Problem Tutorial: "Universal Paperclips"

The main step to the solution is to understand how the number of paperclips differs between whole rounds (of n seconds). Let's simulate the first n seconds and calculate the following values:

1. The number of upgrades: call it U
2. The number of clicks: call it C
3. The number of paperclips after the first round: call it P

Let's consider how the second round will compare to the first. All upgrades are still the same. The number of clicks is the same, but every click creates U more paperclips during the second round. Thus, in the second round, we will get $U \cdot C$ more paperclips than in the first.

Finally, we can calculate the total number of paperclips for the first $k = \lfloor \frac{t}{n} \rfloor$ rounds: $P + (P + UC) + (P + 2UC) + \ldots$. You can then simulate the last ($t \bmod n$) seconds by hand.

Problem Tutorial: "Hanoi Chips"

Let's assume that $x_1 \le x_2 \le x_3$ and $y_1 \le y_2 \le y_3$.

Note that all moves are reversible, so if we can transform both the initial position x and the final y into the same state, then we can also get a solution from x to y. It is clear that most likely it will not be possible to obtain any other position from the initial position, and all possible positions will be divided into classes. Let's try to come up with some "canonical" positions for x and y, by which we can quickly understand the answer. If the normal positions coincided, then we got a suitable solution, and if not coincided — let's try to prove that the solution does not exist.

First, let's try to put the chips as close to each other as possible, that is, to minimize $x_3 - x_1$ (this and the next steps will do the same for y). As long as all values are different, we can decrease the difference (if $x_3 - x_2 > x_2 - x_1$, then we will move x_1, otherwise x_3). Note that $\gcd(x_3 - x_2, x_2 - x_1)$ does not change when a chip is moved, so as soon as we get two or more chips at the same point, we have already obtained the minimum difference (since $x_3 - x_2$ or $x_2 - x_1$ must be at least `gcd`).

If we got three chips at one point, then they were all at the same point initially (since all operations are reversible and we cannot move the chips in any way). It is recommended that such a case be handled separately. If we have only two chips at one point, then we can still move this segment (a segment is a pair of points that contains all three chips), and its left end can contain one or two chips. To unambiguously define a canonical position, say, for example, that the left endpoint must contain exactly one chip and its coordinate must be minimal, but greater than 0. To move a segment, we just move one of its endpoints in the desired direction.

By converting x and y to the canonical position using the algorithm above, we can easily build the solution if their canonical positions coincide. It remains to prove that if two positions are different, then the solution does not exist. Indeed, if the lengths of two segments are different, then the `gcd` of the positions is different, and we can't change it in any way. If the left end doesn't match, then there's no way we can shift it a bit, because x_1, x_2, and x_3 have the same remainder modulo `gcd` regardless of moves. And the last case, when two canonical positions differ by exactly `gcd`. We can consider the case when `gcd` is 1 (if it's not the case, we can shift both positions are divide all coordinates by `gcd`). Now notice that any move preserves the parity of the sum of all three coordinates in each position, but $(0, 1, 1)$ and $(1, 2, 2)$ have different parity, so the answer doesn't exist.

Problem Tutorial: "Sorting Subarrays"

Subtask 1

In this subtask, you could do practically anything. For example, try all subsegments, sort them, and put all arrays into a set `std::set` in C++).

General idea

All following subtasks use the following general idea.

To start, count the initial array, it's possible to get it by sorting a subsegment of length 1.

Now we're only interested in operations that change the array. Suppose we sorted subsegment $[l, r]$. If $a_l = \min\limits_{i=l}^{r} a_i$, then sorting $[l, r]$ is the same as sorting $[l + 1, r]$. Similarly, if $a_r = \max\limits_{i=l}^{r} a_i$, then sorting $[l, r]$ is the same as sorting $[l, r - 1]$.

That means, we can only consider subsegments $[l, r]$, where $a_l > \min\limits_{i=l}^{r} a_i$ and $a_r < \max\limits_{i=l}^{r} a_i$. These segments definitely cover all possible results (except for the initial array).

Now let's prove that sorting all such subsegments results in different arrays. Consider two subsegments $[l_1, r_1]$ and $[l_2, r_2]$. There are two possibilities:

1. $l_1 < l_2$ (without loss of generality, this also covers the case $l_2 < l_1$). In the case l_1 doesn't belong to $[l_2, r_2]$. Thus, when we sort $[l_2, r_2]$, a_{l_1} doesn't change. On the other hand, when we sort $[l_1, r_1]$, a_{l_1} changes because it's greater than the minimum on the first segment.
2. $l_1 = l_2$. Without loss of generality, consider $r_1 < r_2$. In this case, r_2 doesn't belong to $[l_1, r_1]$, and a similar argument applies.

In conclusion, we need to calculate the number of segments $[l, r]$, where the left element is not the minimum and the right element is not the maximum.

Subtask 2

Try all possible left endpoints, and move the right. We can keep track of the current minimum and maximum on the current segment, so we can check if the current segment satisfies the aforementioned criterion. The time complexity is $O(n^2)$.

Subtask 3

Calculate the number of segments that start with 2 and end with 1. For every 1, add the number of 2s left to it to the answer.

Full solution

Fix the left endpoint l. Let's find the closest element to the right that's smaller than it. Let's call this index p_l. The condition for the left endpoint will be satisfied when the right endpoint is $\geq p_l$.

Similarly, for every right endpoint r find the closest greater element to the left of it, calling it q_r. The corresponding left border has to be $\leq q_r$.

The arrays p and q can be calculated either using a data structure that can calculate min/max on a segment (segment tree, sparse table, ...), or using an increasing/decreasing stack in linear time.

