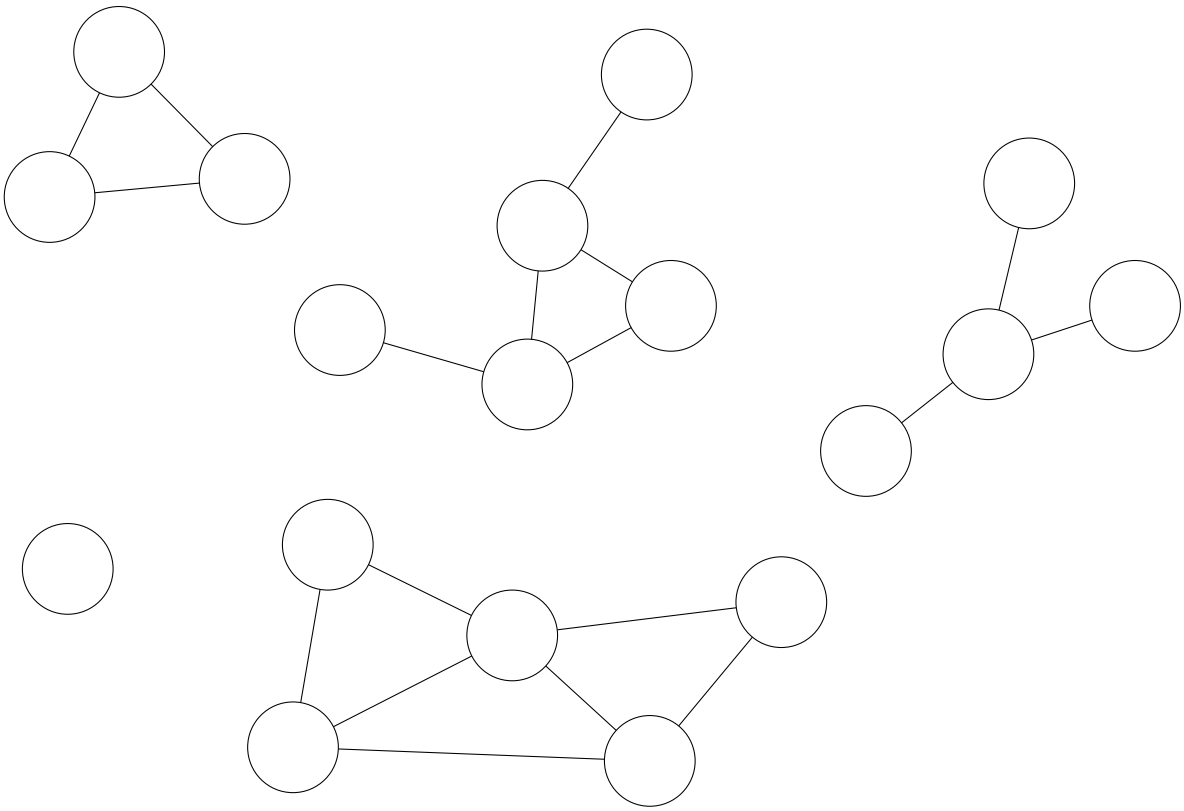


Using the example above we can see that the sortest path with the minimum cost is $A \rightarrow B \rightarrow D \rightarrow G \rightarrow H$

Algorithms: BFS (unweighted graph), Dijkstra's, Bellman-Ford, Floyd-Warshall, A*, and many more...

Connectivity

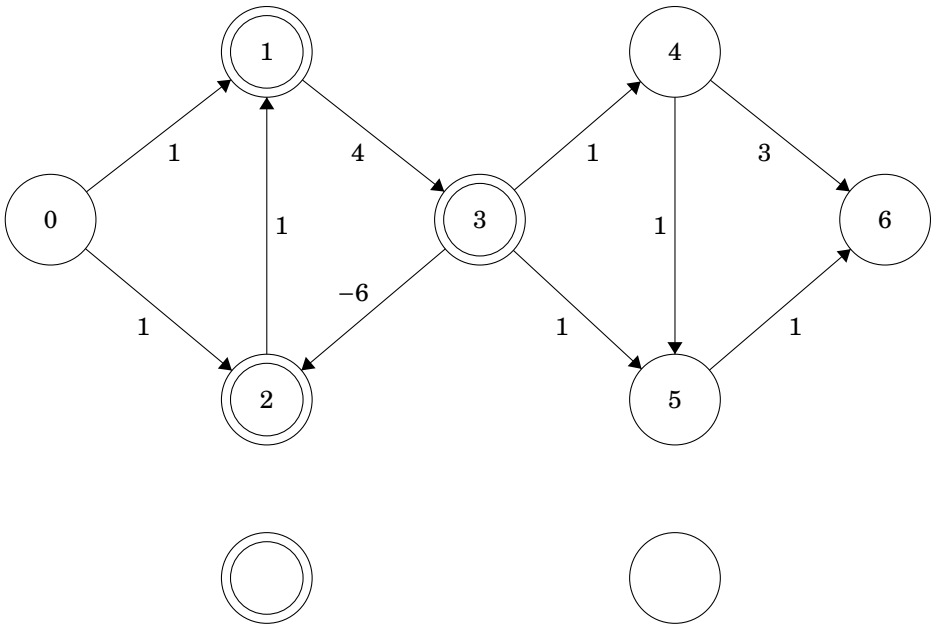
Does there exist a path between node A and node B?



Tipical solution: Use union find data structure or any search algorithm (e.g. DFS)

Negative cycles

Does my weighted digraph (directed graph) have any negative cycles? if so, where?

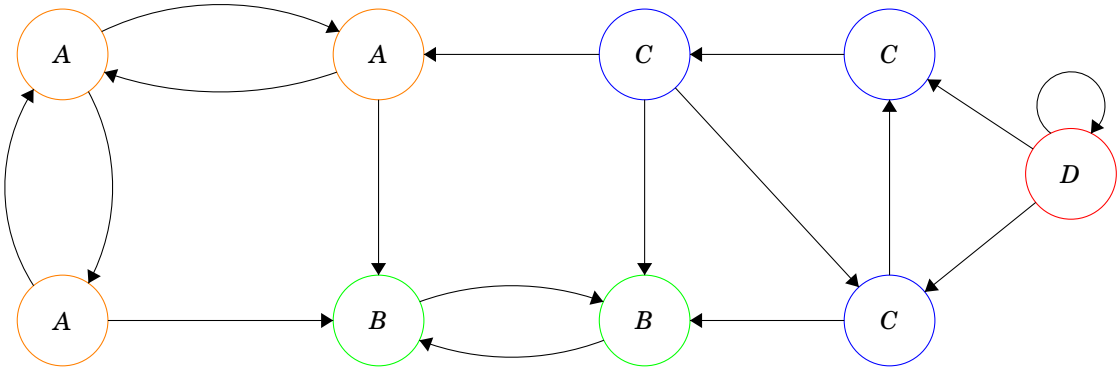


Directly in negative cycle *Unnafected node*

Algorithms: Bellman-Ford, Floyd-Warshall

Strongly connected components

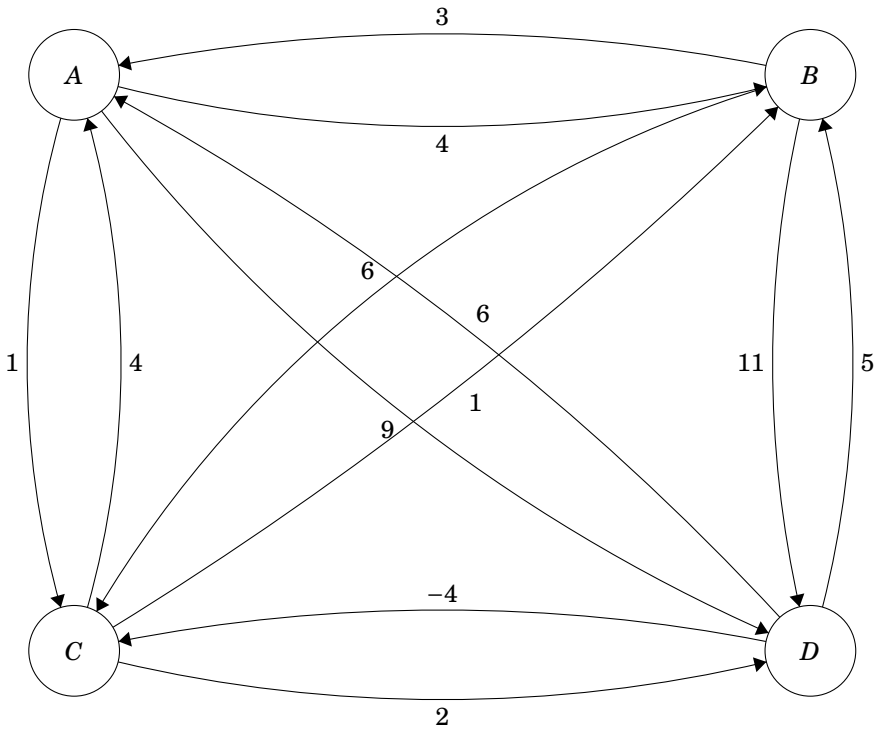
Strongly connected components (SCCs) can be thought of as self-contained cycles within a directed graph where every vertex in a given cycle can reach every other vertex in the same cycle.



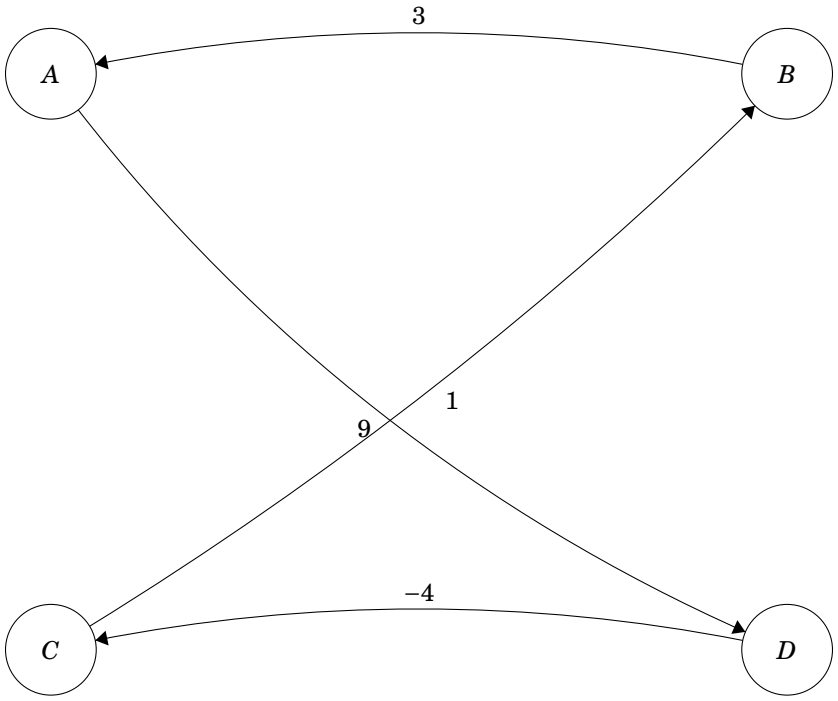
Algorithms: Tarjan's and Kosaraju's algorithm

Traveling salesman problem

"Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visists each city exactly once and returns to the origin city?"



