

## Editorial

Tasks, test data and solutions for CEOI 2022 were prepared by: Dominik Fistrić, Josip Klepec, Krešimir Nežmah, Ivan Paljak and Paula Vidas. Implementation examples are given in attached source code files.

### Task Drawing

**Prepared by:** Ante Đerek, Luka Kalinović, Paula Vidas, and Dominik Fistrić

*Fun fact: This task was originally prepared for CEOI 2013 (also in Croatia). However, it was deemed too difficult, especially because the contestants didn't have full feedback back then, and thus it didn't appear on the contest. Task idea was proposed by Ante Đerek and solution was found by Luka Kalinović. To honor them, this year's statement featured Ante and Luka as main characters.*

Note that in any tree with maximum degree at most three, we can always find a node with degree at most two. If we root the tree in such a node, every node will have at most two children (i.e. it's a binary tree).

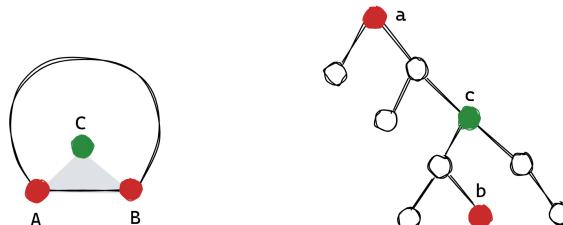
Let's start with the first subtask. Let  $P_1, \dots, P_N$  be the given points, such that they form vertices of a convex polygon in that order. Place the root of the binary tree in any of the points, say  $P_1$ . Let  $k$  be the size of the subtree of one of its children. We can place this child in node  $P_2$  and decide to place its whole subtree in points  $P_2, \dots, P_{k+1}$  in some way. If the node has two children, we place the second child in node  $P_{k+2}$  and its subtree in  $P_{k+2}, \dots, P_N$ . We then solve the subproblems recursively. A careful implementation of this strategy has complexity  $O(N \log N)$ .

We will now describe a solution for subtasks 2 and 3. Similarly to subtask 1, place the root in some point  $A$  on the *convex hull* of the given point set. Sort all other points by the polar angle around point  $A$ . In other words, we will sort them such that if point  $B$  is before point  $C$ , then points  $A, B, C$  are ordered counter-clockwise. Again, if  $k$  is the subtree size of a child, place this child's subtree in the first  $k$  points, choosing the first of the points as the one where the child goes. If there are two children, the other child's subtree is placed in the remaining points, and the child is placed in the first of them.

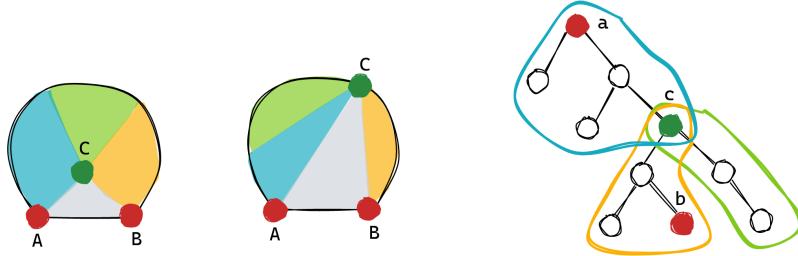
Time complexity of this approach is  $O(N^2 \log N)$  if we sort the points each time, which is enough for the second subtask. It can be speed up to  $O(N^2)$  if we use a some [selection algorithm](#) with expected complexity  $O(N)$  (e.g. `std::nth_element` in C++) to find the  $k$ -th smallest element. This is fast enough to pass the third subtask.

Finally, we describe the full solution. Notice that for now we used a procedure that takes a set of points with one distinguished point on the hull, and a tree with one distinguished node, and draws the tree on those points, such that the distinguished node is placed in the distinguished point. It turns out that it's possible to design a procedure that does the same but given two pairs of distinguished points and nodes, such that the points lie on the hull and are adjacent.

The high level idea is the following: If we're given one point-node pair  $(A, a)$ , such that  $A$  is on the hull and the tree is rooted in  $a$ , then we choose some leaf  $b$  and one of the adjacent points on hull as  $B$ , and  $(B, b)$  is our second pair, so we're in the two pair case. Assume now we're given two point-node pairs  $(A, a)$  and  $(B, b)$ , such that  $a$  is the root of the tree, while  $b$  is a leaf, and  $A$  and  $B$  are two adjacent points on the hull. Take  $c$  to be some node on the path between  $a$  and  $b$ , and take  $C$  to be a point such that the gray triangle  $ABC$  doesn't contain any other points.



