



WEOI Editorials

Blanca Huergo Muñoz

Scientific committee

Andrea Ciprietti (Italian Olympiads in Informatics)

Blanca Huergo Muñoz (Spanish Informatics Olympiad)

Félix Moreno Peñarrubia (Spanish Informatics Olympiad)

Hugo Peyraud-Magnin (France-IOI)

Jelmer Fiset (Dutch Informatics Olympiad)

Luca Versari (Italian Olympiads in Informatics)

Omer Giménez (Google)

Simon Mauras (France-IOI)

Suneet Mahajan (Irish Informatics Olympiad)

Technical committee

Jacobo Vilella Vilahur (Spanish Informatics Olympiad)

Luca Versari (Italian Olympiads in Informatics)

Plane Turbulences (turbulences)

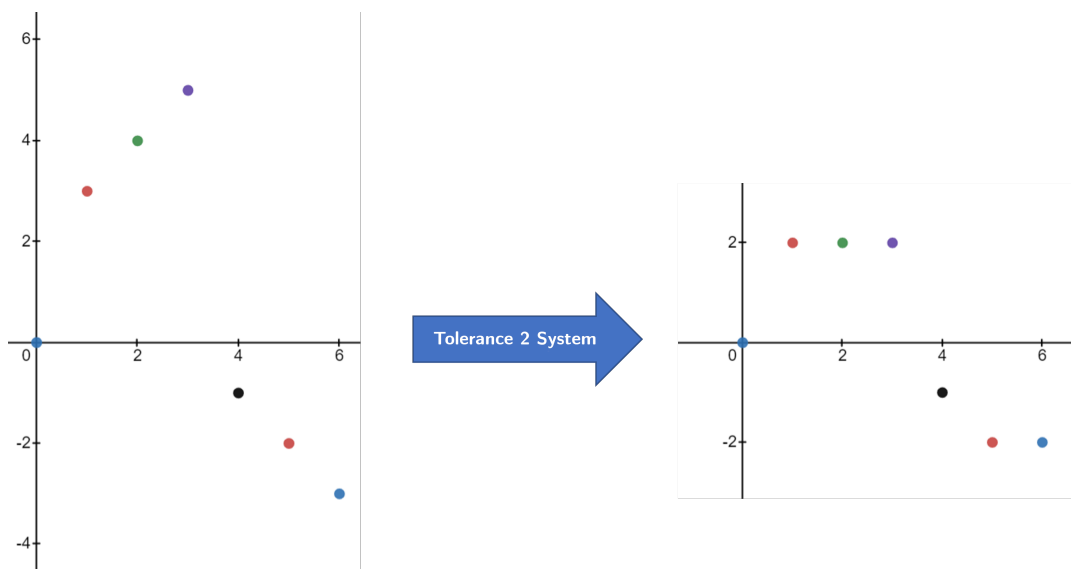
Problem author: Blanca Huergo Muñoz

Problem preparator: Félix Moreno Peñarrubia

Brianair, a reputed airline, is conducting a study about turbulences. They are working to find the optimal stabilising system for their aircraft. This is to be in place during all the flight except landing and take-off, that is, the part of the flight where the plane is supposed to fly *in a straight line*.

A stabilising system of *tolerance* x will ensure that the plane does not deviate from its desired altitude (the one it would have if it were flying in a straight line at constant altitude) by an absolute difference higher than x . It is possible to know in advance the altitude of the plane at each minute of the trip if we do not equip it with a stabilising system. You will be given all these height deviation predictions A_0, \dots, A_{N-1} for the duration of the trip N , in chronological order.

The following example shows how a stabilising system of tolerance 2 takes a flight with deviation predictions $A_0 = 0, A_1 = 3, A_2 = 4, A_3 = 5, A_4 = -1, A_5 = -2, A_6 = -3$ to a flight with actual deviations $B_0 = 0, B_1 = 2, B_2 = 2, B_3 = 2, B_4 = -1, B_5 = -2, B_6 = -2$.



Altitudes before and after applying a stabilising system with tolerance 2.

Brianair knows that customers love high-flying travels, so the customer satisfaction (i.e. the airline's gain from implementing the system) after flying on a plane with a stabilising system of tolerance x equals $\sum_{i=0}^{N-1} B_i$, where B_i is the stabilised altitude at time i . That is, $B_i = \text{sign}(A_i) \cdot \min(|A_i|, x)$.

But the cost of bribing regulators to allow a system with tolerance x equals Kx , where K is some nonnegative constant. The airline therefore wants to maximise its revenue from the flight, i.e. $\left(\sum_{i=0}^{N-1} B_i\right) - Kx$.

Given K and A_0, \dots, A_{N-1} , would you be able to find the maximum revenue that can be attained by setting the optimal tolerance $x \geq 0$?

Implementation

You will have to submit a single `.cpp` source file.

📄 Among this task's attachments you will find a template `turbulences.cpp` with a sample implementation.

You have to implement the following function:

```
C++ | long long revenue(int N, int K, vector<long long> A);
```

- Integer N represents the duration of the flight.
- Integer K represents the cost coefficient.
- The array A , indexed from 0 to $N - 1$, contains the values A_0, A_1, \dots, A_{N-1} , where A_i is the predicted altitude at time i .
- The function should return the maximum revenue that can be obtained.

The grader will call the function `revenue` and will print its return value to the output file.

Sample Grader

The task's directory contains a simplified version of the jury grader, which you can use to test your solution locally. The simplified grader reads the input data from `stdin`, calls the functions that you must implement, and finally writes the output to `stdout`.

The input is made up of 2 lines, containing:

- Line 1: the integers N and K .
- Line 2: the integers A_i , separated by spaces.

The output is made up of a single line, containing the value returned by the function `revenue`.

Constraints

- $1 \leq N \leq 2 \times 10^5$.
- $0 \leq K \leq 2 \times 10^5$.
- $-10^{12} \leq A_i \leq 10^{12}$.

Scoring

Your program will be tested on a set of test cases grouped by subtask. To obtain the score associated to a subtask, you need to correctly solve all the test cases it contains.

- **Subtask 1 [0 points]:** Sample test cases.
- **Subtask 2 [15 points]:** $N = 1$.
- **Subtask 3 [30 points]:** $N \leq 10^2$, $K \leq 10^2$, $-10^2 \leq A_i \leq 10^2$ for each $i = 0, \dots, N - 1$.
- **Subtask 4 [17 points]:** All A_i are equal.
- **Subtask 5 [18 points]:** All A_i are nonnegative.
- **Subtask 6 [20 points]:** No additional constraints.

Examples

stdin	stdout
7 1 0 3 4 5 -1 -2 -3	1
5 1 7 8 -2 5 -10	3
5 0 1000000000000 1000000000000 1000000000000 1000000000000 1000000000000	5000000000000

Explanation

In the **first sample case**, the situation is as described in the picture above. The optimal revenue is obtained with $x = 5$.

In the **second sample case**, the optimal revenue can be obtained by setting $x = 5$. Therefore, the total revenue is $(5 + 5 + -2 + 5 + -5) - 1 \cdot 5 = 3$.

In the **third sample case**, the optimal revenue can be obtained by setting any $x \geq 10^{12}$.