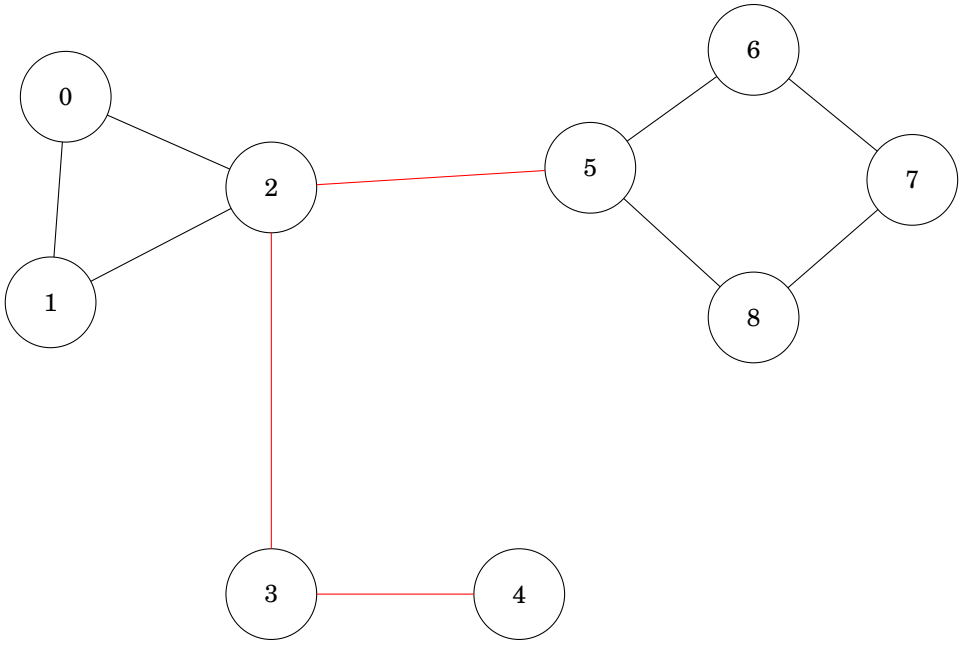
One possible solution is the next:



Algorithms: Held-Karp, branch and bound and many approximation algorithms.

Bridges

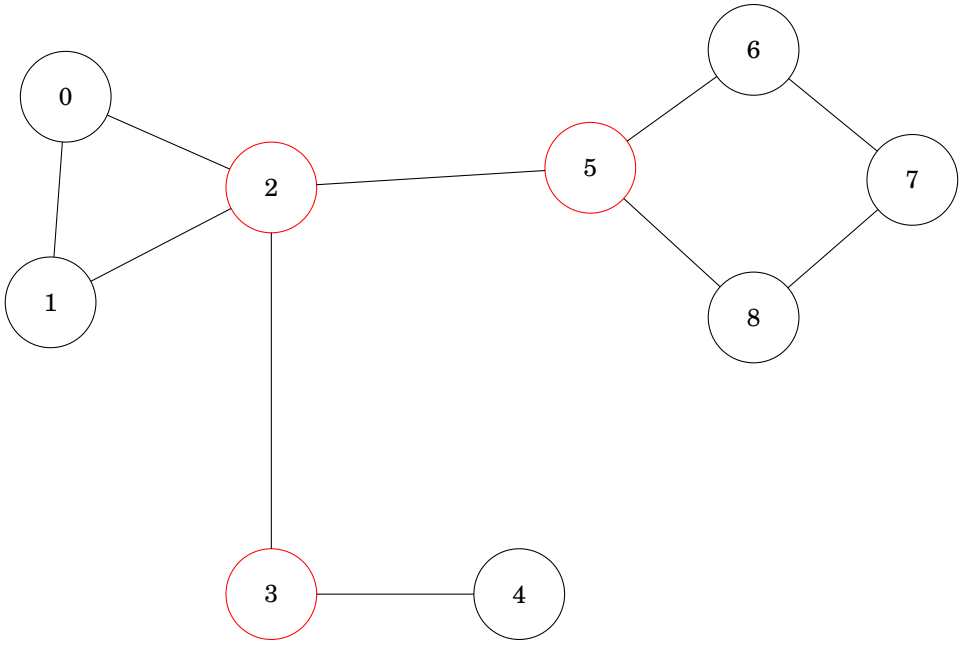
A bridge / cut edge is any edge in a graph whose removal increases the number of connected components



Bridges are important in graph theory because they often hint at weak points, bottlenecks or vulnerabilities in a graph

Articulation points

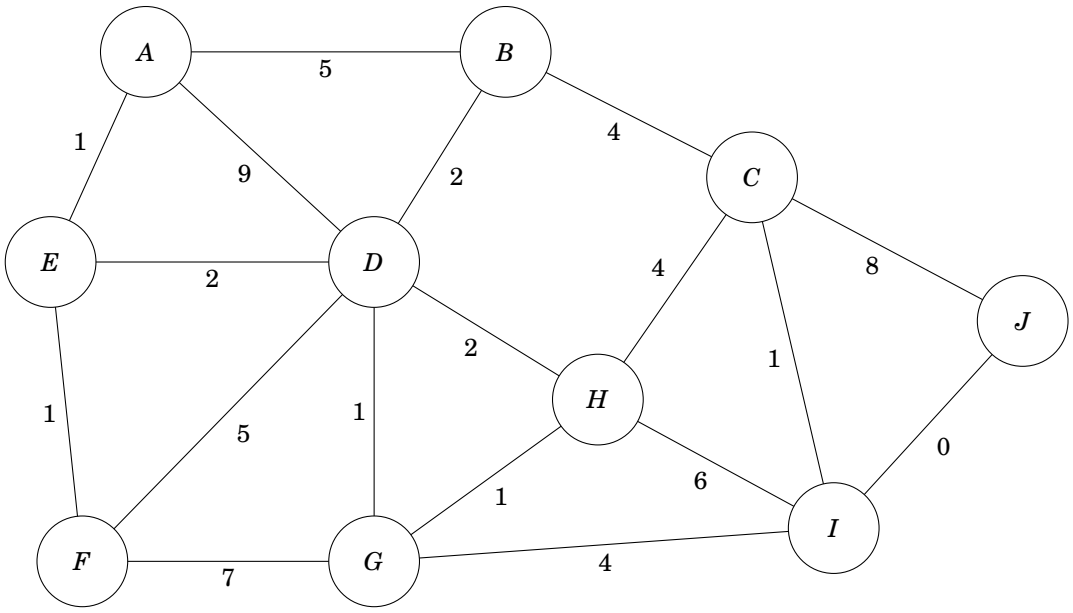
An articulation point / cut vertex is any node in a graph whose removal increases the number of connected components



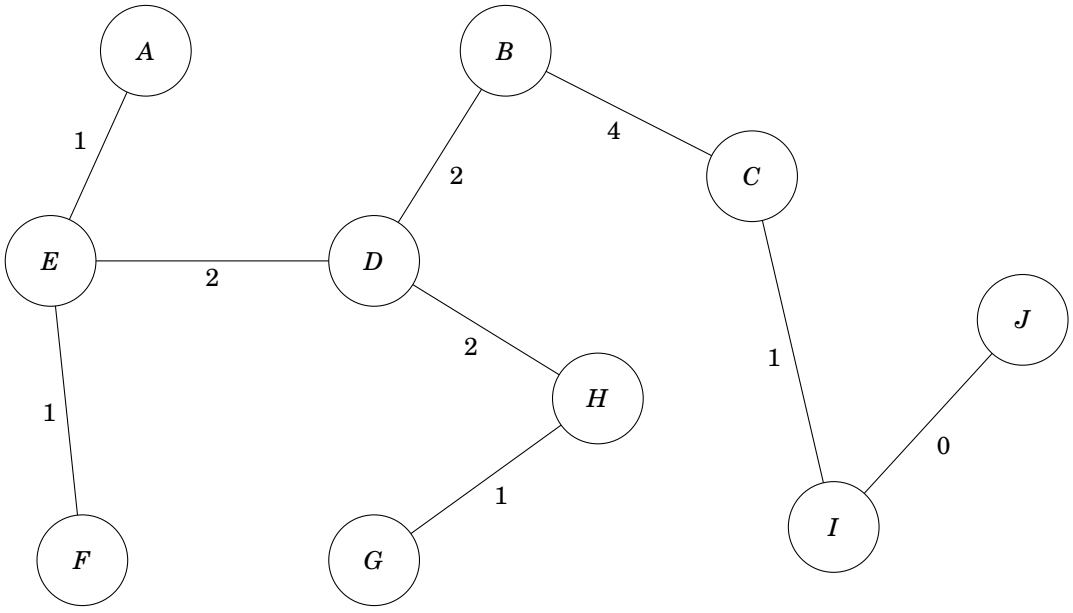
Articulation points are important in graph theory because they often hint at weak points, bottlenecks or vulnerabilities in a graph.

Minimum spaning tree (MST)

A minimum spaning tree (MST) is a subset of the edges connected, edge-weighted graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight.



Using the example above we find the next minimum spaning tree



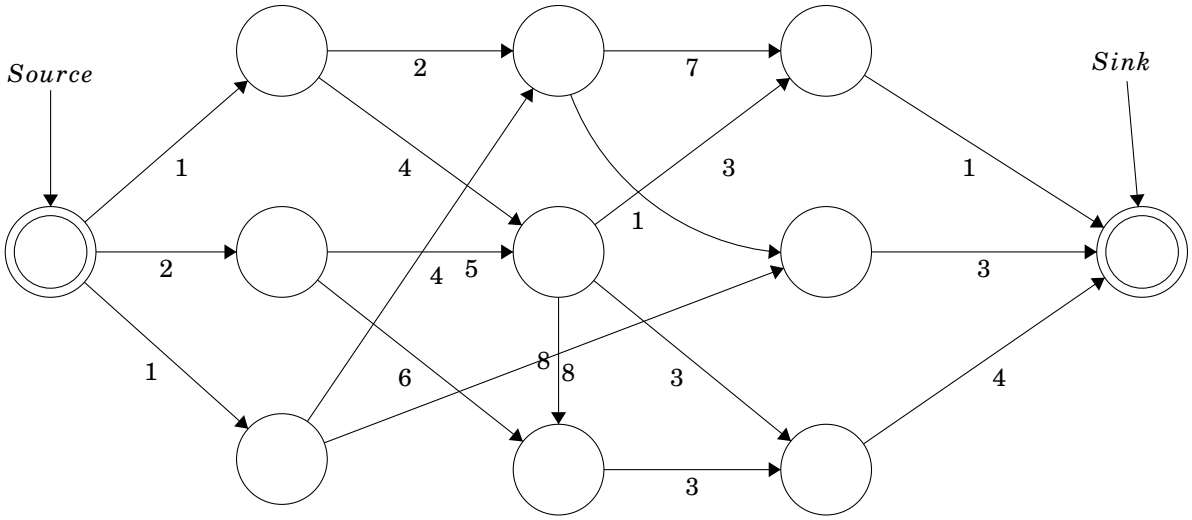
This MST has a total weight of 14. Note that MSTs on a graph are not always unique.

MSTs are seen in many applications including: designing a least cost network, circuit design, transportation networks, etc...

Algorithms: Kruskal's, Prim's and Boruvka's algorithm

Network flow: max flow

Question: with an infinite input source how much "flow" can we push throught the network?



Suppose the edges are roads with cars, pipes with water or hallways with packed with people. Flow represnets the volume of water allowed to flow through the pipes, the number of cars the roads can sustain in traffic and the maximum amount of people that can navigate through the hallways.

