KAIST 8th ACM-ICPC Mock Competition

Solution

School of Computing
KAIST

Problem Statistics

	Onsite (20 teams)	Open (56 teams)	Code	Level
A	1 (287min)	5 (91min)	1263B	Hard
B	1 (286min)	4 (192min)	1280B	Hard
C	5 (136min)	2 (201min)	867B	Medium
D	9 (94min)	18 (25min)	662B	Medium
E	0	4 (157min)	912B	Hard
F	12 (26min)	29 (4min)	413B	Easy
G	3 (54min)	11 (32min)	311B	Medium
H	0	6 (111min)	2058B	Medium
I	20 (3min)	48 (1min)	51B	Easy
J	15 (56min)	25 (16min)	944B	Easy
K	0	4 (39min)	1043B	Hard
L	12 (44min)	22 (37min)	705B	Easy
1st	Thinking Face (8/851)	kjp86201 (12/1328)		

I. Repetitive Palindrome

- Solved by 20+48 team(s)
- First Solve: Ajou Strong Team (3:31)
- Open First Solve: dotorya (1:50)
- Tags: Ad-hoc
- Author: Minkyu Jo

I. Repetitive Palindrome

- You are given a string s and an integer k.
- Is $t = \underbrace{sss \cdots ss}_{k \text{ copies}}$ a palindrome?

I. Repetitive Palindrome

- $t = t^R \iff \underbrace{sss\cdots ss}_{k \text{ copies}} = (\underbrace{sss\cdots ss}_{k \text{ copies}})^R = \underbrace{s^R s^R s^R \cdots s^R s^R}_{k \text{ copies}}$

I. Repetitive Palindrome

- $t = t^R \iff \underbrace{sss\cdots ss}_{k \text{ copies}} = (\underbrace{sss\cdots ss}_{k \text{ copies}})^R = \underbrace{s^R s^R s^R \cdots s^R s^R}_{k \text{ copies}}$
- s and s^R has same length!

I. Repetitive Palindrome

- $t = t^R \iff \underbrace{sss\cdots ss}_{k\text{ copies}} = (\underbrace{sss\cdots ss}_{k\text{ copies}})^R = \underbrace{s^R s^R s^R \cdots s^R s^R}_{k\text{ copies}}$
- s and s^R has same length!
- so $t = t^R \iff s = s^R$

I. Repetitive Palindrome

- $t = t^R \iff \underbrace{sss\cdots ss}_{k \text{ copies}} = (\underbrace{sss\cdots ss}_{k \text{ copies}})^R = \underbrace{s^R s^R s^R \cdots s^R s^R}_{k \text{ copies}}$
- s and s^R has same length!
- so $t = t^R \iff s = s^R$
- Check whether s is palindrome in $\mathcal{O}(|s|)$ time.

F. Fractions

- Solved by 12+29 team(s)
- First Solve: Skai (26:48)
- Open First Solve: rkm0959 (4:21)
- Tags: Math
- Author: Suchan Park

F. Fractions

- $\frac{x}{y}$ is a *Suneung fraction* iff it reduces to $\frac{q}{p}$ and $1 \leq p + q \leq 999$ holds.
- Count the number of *Suneung fraction* $\frac{x}{y}$ where $A \leq x \leq B$ and $C \leq y \leq D$ holds.

F. Fractions

- Instead, count the number of *Suneung fractions* $\frac{x}{y}$ where $\mathbf{1} \leq x \leq P$ and $\mathbf{1} \leq y \leq Q$, $f(P, Q)$.

F. Fractions

- Instead, count the number of *Suneung fractions* $\frac{x}{y}$ where $\mathbf{1} \leq x \leq P$ and $\mathbf{1} \leq y \leq Q$, $f(P, Q)$.
- Then, the answer is equivalent to:

$$f(B, D) - f(A-1, D) - f(B, C-1) + f(A-1, C-1)$$

F. Fractions

- From $\frac{p}{q}$ (where p and q are coprime), the *Suneung fraction* must be in the form of $\frac{k \cdot p}{k \cdot q}$ for some positive k.

F. Fractions

- From $\frac{p}{q}$ (where p and q are coprime), the *Suneung fraction* must be in the form of $\frac{k \cdot p}{k \cdot q}$ for some positive k.
- $1 \leq k \cdot p \leq P$ and $1 \leq k \cdot q \leq Q$ must hold.

F. Fractions

- From $\frac{p}{q}$ (where p and q are coprime), the *Suneung fraction* must be in the form of $\frac{k \cdot p}{k \cdot q}$ for some positive k.
- $1 \leq k \cdot p \leq P$ and $1 \leq k \cdot q \leq Q$ must hold.
- The number of possible k is $\min\left(\left\lfloor \frac{P}{p} \right\rfloor, \left\lfloor \frac{Q}{q} \right\rfloor\right)$.

F. Fractions

- In conclusion,

$$f(P, Q) = \sum_{\gcd(p,q)=1, 1 \le p+q \le 999} \min\left(\left\lfloor \frac{P}{p} \right\rfloor, \left\lfloor \frac{Q}{q} \right\rfloor\right)$$

F. Fractions

- In conclusion,

$$f(P, Q) = \sum_{\gcd(p,q)=1, 1 \le p+q \le 999} \min\left(\left\lfloor \frac{P}{p} \right\rfloor, \left\lfloor \frac{Q}{q} \right\rfloor\right)$$

- The number of such $\frac{p}{q}$ should be something like $\le 1\,000^2$, which is small enough to iterate.

L. Voronoi Diagram Returns

- Solved by 12+22 team(s)
- First Solve: Thinking Face (44:23)
- Open First Solve: 789 (37:52)
- Tags: Implementation
- Author: Hanpil Kang

L. Voronoi Diagram Returns

- You are given n points.
- Construct Voronoi Diagram and answer point query problem.

L. Voronoi Diagram Returns

- Is it really mandatory to construct Voronoi Diagram?

L. Voronoi Diagram Returns

- A point K is included in region i if and only if $d(P_i, K) \leq d(P_j, K)$ holds for all $1 \leq j \leq n$.

L. Voronoi Diagram Returns

- A point K is included in region i if and only if $d(P_i, K) \leq d(P_j, K)$ holds for all $1 \leq j \leq n$.
- i. e., $d(P_i, K) = min_{1 \leq j \leq n}\ d(P_j, K)$

L. Voronoi Diagram Returns

- A point K is included in region i if and only if $d(P_i, K) \leq d(P_j, K)$ holds for all $1 \leq j \leq n$.
- i. e., $d(P_i, K) = min_{1 \leq j \leq n} \; d(P_j, K)$
- You can use the definition directly to test whether the point is in the region or not!

