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

**Subtask 2.** When $b_i = \textit{false}$ 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 $\textit{solve}(A, B)$ which creates a sequence $p$ for which $S(p) \leq 1$ if $A$ is the set of indices $i$ with $b_i = \texttt{true}$ and $B$ is the set of indices $i$ with $b_i = \texttt{false}$. 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 $\mathcal{S} = \{1, \dots, S\}$ 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 $f(\{1, \dots, S\})$. 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 $v[1], \dots, v[M]$, 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 $v[a], \dots, v[M]$. 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 $1, \dots, M$ 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