# Problem A. Cities

Let's assume that we are at point $p$ now. We have two options.

1) Go to $t$ directly, paying extra $|p$ - $t|$ $*a_p$ money.

2) Go to another "optimal"point. There are two choices for that. Either it is minimum $r$ such that $a_r < a_p$ and $r > p$, or maximum $l$ such that $a_l < a_p$ and $l < p$.

So run Dijkstra, maintaining minimum distance. Update answer by the first option. Or go to the next optimal point paying $a_v * dist(r - p$ or $p - l)$ money. We are multiplying $a_v$ because each time we are going to the index with a lower value. Total complexity is $O(nlogn)$.

The graph you get is a DAG, so simple dp is enough. Total complexity is $O(n)$.

Homework: Calculates answers for every $t$ in $O(n)$.

# Problem B. Potatoes

Build a binary tree. If vertex contain array $a_1, a_2$... left child $a_1, a_3$.., right child $a_2, a_4$.... Root vertex contain initial array.

Array *bad* if it contains two equal elements.

For every query: $ans = countofx * x + countofy * y$. Notice $countofx = countofy + 1$. We will calculate $countofx$. $countofx$ equal to a number of *bad* vertexes in the binary tree.

If we fix the right border of the interval for every vertex we can compute the $L_i$-left border such that every interval with such right border and left border at most $L_i$ is bad. $countofx$ equal to a number of $l <= L_i$.

Every element in at most $log(n)$ vertexes in the binary tree. So we will re-calculate $L_i$ only for $log(n)$ vertexes. To calculate a number of $l <= L_i$ we can use segment tree or sqrt decomposition. Total complexity is $O(nlog^2n + qlogn)$ or $O(nlogn + q\sqrt{n})$.

# Problem C. Chips-dips

After each operation number of inversions doesn't change.

If $i < j$ $(p_i = 1, p_j = n)$ call permutations *good*, otherwise *bad*.

The answer exists if the number of inversions is the same and both permutations are *good* or *bad*.

If the permutation *good/bad* we can transform it into every permutations with $p_n = n/1$ and same number of inversion. For every triple $p_{i-1}, p_i, p_{i+1}$ $(p_i = n/1)$ always exist one operation. We will reapat this operations until $p_n = n/1$. It can be proved that the number of operations is at most a polynomial of the second degree in n. Call this operations $take\_out$

Now look to other elements. If they *bad* but initial permutations can be transformed to other with $p_n = n/1$, $p_{n-1} = n - 1/n$ we need make him good using $p_n$. Call this operations $fix$.

Assume initial permutations is *good* and after $take\_out(n)$ and $take\_out(n-1)$ we got permutations $...c, a, b, n-1, 1, n$. If $a < b$ we can transform it to $...c, b, a, 1, n-1, n$, otherwise we need to $order\_suffix$.

pseudo code:

```
$order\_suffix(n)$:

  1. If $c < b < a$ we can transform $c, a, b$ to $b, c, a$
  2. If $b < c < a$ we can transform $c, a, b$ to $a, b, c$
  3. Otherwise $order\_suffix(n-1)$

$take\_out(n)$
```

```
1. If first $n$ elements is *good* find $i$ ($p_i = n$) otherwise $i$ ($p_i = 1$)
2. If $i = n$ break
3. Do operation with $p_{i-1}, p_i, p_{i+1}$
4. $take\_out(n)$
```

$fix(n)$:

```
1. $take\_out(n)$
2. if first $n-1$ elements is *bad* $fix(n-1)$
3. $take\_out(n-1)$
4. $order\_suffix(n-2)$
5. make it *good* using $p_{n-3}, p_{n-2}, p_{n-1}, p_n, p_{n+1}$.
```

$solve(n)$:

```
1. $take\_out(n)$
2. if first $n-1$ elements is *bad* $fix(n-1)$
3. $solve(n-1)$:
```

It can be proved that the number of operations is at most a polynomial of the third degree in n.

# Problem D. Unstable town

**Let's reformulate the statement as follows**. There are $n$ numbers initially painted white. Then, one by one, we recolor them to black. After each change, we build an undirected graph $G$ as follows: for each pair of numbers $(i, p_i)$ add an edge between them if they are of opposite colors. Then, we add the number of connected components in $G$ into our answer.