Solution

Note that, as was done in the problem statement, we assume $\text{sign}: \mathbb{Z} \rightarrow \{-1, 0, 1\}$ to be defined as $\text{sign}(x) = 1$ for $x > 0$, $\text{sign}(x) = -1$ for $x < 0$ and $\text{sign}(0) = 0$.

[Subtask 1 has been omitted since it just comprises the sample cases.]

Subtask 2: $N = 1$.

Letting $z = A[0]$, the constraint in this subtask reduces the problem to finding $\max_{x \geq 0} f(z; x, K) = \text{sign}(z) \min(x, |z|) - Kx$.

When $z = 0$, $f(z; x, K) = -Kx$ for all $x \geq 0$. Hence, since $K \geq 0$, this will be maximised when $x = 0$, with corresponding revenue $f(0; 0, K) = 0$.

Now, when $z > 0$, $f(z; x, K) = \min(x, z) - Kx$. Now, this means that when $x \leq z$, $f(z; x, K) = x - Kx$, whereas $f(z; x, K) = z - Kx$ otherwise. Hence, this function will either be maximised at $x = 0$ or $x = z$, with revenue equal to $\max(f(z; 0, K), f(z; z, K)) = \max(0, z - Kz)$.

Finally, when $z < 0$, $f(z; x, K) = -\min(x, -z) - Kx$, that is, $f(z; x, K)$ will be the sum of two non-negative terms. Therefore, it will be maximised by picking $x = 0$, which will make both terms equal to 0, with revenue therefore equal to 0.

Subtask 3: $N \leq 100$, $K \leq 100$, $-100 \leq A_i \leq 100$ for each $i = 0, \dots, N - 1$.

Let $g(x; K) = \sum_{i=1}^N \text{sign}(A_i) \min(x, |A_i|) - Kx$.

Observe that if $M = \max_i |A_i|$, picking $x \geq M$ yields:

$$\begin{aligned} g(x; K) &= \sum_{i=1}^N \text{sign}(A_i) \min(x, |A_i|) - Kx \\ &= \sum_{i=1}^N A_i - Kx \\ &= \sum_{i=1}^N \text{sign}(A_i) \min(M, |A_i|) - Kx \\ &\leq \sum_{i=1}^N \text{sign}(A_i) \min(M, |A_i|) - KM \\ &= g(M; K) \end{aligned}$$

Hence, we have that it is enough to check in the range $0 \leq x \leq \max_i |A_i|$. For this subtask, it suffices to try all values of x in this range and naively calculate f for each value of x by iterating over all indices of A .

Subtask 4: All A_i are equal.

In a similar fashion to how we approached Subtask 2, letting $z = A[0]$, the constraint in this subtask reduces the problem to finding the (natural) value of x that maximises $h(z; x, K) = N \cdot \text{sign}(z) \min(x, |z|) - Kx = \frac{1}{N}(\text{sign}(z) \min(x, |z|) - KNx) = \frac{1}{N}f(z; x, KN)$.

But then it suffices to use our solution for Subtask 2 with KN in the place of K , remembering to divide the result by N before returning.

Subtask 5: All A_i are non-negative.

This subtask will give us the key insight needed to solve the general problem. We know from the analysis of Subtask 3 that we only need to consider values of x between 0 and $M = \max_i |A_i|$. But M can equal 10^{12} , which is much larger than the number of values we can consider before reaching the time limit.

However, n , the length of A , will not exceed $2 \cdot 10^5$, a much more manageable number. Define the list C

as $C = (c_1, \dots, c_m)$, where $c_1 < \dots < c_m$ and every entry of C is either 0 or an entry of A (and both 0 and all entries of A are entries of C). By definition of C , note that $m \leq N + 1$.

We can in fact observe that an optimal value of x must be an entry of C .

Proof:

For a contradiction, suppose otherwise. That is, all optimal values of x are not entries of C . Since we have already shown that at least one optimal value of x lies between 0 (which is c_1) and M (which is c_m), then at least one optimal value of x lies between entries of C .

Take an optimal value b of x that lies between two entries of C : $\exists j$ s.t. $c_j < b < c_{j+1}$. We hence have that:

$$g(c_j; K), g(c_{j+1}; K) < g(b; K)$$

However, for all i , we have that if $c_j < |A_i|$, then $b < |A_i|$ too (in which case $\min(b, |A_i|) = b$ and $\min(c_j, |A_i|) = c_j$), whereas if $c_j > |A_i|$, then $b > |A_i|$ (in which case $\min(b, |A_i|) = \min(c_j, |A_i|) = |A_i|$).

We therefore have that $\sum_{i=1}^N \text{sign}(A_i) \min(x, |A_i|)$ is a linear function between c_j and c_{j+1} and, since $-Kx$ is also clearly linear in this interval, it follows that $g(x; K)$ is a linear function between c_j and c_{j+1} , contradicting the fact that $g(c_j; K), g(c_{j+1}; K) < g(b; K)$. \square

We have now shown that it an optimal value of x lies in C . It therefore suffices to calculate $g(c_j; K)$ for all j from 1 to m to find the maximal revenue. However, a naïve calculation of the revenue for each entry of C still yields a TLE verdict. Note now that, for any j :

$$\begin{aligned} g(c_j; K) &= \sum_{i=1}^N \text{sign}(A_i) \min(c_j, |A_i|) - Kc_j \\ &= \text{sum}(c_j) + c_j \cdot \text{gt}(c_j) - c_j \cdot \text{lt}(c_j) - Kc_j \\ &= \text{sum}(c_j) + c_j(\text{gt}(c_j) - \text{lt}(c_j) - c_j) \end{aligned}$$

where $\text{sum}(c_j)$ is the sum of all the A_i 's where $|A_i| \leq c_j$, $\text{gt}(c_j)$ is the number of indices of A such that $A_i > c_j$ and $\text{lt}(c_j)$ is the number of indices of A such that $A_i < -c_j$.

Therefore, we can solve this subtask by sorting A in terms of the $|A_i|$'s and calculating the revenue for the values of x in increasing order (that is, from $x = c_1$ to $x = c_m$). We can do this by keeping track of the values of $\text{sum}(x)$, $\text{gt}(x)$ and $\text{lt}(x)$ as we loop through A .

The simplest way to do this is probably to first loop backwards over the sorted array A and calculate for each index i , how many of $A[i], \dots, A[N - 1]$ are positive entries. Once this has been done, we can then loop forwards over A , keeping track of the values of $\text{sum}(x)$ and calculating $\text{gt}(x)$ and $\text{lt}(x)$ based on the number of positive entries that follow the current index.

The complexity of this solution is hence $O(N \log N)$, which is dominated by the sorting of A , since then the calculation over the sorted array actually takes place in $O(N)$.

Subtask 6: No additional constraints.

The solution to Subtask 6 can be coded almost identically to that of Subtask 5. It only requires one extra observation: we define the list D in the same way as C was defined in the explanation above, with the difference that for all negative entries of A , the absolute value instead of the actual value is taken when inserted into D . This way, $D = (d_1, \dots, d_m)$, where $d_1 < \dots < d_m$ and every entry of D is either 0 or the absolute value of an entry of A (and both 0 and the absolute value of all entries of A are an entry of D). Note again that by definition of D , $m \leq N + 1$.

