First Round SOI 2016

Solution Booklet

Swiss Olympiad in Informatics

January 9, 2016

1 Cable Car

Mouse Stofl has bought a small cable car model with N pillars. He puts them in a line such that two adjacent pillar have a distance of 1.

Stofl enjoys the view on a real cablecar only if there is everywhere the same slope. Otherwise the cablecar starts swinging and Stofl can't enjoy the view. Mouse Stofl wants, that the model cablecar, like the real cablecar, has the same slope at each point.

Subtask 1: Small Cable Car (10 points)

In the first task, you are given 3 integers, the heights of the pillars. You have to check, whether the slope is everywhere the same.

You can calculate the slope using the following formula: $\frac{\Delta y}{\Delta x}$. x is the distance between the pillars. y is the difference of height of the pillars. The distance between two adjacent pillars is always 1. Therefore, to calculate the slope, you only need to calculate the difference.

To check, whether the slope is everywhere the same, you only need to check whether the difference between the first and the second pillar is equal to the difference between the second and the third pillar.

This solution runs in $O(1)$.

```cpp
#include<bits/stdc++.h>

using namespace std;

int main(){
        int T;
        cin >> T;
        for(int t=1;t<=T;t++){// There are several testcases
                int N;
                cin >> N;//always 3
                int p1,p2,p3;
                cin >> p1 >> p2 >> p3;
                if(p1-p2==p2-p3){
                        cout << "Case #" << t << ": yes\n";
                } else{
                        cout << "Case #" << t << ": no\n";
                }
        }

}
```

Subtask 2: Long Cable Car (20 points)

The second subtask is the same as subtask 1, but this time with N pillars.

The idea to compare the slope remains the same. The only difference is, that you have to check whether the slope between the first two pillars is equal to all the other slopes.

This solution runs in $O(N)$.

```cpp
#include<bits/stdc++.h>

using namespace std;

int main(){
        int T;
        cin >> T;
        for(int t=1;t<=T;t++){
                int N;
                cin >> N;
                vector<int>pillars(N);
                for(int i=0;i<N;i++){
                        int tmp;
                        cin >> tmp;
                        pillars[i]=tmp;
                }
                bool constant_slope=true;
                int difference = pillars[0]-pillars[1];
                for(int i=2;i<N-1;i++){
                        if(pillars[i]-pillars[i+1]!=difference){
                                constant_slope=false;
                        }
                }
                if(constant_slope){
                        cout << "Case #" << t << ": yes\n";
                } else{
                        cout << "Case #" << t << ": no\n";
                }
        }

}
```

Subtask 3: Find an Enjoyable Part (30 points)

For this subtask, you need to get the longest consecutive subsequence out of the given n pillars, such that the view in this subsequence is enjoyable.

You can solve this problem in $O(N^2)$, by starting at every pillar and checking how far you can get until the slope changes. This solution is fast enought to get the points. But the optimal solution is in $O(N)$. For this solution you need the following observation:

Assume you have a pillar A and a constant slope for some following pillars. If you now choose any of these following pillars, which has still the same slope as the pillar A, the contiguous subsequence from there will end at the same point as the contiguous subsequence starting from A, because the slope at this pillar is the same as at A.

As the choosen pillar is after A, you know, that the contiguous subsequence is shorter, than the one starting from A. Therefore, you don't need to check any contiguous subsequence far a pillar which is part of the contiguous subsequence of another pillar.

This leads to the following solution:

As long as the slope is the same, check whether the slope is also the same for the next pillar. When the slope changes, start checking from that pillar again.

This solution runs in $O(N)$, because every pillar will be visited by our program exactly once.

```cpp
#include<bits/stdc++.h>

using namespace std;

int main(){
        int T;
        cin >> T;
        for(int t=1;t<=T;t++){
                int N,first,second;
                cin >> N;
                cin >> first >> second;

                int difference = first - second;
                int longest = 0;
                int current_length = 2;
                int last = second;

                for(int i=2;i<N;i++){
                        int current;
                        cin >> current;
                        if(last-current==difference){
                                current_length++;
                        }
                        else{
                                longest=max(longest,current_length);
                                current_length=2;
                                difference = last-current;
                        }
                        last=current;
                }
                cout << "Case #" << t << ": " << longest << "\n";
        }

}
```

Subtask 4: Removing Pillars (40 points)

In this section, you have to get the longest (not necessarily contiguous) subsequence, such that the slope in that subsequence is everywhere the same. Again there are two solutions, which are fast enough to get the points. The first one is in $O(N^3)$.

For every two pillars, calculate their slope. Now search the next pillar, such that its height equals the height of the second pillar + the calculated slope. Add this pillar to the subsequence. Now search the next pillar, such that its height equals the height of the last pillar in the subsequence + the calculated slope from the begin. Repeat this up to the end of the sequence, then start with the next two pillars.

The other algorithm is in $O(N^2)$. First, you need a container to store for every pillar and every slope an integer.The height of a pillar can be up to 10^9. Therefore the slope between two pillars can also be that big. If you create a regular array of that size, you will run out of memory. But the number of different slopes from any pillar to some pillar is only N. A map is a container which can store indices, which can be very big. To get or store a value in a map, you need $O(log(N))$ time. If you use a unordered map, it takes $O(1)$. In our container we store for every pillar and every slope the longest subsequence with that slope ending at that pillar.

The following algorithm uses a bottom-up approach to calculate all the values in the container. This means, that you use the lenghts of the subsequences calculated for the previous pillars, to calculate the lenghts of the subsequences for the next pillar.

You iterate through every pillar. For every pillar I you iterate through every pillar J before it. Calculate the difference between I and J. J is before I. Therefore you already calculated the lenghts of all the subsequences ending at J. You look how long the subsequence ending at J with slope equal to the difference between I and J is. Now you can store that lenght +1 at I.

```
#include<bits/stdc++.h>

using namespace std;

int main(){
        int T;
        cin >> T;
        for(int t=1;t<=T;t++){
                int N;
                cin >> N;
                vector<int>pillars(N);
                for(int i=0;i<N;i++){
                        int tmp;
                        cin >> tmp;
                        pillars[i]=tmp;
                }
                vector<unordered_map<int,int> > dp(N);
                int longest = 0;
                for(int i=0;i<N;i++){
                        for(int j=0;j<i;j++){
                                int difference = pillars[i] - pillars[j];
                                dp[i][difference] = dp[j][difference]+1;
                                longest = max(longest, dp[i][difference]);
                        }
                }
                longest++;//default value is 0. The length of a subsequence with 1 element is 1
                cout << "Case #" << t << ": " << N-longest << "\n";
        }

}
```

2 Gotthard Tunnel

In this task you were given the height of the ceiling and the floor of the new Gotthard Base Tunnel. You then had to calculate the length of different parts where the horizontal sight line is not blocked.

