East West University

Department of Computer Science and Engineering
CSE246: Mini Project
Course Instructor: SDIS
ID: 2021-3-60-030 Name: Md. Rakibul Islam

(01)Problem id 03: Graph coloring

Solution:

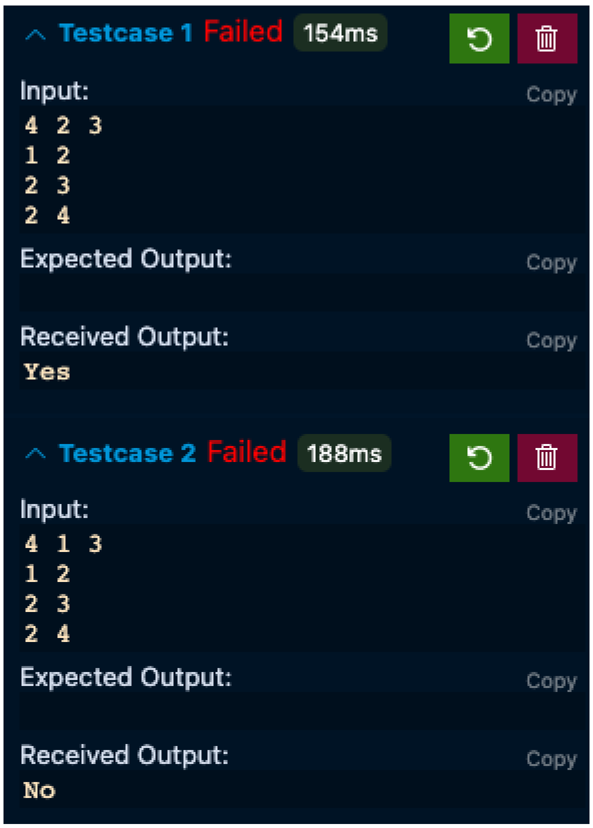
TAG: DFS, Greedy
COMPLEXITY: Approximately O(elog(n)), here e is the number of edges.
Solution:
Whenever we are visiting a node we are checking its adjacent node that is already visited and find out the minimum number k that is not assigned yet. If that minimum number k is greater than the given number "M '' then it's not possible to color the graph with at most M color.
To run a dfs it will cost o(e) and finding the smallest k it will cost log(n) complexity. So total complexity is O(elog(n))

SOURCE CODE:

```
#include <bits/stdc++.h>
using namespace std;

int main(){
   int n, m, e;   cin>>n>>m>>e;
   vector<vector<int>> adj(n+1);
   for(int i=1; i<=e; i++){
      int u, v;   cin>>u>>v;
      adj[u].push_back(v);
      adj[v].push_back(u);
   }
   vector<int> vis(n+1, 0), clr(n+1, -1);
   auto dfs = [&](auto self, int u)->void{
      vis[u] = 1;
      int curClr = 1;
      set<int> s;
      for(auto v:adj[u])if(vis[v]){
         s.insert(clr[v]);
      }
      while(s.count(curClr))curClr++;
      clr[u] = curClr;
```

```
        for(auto v:adj[u])if(!vis[v]){
            self(self, v);
        }
    };
    dfs(dfs, 1);
    int mx = *max_element(clr.begin(), clr.end());
    cout<<(mx > m ? "No\n" : "Yes\n");
    return 0;
}
```

Output:



(02) Problem 14: SCC

Solution:

TAG: DFS, TOPOLOGICAL SORT

Complexity: O(E) here E is the number of edges.

Solution:

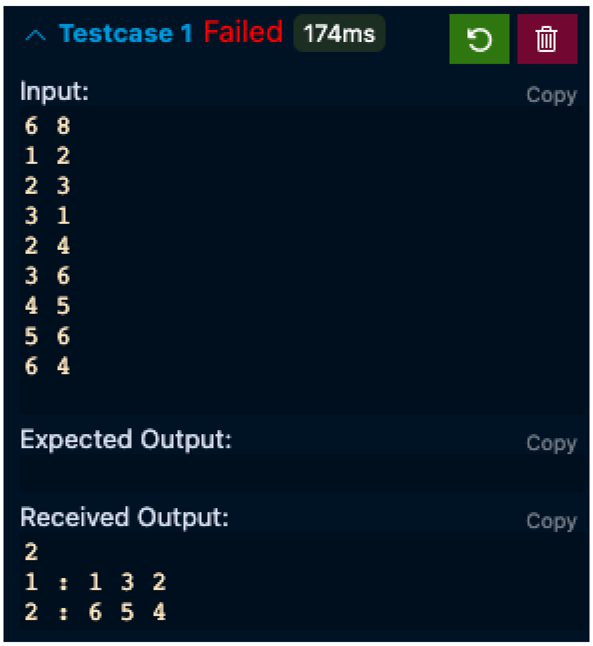
In order to find Strongly connected components in a graph we need two sets of given graphs one is the real one and another one is the transpose of the real one. Then we will run a dfs at the real input graph. Then we will get an order of visited nodes. After that we will run another dfs on a transpose graph with the reverse order of visited nodes. And the number of connected components will be strongly connected.

