First Round SOI 2017/2018

Solution Booklet

Swiss Olympiad in Informatics

October 1 – November 30, 2017

1 Sushi

Given $N > 0$ pieces of sushi with prices $p_1, p_2, \ldots, p_N$, determine the minimum price to buy all of them if you can use a *special* offer arbitrary number of times.

A *special* offer allows you to buy exactly two pieces of sushi and pay only the more expensive one plus $S \geq 0$.

Subtask 1: Happy hour (10 points)

In this case, we have $N = 2$ and $S = 0$. We are never worse off by using the special offer and we have to pay $\max\{p_1, p_2\}$.

Subtask 2: More Sushi (30 points)

Now, N can be between 1 and 10^3, but S stays zero. This means that we are going to use the special offer for all but at most one piece of sushi.

If N is even, we will divide all pieces into pairs and use the special offer for all of them. After we sort the prices in descending order $p_1 \geq p_2 \geq \cdots \geq p_N$, we will take the pairs $(p_1, p_2), (p_3, p_4), \ldots, (p_{N-1}, p_N)$ and pay $p_1 + p_3 + \cdots + p_{N-1}$.

If N is odd, we cannot pair all the pieces. We buy a cheapest piece of sushi separately and pair the remaining pieces just like in the case N is even.

Please refer to the solution of the last subtask to see why this greedy algorithm works.

Subtask 3: The happy hour is over (10 points)

In this subtask, S is arbitrary, but $N = 2$. We have two options – buy the two pieces separately and pay $p_1 + p_2$, or use the special offer and pay $\max\{p_1, p_2\} + S$. We choose the option providing the minimum price.

Subtask 4: Multiple delegations (50 points)

In the last subtask, we have to deal with arbitrary S and N up to 10^4.

We first sort the prices in descending order $p_1 \geq p_2 \geq \cdots \geq p_N$. Then we take the pairs (p_{2i-1}, p_{2i}) for $i \in \{1, 2, \ldots, \lfloor \frac{N}{2} \rfloor\}$ as long as $p_{2i} > S$ (and pay $p_{2i-1} + S$ for each such pair). Finally, we buy the remaining pieces separately.

Let us now prove the correctness of the described greedy approach. Let $S_> = \{p_i \mid p_i > S\}$ and $S_\leq = \{p_i \mid p_i \leq S\}$.

Let T denote the set of pieces for which we use the special offer (in particular, T is of even size).

Lemma. There exists an optimal solution in which T does not contain any piece from $S_\leq$, i.e., any piece with price at most S.

Proof. As long as T contains a piece $p_i \leq S$, which is paired with a piece p_j, we can remove both pieces from T (i.e., buy them separately) without increasing the overall price. This is because we

paid $\max\{p_i, p_j\} + S \geq \max\{p_i, p_j\} + \min\{p_i, p_j\} = p_i + p_j$, i.e., at least as much as after removal of p_i, p_j from T. The inequality holds because $\min\{p_i, p_j\} \leq p_i \leq S$.

Lemma. In each optimal solution T must contain all but at most one piece from $S_>$.

Proof. To arrive at a contradiction, suppose that T does not contain pieces $p_i, p_j \in S_>$, i.e., pieces with $p_i, p_j > S$. Then we bought the two pieces p_i, p_j separately and paid $p_i + p_j$. If we add the pieces p_i, p_j to T and pair them together, we would pay $\max\{p_i, p_j\} + S < \max\{p_i, p_j\} + \min\{p_i, p_j\} = p_i + p_j$ — contradiction to the optimality of the original solution.

Recall that T is of even size, hence $T = S_>$ in some optimal solution if $|S_>|$ is even.

Lemma. If $|S_>|$ is odd, we can take $T = S_> \setminus \{p_i\}$, where p_i is a cheapest piece in $S_>$.

Proof. Suppose $T = S_> \setminus \{p_k\}$ for some $p_k \geq p_i$. Then p_i is paired with some $p_{i'}$ in T and we pay $p_{i'} + S + p_k$ for the pair $(p_i, p_{i'})$ and p_k that we buy separately.

But we could have instead paid

$$\max\{p_{i'}, p_k\} + S + p_i \leq \max\{p_{i'}, p_k\} + S + \min\{p_{i'}, p_k\} = p_{i'} + S + p_k$$

if we had bought the pair $(p_{i'}, p_k)$ and p_i separately, i.e., by choosing $T = S_> \setminus \{p_i\}$.

Now that we know what pieces we consider for the special offer, we need to figure out how to pair them together.

Let $p_1 \geq p_2 \geq \cdots \geq p_{2m}$ be the prices of the pieces in T (with $T = 2m$).

Lemma. We can take the pairs $(p_1, p_2), (p_3, p_4), \ldots, (p_{2m-1}, p_{2m})$ in an optimal solution.

Proof. To arrive at a contradictoin, consider an optimal solution which differs at the latest from our proposed solution, that is which differs at the pair (p_{2k-1}, p_{2k}) for the maximum possible $k \geq 1$ and contains all the previous pairs.

This solution then pairs $(p_{2k-1}, p_l), (p_{2k}, p_{l'})$ for some $l, l' > 2k$, with the cost of $p_{2k-1} + p_{2k} + 2S$.

If we had paired $(p_{2k-1}, p_{2k}), (p_l, p_{l'})$ instead in that solution, we would have paid $p_{2k-1} + \max\{p_l, p_{l'}\} + 2S \leq p_{2k-1} + p_{2k} + 2S$ and obtained a solution which does not differ at the pair (p_{2k-1}, p_{2k}) — contradiction to the maximality of k.