L. Voronoi Diagram Returns

- A point K is included in region i if and only if $d(P_i, K) \leq d(P_j, K)$ holds for all $1 \leq j \leq n$.
- i. e., $d(P_i, K) = min_{1 \leq j \leq n} \; d(P_j, K)$
- You can use the definition directly to test whether the point is in the region or not!
- Time Complexity: $\mathcal{O}(qn)$.

G. Game on Plane

- Solved by 3+11 team(s)
- First Solve: Thinking Face (54:16)
- Open First Solve: 789 (32:52)
- Tags: Games, DP
- Author: Jongwon Lee

G. Game on Plane

- If you draw a segment that meets a previously drawn segment at the endpoints, then your opponent can immediately draw a triangle and you will lose.

G. Game on Plane

- If you draw a segment that meets a previously drawn segment at the endpoints, then your opponent can immediately draw a triangle and you will lose.
- On the other hand, if no two of the drawn segments do not meet, then the next player cannot end the game.

G. Game on Plane

- If you draw a segment that meets a previously drawn segment at the endpoints, then your opponent can immediately draw a triangle and you will lose.
- On the other hand, if no two of the drawn segments do not meet, then the next player cannot end the game.
- Therefore, the game can be interpreted as a game drawing segment where the new segment must not touch any of the previously drawn segments at the endpoint.

G. Game on Plane

- If you draw a line segment separating the n points into i, $n-2-i$ points respectively, then the game is now equivalent to playing on two sets of points with i, $n-2-i$ points respectively.
- Therefore, the grundy number of the game can be computed by the following recurrence

$$f(n) = \min_{k \in \mathbb{Z}_{\geq 0}} \{k \neq f(i) \text{ XOR } f(n-2-i) \text{ for all } i\}$$

J. Rising Sun

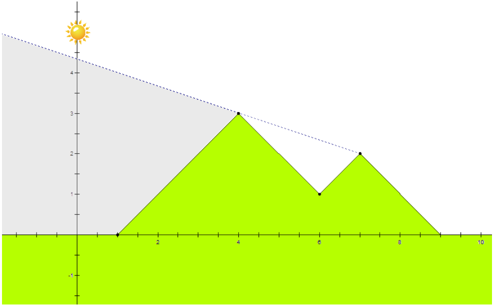
- Solved by 15+25 team(s)
- First Solve: Kkeujeok Kkeujeok (56:16)
- Open First Solve: 1207koo (16:56)
- Tags: Geometry, Implementation
- Author: Joonhyung Shin

J. Rising Sun

- For each mountain, consider the ray starting from Joon's house that passes through the summit of the mountain. Call it *summit ray*.

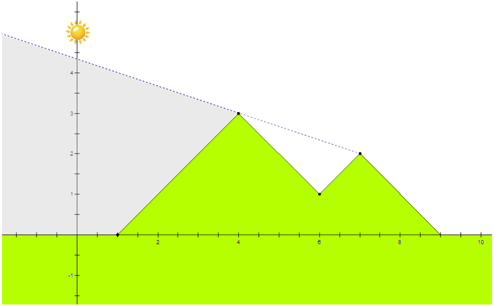
J. Rising Sun

- For each mountain, consider the ray starting from Joon's house that passes through the summit of the mountain. Call it *summit ray*.



J. Rising Sun

- Joon can see the sun *if and only if* for each summit that is in strictly left of Joon's house, the summit ray meets the y-axis below the sun.



J. Rising Sun

- Let (x_i, y_i) be the positions of the summits in the left side of Joon's house which is at (a, b).

J. Rising Sun

- Let (x_i, y_i) be the positions of the summits in the left side of Joon's house which is at (a, b).

- Using the straight line equation, the answer to the problem is

$$\max\left\{\max_i \left\lceil b - \frac{y_i - b}{x_i - a} \cdot a \right\rceil, 0\right\}.$$

J. Rising Sun

- Let (x_i, y_i) be the positions of the summits in the left side of Joon's house which is at (a, b).
- Using the straight line equation, the answer to the problem is

$$\max\left\{\max_i \left\lceil b - \frac{y_i - b}{x_i - a} \cdot a \right\rceil, 0\right\}.$$

- Be careful of overflow!
- Time complexity: $O(n)$

D. Fake Plastic Trees

- Solved by 9+18 team(s)
- First Solve: Thinking Face (94:59)
- Open First Solve: 789 (25:19)
- Tags: Math, Constructive
- Author: Jaehyun Koo

D. Fake Plastic Trees

- You should store N-node tree in $O(\log N)$ memory.

D. Fake Plastic Trees

- You should store N-node tree in $O(\log N)$ memory.
- And you should show off your memory, to prove your construction.

D. Fake Plastic Trees

- You should store N-node tree in $O(\log N)$ memory.
- And you should show off your memory, to prove your construction.
- This setting is quite non-standard, but don't panic!

D. Fake Plastic Trees

- Fake Plastic Trees are recursive, so the solution is.
- Let's take the top-down way: We need a tree with N leaves.

D. Fake Plastic Trees

- Fake Plastic Trees are recursive, so the solution is.
- Let's take the top-down way: We need a tree with N leaves.
- We recursively make two trees with $\lceil N/2 \rceil$, $\lfloor N/2 \rfloor$ leaves.
- Then, we just add one node as a *parent* of two trees.

D. Fake Plastic Trees

- Fake Plastic Trees are recursive, so the solution is.
- Let's take the top-down way: We need a tree with N leaves.
- We recursively make two trees with $\lceil N/2 \rceil$, $\lfloor N/2 \rfloor$ leaves.
- Then, we just add one node as a *parent* of two trees.
- $V = \Omega(N)$.

D. Fake Plastic Trees

- Let's seek some more improvement.
- If N is even, We don't have to make two different trees, one $N/2$-size tree is enough.

D. Fake Plastic Trees

- Let's seek some more improvement.
- If N is even, We don't have to make two different trees, one $N/2$-size tree is enough.
- Unfortunately, this is just a constant optimization.
- $V = \Omega(N)$ still remains.

D. Fake Plastic Trees

- Let's seek some more improvement.
- If N is even, We don't have to make two different trees, one $N/2$-size tree is enough.
- Unfortunately, this is just a constant optimization.
- $V = \Omega(N)$ still remains.
- **No.** Actually $V = \Omega(N^{0.69})$. Do you see why?

D. Fake Plastic Trees

- Nonetheless, $\Omega(N^{0.69})$ is still bad. But we had some significant observation: Only odd N needs two childs.

D. Fake Plastic Trees

- Nonetheless, $\Omega(N^{0.69})$ is still bad. But we had some significant observation: Only odd N needs two childs.
- Let $N = 2K + 1$, then it needs two child with $K + 1$ and K leaves.
- One of $\{K + 1, K\}$ will be even, so they won't branch.
- The other will branch, and again, one of their child is even!

