Graph

How to determine the level of each node in the given tree?

```
vector <int> v[10]; //Vector for maintaining adjacency list
int level[10]; //To determine the level of each node
bool vis[10]; //Mark the node if visited
void bfs(int s) {
    queue <int> q;
    q.push(s);
    level[s] = 0; //Setting the level of the source node as 0
    vis[s] = true;
    while(!q.empty()){
       int p = q.front();
       q.pop();
       for(int i = 0;i < v[ p ].size() ; i++){
            if(vis[ v[ p ][ i ] ] == false){
            //Setting the level of each node
              with an increment in the level of parent node
             level[ v[ p ][ i ] ] = level[ p ]+1;
             q.push(v[ p ][ i ]);
             vis[ v[ p ][ i ] ] = true}}}}
```

0-1 BFS

```
void bfs (int start){
    deque <int > Q;     //Double-ended queue
    Q.push_back( start);
    distance[ start ] = 0;
    while( !Q.empty()){
      int v = Q.front();
      Q.pop_front();
      for( int i = 0 ; i < edges[v].size(); i++){


/* if distance of neighbour of v from start node is greater than
sum of distance of v from start node and edge weight between v and
its neighbour (distance between v and its neighbour of v) ,then
change it */

if(distance[edges[v][i].first] > distance[v] + edges[v][i].second)
{
distance[edges[v][i].first] = distance[v] + edges[v ][i].second;

/*if edge weight between v and its neighbour is 0 then push it to
front of double ended queue else push it to back*/

          if(edges[v][i].second == 0)
                    {
                    Q.push_front( edges[ v ][ i ].first);
                    }
                    else
                    {
                     Q.push_back( edges[ v ][ i ].first);

}}}}}
```

Depth First Search (DFS)

```
    vector <int> adj[10];
    bool visited[10];
    void dfs(int s) {
        visited[s] = true;
        for(int i = 0;i < adj[s].size();++i)    {
         if(visited[adj[s][i]] == false)
             dfs(adj[s][i]);
        }
    }

    void initialize() {
        for(int i = 0;i < 10;++i)
         visited[i] = false;
    }

    int main() {
        int nodes, edges, x, y, connectedComponents = 0;
```

```
        cin >> nodes;   //Number of nodes
        cin >> edges;  //Number of edges
        for(int i = 0;i < edges;++i) {
         cin >> x >> y;

     //Undirected Graph

         adj[x].push_back(y); //Edge from vertex x to vertex y
         adj[y].push_back(x); //Edge from vertex y to vertex x
        }

        initialize();  //Initialize all nodes as not visited

        for(int i = 1;i <= nodes;++i) {
         if(visited[i] == false)     {
             dfs(i);
             connectedComponents++;
         }
        }
        cout << "Number of connected components: " <<
connectedComponents << endl;
        return 0;
    }
```

Time complexty O(V+E)

Kruskal’s Algorithm

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

```
using namespace std;
const int MAX = 1e4 + 5;
int id[MAX], nodes, edges;
pair <long long, pair<int, int> > p[MAX];
void initialize()
{
    for(int i = 0;i < MAX;++i)
        id[i] = i;
}

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

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

long long kruskal(pair<long long, pair<int, int> > p[])
{
    int x, y;
    long long cost, minimumCost = 0;
    for(int i = 0;i < edges;++i)
    {
   /* Selecting edges one by one in increasing order from the
beginning*/

        x = p[i].second.first;
        y = p[i].second.second;
        cost = p[i].first;
        // Check if the selected edge is creating a cycle or not
        if(root(x) != root(y))
        {
            minimumCost += cost;
```

```
            union1(x, y);
        }
    }
    return minimumCost;
}

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

Time complexty O(ElogV)

Prim’s Algorithm

Prim’s Algorithm also use Greedy approach to find the minimum spanning tree. In Prim’s Algorithm we grow the spanning tree

from a starting position. Unlike an **edge** in Kruskal's, we add **vertex** to the growing spanning tree in Prim's.

```
const int MAX = 1e4 + 5;
typedef pair<long long, int> PII;
bool marked[MAX];
vector <PII> adj[MAX];

long long prim(int x)
{
    priority_queue<PII, vector<PII>, greater<PII> > Q;
    int y;
    long long minimumCost = 0;
    PII p;
    Q.push(make_pair(0, x));
    while(!Q.empty())
    {
        // Select the edge with minimum weight
        p = Q.top();
        Q.pop();
        x = p.second;
        // Checking for cycle
        if(marked[x] == true)
            continue;
        minimumCost += p.first;
        marked[x] = true;
        for(int i = 0;i < adj[x].size();++i)
        {
            y = adj[x][i].second;
            if(marked[y] == false)
                Q.push(adj[x][i]);
        }
    }
    return minimumCost;
}

int main()
{
    int nodes, edges, x, y;
    long long weight, minimumCost;
    cin >> nodes >> edges;
    for(int i = 0;i < edges;++i)
    {
        cin >> x >> y >> weight;
        adj[x].push_back(make_pair(weight, y));
        adj[y].push_back(make_pair(weight, x));
```

