Codeforces Round 1019 (Div. 2)

A. Common Multiple

1 second, 256 megabytes

You are given an array of integers $a_1, a_2, \ldots, a_n$. An array $x_1, x_2, \ldots, x_m$ is *beautiful* if there exists an array $y_1, y_2, \ldots, y_m$ such that the elements of y are distinct (in other words, $y_i \neq y_j$ for all $1 \leq i < j \leq m$), and the product of x_i and y_i is the same for all $1 \leq i \leq m$ (in other words, $x_i \cdot y_i = x_j \cdot y_j$ for all $1 \leq i < j \leq m$).

Your task is to determine the maximum size of a subsequence* of array a that is beautiful.

*A sequence b is a subsequence of a sequence a if b can be obtained from a by the deletion of several (possibly, zero or all) element from arbitrary positions.

Input

Each test contains multiple test cases. The first line contains the number of test cases t ($1 \leq t \leq 500$). The description of the test cases follows.

The first line of each test case contains a single integer n ($1 \leq n \leq 100$) — the length of the array a.

The second line of each test case contains n integers $a_1, a_2, \ldots, a_n$ ($1 \leq a_i \leq n$) — the elements of array a.

Note that there are **no** constraints on the sum of n over all test cases.

Output

For each test case, output the maximum size of a subsequence of array a that is beautiful.

input
3 3 1 2 3 5 3 1 4 1 5 1 1

output
3 4 1

In the first test case, the entire array $a = [1, 2, 3]$ is already beautiful. A possible array y is $[6, 3, 2]$, which is valid since the elements of y are distinct, and $1 \cdot 6 = 2 \cdot 3 = 3 \cdot 2$.

In the second test case, the subsequence $[3, 1, 4, 5]$ is beautiful. A possible array y is $[20, 60, 15, 12]$. It can be proven that the entire array $a = [3, 1, 4, 1, 5]$ is not beautiful, so the maximum size of a subsequence of array a that is beautiful is 4.

B. Binary Typewriter

1.5 seconds, 256 megabytes

You are given a binary string s of length n and a typewriter with two buttons: `0` and `1`. Initially, your finger is on the button `0`. You can do the following two operations:

1. Press the button your finger is currently on. This will type out the character that is on the button.
2. Move your finger to the other button. If your finger is on button `0`, move it to button `1`, and vice versa.

The *cost* of a binary string is defined as the minimum number of operations needed to type the entire string.

Before typing, you may reverse at most one substring* of s. More formally, you can choose two indices $1 \leq l \leq r \leq n$, and reverse the substring $s_{l \ldots r}$, resulting in the new string $s_1 s_2 \ldots s_{l-1} s_r s_{r-1} \ldots s_l s_{r+1} \ldots s_n$.

Your task is to find the minimum possible cost among all strings obtainable by performing at most one substring reversal on s.

*A string a is a substring of a string b if a can be obtained from b by the deletion of several (possibly, zero or all) characters from the beginning and several (possibly, zero or all) characters from the end.

Input

Each test contains multiple test cases. The first line contains the number of test cases t ($1 \leq t \leq 10^4$). The description of the test cases follows.

The first line of each test case contains a single integer n ($1 \leq n \leq 2 \cdot 10^5$) — the length of the binary string s.

The second line of each test case contains a binary string $s_1 s_2 \ldots s_n$ ($s_i = 0$ or $s_i = 1$) — the characters of the binary string s.

It is guaranteed that the sum of n over all test cases does not exceed $2 \cdot 10^5$.

Output

For each test case, output the minimum cost of string s after performing at most one substring reversal.

input
6 3 000 3 111 3 011 3 100 5 10101 19 1101010010011011100

output
3 4 4 4 8 29

In the first test case, we can choose not to reverse any substrings. We can do operation 1 three times to type `000`.

In the second test case, we can choose not to reverse any substrings. We can do operation 2 to move our finger to button `1`. Then, we do operation 1 three times to type `111`.

In the third test case, we can choose not to reverse any substring. We can do operation 1 to type `0`. Then, we do operation 2 to move our finger to button `1`. Finally, we do operation 1 two times to type `11`, resulting in the final string `011` using only 4 operations.

In the fourth test case, we can reverse the substring $s_{1 \ldots 3}$, resulting in the string `001`. We can do operation 1 two times to type `00`. Then we do operation 2 to move our finger to button `1`. Finally, we do operation 1 once to type `1`, resulting in the final string `001` using only 4 operations.

In the fifth test case, we can reverse the substring $s_{2 \ldots 3}$, resulting in the string `11001`. The cost of the string is 8 as we can do the following sequence of operations:

- Do operation 2 to move our finger to button `1`.

- Do operation 1 two times to type `11`.
- Do operation 2 to move our finger to button `0`.
- Do operation 1 two times to type `00`.
- Do operation 2 to move our finger to button `1`.
- Do operation 1 once to type `1`.

In the sixth test case, we can reverse the substring $s_{5 \ldots 17}$, resulting in the string `1101`**`11101100100100`**`00`. It can be proven that the minimum number of operations needed to type the binary string is 29.

C. Median Splits

2 seconds, 256 megabytes

The median of an array $b_1, b_2, \ldots b_m$, written as $\operatorname{med}(b_1, b_2, \ldots, b_m)$, is the $\lceil \frac{m}{2} \rceil$-th* smallest element of array b.

You are given an array of integers $a_1, a_2, \ldots, a_n$ and an integer k. You need to determine whether there exists a pair of indices $1 \leq l < r < n$ such that:

$$\operatorname{med}(\operatorname{med}(a_1, a_2, \ldots, a_l), \operatorname{med}(a_{l+1}, a_{l+2}, \ldots, a_r), \operatorname{med}(a_{r+1}, a_{r+2}, \ldots, a_n)) \leq k$$

In other words, determine whether it is possible to split the array into three contiguous subarrays† such that the median of the three subarray medians is less than or equal to k.