Subtask 1: Stationary (20 points)

This subtask can be solved in linear time ($O(n)$). You have to keep track of the current lowest ceiling part and highest floor part and progress from left to right. As soon as the minimum ceiling is lower or equals than the maximum floor, the line of sight is blocked and we have the distance we are looking for.

```cpp
#include <iostream>
#include <vector>

using namespace std;

typedef long long int lli;

int main(){
  int tcn;
  cin >> tcn;

  for(int t = 1; t <= tcn; t++){
    int n;
    cin >> n;

    // ceiling and floor
    vector<lli> c(n), f(n);

    // read input
    for(int i = 0; i < n; i++){
      cin >> c[i];
    }
    for(int i = 0; i < n; i++){
      cin >> f[i];
    }

    // calculate
    lli aminc = c[0], amaxf = f[0];
    lli result = n;
    for(int i = 0; i < n; i++){
      aminc = min(aminc, c[i]);
      amaxf = max(amaxf, f[i]);
      if(aminc <= amaxf){
        result = i;
        break;
      }
    }

    cout << "Case #" << t << ": " << result << "\n";
  }
```

```
41 }
```

Subtask 2: Short Walk (20 points)

In this subtask the limits were small enough to use an inefficient algorithm running in $O(n^2)$. We can reuse the algorithm from subtask 1 and start it from each position.

```cpp
#include <iostream>
#include <vector>

using namespace std;

typedef long long int lli;

int main(){
  int tcn;
  cin >> tcn;

  for(int t = 1; t <= tcn; t++){
    int n;
    cin >> n;

    // ceiling and floor
    vector<lli> c(n), f(n);

    // read input
    for(int i = 0; i < n; i++){
      cin >> c[i];
    }
    for(int i = 0; i < n; i++){
      cin >> f[i];
    }

    // calculate
    lli result = 0;
    for(int s = 0; s < n; s++) {
      lli aminc = c[s], amaxf = f[s];
      lli subresult = n-s;
      for(int i = s; i < n; i++){
        aminc = min(aminc, c[i]);
        amaxf = max(amaxf, f[i]);
        if(aminc <= amaxf){
          subresult = i-s;
          break;
        }
      }
      result = max(result, subresult);
    }

    cout << "Case #" << t << ": " << result << "\n";
  }
}
```

Subtask 3: Long Walk (40 points)

The limits in this subtask are too big for the algorithm from subtask 2 (Well, it should be, but most of you were able to use the same algorithm here anyways...). But we can do better, there exists an algorithm in $O(n)$.

We walk from left to right through the tunnel. For the ceiling and the floor we keep track of the positions we still can see from the current position. For each position we first add the new heights to the list. While we do this we pop all elements that are newly hidden (e.g. all floor positions that are lower than the newly added position). We can then calculate the length of sight to the left from the current position. For this we pop elements from the left side of the list as long as the most left element from the ceiling list and the most left element from the floor list overlap. We only pop the element that is located more to the left and we store the position of the last pop. The length of the line of sight at the current position is now $position(= s) - lastPos$. We keep track of the maximum of those lengths and we have the solution.

As we insert each position exactly once and we pop one element for each comparison we perform, the algorithm runs in linear time.

```cpp
#include <iostream>
#include <vector>
#include <queue>

using namespace std;

typedef long long int lli;

int main(){
  int tcn;
  cin >> tcn;

  for(int t = 1; t <= tcn; t++){
    int n;
    cin >> n;

    // ceiling and floor
    vector<lli> c(n), f(n);

    // read input
    for(int i = 0; i < n; i++){
      cin >> c[i];
    }
    for(int i = 0; i < n; i++){
      cin >> f[i];
    }

    // calculate
    lli result = 0;
    lli lastPos = -1;
    // deque for ceiling and floor
    deque<pair<lli, lli> > cq, fq;
    for(int s = 0; s < n; s++) {

      while(!cq.empty() && cq.back().first >= c[s]){
        cq.pop_back();
```

```
37        }
38        while(!fq.empty() && fq.back().first <= f[s]){
39          fq.pop_back();
40        }
41
42        cq.push_back(make_pair(c[s], s));
43        fq.push_back(make_pair(f[s], s));
44
45        while(cq.front().first <= fq.front().first){
46          if(cq.front().second < fq.front().second){
47            lastPos = cq.front().second;
48            cq.pop_front();
49          } else {
50            lastPos = fq.front().second;
51            fq.pop_front();
52          }
53        }
54
55        result = max(result, s - lastPos);
56      }
57
58      cout << "Case #" << t << ": " << result << "\n";
59    }
60  }
```

Subtask 4: Clearing the Tunnel (20 points)

This subtask is basically the same as the last one. There are only two differences: the numbers are too big for a normal int and you have to use long long int. Secondly you have to output $N - length(= result)$ instead of $length(= result)$. That's all.

```
1  #include <iostream>
2  #include <vector>
3  #include <queue>
4
5  using namespace std;
6
7  typedef long long int lli;
8
9  int main(){
10   int tcn;
11   cin >> tcn;
12
13   for(int t = 1; t <= tcn; t++){
14     int n;
15     cin >> n;
16
17     // ceiling and floor
18     vector<lli> c(n), f(n);
19
20     // read input
21     for(int i = 0; i < n; i++){
22       cin >> c[i];
23     }
```

```
24      for(int i = 0; i < n; i++){
25        cin >> f[i];
26      }
27
28      // calculate
29      lli result = 0;
30      lli lastPos = -1;
31      // deque for ceiling and floor
32      deque<pair<lli, lli> > cq, fq;
33      for(int s = 0; s < n; s++) {
34
35        while(!cq.empty() && cq.back().first >= c[s]){
36          cq.pop_back();
37        }
38        while(!fq.empty() && fq.back().first <= f[s]){
39          fq.pop_back();
40        }
41
42        cq.push_back(make_pair(c[s], s));
43        fq.push_back(make_pair(f[s], s));
44
45        while(cq.front().first <= fq.front().first){
46          if(cq.front().second < fq.front().second){
47            lastPos = cq.front().second;
48            cq.pop_front();
49          } else {
50            lastPos = fq.front().second;
51            fq.pop_front();
52          }
53        }
54
55        result = max(result, s - lastPos);
56      }
57
58      cout << "Case #" << t << ": " << n-result << "\n";
59    }
60  }
```

3 Matryoshka

This task is about Matryoshka dolls, which for our purposes are axis-aligned rectangles with a fixed width and height.

One doll can be put inside another doll if both dimensions of the first doll are strictly smaller than the corresponding dimensions of the second doll.

