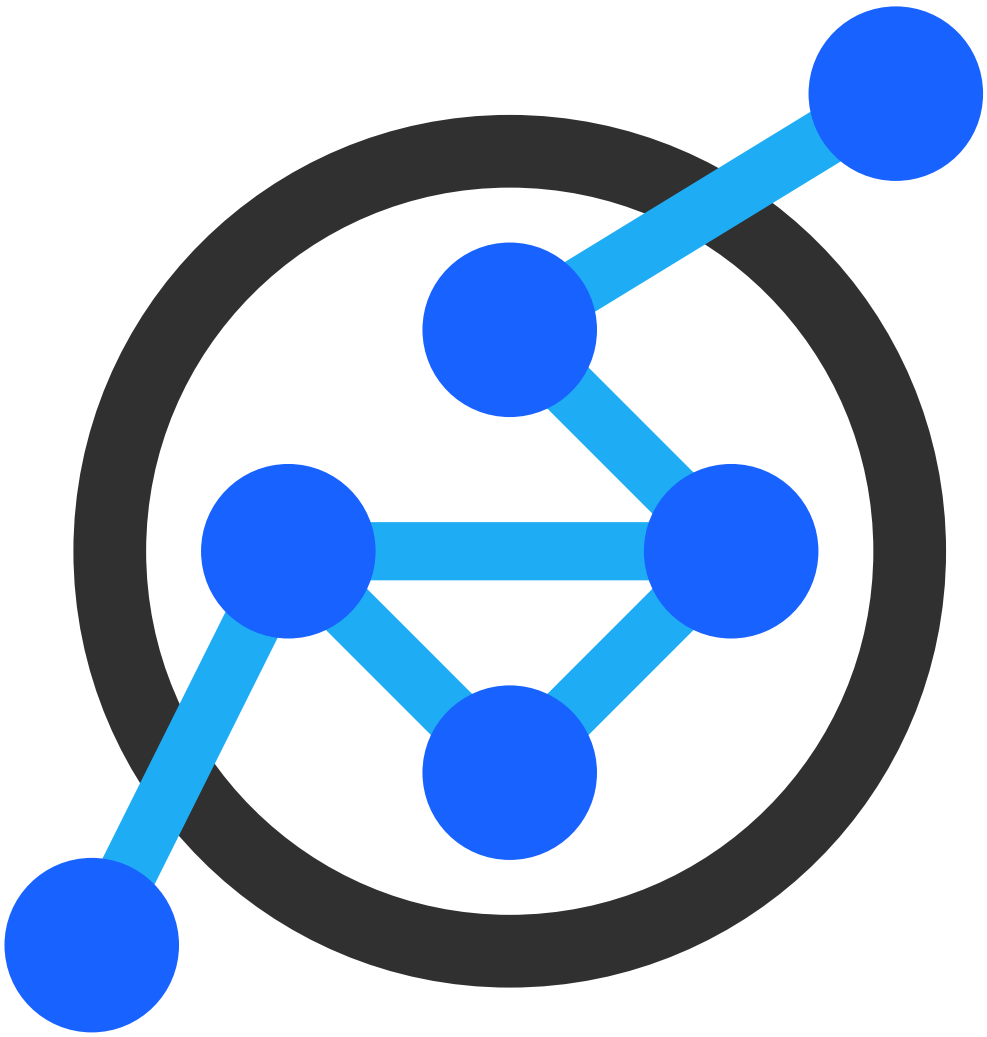
SOI Workshops 2019

Solution Booklet



Swiss Olympiad in Informatics

November 2019

Graph Representation

If we want to work with graphs, we need a way to store them in an efficient way. Most of the time, adjacency lists are the most practical representation as they allow us to quickly loop over all of the neighbours of a certain vertex. One of the other advantages of an adjacency list – in comparison to an adjacency matrix – is, that it only stores the neighbours of a vertex and not a value for each combination of two vertices, so we can run DFS and BFS in $O(n + m)$ instead of $O(n^2)$ (of the adjacency matrix). Also it only needs $O(n + m)$ memory (instead of $O(n^2)$).

