Central European Olympiad in Informatics 2026
Day 2 - solutions

Flower Cutting

The edge growing process adds diagonals to every square (cycle on 4 nodes) and turns it into a clique on 4 nodes. If a part of the graph is sufficiently dense, it will grow into a clique. Any edge that is not a part of some clique can not be regrown and therefore can not be removed. Maximal cliques can't share an edge because they would merge into a larger clique. However, they can share nodes. We need to efficiently find maximal cliques in the graph and figure out how many edges are needed to start the regrowth of a clique of a given size.

subtask 1 There are very few edges, therefore we can consider removing all 2^M subsets of them and simulate the regrowing of the tree.

subtask 2 The graph is one large clique. We can derive a formula for the number of remaining edges in pruned cliques. Consider a pruned graph that will regrow into a clique of size c. Can we adapt this solution so that it will regrow into a clique of size $c+1$ or $c+2$? If we want to extend this graph with one node, we can attach it to any two existing nodes. The missing edges to all other nodes will regrow. However, if we want to extend it with two nodes, we can be more efficient with just three new edges — attach each of them with one edge to existing graph and use the third edge to connect them with each other. You can see that all other edges will again regrow. Therefore, the optimal structure can have a ladder structure. The number of remaining edges for a clique with c nodes is:

$$f(c) = \begin{cases} 3c/2 - 2, & \text{if c is even} \\ 3(c-1)/2, & \text{if c is odd} \end{cases}$$

Proving that this intuitive approach is indeed optimal is rather tricky. The easiest method to convince yourself is to simply submit a solution for this subtask.

subtask 3 The degree of each node is limited to 7, therefore the number of edges can't exceed $4N$. This allows less efficient searches for cliques in the graph (e.g. in $O(M^2)$) and non-explicit formulas for the number of edges in a pruned clique because the size of a clique can't exceed $\sqrt{M} = 2\sqrt{N}$.

subtask 4 This subtask allows potential less efficient polynomial solutions with higher exponents.

subtask 5 In this subtask we're dealing with a large number of edges, therefore we will need a efficient way of finding all maximal cliques. We also need to be careful that we don't consider the same clique multiple times. We can iterate over every edge (a, b) and look for cliques where this is the smallest edge. We need to find all common neighbors of a and b. Because the graph is a result of a completed growth, we know that they form a clique. We already know the minimum number of edges required to regrow a clique. To prevent considering the same clique multiple times, we can iterate over all edges that are part of the discovered clique and mark them as visited. This finds all cliques in $O(MN)$.

Towers

The task requires that we pair up all computers and connect each pair with a cable visiting potentially several towers to maximize the total sum of cable scores. We need to determine the optimal way of using the towers for connecting pairs of computers and the optimal way of pairing the computers. It makes sense to start by sorting the computers and towers (from left to right).

subtask 1 There is only one tower in this subtask. For a given pair of computers, we can check what is more efficient: a direct connection or a connection via the tower. This depends on the distance from the tower to the closest computer in the pair.

We claim that it is optimal to pair up adjacent computers in a sorted order. Consider two overlapping pairs of computers, which should not be necessary for an optimal solution. If neither of the pairs uses the tower, we can obviously reduce the distance and improve the score by changing the pairing. If both pairs connect to the tower, the score depends on the sum of distances from computers to the tower, therefore it doesn't matter how we pair them at all. The final case is a pair that connects to a tower and another that doesn't. The pair that doesn't must be further from the tower, otherwise it would connect to it. We can again see that changing the pairing reduces the sum of distances.

subtask 2 The second subtask has very few computers, where we can afford to consider all possible pairings. It also makes sense to precompute the optimal tower connections for all n^2 pairs of computers. We can simply consider all possible leftmost and rightmost towers. It is obvious that the optimal cable that minimizes the length will start at the left computer, visit the leftmost tower, continue to the rightmost tower and back to the right computer, while visiting all towers in between.

subtask 3 We need to be more efficient in computing optimal connections via some towers. Observe that the cable from computer a to b will have a score comprised of the distance (and towers) between them and the contribution from an extension to the left from a and an extension to the right from b. For every computer we precompute the optimal extensions to the left and to the right (l_i and r_i) in $O(nm)$. This enables us to compute the cost of an optimal cable between two computers in $O(1)$.

Now we have to pair the computers optimally, which we can do with dynamic programming in $O(n^2)$. We will build the pairs from left to right and keep track of how many computers are still unpaired. Let $f(i, u)$ define the maximum score to pair the first i computers so that there are still u unpaired among them (this includes the score of the unpaired computers up to the i-th one). The i-th computer could start a new pair or close some unpaired computer (note that it doesn't matter which one exactly). If it starts a new pair, the score corresponds to $l_i - (u-1)c_i + f(i-1, u-1)$, because $u-1$ unpaired computers span the gap between computers $i-1$ and i, with the contribution c_i for each of them (comprised of the distance and the number of towers in between). Similarly, if it closes an unpaired computer, the score corresponds to $r_i - (u+1)c_i + f(i+1, u+1)$, because there had to be $u+1$ unpaired computers among the first $i-1$.

subtask 4 We can be more efficient in the precomputation of optimal extensions to the left (l_i) and right (r_i) from each computer. The left extension from computer i could stop at some tower between computer $i-1$ and i. If it extends further left than computer $i-1$, we already know the optimal extension for that case — l_{i-1}. This allows us to precompute these values in $O(n+m)$ instead of $O(nm)$ as in the previous subtask.

subtask 5 Let us consider all computer as starting/left ends of the pair: make an optimal left extension from each computer and draw the cable all the way to the far right edge. Then we have to pick some closing/right computer. This adds an optimal right extension, subtracts its previous starting orientation and the contribution from the cable to the far right that it closes. Note that this change d_i doesn't depend on the location from where the cable that it closes actually starts.