```
    }
    // Selecting 1 as the starting node
    minimumCost = prim(1);
    cout << minimumCost << endl;
    return 0;
}
```

Time Complexty O((V+E)logV)

Bellman Ford's Algorithm:

Bellman Ford's algorithm is used to find the shortest paths from the source vertex to all other vertices in a weighted graph. It depends on the following concept: Shortest path contains at

most n-1 edges, because the shortest path couldn't have a cycle.

Assume Source node is (0)

```
vector <int> v [2000 + 10];
    int dis [1000 + 10];

    for(int i = 0; i < m + 2; i++){

        v[i].clear();
        dis[i] = 2e9;
    }

   for(int i = 0; i < m; i++){

        scanf("%d%d%d", &from , &next , &weight);

        v[i].push_back(from);
        v[i].push_back(next);
        v[i].push_back(weight);
   }

    dis[0] = 0;
    for(int i = 0; i < n - 1; i++){
        int j = 0;
        while(v[j].size() != 0){

            if(dis[ v[j][0]  ] + v[j][2] < dis[ v[j][1] ] ){
                dis[ v[j][1] ] = dis[ v[j][0]  ] + v[j][2];
            }
            j++;
        }
    }
```

Time complexity O(V.E)

A very important application of Bellman Ford is to check if there is a negative cycle in the graph,

Dijkstra's Algorithm

Dijkstra's algorithm has many variants but the most common one is to find the shortest paths from the source vertex to all other vertices in the graph.

```
#define SIZE 100000 + 1

vector < pair < int , int > > v [SIZE];   // each vertex has all the connected vertices with the edges weights
int dist [SIZE];
bool vis [SIZE];

void dijkstra(){
 // set the vertices distances as infinity
 memset(vis, false , sizeof vis); // set all vertex as unvisited
    dist[1] = 0;
    multiset < pair < int , int > > s; // multiset do the job as a min-priority queue

    s.insert({0 , 1}); // insert the source node with distance = 0
```

```
    while(!s.empty()){

        pair <int , int> p = *s.begin();// pop the vertex with the
minimum distance
        s.erase(s.begin());

        int x = p.s; int wei = p.f;
        if( vis[x] ) continue; // check if the popped vertex is
visited before
         vis[x] = true;

        for(int i = 0; i < v[x].size(); i++){
            int e = v[x][i].f; int w = v[x][i].s;
            if(dist[x] + w < dist[e]  ){ // check if the next
vertex distance could be minimized
                dist[e] = dist[x] + w;
                s.insert({dist[e],  e} );           // insert the
next vertex with the updated distance
            }
        }
    }
}
```

Floyd–Warshall's Algorithm

```
for(int k = 1; k <= n; k++){
    for(int i = 1; i <= n; i++){
        for(int j = 1; j <= n; j++){
            dist[i][j] = min( dist[i][j], dist[i][k] + dist[k][j]);
        }
    }
}
```

Time Complexity (V^3)

binaryExponentiation

```
int binaryExponentiation(int x,int n)
{
    if(n==0)
        return 1;
    else if(n%2 == 0)        //n is even
        return binaryExponentiation(x*x,n/2);
    else                             //n is odd
        return x*binaryExponentiation(x*x,(n-1)/2);
}
```

modularExponentiation

```
int modularExponentiation(int x,int n,int M)
{
    if(n==0)
        return 1;
    else if(n%2 == 0)        //n is even
        return modularExponentiation((x*x)%M,n/2,M);
    else                             //n is odd
        return (x*modularExponentiation((x*x)%M,(n-1)/2,M))%M;

}
```

```
int modularExponentiation(int x,int n,int M)
{
    int result=1;
    while(n>0)
    {
        if(power % 2 ==1)
            result=(result * x)%M;
        x=(x*x)%M;
        n=n/2;
    }
    return result;
}
```

```
int modInverse(int A,int M)
{
    A=A%M;
    for(int B=1;B<M;B++)
        if((A*B)%M)==1)
            return B;
}
```

Time Complext O(M)

```
int modInverse(int A,int M)
{
    return modularExponentiation(A,M-2,M);
}
```

Sieve of Eratosthenes

You can use the *Sieve of Eratosthenes* to find all the

prime numbers that are less than or equal to a given

number N or to find out whether a number is a prime number.

```
void sieve(int N) {
        bool isPrime[N+1];
        for(int i = 0; i <= N;++i) {
            isPrime[i] = true;
        }
        isPrime[0] = false;
        isPrime[1] = false;
        for(int i = 2; i * i <= N; ++i) {
             if(isPrime[i] == true) { //Mark all the
multiples of i as composite numbers
                 for(int j = i * i; j <= N ;j += i)
                     isPrime[j] = false;
            }
        }
    }
```

Sieve of Eratosthenes on the segment:

Sometimes you need to find all the primes that are in the range

and not in , where is a large number.