Then, we can construct three subproblems, blue, green, and yellow, shown in the figure below. The blue, green, and yellow regions are chosen such that the number of points in each region matches the size of the respective tree, and the points in a region form a consecutive subsequence if we sort all points by polar angle around  $C$ . We distinguish two cases, when  $C$  is not on the hull and when it is.



It can be proven that there always exists a “good” point  $C$ , i.e. a point such that the gray region has no other points inside,  $AC$  is a line segment on the hull of the blue region,  $C$  is on the hull of the green region, and  $BC$  is a line segment on the hull of the yellow region. This is left as an exercise for the reader.

The three subproblems can be solved recursively. In the blue problem we’re given point-node pair  $(A, a)$  as root and  $(C, c)$  as leaf, in the yellow problem we’re given point-node pair  $(C, c)$  as root and  $(B, b)$  as leaf, and in the green problem we’re given a point-node pair  $(C, c)$  as root.

Now it’s time to analyse the time complexity.

When choosing nodes  $b$  or  $c$ , it’s not optimal to simply take any node. When we’re given root  $a$  and need to choose a leaf  $b$ , we’ll do it this way: start from the root, and at each step move down to the “heavy” child, i.e. the one with the larger subtree size, until a leaf is reached. When we’re given root  $a$  and leaf  $b$ , for node  $c$  we’ll take the midpoint of the path between  $a$  and  $b$ .

Consider a two point-node pair subproblem with  $N$  nodes. Let  $k$  denote the length of the path from  $a$  to  $b$ , and let’s look at the subtrees attached to this path that belong to the “light” children. Denote the sizes of these subtrees as  $N_1, N_2, \dots, N_k$ . Notice that  $N_1 + N_2 + \dots + N_k \leq N$ , and  $N_i \leq \frac{N}{2}$  for all  $i = 1, 2, \dots, k$ , because they are the light children. Note that each subproblem can be solved in time that is linear in the size of the subproblem (not counting the recursive calls). Therefore, after  $\log N$  steps of the process described above, we’ll reach all the subproblems of size  $N_i$ , and will have spent  $O(N \log N)$  time up that point. Consequently, if  $T(N)$  denotes the time required to solve a subproblem of size  $N$ , we obtain

$$T(N) = \sum_{i=1}^k T(N_i) + O(N \log N).$$

This recurrence relation works out to be  $T(N) = O(N \log^2 N)$ .

If in each subproblem we sort the points by angle, a factor of  $\log N$  is added to the complexity, and that solution is fast enough for subtask 4. However, sorting is not necessary if we use `std::nth_element`, and that solution should achieve full score for this task.



## Task Measures

**Prepared by:** Ivan Paljak, Paula Vidas, and Dominik Fistrić

Consider the situation when there are  $N$  people standing at coordinates  $a_1, \dots, a_N$ , and without loss of generality let the coordinates be sorted, i.e.  $a_1 \leq \dots \leq a_N$ .

It's easy to see that there exists an optimal rearrangement after which the order of the people remains the same as the initial order. Otherwise, if there is a moment when some two people swap places in the order, the total time won't increase if they instead don't change their relative order at that moment (but instead move where the other person would have moved).

Let  $t_{i,j} = \frac{1}{2}((j-i) \cdot D - (a_j - a_i))$ , for any  $i \leq j$ , and let  $t = \max_{i \leq j} t_{i,j}$ . We claim that  $t_{\text{opt}}$  is equal to  $t$ .

Clearly,  $t$  is a lower bound. After  $t'$  seconds, the distance between people  $i$  and  $j$  can be at most their initial distance  $a_j - a_i$  plus  $2t'$ . Thus  $a_j - a_i + 2t_{\text{opt}} \geq (j-i) \cdot D$ , which implies  $t_{\text{opt}} \geq \frac{1}{2}((j-i) \cdot D - (a_j - a_i)) = t_{i,j}$ .