Algorithms: Ford-Fulkerson, Edmonds-Karp and Dinic's algorithm

2.6.9 Depth First Search (DFS)

Overview

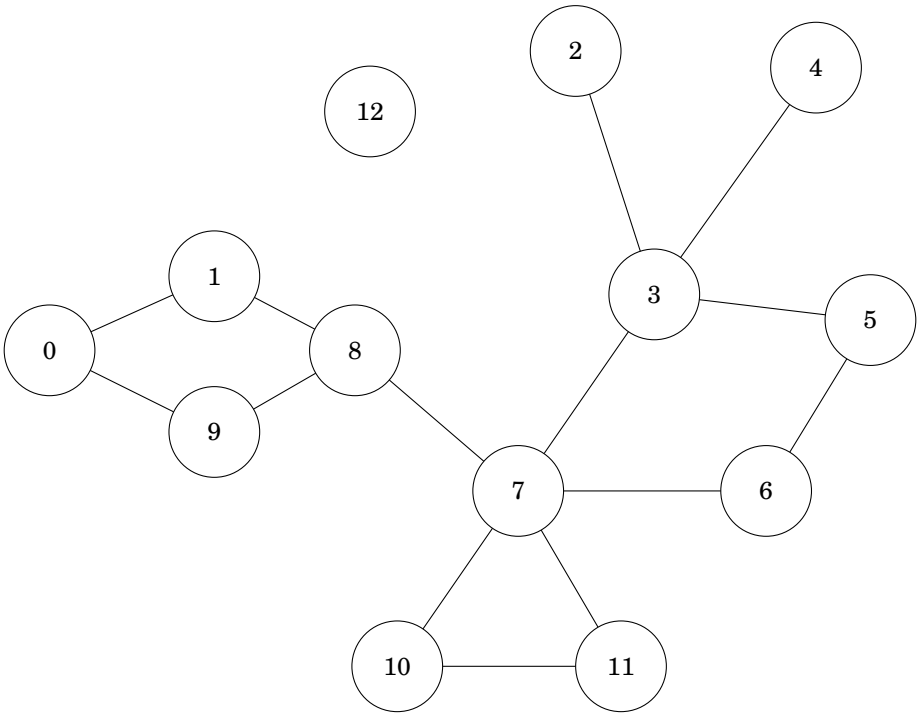
The Depth First Search (DFS) is the most fundamental search algorithm used to explore nodes and edges of a graph. It runs with a time complexity of $O(V+E)$ and is often used as a building block in other algorithms.

By itself the DFS is not all that useful, but when augmented to perform other tasks such as count connected components, determine connectivity, or find bridges / articulation points then DFS really shines.

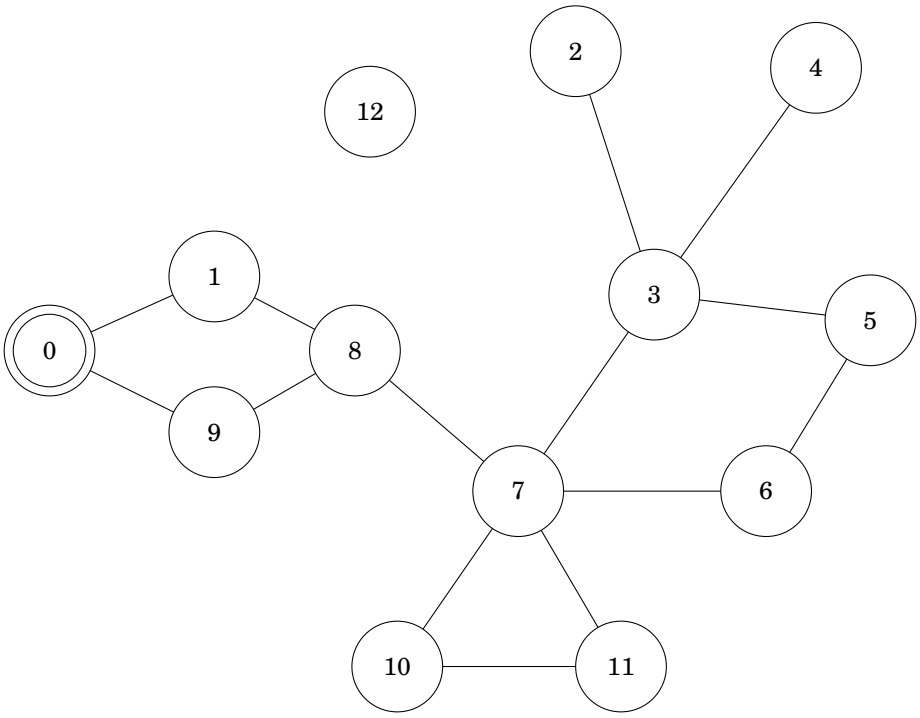
Basic DFS

As the name suggest, a DFS plunges depth first into a graph without regard for which edge it takes next until it cannot go any further at which point it backtracks and continues.

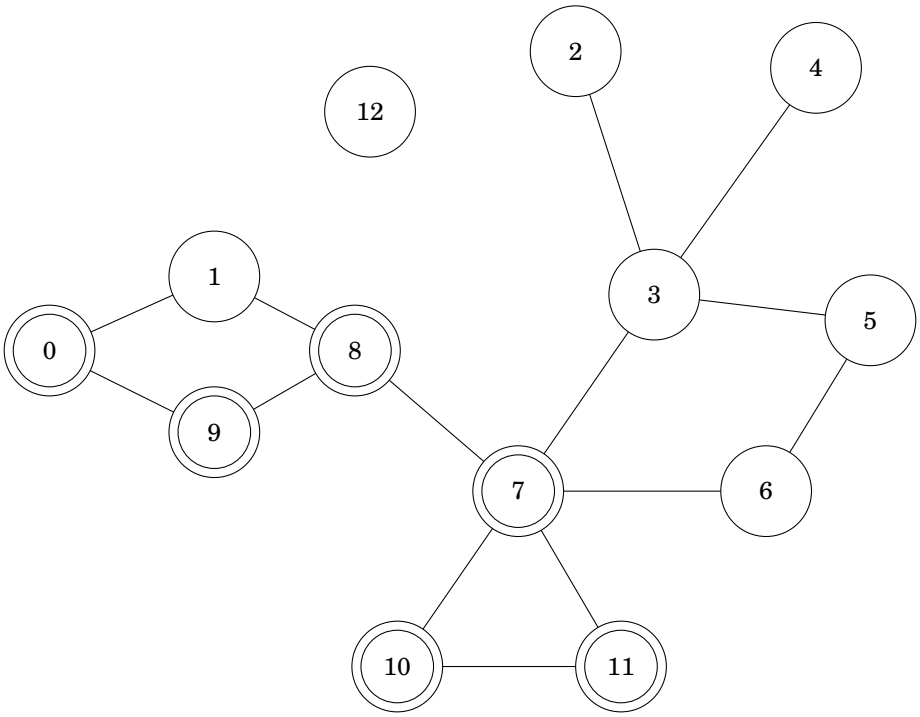
Lets give an example using the next graph



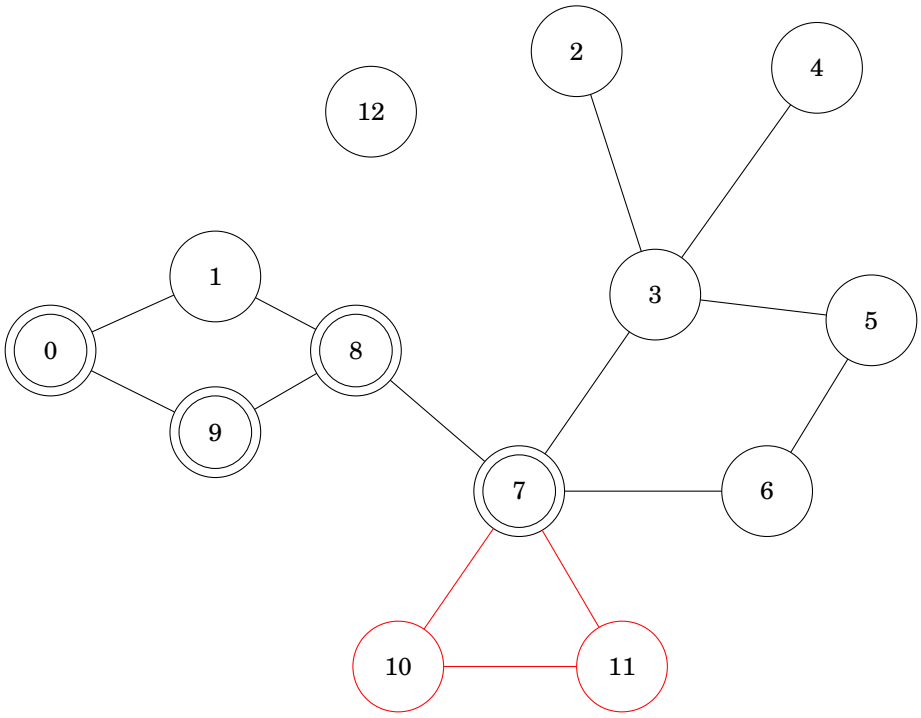
The DFS starts in the node 0



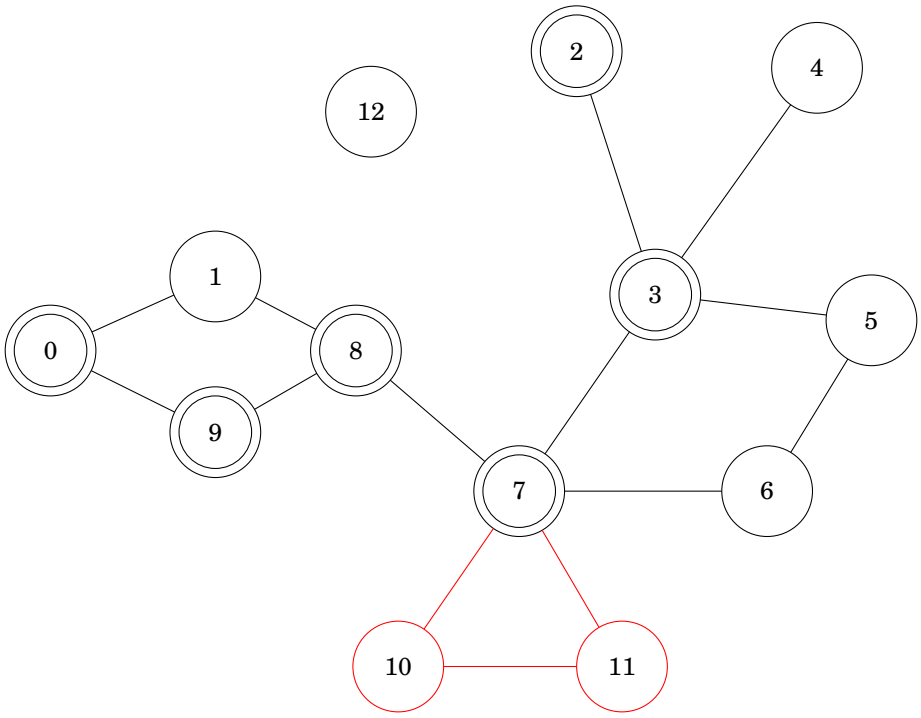
Then the DFS moves through an edge and goes to another node, for example node 9, then node 8, then node 7, then node 10, then node 11 and again node 7



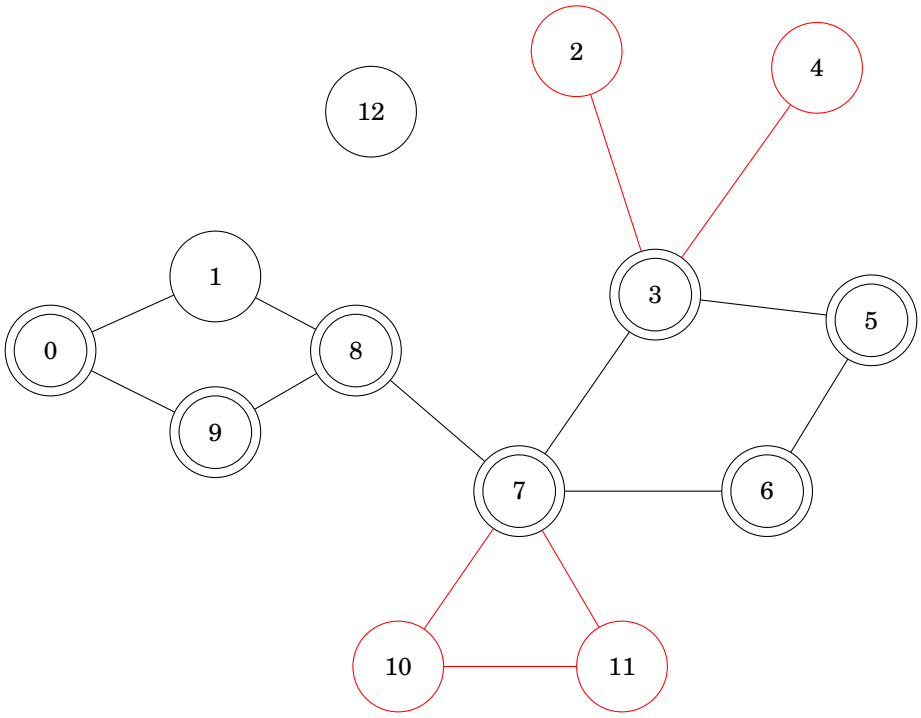
We already visited the node 7, so we need to return using backtracking an visit the rest of neighbours of node 7 so we continue doing the DFS but in another direction and we put a mark in the nodes that we already visited but are not valid.



