Notas:

Calcular el número de dígitos: floor(log10(N)) + 1;

Fórmula números de Catalán:

LCS

if (X[r-1] == Y[c-1]) then //si coinciden +1 arriba a la izquierda

L[r][c] = L[r-1][c-1] + 1

Else //sino el max de arriba y el de la izq

L[r][c] = max { L[r-1][c], L[r][c-1] }

LIS

for (int i = 1; i < n; i++ ) //de 1 a n

{

lis[i] = 1;

for (int j = 0; j < i; j++ ) //de 1 a i

//si es mayor (LIS) y lis[i] puede aumentar cogiendo la nueva serie +1, se coge

if ( arr[i] > arr[j] && lis[i] < lis[j] + 1)

lis[i] = lis[j] + 1;

}

STD

Sort

When we use sort, we can use either comparators (objects) or functions.

Object:

sort(v.begin(),v.end(),greater<**int**>());

Function:

**bool** compare(**int** i,**int** j){

**if**(i>j) **returntrue**;

**elsereturnfalse**;

}

**int** main(){

vector<**int**> v;

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

}

Find

**int** xs[]={5,4,6,7,8,3,69,65};

**if**(find(xs,&xs[8]+1,65)!=&xs[8]+1){

*//The element is present*

*//The element pointed by the last iterator will not be taken into account*

*//This function works with iterators and with addresses*

}

Upper and lower bound

upper\_bound(first, last, val);

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

lower\_bound(first, last, val);

Returns an iterator pointing to the first element in the range [first,last) which does not compare less than *val*.

String

Insert

string s="Hello";

s.insert(0,"a");*//"aHello"*

string s="Hello";

s.insert(0,5,'a'); *//"aaaaaHello"*

Find

string s = "Hello world";

s.find("Hello"); *//Returns the index at which this occurs (0)*

**if**(s.find("Hello")!= string::npos){

*//The substring is present*

}

Substr

string s ="Hello world";

string a=s.substr(0,2);

*//A substring starting from 0 with 2 characters: "He"*

Stringstream

#include <sstream>

stringstream ss;

ss.clear();

contents\_of\_ss=ss.str();

ss.str(string\_to\_insert);

Math

To define the constants, use:

#define USE\_MATH\_DEFINES

Regarding logarithms:

**int** a= log(2); *//natural logarithm*

**int** b= log10(2); *//base 10 logarithm*

**int** c= log2(2); *//base 2 logarithm*

Graphs

Euler theorem

We have a path that goes through every edge exactly once if the degree of every node is even or if there are exactly two nodes with odd degree.

We have a circuit that goes through every edge exactly once if the degree of every node is even.

Dijkstra

**void** dijkstra(**int** start){

**int** dis,pos;

dist.assign(adj.size(), LONG\_MAX);

dist[start]=0;

priority\_queue<ii,vector<ii>,greater<ii>> pq;

pq.push(make\_pair(0,start));

**while**(!pq.empty()){

dis=pq.top().first;

pos=pq.top().second;

pq.pop();

**if**(dis>dist[pos]){

**continue**;

}

**for**(**auto** el:adj[pos]){

**if**(dis+el.second<dist[el.first]){

dist[el.first]=el.second+dis;

pq.push(make\_pair(dist[el.first],el.first));

}

}

}

}

Ufds

**struct** DisjointSets{

**int** n;

*//Constructor*

DisjointSets(**int** nn){

n=nn;

**for**(**int** i=0;i<=n;++i){

ranks[i]=0;

parent[i]=i;

}

}

**int** findRoot(**int** u){*//find function with compression*

**if**(u!=parent[u]) parent[u]=findRoot(parent[u]);

**return**parent[u];

}

**void** Union(**int** x,**int** y){*//Union by ranks*

x=findRoot(x);

y=findRoot(y);

**if**(ranks[x]<ranks[y]) parent[x]=y;

**elseif**(ranks[x]>ranks[y]) parent[y]=x;

**else**{

*//If ranks are the same, make one root and increment its ranks;*

ranks[x]++;

parent[y]=x;