Subtask 1: One Matryoshka-Set (10 points)

Here, you are given heights and widths of N dolls, and your task is to figure out whether it is possible to nest all dolls within each other.

One possible solution was to note that it is sufficient to check for each pair of dolls whether one of them can be nested in the other. This leads to a simple solution in $\Theta(N^2)$.

However, there is also a simple solution in $O(N \log N)$.

Note that if all dolls are nested inside each other, then from innermost to outermost doll, both widths and heights are sorted and there are no duplicates in either dimension (and this is also a sufficient condition). The task can hence be solved by sorting the dolls according to either dimension and checking whether this gives a valid nesting order.

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

int main(){
  int t;
  cin>>t;
  for(int k=1;k<=t;k++){
    int n;
    cin>>n;
    vector<pair<int,int>> d(n);
    for(int i=0;i<n;i++) cin>>d[i].first;
    for(int i=0;i<n;i++) cin>>d[i].second;
    sort(d.begin(),d.end());
    bool r=true;
    for(int i=0;i+1<n;i++){
      r&=d[i].first<d[i+1].first &&
        d[i].second<d[i+1].second;
    }
    cout<<"Case #"<<k<<": "<<(r?"YES\n":"NO\n");
  }
}
```

Subtask 2: Largest Matryoshka-Set (20 points)

For this subtask, you again given N dolls, and your task is to determine the largest subset of dolls such that all of the dolls can be nested within each other.

By a similar argument as in the previous subtask, if we sort all dolls according to width, the problem reduces to finding the longest (strictly) increasing subsequence (LIS) according to both width and height. For this subtask, it was enough to compute the LIS in quadratic time using the recurrence

$$\mathrm{LIS}(j) = \max(\{1\} \cup \{\mathrm{LIS}(i) + 1 \mid 1 \leq i < j, \mathrm{width}(i) < \mathrm{width}(j), \mathrm{height}(i) < \mathrm{height}(j)\}),$$

where $\mathrm{LIS}(j)$ is the longest increasing subsequence ending at index j. The result is given by $\max\{\mathrm{LIS(i)} \mid 1 \leq i \leq n\}$.

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

int main(){
  int t;
  cin>>t;
  for(int k=1;k<=t;k++){
    int n;
    cin>>n;
    vector<pair<int,int>> d(n);
    for(int i=0;i<n;i++) cin>>d[i].first;
    for(int i=0;i<n;i++) cin>>d[i].second;
    sort(d.begin(),d.end());
    vector<int> lis(n,1);
    int r=0;
    for(int j=0;j<n;j++){
      for(int i=0;d[i].first<d[j].first;i++){
        if(d[i].second<d[j].second)
          lis[j]=max(lis[j],lis[i]+1);
      }
      r=max(r,lis[j]);
    }
    cout<<"Case #"<<k<<": "<<r<<'\n';
  }
}
```

Subtask 3: Many Dolls (20 points)

For this subtask the limits on N were increased, with the aim of making solutions that run in $\Omega(N^2)$ too slow. Unfortunately, even with the larger limits, many participants got $\Omega(N^2)$-solutions accepted.

The intended solution was to improve the running time to $O(N \log N)$.

First, assume that all widths are distinct. The innermost loop of the solution to the previous subtask then essentially computes the value $\max\{\mathrm{LIS}(i) \mid i < j, \mathrm{height}(i) < v\}$, for $v := \mathrm{height}(j)$. Our goal will be to optimize this computation such that it runs in logarithmic time instead of linear time.

There is a number of ways to achieve this bound. One can, for example, maintain a list

$$l_j[k] := \min\{\mathrm{height}(i) \mid 1 \leq i \leq j, \mathrm{LIS}(i) = k\}.$$

(The length of l_j is chosen as large as possible such that all minima are taken over non-empty sets.) Note that this list is always sorted in ascending order. The value $\max\{\mathrm{LIS}(i) \mid i < j, \mathrm{height}(i) < v\}$ equals $\max\{k \mid l_{j-1}[k] < v\}$ which can be found by binary search. l_{j+1} can then easily be computed from l_j in-place, modifying at most one list entry. The final result is given by the length of $l(n)$.

Alternatively, one can efficiently maintain a segment tree or fenwick tree for each j, allowing direct retrieval of the value $\max\{\text{LIS}(i) \mid i < j, \text{height}(i) < v\}$ in $O(\log N)$ for any value v.

The algorithms for distinct widths can also be applied to general widths if we order dolls of the same width in order of *decreasing* heights. Any subsequence strictly increasing in heights will then contain at most one doll of the same width.

```cpp
#include <bits/stdc++.h>
using namespace std;
struct D{int w,h;};

int main(){
  int t;
  cin>>t;
  for(int k=1;k<=t;k++){
    int n;
    cin>>n;
    vector<D> d(n);
    for(int i=0;i<n;i++) cin>>d[i].w;
    for(int i=0;i<n;i++) cin>>d[i].h;
    sort(d.begin(),d.end(),
      [](D a,D b){ return a.w<b.w||a.w==b.w&&a.h>b.h; });
    vector<int> l;
    for(int j=0;j<n;j++){
      auto i=lower_bound(l.begin(),l.end(),d[j].h);
      if(i!=l.end()) *i=d[j].h;
      else l.push_back(d[j].h);
    }
    cout<<"Case #"<<k<<": "<<l.size()<<'\n';
  }
}
```

Subtask 4: Group Photo (20 points)

For this subtask, there were three changes to the setup:

1. The given widths and heights now describe *types* of dolls, and for each type there is an arbitrary amount of dolls available.
2. Multiple dolls can be nested side-by-side in a larger doll, given that all their heights are strictly smaller than the height of the larger doll and given that the *sum* of their widths is strictly smaller than the width of the larger doll.
3. It is guaranteed that all given widths are distinct and that all given heights are distinct.

Your task is again to compute the maximal number of dolls that can be nested within each other.

For this subtask, the limits were small enough to allow solutions to pass that for some reason are asymptotically slower than the intended solution. See the discussion of the next subtask for the indended solution.

Subtask 5: Group Photo 2 (30 points)

See previous subtask for a task description. We discuss a solution in $O(N * \max_j \text{width}(j))$.

The intended solution is a simple modification of a dynamic programming algorithm for the subset sum problem. We first sort the dolls in ascending order according to height. We then compute the arrays t_j for each j from 1 to n, where $t_j[i]$ is the maximal number of dolls we can fit within width i[1] using only the first j types of dolls. If we can do this, the result is given by $\max_j t_{j-1}[\text{width}(j) - 1] + 1$.

