**DFS**

#include <bits/stdtr1c++.h>

using namespace std;

int N = 100; // number of nodes

// all the u, v, a, b in this program are 0-indexed, so it ranges from 0 to N-1

// adjacency list

// there is a directed edge from u to v iff adj[u] contains v.

// if the graph is undirected, adj[u] contains v iff adj[v] contains u

vector<int> adjlist[N];

// visited array

// vis[u] true iff the node u have already been visited

bool vis[N];

int main() {

// enter the edges into the adjacency list

// example for directed edge from u to v

adjlist[u].push\_back(v);

// example for undirected edge between u and v

adjlist[u].push\_back(v);

adjlist[v].push\_back(u);

// start the dfs at node a, and find b

// in other words, check if b is reachable from a

bool result = dfs\_iterative(a, b);

// dfs\_recursive(a, b) does the same thing but slower and prone to stack overflow on large graphs

return 0;

}

bool dfs\_iterative(int a, int b) {

memset(vis, 0, sizeof vis);

stack<int> s;

s.push(a);

while (!s.empty()) {

int cur = s.top();

s.pop();

if (vis[cur]) continue;

vis[cur] = true;

/\*\*\* this section can be adapted depending on the purpose of this dfs \*\*\*/

if (cur == b) return true;

/\*\*\*\*\*\*\*\*\*\*\*\*\*/

for (int i : adjlist[cur]) {

s.push(i);

}

}

return false;

}

bool dfs\_recursive(int a, int b) {

/\*\*\* this section can be adapted depending on the purpose of this dfs \*\*\*/

if (a == b) return true;

/\*\*\*\*\*\*\*\*\*\*\*\*\*/

if (vis[a]) return false;

vis[a] = true;

for (int i : adjlist[a]) {

if (dfs\_recursive(i, b)) return true;

}

return false;

}

**BFS**

#include <iostream>

using namespace std;

const int N = 10000;

typedef pair<int, int> pii;

// note: bfs only works when all edge length are equal

vector<int> adj[N]; // there is an edge from node a to node b if adj[a] contains b

bool vis[N]; // visited array

int dist[N]; // result array

void bfs(int a) {

queue<pii> q; // a queue of (location, cost)

// for dijkstra: priority\_queue<pii> q; // (-cost, location)

// ... or for dijkstra: priority\_queue<pii, vector<pii>, greater<pii> > q; // (cost, location)

q.push(pii(a, 0));

while (!q.empty()) {

// for dijkstra: change front to top (std::priority\_queue)

int at = q.front().first;

int cost = q.front().second;

q.pop();

if (vis[at]) continue;

vis[at] = true;

// flexible

dist[at] = cost;

for (int i : adj[at]) {

q.push(pii(i, /\* flexible \*/cost + 1));

//for dijkstra: change cost+1 to cost + the edge cost from at to i

}

}

}

/\*vector<pii> adj[N]; // there is an edge from node a to node b if adj[a] contains b

void dijkstra(int a) {

priority\_queue<pii> q;

// (-cost, location)

q.emplace(0, a);

while (!q.empty()) {

int at = q.top().second;

int cost = -q.top().first;

q.pop();

if (vis[at]) continue;

vis[at] = true;

dist[at] = cost;

// add all adjacent nodes to queue

for (i : adj[at]) {

q.emplace(-(cost + i.first), i.second);

}

}

}\*/

int main() {

ios::sync\_with\_stdio(0);

cin.tie();

/\* // add edges

// ex. directed edge from x to y

adj[x].push\_back(y);

// ex undirected edge between x and y

adj[x].push\_back(y);

adj[y].push\_back(x);\*/

int t; cin >> t;

while (t--) {

int start, end; cin >> start >> end;

adj[start].push\_back(end);

}

// find shortest path from node a to all other nodes

memset(dist, -1, sizeof dist);

memset(vis, 0, sizeof vis);

// specify starting location

int a = 0;

bfs(a);

