STL (STANDARD TEMPLATE LIBRARY)

C++ Templates

Templates are a feature of the C++ programming language that allows functions and classes to operate with generic types. This allows a function or class to work on many different data types without being rewritten for each one.

C++ STL has some containers (pre-build data structures) like **vectors, iterators, pairs etc.** These are all generic class which can be used to represent collection of any data type.

String

C++ provides a powerful alternative for the **char * .** It is not a built-in data type, but is a container class in the **Standard Template Library.** String class provides different string manipulation functions like concatenation, find, replace etc. Let us see how to construct a string type.

```
string s0;                                                    // s0 =
```

```
“”
string s1(“Hello”);                                    // s1 =
 “Hello”
string s2 (s1);                                       // s2 =
“Hello”
string s3 (s1, 1, 2);                                 // s3 =
“el”
string s4 ("Hello World", 5);                          // s4 =
 “Hello”
string s5 (5, ‘*’);                                   // s5 =
“*****”
string s6 (s1.begin(), s1.begin()+3);                  // s6
= “Hel”
```

Here are some member functions:

- **append():** Inserts additional characters at the end of the string (can also be done using ‘+’ or ‘+=’ operator). Its time complexity is **O(N)** where **N** is the size of the new string.
- **begin():** Returns an iterator pointing to the first character. Its time complexity is **O(1).**
- **clear():** Erases all the contents of the string and assign an empty string (“”) of length zero. Its time complexity is **O(1).**
- **compare():** Compares the value of the string with the string passed in the parameter and returns an integer accordingly. Its time complexity is **O(N + M)** where **N** is the size of the first string and **M** is the size of the second string.

- **copy():** Copies the substring of the string in the string passed as parameter and returns the number of characters copied. Its time complexity is **O(N)** where N is the size of the copied string.
- **empty():** Returns a boolean value, **true** if the string is empty and **false** if the string is not empty. Its time complexity is **O(1).**
- **end():** Returns an iterator pointing to a position which is next to the last character. Its time complexity is **O(1).**
- **erase():** Deletes a substring of the string. Its time complexity is **O(N)** where **N** is the size of the new string.
- **find():** Searches the string and returns the first occurrence of the parameter in the string. Its time complexity is **O(N)** where **N** is the size of the string.
- **insert():** Inserts additional characters into the string at a particular position. Its time complexity is **O(N)** where **N** is the size of the new string.
- **length():** Returns the length of the string. Its time complexity is **O(1).**
- **size():** Returns the length of the string. Its time complexity is **O(1).**
- **substr():** Returns a string which is the copy of the substring. Its time complexity is **O(N)** where **N** is the size of the substring.

Vector

Vectors are sequence containers that have dynamic size. In other

words, vectors are dynamic arrays. Just like arrays, vector elements are placed in contiguous storage location so they can be accessed and traversed using iterators. To traverse the vector we need the position of the first and last element in the vector which we can get through ***begin() and end()*** or we can use indexing from ***0 to size()***.

```
vector<int> a;                          // empty vector
of ints
vector<int> b (5, 10);                  // five ints wit
h value 10
vector<int> c (b.begin(),b.end());      // iterating thr
ough second
vector<int> d (c);                     // copy of c
```

Some of the member functions of vectors are:

- **at():** Returns the reference to the element at a particular position (can also be done using '[ ]' operator). Its time complexity is **O(1).**
- **back():** Returns the reference to the last element. Its time complexity is **O(1).**
- **begin():** Returns an iterator pointing to the first element of the vector. Its time complexity is **O(1).**
- **clear():** Deletes all the elements from the vector and assign an empty vector. Its time complexity is **O(N)** where **N** is the size of the vector.
- **empty():** Returns a boolean value, **true** if the vector is empty

and **false** if the vector is not empty. Its time complexity is **O(1).**

- **end():** Returns an iterator pointing to a position which is next to the last element of the vector. Its time complexity is **O(1).**
- **erase():** Deletes a single element or a range of elements. Its time complexity is **O(N + M)** where N is the number of the elements erased and M is the number of the elements moved.
- **front():** Returns the reference to the first element. Its time complexity is **O(1).**
- **insert():** Inserts new elements into the vector at a particular position. ts time complexity is **O(N + M)** where N is the number of elements inserted and M is the number of the elements moved .
- **pop_back():** Removes the last element from the vector. Its time complexity is **O(1).**
- **push_back():** Inserts a new element at the end of the vector. Its time complexity is **O(1).**
- **resize():** Resizes the vector to the new length which can be less than or greater than the current length. Its time complexity is **O(N)** where N is the size of the resized vector.
- **size():** Returns the number of elements in the vector. Its time complexity is **O(1).**