```
bool isPrime[r - l + 1]; //filled by true
for (long long i = 2; i * i <= r; ++i) {
    for (long long j = max(i * i, (l + (i - 1)) / i  * i); j <= r;
j += i) {
        isPrime[j - l] = false;
    }
}
for (long long i = max(l, 2); i <= r; ++i) {
    if (isPrime[i - l]) {
        //then i is prime
    }
}
```

Modification of Sieve of Eratosthenes for fast factorization

```
int minPrime[n + 1];
for (int i = 2; i * i <= n; ++i) {
    if (minPrime[i] == 0) {         //If i is prime
        for (int j = i * i; j <= n; j += i) {
            if (minPrime[j] == 0) {
                minPrime[j] = i;
            }
        }
    }
}

for (int i = 2; i <= n; ++i) {
    if (minPrime[i] == 0) {
        minPrime[i] = i;
    }
}
```

```
vector<int> factorize(int n) {
    vector<int> res;
    while (n != 1) {
        res.push_back(minPrime[n]);
        n /= minPrime[n];
    }
    return res;

}
```

Dis-Joint Set

```
 int root(int Arr[ ],int i)
    {
        while(Arr[ i ] != i) //chase parent of current element
until it reaches root
        {
         i = Arr[ i ];
        }
        return i;
    }


    /*modified union function where we connect the elements by
changing the root of one of the elements*/

    int union(int Arr[ ] ,int A ,int B)
    {
        int root_A = root(Arr, A);
        int root_B = root(Arr, B);

        //setting parent of root(A) as root(B)
        Arr[ root_A ] = root_B ;

    }
    bool find(int A,int B)
    {
 /*if A and B have the same root,
it means that they are connected.*
        if( root(A)==root(B) )
        return true;
        else
        return false;
    }
```

Weghited Union

```
void weighted-union(int Arr[ ],int size[ ],int A,int B)
    {
        int root_A = root(A);
        int root_B = root(B);
        if(size[root_A] < size[root_B ])
        {
    Arr[ root_A ] = Arr[root_B];
    size[root_B] += size[root_A];
    }
        else
        {
    Arr[ root_B ] = Arr[root_A];
    size[root_A] += size[root_B];
    }

    }
```

Important String functions and Techniques

Checking a type of character

```
isdigit(str[0])
isalpha(str[0])
isalnum(str[0])
ispunct(str[0])
isspace(str[0])
islower(str[0])
isupper(str[0])// converting character to upper or lower case
toupper(str[0])
tolower(str[0])

//converting string to integer
string str = "124";
```

```
int i = atoi(str.c_str());
//converting integer to string
int i = 42;
stringstream ss;
ss << i;
//alternative method
for converting integer to string
string s = ss.str();
string numStr = to_string(i);


vector<int> vec;

//getting length of vector

int length = vec.size();

//clearing content of vector

vec.clear();

//defining intial size of vector

vec.reserve(100);

//getting capacity of vector

vec.capacity();

//inserting elemets to vector

vec.push_back(5);

//accessing elements of vector

int temp = vec[0];

//intializing vector with some value

vector<float> vec2(10,0.0);

//removing nth element from vector

int n = 3;

vec.erase(vec.begin()+n);//removing(erasing) last element from
vector

vec.pop_back();//swaping two elements

swap(vec[0],vec[1]);//accessing last element

vec.back();

vec[vec.size()-1];//reversing vector

reverse(vec.begin(), vec.end());//finding element with max or min
value

vector<int> vec3({10,30,20});int mx =
*max_element(vec3.begin(),vec3.end());
int mn = *min_element(vec3.begin(),vec3.end());//finding sum of all
values
int baseVal = 100; //for finding sum keep baseVal zero
int sum = accumulate(vec3.begin(),vec3.end(),baseVal); //returns
160
```

Initializing vector using set

```
set<int> s;
s.insert(1);
s.insert(2);vector<int> vec(s.begin(), s.end());
```

Important search functions and their behavior

binary_search, upper_bound, lower_bound works only on sorted vector

- binary_search returns ‘true/false’.
- To find a position of the first occurrence of a search_element, use a lower_bound function
- To find a position of the last occurrence of a search_element, use a upper_bound function and reduce iterator by 1
- lower_bound returns: *the first occurrence of search_element if it is present otherwise next greater element of search_element*
- upper_bound returns: *the next greater element of search_element*
- If no element in the vector compares greater than search_element, the functions(lower_bound, upper_bound) returns vec.end()

```
vector<int> vec {10,20,20,30,30,40};//vector needs to be sorted for
binary_search, lower_bound
//and upper_bound to work correctly// binary_search
bool found = binary_search(vec.begin(),vec.end(),30);// lower_bound
vector<int>::iterator it = lower_bound(vec.begin(),vec.end(),30);
cout << "pos :" << it - vec.begin() << endl; // pos : 3//
lower_bound
vector<int>::iterator it = lower_bound(vec.begin(),vec.end(),40);
cout << "pos :" << it - vec.begin() << endl; // pos : 5//
upper_bound
vector<int>::iterator it = upper_bound(vec.begin(),vec.end(),30);
cout << "pos :" << it - vec.begin() << endl; // pos : 5//
upper_bound
vector<int>::iterator it = upper_bound(vec.begin(),vec.end(),40);
cout << "pos :" << it - vec.begin() << endl; // pos : 6
```

Copying vector