}

}

};

Kruskal

**struct** Graph{

**int** V,E;

vector<pair<**int**,pair<**int**,**int**>>> edges;

*//Constructor*

Graph(**int** nv, **int** ne){

V=nv;

E=ne;

}

*//Functions*

**void** addEdge(**int** u,**int** v,**int** w){

edges.push\_back(pai(w,u,v));

}

**long** kruskalMST();

};

**long**Graph::kruskalMST(){

**int** output=0;

**int** u,v;

**int** addededges=0;

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

DisjointSets ds(V);

**for**(**auto** it =edges.begin();it!=edges.end();it++){

u=it->second.first; v=it->second.second;

**int** set\_u=ds.findRoot(u), set\_v=ds.findRoot(v);

**if**(set\_u!=set\_v){

addededges++;

output+=it->first;

ds.Union(set\_u, set\_v);

}

**if**(addededges==V-1) **break**;

}

**if**(addededges!=V-1) **return**LONG\_MAX;

**return** output;

}

BFS

BFS to find out if all the nodes are connected in a graph (It also has to be a tree)

**bool** BFS(){

vector<**int**> visited;

queue<**int**> queu;

**int** current;

**bool** stay=**true**;

**bool** success;

visited.assign(n, 0);

queu.push(0);

**while**(!queu.empty()&&stay){

current=queu.front();

queu.pop();

visited[current]=2; *//Close*

**for**(**auto** el:adylist[current]){

**if**(!visited[el]){

queu.push(el);

visited[el]=1;

}

**elseif**(visited[el]==1){ *//Just seen*

stay=**false**;

**break**;

}

}

}

**if**(!stay) **returnfalse**;

success=**true**;

**for**(**auto** el:visited){

**if**(el!=2){

success=**false**;

**break**;

}

}

**return** success;

}

Max Flow

**int** parent[Mmax];

**bool** visited[Mmax]; *//s=source, t=sink MAX FLOW*

**int** graph[Mmax][Mmax];

**bool** BFS(){

**int** u,v;

memset(visited,**false**,M);

queue<**int**> q;

q.push(s);

visited[s]=**true**;

parent[s]=-1;

**while**(!q.empty()){

u=q.front();

q.pop();

**for**(v=0;v<M;++v){

**if**(!visited[v]&&graph[u][v]>0){

q.push(v);

parent[v]=u;

visited[v]=**true**;

}

}

}

**return** (visited[t]==**true**);

}

**int** fordFulkerson(){

**int** maxflow=0,u,v;

**while**(BFS()){

**int** path\_flow=INT\_MAX;

**for**(v=t;v!=s;v=parent[v]){

u=parent[v];

path\_flow=min(path\_flow,graph[u][v]);

}

**for**(v=t;v!=s;v=parent[v]){

u=parent[v];

graph[u][v]-=path\_flow;

graph[v][u]+=path\_flow;

}

maxflow+=path\_flow;

}

**return** maxflow;

}

Rare Algorithms

Maximum of the subsets of a set (258)

Use deque. The deque has to store the elements that are or could be a maximum.

**for**(i=0;i<s\_subset;++i){

/\*\* We will only keep the elements if they could be maximums \*\*/

**while**(!Qi.empty()&&Data[i]>=Data[Qi.back()]){

Qi.pop\_back();

}

Qi.push\_back(i);

}

**for**(;i<s\_total;++i){

cout<<Data[Qi.front()]<<" "; *//print the current maximum*

/\*\* Eliminate the elements that are out of range \*\*/

**while**(!Qi.empty()&&Qi.front()<=i-s\_subset){

Qi.pop\_front();

}

/\*\* Eliminate the elements that will never be a maximum \*\*/

**while**(!Qi.empty()&&Data[i]>=Data[Qi.back()]){

Qi.pop\_back();

}

Qi.push\_back(i);

}

cout<<Data[Qi.front()]<<"\n"; *//print the current maximum*

}