Problem Tutorial: "Classical A+B Problem"

Let $|x|$ denote the number of digits in integer x.

In $n = a + b$, without loss of generality, let $a \geq b$. Then it is easy to see that $|n| - 1 \leq |a| \leq |n|$.

Thus, there are just 2 options for the length of a, and 9 options for the digit a consists of, which makes it just 18 options for a in total. We can try each of them, calculate $b = n - a$ and check if b is a repdigit.

Problem Tutorial: "Classical Counting Problem"

Without loss of generality, suppose that $a_1 \geq a_2 \geq \ldots \geq a_n$.

Let's fix a subset of problems S and try to devise criteria that will tell us whether this subset can be chosen as a problemset.

Let x be the lowest-numbered problem that does not belong to S, and let y be the highest-numbered problem that belongs to S. It means problems $1, 2, \ldots, x-1$ belong to S for sure, problems $y+1, y+2, \ldots, n$ do not belong to S for sure, and problems $x+1, x+2, \ldots, y-1$ either belong to S or not.

If $a_y + m < a_x$, then S is not a valid subset.

Otherwise, let l_i be the smallest number of votes problem i must get, in order for S to be able to become the chosen problemset. For $i \in \{1, 2, \ldots, x-1, x, y+1, y+2, \ldots, n\}$, we have $l_i = 0$. For problem y, we have $l_y = a_x - a_y$. For any other problem $i \in \{x+1, x+2, \ldots, y-1\}$, if it belongs to S, then $l_i = a_x - a_i$; otherwise, $l_i = 0$.

Similarly, let r_i be the largest number of votes problem i can get, in order for S to be able to become the chosen problemset. For $i \in \{1, 2, \ldots, x-1, y, y+1, y+2, \ldots, n\}$, we have $r_i = m$ (note that no problem can get more than m votes). For problem x, we have $r_x = (a_y + m) - a_x$. For any other problem $i \in \{x+1, x+2, \ldots, y-1\}$, if it belongs to S, then $r_i = m$; otherwise, $r_i = (a_y + m) - a_i$.

Note that $l_i \leq r_i$ for any i.

Claim: if $\sum\limits_{i=1}^{n} l_i \leq m \cdot v \leq \sum\limits_{i=1}^{n} r_i$, then S can be chosen as a problemset.

Proof: first, find numbers of votes $b_1, b_2, \ldots, b_n$ such that $l_i \leq b_i \leq r_i$, $\sum\limits_{i=1}^{n} b_i = m \cdot v$, and set S can be chosen. For example, you can start with $b_i = l_i$ and increase the number of votes for problems in S (up to r_i for each i), and once you're done with those, start increasing the number of votes for problems not in S, until the total becomes $m \cdot v$. Then, write down a sequence of integers: b_1 copies of 1, b_2 copies of 2, …, b_n copies of n. This sequence has length $m \cdot v$. Let judge 1 vote for problems on positions $1, m+1, 2m+1, \ldots, (v-1)m+1$ in this list; let judge 2 vote for problems on positions $2, m+2, 2m+2, \ldots, (v-1)m+2$; and so on. You can see that every judge votes for exactly v distinct problems, and every problem i gets exactly b_i votes.

Now, to count possible sets S, one can count sets satisfying $\sum\limits_{i=1}^{n} r_i \geq m \cdot v$, and subtract the number of sets satisfying $\sum\limits_{i=1}^{n} l_i > m \cdot v$.

One way to calculate both numbers is to iterate over problems x and y and run a knapsack-like DP for problems $x+1, x+2, \ldots, y-1$, e.g. $f(i, s)$: how many ways are there to choose a subset of problems $x+1, x+2, \ldots, i$ so that the sum of l_j (or r_j) for the problems in this range is s. This DP has $O(n^2 \cdot a_n)$ states, $O(1)$ transitions from each state, and we need to run it $O(n^2)$ times, for all x and y. Thus, the time complexity of this solution is $O(n^4 \cdot a_n)$, which might be too slow.

To optimize it, note that the values of l_i only depend on x. Hence, to count sets satisfying $\sum\limits_{i=1}^{n} l_i > m \cdot v$, it's enough to fix x, but there is no need to fix y: we can just run the DP forward for $i = x+1, x+2, \ldots, n$.

Similarly, since the values of r_i only depend on y, it's enough to fix y to count sets satisfying $\sum\limits_{i=1}^{n} r_i \geq m \cdot v$, and run the DP backwards for $i = y-1, y-2, \ldots, 1$.