The problem reduces to selecting $n/2$ elements with the maximum sum in an array of changes d_i such that there are at most $i/2$ selected elements in every prefix of length i. This problem has an efficient greedy solution. We maintain a set of the $\lfloor i/2 \rfloor$ selected elements from the first i elements. When we process the next element, we add it to our selected set and if the set exceeds the new capacity, we eject the smallest selected element. At the end, we are left with the optimal solution. We can implement this

greedy algorithm in $O(n \log n)$ by storing the selected elements in a heap. The proofs of optimality are rather cumbersome or rely on advanced techniques, but the algorithm is easy to implement and test with a submission to the judge system.

VIM

We need to find the minimum number of moves to produce a string consisting of n characters starting from a single one.

subtask 1 The state can be described by the current number of characters in the string, the location of the cursor and the content/size of the clipboard. Note that it is more convenient to describe the cursor location as an offset from the right end of the string. This way the location increases by 1 independent of the size of the inserted content. This makes for $O(n^3)$ states and we need to consider 4 operations every time. A simple breadth-first search should suffice for small values of n.

subtask 2 Let us make some observations to decrease the number of considered states. First, it seems that there is no need to ever move the cursor to the right. Second, we can produce a string of length $O(\sqrt{n})$ by copying and pasting a single character. Then copy the entire string and paste it $O(\sqrt{n})$ times. This means that we can produce n characters in $L = O(\sqrt{n})$ operations. Therefore, it doesn't make sense to explore states where the location of the cursor exceeds our upper-bound estimate L (since we could already produce a solution in the number of operations that are required to get the cursor into a position). Consequently, the size of the clipboard cannot exceed L either. This way we can reduce the state-space to $O(n\sqrt{n}\sqrt{n}) = O(n^2)$.

subtask 3 If we observe the solutions for increasing string lengths n, we will notice that they consist of blocks that start with some moves to the left, a copy and a sequence of paste operations. At some point the $O(n^2)$ space complexity becomes a problem because we need to reconstruct solutions. To circumvent this, we can reduce the number of states to $O(n\sqrt{n})$ that represent the size of the string and the cursor location. Transitions between states consist of a sequence of $O(\sqrt{n})$ moves to the left, followed by a copy and a sequence of $O(\sqrt{n})$ pastes. We can optimize each of these two steps (number of moves and number of pastes) independently to get a solution with $O(n^2)$ time complexity that requires just $O(n\sqrt{n})$ space.

subtasks 4-6 To handle larger problems we can again observe the solutions that we are able to compute so far. For small values of n, the solutions behave rather unpredictably. However, for larger values (for example, $n > 1000$), the solutions become nicely increasing. We can also observe that the left moves become irrelevant. We can solve the small cases with previously presented approaches and focus on these larger values. Let us determine the longest string $f(m)$ that we can produce in m moves. Remember that we measure the offset of the cursor from the right side and a paste operation actually increases this offset by 1. To produce the longest string we can clearly ignore any moves to the right as well as moves to the left (because we can achieve the same or more with a paste operation). The optimal sequence will therefore consist of k groups of a copy and several paste operations (YP...P). Let n_i denote the number of paste operations in group i.

$$\begin{aligned} m &= n_1 + n_2 + ... + n_k + k \\ f(n_1, \ldots, n_k) &= 1 + (1)n_1 + (1 + n_1)n_2 + (1 + n_1 + n_2)n_3 + ... \\ &= 1 + \frac{(m-k)^2}{2} + (m-k) - \frac{\sum {n_i}^2}{2} \end{aligned}$$

We can see that the order of groups is irrelevant, just their sizes play a role in the length of the produced string. For a fixed number of groups k, we need to maximize $\sum {n_i}^2$ subject to $n_1 + n_2 + ... + n_k = m - k$. Ideally, n_i would be the same and equal to $(m-k)/k$. Round this value down and distribute the remainder by adding 1 to some terms n_i. We can consider increasing values of m until we find one where $f(m) \geq n$ (we know that the number of required moves is $O(\sqrt{n})$). To determine $f(m)$, we can again simply consider all values of $k \in [1, m/2]$. This gives us an $O(n)$ solution. Note that we could be more efficient to get a sublinear solution, but there is no need. We have described how to determine the magnitude of the solution but not how to actually construct it, which gives us half of the points.

We know that the solution requires m moves. However, the strategy for constructing the longest string might produce a solution that is longer that n. Our goal is to change the values of n_i in such a way that $f(n_1, \ldots, n_k)$ drops to exactly n. One possible strategy is the following. Note that if $n_i = n_j$, decreasing one of them by 1 and increasing the other one actually decreases the solution by 1. It would be beneficial if we had many groups of equal size to give us fine-grained control over the solution. We can start by trying to increase the number of groups in the longest solution as long as it still generates a sufficiently long string. This gives us more groups and decreases the gap. Then, we repeatedly move a paste operation (decrease n_i) from the second largest to the largest group. This effectively increases the largest group as much as possible while maintaining a similar size of the other groups. If an additional increase of the largest group would make the solution too short, we ignore the largest group from there on and try to optimize the length of the solution with the remaining groups that are hopefully of similar size. The process doesn't have to be particularly efficient because we are dealing with relatively few groups. You can test this or any other strategy on a range of values n and check that they indeed manage to adjust the longest solution $f(m)$ to the length of exactly n. We can also formally prove that for sufficiently large n ($n > 10\,000$), this approach always finds a solution.