Now all that's left is to calculate the number of pairs: $l < r$: $l \leq q_r$ and $p_l \leq r$. Notice that $l < r$ is redundant since it's implied by $p_l \leq r$.

Draw points (r, q_r) on a 2D plant. Trying all values of l, we need to add the number of points (x, y), such that $x \geq p_l$ and $y \geq l$. This is a rather standard algorithmic problem about counting points in a rectangle. To answer such queries, we use the sweep line method: sort all points and queries in increasing order of x, and process them in this order. Maintain a 1D data structure (for example, a segment tree). Whenever we encounter a point, add 1 in this data structure at the corresponding y coordinate. When we encounter a query, we ask for the prefix sum until y.

Problem Tutorial: "RestORe"

Let's solve the $n = 1$ case first: find, how many segments $[L, R]$ give the desired bitwise "OR" equal to F. Let's find the longest common prefix of binary representations of L and R: call it P. Then $L = P0\ldots$, and $R = P1\ldots$. Among numbers between L and R, there are $P01\ldots1$ and $P10\ldots0$, their bitwise "OR" is equal to $P11\ldots1$. Let's try all possible positions of the differing bit, check, that all bits after it are equal to one, then add the number of ways to pick such L and R. It is equal to $2^k \cdot 2^k = 4^k$, where k is the number of bits after the one where L and R differ. In a separate case $L = R = F$, just add one to the answer.

This solution doesn't just count the number of pairs, but conveniently describes the set of all possible pairs: it's the union of sets of the form $L \in [P00\ldots0, P01\ldots1]$, $R \in [P10\ldots0, P11\ldots1]$, there are at most $\log_2 f_i + 1 \leq 61$ such sets for every f_i. Notice that for different lengths of the common prefix P, these sets of pairs (L, R) are disjoint, so we're not overcounting.

The full solution uses dynamic programming. The state of DP is a pair i and one of the at most 61 segments that contains the right endpoint of the i-th segment. For every state, we will calculate the number of ways to build i parts such that the right endpoint of the i-th segment is in it. The transition from i to $i + 1$ is done as follows: we try a couple of segments S_1 and S_2, that $R_i \in S_1$ and $L_{i+1} \in S_2$. Since the right endpoints of the i-th part and the left endpoint of the $i + 1$-th part must be adjacent, then the number of ways to transition from S_1 to S_2 is equal to the length of the intersection of S_1 and S_2 minus one.

This solution can be implemented in $O(nB^2)$, where B denotes the bit length of the input values. If implemented optimally enough, you can score 100 points with it. But, further optimization is possible: if we process segments in sorted order, we can work out the transitions from i to $i+1$ using the two pointers method in $O(B)$ time, yielding an overall $O(nB)$ solution.

Problem Tutorial: "Non-adjacent Swaps"

To solve this problem, you need to understand a few critical observations about the described process:

1. Consider positions of numbers within pairs 1 and 2, 2 and 3, $\ldots n-1$ and n. Every two numbers from the same pair will always be in the same relative order, because, to get swapped, at some point, they have to appear in two adjacent positions.
2. Denote the position of i in the permutation as pos_i. A sequence of comparisons $pos_1 \,?\, pos_2$, $pos_2 \,?\, pos3$, $\ldots pos_{n-1} \,?\, pos_n$ is an invariant (let's call it the *profile* of the permutation). Moreover, any permutation with the same profile is reachable.
3. Let's call an *inversion* a pair of elements that appear in a different order in the starting and in the final permutation. The distance between the starting and the final permutations is equal to the number of inversions between those permutations.

Let's prove the third statement. The distance between permutations can't be smaller than the number of inversions, since any swap eliminates at most one inversion. Let's show that the inversion bound is always attainable. Consider two different permutations with equal profiles. Since they are different, there is at least one inversion. If we could show that there is an inversion in two adjacent (in the final permutation)

positions, we could swap them and decrease both the remaining distance and the number of inversions by one.

Let's prove that such an inversion always exists. Consider the inversion (a, b) with the minimum possible distance between two positions in the final permutation. Suppose that $pos_a < pos_b$ in the final permutation, but $pos_a > pos_b$ in the starting one. If the distance is 1, that's exactly what we need. Otherwise, take any element c between a and b: then, either (a, c), or (c, b) is an inversion. Why? By contradiction, if (a, c) are in the correct order and (c, b) are in the correct order, then (a, b) also have to be in the correct order.

Thus, the two numbers you need to output in this problem are the number of permutations with the given profile and the total number of inversions among those reachable permutations. The first number can be calculated with dynamic programming: `dp[i][j]` is the number of ways to arrange the first i in such a way, that i is at position j ($0 \le j < i$, 0-indexed). To insert $i + 1$, try all possible locations for insertion and check if the relative order with i is correct.

Counting inversions is a little harder, but the general scheme is the same. Instead of keeping track of the number of inversions, let's fix one inversion $a < b$ and find the number of permutations where it appears. To do that, we add an additional variable to the DP state that will remember the position of a in the current permutation. Whenever we reach b, we only keep those transitions in which (a, b) is an inversion. The first iteration of this solution works in $O(n^6)$: n^2 possible inversions to consider, n^3 DP states, $O(n)$ per transition.

The transitions can be optimized with prefix sums, yielding a $O(n^5)$ solution. The last optimization aims to process all b's for a given a at once. Instead of trying all inversions $a < b$, we only try all values of a and keep track of its position as before. While calculating the DP, whenever we encounter some $b > a$ such that they form an inversion, we multiply the current number of ways by the number of ways to insert all remaining elements. It doesn't depend on a, so we can precalculate it; the process is identical to just counting the permutations. This solution runs in $O(n^4)$ which is enough for 100 points.