InfO(1)Cup International Round Editorial

InfO(1)Cup Scientific Committee

February 2021

Problem SubarraySort

Author: Alex-Nicolae Pop

Throughout this editorial we will consider the array P indexed from 0 with elements from 1 to n.

Subtask 1 - random permutation

Because a random permutation does not have any particular structure, it is always optimal to use one operation on the segment $[0, n-1]$. Thus, the answer is $\lfloor \sqrt{n} \rfloor$.

Subtask 2 - $n \leq 9$

The small size of the permutation allows the use of a brute-force approach to solve this subtask. One can use, for example, an algorithm similar to Dijkstra's by maintaining a `std::priority_queue` containing pairs `(cost, permutation)` (by using `std::pair<int, std::vector<int>>`) sorted ascending by cost. At each iteration, we pop the pair at the top of the queue and we try to sort all $O(n^2)$ segments comparing the new candidate cost for the new permutation with the best cost found so far (which can be stored, for example, using `std::map<std::vector<int>, int>`). If a better cost is found, we update the map and push the new pair in the queue.

Useful remarks

In order to solve the following subtasks, the following lemma is required:

Lemma. *There is a set S of operations that sort the permutation with minimum cost such that $\forall x, y \in S, x \neq y, x \cap y = \emptyset$.*

Proof. Let S be a set of operations that sort the permutation with minimum cost. We will show that S can be transformed into a set of operations with cost at least as good and disjoint intervals. Denote by S_i the i-th interval in S. Let's build a graph G with $|S|$ vertices in which we consider the edge $(x, y) \iff S_x \cap S_y \neq \emptyset$.

Sublemma. *Segments in S corresponding to a connected component in G can be replaced by their union, and the permutation is still going to be sorted after applying all the operations.*

Proof. Suppose that after the replacement, the permutation will no longer become sorted. This implies the existence of indexes i, j in P such that $P_i > P_j$. This inversion implies that there is no connected component in G whose segment union contains both position i and position j (if there was such a component, after applying its corresponding operation the inversion would become sorted). But if there is no such connected component, there is no segment in S that contains both i and j, so the inversion (i, j) also exists after applying the operations in S, which contradicts the correctness of S. □

Sublemma. *After replacing the segments with their union, the cost of operations is at least as low as the cost of S.*

Proof. Let $f(x) = \begin{cases} \lfloor \sqrt{x} \rfloor & x > 1 \\ 0 & x = 1 \end{cases}$ be the cost with which a segment of length x can be sorted. Note that $f(x + y - 1) \leq f(x) + f(y), \forall x, y \geq 1$. Thus, if there are 2 segments of lengths x, y that intersect in at least one point, their union has a cost less than or equal to the sum of the individual costs. The inequality holds true for multiple segments because we are able to merge 2 of them repeatedly. □

Thus, using the results of the two sublemmas, we can transform S into a set of disjoint segments (by definition of connected components) with a cost at least as low as the cost of S. □

Let us partition the permutations in minimal, consecutive, disjoint segments $[0, a_1], [a_1 + 1, a_2] \dots [a_k, n - 1]$ with the property that the elements in range $[a_i + 1, a_{i+1}]$ will stay in that range after the sorting. For example, for `P = [2, 1, 3, 6, 5, 4, 8, 7]`, the partition is $[0, 1], [2, 5], [6, 7]$. Note that $[0, 5], [6, 7]$ is not correct because the first segment can be further split in two. Let us call the segments in the partition *compacts*.

It is easy to see that in a correct solution we have to cover every compact with length greater that 1 by a segment.

To compute all the compacts we traverse the array from 0 to $n - 1$ and we maintain the current maximum of P_i in a variable called max_val. If $max\_val = i + 1$, than we end the current compact. This process takes $O(n)$ time and $O(1)$ extra memory.

Subtask 3 - $n \leq 2000$

We will solve the problem using dynamic programming. Let min_cost_i denote the minimum cost to sort the prefix up until position i, where i is a right

end of a compact. Then,

$$min\_cost_i = \min_{\substack{j < i \\ j \text{ is the right end} \\ \text{of a compact}}} (min\_cost_j + f(i-j)).$$

A direct implementation of this formula runs in $O(n^2)$ time and it is enough to solve subtask 3.

Subtask 4 - $n \leq 10^5$

In order to solve this subtask an optimization of the last solution was required. It should run in $O(n\sqrt{n})$time. Let us try to reduce the number of js that we consider in the formula.