By a symmetrical argument to the one in the analysis of Subtask 5, we can conclude that an optimal value of x will be an entry of D . Hence, by substituting each entry of A with its absolute value to define D instead of C (note that the actual entry of A should be used to calculate the values of $\text{sum}(x)$), we can practically reuse the solution to Subtask 5 to obtain an AC verdict.

[Spoiler alert: the following page contains C++ code for a 100 points solution.]

Accepted Solution

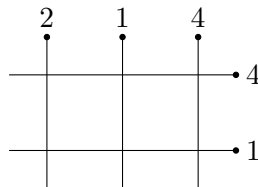
```
1 #include <algorithm>
2 #include <vector>
3 using namespace std;
4 typedef long long int ll;
5
6 long long revenue(int N, int K, vector<long long> A) {
7     // we first sort A by absolute value
8     sort(A.begin(), A.end(), [](ll i, ll j) { return abs(i) < abs(j); });
9
10    ll highestRevenue = 0ll; // with x = 0, the revenue is always 0
11    // we now skip over the zeros in A:
12    size_t i = 0;
13    while (i < A.size() && A[i] == 0) {
14        i++;
15    }
16
17    // in order to calculate count_gt and count_lt it will be useful to know, for
18    // each index of A, how many positive entries follow it (including, if it is
19    // positive, itself).
20    int positives[N];
21    positives[N - 1] = A[N - 1] > 0;
22    for (int j = N - 2; j >= 0; j--) {
23        positives[j] = positives[j + 1] + (A[j] > 0);
24    }
25
26    ll x;
27    ll sum = 0ll;
28    int count_gt, count_lt, cnt_current;
29    while (i < A.size()) {
30        // we count how many entries of A have the absolute value of A[i] and use
31        // this to update sum(|A[i]|), count_gt(|A[i]|) and count_lt(|A[i]|)
32        cnt_current = 1;
33        sum += A[i];
34        while (i + 1 < A.size() && abs(A[i + 1]) == abs(A[i])) {
35            i++;
36            cnt_current++;
37            sum += A[i];
38        }
39        count_gt = positives[i] - (A[i] > 0);
40        count_lt = N - i - 1 - count_gt;
41        // it now just suffices to calculate the new revenue and compare with the
42        // current max
43        x = abs(A[i]);
44        highestRevenue = max(highestRevenue, sum + x * (count_gt - count_lt - K));
45        i++;
46    }
47    return highestRevenue;
48 }
```


Political cost (cost)

Problem author: Hugo Peyraud-Magnin

Problem preparator: Omer Giménez

In the city where you live there are N streets going East-to-West (from 0th Street to $(N - 1)$ th Street) and M avenues going North-to-South (from 0th Avenue to $(M - 1)$ th Avenue). Every street or avenue has a *political weight*, which is the importance of the most important citizen living in it. We represent political weights as two arrays $A[0 \dots N - 1]$ and $B[0 \dots M - 1]$ of integers from 1 to K . The following plot represents such a city with 2 streets and 3 avenues, with political weights of $A = [1, 4]$ and $B = [2, 1, 4]$ respectively.




The major wants to organize a parade through the city. If the parade goes through the intersection of x th Street with y th Avenue, the traffic of both roads will be disrupted, and the major will incur a *political cost* of $\max(A[x], B[y])$. If the parade crosses goes through multiple intersections, the political cost will be the *maximum* of the political cost of each intersection. Notice that costs are not summed: what matters is not how many people the parade bothers, but how important is the most important citizen the parade bothers.

The *political distance* between two intersections is the smallest *political cost* of a parade departing from the first intersection and arriving at the second intersection. Your job is to compute the sum of the political distances between all pairs of intersections in the city.

Implementation

You will have to submit a single `.cpp` source file.

 Among this task's attachments you will find a template `cost.cpp` with a sample implementation.

You have to implement the following function:

```
C++ | int solve(int N, int M, int K, vector<int> A, vector<int> B);
```

- Integer N represents the number of East-to-West streets.
- Integer M represents the number of North-to-South avenues.
- The array A , indexed from 0 to $N - 1$, contains the values A_0, A_1, \dots, A_{N-1} , where A_i is the political weight of the i -th East-to-West street.
- The array B , indexed from 0 to $M - 1$, contains the values B_0, B_1, \dots, B_{M-1} , where B_i is the political weight of the i -th North-to-South street.
- The function should return the sum of the political distances between all possible pairs of intersections, **modulo** 1000003.

The grader will call the function `solve` and will print its return value to the output file.

Sample Grader

The task's directory contains a simplified version of the jury grader, which you can use to test your solution locally. The simplified grader reads the input data from `stdin`, calls the functions that you must implement, and finally writes the output to `stdout`.

The input is made up of 3 lines, containing:

- Line 1: the integers N , M and K .
- Line 2: the integers A_i , space-separated.
- Line 3: the integers B_i , space-separated.

The output is made up of a single line, containing the value returned by the function `solve`.

Constraints

- $1 \leq N \leq 3 \times 10^5$.
- $1 \leq M \leq 3 \times 10^5$.
- $1 \leq K \leq N + M$.
- $1 \leq A_i \leq K$ for $i = 0 \dots N - 1$.
- $1 \leq B_i \leq K$ for $i = 0 \dots M - 1$.

Scoring

Your program will be tested on a set of test cases grouped by subtask. To obtain the score associated to a subtask, you need to correctly solve all the test cases it contains.

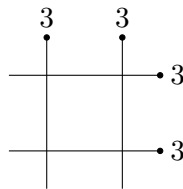
- **Subtask 1 [0 points]**: Sample test cases.
- **Subtask 2 [10 points]**: $N \leq 10^1, M \leq 10^1$.
- **Subtask 3 [10 points]**: $N \leq 10^2, M \leq 10^2$.
- **Subtask 4 [10 points]**: $N = 1, M \leq 10^4$.
- **Subtask 5 [10 points]**: $N = 1, M \leq 10^5$.
- **Subtask 6 [10 points]**: $N \leq 10^3, M \leq 10^3$.
- **Subtask 7 [10 points]**: $N \leq 10^4, M \leq 10^4$.
- **Subtask 8 [10 points]**: $N \leq 10^5, M \leq 10^5$ and the arrays A and B are non-decreasing, that is, if $i < j$, then $A_i \leq A_j$ and $B_i \leq B_j$.
- **Subtask 9 [10 points]**: $N \leq 10^5, M \leq 10^5, K \leq 10^1$.
- **Subtask 10[10 points]**: $N \leq 10^5, M \leq 10^5$.
- **Subtask 11[10 points]**: No additional constraints.

Examples

stdin	stdout
2 2 4 3 3 3 3	48
1 3 4 2 2 3 1	25
2 3 5 1 4 2 1 4	135

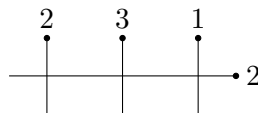
Explanation

In the **first sample case**, we have a city with 2 streets and 2 avenues, all of them of political weight 3:



There are 16 different pairs of intersections. Since the political distance between each pair of intersections is 3, the solution is $3 \cdot 16 = 48$.