We compute t_j in-place from t_{j-1}. The only thing that changes is that now dolls of type j may be used. Also, it is clear that any dolls that are nested within a doll of type j are from types strictly smaller than j. Hence, if the doll of type j is the outermost doll, the maximal number of nested dolls is $x := t_{j-1}[\text{width}(j) - 1] + 1$, which we can compute from the given information. With the new doll, it is now possible to nest x dolls within a width of $\text{width}(i)$. Now, for each i, the optimal number of dolls for width i either uses this new possibility[2], then $t_j[i] = t_j[i - \text{width}(j)] + x$, or not, then $t_j[i] = t_{j-1}[i]$. We pick whatever is better. Note how we have used t_j for an index smaller than i to compute t_j: this way, we allow multiple outermost dolls of type j to be placed side-by-side.

```cpp
#include <bits/stdc++.h>
using namespace std;
struct D{int w,h;};

int main(){
  int t;
  cin>>t;
  for(int k=1;k<=t;k++){
    int n;
    cin>>n;
    vector<D> d(n);
    for(int i=0;i<n;i++) cin>>d[i].w;
    for(int i=0;i<n;i++) cin>>d[i].h;
    sort(d.begin(),d.end(),
      [](D a,D b){ return a.h<b.h; });
    vector<int> t(100001);
    t[0]=0;
    int r=0;
    for(int j=0;j<n;j++){
      int x=t[d[j].w-1]+1;
      r=max(r,x);
      for(int i=d[j].w;i<(int)t.size();i++)
        t[i]=max(t[i],t[i-d[j].w]+x);
    }
    cout<<"Case #"<<k<<": "<<r<<'\n';
  }
}
```

[1] I.e. they use at *most* width i.
[2] In this case, i must be at least $\text{width}(j)$.

4 Waterslides

The program is given a directed graph where every node has exactly one incoming edge. The task is to determine the longest path in this graph.

For the first subtask, the starting node of the path is specified and the path only has to be the longest path starting from this node.

Subtask 1: Subtask 1 (20 points)

The graph can consist of multiple components. Each of those consists of one cycle and possibly multiple trees having their root in the cycle. The starting node can either be part of a cycle or a tree.

If the starting node is part of a tree and not part of a cycle, the solution is to find the longest path starting from the node. In a tree this can be done easily using a depth first search.

If the starting node is part of a cycle, the solution can be found by evaluating all trees connected to the cycle and adding the depth of the deepest tree to the length of the cycle. This corresponds to Stofl walking all the way around the cycle and then taking its path down the tree. The cycle can be detected by starting a depth first search and returning as soon as a node that is already on the stack is discovered.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <sys/resource.h>

using namespace std;

vector<vector<int> > conns;

vector<int> dfsDepth;   //stores at which distance from the starting point this node has
                        //been visited in the dfs
vector<int> cycleLength; //stores length of cycle containing this node

//returns longest path in a tree starting from pos
int getLongestPath(int pos, int depth){
  int ret = depth;
  for (int i = 0 ; i < conns[pos].size(); i++){
    if (cycleLength[conns[pos][i]] > 0 ) continue;  //ignore nodes that are
                                                    //part of a cycle(no tree)
    ret = max(getLongestPath(conns[pos][i],depth + 1), ret);
  }
  return ret;
}

//returns pair of cycle length and lowest dfs depth of a node in the cycle
pair<int,int> detectCycles(int pos, int depth){
  if (cycleLength[pos] >= 0) return {0,0};
  if (dfsDepth[pos] >= 0) {//We've been here before, return distance as cycle length
    return {depth - dfsDepth[pos], dfsDepth[pos]};
  }
  dfsDepth[pos] = depth; //store dfs path length
  pair<int,int> retval = {0,0};
```

```
  for (int i = 0 ; i < conns[pos].size(); i++){
    auto ret = detectCycles(conns[pos][i],depth + 1);
    if (ret.first > 0 && ret.second <= depth){  // making sure that node
                                                //is actually part of cycle
      retval = ret;
    }
  }
  cycleLength[pos] = retval.first;  //store the length of the cycle
                                    //which contains this node
  return retval;                    //if this node is part of a tree, store 0
}


int solve_testcase(){
  int nodes, startnode;
  cin >> nodes;
  conns.clear();
  conns.resize(nodes);
  dfsDepth.clear();
  dfsDepth.resize(nodes, -1);
  cycleLength.clear();
  cycleLength.resize(nodes, -1);
  for (int i = 0; i < nodes; i++){ //read input
    int num;
    cin >> num;
    for (int connectionIndex = 0; connectionIndex < num; connectionIndex++){
      int node;
      cin >> node;
      conns[i].push_back(node - 1);
    }
  }
  detectCycles(startnode, 0);//detect cycle containing the startnode

  int result = 0;
  for (int node = 0; node < nodes; node++){
    if (cycleLength[node] > 0){ //start a dfs from each node that is part of the
                                //cycle the startnode is in (if it exists)
      result = max(getLongestPath(node, 0) + cycleLength[node], result);
    }else if (node == startnode){
          result = max(getLongestPath(node, 0) + 1, result); //also start a dfs from the
                              //startnode directly, in case it isn't part of a cycle
        }
  }

  cout << result << endl;
}

int main() {
  ios_base::sync_with_stdio(0);
  cin.tie(0);
  int testcases;
  cin >> testcases;
  for (int testcase = 1; testcase <= testcases; ++testcase) {
    cout << "Case #" << testcase << ": ";
    solve_testcase();
  }
```