```
vector<int> vec1(10,0);// Copying while creating
vector<int> vec2(vec1);// Copying after creation
vector<int> vec3;
vec3 = vec1;// Copying specific section after creation
vector<int> vec4;
vec4.assign(vec1.begin()+1, vec1.end());
```

Longest Increasing Subsequence

```
/* Dynamic Programming C++ implementation of LIS problem */

#include<bits/stdc++.h>

using namespace std;


/* lis() returns the length of the longest increasing

subsequence in arr[] of size n */

int lis( int arr[], int n )

{

    int lis[n];

    lis[0] = 1;

    /* Compute optimized LIS values in bottom up manner */

    for (int i = 1; i < n; i++ )

    {

        lis[i] = 1;

        for (int j = 0; j < i; j++ )

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

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

    }

    // Return maximum value in lis[]

    return *max_element(lis, lis+n);

}

/* Driver program to test above function */

int main()

{

    int arr[] = { 10, 22, 9, 33, 21, 50, 41, 60 };

    int n = sizeof(arr)/sizeof(arr[0]);

    printf("Length of lis is %d\n", lis( arr, n ) );

    return 0;

}
```

Longest Common Subsequence | DP-4

```
int max(int a, int b);

/* Returns length of LCS for X[0..m-1], Y[0..n-1] */

int lcs( char *X, char *Y, int m, int n )
```

```
{
    int L[m + 1][n + 1];
    int i, j;
    /* Following steps build L[m+1][n+1] in
    bottom up fashion. Note that L[i][j]
    contains length of LCS of X[0..i-1]
    and Y[0..j-1] */
    for (i = 0; i <= m; i++)
    {
        for (j = 0; j <= n; j++)
        {
        if (i == 0 || j == 0)
            L[i][j] = 0;

        else if (X[i - 1] == Y[j - 1])
            L[i][j] = L[i - 1][j - 1] + 1;

        else
            L[i][j] = max(L[i - 1][j], L[i][j - 1]);
        }
    }

    /* L[m][n] contains length of LCS
    for X[0..n-1] and Y[0..m-1] */
    return L[m][n];
}

/* Utility function to get max of 2 integers */
int max(int a, int b)
{
    return (a > b)? a : b;
}

// Driver Code
int main()
{
    char X[] = "AGGTAB";
    char Y[] = "GXTXAYB";

    int m = strlen(X);
    int n = strlen(Y);

    cout << "Length of LCS is "
        << lcs( X, Y, m, n );

    return 0;
}


// This code is contributed by rathbhupendra
```

Time Complexty (mn)

Edit Distance | DP-5

Given two strings str1 and str2 and below operations that can performed on str1. Find minimum number of edits (operations) required to convert ‘str1’ into ‘str2’.

```
// Utility function to find the minimum of three numbers
int min(int x, int y, int z)
{
    return min(min(x, y), z);
}
int editDistDP(string str1, string str2, int m, int n)
{
    // Create a table to store results of subproblems
    int dp[m+1][n+1];
```

```
    // Fill d[][] in bottom up manner
    for (int i=0; i<=m; i++)
    {
        for (int j=0; j<=n; j++)
        {
            // If first string is empty, only option is to
            // insert all characters of second string
            if (i==0)
                dp[i][j] = j; // Min. operations = j

            // If second string is empty, only option is to
            // remove all characters of second string
            else if (j==0)
                dp[i][j] = i; // Min. operations = i

            // If last characters are same, ignore last char
            // and recur for remaining string
            else if (str1[i-1] == str2[j-1])
                dp[i][j] = dp[i-1][j-1];

            // If the last character is different, consider all
            // possibilities and find the minimum
            else
                dp[i][j] = 1 + min(dp[i][j-1], // Insert
                                   dp[i-1][j], // Remove
                                   dp[i-1][j-1]); // Replace
        }
    }
    return dp[m][n];
}

// Driver program
int main()
{
    // your code goes here
    string str1 = "sunday";
    string str2 = "saturday";

    cout << editDistDP(str1, str2, str1.length(), str2.length());

    return 0;
}
```

Min Cost Path | DP-6

Given a cost matrix cost[][] and a position (m, n) in cost[][], write a function that returns cost of minimum cost path to reach (m, n) from (0, 0)

```
/* Dynamic Programming implementation of MCP problem */
#include<stdio.h>
#include<limits.h>
#define R 3
#define C 3
int min(int x, int y, int z);
int minCost(int cost[R][C], int m, int n)
{
    int i, j;
// Instead of following line, we can use int tc[m+1][n+1] or
// dynamically allocate memory to save space. The following line is
// used to keep the program simple and make it working on all
compilers.
    int tc[R][C];
```

```
    tc[0][0] = cost[0][0];
    /* Initialize first column of total cost(tc) array */
    for (i = 1; i <= m; i++)
        tc[i][0] = tc[i-1][0] + cost[i][0];
    /* Initialize first row of tc array */
    for (j = 1; j <= n; j++)
        tc[0][j] = tc[0][j-1] + cost[0][j];