SOURCE CODE:

```
#include <bits/stdc++.h>
using namespace std;

int main(){
   int n, e;   cin>>n>>e;
   vector<vector<int>> adj(n+1), tadj(n+1);
   for(int i=1; i<=e; i++){
      int u, v;   cin>>u>>v;
      adj[u].push_back(v);
      tadj[v].push_back(u);
   }

   vector<int> vis(n+1, 0);
   vector<int> ord;
   auto dfs = [&](auto self, int u)->void{
      vis[u] = 1;
      for(auto v:adj[u])if(!vis[v]){
         self(self, v);
      }
      ord.push_back(u);
   };

   for(int i=1; i<=n; i++)if(!vis[i]){
      dfs(dfs, i);
   }
   vis.assign(n+1, 0);

   vector<int> scc;
   auto Dfs = [&](auto self, int u)->void{
      vis[u] = 1;
      scc.push_back(u);
      for(auto v:tadj[u])if(!vis[v]){
         self(self, v);
      }
   };
   vector<vector<int>> ans;
   reverse(ord.begin(), ord.end());
   for(auto v:ord)if(!vis[v]){
      scc.clear();
```

```
      Dfs(Dfs, v);
      ans.push_back(scc);
   }
   cout<<ans.size()<<"\n";
   for(int i=0; i<ans.size(); i++){
      cout<<i+1<<" : ";
      for(auto v:ans[i]){
         cout<<v<<" ";
      }
      cout<<"\n";
   }
   return 0;
}
```

OUTPUT:



(03) Problem ID 12 : Maximum Sum Interval

Solution:

TAG: GREEDY

Complexity: O(n) or linear.

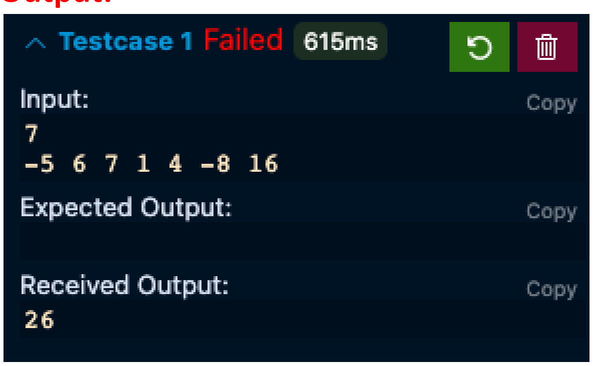
Solution:

This is a pretty straightforward solution. First of all we will have a sum variable and we will calculate the sum of adjacent elements whenever we get a negative sum we will update our sum variable with 0 because if we don't get a positive sum we will not take any elements from the given array. And this algorithm is called KADANE'S algorithm.

Source Code:

```
#include <bits/stdc++.h>
using namespace std;

int main(){
    int n;  cin>>n;
    vector<int> a(n);
    for(int i=0; i<n; i++){
        cin>>a[i];
    }

    long long cur = 0, ans = 0;
    for(int i=0; i<n; i++){
        cur += a[i];
        cur = max(cur, 0ll);
        ans = max(ans, cur);
    }

    cout<<ans<<"\n";
    return 0;
}
```

Output:



(04) Problem Id 13 : MCM

Solution:

TAG: Dynamic Programming, Implementation

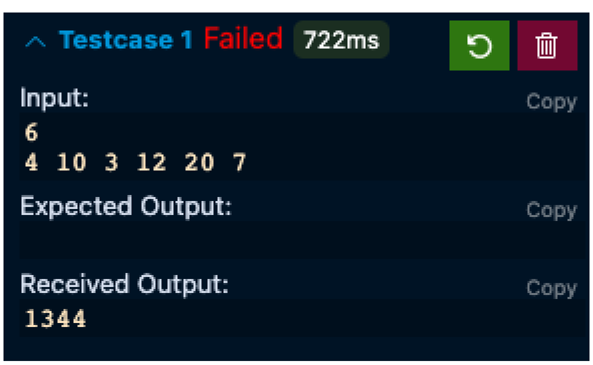
COMPLEXITY: O(N^2)

SOLUTION:

In this problem we will use a dynamic programming approach. First of all, what is the cost of multiplying two matrices? The cost of two matrices if [a x b] [b x c] is two different matrix then cost of them will be (a*c). By putting that observation in our head we are gonna model a DP solution. There will be two states. DP[I][J] will tell us the minimum number of costs we need if we want to multiply the matrices started from i to j.

```
#include <bits/stdc++.h>
using namespace std;

int main(){
   int n;  cin>>n;
   vector<int> mat(n);
   for(int i=0; i<n; i++){
      cin>>mat[i];
   }

   vector dp(n+1, vector<int>(n+1, -1));
   auto DP = [&](auto self, int i, int j)->int{
      if(i == j)return 0;
      if(dp[i][j] != -1)return dp[i][j];
      dp[i][j] = 1000000000;
      for(int k = i; k<j; k++){
         dp[i][j] = min(dp[i][j], self(self, i, k) + self(self, k+1, j) + mat[i-1] * mat[k] * mat[j]);
      }
      return dp[i][j];
   };
   cout<<DP(DP, 1, n-1)<<"\n";
   return 0;
}
```

Output:



(05) Problem id 14: Articulation Point And Bridges

Solution:

TAG: DFS, IMPLEMENTATION

COMPLEXITY: O(E+N) here E is the number of edges and N is the number of nodes.

SOLUTION:

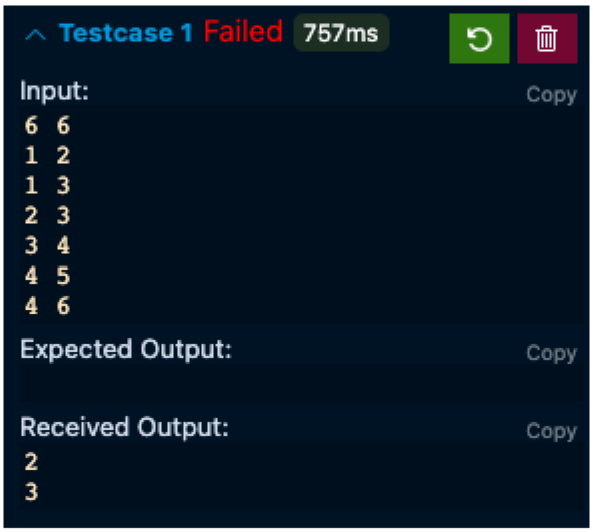
In this problem we are not given a source node. But we are gonna assume node 1 is a source node here. By taking 1 as a source node we are gonna run a dfs from node 1 now if we are going to visit some node y from some node x. Then if the node y is not previously and its depth is greater than the node x. Then we can say node x is an articulation point. Because any child of x is not connected with any child of y.

Source Code:

```
#include<bits/stdc++.h>
using namespace std;

const int inf = 1e9;
int main(){
  int n, e;   cin>>n>>e;
  vector<vector<int>> adj(n+1);
  for(int i=1; i<=e; i++){
    int u, v;   cin>>u>>v;
    adj[u].push_back(v);
    adj[v].push_back(u);
  }

  vector<int> disc(n+1, 0), ap(n+1, 0), low(n+1, inf);
  vector<array<int, 2>> br;
  int Time = 0;
  auto dfs = [&](auto self, int u, int p)->int{
   int children = 0;
   low[u] = disc[u] = ++Time;
   for (auto v : adj[u]) {
    if (v == p) continue;
    if (!disc[v]) {
     children++;
     self(self, v, u);
     if(disc[u] < low[v])br.push_back({u, v});
     if (disc[u] <= low[v])
      ap[u] = 1;
      low[u] = min(low[u], low[v]);
    } else
     low[u] = min(low[u], disc[v]);
   }
   return children;
  };

  auto AP = [&]() {
```

```
    Time = 0;
    for (int u = 1; u <=n; u++)
      if (!disc[u])
        ap[u] = dfs(dfs, u, u) > 1;
  };
  AP();
  int cnt = accumulate(ap.begin(), ap.end(), 0ll);
  cout<<cnt<<"\n";
  cout<<br.size()<<"\n";
  return 0;
}
```

Output:



(06) Problem id 16: LCS(Longest Common Subsequences)

Solution:

TAG: Dynamic Programming, Implementation

COMPLEXITY: O(N*M) Where N is the length of the first string and M is the second string.