In most of the tasks, graphs are given by two numbers n and m, the number of vertices and the number of edges, followed by m lines with two numbers a and b, each representing an edge from a to b. To read it into an adjacency list, just push a back on the list of b, and vice versa. The code looks like this:

```
#include <soi>

signed main()
{
    int n = read_int(); // The number of vertices
    int m = read_int(); // The number of edges

    vector<vector<int>> graph(n); // Our adjacency list

    for (int i = 0; i < m; i++) // For each edge
    {
        int from = read_int(); // Read the first vertex
        int to = read_int(); // Read the second vertex
        graph[from].push_back(to); // Push a to b so we can go from a to b.
        graph[to].push_back(from); // Push b to a so we can go from b to a.
    }

    for (int i = 0; i < n; i++) {
        print(graph[i]); // Print the adjacency list.
    }
}
```

Stars in the Sky

Given a graph, we should find out if it's a star. There are many approaches which try to group the vertices into center and not-center, but the code often gets uglier than it should be. In fact, the solution of this problem is rather short. The first observation one needs to make, is that a star consisting of n vertices has exactly $n-1$ vertices: for each vertex except for the one in the center, there has to be a connection to the one in the center. This condition is necessary, but not yet sufficient (look at a path).

The second condition for our graph is, that it has to contain at least one vertex with degree $m = n-1$, namely the center. This doesn't sound interesting yet, the only thing we did is reformulating the problem. But in fact, these two conditions together are sufficient: If the degree of a vertex is $n-1$, then all of the other vertices have each one connection to the center-vertex. And there can't be any other edge which connects two of the non-center vertices, because each edge starts at the center vertex (otherwise his degree would be less than m). So the only thing we need to check is the number of edges, and if there exists a vertex with degree $n-1 = m$.

The code looks like this:

```cpp
#include <soi>

signed main()
{
    int n = read_int(); // Number of vertices
    int m = read_int(); // Number of edges

    if (n - 1 != m) // Our graph has not the right number of edges
    {
        print("other");
        return 0;
    }

    vector<vector<int>> graph(n);

    for (int i = 0; i < m; i++) // Read the graph.
    {
        int from = read_int();
        int to = read_int();
        graph[from].push_back(to);
        graph[to].push_back(from);
    }

    for (int i = 0; i < n; i++) // Loop over every vertex
    {
        if (graph[i].size() == m)
        {
            print("star"); // We found a vertex with degree m = n - 1
            return 0;
        }
    }
    print("other"); // We didn't find any vertex with degree m
}
```

Components

The task was to count the number of components in a graph. We can do this by looping through all vertices. If a vertex is unvisited, we found a new component! To "discover" all other vertices and mark them as visited, we use a simple dfs which does nothing else than "visiting" vertices. If we arrive at a vertex we have already visited, we just ignore it. In code:

```cpp
#include <soi>

vector<vector<int>> graph;
vector<bool> visited;

void dfs(int curr) { // Simple dfs which only "visites vertices".
{
    if (visited[curr]) return;
    visited[curr] = true;

    for (int next : graph[curr])
    {
        dfs(next);
    }
}

signed main()
{
    int n = read_int(); // Read the input
    int m = read_int();

    graph.resize(n);

    for (int i = 0; i < m; i++)
    {
        int from = read_int();
        int to = read_int();
        graph[from].push_back(to);
        graph[to].push_back(from);
    }

    visited.resize(n, false); // Initialize our visited-array

    int cnt = 0;

    for (int i = 0; i < n; i++) // Loop through the vertices
    {
        if (!visited[i])
        {
            cnt++;  // We found a unvisited vertex!
            dfs(i);
        }
    }

    print(cnt);
}
```