Exploiting the boundedness of *min_cost*

The values of *min_cost* are bounded by $\lfloor\sqrt{n}\rfloor$ because we always can take this time by sorting segment $[0, n-1]$.

Solution in $O(n\sqrt{n})$. Note that for equal values of $min\_cost[j]$ we only care about the greatest j of them, because as j increases, $f(i-j)$ decreases due to its monotonicity (it is an increasing function). That means that we can maintain the best j for every value of *min_cost* in an array or by keeping a stack. This allows us to iterate $O(\lfloor\sqrt{i}\rfloor)$ values instead of $O(i)$ for each i and thus making the time complexity $O(n\sqrt{n})$.

Exploiting the boundness of f

It is easy to see from the formula that the values of $f(x)$ are bounded by $\lfloor\sqrt{n}\rfloor$.

Solution in $O(n\sqrt{n})$. Note that *min_cost* has increasing values (we can take an optimal solution for $i+1$ and take out $i+1$ from all segments without increasing the cost and we get a solution for i). Also, all the positions j for which $f(i-j)$ has the same value form a continuous segment so we can iterate this value instead of j and compute its range of positions that give this value. Due to the monotonicity of *min_cost* we only care about the leftmost position in the range that is the right end of a compact. We can precompute this while finding the compacts in linear time.

Solution in $O(n\sqrt{n})$ Say we did not remark that *min_cost* is increasing, we still need to find the position in a range with minimum *min_cost*. Let us assume that we have a data structure that can maintain a set of items (by insertions and deletions) and we can also query it for the minimum. We will maintain all the values for which $f(i-j)$ is equal in one of them and we will query them for minimum. Then, when moving from position i to $i+1$ only $O(\lfloor\sqrt{n}\rfloor)$

positions will move from one structure to another one. The solution will run in $O(T(n)n\sqrt{n})$ where $T(n)$ is equal to the time of insert / delete / query of our data structure.

We could use a queue for the insertions and deletions, but how can we get the minimum? Let us think of how would we implement minimum query on a stack. We can maintain an additional `int` representing the minimum of all the elements below that element. When we push a new element, its minimum is the minimum of it and the minimum of the top element of the stack. Then, the minimum in the stack is just the minimum at the top of the stack. To implement minimum query on queue we can just implement the queue using two stacks that support minimum query. Then, all operations on the queue take amortized $O(1)$ resulting in an $O(n\sqrt{n})$ solution.

Exploiting the boundness of both min_cost and f

Solution in $O(n\sqrt{n})$ We will maintain an array $cost$ such that $cost_j = min\_cost_j + f(i-j)$. Then, $min\_cost_i = \min_{\substack{j < i \\ j \text{ is the right end} \\ \text{of a compact}}} cost_j$. Let us do square root decomposition on $cost$ and maintain for each bucket the minimum value of $cost$ inside it and also how many positions have this value. Thus, to find min_cost_i we just take the minimum of minimum values of all the buckets to the left of i. It is obvious that this takes $O(\sqrt{n})$ time for each i. When we move from i to $i+1$ only $\lfloor\sqrt{i}\rfloor$ values of $cost$ will change (due to changing of $f(i-j)$). If a value that gets changed was equal to the minimum in its corresponding bucket, we decrease its frequency; else we don't do anything. When the frequency of what we knew was the minimum value becomes 0, we recompute it by going through all the positions inside the bucket in $O(\sqrt{n})$. We claim this works in total $O(n\sqrt{n})$ time. Values of $cost$ are bounded by $2 \cdot \lfloor\sqrt{n}\rfloor$ by being the sum of 2 bounded terms. This means that the expensive computation in $O(\sqrt{n})$ can only be done $O(\sqrt{n})$ times per bucket. Since there are $\sqrt{n}$ buckets, this results in $O((\sqrt{n})^3) = O(n\sqrt{n})$ amortized time.

