EJOI 2021, Day 1, English Editorial

August 26, 2021

Problem 1: Addk

(Proposed by Mihai Bunget.)

Subtask 1 A type 1 requirement is not processed because the array A does not change.

For query of type 2 we calculate the sum $S = A_l + A_{l+1} + ... + A_{l+m-1}$, which is added to the result, then for every i from $l + m$ to r $S = S - A_{i-m} + A_i$ is updated, which is added to the result.

Subtask 2 A type 1 requirement is not processed because the array A does not change.

For the type 2 requirement we observe the following two cases:

- If the length of the sequence A_l, A_{l+1}, ..., A_{r-1}, A_r, ie $r - l + 1$, is at least $2 \cdot m$, then the sum of the elements of all subsequences of length m in this sequence will be $S = S_1 + S_2 + S_3$, where $S_1 = 1 \cdot A_l + 2 \cdot A_{l+1} + ... + (m-1) \cdot A_{l+m-2}$, $S_2 = (r-l+1-2\cdot(m-1)) \cdot (A_{l+m} + A_{l+m+1} + ... + A_{r-m+1})$, $S_3 = (m-1) \cdot A_{r-m+2} + ... + 2 \cdot A_{r-1} + 1 \cdot A_r$.
- If the length of the sequence A_l, A_{l+1}, ..., A_{r-1}, A_r is less than $2 \cdot m$, then the sum of the elements of all subsequences of length m in this sequence will be $S = S_1 + S_2 + S_3$, where $S_1 = 1 \cdot A_l + 2 \cdot A_{l+1} + ... + (rlm) \cdot A_{rm-1}$, $S_2 = (r-l+2-m) \cdot (A_{rm} + A_{r-m+1} + ... + A_{l+m-1})$, $S_3 = (rl-m+1) \cdot A_{l+m} + ... + 2 \cdot A_{r-1} + 1 \cdot A_r$.

For this we construct the arrays of partial sums B and C, so that $B_i = A_1 + A_2 + ... + A_i$ and $C_i = 1 \cdot A_1 + 2 \cdot A_2 + ... + i \cdot A_i$, for any i from 1 to N. These being constructed, for the first case above we will have $S_1 = C_{l+m-2} - C_{l-1} - (l-1) \cdot (B_{l+m-2} - B_{l-1})$, $S_2 = (r - l + 1 - 2 \cdot (m-1)) \cdot (B_{r-m+1} - B_{l+m-1}$, $S3 = (r+1) \cdot (Br - Br - m + 1) - (Cr - Cr - m + 1)$.

Subtask 3 Because in this case we also have the update operation on array A we will use the binary indexed tree structure for arrays B and C defined above.

Thus for the type 1 requirement we will update these BIT's (binary indexed trees), for each of the elements A_{i_1}, A_{i_2}, ..., A_{i_K} .

For the type 2 requirement we will use two interval calculation functions for arrays B and C, using the BIT's associated with these arrays. The calculation formulas are the same from subtask 2.

Problem 2: Kpart

(Proposed by Ionel-Vasile Piț-Rada.)

We will further note by $ksir(A, N, pos, kValues)$ the context of the solution for the string A_1, A_2, ..., A_N. The $kValues$ string contains, in ascending order, all the values K for which A is K-string. The pos string is defined by $pos[s]$= the highest index, between $1, 2, 3, ..., N$, for which there is a sub-sequence that starts with $A[pos[s]]$ and has the amount s. The values $pos[s]$ will be equal to -1 for the amounts s, $1 \leq s \leq 50000$, which have not yet been reached. The value $pos[0]$ is initially equal to 0.

Suppose we have solved the problem $ksir(A, N-1, pos, kValues)$ and we want to get the solution for the problem $ksir(A, N, pos, kValues)$. How can we proceed?