    /* Construct rest of the tc array */
    for (i = 1; i <= m; i++)
        for (j = 1; j <= n; j++)
            tc[i][j] = min(tc[i-1][j-1],
                           tc[i-1][j],
                           tc[i][j-1]) + cost[i][j];

    return tc[m][n];
}
/* A utility function that returns minimum of 3 integers */
int min(int x, int y, int z)
{
if (x < y)
    return (x < z)? x : z;
else
    return (y < z)? y : z;
}
```

```
/* Driver program to test above functions */
int main()
{
int cost[R][C] = { {1, 2, 3},
                   {4, 8, 2},
                   {1, 5, 3} };
printf(" %d ", minCost(cost, 2, 2));
return 0;
}
Time Complexity of the DP implementation is O(mn)
```

Longest Palindromic Subsequence | DP-12

```
// A utility function to get max of two integers
int max (int x, int y) { return (x > y)? x : y; }
// Returns the length of the longest palindromic subsequence in seq
int lps(char *str) {
int n = strlen(str);
int i, j, cl;
int L[n][n]; // Create a table to store results of subproblems

for (i = 0; i < n; i++)
    L[i][i] = 1;
    for (cl=2; cl<=n; cl++)
    {
        for (i=0; i<n-cl+1; i++)
```

```
        {
            j = i+cl-1;
            if (str[i] == str[j] && cl == 2)
            L[i][j] = 2;
            else if (str[i] == str[j])
            L[i][j] = L[i+1][j-1] + 2;
            else
            L[i][j] = max(L[i][j-1], L[i+1][j]);
        }
    }
    return L[0][n-1];
}
/* Driver program to test above functions */
int main()
{
    char seq[] = "GEEKS FOR GEEKS";
    int n = strlen(seq);
    printf ("The lnegth of the LPS is %d", lps(seq));
    getchar();
    return 0;
}
```

Maximum Sum Increasing Subsequence | DP-14

```
/* Dynamic Programming implementation
of Maximum Sum Increasing Subsequence
(MSIS) problem */
#include <bits/stdc++.h>
using namespace std;


/* maxSumIS() returns the maximum
sum of increasing subsequence
in arr[] of size n */
int maxSumIS(int arr[], int n)
{
    int i, j, max = 0;
    int msis[n];

    /* Initialize msis values
    for all indexes */
    for ( i = 0; i < n; i++ )
        msis[i] = arr[i];

    /* Compute maximum sum values
    in bottom up manner */
    for ( i = 1; i < n; i++ )
        for ( j = 0; j < i; j++ )
            if (arr[i] > arr[j] &&
                msis[i] < msis[j] + arr[i])
                msis[i] = msis[j] + arr[i];

    /* Pick maximum of
    all msis values */
    for ( i = 0; i < n; i++ )
        if ( max < msis[i] )
            max = msis[i];

    return max;
}
```

```
// Driver Code
int main()
{
    int arr[] = {1, 101, 2, 3, 100, 4, 5};
    int n = sizeof(arr)/sizeof(arr[0]);
    cout << "Sum of maximum sum increasing "
            "subsequence is " << maxSumIS( arr, n ) << endl;
    return 0;
}
time complexity O(N*N)
#include <stdio.h>

void multiply(int F[2][2], int M[2][2]);

void power(int F[2][2], int n);

/* function that returns nth Fibonacci number */
int fib(int n)
{
int F[2][2] = {{1,1},{1,0}};
if (n == 0)
    return 0;
power(F, n-1);
return F[0][0];
}

/* Optimized version of power() in method 4 */
void power(int F[2][2], int n)
{
if( n == 0 || n == 1)
    return;
int M[2][2] = {{1,1},{1,0}};

power(F, n/2);
multiply(F, F);
if (n%2 != 0)
    multiply(F, M);
}

void multiply(int F[2][2], int M[2][2])
{
int x = F[0][0]*M[0][0] + F[0][1]*M[1][0];
int y = F[0][0]*M[0][1] + F[0][1]*M[1][1];
int z = F[1][0]*M[0][0] + F[1][1]*M[1][0];
int w = F[1][0]*M[0][1] + F[1][1]*M[1][1];

F[0][0] = x;
F[0][1] = y;
F[1][0] = z;
F[1][1] = w;
}

/* Driver program to test above function */
int main()
{
int n = 9;
printf("%d", fib(9));
getchar();
return 0;
}
```

Minimum number of jumps to reach end

```
// CPP program to find Minimum
// number of jumps to reach end
#include<bits/stdc++.h>
using namespace std;


// Returns Minimum number of
// jumps to reach end
```

```
int minJumps(int arr[], int n)
{
    // jumps[0] will hold the result
    int *jumps = new int[n];
    int min;

    // Minimum number of jumps needed
    // to reach last element from last
    // elements itself is always 0
    jumps[n-1] = 0;


    // Start from the second element,
    // move from right to left and
    // construct the jumps[] array where
    // jumps[i] represents minimum number
    // of jumps needed to reach
    // arr[m-1] from arr[i]
    for (int i = n-2; i >=0; i--)
    {
        // If arr[i] is 0 then arr[n-1]
        // can't be reached from here
        if (arr[i] == 0)
            jumps[i] = INT_MAX;

        // If we can direcly reach to
        // the end point from here then
        // jumps[i] is 1
        else if (arr[i] >= n - i - 1)
            jumps[i] = 1;