In the **second sample case** there is 1 street and 3 avenues, with political weights $A = [2]$ and $B = [2, 3, 1]$ respectively:



There are 9 pairs of intersections. Three of these pairs start and end at the same intersection, and have political distances of 2, 3 and 2 respectively (the rightmost avenue has a political weight of 1, but the political weight of the only street is 2, so the political distance of any parade is at least 2). For each remaining pair of intersections, the parade joining them must cross the middle avenue, and hence must have political distance of 3. So the total sum is $2 + 3 + 2 + 6 \cdot 3 = 25$.

The **third sample case** corresponds to the example given in the statement of the problem. Here, there are 2 streets and 3 avenues. You can check, with some patience, that the sum of the political distances of the 36 pairs of intersections is 135.

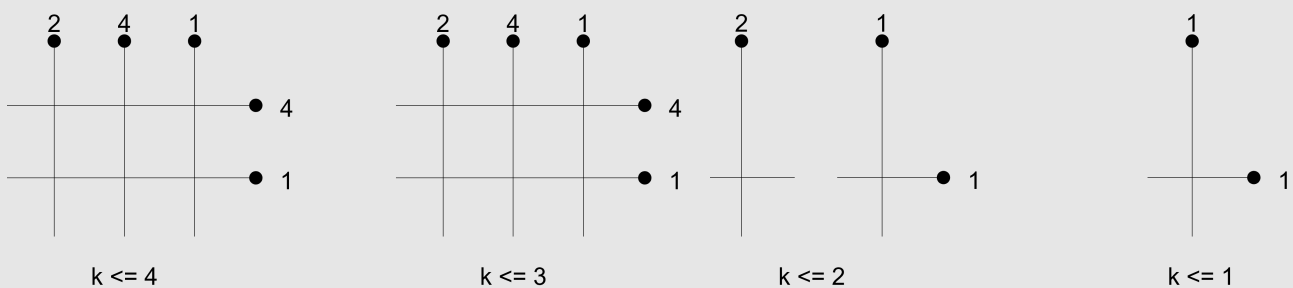
Solution

Note that here, as was done in the presentation of the solutions at the end of the contest, I use the word “road” as a generic term for both avenues and streets.

Take some $k \leq K$ and consider the number of pairs of intersections p_k that have a political cost of at most k . The number of such pairs of intersections will be the number of pairs of intersections in the map obtained by removing all streets and avenues with a weight greater than k from the original map, since these will be the roads that can be traversed in a path of cost $\leq k$ between any pair of intersections.

Now, let q_k equal the number of pairs of intersections that have a cost of exactly k . We can see that $q_k = p_k - p_{k-1}$ for $0 \leq k \leq K$, letting $p_{-1} = 0$ for simplicity of notation. The answer to the original problem is $\sum_{k=0}^K kq_k$ (all modulo 1000003).

In order to understand the relationship between p_{k-1} and p_k , it will be helpful to draw the maps that are generated in the process. Take the following example:



We first make a simple observation: if there are no roads of weight k , then $p_k = p_{k-1}$ and $q_{k-1} = 0$. Suppose otherwise.

The key to solving this problem in a way that is simple and efficient to implement is to observe that it is not necessary to calculate q_k itself, but we can instead remove from the map with weights $\leq k$ all roads of weight k one by one, adding to the total cost all pairs of intersections of cost k that cease to exist after this removal, multiplying this number by k . This works because if we count all these pairs of intersections we are removing, the total is q_k .

It now remains to understand the effect of removing a road in the total of pairs of intersections. We carry out two steps as preprocessing. We first apply the monotonic stack technique twice to A to calculate the following: for each index i , the largest j such that $j < i$ and $A_j \leq A_i$ (letting $j = -1$ if no such j exists) and the smallest h such that $i < h$ and $A_i < A_h$ (letting $h = N$ if no such h exists). We do the same to B . Let $idxLeft_A$, $idxLeft_B$, $idxRight_A$ and $idxRight_B$ represent these mappings. Then, we sort the avenues and streets (separately) in descending order of weight. In the case where two roads have the same weight, pick the one with the smaller index first. The use of `structs` or a similar idea is recommendable, since this allows us to also store the corresponding initial index inside A or B of each road.

Suppose we are now at a point where our current map is the original map with all roads of weight $> k$ removed, for some $k \leq K$. We then remove the streets with weight k from bottom to top and subsequently remove the avenues with weight k from left to right, i.e. in the order of the indices of A and B respectively. We can iterate over the roads in this order easily, since we will have already sorted A and B so that the roads of weight k appear together as a block, sorted by index.

When we reach a street of weight k in this processing, we have already removed all streets of weight k below it, whereas all streets of weight k above it are still there. Therefore, the pairs of intersections formed by an intersection below the current street and an intersection on the current street with a political cost k will have already been counted if and only if the bottom intersection was in a street of weight k . Hence, it suffices to remove the pairs of intersections formed by an intersection on the current street (let its index be i) and an intersection on streets in indices $[idxLeft_A(i) + 1, idxRight_A(i) - 1]$, both on avenues that

have not yet been removed. The reason for this upper bound on the interval is that all streets of weight $> k$ will have been removed already, so streets from $idxRight_A(i)$ onwards will be unreachable from street i . Hence, the removal of street i incurs the removal of $s_i = 2(i - idxLeft_A(i))(idxRight_A(i) - i) - 1$ pairs of streets in the current map and therefore $s_i \cdot a$ (where a is the current number of avenues) intersections of weight k . We therefore add $k \cdot s_i \cdot a$ to the total. Note that since we know that in the original map the number of pairs of streets is N^2 and that of avenues is M^2 , from these two numbers we can calculate the rest.

A symmetrical analysis of the removal of avenues yields the full solution. The cost of this solution is $O(N \log N + M \log M)$, dominated by the sorting of both streets and avenues.

[Spoiler alert: the following page contains C++ code for a 100 points solution.]