Subtask 5 - $n \leq 4 \cdot 10^6$

We solved the previous subtasks by computing min_cost_i, the minimum cost required to sort the prefix of P up until position i. We will change things up a bit by computing max_pref_{cost}, the longest prefix we can sort with cost equal to $cost$. The answer will be the smallest index $cost$ such that $max\_pref_{cost} = n$. Note that $cost$ is still bounded by $\lfloor\sqrt{n}\rfloor$ so we only care about $O(\sqrt{n})$ states in this approach. By doing the transitions in $O(\sqrt{n})$ we get a complexity of $O(\sqrt{n}^2) = O(n)$. For this, we need to precompute $left_i$, the left end of the compact in which i lies. We will also precompute $free_i$, the maximum numbers of compacts of length 1 starting from position i and going to the right. For example, for `P = [2, 1, 3, 4, 5, 8, 6, 7, 9, 10]`, `left = [0, 0, 2, 3, 4, 5, 5, 5, 8, 9]` and `free = [0, 0, 3, 2, 1, 0, 0, 0, 2, 1]`. Say we

are processing max_pref_{cost}. First, we try to improve it with free positions (compacts of length 1) by doing $max\_pref_{cost} += free_{max\_pref_{cost}}$. Then, we iterate over the next cost of a segment that we choose to sort: $next\_cost$. We use the array $left$ to find out the length of the prefix we can sort with $cost + next\_cost$ and update $max\_pref_{cost+next\_cost}$ by taking the maximum of the current candidate and its current value.

Problem Fiboxor

For this problem, the main observation is that the Fibonacci sequence, modulo $m = 2^k$, is periodic, with period $O(m)$. Let p be the period of the Fibonacci sequence modulo m. Note that the the XOR of any subsequence of the Fibonacci sequence modulo m is the XOR of a number of repetitions of the complete period of the sequence, XOR-ed with a prefix and a suffix of this period. All of these values can be precomputed in $O(m)$, and thus we can answer each query in constant time.

Problem Bricks

Author: Alexa Tudose; *Prepared by: Alexa Tudose, Mihai Popescu*

Subtask 1 - no changes need to be made

In this subtask, we only need to find the interesting bricks in the original configuration. For this, we create the stack of maximums for purple bricks and for red bricks. Let's denote them st_0 and st_1, respectively.

To calculate st_0, we iterate through the purple bricks from left to right, and for each brick i we first remove from the stack all bricks that are shorter than it, and then add it to the stack. Of course, st_1 can be calculated in a similar manner, but consider the red bricks instead of purple bricks.

The number of interesting bricks equals the sum of sizes of the two stacks.

This leads to an $O(N)$ time complexity.

Notations

We will call a brick *lost* if it was interesting in the beginning, but it ceases to be interesting after the change.

Similarly, we will call a brick *gained* if it wasn't interesting in the beginning, but it becomes interesting after the change.

Also, if there is exactly one higher brick than i of the same colour as i and on the right of i, let $p[i]$ be the position of that brick. Otherwise, let $p[i] = -1$. In other words, $p[i]$ tells us which brick should have its colour changed in order for brick i to be gained. Note that $p[i]$ can be calculated easily, e.g. by calculating the maximum and second maximum for all suffixes.

Subtask 2 - all bricks are red

In this subtask, no bricks can be lost. We therefore aim to maximize the number of gained bricks. There are 2 cases:

- change the colour of an interesting brick i

  In this case, all bricks j with $p[j] = i$ are gained. We can keep a frequency table such that $cnt[i]$ is the number of indices j with $p[j] = i$.

- change the colour of a non-interesting brick i

  In this case, only one brick will be gained: brick i.

The answer will be $size(st_0) + size(st_1) + max(maxElement(Cnt), 1)$. This approach can be implemented in $O(N)$.

Subtask 3 - $N \leq 1.000$

For this subtask, we can simulate all of the N possible changes and use the approach from the first subtask to calculate how many interesting bricks we get for each possible change. This gives the time complexity $O(N^2)$

Subtask 4 - $N \leq 200.000$

To solve this subtask, we should go back to the ideas involved in Subtask 2. However, this time we can have not only gained bricks, but also lost bricks. We will try to change the colour of each brick and see how many gained and lost bricks we get in each case.

We will consider from now on that we change the colour of a brick from purple to red. Changing the colour from red to purple is symmetrical and can be treated similarly.

Suppose we change the colour of brick i from purple (false) to red (true). Then:

- Bricks j with $p[j] = i$ are gained. We can againg compute $cnt[i]$, as we previously did in Subtask 2.

- Interesting red bricks j with $j < i$ and $H[j] < H[i]$ are lost. These lost bricks are actually $st_1[k]$, $st_1[k+1]$, $st_1[k+2]$, ..., $st_1[t]$, where k is the smallest position such that $H[st_1[k]] < H[i]$ and t is the largest position such that $st_1[t] < i$. We can use binary search to find k and t.