// distance from node a to node c is now dist[c], or dist[c] == -1 if c is unreachable from a

}

**Find number of permutations of string:**

#include <iostream>

#include <vector>

#include <string>

#include <algorithm>

using namespace std;

int main(){

string s; cin >> s;

vector<string> v;

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

do {

v.push\_back(s);

} while (next\_permutation(s.begin(), s.end()));

unique(v.begin(),v.end());

int size = v.size();

cout << size << endl;

return 0;

}

**Find permutations of array:**

#include <iostream> // std::cout

#include <algorithm> // std::next\_permutation, std::sort

int main () {

int myints[] = {1,2,3};

std::sort (myints,myints+3);

std::cout << "The 3! possible permutations with 3 elements:\n";

do {

std::cout << myints[0] << ' ' << myints[1] << ' ' << myints[2] << '\n';

} while ( std::next\_permutation(myints,myints+3) );

std::cout << "After loop: " << myints[0] << ' ' << myints[1] << ' ' << myints[2] << '\n';

}

**Sample DP:**

#include <iostream>

using namespace std;

int A[1005];

int totalValue, d1, d2, d3;

int mincoins(int n) {

if (n == 0) return 0;

if (n < 0) return totalValue+1;

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

return A[n] = min(min(mincoins(n - d1), mincoins(n - d2)), mincoins(n - d3)) + 1;

}

int main() {

memset(A,-1,sizeof A);

d1 = 1;

d2 = 3;

d3 = 7;

totalValue = 1000;

cout << mincoins(totalValue) << endl;

return 0;

}

|  |  |
| --- | --- |
| // lexicographical\_compare example  #include <iostream> // std::cout, std::boolalpha  #include <algorithm> // std::lexicographical\_compare  #include <cctype> // std::tolower  // a case-insensitive comparison function:  bool mycomp (char c1, char c2)  { return std::tolower(c1)<std::tolower(c2); }  int main () {  char foo[]="Apple";  char bar[]="apartment";  std::cout << std::boolalpha;  std::cout << "Comparing foo and bar lexicographically (foo<bar):\n";  std::cout << "Using default comparison (operator<): ";  std::cout << std::lexicographical\_compare(foo,foo+5,bar,bar+9);  std::cout << '\n';  std::cout << "Using mycomp as comparison object: ";  std::cout << std::lexicographical\_compare(foo,foo+5,bar,bar+9,mycomp);  std::cout << '\n';  return 0;  } |  |

**Binary search:**

int binarySearch(int arr[], int value, int left, int right) {

      while (left <= right) {

            int middle = (left + right) / 2;

            if (arr[middle] == value)

                  return middle;

            else if (arr[middle] > value)

                  right = middle - 1;

            else

                  left = middle + 1;

      }

      return -1;

}

// OR:

std::binary\_search (v.begin(), v.end(), 3) // returns boolean

**Ternary search:**

#include <iostream>

using namespace std;

#define s 12 //I gave space in the array as 12,but you can give the number you want.

int ternary\_search (int v[],int n, int left, int right, int x);

int main()

{

int v[s];

short x;

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

{ v[i-1] = i; }

cout << "Enter number for research:\n";

cin >> x;

int left = s/3;

int right = (s/3)\*2;

if(ternary\_search(v,s,left-1,right-1,x) == -1)

{cout<<"Number does not exist in array.\n";}

else

{cout<<"The index is:"<<ternary\_search(v,s,left-1,right-1,x)<<"\n";}

return 0;

}

int ternary\_search (int v[],int n, int left, int right, int x)

{

if(left < 0 || right > n-1 || left > right) return -1;

if(x == v[left]) return left;

if(x == v[right]) return right;

// Update the two index left and right if the element is not found.

if(x < v[left]) return ternary\_search(v,n,left-1,right,x);

if(x > v[left] && x < v[right]) return ternary\_search(v,n,left+1,right-1,x);

if(x > v[right]) return ternary\_search(v,n,left,right+1,x);

}