CS3233 Final Contest 2025 Editorial

Hua Jun	Xing Chen	Benson	Rama
huajun	lovemathboy	benson1029	rama_pang

Maxi	Hong Duc	Yu Peng
tzaph_	user202729	feeder1

14 April 2025

	Title	# Solves	First Solve
A	Anti-Diagonal Game	12 teams	14 minutes
B	BrIIIIance of Wings	1 team	172 minutes
C	Chimchar Defense	9 teams	22 minutes
D	Double String	1 team	296 minutes
E	Energy Extraction	0 teams	
F	Fair Forgery	1 team	206 minutes
G	Game of Two Choices	5 teams	97 minutes
H	Help Eevee Pls Eh	11 teams	28 minutes
I	Independent Inversions	6 teams	98 minutes
J	Job Interview	0 teams	
K	Kanto To Johto	12 teams	29 minutes
L	Last Goal	5 teams	73 minutes
M	Miracles can be Created	12 teams	8 minutes

A. Anti-Diagonal Game

Let's fix the string S and determine Alice's winning condition. Suppose that N is odd, so Alice will make the last move. Let $N = 2K + 1$.

Claim. *If one of* $(K, K+1)$ *or* $(K+1, K)$ *is a winning position for Alice, then Alice wins.*

Proof. WLOG, $(K, K+1)$ is a winning position for Alice. In the first move, Alice moves the token to $(0, 1)$. After this, Alice moves the opposite of Bob's last move (i.e., if Bob moves the token by incrementing i, then Alice moves the token by incrementing j; and vice versa). This way, Alice will "force" the token to go to $(i, i+1)$ every time Alice moves, eventually reaching $(K, K+1)$, which is a winning position for Alice. □

Claim. *If both* $(K, K+1)$ *and* $(K+1, K)$ *are losing positions for Alice, then Alice loses.*

Proof. Bob always moves opposite of Alice's last move. After Bob's final move, the token will be at (K, K), and no matter what Alice does, the token will end up at a losing position for Alice. □

Thus, for odd N, Alice wins depending only on the values of (A_K, A_{K+1}).

For even $N = 2K$, after Alice's first move, Bob wins only depending on the values of either (A_{K-1}, A_K) or (A_K, A_{K+1}). So Alice wins depending only on the values of (A_{K-1}, A_K, A_{K+1}).

We can easily check which values (A_K, A_{K+1}) for odd N, or values (A_{K-1}, A_K, A_{K+1}) for even N in which Alice wins by either writing a brute-force program or by hand. Then, the remaining values can be either 0 or 1 freely.

In the end, the final answer is $4^{3\times\left\lfloor\frac{N-1}{2}\right\rfloor}$.

B. BrIllIance of Wings

Let T be the original tree and T' be the new tree. Let $T \cap T'$ be the set of edges that are in both trees.

Claim. *The minimum number of operations is* $N - 1 - |T \cap T'|$ *and the operations done are removing edges from* T *that are not in* T' *and adding edges to* T' *that are not in* T*.*

Proof. Proven in midterm, and by now this should sound intuitive to you. □

We are tasked to construct such order of operations.

Arbitrarily root the tree T and start from the leaves of T. Consider the leaf u with parent p, and check if edge (u, p) needs to be replaced. If it does, we find another edge (x, y) in T' such that during the operation, x is in one connected component with u, and y is in one connected component with p. This is the high-level idea of the solution, but finding this (x, v) requires more details.

We maintain the connectivity between vertices already joined by using a Disjoint Set Union (DSU), but we cannot simply only store the DSU parent of each node. We need to store what are the edges bridging out from nodes in our connected component (i.e. store all (u, v) where u is in our connected component), and in an operation select one edge such that v is not in the connected component.

This may sound exhaustive, but with proper analysis we can prove that this is fast enough.

Initially, we store all edges connecting to u in its own connected component (no vertices have been joined yet). When we are to go through the edges in the original tree (starting from the leaves), if (u, p) is already in T', we can simply join u and p in the DSU data structure. We also store all edges of u and p in this DSU component. To do this effectively, simply set the vertex with a larger edge set to be the DSU parent of the other, and move edges from the smaller set to the larger set.

If (u, p) is not in T', we iterate through all edges in the DSU component of u, checking if the edge (x, y) satisfies the requirement. With the way we set up our algorithm, x must be in one DSU component with u, so we must check if y is not in one DSU component with u. If a y is in one DSU component with u, edge (x, y) is no longer important (as the edge is in the component) and we can remove this edge. If we found a y that is not in one DSU component with u (which must exist given our argument to the lemma), we join u and y in a similar matter.

