Algorithmic Engagements contest

Presentation of solutions

A. Interesting Paths

Fastest solution: **UTokyo: Time Manipulators (0:12)**

A. Interesting Paths

Problem statement

Given is a DAG with n vertices and m edges.
What is the longest possible sequence of paths in which each path:

- starts in the source (vertex 1) and finishes in the sink (vertex n)
- contains at least one edge not contained in any of the previous paths

A. Interesting Paths

Solution

- If you can't get to the sink from the source, the answer is 0.

A. Interesting Paths

Solution

- If you can't get to the sink from the source, the answer is 0.
- Let N be the number of vertices reachable both from the source and the sink (in the later case in the reversed graph).

A. Interesting Paths

Solution

- If you can't get to the sink from the source, the answer is 0.
- Let N be the number of vertices reachable both from the source and the sink (in the later case in the reversed graph).
- Let M be the number of such edges.

A. Interesting Paths

Solution

- If you can't get to the sink from the source, the answer is 0.
- Let N be the number of vertices reachable both from the source and the sink (in the later case in the reversed graph).
- Let M be the number of such edges.
- The answer is $M - N + 2$.

A. Interesting Paths

Why?

- If there is an unvisited edge, we can extend the sequence of paths.

Why?

- If there is an unvisited edge, we can extend the sequence of paths.
- If the new path visits v unvisited vertices, it musts visit at least $v + 1$ unvisited edges (at it is enough).

A. Interesting Paths

Why?

- If there is an unvisited edge, we can extend the sequence of paths.
- If the new path visits v unvisited vertices, it musts visit at least $v + 1$ unvisited edges (at it is enough).
- So $N - 2 = \sum_p v$ and $M = \sum_p (v + 1) = p + \sum_p v$, thus $p = M - N + 2$.

Why?

- If there is an unvisited edge, we can extend the sequence of paths.
- If the new path visits v unvisited vertices, it musts visit at least $v+1$ unvisited edges (at it is enough).
- So $N-2=\sum_p v$ and $M=\sum_p (v+1) = p + \sum_p v$, thus $p = M - N + 2$.

Complexity

A. Interesting Paths

Why?

- If there is an unvisited edge, we can extend the sequence of paths.
- If the new path visits v unvisited vertices, it musts visit at least $v + 1$ unvisited edges (at it is enough).
- So $N - 2 = \sum_p v$ and $M = \sum_p (v + 1) = p + \sum_p v$, thus $p = M - N + 2$.

Complexity

$\mathcal{O}(n + m)$

B. Roars III

Fastest solution: **none :(**

B. Roars III

Problem statement

Given is a tree which in some of its vertices contain tokens. For each of its vertices independently we should assume that it is a root and solve the following problem:

B. Roars III

Problem statement

Given is a tree which in some of its vertices contain tokens. For each of its vertices independently we should assume that it is a root and solve the following problem:
In one move we can choose a vertex (different that the root) which contains a token and move this token one edge towards the root. We can do it only if the target vertex doesn't contain a token. We have to calculate the maximum possible number of moves.

B. Roars III

One root

- Let's consider the vertices from the deepest ones.

B. Roars III

One root

- Let's consider the vertices from the deepest ones.
- If the vertex doesn't contain a token, but there is at least one token somewhere in its subtree, it's optimal to move all the tokens on the path to the deepest one one vertex towards the root.

B. Roars III

One root

- Let's consider the vertices from the deepest ones.
- If the vertex doesn't contain a token, but there is at least one token somewhere in its subtree, it's optimal to move all the tokens on the path to the deepest one one vertex towards the root.

Proof

B. Roars III

One root

- Let's consider the vertices from the deepest ones.
- If the vertex doesn't contain a token, but there is at least one token somewhere in its subtree, it's optimal to move all the tokens on the path to the deepest one one vertex towards the root.

Proof

Exchange argument.

B. Roars III

One root

- Let's consider the vertices from the deepest ones.
- If the vertex doesn't contain a token, but there is at least one token somewhere in its subtree, it's optimal to move all the tokens on the path to the deepest one one vertex towards the root.

Proof

Exchange argument.

New move

We can treat such sequence of moves as moving the deepest token directly to the root of the subtree.

B. Roars III

One root

- Let's consider the vertices from the deepest ones.
- If the vertex doesn't contain a token, but there is at least one token somewhere in its subtree, it's optimal to move all the tokens on the path to the deepest one one vertex towards the root.

Proof

Exchange argument.

New move

We can treat such sequence of moves as moving the deepest token directly to the root of the subtree.
We can find the deepest token in the subtree in time $\mathcal{O}(\log(n))$ using segment tree.

Many roots

- Let's simulate the mentioned process for any root.

Many roots

- Let's simulate the mentioned process for any root.
- Note that the operations are reversible (we can choose a token brought to the root and move it back to its position)

Many roots

- Let's simulate the mentioned process for any root.
- Note that the operations are reversible (we can choose a token brought to the root and move it back to its position) – we can rollback the operations.

B. Roars III

Many roots

- Let's simulate the mentioned process for any root.
- Note that the operations are reversible (we can choose a token brought to the root and move it back to its position) – we can rollback the operations.
- Note that the vertex we are calculating the answer for doesn't have to be the root for our data structures

B. Roars III

Many roots

- Let's simulate the mentioned process for any root.
- Note that the operations are reversible (we can choose a token brought to the root and move it back to its position) – we can rollback the operations.
- Note that the vertex we are calculating the answer for doesn't have to be the root for our data structures – we can "reroot" the tree in complexity $\mathcal{O}(\log(n))$ (because the new root is going to be closer to a directed subtree and further from the rest of the vertices).

Many roots

- Let's simulate the mentioned process for any root.
- Note that the operations are reversible (we can choose a token brought to the root and move it back to its position) – we can rollback the operations.
- Note that the vertex we are calculating the answer for doesn't have to be the root for our data structures – we can "reroot" the tree in complexity $\mathcal{O}(\log(n))$ (because the new root is going to be closer to a directed subtree and further from the rest of the vertices).
- What if we calculated the answer for some vertex v and we want to calculate it for its neighbor u?

Many roots

- Let's simulate the mentioned process for any root.
- Note that the operations are reversible (we can choose a token brought to the root and move it back to its position) – we can rollback the operations.
- Note that the vertex we are calculating the answer for doesn't have to be the root for our data structures – we can "reroot" the tree in complexity $\mathcal{O}(\log(n))$ (because the new root is going to be closer to a directed subtree and further from the rest of the vertices).
- What if we calculated the answer for some vertex v and we want to calculate it for its neighbor u? – Only two moves can be different!

B. Roars III

Many roots

- Let's simulate the mentioned process for any root.
- Note that the operations are reversible (we can choose a token brought to the root and move it back to its position) – we can rollback the operations.
- Note that the vertex we are calculating the answer for doesn't have to be the root for our data structures – we can "reroot" the tree in complexity $\mathcal{O}(\log(n))$ (because the new root is going to be closer to a directed subtree and further from the rest of the vertices).
- What if we calculated the answer for some vertex v and we want to calculate it for its neighbor u? – Only two moves can be different!
- Let's rollback these two moves (firstly rollback bringing the token to v and then to u) and then bring tokens to v and to u from correct subtrees.

Complexity

If we maintain a segment tree over the tree we can perform each operation in time $\mathcal{O}(\log(n))$ which gives the final complexity $\mathcal{O}(n\log(n))$.

C. Radars

Fastest solution: **UTokyo: Time Manipulators (0:07)**

C. Radars

Problem statement

Given is a square board $n \times n$. For each of its cells we know the cost of building a radar in it which will cover a square with side n centered in this cell. What is the minimal cost to cover the whole board?

C. Radars

First easy observation

We need to cover each corner of the board.

C. Radars

First easy observation

We need to cover each corner of the board.

Second easy observation

If we covered all corners, we surely covered the whole board.

C. Radars

First easy observation

We need to cover each corner of the board.

Second easy observation

If we covered all corners, we surely covered the whole board.