```
89 }
```

Subtask 2: Subtask 2 (20 points)

Unlike the first subtask, the starting node is no longer given for this subtask. A longest path in a tree always starts in the root which also happens to always be part of a cycle. Thus it is sufficient to check for the cycle that allows the longest path. In each cycle, this longest path is calculated by adding the depth of the deepest tree to the length of the cycle. As a result, the maximum of all cycles is returned. Stofl consequently starts by riding around a cycle and then down the longest tree branching off of the cycle.

Alternatively, one could reuse the solution from subtask one and evaluate all possible starting positions, returning the maximum. This solution would run in quadratic time.

Subtask 3: Subtask 3 (20 points)

This subtask is the same as Subtask 2, except that solutions should run in linear time.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <sys/resource.h>

using namespace std;

vector<vector<int> > conns;

vector<int> dfsDepth;   //stores at which distance from the starting point this node has
                        //been visited in the dfs
vector<int> cycleLength; //stores length of cycle containing this node

//returns longest path in a tree starting from pos
int getLongestPath(int pos, int depth){
  int ret = depth;
  for (int i = 0 ; i < conns[pos].size(); i++){
    if (cycleLength[conns[pos][i]] > 0 ) continue;  //ignore nodes that a part
                                                    //of a cycle(no tree)
    ret = max(getLongestPath(conns[pos][i],depth + 1), ret);
  }
  return ret;
}

//returns pair of cycle length and lowest dfs depth of a node in the cycle
pair<int,int> detectCycles(int pos, int depth){
  if (cycleLength[pos] >= 0) return {0,0};
  if (dfsDepth[pos] >= 0) {//We've been here before, return distance as cycle length
    return {depth - dfsDepth[pos], dfsDepth[pos]};
  }
  dfsDepth[pos] = depth; //store dfs path length
  pair<int,int> retval = {0,0};
  for (int i = 0 ; i < conns[pos].size(); i++){
    auto ret = detectCycles(conns[pos][i],depth + 1);
    if (ret.first > 0 && ret.second <= depth){  // making sure that node is
```

```
                                                //actually part of cycle
      retval = ret;
    }
  }
  cycleLength[pos] = retval.first;  //store the length of the cycle which
                                    //contains this node
  return retval;                    //if this node is part of a tree, store 0
}


int solve_testcase(){
  int nodes;
  cin >> nodes;
  conns.clear();
  conns.resize(nodes);
  dfsDepth.clear();
  dfsDepth.resize(nodes, -1);
  cycleLength.clear();
  cycleLength.resize(nodes, -1);
  for (int i = 0; i < nodes; i++){ //read input
    int num;
    cin >> num;
    for (int connectionIndex = 0; connectionIndex < num; connectionIndex++){
      int node;
      cin >> node;
      conns[i].push_back(node - 1);
    }
  }
  for (int node = 0; node < nodes; node++){ //detect cycles for each subgraph
    if (cycleLength[node] >= 0) continue;
    detectCycles(node, 0);
  }
  int result = 0;
  for (int node = 0; node < nodes; node++){
    if (cycleLength[node] > 0){ //start a dfs from each node that is part of a cycle
      result = max(getLongestPath(node, 0) + cycleLength[node], result);
    }
  }

  cout << result << endl;
}

int main() {
  ios_base::sync_with_stdio(0);
  cin.tie(0);
  int testcases;
  cin >> testcases;
  for (int testcase = 1; testcase <= testcases; ++testcase) {
    cout << "Case #" << testcase << ": ";
    solve_testcase();
  }
}
```

Subtask 4: Subtask 4 (40 points)

Every connected subgraph containg N nodes also has to contain exactly N edges. This is because every node is guaranteed to have exactly one incoming edge. There has to be a cycle, because every node has a parent and it's thus possible to endlessly go up the graph. If one edge was removed from the cycle, the graph would now have N-1 edges, thus being transformed into a tree. Hence the graph only contains one cycle.

The starting node always has to be inside the cycle, because if a node that is not part of the cycle is chosen as starting point, choosing its parent would yield a path that contains one additional edge and is consequently longer than the previous path. Which node in the cycle is chosen as starting point doesn't matter, as one can traverse the cycle arbitrarily many times, therefore visiting each edge in the cycle and being able to choose any exiting edge.

It is thus sufficient to determine the longest path for each subgraph by finding the longest path starting from one node in the cycle.

Calculating the length of a cycle requires each node in the subgraph to be evaluated exactly once and can hence be done in linear time. The depth of a tree is also derived by visiting each node in the tree and only done once per tree, subsequently running in linear time as well. The solution runs in linear time and also has a linear memory consumption for storing the graph.

5 Salesman

This task is about the distance that two Salesman with different strategies and starting points have to cover in order to visit all villages exactly once. The first Salesman "Stofl" always visits the village which is the furthest away from him next. The second Salesman "Vladimir" on the other hand always visites the village which is the nearest from his location.

Subtask 1: Calculate Paths (20 points)

We are given a Graph G=(V,E) which is

- fully connected, every vertex is connected with every other vertex.
- weighted, each edge has an associeted number which we call weight or cost.
- undirected, the edges have no direction i.e. the edges (x,y) and (y,x) are identical.
- weights are unique i.e. no two different edges have the same weight.

Further we're given two vertices, A and B, the starting Point of Stofl and Vladimir. Our task is to find the distance that "Stofl" and "Vladimir" will cover. We define N as the number of villages, i.e $N = |V|$. Note that we will use the terms village, node and vertex interchangeably.

Because all edges have a unique weight, all problems have a unique solution. This makes this task really straight forward to solve with a greedy algorithm. They both visit every village exactly once and their path will therefor be of length N. This means we can simply loop N-times and choose at each turn the min/max edge leading to a city that we haven't already visited. We make N comparisons for each village which gives a running time of $O(N^2)$ which is optimal because it already takes $O(N^2)$ to read the input.

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

int main() {
    int T;
    cin >> T;
    for(int t = 0; t < T; t++) {
        int N, A, B;
        cin >> N >> A >> B;
        vector<vector<int> > graph(N, vector<int>(N));

        for(int i = 0; i < N; i++) {
            for(int j = 0; j < N; j++) {
                cin >> graph[i][j];
            }
        }

        int S = 0, V = 0;
        int initValue[2] = {-1, 10001};
        bool compareResult[2] = {false, true};
        for(int z = 0; z < 2; z++) {
            pair<int, int> nextNode(A-1, 0);
```

```
            vector<bool> visited(N);

            for(int i = 0; i < N; i++) {
                S += nextNode.second;

                int current = nextNode.first;
                visited[current] = true;

                nextNode.second = initValue[z];
                for(int k = 0; k < N; k++) {
                    if(visited[k])
                        continue;
                    if((graph.at(current).at(k) < nextNode.second) == compareResult[z]){
                        nextNode = make_pair(k, graph[current][k]);
                    }
                }
            }