Repeat this until we reach the root of T.

Now, let us analyze the time complexity of our algorithm. The algorithm looks heavy from the idea of removing unimportant edges and merging edge sets.

Notice that the edge set will only be as large as the size of the tree itself ($O(N)$), so the number of times we have to remove unimportant edges is also $O(N)$ in total, regardless on when we do this removal in the DFS traversal. Hence, removing unimportant edges are not a problem.

Now, consider the merging of two edge sets. We established the fact that we should move edges from the smaller set to the larger set. We can formulate the time complexity of the algorithm to be $T(N) = T(small) + T(N - small) + O(small)$ where $T(X)$ represents the recurrence onto the subtrees and $O(small)$ represents the pushing of small number of elements from the smaller set to the larger set. As $|small| \leq \frac{N}{2}$ (we can't have two edge sets with size more than $\frac{N}{2}$, hence the smaller set must have a size less than $\frac{N}{2}$), we have $T(N) = 2T(N/2) + O(N/2)$ in the worst case, and we know that this yields $T(N) = O(N \log N)$. Hence, the merging of sets in this way is efficient.

The final complexity is $O(N \log N)$ from the DSU set merging.

Note: This set merging technique is known as "small-to-large" technique in CP (rarely known as Sack DSU, DSU on Tree, etc.). You can read more of that in the USACO blog linked. Also, read the references linked in the USACO guide for more advanced exploration.

C. Chimchar Defense

Notice that if we process the chimchars from left to right, it is difficult to determine how much damage a chimchar will take as it is dependent on piplups that have yet to be processed. Let us instead process the problem from the back to the front and try to formulate a DP solution from there. As mentioned, we need to keep track how much damage a chimchar will take, to do this we can simply keep track of the amount of cumulative damage from the chosen piplups. Let $dp[i][d]$ be the maximum score (damage - cost) we can achieve only considering the piplups and chimchars in range $[i, N]$, such that the total damage of piplups used is d. We can then get the following relation:

$$dp[i][d] = \max\{dp[i+1][d], dp[i+1][d - D_i] - C_i\} + \min\{d, H_i\}$$

Where at each state, we have the choice to either fire or not fire the i-th piplup at a cost of C_i. Note that there are $N(\max H_i + \max D_i)$ states since it is never optimal do more than $(\max H_i + \max D_i)$ damage and with transition of $O(1)$. Leading to a $O(N(\max H_i + \max D_i))$ solution.

D. Double String

We construct the suffix array sa and the LCP array lcp. We denote $sa[i]$ as the i-th lexicographically lowest suffix, $lcp[i]$ as the longest common prefix between the suffixes starting from $sa[i]$ and $sa[i+1]$, and the function $lcp(i, j)$ as the longest common prefix between the suffixes starting from indices i and j. In particular, we can note that

$$lcp(i, j) = \min_{sa^{-1}[i] \leq k \leq sa^{-1}[j]-1} lcp[k]$$

The substring $S[i_1, j_1]$ is a double string if and only if it has even length, and:

- Let $k = (j_1 - i_1 + 1)/2$
- $lcp(i_1, i_1 + k) \geq k$

In other words, there is a bijection between "$S[i_1, j_1]$ is a double string" and "indices (i_1, j_2) satisfy $lcp(i_1, j_2) \geq j_2 - i_1$".

Therefore, if we can count the number of indices (i, j) satisfying $lcp(i, j) \geq |i - j|$, we have counted the number of double strings. We call these pairs of indices *good pairs*.

Consider the LCP array. Note that $lcp(i, j)$ can be computed by a range-minimum of the interval between $sa^{-1}[i]$ and $sa^{-1}[j]$. This motivates us to do divide-and-conquer on the LCP array in the following way:

- Suppose we now want to compute the good pairs with $L \leq sa^{-1}[i] < sa^{-1}[j] \leq R$.
- Consider $M = \arg\min_{L \leq k < R} lcp[k]$.
- Notice that for any $x \in \{sa[L], sa[L + 1], \ldots, sa[M]\}$ and $y \in \{sa[M + 1], sa[M + 2], \ldots, sa[R]\}$, we have $lcp(x, y) = lcp[M]$.
- In time $O((\min\{M - L, R - M\}) \cdot F)$, we count good pairs (x, y) where x belongs to the left half and y belongs to the right half, then recurse to each half independently, where F is some additional log factor that we might need.

It can be proven that if we only take $O(\min\{M - L, R - M\})$ to process each interval, the overall complexity is $O(NF \log N)$. The proof is left as an exercise to the reader (hint: instead of consider the process of splitting intervals, consider the process in reverse of merging these intervals together).

