Problem Tutorial: "Easy Problem"

We are asked to find maximum flow in some convex bipartite graphs (node in one part connected to segment in other part). First, there exists online $O(n \log^2 n)$ algorithm for doing so `https://www.researchgate.net/publication/220975392_Dynamic_Matchings_in_Convex_Bipartite_Graphs`. We will use different approach that doesn't use any ideas from this paper. We will use matching terminology, since its equivalent.

There exists simple greedy algorithm of finding maximum matching with fixed feeders. Go left to right, maintain open feeders, for current chicken use feeder that closes earliest. We will do this algorithm right to left, but implicitly.

Lets do scanline on chicken representative x left to right. We will maintain maximum matching to the right of x by maintaining how much grains goes to right of x from each feeder. Then for each feeder we will know how many grains will be left for chickens left of x, for this kind of task (each feeder ends in x), maximum matching can be found with segment tree (maintain minimum prefix balance).

Lets fix x, we build matching greedily right to left. For each feeder (l, r) we maintain size of matching to the right of x. We have operations of adding a feeder, starting at x, and moving x to the right. After adding feeder (x, r), some feeder (possibly (x, r) itself) can have surplus of grains, which we need to subtract from right matching, therefore adding it to left matching. First candidate to check is feeder with lowest priority, priority in this case is simply left end of feeder (therefore (x, r) has highest priority).

Consider array right of x. Let w_i equal to sum of all feeders that end at i minus a_i. Calculate prefix sums p_i on array w_i. Then there is the longest prefix with surplus, say p_o (note that surplus prefixes are records of p and p_o is rightmost record, which is simply maximum of p). Then all feeders that end right of o dont have surplus, meaning all their grains are eaten to the right of x (in fact to the right of o). On the other hand, removing one grain from any feeder left of o doesn't change size of matching to the right of x, therefore we need to find a feeder with lowest priority, that ends left of o, and subtract some surplus from it.

Consider array p, add 0 to the beginning of it. Smallest prefix with surplus is leftmost record, which is first position where $p_i > 0$ (denote it as i_1), next suprlus prefix is leftmost position, where $p_i > p_{i_1}$ and so on, rightmost surplus prefix is p_{i_k} (global maximum on p). Let $i_1, i_2, \dots, i_k$ be positions of records of p, they are surplus prefixes as well. Then if feeder with smallest priority left of i_k ends in r, and $i_t < r$ is rightmost record left of r, then surplus of this feeder equals to $p_{i_k} - p_{i_t}$. After subtracting its surplus we need to do this process again until there is no surplus to the right of x, that is when maximum on p is 0.

Now lets prove that its not possible to subtract surplus for too long. We will prove that the number of operations is bound by $O(n \log n)$, however we didn't come up with example that achieves more than linear amount.

Rewrite our problem, say we are given an array a and we build prefix sums p on it. Initially $a_i = p_i = 0$. Say we have two type of operations

- $a_i = a_i + w$ This operation is performed n times
- Let p_t be global maximum on p, choose $j < t$, let p_k be global maximum on prefix j, set $a_j = a_j - (p_t - p_k)$. This effectively makes p_t equal to p_k, making p_k new global maximum and cutting records of p that are right of k.

We need to prove that number of second operations is $O(n \log n)$ where n is number of operations of first type.

To do that, build segment tree on array p, for a non-leaf node of tree set 1 if maximum on this segment is in right son, otherwise set 0 (if equal, we assume maximum is in left son). We will estimate sum of this values, initially its 0. First operation changes this value only for nodes on path from root to respective index i (in other nodes either all values change by w or dont change at all). When doing second type operation, find node that divides positions k and t. $p_t > p_k$ before this operation, and after this operation $p_t = p_k$, so value in this node changes from 1 to 0. In total, second type operations is bound by $O(n \log n)$.

Total time complexity is $O(n \log^2 n)$, since we need one segment tree to maintain surplus on the right and one segment tree to maintain matching on the left.

Problem Tutorial: "Standard Problem"

First lets compute the maximum weight of a good subsequence. Define f_i — maximum weight of a subsequence, whose lexicographically minimum integer sequence ends with i. Initially $f_i = 0$ for $0 \le i \le m$. When we add segment $[l; r]$ with weight c we need to do

- $f'_l = \max_{j \le l} f_j + c$
- $f'_j = f_j + c$ for $l < j \le r$

Note that we actually need to update $f'_{l<j\le r}$ using prefix maximums as for f'_l, however doing it only for f'_l will preserve prefix maximums in array f', which is enough.

Now lets calculate number of maximums as well. For each subsequence of segments with maximum weight, we will count its occurence only for minimum i as in array f. We will store array of pairs $f_i = (v_i, c_i)$ — weight v_i, number of subsequences of weight v_i is c_i. Initially $f_0 = (0, 1)$, $f_{i>0} = (0, 0)$.

- $f'_l = \max_{j \le l} f_j$, $f'_l[v] += c$
- $f'_j = f_j$, $f'_j[v] += c$ for $l < j \le r$

The only difference is that $\max_{j \le l} f_j$ is calculated not just as the maximum value, but also as the number of its occurences. Again, we should update f'_j with prefix maximums, but its easy to see that for those f'_j that need to be updated from some $f_{i<j}$ its true that $f'_j[c]$ will be 0, so updating it doesn't change final result.

This can be done with segment tree, complexity $O(n \log n)$.

Problem Tutorial: "Network Transfer"

Let's process all events in the order of increasing time and maintain array f_i — when will i-th transfer finish if the throughput for given transfer doesn't change.

Now suppose that next transfer starts at the moment t_0, has priority p_0 and size s_0. Denote P as the current sum of prioirities. Now, the remaining size of i-th transfer is equal to $(f_i - t_0) \cdot \frac{p_i}{P} w$ — the product of time and available throughput. New throughput will be $\frac{p_i}{P+p_0} w$, so new finish time is $f'_i = t_0 + \frac{(f_i - t_0) \cdot \frac{p_i}{P} w}{\frac{p_i}{P+p_0} w} = \frac{P+p_0}{P} f_i - \frac{p_0}{P} t_0$. In other words, you need apply some linear function to all f_i, the same for all i. So we need a data structure which supports 3 operations:

- Apply linear function $f(x) = kx + b$ to all values.
- Get minimum value and possibly erase it. This is for comparing closest transfer finish with next starting event to decide which one is earlier.
- Insert new value.

One can notice that it is enough to keep all values in a priority queue and store linear function separately. Then first operation is just calculating a composition between old linear function and new one. Third operation is applying inverse function to the value and adding result to the priority queue. And second operation is working with minimum element in a priority queue, since linear function with positive slope doesn't change the relative order of elements.

The total complexity is $O(n \log n)$.

Problem Tutorial: "Hard Problem"

Slow solution: loop over segments $[l; r]$, let $m = \frac{l+r}{2}$. We are interested in maximum on $[l; m]$ (let it be L), maximum on $[m+1; r]$ (let it be R), and central position m. Segment is good if $|L - R| \le k$. Write down triples (L, R, m) over all good segments. Key claim is that there are only $O(nk \log n)$ valid triples (valid means there exists corresponding segment $[l; r]$). We will prove it later. Now we need to find all valid triples, then loop over them and add $(a_m + 10) \cdot \sum_{i=p}^{q} f_i$ for some $p \le q$ which is easy to find with stack.

We will prove the claim for the case when a is permutation, proof for general case can be generalized as well.

First lets provide an algorithm of finding all triples. Let $L < R$ (solve twice, for original and reversed array). Loop over position that will correspond to L, say i. Let nearest from left bigger than a_i element be on position L_1. There are no more than k candidates (with values $L+1, L+2, \ldots, L+k$) for value R, we can find their positions with stack, let them be $p_1, p_2, \ldots, p_k$. We can find pairs (L, R) in $O(nk)$.

Fix a_{p_j} as R, lets find which m are valid. There are 4 conditions on m:

- $i \le m < p_1$
- $m < \frac{i+p_{j+1}}{2}$ [otherwise if m is to the right, then because m is the middle of a segment, if segment contains i, then it must contain p_{j+1}]
- $m \ge \frac{L_1+p_j}{2}$ [same reasons, if m is to the left and contains p_j, then it must contain L_1]

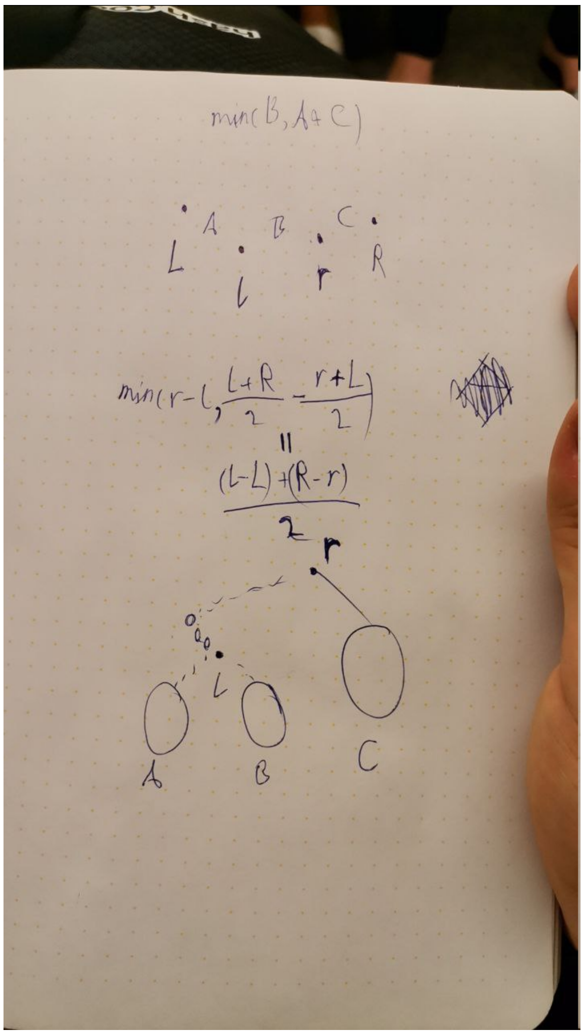
Each condition is necessary and together they are sufficient, so the set of valid m for $L = a_i$ and $R = a_{p_j}$ is a segment constructed as intersection of these conditions. Description of algorithm is finished.

Now back to proof. Divide 4 conditions in 2 segments: $[i; p_1]$ and $[\frac{L_1+p_j}{2}; \frac{i+p_{j+1}}{2}]$. Turns out, if we loop over smaller of the two segments, we can prove a complexity similar to small to large on tree.

First lets do that for $k = 1$. Let $l = i$ (position of left maximum), $r = p_1$ (position of right maximum), $L = L_1$, $R = p_2$. Then $l \le m \le r$ and $\frac{L+r}{2} \le m \le \frac{l+R}{2}$.

Estimating minimum, $\min(r - l, \frac{(l+R)-(L+r)}{2}) \le \min(r - l, (l+R) - (L+r)) = \min(r - l, (l - L) + (R - r))$. Build cartesian tree for maximum on array a — root of the tree is global maximum in array, subtrees constructed recursively from subsegments.

Then segment $(L; l]$ will be left subtree of l, segment $[l; r)$ will be right subtree of l. Minimum can be rewritten as $\min(B, A + C)$ (see image).



If $B \leq A$, then B is chosen and B is smaller son of l (small to large is satisfied). Otherwise $B > A$, lets add A to estimating minimum (since A is smaller son), and subtract it from $A+C$, leaving $\min(B, C)$. Notice that $\min(B, C)$ is not greater than smaller son of r, so for r small to large is satisfied. Now notice that node l is considered once, node r is considered no more than once, since its required that $a[l] \geq a[r] - 1$ (because difference of maximums is no more than k). Proof is easily generalized for $k > 1$, l is considered k times, r is considered no more than k times because we are considering permutation.

Proof can be generalized for non permutation arrays.

Problem Tutorial: "String Strange Sum"

If you know Russian language you can read the bachelor thesis of Ivan Safonov (https://www.hse.ru/en/ba/ami/students/diplomas/624888932) about borders of substrings. The solution of this problem is described in chapters 7.4.3 - 7.4.4 with all proofs and details.

Let's define $hook(l, r)$ as the minimal $i \leq l$, such that $s[i, l-1]$ can be divided into prefixes of $s[l, r]$.

We can find the sum of $hook(l, r)$ instead of the sum of $f(l, r)$.

Let's note, that $hook(l, r) \leq hook(l, r-1)$. In what cases $hook(l, r) < hook(l, r-1)$?

Let's note, that if $hook(l, r) < hook(l, r-1)$ when $s[hook(l, r-1) - (r-l+1), hook(l, r-1) - 1] = s[l, r]$ $(*)$. Also $s[l, r]$ is unbordered substring (the string without equal prefixes and suffixes).

The condition $(*)$ is the criteria of $hook(l, r) < hook(l, r-1)$. Let's note, that if $(*)$ holds, the value $hook(l, r) = hook(hook(l, r-1) - (r-l+1), r)$ $(**)$.

Let's iterate r from 1 to $|s|$ and store an array of values $hook(l, r)$. Let's change our array when $r-1 \to r$. Let's find an array of positions C^r, such that $hook(l, r) < hook(l, r-1)$. If we will find C^r we can sort it, iterate all l from it in increasing order and using the formula $(**)$ recalculate correct values of $hook(l, r)$.

How to find C^r? Firstly, let's add all positions $l \geq r - \lfloor\sqrt{n}\rfloor$ into it.

Now we should find all $l < r - \lfloor\sqrt{n}\rfloor$, such that $hook(l, r) < hook(l, r - 1$. We know, that the substring $s[l, r]$ is unbordered, has the length $> \sqrt{n}$ and appears at least twice in the string s: the first occurence ends in the position $hook(l, r - 1) - 1$, the second ends in the position r.

Let's build a suffix array for the reversed string s. Let's note, that the total numbers of occurences of $s[l, r]$ into s is at most $\frac{n}{r-l+1} \leq \sqrt{n}$, since $s[l, r]$ is unbordered. So, the positions r, $hook(l, r - 1) - 1$ has the distance at most $\sqrt{n}$ in the suffix array.

Let's iterate all possible $t = hook(l, r - 1) - 1$. There are at most $\sqrt{n}$ such values t. Now we should find all l, such that $hook(l, r - 1) = t + 1$ and $hook(l, r - 1) > hook(l, r)$. It is equivalent to $lcs(t, r) \geq r - l + 1 \Rightarrow l \geq r + 1 - lcs(t, r)$ (by the $(*)$).

So we should add all l, such that $hook(l, r - 1) = t + 1$ and $l \geq r + 1 - lcs(t, r)$.

Additionally to the array of $hook(l, r)$ for each value x let's store the list $L_{x,r}$ of the positions l, such that $hook(l, r) = t$ and $l < r - \lfloor\sqrt{n}\rfloor$. Also for each list let's store the value $\max(L_{x,r})$.

Now using these lists we can find possible l: let's firstly check, that $\max(L_{t+1,r-1}) \geq r + 1 - lcs(t, r)$ (it means that at least one possible l exists for such t). If it is true let's just iterate all possible elements of $L_{t+1,r-1}$ and add elements $l \geq r + 1 - lcs(t, r)$ to C^r. During this iteration let's remove all such l from $L_{t+1,r-1}$ (it can be made by constructing the new list, because we iterate all elements of $L_{t+1,r-1}$). Our complexity for exact value of t is $O(|L_{t+1,r-1}|)$.

There is a statement, that I will leave without the proof: the total sum of $|L_{t+1,r-1}|$ during this algorithm is $O(n\sqrt{n})$. It is true, if we will iterate only sets for which at most one suitable value l exists.

This statement means, that our construction of C^r part of the algorithm runs in $O(n\sqrt{n})$ and the sum of $|C^r|$ is $O(n\sqrt{n})$. We can sort C^r with count sort in $O(|C^r| + \sqrt{n})$ time (or just use the sort with log factor, this solution must pass). After iterating C^r and calculating the correct values of $hook(l, r)$ we should add these l into the lists $L_{hook(l,r),r}$.

Time complexity is $O(n\sqrt{n})$.

Challenge:

Let's define $\xi(s)$ as the number of (l, r), such that $hook(l, r) < hook(l, r - 1)$. We proved, that $\xi(s) = O(n\sqrt{n})$ but this bound seems to be not strict and maybe cannot be achieved.

1. Can you construct an example with $\xi(s) = \omega(n)$.
2. (here I don't know the answer) What is the correct bound for $\xi(s)$?

Problem Tutorial: "Bayan Testing"

Let's consider some integer d and periodic array $\{1, 2, \ldots, d, 1, 2, \ldots, d, \ldots\}$ with period d. For this array for any subsegment of length at most d the answer is negative (all elements are different), for others is positive.

Of course, it is not always possible to select such d, that exactly m segments have length at most d, so let's upgrade this array a bit.

Let's sort all segments by their length (in case of equal length by left bound). Let d be the length of $(m + 1)$-st segment, r be it's right bound.

- If $d = 1$, the answer is impossible, because there are $\geq m + 1$ segments of length 1, for them the answer will be always negative.
- If $d \geq 2$ the answer always exists. Let's consider an array $\{1, 2, \ldots, d, 1, 2, \ldots, d, \ldots\}$. From the position r let's change the period from d to $d - 1$. It means, that $a_i = a_{i-d}$ for $i < r$ and $a_i = a_{i-d+1}$

for $i \geq r$. For such array the answer for the first m segments in the sorted order will be negative, for others will be positive.

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

Problem Tutorial: “Battleship: New Rules”

For each ship let’s consider it’s right/bottom corner. By this operation ships of sizes $1 \times a$, $a \times 1$ will be changed to strips of sizes $2 \times (a+1)$ and $(a+1) \times 2$ respectively.

Let’s note, that each pair of strips do not intersect. Also these strips lie at the square of size $(n+1)\times(n+1)$ (the board with its right/bottom corner).

Let s be the sum of ships lenghts. The sum of strips areas is equal to $2(k+s)$. So, $2(k+s) \leq (n+1)^2$.

By the statement we select positions of ships in such way, that s is maximum possible. It can be proved, that with condition $n \leq k \leq \lceil \frac{n}{2} \rceil^2$ the maximum value $s = \lfloor \frac{(n+1)^2}{2} \rfloor - k$.

Let’s note, that if all cells of 2×2 square are covered with strips, there exists a cell covered with ship in this square.

So:

- If n is odd, $2(s+k) = (n+1)^2$ and all cells of $(n+1) \times (n+1)$ square are covered with strips. So the answer is always -1.
- If n is even, $2(s+k) = (n+1)^2 - 1$ and there exists exactly one cell (x_0, y_0), such that it is not covered with strips. It can be noticed, that the 2×2 square with cells (x_0-1, y_0-1), (x_0, y_0-1), (x_0-1, y_0), (x_0, y_0) is empty and it is the only empty 2×2 square.

So, n is even and to solve the problem we should find the only cell (x_0, y_0) not covered with strips.

Let’s use a divide and conquer algorithm. During the algorithm we will color some cells, covered with strips.

We will store the angles of the rectangle (x_1, y_1), (x_2, y_2), such that $x_1 \leq x_0 \leq x_2$, $y_1 \leq y_0 \leq y_2$. Values $x_1 = y_1 = 1$, $x_2 = y_2 = n+1$, initially.

Let’s divide the longest side of the rectangle into two parts. WLOG $x_2 - x_1 \geq y_2 - y_1$. Let $x_{mid} = \frac{x_1+x_2}{2}$. Let’s ask all cells $(x_{mid}-1, y)$, (x_{mid}, y) with $y_1 \leq y \leq y_2$. We will ask $2(y_2 - y_1 + 1)$ queries. If we will find an empty 2×2 square — finish the algorithm immediately. Otherwise for each found cell (i, j), such that (i, j) is covered with ship let’s color cells (i, j), $(i+1, j)$, $(i, j+1)$, $(i+1, j+1)$ (they are covered with strips). Let’s note, that we will color all cells (x_{mid}, y) $(y_1 \leq y \leq y_2)$.

Let’s consider two parts of the rectangle, for which it was divided. For each of them let’s calculate the number of uncolored cells inside it. Let’s note, that during the algorithm we colored some parts of strips of sizes $2 \times b$, so the total number colored cells is even, so the total number of uncolored cells is odd. So, in one of the parts the number of uncolored cells will be odd. Let’s note, that the cell (x_0, y_0) should be in this part, so we can divide the longest side of the rectangle by 2 and continue the algorithm.

The maximum number of queries we will ask is equal to $2(n + \frac{n}{2} + \frac{n}{2} + \frac{n}{4} + \frac{n}{4} + \ldots) = 6n$.

Problem Tutorial: “Triangular Cactus Paths”

Let’s consider all simple paths between vertices s, f. How they look like? Let’s consider the shortest path. It’s easy to prove, that it is unique (because in our cactus all cycles have odd length). There are some triangles, for which an edge lies on this shortest path. We can change this edge into two other edges on this triangle. Applying this operation multiple times we will get all simple paths between s and f. So, if the length of the shortest path is l, the number of triangles is t, the number of simple paths with length k is $\binom{t}{k-l}$.

To find the length of the shortest path and the number of triangles for each pair, let's build a tree from our graph: let's consider each cycle between vertices (a, b, c), remove edges between them, add new vertex v to the graph, connect it to a, b, c. As a result we will get the tree with $m+1$ vertices. Let's write a pair $(1, 0)$ on edges, that were in the initial graph, a pair $(1, 1)$ on edges, that were added. Let's build the structure on the tree for calculating the sum of pairs on paths (for example, with binary liftings).

For two vertices s, f let the pair (a, b) be the sum of pairs on the path between them. The number $\frac{b}{2}$ is equal to the number of triangles, the number $a - \frac{b}{2}$ is equal to the length of the shortest path.

Time complexity: $O((m + q) \log m)$.

Problem Tutorial: "Best Sun"

Required tricks:

- Binary search by the answer.
- Radewoosh trick (https://codeforces.com/blog/entry/62602).
- *dp* by angle to find the convex polygon (with fixed lowest point).

Let's fix the lowest point of the sun s and make a binary search. We want to check if there exists a sun with lowest point A_s, such that $S - xP \geq 0$ for given x.

Let's make a standart *dp*. Let's sort all pairs of points (A_i, A_j), such that the triangle $A_sA_iA_j$ doesn't contain other points. Pairs will be sorted by angle of vector $A_j - A_i$. Let dp_p equal to the maximum score of the path from s to p ($dp_p = -\infty$ initially). After that we iterate pairs (i, j) in the sorted order and update:

$$dp_j = \max\left(dp_j, dp_i + area(A_sA_iA_j) - x|A_iA_j|\right)$$

Pairs (i, j) can be sorted before iteration of s and the binary search once, so the complexity of this check is $O(n^2)$. At the end we should check, that $dp_s \geq 0$.

But this solution does not take into account points outside of the sun. How we could add them? For each point A_p outside of the sun let's consider it's projection to the perimeter of the sun.

- If this projection is on some side A_iA_j, it is easy to add such points into *dp*. For each pairs of points A_i, A_j let's calculate the sum of $\min(|A_iA_p|, |A_jA_p|)$, for all p such that the projection of A_p to the line A_iA_j lies on the segment A_iA_j and A_p lies on the right side to the vector from A_i to A_j. After that just change $|A_iA_j|$ to the $|A_iA_j| + sum_{i,j}$ in *dp* formula.
- This projection is some point A_i. To add this case to our *dp* let's add events corresponding to it. Let's add a second type of events for every pair (i, p). In the moment $angle(A_p - A_i) + \frac{\pi}{2}$ let's add this event. To update *dp* for event (i, p) of this type we should make:

$$dp_i -= x|A_iA_p|$$

The time complexity of one check is still $O(n^2)$.

Now let's iterate all s in random order and make a binary search only if the answer can be increased on this step (it can be checked with one check). The total number of checks will be $O(n + \log n \log A)$.

So, the total time complexity is $O(n^3 + n^2 \log n \log A)$.

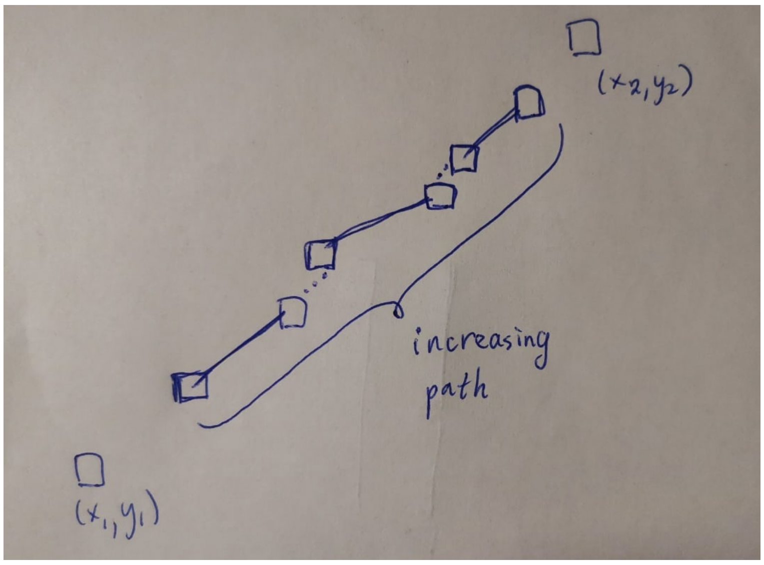
Problem Tutorial: "Fast Bridges"

Let's find how the shortest path between cells (x_1, y_1), (x_2, y_2) can look like.

If $x_1 = x_2$ or $y_1 = y_2$ it's length is still $|x_1 - x_2| + |y_1 - y_2|$.

WLOG $x_1 < x_2$, $y_1 < y_2$.

It can be proved, that the manhattan path can be optimized with *increasing path* — the sequence of fast bridges where each next fast bridge is located upper right to the previous. Let's define $f(x_1, y_1, x_2, y_2)$ as the length of the longest increasing path between (x_1, y_1), (x_2, y_2).



The shortest path between (x_1, y_1), (x_2, y_2) has length $|x_1 - x_2| + |y_1 - y_2| - f(x_1, y_1, x_2, y_2)$.

So, the answer is equal to the sum of manhattan distances between all pairs of cells (it is equal to $n^4(n+1) - \frac{n^3(n+1)(2n+1)}{3}$ minus the sum of $f(x_1, y_1, x_2, y_2)$ for all pairs of cells.

The sum of $f(x_1, y_1, x_2, y_2)$ can be found for pairs of cells with $y_1 < y_2$ and $y_1 > y_2$ separately.

Now we should calculate the sum of $f(x_1, y_1, x_2, y_2)$ between all pairs of cells (x_1, y_1), (x_2, y_2) such that $x_1 < x_2$, $y_1 < y_2$.

Let's find $dp_{i,j}$ as the length of the longest increasing path starting with the fast bridge i, ending with the fast bridge j. It can be found easily in $O(n^3)$.

Let's compress all coordinates of left down cells of bridges and iterate over cells in compressed grid from right to left from up to down. During the iteration let's calculate the value $f_{x,y,i}$ — the length of the longest increasing path with x coordinates $\geq x$, y coordinates $\geq y$ ending with the fast bridge i. They can be calculated as a maximum from $f_{x+1,y,i}$, $f_{x,y+1,i}$ and values $dp_{t,i}$, such that $x_{1,t} = x$, $y_{1,t} = y$. It's easy to see, that only the last layer of f can be maintained, so we need only $O(n^2)$ memory. The total calculation time will be $O(n^3)$.

During the iteration let's calculate the sum of $f(x_1, y_1, x_2, y_2)$, such that (x_1, y_1) is located in the compressed grid cell (x, y). Let's note, that for all $x_2 \geq x_{2,i}$, $y_2 \geq y_{2,i}$ the value $f(x_1, y_1, x_2, y_2) \geq f_{x,y,i}$. Let's find the area of union of rectangles $[x_{2,i}, k] \times [y_{2,i}, k]$ with $f_{x,y,i} = t$. This area is equal to the number of $f(x_1, y_1, x_2, y_2) \geq t$. So we should just sum these areas for all t.

To find the area of union for each value t we can iterate all points $(x_{2,i}, y_{2,i})$ in increasing order (by x) and store the lowest previous point (with smallest y). Area can be recalculated in $O(1)$ easily by adding the new point. We can sort all points $(x_{2,i}, y_{2,i})$ before the algorithm, so this part of the solution is linear for all compressed grid cells (x, y).

The total time complexity is $O(n^3)$, memory complexity is $O(n^2)$.

Problem Tutorial: "Decoding The Message"

We should find the answer by modulo $255 \cdot 257$. Let's find the answer by modulo 255 and by modulo 257 separately and merge them by Chinese Remainder Theorem.

Finding the answer by modulo 255 is simple: it is equal to $(d_1 + \ldots + d_n)^{n!}$.

Now let's solve the problem by the prime modulo 257. Let's consider a number $\overline{d_{p_{n-1}} \ldots d_{p_1} d_{p_0}}$. By modulo 257 it is equal to $\sum\limits_{2|i} d_{p_i} - \sum\limits_{2\nmid i} d_{p_i}$.

Let's define S as a set of indices on odd positions. The size $|S| = \lfloor \frac{n}{2} \rfloor$. The number by modulo 257 will be equal to $f(S) = \sum\limits_{i=0}^{n-1} d_i - 2\sum\limits_{i\in S} d_i$. For each set S there are exactly $\lfloor \frac{n}{2} \rfloor ! \lceil \frac{n}{2} \rceil !$ permutations. So the answer is equal to the multiply of $f(S)$ for all sets S in power $\lfloor \frac{n}{2} \rfloor ! \lceil \frac{n}{2} \rceil !$.

If $n \le 11$, let's find the multiply iterating all possible subsets S in exponential time.

It $n \ge 12$, the number $\lfloor \frac{n}{2} \rfloor ! \lceil \frac{n}{2} \rceil !$ is divisible by $256 = \phi(257)$. So the answer will be always 0 or 1.

The answer is 0 if there exists set S of positions of size $\lfloor \frac{n}{2} \rfloor$, such that $f(S)$ is divisible by 257. It means that $sum(S) = \sum\limits_{i\in S} d_i \equiv \frac{\sum\limits_{i=0}^{n-1} d_i}{2} \pmod{257}$.

If it is possible to select 256 pairs of different digits (d_{x_i}, d_{y_i}) $(d_{x_i} \ne d_{y_i})$, such set always exists. To prove it let's select any set S_0 of positions with size $\lfloor \frac{n}{2} \rfloor$, such that all $x_i \in S_0, y_i \notin S_0$. After that, by replacing the digit d_{x_i} in S_0 to d_{y_i} we will add $d_{y_i} - d_{x_i}$ to $sum(S_0)$. All differences $d_{y_i} - d_{x_i}$ are non-zero. There are 256 differences, so every remainder by modulo 257 (including $-sum(S_0)$) can be made as a sum of subset of differences.

Let's sort all digits by their counts. So $c_{i_1} \le c_{i_2} \le \ldots \le c_{i_k}$.

So the answer is 0 if $\sum\limits_{j=1}^{k-1} c_{i_j} \ge 2 \cdot 256$.

Now we have $\sum\limits_{j=1}^{k-1} c_{i_j} \le 2 \cdot p$ $(p = 257)$.

Let's use knapsack with bitsets to find all pairs (b, s), such that there exists subset of digits $\{i_1, i_2, \ldots, i_{k-1}\}$ of size b with sum s. After that for all such pairs with $b + c_{i_k} \ge \lfloor \frac{n}{2} \rfloor$ let's check that $s + (\lfloor \frac{n}{2} \rfloor - b) i_k \equiv \frac{\sum\limits_{i=0}^{n-1} d_i}{2} \pmod{257}$.

Time complexity: $O(\frac{p^3}{64})$ per test case.