Clawcrane

In the task clawcrane, we want to maximize the sum of the values of the prizes we can get in a single try, as some prizes clump together. If we choose to take the toy at vertex, we get all the toys in the same component. So we can reformulate the problem a bit: find the maximal value among the sums of the vertices in a component. Again, we can use dfs. Our dfs function returns the sum of the values of the vertices we did visit, so after we visited all vertices in a component, it returns the sum of the component. In total, we do this: We loop over all vertices. If we didn't visit a vertex, we use dfs to calculate the sum in the component of this vertex. If it is already visited, just ignore it. During all this we store the current maximum in a variable. In code:

```
#include <soi>

vector<vector<int>> graph;
vector<int> visited;

vector<int> values;

int dfs(int curr) // Returns the sum of the vertices our dfs has already discovered.
{
    if (visited[curr]) return 0; // We have been here already
    visited[curr] = true;

    int sum = values[curr]; // Add the value of this vertex to the result

    for (int next: graph[curr])
    {
        sum += dfs(next); // Add the values from the dfs of the neighbours to the result
    }

    return sum;
}

signed main()
{
    int n = read_int(); // Read input
    int m = read_int();

    values.resize(n);

    for (int i = 0; i < n; i++)
    {
        int v = read_int();
        values[i] = v;
    }

    graph.resize(n);

    for (int i = 0; i < m; i++)
    {
        int from = read_int();
        int to = read_int();
        graph[from].push_back(to);
        graph[to].push_back(from);
    }

    visited.resize(n, false);

    int maxval = 0; // Our global variable with the maximum

    for (int i = 0; i < n; i++)
    {
        if (visited[i]) continue;
        maxval = max(maxval, dfs(i)); // Call dfs and update the maximum
```

```
54      }
55
56      print(maxval);
57  }
```

Passports

The problem passports introduces the concept of vertex-coloring. In passports, we want to assign one of the two passports to each of the nations such that we don't assign the same passport to two nations in conflict. To color a graph, we assign to each vertex a number ("color") such that no two adjacent vertices have the same number. Here we just want to find a coloring of the graph with 2 colors. Let's look at a vertex v. As we didn't assign any other color to the graph, we can – without loss of generality – assign the color 0 to it. Then for each neighbour, we have to use the value 1 (otherwise the two adjacent vertices have the same number). But each neighbour of a vertex with number 1, must have the color 0, and so on. We can do this with a dfs: memorize the current color, and when we call dfs for all the neighbours, we need to inverse the colors. If our dfs arrives at a vertex which already has a color, if the color we want to assign to it and the color it already has are the same, then we can just continue. But if these two colors are different, then it's impossible to assign the colors. The two different colors mark that we have found an odd cycle, and to color an odd cycle with 2 colors is impossible. So we can just use dfs. As there may be several components in the graph and dfs just goes through one of it, we again have to loop through all the vertices and run dfs from each unvisited vertex.

```cpp
#include <soi>

vector<vector<int>> graph;
// We can combine the visited and the color array to one array:
// -1 when not visited yet, otherwise the color
vector<int> color;

// Dfs taking the current color as the second argument.
// Returns true when it's possible, false when not
bool dfs(int curr, int col)
{
    // If our new color is equal to the old, we can return true.
    // If it's different, it's not possible so we return false
    if (color[curr] != -1) return col == color[curr];
    color[curr] = col; // Set the current color

    for (int next : graph[curr])
    {
        // Use dfs on the next vertex with the other color.
        // If it returns false, we also return false
        if (!dfs(next, !col)) return false;
    }
    return true; // Nothing returned false, so we can return true
}

signed main()
{
    int n = read_int(); // Read input
    int m = read_int();

    graph.resize(n);

    for (int i = 0; i < m; i++)
    {
        int from = read_int();
        int to = read_int();
        graph[from].push_back(to);
        graph[to].push_back(from);
    }

    color.resize(n, -1);

    bool possible = true;

```

```
45      for (int i = 0; i < n; i++) // Loop through all the vertices
46      {
47          if (color[i] != -1) continue; // if (visited[i]) continue;
48
49          if (!dfs(i, 0))
50          {
51              possible = false; // It's not possible
52              break;
53          }
54      }
55
56      if (possible)
57      {
58          vector<vector<int>> groups(2); // The two groups
59          for (int i = 0; i < n; i++)
60          {
61              groups[color[i]].push_back(i); // Add each vertex to his group
62          }
63          print(groups[0]); // Print the groups
64          print(groups[1]);
65      }
66      else
67      {
68          print("Order more passports."); // Not possible, we need more
69      }
70  }
```

Erasmus

The task of erasmus looks a bit strange at first. We have these talents and we don't really know what to do. In these tasks, it's often useful to look at just a single talent, try to solve it for a single talent, and then look at the solution of the whole problem. Sometimes we can just apply the same algorithm for each "talent", sometimes we have to be more efficient and combine them.

We have one single talent and want to make our gang as big as possible. Let's look at a mouse which is not impressed. This mouse doesn't join us, and it's also not going to tell his friends about this talent. This means, we could basically just remove every unimpressed mouse from the graph. What we have left are the mice that are impressed and will join us, and tell all their friends about us. If we look at the mice as a graph, we can see that every mouse in the same component joins our gang. Again reformulating this a bit, we can see that we want to find the component with the biggest size. We can do this with dfs in $O(n + m)$. If we do this for each talent, we get a running time of $O(nt + mt)$, which fits into the limits! Our algorithm looks like this: For each talent, remove all mice with a skill-level of this talent bigger than Stefl's. Then calculate the size of the biggest component using dfs. The result is the maximum of this for each talent. We can do a bit better: Instead of removing the mice from the graph, we can just ignore them when we "visit" them in our dfs. This makes the code much simpler.

```cpp
#include <soi>

vector<vector<int>> graph;
vector<vector<int>> talents;

vector<bool> visited;

int dfs(int curr, int talent) // Dfs at vertex curr looking at talent t
{
    if (visited[curr]) return 0; // We have been here before
    visited[curr] = true;

    // If this mouse has a bigger talent value than stefl, so we should ignore it, so we return 0
    if (talents[0][talent] <= talents[curr][talent]) return 0;

    int sum = 1;

    for (int next : graph[curr])
    {
        sum += dfs(next, talent); // Calculate the component size
    }
    return sum;
}

signed main()
{
    int n = read_int(); // Read Input
    int m = read_int();
    int t = read_int();

    talents.resize(n, vector<int>(t));

    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < t; j++)
        {
            int v = read_int();
            talents[i][j] = v;
        }
    }

    graph.resize(n);

```

```
44      for (int i = 0; i < m; i++)
45      {
46          int from = read_int();
47          int to = read_int();
48          graph[from].push_back(to);
49          graph[to].push_back(from);
50      }
51
52      int best = -1; // Biggest gang size
53      int btal = -1; // Talent with biggest gang size
54
55      for (int ctal = 0; ctal < t; ctal++)
56      {
57          visited.assign(n, false); // Reset all the visited flags
58          for (int i = 1; i < n; i++)
59          {
60              if (visited[i]) continue;
61              int nv = dfs(i, ctal); // Calculate the value
62              if (best < nv)
63              {
64                  // Set the current maximum and the current talent
65                  best = nv;
66                  btal = ctal;
67              }
68          }
69      }
70
71      print(btal, best);
72  }
```

Shortest Path

This task was about computing the shortest path from vertex a to vertex b in an unweighted graph. Thus, we can directly apply BFS (Breath-First Search).

Below is a code that has BFS in its own function. The function takes as input a graph, and returns the distance array to all vertices, relative to some source vertex `start`.

```cpp
#include <soi>

vector<int> bfs(vector<vector<int>> const& graph, int start) {
  vector<int> dist(graph.size(), -1); // -1 means unvisited
  // dist[v] is the distance between start and v
  queue<int> q; // BFS queue, the pending vertices

  // first we set the distance, then we push it to the queue
  dist[start] = 0;
  q.push(start);

  while (!q.empty()) {
    int v = q.front();
    q.pop();
    for (auto w : graph[v]) {
      if (dist[w] == -1) { // if unvisited:
        // set the distance, then we push it to the queue
        dist[w] = dist[v] + 1;
        q.push(w);
      }
    }
  }
  // return the distance array
  return dist;
}

signed main() {
  int n = read_int();
  int m = read_int();
  vector<vector<int>> graph(n);
  for (int i=0; i<m; ++i) {
    int a = read_int();
    int b = read_int();
    graph[a].push_back(b);
    graph[b].push_back(a);
  }

  int start = read_int();
  int target = read_int();

  auto dist = bfs(graph, start);
  print(dist[target]);
}
```

Seven-segment display

We are given a display where we can go from displaying number a to $a + 1 \bmod 10^6$ and $C \cdot a \bmod 10^6$ and have to find the minimum number of steps to reach number Y starting from number X.

The first step is to model this as a graph problem. We represent each number i from 0 to $10^6 - 1$ as vertex i. We draw an edge from i to $i + 1 \bmod 10^6$ and from i to $C \cdot i \bmod 10^6$. Note that in this case, the graph is *directed*. We can go from a to $a + 1$ but not vice versa. What this means is that we only insert one direction when we build the graph. Then, the length of the shortest path from X to Y is exactly the minimum number of operations that we need to go from X to Y.

We first construct the graph and the use exactly the same BFS function as before:

```cpp
#include <soi>

vector<int> bfs(vector<vector<int>> const& graph, int start) {
  vector<int> dist(graph.size(), -1);
  queue<int> q;

  dist[start] = 0;
  q.push(start);

  while (!q.empty()) {
    int v = q.front();
    q.pop();
    for (auto w : graph[v]) {
      if (dist[w] == -1) {
        dist[w] = dist[v] + 1;
        q.push(w);
      }
    }
  }
  return dist;
}

const int M = 1e6;
signed main() {
  int c = read_int();
  vector<vector<int>> graph(M);
  for (int i=0; i<M; ++i) {
    graph[i].push_back((i+1)%M);
    graph[i].push_back((c*i)%M);
  }

  int start = read_int();
  int target = read_int();

  auto dist = bfs(graph, start);
  print(dist[target]);
}
```

We define $M = 10^6$ using the line "`const int M=1e6`". "`const`" means that we are not allowed to change it in the program. This prevents us from acidentally changing "`M`" (which is usually a bug). Also, it allows the compiler to perform some optimizations, and makes the code a bit faster. "`1e6`" is a shorthand for 10^6.

Note that we don't actually have to construct the graph ourselves. This is a case of an *implicit* graph. We can just compute the neighbors on the fly:

```cpp
#include <soi>

const int M = 1e6;

signed main() {
```

```
  int c = read_int();
  int start = read_int();
  int target = read_int();

  vector<int> dist(M, -1);
  queue<int> q;
  dist[start] = 0;
  q.push(start);

  while (!q.empty()) {
    int v = q.front();
    q.pop();
    for (auto w : {(v+1)%M, c*v%M}) {
      if (dist[w] == -1) {
        dist[w] = dist[v] + 1;
        q.push(w);
      }
    }
  }
  print(dist[target]);
}
```

While both codes are $O(M)$ (which is actually just $O(1)$ since M is a constant), the second code is quite a bit faster since we don't need to construct the whole graph.

Tiles

We have to calculate the number of possibilities to fill a $d * n$ field with tiles of size $d * 1$. Counting stuff is a good indicator for dp, and if we look at the problem closer, we can see that we can calculate the number of possibilities recursively! Let the number of possibilites to fill a $d * i$ field be dp_i. To continue our pattern, there are two possibilities: either we place one tile vertically using only the next field, or d tiles horizontally, using the next d fields. Or looking from the other direction, the number of possibilites for the $d * i$ field is equal to the number of possibilities for the $d * (i - 1)$ field (the last thing we placed was a vertical tile) plus the number of possibilities of a $d * (i - d)$ field (we placed d horizontal tiles). We can write this as a recursion formula: $dp_i = dp_{i-1} + dpi - d$. The base cases for this formula are $dp_0 = 1$ (there is exactly one possibility for an emtpy field) and $dp_{i<0} = 0$ (negative sizes are not allowed). This is everything we need: we know how to split the problem up in subproblems (recursion formula), we know our base cases, and we know which subproblem is the relevant subproblem. So let's start coding!

```cpp
#include <soi>

vector<int> table; // Memorize the values we know already
int d;

int dp(int i)
{
    if (i == 0) return 1; // Base Cases
    if (i < 0) return 0;
    if (table[i] != -1) return table[i]; // We know this value already

    return table[i] = dp(i - 1) + dp(i - d); // Calculate the value using the formula
}

signed main()
{
    d = read_int();
    int n = read_int();

    if (d == 1) // Special case if d == 1: It doesn't matter how we arrange them, it's always the same
    {
        print(1);
        return 0;
    }

    table.resize(n + 1, -1); // Resize the table. Important: REsize it to n + 1 as we query the n-th value
    print(dp(n)); // Calculate the value
}
```

Triangle Sum (with path)

Given a triangle of numbers, we had to find the way from the top to the bottom with the highest sum. In total, there are 2^n different paths from top to bottom, so testing each path is too slow. The crucial observation one had to make in this task, was that if we know the best path from the top to a number a in the triangle, then every best path to any other number b which contains a will take the best path to a and from there the best path to this b. To see exactly what this means, let's look at a specific number in the triangle. There are only two possibilities of where we can come from, the number above it to the left, and the number above it to the right. If we know the sum of the best path to these two numbers, we can easily calculate the best path to our number: We choose the maximum of these 2 paths and add the value of our number to it. And now this really looks like dp: our problem depends on two smaller subproblems, namely the two numbers above our number! Now, to calculate the maximal path, we start at the top, and for each row, we calculate the best sum for each number. These values depend from the values above them, but since we loop from top to bottom, we already now these values. To find the maximal path, we just have to take the maximal value in the last row.

This is enough to solve the trisum problem, but in trisumpath, we also have to print the path (print "L" if we go left or "R" if we go right for each row). It looks very easy to just memorize for each number the best path to it and for each number copy the path of the correct "parent" and append the direction we have chosen, but this is adds another factor of n to our running time, which would make our code to slow. There are two solutions of how to do it, and which one to choose really depends on the problem. The first option is to memorize a value for each number in the triangle which tells us if we came from the left or from the right. In the end we know which number is the end of the best path, so we can look there from which direction we came, go to this number, look from where we cam to this number and so on, until we arrive at the top. The other option is just to look at the parents, and look at which one our program has chosen by recalculating it. I chose the second option here, but here, both work well.

```cpp
#include <soi>

signed main()
{
    vector<vector<int>> triangle;
    vector<vector<int>> sums;

    int n = read_int();

    for (int i = 0; i < n; i++) // Initialize the vectors and read the input
    {
        sums.push_back(vector<int>(i + 1, -1));
        triangle.push_back(vector<int>(i + 1, -1));

        for (int j = 0; j <= i; j++)
        {
            triangle[i][j] = read_int();
        }
    }

    for (int i = 0; i < n; i++) // Calculate our dp
    {
        for (int j = 0; j <= i; j++) // For each number in our triangle
        {
            int best = 0;
            // We have to check wheter we are on the borders, the leftmost number has
            // no left parent and the same for the rightmost
            if (j < i) best = max(best, sums[i - 1][j]); // Come from the right
            if (j > 0) best = max(best, sums[i - 1][j - 1]); // Come from the left

            sums[i][j] = best + triangle[i][j]; // The best possibility
        }
```

```
    }

    int mval = -1; // Find the best value
    int mind = -1; // Save the position of the best value so we can start to calculate the path from there

    for (int i = 0; i < n; i++)
    {
        if (sums[n - 1][i] > mval)
        {
            mval = sums[n - 1][i];
            mind = i;
        }
    }

    print(mval); // Print the best value

    string path = "";
    int curr = mind;

    for (int i = n - 1; i > 0; i--)
    {
        if (curr < i && sums[i][curr] == sums[i - 1][curr] + triangle[i][curr]) // Check if we came from the right
        {
            path.push_back('L'); // If we came from the right, we chose left there to get here.
        }
        else
        {
            path.push_back('R');
            curr--;
        }
    }

    reverse(path.begin(), path.end()); // Reverse the path, because we recalculated it from the bottom to the top
    print(path);
}
```

