SJTU Dreadnought Contest Editorial

November 10th, 2016

by Mikhail Tikhomirov (MIPT)

Moscow ACM ICPC Workshop, MIPT, 2016

A. The Ultimate Duel

Two players play an infinite rock-paper-scissors game. Each player chooses symbols from a given string in a cyclic fashion. Determine if one of the players wins a larger portion of all games than the other.

A. The Ultimate Duel

Two players play an infinite rock-paper-scissors game. Each player chooses symbols from a given string in a cyclic fashion. Determine if one of the players wins a larger portion of all games than the other.

Outline: simple counting of all pairwise outcomes (i, j) under the condition $i \equiv j \bmod \mathrm{GCD}(n, m)$.

A. The Ultimate Duel

Let n and m be lengths of the first and second player's string respectively. Let $a_i = i \bmod n$, $b_i = i \bmod m$ — indices of symbols chosen by the players on i-th move.

A. The Ultimate Duel

Let n and m be lengths of the first and second player's string respectively. Let $a_i = i \bmod n$, $b_i = i \bmod m$ — indices of symbols chosen by the players on i-th move.

The infinite sequence of pairs (a_i, b_i) loops after nm/k iterations, where k is the GCD of n and m. Moreover, a pair (a, b) occurs in the sequence iff $a \equiv b \bmod k$, and if it does, it occurs exactly once in each loop.

A. The Ultimate Duel

Let n and m be lengths of the first and second player's string respectively. Let $a_i = i \bmod n$, $b_i = i \bmod m$ — indices of symbols chosen by the players on i-th move.

The infinite sequence of pairs (a_i, b_i) loops after nm/k iterations, where k is the GCD of n and m. Moreover, a pair (a, b) occurs in the sequence iff $a \equiv b \bmod k$, and if it does, it occurs exactly once in each loop.

Fix a remainder $l \in [0; k-1]$. Let x_P, x_R, x_S denote the number of corresponding symbols among positions with $i \equiv l \bmod k$ in the first string; define y_P, y_R, y_S similarly.

A. The Ultimate Duel

Let n and m be lengths of the first and second player's string respectively. Let $a_i = i \bmod n$, $b_i = i \bmod m$ — indices of symbols chosen by the players on i-th move.

The infinite sequence of pairs (a_i, b_i) loops after nm/k iterations, where k is the GCD of n and m. Moreover, a pair (a, b) occurs in the sequence iff $a \equiv b \bmod k$, and if it does, it occurs exactly once in each loop.

Fix a remainder $l \in [0; k-1]$. Let x_P, x_R, x_S denote the number of corresponding symbols among positions with $i \equiv l \bmod k$ in the first string; define y_P, y_R, y_S similarly.

Among the games with $i \equiv j \equiv l \bmod k$, the first player wins $x_P y_R + x_R y_S + x_S y_P$ games and loses $x_R y_P + x_S y_R + x_P y_S$ games.

A. The Ultimate Duel

Let n and m be lengths of the first and second player's string respectively. Let $a_i = i \bmod n$, $b_i = i \bmod m$ — indices of symbols chosen by the players on i-th move.

The infinite sequence of pairs (a_i, b_i) loops after nm/k iterations, where k is the GCD of n and m. Moreover, a pair (a, b) occurs in the sequence iff $a \equiv b \bmod k$, and if it does, it occurs exactly once in each loop.

Fix a remainder $l \in [0; k-1]$. Let x_P, x_R, x_S denote the number of corresponding symbols among positions with $i \equiv l \bmod k$ in the first string; define y_P, y_R, y_S similarly.

Among the games with $i \equiv j \equiv l \bmod k$, the first player wins $x_P y_R + x_R y_S + x_S y_P$ games and loses $x_R y_P + x_S y_R + x_P y_S$ games.

These formulas allow us to find the number of games won by each player in $O(n)$ time.

B. Mighty Spell

A string s of n characters of m types is given. Consider all subsequences of s that contain at least one character of each of m types. Find the sum of $f(x) = 2x^3 + 3x^2 + 3x + 3$, where x ranges over all maximal subsegments of all admissible subsequences.

B. Mighty Spell

A string s of n characters of m types is given. Consider all subsequences of s that contain at least one character of each of m types. Find the sum of $f(x) = 2x^3 + 3x^2 + 3x + 3$, where x ranges over all maximal subsegments of all admissible subsequences.

Outline: for a left endpoint of a segment there are at most m values of right endpoint when the set of types inside the segment changes. After some scaling we can quickly process each of these “constant” ranges.

B. Mighty Spell

We can express $f(n) = \sum_{k=1}^{n}(n-k+1)g(k)$ for a certain function $g(k)$. Now we can sum $g(k)$ over all suitable segments instead of only maximal segments.

B. Mighty Spell

We can express $f(n) = \sum_{k=1}^{n}(n-k+1)g(k)$ for a certain function $g(k)$. Now we can sum $g(k)$ over all suitable segments instead of only maximal segments.

Consider a segment $[i;j]$. Let $S(i,j_1)$ be the set of types inside $[i;j]$, and $q(c)$ the number of symbols c outside the segment.

B. Mighty Spell

We can express $f(n) = \sum_{k=1}^{n}(n-k+1)g(k)$ for a certain function $g(k)$. Now we can sum $g(k)$ over all suitable segments instead of only maximal segments.

Consider a segment $[i;j]$. Let $S(i,j_1)$ be the set of types inside $[i;j]$, and $q(c)$ the number of symbols c outside the segment.

The weight of $g(j-i+1)$ for this segment is

$$w(i,j) = \prod_{c \in S} 2^{q(c)} \cdot \prod_{c \notin S} \left(2^{q(c)} - 1\right)$$

B. Mighty Spell

We can express $f(n) = \sum_{k=1}^{n} (n - k + 1) g(k)$ for a certain function $g(k)$. Now we can sum $g(k)$ over all suitable segments instead of only maximal segments.

Consider a segment $[i; j]$. Let $S(i, j_1)$ be the set of types inside $[i; j]$, and $q(c)$ the number of symbols c outside the segment.

The weight of $g(j - i + 1)$ for this segment is

$$w(i, j) = \prod_{c \in S} 2^{q(c)} \cdot \prod_{c \notin S} \left(2^{q(c)} - 1 \right)$$

Notice that if for different j_1, j_2 we have $S(i, j_1) = S(i, j_2)$, then the right half of the product (over $c \notin S$) stays the same, since all these symbols are outside. Moreover, this half depends only on S.

B. Mighty Spell

For each left endpoint i consider all right endpoints $j_1, \ldots, j_k$ such that $S(i, j_?) \neq S(i, j_? - 1)$. If j ranges between consecutive j_k and j_{k+1}, the value of $w(i, j)$ halves for each character appended.

B. Mighty Spell

For each left endpoint i consider all right endpoints $j_1, \ldots, j_k$ such that $S(i, j_?) \neq S(i, j_? - 1)$. If j ranges between consecutive j_k and j_{k+1}, the value of $w(i,j)$ halves for each character appended.

Let $w'(i,j) = w(i,j)2^{j-i+1}$. The above reasoning is equivalent to the fact that $w'(i,j)$ is constant between consecutive j_k and j_{k+1}.

B. Mighty Spell

For each left endpoint i consider all right endpoints $j_1, \ldots, j_k$ such that $S(i, j_?) \neq S(i, j_? - 1)$. If j ranges between consecutive j_k and j_{k+1}, the value of $w(i,j)$ halves for each character appended.

Let $w'(i,j) = w(i,j)2^{j-i+1}$. The above reasoning is equivalent to the fact that $w'(i,j)$ is constant between consecutive j_k and j_{k+1}.

We will count all A_l — sums of $w'(i,j)$ for all segment lengths $l = j - i + 1$ separately. That way we will be able to easily obtain the answer at the end:

$$Ans = \sum_{l=1}^{n} A_l 2^{-l} g(l)$$

B. Mighty Spell

For each left endpoint i consider all right endpoints $j_1, \ldots, j_k$ such that $S(i, j_?) \neq S(i, j_? - 1)$. If j ranges between consecutive j_k and j_{k+1}, the value of $w(i, j)$ halves for each character appended.

Let $w'(i, j) = w(i, j)2^{j-i+1}$. The above reasoning is equivalent to the fact that $w'(i, j)$ is constant between consecutive j_k and j_{k+1}.

We will count all A_l — sums of $w'(i, j)$ for all segment lengths $l = j - i + 1$ separately. That way we will be able to easily obtain the answer at the end:

$$Ans = \sum_{l=1}^{n} A_l 2^{-l} g(l)$$

Since $w'(i, j)$ has $O(m)$ ranges of constant value, accounting for a single endpoint i requires $O(m)$ offline range additions.

B. Mighty Spell

We now process values of i by decreasing. Consider costs of changing from $i + 1$ to i.

B. Mighty Spell

We now process values of i by decreasing. Consider costs of changing from $i+1$ to i.

Among boundaries j_k only one gets changed (namely, $j_1 = i$), that is, the list can be updated in $O(m)$.

B. Mighty Spell

We now process values of i by decreasing. Consider costs of changing from $i+1$ to i.

Among boundaries j_k only one gets changed (namely, $j_1 = i$), that is, the list can be updated in $O(m)$.

Range-constant values of $w'(i, j)$ can be computed by increasing of j_k in $O(m)$: we only to omit a single factor from $\prod_{c \notin S} \left(2^{q(c)} - 1\right)$ and account for different length of the segment.

B. Mighty Spell

We now process values of i by decreasing. Consider costs of changing from $i+1$ to i.

Among boundaries j_k only one gets changed (namely, $j_1 = i$), that is, the list can be updated in $O(m)$.

Range-constant values of $w'(i,j)$ can be computed by increasing of j_k in $O(m)$: we only to omit a single factor from $\prod_{c \notin S} \left(2^{q(c)} - 1\right)$ and account for different length of the segment.

The resulting complexity of the solution is $O(nm)$.

C. The Defense Fence

We are given n points in the plane. Two of them are $(0, 0)$ and $(d, 0)$. Build a polygon with points in the set that contains $(0, 0)$ and $(d, 0)$ as vertices and also doesn't have two vertices with distance more than d apart. Maximize area of the polygon.

C. The Defense Fence

We are given n points in the plane. Two of them are $(0, 0)$ and $(d, 0)$. Build a polygon with points in the set that contains $(0, 0)$ and $(d, 0)$ as vertices and also doesn't have two vertices with distance more than d apart. Maximize area of the polygon.

Obviously it suffices to consider only convex polygons.

C. The Defense Fence

We are given n points in the plane. Two of them are $(0, 0)$ and $(d, 0)$. Build a polygon with points in the set that contains $(0, 0)$ and $(d, 0)$ as vertices and also doesn't have two vertices with distance more than d apart. Maximize area of the polygon.

Obviously it suffices to consider only convex polygons.

Outline: construct upper and lower halves of the polygon while simulating rotating calipers, DP on all states of the process with angle sorting optimization.

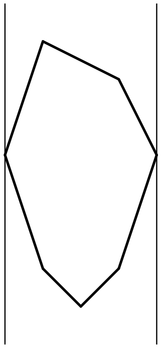
C. The Defense Fence

Recall the “rotating calipers” algorithm for finding diameter (largest distance between vertices) of a convex polygon.

C. The Defense Fence

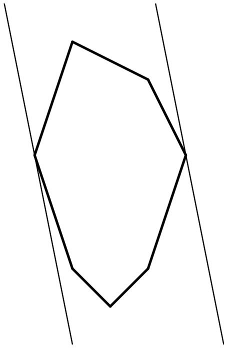
Recall the “rotating calipers” algorithm for finding diameter (largest distance between vertices) of a convex polygon.

Fix a direction d and “lock” the polygon between two lines parallel to d. Let us call vertices that lie on the border lines the *resting* points for the direction d.



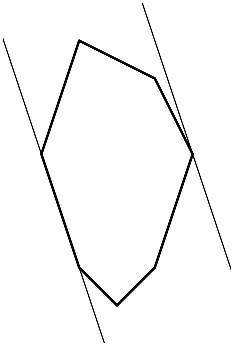
C. The Defense Fence

Gradually rotate d (say, counterclockwise), the lines will rotate around corresponding resting points.



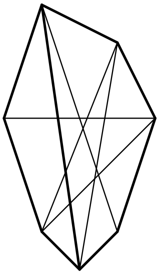
C. The Defense Fence

One (or both) resting point will shift to the next vertex when d becomes parallel to a side of the polygon. Among two sides the one with smaller angle with Oy will be first to be shifted along.



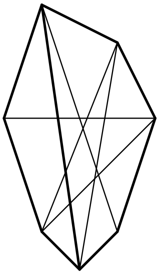
C. The Defense Fence

Continue this process until d travels full circle (it's actually enough to travel half of a circle). The maximal possible distance is attained between a pair of resting point for a certain value of d. (Showing this is an exercise).



C. The Defense Fence

Continue this process until d travels full circle (it's actually enough to travel half of a circle). The maximal possible distance is attained between a pair of resting point for a certain value of d. (Showing this is an exercise).



Clearly, there are only $O(n)$ pairs of points to try.

C. The Defense Fence

The solution to the actual problem proceeds as follows: we will be considering all ways to construct lower and upper parts of polygon in parallel between $(0, 0)$ and $(d, 0)$ counterclockwise.

C. The Defense Fence

The solution to the actual problem proceeds as follows: we will be considering all ways to construct lower and upper parts of polygon in parallel between $(0, 0)$ and $(d, 0)$ counterclockwise.

We will consider the points to be appended in the same order as the “rotating calipers” algorithm would make them resting points.

C. The Defense Fence

The solution to the actual problem proceeds as follows: we will be considering all ways to construct lower and upper parts of polygon in parallel between $(0, 0)$ and $(d, 0)$ counterclockwise.

We will consider the points to be appended in the same order as the "rotating calipers" algorithm would make them resting points. Meanwhile we will make sure that no situation when two resting points are at distance greater than d arises.

C. The Defense Fence

The solution to the actual problem proceeds as follows: we will be considering all ways to construct lower and upper parts of polygon in parallel between $(0, 0)$ and $(d, 0)$ counterclockwise.

We will consider the points to be appended in the same order as the "rotating calipers" algorithm would make them resting points. Meanwhile we will make sure that no situation when two resting points are at distance greater than d arises.

A *situation* (that is, a DP state) is fully described by the pair of resting points and the direction d.

C. The Defense Fence

The solution to the actual problem proceeds as follows: we will be considering all ways to construct lower and upper parts of polygon in parallel between $(0,0)$ and $(d,0)$ counterclockwise.

We will consider the points to be appended in the same order as the "rotating calipers" algorithm would make them resting points. Meanwhile we will make sure that no situation when two resting points are at distance greater than d arises.

A *situation* (that is, a DP state) is fully described by the pair of resting points and the direction d. If we will consider only "critical" moments, i.e. when a resting point is shifted, there are $O(n^3)$ possible situations since d is a vector from a previous resting point ($O(n)$ options) to one of the two current resting point ($O(1)$).

C. The Defense Fence

The solution to the actual problem proceeds as follows: we will be considering all ways to construct lower and upper parts of polygon in parallel between $(0, 0)$ and $(d, 0)$ counterclockwise.

We will consider the points to be appended in the same order as the "rotating calipers" algorithm would make them resting points. Meanwhile we will make sure that no situation when two resting points are at distance greater than d arises.

A *situation* (that is, a DP state) is fully described by the pair of resting points and the direction d. If we will consider only "critical" moments, i.e. when a resting point is shifted, there are $O(n^3)$ possible situations since d is a vector from a previous resting point ($O(n)$ options) to one of the two current resting point ($O(1)$).

For each of these situations we will maximize the partial area (signed sum of directed trapezoids areas) over all ways to reach the particular situation.

C. The Defense Fence

The solution to the actual problem proceeds as follows: we will be considering all ways to construct lower and upper parts of polygon in parallel between $(0, 0)$ and $(d, 0)$ counterclockwise.

We will consider the points to be appended in the same order as the "rotating calipers" algorithm would make them resting points. Meanwhile we will make sure that no situation when two resting points are at distance greater than d arises.

A *situation* (that is, a DP state) is fully described by the pair of resting points and the direction d. If we will consider only "critical" moments, i.e. when a resting point is shifted, there are $O(n^3)$ possible situations since d is a vector from a previous resting point ($O(n)$ options) to one of the two current resting point ($O(1)$).

For each of these situations we will maximize the partial area (signed sum of directed trapezoids areas) over all ways to reach the particular situation. The answer for the situation with $(d, 0)$ and $(0, 0)$ the resting points (note that the order is swapped) is the answer to the global problem.

C. The Defense Fence

An evident $O(n^4)$ solution follows: for each situation consider all options for the next resting point shift. An option is admissible if d rotates counterclockwise.

C. The Defense Fence

An evident $O(n^4)$ solution follows: for each situation consider all options for the next resting point shift. An option is admissible if d rotates counterclockwise.

We can use a standard angle sorting trick to optimize to $O(n^3 \log n)$.

C. The Defense Fence

An evident $O(n^4)$ solution follows: for each situation consider all options for the next resting point shift. An option is admissible if d rotates counterclockwise.

We can use a standard angle sorting trick to optimize to $O(n^3 \log n)$.

Consider all situations with resting points v and u (with many choices of a previous resting point).

C. The Defense Fence

An evident $O(n^4)$ solution follows: for each situation consider all options for the next resting point shift. An option is admissible if d rotates counterclockwise.

We can use a standard angle sorting trick to optimize to $O(n^3 \log n)$.

Consider all situations with resting points v and u (with many choices of a previous resting point).

Consider all possible preceding (with an earlier resting point) and current situations, order all of them by rotating d counterclockwise.

C. The Defense Fence

An evident $O(n^4)$ solution follows: for each situation consider all options for the next resting point shift. An option is admissible if d rotates counterclockwise.

We can use a standard angle sorting trick to optimize to $O(n^3 \log n)$.

Consider all situations with resting points v and u (with many choices of a previous resting point).

Consider all possible preceding (with an earlier resting point) and current situations, order all of them by rotating d counterclockwise.

Since a transfer from a preceding to a current situation is possible whenever d rotates counterclockwise, for each current situation admissible preceding situations from a prefix in the constructed order.

C. The Defense Fence

An evident $O(n^4)$ solution follows: for each situation consider all options for the next resting point shift. An option is admissible if d rotates counterclockwise.

We can use a standard angle sorting trick to optimize to $O(n^3 \log n)$.

Consider all situations with resting points v and u (with many choices of a previous resting point).

Consider all possible preceding (with an earlier resting point) and current situations, order all of them by rotating d counterclockwise.

Since a transfer from a preceding to a current situation is possible whenever d rotates counterclockwise, for each current situation admissible preceding situations from a prefix in the constructed order.

Now follow this order and maintain optimal sub-answer for the prefix that is currently covered. This can be used to obtain the answer for each current situation in $O(1)$.

D. The Road Networks

We are given a sequence $w_1, \ldots, w_n$. Connect i and j with an edge iff $w_i + w_j \geqslant d$. Find the size of the maximal cut in the resulting graph, also find the number of cuts with such size.

D. The Road Networks

We are given a sequence $w_1, \ldots, w_n$. Connect i and j with an edge iff $w_i + w_j \geqslant d$. Find the size of the maximal cut in the resulting graph, also find the number of cuts with such size.

Outline: choose a suitable ordering of vertices, see a simple DP.

D. The Road Networks

Let us call a number x *large* if $2x \geqslant d$, and *small* otherwise. Clearly, all large numbers form a clique, and all small number form an anti-clique.

D. The Road Networks

Let us call a number x *large* if $2x \geqslant d$, and *small* otherwise. Clearly, all large numbers form a clique, and all small number form an anti-clique.

The main idea is to choose a suitable order of vertices to process so that for each small number all adjacent (large) numbers are exactly the large numbers that occur earlier in the order. That allows us to apply DP with parameters (number of vertices processed, number of large numbers in each half of the cut).

D. The Road Networks

Let us call a number x *large* if $2x \geqslant d$, and *small* otherwise. Clearly, all large numbers form a clique, and all small number form an anti-clique.

The main idea is to choose a suitable order of vertices to process so that for each small number all adjacent (large) numbers are exactly the large numbers that occur earlier in the order. That allows us to apply DP with parameters (number of vertices processed, number of large numbers in each half of the cut).

Value of each DP subproblem is a pair (the size of maximal cut with these parameters, the number of such cuts).

D. The Road Networks

First, replace all $w_i > d$ with d.

D. The Road Networks

First, replace all $w_i > d$ with d. Then, let us reorder the sequence w_1, . . . , w_n according to the following order:

$$d, 0, d-1, 1, d-2, 2, \ldots$$

That is, put all occurences of d first, all occurences of 0 immediately after, and so on.

D. The Road Networks

Suppose we are in a state $dp_{k,i}$: we processed k vertices, and we put exactly i of *large* of them in the first part of the cut. Denote L_k the number of large vertices among first k, then exactly $L_k - i$ large vertices are in the second half.

D. The Road Networks

Suppose we are in a state $dp_{k,i}$: we processed k vertices, and we put exactly i of *large* of them in the first part of the cut. Denote L_k the number of large vertices among first k, then exactly $L_k - i$ large vertices are in the second half.

If the next number goes to the first or second half of the cut, the size of cut increases by $L_k - i$ or i respectively (regardless if the new number is large or small).

D. The Road Networks

Suppose we are in a state $dp_{k,i}$: we processed k vertices, and we put exactly i of *large* of them in the first part of the cut. Denote L_k the number of large vertices among first k, then exactly $L_k - i$ large vertices are in the second half.

If the next number goes to the first or second half of the cut, the size of cut increases by $L_k - i$ or i respectively (regardless if the new number is large or small).

Additionally, if the new number is large and goes to the first half, the value of i increases by 1.

D. The Road Networks

Suppose we are in a state $dp_{k,i}$: we processed k vertices, and we put exactly i of *large* of them in the first part of the cut. Denote L_k the number of large vertices among first k, then exactly $L_k - i$ large vertices are in the second half.

If the next number goes to the first or second half of the cut, the size of cut increases by $L_k - i$ or i respectively (regardless if the new number is large or small).

Additionally, if the new number is large and goes to the first half, the value of i increases by 1.

The resulting DP scheme has $O(n^2)$ states and transitions, thus this also is a time complexity bound.

E. Great Hunt

We have a tree on n vertices which is a root with some paths growing out of it (we'll call them *vines*). We also have n simple paths in the tree. Build a bijection between paths and vertices so that each path contains its vertex.

E. Great Hunt

We have a tree on n vertices which is a root with some paths growing out of it (we'll call them *vines*). We also have n simple paths in the tree. Build a bijection between paths and vertices so that each path contains its vertex.

Outline: combine greediness inside vines with maxflow to deal with path between vines.

E. Great Hunt

Let’s handle two partial cases of this problem that will comprise a full solution.

E. Great Hunt

Let's handle two partial cases of this problem that will comprise a full solution.

Case 1: No path contains more than one child of the root (that is, doesn't pass through the root, but might contain it).

E. Great Hunt

Let's handle two partial cases of this problem that will comprise a full solution.

Case 1: No path contains more than one child of the root (that is, doesn't pass through the root, but might contain it).

Now each path is contained in some vine. We now have a set of independent one-dimensional problems: representing each vine as an array and paths as its subsegments, we have to find a distinct point in each subsegment. This can be done by a standard greedy algorithm:

E. Great Hunt

Let's handle two partial cases of this problem that will comprise a full solution.

Case 1: No path contains more than one child of the root (that is, doesn't pass through the root, but might contain it).

Now each path is contained in some vine. We now have a set of independent one-dimensional problems: representing each vine as an array and paths as its subsegments, we have to find a distinct point in each subsegment. This can be done by a standard greedy algorithm:

- Sort all subsegments by increasing the left endpoint position.

E. Great Hunt

Let's handle two partial cases of this problem that will comprise a full solution.

Case 1: No path contains more than one child of the root (that is, doesn't pass through the root, but might contain it).

Now each path is contained in some vine. We now have a set of independent one-dimensional problems: representing each vine as an array and paths as its subsegments, we have to find a distinct point in each subsegment. This can be done by a standard greedy algorithm:

- Sort all subsegments by increasing the left endpoint position.
- Iterate over position x from left to right. Mark subsegments as active as soon as they contain x.

E. Great Hunt

Let's handle two partial cases of this problem that will comprise a full solution.

Case 1: No path contains more than one child of the root (that is, doesn't pass through the root, but might contain it).

Now each path is contained in some vine. We now have a set of independent one-dimensional problems: representing each vine as an array and paths as its subsegments, we have to find a distinct point in each subsegment. This can be done by a standard greedy algorithm:

- Sort all subsegments by increasing the left endpoint position.
- Iterate over position x from left to right. Mark subsegments as active as soon as they contain x.
- For each position x choose an active subsegment with the smallest right endpoint position. If it doesn't contain x, then FAIL.

E. Great Hunt

Let's handle two partial cases of this problem that will comprise a full solution.

Case 1: No path contains more than one child of the root (that is, doesn't pass through the root, but might contain it).

Now each path is contained in some vine. We now have a set of independent one-dimensional problems: representing each vine as an array and paths as its subsegments, we have to find a distinct point in each subsegment. This can be done by a standard greedy algorithm:

- Sort all subsegments by increasing the left endpoint position.
- Iterate over position x from left to right. Mark subsegments as active as soon as they contain x.
- For each position x choose an active subsegment with the smallest right endpoint position. If it doesn't contain x, then FAIL.

This will work in $O(n \log n)$ in total if we use an `std::set` or such for extracting minimums.

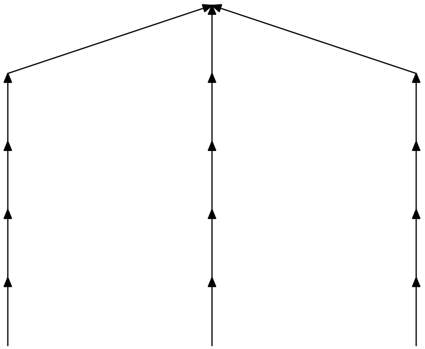
E. Great Hunt

Case 2: each path contains the root.

E. Great Hunt

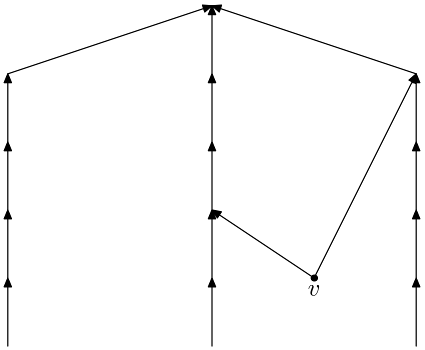
Case 2: each path contains the root.

Let's build a flow network as follows. Direct all edges of the tree towards the root and assign infinite capacities to them. For each vertex add edge to the sink with capacity 1 (not shown on the picture);



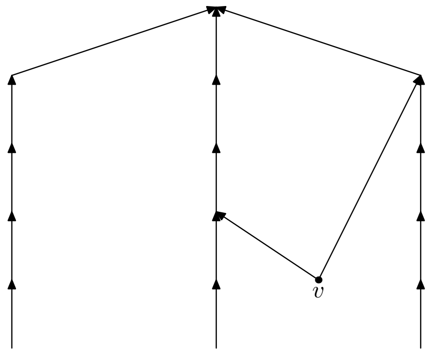
E. Great Hunt

For each path introduce a new vertex v with a 1-capacity edge from the source (not shown). Add 1-capacity edges from v to endpoints of the path.



E. Great Hunt

For each path introduce a new vertex v with a 1-capacity edge from the source (not shown). Add 1-capacity edges from v to endpoints of the path.



Consider a flow in this network. Each path in the decomposition flows through some "path" vertex and out of a tree vertex inside the path. Thus a correct matching can be established iff flow of amount n exists.

E. Great Hunt

To obtain the full solution we can first greedily assign the lowest possible point to each path inside a vine (much like in case 1), and then match the unused points with paths passing through the root (like in case 2).

E. Great Hunt

To obtain the full solution we can first greedily assign the lowest possible point to each path inside a vine (much like in case 1), and then match the unused points with paths passing through the root (like in case 2).

The heaviest part is obviously the second. The constructed network has $O(n)$ vertices and edges, thus a simple FF algorithm will find a maximal flow in $O(n^2)$ time (and Dinic and the likes probably even faster).

E. Great Hunt

To obtain the full solution we can first greedily assign the lowest possible point to each path inside a vine (much like in case 1), and then match the unused points with paths passing through the root (like in case 2).

The heaviest part is obviously the second. The constructed network has $O(n)$ vertices and edges, thus a simple FF algorithm will find a maximal flow in $O(n^2)$ time (and Dinic and the likes probably even faster).

Oh, and a simple Kuhn on vertex-path adjacency matrix passes despite $O(n^3)$ worst-case complexity.

F. The Jump Address

Count the number of permutations on n numbers with sum of indices of all ascents equal to k.

F. The Jump Address

Count the number of permutations on n numbers with sum of indices of all ascents equal to k.

Outline: magic recurrence.

F. The Jump Address

Denote $f_{n,k}$ the answer to the problem. Let us look at first few values of $f_{n,k}$:

$n \backslash k$	0	1	2	3	4	5	6	7	8	9	10
1	1										
2	1	1									
3	1	2	2	1							
4	1	3	5	6	5	3	1				
5	1	4	9	15	20	22	20	15	9	4	1

F. The Jump Address

Denote $f_{n,k}$ the answer to the problem. Let us look at first few values of $f_{n,k}$:

$n \backslash k$	0	1	2	3	4	5	6	7	8	9	10
1	1										
2	1	1									
3	1	2	2	1							
4	1	3	5	6	5	3	1				
5	1	4	9	15	20	22	20	15	9	4	1

Can you guess the rule that governs those numbers?

F. The Jump Address

Turns out $f_{n,k} = \sum_{i=0}^{k-1} f_{n-1,k-i}$ (e.g., in the table below the red number is the sum of blue numbers).

$n \backslash k$	0	1	2	3	4	5	6	7	8	9	10
1	1										
2	1	1									
3	1	2	2	1							
4	1	3	5	6	5	3	1				
5	1	4	9	15	20	22	20	15	9	4	1

F. The Jump Address

Turns out $f_{n,k} = \sum_{i=0}^{k-1} f_{n-1,k-i}$ (e.g., in the table below the red number is the sum of blue numbers).

$n \backslash k$	0	1	2	3	4	5	6	7	8	9	10
1	1										
2	1	1									
3	1	2	2	1							
4	1	3	5	6	5	3	1				
5	1	4	9	15	20	22	20	15	9	4	1

This recurrence can be used to simple compute each number in $O(n^3)$ using prefix sums in each row.

F. The Jump Address

Turns out $f_{n,k} = \sum_{i=0}^{k-1} f_{n-1,k-i}$ (e.g., in the table below the red number is the sum of blue numbers).

$n\backslash k$	0	1	2	3	4	5	6	7	8	9	10
1	1										
2	1	1									
3	1	2	2	1							
4	1	3	5	6	5	3	1				
5	1	4	9	15	20	22	20	15	9	4	1

This recurrence can be used to simple compute each number in $O(n^3)$ using prefix sums in each row.

The simplest way to discover the recurrence probably is meditating long enough while looking at this table.

F. The Jump Address

The number $f_{n,k}$ doesn't change if we count descents positions instead of ascents; in that case the value is called the *major index* of a permutation.

F. The Jump Address

The number $f_{n,k}$ doesn't change if we count descents positions instead of ascents; in that case the value is called the *major index* of a permutation.

To see why the recurrence holds have a look at the following example (borrowed from the paper "Inversions and Major Index for Permutations" by T. Thanatipanonda):

The permutation 5**7**24**63**1 has major index $2 + 5 + 6 = 13$. Consider all ways to insert 8 into the permutation and observe how the major index changes.

F. The Jump Address

$5\mathbf{7}24\mathbf{63}1\underline{8}$: $2+5+6=13$.

$5\mathbf{7}24\mathbf{63}\underline{8}1$: $2+5+7=14$.

$5\mathbf{7}24\mathbf{6}\underline{8}\mathbf{3}1$: $2+6+7=15$.

$5\mathbf{7}24\underline{8}\mathbf{63}1$: $2+5+6+7=20$.

$5\mathbf{7}2\underline{8}4\mathbf{63}1$: $2+4+6+7=19$.

$5\mathbf{7}\underline{8}24\mathbf{63}1$: $3+6+7=16$.

$5\underline{8}\mathbf{7}24\mathbf{63}1$: $2+3+6+7=18$.

$\underline{8}5\mathbf{7}24\mathbf{63}1$: $1+3+6+7=17$.

F. The Jump Address

$5\mathbf{7}24\mathbf{63}1\underline{8}$: $2+5+6=13$.

$5\mathbf{7}24\mathbf{63}\underline{8}1$: $2+5+7=14$.

$5\mathbf{7}24\mathbf{6}\underline{8}\mathbf{3}1$: $2+6+7=15$.

$5\mathbf{7}24\underline{8}\mathbf{63}1$: $2+5+6+7=20$.

$5\mathbf{7}2\underline{8}4\mathbf{63}1$: $2+4+6+7=19$.

$5\mathbf{7}\underline{8}24\mathbf{63}1$: $3+6+7=16$.

$5\underline{8}\mathbf{7}24\mathbf{63}1$: $2+3+6+7=18$.

$\underline{8}5\mathbf{7}24\mathbf{63}1$: $1+3+6+7=17$.

We leave finding a pattern in these numbers and the details of the proof as an exercise.

F. The Jump Address

Still, is there a tangible reason *why* major index is distributed exactly as the number of inversions?

F. The Jump Address

Still, is there a tangible reason *why* major index is distributed exactly as the number of inversions?

It is possible to establish a bijection between permutations with major index k and inversion number k, but no simple and/or natural bijection is known.

F. The Jump Address

Still, is there a tangible reason *why* major index is distributed exactly as the number of inversions?

It is possible to establish a bijection between permutations with major index k and inversion number k, but no simple and/or natural bijection is known.

So, the best answer to the question above is

F. The Jump Address

Still, is there a tangible reason *why* major index is distributed exactly as the number of inversions?

It is possible to establish a bijection between permutations with major index k and inversion number k, but no simple and/or natural bijection is known.

So, the best answer to the question above is

MAGIC

G. The Imaginary Girlfriend

Given a set of vertical and horizontal segments in the plane, find the shortest path length between two points.

G. The Imaginary Girlfriend

Given a set of vertical and horizontal segments in the plane, find the shortest path length between two points.

Outline: convoluted sweep-line+SQRT algorithm that optimizes against full grid cases.

G. The Imaginary Girlfriend

Finding all pairwise intersections within a set of H/V segments is a fairly common sweep-line task. Thus we can build an explicit graph of all intersections and endpoints with segment parts between them. The catch is there can be $\Omega(n^2)$ intersections, and Dijkstra will not be fast enough to deal with such a large graph.

G. The Imaginary Girlfriend

Finding all pairwise intersections within a set of H/V segments is a fairly common sweep-line task. Thus we can build an explicit graph of all intersections and endpoints with segment parts between them. The catch is there can be $\Omega(n^2)$ intersections, and Dijkstra will not be fast enough to deal with such a large graph.

The bad case is a full rectangular grid. The key observation is that if start and finish points are outside a subgraph which is a grid, then most of the intersection points can be erased without changing distances.

G. The Imaginary Girlfriend

Here is a brief description of the algorithm. Suppose that coordinates are compressed, and $x \in [0; n)$.

G. The Imaginary Girlfriend

Here is a brief description of the algorithm. Suppose that coordinates are compressed, and $x \in [0; n)$. Divide $[0; n)$ into $\sim \sqrt{n}$ subsegments, and for each subsegment construct the subgraph that falls within the corresponding vertical strip.

G. The Imaginary Girlfriend

Here is a brief description of the algorithm. Suppose that coordinates are compressed, and $x \in [0; n)$. Divide $[0; n)$ into $\sim \sqrt{n}$ subsegments, and for each subsegment construct the subgraph that falls within the corresponding vertical strip.

Each strip is processed as follows: perform a vertical sweep-line and process all events: start/end of a vertical segment, a horizontal segment. While there are no endpoints inside the strip, the part of the graph is considered a *complete grid*.

G. The Imaginary Girlfriend

Here is a brief description of the algorithm. Suppose that coordinates are compressed, and $x \in [0; n)$. Divide $[0; n)$ into $\sim \sqrt{n}$ subsegments, and for each subsegment construct the subgraph that falls within the corresponding vertical strip.

Each strip is processed as follows: perform a vertical sweep-line and process all events: start/end of a vertical segment, a horizontal segment. While there are no endpoints inside the strip, the part of the graph is considered a *complete grid*.

If the current horizontal segment crosses the strip entirely, it is appended to the complete grid structure.

G. The Imaginary Girlfriend

Here is a brief description of the algorithm. Suppose that coordinates are compressed, and $x \in [0; n)$. Divide $[0; n)$ into $\sim \sqrt{n}$ subsegments, and for each subsegment construct the subgraph that falls within the corresponding vertical strip.

Each strip is processed as follows: perform a vertical sweep-line and process all events: start/end of a vertical segment, a horizontal segment. While there are no endpoints inside the strip, the part of the graph is considered a *complete grid*.

If the current horizontal segment crosses the strip entirely, it is appended to the complete grid structure.

If a start/finish point or an endpoint of any kind is encountered, explicit edges between intersections are needed, and the currently accumulated complete grid is “flushed”.

G. The Imaginary Girlfriend

While flushing the grid we create all segments in the grid, but only intersections which are on the border are introduced. No endpoints or start/finish are inside, thus the distances are preserved.

G. The Imaginary Girlfriend

While flushing the grid we create all segments in the grid, but only intersections which are on the border are introduced. No endpoints or start/finish are inside, thus the distances are preserved.

Finally a careful “gluing” of adjacent strips is required.

G. The Imaginary Girlfriend

While flushing the grid we create all segments in the grid, but only intersections which are on the border are introduced. No endpoints or start/finish are inside, thus the distances are preserved.

Finally a careful "gluing" of adjacent strips is required.

One can ensure that the described process constructs a graph of size $O(n\sqrt{n})$. Finally, Dijkstra algorithm is used to find the shortest path.

G. The Imaginary Girlfriend

For more details refer to the attached paper (please share responsibly).

The algorithm in the paper is actually capable to answer online distance queries in $O(\log n)$ after $O(n\sqrt{n})$ preprocessing.

H. The Kirakira Cycle

We are given an integer n. Define a function $f(x) = \sum_{k=1}^{n} x \bmod k$. Find the largest length of a cycle of this function.

H. The Kirakira Cycle

We are given an integer n. Define a function $f(x) = \sum_{k=1}^{n} x \bmod k$. Find the largest length of a cycle of this function.

Outline: Straightforward $O(n^2 \log n)$ is not too much, but even if it is, lots of room to optimize.

H. The Kirakira Cycle

Since $f(x) \leqslant 0 + \ldots + n - 1 = n(n-1)/2$ for every x, we can ignore numbers larger than $n(n-1)/2$ since they don't lie on any cycle.

H. The Kirakira Cycle

Since $f(x) \leqslant 0 + \ldots + n - 1 = n(n-1)/2$ for every x, we can ignore numbers larger than $n(n-1)/2$ since they don't lie on any cycle.

Let us compute $f(x)$ for all $x = 0, \ldots, n(n-1)/2$. Explicit computation would require $O(n^3)$ time.

H. The Kirakira Cycle

Since $f(x) \leqslant 0 + \ldots + n - 1 = n(n-1)/2$ for every x, we can ignore numbers larger than $n(n-1)/2$ since they don't lie on any cycle.

Let us compute $f(x)$ for all $x = 0, \ldots, n(n-1)/2$. Explicit computation would require $O(n^3)$ time.

Consider $f(x) - f(x-1) = \sum_{k=1}^{n} \delta(x,k)$, where $\delta(x,k) = (x \bmod k) - ((x-1) \bmod k)$.

H. The Kirakira Cycle

Since $f(x) \leqslant 0 + \ldots + n - 1 = n(n-1)/2$ for every x, we can ignore numbers larger than $n(n-1)/2$ since they don't lie on any cycle.

Let us compute $f(x)$ for all $x = 0, \ldots, n(n-1)/2$. Explicit computation would require $O(n^3)$ time.

Consider $f(x) - f(x-1) = \sum_{k=1}^{n} \delta(x, k)$, where $\delta(x, k) = (x \bmod k) - ((x-1) \bmod k)$.

$\delta(x, k) = -(k-1)$ if x is divisible by k, and 1 otherwise. Thus, there are $O(n^2 \log n)$ values of $\delta(x, k) \neq 1$.

H. The Kirakira Cycle

Since $f(x) \leqslant 0 + \ldots + n - 1 = n(n-1)/2$ for every x, we can ignore numbers larger than $n(n-1)/2$ since they don't lie on any cycle.

Let us compute $f(x)$ for all $x = 0, \ldots, n(n-1)/2$. Explicit computation would require $O(n^3)$ time.

Consider $f(x) - f(x-1) = \sum_{k=1}^{n} \delta(x, k)$, where $\delta(x, k) = (x \bmod k) - ((x-1) \bmod k)$.

$\delta(x, k) = -(k-1)$ if x is divisible by k, and 1 otherwise. Thus, there are $O(n^2 \log n)$ values of $\delta(x, k) \neq 1$.

This allows us to compute $f(x) - f(x-1)$ for all $x = 1, \ldots, n(n-1)/2$ in $O(n^2 \log n)$ time, and then compute prefix sums to obtain values of $f(x)$.

H. The Kirakira Cycle

Finally, use the values of $f(x)$ to traverse and find all cycles in the function graph of f. Note that memory limit is not generous, so one should use an array of `char`'s or a `bitset` for marking visited vertices.

H. The Kirakira Cycle

Finally, use the values of $f(x)$ to traverse and find all cycles in the function graph of f. Note that memory limit is not generous, so one should use an array of `char`'s or a `bitset` for marking visited vertices.

The resulting solution is $O(n^2 \log n)$ time and $O(n^2)$ space. It also may require additional optimization to fit in TL.

H. The Kirakira Cycle

Finally, use the values of $f(x)$ to traverse and find all cycles in the function graph of f. Note that memory limit is not generous, so one should use an array of `char`'s or a `bitset` for marking visited vertices.

The resulting solution is $O(n^2 \log n)$ time and $O(n^2)$ space. It also may require additional optimization to fit in TL.

Authors' solution uses additional ideas:

H. The Kirakira Cycle

Finally, use the values of $f(x)$ to traverse and find all cycles in the function graph of f. Note that memory limit is not generous, so one should use an array of `char`'s or a `bitset` for marking visited vertices.

The resulting solution is $O(n^2 \log n)$ time and $O(n^2)$ space. It also may require additional optimization to fit in TL.

Authors' solution uses additional ideas:

- since $f(x)$ is somewhat "random", we can assume that all values of $f(x)$ are concentrated around $n(n-1)/4$.

H. The Kirakira Cycle

Finally, use the values of $f(x)$ to traverse and find all cycles in the function graph of f. Note that memory limit is not generous, so one should use an array of `char`'s or a `bitset` for marking visited vertices.

The resulting solution is $O(n^2 \log n)$ time and $O(n^2)$ space. It also may require additional optimization to fit in TL.

Authors' solution uses additional ideas:

- since $f(x)$ is somewhat "random", we can assume that all values of $f(x)$ are concentrated around $n(n-1)/4$.
- to optimize for memory, one can choose random x as a starting point of the cycle and find the cycle in $O(l)$, where l is the length of the cycle. Again, f is "random", so there won't be really long cycles. Repeat as many times as needed to find the longest cycle (note that it has the highest probability to be found randomly).

I. The Impressive Path

Construct a simple path of length t between opposite corners of a rectangular $n \times m$ field. $n, m \geqslant 8$, $n + m \leqslant t \leqslant \frac{3}{4}nm$.

I. The Impressive Path

Construct a simple path of length t between opposite corners of a rectangular $n \times m$ field. $n, m \geqslant 8$, $n + m \leqslant t \leqslant \frac{3}{4}nm$.

Outline: brute-force with clever local-global decision making. Or not.

I. The Impressive Path

A summary of authors’ solution (to my best understanding). Our path will consist of *dense* and *sparse* parts.

I. The Impressive Path

A summary of authors' solution (to my best understanding). Our path will consist of *dense* and *sparse* parts.

A *dense* part starts and ends in particular points of a 4×4 square and fills it (almost) completely. A sufficient number of dense parts of different kinds can be constructed using brute-force or manually.

I. The Impressive Path

A summary of authors' solution (to my best understanding). Our path will consist of *dense* and *sparse* parts.

A *dense* part starts and ends in particular points of a 4×4 square and fills it (almost) completely. A sufficient number of dense parts of different kinds can be constructed using brute-force or manually.

A *sparse* part just moves from point to point using a short path.

I. The Impressive Path

A summary of authors' solution (to my best understanding). Our path will consist of *dense* and *sparse* parts.

A *dense* part starts and ends in particular points of a 4×4 square and fills it (almost) completely. A sufficient number of dense parts of different kinds can be constructed using brute-force or manually.

A *sparse* part just moves from point to point using a short path.

Construct a part with an optimized recursion: if there are a lot more steps to make than the distance to finish, place an appropriate dense block. Otherwise, recursively try to find a short path with appropriate length.

I. The Impressive Path

A summary of authors' solution (to my best understanding). Our path will consist of *dense* and *sparse* parts.

A *dense* part starts and ends in particular points of a 4×4 square and fills it (almost) completely. A sufficient number of dense parts of different kinds can be constructed using brute-force or manually.

A *sparse* part just moves from point to point using a short path.

Construct a part with an optimized recursion: if there are a lot more steps to make than the distance to finish, place an appropriate dense block. Otherwise, recursively try to find a short path with appropriate length.

A summary of a team's AC solution: brute-force the path pruning the case when we don't have enough steps to reach the finish.

J. The Magic Square

For a given n, present a way to cut a square $s \times s$ (with s of your choice) into exactly n squares.

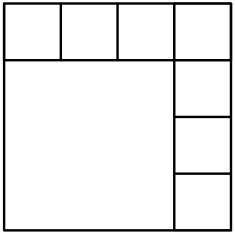
J. The Magic Square

For a given n, present a way to cut a square $s \times s$ (with s of your choice) into exactly n squares.

Outline: simple construction with a couple of cases.

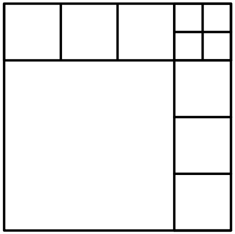
J. The Magic Square

If n is even and greater than 2, we can use a construction depicted below:



J. The Magic Square

If n is odd and greater than 5, we can cut the square into $n - 3$ squares as before, and then cut any square into four equal parts:



J. The Magic Square

The only values without a solution are $n = 2, 3, 5$ ($n = 1$ is trivial).

J. The Magic Square

The only values without a solution are $n = 2, 3, 5$ ($n = 1$ is trivial).

Let us prove that these values have no solution. If the partition of the initial square S is non-trivial, the corners of S end up belonging to different squares s_{LU}, s_{RU}, s_{LD}, s_{RD}, thus $n \geqslant 4$.

J. The Magic Square

The only values without a solution are $n = 2, 3, 5$ ($n = 1$ is trivial).

Let us prove that these values have no solution. If the partition of the initial square S is non-trivial, the corners of S end up belonging to different squares s_{LU}, s_{RU}, s_{LD}, s_{RD}, thus $n \geqslant 4$.

If $n = 5$, there is at most one additional square in the partition (that is, not containing any corner of S). It is adjacent to at most one side of S, thus for other three sides we have $l(s) + l(s') = l(S)$, where $l(s)$ is the length of a side of a square s, s and s' are squares adjacent to the side of S.

J. The Magic Square

The only values without a solution are $n = 2, 3, 5$ ($n = 1$ is trivial).

Let us prove that these values have no solution. If the partition of the initial square S is non-trivial, the corners of S end up belonging to different squares s_{LU}, s_{RU}, s_{LD}, s_{RD}, thus $n \geqslant 4$.

If $n = 5$, there is at most one additional square in the partition (that is, not containing any corner of S). It is adjacent to at most one side of S, thus for other three sides we have $l(s) + l(s') = l(S)$, where $l(s)$ is the length of a side of a square s, s and s' are squares adjacent to the side of S.

That implies that $l(s_{LU}) = l(s_{RD}) = l(S) - l(s_{LD}) = l(S) - l(s_{RU})$. But we also have $l(s_{LU}) + l(s_{RD}) \leqslant l(S)$, $l(s_{LD}) + l(s_{RU}) \leqslant l(S)$. All these conditions imply that all four squares are equal and partite S. Contradiction.