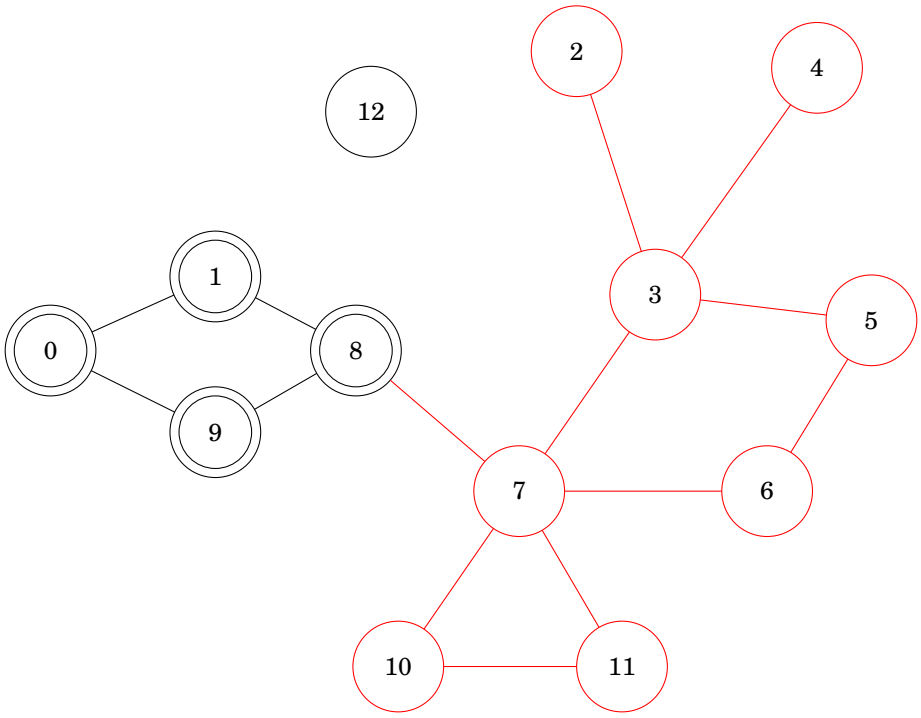
Now we move to the node 3 and then to the node 2 so we need to make backtracking again and continue with our DFS



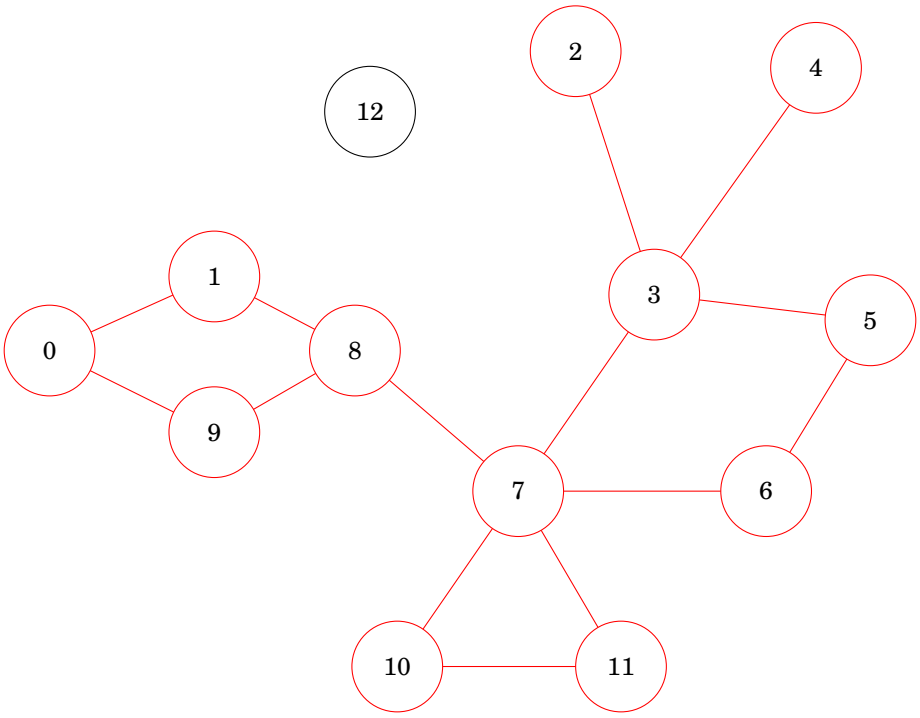
Now we move to the node 4 but again it is a dead end so we move to the node 5 then to the node 6, then to the node 7 but we already visited



Now we backtrack and we return to the node 8, and we continue doing the DFS



Again we have a cycle so we bakctrack and we have visited all the nodes



So, how can we made this using some code? I've written some code to explain this clearly

```
#include <bits/stdc++.h>

using namespace std;

typedef long long int lli;
typedef vector< lli > vi;

/* A graph is a set of vertices and edges. Also we need to
    know that
    we can represent a graph using:
    -> An adjacency matrix G[u][v] = w
    -> An adjacency list u -> (v, w)
    -> An edge list (u, v, w)
    To this implementation I've used an adjacency list
 */
struct Graph{

    /* As I mentioned before I'm using an adjacency list, so
         each
        node has an adjacency list, and each node has it's
            value and it's
        visited state
     */
    struct Node{
```

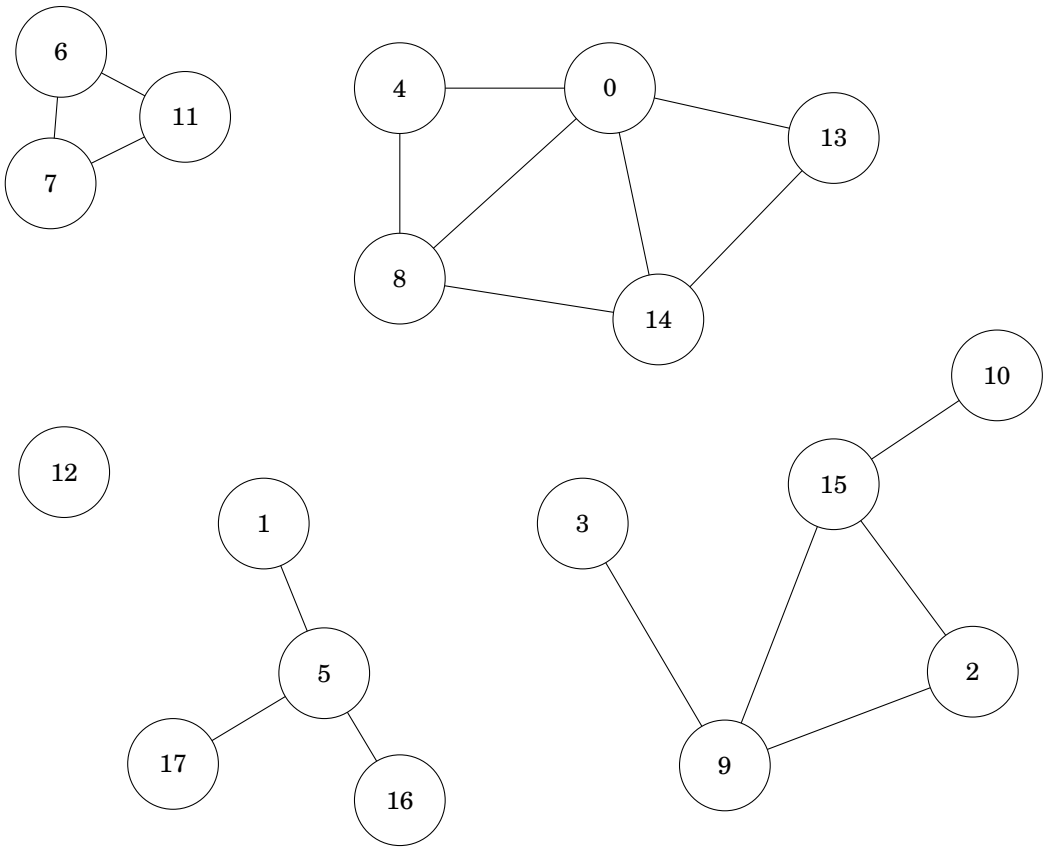
```
            bool visited;
            lli value;
            vi adjList;

            Node(){
                value = 0;
                visited = false;
            }
        };

        //Set of nodes
        vector< Node > nodes;
        lli size;

        Graph( lli size ){
            this -> size = size;
            nodes.assign( size, Node() );
        }

        /* This function connects a node to our graph
        param: (lli)from: the source node
        param: (lli)to: the destination node
        return: nothing    */
        void connect(lli from, lli to){
            nodes[ from ].adjList.push_back( to );
            nodes[ to ].adjList.push_back( from );
        }

        /* This function makes a Depth First Search
        param: (lli)i: the index of the node to search
        return: nothing    */
        void DFS( lli i ){
            nodes[ i ].visited = true;
            //This line is using to verify if the DFS made is
                correct
            nodes[ i ].value = 1;
            for(int j = 0; j < nodes[ i ].adjList.size(); j++){
                lli u = nodes[ i ].adjList[ j ];
                if( !nodes[ u ].visited )
                    DFS( u );
            }
        }

        void print(){
            cout << endl;
            for( int i = 0; i < nodes.size(); i++){
                cout << i << " " << nodes[ i ].value << endl;
            }
            cout << endl;
        }
    };

    int main(){

        lli n_nodes, n_connections, from, to;
```

```
        cin >> n_nodes >> n_connections;
        Graph graph( n_nodes );
        while(n_connections--){
            cin >> from >> to;
            graph.connect( from, to );
        }

        graph.print();

        graph.DFS( 0 );

        graph.print();

        return 0;
    }
```

Just consider that this implementations of the Depth First Search is recusive and in some cases it does not work.

Finding components

An application of the DFS is find components, it is very simple to understand, let's use the next:



As we can see we have components that are not connected. If we separate in subsets we get:

- $A = 6, 11, 7$
- $B = 4, 0, 13, 14, 8$
- $C = 12$
- $D = 1, 5, 16, 17$
- $E = 3, 9, 2, 15, 10$

Each subset we go to call it as component. Now we know what is a component but we do not know how to identify each component yet. To solve this problem we can use the Depth First Search. Maybe you noticed that if the graph have a node that is not connected the DFS never verify that node, so we can made n DFS, one for each component if and just if the current node has not been visited. So, if we code this we need just modify the next:

Modify the struct **Node**, we need to add the attribute component, something like this:

```
struct Node{
    lli value;
    bool visited;
    lli component;
    vector< lli > adjList;
    Node(){
        value = 0;
        visited = false;
        component = 0;
    }
};
```

Then we need to modify our DFS function, because we go to assign something to the component attribute:

```
void DFS( lli i, lli count ){
    nodes[ i ].visited = true;
    nodes[ i ].value = 1;
    nodes[ i ].component = count;
    for( int j = 0; j < nodes[ i ].adjList.size(); j++ ){
        lli u = nodes[ i ].adjList[ j ];
        if( !nodes[ u ].visited )
            DFS( u, count );
    }
}
```

And finally we need to write the code to the fundComponents function. Here we need to made n callings to the DFS function, many as we need, and we go to made the DFS if and just if the current node has not been visited yet.