To process the interval, suppose we have some data structure D that encompasses the information of the whole interval. WLOG, $M - L \leq R - M$. Then, we can construct a new D_L for the left half, remove the left half indices from D, and use D for the right half. The data structure D should be used to count the indices y in $O(F)$ time by fully iterating each x in the left half.

We can use any data structure that maintains order statistics: e.g., a BBST with order statistics or a variable-length segment tree or fenwick tree for the data structure D. The most straightforward to do this would be to leverage GNU's Policy-Based Data Structure for a ready-use BBST with order statistics. With this, we can process with $F = O(\log N)$.

Therefore, the total time complexity is $O(N \log^2 N)$. `https://www.sciencedirect.com/science/article/pii/S0022000004000364` may also be of interest.

E. Energy Extraction

See proposition 19 in section 3 of the 2nd result in `https://arxiv.org/search/?searchtype=author&query=Bui%2C+H+D&order=announced_date_first` (only available from 2025 April 15, 8am SGT), written by the author to this problem. Section 6.3 and appendix 7.1 may also be of interest.

F. Fair Forgery

In this solution, we will refer to candidates as numbers for convenience.

We first compute how many times each number must appear in the top l positions in the output permutations for every l. Note that if a number appears t times in the l highest ranks in the M given permutations, then it must appear at least $\left\lfloor \frac{t \cdot K}{M} \right\rfloor$ in the highest l ranks in the output permutations. Thus we may compute $c_{i,j}$: the number of times i must appear in the top j positions in the output permutations, for each $1 \leq i \leq N, 1 \leq j \leq N$.

An easier problem

Consider the K output permutations as a $K \times N$ table i.e. K rows, N columns, then the highest l ranks in a permutation would correspond to the leftmost l columns in this table. We first try to construct this table such that every number appears K times in this table, but each row is not necessarily a permutation, and the following is satisfied:

(*) For all integers $1 \leq t \leq K$, $1 \leq l \leq N$, if a number appears in the highest l ranks in at least $\lceil \frac{t \cdot M}{K} \rceil$ of the M received votes, then he/she should appear in the leftmost l columns at least t times.

The main point of this rephrasing is that if a number appears a times in the leftmost l positions in a row, this would count as a appearances in the leftmost l columns. Constructing this table is a strictly easier problem than the original problem, which is why we try to do this first.

Constructing the table

Intuitively, we just perform a greedy algorithm: greedily fill the columns of the table from left to right, putting the numbers that need to be in column 1 the specified number of times, followed by the numbers that need to be in the top 2 ranks, etc. Formally, for each $1 \leq i \leq N$, define $c_{i,0} = 0$ for convenience, and then for each $1 \leq j \leq N$ let $d_{i,j} = c_{i,j} - c_{i,j-1}$. We then perform the following, filling the columns from left to right:

for $j = 1$ to K:

  for $i = 1$ to N:

    fill $d_{i,j}$ cells with i

We will show this constructs a table satisfying the conditions we want.

Proof. At the end of the $j = l$ iteration, we would have filled $c_{i,l}$ occurrences of i for each $1 \leq i \leq N$. We wish to show that we have not filled anything past column l at this point.

Note that i appears at least $\frac{c_{i,l} \cdot M}{K}$ times in the top l ranks in the original permutations, and there are in total $M \cdot l$ positions which are considered the top l ranks in the original permutations. Hence we have

$$\sum_{i=1}^{N} \frac{c_{i,l} \cdot M}{K} \leq M \cdot l \implies \sum_{i=1}^{N} c_{i,l} \leq l \cdot K$$

which means that we've filled at most $l \cdot K$ numbers at this point, and thus have not exceeded column l. Thus the table constructed by this process fulfills (*). Also note that $c_{i,N} = K$ for all $1 \leq i \leq N$ and thus at the end of the process, every number will appear K times in the table. Hence the table fulfills all the properties we want. □

Constructing the solution

Now, we would like to permute each column of this table such that each of the table forms a permutation from 1 to N. Again, we aim to be less ambitious first: let's try to permute the columns such that the **first** row forms a permutation. Suppose that in this desired state, for each $1 \leq i \leq N$, i appears in position m_i in the first row. Then in the table we constructed, i must appear in column m_i, and m_i must form a permutation of 1 to N.

Hence to do this, consider a bipartite graph with N vertices on each side numbered 1 to N, such that there is an edge from i on the left side to j on the right side if and only if number i appears in column j. Then the pairing (i, m_i) above corresponds to a perfect matching in this bipartite graph.