The newly appeared value is $A[N]$, so we will update in a first step the string pos by going through decreasing all the positive amounts s already reached and updating the $pos[s + A[N]]$ each time $s + A[N] \leq 50000$. The update will be made using the expression $pos[s + A[N]] = max(pos[s + A[N]], pos[s])$. Then it will update $pos[A[N]]$with N.

In the second stage we will go through the string $kValues$ and update it. Note that the values $kValues[j], 1 \leq j \leq length(kValues)$ are all already validated for the sub-string A_1, A_2, ..., A_{N-1}. What we are interested in now is to check/validate the sub-strings $A_{N-kValues[j]+1}...A_N$, checking if the amounts $s = A_{N-kValues[j]+1}$ + $A_{N-kValues[j]+2}$ + ... + A_N are even and $pos[s/2] \geq N - kValues[j] + 1$, where $1 \leq j \leq length(kValues)$, which ensures that the amounts $s/2$ can be obtained using only elements of a sub-sequence inside the verified sub-string. The sub-string $A_1...A_N$ is checked/validated separately using the same idea. Obviously, all valid $kValues[j]$ values will be kept.

Initially it will start with the context of the $ksir(A, 0, pos, kValues)$ in which A is the empty string, $length(kValues) = 0$, $pos[0] = 0$ and $pos[j] = -1$ for $1 \leq j \leq 50000$, and then step by step the contexts will be updated $ksir(A, i, pos, kValues)$, for $1 \leq i \leq N$. The complexity is $O(T \cdot N \cdot S)$, where $S \leq 50000$.

Problem 3: Xcopy

(Proposed by Tulba-Lecu Theodor-Gabriel & Pop Ioan Cristian)

Disclaimer The subtasks were designed to give contestants a way to solve the task step by step. Each subtask adds a new level of difficulty and brings you closer to the optimal solution. The proofs for all the observations can be found at the end of the editorial in the Appendix section.

1 Solution for subtask 1

When we set $N = 1$, the task is reduced to finding an array such that any two consecutive numbers differ by exactly one bit. This is well known as a Gray code.

We can greedily generate a partial Gray code of length M that contains all the elements from 0 to $M - 1$ exactly once. For the rest of the editorial we will refer to this as a "compact Gray code". A constructive algorithm can be found in the section A of the Appendix.

2 Solution for subtask 2

For this subtask we need to worry if and how the bits that change when moving on a row, influence those that change when moving on a column. It turns out that the two sets of bits are completely independent of each other (no bit can be influenced by both moving on a row or by moving on a column). A proof for this can be found in the section B of the Appendix.

Since both N and M are powers of 2, the Gray codes are complete and thus we only need to concatenate the Gray code for the rows, and columns. An example for such a test would be $N = 4$ and $M = 8$. A possible optimal answer would be:

0	1	3	2	6	7	5	4
8	9	11	10	14	15	13	12
24	25	27	26	30	31	29	28
16	17	19	18	22	23	21	20

3 Solution for subtask 3

The third subtask is a combination of the first two subtasks. You need to compute a "compact Gray code" for the column set of bits. And then concatenate the bits of the two sets in a way to allow an optimal answer.

4 Complete solution

The last observation we need to make in order to solve the problem is how to combine the two sets of bits to obtain a minimal value. Just concatenating doesn't necessarily produce an optimal outcome. For example for $N, M = 6$ if we generate the following "compact Gray code" for both sets of bits:

$$5 = 101_{(2)}, 1 = 001_{(2)}, 3 = 011_{(2)}, 2 = 010_{(2)}, 0 = 000_{(2)}, 4 = 100_{(2)}$$

and concatenate the answers we would obtain the following matrix:

101101	101001	101011	101010	101000	101100
001101	001001	001011	001010	001000	001100
011101	011001	011011	011010	011000	011100
010101	010001	010011	010010	010000	010100
000101	000001	000011	000010	000000	000100
001101	001001	001011	001010	001000	001100

with a maximum value of $45 = 101101$. Whereas if we look at the maximum value ($5 = 101_{(2)}$), we can observe that we can merge the two 5s in the following way: $43 = 101011$. Thus we can minimize the maximum value of the matrix to 43 instead of 45 and obtain the following matrix:

101011	100011	100111	100110	100010	101010
001011	000011	000111	000110	000010	001010
011011	010011	010111	010110	010010	011010
011001	010001	010101	010100	010000	011000
001001	000001	000101	000100	000000	001000
101001	100001	100101	100100	100000	101000

The merging can be done in various ways, one of which would be using dynamic programming:

$dp_{i,j}$ = the minimum value that can be obtained using the first i bits of the maximum of the first set and the first j bits of the maximum of the second set.

A Compact Gray code constructive algorithm

Gray codes of power-of-two length. Note that a Gray code of length 2^n can be created by setting element i (indexed from 0) to $i \oplus (i \texttt{ div } 2)$. This Gray code can also be rotated (thus if $g_1, \ldots, g_{2^n}$ is a Gray code, so is $g_k, \ldots, g_{2^n}, g_1, \ldots, g_{k-1}$. We leave as an exercise to the reader proving that this works.

Gray codes of arbitrary length. Observe the following facts:

- We can create a Gray code of length 2^n starting with any number from 0 to $2^n - 1$, using the previous construction suitably rotated.
- Any integer is a sum of distinct powers of 2.

These facts suggest immediately an algorithm: create Gray codes of lengths equal to the powers of 2 that compose the length of the desired Gray code, and put them together in some way. We will show how this is done by example, for length 7.

Note that Gray codes for length $1, 2$ and 4 respectively are $0, 1, 3, 2$, $0, 1$ and 0 respectively. We now "add in" the high order bit in the second and third Gray codes to get $0, 1, 3, 2$, $4, 5$, 6. Now we need to rotate the Gray codes so that they can be adjacent, from right to left. 6 can only be adjacent to 4, so the array ends with $5, 4, 6$. 5 can only be adjacent to 1, so the full array is $3, 2, 0, 1, 5, 4, 6$.

B The row-column independence

First let's note that if we fix the elements at the indexes (i, j), $(i + 1, j)$ and $(i, j + 1)$, then the element at the index $(i + 1, j + 1)$ is uniquely determined by the following formula ($\oplus$ represents the bitwise XOR operation):

$$M[i+1][j+1] = M[i][j+1] \oplus M[i+1][j] \oplus M[i][j]$$

More than that, we can recursively extend this formula to:

$$M[i][j] = M[0][j] \oplus M[i][0] \oplus M[0][0]$$

Now let's suppose that there exists a bit such that it is changed both on a row and a column. That means that there exists a bit k and a row i and column j such that:

- $M[i+1][x] = M[i][x] \oplus 2^k, \forall\ x \in \{0, 1, \ldots, M-1\}$
- $M[x][j+1] = M[x][j] \oplus 2^k, \forall\ x \in \{0, 1, \ldots, N-1\}$

From the previous relation we get the following result:

$$M[i+1][j+1] = M[i][j+1] \oplus M[i+1][j] \oplus M[i][j]$$

$$M[i+1][j+1] = M[i][j] \oplus 2^k \oplus M[i][j] \oplus 2^k \oplus M[i][j]$$

$$M[i+1][j+1] = M[i][j]$$

In conclusion, if there would exist a bit that is influenced both by a line and by a column, then the condition of distinctness would not be satisfied.