**First subtask**. The solution to the first subtask is very simple, but necessary — to check if you got the problem statement correctly. Any $O(n!)$ solution will do.

**Second subtask**. To solve the second subtask we can, instead of brute forcing permutations, go through all possible $2^n$ colorings of numbers. For the current graph $G$, we count the number of its connected components $x$ and calculate its contribution to the answer. Let's say we have $a$ white numbers and $b$ black numbers. Then, all black numbers should go before all white numbers. But the order of numbers of the same color is not important. So the contribution is going to be $x \times a! \times b!$. Time complexity is $O(2^n)$.

**Full solution**. I could go on describing all of the partial solutions, but I fear there are a lot of them, so I will not go into details and hop straight into the full solution.

You should have noticed that the graph formed by the edges $(i, p_i)$ is just a bunch of simple cycles. The solution relies on the very important observation — the number of connected components can be expressed as (the number of vertices $V$) - (the number of edges $E$) + (the number of complete cycles $C$). As all these variables are independent of each other, we can calculate them separately.

The sum of number of vertices $V$ over all permutations is simply $n! \times n^2$.

The sum of number of edges $E$ over all permutations is (the number of indices satisfying $p_i \neq i$) $\times$ $(n+1)!/3$ (Try to come up with the formula yourself).

Now, we focus on counting the number of complete cycles $C$ over all permutations. Let's focus on a particular cycle of length $k$. If $k$ is odd, this cycle will not affect the answer because our graph $G$ will always be bipartite.

Otherwise, $k$ is even and there are exactly 2 ways to color this cycle. We will have $k/2$ white and $k/2$ black numbers. All black numbers should go before all white numbers. And the contribution is (the position of the first white number) - (the position of the last black number) + 1.

We could try brute forcing all possible separators $1 \le t \le n$, so that all black numbers are in the range $[1, t]$ and all white numbers are in the range $[t+1, n]$. There are $C_t^{k/2} \times C_{n-t}^{k/2}$ ways to achieve that. The order of these halves also does not matter, so we should also multiply by $(k/2)!^2$.

This solution solves for a fixed length $k$ in $O(n)$ time, so exploiting the fact that there are at most $O(\sqrt{n})$ different cycle lengths, we arrive at a $O(n\sqrt{n})$ **solution for** $81$ **points**.

**To get the full score** we should get rid of brute forcing the separator. There is a beautiful trick. Let's say that separator is just another item between those halves of size $k/2$. So there are $C_n^{k+1}$ combinations in total, where the separator is exactly the $k+1$-th item. However, we did not count cases where the separator is the same as the $k$-th item. There are $C_n^k$ such combinations. So, we have a nice and clean solution in $O(n \log n)$ or $O(n)$ depending on your way of calculating factorial inverses.

# Problem E. Rooms

**Let's discuss the greedy solution first**. Our current state can be described by the segment of available rooms $(l, r)$ and our experience $xp$. Whenever we can open the leftmost or rightmost door and increase our segment, we should do so, because the constraint $a_i \ge 0$ ensures it is always optimal. If we cannot open both doors, we should buy some experience so that at least one of them becomes open and repeat our process. We stop when we can already escape the house.

**The naive implementation** solves the problem in $O(n^2)$ time, which is enough for first two subtasks. It can be further optimized to solve the problem in $O(n \cdot max(b_i))$ time, **which is enough for the fifth subtask**.

**For the third subtask**, calculate prefix and suffix maximums on $b$ and choose the optimal direction.

**For the fourth subtask**, the answer is simply $max(0, b_i - a_i)$.

Now, let's come up with a pure mathematical solution using our greedy approach. Say, we want to calculate the answer for the room $i$. Imagine our segment of available rooms at some point is $(l, r)$ satisfying $l \le i \le r$. Then, to extend the segment, we would have to have at least $min(b_l, b_{r+1}) - \sum_{j=l}^{r} a_j$ coins donated. Since our donations accumulate, it is enough to take the maximum over those.

$$ans_i = max_{l \le i \le r}(min(b_l, b_{r+1}) - \sum_{j=l}^{r} a_j)$$

**Now, there are several approaches leading to the full solution**. I will describe the hardest (but intended) one — divide and conquer. Let's define a function `calc(L, R)` which will solve for all segments within the range $(L, R)$. Let $M = (L+R)/2$. As usual, we recursively run `calc(L, M)` and `calc(M+1, R)`. Now we have to consider all segments connecting both halves. Let's calculate the suffix sum `suff` on $a$ for the first half, and the prefix sum `pref` on $a$ for the second half.