Now we claim that a perfect matching definitely exists in this graph.

Proof. We use Hall's marriage theorem. Consider a subset of vertices A on the left side, and let B be the neighbour set of A on the right side. Note that for each $i \in A$, i appears at least K times in the table. Hence the numbers in A appear a total of $K|A|$ times in the table, and since each column has only K numbers, the numbers must appear in at least $|A|$ columns. Thus $|B| \geq |A|$, so Hall's condition is satisfied, and a perfect matching exists. □

Thus we may permute the columns to make the first row a permutation of 1 to N. Now notice that in the remaining $K-1$ rows, each number now appears $K-1$ times, so we may actually repeat the above argument to permute the columns in the remaining $K-1$ rows to make the second row a permutation, and so on.

Hence the problem can be solved by doing bipartite matching K times. In each iteration, the graph has $2N$ nodes, and since each number appears at most K times in the table (less in iterations after the first), there are $O(NK)$ edges.

Therefore, each bipartite matching takes $O(N^2K)$ with Kuhn's or $O(N^{1.5}K)$ with Hopcroft-Karp, so K matchings take $O(N^2K^2)$ or $O(N^{1.5}K^2)$ time in total.

Computing $c_{i,j}$ and constructing the table takes $O(NM)$ time, so overall this solution is fast enough to pass.

Remark. If we think about the bipartite graph corresponding to the initial $K \times N$ table we constructed, we realise that it is actually a K-regular bipartite multigraph if we draw an edge between i on the left and j on the right each time i appears in column j. Having each row being a permutation would then correspond to having K disjoint perfect matchings on this graph, which is equivalent to finding a K-edge colouring on this bipartite multigraph. There are faster ways to do this, for example shown in this blog and references linked inside: `https://codeforces.com/blog/entry/75431`.

G. Game of Two Choices

Let $ans[u]$ be the maximum possible score if the game starts at vertex u, or ∞ if the game can be played indefinitely. Let $\infty + 1 = \infty$ for convenience. Note that

- If $|N(u)| = 0$, then certainly $ans[u] = 0$.
- If $|N(u)| = 1$, then $ans[u] = ans[v] + 1$.
- If $|N(u)| = 2$, then suppose that when the game starts at vertex u, you choose the vertices v_1, v_2. Since SoCCat wants to minimise the game score, their optimal play is to pick the vertex that gives the minimum between $ans[v_1], ans[v_2]$. Therefore, your optimal play is to pick v_1, v_2 with the largest two ans values, and we would then have $ans[u] = \min(ans[v_1], ans[v_2]) + 1$.

To compute the ans values for all vertices, it turns out that a BFS-like solution works. Throughout this process, we will call a vertex *unprocessed* if we haven't determined its ans value. We'll keep layers for the vertices for the sake of proving correctness, in which we aim to have vertices u with $ans[u] = k$ in layer k, and those with $ans[u] = \infty$ in no layer. We will also keep another counter $deg[u]$ for every vertex u in this process, which represents the number of **unprocessed** vertices v such that there is an edge $u \to v$.

Firstly, add all vertices with outdegree 0 to layer 0, and set $ans[u] = 0$ for all these vertices u. It's clear that these are all the vertices u with $ans[u] = 0$, since if a vertex has at least one out-edge, the game starting at that vertex cannot immediately end.

We now iterate $i = 0, 1, \ldots$, and stop when layer i is empty. When we are at layer i, for each vertex u in the layer, for each **unprocessed** vertex v such that there is an edge $u \to v$, decrement $deg[v]$. We call this action *visiting* u for convenience. If $deg[v] \leq 1$, set $ans[v] = i + 1$, and add vertex v to layer $i + 1$.

At the end of the process, for all vertices u not in any layer, set $ans[u] = \infty$.

Intuitively this algorithm works for similar reasons as why BFS works, but we will prove it anyway. We induct on k to show the following: if layer k is not empty, or is the first empty layer, then for each $i = 0, 1, \ldots, k$,

for each vertex u, $ans[u] = i \iff u$ is in layer i. This statement is true for $k = 0$ as we've previously discussed. So suppose the statement is true for some k, then consider an unprocessed vertex u. By our inductive hypothesis, $ans[u] \geq k + 1$. We split cases:

Case 1: $ans[u] = k + 1$. In this case, it means that among the vertices v such that there is an edge $u \to v$, at most 1 of them has $ans[v] > k$. Thus, after we are done with iterating through layer k, u must be added to layer $k + 1$, since $deg[v] \leq 1$ after this iteration.