This way, the time complexity improves to $O(n^3 \cdot a_n)$.

Problem Tutorial: "Classical Data Structure Problem"

This problem looks like a standard dynamic segment tree problem.

Initially, you create a single node corresponding to segment $[0; 2^m)$.

Whenever a query $[l; r]$ happens, let's say you traverse the tree from top to bottom and arrive at a node that corresponds to segment $[x; y)$ and has no children such that $x < l < y$. Then, you create two new nodes corresponding to segments $[x; \frac{x+y}{2})$ and $[\frac{x+y}{2}; y)$ and continue going down. Similar thing happens to the segment containing r.

This way, you create $O(m)$ new nodes per query, resulting in $O(nm)$ space usage, which might be too much since the memory limit is tight.

Instead of splitting $[x; y)$ into $[x; \frac{x+y}{2})$ and $[\frac{x+y}{2}; y)$, you could also split into $[x; l)$ and $[l; y)$. This way, you will only split once for both l and r, which means you need $O(n)$ space overall since only create at most 4 new nodes per query. However, now time usage can be an issue: the height of the tree might grow to $\Omega(n)$, e.g. if all query segments are nested into each other.

To fix this, note that you can perform rotations to rebalance your segment tree, similarly to how many balanced binary search trees work. This way, you can use the techniques from e.g. AVL-tree or splay tree to achieve $O(n \log n)$ time complexity and $O(n)$ space complexity.

Problem Tutorial: "Classical DP Problem"

For convenience, reverse a so that $a_1 \geq a_2 \geq \ldots \geq a_n$.

Find the side length k of the largest square subgrid that fits inside the grid. Formally, k is the only integer that satisfies $a_k \geq k$ and $a_{k+1} < k+1$. Then the smallest number of rooks needed is equal to k. Why?

- To cover the whole $k \times k$ square, at least k rooks are needed.
- All other cells of the grid belong either to one of the first k rows or to one of the first k columns. Thus, if we place k rooks on the main diagonal of the $k \times k$ square, all cells will be covered.

Now let's focus on counting valid ways to place k rooks. At least one of the following two conditions must be satisfied:

1. Each of the first k rows contains a rook.
2. Each of the first k columns contains a rook.

If neither condition is satisfied, at least one cell inside the $k \times k$ square will not be covered: specifically, any cell that belongs to a row without a rook and a column without a rook. Thus, the answer to the problem is the number of placements satisfying condition 1, plus the number of placements satisfying condition 2, minus the number of placements satisfying both conditions. It's easy to see that the last number is equal to $k!$.

From now on, let's count placements satisfying condition 1: each of the first k rows contains a rook. The number of placements satisfying condition 2 can be found similarly after transposing the grid.

Let $t = a_{k+1}$. Then t is the number of columns that must contain at least one rook to make sure that the whole grid is covered.

Let $f(i, j)$ be the number of ways to place rooks into the first i rows so that exactly j of the first t columns contain at least one rook. As a base case, $f(0, 0) = 1$, and the number we are looking for is $f(k, t)$.

What are the transitions? Say we are at state (i, j), and we want to put a rook into row $i+1$. If the rook belongs to any of $t-j$ columns that need a rook but don't currently have one, we will move to state

$(i+1, j+1)$, and there are $t-j$ ways to do so. Otherwise, the number of columns containing a rook among the first t columns does not change, and we will move to state $(i+1, j)$ in $a_{i+1} - (t-j)$ ways.

The time complexity of this solution is $O(n^2)$.

Problem Tutorial: "Classical FFT Problem"

Improving on the solution to the previous problem: we have k rows containing $a_1 \geq a_2 \geq \ldots \geq a_k$ cells, and we need to find the number of ways to put a rook into every row so that each of the first t columns ($t \leq a_k$) contains at least one rook.

Let's use the inclusion-exclusion principle on the set of columns among the first t columns that contain a rook. Our answer is the number of ways to put the rooks arbitrarily, minus the number of ways to put the rooks so that column i is not covered (for every i), plus the number of ways to put the rooks so that columns i and j are not covered (for every $i < j$), etc.