Scientific committee

The problems were prepared by:

- Adrian Panaete (chair) - “A.T. Laurian” National College, Botoșani
- Ionel-Vasile Piț-Rada - “Traian” National College, Drobeta Turnu Severin
- Maria-Alexa Tudose - University of Oxford, UK
- Gheorghe-Eugen Nodea - “Tudor Vladimirescu” National College, Târgu Jiu
- Tamio-Vesa Nakajima - Oxford, Computer Science department, UK
- Mihai Bunget - “Tudor Vladimirescu” National College, Târgu Jiu
- Andrei-Costin Constantinescu - University of Oxford, UK
- Ciprian-Daniel Cheșcă - “Grigore C. Moisil” Technological High School, Buzău
- Lucian Bicsi - University of Bucharest
- Dan Pracsiu - “Emil Racoviță”, Theoretical High School, Vaslui
- Ioan-Cristian Pop - Polytechnical University, Bucharest
- Theodor-Gabriel Tulbă-Lecu - Polytechnical University, Bucharest
- Raluca-Veronica Costineanu - “Ștefan cel Mare” National College, Suceava

EJOI 2021, Day 2, English Editorial

August 27, 2021

Problem 1: Binsearch

(Proposed by Maria-Alexa Tudose.)

Let us call the value $\frac{n-1}{2}$ "the middle value" and the position $\frac{n-1}{2}$ (in the permutation p) "the middle position".

Subtask 1. When $b_i = \mathit{true}$ for all i, we can achieve $S(p) = 0$ by setting p to be the increasing permutation.

Subtask 2. When $b_i = \mathit{false}$ for all i, we can achieve $S(p) = 1$ in the following way:

- Place the middle value on the middle position.
- Place all values smaller than the middle value on positions $\frac{n-1}{2} + 1, \ldots, n$, in any order.
- Place all values bigger than the middle value on positions $1, \ldots, \frac{n-1}{2} - 1$, in any order.

In particular, note that the decreasing permutation works.

Subtask 3. In this subtask $N = 7$ holds. We can use the fact that $7! = 5040$ is small. This allows us to generate all the possible permutations of size n, and for each such permutation we can run binary search on all values from 1 to n and compare the returned result with the desired value in b. We calculate $S(p)$ for all permutations p, and print any permutation p that achieves $S(p) \leq 1$.

Subtask 4. This subtask encourages solutions with sub-optimal (but polynomial) time complexities. For example, a poor implementation of the "Solution 1" presented below might have a time complexity of $O(n^2)$ instead of the optimal $O(n)$.

Subtask 5. In this subtask, the sequence b is guaranteed to be generated randomly. This allows us to design solutions which use different facts, such as:

- The numbers of ones and zeros in b should be approximately equal.
- There are not many consecutive equal entries in b.

Subtask 6. The problem admits a wide variety of full solutions which promote different types of thinking. We present only a few of them.

Solution 1. We try to place, in turn, each possible value on the middle position. Let X be our current try. Our aim is to find p for which $b_i =$ `binary_search(n, p, i)` for all values $i \neq X$. This would give $S(p) = 0$ if $b_X = 1$ and would give $S(p) = 1$ if $b_X = 0$.

We define two sets L and R:

- $L = \{i \mid b_i = \texttt{true}, i < X\} \cup \{i \mid b_i = \texttt{false}, i > X\}$
- $R = \{i \mid b_i = \texttt{true}, i > X\} \cup \{i \mid b_i = \texttt{false}, i < X\}$

We also define two sets A and B:

- $A = \{i \mid b_i = \texttt{true}\}$
- $B = \{i \mid b_i = \texttt{false}\}$

Lemma. *If $|L| = |R| = \frac{n-1}{2}$ for some X, then we can find a good permutation.*

Proof. One way to build such a permutation is:

- First place the values in L in increasing order
- Then place the values in R in increasing order

□

Lemma. *There exists an X such that $|L| = |R| = \frac{n-1}{2}$.*

Proof. We aim to achieve $|L| = \frac{n-1}{2}$.

When we change X to $X + 1$, $|L|$ either stays constant, increases by 1, or decreases by 1.