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

int T;

int N, S;
int p[10005];

int price;

int main()
{
        scanf("%d", &T);
        for(int t = 0; t < T; t++) {
                scanf("%d%d", &N, &S);
                for(int i = 0; i < N; i++) {
                        scanf("%d", &p[i]);
                }

                sort(p, p + N, greater<int>());

```

```
22                  int cur = 0;
23                  price = 0;
24                  while(cur + 1 < N && p[cur + 1] > S) {
25                          price += p[cur] + S;
26                          cur += 2;
27                  }
28                  while(cur < N) {
29                          price += p[cur++];
30                  }
31
32                  printf("Case #%d: %d\n", t, price);
33          }
34
35          return 0;
36  }
```

2 Wagashi

Mouse Stofl promised the participants one wagashi (traditional Japanese candy) for each training task they solve. Every task has an assigned wagashi type. Your task is to find the amount of money required to buy enough wagashi of each of the different types.

Subtask 1: Order single wagashi (20 points)

There are N types of wagashi, and Stofl needs to buy a_i wagashi of type i. A wagashi of type i costs c_i Yen.

This subtask is super easy, to solve it simply calculate $a_i \cdot c_i$ for each i and output the sum of the results. This runs in $O(N)$ time.

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

int main () {
  int T, N;
  cin >> T;
  for (int t = 0; t < T; t++) {
    cin >> N;
    vector<int> a(N);
    vector<int> c(N);
    for (int i = 0; i < N; i++) cin >> a[i];
    for (int i = 0; i < N; i++) cin >> c[i];
    int result = 0;
    for (int i = 0; i < N; i++) result += a[i] * c[i];
    cout << "Case #" << t << ": " << result << "\n";
  }
}
```

Subtask 2: The wagashi package (30 points)

In addition to single wagashi, the wagashi store also offers the *wagashi package*, which contains p_i wagashi of type i and costs K Yen. Stofl can possibly save money by buying some number of wagashi packages, instead of everything as single wagashi.

The numbers of wagashi in this subtask are small; to find the optimal number of wagashi packages, you can just try every number from 0 to some upper limit. A good value for this upper limit is the maximum of all a_i. For every number of packages, you can calculate the total cost by calculating the number of single wagashi you still need of each type multiplied by the single wagashi cost, taking the sum and adding K times the number of packages. Finally you find the minimum cost over all numbers of wagashi packages. This algorithm runs in $O(N \cdot M)$ time, where $M = \max\{a_i\}$.

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

int main () {
```

```
  int T;
  cin >> T;
  for (int t = 0; t < T; t++) {
    ll N, K;
    cin >> N >> K;
    vector<ll> a(N);
    vector<ll> c(N);
    vector<ll> p(N);
    for (int i = 0; i < N; i++) cin >> a[i];
    for (int i = 0; i < N; i++) cin >> c[i];
    for (int i = 0; i < N; i++) cin >> p[i];

    ll maxA = 0;
    for (int i = 0; i < N; i++) maxA = max(maxA, a[i]);

    ll result = numeric_limits<ll>::max();
    for (ll packages = 0; packages <= maxA; packages++) {
      ll cost = packages * K;
      for (int i = 0; i < N; i++) {
        cost += max(a[i] - packages * p[i], 0ll) * c[i];
      }
      result = min(result, cost);
    }
    cout << "Case #" << t << ": " << result << "\n";
  }
}
```

Subtask 3: Ordering wagashi from relatives (25 points)

Now Stofl needs a lot more wagashi. To reduce costs, he orders them from his Japanese relatives. That means he only has to pay the shipping cost, which is the same for each type. He still can buy wagashi packages. The brute force approach from before is too slow for this many wagashi, optimizations are necessary.

You have to find a way to calculate the ideal number of packages more efficiently. If you draw a graph of the total cost at a given number of packages, the points will form a U-shape. The goal is to find the lowest point. For these kinds of problems, the binary search algorithm is usually the right answer. In binary search, you divide the search range in two in each step, look at the middle element and then decide to continue in either the upper or lower half. In this case, you calculate whether costs increase or decrease when taking one additional package, and continue in the upper or lower half accordingly. In other words, you are taking the derivative and finding the zero-intersection. Binary search over N values takes $O(\log N)$ time, so in total this algorithm runs in $O(N \cdot \log M)$.

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

ll N, K;
vector<ll> a;
vector<ll> c;
vector<ll> p;

```

```
ll calcCost (ll packages) {
  ll cost = packages * K;
  for (int i = 0; i < N; i++) {
    cost += max(a[i] - packages * p[i], 0ll) * c[i];
  }
  return cost;
}

int main () {
  int T;
  cin >> T;
  for (int t = 0; t < T; t++) {
    cin >> N >> K;
    a.resize(N);
    c.resize(N);
    p.resize(N);
    for (int i = 0; i < N; i++) cin >> a[i];
    for (int i = 0; i < N; i++) cin >> c[i];
    for (int i = 0; i < N; i++) cin >> p[i];

    ll maxA = 0;
    for (int i = 0; i < N; i++) maxA = max(maxA, a[i]);

    ll left = 0, right = maxA + 1;
    while (left < right) {
      ll middle = (left + right) / 2;
      if (calcCost(middle) < calcCost(middle + 1)) {
        right = middle;
      } else {
        left = middle + 1;
      }
    }
    cout << "Case #" << t << ": " << calcCost(left) << "\n";
  }
}
```

Subtask 4: Many wagashi with different prices (25 points)

The code from the previous subtask also works when the prices for different types are not the same, and is fast enough for this subtask too.

Slightly faster solution with scanline

However we can improve the speed a bit by using a scanline algorithm. The main idea is that the only *interesting* points (numbers of wagashi packages) are where there is a type of wagashi such that we have enough of this type just from packages if and only if we increase the number of packages by at least one. We only have to look at these points and one step after them, because only there things are happening. For each wagashi type, we can calculate this point. Then we sort this list of events and go through it step by step. This runs in $O(N \cdot \log N)$, which is an improvement but only very small in practice. However, scanline can be very useful in other tasks.

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

pair<ll, ll> operator+= (pair<ll, ll> &a, const pair<ll, ll> &b) {
  a.first += b.first;
  a.second += b.second;
  return a;
}

int main () {
  int T;
  cin >> T;
  for (int t = 0; t < T; t++) {
    ll N, K;
    cin >> N >> K;
    vector<ll> a(N);
    vector<ll> c(N);
    vector<ll> p(N);
    for (int i = 0; i < N; i++) cin >> a[i];
    for (int i = 0; i < N; i++) cin >> c[i];
    for (int i = 0; i < N; i++) cin >> p[i];