Focus on the left half. Let the left endpoint of a segment be $l \le M$. Then, we would have to update all values $ans_l, \ldots, ans_M$ with the value $max_{M<r\le R}(min(b_l, b_{r+1}) - \texttt{suff}_l - \texttt{pref}_r)$. If we let $b_l \le b_{r+1}$, we are maximizing $-\texttt{pref}_r$ on the right half. If we let $b_l > b_{r+1}$, we are maximizing $b_r - \texttt{pref}_r$ on the right half. The equations for the right half are almost identical. All of those values can be easily calculated if we also do a merge sort on $b$. **Note that without merge sort** we would have had a $O(n \log^2 n)$ solution for 80 points.

So, we get a not-easy-but-not-that-hard-either full solution in $O(n \log n)$ time complexity. There are also some $DSU$ and stack solutions, which are probably easier to implement and work better in practice (but whatever).

# Problem F. Dendrology

1. $n \leq 1000$, $q = 0$. It is guaranteed that $a_i = i$, $b_i = i + 1$ for all $i$ $(1 \leq i < n)$.

Given tree is bamboo. $S_{l,r}$contains all vertexes between leftmost and rightmost vertexes in bamboo. Fix $l$, iterate $r$ and keep leftmost and rightmost vertexes. $O(n^2)$

2. $n \leq 10^5$, $q = 0$. It is guaranteed that $a_i = 1$, $b_i = i + 1$ for all $i$ $(1 \leq i < n)$.

Given tree is star. $S_{l,r}$ contains all vertexes in interval and center vertex if interval contains at least two vertexes. $S_{l,r} = r - l + 1$ or $r - l + 2$. $O(n)$

3. $n \leq 10^5$, $q = 0$. It is guaranteed that $a_i = i$, $b_i = i + 1$ for all $i$ $(1 \leq i < n)$.

Optimization for subtask 1. To calculate $\sum maximum - minimum$ over all intervals we calculate $\sum maximum$ and $\sum minimum$ separately. It's a classic "sum of subarray minimums"problem. $O(n)$

4. $n \leq 1000$, $q = 0$.

Fix $l$, iterate $r$ and answer the query which requires the minimal number of size of subtree which contains all the vertices in the set, after the "add vertices to the set"operation.

We can calculate the distance between two nodes with the LCA algorithm, then when we order the nodes by dfs order, we can answer the "add vertice"query that adds the vertice which is numbered $s$ in dfs order, and assume that the previous numbered vertices in dfs order in the set is vertex $t$, and next is vertex $u$, we can get rid of the "add"query that $(thecurrentsizeofthesubtree) + distance(s, t) + distance(t, u) - distance(s, u)$.

The time complexity of the LCA algorithm is $O(logn)$. $O(n^2 logn)$

5. $n, q \leq 10^5$. It is guaranteed that $a_i = i$, $b_i = i + 1$ for all $i$ $(1 \leq i < n)$.

Find answer without queries(like in subtask 3). Because all operations change two adjacent values only for these values borders will change. We can find new borders with data structure, for example, segment tree. $O(nlogn)$

6. $n \leq 10^5$, $q = 0$.

We will calculate the number of edges in $S_{l,r}$. Edge will in $S_{l,r}$ if in both components separated with this edge have at least one vertex from the interval. We will calculate the number of intervals that don't. Fix any root. Keep set for every vertex that contains all vertex from subtree. Calculate the number of the interval that fully in or fully not in set. To calculate this set for vertex calculate it for child vertexes, merge them(Small to Large), and add this vertex. We can keep this set using segment tree or std::set.$O(nlog^2 n)$

7. Original problem constraints.

Like in subtask 5 only for ends of edges answer will change. Do all like in subtask 6 but persistently. $O(nlog^2 n)$