For $X = 1$, we have $|L| = |B - \{1\}|$. Also, when we set $X = n$, we have $|L| = |A - \{n\}|$. Therefore, $|L| \leq \frac{n-1}{2}$ for $X = 1$ or for $X = n$, and $|L| \geq \frac{n-1}{2}$ for $X = 1$ or for $X = n$. Combining these observations, we get the conclusion. □

Solution 2. If $A = \emptyset$, then we return the decreasing permutation.

From now on, assume that $A \neq \emptyset$

We choose X to be any value from A for which $|L \cap A|, |R \cap A| \leq \frac{n-1}{2}$.

Intuitively, this means that we place on the middle position any value from A for which all remaining values from A "fit properly" in the remaining halves.

There are 2 cases now: $|L| \geq |R|$ and $|L| < |R|$. We will treat only the first case, because the second one can be solved symmetrically.

Assuming $|L| \geq |R|$, let W be a set of size $|L| - \frac{n-1}{2}$ s.t. $W \subseteq L \cap B$. We will place the values in $L - W$ in the first half of the permutation in increasing order and the values in $R \cup W$ in the second half (in an order to be determined).

All elements in the first half will get the required result when calling binary search. The value on the middle position also gets the required result.

We therefore need to arrange the elements in the second half such that we get at most one position i for which $b_i \neq$ `binary_search(n, p, i)`. We can solve this recursively.

Solution 3. (This solution is due to Tamio-Vesa Nakajima.) We will create a procedure *solve*(A, B) which creates a sequence p for which $S(p) \leq 1$ if A is the set of indices i with $b_i =$ `true` and B is the set of indices i with $b_i =$ `false`. This will be a recursive procedure, using the following cases:

- If $A = \emptyset$ then we are in subtask 2 – output B sorted in decreasing order.
- If $|A| + |B| \leq 3$ then the answer can be easily found by hand (there are only 8 cases).
- Now suppose $A \neq \emptyset$ and $|A| + |B| \geq 7$. Thus at least one element should be "found". We have two further cases:

- Suppose $|A| > |B|$. Suppose there are $2^k - 1$ elements overall. Let A' contain the first 2^{k-1} elements of A in increasing order. Then output A' in increasing order first, followed by *solve*$(A - A', B)$. For example if $A = \{1, 3, 5, 6, 7\}, B = \{2, 4\}$, then $A' = \{1, 3, 5, 6\}$ and our output starts with $1, 3, 5, 6$ followed by the result of *solve*$(A - A' = \{7\}, B = \{2, 4\})$.
- Suppose $|A| < |B|$. Suppose $|A| + |B| = 2^k - 1$. (Note that the case $|A| = |B|$ is impossible as $|A| + |B| = 2^k - 1$ is odd.) Since $|B| > |A|$ we deduce that $|B| \geq 2^{k-1} - 1$. Let $t = 2^{k-2} - 1$. Let X be the first t elements of B in increasing order and Y be the last t elements of B in increasing order. Since $|B| \geq 2t$ we deduce that $X \cap Y = \emptyset$ i.e. X and Y have no common elements. Let b be an arbitrary element from $B - X - Y$. There are two cases: either $b < \min A$ or $b > \min A$ – we will assume without loss of generality that $b < \min A$, since the other case is treated symmetrically. We construct our array as follows:
  - The first t elements of the result are Y (in any order).
  - The next element of the result is b.
  - The next t elements of the result are X (in any order).
  - The next element (in fact the middle element) should be $a = \min A$.
  - The second half of the result should be *solve*$(A - \{a\}, B - X - Y - \{b\})$.

  Observe that:
  - All of the elements in Y are greater than b, and thus are not found.
  - $b < \min A$ and thus is not found.
  - All of the elements in X are less than b and thus are not found.
  - $\min A$ is immediately found.
  - By the correctness of *solve* the elements in the second half contribute at most 1 to $S(p)$.