Case 2: $ans[u] \geq k + 2$. If u has outdegree 1, to a vertex v, then $ans[v] = ans[u] - 1 \geq k + 1$. Then after we are done with layer k, we would still have not visited v, and thus we also have not yet added u to any layer. Otherwise, if u has outdegree ≥ 2, among all vertices v such that there is an edge $u \to v$, let v_1, v_2 be the ones with the two largest ans values. As $ans[u] \geq k + 2$, $ans[v_1], ans[v_2] \geq k + 1$. Therefore, after we are done with layer k, vertices v_1, v_2 would still be unvisited, and hence $deg[u] \geq 2$, so u is not added to any layer yet.

Thus, this shows that a vertex u is in layer $k + 1$ if and only if $ans[u] = k + 1$, completing the induction.

Now suppose layer k is the first empty layer. Then note that for each vertex u such that $ans[u] \neq \infty$, there exists a vertex v such that there is an edge $u \to v$, and $ans[v] = ans[u] - 1$. Our induction shows that layer k being empty means there aren't any vertices u with $ans[u] = k$. Consequently there aren't any vertices u with $ans[u] \neq \infty$ but $ans[u] > k$. Thus the algorithm is correct.

H. Help Eevee Pls Eh

As mentioned in the question statement, we have the following equivalent definition of a palindrome:

- Consider the $\lfloor \frac{n}{2} \rfloor$ pairs of characters $(s_1, s_n), (s_2, s_{n-1}), \ldots, (s_{\lfloor \frac{n}{2} \rfloor}, s_{\lfloor \frac{n}{2} \rfloor + 1})$.
- The number of pairings above that are of identical characters should be $\lfloor \frac{n}{2} \rfloor$.

Define the string after removing character s_x as $S_x = s_1 \ldots s_{x-1} s_{x+1} \ldots s_n$. Construct the $\lceil \frac{n-1}{2} \rceil$ pairs of S_x, let this set of pairs be P_x. We notice that P_x and P_{x+1} actually only differs by one pair. WLOG assume $x + 1 \leq \lfloor \frac{n}{2} \rfloor$, in P_x is then paired with (s_{x+1}, s_{n-x-1}), but in P_{x+1}, all the other pairs remain the same but (s_{x+1}, s_{n-x+1}) changes to (s_x, s_{n-x+1}) instead. As such for each x, we can simply maintain a counter of the number of identical pairs and as we move x, update that single pair and update the counter accordingly and check if its equivalent to $\lfloor \frac{n-1}{2} \rfloor$. This will run in $O(N)$.

There are other (possibly better) solutions involving more observations or just plain hashing or manacher, etc.

I. Independent Inversions

There are quite a few possible solutions to this problem.

Solution 1 – $O\left(N^{1.63\ldots}\right)$

For each index number i, the number of possible independent pairs of indices (i, j) is 2^K. Hence, we can brute force all possible independent pairs of indices in $O\left(N \times 2^K\right) = O\left(N^{1+\log_3 2}\right) = O\left(N^{1.63\ldots}\right)$ time.

An efficient implementation can be done via divide & conquer. Define $dnc(k, x, y)$, where the last k ternary bits of x and y are independent, the other ternary bits of x and y are empty. Then, we can iterate all possible

combinations of the current bit:

$$\begin{aligned} dnc(k,x,y) = &\ dnc(k+1,x,y+3^k) \\ &+ dnc(k+1,x,y+2\times 3^k) \\ &+ dnc(k+1,x+3^k,y) \\ &+ dnc(k+1,x+3^k,y+2\times 3^k) \\ &+ dnc(k+1,x+2\times 3^k,y) \\ &+ dnc(k+1,x+2\times 3^k,y+3^k) \end{aligned}$$

At the end, we can check whether $x < y$ and $B_x > B_y$ to compute the number of independent inversions.

Solution 2 – $O\left(N^{1.31\ldots}\right)$

Instead of computing all independent pairs of indices, process B_i from smallest to largest. Denote this new array as Re with $Re[B_i] = i$. Then, the problem is reduced to finding $(Re[i], Re[j])$ pairs such that $Re[i] < Re[j]$, $i > j$, and $Re[i]$ and $Re[j]$ are independent.

Apply meet-in-the-middle ideas: Split each number into first and second $\frac{K}{2}$ ternary bit halves. Define $S[x][y]$ as the number of $Re[i]$s before the current index i where x and the first $\frac{K}{2}$ ternary bits of $Re[i]$ are independent, and y is equal to the second $\frac{K}{2}$ ternary bits of $Re[i]$. For each update, we $2^{\frac{K}{2}}$ brute force the first half; for each query, we $2^{\frac{K}{2}}$ brute force the second half.