        // Otherwise, to find out the minimum
        // number of jumps needed to reach
        // arr[n-1], check all the points
        // reachable from here and jumps[]
        // value for those points
        else
        {
            // initialize min value
            min = INT_MAX;

            // following loop checks with all
            // reachable points and takes
            // the minimum
            for (int j = i + 1; j < n && j <=
                           arr[i] + i; j++)
            {
                if (min > jumps[j])
                    min = jumps[j];
            }
            // Handle overflow
            if (min != INT_MAX)
            jumps[i] = min + 1;
            else
            jumps[i] = min; // or INT_MAX
        }
    }
    return jumps[0];
}

// Driver program to test above function
int main()
{
    int arr[] = {1, 3, 6, 1, 0, 9};
    int size = sizeof(arr)/sizeof(int);
```

```
    cout << "Minimum number of jumps to reach"
        << " end is " << minJumps(arr, size);
    return 0;
}
```

time complexit O(N*N)

Ugly Numbers

Ugly numbers are numbers whose only prime factors are 2, 3 or 5. The sequence 1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, … shows the first 11 ugly numbers. By convention, 1 is included.

```
// C++ program to find n'th Ugly number
# include<bits/stdc++.h>
using namespace std;


/* Function to get the nth ugly number*/
unsigned getNthUglyNo(unsigned n)
{
    unsigned ugly[n]; // To store ugly numbers
    unsigned i2 = 0, i3 = 0, i5 = 0;
    unsigned next_multiple_of_2 = 2;
    unsigned next_multiple_of_3 = 3;
    unsigned next_multiple_of_5 = 5;
    unsigned next_ugly_no = 1;

    ugly[0] = 1;
    for (int i=1; i<n; i++)
    {
    next_ugly_no = min(next_multiple_of_2,
                       min(next_multiple_of_3,
                           next_multiple_of_5));
    ugly[i] = next_ugly_no;
    if (next_ugly_no == next_multiple_of_2)
    {
        i2 = i2+1;
        next_multiple_of_2 = ugly[i2]*2;
    }
    if (next_ugly_no == next_multiple_of_3)
    {
        i3 = i3+1;
        next_multiple_of_3 = ugly[i3]*3;
    }
    if (next_ugly_no == next_multiple_of_5)
    {
        i5 = i5+1;
        next_multiple_of_5 = ugly[i5]*5;
    }
    } /*End of for loop (i=1; i<n; i++) */
    return next_ugly_no;
}
/* Driver program to test above functions */
int main()
{
    int n = 150;
    cout << getNthUglyNo(n);
    return 0;
}
```

chapi, [25.10.19 13:27]

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

// Maximum number of digits in output
// x^n where 1 <= x, n <= 10000 and overflow may happen
#define MAX 100000

// This function multiplies x
// with the number represented by res[].
// res_size is size of res[] or
// number of digits in the number
```

```
// represented by res[]. This function
// uses simple school mathematics
// for multiplication.
// This function may value of res_size
// and returns the new value of res_size
int multiply(int x, int res[], int res_size)
{

  // Initialize carry
  int carry = 0;

  // One by one multiply n with
  // individual digits of res[]
  for (int i = 0; i < res_size; i++)
  {
    int prod = res[i] * x + carry;

    // Store last digit of
    // 'prod' in res[]
    res[i] = prod % 10;

    // Put rest in carry
    carry = prod / 10;
  }

  // Put carry in res and
  // increase result size
  while (carry)
  {
    res[res_size] = carry % 10;
    carry = carry / 10;
    res_size++;
  }
  return res_size;
}

// This function finds
// power of a number x
void power(int x, int n)
{
  //printing value "1" for power = 0
  if (n == 0)
  {
    cout << "1";
    return;
  }

  int res[MAX];
  int res_size = 0;
  int temp = x;

  // Initialize result
  while (temp != 0)
  {
    res[res_size++] = temp % 10;
    temp = temp / 10;
  }

  // Multiply x n times
  // (x^n = x*x*x....n times)
  for (int i = 2; i <= n; i++)
    res_size = multiply(x, res, res_size);

  cout << x << "^" << n << " = ";
  for (int i = res_size - 1; i >= 0; i--)
    cout << res[i];
}

// Driver program
int main()
{
  int exponent, base;
  printf("Enter base ");
  scanf("%id \n", &base);
  printf("Enter exponent ");
  scanf("%id", &exponent);
  power(base, exponent);
  return 0;
}
```

```
//

//

catalan_dict = {0: 1, 1: 1}


def catalan(n):

    if catalan_dict.contains(n):

        return catalan_dict[n]

    if n <= 500:

        catalan_dict[n] = ((n << 2) - 2) * catalan(n-1) // (n + 1)

    else:

        catalan(n-500)

        catalan_dict[n] = ((n << 2) - 2) * catalan(n-1) // (n + 1)

    return catalan_dict[n]


for _ in range(int(input())):

    print(catalan(int(input())))
```