  As an example, suppose $A = \{3, 4\}, B = \{1, 2, 5, 6, 7\}$. Then $X = \{1\}, Y = \{7\}, b = 2$, and $2 = b < \min A = 3$. Thus the array begins with $7, 2, 1, 3$ followed by the result of $(\{4\}, \{5, 6\})$, which can be $5, 4, 6$ for instance. Thus the result is $7, 2, 1, 3, 5, 4, 6$, with $S(p) = 1$.

Problem 2: Dungeon

(Proposed by Lucian Bicsi. Primarily prepared by Teodor-Gabrial Tulba-Lecu. We thank Tamio-Vesa Nakajima for this editorial.)

To solve this problem, first note that the dungeon explorer's "mental state" consists of:

- The current position relative to the starting position.
- The set of possible starting positions.

Thus initially the state consists of position $(+0, +0)$ relative to the starting position, and the set $\mathcal{S} = \{1, \ldots, S\}$ of starting position (represented by their indices).

Observe that if we know that we started at some *subset* of starting positions $\mathcal{S}'$, then we will never go to a relative position that corresponds to a mine relative to *any* starting position from $\mathcal{S}'$. This is because doing this would risk getting 0 coins. Other than this we can visit any relative position.

Thus, we can imagine the following algorithm for the explorer:

- Consider the current set of starting positions that we could have started at $\mathcal{S}'$.
- Visit all possible squares that do not require us to visit a relative position that could correspond to a bomb for any of the starting positions in $\mathcal{S}'$.
- Check if we see any wall or coin that only appears in some *proper* subset of starting positions $\mathcal{S}'' \subset \mathcal{S}'$.
- If such a position exists, continue the search from that subset.
- Otherwise give up with the coins we can collect at this point.

How can we simulate this algorithm in our case? It is not difficult to make a version that does $O(NMS)$ complexity for each starting set (simply do a breadth first search, checking if a position is visitable, or respectively is a wall or a coin that only appears for some starting positions, by iterating over the current starting positions). To optimise this to use $O(NM)$ time for each set of starting positions that we check, store the map "relative to each starting position" in a bit mask. Thus we will have several matrices `wall/bomb/coin` that, at position (i, j) will contain a bit mask that have bit k equal to 1 if and only if position (i, j) relative to starting position k contains a wall/bomb/coin. This allows us to check (a) if a certain relative position contains a bomb for a subset of starting positions, and (b) if a certain relative position contains a wall/coin in some starting positions but not others. These can be done by representing the set of starting positions with a bit mask, and the using a bitwise "and" operation. Then for (a) we check if the result is nonzero, and for (b) we check if it is neither zero or equal to the bit mask that represents the set of starting positions.

Thus with this approach we can find, in $O(NM)$, for any set of starting positions:

- The number of coins we can safely collect.
- If any wall or coin is visible for only some subset of starting positions.
- What that subset of starting positions is.

Now suppose $f(\mathcal{S})$ gives us the result for some set of starting positions $\mathcal{S}$. The full result will be $f(\{1, \ldots, S\})$. To compute $f(\mathcal{S})$ use the following algorithm:

- Check if some coin or wall is visible only in some subset $\mathcal{S}' \subset \mathcal{S}$.
- If so, then return $\min(f(\mathcal{S}'), f(\mathcal{S} - \mathcal{S}'))$.
- Otherwise return the number of coins safely collectable if we would start at $\mathcal{S}$.

It can be proved that this does at most S matrix traversals. Thus the final complexity is $O(NMS)$.

Problem 3: Waterfront

(Proposed by Eugen Nodea. We thank Tamio-Vesa Nakajima for this editorial.)

We will describe the solution in several stages, starting from a brute-force solution, and then gradually optimising it.

Brute force solution. A simple brute-force solution is the following:

- Find the tree which will be the tallest *at the end of the M days.*
- Find the first moment at which this tree can be cut, if such a moment exists.
- Cut the tree at that moment, if it exists.
- If not, output it's height.

