

Problem A. Black Friday

Let's consider if swapping two adjacent coupons, (a_i, d_i) and (a_{i+1}, d_{i+1}) , results in a better final total. Let the total before using either coupon be T . Let $p_i = 1 - \frac{d_i}{100}$. The total after using both coupons can then be computed as:

$$T' = p_{i+1}(p_i(T - a_i) - a_{i+1})$$

So in order for swapping coupon i to come before coupon $i + 1$:

$$\begin{aligned} p_{i+1}(p_i(T - a_i) - a_{i+1}) &< p_i(p_{i+1}(T - a_{i+1}) - a_i) \\ p_{i+1}p_iT - p_{i+1}p_ia_i - p_{i+1}a_{i+1} &< p_{i+1}p_iT - p_{i+1}p_ia_{i+1} - p_ia_i \\ p_{i+1}p_ia_i + p_{i+1}a_{i+1} &> p_{i+1}p_ia_{i+1} + p_ia_i \\ p_{i+1}p_ia_i - p_ia_i &> p_{i+1}p_ia_{i+1} - p_{i+1}a_{i+1} \\ p_ia_i(p_{i+1} - 1) &> p_{i+1}a_{i+1}(p_i - 1) \\ \frac{p_ia_i}{p_i - 1} &> \frac{p_{i+1}a_{i+1}}{p_{i+1} - 1} \end{aligned}$$

Therefore, we can sort the coupons in decreasing order of $\frac{p_ia_i}{p_i - 1}$. Be wary of dividing by 0.

Problem B. Collecting Strawberries

Badelines's optimal strategy is to greedily increase all edges on a min cut of the cost graph, uniformly. This can be proven using LP Duality. Let's start by looking at the LP for shortest path from s to t for the weighted graph $G = (V, E)$, where $\text{in}(u)$ denotes the set of edges entering u and $\text{out}(u)$ denotes the set of edges leaving u :

$$\begin{aligned} \min \quad & \sum_{e \in E} e_w x_e \\ \text{s.t.} \quad & \sum_{e \in \text{in}(u)} x_e - \sum_{e \in \text{out}(u)} x_e = \begin{cases} -1 & \text{if } u = s \\ 1 & \text{if } u = t \\ 0 & \text{otherwise} \end{cases} \\ & x \geq 0 \end{aligned}$$

The dual of this LP is:

$$\begin{aligned} \max \quad & \pi_t - \pi_s \\ \text{s.t.} \quad & \pi_{e_u} - \pi_{e_v} \leq e_w \quad \forall e \in E \end{aligned}$$

Let's modify the dual of the shortest path LP for Badeline's decisions:

$$\begin{aligned} \max \quad & \pi_t - \pi_s \\ \text{s.t.} \quad & \pi_{e_u} - \pi_{e_v} - l_e \leq 0 \quad \forall e \in E \\ & \sum_{e \in E} l_e e_c \leq k \\ & l \geq 0 \end{aligned}$$

Taking the dual of this new LP gives:

$$\begin{aligned}
& \min k\lambda \\
& \text{s.t.} \quad \sum_{e \in \text{in}(u)} x_e - \sum_{e \in \text{out}(u)} x_e = \begin{cases} -1 & \text{if } u = s \\ 1 & \text{if } u = t \\ 0 & \text{otherwise} \end{cases} \\
& \quad x_e \leq e_c \lambda \quad \forall e \in E \\
& \quad x \geq 0 \\
& \quad \lambda \geq 0
\end{aligned}$$

Let $\lambda = \frac{1}{F}$ and $x = \frac{f}{F}$. The LP now becomes:

$$\begin{aligned}
& \min \frac{k}{F} \\
& \text{s.t.} \quad \sum_{e \in \text{in}(u)} f_e - \sum_{e \in \text{out}(u)} f_e = \begin{cases} -F & \text{if } u = s \\ F & \text{if } u = t \\ 0 & \text{otherwise} \end{cases} \\
& \quad f_e \leq e_c \\
& \quad f \geq 0 \\
& \quad F \geq 0
\end{aligned}$$

Since minimizing $\frac{k}{F}$ is equivalent to maximizing F , this LP is equivalent to maximizing the flow of between the starting and ending nodes, such that each edge has a maximum capacity of e_c . Since the maximum flow is equivalent to the minimum cut, Badeline's optimal strategy is to uniformly increase the weight of all edges on the minimum cut, achieving an answer of $\frac{k}{C}$ where C is the total cost of all edges on the minimum cut.

On a 1000 by 1000 grid graph, any standard flow algorithm will be too slow. Let's consider the planar dual of the grid graph. In this dual graph, we will create a node for each finite face, and connect the corresponding nodes of each pair of adjacent faces with an edge with weight equal to the cost of the edge separating those two faces. For all faces bordering the left and top sides of the grid, connect them to a starting node. For all faces bordering the right and bottom sides of the grid, connect them to an ending node. These edges will also have weight equal to the edge separating the cell from the infinite region.

In the dual graph, any path from the start to end node corresponds to a cut in the original graph, and vice versa. Therefore we can compute the minimum cut of the original graph by finding the shortest path in the dual graph using Dijkstra's algorithm in $O(nm \lg nm)$ time.

Problem C. File Your Taxes

Let $f_i(x)$ be the amount of taxes paid on x dollars in tax system i , and let $g(x) = f_2(x) - f_1(x)$. Note that $f_1(x)$, $f_2(x)$, and $g(x)$ are all piecewise linear. For some value of x , we will start by placing all earnings into tax system 2. Then we need to find the value y such that $y \leq x$ and $g(y)$ is maximal. Our answer will then be $f_2(x) - g(y)$. To answer queries of x , we will use a sweep. All event points will be entering a new bracket in system 1, entering a new bracket in system 2, or hitting a query amount. As we sweep we can then maintain the total amount of taxes for placing all money into tax systems 1 and 2 and the optimal value of $g(y)$. Thus, we can answer all queries offline in $O(n + q \lg q)$.

Problem D. Hurricane Mapping

In pass one, add a point to the corner of the initial bounding square that maximizes the area of the convex hull. This guarantees that the area of the convex hull is at least $\frac{1000^2}{2}$.

To prove this, let's consider four points, one on each edge of the bounding square. Any additional points only increases the area, so if this quadrilateral covers over half the area of the bounding square, then so

will any convex hull with more than four points. For simplicity, we scale the initial coordinate system down to a 1 by 1 square. Let the four points be $B = (b, 0)$, $L = (0, l)$, $T = (t, 1)$, and $R = (1, r)$. Each corner is a right triangle so we can express the area of each of these triangles as $\frac{bl}{2}$, $\frac{t(1-l)}{2}$, $\frac{(1-t)(1-r)}{2}$, and $\frac{(1-b)r}{2}$. Using shoelace, we can find the area of the initial convex hull to be $\frac{1}{2}(1 + br - bl + tl - tr)$. Letting $m = \max(\frac{bl}{2}, \frac{t(1-l)}{2}, \frac{(1-t)(1-r)}{2}, \frac{(1-b)r}{2})$, the area of the new convex hull will be $\frac{1}{2}(1 + br - bl + tl - tr + 2m)$. Our goal is to prove that $\frac{1}{2}(1 + br - bl + tl - tr + 2m) \geq \frac{1}{2}$. Rearranging this inequality gives $br - bl + tl - tr = (t - b)(r - l) \leq 2m$. Let's break this into a few cases:

- If $t \leq b$ and $r \geq l$, or $t \geq b$ and $r \leq l$, then $(t - b)(r - l) \leq 0$ and therefore also less than or equal to $2m$ since m must be non-negative.
- If $t \geq b$ and $r \geq l$, then $(t - b)(r - l) \leq (1 - b)r$, so $(t - b)(r - l)$ must be less than or equal to double the area of the bottom-right triangle, and therefore must be at most $2m$.
- If $t \leq b$ and $r \leq l$, then $(t - b)(r - l) \leq bl$, so $(t - b)(r - l)$ must be less than or equal to double the area of the bottom-left triangle, and therefore must be at most $2m$.

So in all cases, $(t - b)(r - l) \leq 2m$ and therefore the area of the new convex hull must be at least half of the bounding square. Any additional points can only increase the area of the convex hull.

When moving to the new coordinate system where the convex hull can exist anywhere in a 5000 by 5000 bounding square, the new convex hull will contain at least $\frac{1}{50}$ of the area, which means that we have roughly a 2% chance of randomly choosing a point within the convex hull. If we choose 800 points randomly, we have roughly a 10^{-7} chance of not finding any point inside the convex hull. Note that deterministic methods exist for finding a point in the convex hull.

Now, we need to use the point we found to build back the convex hull. To do this we will walk to the left until we find the furthest left point for the current y value, which will use less than 1000 queries. Then, we need to walk around the perimeter of the convex hull to find all boundary points. This can be done by moving in one direction as far as possible while staying in the convex hull, then move one point in the direction after it. As soon as this step no longer remains in the convex hull, we will change the direction we walk in. Since the convex hull is bounded by a width of 1000 and height of 1000, this will use at most 8000 queries, giving a total number of queries of up to 9800. This is quite close to the query limit so extra care should be taken during implementation to avoid going over the query limit.

Problem E. IPv6 Address Compression

This problem can be solved by using a segment tree with lazy propagation. Let's consider only the characters 0 and 1, and that we can only remove 0's. In each node we will store L , R , c_L , c_R , and t , corresponding to the left end point of the range, the right end point of the range, the number of colons on the left side of the range, the number of colons on the right side of the range, and the total number of characters in the range. In order to store this data in a segment tree, we need to be able to merge two ranges together, and be able to apply an update over the entire $[L, R]$ range. To avoid double counting, we will count any colon in between two ranges as a part of the right range's left colon count.

Update:

- If updating a range to 0, set t equal to 0, c_L to $\min(2, \lfloor \frac{R}{k} \rfloor - \lfloor \frac{L-1}{k} \rfloor)$, and c_R to 0.
- If updating a range to 1, set t equal to $R - L + 1 + \lfloor \frac{R}{k} \rfloor - \lfloor \frac{L}{k} \rfloor$, c_L to 1 if L is larger than 0 and divisible by k (assuming 0 indexing), otherwise set it to 0, and set c_R to 0.

Merge:

- If the left total is 0, merge the left range's colons into the right range's c_L .

- If the right total is 0, merge the right range's colons into the left range's c_R .
- Otherwise:
 - The merged c_L value is equivalent to the left range's c_L .
 - The merged c_R value is equivalent to the right range's c_R .
 - The merged t is equal to the sum of the left range's t , the right range's t , and the merged value of the left range's c_R and the right range's c_L .

To generalize this to strings using hexadecimal characters, create 16 segment trees for each of the characters. For the c -th segment tree, a 0 will correspond to c occurring in the corresponding position and a 1 will correspond to any other character. That is, the c -th segment tree computes the compressed character count for deleting the character c . This allows us to answer each update in $O(16(n + q \lg n))$ time.

Problem F. K-Star Sets

This problem can be solved using dynamic programming. The state will be comprised of two values: the current node and a value c , which denotes the type of connection between the current node and its parent. If c is 0, then the current node is not connected to its parent, if it is 1, then the current node is connected to its parent and the parent is the center of a k -star, and if it is 2, then the current node is connected to its parent and the current node is the center of a k -star.

There are three different types of transitions that can occur for each state:

1. Connect to no children if $c = 0$ or $c = 1$: Add $\prod_{nxt \in \text{children}(cur)} f(nxt, 0)$ to the answer.
2. Connect to one child which is the center of a k -star if $c = 0$: For each child, v add $f(v, 2) \prod_{nxt \in \text{children}(cur) \setminus \{v\}} f(nxt, 0)$ to the answer. Use prefix/suffix products to efficiently compute this value.
3. Connect to k or $k - 1$ children if $c = 0$ or $c = 2$, respectively: Letting t be the target number of children, add $[x^t] \prod_{nxt \in \text{children}(cur)} (f(nxt, 0) + x \cdot f(nxt, 1))$. We can compute this polynomial efficiently using NTT and a divide and conquer.

In total, computing this DP takes $O(n \lg^2 n)$.

Problem G. LCM Subset Count

Let the prime factorization of k be $p_1^{a_1} p_2^{a_2} \dots p_m^{a_m}$. Since the product of the first 16 prime numbers is larger than 10^{18} , $m \leq 15$. Let x be a bitmask of m bits, where the i -th bit corresponds to the i -th prime power in the prime factorization of k . Define a_x to be the number of integers from 1 to n that are divisible by the product of the prime powers in x :

$$a_x = \frac{n}{\prod_{i \in x} p_i^{a_i}}$$

Let b_x be the number of integers divisible by each prime power in x , but not divisible by any prime power not in x . This can be computed using a_x using inclusion-exclusion:

$$b_x = \sum_{y \text{ s.t. } x \subseteq y} (-1)^{|y| - |x|} a_y$$

Let c_x be the number of integers not divisible by any prime power not in x . This can be computed from b_x as:

$$c_x = \sum_{y \text{ s.t. } y \subseteq x} b_y$$

From c_x , the number of non-empty subsets of the integers 1 to n whose LCM is not divisible by any prime power not in x can be computed as $2^{c_x} - 1$. Let d_x be the number of non-empty subsets of integers from 1 to n whose LCM is divisible by all prime powers in x and not divisible by any prime power not in x . This can be computed using inclusion-exclusion:

$$d_x = \sum_{y \text{ s.t. } y \subseteq x} (-1)^{|x|-|y|} (2^{c_y} - 1)$$

The final answer is d_{2^m-1} . Using the Zeta Transform (SOS DP) and Mobius Transform (Inverse SOS DP), this can be computed in $O(m2^m + 2^m \lg(n) + F)$ per test case, where F is the time complexity used for computing the prime factorization of k . Note that the time complexity can be optimized to $O(m2^m + 2^m \lg(M) + F)$, where M is the modulo, using Fermat's Little Theorem.

Problem H. New Password

For simplicity, we will treat all ranges as inclusive-exclusive and 0-indexed. Let's consider the trie built on the suffixes of p . For node i in the preorder traversal of the trie, let s_i be the substring of p corresponding to the i -th node and let c_i be the number of suffixes that pass through the i -th node. Assuming the trie edges for each node are stored in alphabetical order, for $i < j$, s_i will be lexicographically smaller than s_j . Thus, we can write p' as $s_1^{c_1} s_2^{c_2} s_3^{c_3} \dots$ where $s_i^{c_i}$ denotes the string s_i repeated c_i times. To solve this problem, we must be able to compute the substring over the range $[L_i, R_i)$ of $r_i = s_i^{c_i} s_{i+1}^{c_{i+1}} \dots$ for the i -th node where L_1 and R_1 correspond to the range given in input.

We will begin by computing t_i which corresponds to the total number of characters that the subtree rooted at the i -th node can contribute to the answer. This can be done by adding $c_j T(|s_j|)$ for each node j that is in the subtree of the i -th node, where $T(x)$ is the x -th triangle number. Now to compute the substring over $[L_i, R_i)$ of r_i , we will begin by adding in the contribution of $s_i^{c_i}$. If $L_i < |s_i^{c_i}|$, we will add in the substring of $s_i^{c_i}$ over the range $[L_i, \min(R_i, |s_i^{c_i}|))$. Otherwise, the initial answer will be the empty string. Let the number of letters we added be a_i . We will now compute the rest of the answer by recursing to each of our children. To get the answer for one child, c , we will compute the substring over $[L_c, R_c)$ of r_c , where $L_c = \max(0, \min(t_c, L_i - h_c - a_i))$ and $R_c = \max(0, \min(t_c, R_i - h_c - a_i))$, where h_c is the sum of all letters that all previous children's subtrees could contribute. We can then append this result to our answer for the i -th node.

The issue with this approach is that the trie has $O(n^2)$ nodes. One optimization we can make is to return early if the size of the range $[L_i, R_i)$ is zero. Doing this means we will only ever traverse $O(k)$ nodes, where k is the size of the input range. Thus, if we can build and store the trie more efficiently, then we can still simulate the trie traversal.

To do this, we will begin by constructing the suffix array on p along with the LCP array. We can then bucket each i by the length of the LCP at i . Now each node in the trie corresponds to a range of indices in the suffix array and the length of the current substring of p . To traverse to a child node, we then need to search for the indices in the current range corresponding to an LCP value equal to the size of the current substring. We can maintain s_i as we simulate the traversal and compute c_i as the length of the current range in the suffix array. To compute t_i efficiently, we can use a prefix sum on the triangle number corresponding to the length of the suffix at each position in the suffix array.

Problem I. Pirates on a Boat

Let p_1 be the starting polyline and p_2 be the ending polyline. Let's begin by determining the probability that p_2 is reached from a single point, q , on p_1 . For some small portion of p_2 , dp_2 , define r as the distance from q to dp_2 , θ_1 as the angle between the chosen direction and the normal vector at q , and θ_2 as the

angle between the chosen direction and the normal vector for dp_2 . The probability of choosing a certain direction is given in the problem as $\frac{\cos \theta_1}{2}$. The probability of choosing a direction such that it hits dp_2 is $V(q, dp_2) \frac{\cos \theta_2 dp_2}{r}$, where $V(q, dp_2)$ is 0 if something blocks the path from q to dp_2 , and 1 otherwise. This is because we must first project dp_2 on the perpendicular of the direction vector, and then project this onto the unit circle centered on q . This means the probability of reaching p_2 from q is:

$$\int_{p_2} V(q, dp_2) \frac{\cos \theta_1 \cos \theta_2}{2r} dp_2$$

Let $|p_1|$ denote the total length of p_1 . Integrating over all choices of q gives:

$$f(p_1, p_2) = \frac{1}{|p_1|} \int_{p_1} \int_{p_2} V(q, dp_2) \frac{\cos \theta_1 \cos \theta_2}{2r} dp_2 dp_1$$

One important consequence of this equation is that $|p_1|f(p_1, p_2) = |p_2|f(p_2, p_1)$. This is referred to as reciprocity. Using this fact, along with several rules of probability, we can now compute the answer algebraically!

For now, let's assume that p_1 is an upper hull and p_2 is a lower hull. Let's denote A and B as the left and right endpoints of the starting polyline and C and D as the left and right endpoints of the ending polyline. Let $s(q, r)$ denote the shortest path from q to r without crossing the starting or ending polylines. Let's consider the enclosed space formed by p_1 , $s(A, C)$, and $s(B, C)$. In an enclosed space, no matter where we start on one side, we must cross over one of the other two, so we can form the following equations:

$$\begin{aligned} f(p_1, s(A, C)) + f(p_1, s(B, C)) &= 1 \\ f(s(A, C), p_1) + f(s(A, C), s(B, C)) &= 1 \\ f(s(B, C), p_1) + f(s(B, C), s(A, C)) &= 1 \end{aligned}$$

In the first equation, we can multiply everything by $|p_1|$. Similarly, for the second and third equations, we can multiply everything by $|s(A, C)|$ and $|s(B, C)|$, respectively.

$$\begin{aligned} |p_1|f(p_1, s(A, C)) + |p_1|f(p_1, s(B, C)) &= |p_1| \\ |s(A, C)|f(s(A, C), p_1) + |s(A, C)|f(s(A, C), s(B, C)) &= |s(A, C)| \\ |s(B, C)|f(s(B, C), p_1) + |s(B, C)|f(s(B, C), s(A, C)) &= |s(B, C)| \end{aligned}$$

Now applying reciprocity:

$$\begin{aligned} |p_1|f(p_1, s(A, C)) + |p_1|f(p_1, s(B, C)) &= |p_1| \\ |p_1|f(p_1, s(A, C)) + |s(A, C)|f(s(A, C), s(B, C)) &= |s(A, C)| \\ |p_1|f(p_1, s(B, C)) + |s(A, C)|f(s(A, C), s(B, C)) &= |s(B, C)| \end{aligned}$$

Finally, add together the first two equations and subtract the third:

$$2|p_1|f(p_1, s(A, C)) = |p_1| + |s(A, C)| - |s(B, C)| \rightarrow f(p_1, s(A, C)) = \frac{|p_1| + |s(A, C)| - |s(B, C)|}{2|p_1|}$$

Using a similar system of equations, we can derive that:

$$f(p_1, s(B, D)) = \frac{|p_1| + |s(B, D)| - |s(A, D)|}{2|p_1|}$$

Now let's consider the enclosed region formed by p_1 , p_2 , $s(A, C)$, and $s(B, D)$. Since any path must either reach p_2 or cross $s(A, C)$ or $s(B, D)$:

$$f(p_1, p_2) = 1 - f(p_1, s(A, C)) - f(p_1, s(B, D)) = 1 - \frac{|p_1| + |s(A, C)| - |s(B, C)|}{2|p_1|} - \frac{|p_1| + |s(B, D)| - |s(A, D)|}{2|p_1|}$$

Simplifying this gives us:

$$f(p_1, p_2) = \frac{|s(B, C)| + |s(A, D)| - |s(A, C)| - |s(B, D)|}{2|p_1|}$$

All shortest path lengths can be computed efficiently in $O(n + m)$ time by computing the hull tangents using a two pointer sweep. The challenge now lays in how to handle concavities. Concavities in the ending polyline do not affect the answer. This is because once a path crosses the hull of the polyline, it must reach the polyline. For the starting polyline, let p_1 correspond to the corresponding upper hull and let q_1 correspond to the original polyline with concavities. Since $f(p_1, q_1)$ must be equal to 1, by reciprocity $f(q_1, p_1) = \frac{|p_1|}{|q_1|} f(p_1, q_1)$. Thus the final answer is

$$\frac{|s(B, C)| + |s(A, D)| - |s(A, C)| - |s(B, D)|}{2|q_1|}$$

Note: This result is an important result in the study of radiative heat transfer. It is known as Hottel's crossed strings method.

Problem J. Return of Puzzle Boy

Let a_i be the number of letters that haven't been used yet in index i for any string. We can now convert this to a game on m pile of stones, where the i -th pile has a_i stones. Creating a new string is equivalent now to picking at least 1 pile but less than m piles and removing a single stone from each of those piles.

If all piles have an even number of stones, the game is in a losing state. This is because no matter what subset of piles is chosen, the other player can pick the same piles to bring the game back to all even stones, until all piles are empty and the game ends.

Since a player cannot choose all piles to remove stones from, all piles having an odd number of stones is also a losing state. This is because the other player can choose the complement of the stones chosen, which results in all piles having an even number of stones.

If the parity of all piles of stones is not the same, the current player can either choose all non-zero even piles or all odd piles to force their opponent into a losing state. Thus, a game is in a losing state if and only if the parity of all piles of stones is equal.

If Ben is in a winning state, we need to count the number of words he can form. Let $e(a)$ be the set of indices where a_i is even. If Ben chooses even piles:

- For each index, i in $e(a)$ has a_i options for letters he can place at the i -th index of the new string.
- For each index, i not in $e(a)$ has $c - a_i$ options for letters he can place at the i -th index of the new string.

A similar argument can be made for Ben choosing odd piles. This means that the total number of moves Ben can make is:

$$\left(\prod_{i \in e(a)} a_i \right) \left(\prod_{i \notin e(a)} c - a_i \right) + \left(\prod_{i \in e(a)} c - a_i \right) \left(\prod_{i \notin e(a)} a_i \right)$$

Problem K. Trapping Rain Water

We can break this problem into two parts: what cells can retain water, and where will the water flow. To determine if a cell can retain water, we first must compute the minimum max height of any path from the cell to the outside border. Let this value be $h_{i,j}$. This can be computed using a modified version of Dijkstra's algorithm where we start by populating a min-heap priority queue with any empty border cell, where $h_{i,j} = n - i$ for these cells. When processing the top of the priority queue, we can then consider each neighbor of the current cell, (i', j') and try updating $h_{i',j'}$ to $\max(h_{i,j}, n - i')$. Any cell with $h_{i,j} > n - i$ can retain water.

We can then simulate the flow of water using a BFS. For the current cell, if the cell beneath is empty, let the water flow down. If the cell beneath is a wall, or the cell beneath us can retain water, let water flow to the left and right. Finally, if the current cell can retain water, let water flow upwards. When we enter a cell that can contain water, update the grid.

This solution has a total time complexity of $O(nm \lg nm)$, however many linear time solutions exist as well.

Problem L. Trimming Hedges

If all hedges are already the same height, then 0 operations are needed. Otherwise, we will trim every index down to the minimum pre-existing height. This can take at most two operations because we could first select all even indices, then all odd indices which covers the entire array. So we just need to check if one operation is needed which can be done by checking if one arithmetic progression (with common difference larger than 1) can cover all indices that have a height larger than the minimum. To do this, we can take the GCD of the gap sizes between these indices. If it is larger than 1, then the answer is 1. Otherwise it is 2.