Accepted Solution

```
1  #include <algorithm>
2  #include <stack>
3  #include <vector>
4  using namespace std;
5  typedef long long int ll;
6
7  ll mod = 1000003;
8
9  struct road {
10     int weight, index;
11     road() {}
12     road(int w, int i) {
13         weight = w;
14         index = i;
15     }
16     bool operator<(const road &other) const {
17         if (weight != other.weight)
18             return weight > other.weight;
19         return index < other.index;
20     }
21 };
22
23 // Uses monotonic stack technique to find, for each index i of A,
24 // the largest index j such that j < i and A[j] > i
25 vector<int> leftGeq(vector<int> &A) {
26     vector<int> indices(A.size());
27     indices[0] = -1;
28     stack<int> S;
29     S.push(0);
30     for (int i = 1; i < (int)A.size(); i++) {
31         while (S.size() && A[S.top()] < A[i]) {
32             S.pop();
33         }
34         if (S.size()) {
35             indices[i] = S.top();
36         } else {
37             indices[i] = -1;
38         }
39         S.push(i);
40     }
41     return indices;
42 }
43
44 // Uses monotonic stack technique to find, for each index i of A,
45 // the smallest index j such that j > i and A[j] >= i
46 vector<int> rightGeq(vector<int> &A) {
47     vector<int> indices(A.size());
48     int N = (int)A.size();
49     indices[N - 1] = N;
50     stack<int> S;
51     S.push(N - 1);
52     for (int i = N - 2; i >= 0; i--) {
```

```

53     while (S.size() && A[S.top()] <= A[i]) {
54         S.pop();
55     }
56     if (S.size()) {
57         indices[i] = S.top();
58     } else {
59         indices[i] = N;
60     }
61     S.push(i);
62 }
63 return indices;
64 }
65
66 vector<road> createRoadsSet(vector<int> &C) {
67     vector<road> roads(C.size());
68     for (int i = 0; i < (int)C.size(); i++) {
69         roads[i] = road(C[i], i);
70     }
71     sort(roads.begin(), roads.end());
72     return roads;
73 }
74
75 void update(ll left, ll right, ll k, ll &result, ll &pairsCurDir,
76            ll pairsOtherDir) {
77     ll pairsToRemove = (((left * right * 2) % mod) + mod - 1) % mod;
78     ll pairsInterRemoved = (pairsToRemove * pairsOtherDir) % mod;
79     k %= mod;
80     result += (pairsInterRemoved * k) % mod;
81     result %= mod;
82     pairsCurDir = (pairsCurDir - pairsToRemove + mod) % mod;
83 }
84
85 int solve(int N, int M, int K, vector<int> A, vector<int> B) {
86     // Preprocessing step 1: monotonic stacks
87     vector<int> leftBoundsStreets = leftGeq(A);
88     vector<int> leftBoundsAvenues = leftGeq(B);
89     vector<int> rightBoundsStreets = rightGeq(A);
90     vector<int> rightBoundsAvenues = rightGeq(B);
91     // Preprocessing step 2: sort roads
92     vector<road> streets = createRoadsSet(A);
93     vector<road> avenues = createRoadsSet(B);
94
95     ll pairsStreets = ((ll)N * (ll)N) % mod;
96     ll pairsAvenues = ((ll)M * (ll)M) % mod;
97     ll result = 0;
98     K = max(streets[0].weight, avenues[0].weight);
99
100    int left, right, idx;
101    int streetsIt = 0, avenuesIt = 0;
102    for (int k = K; k > 0; k--) {
103        // remove all streets of weight k
104        while (streetsIt < N) {
105            if (streets[streetsIt].weight != k)
106                break;

```

```
107     idx = streets[streetsIt].index;
108     left = idx - leftBoundsStreets[idx];
109     right = rightBoundsStreets[idx] - idx;
110     update(left, right, k, result, pairsStreets, pairsAvenues);
111     streetsIt++;
112 }
113
114 // remove all avenues of weight k
115 while (avenuesIt < M) {
116     if (avenues[avenuesIt].weight != k)
117         break;
118     idx = avenues[avenuesIt].index;
119     left = idx - leftBoundsAvenues[idx];
120     right = rightBoundsAvenues[idx] - idx;
121     update(left, right, k, result, pairsAvenues, pairsStreets);
122     avenuesIt++;
123 }
124 }
125 return result;
126 }
```


Shop Tour (tour)

Problem author: Félix Moreno Peñarrubia

Problem preparator: Félix Moreno Peñarrubia

In Lineland there are N cookie shops in a row, numbered from 0 to $N - 1$. Baq wants to do a *shop tour* through the shops. A shop tour is determined by N **distinct** integers P_0, \dots, P_{N-1} between 0 and $N - 1$.

For a given shop tour, Baq will start at shop P_0 . For each $i = 0, \dots, N - 1$, Baq will move from shop P_i to shop P_{i+1} (here we say $P_N = P_0$) buying one cookie from each of the shops between P_i and P_{i+1} , inclusive. Formally, if $L_i = \min(P_i, P_{i+1})$ and $R_i = \max(P_i, P_{i+1})$, then in the i -th step Baq will buy one cookie from each of the shops $L_i, L_i + 1, \dots, R_i$.

Baq now has the numbers A_0, \dots, A_{N-1} , where A_i denotes the total number of cookies bought in the i -th shop, but does not remember the shop tour. Your task is to determine if the information in the array A is consistent with a valid shop tour, and if it is, construct such a valid tour. Additionally, in order to obtain a full score (see the scoring section for details) the tour you construct must be the *lexicographically smallest* such tour.


We say that a tour P_0, \dots, P_{N-1} is *lexicographically smaller* than a different tour Q_0, \dots, Q_{N-1} if there exists a $0 \leq k \leq N - 1$ such that:

- $P_i = Q_i$ for all $0 \leq i < k$.
- $P_k < Q_k$.

A tour Q is the lexicographically smallest among those consistent with the information in the array A if there does not exist a different tour P with the same array A of bought cookies in each store which is lexicographically smaller than Q .

Implementation

You will have to submit a single `.cpp` source file.

 Among this task's attachments you will find a template `tour.cpp` with a sample implementation.

You have to implement the following function:

```
C++ | variant<bool, vector<int>> find_tour(int N, vector<int> A);
```

- Integer N represents the number of shops.
- The array A , indexed from 0 to $N - 1$, contains the values A_0, A_1, \dots, A_{N-1} , where A_i is the number of cookies bought at the i -th store.
- The function should return either a boolean or an array of integers.
 - If no valid shop tour exists which corresponds to the array A , the function should return `false`.
 - If a valid shop tour exists, you have multiple options:

- * To be awarded the full score, the procedure should return an array of N integers P_0, \dots, P_{N-1} representing the **lexicographically smallest** shop tour resulting in the array A .
- * To be awarded a partial score, the procedure should return an array of N integers P_0, \dots, P_{N-1} representing any not-lexicographically-smallest shop tour resulting in the array A .
- * To be awarded a smaller partial score, the procedure should return `true` or any array of integers not describing a valid shop tour resulting in the array A .

The grader will call the function `tour` and will print the following to the output file:

- If the return value is `false`, it will print a single line with the string `NO`.
- If the return value is `true` or an array of integers of length not equal to N , it will print a single line with the string `YES`.
- If the return value is an array P of N integers, it will print a single line with the string `YES`, followed by a line with the N integers P_0, \dots, P_{N-1} separated by spaces.

Sample Grader

The task's directory contains a simplified version of the jury grader, which you can use to test your solution locally. The simplified grader reads the input data from `stdin`, calls the functions that you must implement, and finally writes the output to `stdout`.

The input is made up of two lines, containing:

- Line 1: the integer N .
- Line 2: the integers A_i , separated by spaces.

The output is made up of one or two lines, containing the values returned by the function `tour`.

Constraints

- $2 \leq N \leq 10^6$.
- $0 \leq A_i \leq 10^6$.

Scoring

Your program will be tested on a set of test cases grouped by subtask. The score associated to a subtask will be the minimum of the scores obtained in each of the test cases.

- **Subtask 1 [0 points]:** Sample test cases.
- **Subtask 2 [8 points]:** $N \leq 8$.
- **Subtask 3 [32 points]:** $N \leq 2 \times 10^3$.
- **Subtask 4 [16 points]:** $A_i \leq 4$ for all $i = 0, \dots, N - 1$.
- **Subtask 5 [20 points]:** There exists a $0 \leq j \leq N - 1$ such that $A_i \leq A_{i+1}$ for all $0 \leq i < j$ and $A_i \geq A_{i+1}$ for all $j \leq i \leq N - 2$.
- **Subtask 6 [24 points]:** No additional constraints.

For each test case in which a valid shop tour exists, your solution:

- gets full points if it returns the lexicographically smallest valid shop tour.