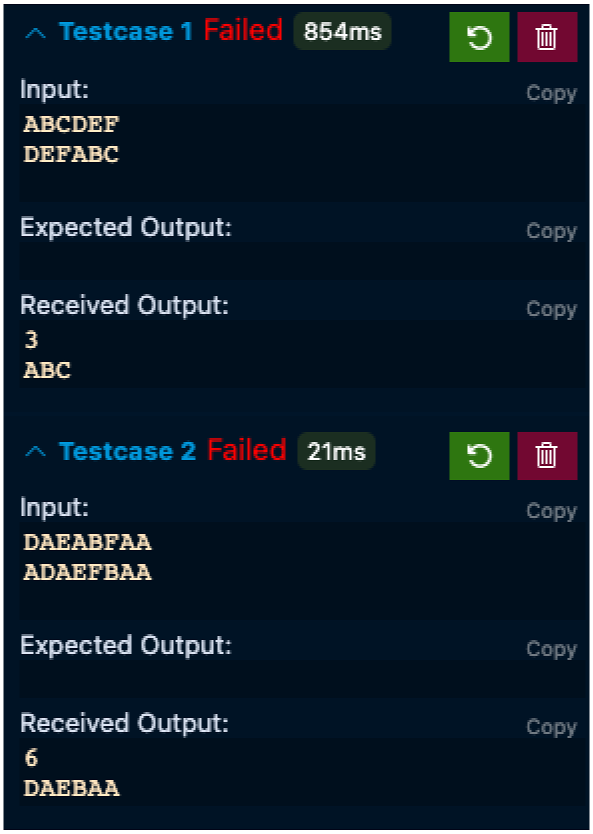
SOLUTION:

We are gonna use a tabulation form of dynamic programming approach.It's again a dynamic programming solution, so what is the state of our approach? The state is dp[i][j]. Where dp[i][j] will tell us from 0 to i of the first string and 0 to j of second string maximum length of common subsequences.For that such operation complexity will be O(N*M).

SOURCE CODE:

```
#include<bits/stdc++.h>
using namespace std;

int main() {
	string a, b;	cin>>a>>b;
	int n = a.size(), m = b.size();
	a = ' ' + a;
	b = ' ' + b;
	vector<vector<int>> dp(n+1, vector<int>(m+1));
	for (int i = 1; i <= n; i++) {
		for (int j = 1; j <= m; j++) {
			dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
			if (a[i] == b[j])dp[i][j] = max(dp[i][j], dp[i - 1][j - 1] + 1);
		}
	}

	int x = n, y = m;
	int c = dp[x][y];
	cout<<c<<"\n";
	string ans = "";
	while (x && y) {
		if (a[x] == b[y]) {
			ans += a[x];
			x--, y--;
		}
		else if (dp[x][y] == dp[x - 1][y])x--;
		else y--;
	}
	reverse(ans.begin(), ans.end());
   cout<<ans<<"\n";
}
```

Output:



(07) Problem id 06: Optimal Binary search tree

Solution:

TAG: DYNAMIC PROGRAMMING, IMPLEMENTATION

COMPLEXITY: O(N^2), where N is the length of the given input.

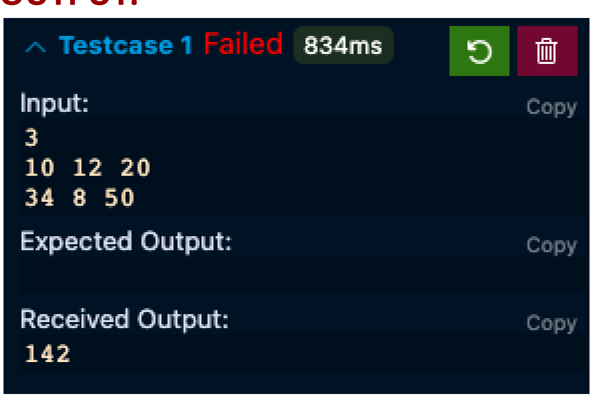
SOLUTION: First of all, what is a binary tree ? Basically every node of a tree has two children, that is why it's called a binary tree. Then in the binary search tree there is a searching cost of an element. And that cost depends on the level of that node.If a element in a root node then it's searching cost is 0, if it's were level 1 node then the cost would be 1.In the given problem you have some element and the search frequency (number of times you need to search that element). And by that priority we are gonna build a binary search tree. And there we are gonna use a dynamic programming approach where dp[i][j] will tell us the minimum cost of searching the element from i to j.