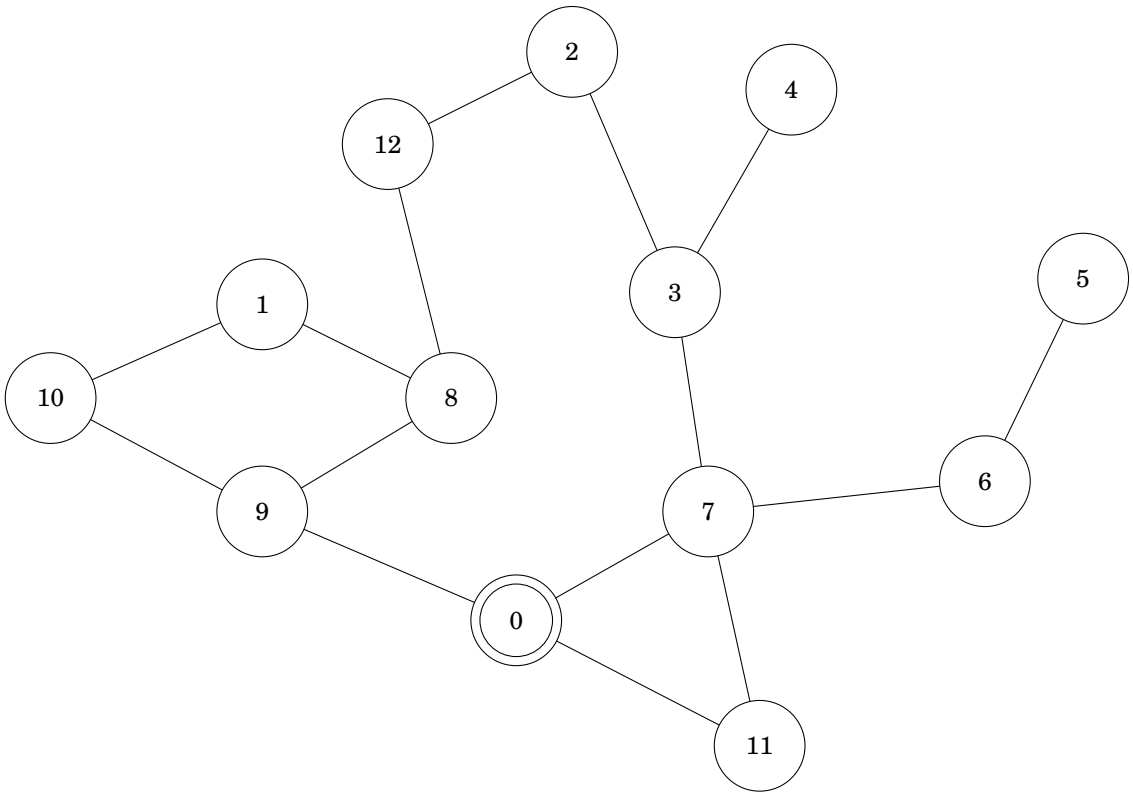
```
lli findComponents(){
    lli count = 0;
    for( int i = 0; i < size; i++ )
        if( !nodes[ i ].visited )
            DFS( i, count++ );
    return count;
}
```

2.6.10 Breadth First Search (BFS)

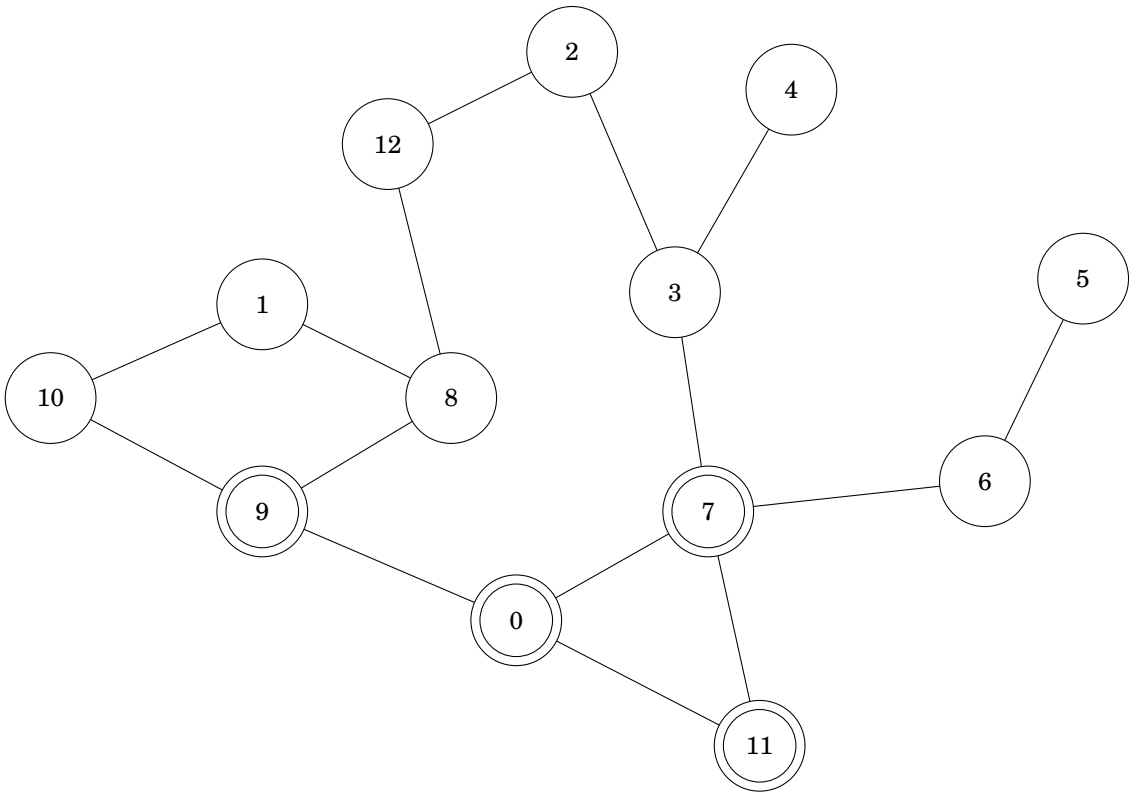
Overview

The Breath First Search (BFS) is another fundamental search algorithm used to explore nodes and edges of a graph. It runs with a time complexity of $O(V+E)$ and is often used as a building block in other algorithms.

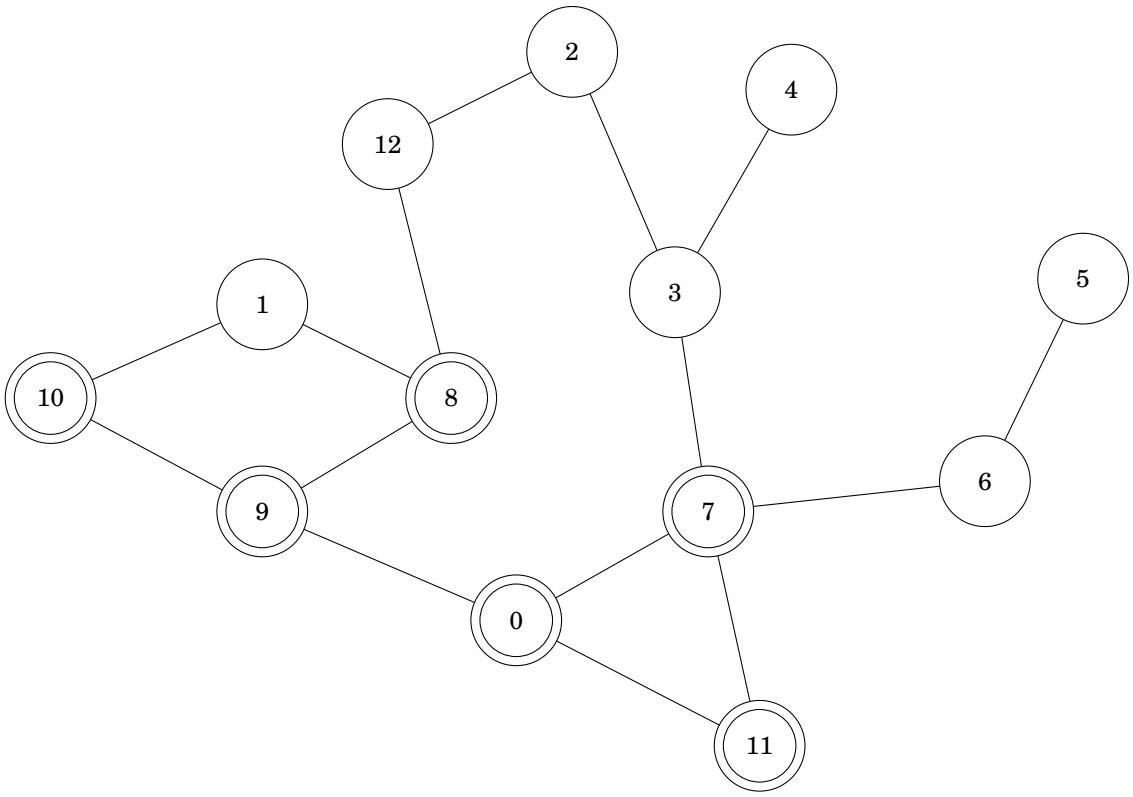
The BFS algorithm is particularly useful for one thing: finding the shortest path on unweighted graphs. Let's show the next graph.

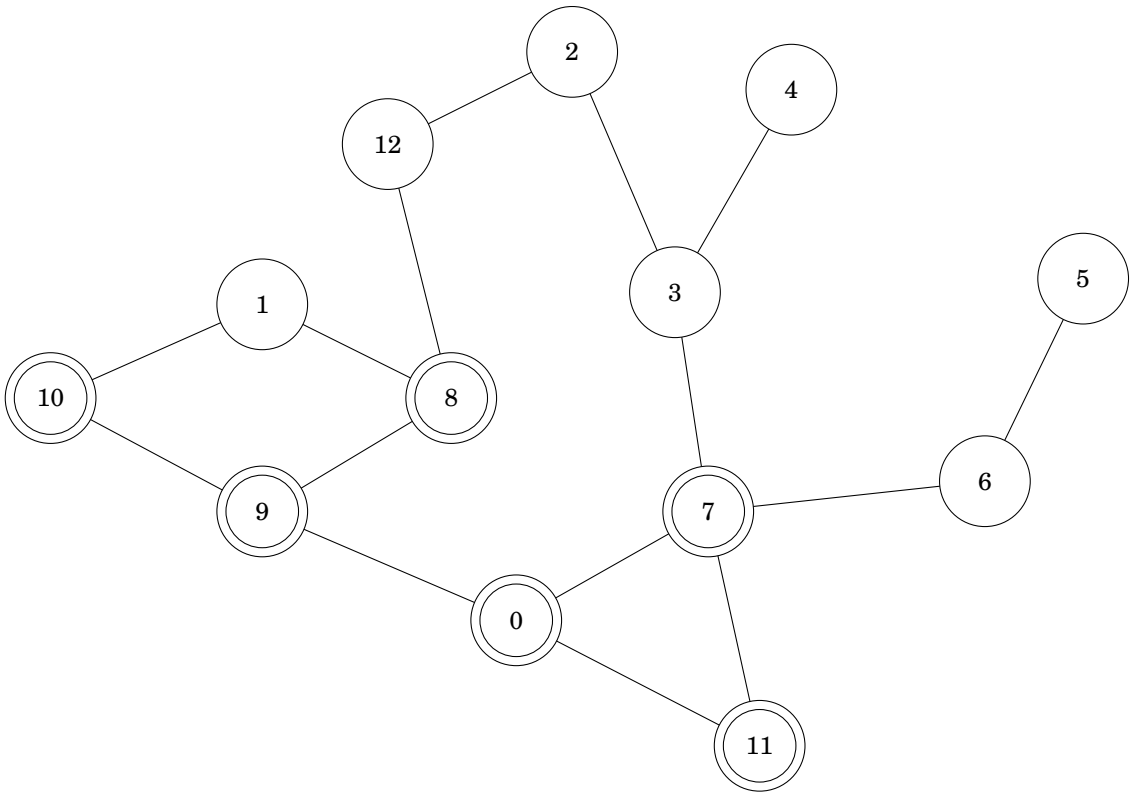


If we start in the node 0, using the BFS then we need to move to the neighborhoods of the current node, so we get the next:

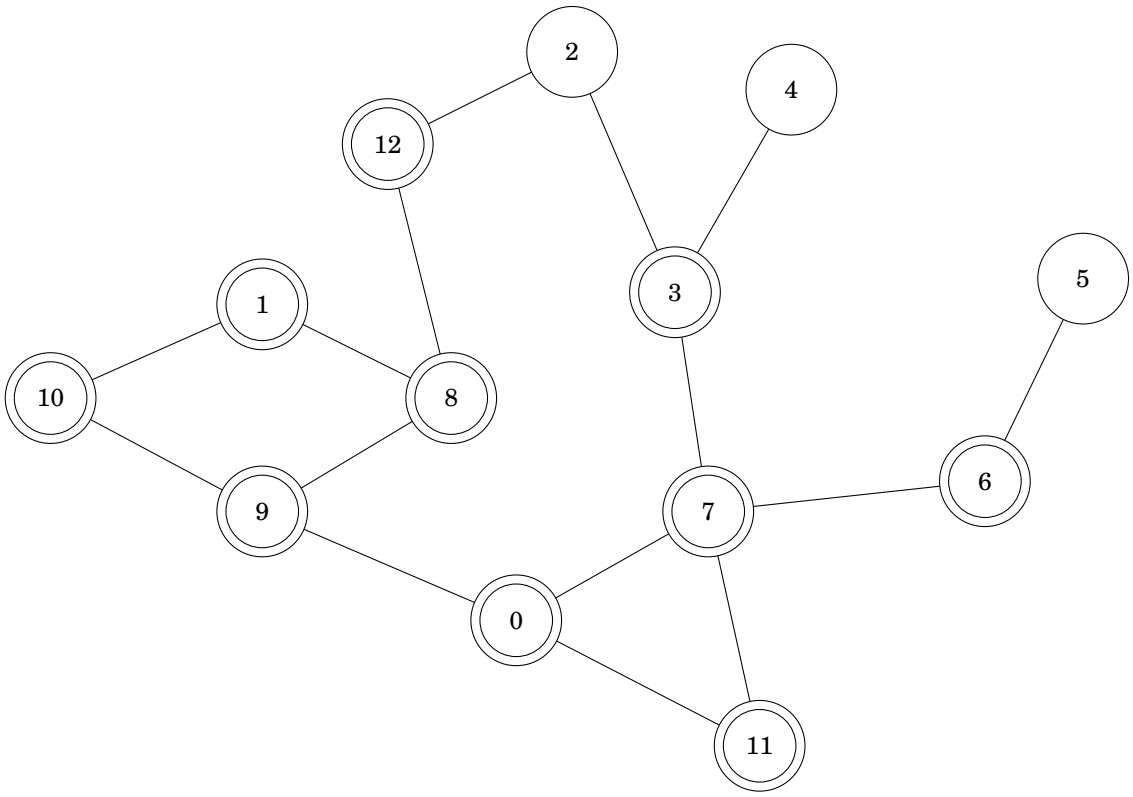


Again we choose arbitrarily a new node for example the node 9 and we repeat the same process.

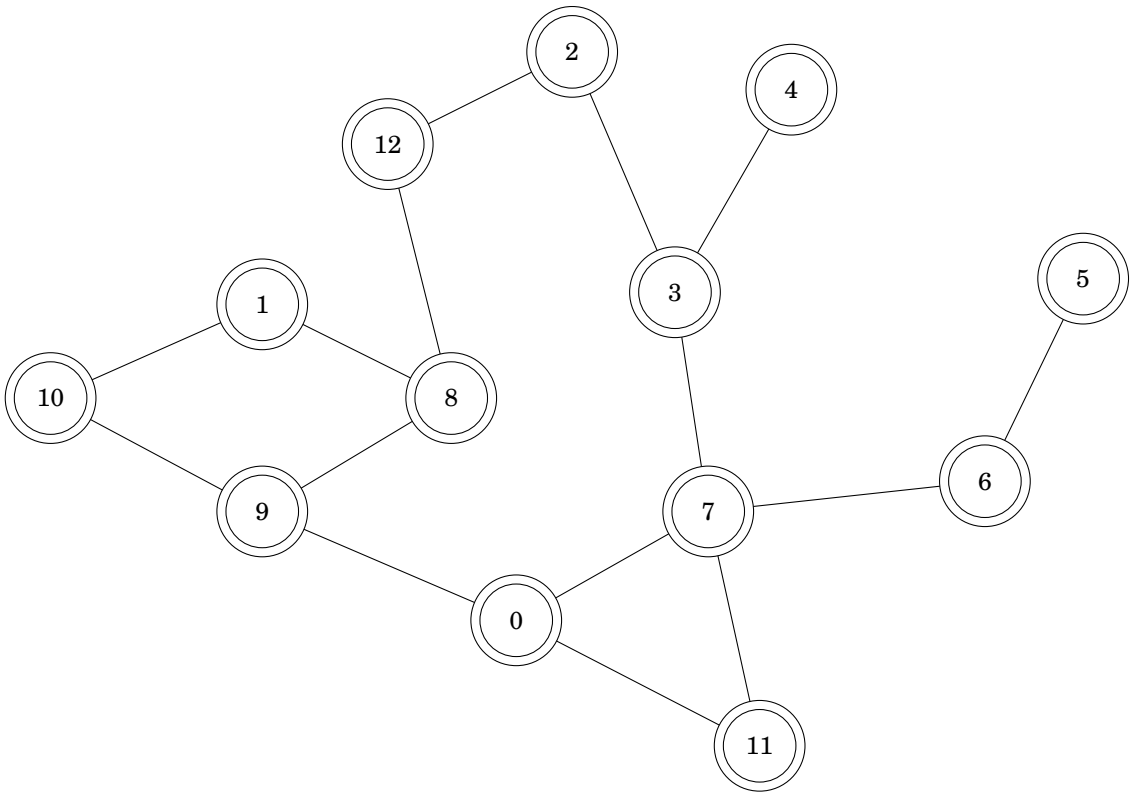

2
4
12
3
5
1
10
8
6
9
7
0
11



We make the same with the nodes 10, 8 and 7.

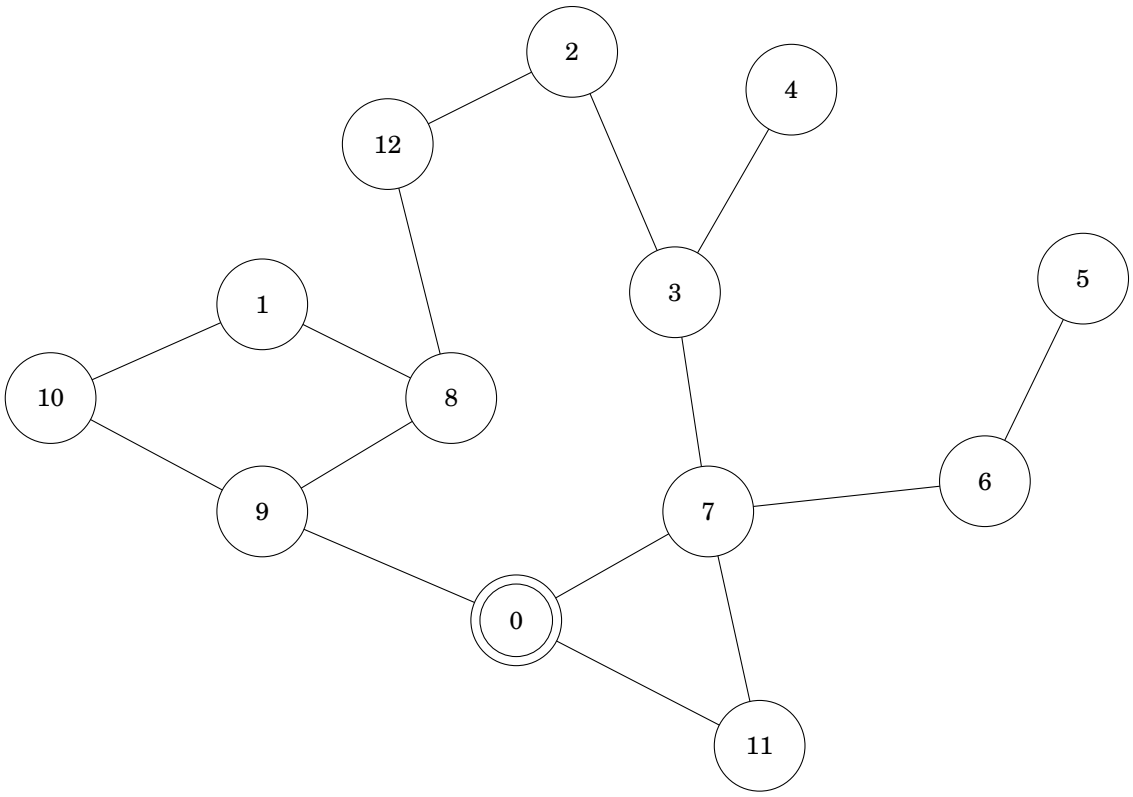


And finally repeat the same process with the next nodes



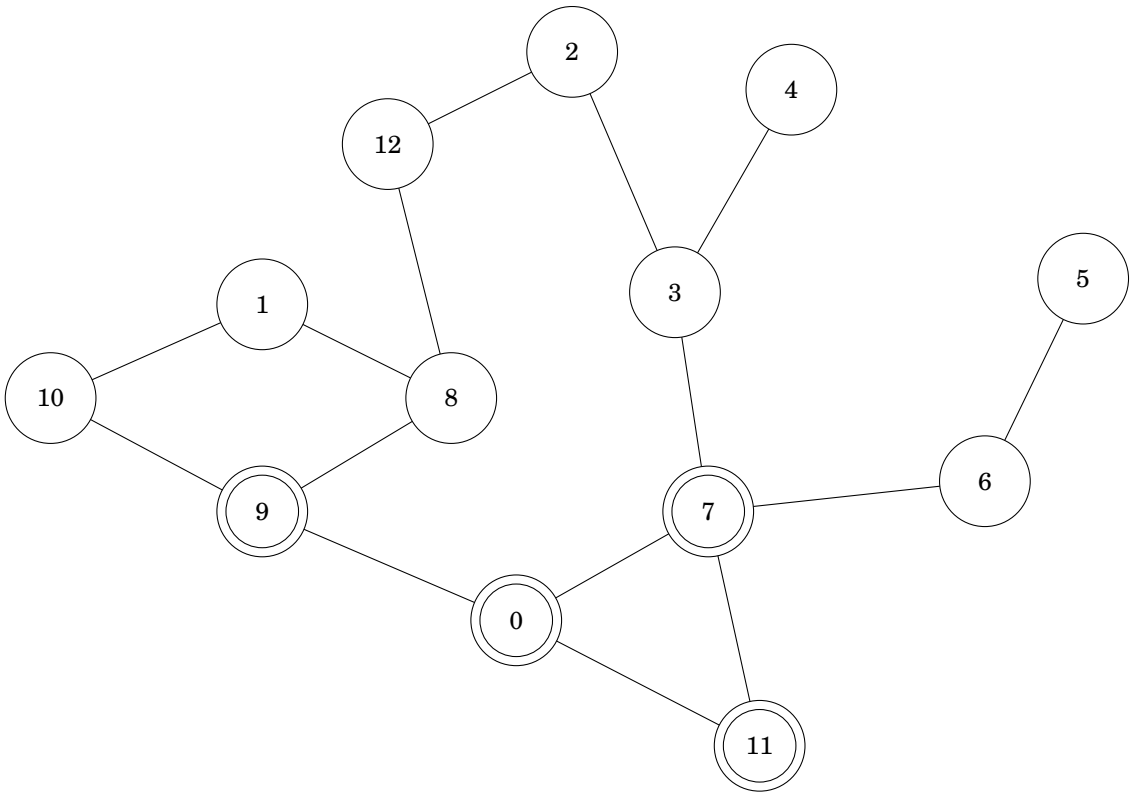
Maybe understand how it works is really easy, but you need to considet that to this operation we need to use a queue. Let's see the same example but step by step.