To see (broadly) why this is correct, suppose that we try to see if the solution can be at most S. Then, based on the heights at the end of the M days, we know how many times each tree must be cut (according to the formula $\lceil S - \mathit{finalHeight}/x \rceil$). Each cut can then be done in some suffix of days. We then need to assign cuts to days in some way. This is an example of an *activity selection* problem. We have M days in which we can do k activities daily, and each activity can be done in some suffix of days. It is well known that such problems can be solved using a greedy approach – which is exactly what we do.

The previous remarks justify our brute force algorithm. If the real solution is S, then we note that the set of tree cuts done by our brute force algorithm are precisely the tree cuts necessary for solution all solutions greater or equal to S in decreasing order – thus stopping at precisely the correct solution. Furthermore, we can see that our greedy solution is correct, since the set of assigned activities will be the same as if we only considered those cuts (even if certain cuts may be done in different days).

Optimisation 1. First we optimise the part of the solution where we find the first point at which a cut can be made. Note that it is easy to check using arithmetic and some bookkeeping the first point in time in which a certain tree is tall enough to be cut. We now need to find the first day in which it can be cut. If we model the days as an array $v[1], \ldots, v[M]$, where $v[i]$ is the number of cuts allowed in day i, then we want to find the first non-zero value in some suffix $v[a], \ldots, v[M]$. Then we want to decrement that value (that is where the cut is done).

How do we do this? Suppose we consider a partition of indices $1, \ldots, M$ where all adjacent indices i where $v[i] = 0$ are joined into the same partition. In this case to find the first nonzero value to the right of index a, we find the set to which a belongs, we find the rightmost index r in this partition, and:

- if $v[r] = 0$, then by the definition of the partition the next nonzero index if $r + 1$;
- otherwise $r = a$ and a itself can be decremented.

This partition can be efficiently maintained in $\log^* M$ complexity using a disjoint set data structure.

Optimisation 2. We now optimise the way of choosing the tree to cut. We previously said that we always choose the tree that is tallest at the end of the M days. In essence this can be simulated using a priority queue. The keys are the heights at the end of the M days, and the values are the indices of the trees.

Optimisation 3. The idea from the previous paragraph can be optimised further. Note that we actually have the following operations on our priority queue:

- Finding the maximum key-value pair.
- Decrementing the maximum key by a constant, x.

This type of priority queue can be implemented in constant time for each operation, with $n \log n$ precalculation time. To do this, maintain two data structures, a stack S and a queue Q. Insert all the key-value pairs into S in increasing order (so that the maximum value is at the top of the stack). When trying to get the maximum key-value pair, take the maximum of the key-value pair from among the top of S and the front of Q. To decrement it's key, remove it from S or Q respectively, and add it into Q (at the back), with a decremented key. We leave the proof of correctness as an exercise.

Scientific committee

The problems were proposed and prepared by:

- Adrian Panaete (chair) - "A.T. Laurian" National College, Botoșani
- Ionel-Vasile Piț-Rada - "Traian" National College, Drobeta Turnu Severin
- Maria-Alexa Tudose - University of Oxford, UK
- Gheorghe-Eugen Nodea - "Tudor Vladimirescu" National College, Târgu Jiu
- Tamio-Vesa Nakajima - Oxford, Computer Science department, UK
- Mihai Bunget - "Tudor Vladimirescu" National College, Târgu Jiu
- Andrei-Costin Constantinescu - University of Oxford, UK
- Ciprian-Daniel Cheșcă - "Grigore C. Moisil" Technological High School, Buzău
- Lucian Bicsi - University of Bucharest
- Dan Pracsiu - "Emil Racoviță", Theoretical High School, Vaslui
- Ioan-Cristian Pop - Polytechnical University, Bucharest
- Theodor-Gabriel Tulbă-Lecu - Polytechnical University, Bucharest
- Raluca-Veronica Costineanu - "Ștefan cel Mare" National College, Suceava