Denote $Re[i]_x$ as the first $\frac{K}{2}$ digits of $Re[i]$ and $Re[i]_y$ as the second $\frac{K}{2}$ digits of $Re[i]$. Iterate from $i = 0$ to $N-1$, the number of valid $(Re[j], Re[i])$ pairs (with $j < i$) is obtained by summing $S[Re[i]_x][y]$ over all $2^{\frac{K}{2}}$ options of y. (The $2^{\frac{K}{2}}$ values of y that are independent with $Re[i]_y$) Then, we update S by incrementing all $S[x][Re[i]_y]$, where x and $Re[i]_x$ are independent and $x < Re[i]_x$.

Overall, the algorithm runs in $O\left(N \times 2^{\frac{K}{2}}\right) = O\left(N^{1+\frac{1}{2}\log_3 2}\right) = O\left(N^{1.31\ldots}\right)$

J. Job Interview

Note that problem is basically equivalent to the following DP:

$$dp[i] = \min_{j<i, j\notin S_i} dp[j] + cost(j,i)$$

Where $cost(j,i) = (i-j) * (A[i] - A[j+1])$. Also, S_i differs by S_{i+1} by one element (either removed or added) and this set is computed online. Let $f(j,i) = dp[j] + cost(j,i)$

Observation 1: If $f(j,i) > f(k,i)$ for $j < k < i$. Then $f(j,i+1) > f(k,i+1)$.

Proof. $dp[j] + cost(j,i) > dp[k] + cost(k,i) \iff dp[k] - dp[j] < cost(j,i) - cost(k,i)$

$$\begin{aligned} cost(k,i+1) - cost(k,i) &= (A[i+1] - A[k+1])(i+1-k) - (A[i] - A[k+1])(i-k) \\ &= (A[i+1] - A[k+1]) + (A[i+1] - A[i])(i-k) \\ &< (A[i+1] - A[j+1]) + (A[i+1] - A[i])(i-j) \\ &= cost(j,i+1) - cost(j,i) \\ \therefore cost(j,i) - cost(k,i) &< cost(j,i+1) - cost(k,i+1) \end{aligned}$$

$$\therefore dp[k] - dp[j] < cost(j,i+1) - cost(k,i+1) \iff dp[k] + cost(k,i+1) < dp[j] + cost(j,i+1)$$

□

Using this observation we notice that between every index $j < k$, there exists a point $t_{j,k}$ such that for $i \leq t_{j,k}$ j is a more optimal ($f(j,i) < f(k,i)$) dp transition, while for $i > t_{j,k}$ k is more optimal. We can compute this value in $O(\log N)$ with binary search.

Thus, we can construct and maintain a segment tree of possible dp transitions indices. For simplicity, we shall first assume that $S_i = \emptyset$. The j-th leaf node will represent $f(j,i)$ where i is the current index we are processing. In the intermediate nodes we can then retrieve the current optimal index in the left and right child nodes, let them be opt_l and opt_r. We can then compute t_{opt_l,opt_r} and check if $i \leq t_{opt_l,opt_r}$ then the optimal index for the intermediate node is opt_l, otherwise it is opt_r. For those nodes with $i \leq t_{opt_l,opt_r}$, we can maintain some DS such that when $i = t_{opt_l,opt_r} + 1$, we can update this node to change its optimal index from opt_l to opt_r. Thus, with this maintenance, the optimal index in the root node would be the optimal index for i and we can compute $dp[i]$ in $O(1)$. However, note that when we update a node we will also have to update all its ancestors.

Observation 2: The total number of times we will update is bounded by $O(N \log N)$.

Proof. Let's consider a node that covers the range of indices $[l, r]$. Let us consider the sequence $\{opt_0, \ldots, opt_k\}$ to be the optimal index of this node as it changes through the execution of this algorithm. Note that for $i \neq j, opt_i \neq opt_j$. Since if opt_i is replaced by another index opt', then by observation 2 for all following i, opt' would still be more optimal. As such the sequence length is bounded by $(r - l + 1)$ and the optimal index of a node can only change $(r - l + 1)$ times.

Now, we note that a node will only be updated if one of its children optimal values are updated or if its own optimal value is to be updated. As such from our previous observation, we can see that each node will be updated $O(r - l + 1)$ times.