Python Pow implementation

```
 unsigned long pow_mod(unsigned short x, unsigned long y, unsigned
short z)

{
    unsigned long number = 1;
    while (y)
    {
        if (y & 1)
            number = number * x % z;
        y >>= 1;
        x = (unsigned long)x * x % z;
    }
    return number;
}

int main()
{
    printf("%d\n", pow_mod(63437, 3935969939, 20628));
    return 0;
}
```

a x b = LCM(a, b) * GCD (a, b)

```
// C++ program of finding nth palindrome

// of k digit

#include<bits/stdc++.h>

using namespace std;


void nthPalindrome(int n, int k)

{

  // Determine the first half digits

  int temp = (k & 1) ? (k / 2) : (k/2 - 1);

  int palindrome = (int)pow(10, temp);

  palindrome += n - 1;


  // Print the first half digits of palindrome

  printf("%d", palindrome);


  // If k is odd, truncate the last digit

  if (k & 1)

    palindrome /= 10;


  // print the last half digits of palindrome
```

```
  while (palindrome)
  {
    printf("%d", palindrome % 10);
    palindrome /= 10;
  }
  printf("\n");
}

// Driver code
int main()
{
  int n = 6, k = 5;
  printf("%dth palindrome of %d digit = ",n ,k);
  nthPalindrome(n ,k);

  n = 10, k = 6;
  printf("%dth palindrome of %d digit = ",n ,k);
  nthPalindrome(n, k);
  return 0;
}
```

a x b = LCM(a, b) * GCD (a, b)

```
// combination c(n,k)
int binomialCoeff(int n, int k)
{
  int res = 1;

  // Since C(n, k) = C(n, n-k)
  if ( k > n - k )
    k = n - k;

  // Calculate value of
  // [n * (n-1) *---* (n-k+1)] / [k * (k-1) *----* 1]
  for (int i = 0; i < k; ++i)
  {
    res *= (n - i);
    res /= (i + 1);
  }

  return res; }
void nthPalindrome(int n, int k)
{
  // Determine the first half digits
  int temp = (k & 1) ? (k / 2) : (k/2 - 1);
  int palindrome = (int)pow(10, temp);
  palindrome += n - 1;

  // Print the first half digits of palindrome
  printf("%d", palindrome);

  // If k is odd, truncate the last digit
```

```
  if (k & 1)
    palindrome /= 10;

  // print the last half digits of palindrome
  while (palindrome)
  {
    printf("%d", palindrome % 10);
    palindrome /= 10;
  }
  printf("\n");
}
//check if it is power of 2 or not
    int arr[N];
    bool isPowerOfTwo (ll x)
    {
        return x && (!(x&(x-1)));
    }
```

Topological Sort

```
def dfs(graph,start):
    path = []
    stack = [start]
    while stack != []:
        v = stack.pop()
        if v not in path:
            path.append(v)
        for w in reversed(graph[v]):
            if w not in path and not w in stack:
                stack.append(w)
    return path
```

Graph bipartite check

```
def check_bipartite_dfs(l):
      visited = [False] * len(l)
      color = [-1] * len(l)
      def dfs(v, c):
        visited[v] = True
        color[v] = c
        for u in l[v]:
         if not visited[u]:
          dfs(u, 1 - c)
      for i in range(len(l)):
        if not visited[i]:
          dfs(i,0)
      for i in range(len(l)):
        for j in l[i]:
          if color[i] == color[j]:
            return False
      return True
```

DFS

```
from collections import defaultdict
# adjacency list representation
class Graph:
    def __init__(self):
        self.graph = defaultdict(list)
    def addEdge(self, u, v):
        self.graph[u].append(v)
    def DFSUtil(self, v, visited):
        visited[v] = True
        print(v, end = ' ')
        for i in self.graph[v]:
           if visited[i] == False:
                  self.DFSUtil(i, visited)
    def DFS(self, v):
        visited = [False] *
(len(self.graph))
        self.DFSUtil(v, visited)
g = Graph()
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)

##DFS starting from vertex 2
g.DFS(2)
```

I LOVE OBSTACLE

Union

```
# A function that does union of two
# nodes x and y where xr is root node
# of x and yr is root node of y
def _union(xr,yr):
   global Arr, size
   if (size[xr] < size[yr]):
   # Make yr parent of xr
     Arr[xr] = Arr[yr]
     size[yr] += size[xr]
   else:
      # Make xr parent of yr
      Arr[yr] = Arr[xr]
      size[xr] += size[yr]
```

BFS

```
graph={ 0:[1,3,4], 1:[0,2,4], 2:[1,6], 3:[0,4,6], 4:
[0,1,3,5], 5:[4], 6:[2,3] }

def bfs(graph, start, path=[]):
  queue = [start]
  while queue:
     vertex = queue.pop(0)
     if vertex not in path:
         path.append(vertex)
         queue.extend(graph[vertex] - path)
  return path

print bfs(graph, 0)
```

DFS 2

```
graph ={'A':['B','S']...}
visited=[]
def dfs(graph,node):
   global visited
   if node not in visited:
      visited.append(node)
      for n in graph[node]:
         dfs(graph,n)
dfs(graph1,'A')
primt(visited)
```