Again we begin with the node 0.



And we add the element to our queue.

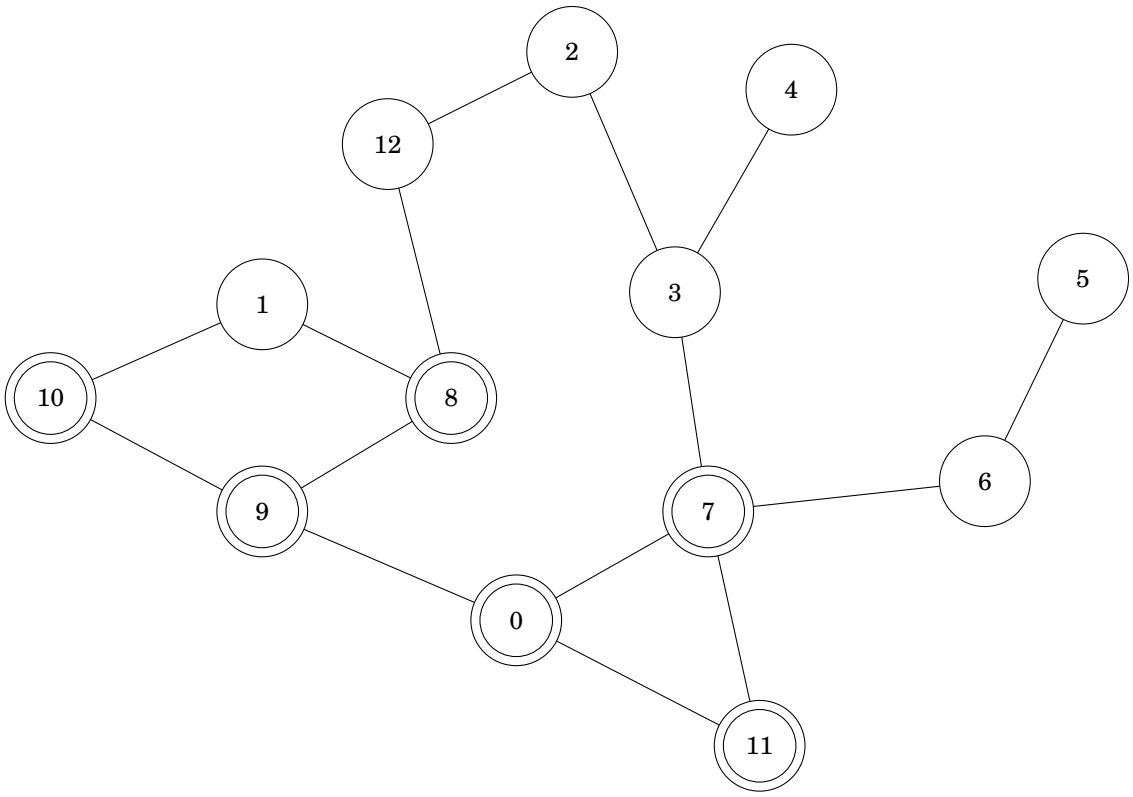
Then we visit the neighbors of the current node.



Also we add the elements to our queue.

11
7
9
0

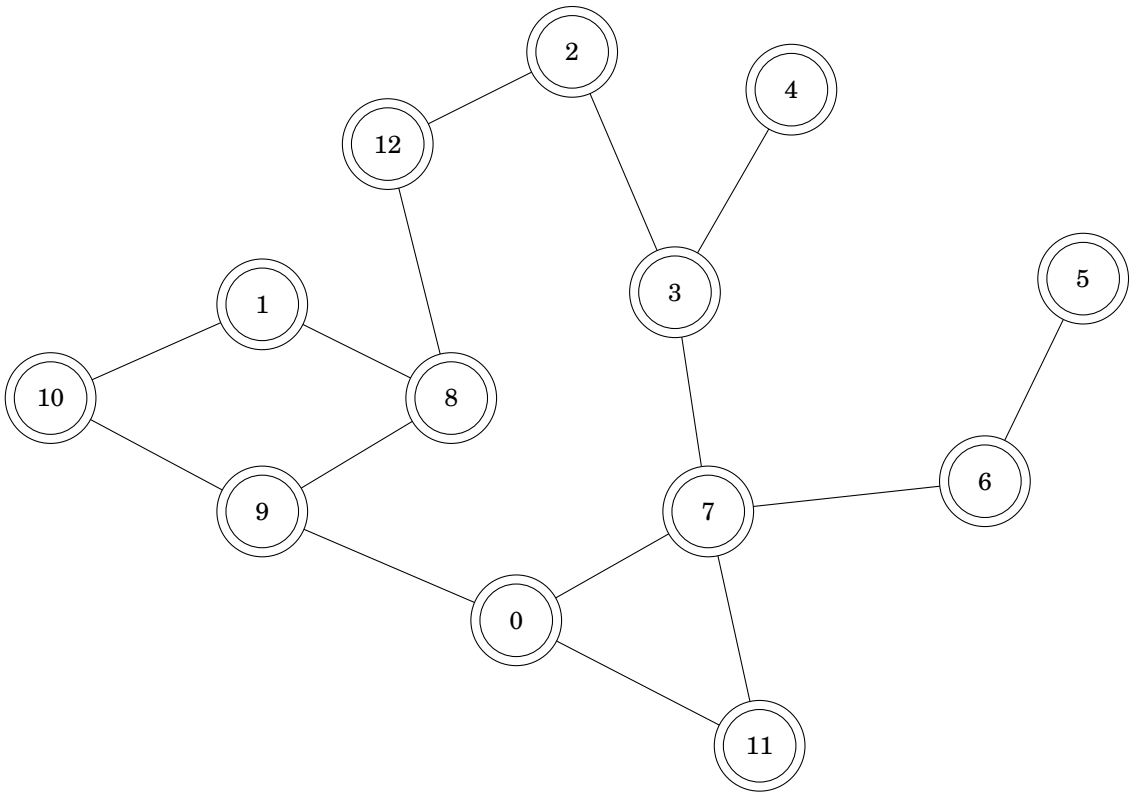
Now we need to visit the neighbors of the nodes 9, 7 and 11.



We add the nodes to our queue.

8
10
11
7
9
0

We continue doing the same until we have all the nodes in the queue.



4
2
5
12
1
3
6
8
10
11
7
9
0

Finally we need to continue doing the same process until the queue is empty.

Using a queue

The BFS algorithm uses a queue data structure to track which node to visit next. Upon reaching a new node the algorithm adds it to the queue to visit it later. The queue data structure works just like a real world queue such a waiting line at a restaurant. People can either enter the waiting line (enqueue) or get seated (dequeue).

Here you are the code to the function to make the Breath First Search. Consider the same structure for a graph that I implemented in the DFS section.

```
void BFS( lli u ){
    queue< lli > Queue;
    Queue.push( u );

    while( !Queue.empty() ){
        u = Queue.front();
        Queue.pop();
        nodes[ u ].visited = true;
        nodes[ u ].value = 1;

        for( int i = 0; i < nodes[ u ].adjList.size(); i++ )
            {
            lli v = nodes[ u ].adjList[ i ];
            if( !nodes[ v ].visited )
                Queue.push( v );
        }
    }
}
```

Shortest path in an unweighted graph

2.7 Trees

A tree is a special type of graph. It is an undirected graph with no cycles. Equivalently, it is a connected graph with N nodes and N - 1 edges.

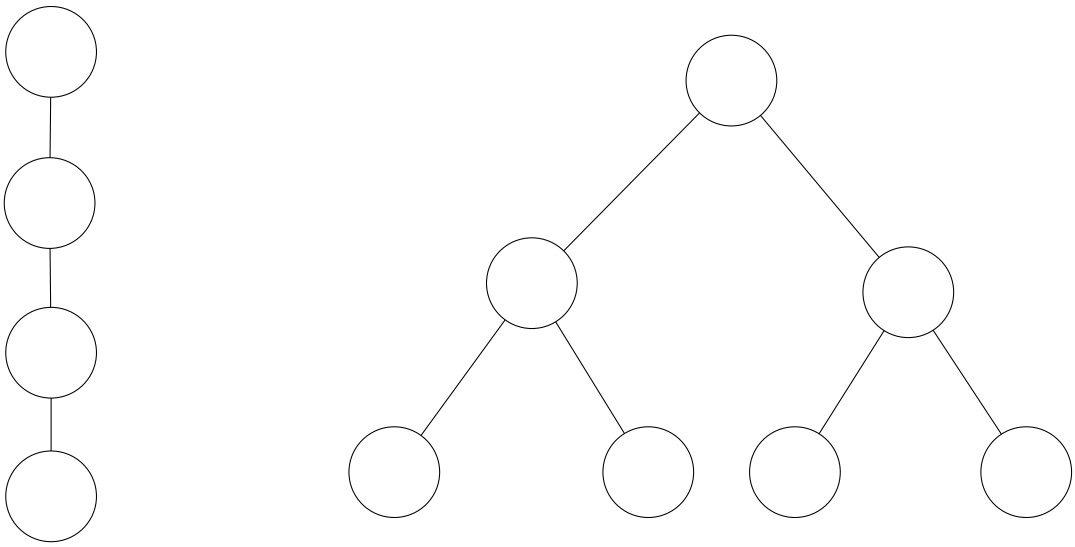


Figure 2.13: Basic tree diagram 01

Or maybe something more strange but stills being a tree:

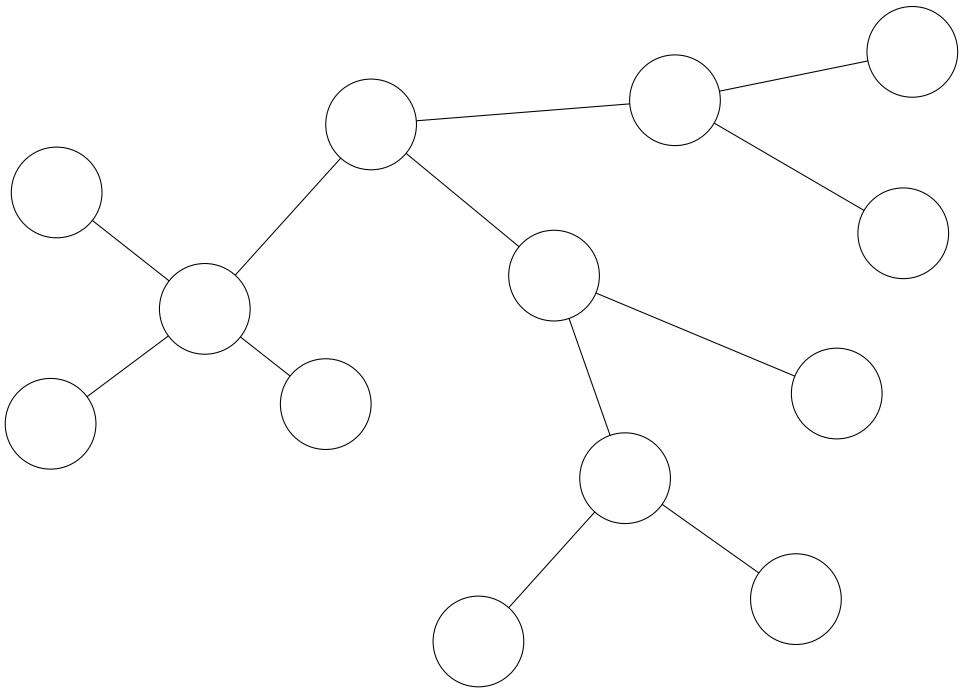


Figure 2.14: Basic tree diagram 02

2.7.1 Roted trees

A roted tree is a tree with a designated root node where every edge either points away from or towards the root node. When edges point away from the root the graph is called an arborescence (out-tree) and anti-arborescence (in-tree) otherwise. In the next figure you can see an anti-arborescence (in-tree) and an arborescence (out-tree)

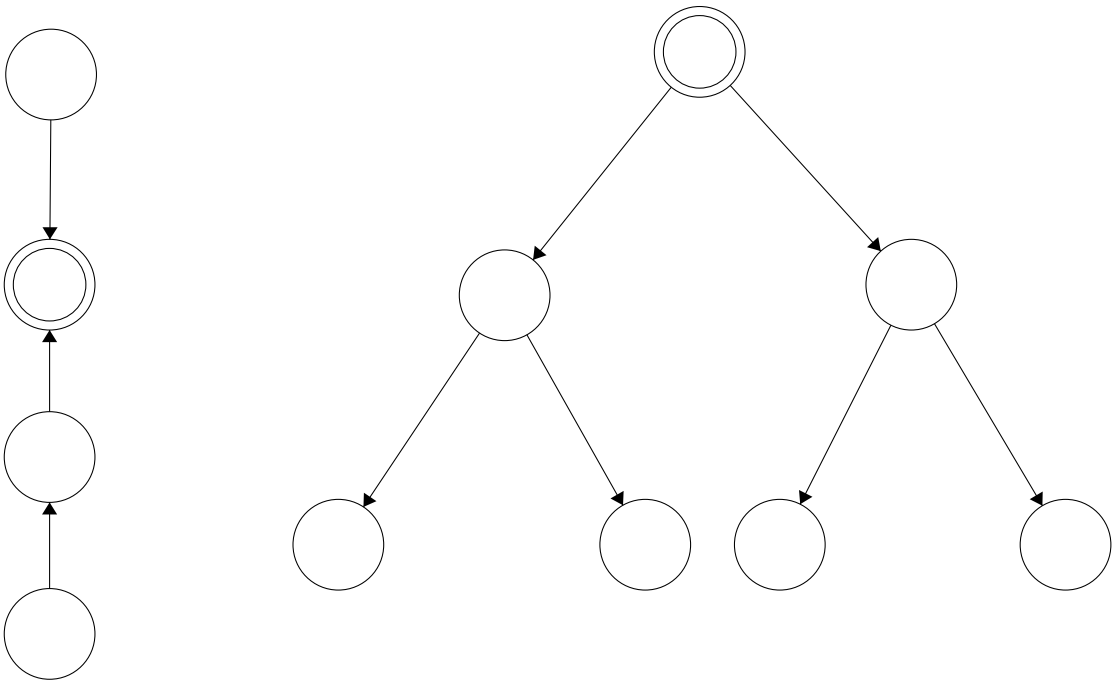


Figure 2.15: Rooted tree diagram 01

And here you are another example with a little bit strage tree (arborescence)

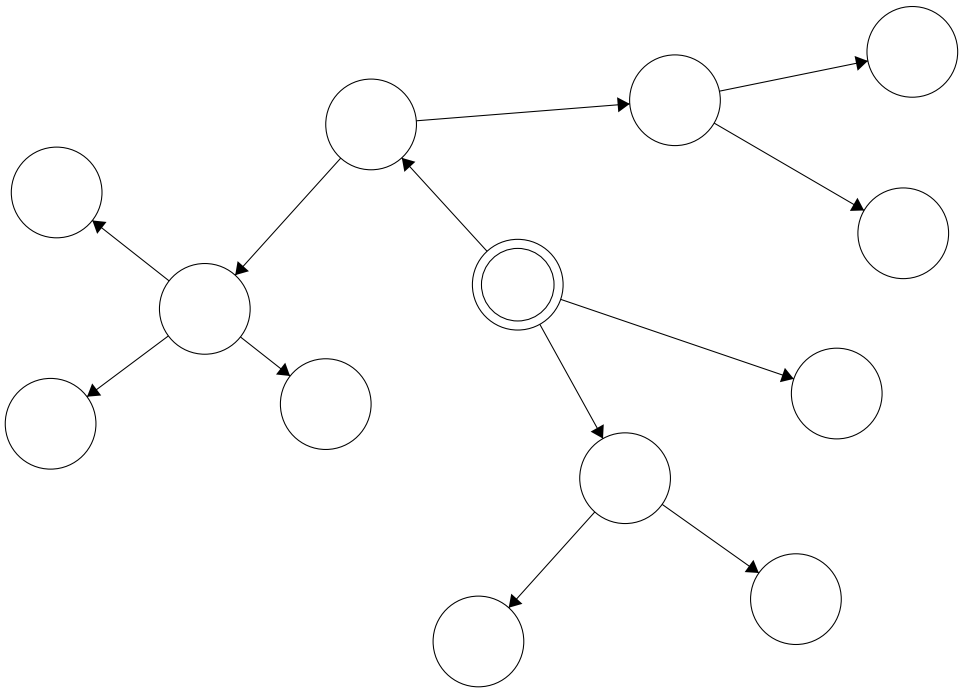


Figure 2.16: Rooted tree diagram 02

2.8 Tries

3 Algorithms

3.1 Dynamic Programming (DP)

If we are talking about dynamic programming most of the times it is an optimization of a solution that use recursion. As we know most of the times use recursion results in an exponential complexity in time so if we find a way to use dynamic programming we can reduce this complexity.

What is the main idea of use Dynamic programming? The idea is very simple, if we can store the result of the subproblems, we do not need to recalculate this subproblem and it reduces the complexity from exponential to a polynomial.

3.1.1 Examples

We can use the idea of calculate a factorial number, we know that a factorial number it is the multiplication of all the numbers from 1 to n, for example

- Calculate 2! = 1 * 2 = 2
- Calculate 3! = 1 * 2 * 3 = 6
- Calculate 5! = 1 * 2 * 3 * 4 * 5 = 120

But we need to consider some things about factorial numbers

1. 0! = 1
2. Factorial numbers exist in range of $[0,\infty)$

If we know this we can notice that we can get a formula to simplify this:

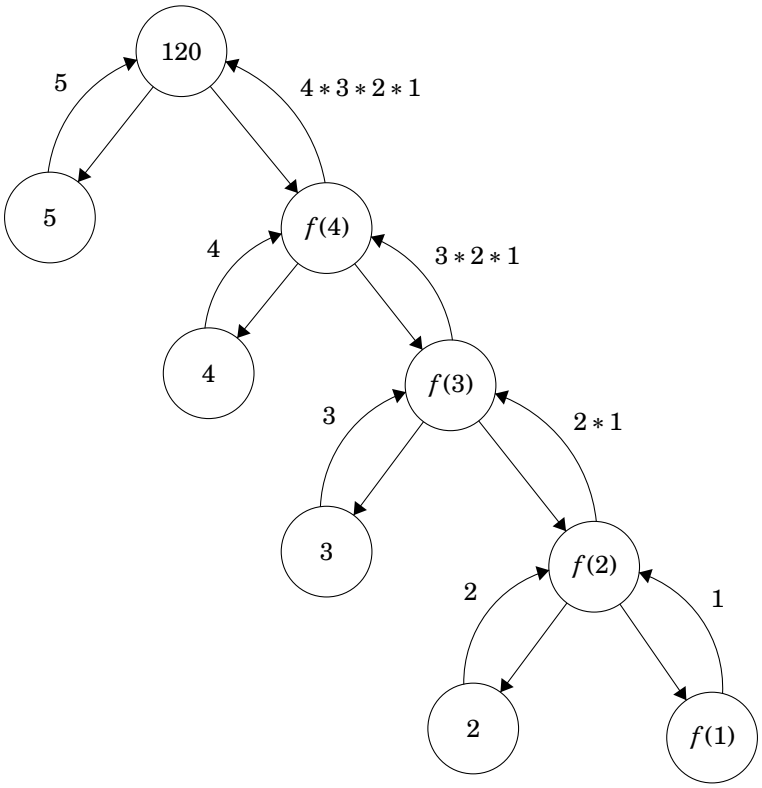
$$factorial(n) = \begin{cases} n * factorial(n-1) & \text{if } n > 1 \\ 1 & \text{otherwise} \end{cases}$$

Now we can solve this problem using a recursive program, it is very simple because we just need to programm the formula

```
#include<bits/stdc++.h>

using namespace std;
typedef long long int lli;

lli factorial( int n ){
    if( n > 1 )
        return n * factorial(n);
    return 1;
}

int main(){
    int n;
    cin >> n;
    cout << factorial( n ) << endl;
    return 0;
}
```

If we use the last program we can get the factorial of n, but the complexity of this algorithm is $O(n!)$ so it is awful, but using dynamic programming can we get a better complexity?

For me is easier to find a solution using dynamic programming using draws, so I go to make the draw of the recursive calls of the example 5!



As we can see we the small problem contains information of the base case has information of the next case and this case has information for the next case and so on, so we do not need to call. Maybe if you need to calculate just once a factorial number it is not necessary to store the information about a factorial number but imagine that you are doing a program that needs a lot of factorial numbers, if we use the recursive algorithm without dynamic programming we need to do for each query $O(n!)$ operations, and that is a lot. So we can store this information and make each query a lot of times and if we already calculated a factorial number we just need to return the information and it is $O(1)$

```
#include <bits/stdc++.h>

using namespace std;
typedef long long int lli;

vector< lli > factorial_number( 1000 + 1, -1 )
lli factorial( int n ){

    if( factorial_number[ n ] != -1)
        return factorial_number[ n ];

    if( n > 1 )
```

```
        return ( factorial_number[ n ] = (n * factorial( n -
            1 ) );

    return factorial_number[ n ] = 1;
}

int main(){
    int t, n;
    cin >> t;
    while( t-- ){
        cin >> n;
        cout << n << "! = " << factorial( n ) << endl;
    }
    return 0;
}
```

As you can see we have a little bit of code but we make more efficient get a factorial number, from $O(n!)$ complexity to $O(1)$ in the case of we already calculated the number, but we are using more memory in this case $O(n)$ in other words we are using a container which size is the biggest factorial that we already calculated.

Other example is calculate Fibonacci numbers, but first of all we need to know what is a factorial number. A factorial number is defined using the next function.

$$fibonacci(n) = \begin{cases} 0 & \text{if } n \leq 0 \\ n*(n-1)+(n-2) & \text{otherwise} \end{cases}$$

As the las example we first of all can sole this problem using a recursive algorithm, we just need to program the previous definition of a fibonacci number.

```
#include<bits/stdc++.h>

using namespace std;
typedef long long int lli;

lli fibonacci( int n ){
    if( n <= 0 )
        return 0;
    return ( n + fibonacci( n - 1 ) + fibonacci( n - 2 ) );
}

int main(){
    int n;
    return 0;
}
```