            swap(S, V);
            swap(A, B);
        }
        cout << "Case #" << t+1 << ": " << S << " " << V << "\n";
    }
}
```

Subtask 2: Examples (20 points total)

Now we asked you to generate input data with N villages for Subtask1 in a way such that the difference between Stofl's and Vladimir's path is a given number D. There are different strategies to achieve this goal, but most of them boil down to have both Vladimir and Stofl traverse the same path in the opposite direction with one different edge which results in the difference. We are going to describe one such approach.

Different Distances (10 points)

We enumerate the villages from 1 to N. Vladimir is going to start from village 1 and is going to visit the villages in ascending order e.g. from 1 to 2 to 3 ... to N. Stofl on the other hand will start at village N and is going to travel them in descending order e.g from N to N-1 .. to 1. How do we set the edge weights such that they both take these paths while keep all costs unique? We define the weight between village i and village j ($j \neq i$) to be $(i-1)*N+(j-1)$. The $(i-1)*N$ part is to assure that each weight is unique as $i-1<N$. The $(j-1)$ part is to assure that for evey village u and v the edge to u is shorter than to v iff u < v.

For example, if $N=6$ and Vladimir is at village 4, the shortest edge is to village 1, but he has already visited village 1 as well as 2, 3 and now 4. The next village he will visit is 5, because he has already visited village 1 to 4 and the edges to village 6 will be greater than the edge to 5. $W(\{4,5\}) = (4-1)*6+(5-1) = 22 <= 23 = (4-1)*6+(6-1) = W(\{4,6\})$ where $W(\{i,j\})$ is the weight of the edge between i and j.

Now we have a graph where Stofl and Vladimir take the same path and therfore cover the same distance. We solved the problem in the case $D=0$.

Equal Distances (10 points)

If $D \neq 0$ we simply change 3 costs:

- w(1, 2) becomes 1
- w(2, 3) becomes 2
- w(1, 3) becomes $3 + (D - 1)$

The Path Vladimir takes stays the same: $1 \to 2 \to 3 \to ... \to N$. The Path Stolf takes on the other hand becomes $N \to (N-1) \to ... \to 3 \to 1 \to 2$ because $W(\{3,1\}) = 3 + (D-1) = 2 + D > 2 = W(\{3,2\})$. Stofl and Vladimir travel the same path with the difference that Stofl takes $1 \leftrightarrow 3$ instead of $2 \leftrightarrow 3$ and the difference of their distances is $|W(\{2,3\}) - W(\{1,3\})| = |2 - 3 + (D-1)| = |D| = D$ as desired.

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

int main() {
    int T;
    cin >> T;
    for(int t = 0; t < T; t++) {
        int N, D;
        cin >> N >> D;
        vector<vector<int> > graph(N, vector<int>(N));
        for(int i = 0; i < N; i++) {
            for(int j = i+1; j < N; j++) {
                graph[i][j] = i*N + j + D;
            }
        }
        graph[0][1] = 1;
        graph[1][2] = 2;
        graph[0][2] = 3 + (D-1);

        cout << "Case #" << t+1<< " : ";
        cout << N << " " << N << " 1 " << "\n";
        for(int i = 0; i < N; i++) {
            for(int j = 0; j < N; j++) {
                cout << graph[min(i,j)][max(i,j)];
                if(j != N-1)
                    cout << " ";
                else
                    cout << "\n";
            }
        }
    }
}
```

Subtask 3: Proof (60 points)

Given a undirected weighted complet graph you had to proof that the path of Vladimir is never longer than the path of Stofl. We are going to demonstrate rigourousy why that statement is true.

You were not required to do that. It was sufficent for all points to give all the arguments we are going to use verbally.

The first Lemma was easier than the other two, a simple case-by-case analysis was enough and most participants that attempted is got all 20 points or 18 if they forgot the trivial case when all weights are zero. Lemma 2 was probably the least well solved problem and lots of participants struggled on how to attack the problem. If you want to show a property for every possible valid path the easiest way is generally to just assume you are given a valid path and then show the property for this arbitrary valid path. We focus on a village which Vladimirs leaves with cost 0 and Stofl leaves with cost 1 and show that once Vladimir arrives there, his total cost is already bigger than that of Stofl. For Lemma 3 we are going to reuse Lemma 2 by setting all edges weights greater than a certain number to 1 and all other to 0.

For Lemma 3 we proof the equivalent statement: The k-th most expensive edge Vladimir takes weights at most as much as the k-the most expensive edge Stofl takes. We then use Lemma 2 by setting all edge weights greater than the k-th most expensive edge Vladimir takes to 0 and as both path remain valid we conclud the statement.

Theorem

More formally we want to proof the following statement:
Given a graph $G = (V, E)$ where V is a set of vertices and $E \subset V \times V$ a set of edges and a function W : $E \to \mathbb{R}$ which assign a weight to each edge. Suppose we are give a valid path $S = (s_1, s_2, \ldots, s_N)$ for Stofl and valid path $V = (v_1, v_2, \ldots, v_N)$ where we denote the lengths of the paths with $|S| = \sum_{n=0}^{N-1} w(s_n, s_n + 1)$ and $|V| = \sum_{n=0}^{N-1} w(v_n, v_n + 1)$.

Lemma 1

Assume all edges have weight 0 or 1 and every edge with weight 1 is connected to the vertex A. More preciese, let $A \in V$ be the vertex such that if $W(e) = 1$ then $A \in e$. If all weights are zero i.e $W(e) = 0 \; \forall e \in E$ then both paths have obviously length 0.

We can now assume that there is at least one edge with weight 1. We are going to prove that $|S| \geq 1$ and $|V| \leq 1$ and conclude that $|S| \geq |V|$.

If Vladimir starts at A then the first edge he takes has either weight 1 or 0, depending on wheter there is an edge of weight 0 from A to any other node. By assumption every other edge, he is gonna take, has weight 0 because he can't visit A a second time. If Vladimir doesn't start at A he certainly takes $N - 1$ edges of weight 0 because there's always an edge with cost 0 available as long there's more than one vertex remaining(which could be A). In both cases we have $|V| \leq 1$.

If Stofl starts at A then he takes an edge with weight 1 because by assumption there is at least one such edge. If Stofl doesn't start at A then he will reach at one point the first vertex with an outgoing edge of weight 1. He will take this edge because the vertex on the other site is unvisited(otherwise this woludn't be the first such vertex). In both cases we have $|S| \geq 1$.

Lemma 2

Assume all edges in E have weight 0 or 1 i.e W : $E \to \{0, 1\}$. We want to show that for all possible paths for Stofl and Vladimir(S,V) we have $|S| \geq |V|$. As mentioned in the introduction we just assume that we are given $S = (s_1, s_2, ..., s_N)$ and $V = (v_1, v_2, ..., v_N)$, two arbitrary valid paths for Stofl and Vladimir.

Now we look at all vertices which Vladimir leaves with a edge of weight 1. We name these vertices by $c_1, c_2, ..., c_{|S|}$ where c_1 is the first vertex Vladimir leaves with cost 1, c_2 the second and so on. There are exactly $|S|$ such vertices, because by our assumption, there are only weights of 0 or 1.

Now we look at the first vertex in Stofl's path where Vladimir leaves the vertex with cost 1 but Stofl leaves with cost 0. If that happens after he visited $|S|$ vertices, then we already have $|S| \geq |V|$. Otherwise we can assume that the first such village in Stofl's path is the village c_k where $k \leq |S|$.

We observe that c_k is connected with a edge of weight 1 to every v_j that comes after c_k in V. Namely if $i < j$ and $c_k = v_i$ then Vladimir was forced to take a edge of weight 1 on v_i when he hasn't visited v_i yet. Meaning $W(c_k, v_j) = W(v_i, v_j) = 1 \; \forall \, i < j \leq |S|$. Otherwise he would have taken the edge to v_j from c_k instead.