- Brick i can be gained, lost, or none of these. To see which one happens, we need to see whether i will be interesting after the change or not. We can do this by keeping maximums on suffixes, or simply by checking whether $H[st_1[t+1]] < H[i]$.

We obtain a time complexity of O(NlogN) for this solution.

Subtask 5 - $N \leq 6.000.000$

To solve this subtask, we use the same main ideas as in the solution for Subtask 4, but get rid of the binary search.

Note that if i is not interesting, then we will not gain anything by changing the colour of i, except for possibly i. In other words, purple bricks which are not in st_0 can lead to a gain of at most 1. Therefore, if changing the colour of i leads to a lost of at least 1, then we cannot get a better solution than the original one in which we were not making any changes. This means that we don't need to find k and t as in the previous subtask. We only need to check whether there exists any red interesting brick r such that $r < i$ and $H[r] < H[i]$. If such an r does not exist, there are no lost bricks and we need to check whether i will be interesting after the colour change. If i will be interesting, than we found a solution which is better by 1 than the original solution in which we don't make any modifications. Otherwise, if i will not be interesting, or if an r which satisfies the above conditions exists, then changing the colour of i cannot give a better solution than we already have.

It remains to check how many bricks we gain and how many we lose if we try to change each of the interesting purple bricks. We will therefore compute k and t for each brick i from st_0. Remember that k is the smallest position such that $H[st_1[k]] < H[i]$ and t is the largest position such that $st_1[t] < i$. Since the heights of bricks from st_0 are in decreasing order, the sequence of k-s obtained will be in increasing order, so we can just increase the k obtained for the previous step until it satisfies the condition $H[st_1[k]] < H[i]$. Similarly, the sequence of t-s obtained will be in increasing order and we can apply the same idea.

This solution gives an O(N) amortized complexity.

Problem CoolRot

Author: Tamio-Vesa Nakajima; *Prepared by: Tamio-Vesa Nakajima, Tinca matei, George Alexandru Râpeanu*

Firstly, an operation of type $update(d, x)$ will rotate the array with $d \times x$ positions. Because of that, there are N different obtainable arrays, so we can compute the number of inversions for every possible rotation. One problem is that, for a set of updates we won't exactly know which of the rotations will be obtainable.

We can express the number of rotations r of the base array by using the following formula:

$r \equiv ds_0 * x_0 + ds_1 * x_1 + ... + ds_{k-1} * x_{k-1} \pmod{n}$ where ds are the available operations from the query, and x_i represents what's the value of the x argument from the *update* function.

That is, if we know that the optimal rotation is r, and we know the values of the array x, then the operations applied will be:

- $update(ds_0, x_0)$

- $update(ds_1, x_1)$
- ...
- $update(ds_{k-1}, x_{k-1})$

Notice that the order of the update functions doesn't matter, because in the end we will get the same r.

If we look at $d = gcd(ds_0, ds_1, ds_2, ..., ds_{k-1})$, then we can observe that r must be a multiple of d. Notice that since all elements of ds and d are divisors of n, then we can ignore the modulo. Now we know that r must be a multiple of d, but we don't really know if we can obtain all multiples of d.

Lemma: If we can obtain the rotations x and y, then we can obtain $gcd(x, y)$.

Proof: We can simulate Euclid's algorithm of computing the greatest common divisor of two numbers. If we can obtain x and y, then we can obtain $|x - y|$. By doing this subtraction repeatedly, we will eventually obtain the gcd of the two numbers.

Using the above lemma repeatedly, then we can obtain:

- $gcd(ds_0, ds_1)$
- $gcd(gcd(ds_0, ds_1), ds_2) = gcd(ds_0, ds_1, ds_2)$
- ...
- $gcd(gcd(ds_0, ds_1, ..., ds_{k-2}), ds_{k-1}) = gcd(ds_0, ds_1, ..., ds_{k-1})$

Since we have a method to obtain d, then we can obtain all multiples of d (including 0).

At this point, the problem can be split into two independent subproblems:

- We must calculate the number of inversions for every possible rotation of the initial array, and then for each possible d we must calculate which multiple of d yields the rotation with the least amount of inversions;
- If the rotation is a multiple of d, we must find a way to generate that rotation by using the given operations.

Number of inversions

We can do a brute force algorithm. For each rotation, we calculate the number of inversions, by trying every pair. The complexity of this will be $O(N^3)$, because we have N possible rotations and for a rotation we have to check N^2 pairs.