    ll cost = 0;
    ll slope = K; // slope: How much more/less do we spend with 1 additional package?
    map<ll, pair<ll, ll> > events;
    for (int i = 0; i < N; i++) {
      a[i] *= c[i];
      p[i] *= c[i];
      events[a[i] / p[i]] += make_pair(p[i], a[i] % p[i]);
      cost += a[i];
      slope -= p[i];
    }
    ll result = cost;
    for (auto it = events.begin(); it != events.end(); it++) {
      // Fast forward to just before the next interesting point
      cost += slope * it->first;
      slope += it->second.first;
      result = min(result, cost);
      // One package more, now there are more types of wagashi of which we have too many
      cost -= it->second.second;
      result = min(result, cost + slope);
      cost -= slope * it->first;
    }
    cout << "Case #" << t << ": " << result << "\n";
  }
}
```

3 Cheesepatrol

In this task, Mouse Stofl exports cheese to Japan, where he can sell it for profit. However, recently his revenue in Tokyo has decreased. Stofl thinks that someone is trading cheese in the streets. To solve this problem, he wants to hire private investigators.

A standard graph with m streets and n intersections are given. The i-th street connects intersections a_i and b_i. Two intersections are never connected by two streets and every street connects two different intersections.

A private investigator can either stand along a street and oversee it, or he can stand at an intersection and look at two streets at that intersection, one with each eye.

Mouse Stofl now wants that each street is under surveillance from at least one private investigator, and wants to hire as few investigators as possible.

The goal is to help Stofl to determine the minimal number of investigators and how to place them.

In subtask 1, the graph is a star, in subtask 2, it is a tree and in subtask 3, it is a general graph. Subtask 4 is about describing the idea and proving why the solutions for subtasks 1-3 are correct.

General Idea

The first main observation for this task is to see that if we place a detective on a street, then we can place him to an intersection connecting the street and as a result, the original street and another street can be covered. So its a good idea to basically only place detectives on intersections.

The second is that it is enough for each street to be covered by one detective only, i. e. it is not needed to use two or more detectives for one street.

Actually, it can and will be proven below, that for optimal result, for each connected component i with m_i streets, it is optimal to have $m_i/2$ detectives if m_i is even, and $(m_i + 1)/2$ if m_i is odd. I. e., if m_i is even, we can place $m_i/2$ detectives on exactly $m_i/2$ (not necessesarily distinct) intersections such that each detective covers two streets in such a way that each street is covered by exactly one detective. If m_i is odd, then $(m_i - 1)/2$ detectives cover $m_i - 1$ streets as in the even case and we have one detective who can guard the remaining street or cover two streets: the remaining one and another one which is already covered by another detective.

The main challenge for subtasks 1, 2 and 3 is to find such a distribution of detectives.

Subtask 1: Star (10 points)

In a star graph, if m is even, we can put each detective on the intersection in the middle and assign each of them two streets such that each street is covered by exactly one detective. In the odd case, $m - 1$ streets can be covered as in the even case. The remaining street can be either covered by one detective directly or by putting him on one intersection it is connected to covering the remaining street and potentially another one.

There are many ways to implement this, the probably most straightforward is to assign streets $0, 1$ to the first detective, $2, 3$ to the second and so on until $m - 2, m - 1$ to the last one, if m is even, or street $m - 1$ if m is odd.

Note that this solution is independant of the concrete graph, all we need is the value of m (or n since $m = n + 1$ in this subtask). The running time is $O(n) = O(m)$ since we need to output m

numbers. The memory is in $O(1)$ as we basically only need to save m.

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


#define max_n 10000
#define max_m 100000000

using namespace std;



int solution; //minimal number of policeman necessary




int main(int argc, const char * argv[]) {

    int testcases = 0;
    cin >> testcases;
    for(int t = 0; t < testcases; t++){


        //reading the input - creating the graph
        int n; //number of vertices
        int m; //number of edges
        cin >> n >> m;
        for(int i = 0; i < m; i++){
            int a, b;
            cin >> a >> b;
            //note that this information is not relevant in this subtask
        }

        //calculating optimal value
        solution = (m+1)/2;

        //printing case number
        cout << "Case #" << t << ": ";

        //printing solution
        cout << solution << endl;

        //printing the edges
        for(int i = 0; i < m; i=i+2){
            if(m - i > 1){
                cout << i << " " << i+1 << endl;
            }else if(m - 1 == i){
                cout << i << endl;
            }
        }


    }
```

```
55
56
57      return 0;
58  }
```

Subtask 2: Tree (20 points)

For this subtask, we need the observation that a tree with more than 1 vertice has at least one leaf a (a vertex which is connected to exactly one another vertex). We call the intersection connected to it b and the street connecting a and b is ab. We can now assign a detective to cover ab by being on the vertex b. Note that this is optimal for this street since the street connected to this intercection needs to be covered, and there are only three cases: the detective can be on a, ab or b. In the first two cases, only ab can be covered while in the last case, the detective can cover ab and an additional incident street to ab.

Then we can save that there is a detective on b which can be assigned later. Afterwards, we transform the graph in such a way that we ignore a and ab. Now we have another tree and can recursively find a leaf and do what we did before until there is no street left. The only difference is that we need to check whether there is already a detective on the leaf which can be assigned another street in which case we do exactly that since otherwise we wouldn't make the best use of that detective.

Note that this method has the invariant that in each step, we have one intersection less and that the remaining graph is a tree. Therefore, we must have 2 intersections at one point (our "base case") which are connected by exactly one street. Then with the recursive step, the last street is covered as well. At this point, if there is a detective on exactly one intersection, we have to take that interception. With this method, it is guaranteed that each detective covers exactly two streets in such a way that each street is covered by exactly one detective since each detective who is on an intersection will at some point be at a leaf and cover another street, except the last one who only covers one street if m is odd and obviously, one street is only covered by one detecive.

We can implement this idea by doing a dfs (depth-first-search), then recursively finding leafs and placing detectives the way described above. Note that finding leafs can be done in constant additional time by constantly saving them with a boolean array of size n during the dfs algorithm and the following recursion.

So overall, we get that this solution works in $O(n)$ as we can implement the dfs for a tree in $O(n)$ and the recursion consists of $O(n)$ steps, each of which can be implemented in $O(n)$. The memory is in $O(n)$ for the boolean array and for saving the detectives.

We have only added the implementation for subtask 3 as the algorithm for subtask 2 is similar and a special case of the one of subtask 3.

Subtask 3: General Graph (20 points)

This subtask can be done similarly to subtask 2.

The first major difference is that the graph can have multiple connected components, which actually does not make the main idea more difficult: we simply need to implement our algorithm for each connected component seperately.

The second difference is that the connected components don't have to be trees. We can, however, still construct a dfs tree and solve the problem similarly as in subtask 2 by starting with with a leaf

(specific intersection) a with a_m streets having an end in it. If a_m is even, we put $a_m/2$ detectives on it, covering all the streets and transform the graph such that we ignore a and all streets covered by detectives. If a_m is odd, we cover $(a_m - 1)/2$ streets with detectives. The remaining street can be solved recursively the same way as in subtask 2 as a is now a real leaf in the graph ignoring the $(a_m - 1)$ streets covered with detectives.

For each connected component i with m_i streets, we use $m_i/2$ detectives if m_i is even, and $(m_i + 1)/2$ if m_i is odd which can be analgously argued as in subtask 2.

This algorithm can be implemented similarly as in subtask 2 by finding a dfs tree and recursively putting detectives as described above. Saving where detectives are is a potential challenge, see the master solution for a possible implementation of it.

This algorithm works in $O(n^2)$ as the dfs algorithm for a general graph is in $O(n \log n)$, but each detective has to be assigned and since it is a general graph, it can have $O(n^2)$ streets. Finding the leafs can be achieved in $O(1)$ by saving each leave in a vector, for example. The memory is in $O(n^2)$ as well, for saving the detectives and the graph.

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


#define max_n 10000
#define max_m 100000000

using namespace std;

vector<pair<int, int> > adjl[max_n]; //graph representation as adjacency list
vector<pair<int, int> > matches; //pairs of edges that are matched
set<pair<int, int> > seen; //flag whether edge already seen
bool vis[max_n]; //whether v was already visited
int solution; //minimal number of detectives necessary
bool matched[max_m]; //whether edge matched
vector<pair<int, int> > edges;//saves for each edge its incident vertices

int dfs(int v, int p, int e);//function prototype

int main(int argc, const char * argv[]) {

    int testcases = 0;
    cin >> testcases;
    for(int t = 0; t < testcases; t++){


        //reading the input - creating the graph
        int n; //number of vertices
        int m; //number of edges
        cin >> n >> m;
        for(int i = 0; i < m; i++){
            int a, b;
            cin >> a >> b;
            adjl[a].push_back(make_pair(b, i));
            adjl[b].push_back(make_pair(a, i));
            edges.push_back(make_pair(a, b));
        }
```

```
39
40
41          //matching vertices with the dfs algorithm
42          for(int i = 0; i < n; i++){
43              if(!vis[i]){
44                  seen.insert(make_pair(i, i));
45                  dfs(i, i, -1);
46              }
47          }
48
49
50
51          //calculating solution
52          solution = m - (int) matches.size();
53
54
55          //printing case number
56          cout << "Case #" << t << ": ";
57
58          //printing optial number
59          cout << solution << endl;
60
61          //printing the edges which are matched
62          for(int i = 0; i < matches.size(); i++){
63              int e1 = matches[i].first;
64              int e2 = matches[i].second;
65              cout << e1 << " " << e2 << endl;
66          }
67
68          //printing unmatched edges
69          for(int i = 0; i < m; i++){
70              if(!matched[i]){
71                  cout << i << endl;
72              }
73          }
74
75          //clearing the reuasable data after testcase is finished
76          for(int i = 0; i < n; i++){
77              vis[i] = false;
78              adjl[i].clear();
79          }
80          for(int i = 0; i < m; i++){
81              matched[i] = false;
82          }
83          seen.clear();
84          matches.clear();
85          edges.clear();
86      }
87
88
89      return 0;
90  }
91
92
93  //dfs algorithm for finding an optimal solution
94  int dfs(int v, int p, int e){
```

```
    vector<int> unmatched;
    if((seen.find(make_pair(p, v)) != seen.end()) || (seen.find(make_pair(v, p)) != seen.end())){
        //do nothing
    }else{
        unmatched.push_back(e);
    }
    seen.insert(make_pair(v, p));
    if(vis[v]){
        if(unmatched.size() >= 1){
            return unmatched[0];
        }else{
            return -1;
        }
    }
    vis[v] = true;

    for(int i = 0; i < adjl[v].size(); i++){
        int w = adjl[v][i].first;
        int e1 = adjl[v][i].second;
        int temp = dfs(w, v, e1);
        if(temp != -1){
            unmatched.push_back(temp);
        }
    }

    while(unmatched.size() >= 2){
        matches.push_back(make_pair(unmatched[unmatched.size()-1], unmatched[unmatched.size()-2]));
        matched[unmatched[unmatched.size()-1]]=true;
        matched[unmatched[unmatched.size()-2]]=true;
        unmatched.pop_back();
        unmatched.pop_back();
    }

    if(unmatched.size() >= 1){
        return unmatched[0];
    }else{
        return -1;
    }
}
```

Subtask 4: Description and Proof (50 points)

Now we prove that each connected component i with m_i streets needs at least $m_i/2$ detectives if m_i is even and at least $(m_i + 1)/2$ if m_i is odd. First of all, each detective can at most cover 2 streets which are in the same connecte component. There are m_i streets, each of which need to be covered by at least one detective. Therefore, if m_i is even, at least $m_i/2$ detectives are needed because $m_i/2 - 1$ or less detectives can at most cover $m_i - 2$ streets wile $m_i/2$ could cover m_i streets, each covering two streets accordingly. Similarly, if m_i is odd, at least $(m_i + 1)/2$ detectives are needed.

For the description of the algorithm and what value they calculate, see the respective subtasks above.

With our algorithms for the given subtasks, we have already proven that we can actually get exactly these values so they are optimal and our algorithm calculates the correct value.

The grading scheme for this subtask was the following: There are 25 points for describing the

algorithm and proving that it calculates the right value. The other 25 points can be achieved by proving that the calculated number is optimal indeed. The proof does not have to be formal, but all the ideas of a complete proof have to be included to get a full score.

If, for example, only subtask 1 or 2 were proven, then the points were adjusted accordingly.

4 Hanabi

In this task you were given n cities connected by m bidirectional roads together with the times needed to travel between pairs of cities using these roads. In addition you were given a description of k fireworks that take place in the cities. Your goal was to schedule a plan to attend as many fireworks as possible while traveling between cities accordingly.

Subtask 1: Two cities (15 points)

In this subtask the small limits enabled various inefficient algorithms, as the emphasis was on the correctness of your algorithm. We won't provide any particular sample solution in this case.

Subtask 2: Two fireworks (15 points)

In this subtask there were exactly two fireworks in two distinct cities. To decide whether one can attend both we can use Dijkstra's algorithm to calculate the shortest path between the two cities. The implementation below has running time $O(m \log m)$. (With a set datastructure and a simple trick it can get improved to $O(m \log n)$.)

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

#define INF (1LL<<62)

using namespace std;

struct F {
  long long start_time;
  long long duration;
  int city;
};

int main(void) {
  int T, N, M, K, G;
  cin >> T;
  for(int t=0; t<T; ++t) {
    cin >> N >> M >> K >> G;

    vector<vector<pair<int, long long> > > edges(N);// (city2, distance)
    vector<long long> distance(N, INF);
    vector<int> S(N, -1);                          // S[i] = j <-> the city i is visited from j

    // read roads between cities
    for(int m=0; m<M; ++m) {
      int a,b; long long dist;
      cin >> a >> b >> dist;
      edges[a].push_back({b, dist});
      edges[b].push_back({a, dist});
```

```
32      }
33
34      // read fireworks
35      vector<F> fireworks(K);
36      for(int k=0; k<K; ++k) {
37        cin >> fireworks[k].city >> fireworks[k].start_time >> fireworks[k].duration;
38      }
39
40      // start time of fireworks at 0 <= start time of fireworks at 1
41      if (fireworks[0].start_time > fireworks[1].start_time) swap(fireworks[0], fireworks[1]);
42
43      // stores tuples (time in which city is reached, processed city, city from we arrived)
44      priority_queue< pair<long long, pair<int,int> >,  // declare the elements stored in the queue
45                      vector<pair<long long, pair<int,int> > >, // container
46                      greater<pair<long long, pair<int, int> > > > pqueue; // operator
47      pqueue.push( { fireworks[0].start_time + fireworks[0].duration, { fireworks[0].city, -1 } } );
48
49      // Dijkstra's algo
50      while(!pqueue.empty()) {
51        pair<long long, pair<int, int> > p = pqueue.top();  // get the currently processed city
52        pqueue.pop();
53
54        long long dist = p.first;           // distance
55        int city = p.second.first;          // processed city
56        int from_city = p.second.second;    // the city from which we came
57
58        if (dist >= distance[city]) continue;
59
60        distance[city] = dist;          // store the shortest distance
61        S[city] = from_city;            // store the city from which we arrived
62
63        for(int i=0; i<(int)edges[city].size(); ++i)
64          pqueue.push({ dist + edges[city][i].second, { edges[city][i].first, city} });
65      }
66
67      if (distance[fireworks[1].city] <= fireworks[1].start_time) {     // visit both concerts!
68        vector<int> r;
69        int c = fireworks[1].city;
70        while( c != -1) {
71          r.push_back(c);
72          c = S[c];
73        }
74        reverse(r.begin(), r.end());
75        cout << "Case #" << t << ": " << r.size() << endl;
76        for(int i=0; i<(int)r.size(); ++i)
77          cout << r[i] << (i == (int)r.size() - 1 ? "\n" : " ");
78      }
79      else {
80        cout << "Case #" << t << ": 1" << endl;
81        cout << fireworks[0].city << endl;
82      }
83    }
84    return 0;
85  }
```

Subtask 3: Sparse fireworks (30 points)

First we calculate the minimum time to travel between any pair of cities. This can be achieved by n-times executing Dijkstra's algorithm from above or in $O(n^3)$ by Floyd-Warshall's algorithm (which is implemented in the sample source code below).

The solution of this subtask uses a dynamic programming approach. We store value S_i: the maximum number of fireworks one can see when ending by the fireworks i. To be able to reconstruct Stofl's travel plan we also store the fireworks from which we arrived to i.

We first sort the fireworks by increasing start time in $O(k \log k)$ and process them in this order.

For a fireworks i (taking place in city c_i from time s_i to time $s_i + d_i$) we consider all fireworks j (taking place in city c_j from time s_j to time $s_j + d_j$) such that we can attend the fireworks j and arrive to city c_i just in time to see fireworks i. Out of all such fireworks we pick the one with allows us to see the maximum number of fireworks. Formally

$$S_i = 1 + \max_j \left\{ S_j \,\middle|\, s_j + d_j + dist(j,i) \le s_i \right\},$$

where the +1 is for the fireworks i. The running time of the below code is $O(n^3 + k^2)$ and the space needed is $O(n^2 + k)$.

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

#define INF (1LL<<60)

using namespace std;

struct F {
  long long start_time;
  long long duration;
  int city;
  int id;  // the order in which the fireworks is in the input (just for output purposes)

  bool operator<(const F&o) const {
    return start_time < o.start_time;
  }
};

int main(void) {
  int T, N, M, K, G;
  cin >> T;
  for(int t=0; t<T; ++t) {
    cin >> N >> M >> K >> G;

    vector<vector<long long> > edges(N, vector<long long>(N, INF)); // adjacency matrix
    // read roads between cities
    for(int m=0; m<M; ++m) {
      int a, b; long long dist;
      cin >> a >> b >> dist;
      edges[a][b] = edges[b][a] = dist;
    }
    for(int k=0; k<N; ++k) edges[k][k] = 0;

```

```
35      // Floyd-Warshall's algo
36      for(int k=0; k<N; ++k)
37        for(int i=0; i<N; ++i)
38          for(int j=0; j<N; ++j)
39            edges[i][j] = min(edges[i][j], edges[i][k] + edges[k][j]);
40
41      // read fireworks
42      vector<F> fireworks(K);
43      for(int k=0; k<K; ++k) {
44        cin >> fireworks[k].city >> fireworks[k].start_time >> fireworks[k].duration;
45        fireworks[k].id    = k;
46      }
47      sort(fireworks.begin(), fireworks.end());
48
49      // S[i] = (c, j) <=> the city i is visited from j and c fireworks can be viewed
50      vector<pair<int, int> > S(K, {1, -1});
51      for(int k=0; k<K; ++k) {
52        for(int i=0; i<k; ++i) {
53          long long end_time = fireworks[i].start_time + fireworks[i].duration;
54          long long dist = edges[fireworks[i].city][fireworks[k].city];
55          if (end_time + dist <= fireworks[k].start_time) {
56            if ( S[k].first < S[i].first + 1) {
57              S[k] = { S[i].first + 1, i };
58            }
59          }
60        }
61      }
62
63      int best_count = 0;
64      int best_city = -1;
65      for(int k=0; k<K; ++k) {
66        if (best_count < S[k].first) {
67          best_count = S[k].first;
68          best_city = k;
69        }
70      }
71
72      cout << "Case #" << t << ": " << best_count << endl;
73      if (G == 1) {
74        vector<int> r;
75        int f = best_city;
76        while( f != -1) {
77          r.push_back(f);
78          f = S[f].second;
79        }
80        reverse(r.begin(), r.end());
81        for(int i=0; i < (int)r.size(); ++i)
82          cout << fireworks[r[i]].id << (i == (int)r.size() - 1 ? "\n" : " ");
83      }
84    }
85    return 0;
86  }
```

Subtask 4: Many fireworks (40 points)

To further speed up the above solution, we need the following observation about monotonicity. Let i and j be two fireworks taking place in the same city, having $s_i < s_j$. We will prove that in that case $S_i \leq S_j$. The reason for this is simple, if we attend $S_i - 1$ fireworks before i, we can always skip the fireworks i and wait for j instead. Thus the case with j can be no worse than i.

This means that in order to find the best schedule for the given fireworks i, there is only one fireworks display in each city to consider – the one with the latest start time such that the journey to the city of i's fireworks can still be made in time. We can thus for each city store the list of the fireworks ordered by the start time, and use binary search to find the respective fireworks.

The running time of the dynamic programming becomes $O(n \cdot k \cdot \log k)$ which is better than the solution for the third subtask when $n \ll k$.

```
#include<iostream>
#include<algorithm>
#include<vector>

#define INF (1LL<<60)

using namespace std;

struct F {
  long long start_time;
  long long duration;
  int city;
  int id;  // the order in which the fireworks is in the input (just for output purposes)

  bool operator<(const F&o) const {
    return start_time < o.start_time;
  }
};

typedef pair<pair<long long,int>, int> PP;              // (end time, count, fireworks)

int main(void) {
  int T, N, M, K, G;
  cin >> T;
  for(int t=0; t<T; ++t) {
    cin >> N >> M >> K >> G;

    vector<vector<long long> > edges(N, vector<long long>(N, INF)); // adjacency matrix
    // read roads between cities
    for(int m=0; m<M; ++m) {
      int a, b; long long dist;
      cin >> a >> b >> dist;
      edges[a][b] = edges[b][a] = dist;
    }
    for(int k=0; k<N; ++k) edges[k][k] = 0;

    // Floyd-Warshall's algo
    for(int k=0; k<N; ++k)
      for(int i=0; i<N; ++i)
        for(int j=0; j<N; ++j)
          edges[i][j] = min(edges[i][j], edges[i][k] + edges[k][j]);
```

```
42
43      // read fireworks
44      vector<F> fireworks(K);
45      for(int k=0; k<K; ++k) {
46        cin >> fireworks[k].city >> fireworks[k].start_time >> fireworks[k].duration;
47        fireworks[k].id = k;
48      }
49      sort(fireworks.begin(), fireworks.end());
50
51      // MAIN ALGORITHM
52      vector<int> P(K, -1);       // The fireworks from which we arrived
53      vector<vector<PP> > S(N);  // (end time, # fireworks, fireworks id)
54      for(int k=0; k<K; ++k) {
55        int best_count = 0;
56        int current_city = fireworks[k].city;
57        for(int n=0; n<N; ++n) {
58          long long t = fireworks[k].start_time - edges[n][current_city];
59          auto it = upper_bound(S[n].begin(), S[n].end(), PP{{t, K+1}, K});
60          if (it == S[n].begin()) continue;   // no record found
61          --it;
62          if( best_count < it->first.second ) {
63            best_count = it->first.second;
64            P[k] = it->second;
65          }
66        }
67        S[current_city].push_back({{fireworks[k].start_time + fireworks[k].duration, best_count + 1}, k} );
68      }
69      // END OF MAIN ALGORITHM
70
71      int best_count = 0;
72      int best_city = -1;
73      for(int n=0; n<N; ++n) {
74        if (!S[n].empty() && best_count < S[n].back().first.second) {
75          best_count = S[n].back().first.second;
76          best_city = n;
77        }
78      }
79
80      cout << "Case #" << t << ": " << best_count << endl;
81      if (G == 1) {
82        vector<int> r;
83        int f = S[best_city].back().second;
84        while (f != -1) {
85          r.push_back(f);
86          f = P[f];
87        }
88        reverse(r.begin(), r.end());
89        for(int i=0; i < (int)r.size(); ++i)
90          cout << fireworks[r[i]].id << (i == (int)r.size() - 1 ? "\n" : " ");
91      }
92    }
93    return 0;
94  }
```

In fact, we can get rid of the binary search, too. Notice that for each pair of cities i, j, the queries for fireworks in city i from which the firewokrs in j can be watched are only increasing (since the fireworks are ordered by their start time). Thus, a pointer advanced in linear fashion can be used to

handle all queries fo a given pair of cities.

We now need to upper bound the number of advancemenets of these pointers. In order to do that, denote by F_i the number of fireworks in city i. For a pointer $p_{i,j}$, the number of advancements can be at most $F_i + 1$ (the pointer starts before the first element, and can end no further than after the last). The total cost for all pointers is thus

$$\sum_{i=1}^{n} \sum_{j=i}^{n} F_i + 1 = n(n + \sum_{i=1}^{n} F_i) = n(n + k)\,.$$

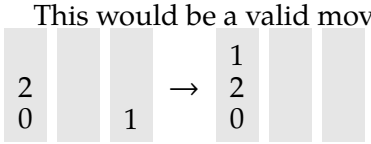
This yields total running time $O(n^3 + k \cdot n)$ and space $O(k + n^2)$.

```
    // MAIN ALGORITHM
    vector<int> P(K, -1);      // The fireworks from which we arrived
    vector<vector<PP> > S(N);  // (end time, # fireworks, fireworks id)
    vector<vector<int> > pointers(N, vector<int>(N, 0));
    for(int k=0; k<K; ++k) {
      int best_count = 0;
      int current_city = fireworks[k].city;
      for(int n=0; n<N; ++n) {
        long long t = fireworks[k].start_time - edges[n][current_city];
        int& p = pointers[n][current_city];          // with & we will be modifying the value in p
        while(p < (int)S[n].size() &&        // not at end of pointer list
              S[n][p].first.first <= t)      // fireworks n ends early enough
          ++p;
        if (p > 0) {     // there was at least 1 fireworks in city n
          if (best_count < S[n][p-1].first.second) { // it is -1 because we want last value <= t
            best_count = S[n][p-1].first.second;
            P[k] = S[n][p-1].second;
          }
        }
      }
      S[current_city].push_back({ {fireworks[k].start_time + fireworks[k].duration, best_count + 1}, k });
    }
    // END OF MAIN ALGORITHM
```

5 Mahjong

Mahjong is about a solo game with three stacks (in the original task description this has been called "pile") and N tiles. The tiles are numbered from 0 to $N-1$. A game consists of a sequence of the following moves: Take the top tile of a stack and to move it on top of another stack.

This would be a valid move:



When all tiles lie on the same stack and are in sorted order (with the 0 on top), the game is over.

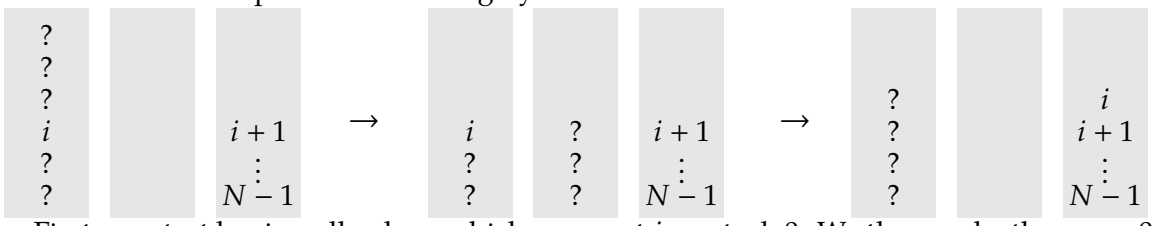
We call a specific configuration of the tiles on the three stacks as a "state". A state where all tiles are on one stack and in sorted order is referred to as "end state".

Subtask 1: Construction of a Solution (25 points)

Given an arrangement of the n tiles, the task is to find a possible sequence of moves that results in one of the three end states.

There is a restriction on the number of moves, but it is large enough that it doesn't really matter unless one tries a solution that goes through all permutations.

Many approaches exist, but the following one leads to a fairly clean code: Use stacks 0 and 1 as "shunting yards" and extract the values $N-1, N-2, N-3 \ldots, 1, 0$ one by one and put them on stack 2. The i-th step would look roughly like this:



First, we start having all values which are most i on stack 0. We then make the move $0 \to 1$ as long as i is not on top of stack 0. Then, we move it to stack 2 with the move $0 \to 2$. Finally, we put everything back on the first stack with the move $1 \to 0$.

Using Python's `yield`[1], this can be done with the following code:

```python
def parse():
    n = int(input())
    s = [list(map(int, input().split()))[1:] for _ in range(3)]
    return n, s

def solve(n, s):
    # Perform the move on s and return the move as tuple
    def move(a, b):
        s[b].append(s[a][-1])
        s[a].pop()
        return a, b

    # Move everything to stack 0
```

[1] You can think of the function as returning a list, and yield as appending to that list

```
    while s[1]:
        yield move(1, 0)
    while s[2]:
        yield move(2, 0)

    # Extract largest number and move it to stack 2
    for i in reversed(range(n)):
        while s[0][-1] != i:
            yield move(0, 1)
        yield move(0, 2)
        while s[1]:
            yield move(1, 0)

def display(moves):
    moves = list(moves)
    return '\n'.join([f'{len(moves)}'] + [f'{a} {b}' for a, b in moves])

for i in range(int(input())):
    print(f'Case #{i}: {display(solve(*parse()))}')
```

Subtask 2: Move Table with Full Information (25 points)

This time you need to solve *any* possible state on N tiles.

There are three basic approaches for this subtask:

1. Any solution for the later subtasks is also a solution for this subtask.
2. Use your solution for subtask 1 and do some ad hoc DFS.
3. Use DFS on the tree of possible moves.

What does this mean? DFS is short depth-first search and you can look it up in on `https://soi.ch/wiki`. Initially, all states are "pending". You mark the end states as "final". Then, you look at some state which is "pending" and decide what to do with it, i.e. what should be its successor state. Then you mark this node as final. Once all nodes are final, you print out your solution.

In general this can produce loops: If from state A you go to B, from B to C, and from C back to A, you have a loop. To avoid this, you either need to be careful when computing the successor state (which would be the approach 2), or you can modify the algorithm as follows.

Again, you mark the end states as "final". Then, you look at all states which can reach another state that is marked "final". For those you know what the successor should be: that final state you can reach. You repeat this until all nodes are final.

Why does this avoid loops? Give each state an index, the step in which you painted it. The end states have indices 0, the first other state you mark final has index 1, the second state you marked final has index 2, etc. You note that for each state it sucessor has a *strictly lower* index (because it was marked final before). When you follow the actions starting at a given state, all states have strictly lower index, but you can not decrease the index infinitely, so after a finite number of steps you are in an end state.

The code below implements this last approach.

```python
def parse():
    return int(input())

def neighbors(s):
    def move(a, b):
        if not s[a]: return a, b, None
        t = [list(x) for x in s]
        t[b].append(t[a][-1])
        t[a].pop()
        return a, b, tuple(map(tuple, t))
    yield move(0, 1)
    yield move(1, 0)
    yield move(0, 2)
    yield move(2, 0)
    yield move(1, 2)
    yield move(2, 1)

def solve(n):
    state = {}  # move table
    stack = []  # dfs stack

    # generate end positions
    for i in range(3):
        s = [()]*3
        s[i] = tuple(reversed(range(n)))
        stack.append(s)
        state[tuple(s)] = None

    # perform dfs
    while stack:
        s = stack.pop()
        for a, b, t in neighbours(s):
            if t is not None and t not in state:
                state[t] = b, a
                stack.append(t)

    return state.items()

def display(moves):
    moves = list(moves)
    def print_state(s): return ' | '.join(' '.join(map(str, reversed(x))) for x in s).strip()
    def print_move(s): return "done" if s is None else f"{s[0]} -> {s[1]}"
    return '\n'.join([print_state(t) + ': ' + print_move(s) for t, s in moves])

for i in range(int(input())):
    print(f'Case #{i}:\n{display(solve(parse()))}')
```

Subtask 3: Move Table with Top Only (20 points)

Let's move on to the next variation of this game. This time, you don't know the complete arrangement of a stack, but just the value of the topmost tile. You need to give your move for each possible arrangement of top tiles.

The limits for this subtask are small enough that you can bruteforce the possible move tables for partial points, or even full points if you do it clever.

This subtask exists so that you can try your ideas for the last subtask.

Subtask 4: Theoretical Move Function (30 points)

You should to define a function f(n, a, b, c), which takes the three topmost numbers a, b, c of the stacks (empty piles are encoded as -1) and returns $0 \to 1$, $0 \to 2$, $1 \to 0$, $1 \to 2$, $2 \to 0$ or $2 \to 1$. There are n tiles in total. Your function needs to be fast.

There are many ways to solve this. The basic idea that is common behind all of the solutions is that they group the states into different "phases", where one phase performs a sorting step and the other phases put everything together and prepare the next sorting step.

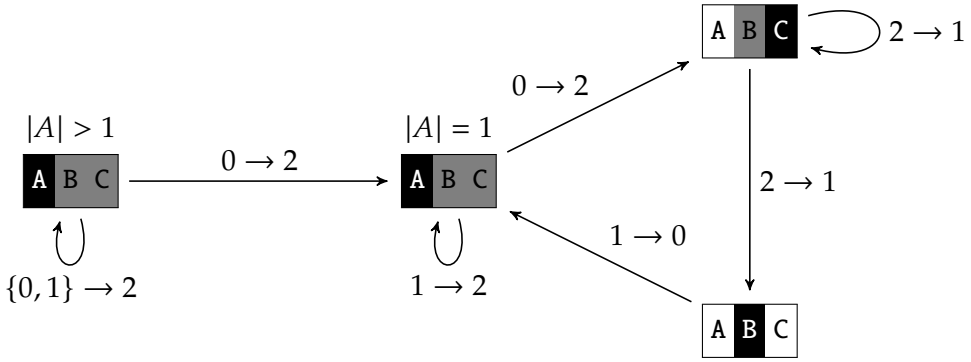
Below we give one possible solution together with an explanation.