Since the columns are indistinguishable, we can just iterate on the number of uncovered columns p. The number of ways to put the rooks so that the given p columns do not contain any rooks is $(a_1 - p) \cdot (a_2 - p) \cdot \ldots \cdot (a_k - p)$. The number of ways to choose p columns from the first t columns is $\binom{t}{p}$. Thus, the overall answer is $\sum\limits_{p=0}^{t} (-1)^p \cdot \binom{t}{p} \cdot (a_1 - p) \cdot (a_2 - p) \cdot \ldots \cdot (a_k - p)$.

Let $f(x) = (a_1 - x) \cdot (a_2 - x) \cdot \ldots \cdot (a_k - x)$ be a polynomial. We can find the coefficients of this polynomial in $O(k \log^2 k)$ using divide-and-conquer and FFT, and we can find the values of $f(0), f(1), \ldots, f(t)$ in $O(k \log^2 k)$ using multipoint evaluation. Then, the answer is just $\sum\limits_{p=0}^{t} (-1)^p \cdot \binom{t}{p} \cdot f(p)$.

Problem Tutorial: "Classical Geometry Problem"

Let's define the *rank* $f(p)$ of a point $p = (r, g, b)$ to be the number of its coordinates not equal to either 0 or 255.

If $f(p) = 0$, it means that p is one of the basic colors, and you can just go to p in one move.

If $f(p) = 1$, it means p lies on an edge between two basic colors a and b. You can go to a, then start moving towards b and stop at the right moment.

If $f(p) = 2$, it means p lies inside a face of the cube. Pick any vertex v of this face, draw a half-line from v through p, and find where this half-line intersects a cube's edge. The point of intersection has rank 0 or 1, so you can first go to that point as described above, and then move from that point towards v to arrive at p.

If $f(p) = 3$, it means p lies strictly inside the cube. Pick any cube vertex v, draw a half-line from v through p, and find where this half-line intersects a cube's face. The point of intersection has rank 0, 1, or 2, so you can first go to that point as described above, and then move from that point towards v to arrive at p.

Problem Tutorial: "Classical Graph Theory Problem"

See paper "The even adjacency split problem for graphs", section 3.

Problem Tutorial: "Classical Maximization Problem"

Build a bipartite graph that has a vertex for each value of x-coordinate and a vertex for each value of y-coordinate. Each point (x_i, y_i) becomes an edge that connects two vertices corresponding to $x = x_i$ and $y = y_i$.

In this graph, we need to form the maximum number of non-intersecting pairs of edges that have a common vertex.

For each connected component of the graph, the problem can be solved separately. Now let's show that in a connected component that contains m edges, we can always form $\lfloor \frac{m}{2} \rfloor$ pairs.

Build a depth-first search (DFS) tree of the component. In the component, every edge either belongs to the tree, or goes from an ancestor to a descendant. In other words, there are no cross edges.

Now let's solve the problem recursively from bottom to top. For each subtree of the DFS tree, we will split all the edges inside it into friendly pairs, and we will also use the edge going from the subtree root to its parent if the number of edges inside the subtree is odd.

The recursive function $f(v)$ works as follows. First, call the function $f(u)$ for all children u of the subtree root v. From each u, we will either use the $u - v$ edge or keep it unpaired. Now, form a set S containing all such unpaired edges, and also all edges going down from v to its descendants (not direct children). Note that all edges in S have v as a common vertex. Split all edges in S into pairs arbitrarily, and if the size of S is odd, match the remaining edge with the edge going from v to its parent.

In the end, we will split all edges into friendly pairs as promised, except maybe for one edge incident to the root.

Problem Tutorial: "Classical Minimization Problem"

In this problem, we want to form as many unfriendly pairs as possible. Let k be the maximum number of points belonging to the same vertical or horizontal line. Then, if $k \le n$, we will show that the answer is n. Otherwise, if $k > n$, the answer obviously does not exceed $2n - k$, and we will show that the answer is exactly that.

We will prove these claims using induction on n, forming pairs one by one.

If $k > n$:

- Note that at most one line can contain more than n points; thus, we need to form a pair that uses a point on that line. In that case, both n and k will decrease by 1, and if we form $2(n-1) - (k-1) = 2n - k - 1$ pairs from the remaining points, we will arrive at $2n - k$ pairs in total, as desired.

If $k \le n$, we need to form a pair that uses at least one point on every line containing n points:

- If both a horizontal line with n points and a vertical line with n points exist, the pair should contain a point on one line and a point on the other line.
- If only a horizontal or a vertical line with n points exists, the pair should include a point on that line.
- If no such lines exist, we can form any valid pair.