D. Fake Plastic Trees

- Nonetheless, $\Omega(N^{0.69})$ is still bad. But we had some significant observation: Only odd N needs two childs.
- Let $N = 2K + 1$, then it needs two child with $K + 1$ and K leaves.
- One of $\{K + 1, K\}$ will be even, so they won't branch.
- The other will branch, and again, one of their child is even!
- If we carefully follow their traces, we might only need $2\log_2(N) + 2$ nodes.

D. Fake Plastic Trees

- Let $f(n)$ be a function that returns a **pair** of FPT: One with size $n+1$, the other with size n.

D. Fake Plastic Trees

- Let $f(n)$ be a function that returns a **pair** of FPT: One with size $n+1$, the other with size n.

- $f(1)$ is easy, and we need two nodes for it.

D. Fake Plastic Trees

- Let $f(n)$ be a function that returns a **pair** of FPT: One with size $n+1$, the other with size n.
- $f(1)$ is easy, and we need two nodes for it.
- For even $2k \geq 2$, we need to build two FPT with size $2k+1$, $2k$. You need two FPT with size $k+1$, k.
- For odd $2k+1 \geq 2$, we need to build two FPT with size $2k+2$, $2k+1$. You need two FPT with size $k+1$, k.

D. Fake Plastic Trees

- Let $f(n)$ be a function that returns a **pair** of FPT: One with size $n+1$, the other with size n.
- $f(1)$ is easy, and we need two nodes for it.
- For even $2k \geq 2$, we need to build two FPT with size $2k+1$, $2k$. You need two FPT with size $k+1$, k.
- For odd $2k+1 \geq 2$, we need to build two FPT with size $2k+2$, $2k+1$. You need two FPT with size $k+1$, k.
- In any case, you only need $f(n/2)$. $V \leq 2\log_2(N)+2$.

C. Electronic Circuit

- Solved by 5+2 team(s)
- First Solve: Thinking Face (136:10)
- Open First Solve: kjp86201 (201:28)
- Tags: Graph
- Author: Joonhyung Shin

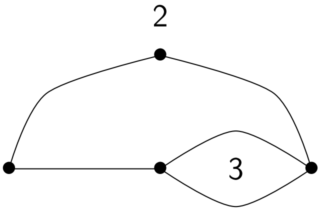
C. Electronic Circuit

- The key of this problem is to take a closer look at 'nice' circuits.

C. Electronic Circuit

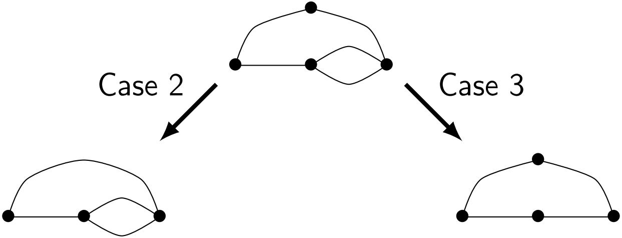
- The key of this problem is to take a closer look at 'nice' circuits.

- One may see that at least one of the following is true for nice circuits.
  1. It consists of a single wire connecting two nodes.
  2. It contains a node which is directly connected to exactly two nodes.
  3. It contains multiple wires connecting the same two nodes.



C. Electronic Circuit

- Moreover, smoothing a node (removing the node and combining the attached wires into one) in Case 2 or removing extra wires in Case 3 does not violate 'nice property'.



C. Electronic Circuit

- **These two actions both reduce the number of wires**, which means that repeating this process, a nice circuit will eventually become a single wire with two nodes (Case 1).

C. Electronic Circuit

- **These two actions both reduce the number of wires**, which means that repeating this process, a nice circuit will eventually become a single wire with two nodes (Case 1).

- In fact, one can prove by induction that the circuit is nice *if and only if* it reduces to Case 1 after applying this process repeatedly.

C. Electronic Circuit

- **These two actions both reduce the number of wires**, which means that repeating this process, a nice circuit will eventually become a single wire with two nodes (Case 1).

- In fact, one can prove by induction that the circuit is nice *if and only if* it reduces to Case 1 after applying this process repeatedly.

- This can be implemented efficiently by maintaining adjacency list with `set` in C++ or `TreeSet` in Java.

- Time complexity: $O(m + n \log n)$

B. Dumae

- Solved by 1+4 team(s)
- First Solve: Thinking Face (286:39)
- Open First Solve: 789 (192:17)
- Tags: Graph, Greedy, DP
- Author: Sunghyeon Jo (Seoul National Univ)

B. Dumae

- Goal : Find a permutation $p_1, ..., p_n$ which satisfies $L_i \leq p_i \leq R_i,\ p_{u_i} < p_{v_i}$
- Inverse of the permutation p is an answer of the problem.

B. Dumae

- Directed graph
  $G = (V, E), V = \{1, 2, .., n\}, E = \{(u_i, v_i) \mid 1 \leq i \leq n\}$
- If G is not a DAG, then solution does not exist.
- Now we assume G to be DAG.

B. Dumae

- If there is a condition $p_x < p_y$, then we can replace L_y as $max(L_y, L_x + 1)$.
- In the same manner, we can replace R_x as $max(R_x, R_y - 1)$.
- This doesn't change the validity of any solution.

B. Dumae

- By scanning in increasing topological order of G, we can update L_i to satisfy all above condition.
- Do the same for R_i.
- Then we can find a 'tighter interval' of p_i.

B. Dumae

- After finding a 'tighter interval', we can completely ignore the topological order!
- We can use the standard "deadline first" greedy algorithm for matching intervals with numbers.

B. Dumae

- After finding a 'tighter interval', we can completely ignore the topological order!
- We can use the standard "deadline first" greedy algorithm for matching intervals with numbers.
- We match each number $x \in \{1, \cdots, N\}$ in increasing order of x.
- Among all interval that contains x, choose one that have minimum endpoint.
- Remove the matched interval.

B. Dumae

- The proof of this greedy algorithm is done with standard "exchange argument".
- Then, why can we ignore the topological order?

B. Dumae

- The proof of this greedy algorithm is done with standard "exchange argument".
- Then, why can we ignore the topological order?
- For each edge $(u, v) \in E$, $L_u < L_v$ and $R_u < R_v$ holds after the 'scanning procedure'.

B. Dumae

- The proof of this greedy algorithm is done with standard "exchange argument".
- Then, why can we ignore the topological order?
- For each edge $(u, v) \in E$, $L_u < L_v$ and $R_u < R_v$ holds after the 'scanning procedure'.
- u is always chosen before v in above greedy procedure.
- Therefore, any matching found by above greedy satisfies $p_u < p_v$ for all $(u, v) \in E$.

A. Coloring Roads

- Solved by 1+5 team(s)
- First Solve: Deobureo Minkyu Party (287:35)
- Open First Solve: 789 (91:13)
- Tags: Data Structures, Tree
- Author: Jongwon Lee

A. Coloring Roads

- Translate the problem into graph theoretical terms, so that we are coloring the edges of a rooted tree.

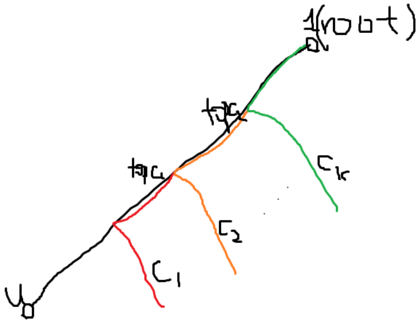
A. Coloring Roads

- Translate the problem into graph theoretical terms, so that we are coloring the edges of a rooted tree.
- Suppose that the color is different for each query.
- For each color c, we shall keep track of the vertex top_c which is the topmost (closest to the root) vertex incident on an edge with color c.
- With this information the answer to the query can be easily computed by precomputing the depth of each vertex.

A. Coloring Roads

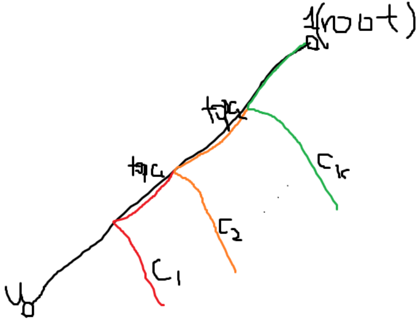
- Also, the color of some edge is the color of the most recent query applied to one of the vertices of its subtree.
- This can be computed in $O(\log n)$ time by maintaining a segment tree of the vertices in dfs order.

A. Coloring Roads



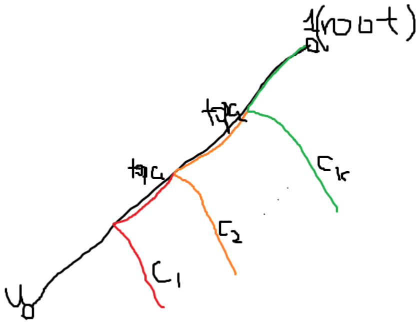
- Suppose there are k colors from the path from u to the root, say $c_1, \ldots, c_k$.

A. Coloring Roads



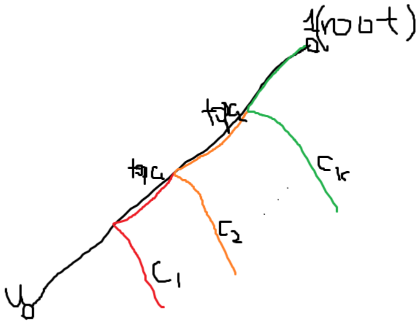
- First, traverse up from u until you meet the first color, c_1.

A. Coloring Roads



- First, traverse up from u until you meet the first color, c_1.
- Change top_{c_1} to the appropriate vertex and jump to the previous value of the top_{c_1} where you can meet the next color c_2.

A. Coloring Roads



- Continue this until you reach the root.

A. Coloring Roads

It might seem that this solution is slow at first sight, but we shall prove that this indeed works.

A. Coloring Roads

It might seem that this solution is slow at first sight, but we shall prove that this indeed works.
To analyze the time complexity, note that we have two parts:

1. traversing up until we meet the first colored edge,
2. traversing the colored edges.

The total time of the first part can be done in $O(n \log n)$ time since each edge appears at most once in this process, and never appears again after it gets a color.

A. Coloring Roads

It might seem that this solution is slow at first sight, but we shall prove that this indeed works.
To analyze the time complexity, note that we have two parts:

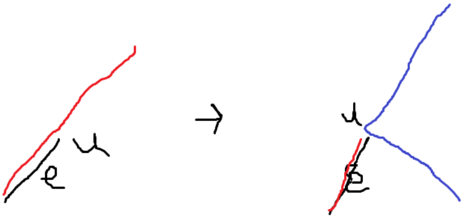
1. traversing up until we meet the first colored edge,
2. traversing the colored edges.

The total time of the first part can be done in $O(n \log n)$ time since each edge appears at most once in this process, and never appears again after it gets a color.
The second part takes $O(k \log n)$ time per query where k is the number of colors you meet in that query. We shall show that the sum of k for all queries is bounded by $O((n+q)\log(n+q))$ so that the time complexity in total is $O((n+q)\log^2(n+q))$.

A. Coloring Roads

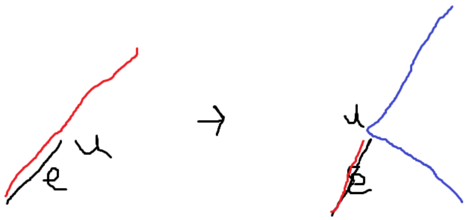
- Since k, for each query, is the number of colors c such that top_c changes, the sum of all k is equal to the sum of, for each vertex u, the number of times when u becomes the top vertex of a color.
- For the time being, assume that the queries were applied on different vertices.

A. Coloring Roads



- For each edge e under u, it becomes the top edge of a color when a query is applied to a vertex in the subtree of e and some time later a query is applied to a vertex in the subtree of u but not in the subtree of e. (See the figure)

A. Coloring Roads



- The number of such events would be bounded by

$$\min(\text{subtree of } e, \text{subtree of } u \text{ minus that of } e)$$

A. Coloring Roads

- Suppose that u has m children and let the size of the subtree of each be $s_1, \ldots, s_m$ in the increasing order. Let $S = s_1 + \cdots + s_m + 1$ be the size of the subtree of u.
- Then, the number of times u becomes the top is bounded by

$$\begin{aligned} &\min(s_1, S - s_1) + \cdots + \min(s_m, S - s_m) \\ &\leq s_1 + \cdots + s_{m-1} + S - s_m = 2 * (S - s_m) - 1 \end{aligned}$$

A. Coloring Roads

- It can be proven that the sum of such number for all vertices is $O(n \log n)$. (Compare heavy-light decomposition)
- For the case where queries can be applied to the same vertex many times, add one children to the vertex for each query applied on it, then the proof above applies again.
- Finally, only the final computation of the answers changes slightly if we allow multiple queries to have the same color.

A. Coloring Roads

- Extra challenge: Solve this problem in $O(n + q\log(n))$.

H. Histogram Sequence

- Solved by 0+6 team(s)
- No solve in onsite contest.
- Open First Solve: 789 (111:07)
- Tags: Binary Search, Data Structure
- Author: Suchan Park

H. Histogram Sequence

- What if we are to only compute A_k?

H. Histogram Sequence

- What if we are to only compute A_k?
- It is tempting to use the classical *binary search on answer* technique

H. Histogram Sequence

- What if we are to only compute A_k?
- It is tempting to use the classical *binary search on answer* technique
- Define $f(x)$ as the number of elements of A less than or equal to x

H. Histogram Sequence

- What if we are to only compute A_k?
- It is tempting to use the classical *binary search on answer* technique
- Define $f(x)$ as the number of elements of A less than or equal to x
    - f is a monotonically increasing function

H. Histogram Sequence

- What if we are to only compute A_k?
- It is tempting to use the classical *binary search on answer* technique
- Define $f(x)$ as the number of elements of A less than or equal to x
  - f is a monotonically increasing function
  - $f(A_k - 1) < k \leq f(A_k)$ holds

H. Histogram Sequence

- What if we are to only compute A_k?
- It is tempting to use the classical *binary search on answer* technique
- Define $f(x)$ as the number of elements of A less than or equal to x
  - f is a monotonically increasing function
  - $f(A_k - 1) < k \leq f(A_k)$ holds
  - Find minimum m where $f(m) \geq k$, then $A_k = m$

H. Histogram Sequence

- How to compute $f(x)$ efficiently?

H. Histogram Sequence

- How to compute $f(x)$ efficiently?
- By definition, $f(x)$ equals to the number of rectangles whose area $\leq x$

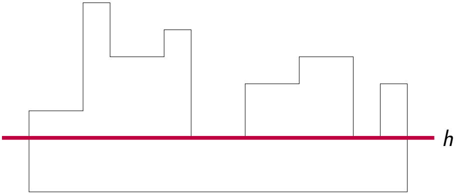
H. Histogram Sequence

- How to compute $f(x)$ efficiently?
- By definition, $f(x)$ equals to the number of rectangles whose area $\leq x$
- If we fix the height h, we have to count the number of rectangles whose width $\leq \lfloor \frac{x}{h} \rfloor$

H. Histogram Sequence

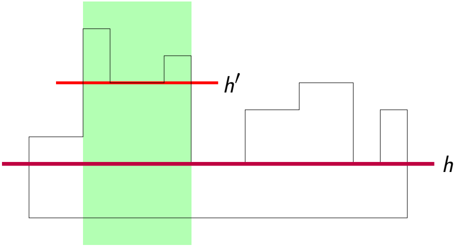
- How to compute $f(x)$ efficiently?
- By definition, $f(x)$ equals to the number of rectangles whose area $\leq x$
- If we fix the height h, we have to count the number of rectangles whose width $\leq \lfloor \frac{x}{h} \rfloor$
  - Goal: compute the number of rectangles with height exactly h, and width exactly $1, 2, \cdots, \lfloor \frac{x}{h} \rfloor$.

H. Histogram Sequence



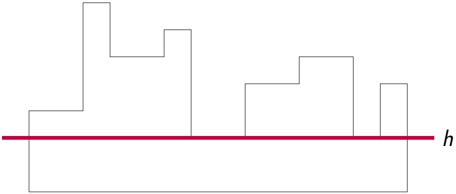
We may assume h is the minimum height among all bars.

H. Histogram Sequence



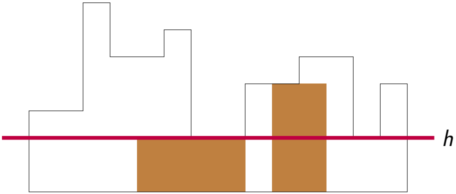
We may assume h is the minimum height among all bars.
For all other heights h' ($> h$), we can use a stack to find the maximal interval $[l, r]$ of H, where $\min_{l \le i \le r} H[i] \ge h$, and solve the same problem. (Google 'largest rectangle in histogram')

H. Histogram Sequence



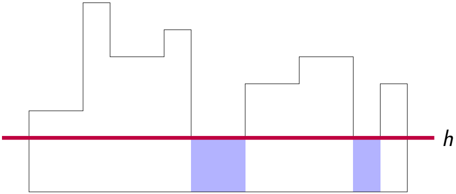
When does the height of the rectangle made by $[i, j]$ equal h?

H. Histogram Sequence



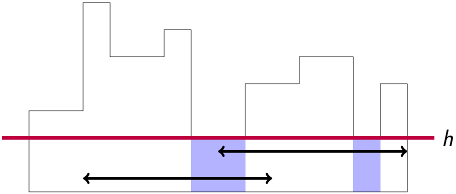
When does the height of the rectangle made by $[i, j]$ equal h?
$\rightarrow$ Since h is the *minimum* height, at least one of $H_i, H_{i+1}, \cdots, H_j$ should be h.

H. Histogram Sequence



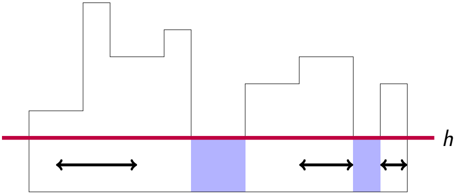
Mark all bars whose height is exactly h.

H. Histogram Sequence



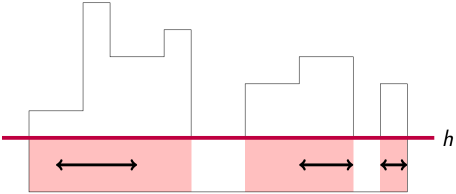
Mark all bars whose height is exactly h.
For $[i, j]$ to have height exactly h, the interval should touch at least one of the marked bars.

H. Histogram Sequence



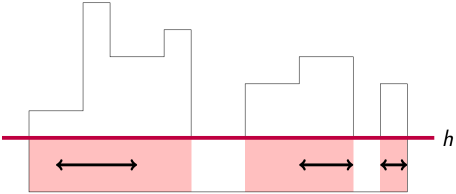
Mark all bars whose height is exactly h.
For $[i, j]$ to have height exactly h, the interval should touch at least one of the marked bars.
For $[i, j]$ to **NOT** have height exactly h, the interval should touch **NO** marked positions.

H. Histogram Sequence



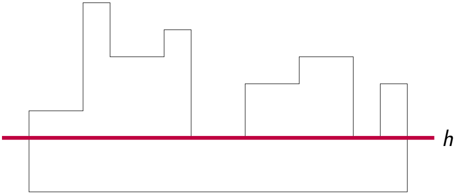
For $[i, j]$ to **NOT** have height exactly h, the interval should be entirely contained in one of the unmarked areas.

H. Histogram Sequence



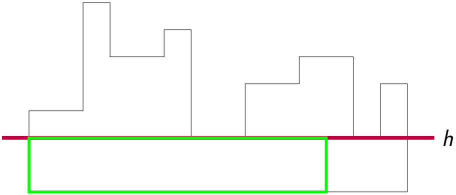
For $[i, j]$ to **NOT** have height exactly h, the interval should be entirely contained in one of the unmarked areas.
This seems easier to deal with! With this observation, let's compute the number of rectangles with height exactly h, and width exactly $1, 2, \cdots, \lfloor \frac{x}{h} \rfloor$.

H. Histogram Sequence



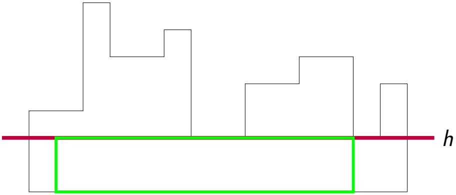
The number of rectangles of width i is obviously $n - i + 1$.

H. Histogram Sequence



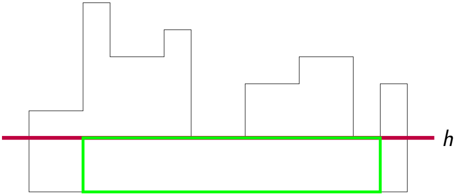
The number of rectangles of width i is obviously $n - i + 1$.

H. Histogram Sequence



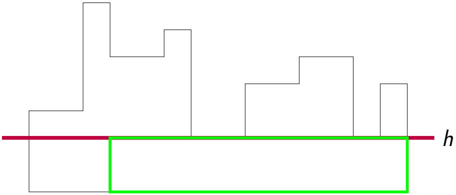
The number of rectangles of width i is obviously $n - i + 1$.

H. Histogram Sequence



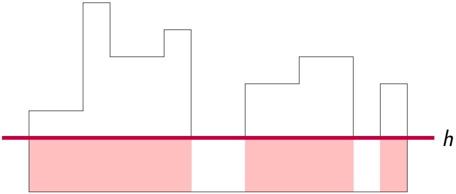
The number of rectangles of width i is obviously $n - i + 1$.

H. Histogram Sequence



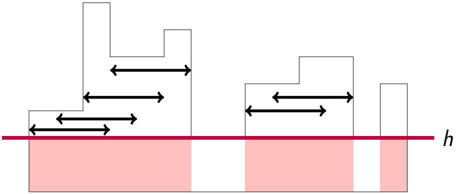
The number of rectangles of width i is obviously $n - i + 1$.

H. Histogram Sequence



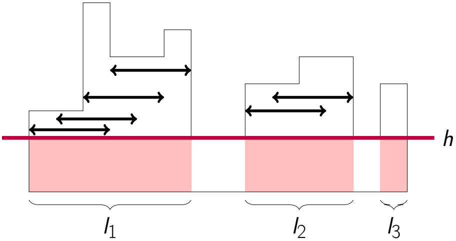
The number of rectangles of width i whose height is **NOT** h is..

H. Histogram Sequence



The number of rectangles of width i whose height is **NOT** h is.. just as same as the case we've seen before!

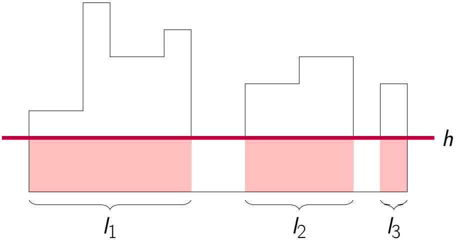
H. Histogram Sequence



The number of rectangles of width i whose height is **NOT** h is.. just as same as the case we've seen before!

$$\sum_{l} \max(l - i + 1, 0)$$

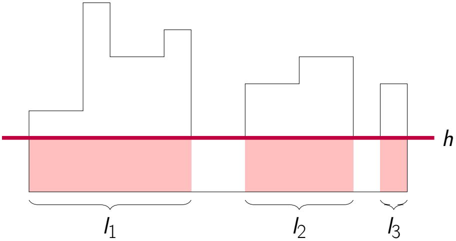
H. Histogram Sequence



In conclusion, the number of rectangles of width i, whose height is exactly h, is:

$$(n - i + 1) - \sum_{l} \max(l - i + 1, 0)$$

H. Histogram Sequence



In conclusion, the number of rectangles of width i, whose height is exactly h, is:

$$(n - i + 1) - \sum_{l} \max(l - i + 1, 0)$$

H. Histogram Sequence

Let $m = \lfloor \frac{x}{h} \rfloor$, then effectively we need to compute

$$\sum_{i=1}^{m} \left\{ (n - i + 1) - \sum_{l} \max(l - i + 1, 0) \right\}$$

H. Histogram Sequence

Let $m = \lfloor \frac{x}{h} \rfloor$, then effectively we need to compute

$$\sum_{i=1}^{m} \left\{ (n - i + 1) - \sum_{l} \max(l - i + 1, 0) \right\}$$

$$= \sum_{i=1}^{m} (n - i + 1) - \sum_{l} \sum_{i=1}^{m} \max(l - i + 1, 0)$$

H. Histogram Sequence

Let $m = \lfloor \frac{x}{h} \rfloor$, then effectively we need to compute

$$\begin{aligned}
&\sum_{i=1}^{m} \left\{ (n - i + 1) - \sum_{l} \max(l - i + 1, 0) \right\} \\
&= \sum_{i=1}^{m} (n - i + 1) - \sum_{l} \sum_{i=1}^{m} \max(l - i + 1, 0) \\
&= \mathtt{sum1}(n - m + 1, n) - \sum_{l} \mathtt{sum1}\left(\max(l - m + 1, 0), l\right)
\end{aligned}$$

where $\mathtt{sum1}(p, q) = \sum_{t=p}^{q} t$.

H. Histogram Sequence

Let $m = \lfloor \frac{x}{h} \rfloor$, then effectively we need to compute

$$\sum_{i=1}^{m} \left\{ (n-i+1) - \sum_{l} \max(l-i+1, 0) \right\}$$
$$= \sum_{i=1}^{m} (n-i+1) - \sum_{l} \sum_{i=1}^{m} \max(l-i+1, 0)$$
$$= \mathtt{sum1}(n-m+1, n) - \sum_{l} \mathtt{sum1}\left(\max(l-m+1, 0), l\right)$$

where $\mathtt{sum1}(p, q) = \sum_{t=p}^{q} t$.
The total number of "uncovered areas" for all h is $O(n)$, so it takes only $O(n)$ time to compute $f(x)$.

H. Histogram Sequence

So far, we were able to compute A_L in $O(n \log \texttt{maxanswer})$ time.

H. Histogram Sequence

So far, we were able to compute A_L in $O(n \log \texttt{maxanswer})$ time. What about $A_{L+1}, \cdots, A_R$?

H. Histogram Sequence

So far, we were able to compute A_L in $O(n \log \texttt{maxanswer})$ time. What about $A_{L+1}, \cdots, A_R$?
WLOG assume $L = f(A_L)$, so that $A_{L+1} \neq A_L$. We can do this by printing 'A_L' $f(A_L) - L$ times.

H. Histogram Sequence

So far, we were able to compute A_L in $O(n \log \texttt{maxanswer})$ time. What about $A_{L+1}, \cdots, A_R$?
WLOG assume $L = f(A_L)$, so that $A_{L+1} \neq A_L$. We can do this by printing 'A_L' $f(A_L) - L$ times.
If we fix h, we considered all rectangles with width $\leq \lfloor \frac{A_L}{h} \rfloor$.
The smallest rectangle we didn't cover has width $\lfloor \frac{A_L}{h} \rfloor + 1$.

H. Histogram Sequence

Given A_L, how to compute $A_{L+1}, \cdots, A_R$?
First, for all h, push $\left(\left(\lfloor \frac{A_L}{h} \rfloor + 1\right) \cdot h, h\right)$ to a min heap. Then, repeat the following:

H. Histogram Sequence

Given A_L, how to compute $A_{L+1}, \cdots, A_R$?
First, for all h, push $\left(\left(\lfloor \frac{A_L}{h} \rfloor + 1\right) \cdot h, h\right)$ to a min heap. Then, repeat the following:

1. Pop the smallest element (a', h') from the heap.

H. Histogram Sequence

Given A_L, how to compute $A_{L+1}, \cdots, A_R$?
First, for all h, push $\left(\left(\lfloor \frac{A_L}{h} \rfloor + 1\right) \cdot h, h\right)$ to a min heap. Then, repeat the following:

1. Pop the smallest element (a', h') from the heap.
2. With the formula in the previous slides, we can compute c, the number of rectangles with height h' and width a'/h'.

H. Histogram Sequence

Given A_L, how to compute $A_{L+1}, \cdots, A_R$?
First, for all h, push $\left(\left(\lfloor \frac{A_L}{h} \rfloor + 1\right) \cdot h, h\right)$ to a min heap. Then, repeat the following:

1. Pop the smallest element (a', h') from the heap.
2. With the formula in the previous slides, we can compute c, the number of rectangles with height h' and width a'/h'.
3. Print a' c times.

H. Histogram Sequence

Given A_L, how to compute $A_{L+1}, \cdots, A_R$?
First, for all h, push $\left(\left(\lfloor \frac{A_L}{h} \rfloor + 1\right) \cdot h, h\right)$ to a min heap. Then, repeat the following:

1. Pop the smallest element (a', h') from the heap.
2. With the formula in the previous slides, we can compute c, the number of rectangles with height h' and width a'/h'.
3. Print a' c times.
4. Push $(a' + h', h')$ to the min heap.

H. Histogram Sequence

Given A_L, how to compute $A_{L+1}, \cdots, A_R$?
First, for all h, push $\left(\left(\lfloor \frac{A_L}{h} \rfloor + 1\right) \cdot h, h\right)$ to a min heap. Then, repeat the following:

1. Pop the smallest element (a', h') from the heap.
2. With the formula in the previous slides, we can compute c, the number of rectangles with height h' and width a'/h'.
3. Print a' c times.
4. Push $(a' + h', h')$ to the min heap.

Step 2 might take $O(n)$ time, but the total number of summations seems to be $O(n\sqrt{n})$ with a small constant, so this works. However, we can make Step 2 work in $O(\log n)$ or amortized $O(1)$ with some preprocessing.

K. Utilitarianism

- Solved by 0+4 team(s)
- No solve in onsite contest.
- Open First Solve: kjp86201 (39:05)
- Tags: DP, Binary Search
- Author: Jongwon Lee

K. Utilitarianism

A subset of edges of a graph such that no two edges are adjacent is called a *matching*. In this problem, given a tree, our goal is to choose a matching of size k which maximizes the weight.

K. Utilitarianism

- First, consider the problem where you have to find the size of matching which maximizes the weight with no a priori constraint on the size of the matching. Call this problem K2

K. Utilitarianism

- First, consider the problem where you have to find the size of matching which maximizes the weight with no a priori constraint on the size of the matching. Call this problem K2
- This problem can be solved by a simple depth first search. Precisely, set any node as the root node, and for each node u, find the maximum weighted matching's weight/size of its subtree, distinguishing the cases
  1. u is included in the matching
  2. u is not included in the matching

K. Utilitarianism

- First, consider the problem where you have to find the size of matching which maximizes the weight with no a priori constraint on the size of the matching. Call this problem K2
- This problem can be solved by a simple depth first search. Precisely, set any node as the root node, and for each node u, find the maximum weighted matching's weight/size of its subtree, distinguishing the cases
  1. u is included in the matching
  2. u is not included in the matching
- It is easy to compute the above values for a node using its children's results.

K. Utilitarianism

- Suppose you add a constant X to the weight of each edge and solve problem K2 on this graph.

K. Utilitarianism

- Suppose you add a constant X to the weight of each edge and solve problem K2 on this graph.
- If X is super big(∞), then the output of problem K2 would be the size of maximum matching possible,

K. Utilitarianism

- Suppose you add a constant X to the weight of each edge and solve problem K2 on this graph.
- If X is super big(∞), then the output of problem K2 would be the size of maximum matching possible,
- and if X is super small($-\infty$), then the output of problem K2 would be zero.

K. Utilitarianism

- Suppose you add a constant X to the weight of each edge and solve problem K2 on this graph.
- If X is super big(∞), then the output of problem K2 would be the size of maximum matching possible,
- and if X is super small($-\infty$), then the output of problem K2 would be zero.
- The output of problem K2 increases when X increases. Therefore, one can binary search on X and solve problem K2 to find X such that the output of problem K2 is exactly k.

K. Utilitarianism

This was the general idea of the solution. In reality one should be more careful since

1. There might be many answers to K2.
2. There might be no X such that the output of K2 is exactly k.

To overcome the first issue, let the output of K2 be the maximum if there are multiple answers.
Now define $f(y)$ as the answer to the problem when $k = y$.
The important observation is that $f(y)$ is a concave function on y, i.e.

$$f(y) - f(y-1) \geq f(y+1) - f(y) \quad \forall y$$

K. Utilitarianism

- To see why $f(y)$ is concave, observe that this problem can be modeled as a min-cost max-flow problem by negating all the costs.

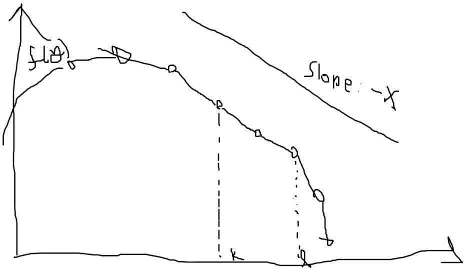
K. Utilitarianism

- To see why $f(y)$ is concave, observe that this problem can be modeled as a min-cost max-flow problem by negating all the costs.
- In the algorithm for solving mcmf using Dijkstra, as an extension of Edmonds-Karp algorithm, the distance from the sink to source increases every iteration, which actually is the whole point of Edmonds-Karp algorithm.

K. Utilitarianism

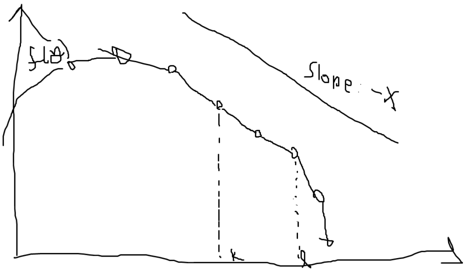
- To see why $f(y)$ is concave, observe that this problem can be modeled as a min-cost max-flow problem by negating all the costs.
- In the algorithm for solving mcmf using Dijkstra, as an extension of Edmonds-Karp algorithm, the distance from the sink to source increases every iteration, which actually is the whole point of Edmonds-Karp algorithm.
- In the algorithm, this distance equals the change of the minimal cost when the flow increases by 1.
- This implies, in our situation, that the difference $f(y+1) - f(y)$ decreases as y increases (since we have negated all the costs).

K. Utilitarianism



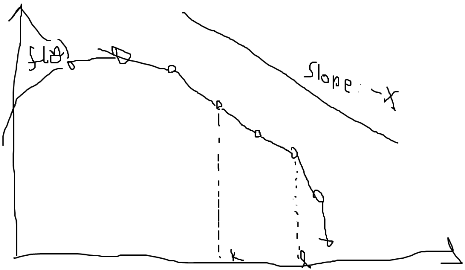
Back to our problem, note that the output of K2 is y where $f(y) + yX$ becomes maximum, and this is exactly where the line with slope $-X$ touches the graph of $f(y)$ tangently.

K. Utilitarianism



Find the smallest X among the ones that the output of K2 is at least k. In the above figure, the output of K2 for such X would be ℓ.

K. Utilitarianism



Note that $f(k) + kX = f(\ell) + \ell X$. Since $f(\ell) + \ell X$ is the maximum weight you can obtain from K2, the final answer is that value minus kX.

E. Fascination Street

- Solved by 0+4 team(s)
- No solve in onsite contest.
- Open First Solve: kjp86201 (157:09)
- Tags: DP
- Author: Jaehyun Koo

E. Fascination Street

- Assume $K = 0$.
- This is a standard DP exercise, where $DP[i][j] =$ (minimum cost to place the streetlight in $[1, i]$ blocks, where your rightmost streetlight lies in position j).
- $i - j \leq 2$ should hold. Thus, this DP requires only $O(N)$ states. We have $O(N)$ time solution.

E. Fascination Street

- Assume $K = 0$.
- This is a standard DP exercise, where $DP[i][j] =$ (minimum cost to place the streetlight in $[1, i]$ blocks, where your rightmost streetlight lies in position j).
- $i - j \leq 2$ should hold. Thus, this DP requires only $O(N)$ states. We have $O(N)$ time solution.
- Wait, you can't assume $K = 0$...

E. Fascination Street

- ...but K is very small, smells like some exhaustive search..

E. Fascination Street

- ...but K is very small, smells like some exhaustive search..

E. Fascination Street

- ...but K is very small, smells like some exhaustive search..

- No! N is too large to make this work :(

E. Fascination Street

- We fix the subset of **location** that will cover the street in the end.
- If $K = 0$, the minimum cost is simply the sum of W_i in the subset.

E. Fascination Street

- We fix the subset of **location** that will cover the street in the end.
- If $K = 0$, the minimum cost is simply the sum of W_i in the subset.
- If $K > 0$, we should swap wisely to minimize the cost.

E. Fascination Street

- We fix the subset of **location** that will cover the street in the end.
- If $K = 0$, the minimum cost is simply the sum of W_i in the subset.
- If $K > 0$, we should swap wisely to minimize the cost.
- This is easy: We just swap the largest W_i in subset, with the smallest W_i not in subset.

E. Fascination Street

- For $K > 0$, let $S(\leq K)$ be the number of swaps we've done.

E. Fascination Street

- For $K > 0$, let $S(\leq K)$ be the number of swaps we've done.
- We drop the top-S elements in the location set, and obtain the bottom-S elements not in the location set.

E. Fascination Street

- For $K > 0$, let $S(\leq K)$ be the number of swaps we've done.
- We drop the top-S elements in the location set, and obtain the bottom-S elements not in the location set.
- If we are doing some kind of DP, then we should take account of those top-S element (and vice versa).

E. Fascination Street

- For $K > 0$, let $S(\leq K)$ be the number of swaps we've done.
- We drop the top-S elements in the location set, and obtain the bottom-S elements not in the location set.
- If we are doing some kind of DP, then we should take account of those top-S element (and vice versa).
- It seems pretty complicated.

E. Fascination Street

- But should we really care about top-S elements?

E. Fascination Street

- But should we really care about top-S elements?
- We can just drop **any** S element in set, and obtain **any** S element not in set.

E. Fascination Street

- But should we really care about top-S elements?
- We can just drop **any** S element in set, and obtain **any** S element not in set.
- If we do the DP well, the optimality is guaranteed anyway.

E. Fascination Street

- Long story short, we should find a partition of streetlight into those four sets:
    1. It is in the location subset, and it's not dropped.
    2. **It is in the location subset, and it's dropped.**
    3. **It is not in the location subset, and it's obtained.**
    4. It is not in the location subset, and it's not obtained.

E. Fascination Street

- Long story short, we should find a partition of streetlight into those four sets:
    1. It is in the location subset, and it's not dropped.
    2. **It is in the location subset, and it's dropped.**
    3. **It is not in the location subset, and it's obtained.**
    4. It is not in the location subset, and it's not obtained.

- Location subset should cover the whole street.

E. Fascination Street

- Long story short, we should find a partition of streetlight into those four sets:
  1. It is in the location subset, and it's not dropped.
  2. **It is in the location subset, and it's dropped.**
  3. **It is not in the location subset, and it's obtained.**
  4. It is not in the location subset, and it's not obtained.
- Location subset should cover the whole street.
- The size of Case 2 / 3 should remain same, and should be at most K.

E. Fascination Street

- To do this, you can add size of dropped ones, size of obtained ones in the DP states.

E. Fascination Street

- To do this, you can add size of dropped ones, size of obtained ones in the DP states.
- Just add two or three for/if statement. You are done :D

E. Fascination Street

- To do this, you can add size of dropped ones, size of obtained ones in the DP states.
- Just add two or three for/if statement. You are done :D
- Time complexity is $O(NK^2)$.
- Memory complexity is $O(K^2)$. Our limits were lenient, thus $O(NK^2)$ also passes easily.