List

List is a sequence container which takes constant time in inserting and removing elements. List in STL is implemented as Doubly Link List. The elements from List cannot be directly accessed. For example to

access element of a particular position ,you have to iterate from a known position to that particular position.

```
list <int> LI;
list<int> LI(5, 100)    //here LI will have 5 int element
s of value 100
```

Some of the member function of List:

- **begin( ):** It returns an iterator pointing to the first element in list.Its time complexity is **O(1).**
- **end( ):** It returns an iterator referring to the theoretical element(doesn’t point to an element) which follows the last element.Its time complexity is **O(1).**
- **empty( ):** It returns whether the list is empty or not.It returns 1 if the list is empty otherwise returns 0.Its time complexity is **O(1).**
- **back( ):** It returns reference to the last element in the list.Its time complexity is **O(1).**
- **assign( ):** It assigns new elements to the list by replacing its current elements and change its size accordingly.It time complexity is **O(N).**
- **erase( ):** It removes a single element or the range of element from the list.Its time complexity is **O(N).**
- **front( ):** It returns reference to the first element in the list.Its time complexity is **O(1).**
- **push_back( ):** It adds a new element at the end of the list,

after its current last element. Its time complexity is **O(1).**

- **push_front( ):** It adds a new element at the beginning of the list, before its current first element. Its time complexity is **O(1).**
- **remove( ):** It removes all the elements from the list, which are equal to given element. Its time complexity is **O(N).**
- **pop_back( ):** It removes the last element of the list, thus reducing its size by 1. Its time complexity is **O(1).**
- **pop_front( ):** It removes the first element of the list, thus reducing its size by 1. Its time complexity is **O(1).**
- **insert( ):** It insert new elements in the list before the element on the specified position. Its time complexity is **O(N).**
- **reverse ( ):** It reverses the order of elements in the list. Its time complexity is **O(N).**
- **size( ):** It returns the number of elements in the list. Its time complexity is **O(1).**

Pair

Pair is a container that can be used to bind together a two values which may be of different types. Pair provides a way to store two heterogeneous objects as a single unit.

```
pair <int, char> p1;                    // default
pair <int, char> p2 (1, 'a');            // value inititi
alization
pair <int, char> p3 (p2);                // copy of p2
```

We can also initialize a pair using **make_pair()** function.

make_pair(x, y) will return a pair with first element set to **x** and second element set to **y**.

```
p1 = make_pair(2, ‘b’);
```

To access the elements we use keywords, **first** and **second** to access the first and second element respectively.

```
cout << p2.first << ‘ ‘ << p2.second << endl;
```

Set

Sets are containers which store only **unique values** and permit easy look ups. The values in the sets are stored in some specific order (like ascending or descending). Elements can only be inserted or deleted, but cannot be modified. We can access and traverse set elements using iterators just like vectors.

Multisets are containers that store elements following a specific order, and where **multiple elements can have equivalent values**. In a multiset, the value of an element also identifies it (the value is itself the key, of type T). The value of the elements in a multiset cannot be modified once in the container (the elements are always const), but they can be inserted or removed from the container.

```
set<int> s1;                                // Empty Set
```

```
int a[]= {1, 2, 3, 4, 5, 5};
set<int> s2 (a, a + 6);                 // s2 = {1, 2, 3, 4,
5}
set<int> s3 (s2);                       // Copy of s2
set<int> s4 (s3.begin(), s3.end());     // Set created usi
ng iterators

multiset<int> first;                            // empty mu
ltiset of ints
int myints[]= {10,20,30,20,20};
multiset<int> second (myints,myints+5);         // pointers
 used as iterators
multiset<int> third (second);                   // a copy o
f second
multiset<int> fourth (second.begin(), second.end()); //mu
ltiset created using iterators
```

Some of the member functions of set are:

- **begin():** Returns an iterator to the first element of the set. Its time complexity is **O(1).**
- **clear():** Deletes all the elements in the set and the set will be empty. Its time complexity is **O(N)** where N is the size of the set.
- **count():** Returns 1 or 0 if the element is in the set or not respectively. Its time complexity is **O(logN)** where N is the size of the set.