UNION-FIND SET (O(log(n))) {c++}

```
''' use this to check for connectivity by
decreasing
the count every time union_set function is
called,
also use this mainly to check cycle when
using
kruskal algorithm for minimum spanning
tree(mst)'''
int parent[1005], _size[1005];
void make_set(int v)
{
parent[v] = v; _size[v] = 1;
}
int find_set(int i)
{
while(i != parent[i]) i = parent[i];
return i;
}
```

```
void union_set(int i, int j)
{
int a = find_set(i); int b = find_set(j);
if(_size[a] < _size[b]) swap(a, b);
parent[b] = a;
_size[a] += _size[b];
}
```

Bellman Ford

```
from collections import defaultdict
class Graph:
  def __init__(self, vertices):
    self.V = vertices
    self.graph = []
  def addEdge(self, u, v, w):
    self.graph.append([u, v, w])
  def printArr(self, dist):
    print("Vertex Distance from Source")
    for i in range(self.V):
        print("% d \t\t % d" % (i, dist[i]))
def BellmanFord(self, src):
  dist = [float("Inf")] * self.V
  dist[src] = 0
  for i in range(self.V - 1):
    for u, v, w in self.graph:
      if dist[u] != float("Inf") and
  dist[u] + w < dist[v]:
        dist[v] = dist[u] + w
    for u, v, w in self.graph:
      if dist[u] != float("Inf") and
      dist[u] + w < dist[v]:
           print('negative weight cycles')
           return
  self.printArr(dist)

g = Graph(2)
g.addEdge(0, 1, -1)
g.addEdge(0, 2, 4)
g.BellmanFord(0)
```

Floyed-warsahall

```
V = 4
def floydWarshall(graph):
  dist = map(lambda i : map(lambda j : j ,
  i) , graph)
  for k in range(V):
    for i in range(V):
      for j in range(V):
        dist[i][j] = min(dist[i][j],
        dist[i][k]+ dist[k][j])
  printSolution(dist)
def printSolution(dist):
  for i in range(V):
```

```
    for j in range(V):
      if(dist[i][j] == INF):
        print "%7s" %("INF"),
      else:
        print "%7d\t" %(dist[i][j]),
      if j == V-1:
        print ""
graph = [[0,5,INF,10],[INF,0,3,INF],[INF,INF,0,1],[INF, INF, INF,
0]]
floydWarshall(graph);
```

Dijkastra

```
def extract(Q, w):
 m=0
 minimum=w[0]
 for i in range(len(w)):
  if w[i]<minimum:
    minimum=w[i]
    m=i
 return m, Q[m]
def dijkstra(G, s, t='B'):#assuming the
                   start vertex to be ‘B’#
 Q=[s]
 p={s:None}
 w=[0]
 d={}
 for i in G:
   d[i]=float('inf')
   Q.append(i)
   w.append(d[i])
 d[s]=0
 S=[]
 n=len(Q)
 while Q:
   u=extract(Q,w)[1]
   S.append(u)
   #w.remove(extract(Q, d, w)[0])
   Q.remove(u)
   for v in G[u]:
     if d[v]>=d[u]+G[u][v]:
        d[v]=d[u]+G[u][v]
        p[v]=u
 return d, p
```

From Hacker Earth

Min-coast Max-flow

```
function: CostNetwork(Graph G, Graph Gf):

    Gc <- empty graph
    for i in edges E:
        if E(u,v) in G:
            cf(u,v) = c(u,v)
        else if E(u,v) in Gf:
            cf(u,v) = -c(u,v)
'''Find a feasible maximum flow of G using Ford Fulkerson and
construct residual'''

function: MinCost(Graph G):
    graph(Gf)
    Gc = CostNetwork(G, Gf)
```

```
    while(negativeCycle(Gc)):
'''Increase the flow along each edge in cycle C by minimum
capacity in the cycle C
        Update residual graph(Gf)'''
        Gc = CostNetwork(G,Gf)
    mincost = sum of Cij*Fij for each of the flow in residual
graph
    return mincost
```

GEEk MAX-FLOW MIN-COAST

```
C = 3
R = 3
def minCost(cost, m, n):
 tc=[[0 for x in range(C)]forx in range(R)]
 tc[0][0] = cost[0][0]
 for i in range(1, m + 1):
   tc[i][0] = tc[i-1][0] + cost[i][0]
 for j in range(1, n + 1):
   tc[0][j] = tc[0][j-1] + cost[0][j]
 for i in range(1, m + 1):
   for j in range(1, n + 1):
     tc[i][j] = min(tc[i-1][j-1], tc[i-1]
     [j],tc[i][j-1]) + cost[i][j]
 return tc[m][n]
```

prime factorization

```
def prime_factors(n):
    i = 2
    factors = []
    while i * i <= n:
        if n % i:
            i += 1
        else:
            n //= i
            factors.append(i)
    if n > 1:
        factors.append(n)
    return factors
```

DAY CALCULATOR

```
def day(d,m,y):

t = [0,3,2,5,0,3,5,1,4,6,2,4]

    y-= m < 3

    return (((y+int(y / 4)) - int(y / 100) + int(y/400) + t[m - 1]
+ d) % 7)

day = day(30,8,2010)

print(day)
```