Now we prove  $t$  is an upper bound. Consider the following greedy strategy. Let  $b_1, \dots, b_N$  be the final coordinates, calculated as:  $b_1 = a_1 - t$ , and  $b_i = \max(a_i - t, b_{i-1} + D)$  for  $i > 1$ . First, we prove that nobody moved more than  $t$ . For some person  $j$ , let  $i \leq j$  be the maximal index such that  $b_i = a_i - t$ . Then, we can see that  $b_j = b_i + (j-i) \cdot D$ , so  $b_j - a_j = (j-i) \cdot D + b_i - a_j = (j-i) \cdot D + a_i - t - a_j \leq 0$ . On the other hand,  $b_j \geq a_j - t$ , i.e.  $b_j - a_j \geq -t$ , so we're done. Second, we prove that after the rearrangement, no two consecutive people are standing closer than  $D$ . This follows from  $b_i \geq b_{i-1} + D$ , and the proof is complete.

Naively calculating all  $t_{i,j}$  and taking the maximum one, for each of the  $M$  scenarios, is enough to solve the first subtask. Time complexity is  $O(M(N + M)^2)$ .

To solve the second subtask, we don't need the formula. Instead, we can binary search  $t_{\text{opt}}$  directly. To check if some  $t'$  is feasible, we use the greedy strategy from the proof above, and at the end check if all consecutive distances are at least  $D$ . Time complexity is  $O(M(N + M) \log T)$ , where  $T$  is the maximum possible value of  $t_{\text{opt}}$  given the constraints.

The third subtask can be solved using the formula for  $t$ . We can rewrite  $t_{i,j} = \frac{1}{2}((a_i - iD) - (a_j - jD))$ . Since each new person is added to the end of the line, it's enough to maintain the current maximum value of  $a_i - iD$ . Then, when  $j$ -th person is added, it's easy to find the maximal  $t_{i,j}$  over all  $i$ . Time complexity is  $O(N + M)$ .

In order to solve the fourth subtask, we need to be able to insert a person at an arbitrary place in the line. Consider the value  $a_i - iD$ . When a person is inserted somewhere in the middle of the line, this value remains the same for everyone on the left side, and decreases by  $D$  for everyone on the right side of this person. Furthermore, values  $t_{i,j}$  remain the same between people who are on the same sides relative to the new person, and it increases by  $D$  between people on opposite sides. Therefore, to find the new  $t_{\text{opt}}$ , it's enough to find the maximum of  $a_i - iD$  in the left part and the minimum in the right part.

This process can be simulated using a segment tree which supports adding a number to a range and querying for minimum and maximum of a range. This is a classic problem, for more info check this [link](#). Time complexity is  $O((N + M) \log(N + M))$ .

## Task Parking

Prepared by: Ivan Paljak and Krešimir Nežmah

The first subtask can be solved in various ways. Since  $M$  is small, it's possible to go over all the cases systematically with pen and paper. Alternatively, we can have a brute force solution which uses BFS to go over the state space.

If at any point there are two vehicles of the same color in the same parking spot, we can ignore this parking spot for the rest of the process since these vehicles are in the correct spot. If two vehicles of the same color are not initially in the same parking spot, then we obviously need to spend at least one move for this color. Additionally, if both of these vehicles are top vehicles, we clearly need at least two moves. As we'll see later, there is one more case where we have to use an additional move (one of the cycle cases). We will present a solution which constructs a sequence of moves that achieves this bound, in case a solution exists.

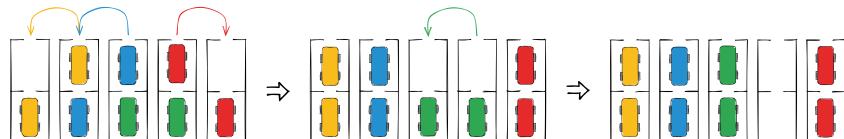
Each parking spot contains two positions (the bottom one and the top one). We'll create a graph with  $2M$  nodes representing the parking spot positions. The edges are defined as follows:

- if two vehicles of the same color are in different parking spots, we connect their positions,
- if a parking spot is full, we connect the bottom and the top position.