SOURCE CODE:

```
#include<bits/stdc++.h>
using namespace std;

int main() {
```

```
    int n;  cin>>n;
            vector<int> key(n), freq(n);
    for(int i=0; i<n; i++){
        cin>>key[i];
    }

    for(int i=0; i<n; i++){
        cin>>freq[i];
    }
    vector dp(n+1, vector<int>(n+1));

    auto cal = [&](auto self, int i, int j){
        if(dp[i][j])return dp[i][j];
        int sum = 0;
        for(int k=i; k<=j; k++){
            sum += freq[k];
        }

        int mn = 1000000000;
        for (int r = i; r <= j; r++) {
            int c = self(self, i, r - 1) + self(self, r + 1, j) + sum;
            if (c < mn) {
                mn = c;
                dp[i][j] = c;
            }
        }

        return dp[i][j];
    };

    cout<<cal(cal, 0, n-1)<<"\n";
}
```

OUTPUT:



(08) Problem id(15) : Closest Pair of Points

Solution:

TAG: BRUTEFORCE

COMPLEXITY: O(N^2)

SOLUTION: First of all this solution is not an optimal solution. There is also a solution to this problem which is using the concept of divide and conquer algorithm. We should have used that one but I had faced some issues with that approach and then I decided to use bruteforce here.Where i am calculating the distance of every point to all other point and take the minimum answer.

SOURCE CODE:

```
#include<bits/stdc++.h>
using namespace std;


int main() {
   int n;  cin>>n;
   vector<array<int, 2>> a(n);
   for(int i=0; i<n; i++){
      cin>>a[i][0]>>a[i][1];
   }
   sort(a.begin(), a.end());

   auto dis = [&](int i, int j){
      int d = abs(a[i][0]-a[j][0]) * abs(a[i][0]-a[j][0]);
      d += abs(a[i][1]-a[j][1]) * abs(a[i][1]-a[j][1]);
      return int(sqrt(d));
   }

   int ans = 100000;
   for(int i=0; i<n; i++){
      for(int j=i+1; j<n; j++){
         ans = min(ans, dis(i, j));
      }
   }
   cout<<ans<<"\n";
   return 0;
}
```

Output:

(09) Problem ID(11) : Multiple Shortest Path

Solution:

TAG: DFS, SHORTEST PATH

COMPLEXITY: O(E log(v)) where E is the number of edges and v is the number of nodes.

SOLUTION:

In this problem we are given a graph and we need to find if that node has a multiple shortest path and we can do that easily with the dijkstra algorithm. So first of all we will run our algorithm from node 1 if any node got visited twice that means it has multiple shortest paths and we will print "yes" otherwise "no".

SOURCE CODE:

```
#include<bits/stdc++.h>
using namespace std;

int inf = 1e9;

int main(){
   int n, e;   cin>>n>>e;
   vector<vector<int>> adj(n+1);
   for(int i=1; i<=e; i++){
      int u, v;   cin>>u>>v;
      adj[u].push_back(v);
      adj[v].push_back(u);
   }
   int s;  cin>>s;

   priority_queue<array<int, 2>, vector<array<int, 2>>, greater<array<int, 2>>> pq;
   vector<int> dis(n+1, inf), mul(n+1, 0);
   dis[s] = 0;
   pq.push({0, s});
   while(!pq.empty()){
      auto top = pq.top();
      pq.pop();
      for(auto v:adj[top[1]]){
         if(dis[v] == dis[top[1]] + 1){
            mul[v] = 1;
         }
         if(dis[v] > dis[top[1]] + 1){
            dis[v] = dis[top[1]] + 1;
            pq.push({dis[top[1]], v});
            mul[v] |= mul[top[1]];
         }
      }
```

```
    }
    int mx = *max_element(mul.begin(), mul.end());
    cout<<(mx ? "YES\n" : "NO\n");
    return 0;
}
```

Output: