2026 North America Championship Solutions

The Judges

March 22, 2026

Acorn Quarrels

Problem

Given a tree with $N \leq 10^5$ vertices, each with an associated integer value d_i, do the following for each $1 \leq i \leq n$ in order:

- Let m be the maximum value of any neighbor of i. For each neighbor j with value m, subtract 1 from d_j and add 1 to d_i.

Assign values d_i such that $N \leq d_i \leq 2N - 1$ and each vertex has its original value after running the above.

Acorn Quarrels

Problem

Given a tree with $N \leq 10^5$ vertices, each with an associated integer value d_i, do the following for each $1 \leq i \leq n$ in order:

- Let m be the maximum value of any neighbor of i. For each neighbor j with value m, subtract 1 from d_j and add 1 to d_i.

Assign values d_i such that $N \leq d_i \leq 2N - 1$ and each vertex has its original value after running the above.

Initial Observations

- Trying to prove that the above is always possible may give a path to a solution.

Acorn Quarrels

Problem

Given a tree with $N \leq 10^5$ vertices, each with an associated integer value d_i, do the following for each $1 \leq i \leq n$ in order:

- Let m be the maximum value of any neighbor of i. For each neighbor j with value m, subtract 1 from d_j and add 1 to d_i.

Assign values d_i such that $N \leq d_i \leq 2N - 1$ and each vertex has its original value after running the above.

Initial Observations

- Trying to prove that the above is always possible may give a path to a solution.
- One way would be to give a construction such that along each edge, the addition/subtraction happens either 0 or 2 times, i.e. if node u steals from node v, then a valid construction will have node v steal back from node u.

Acorn Quarrels

Solution

- We construct the array of values such that at every step u, all neighbors of v have the same number of acorns T_u (which guarantees all squirrels start and end with same number of acorns).

Acorn Quarrels

Solution

- We construct the array of values such that at every step u, all neighbors of v have the same number of acorns T_u (which guarantees all squirrels start and end with same number of acorns).
- For any node u and a neighbor v, we can relate T_u to d_v:

$$d_v = T_u + L_v(u) - 1[v < u]\deg(v),$$

  where $L_v(u)$ is the number of neighbors of v w/ index less than u.
- In order for squirrel v to have T_u acorns when it's squirrel u's turn to steal, it must have started with T_u acorns, plus one acorn for every squirrel that has already stolen from node v. Additionally, if $v < u$ then node v will have $\deg(v)$ extra acorns from its own thefts.

Acorn Quarrels

Solution

- We construct the array of values such that at every step u, all neighbors of v have the same number of acorns T_u (which guarantees all squirrels start and end with same number of acorns).
- For any node u and a neighbor v, we can relate T_u to d_v:

$$d_v = T_u + L_v(u) - 1[v < u]\deg(v),$$

  where $L_v(u)$ is the number of neighbors of v w/ index less than u.
- In order for squirrel v to have T_u acorns when it's squirrel u's turn to steal, it must have started with T_u acorns, plus one acorn for every squirrel that has already stolen from node v. Additionally, if $v < u$ then node v will have $\deg(v)$ extra acorns from its own thefts.
- Swapping the roles of u and v, we also have $T_v = d_u - L_u(v) + 1[u < v]\deg(u)$.

Acorn Quarrels

Solution

- We can root the tree at node 1 and arbitrarily assign $d_1 = T_1 = 0$. We can then DFS through the tree and use the two equations on the previous slide to compute d_v and T_v for all nodes, using the values from its parent.

Acorn Quarrels

Solution

- We can root the tree at node 1 and arbitrarily assign $d_1 = T_1 = 0$. We can then DFS through the tree and use the two equations on the previous slide to compute d_v and T_v for all nodes, using the values from its parent.
- We can then add a constant to all d_i to place them in the range $[N, 2N - 1]$.
- Every step is $\mathcal{O}(n)$, and the memory required is also $\mathcal{O}(n)$.

Boss Rush

Problem

There are $n \leq 3 \cdot 10^5$ bosses that all attack with a period of $d \leq 10^9$ seconds, with their first attack at some second $f_i < d$. You have a parry move that can be used when a boss attacks you which defeats it. It takes $w \leq 10^9$ seconds before you are able to use the move again. What is the minimum amount of time to defeat all n bosses if you parry optimally.

Boss Rush

Problem

There are $n \leq 3 \cdot 10^5$ bosses that all attack with a period of $d \leq 10^9$ seconds, with their first attack at some second $f_i < d$. You have a parry move that can be used when a boss attacks you which defeats it. It takes $w \leq 10^9$ seconds before you are able to use the move again. What is the minimum amount of time to defeat all n bosses if you parry optimally.

Initial Observations

- The decision question on whether you can defeat all bosses in $\leq t$ seconds is monotonic, so solving this would be enough to binary search.

Boss Rush

Problem

There are $n \leq 3 \cdot 10^5$ bosses that all attack with a period of $d \leq 10^9$ seconds, with their first attack at some second $f_i < d$. You have a parry move that can be used when a boss attacks you which defeats it. It takes $w \leq 10^9$ seconds before you are able to use the move again. What is the minimum amount of time to defeat all n bosses if you parry optimally.

Initial Observations

- The decision question on whether you can defeat all bosses in $\leq t$ seconds is monotonic, so solving this would be enough to binary search.
- For a fixed time t, the order to parry bosses is not obvious (things that don't work: greedy, reverse greedy).

Boss Rush

Problem

There are $n \leq 3 \cdot 10^5$ bosses that all attack with a period of $d \leq 10^9$ seconds, with their first attack at some second $f_i < d$. You have a parry move that can be used when a boss attacks you which defeats it. It takes $w \leq 10^9$ seconds before you are able to use the move again. What is the minimum amount of time to defeat all n bosses if you parry optimally.

Initial Observations

- The decision question on whether you can defeat all bosses in $\leq t$ seconds is monotonic, so solving this would be enough to binary search.
- For a fixed time t, the order to parry bosses is not obvious (things that don't work: greedy, reverse greedy).
- Instructive to think about how to solve $d \leq 10^5$ first.

Boss Rush

Solution: Simplified case

- Suppose first that $d \leq 10^5$, $d \mid t$: we check if answer $\leq t - w$.

Boss Rush

Solution: Simplified case

- Suppose first that $d \leq 10^5$, $d \mid t$: we check if answer $\leq t - w$.
- For any $0 \leq j \leq d - 1$, the number of times we "wait" between time $j + a \cdot d$ and $j + 1 + a \cdot d$ for $0 \leq ad \leq t$ is exactly $s_j = \frac{t}{d} - \sum_{i=1}^{n} \#[j \in [f_i, f_i + w)]$ where the sum subtracts the number of times j appears in the interval $[f_i, f_i + w)$, counting each wraparound.

Solution: Simplified case

- Suppose first that $d \leq 10^5$, $d \mid t$: we check if answer $\leq t - w$.
- For any $0 \leq j \leq d-1$, the number of times we "wait" between time $j + a \cdot d$ and $j + 1 + a \cdot d$ for $0 \leq ad \leq t$ is exactly $s_j = \frac{t}{d} - \sum_{i=1}^{n} \#[j \in [f_i, f_i + w)]$ where the sum subtracts the number of times j appears in the interval $[f_i, f_i + w)$, counting each wraparound.
- Any path in time from 0 to t can be seen as a directed graph with $\sum s_j + n$ edges (but d vertices): one edge for each boss $f_i \to f_i + w \bmod d$ with weight 1, and s_j edges from j to $j + 1$.

Boss Rush

Solution: Simplified case

- Suppose first that $d \leq 10^5$, $d \mid t$: we check if answer $\leq t - w$.
- For any $0 \leq j \leq d - 1$, the number of times we "wait" between time $j + a \cdot d$ and $j + 1 + a \cdot d$ for $0 \leq ad \leq t$ is exactly $s_j = \frac{t}{d} - \sum_{i=1}^{n} \#[j \in [f_i, f_i + w)]$ where the sum subtracts the number of times j appears in the interval $[f_i, f_i + w)$, counting each wraparound.
- Any path in time from 0 to t can be seen as a directed graph with $\sum s_j + n$ edges (but d vertices): one edge for each boss $f_i \to f_i + w \bmod d$ with weight 1, and s_j edges from j to $j + 1$.
- Such a path exists if an only if an Eulerian trail from 0 to $t \bmod d$ exists: here, the graph must be connected and every vertex must have the same in and out degree except 0 and t (or, if $t \bmod d = 0$ and all are equal). If $d \mid t$ we end up in the latter case.

Boss Rush

Solution: Simplified case

- Suppose first that $d \leq 10^5$, $d \mid t$: we check if answer $\leq t - w$.
- For any $0 \leq j \leq d-1$, the number of times we "wait" between time $j + a \cdot d$ and $j + 1 + a \cdot d$ for $0 \leq ad \leq t$ is exactly $s_j = \frac{t}{d} - \sum_{i=1}^{n} \#[j \in [f_i, f_i + w)]$ where the sum subtracts the number of times j appears in the interval $[f_i, f_i + w)$, counting each wraparound.
- Any path in time from 0 to t can be seen as a directed graph with $\sum s_j + n$ edges (but d vertices): one edge for each boss $f_i \to f_i + w \bmod d$ with weight 1, and s_j edges from j to $j + 1$.
- Such a path exists if an only if an Eulerian trail from 0 to $t \bmod d$ exists: here, the graph must be connected and every vertex must have the same in and out degree except 0 and t (or, if $t \bmod d = 0$ and all are equal). If $d \mid t$ we end up in the latter case.
- We can check connectedness in $\mathcal{O}(d)$ after a $\mathcal{O}(d + n)$ graph creation.

Solution, optimizations

- To deal with $d \nmid t$, just need to change calculation of s_j depending on whether $j < t \bmod d$.

Boss Rush

Solution, optimizations

- To deal with $d \nmid t$, just need to change calculation of s_j depending on whether $j < t \bmod d$.
- We need to optimize graph creation to be able to solve $d \leq 10^9$.

Boss Rush

Solution, optimizations

- To deal with $d \nmid t$, just need to change calculation of s_j depending on whether $j < t \bmod d$.
- We need to optimize graph creation to be able to solve $d \leq 10^9$.
- Observation: can compress many edges where there is no boss start/end (or $t \bmod d$) between them, since s_j must be the same for each.

Boss Rush

Solution, optimizations

- To deal with $d \nmid t$, just need to change calculation of s_j depending on whether $j < t \bmod d$.
- We need to optimize graph creation to be able to solve $d \leq 10^9$.
- Observation: can compress many edges where there is no boss start/end (or $t \bmod d$) between them, since s_j must be the same for each.
- Can do this in $\mathcal{O}(n)$ if boss "events" are pre-sorted, and connectedness is now $\mathcal{O}(n)$.

Boss Rush

Solution, optimizations

- To deal with $d \nmid t$, just need to change calculation of s_j depending on whether $j < t \bmod d$.
- We need to optimize graph creation to be able to solve $d \leq 10^9$.
- Observation: can compress many edges where there is no boss start/end (or $t \bmod d$) between them, since s_j must be the same for each.
- Can do this in $\mathcal{O}(n)$ if boss "events" are pre-sorted, and connectedness is now $\mathcal{O}(n)$.
- Answer is at most $2\max(d, w) \cdot n$, so total runtime is $\mathcal{O}(n(\log n + \log d + \log w))$.

Boss Rush

Solution, optimizations

- To deal with $d \nmid t$, just need to change calculation of s_j depending on whether $j < t \bmod d$.
- We need to optimize graph creation to be able to solve $d \leq 10^9$.
- Observation: can compress many edges where there is no boss start/end (or $t \bmod d$) between them, since s_j must be the same for each.
- Can do this in $\mathcal{O}(n)$ if boss "events" are pre-sorted, and connectedness is now $\mathcal{O}(n)$.
- Answer is at most $2\max(d, w) \cdot n$, so total runtime is $\mathcal{O}(n(\log n + \log d + \log w))$.
- **Bonus:** Increasing t only adds wait edges (and there are $\mathcal{O}(n)$ "events" to consider), so we can simulate this and check online connectedness with Union Find to get a final $\mathcal{O}(n \log n)$ runtime.

Cable Pruning

Problem

Given a weighted connected graph G with $n \leq 10^5$ vertices *and* edges, and $k \leq 10^5$ pairs of vertices, compute the minimum sum of edges that are required to exist so as to keep these pairs of vertices connected.

Note that the resulting G' after doing this need not continue to be connected as a whole.

Cable Pruning

Problem

Given a weighted connected graph G with $n \leq 10^5$ vertices *and* edges, and $k \leq 10^5$ pairs of vertices, compute the minimum sum of edges that are required to exist so as to keep these pairs of vertices connected.
Note that the resulting G' after doing this need not continue to be connected as a whole.

Initial Observations

- For a general graph G, this is the weighted Steiner Forest problem, which is NP-hard.

Cable Pruning

Problem

Given a weighted connected graph G with $n \leq 10^5$ vertices *and* edges, and $k \leq 10^5$ pairs of vertices, compute the minimum sum of edges that are required to exist so as to keep these pairs of vertices connected.
Note that the resulting G' after doing this need not continue to be connected as a whole.

Initial Observations

- For a general graph G, this is the weighted Steiner Forest problem, which is NP-hard.
- Any connected graph with n vertices and n edges must be a "tree + one edge": one cycle, with trees growing off of the cycle vertices.

Cable Pruning

Solution

- We can identify the cycle (and the trees coming off of it) in $\mathcal{O}(n)$ with DFS.
- Try to reduce the problem to G being a cycle, along with some extra tree subproblems.
  - In a tree, every pair of vertices has a unique path between them. If a pair (u, v) is in two different trees rooted at r_1, r_2, we can decompose it into three requirements: $(u, r_1), (r_1, r_2), (r_2, v)$.

Cable Pruning

Solution

- We can identify the cycle (and the trees coming off of it) in $\mathcal{O}(n)$ with DFS.
- Try to reduce the problem to G being a cycle, along with some extra tree subproblems.
  - In a tree, every pair of vertices has a unique path between them. If a pair (u, v) is in two different trees rooted at r_1, r_2, we can decompose it into three requirements: $(u, r_1), (r_1, r_2), (r_2, v)$.
- **Part 1: Connections within a tree**
  - Many solutions, since paths are unique. For an overkill example, construct an LCA and HLD with a range-set seg tree on the edges of paths.
  - This takes time $\mathcal{O}((n + k) \log^2 n)$ (For each pair (u, v), range-set 1 on the paths from u and v to the LCA).

Cable Pruning

Solution

- We can identify the cycle (and the trees coming off of it) in $\mathcal{O}(n)$ with DFS.
- Try to reduce the problem to G being a cycle, along with some extra tree subproblems.
  - In a tree, every pair of vertices has a unique path between them. If a pair (u, v) is in two different trees rooted at r_1, r_2, we can decompose it into three requirements: $(u, r_1), (r_1, r_2), (r_2, v)$.
- **Part 1: Connections within a tree**
  - Many solutions, since paths are unique. For an overkill example, construct an LCA and HLD with a range-set seg tree on the edges of paths.
  - This takes time $\mathcal{O}((n + k) \log^2 n)$ (For each pair (u, v), range-set 1 on the paths from u and v to the LCA).
  - Can also do this in $\mathcal{O}((n + k) \log n)$ with Small-to-Large merging.

Cable Pruning

Solution

- We can identify the cycle (and the trees coming off of it) in $\mathcal{O}(n)$ with DFS.
- Try to reduce the problem to G being a cycle, along with some extra tree subproblems.
  - In a tree, every pair of vertices has a unique path between them. If a pair (u, v) is in two different trees rooted at r_1, r_2, we can decompose it into three requirements: $(u, r_1), (r_1, r_2), (r_2, v)$.
- **Part 1: Connections within a tree**
  - Many solutions, since paths are unique. For an overkill example, construct an LCA and HLD with a range-set seg tree on the edges of paths.
  - This takes time $\mathcal{O}((n + k) \log^2 n)$ (For each pair (u, v), range-set 1 on the paths from u and v to the LCA).
  - Can also do this in $\mathcal{O}((n + k) \log n)$ with Small-to-Large merging.
- **Part 2: Connections on a cycle** is less standard

Cable Pruning

Solution (cont.)

- Part 2: Connections on a cycle
- Idea: If we remove any edge from the cycle, each connection becomes unique. We should try removing each edge while walking around the cycle, and take the minimum answer.

Cable Pruning

Solution (cont.)

- Part 2: Connections on a cycle
- Idea: If we remove any edge from the cycle, each connection becomes unique. We should try removing each edge while walking around the cycle, and take the minimum answer.
- Data structure: Lazy seg tree s with a static backing array a such that the value of a range is the sum of a_i with $s_i > 0$.

Cable Pruning

Solution (cont.)

- Part 2: Connections on a cycle
- Idea: If we remove any edge from the cycle, each connection becomes unique. We should try removing each edge while walking around the cycle, and take the minimum answer.
- Data structure: Lazy seg tree s with a static backing array a such that the value of a range is the sum of a_i with $s_i > 0$.
  - Can be done by keeping track of the minimum s_i in a given block, and the number of occurrences of it (as long as $s_i \geq 0$ always).

Cable Pruning

Solution (cont.)

- Part 2: Connections on a cycle
- Idea: If we remove any edge from the cycle, each connection becomes unique. We should try removing each edge while walking around the cycle, and take the minimum answer.
- Data structure: Lazy seg tree s with a static backing array a such that the value of a range is the sum of a_i with $s_i > 0$.
  - Can be done by keeping track of the minimum s_i in a given block, and the number of occurrences of it (as long as $s_i \geq 0$ always).
- Walk around edges of the cycle. When we hit the endpoint of a pair of vertices, switch between the two possible paths (-1 on one, $+1$ on the other).

Cable Pruning

Solution (cont.)

- Part 2: Connections on a cycle
- Idea: If we remove any edge from the cycle, each connection becomes unique. We should try removing each edge while walking around the cycle, and take the minimum answer.
- Data structure: Lazy seg tree s with a static backing array a such that the value of a range is the sum of a_i with $s_i > 0$.
  - Can be done by keeping track of the minimum s_i in a given block, and the number of occurrences of it (as long as $s_i \geq 0$ always).
- Walk around edges of the cycle. When we hit the endpoint of a pair of vertices, switch between the two possible paths (-1 on one, $+1$ on the other).
- Total complexity for this step: $\mathcal{O}((n+k)\log n)$. Therefore the total is $\mathcal{O}((n+k)\log^2 n)$.

Draw Your Deck

Problem

Given $n \leq 1500$ cards, each labelled with a *nonnegative* integer at most 3, consider "drawing the deck": shuffle the deck and draw the top card. Then, draw as many cards as the integer on your card. Repeat until the deck is exhausted or you do not have cards. Assuming a random shuffle, compute the probability that you exhaust the deck.

Draw Your Deck

Problem

Given $n \leq 1500$ cards, each labelled with a *nonnegative* integer at most 3, consider "drawing the deck": shuffle the deck and draw the top card. Then, draw as many cards as the integer on your card. Repeat until the deck is exhausted or you do not have cards. Assuming a random shuffle, compute the probability that you exhaust the deck.

Initial Observations

- If the sum of the cards is less than $n - 1$, it is impossible to draw the deck.

Draw Your Deck

Problem

Given $n \leq 1500$ cards, each labelled with a *nonnegative* integer at most 3, consider "drawing the deck": shuffle the deck and draw the top card. Then, draw as many cards as the integer on your card. Repeat until the deck is exhausted or you do not have cards. Assuming a random shuffle, compute the probability that you exhaust the deck.

Initial Observations

- If the sum of the cards is less than $n - 1$, it is impossible to draw the deck.
- If the sum of the cards is $n - 1$, then you draw the deck with probability $\frac{1}{n}$ (to be proved on the next slide).

Draw Your Deck

Problem

Given $n \leq 1500$ cards, each labelled with a *nonnegative* integer at most 3, consider "drawing the deck": shuffle the deck and draw the top card. Then, draw as many cards as the integer on your card. Repeat until the deck is exhausted or you do not have cards. Assuming a random shuffle, compute the probability that you exhaust the deck.

Initial Observations

- If the sum of the cards is less than $n-1$, it is impossible to draw the deck.
- If the sum of the cards is $n-1$, then you draw the deck with probability $\frac{1}{n}$ (to be proved on the next slide).
- Otherwise, a 1 is functionally useless and can be disregarded.

Draw Your Deck

Solution

- Suppose that the sum is $n-1$ and consider any fixed arrangement of the cards. We show that there exists exactly one cyclic shift of this arrangement which allows drawing the deck, which implies that the probability is $\frac{1}{n}$.

Draw Your Deck

Solution

- Suppose that the sum is $n-1$ and consider any fixed arrangement of the cards. We show that there exists exactly one cyclic shift of this arrangement which allows drawing the deck, which implies that the probability is $\frac{1}{n}$.
  - Repeatedly do the following to the arrangement (viewed cyclically): take a nonnegative value x which has at least $x-1$ zeroes following it, and remove all x of these cards from the string.

Draw Your Deck

Solution

- Suppose that the sum is $n-1$ and consider any fixed arrangement of the cards. We show that there exists exactly one cyclic shift of this arrangement which allows drawing the deck, which implies that the probability is $\frac{1}{n}$.
  - Repeatedly do the following to the arrangement (viewed cyclically): take a nonnegative value x which has at least $x-1$ zeroes following it, and remove all x of these cards from the string.
  - To prove that we remove all nonzero cards, note that at every step we preserve the invariant that the sum is $n'-1$.
  - For each nonzero card i, let s_i be the number of zeros immediately following it.
  - If $s_i+1 < a_i$ for all i, then by summing it follows that the total number of cards is less than $n-1$, contradiction.

Draw Your Deck

Solution

- Suppose that the sum is $n-1$ and consider any fixed arrangement of the cards. We show that there exists exactly one cyclic shift of this arrangement which allows drawing the deck, which implies that the probability is $\frac{1}{n}$.
  - Repeatedly do the following to the arrangement (viewed cyclically): take a nonnegative value x which has at least $x-1$ zeroes following it, and remove all x of these cards from the string.
  - To prove that we remove all nonzero cards, note that at every step we preserve the invariant that the sum is $n'-1$.
  - For each nonzero card i, let s_i be the number of zeros immediately following it.
  - If $s_i + 1 < a_i$ for all i, then by summing it follows that the total number of cards is less than $n-1$, contradiction.
  - Since the total sum is $n-1$, there must be exactly one zero remaining, and the only valid cyclic shift must place this last.

Draw Your Deck

Solution (cont.)

- Suppose that the sum is at least n (then, 1s do not matter).

Draw Your Deck

Solution (cont.)

- Suppose that the sum is at least n (then, 1s do not matter).
- Else, we can do DP on the number of 0s (a), the number of 2s (b), and the number of 3s (c).
- Since $a + b + c = n$, $abc \leq (\frac{n}{3})^3$ by AM-GM, which is fast enough and fits in memory when $n \leq 1500$.

Draw Your Deck

Solution (cont.)

- Suppose that the sum is at least n (then, 1s do not matter).
- Else, we can do DP on the number of 0s (a), the number of 2s (b), and the number of 3s (c).
- Since $a + b + c = n$, $abc \leq (\frac{n}{3})^3$ by AM-GM, which is fast enough and fits in memory when $n \leq 1500$.
- At a state $\{a, b, c\}$, we can compute the number of "accumulated" draws and short circuit if this is 0 or more than the number of remaining cards.
- Else, we can recurse to $\{a - 1, b, c\}$ and the other two possibilities.

Draw Your Deck

Solution (cont.)

- Suppose that the sum is at least n (then, 1s do not matter).
- Else, we can do DP on the number of 0s (a), the number of 2s (b), and the number of 3s (c).
- Since $a + b + c = n$, $abc \leq (\frac{n}{3})^3$ by AM-GM, which is fast enough and fits in memory when $n \leq 1500$.
- At a state $\{a, b, c\}$, we can compute the number of "accumulated" draws and short circuit if this is 0 or more than the number of remaining cards.
- Else, we can recurse to $\{a - 1, b, c\}$ and the other two possibilities.
- Total complexity: $\mathcal{O}((n/3)^3)$ time and space.

Draw Your Deck

Solution (cont.)

- Suppose that the sum is at least n (then, 1s do not matter).
- Else, we can do DP on the number of 0s (a), the number of 2s (b), and the number of 3s (c).
- Since $a + b + c = n$, $abc \leq (\frac{n}{3})^3$ by AM-GM, which is fast enough and fits in memory when $n \leq 1500$.
- At a state $\{a, b, c\}$, we can compute the number of "accumulated" draws and short circuit if this is 0 or more than the number of remaining cards.
- Else, we can recurse to $\{a - 1, b, c\}$ and the other two possibilities.
- Total complexity: $\mathcal{O}((n/3)^3)$ time and space.
- **Bonus:** Solve this in $\mathcal{O}(n^2)$ time and $\mathcal{O}(n)$ space with some clever counting.

Evil Judges

Problem

Given a string s with $|s| \leq 2 \cdot 10^5$ consisting only of the characters `A`, `C`, and `K`, define an operation as swapping two adjacent characters. Given at most $M \leq 10^9$ operations, minimize the combined total number of `AC` and `AK` subsequences.

Evil Judges

Problem

Given a string s with $|s| \leq 2 \cdot 10^5$ consisting only of the characters `A`, `C`, and `K`, define an operation as swapping two adjacent characters. Given at most $M \leq 10^9$ operations, minimize the combined total number of `AC` and `AK` subsequences.

Initial Observations

- Actually doing all M operations is too costly.

Evil Judges

Problem

Given a string s with $|s| \leq 2 \cdot 10^5$ consisting only of the characters `A`, `C`, and `K`, define an operation as swapping two adjacent characters. Given at most $M \leq 10^9$ operations, minimize the combined total number of `AC` and `AK` subsequences.

Initial Observations

- Actually doing all M operations is too costly.
- Equivalently, we may consider this as a binary string with $\texttt{A} = 1$ and $\texttt{C}, \texttt{K} = 0$. We want to minimize the number of inversions in this string.

Evil Judges

Solution

- As long as the string is not sorted, swapping any 10 to 01 decreases the number of inversions by 1.

Evil Judges

Solution

- As long as the string is not sorted, swapping any 10 to 01 decreases the number of inversions by 1.
- Therefore, the answer is just the number of "inversions" in the original string, minus M (or 0, if we can sort the whole thing).

Evil Judges

Solution

- As long as the string is not sorted, swapping any 10 to 01 decreases the number of inversions by 1.
- Therefore, the answer is just the number of "inversions" in the original string, minus M (or 0, if we can sort the whole thing).
- For each `C`/`K`, the number of inversions it contributes is exactly the number of `A`'s before it.

Evil Judges

Solution

- As long as the string is not sorted, swapping any 10 to 01 decreases the number of inversions by 1.
- Therefore, the answer is just the number of "inversions" in the original string, minus M (or 0, if we can sort the whole thing).
- For each `C`/`K`, the number of inversions it contributes is exactly the number of `A`'s before it.
- We can compute this in $\mathcal{O}(n)$ by sweeping once along the string. Be careful not to use `int`!

Friend Meetup

Problem

Given $n \leq 2 \cdot 10^5$ people on the lattice, each with an walking speed $v_i > 0$, compute the maximum time required for any pair of people to meet, if they can only walk along grid lines.

Friend Meetup

Problem

Given $n \leq 2 \cdot 10^5$ people on the lattice, each with an walking speed $v_i > 0$, compute the maximum time required for any pair of people to meet, if they can only walk along grid lines.

Initial Observations

- Equivalently, we want to compute $M = \max_{i,j} \frac{|x_i - x_j| + |y_i - y_j|}{v_i + v_j}$.

Friend Meetup

Problem

Given $n \leq 2 \cdot 10^5$ people on the lattice, each with an walking speed $v_i > 0$, compute the maximum time required for any pair of people to meet, if they can only walk along grid lines.

Initial Observations

- Equivalently, we want to compute $M = \max_{i,j} \frac{|x_i - x_j| + |y_i - y_j|}{v_i + v_j}$.
- Recall that $|r - s| = \max(r - s, s - r)$. If we let $a_i = x_i + y_i$ and $b_i = x_i - y_i$, then

$$M = \max_{i,j} \frac{|x_i - x_j| + |y_i - y_j|}{v_i + v_j} = \max_{i,j} \left\{ \frac{a_i - a_j}{v_i + v_j}, \frac{b_i - b_j}{v_i + v_j} \right\}.$$

Friend Meetup

Solution

- Binary search on the answer.

Friend Meetup

Solution

- Binary search on the answer.
- Then, $M \leq t$ iff both $\frac{a_i - a_j}{v_i + v_j}$ and $\frac{b_i - b_j}{v_i + v_j}$ are at most t for every pair i, j.

Friend Meetup

Solution

- Binary search on the answer.
- Then, $M \leq t$ iff both $\frac{a_i - a_j}{v_i + v_j}$ and $\frac{b_i - b_j}{v_i + v_j}$ are at most t for every pair i, j.
- We may rewrite $\frac{a_i - a_j}{v_i + v_j} \leq m$ as $a_i - tv_i \leq a_j + tv_j$, so equivalently we need $\max_i(a_i - tv_i) \leq \min_i(a_i + tv_i)$ (and the same for b).

Friend Meetup

Solution

- Binary search on the answer.
- Then, $M \leq t$ iff both $\frac{a_i - a_j}{v_i + v_j}$ and $\frac{b_i - b_j}{v_i + v_j}$ are at most t for every pair i, j.
- We may rewrite $\frac{a_i - a_j}{v_i + v_j} \leq m$ as $a_i - tv_i \leq a_j + tv_j$, so equivalently we need $\max_i(a_i - tv_i) \leq \min_i(a_i + tv_i)$ (and the same for b).
- Each such check takes time $\mathcal{O}(n)$, so the overall runtime is $\mathcal{O}(n \log \frac{\text{MAX}}{\varepsilon})$.

Gemstone Dowsing

Problem

There are $n \leq 10^5$ gemstones hidden in the ground ($y_i < 0$). You know the locations of k of these, and additionally you are given n *readings* from the surface: each reading gives the closest gemstone. It is guaranteed that each gemstone is in exactly one reading. For each undiscovered gemstone, compute the furthest from the surface it could be at, *independently* of the other undiscovered gemstones.

Gemstone Dowsing

Problem

There are $n \leq 10^5$ gemstones hidden in the ground ($y_i < 0$). You know the locations of k of these, and additionally you are given n *readings* from the surface: each reading gives the closest gemstone. It is guaranteed that each gemstone is in exactly one reading. For each undiscovered gemstone, compute the furthest from the surface it could be at, *independently* of the other undiscovered gemstones.

Initial Observations

- Each reading defines a semicircle with radius being the distance to the closest known gemstone, and the gemstone corresponding to the reading must lie within (or on the boundary of) this semicircle.

Gemstone Dowsing

Problem

There are $n \leq 10^5$ gemstones hidden in the ground ($y_i < 0$). You know the locations of k of these, and additionally you are given n *readings* from the surface: each reading gives the closest gemstone. It is guaranteed that each gemstone is in exactly one reading. For each undiscovered gemstone, compute the furthest from the surface it could be at, *independently* of the other undiscovered gemstones.

Initial Observations

- Each reading defines a semicircle with radius being the distance to the closest known gemstone, and the gemstone corresponding to the reading must lie within (or on the boundary of) this semicircle.
- The x positions of gems must be in the same order as their readings (curvature considerations).

Gemstone Dowsing

Solution

- Observation: as long as the known gemstones are in the same order as their readings, there always exists a way of placing the undiscovered gemstones.
  - From the first unknown gemstone, iteratively place at the closest known gemstone, and mark as known.

Gemstone Dowsing

Solution

- Observation: as long as the known gemstones are in the same order as their readings, there always exists a way of placing the undiscovered gemstones.
  - From the first unknown gemstone, iteratively place at the closest known gemstone, and mark as known.
- Consider an arbitrary unknown gemstone and place it anywhere in its semicircle that isn't in any known semicircle: then, the above argument works to place all other gemstones (go iteratively in both directions)

Gemstone Dowsing

Solution

- Observation: as long as the known gemstones are in the same order as their readings, there always exists a way of placing the undiscovered gemstones.
  - From the first unknown gemstone, iteratively place at the closest known gemstone, and mark as known.
- Consider an arbitrary unknown gemstone and place it anywhere in its semicircle that isn't in any known semicircle: then, the above argument works to place all other gemstones (go iteratively in both directions)
- Only the nearest known gemstones from both sides matter (curvature considerations again).

Gemstone Dowsing

Solution

- Observation: as long as the known gemstones are in the same order as their readings, there always exists a way of placing the undiscovered gemstones.
  - From the first unknown gemstone, iteratively place at the closest known gemstone, and mark as known.
- Consider an arbitrary unknown gemstone and place it anywhere in its semicircle that isn't in any known semicircle: then, the above argument works to place all other gemstones (go iteratively in both directions)
- Only the nearest known gemstones from both sides matter (curvature considerations again).
- For each unknown reading U with A and B known around it, only the extreme point of the U semicircle and the intersections with A and B semicircles matter.
- Total runtime: $O(n)$ to sweep and do $O(1)$ work per reading.

Heist of the Century

Problem

There is a hidden array h of length $n \leq 500$ with $1 \leq h_i \leq 2n$. Up to $4n$ times, you can query an array d with $1 \leq d_i \leq 2n$ and receive back $q_d = \max_i |d_i - h_i|$. Reconstruct h.

Heist of the Century

Problem

There is a hidden array h of length $n \leq 500$ with $1 \leq h_i \leq 2n$. Up to $4n$ times, you can query an array d with $1 \leq d_i \leq 2n$ and receive back $q_d = \max_i |d_i - h_i|$. Reconstruct h.

Initial Observations

- The bounds suggest using at most 4 queries to reconstruct each h_i.

Heist of the Century

Problem

There is a hidden array h of length $n \leq 500$ with $1 \leq h_i \leq 2n$. Up to $4n$ times, you can query an array d with $1 \leq d_i \leq 2n$ and receive back $q_d = \max_i |d_i - h_i|$. Reconstruct h.

Initial Observations

- The bounds suggest using at most 4 queries to reconstruct each h_i.
- We can always force $q_a \leq n$ by setting each d_i to n or $n+1$.

Heist of the Century

Problem

There is a hidden array h of length $n \leq 500$ with $1 \leq h_i \leq 2n$. Up to $4n$ times, you can query an array d with $1 \leq d_i \leq 2n$ and receive back $q_d = \max_i |d_i - h_i|$. Reconstruct h.

Initial Observations

- The bounds suggest using at most 4 queries to reconstruct each h_i.
- We can always force $q_a \leq n$ by setting each d_i to n or $n+1$.
- If $h_i \neq n, n+1$ then trying $d_i = 1, 2n$ and setting all others to n will decode h_i.

Heist of the Century

Solution

- We will try to uncover each h_i in order.

Heist of the Century

Solution

- We will try to uncover each h_i in order.
- WLOG look at $i = 1$. Try the arrays $a = [1, n, n, \ldots, n]$ and $b = [2n, n, n, \ldots, n]$.
  - If $q_a > n$, then $h_i = q_a + 1$ and if $q_b > n$, then $h_i = 2n - q_b$.
  - Else, h_i must be either n or $n + 1$; set it to n and keep track of this index.

Heist of the Century

Solution

- We will try to uncover each h_i in order.
- WLOG look at $i = 1$. Try the arrays $a = [1, n, n, \ldots, n]$ and $b = [2n, n, n, \ldots, n]$.
  - If $q_a > n$, then $h_i = q_a + 1$ and if $q_b > n$, then $h_i = 2n - q_b$.
  - Else, h_i must be either n or $n + 1$; set it to n and keep track of this index.
- The new guess for h has $q_h \leq 1$, so for each index where $h_i = n$, we can try d which is h except $d_i = 1$. We can then recover $h_i = q_d + 1$ since this must be the "binding" index.

Heist of the Century

Solution

- We will try to uncover each h_i in order.
- WLOG look at $i = 1$. Try the arrays $a = [1, n, n, \ldots, n]$ and $b = [2n, n, n, \ldots, n]$.
  - If $q_a > n$, then $h_i = q_a + 1$ and if $q_b > n$, then $h_i = 2n - q_b$.
  - Else, h_i must be either n or $n + 1$; set it to n and keep track of this index.
- The new guess for h has $q_h \leq 1$, so for each index where $h_i = n$, we can try d which is h except $d_i = 1$. We can then recover $h_i = q_d + 1$ since this must be the "binding" index.
- Finally, guess the correct h if this hasn't happened yet.

Heist of the Century

Solution

- We will try to uncover each h_i in order.
- WLOG look at $i = 1$. Try the arrays $a = [1, n, n, \ldots, n]$ and $b = [2n, n, n, \ldots, n]$.
  - If $q_a > n$, then $h_i = q_a + 1$ and if $q_b > n$, then $h_i = 2n - q_b$.
  - Else, h_i must be either n or $n + 1$; set it to n and keep track of this index.
- The new guess for h has $q_h \leq 1$, so for each index where $h_i = n$, we can try d which is h except $d_i = 1$. We can then recover $h_i = q_d + 1$ since this must be the "binding" index.
- Finally, guess the correct h if this hasn't happened yet.
- We use at most $3n + 1$ queries, each query taking time $\mathcal{O}(n)$.

I Don't Miss Pennies

Problem

Given $n \leq 3 \cdot 10^5$ item costs (in cents), group them optimally so that the "savings" from rounding to the nearest multiple of 5 is maximized.

I Don't Miss Pennies

Problem

Given $n \leq 3 \cdot 10^5$ item costs (in cents), group them optimally so that the "savings" from rounding to the nearest multiple of 5 is maximized.

Some Observations

- Consider item costs only mod 5.

I Don't Miss Pennies

Problem

Given $n \leq 3 \cdot 10^5$ item costs (in cents), group them optimally so that the "savings" from rounding to the nearest multiple of 5 is maximized.

Some Observations

- Consider item costs only mod 5.
- If (mod 5) the cost is a 0, 1, or 2, then group it on its own.

I Don't Miss Pennies

Problem

Given $n \leq 3 \cdot 10^5$ item costs (in cents), group them optimally so that the "savings" from rounding to the nearest multiple of 5 is maximized.

Some Observations

- Consider item costs only mod 5.
- If (mod 5) the cost is a 0, 1, or 2, then group it on its own.
- Only have to handle 3s and 4s.

I Don't Miss Pennies

Solution

- As noted above, put 0s, 1s, and 2s on their own (one way to convince yourself that this is good is to consider the "per capita" savings of each of these).

I Don't Miss Pennies

Solution

- As noted above, put 0s, 1s, and 2s on their own (one way to convince yourself that this is good is to consider the "per capita" savings of each of these).
- As long as there are both 3s and 4s, pair them together since $3 + 4 = 2 \bmod 5$ (the marginal savings is 1)

I Don't Miss Pennies

Solution

- As noted above, put 0s, 1s, and 2s on their own (one way to convince yourself that this is good is to consider the "per capita" savings of each of these).
- As long as there are both 3s and 4s, pair them together since $3 + 4 = 2 \bmod 5$ (the marginal savings is 1)
- The remaining 3s should be grouped in pairs ($3 + 3 = 1 \bmod 5$) and 4s in threes ($4 + 4 + 4 \equiv 2 \bmod 5$).

I Don't Miss Pennies

Solution

- As noted above, put 0s, 1s, and 2s on their own (one way to convince yourself that this is good is to consider the "per capita" savings of each of these).
- As long as there are both 3s and 4s, pair them together since $3 + 4 = 2 \bmod 5$ (the marginal savings is 1)
- The remaining 3s should be grouped in pairs ($3 + 3 = 1 \bmod 5$) and 4s in threes ($4 + 4 + 4 \equiv 2 \bmod 5$).
- Expected complexity: $\mathcal{O}(n)$ (for reading the input). The rest can be done in $\mathcal{O}(1)$.

Jelly Fusion

Problem

Consider a circular array of N jellies. The i^{th} jelly has width w_i and height h_i. In one move, you can "fuse" two adjacent jellies into a new one with width $\max(w_i, w_{i+1})$ and height $\max(h_i, h_{i+1})$. What is the maximum area over all jellies after some (possibly none) fusion operations?

Jelly Fusion

Problem

Consider a circular array of N jellies. The i^{th} jelly has width w_i and height h_i. In one move, you can "fuse" two adjacent jellies into a new one with width $\max(w_i, w_{i+1})$ and height $\max(h_i, h_{i+1})$. What is the maximum area over all jellies after some (possibly none) fusion operations?

Hints

- Solve this problem on a (non-circular) array.

Jelly Fusion

Problem

Consider a circular array of N jellies. The i^{th} jelly has width w_i and height h_i. In one move, you can "fuse" two adjacent jellies into a new one with width $\max(w_i, w_{i+1})$ and height $\max(h_i, h_{i+1})$. What is the maximum area over all jellies after some (possibly none) fusion operations?

Hints

- Solve this problem on a (non-circular) array.
- Call a contiguous sequence of jellies a "segment". Where in the segment is the maximum-width jelly?

Jelly Fusion

Problem

Consider a circular array of N jellies. The i^{th} jelly has width w_i and height h_i. In one move, you can "fuse" two adjacent jellies into a new one with width $\max(w_i, w_{i+1})$ and height $\max(h_i, h_{i+1})$. What is the maximum area over all jellies after some (possibly none) fusion operations?

Hints

- Solve this problem on a (non-circular) array.
- Call a contiguous sequence of jellies a "segment". Where in the segment is the maximum-width jelly?

Jelly Fusion

Solving on a line

- Consider solving on a line. The naive dynamic programming is $dp_i = \max_{j<i} dp_j + \max(w_j, w_{j+1}, \ldots, w_i) \cdot \max(h_j, h_{j+1}, \ldots, h_i)$. This is $\mathcal{O}(n^3)$ if implemented naively or $\mathcal{O}(n^2)$ by keeping suffix maxima.

Jelly Fusion

Solving on a line

- Consider solving on a line. The naive dynamic programming is $dp_i = \max_{j<i} dp_j + \max(w_j, w_{j+1}, \ldots, w_i) \cdot \max(h_j, h_{j+1}, \ldots, h_i)$. This is $\mathcal{O}(n^3)$ if implemented naively or $\mathcal{O}(n^2)$ by keeping suffix maxima.
- By exchange argument, the maximum-width jelly must be at an end of a fused segment. Same for the maximum height.

Jelly Fusion

Solving on a line

- Consider solving on a line. The naive dynamic programming is $dp_i = \max_{j<i} dp_j + \max(w_j, w_{j+1}, \ldots, w_i) \cdot \max(h_j, h_{j+1}, \ldots, h_i)$. This is $\mathcal{O}(n^3)$ if implemented naively or $\mathcal{O}(n^2)$ by keeping suffix maxima.
- By exchange argument, the maximum-width jelly must be at an end of a fused segment. Same for the maximum height.
- Stated another way, each segment is either of size 1, or has its maximum-width jelly at one end and maximum-height jelly at the other end.

Jelly Fusion

Solving on a line

- Consider solving on a line. The naive dynamic programming is $dp_i = \max_{j<i} dp_j + \max(w_j, w_{j+1}, \ldots, w_i) \cdot \max(h_j, h_{j+1}, \ldots, h_i)$. This is $\mathcal{O}(n^3)$ if implemented naively or $\mathcal{O}(n^2)$ by keeping suffix maxima.
- By exchange argument, the maximum-width jelly must be at an end of a fused segment. Same for the maximum height.
- Stated another way, each segment is either of size 1, or has its maximum-width jelly at one end and maximum-height jelly at the other end.
- dp_i is therefore always of the form $dp_{i-1} + w_i \cdot h_i$, $dp_j + w_i \cdot h_j$ or $dp_j + w_j \cdot h_i$.

Jelly Fusion

Solving on a line

- Consider solving on a line. The naive dynamic programming is $dp_i = \max_{j<i} dp_j + \max(w_j, w_{j+1}, \ldots, w_i) \cdot \max(h_j, h_{j+1}, \ldots, h_i)$. This is $\mathcal{O}(n^3)$ if implemented naively or $\mathcal{O}(n^2)$ by keeping suffix maxima.
- By exchange argument, the maximum-width jelly must be at an end of a fused segment. Same for the maximum height.
- Stated another way, each segment is either of size 1, or has its maximum-width jelly at one end and maximum-height jelly at the other end.
- dp_i is therefore always of the form $dp_{i-1} + w_i \cdot h_i$, $dp_j + w_i \cdot h_j$ or $dp_j + w_j \cdot h_i$.
- The best index j for the latter two forms can be computed using the *convex hull trick*.

Jelly Fusion

Solving on a line

- Consider solving on a line. The naive dynamic programming is $dp_i = \max_{j<i} dp_j + \max(w_j, w_{j+1}, \ldots, w_i) \cdot \max(h_j, h_{j+1}, \ldots, h_i)$. This is $\mathcal{O}(n^3)$ if implemented naively or $\mathcal{O}(n^2)$ by keeping suffix maxima.
- By exchange argument, the maximum-width jelly must be at an end of a fused segment. Same for the maximum height.
- Stated another way, each segment is either of size 1, or has its maximum-width jelly at one end and maximum-height jelly at the other end.
- dp_i is therefore always of the form $dp_{i-1} + w_i \cdot h_i$, $dp_j + w_i \cdot h_j$ or $dp_j + w_j \cdot h_i$.
- The best index j for the latter two forms can be computed using the *convex hull trick*.
- Because the lines are added to the data structure in increasing order of intercept, not slope, this requires either a balanced binary search tree of lines, or a Li-Chao segment tree. Complexity is $\mathcal{O}(N \log N)$.

Jelly Fusion

Solving on a circle

- Even on a cycle, the maximum-width jelly must be at one end of its fused segment.

Jelly Fusion

Solving on a circle

- Even on a cycle, the maximum-width jelly must be at one end of its fused segment.
- Find the first occurrence of a jelly with maximum width. Consider the two rotations where this jelly is at the beginning or end. Take the maximum over both cases.

Kindergarten Revisited

Problem

Given $n \leq 100$ rectangles on an infinitely large piece of paper, determine the minimum number of straight-line cuts of the paper required to make each rectangle be on its own piece.

Kindergarten Revisited

Problem

Given $n \leq 100$ rectangles on an infinitely large piece of paper, determine the minimum number of straight-line cuts of the paper required to make each rectangle be on its own piece.

Some Observations

- The answer is always either $n - 1$ or `impossible`, by induction.

Kindergarten Revisited

Problem

Given $n \leq 100$ rectangles on an infinitely large piece of paper, determine the minimum number of straight-line cuts of the paper required to make each rectangle be on its own piece.

Some Observations

- The answer is always either $n - 1$ or `impossible`, by induction.
- We only have to consider lines which go through pairs of vertices of (all) the rectangles, as any line can be adjusted to go through vertices.

Kindergarten Revisited

Problem

Given $n \leq 100$ rectangles on an infinitely large piece of paper, determine the minimum number of straight-line cuts of the paper required to make each rectangle be on its own piece.

Some Observations

- The answer is always either $n - 1$ or `impossible`, by induction.
- We only have to consider lines which go through pairs of vertices of (all) the rectangles, as any line can be adjusted to go through vertices.
- If we can find any line cutting at least one rectangle away, we can recurse on both sides.

Kindergarten Revisited

Solution

- For two vertices, we can compute the number of rectangles to the left, right, and nicked by it in $\mathcal{O}(n)$.
- Can precompute for all pairs of vertices the above quantities in $\mathcal{O}(n^3)$ time, $\mathcal{O}(n^2)$ space.

Kindergarten Revisited

Solution

- For two vertices, we can compute the number of rectangles to the left, right, and nicked by it in $\mathcal{O}(n)$.
- Can precompute for all pairs of vertices the above quantities in $\mathcal{O}(n^3)$ time, $\mathcal{O}(n^2)$ space.
- Given this computation, can find a line not nicking any rectangles in $\mathcal{O}(n^2)$.
- Let L, R be the rectangles on the left and right of the line.
- Before recursing on each, subtract from the precomputed values the contribution of (rectangle in L, pairs of vertices in R) and vice versa.

Kindergarten Revisited

Solution

- For two vertices, we can compute the number of rectangles to the left, right, and nicked by it in $\mathcal{O}(n)$.
- Can precompute for all pairs of vertices the above quantities in $\mathcal{O}(n^3)$ time, $\mathcal{O}(n^2)$ space.
- Given this computation, can find a line not nicking any rectangles in $\mathcal{O}(n^2)$.
- Let L, R be the rectangles on the left and right of the line.
- Before recursing on each, subtract from the precomputed values the contribution of (rectangle in L, pairs of vertices in R) and vice versa.
- Total number of subtractions is $\mathcal{O}(n^3)$ (each addition at the start is subtracted exactly once) and total cost of recursion is $\mathcal{O}(n^3)$ as well, so the total time is $\mathcal{O}(n^3)$.

Kindergarten Revisited

Solution

- For two vertices, we can compute the number of rectangles to the left, right, and nicked by it in $\mathcal{O}(n)$.
- Can precompute for all pairs of vertices the above quantities in $\mathcal{O}(n^3)$ time, $\mathcal{O}(n^2)$ space.
- Given this computation, can find a line not nicking any rectangles in $\mathcal{O}(n^2)$.
- Let L, R be the rectangles on the left and right of the line.
- Before recursing on each, subtract from the precomputed values the contribution of (rectangle in L, pairs of vertices in R) and vice versa.
- Total number of subtractions is $\mathcal{O}(n^3)$ (each addition at the start is subtracted exactly once) and total cost of recursion is $\mathcal{O}(n^3)$ as well, so the total time is $\mathcal{O}(n^3)$.
- **Note:** With some careful counting of operations, an $\mathcal{O}(n^4)$ can pass as well.
- **Bonus:** Solve the problem in $\mathcal{O}(n^2 \log n)$.

Leaking Santa's Secrets

Problem

Deduce a permutation H of length $N \leq 220$ using at most 1 510 queries. Each query is a permutation Q and gives the number of cycles in the permutation $H \circ Q$.

Leaking Santa's Secrets

Problem

Deduce a permutation H of length $N \leq 220$ using at most 1 510 queries. Each query is a permutation Q and gives the number of cycles in the permutation $H \circ Q$.

Hints

- Say the number of cycles is N. How can we determine H immediately?

Leaking Santa's Secrets

Problem

Deduce a permutation H of length $N \leq 220$ using at most 1 510 queries. Each query is a permutation Q and gives the number of cycles in the permutation $H \circ Q$.

Hints

- Say the number of cycles is N. How can we determine H immediately?
- What happens to the number of cycles when only two elements are swapped?

Leaking Santa's Secrets

Problem

Deduce a permutation H of length $N \leq 220$ using at most 1 510 queries. Each query is a permutation Q and gives the number of cycles in the permutation $H \circ Q$.

Hints

- Say the number of cycles is N. How can we determine H immediately?
- What happens to the number of cycles when only two elements are swapped?
- How can we do many swaps in a single query?

Leaking Santa's Secrets

Solution

- First query the identity permutation to get a baseline. Call the initial answer C.

Leaking Santa's Secrets

Solution

- First query the identity permutation to get a baseline. Call the initial answer C.
- Query a permutation Q which swaps elements i and j. If these two elements are part of the same cycle, this will return $C + 1$, otherwise $C - 1$.

Leaking Santa's Secrets

Solution

- First query the identity permutation to get a baseline. Call the initial answer C.
- Query a permutation Q which swaps elements i and j. If these two elements are part of the same cycle, this will return $C + 1$, otherwise $C - 1$.
- This works if you compose multiple swaps together. Say you query permutation Q, which is the composition of K swaps. Then if all elements are in different cycles, the query returns $C - K$.

Leaking Santa's Secrets

Solution

- First query the identity permutation to get a baseline. Call the initial answer C.
- Query a permutation Q which swaps elements i and j. If these two elements are part of the same cycle, this will return $C + 1$, otherwise $C - 1$.
- This works if you compose multiple swaps together. Say you query permutation Q, which is the composition of K swaps. Then if all elements are in different cycles, the query returns $C - K$.
- When adding a new element i, we can binary search on the found cycles to determine which cycle this new element belongs to.

Leaking Santa's Secrets

Solution

- First query the identity permutation to get a baseline. Call the initial answer C.
- Query a permutation Q which swaps elements i and j. If these two elements are part of the same cycle, this will return $C + 1$, otherwise $C - 1$.
- This works if you compose multiple swaps together. Say you query permutation Q, which is the composition of K swaps. Then if all elements are in different cycles, the query returns $C - K$.
- When adding a new element i, we can binary search on the found cycles to determine which cycle this new element belongs to.
- If the element is in the same cycle as a previous element, incorporate a swap into all future queries. Once the number of cycles is N, then you know that $H = Q^{-1}$, and can invert Q to find the final solution.

Leaking Santa's Secrets

Solution

- First query the identity permutation to get a baseline. Call the initial answer C.
- Query a permutation Q which swaps elements i and j. If these two elements are part of the same cycle, this will return $C + 1$, otherwise $C - 1$.
- This works if you compose multiple swaps together. Say you query permutation Q, which is the composition of K swaps. Then if all elements are in different cycles, the query returns $C - K$.
- When adding a new element i, we can binary search on the found cycles to determine which cycle this new element belongs to.
- If the element is in the same cycle as a previous element, incorporate a swap into all future queries. Once the number of cycles is N, then you know that $H = Q^{-1}$, and can invert Q to find the final solution.
- With a careful binary search implementation, the total number of queries is $1 + \sum_{i=1}^{220} \log_2(i+1) = 1506$.

Maki Conveyor Belt

Problem

There are two people, Alice and Bob, at a maki restaurant. As time goes on, they have the option of purchasing maki in front of them. What is the minimum amount of time for both of them to get the maki they want? Among all strategies that use the least amount of time, what is the minimum cost?

Maki Conveyor Belt

Problem

There are two people, Alice and Bob, at a maki restaurant. As time goes on, they have the option of purchasing maki in front of them. What is the minimum amount of time for both of them to get the maki they want? Among all strategies that use the least amount of time, what is the minimum cost?

Some Observations

- For each pair of (plate, person), we can compute exactly when that plate appears in front of that person.

Maki Conveyor Belt

Problem

There are two people, Alice and Bob, at a maki restaurant. As time goes on, they have the option of purchasing maki in front of them. What is the minimum amount of time for both of them to get the maki they want? Among all strategies that use the least amount of time, what is the minimum cost?

Some Observations

- For each pair of (plate, person), we can compute exactly when that plate appears in front of that person.
- Maki types are completely independent.

Maki Conveyor Belt

Problem

There are two people, Alice and Bob, at a maki restaurant. As time goes on, they have the option of purchasing maki in front of them. What is the minimum amount of time for both of them to get the maki they want? Among all strategies that use the least amount of time, what is the minimum cost?

Some Observations

- For each pair of (plate, person), we can compute exactly when that plate appears in front of that person.
- Maki types are completely independent.
- If we know the set of plates that appear in front of each person, we can see if they can both be satisfied with a greedy algorithm.

Maki Conveyor Belt

Greedy Algorithm

- Say the conveyor belt rotates a fixed r times.

Maki Conveyor Belt

Greedy Algorithm

- Say the conveyor belt rotates a fixed r times.
- This allows us to categorize each plate into one of 4 types, depending on which set of people have access to it (Alice-only, Bob-only, shared, or unavailable). This can be computed by comparing the distance from the plate to each diner and comparing with r.

Maki Conveyor Belt

Greedy Algorithm

- Say the conveyor belt rotates a fixed r times.
- This allows us to categorize each plate into one of 4 types, depending on which set of people have access to it (Alice-only, Bob-only, shared, or unavailable). This can be computed by comparing the distance from the plate to each diner and comparing with r.
- To see if Alice and Bob can be satisfied with maki type i, they need to eat $a_i + b_i$ total maki, with at most a_i Alice maki and at most b_i Bob maki.

Maki Conveyor Belt

Greedy Algorithm

- Say the conveyor belt rotates a fixed r times.
- This allows us to categorize each plate into one of 4 types, depending on which set of people have access to it (Alice-only, Bob-only, shared, or unavailable). This can be computed by comparing the distance from the plate to each diner and comparing with r.
- To see if Alice and Bob can be satisfied with maki type i, they need to eat $a_i + b_i$ total maki, with at most a_i Alice maki and at most b_i Bob maki.
- Therefore, maki-type i can be satisfied iff $\min(a_i, \text{A-only}) + \min(b_i, \text{B-only}) + \text{shared} \geq a_i + b_i$.

Maki Conveyor Belt

Greedy Algorithm

- Say the conveyor belt rotates a fixed r times.
- This allows us to categorize each plate into one of 4 types, depending on which set of people have access to it (Alice-only, Bob-only, shared, or unavailable). This can be computed by comparing the distance from the plate to each diner and comparing with r.
- To see if Alice and Bob can be satisfied with maki type i, they need to eat $a_i + b_i$ total maki, with at most a_i Alice maki and at most b_i Bob maki.
- Therefore, maki-type i can be satisfied iff $\min(a_i, \text{A-only}) + \min(b_i, \text{B-only}) + \text{shared} \geq a_i + b_i$.
- To incorporate costs, sort by cost and take the minimum of the sushi available and a_i, b_i or $a_i + b_i$ depending on the plate type.

Maki Conveyor Belt

Greedy Algorithm

- Say the conveyor belt rotates a fixed r times.
- This allows us to categorize each plate into one of 4 types, depending on which set of people have access to it (Alice-only, Bob-only, shared, or unavailable). This can be computed by comparing the distance from the plate to each diner and comparing with r.
- To see if Alice and Bob can be satisfied with maki type i, they need to eat $a_i + b_i$ total maki, with at most a_i Alice maki and at most b_i Bob maki.
- Therefore, maki-type i can be satisfied iff $\min(a_i, \text{A-only}) + \min(b_i, \text{B-only}) + \text{shared} \geq a_i + b_i$.
- To incorporate costs, sort by cost and take the minimum of the sushi available and a_i, b_i or $a_i + b_i$ depending on the plate type.
- Total runtime of this algorithm is $\mathcal{O}(K \log K + M)$ for sorting.

Maki Conveyor Belt

Optimizing: Binary Search

One path to a solution is binary searching on the number of rotations. If a number of rotations r is feasible, then any $r' > r$ is also feasible, satisfying the monotonicity requirement. Total runtime $\mathcal{O}((K \log K + M) \log N)$

Maki Conveyor Belt

Optimizing: Binary Search

One path to a solution is binary searching on the number of rotations. If a number of rotations r is feasible, then any $r' > r$ is also feasible, satisfying the monotonicity requirement. Total runtime $\mathcal{O}((K \log K + M) \log N)$

Optimizing: Incremental

Another option is to find all "events" where a plate is in front of a diner and sort them by time. Then adapt the above computation incrementally. This requires adjusting the types of the plates on each update and updating their assignments. Slightly better time complexity of $\mathcal{O}(K \log K + M)$ but for this data size the runtimes are comparable.

For both solution strategies, it is easier to compute the minimum time first, then compute the minimum cost in a separate pass.