Conclusion

We can focus only on covering the corners.

Algorithm

We maintain the cost to cover each mask of the corners

Algorithm

We maintain the cost to cover each mask of the corners – $DP[16]$ will be helpful.

Algorithm

We maintain the cost to cover each mask of the corners – $DP[16]$ will be helpful.
To consider a cell with a mask of corners m and cost x, we need to relax the dynamic programming

Algorithm

We maintain the cost to cover each mask of the corners – $DP[16]$ will be helpful.
To consider a cell with a mask of corners m and cost x, we need to relax the dynamic programming – for each i we execute $DP[i|m] = \min(DP[i|m], DP[i] + x)$.

C. Radars

Algorithm

We maintain the cost to cover each mask of the corners – $DP[16]$ will be helpful.
To consider a cell with a mask of corners m and cost x, we need to relax the dynamic programming – for each i we execute $DP[i|m] = \min(DP[i|m], DP[i] + x)$.

Complexity

Linear in the size of the board – $\mathcal{O}(n^2)$.

D. Xor Partitions

Fastest solution: **Harbour.Space: P+P+P (0:08)**

D. Xor Partitions

Problem statement

Given is a sequence of integers $a_1, a_2, \ldots, a_n$.

D. Xor Partitions

Problem statement

Given is a sequence of integers $a_1, a_2, \dots, a_n$.
The value of an interval of the sequence is the xor of its elements.

D. Xor Partitions

Problem statement

Given is a sequence of integers $a_1, a_2, \ldots, a_n$.
The value of an interval of the sequence is the xor of its elements.
The value of a partition of the sequence into intervals is the product of the values of each interval.

D. Xor Partitions

Problem statement

Given is a sequence of integers $a_1, a_2, \ldots, a_n$.
The value of an interval of the sequence is the xor of its elements.
The value of a partition of the sequence into intervals is the product of the values of each interval.
Calculate the sum of the values of all the partitions of a.

D. Xor Partitions

Slow solution

- $dp[i]$ - sum of the values of all partitions of $a_1, a_2, \ldots, a_i$

D. Xor Partitions

Slow solution

- $dp[i]$ - sum of the values of all partitions of $a_1, a_2, \ldots, a_i$
- Iterate over the length of the last interval in the partition.

D. Xor Partitions

Slow solution

- $dp[i]$ - sum of the values of all partitions of $a_1, a_2, \ldots, a_i$
- Iterate over the length of the last interval in the partition.
- The sum of the values of such partitons – the value of the last segment multiplied by sum of the values of all partitions of the prefix.

D. Xor Partitions

Slow solution

- $dp[i]$ - sum of the values of all partitions of $a_1, a_2, \ldots, a_i$
- Iterate over the length of the last interval in the partition.
- The sum of the values of such partitons – the value of the last segment multiplied by sum of the values of all partitions of the prefix.
- $$dp[i] = \sum_{j=0}^{i-1} dp[j] \cdot XOR(a_{j+1}, a_{j+2}, \ldots, a_i)$$

D. Xor Partitions

Slow solution

- $dp[i]$ - sum of the values of all partitions of $a_1, a_2, \ldots, a_i$
- Iterate over the length of the last interval in the partition.
- The sum of the values of such partitons – the value of the last segment multiplied by sum of the values of all partitions of the prefix.
- $$dp[i] = \sum_{j=0}^{i-1} dp[j] \cdot XOR(a_{j+1}, a_{j+2}, \ldots, a_i)$$

Complexity $\mathcal{O}(n^2)$ – too slow.

D. Xor Partitions

Better idea

- We can use distributive property of multiplication over addition.

D. Xor Partitions

Better idea

- We can use distributive property of multiplication over addition.
- Consider each bit in the last interval separately.

D. Xor Partitions

Better idea

- We can use distributive property of multiplication over addition.
- Consider each bit in the last interval separately.
- $$dp[i] = \sum_{b} \sum_{j=0}^{i-1} dp[j] \cdot 2^b \cdot [\text{b is set in } XOR(a_{j+1}, a_{j+2}, \ldots, a_i)]$$

D. Xor Partitions

Better idea

- We can use distributive property of multiplication over addition.
- Consider each bit in the last interval separately.
- $$dp[i] = \sum_{b} \sum_{j=0}^{i-1} dp[j] \cdot 2^b \cdot [\text{b is set in } XOR(a_{j+1}, a_{j+2}, \ldots, a_i)]$$
- Set $pref_i = XOR(a_1, a_2, \ldots, a_i)$

D. Xor Partitions

Better idea

- We can use distributive property of multiplication over addition.
- Consider each bit in the last interval separately.
- $$dp[i] = \sum_{b} \sum_{j=0}^{i-1} dp[j] \cdot 2^b \cdot [\text{b is set in } XOR(a_{j+1}, a_{j+2}, \ldots, a_i)]$$
- Set $pref_i = XOR(a_1, a_2, \ldots, a_i)$
- $$dp[i] = \sum_{b} \sum_{j=0}^{i-1} dp[j] \cdot 2^b \cdot [\text{state of b is different in } pref_i \text{ and in } pref_j]$$

D. Xor Partitions

Algorithm

We can calculate $DP_2[i][b][2]$ – the sum of $DP[j]$ with $j \leq i$ such that the bit b is set or not in $pref_i$. It's easy to update it and calculate DP with it.

D. Xor Partitions

Algorithm

We can calculate $DP_2[i][b][2]$ – the sum of $DP[j]$ with $j \leq i$ such that the bit b is set or not in $pref_i$. It's easy to update it and calculate DP with it.

Memory optimization

We can only remember the last layer of DP_2.

D. Xor Partitions

Algorithm

We can calculate $DP_2[i][b][2]$ – the sum of $DP[j]$ with $j \leq i$ such that the bit b is set or not in $pref_i$. It's easy to update it and calculate DP with it.

Memory optimization

We can only remember the last layer of DP_2.

Complexity

$\mathcal{O}(n \cdot \log(max(a_i)))$

E. Pattern Search II

Fastest solution: **UTokyo: Time Manipulators (1:21)**

E. Pattern Search II

Problem statement

Given is a string t over binary alphabet. We have to choose an equal to it subsequence of the infinite Fibonacci word, so that the distance between the first and the last chosen position is minimal.

Infinite word

Each 3 consecutive characters in Fibonacci word contain both letters, so the result won't exceed $3n$.

E. Pattern Search II

Infinite word

Each 3 consecutive characters in Fibonacci word contain both letters, so the result won't exceed $3n$.

If we fit in S_k, then we are either in its left part (S_{k-1}) or in its right part (S_{k-2}) or in both of them.

Infinite word

Each 3 consecutive characters in Fibonacci word contain both letters, so the result won't exceed $3n$.

If we fit in S_k, then we are either in its left part (S_{k-1}) or in its right part (S_{k-2}) or in both of them.

We can rewrite the left part as $S_{k-2}S_{k-3}$ and the right part as $S_{k-4}S_{k-5}S_{k-4}$.

Infinite word

Each 3 consecutive characters in Fibonacci word contain both letters, so the result won't exceed $3n$.

If we fit in S_k, then we are either in its left part (S_{k-1}) or in its right part (S_{k-2}) or in both of them.

We can rewrite the left part as $S_{k-2}S_{k-3}$ and the right part as $S_{k-4}S_{k-5}S_{k-4}$. If $|S_{k-4}| \geq 3n$, then surely S_{k-3} is enough in the left part and S_{k-4} is enough in the right part, so we fit S_{k-2}.

E. Pattern Search II

Infinite word

Each 3 consecutive characters in Fibonacci word contain both letters, so the result won't exceed $3n$.

If we fit in S_k, then we are either in its left part (S_{k-1}) or in its right part (S_{k-2}) or in both of them.

We can rewrite the left part as $S_{k-2}S_{k-3}$ and the right part as $S_{k-4}S_{k-5}S_{k-4}$. If $|S_{k-4}| \geq 3n$, then surely S_{k-3} is enough in the left part and S_{k-4} is enough in the right part, so we fit S_{k-2}.