- gets 75% of the points if it returns a valid shop tour which is not the lexicographically smallest one.
- gets 50% of the points if it returns `true` or an array that does not describe a valid shop tour.
- gets 0 points otherwise.

For each test case in which a valid shop tour does not exist, your solution:

- gets full points if it returns `false`.
- gets 0 points otherwise.

Examples

stdin	stdout
4 2 4 4 2	YES 0 2 1 3
3 2 2 2	NO

Explanation

In the **first sample case**, the tour $P = [0, 2, 1, 3]$ generates the array $A = [2, 4, 4, 2]$ as follows:

- Initially, the number of cookies bought from each shop is $[0, 0, 0, 0]$.
- Baq moves from shop $P_0 = 0$ to shop $P_1 = 2$, so the array after this move is $[1, 1, 1, 0]$.
- Baq moves from shop $P_1 = 2$ to shop $P_2 = 1$, so the array after this move is $[1, 2, 2, 0]$.
- Baq moves from shop $P_2 = 1$ to shop $P_3 = 3$, so the array after this move is $[1, 3, 3, 1]$.
- Finally, Baq moves from shop $P_3 = 3$ to shop $P_0 = 0$, so the final array is $[2, 4, 4, 2]$.

It can be shown that $[0, 2, 1, 3]$ is the lexicographically smallest such tour.

In the **second sample case**, it can be shown that there does not exist a valid tour resulting in the array $A = [2, 2, 2]$.

Solution

To start with, we can think about the tour (P_0, \dots, P_{N-1}) (letting $P_N = P_0$, as was done in the problem statement, for simplicity of notation) as a list of intervals: $[\min(P_0, P_1), \max(P_0, P_1)], [\min(P_1, P_2), \max(P_1, P_2)], \dots, [\min(P_{N-1}, P_N), \max(P_{N-1}, P_N)]$. Note that each point in P_0, \dots, P_{N-1} is an endpoint of exactly two intervals. We can imagine these intervals all drawn on top of each other over a number line, with some overlaps. This “number line” intuition will help us understand the problem better.

Let $f: \{0, \dots, N-1\} \rightarrow \{0, \dots, N-1\}$ map each cookie shop number to its index in the permutation P . That is, $f(i) = P_j \iff P_j = i$. This is clearly well-defined.

By definition, A_i is equal to the number of intervals that include $f(i)$ in them, that is, the number of indices $0 \leq j \leq N-1$ where $\min(P_j, P_{j+1}) \leq f(i) \leq \max(P_j, P_{j+1})$.

Note that, since at least intervals $j-1$ (or N when $j=0$) and j contain the integer P_j , then $A_i \geq 2$ for all i or there is no solution. We can now assume therefore that $A_i \geq 2$ for all $0 \leq i \neq N-1$.

Now, observe that the only way for shop 0 to be included in an interval is for it to be the left endpoint. Hence, if $A_0 \neq 2$ no valid permutation exists. Assume now that $A_0 = 2$. Note that by a symmetric argument, we can conclude that $A_{N-1} = 2$ or no valid permutation exists.

Extending this logic further, we observe that since for all shops along the line, exactly two intervals have the shop as an endpoint, shops can be divided into three categories:

1. The shop is the left endpoint of both intervals that have it as an endpoint.
2. The shop is the right endpoint of both intervals that have it as an endpoint.
3. The shop is the left endpoint of one of the intervals that has it as an endpoint and the right endpoint of the other.

By iterating over A_0, \dots, A_{N-1} , we can keep track of which intervals are currently *open*, that is, that we have found their left interval but not the corresponding right interval. This information will allow us to categorise each shop, which will be the key information we need to construct the required tour.

From above, we have that $A_0 = 2$ and hence that shop 0 is the left endpoint of 2 intervals, which we now store as open.

Suppose that we reach some A_i , with $i \geq 1$, and k intervals that are currently open. If shop i is in category 1, then it is left endpoint of two intervals that have not been opened yet and is hence not the right endpoint of any of the open intervals. Thus we have that $A_i = k + 2$. If this is the case, we then update shop i 's category and increment the number of open intervals by 2.

If shop i is in category 2, then it is the right endpoint of two of the open intervals and is not the left endpoint of any new interval. Therefore, $A_i = k$. If this is the case, we then update shop i 's category and decrement the number of open intervals by 2.

If shop i is in category 3, it is the right endpoint of one of the open intervals and the left endpoint of an interval that has not been opened yet. Hence, $A_i = k + 1$ and, if this is the case, we update shop i 's category and leave the number of open intervals as is.

Finally, if A_i has any other value, no valid permutation exists, since we have already gone over all three shop categories.

Also, we must remember to check on each iteration of the loop over A that the number of open intervals is non-negative.

Once we have recorded the category of all N shops, we have to ensure that all open intervals have been closed.

It now simply remains to try to construct a valid permutation that respects this (note that having found these categories does not imply that a solution exists, as we will soon see) and is the lexicographically

smallest. This second iteration over A could be merged with the first and both steps done at once, but the explanation is a bit cleaner when split into two steps and the code still yields AC.

Since we are searching for the smallest lexicographically valid tour and these tours have a cyclic definition, we can trivially let $P_0 = 0$. We now fill in the rest of the indices of P greedily. Essentially, the idea behind this algorithm is to iterate over the indices 1 to $N - 2$ and, whenever it does not lead to a wrong tour, append the current index to the tour. This idea abstractly makes sense, since it will always be more convenient to append the current index than a future, larger index to the tour. And how do we carry out these *wrong tour* checks?

We keep a queue where we store the indices of category 1 shops that have not yet been added to the queue, sorted by their index from smallest to largest. It suffices to use a standard queue instead of a heap/priority queue since this is already the order in which we are visiting the indices. When we visit index 1, this queue is empty, since shop 0 was already appended to the tour.

Whenever we reach an index with its shop being in category 2, we first check to make sure that Q is not empty and, otherwise, conclude that no valid permutation exists. Essentially, we are ensuring that our tour does not at any point close any intervals that it has not yet opened. After this check, we can safely append it to the tour and follow this with dequeuing the index at the front of Q and appending this to the tour too.

Also, whenever we reach an index with its shop being in category 1, we enqueue it to Q , where it will wait for the next index in category 2 to be dequeued and added to the tour.

Finally, whenever we reach an index with its shop being in category 3, we append it to the tour but do not touch Q , since the number of open intervals is unaffected by it.

Once we have iterated over all indices from 1 to $N - 2$, the queue will have been emptied and P will have length $N - 1$. Shop $N - 1$ will close the last open intervals and we can return the tour obtained by appending it to P .

[Spoiler alert: the following page contains C++ code for a 100 points solution.]