Summing this across all nodes in the segment tree, we get a sum of $O(N \log N)$. □

For each change to S_i, if it is disabling an index, we can simply update the leaf node to have an invalid index and intermediate nodes would ignore this invalid index. For re-enabling an index, we will just revert the leaf node back to its original value and update its ancestors. Note that this operation will take $O(\log^2 N)$ as $O(\log N)$ nodes are updated, also the size of the sequence of optimal index changes is now bounded by $O(r - l + 1 + \text{update in subtree})$, thus resulting in an additional $O(\log N)$ updates overall. As there are N updates, this will not affecting the amortised complexity in observation 2.

Therefore, as each update takes $O(\log N)$ to compute t_{opt_l,opt_r}, the total amortised complexity is $O(N \log^2 N)$.

Note: This question was actually motivated as a variation of 1d1d optimisation with updates.

K. Kanto To Johto

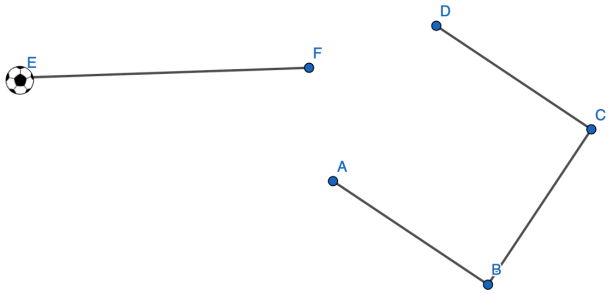
We call passes edges for convenience. Note that if we buy $\geq K$ edges, then we will not be worse off by buying the K edges with smallest costs in the connected component station 1 is in. We can buy all edges in the connected component by simply traversing through the entire component.

Otherwise, if we buy $< K$ edges, then the cost is at least the cost of the path we take from station 1 to N, which in turn is at least the path with the minimum total cost from station 1 to N.

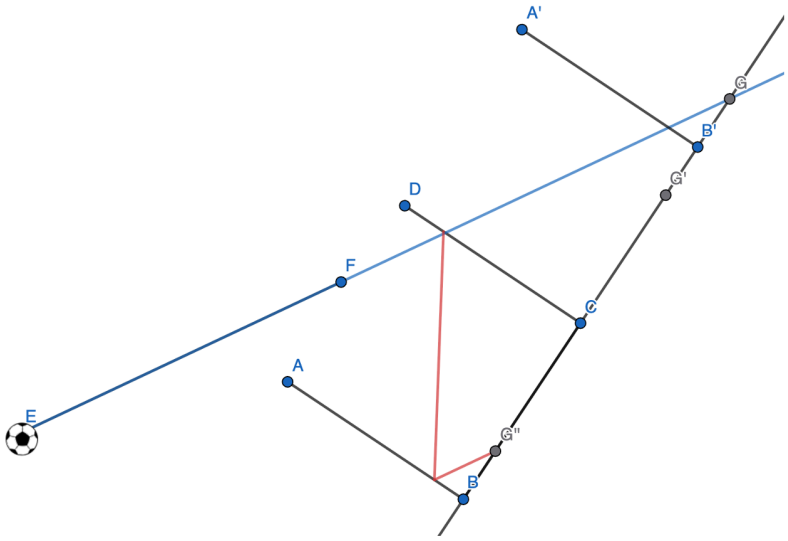
Hence the answer is the $\min(A, B)$ where A = sum of smallest K costs of edges in the connected component station 1 is in, and B = minimum total cost of a path from 1 to N. To compute A, we can DFS to determine which edges are in the same connected component as station 1, sort them by cost, and sum up the minimum K. To compute B, perform Dijkstra from station 1.

L. Last Goal

A simple geometry problem. Let the problem be described by the following image: The goal is ABCD and the path of the ball is EF.



Instead of computing the intersection of each reflection (where there can possibly be a large number of reflections). We can instead pretend that the ray is not reflected but the goal is reflected instead (A'B'CD) and keep doing this until the ray hits line BC.



We then need to compute the maximum of BG'' or $G''C$. To do this we can compute the distance of GB and take the fmod of the distance (or use other functions like division and flooring) to get a distance $d < |BC|$. We can then output $\max(d, |BC| - d)$.

M. Miracles can be Created

Let's solve a simpler problem. How do we calculate $0 \oplus 1 \oplus 2 \oplus \cdots \oplus N$ in sublinear time complexity ($\oplus$ denotes the bitwise XOR operation)?