We fit in S_k and $|S_{k-4}| \geq 3n \rightarrow$ we fit in S_{k-1}.

E. Pattern Search II

Infinite word

Each 3 consecutive characters in Fibonacci word contain both letters, so the result won't exceed $3n$.
If we fit in S_k, then we are either in its left part (S_{k-1}) or in its right part (S_{k-2}) or in both of them.
We can rewrite the left part as $S_{k-2}S_{k-3}$ and the right part as $S_{k-4}S_{k-5}S_{k-4}$. If $|S_{k-4}| \geq 3n$, then surely S_{k-3} is enough in the left part and S_{k-4} is enough in the right part, so we fit S_{k-2}.
We fit in S_k and $|S_{k-4}| \geq 3n \rightarrow$ we fit in S_{k-1}.

~~In~~finite word

We don't have to look for the optimal subseqnece too far.

E. Pattern Search II

Slow solution

Iterate over the position in S at which the subsequence should start.

E. Pattern Search II

Slow solution

Iterate over the position in S at which the subsequence should start. While going to the right we can greedily match characters from t and find the earliest possible last position.

E. Pattern Search II

Slow solution

Iterate over the position in S at which the subsequence should start. While going to the right we can greedily match characters from t and find the earliest possible last position.

Speed up

We need to be able to answer the following queries: if we'd want to match to S_k the characters of t starting from the i-th one, how many of them would we match?

E. Pattern Search II

Slow solution

Iterate over the position in S at which the subsequence should start. While going to the right we can greedily match characters from t and find the earliest possible last position.

Speed up

We need to be able to answer the following queries: if we'd want to match to S_k the characters of t starting from the i-th one, how many of them would we match?

This will help us quickly jump over the fragments of S.

E. Pattern Search II

Slow solution

Iterate over the position in S at which the subsequence should start. While going to the right we can greedily match characters from t and find the earliest possible last position.

Speed up

We need to be able to answer the following queries: if we'd want to match to S_k the characters of t starting from the i-th one, how many of them would we match?
This will help us quickly jump over the fragments of S.
Answers for all such queries can be easily calculated

E. Pattern Search II

Slow solution

Iterate over the position in S at which the subsequence should start. While going to the right we can greedily match characters from t and find the earliest possible last position.

Speed up

We need to be able to answer the following queries: if we'd want to match to S_k the characters of t starting from the i-th one, how many of them would we match?
This will help us quickly jump over the fragments of S.
Answers for all such queries can be easily calculated – if we denote the answer for the above question by $DP[i][k]$, then
$DP[i][k] = DP[i][k-1] + DP[i + DP[i][k-1]][k-2]$ holds.

E. Pattern Search II

Fast solution

We can calculate each cell of the mentioned array in constant time, so we can calculate all of them in time $\mathcal{O}(n \cdot \log(n))$.

Fast solution

We can calculate each cell of the mentioned array in constant time, so we can calculate all of them in time $\mathcal{O}(n \cdot \log(n))$.
Using it we can for each possible starting positon find the earliest possible last position in time $\mathcal{O}(\log(n))$.

E. Pattern Search II

Fast solution

We can calculate each cell of the mentioned array in constant time, so we can calculate all of them in time $\mathcal{O}(n \cdot \log(n))$.
Using it we can for each possible starting positon find the earliest possible last position in time $\mathcal{O}(\log(n))$.

Complexity

Dynamic programming and looking for all the subsequences take time $\mathcal{O}(n \cdot \log(n))$ each.

F. Waterfall Matrix

Fastest solution: **Add Train Team (1:41)**

F. Waterfall Matrix

Problem statement

We want to create a matrix $n \times n$ in which the values in all columns and rows are nonincreasing. For some subset of its cell we are told what should be in them. For each of these cells the penalty is the absolute difference between the required value and the value in our matrix. We have to minimize the sum of penalties.

F. Waterfall Matrix

Preparation

We can move given cells without changing the answer, so that all of them are in different rows and columns.

F. Waterfall Matrix

What is a difference?

For any integer x we can imagine a "barrier" on the number line at position $x + \frac{1}{2}$.

What is a difference?

For any integer x we can imagine a "barrier" on the number line at position $x + \frac{1}{2}$.
$|a - b|$ is the number of barriers between a and b.

F. Waterfall Matrix

What is a difference?

For any integer x we can imagine a "barrier" on the number line at position $x + \frac{1}{2}$.
$|a - b|$ is the number of barriers between a and b.

One barrier

We could find an optimal matrix if only one barrier would count – the one on position $x + \frac{1}{2}$.

F. Waterfall Matrix

What is a difference?

For any integer x we can imagine a "barrier" on the number line at position $x + \frac{1}{2}$.
$|a - b|$ is the number of barriers between a and b.

One barrier

We could find an optimal matrix if only one barrier would count – the one on position $x + \frac{1}{2}$. We can change numbers $\leq x$ into zeroes and $> x$ into ones.

F. Waterfall Matrix

What is a difference?

For any integer x we can imagine a "barrier" on the number line at position $x + \frac{1}{2}$.
$|a - b|$ is the number of barriers between a and b.

One barrier

We could find an optimal matrix if only one barrier would count – the one on position $x + \frac{1}{2}$. We can change numbers $\leq x$ into zeroes and $> x$ into ones.
We are looking for a "border" which goes from the top-right corner to the bottom-left corner of the matrix

F. Waterfall Matrix

What is a difference?

For any integer x we can imagine a "barrier" on the number line at position $x + \frac{1}{2}$.
$|a - b|$ is the number of barriers between a and b.

One barrier

We could find an optimal matrix if only one barrier would count – the one on position $x + \frac{1}{2}$. We can change numbers $\leq x$ into zeroes and $> x$ into ones.
We are looking for a "border" which goes from the top-right corner to the bottom-left corner of the matrix – for each of the cells we know at which side of the borter it wants to be

F. Waterfall Matrix

What is a difference?

For any integer x we can imagine a "barrier" on the number line at position $x + \frac{1}{2}$.
$|a - b|$ is the number of barriers between a and b.

One barrier

We could find an optimal matrix if only one barrier would count – the one on position $x + \frac{1}{2}$. We can change numbers $\leq x$ into zeroes and $> x$ into ones.
We are looking for a "border" which goes from the top-right corner to the bottom-left corner of the matrix – for each of the cells we know at which side of the borter it wants to be – we pay an unit penalty for each cell that is at the wrong side.

F. Waterfall Matrix

Sweepline

- We sweep the matrix from top to bottom adding rows one by one.

F. Waterfall Matrix

Sweepline

- We sweep the matrix from top to bottom adding rows one by one.
- For each edge between columns we want to remember the penalty if the border goes there

F. Waterfall Matrix

Sweepline

- We sweep the matrix from top to bottom adding rows one by one.
- For each edge between columns we want to remember the penalty if the border goes there – these values are nondecreasing.

Sweepline

- We sweep the matrix from top to bottom adding rows one by one.
- For each edge between columns we want to remember the penalty if the border goes there – these values are nondecreasing. If they aren't and some edge has lower penalty that an edge on its left, we decrease the penalty for the edge on the left.

F. Waterfall Matrix

Sweepline

- We sweep the matrix from top to bottom adding rows one by one.
- For each edge between columns we want to remember the penalty if the border goes there – these values are nondecreasing. If they aren't and some edge has lower penalty that an edge on its left, we decrease the penalty for the edge on the left.
- When passing by an important cell we should either increase a suffix by 1 or increase a prefix by 1.

F. Waterfall Matrix

Sweepline

- We sweep the matrix from top to bottom adding rows one by one.
- For each edge between columns we want to remember the penalty if the border goes there – these values are nondecreasing. If they aren't and some edge has lower penalty that an edge on its left, we decrease the penalty for the edge on the left.
- When passing by an important cell we should either increase a suffix by 1 or increase a prefix by 1. In the later case we might have to decrease some interval by 1 to keep the penalties nondecreasing.

F. Waterfall Matrix

Sweepline

- We sweep the matrix from top to bottom adding rows one by one.
- For each edge between columns we want to remember the penalty if the border goes there – these values are nondecreasing. If they aren't and some edge has lower penalty that an edge on its left, we decrease the penalty for the edge on the left.
- When passing by an important cell we should either increase a suffix by 1 or increase a prefix by 1. In the later case we might have to decrease some interval by 1 to keep the penalties nondecreasing.
- We can use multiset and store the places in which the result increases.

F. Waterfall Matrix

Sweepline

- We sweep the matrix from top to bottom adding rows one by one.
- For each edge between columns we want to remember the penalty if the border goes there – these values are nondecreasing. If they aren't and some edge has lower penalty that an edge on its left, we decrease the penalty for the edge on the left.
- When passing by an important cell we should either increase a suffix by 1 or increase a prefix by 1. In the later case we might have to decrease some interval by 1 to keep the penalties nondecreasing.
- We can use multiset and store the places in which the result increases.

Border recovery

We can reverse this process and recover the optimal border

F. Waterfall Matrix

Sweepline

- We sweep the matrix from top to bottom adding rows one by one.
- For each edge between columns we want to remember the penalty if the border goes there – these values are nondecreasing. If they aren't and some edge has lower penalty that an edge on its left, we decrease the penalty for the edge on the left.
- When passing by an important cell we should either increase a suffix by 1 or increase a prefix by 1. In the later case we might have to decrease some interval by 1 to keep the penalties nondecreasing.
- We can use multiset and store the places in which the result increases.

Border recovery

We can reverse this process and recover the optimal border – this will tell us which cells should be $\leq x$ and which should be $> x$.

F. Waterfall Matrix

Key observation

Optimal border for x will be below and to the right of the optimal border of $x + 1$

F. Waterfall Matrix

Key observation

Optimal border for x will be below and to the right of the optimal border of $x + 1$ – we can find them independently and sum the results!

F. Waterfall Matrix

Key observation

Optimal border for x will be below and to the right of the optimal border of $x + 1$ – we can find them independently and sum the results!
It would give us correct algorithm working in time $\mathcal{O}(n^2 \log(n))$.

F. Waterfall Matrix

Divide and conquer!

For any x we can check which cells should be greater than x and which shouldn't.

Divide and conquer!

For any x we can check which cells should be greater than x and which shouldn't.
We can run the algorithm for the "middle" value of x and split recursively on two sides passing each cell only to one side.

F. Waterfall Matrix

Divide and conquer!

For any x we can check which cells should be greater than x and which shouldn't.
We can run the algorithm for the "middle" value of x and split recursively on two sides passing each cell only to one side.
There will be $\mathcal{O}(\log(n))$ layers of the recurrence and each of them will contain n cells at total – it will take $\mathcal{O}(n \log(n))$ to consider them all.

F. Waterfall Matrix

Divide and conquer!

For any x we can check which cells should be greater than x and which shouldn't.
We can run the algorithm for the "middle" value of x and split recursively on two sides passing each cell only to one side.
There will be $\mathcal{O}(\log(n))$ layers of the recurrence and each of them will contain n cells at total – it will take $\mathcal{O}(n \log(n))$ to consider them all.

Complexity

If we use a data structure (such as multiset) which works in $\mathcal{O}(\log(n))$ per operation we will end up with complexity $\mathcal{O}(n \log^2(n))$.

G. Puzzle II

Fastest solution: **Add Train Team (2:04)**

G. Puzzle II

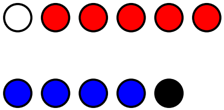
Problem statement

Given two binary sequences of length n and a number k. In one move we can choose a cyclic segment of length k from the first sequence and a cyclic segment of the same length from the second sequence and swap them. We need to make both sequences monochromatic in at most n moves.

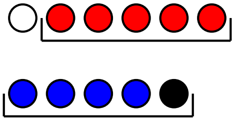
What to do?

What organized moves can we do?

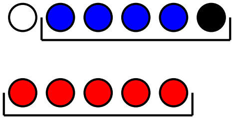
G. Puzzle II



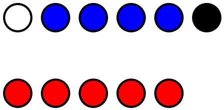
G. Puzzle II



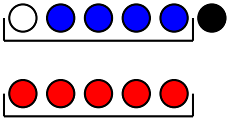
G. Puzzle II



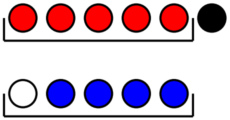
G. Puzzle II



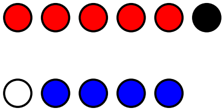
G. Puzzle II



G. Puzzle II



G. Puzzle II



What to do?

In two moves we can move an element from the first sequence to the second and one from the second to the first.

G. Puzzle II

What to do?

In two moves we can move an element from the first sequence to the second and one from the second to the first. As we can choose which sequence will be white we can do at most $\frac{n}{2}$ such operations, resulting in $\leq n$ moves.

G. Puzzle II

Should we start?

An experienced eye should spot a solution which uses some BST and would result in a $\mathcal{O}(n \cdot \log(n))$ complexity...

G. Puzzle II

Should we start?

An experienced eye should spot a solution which uses some BST and would result in a $\mathcal{O}(n \cdot \log(n))$ complexity...However it might be a good idea to look for something simpler and faster.

Clever way

Let's set a sliding window of size $k + 1$ on first elements of the first sequence.

Clever way

Let's set a sliding window of size $k+1$ on first elements of the first sequence.
Let's set a sliding window of size k on the last elements of the second sequence.

G. Puzzle II

Clever way

Let's set a sliding window of size $k + 1$ on first elements of the first sequence.
Let's set a sliding window of size k on the last elements of the second sequence.
We'll store both windows on deques – described operation can be done in constant time.

G. Puzzle II

Clever way

Let's set a sliding window of size $k + 1$ on first elements of the first sequence.
Let's set a sliding window of size k on the last elements of the second sequence.
We'll store both windows on deques – described operation can be done in constant time.
We can move the windows to the right and to the left by one also in constant time.

G. Puzzle II

Clever way

Let's set a sliding window of size $k+1$ on first elements of the first sequence.
Let's set a sliding window of size k on the last elements of the second sequence.
We'll store both windows on deques – described operation can be done in constant time.
We can move the windows to the right and to the left by one also in constant time.
As long as we are not happy with the first element of the first window we can slide it to the right – and the second window to the left.

G. Puzzle II

Clever way

Let's set a sliding window of size $k + 1$ on first elements of the first sequence.
Let's set a sliding window of size k on the last elements of the second sequence.
We'll store both windows on deques – described operation can be done in constant time.
We can move the windows to the right and to the left by one also in constant time.
As long as we are not happy with the first element of the first window we can slide it to the right – and the second window to the left.
We'll move both windows at most $\mathcal{O}(n)$ times.

G. Puzzle II

Complexity

$\mathcal{O}(n \cdot \log(n))$ with a BST of your choice or $\mathcal{O}(n)$ with some tricks.

H. Weather Forecast

Fastest solution: **LNU: LNU Stallions (0:17)**

H. Weather Forecast

Problem statement

We are given a sequence of integers and a number k. We need to find a partition of this sequence into k intervals which maximizes the minimum mean over all intervals.

Key observation

If we can have all means $\geq x$, we surely can have all means $\geq y$ if $x \geq y$.

H. Weather Forecast

Key observation

If we can have all means $\geq x$, we surely can have all means $\geq y$ if $x \geq y$.

Conclusion

Binary search to find the answer.

H. Weather Forecast

Can we have all means $\geq x$?

H. Weather Forecast

Can we have all means $\geq x$?

- $\frac{a_1+\ldots+a_\ell}{\ell} \geq x \iff a_1 + \ldots + a_\ell \geq x \cdot \ell \iff (a_1 - x) + \ldots + (a_\ell - x) \geq 0$

Can we have all means $\geq x$?

- $\frac{a_1+\ldots+a_\ell}{\ell} \geq x \iff a_1 + \ldots + a_\ell \geq x \cdot \ell \iff (a_1 - x) + \ldots + (a_\ell - x) \geq 0$
- Subtract x from each number in the sequence.

H. Weather Forecast

Can we have all means $\geq x$?

- $\frac{a_1+\ldots+a_\ell}{\ell} \geq x \iff a_1 + \ldots + a_\ell \geq x \cdot \ell \iff (a_1 - x) + \ldots + (a_\ell - x) \geq 0$
- Subtract x from each number in the sequence.
- Calculate prefix sums and set sequence $0 = e_0, e_1, e_2, e_3, \ldots, e_k = n$ to be the ends of the intervals in the partition.

H. Weather Forecast

Can we have all means $\geq x$?

- $\frac{a_1+\ldots+a_\ell}{\ell} \geq x \iff a_1 + \ldots + a_\ell \geq x \cdot \ell \iff (a_1 - x) + \ldots + (a_\ell - x) \geq 0$
- Subtract x from each number in the sequence.
- Calculate prefix sums and set sequence $0 = e_0, e_1, e_2, e_3, \ldots, e_k = n$ to be the ends of the intervals in the partition.
- Prefix sums in these points must form a nondecreasing sequence. We have to choose at least $k+1$ of them (including 0 and n).

H. Weather Forecast

Can we have all means $\geq x$?

- $\frac{a_1+\ldots+a_\ell}{\ell} \geq x \iff a_1 + \ldots + a_\ell \geq x \cdot \ell \iff (a_1 - x) + \ldots + (a_\ell - x) \geq 0$
- Subtract x from each number in the sequence.
- Calculate prefix sums and set sequence $0 = e_0, e_1, e_2, e_3, \ldots, e_k = n$ to be the ends of the intervals in the partition.
- Prefix sums in these points must form a nondecreasing sequence. We have to choose at least $k+1$ of them (including 0 and n).

Solution

Longest increasing sequence.

H. Weather Forecast

Can we have all means $\geq x$?

- $\frac{a_1+\ldots+a_\ell}{\ell} \geq x \iff a_1 + \ldots + a_\ell \geq x \cdot \ell \iff (a_1 - x) + \ldots + (a_\ell - x) \geq 0$
- Subtract x from each number in the sequence.
- Calculate prefix sums and set sequence $0 = e_0, e_1, e_2, e_3, \ldots, e_k = n$ to be the ends of the intervals in the partition.
- Prefix sums in these points must form a nondecreasing sequence. We have to choose at least $k+1$ of them (including 0 and n).

Solution

Longest increasing sequence.

Complexity

$\mathcal{O}(n \cdot \log(n) \cdot \log(precision))$

I. Mercenaries

Fastest solution: **UTokyo: Time Manipulators (4:10)**

I. Mercenaries

Problem statement

We are given a straight road on which we can move only to the right. On the road there are n cities and the mercenary living in the i-th city is parametrized by pair (s_i, m_i). Between each two neighboring cities there is a shop which allows to buy one item in it and each item will add some values to both statistics of the mercenary. When mercenary moves from one city to another (he can move only to the right) he can buy one item in each shop and their bonuses will accumulate. We have to consider monster attack scenarios – a monster can attack some city and this monster is parametrized by three values – A, B and C. A mercenary can defeat the monster if he can get to it having statistics (S, M) so that $A \cdot S + B \cdot M \geq C$. We have to find the rightmost mercenary which could defeat each monster.

Is this geometry?

Let's interpret mercenaries and items as vectors in the first quarter of a coordinate plane

Is this geometry?

Let's interpret mercenaries and items as vectors in the first quarter of a coordinate plane – defeating the monster means being in a given halfplane.

I. Mercenaries

Is this geometry?

Let's interpret mercenaries and items as vectors in the first quarter of a coordinate plane – defeating the monster means being in a given halfplane.

Is this convex hulls?

To check if a monster can be defeated we need to consider only upper-right convex hull of the statistics of the mercenaries that can get to this monster.

Organized approach

Let's build a segment tree on the sequence of cities.

Organized approach

Let's build a segment tree on the sequence of cities.
In each base segment let's calculate a convex hull of possible bonuses that we can get if we pass through this segment.

Organized approach

Let's build a segment tree on the sequence of cities.

In each base segment let's calculate a convex hull of possible bonuses that we can get if we pass through this segment.

A convex hull for a segment is a Minkowski sum of convex hulls for its two subsegments

Organized approach

Let's build a segment tree on the sequence of cities.
In each base segment let's calculate a convex hull of possible bonuses that we can get if we pass through this segment.
A convex hull for a segment is a Minkowski sum of convex hulls for its two subsegments – we can merge them in linear time.

Organized approach part two

For each base segment let's also calculate a convex hull of possible statistics of mercenaries which start in this interval in the moment they leave it.

Organized approach part two

For each base segment let's also calculate a convex hull of possible statistics of mercenaries which start in this interval in the moment they leave it.
Such a mercenary can start in the right subsegment

Organized approach part two

For each base segment let's also calculate a convex hull of possible statistics of mercenaries which start in this interval in the moment they leave it.
Such a mercenary can start in the right subsegment or in the left one and strengthen himself with items from the right interval

Organized approach part two

For each base segment let's also calculate a convex hull of possible statistics of mercenaries which start in this interval in the moment they leave it.
Such a mercenary can start in the right subsegment or in the left one and strengthen himself with items from the right interval (again Minkowski sum).

Organized approach part two

For each base segment let's also calculate a convex hull of possible statistics of mercenaries which start in this interval in the moment they leave it.
Such a mercenary can start in the right subsegment or in the left one and strengthen himself with items from the right interval (again Minkowski sum).
Convex hull of a set of points also can be calculated in linear time.

Attack scenario

When a monster attacks let's split the prefix into base segments and consider them from right to left.

Attack scenario

When a monster attacks let's split the prefix into base segments and consider them from right to left.
To check if in the segment there is a mercenary that can defeat it we can use binary search (or ternary search) on the hull.

Attack scenario

When a monster attacks let's split the prefix into base segments and consider them from right to left.
To check if in the segment there is a mercenary that can defeat it we can use binary search (or ternary search) on the hull.
If yes, we go deeper in the tree.

Attack scenario

When a monster attacks let's split the prefix into base segments and consider them from right to left.
To check if in the segment there is a mercenary that can defeat it we can use binary search (or ternary search) on the hull.
If yes, we go deeper in the tree.
If no, we have to consider the items that the mercery can buy.

Attack scenario

When a monster attacks let's split the prefix into base segments and consider them from right to left.
To check if in the segment there is a mercenary that can defeat it we can use binary search (or ternary search) on the hull.
If yes, we go deeper in the tree.
If no, we have to consider the items that the mercery can buy. Their maximum possible impact on the monster (information how much should we decrease the C parameter) also can be found with binary search.

Optimization

Described approach for $\mathcal{O}(\log(n))$ base segments does a binary search

Optimization

Described approach for $\mathcal{O}(\log(n))$ base segments does a binary search – we can be better.

Optimization

Described approach for $\mathcal{O}(\log(n))$ base segments does a binary search – we can be better.
Let's consider the monsters sorted by the angle of their halfplane

Optimization

Described approach for $\mathcal{O}(\log(n))$ base segments does a binary search – we can be better.

Let's consider the monsters sorted by the angle of their halfplane – the optimal point on all convex hulls will move only to one direction

Optimization

Described approach for $\mathcal{O}(\log(n))$ base segments does a binary search – we can be better.
Let's consider the monsters sorted by the angle of their halfplane – the optimal point on all convex hulls will move only to one direction – we'll be able to just remove non-optimal vectors from the convex hulls if we store them as lists.

Optimization

Described approach for $\mathcal{O}(\log(n))$ base segments does a binary search – we can be better.
Let's consider the monsters sorted by the angle of their halfplane – the optimal point on all convex hulls will move only to one direction – we'll be able to just remove non-optimal vectors from the convex hulls if we store them as lists.

Complexity

Mentioned optimization will allow us to multiply the size of the input by only one logarithm – the height of the segment tree and the need of sorting the monsters and items by angle.

Optimization

Described approach for $\mathcal{O}(\log(n))$ base segments does a binary search – we can be better.
Let's consider the monsters sorted by the angle of their halfplane – the optimal point on all convex hulls will move only to one direction – we'll be able to just remove non-optimal vectors from the convex hulls if we store them as lists.

Complexity

Mentioned optimization will allow us to multiply the size of the input by only one logarithm – the height of the segment tree and the need of sorting the monsters and items by angle. We'll end up with comlexity $\mathcal{O}((n + \sum_i r_i + q) \cdot \log(n + \sum_i r_i))$.

J. Polygon II

Fastest solution: **Harbour.Space: P+P+P (3:24)**

J. Polygon II

Problem statement

Random variables $X_1, X_2, \ldots, X_n$, where $X_i = U(0, 2^{a_i})$. Find the probability that we can construct a nondegenerate polygon with sides of lengths X_i.

J. Polygon II

Triangle inequality

Bad if and only if for some i

$$X_i \geq \sum_{j \neq i} X_j.$$

J. Polygon II

Triangle inequality

Bad if and only if for some i

$$X_i \geq \sum_{j \neq i} X_j.$$

Answer

$$1 - \sum_i P(X_i \geq \sum_{j \neq i} X_j)$$

– bad events are disjoint.

Helpful lemma

$$P(X_i \geq \sum_{j \neq i} X_j) = P(2^{a_i} \geq \sum_{j} X_j)$$

J. Polygon II

Helpful lemma

$$P(X_i \geq \sum_{j \neq i} X_j) = P(2^{a_i} \geq \sum_{j} X_j)$$

Proof

$$X_i \sim 2^{a_i} - X_i$$

J. Polygon II

Helpful lemma

$$P(X_i \geq \sum_{j \neq i} X_j) = P(2^{a_i} \geq \sum_{j} X_j)$$

Proof

$$X_i \sim 2^{a_i} - X_i$$

$$P(X_i \geq \sum_{j \neq i} X_j) = P(2^{a_i} - X_i \geq \sum_{j \neq i} X_j) = P(2^{a_i} \geq \sum_{j} X_j)$$

J. Polygon II

Main idea

Let Y_i be a random variable with only two possible values:

$$P(Y_i = 0) = \frac{1}{2}, P(Y_i = 2^i) = \frac{1}{2}$$

J. Polygon II

Main idea

Let Y_i be a random variable with only two possible values:

$$P(Y_i = 0) = \frac{1}{2}, P(Y_i = 2^i) = \frac{1}{2}$$

Bits decomposition

$$X_i = U(0,1) + Y_0 + Y_1 + \ldots + Y_{a_i - 1}$$

Dynamic programming

$DP[i][j]$ – probability, that we carry j bits (of value 2^i) after deciding on all $U(0,1)$, Y_0, Y_1, … Y_{i-1}.

J. Polygon II

Dynamic programming

$DP[i][j]$ – probability, that we carry j bits (of value 2^i) after deciding on all $U(0,1)$, Y_0, Y_1, ... Y_{i-1}.

Transitions

k_i – number of variables of type Y_i

$$DP[i+1][j] = \sum_{l=0}^{k_i} (DP[i][2j-l] + DP[i][2j-l+1]) \frac{\binom{k_i}{l}}{2^{k_i}}$$

J. Polygon II

Dynamic programming

$DP[i][j]$ – probability, that we carry j bits (of value 2^i) after deciding on all $U(0,1)$, Y_0, Y_1, … Y_{i-1}.

Transitions

k_i – number of variables of type Y_i

$$DP[i+1][j] = \sum_{l=0}^{k_i} (DP[i][2j-l] + DP[i][2j-l+1]) \frac{\binom{k_i}{l}}{2^{k_i}}$$

Initialization – only $U(0,1)$

$\sum_{i=0}^{j} DP[0][i] =$ volume of an n-dimentional polyhedron $\sum x_i < j$ and $0 \leq x_i \leq 1$.

J. Polygon II

Dynamic programming

$DP[i][j]$ – probability, that we carry j bits (of value 2^i) after deciding on all $U(0,1)$, Y_0, Y_1, ... Y_{i-1}.

Transitions

k_i – number of variables of type Y_i

$$DP[i+1][j] = \sum_{l=0}^{k_i} (DP[i][2j-l] + DP[i][2j-l+1]) \frac{\binom{k_i}{l}}{2^{k_i}}$$

Initialization – only $U(0,1)$

$\sum_{i=0}^{j} DP[0][i] =$ volume of an n-dimentional polyhedron $\sum x_i < j$ and $0 \leq x_i \leq 1$.
Inclusion-exclusion principle on how many $x_i \leq 1$ are not met.

J. Polygon II

Complexity

$\mathcal{O}(max(a_i) \cdot n^2)$

J. Polygon II

Complexity

$\mathcal{O}(max(a_i) \cdot n^2)$ or $\mathcal{O}(max(a_i) \cdot n \cdot \log(n))$ with FFT.

K. Power Divisions

Fastest solution: **Add Train Team (1:18)**

K. Power Divisions

Problem statement

Given is a sequence $b_1, b_2, \ldots b_n$ of form $2^{a_1}, 2^{a_2}, \ldots, 2^{a_n}$.
An interval $[l, r]$ is good $\iff b_l + b_{l+1} + \ldots + b_r = 2^k$ (for $k \in \mathbb{N}$)
Calculate the number of partitions of the sequence into good intervals (modulo prime number).

Workflow

K. Power Divisions

Workflow

- Find all good intervals

Workflow

- Find all good intervals – divide&conquer.

K. Power Divisions

Workflow

- Find all good intervals – divide&conquer.
- Count all good partitions

K. Power Divisions

Workflow

- Find all good intervals – divide&conquer.
- Count all good partitions – dynamic programming.

Representation of sum S and increasing it by b_i

K. Power Divisions

Representation of sum S and increasing it by b_i

S	101110
b_i	000100
$S + b_i$	110010

Representation of sum S and increasing it by b_i

S	101110
b_i	000100
$S + b_i$	110010

Time – amortized $\mathcal{O}(1)$

Representation of sum S and increasing it by b_i

S	101110
b_i	000100
$S + b_i$	110010

Time – amortized $\mathcal{O}(1)$ (potential – number of set bits).

K. Power Divisions

Constants

P – big prime number, for example $2^{61} - 1$.

$c_0, c_1, \ldots, c_{10^6+20}$ – random coefficients.

K. Power Divisions

Constants

P – big prime number, for example $2^{61} - 1$.

$c_0, c_1, \ldots, c_{10^6+20}$ – random coefficients.

Hash

$S = \sum_i b_i \cdot 2^i$

K. Power Divisions

Constants

P – big prime number, for example $2^{61} - 1$.
$c_0, c_1, \ldots, c_{10^6+20}$ – random coefficients.

Hash

$S = \sum_i b_i \cdot 2^i$
$h(S) = \sum_i b_i \cdot c_i$ modulo P.

K. Power Divisions

Constants

P – big prime number, for example $2^{61} - 1$.
$c_0, c_1, \ldots, c_{10^6+20}$ – random coefficients.

Hash

$S = \sum_i b_i \cdot 2^i$
$h(S) = \sum_i b_i \cdot c_i$ modulo P.
Together with the binary reprezentation we will keep track of the hash of the sum (still amortized constant time).

K. Power Divisions

Constants

P – big prime number, for example $2^{61} - 1$.
$c_0, c_1, \dots, c_{10^6+20}$ – random coefficients.

Hash

$S = \sum_i b_i \cdot 2^i$
$h(S) = \sum_i b_i \cdot c_i$ modulo P.
Together with the binary reprezentation we will keep track of the hash of the sum (still amortized constant time).

Probability of a collision

$S_1 \neq S_2 \Rightarrow P(h(S_1) = h(S_2)) = \frac{1}{P}$

All good intervals – divide&conquer

K. Power Divisions

All good intervals – divide&conquer

- split the interval into L and R

K. Power Divisions

All good intervals – divide&conquer

- split the interval into L and R
- recursion on L

K. Power Divisions

All good intervals – divide&conquer

- split the interval into L and R
- recursion on L
- recursion on R

All good intervals – divide&conquer

- split the interval into L and R
- recursion on L
- recursion on R
- intervals $suf_L + pref_R = 2^k$

K. Power Divisions

Intervals $suf_L + pref_R = 2^k$

K. Power Divisions

Intervals $suf_L + pref_R = 2^k$

WLOG $pref_R \geq suf_L$ (case $pref_R < suf_L$ is analogical).

K. Power Divisions

Intervals $suf_L + pref_R = 2^k$

WLOG $pref_R \geq suf_L$ (case $pref_R < suf_L$ is analogical).

Key observation

For $pref_R$ there is only one possible value of suf_L.

K. Power Divisions

Intervals $suf_L + pref_R = 2^k$

WLOG $pref_R \geq suf_L$ (case $pref_R < suf_L$ is analogical).

Key observation

For $pref_R$ there is only one possible value of suf_L.

$pref_R$	0101110
2^k	1000000
suf_L	0010010

K. Power Divisions

Intervals $suf_L + pref_R = 2^k$

WLOG $pref_R \geq suf_L$ (case $pref_R < suf_L$ is analogical).

Key observation

For $pref_R$ there is only one possible value of suf_L.

$pref_R$	0101110
2^k	1000000
suf_L	0010010

Calculating $h(suf_L)$

a – rightmost set bit in $pref_R$.
b – leftmost set bit in $pref_R$.

K. Power Divisions

Intervals $suf_L + pref_R = 2^k$

WLOG $pref_R \geq suf_L$ (case $pref_R < suf_L$ is analogical).

Key observation

For $pref_R$ there is only one possible value of suf_L.

$pref_R$	0101110
2^k	1000000
suf_L	0010010

Calculating $h(suf_L)$

a – rightmost set bit in $pref_R$.

b – leftmost set bit in $pref_R$.

$$h(suf_L) + h(pref_R) = c_a + \sum_{i=a}^{b} c_i.$$

K. Power Divisions

Intervals $suf_L + pref_R = 2^k$ – algorithm

- Memorize $h(suf_L)$ – (hash)map from $h(suf_L)$ into L.

Intervals $suf_L + pref_R = 2^k$ – algorithm

- Memorize $h(suf_L)$ – (hash)map from $h(suf_L)$ into L.
- Iterate over $pref_R$, keep frack of S, $h(S)$, a and b.

K. Power Divisions

Intervals $suf_L + pref_R = 2^k$ – algorithm

- Memorize $h(suf_L)$ – (hash)map from $h(suf_L)$ into L.
- Iterate over $pref_R$, keep frack of S, $h(S)$, a and b.
- Check if good suf_L exists.

K. Power Divisions

Intervals $suf_L + pref_R = 2^k$ – algorithm

- Memorize $h(suf_L)$ – (hash)map from $h(suf_L)$ into L.
- Iterate over $pref_R$, keep frack of S, $h(S)$, a and b.
- Check if good suf_L exists.

Probability of a collision

$\sum_{suf_L} \sum_{pref_R} P(h(suf_L) = h(2^k - pref_R)) \leq \frac{n^2}{P}$

Complexity

- One level of divide and conquer:

K. Power Divisions

Complexity

- One level of divide and conquer: $\mathcal{O}(n)$ (hashmap) or $\mathcal{O}(n \log n)$ (map).

Complexity

- One level of divide and conquer: $\mathcal{O}(n)$ (hashmap) or $\mathcal{O}(n \log n)$ (map).
- Whole divide and conquer:

K. Power Divisions

Complexity

- One level of divide and conquer: $\mathcal{O}(n)$ (hashmap) or $\mathcal{O}(n \log n)$ (map).
- Whole divide and conquer: $\mathcal{O}(n \log n)$ (hashmap) or $\mathcal{O}(n \log^2 n)$ (map).

K. Power Divisions

Complexity

- One level of divide and conquer: $\mathcal{O}(n)$ (hashmap) or $\mathcal{O}(n \log n)$ (map).
- Whole divide and conquer: $\mathcal{O}(n \log n)$ (hashmap) or $\mathcal{O}(n \log^2 n)$ (map).
- Dynamic programming:

K. Power Divisions

Complexity

- One level of divide and conquer: $\mathcal{O}(n)$ (hashmap) or $\mathcal{O}(n \log n)$ (map).
- Whole divide and conquer: $\mathcal{O}(n \log n)$ (hashmap) or $\mathcal{O}(n \log^2 n)$ (map).
- Dynamic programming: $\mathcal{O}(\text{number of good intervals}) = \mathcal{O}(n \log n)$.

L. Chords

Fastest solution: **Harbour.Space: P+P+P (2:36)**

L. Chords

Problem statement

$2n$ points on a circle were randomly paired creating n chords. We need to find the biggest subset of chords such that no two of them intersect.

Simpler look

We can cut the circle in any position

Simpler look

We can cut the circle in any position – now any two chosen interval has to be either disjoint or one must be contained in the other.

Simpler look

We can cut the circle in any position – now any two chosen interval has to be either disjoint or one must be contained in the other.

Deterministic solution

L. Chords

Simpler look

We can cut the circle in any position – now any two chosen interval has to be either disjoint or one must be contained in the other.

Deterministic solution

$DP[\ell][r]$ – maximum number of nonintersecting chords if we choose only from interval $[\ell, r]$.

Simpler look

We can cut the circle in any position – now any two chosen interval has to be either disjoint or one must be contained in the other.

Deterministic solution

$DP[\ell][r]$ – maximum number of nonintersecting chords if we choose only from interval $[\ell, r]$.
To calculate $DP[\ell][r]$ we surely have to consider $DP[\ell][r-1]$.

L. Chords

Simpler look

We can cut the circle in any position – now any two chosen interval has to be either disjoint or one must be contained in the other.

Deterministic solution

$DP[\ell][r]$ – maximum number of nonintersecting chords if we choose only from interval $[\ell, r]$.
To calculate $DP[\ell][r]$ we surely have to consider $DP[\ell][r-1]$.
If in point r some chord ends, its begining is on the point $left_r$ and $left_r \geq \ell$ holds, we need to consider $DP[\ell][left_r - 1] + 1 + dp[left_r + 1][r - 1]$.

L. Chords

Simpler look

We can cut the circle in any position – now any two chosen interval has to be either disjoint or one must be contained in the other.

Deterministic solution

$DP[\ell][r]$ – maximum number of nonintersecting chords if we choose only from interval $[\ell, r]$.
To calculate $DP[\ell][r]$ we surely have to consider $DP[\ell][r-1]$.
If in point r some chord ends, its begining is on the point $left_r$ and $left_r \geq \ell$ holds, we need to consider $DP[\ell][left_r - 1] + 1 + dp[left_r + 1][r - 1]$.
Described dynamic programming calculates correct answer in time $\mathcal{O}(n^2)$ and returns it in $DP[1][2n]$.

L. Chords

Why randomness?

Chords very often intersect each other in a chaotic way

Why randomness?

Chords very often intersect each other in a chaotic way – it's difficult to pick a big subset so that no two of them will intersect

Why randomness?

Chords very often intersect each other in a chaotic way – it's difficult to pick a big subset so that no two of them will intersect – the answer will be small!

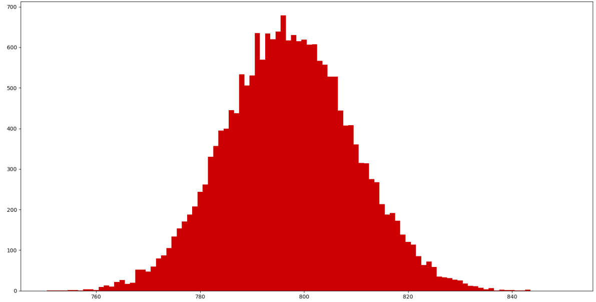
Why randomness?