Accepted Solution

```
1  #include <queue>
2  #include <variant>
3  #include <vector>
4  using namespace std;
5
6  variant<bool, vector<int>> find_tour(int N, vector<int> A) {
7      // Easy check to save us time:
8      if (A[0] != 2 || A[N - 1] != 2) {
9          return false;
10     }
11
12     // Let us first find the category of each shop
13     // Endpoints category notation: 0 = both left, 1 = mix, 2 = both right
14     vector<int> shopCategory(N);
15     shopCategory[0] = 0;
16     int openIntervals = 2;
17     for (int i = 1; i < N; i++) {
18         if (A[i] - openIntervals == 2) {
19             shopCategory[i] = 0;
20             openIntervals += 2;
21         } else if (A[i] - openIntervals == 1) {
22             shopCategory[i] = 1;
23         } else if (A[i] - openIntervals == 0) {
24             shopCategory[i] = 2;
25             openIntervals -= 2;
26         } else {
27             return false;
28         }
29         if (openIntervals < 0)
30             return false;
31     }
32     if (openIntervals != 0)
33         return false;
34
35     // Let us now construct the tour
36     vector<int> shopTour;
37     shopTour.push_back(0);
38     queue<int> Q;
39     for (int i = 1; i < N - 1; i++) {
40         if (shopCategory[i] == 0) {
41             Q.push(i);
42         } else if (shopCategory[i] == 1) {
43             shopTour.push_back(i);
44         } else {
45             shopTour.push_back(i);
46             if (Q.empty())
47                 return false;
48             shopTour.push_back(Q.front());
49             Q.pop();
50         }
51     }
52     shopTour.push_back(N - 1);
```

```
53     return shopTour;  
54 }
```

Numbering (numbering)

Problem author: Andrea Ciprietti

Problem preparator: Suneet Mahajan

Given a forest on N nodes, a numbering of it is an assignment of positive integers to each edge of the forest. A numbering is beautiful if, for every node, its edges have the numbers $1, 2, \dots, d$ in some order (where d is the degree of the node).


You are given N positive integers A_0, \dots, A_{N-1} . Determine if there exists a forest on N nodes such that:

- for every $0 \leq i \leq N - 1$, the degree of the node i is A_i ;
- it admits at least one beautiful numbering.

Additionally, if there exists a such a forest, construct an example.

Implementation

You will have to submit a single `.cpp` source file.

 Among this task's attachments you will find a template `numbering.cpp` with a sample implementation.

You have to implement the following function:

```
C++ | variant<bool, vector<pair<int, int>>> find_numbering(int N, vector<int> A);
```

- Integer N represents the number of nodes.
- The array A , indexed from 0 to $N - 1$, contains the values A_0, A_1, \dots, A_{N-1} , where A_i is the degree of the i -th node.
- The function should return either a boolean or an array of pairs of integers.
 - If no valid (satisfying the conditions of the statement) forest exists, the function should return `false`.
 - If a valid forest exists, you have two options:
 - * To be awarded the full score, the procedure should return an array of pairs of integers, representing the edges of a valid forest.
 - * To be awarded a partial score, the procedure should return `true` or any array of integers not describing a valid forest.

The grader will call the function `find_numbering` and will print the following to the output file:

- If the return value is `false`, it will print a single line with the string `NO`.
- If the return value is `true`, it will print a single line with the string `YES`.
- If the return value is an array of pairs of integers of length M , it will print a line with the string `YES`, followed by one line with M , followed by M lines with the pairs of the array.

Sample Grader

The task's directory contains a simplified version of the jury grader, which you can use to test your solution locally. The simplified grader reads the input data from `stdin`, calls the functions that you must implement, and finally writes the output to `stdout`.

The input is made up of 2 lines, containing:

- Line 1: the integer N .
- Line 2: A_0, A_1, \dots, A_{N-1} .

The output is made up of multiple lines, containing the values returned by the function `find_numbering`.

Constraints

- $2 \leq N \leq 10^5$.
- $0 \leq A_i \leq N - 1$.

Scoring

Your program will be tested on a set of test cases grouped by subtask. The score associated to a subtask will be the minimum of the scores obtained in each of the test cases.

- **Subtask 1 [0 points]**: Sample test cases.
- **Subtask 2 [16 points]**: $A_i \leq 2$.
- **Subtask 3 [12 points]**: $A_i \leq 3$.
- **Subtask 4 [16 points]**: Let `count(i)` be the number of occurrences of i in A . You are guaranteed that `count(i) \geq count($i + 1$) + count($i + 2$) + ...` for all $1 \leq i \leq N - 1$.
- **Subtask 5 [10 points]**: $N \leq 12$.
- **Subtask 6 [24 points]**: $N \leq 500$.
- **Subtask 7 [22 points]**: No additional constraints.

For each test case in which a valid forest exists, your solution:

- gets full points if it returns a valid forest.
- gets 50% of the points if it returns `true` or an array that does not describe a valid forest.
- gets 0 points otherwise.

For each test case in which a valid forest does not exist, your solution:

- gets full points if it returns `false`.
- gets 0 points otherwise.

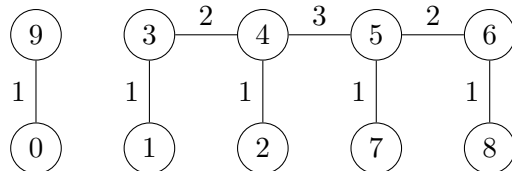
Examples

stdin	stdout
4 1 1 2 1	NO
10 1 1 1 2 3 3 2 1 1 1	YES 8 0 9 1 3 2 4 3 4 4 5 5 6 5 7 6 8

Explanation

In the **first sample case**, we want a valid forest with 4 nodes: 3 with degree 1 and 1 with degree 2. We can show that this is not possible. Suppose such a forest exists, then there should be an edge with number 2 out of the node with degree 2. This edge connects to another node that should have degree at least 2. However, such a node does not exist, since all other nodes have degree 1.

In the **second sample case**, we want a valid forest with 10 nodes: 6 with degree 1, 2 with degree 2 and 2 with degree 3. Such a forest exists and the output is depicted below:



Notice that nodes 4 and 5 have three edges labeled 1, 2 and 3. Furthermore nodes 3 and 6 have two edges labeled 1 and 2. Finally nodes 0, 1, 2, 7, 8 and 9 have one edge labeled 1.

Solution

Let us say a forest (V, E) is beautiful if it admits at least one beautiful numbering, that is, if we can define a mapping $f: E \rightarrow \{1, \dots, |V| - 1\}$ such that for all vertices $v \in V$, we have that the labels on the edges with v as an endpoint are $(1, \dots, d_v)$, where d_v is the degree of vertex v .

Take now a beautiful forest, with its beautiful numbering. Note that a node with degree d will have edges with numbers $1, 2, \dots, d$. This means that it will have an edge connecting it to a node of degree at least d , another node of degree at least $d - 1$, and so on.

Let the largest degree in A be some number D and let us start with an *empty slate*, that is, the forest with nodes $0, \dots, N - 1$ and no edges. Before any of the "hard work", we can carry out two simple checks on A to find some early diagnoses that no forest exists: calculate $S = A_0 + \dots + A_{N-1}$. Any forest with degrees A will have $\frac{S}{2}$ edges, so we can already conclude that no forest exists if S is odd or $\frac{S}{2} > N - 1$. But this will not always be the case and we need to have stronger methods to find more subtle inconsistencies.

We will start by adding in the edges with label D . Then, all nodes with degree D must be paired with each other, since only nodes with degree D or higher (and since D is the maximum, no node will have degree $> D$) will have outgoing edges labelled with a D . If after this pairing is done there is a node left over, then we can conclude that no beautiful forest with degrees A exists.