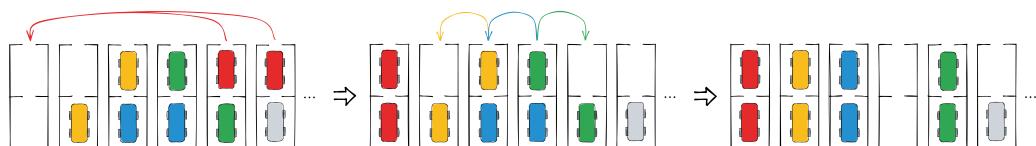
Each connected component of such a graph will be either a cycle, or a chain, where the ends of the chain are two bottom vehicles. If a connected component has two adjacent top nodes, we'll call such a pair a *top pair* for this connected component. Note that a top pair corresponds to two top vehicles of the same color in different parking spaces. We define the term *bottom pair* analogously.

We'll analyze chains and cycles separately. It turns out that each chain can be solved with at most one additional empty parking spot. Similarly, each cycle requires at least one empty parking spot and each cycle can be solved with at most two empty parking spots.

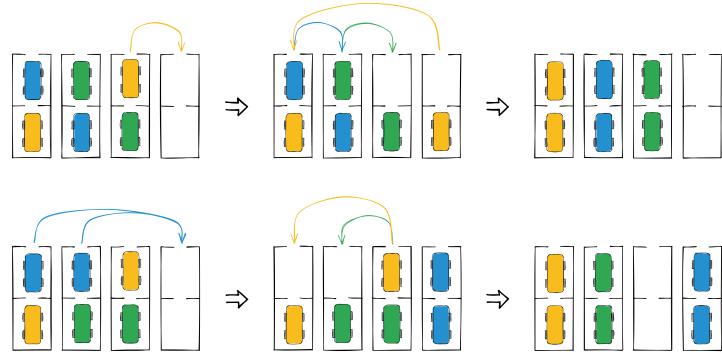
**Observation 1:** A chain with no top pair can be solved with no additional empty parking spots. Note that every chain must contain at least one bottom pair. In this case there is exactly one. The image bellow depicts how to solve such a chain.



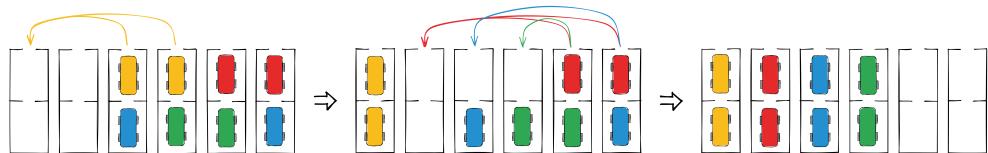
**Observation 2:** A chain with a top pair requires at least one additional empty parking spot. Furthermore, one parking spot is sufficient. Starting from one end of the chain, we can find the first time a top pair appears and move this top pair to the empty parking spot. We are now left with two chains, where one of them requires no additional parking spots. We first solve this chain, giving us an additional empty parking spot, and then continue solving the other chain. The image bellow depicts this process.



**Observation 3:** A cycle with at most one top pair can be solved with only one additional empty parking spot. If there are no top pairs, then the top vehicles of the cycle are a permutation of the bottom ones. We can solve this case by removing any top vehicle to an empty parking spot, reducing the problem to a chain as in observation 1. If the cycle contains exactly one top pair, it contains exactly one bottom pair as well. We move the top pair to an empty parking spot, reducing the problem to a chain as in observation 1.



**Observation 4:** A cycle with more than one top pair requires at least two empty parking spots. We can take any top pair, move it to an empty parking spot, reducing the problem to a chain as in observation 2.



Combining these observations yields a full solution. Note that we should prioritize solving chains, to solving cycles, because after solving a chain we gain an additional empty parking spot. Similarly, we should prioritize chains and cycles with a smaller number of top pairs.

The subtasks were intended to be solved by using only a subset of these observations, or by alternative approaches. Everything mentioned above can be implemented in  $O(N)$  or  $O(N \log N)$ . The subtasks where  $N \leq 1000$  allow for slower solutions with easier implementation.

Subtask 2 can be solved by first moving all top pairs to empty parking spots. There will always be enough space because  $2N \leq M$ . After that, we are left with only chains and cycles that can be solved as described in observations 1 and 3.

Subtasks 3 and 4 contain no chains. The following greedy approach works for this case:

- if at some point we can pair a vehicle with another one of the same color, we do it,
- otherwise, if there is a top pair, move it to an empty parking spot,
- otherwise, move any vehicle from the top to an empty parking spot.

If at any point there are no parking spots available, there is no solution. Note that this approach does not work for the other subtasks because, we should prioritize moving top pairs from chains.