To see for each d which rotation is the best, then we can also do a brute force algorithm. For each possible d, we look at all the multiples of d and take

the one with minimum number of inversions. The complexity of this is going to be $O(N * (\frac{1}{1} + \frac{1}{2} + ... + \frac{1}{N})) \approx O(N * logN)$. To prove that the sum of fractions from 1 to N is approximately equal to $log_2(N)$, we can compare each term like the following: $\frac{1}{N} \geq \frac{1}{2^t}$ where t is the highest natural number such that $2^t \leq N$.

Sadly, because of the first step which is done in $O(N^3)$, this approach will fail all the subtasks.

To optimise this step, we can observe that if we erase the first element from the sequence and insert it at the end of the array, then all the inversions will be the same except for the ones generated by that moved element. Suppose that element is x. If we erase it, then the number of inversions will decrease by the number of elements smaller than x, and after we insert it at the end, the number of inversions will increase by the number of elements greater than x.

If we calculate the number of inversions for the initial array in $O(N^2)$, then we can compute the number of inversions for every rotation by doing $O(N)$ for each rotation, because we have to compute the number of elements smaller and greater than the first element. In total, the complexity will be $O(N^2)$. This approach will fail all subtasks except for the first one.

To simplify the above approach a little bit, we can use the fact that the initial array is a permutation. That means that if we move the first element, let's call it x, then the number of elements smaller will be x (remember that the elements are indexed from 0), and the number of elements greater will be $N - x - 1$. The complexity is still going to be $O(N^2)$.

Notice that the only "expensive" step right now is calculating the number of inversions for the initial array. To optimise that, we can use two different approaches:

- We can use a divide-and-conquer method that implements merge-sort. We split the array in two halves, we calculate the number of inversions in the first and second half recursively, then we must combine the two-solutions. Since we also do a merge-sort, the two halves will also be sorted, so we can count the number of inversions between the two halves during the merge step. If we have to take an element from the left half, then the number of inversions will increase by the number of elements greater than the extracted element from the right half, that is the number of remaining elements in the respective half. Otherwise, we don't add anything to the number of inversions

- We iterate through each element of the array. When we process element x, we add to the number of inversions the number of marked elements greater than x, then we mark element x. We can see the number of marked elements greater than a particular value with a data structure such as a Segment Tree, or a Fenwick Tree (also called a Binary Indexed Tree).

The above approaches will both run in $O(NlogN)$ complexity, which will be enough to not fail any subtask.

Generating the solution

In this solution, we know that if we know how to write the number x, then we will also know how to write the elements $x + ds_0, x + ds_1, ..., x + ds_{m-1}$, everything modulo N. We can construct a graph by using this idea, by creating the edges from x to $x + ds_0, x + ds_1, ..., x + ds_{m-1}$ for every x from 0 to $N - 1$ and for every i from 0 to $m - 1$ (keep in mind that every element is going to be modulo N). Now to see how we can write the number r, then we can look at the path from the node 0 to node r. We can use a DFS or BFS algorithm to find that path, so the complexity of this solution will be $O(Q * N * m)$. This solution will fail every subtask except for ...

If $m = 1$ and the best rotation is r, then the operation we must apply is $update(ds[0], r/ds[0])$. Using this solution will result in the failure of the subtasks ...

If $m = 2$, then we must solve the following equation (modulo N): $ds_0 * x_0 + ds_1 * x_1 = d$ where $d = gcd(ds_0, ds_1)$. We can use the Extended Euclidean algorithm which solves this problem. At the end, we will multiply everything by r/d. Using this method, the failed subtasks will be ...

If $m = 3$, then we have to solve the following equation: $ds_0 * x_0 + ds_1 * x_1 + ds_2 * x_2 = d$ where $d = gcd(ds_0, ds_1, ds_2)$. To do this, suppose we have solutions for the following equations:

- $ds_0 * x_0 + ds_1 * x_1 = gcd(ds_0, ds_1) = d_1$
- $d_1 * y_0 + ds_2 * y_1 = gcd(d_1, ds_2) = gcd(ds_0, ds_1, ds_2) = d$

We can combine the above solutions by replacing d_1 in the second equation with the solutions from the first equation:
$(ds_0 * x_0 + ds_1 * x_1) * y_0 + ds_2 * y_1 = d \iff$
$ds_0 * (x_0 * y_0) + ds_1 * (x_1 * y_0) + ds_2 * y_1 = d$
Implementing the above solution will result in the failure of subtask ...