Let us now include the edges with label $D - 1$. All nodes with degree $D - 1$ or D will have one outgoing edge with this label, and we again have to find a way to pair them up, this time ensuring that no cycles are created in the process. If there is no way to pair them up, we again conclude that this is impossible.

We decrement the label d on the edges we are adding to the forest, pairing up all nodes with degree $\geq d$ ensuring no cycles are created in the process, until we finish with $d = 1$, or reach a point where we conclude that the construction is impossible.

But how can we ensure that the choices we make at each step do not generate *false negatives* in the future? That is, we must pair nodes in each step maintaining the invariant that if a beautiful forest existed in the previous step, then at least one beautiful forest exists after the current step.

We can do this by keeping track of the current connected components in the forest and making greedy choices. As a preprocessing step, we iterate over A and store, for each value of $d = 1, \dots, N - 1$, the indices of the nodes in A that have such a degree in an array `nodesPerDegree`.

We then iterate from $d = N - 1$ to $d = 1$ and find, for each connected component in the current forest, which nodes have degree $\geq d$. We can do so by, after each iteration, copying all nodes in `nodesPerDegree[d]` to `nodesPerDegree[d-1]`. This way, at the start of each iteration, `nodesPerDegree[d]` will already contain the nodes that we need.

We sort the connected components in descending order of the number of nodes with degree $\geq d$ that the component contains. We now proceed to pair these nodes up with edges with label d . In order to do so, we iterate over the list of connected components in this order. We keep at all times the index of the last connected component we have seen and how many unpaired nodes it has. Initially, this number is set to zero for both variables. We wish to maintain the invariant that for all connected components we have already iterated over, either their degree- $\geq d$ nodes have already been paired up, or the connected component is the one we are keeping the index of (and hence the last visited). Whenever we reach a new connected component, we first check if there are unpaired nodes in the last visited component.

If there are, we then add an edge between an unpaired node of the last component and an unpaired node of the current component. This now means the two components have merged into one. We then update the index of the last visited component and the number of unpaired nodes in it, which may now be zero, and hence maintain the invariant after the iteration.

On the other hand, if all nodes with degree $\geq d$ in the last connected component have been paired up already, then by the invariant we can conclude that all degree $\geq d$ nodes we have iterated over are now paired up. We hence simply update the index of the last visited component and the number of unpaired

degree- $\geq d$ nodes to those of the current connected component, and also maintain the invariant in this case.

Once we have finished traversing the list of connected components, either all nodes of degree $\geq d$ have been paired up, in which case we decrement d and continue; or the last component remains unpaired, in which case we conclude no beautiful forest exists. It can be proven that this greedy algorithm is optimal.

It now remains to program it efficiently. For that, we can use the disjoint set union data structure to keep track of current connected components and use small-to-large merging when merging two connected components. With this I mean that when we merge two connected components, the tree of the one with fewer unpaired nodes is hung under the root of the tree of the one with the most unpaired nodes, breaking ties arbitrarily.

[Spoiler alert: the following page contains C++ code for a 100 points solution.]

Accepted Solution

```
1  #include <algorithm>
2  #include <iostream>
3  #include <variant>
4  #include <vector>
5  using namespace std;
6
7  struct dsu {
8      int N;
9      vector<int> par;
10     vector<vector<int>> unpairedPerSet;
11     dsu(int n) {
12         N = n;
13         par = vector<int>(N);
14         for (int i = 0; i < N; i++) {
15             par[i] = i;
16         }
17         unpairedPerSet = vector<vector<int>>(N);
18     }
19
20     int root(int x) {
21         if (par[x] == x)
22             return x;
23         return par[x] = root(par[x]);
24     }
25
26     void mergeTrees(int x, int y) {
27         int parX = root(x);
28         int parY = root(y);
29         if (parX == parY)
30             return;
31         if (unpairedPerSet[parX].size() < unpairedPerSet[parY].size()) {
32             swap(parX, parY);
33         }
34         par[parY] = parX;
35         // the only difference between this implementation of DSU and the canonical
36         // one is that we keep track of all the unpaired nodes in each set
37         for (int &node : unpairedPerSet[parY]) {
38             unpairedPerSet[parX].push_back(node);
39         }
40     }
41
42     void removeEdge(int node) { unpairedPerSet[root(node)].push_back(node); }
43
44     int unpairedInComponent(int node) {
45         return (int)unpairedPerSet[root(node)].size();
46     }
47
48     pair<int, int> mergeUnpaired(int node1, int node2) {
49         int root1 = root(node1);
50         int root2 = root(node2);
51         pair<int, int> edge =
52             make_pair(unpairedPerSet[root1].back(), unpairedPerSet[root2].back());
```

```

53     unpairedPerSet[root1].pop_back();
54     unpairedPerSet[root2].pop_back();
55     mergeTrees(root1, root2);
56     return edge;
57 }
58 };
59
60 bool checkValidNumEdges(int N, vector<int> &A) {
61     long long numEdges = 0;
62     for (int i = 0; i < N; i++) {
63         numEdges += A[i];
64     }
65
66     // the number of edges of any graph equals sum(degrees) / 2
67     // so sum(degrees) must be even
68     if (numEdges & 1) {
69         return false;
70     }
71
72     numEdges /= 2;
73     // numEdges now contains the real number of edges in any beautiful forest with
74     // degrees A
75     return numEdges <= N - 1;
76 }
77
78 variant<bool, vector<pair<int, int>>> find_numbering(int N, vector<int> A) {
79     if (!checkValidNumEdges(N, A)) {
80         return false;
81     }
82
83     dsu components = dsu(N);
84
85     vector<vector<int>> nodesPerDegree(N);
86     for (int i = 0; i < N; i++) {
87         // we know 0 <= A[i] <= N-1 from the constraints, so there is no need to
88         // check this
89         nodesPerDegree[A[i]].push_back(i);
90     }
91
92     vector<pair<int, int>> edges;
93
94     for (int degree = N - 1; degree >= 1; degree--) {
95         if (!nodesPerDegree.size())
96             continue;
97
98         // remove 1 edge from all nodes with current degree
99         // this leaves one unpaired endpoint per node, so we note that down for
100        // later
101        for (int &node : nodesPerDegree[degree]) {
102            nodesPerDegree[degree - 1].push_back(node);
103            components.removeEdge(node);
104        }
105
106        // for each set in the DSU, note how many unpaired nodes it contains

```

```

107 // and the root of the tree that represents the set
108 vector<pair<int, int>> toPair;
109 for (int &node : nodesPerDegree[degree]) {
110     if (components.root(node) == node &&
111         components.unpairedInComponent(node)) {
112         toPair.push_back(make_pair(components.unpairedInComponent(node), node));
113     }
114 }
115
116 // sort these values by descending order of how many unpaired nodes the set
117 // has
118 sort(toPair.rbegin(), toPair.rend());
119 int lastUnpairedInd = 0, leftUnpaired = 0;
120 for (auto [treeUnpaired, treeRoot] : toPair) {
121     if (leftUnpaired > 0) {
122         edges.push_back(components.mergeUnpaired(treeRoot, lastUnpairedInd));
123     }
124     lastUnpairedInd = components.root(treeRoot);
125     leftUnpaired = components.unpairedInComponent(lastUnpairedInd);
126 }
127 if (leftUnpaired) {
128     return false;
129 }
130 }
131 return edges;
132 }

```