As a remainder, Stofl takes an edge of weight 0 at c_k, but c_k is connected with weight 1 to all vertices v_j that come after c_k in V. Therefor Stofl has already visited them when he arrives at c_k and has already a path of length $N - i$ behind him($v_i = c_k$).

Note that $(N-1) - i \leq |S| - k$ because the vertices $v_i, ..., v_{N-1}$ contain $c_k, ..., c_{|S|}$. Which means $N - i \leq |S| - k + 1$ ($\star$).

As k is minimal, Stofl leaves the vertices $c_1, ..., c_{k-1}$ with edge weight 1. Combined with ($\star$) we reach the conclusion $|V| = (k-1) + N - i \leq k - 1 + |S| - k + 1 = |S|$.

Lemma 3

We prove the following equivalent statement: The k-th most expensive connection of Vladimir a_k costs at most as much as the k-th most expensive connection of Stofl b_k. Lets proof that this statement is equivalent:
$\Leftarrow$ is trivial, if $a_k \leq b_k$ then $\sum_{k=1}^{N} a_k \leq \sum_{k=1}^{N} b_k$.
$\Rightarrow$ follows by contraposition. Suppose it exists a k such that $a_k > b_k$. Now we add a large constant c to every edge with weight at least a_k. Both paths S and V remain valid. Stofl has one edge more in his path that has c added to it. If c is sufficent big, we have $\sum_{k=1}^{N} a_k > \sum_{k=1}^{N} b_k$ and therefor a valid counterexample.

Now we want to proof the original statement by proofing the equivalent statement. It is sufficent to proof for a given k that $a_k \leq b_k$. We change the cost of all edges that weight no less than the k-th most expensive connection of Vladimir a_k to 1 and the rest to 0. Obviously, both paths remain valid in the new graph.

In the new graph we have by definition $|V| = k$. Since both paths are valid we use Lemma 2 to deduce that $|S| \geq |V| = k$. Thus there are at least k edges in Stofls path that cost at least as much as a_k. It immeadeatly follows that $a_k \leq b_k$ which concludes our proof.

6 Racing

In this task you had to play the game vector racing.

Subtask 1: Find a path (20 points)

In this subtask you had to find any path visiting all checkpoints. For this you could choose an arbitrary order of visit for the checkpoints and use BFS to find a valid path between two checkpoints. As the given map didn't require a "jump across a wall" this was enough to find a solution.

```cpp
#include <bits/stdc++.h>

using namespace std;

// All possible directions
int dx[] = {0,0,0,1,1,1,-1,-1,-1};
int dy[] = {0,1,-1,0,1,-1,0,1,-1};

typedef pair<int, int> pii;

// Add the path specified by parent from start to activeElement to path recursively
void addall(vector<pii>& path, map<pii, pii>& parent, pii activeElement){
  if(parent[activeElement] == activeElement){
    path.push_back(activeElement);
    return;
  }
  addall(path, parent, parent[activeElement]);
  path.push_back(activeElement);
}

int main() {
  int w,h,players;
  vector<char> checkpoints;
  vector<vector<char> > rmap;
  vector<pii> path;

  // read map info
  cin >> w >> h >> players;
  rmap.resize(w, vector<char>(h));

  for (int y = 0; y < h; y++){
    for (int x = 0; x < w; x++){
      cin >> rmap[x][y];
      if (rmap[x][y] != '#' && rmap[x][y] != '$' && rmap[x][y] != '=' && rmap[x][y] != '.')
        checkpoints.push_back(rmap[x][y]);
    }
  }

  // find one possible solution
  for (int y = 0; y < h; y++){
    for (int x = 0; x < w; x++){
      // start at first start we see
```

```
43      if (rmap[x][y] == '='){
44        set<char> visitedCheckpoints;
45        pii activeElement;
46        activeElement = {x, y};
47        while(true){
48          // choose the next checkpoint we didn't yet visit
49          char nextCheckpoint = '$';
50          for(int i = 0; i < checkpoints.size(); i++){
51            if(visitedCheckpoints.find(checkpoints[i]) == visitedCheckpoints.end()){
52              nextCheckpoint = checkpoints[i];
53              break;
54            }
55          }
56          cerr << "search " << nextCheckpoint << endl;
57
58          // perform BFS from the current location to the chosen checkpoint
59          map<pii, pii> parent;
60          queue<pii> q;
61          set<pii> visited;
62          parent[activeElement] = activeElement;
63          q.push(activeElement);
64          visited.insert(activeElement);
65          while(!q.empty()){
66            activeElement = q.front();
67            q.pop();
68            char activeChar = rmap[activeElement.first][activeElement.second];
69            // we reached our chosen checkpoint
70            if(activeChar == nextCheckpoint) break;
71            // go through all possible next positions
72            // we only use velocity of one
73            for(int i = 0; i < 9; i++){
74              pii nextPosition = {activeElement.first + dx[i], activeElement.second + dy[i]};
75              // position is out of bounds
76              if(nextPosition.first < 0 || nextPosition.first >= w
77                  || nextPosition.second < 0 || nextPosition.second >= h)
78                continue;
79              // position is blocked
80              if(rmap[nextPosition.first][nextPosition.second] == '#')
81                continue;
82              // push new position in queue
83              if(visited.find(nextPosition) == visited.end()){
84                q.push(nextPosition);
85                visited.insert(nextPosition);
86                // we store the position from which we discovered this position
87                // with this we can later find the path from the start to this position
88                parent[nextPosition] = activeElement;
89              }
90            }
91          }
92
93          addall(path, parent, activeElement);
94          visitedCheckpoints.insert(nextCheckpoint);
95
96          // all checkpoints were visited, we're done
97          if(nextCheckpoint == '$')
98            break;
```

```
99          }
100         y=h;
101         break;
102       }
103     }
104   }
105
106   cerr << "path has length of " << path.size() << endl;
107
108   // print solution
109   cout << path[0].first << " "<< path[0].second<<endl;
110   for (int i = 1; i < path.size(); i++){
111     // read input for each round so the input stream doesn't block
112     // As we don't need it, we ignore it
113     int ignore;
114     cin >> ignore;
115     for(int j = 0; j < players; j++){
116       // player information
117       cin >> ignore >> ignore >> ignore >> ignore >> ignore >> ignore >> ignore;
118     }
119     int bmb, frz;
120     cin >> bmb >> frz;
121     for(int j = 0; j < bmb+frz; j++){
122       // cheese and mouse traps
123       cin >> ignore >> ignore >> ignore;
124     }
125     // output new position
126     cout << "M " << path[i].first << " " << path[i].second  << endl;
127   }
128   return 0;
129 }
```

Subtask 2: Find the optimal path (30 points)

In this subtask you had to find the optimal path on a very small map (20x15). As the map was very small the algorithm doesn't have to be very fast. Note that you had to use a mouse trap on yourself to find the shortest path.