To further generalize this for $m \geq 4$, we can use the same idea as $m = 3$, by doing induction.

If we solved the equations:

- $(ds_0 * x_0 + ds_1 * x_1 + ... + ds_{m-2} * y_{m-2} = gcd(ds_0, ds_1, ds_2, ..., ds_{m-2}) = d'$
- $d' * y_0 + ds_{m-1} * y_1 = gcd(d', ds_{m-1}) = gcd(ds_0, ds_1, ..., ds_{m-1}) = d$

We can replace d' from the second equation with the first equation, and we will obtain:
$(ds_0 * x_0 + ds_1 * x_1 + ... + ds_{m-2} * x_{m-2}) * y_0 + ds_{m-1} * y_1 = d \iff$
$ds_0 * (x_0 * y_0) + ds_1 * (x_1 * y_0) + ... + ds_{m-2} * (x_{m-2} * y_0) + ds_{m-1} * y_1 = d$
We can implement this solution by solving the following $m - 1$ equations:

- $ds_0 * x_0 + ds_1 * y_0 = gcd(ds_0, ds_1) = d_0$
- $d_0 * x_1 + ds_2 * y_1 = gcd(ds_0, ds_1, ds_2) = d_1$

- $d_1 * x_2 + ds_3 * y_2 = gcd(ds_0, ds_1, ds_2, ds_3) = d_2$
- ...
- $d_{m-3} * x_{m-2} + ds_{m-1} * y_{m-2} = gcd(ds_0, ds_1, ..., ds_{m-1}) = d$

Then by going from the last equation to the first and using the observation from above, we can compute the coefficient of each element from a query. The complexity of this solution will be $O(m*log(N))$. This will not fail any subtask.

Problem Jumpy

Author: Tamio-Vesa Nakajima; *Prepared by: Tamio-Vesa Nakajima, Tinca matei, George Alexandru Râpeanu*

The first observation is that this game can be represented by a bipartite graph. Each node in this graph represents a set of cells accessible in one move by one of the players. An edge is drawn between graph nodes that contain a common cell. Thus the graph is bipartite, with one "side" of the bipartite graph containing cells accessible in one move by Little Square, and one "side" of the bipartite graph containing cells accessible in one move by Little Triangle.

Consider, as an example, the following game board:

```
...
.#.
```

Then there are nodes for the cell sets $A = \{(1,1),(1,2),(1,3)\}, B = \{(2,1)\}, C = \{(2,3)\}$ (for left-right moves) and for the cell sets $X = \{(1,1),(2,1)\}, Y = \{(1,2)\}, Z = \{(1,3),(2,3)\}$ (for up-down moves). Edges are now drawn between $A - X$, $A - Y$, $A - Z$, $B - X$, $C - Y$. The graph is bipartite, with one "side" of the graph containing A, B, C and one X, Y, Z.

With this representation, it turns out that the game can be proven to be equivalent with the following game. If the player starts from cell (x, y), put a piece in the cell set containing all the cells accessible from (x, y) for that player. On each move, the moving player is allowed to move the piece to an adjacent node that has not yet been visited. The player who cannot move to any adjacent node loses. (This equivalence is not immediate – but if you closely inspect the rules for cell colors, you can prove this).

Now, how can we see who wins this game? It can be proven that this game is winnable when starting from a node if and only if there exists a maximal matching that does not contain it (the idea of the proof is to consider a perfect matching as a strategy for the second player, who decides to move always along an edge from the matching). To check if there exists a maximal matching that does not contain a particular node, first construct a maximal matching. If the maximal matching happens not to include the node we are interested in, then we are done. Otherwise, remove the node and the edge in the matching adjacent to it. Then try to augment the matching with one extra node. This can be done efficiently by doing one pass of the Hopcroft-Karp algorithm (the

"maximal bipartite matching" algorithm that is usually taught to competitive programmers).

Team: The InfO(1)Cup Scientific Committee members are:

- Tamio-Vesa Nakajima, Executive President;
- Costin Andrei Oncescu;
- Stefan Taga;
- Mihai Popescu;
- Andrei Onut;
- Alexandru Rapeanu;
- Tiberiu Musat;
- Alex Nicolae Pop;
- Andrei Cotor;
- Alexandra Udristoiu;
- Tinca Matei;
- Alexa Tudose;
- Apostol Daniel;
- Alexandru Petrescu;
- Stefan Savulescu.