Note that there could be two horizontal or two vertical lines with n points, but in that case, any valid pair uses a point on both of them, which is fine for us.

Repeat the following algorithm to handle all the cases above:

- Find H, the horizontal line containing the most points among all horizontal lines, and V, the vertical line containing the most points among all vertical lines.
- If H or V contains all $2n$ points, stop. No more pairs can be formed.
- If H contains just 1 point, and that point lies on V, let V' be the vertical line with the second biggest number of points. Form a pair using the only point on H and any point on V'.
- Otherwise, if V contains just 1 point, and that point lies on H, let H' be the horizontal line with the second biggest number of points. Form a pair using the only point on V and any point on H'.
- Otherwise, let h be any point on H and v be any point on V. If h lies on V, change h to any other point on H. Similarly, if v lies on H, change v to any other point on V. (Since both H and V contain at least 2 points, this is always possible.) Then, form a pair using h and v.

For implementation, one could use priority queues to find H and V and arrive at an $O(n \log n)$ time complexity. However, since the number of points on each line is bounded by $2n$, one could also use $2n$ linked lists for horizontal/vertical lines, where list i stores all lines containing i points, and maintain pointers to non-empty lists with the highest number to achieve an $O(n)$ time complexity.

Problem Tutorial: "Classical Scheduling Problem"

Without loss of generality, reorder the topics so that $b_1 \le b_2 \le \ldots \le b_n$.

Let's use binary search on x, the number of topics you will be confident about. How to check whether you can be confident about x topics?

Let $p_1 < p_2 < \ldots < p_k$ be the topics you learn ($k \ge x$). Since the topics are ordered by b, you must be confident in topics $p_1, p_2, \ldots, p_x$. Thus, $k \ge b_{p_x}$ must hold.

Let's iterate over topic i then. Is it possible to have $p_x = i$? In this case, we should choose $x-1$ topics with the smallest a_j from topics $1, 2, \ldots, i-1$, and we should choose $\max(0, b_i - x)$ topics with the smallest a_j from topics $i+1, i+2, \ldots, n$. If the total time needed to learn the chosen topics (including topic i) does not exceed t, we are fine.

To find $x-1$ topics with the smallest a_j on every prefix, we can traverse the array from left to right and use a priority queue. To find $\max(0, b_i - x)$ topics with the smallest a_j on every suffix, we can traverse the array from right to left and use a priority queue as well.

The overall time complexity is $O(n \log^2 n)$ (one log from using binary search on x, the other log from using priority queue).

Problem Tutorial: "Classical Summation Problem"

Let's reorder the cities where the friends live so that $a_1 \le a_2 \le \ldots \le a_k$. Then, if k is odd, the friends will meet in city $a_{(k+1)/2}$, and if k is even, they will meet in city $a_{k/2}$. Let's focus on the expected value of the answer assuming every friend chooses the city to live in uniformly at random, and multiply it by n^k at the end.

When k is odd, due to symmetry, the expected value $E(a_{(k+1)/2}) = \frac{n+1}{2}$. Thus, the answer to the problem is $\frac{n+1}{2} \cdot n^k$.

When k is even, we can not reason about $a_{k/2}$ in the same way, but we can look at the average of two medians, $m = \frac{a_{k/2} + a_{k/2+1}}{2}$, and note that $E(m) = \frac{n+1}{2}$. Let $d = a_{k/2+1} - a_{k/2}$ be the distance between two medians. Then $a_{k/2} = m - \frac{d}{2}$, and $E(a_{k/2}) = E(m - \frac{d}{2}) = E(m) - E(\frac{d}{2}) = \frac{n+1}{2} - \frac{1}{2} \cdot E(d)$.

How do we find $E(d)$? For each $i = 1, 2, \ldots, n-1$, let's find the probability p_i that line segment between cities i and $i+1$ lies between two medians, then $E(d) = \sum_{i=1}^{n-1} p_i$. For that to happen, we need exactly $\frac{k}{2}$ friends to live in cities $1, 2, \ldots, i$, and exactly $\frac{k}{2}$ friends to live in cities $i+1, i+2, \ldots, n$. The probability of this can be calculated as $p_i = (i^{k/2} \cdot (n-i)^{k/2} \cdot \binom{k}{k/2})/n^k$.

The time complexity of this solution is $O(n \log k + k)$.