```python
def move(a, b, c):
    # _?C -> _??: move everything to B
    if a == -1 and c != -1:
        return (2, 1)
    # _B_ -> A?_: start a new round
    if a == -1:
        return (1, 0)
    # A?? -> ???: sort as long as A is non empty
    if b == -1 or (c != -1 and b > a > c):
        return (0, 2)
    return (1, 2)
```

Below is the phase diagram for this code. Such drawings are known as state machines, where the vertices are states and edges are transitions. Since the term state is already used for a specific configuration of tiles, we use phase instead.

The three stacks are represented by the letters A, B and C. White background means that the stack is empty. Black means that the stack is not empty. Gray background means that we don't care – the stack may or may not be empty.

For each phase, the code identifies in which phase he is in and then decides on the move. The phase may stay the same (which is indicated by a loop) or may transition to a different phase. All possible phase transitions are shown. Note there may more than one transition in the diagram (if the decision is not so easy), but for any given state, there is only one transition.



The condition $|A| > 1$ means that the stack A must have at least two tiles on it.

First, you need to convince yourself that (1) every state is assigned to at least one phase and (2) a state can have at most one phase assigned to him.

If we start at the left-most phase, A B C with $|A| > 1$, we move something from either A or B to C. This can't be repeated infinitely and we inevitably end up in the phase A B C with $|A| = 1$. As there is no other way to reach phase A B C with $|A| > 1$, we can ignore it for the further analysis. If we speak from phase A B C, we from now on assume that we have $|A| = 1$.

Also, it is pretty clear that from phase A B C we reach phase A B C after at most n moves, from A B C we reach A B C after at most n moves and after at most 1 move, we reach A B C.

Thus we can assume we start at A B C. We'll look at how the state changes after transitioning to phases A B C, A B C and A B C again. We call this a "round".

Let's say, the state initially looks like this:

$$A = ()$$
$$B = (x, y, \ldots, z_1, z_2, \ldots), \text{ where } x < y, z_1 < x < z_2$$
$$C = ()$$

Or, to keep it short: $((), (x, y, \ldots, x_-, x_+, \ldots), ())$ – with x_- we mean a number smaller than x, and with x_+ a number larger than x. There may be many adjacent x_- and x_+, so by convention we take the first such pair. After one round, we end up with this: $((), (y, \ldots, x_-, x, x_+, \ldots), ())$. Why? Let's simulate the steps.

- First, we apply $1 \to 0$ and transition to phase A B C. We end up with $(x), (y, \ldots, x_-, x, x_+, \ldots), ()$.
- At first, the condition $b = -1$ or ($c \neq -1$ and $b > a > c$) is always false. So we shuffle the elements on top of stack C in reversed order. $(x), (\ldots, x_-, x_+, \ldots), (\ldots, y)$
- At one point we reach this state: $(x), (x_+, \ldots), (x_-, \ldots, y)$.
- The second condition becomes true and we move x to C. $(), (x_+, \ldots), (x, x_-, \ldots, y)$.
- We're now in phase A B C, where we put everything back again on stack B until we are finished: $(), (y, \ldots, x, x, x_+, \ldots), ()$.

As you can see, we end up with what was claimed above. Intuitively, we put x into the spot it belongs.

To make a proof out of this, the easiest way is to formulate variants. A variant is a number that changes after every round, in a well-defined fashion. Here, we use a variant consisting of two numbers:

- The number adjacent inversions
- The value of the front element

With adjacent inversions we mean adjacent elements a_i, a_{i+1} having $a_i > a_{i+1}$ (as opposed to $a_i < a_{i+1}$, which would be the correct order). Furthermore, we say that $N - 1, 0$ is not an adjacent inversion (to solve a special case below).

Let's see this in action. We start with $((), (x, y, \ldots), ())$ and see what happens after one round.

- For $(x, y, \ldots, x_-, x_+, \ldots)$, if $x > y$, we end up with $(y, \ldots, x_-, x, x_+, \ldots)$. We get rid of the adjacent inversion $x > y$, thus reducing the number of adjacent inversions.
- For $(x, y, \ldots, x_-, x_+, \ldots)$, if $x < y$, we end up with $(y, \ldots, x_-, x, x_+, \ldots)$. The number of adjacent inversions stays the same, but we have increased the value of the front element (from x to y).
- Assume there are no such x_- and x_+. In this case, we perform a cyclic shift on the whole list, i.e. we go from $(x, y, \ldots)$ to $(y, \ldots, x)$ (verify this yourself). This can only happen if:
  - The first element is 0 and $N-1$ is the last element. Note that we can always put 0 after $N-1$ since we defined this as a special case. In case $N-1$ is the last element, this coincidences with a cyclic shift.
  - The first element is $N-1$: This case can't happen, since we would have put it in front of the 0 before.
  - The list is almost in sorted order like this: $(x, x+1, x+2, \ldots, N-1, 0, 1, \ldots, x-1)$. Then we increase the front element while keeping the number of inversions as 0.

  Except for the state $(N-1, 0, 1, \ldots, N-2)$, these are all cases (check that for yourself).

What happens in the state $(N-1, 0, 1, \ldots, N-2)$? According to the algorithm, we'll perform a cyclic shift and end up with $(0, 1, \ldots, N-2, N-1)$. We violate our variant, but this doesn't matter: This is exactly our sorted list, so we are done.

We didn't ask for this in the task description, but how many operations do we need? Our variant can take at most $O(n^2)$ values and one round takes $O(n)$ operations. So from any position, we can reach an end position after $O(n^3)$ steps.

The function provided runs in constant time with only constant additional memory.

6 Samurai

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

We played many games with different configurations to determine the quality of the submitted algorithms. In the first few rounds we determined the best bot per person and then in later rounds only played with the best bot per participant. According to all those games a ranking was calculated and points were awarded according to it. If a bot did not follow the rules or no source code was submitted (i.e. only a compiled binary) some points were deducted.