Springshoes

If we look at the task, there are very strong indicators that this is a dp task: One of them is, that we have to count something. Another is, that the solution of the problem depends on the solution for the same problem, but for smaller n's. Then let's apply DP on it! The first thing we have to do, is to find some the recursion formula for the task. Let DP[i] be the number of ways we can get to a point with distance i from the beginning. If there is a tree on i, then DP[i] = 0, because we can't land on a tree. Now let's look from where we can reach i. Let's say we're at distance j. Then we can jump to i if $i - b \leq j \leq i - a$. This means, we can add up the possibilities for each such j, giving us a recurrence formula which looks like this: $\text{DP}[i] = \sum_{j=i-b}^{i-a} \text{DP}[j]$. The thing is, if we make a loop to calculate this sum, we'll get a running time of $O(n^2)$ which is too slow. The solution is to store something like a global sum variable, and when we calculate the next value of our DP, we just need to move the sum one to the right, this means removing DP[$i - b - 1$] and adding DP[$i - a$].

```cpp
#include <soi>

signed main()
{
    // Read the input
    int n = read_int();
    int a = read_int();
    int b = read_int();
    int k = read_int();

    vector<int> vals(n + 1, 0);
    vector<bool> trees(n + 1, false); // Saves for each place wheter it's a tree

    for (int i = 0; i < k; i++)
    {
        int p = read_int();
        trees[p] = true;
    }

    vals[0] = 1; // Our table

    int sum = 0; // Our sum variable

    for (int i = 1; i <= n; i++) // The dp
    {
        if (i - b - 1 >= 0) sum -= vals[i - b - 1];
        if (i - a >= 0) sum += vals[i - a];
        if (!trees[i]) vals[i] = sum;
        vals[i] %= (int)(1e9+7); // Don't forget to take the modulo!
    }

    print(vals[n]);
}
```

Hoist the flags!

At first the task looks kind of strange. We send n mice, the i-th mouse changes the state of the flags $1, i, i, 3i, 4i \ldots$ And we should calculate if the n-th flag is up or down at the end, where n can get up to 10^{10000}.

The first algorithm we could think of is to just simulate the mice. At the beginning, we start with the n-th flag down. Then for each mouse i with $1 \leq i \leq n$, if n is divisible by i, we change the state of the flag. In the end, we just check, if the flag is up or down. This approach runs in $O(n)$ and gives 30 points:

```
#include <soi>

signed main() {
    int n = read_int();
    bool flag = false;

    for (int i = 1; i <= n; i++)
    {
        flag = flag ^ !(n % i);
    }

    if (flag)
    {
        print("....o\n####|\n####|\n####|\n....|\n....|\n....|\n....|\n....|\n....|");
    }
    else
    {
        print("....o\n....|\n....|\n....|\n....|\n....|\n####|\n####|\n####|\n....|");
    }
}
```

But since this runs in $O(n)$, this is ways too slow (we want to go up to 10^{10000}). So let's try to reformulate the problem. In problems like this, it is often very useful to calculate some of the values by hand, or if we already have a slow solution, we can use it to calculate the flags for the some n's. When we run our program on small inputs, we see that the flags are up for $1, 4, 9, 16, 25, 36, 49 \ldots$ This looks interesting: these numbers are all square numbers! And it's not only that all these numbers are square numbers, but for each square, the flag is up! Let's try to prove this!

The first thing we see, is that the divisors of n themselves have no influence on the flag, it's only the number of divisors that has to be odd for the flag to be up. Now for each divisor d of n, we know that n/d is also a divisor of n. So we can group the divisors in pairs. But if n is a square, then $d = \sqrt{n}$ is in the pair with $n/d = d$, so it's alone. This means, that if n is a square, then the number of divisors is odd. Otherwise, it's even. So a better algorithm would be to check if a number is a square, by calculating the squareroot and "resquare" it:

```
#include <soi>

signed main() {
    int n = read_int();

    int sqroot = floor(sqrt(n));

    if (sqroot*sqroot == n)
    {
        print("....o\n####|\n####|\n####|\n....|\n....|\n....|\n....|\n....|\n....|");
    }
    else
    {
        print("....o\n....|\n....|\n....|\n....|\n....|\n####|\n####|\n####|\n....|");
    }
}
```

This approach runs in $O(\log(n))$ and gives 90 points. Now I'm going to explain a 100 points solution. This solution is mathematically a bit more difficult, if you're new, it may be too difficult. But if you want, you can give it a try.