The sample solution below uses a few shortcuts that only work on the map used for this subtask. For example there is only one checkpoint so we don't have to worry about the order of them.

To find the shortest path we use again a BFS. This time the current "node" in the graph doesn't only hold information about the position, but also about the current velocity, whether we used the mouse trap and whether we already visited the checkpoint. After we started the BFS from every possible start point we just take the shortest of all found solutions.

```
1 #include <bits/stdc++.h>
2
3 // don't do this at home, use a struct
4 #define px first.first.first
5 #define py first.first.second
6 #define vx first.second.first
7 #define vy first.second.second
8 #define vc second.first
9 #define bomb second.second
```

```cpp

using namespace std;

// All possible directions
int dx[] = {0,0,0,1,1,1,-1,-1,-1};
int dy[] = {0,1,-1,0,1,-1,0,1,-1};

typedef pair<int, int> pii;
typedef pair<pair<pii, pii>, pair<bool, int> > status;

struct sol{
  vector<pii> path;
  int bombTime;
};

// Add the path specified by parent from start to activeElement to path recursively
void addall(sol& solution, map<status, status>& parent, status activeElement){
  if(parent[activeElement] == activeElement){
    solution.path.push_back({activeElement.px, activeElement.py});
    return;
  }
  addall(solution, parent, parent[activeElement]);
  solution.path.push_back({activeElement.px, activeElement.py});
  if(activeElement.bomb == 0){
    solution.bombTime = solution.path.size() - 1;
  }
}

int main() {
  int w,h,players;
  vector<char> checkpoints;
  vector<vector<char> > rmap;
  sol solution, bestSolution;

  // read map info
  cin >> w >> h >> players;
  rmap.resize(w, vector<char>(h));

  for (int y = 0; y < h; y++){
    for (int x = 0; x < w; x++){
      cin >> rmap[x][y];
      if (rmap[x][y] != '#' && rmap[x][y] != '$' && rmap[x][y] != '=' && rmap[x][y] != '.')
        checkpoints.push_back(rmap[x][y]);
    }
  }

  // find best solution
  for (int y = 0; y < h; y++){
    for (int x = 0; x < w; x++){
      // start from each possible start
      if (rmap[x][y] == '='){
        set<char> visitedCheckpoints;
        status activeElement;
        activeElement = {{{x, y}, {0, 0}}, {false, -1}};
        // choose next checkpoint we didn't yet visit (there is only one)
        char nextCheckpoint = '$';
```

```
66        for(int i = 0; i < checkpoints.size(); i++){
67          if(visitedCheckpoints.find(checkpoints[i]) == visitedCheckpoints.end()){
68            nextCheckpoint = checkpoints[i];
69            break;
70          }
71        }
72        cerr << "search " << nextCheckpoint << endl;
73
74        // perform BFS from current location to goal, visiting the chosen checkpoint
75        map<status, status> parent;
76        queue<status> q;
77        set<status> visited;
78        parent[activeElement] = activeElement;
79        q.push(activeElement);
80        visited.insert(activeElement);
81        while(!q.empty()){
82          activeElement = q.front();
83          q.pop();
84          char activeChar = rmap[activeElement.px][activeElement.py];
85          // we reached the goal and visited the checkpoint
86          if(activeChar == '$' && activeElement.vc) break;
87          // go through all possible next positions
88          for(int i = 0; i < 9; i++){
89            status nextPosition = {{{activeElement.px + activeElement.vx + dx[i],
90              activeElement.py + activeElement.vy + dy[i]},
91              {activeElement.vx + dx[i], activeElement.vy + dy[i]}},
92              // check whether we reached the checkpoint
93              {activeElement.vc || activeChar == nextCheckpoint,
94              activeElement.bomb == -1 ? -1 : 1}};
95            // position is out of bounds
96            if(nextPosition.px < 0 || nextPosition.px >= w ||
97                nextPosition.py < 0 || nextPosition.py >= h)
98              continue;
99            // position is blocked
100           if(rmap[nextPosition.px][nextPosition.py] == '#')
101             continue;
102           // push new position in queue
103           if(visited.find(nextPosition) == visited.end()){
104             q.push(nextPosition);
105             visited.insert(nextPosition);
106             // we store the position from which we discovered this position
107             // with this we can later find the path from the start to this position
108             parent[nextPosition] = activeElement;
109           }
110           // if we didn't yet use the mouse trap, there are more options
111           if(nextPosition.bomb == -1){
112             nextPosition.bomb = 0;
113             nextPosition.vx = 0;
114             nextPosition.vy = 0;
115             // push new position in queue
116             if(visited.find(nextPosition) == visited.end()){
117               q.push(nextPosition);
118               visited.insert(nextPosition);
119               // we store the position from which we discovered this position
120               // with this we can later find the path from the start to this position
121               parent[nextPosition] = activeElement;
```

```
122               }
123             }
124           }
125         }
126
127         solution.path.clear();
128         addall(solution, parent, activeElement);
129
130         // check whether the new solution is better than the old best solution
131         if(bestSolution.path.size() == 0 || solution.path.size() < bestSolution.path.size()){
132           bestSolution = solution;
133         }
134       }
135     }
136   }
137
138   cerr << "path has length of " << bestSolution.path.size() << endl;
139
140   // print solution
141   cout << solution.path[0].first << " "<< solution.path[0].second << endl;
142   for (int i = 1; i < solution.path.size(); i++){
143     // read input for each round so the input stream doesn't block
144     // As we don't need it, we ignore it
145     int ignore;
146     cin >> ignore;
147     for(int j = 0; j < players; j++){
148       // player information
149       cin >> ignore >> ignore >> ignore >> ignore >> ignore >> ignore >> ignore;
150     }
151     int bmb, frz;
152     cin >> bmb >> frz;
153     for(int j = 0; j < bmb+frz; j++){
154       // cheese and mouse traps
155       cin >> ignore >> ignore >> ignore;
156     }
157     if(solution.bombTime == i)
158       // output the mouse trap
159       cout << "B " << solution.path[i].first << " " << solution.path[i].second << endl;
160     // output new position
161     cout << "M " << solution.path[i].first << " " << solution.path[i].second  << endl;
162   }
163   return 0;
164 }
```

Subtask 3: Creativity Tourney (50 points)

As this is the creativity task, there are many possible solutions and we don't know the best one either.

We played several rounds on several maps to determine the quality of the submitted algorithms. According to those games we calculated a ranking and awarded points according to it.

Unfortunately most submitted programs contained more or less serious bugs we had to fix before we were able to test the algorithms. Depending on what we had to fix we deducted some points.

For the base score before the deduction we used the following table:

1. 50P
2. 40P
3. 30P
4. 20P
5. 10P
6. 7P
7. 5P