- **empty():** Returns true if the set is empty and false if the set has at least one element. Its time complexity is **O(1).**
- **end():** Returns an iterator pointing to a position which is next to the last element. Its time complexity is **O(1).**
- **erase():** Deletes a particular element or a range of elements from the set. Its time complexity is **O(N)** where N is the number of element deleted.
- **find():** Searches for a particular element and returns the iterator pointing to the element if the element is found otherwise it will return the iterator returned by **end().** Its time complexity is **O(logN)** where N is the size of the set.
- **insert():** insert a new element. Its time complexity is **O(logN)** where N is the size of the set.
- **size():** Returns the size of the set or the number of elements in the set. Its time complexity is **O(1).**

Maps

Maps are containers which store elements by mapping their **value against a particular key.** It stores the combination of key value and mapped value following a specific order. Here key value are used to uniquely identify the elements mapped to it. The data type of key value and mapped value can be different. Elements in map are always in sorted order by their corresponding key and can be accessed directly by their key using bracket operator ([ ]).
In map, key and mapped value have a pair type combination,i.e both key and mapped value can be accessed using pair type functionalities

with the help of iterators.

```
map <char ,int > mp;
mp[‘b’]  = 1;
```

In map mp , the values be will be in sorted order according to the key.

Some Member Functions of map:

- **at( ):** Returns a reference to the mapped value of the element identified with key.Its time complexity is **O(logN).**
- **count( ):** searches the map for the elements mapped by the given key and returns the number of matches. As map stores each element with unique key, then it will return 1 if match if found otherwise return 0.Its time complexity is **O(logN).**
- **clear( ):** clears the map, by removing all the elements from the map and leaving it with its size 0.Its time complexity is **O(N).**
- **begin( ):** returns an iterator(explained above) referring to the first element of map.Its time complexity is **O(1).**
- **end( ):** returns an iterator referring to the theoretical element(doesn’t point to an element) which follows the last element.Its time complexity is **O(1).**
- **empty( ):** checks whether the map is empty or not. It doesn’t modify the map.It returns 1 if the map is empty otherwise returns 0.Its time complexity is **O(1).**
- **erase( ):** removes a single element or the range of element from the map.

- **find( ):** Searches the map for the element with the given key, and returns an iterator to it, if it is present in the map otherwise it returns an iterator to the theoretical element which follows the last element of map.Its time complexity is **O(logN).**
- **insert( ):** insert a single element or the range of element in the map.Its time complexity is **O(logN)**, when only element is inserted and **O(1)** when position is also given.

Stack

Stack is a container which follows the LIFO (Last In First Out) order and the elements are inserted and deleted from one end of the container. The element which is inserted last will be extracted first.

```
stack <int> s;
```

Some of the member functions of Stack are:

- **push( ):** Insert element at the top of stack. Its time complexity is **O(1).**
- **pop( ):** removes element from top of stack. Its time complexity is **O(1).**
- **top( ):** access the top element of stack. Its time complexity is **O(1).**
- **empty( ):** checks if the stack is empty or not. Its time complexity is **O(1).**

- **size( ):** returns the size of stack. Its time complexity is **O(1).**

Queue

Queue is a container which follows **FIFO** order **(First In First Out)** . Here elements are inserted at one end (rear) and extracted from another end(front).

```
queue <int> q;
```

Some member function of Queues are:

- **push( ):** inserts an element in queue at one end(rear). Its time complexity is **O(1).**
- **pop( ):** deletes an element from another end if queue(front). Its time complexity is **O(1).**
- **front( ):** access the element on the front end of queue. Its time complexity is **O(1).**
- **empty( ):** checks if the queue is empty or not. Its time complexity is **O(1).**
- **size( ):** returns the size of queue. Its time complexity is **O(1).**

Priority Queue

A priority queue is a container that provides constant time extraction of the largest element, at the expense of logarithmic insertion. It is similar to the heap in which we can add element at any time but only

the maximum element can be retrieved. In a priority queue, an element with high priority is served before an element with low priority.

```
priority_queue<int> pq;
```

To make a min-priority queue, declare priority queue as:

```
#include <functional> //for greater <int>
//min priority queue
priority_queue < int, vector < int > , greater <int> > pq
;
```

Some member functions of priority queues are:

- **empty():** Returns true if the priority queue is empty and false if the priority queue has at least one element. Its time complexity is **O(1).**
- **pop():** Removes the largest element from the priority queue. Its time complexity is **O(logN)** where N is the size of the priority queue.
- **push():** Inserts a new element in the priority queue. Its time complexity is **O(logN)** where N is the size of the priority queue.
- **size():** Returns the number of element in the priority queue. Its time complexity is **O(1).**
- **top():** Returns a reference to the largest element in the priority

queue. Its time complexity is **O(1).**

Deque

Double-ended queues are sequence containers with dynamic sizes that can be expanded or contracted on both ends (either its front or its back).

```
deque<int> first;                        // empty deque of in
ts
deque<int> second (4,100);              // four ints with val
ue 100
deque<int> third (second.begin(),second.end());// iterati
ng through second
deque<int> fourth (third);                 // a copy of thir
d
```

Some member functions of deque are:

- **assign():** Assigns new contents to the deque container, replacing its current contents, and modifying its size accordingly.
- **at(n):** Returns a reference to the element at position n in the deque container object.
- **back():** Returns a reference to the last element in the container.
- **begin():** Returns an iterator pointing to the first element in the

deque container.

- **empty():** Returns whether the deque container is empty (i.e. whether its size is 0).
- **end():** Returns an iterator referring to the past-the-end element in the deque container.
- **erase():** Removes from the deque container either a single element (position) or a range of elements ([first,last)).
- **front():** Returns a reference to the first element in the deque container.
- **pop_back():** Removes the last element in the deque container, effectively reducing the container size by one.
- **pop_front():** Removes the first element in the deque container, effectively reducing its size by one.
- **push_back():** Adds a new element at the end of the deque container, after its current last element.
- **push_front():** Inserts a new element at the beginning of the deque container, right before its current first element.
- **size():** Returns the number of elements in the deque container.

Iterator

An iterator is any object that, points to some element in a range of elements (such as an array or a container) and has the ability to iterate through those elements using a set of operators (with at least the increment (++) and dereference (*) operators).

```
For Vector:
```

```
vector <int>::iterator  it;
For List:
list <int>::iterator it;
etc....
```

<algorithm>

The header <algorithm> defines a collection of functions especially designed to be used on ranges of elements.

binary_search(first,last,val)

Returns true if any element in the range [first,last) is equivalent to val, and false otherwise.

```
binary_search (v.begin(), v.end(), 3)) //v is a vector
```

find(first,last,val)

Returns an iterator to the first element in the range [first,last) that compares equal to val. If no such element is found, the function returns last.

```
it = find (myvector.begin(), myvector.end(), 30); //it is
 an iterator
```

lower_bound(first,second,val)

Returns an iterator pointing to the first element in the range [first,last)

which does not compare less than val.

```
it = lower_bound (v.begin(), v.end(), 20); //v is a vecto
r
```

upper_bound(first,second,val)

Returns an iterator pointing to the first element in the range [first,last) which compares greater than val.

```
it = upper_bound (v.begin(), v.end(), 20); //v is a vecto
r
```

max(a,b)

Returns the largest of a and b. If both are equivalent, a is returned.

```
cout << max(a,b);
```

min(a,b)

Returns the smallest of a and b. If both are equivalent, a is returned.

```
cout << min(a,b);
```

reverse(first,last)

Reverses the order of the elements in the range [first,last).

```
reverse(myvector.begin(),myvector.end());
```

rotate(first,middle,last)

Rotates the order of the elements in the range [first,last), in such a way that the element pointed by middle becomes the new first element.

```
rotate(myvector.begin(),myvector.begin()+3,myvector.end()
);
```

sort(first,last)

Sorts the elements in the range [first,last) into ascending order.

```
sort(v.begin(),v.end());
```

swap(a,b)

Exchanges the values of a and b.

```
swap(a,b);
```

next_permutation(first,last)

Rearranges the elements in the range [first,last) into the next lexicographically greater permutation.

```
next_permutation(v.begin(),v.end());
```

GRAPH (Adjacency List

Implementation)

An adjacency list implementation of graph can be easily represented using a vector.

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

vector < pair < int,int > > graph[100005];  //graph
int visited[100005];    //visited array

void dfs(int cur){  //dfs method
    if(visited[cur])return;
    cout<<cur<<" "; //printing current node
    visited[cur] = 1;   //setting current node as visited
    for(int i=0;i<graph[cur].size();++i){
        dfs(graph[cur][i].first);
    }
}

int main() {
    int n,m;
    int a,b,w;
    cin>>n>>m;
    for(int i=0;i<m;++i){
        cin>>a>>b>>w;   //edge between a and b with weigh
```

```
t w
        graph[a].push_back(make_pair(b,w));
        graph[b].push_back(make_pair(a,w));
    }
    cin>>a;
    dfs(a);
    return 0;
}
```

Created By *Shubham Rawat*