Since the number is very big, we can't read it in as `int` anymore. We should read it in as a `string`. But the problem is, we can't really perform that many useful operations on this string, so it's very difficult to calculate with it. But luckily, there's modular arithmetic! Calculating this umber modulo a other number is very easy. We just loop over the string, when we look at a digit, just multiply the current number by 10, add the digit, and then use the modulo operator. So we can calculate the number modulo an other number to get an integer! But what does the residue of this number help us? We know, that not each residue class of a number is a square. In fact, if we look modulo a number p, there are at most $\lceil p/2 \rceil$ residue classes modulo p (easy proof, try to prove it if you don't see it). And Euler gives us a very useful criterion to check wheter residue a modulo an odd prime p can be the residue of a square or not: if $a^{(p-1)/2} \equiv 1$ modulo p, then a can be a square (it doesn't have to be). If it is $\equiv -1$, then a can not be the residue class of a square. We can calculate $a^{(p-1)/2}$ naively in $O(p)$, but if we use fastpow, we can calculate it in $O(\log(p))$. The thing is, if we just do this for one prime, we can't be sure: if n is not a square, it has a chance of roughly $1/2$ that it's also a squared residue modulo p, even if it's not a square. So we have to look at many p's. But how to choose them? At this point, randomizing joins the party! We can just take m different random primes. We have to choose m such that our program is fast enough, but m has to be big enough that the chance that n is not a square even if we claim it, is small. If we look at our runtime, we get $O(m \log(n) + m \log(\max(p)))$, and the chance that our program claims the wrong thing is roughly $(1/2)^m$. Here is the code (which gets 100 points):

```cpp
#include <vector>
#include <iostream>
#include <set>
#include <cmath>
#include <random>
#include <ctime>

#define int int64_t
#define runs 100

using namespace std;

struct solver
{
    string ip;
    int iplen;
    mt19937 rng;
    uniform_int_distribution<int> uni;

    int fastpowmod(int base, int exp, int mod) // Fastpow modulo an integer
    {
        int res = 1;
        int b = base;
        int cr = 1;
        for (int i = 0; i <= ceil(log2(exp)); i++)
        {
            if (exp & cr) res *= b;
            cr <<= 1;
            b *= b;
            b %= mod;
            res %= mod;
        }
        return res;
    }

    int calcresidue(int mod) // Calculate the residue of n modulo a number
    {
        int curr = 0;
        for (int i = 0; i < iplen; i++)
```

```
        {
            curr *= 10;
            curr += (ip[i] - 48);
            curr %= mod;
        }
        return curr;
    }

    void read_ip() // Read the number
    {
        cin >> ip;
        iplen = ip.size();
    }

    void solve()
    {
        rng = mt19937(time(0));
        uni = uniform_int_distribution<int>(0, 1e12);
        read_ip();
        vector<int> primes;
        for (int i = 3; i < 1e5; i++) // Generate a list with primes
        {
            bool pr = true;
            for (int j = 2; j <= sqrt(i); j++)
            {
                if (i % j == 0)
                {
                    pr = false;
                    break;
                }
            }
            if (pr) primes.push_back(i);
        }
        bool sq = true;
        for (int r = 0; r < runs; r++) // Run it "runs" times
        {
            int cpr = uni(rng) % ((int)primes.size());
            int prime = primes[cpr];
            int calcres = calcresidue(prime);
            int crit = fastpowmod(calcres, (prime - 1)/2, prime);
            if (crit == (prime - 1))
            {
                sq = false;
            }
        }
        if (sq) // Print the output
        {
            cout << "....o\n####|\n####|\n####|\n....|\n....|\n....|\n....|\n....|\n....|\n";
        }
        else
        {
            cout << "....o\n....|\n....|\n....|\n....|\n....|\n####|\n####|\n####|\n....|\n";
        }
    }
};

signed main()
{
    solver s;
    s.solve();
}
```