Chords very often intersect each other in a chaotic way – it's difficult to pick a big subset so that no two of them will intersect – the answer will be small!
It's best to check empirically – for $n = 10^5$ the answer won't be very far from 800.

L. Chords

Why randomness?

Chords very often intersect each other in a chaotic way – it's difficult to pick a big subset so that no two of them will intersect – the answer will be small!
It's best to check empirically – for $n = 10^5$ the answer won't be very far from 800.



L. Chords

How to use it?

For fixed r and any ℓ $DP[\ell][r]$ won't be very big.

How to use it?

For fixed r and any ℓ $DP[\ell][r]$ won't be very big. We also know, that $DP[\ell-1][r] \geq DP[\ell][r]$.

How to use it?

For fixed r and any ℓ $DP[\ell][r]$ won't be very big. We also know, that $DP[\ell-1][r] \geq DP[\ell][r]$.

For each r we can only memorize for which ℓs the answer increases!

L. Chords

How to use it?

For fixed r and any ℓ $DP[\ell][r]$ won't be very big. We also know, that $DP[\ell-1][r] \geq DP[\ell][r]$.

For each r we can only memorize for which ℓs the answer increases!

Compressing the square array this way we decrease the size of stored information to $\mathcal{O}(n \cdot ans)$

L. Chords

How to use it?

For fixed r and any ℓ $DP[\ell][r]$ won't be very big. We also know, that $DP[\ell-1][r] \geq DP[\ell][r]$.

For each r we can only memorize for which ℓs the answer increases!

Compressing the square array this way we decrease the size of stored information to $\mathcal{O}(n \cdot ans)$ In time $\mathcal{O}(ans)$ we can also calculate everything for fixed r.

L. Chords

How to use it?

For fixed r and any ℓ $DP[\ell][r]$ won't be very big. We also know, that $DP[\ell - 1][r] \geq DP[\ell][r]$.

For each r we can only memorize for which ℓs the answer increases!

Compressing the square array this way we decrease the size of stored information to $\mathcal{O}(n \cdot ans)$ In time $\mathcal{O}(ans)$ we can also calculate everything for fixed r.

Complexity

We end up with time and memory complexity $\mathcal{O}(n \cdot ans)$.

M. Balance of Permutation

Fastest solution: **Harbour.Space: P+P+P (1:31)**

M. Balance of Permutation

Problem statement

A balance of a permutation p is defined as the sum of $|p_i - i|$. We have to find k-th lexicographically smallest n-element permutation with balance equal to b.

M. Balance of Permutation

Simple look

A permutation is as assignment of values to positions – let's imagine n red numbers (the positions) and n blue numbers (the values) and sort them as numbers.

M. Balance of Permutation

Simple look

A permutation is as assignment of values to positions – let's imagine n red numbers (the positions) and n blue numbers (the values) and sort them as numbers.

1 1 2 2 3 3 4 4 5 5

M. Balance of Permutation

Simple look

A permutation is as assignment of values to positions – let's imagine n red numbers (the positions) and n blue numbers (the values) and sort them as numbers.

1 1 2 2 3 3 4 4 5 5

Dynamic programming

Each pair will be created from the perspective of the right number.

M. Balance of Permutation

Simple look

A permutation is as assignment of values to positions – let's imagine n red numbers (the positions) and n blue numbers (the values) and sort them as numbers.

1 1 2 2 3 3 4 4 5 5

Dynamic programming

Each pair will be created from the perspective of the right number. Let $DP[i][j]$ be the number of ways to create j pairs on the prefix of size i (in the above sequence).

M. Balance of Permutation

Simple look

A permutation is as assignment of values to positions – let's imagine n red numbers (the positions) and n blue numbers (the values) and sort them as numbers.

1 1 2 2 3 3 4 4 5 5

Dynamic programming

Each pair will be created from the perspective of the right number. Let $DP[i][j]$ be the number of ways to create j pairs on the prefix of size i (in the above sequence). Moving from prefix i to $i+1$ we can match the number with one of the previous unpaired numbers – we know the number of ways to do so.

M. Balance of Permutation

Simple look

A permutation is as assignment of values to positions – let's imagine n red numbers (the positions) and n blue numbers (the values) and sort them as numbers.

1 1 2 2 3 3 4 4 5 5

Dynamic programming

Each pair will be created from the perspective of the right number. Let $DP[i][j]$ be the number of ways to create j pairs on the prefix of size i (in the above sequence). Moving from prefix i to $i+1$ we can match the number with one of the previous unpaired numbers – we know the number of ways to do so. We can also not match it and assume that we want to match it with something on its right.

What about the balance?

Such dynamic programming will simply count all the permutations

M. Balance of Permutation

What about the balance?

Such dynamic programming will simply count all the permutations – we can extend the state.

M. Balance of Permutation

What about the balance?

Such dynamic programming will simply count all the permutations – we can extend the state.

Let $DP[i][j][\ell]$ be the number of ways to create j pairs on the prefix of size i, so that the sum of numbers that are right in their pairs is equal to ℓ.

What about the balance?

Such dynamic programming will simply count all the permutations – we can extend the state.

Let $DP[i][j][\ell]$ be the number of ways to create j pairs on the prefix of size i, so that the sum of numbers that are right in their pairs is equal to ℓ.

Knowing the sum of right numbers, we also know the sum of left numbers, so we know the balance.

M. Balance of Permutation

What about the balance?

Such dynamic programming will simply count all the permutations – we can extend the state.
Let $DP[i][j][\ell]$ be the number of ways to create j pairs on the prefix of size i, so that the sum of numbers that are right in their pairs is equal to ℓ.
Knowing the sum of right numbers, we also know the sum of left numbers, so we know the balance.
Such dynamic programming has $\mathcal{O}(n^4)$ states and we calculate each of them in constant time.

M. Balance of Permutation

What about k-th lexicographically smallest one?

In subsequent positions we will try to manually insert subsequent values and count in how many ways we can finish the permutation so it has the required balance.

M. Balance of Permutation

What about k-th lexicographically smallest one?

In subsequent positions we will try to manually insert subsequent values and count in how many ways we can finish the permutation so it has the required balance.
How to calculate 5-element permutations that start with $[4, 1]$ and have desired balance?

M. Balance of Permutation

What about k-th lexicographically smallest one?

In subsequent positions we will try to manually insert subsequent values and count in how many ways we can finish the permutation so it has the required balance.
How to calculate 5-element permutations that start with $[4, 1]$ and have desired balance?

1 1 2 2 3 3 4 4 5 5

M. Balance of Permutation

What about k-th lexicographically smallest one?

In subsequent positions we will try to manually insert subsequent values and count in how many ways we can finish the permutation so it has the required balance.
How to calculate 5-element permutations that start with $[4, 1]$ and have desired balance?

1 1 2 2 3 3 4 4 5 5

$\downarrow$

(1, 4) (1, 2)

M. Balance of Permutation

What about k-th lexicographically smallest one?

In subsequent positions we will try to manually insert subsequent values and count in how many ways we can finish the permutation so it has the required balance.
How to calculate 5-element permutations that start with $[4, 1]$ and have desired balance?

1 1 2 2 3 3 4 4 5 5

↓

(1, 4) (1, 2)

2 3 3 4 5 5

M. Balance of Permutation

Complexity

On each position we will try to insert each value at most once

M. Balance of Permutation

Complexity

On each position we will try to insert each value at most once – $\mathcal{O}(n^2)$ times we will run an $\mathcal{O}(n^4)$ algorithm.

M. Balance of Permutation

Complexity

On each position we will try to insert each value at most once – $\mathcal{O}(n^2)$ times we will run an $\mathcal{O}(n^4)$ algorithm.
The final complexity will be $\mathcal{O}(n^6)$.