Notice that $(2k) \oplus (2k+1) = 1$, as the two integers only differ at the last bit in their binary representation. As the bitwise XOR operation is associative, we can calculate $0 \oplus 1 \oplus 2 \oplus \cdots \oplus N$ as $(0 \oplus 1) \oplus (2 \oplus 3) \oplus \cdots \oplus N$. This is beneficial as the expression becomes $1 \oplus 1 \oplus 1 \oplus \ldots$. Notice that $X \oplus X = 0$ for any X, so we can simply count the number of 1s made and paired to determine the result.

From here, we notice different cases:

- When $N \bmod 2 \equiv 0$, there are $\frac{N}{2}$ complete pairs that calculate to 1, and the number N has no pair. The resulting value depends on the parity of $\frac{N}{2}$,
  - If $\frac{N}{2} \bmod 2 \equiv 0$, (this is equivalent to saying $N \bmod 4 \equiv 0$), there are an even number of 1s, thus doing XOR on them gives you $0 \oplus N = N$.
  - If $\frac{N}{2} \bmod 2 \equiv 1$, (this is equivalent to saying $N \bmod 4 \equiv 2$), there are an odd number of 1s, thus doing XOR on them gives you $1 \oplus N = N + 1$ (as N is even).
- When $N \bmod 2 \equiv 1$, there are $\frac{N+1}{2}$ complete pairs that calculate to 1. The resulting value depends on the parity of $\frac{N+1}{2}$,
  - If $\frac{N+1}{2} \bmod 2 \equiv 0$, (this is equivalent to saying $N \bmod 4 \equiv 3$), there are an even number of 1s, thus doing XOR on them gives you 0.
  - If $\frac{N+1}{2} \bmod 2 \equiv 1$), (this is equivalent to saying $N \bmod 4 \equiv 1$, there are an odd number of 1s, thus doing XOR on them gives you 1.

To summarize,

$$0 \oplus 1 \oplus 2 \oplus \cdots \oplus N = \begin{cases} N & \text{if } N \equiv 0 \pmod 4, \\ 1 & \text{if } N \equiv 1 \pmod 4, \\ N+1 & \text{if } N \equiv 2 \pmod 4, \\ 0 & \text{if } N \equiv 3 \pmod 4. \end{cases}$$

Coming back to the original problem, notice that removing an integer from the expression is actually the same thing as adding an integer to that expression (as $X \oplus X = 0$). Thus, we can consider the resulting value of $0 \oplus 1 \oplus 2 \oplus \cdots \oplus N$ and do a bitwise XOR of this result and a number not more than N, and see if we can get a larger value than N.

- For $N \bmod 4 \equiv 0$, we can remove the number 1. This results in $0 \oplus 2 \oplus 3 \oplus \cdots \oplus N = (0 \oplus 1 \oplus \cdots \oplus N) \oplus 1 = N \oplus 1 = N + 1$ which is larger than N. Thus, the answer is YES for $N \bmod 4 \equiv 0$.
- For $N \bmod 4 \equiv 1$, we cannot get a larger number than N. This is because $(0 \oplus 1 \oplus \cdots \oplus N) \oplus N = 1 \oplus N = N - 1$, $(0 \oplus 1 \oplus \cdots \oplus N) \oplus (N-1) = 1 \oplus (N-1) = N$, and any other number will result in a smaller number than the two cases mentioned. Thus, the answer is NO for $N \bmod 4 \equiv 1$.
- For $N \bmod 4 \equiv 2$, we can remove the number 0. This results in $1 \oplus 2 \oplus 3 \oplus \cdots \oplus N = (0 \oplus 1 \oplus \cdots \oplus N) \oplus 0 = (N+1) \oplus 0 = N + 1$ which is larger than N. Thus, the answer is YES for $N \bmod 4 \equiv 2$.
- For $N \bmod 4 \equiv 3$, we cannot get a larger number than N. This is because $(0 \oplus 1 \oplus \cdots \oplus N) \oplus N = 0 \oplus N = N$, $(0 \oplus 1 \oplus \cdots \oplus N) \oplus (N-1) = 0 \oplus (N-1) = N - 1$, and any other number will result in a smaller number than the two cases mentioned. Thus, the answer is NO for $N \bmod 4 \equiv 3$.

Hence,

$$\texttt{answer} = \begin{cases} \texttt{YES} & \text{if } N \equiv 0 \pmod 2, \\ \texttt{NO} & \text{otherwise} \end{cases}$$

Note: one can implement a $O(N)$ (or even $O(N^2)$) brute force algorithm to check for patterns of the output, which should be easy to spot. While rigorously proving a solution is always preferred to prevent any penalties, sometimes simply finding and verifying patterns for a simple problem should suffice, especially when you are running out of time. Conversely, hastily submitting a wrong solution out of unproven intuition (without any form of verification) can damage your performance.