* $\lceil x \rceil$ is the ceiling function which returns the least integer greater than or equal to x.

† An array x is a subarray of an array y if x can be obtained from y by the deletion of several (possibly, zero or all) elements from the beginning and several (possibly, zero or all) elements from the end.

Input

Each test contains multiple test cases. The first line contains the number of test cases t ($1 \leq t \leq 10^4$). The description of the test cases follows.

The first line of each test case contains two integers n and k ($3 \leq n \leq 2 \cdot 10^5$, $1 \leq k \leq 10^9$) — the length of the array a and the constant k.

The second line of each test case contains n integers $a_1, a_2, \ldots, a_n$ ($1 \leq a_i \leq 10^9$) — the elements of the array a.

It is guaranteed that the sum of n over all test cases does not exceed $2 \cdot 10^5$.

Output

For each testcase, output "`YES`" if such a split exists, and "`NO`" otherwise.

You can output the answer in any case (upper or lower). For example, the strings "`yEs`", "`yes`", "`Yes`", and "`YES`" will be recognized as positive responses.

input

```
6
3 2
3 2 1
3 1
3 2 1
6 3
8 5 3 1 6 4
8 7
10 7 12 16 3 15 6 11
6 8
7 11 12 4 9 17
3 500000000
1000 1000000000 1000
```

output

```
YES
NO
NO
YES
YES
YES
```

In the first and second test case, the only possible partition of the array into three contiguous subarrays is $[3]$, $[2]$, $[1]$. Their respective medians are 3, 2, and 1. The median of the three subarray medians is $\operatorname{med}(3, 2, 1) = 2$. Therefore, the answer for the first test case is "`YES`" since $2 \leq 2$, while the answer for the second test case is "`NO`" since $2 > 1$.

In the third test case, it can be proven that no partition satisfies the constraints.

In the fourth test case, one of the partitions satisfying the constraints is $[10, 7]$, $[12, 16, 3, 15]$, $[6, 11]$. The respective medians of subarrays are 7, 12, and 6. The median of the three subarray medians is $\operatorname{med}(7, 12, 6) = 7 \leq k$, hence this partition satisfies the constraints.

In the fifth test case, one of the partitions satisfying the constraints is $[7, 11]$, $[12, 4]$, $[9, 17]$. The respective medians of the subarrays are 7, 4, and 9. The median of the three subarray medians is $\operatorname{med}(7, 4, 9) = 7 \leq k$, hence this partition satisfies the constraints.

In the sixth test case, the only possible partition of the array into three contiguous subarrays is $[1000]$, $[10^9]$, $[1000]$. The respective medians of the subarrays are 1000, 10^9, and 1000. The median of the three subarray medians is $\operatorname{med}(1000, 10^9, 1000) = 1000 \leq k$, hence this partition satisfies the constraints.

D. Local Construction

2 seconds, 256 megabytes

An element b_i ($1 \leq i \leq m$) in an array $b_1, b_2, \ldots, b_m$ is a local minimum if at least one of the following holds:

- $2 \leq i \leq m - 1$ and $b_i < b_{i-1}$ and $b_i < b_{i+1}$, or
- $i = 1$ and $b_1 < b_2$, or
- $i = m$ and $b_m < b_{m-1}$.

Similarly, an element b_i ($1 \leq i \leq m$) in an array $b_1, b_2, \ldots, b_m$ is a local maximum if at least one of the following holds:

- $2 \leq i \leq m - 1$ and $b_i > b_{i-1}$ and $b_i > b_{i+1}$, or
- $i = 1$ and $b_1 > b_2$, or
- $i = m$ and $b_m > b_{m-1}$.

Note that local minima and maxima are not defined for arrays with only one element.

There is a hidden permutation* p of length n. The following two operations are applied to permutation p alternately, starting from operation 1, until there is only one element left in p:

- **Operation 1** — remove all elements of p which are **not** local minima.
- **Operation 2** — remove all elements of p which are **not** local maxima.

More specifically, operation 1 is applied during every odd iteration, and operation 2 is applied during every even iteration, until there is only one element left in p.

For each index i ($1 \leq i \leq n$), let a_i be the iteration number that element p_i is removed, or -1 if it was never removed.

It can be proven that there will be only one element left in p after at most $\lceil \log_2 n \rceil$ iterations (in other words, $a_i \leq \lceil \log_2 n \rceil$).

You are given the array $a_1, a_2, \ldots, a_n$. Your task is to construct any permutation p of n elements that satisfies array a.

*A permutation of length n is an array consisting of n distinct integers from 1 to n in arbitrary order. For example, $[2, 3, 1, 5, 4]$ is a permutation, but $[1, 2, 2]$ is not a permutation (2 appears twice in the array), and $[1, 3, 4]$ is also not a permutation ($n = 3$ but there is 4 in the array).

Input

Each test contains multiple test cases. The first line contains the number of test cases t ($1 \leq t \leq 10^4$). The description of the test cases follows.

The first line of each test case contains a single integer n ($2 \le n \le 2 \cdot 10^5$) — the number of elements in permutation p.

The second line of each test case contains n integers $a_1, a_2, \ldots, a_n$ ($1 \le a_i \le \lceil \log_2 n \rceil$ or $a_i = -1$) — the iteration number that element p_i is removed.

It is guaranteed that the sum of n over all test cases does not exceed $2 \cdot 10^5$.

It is guaranteed that there exists at least one permutation p that satisfies array a.

Output

For each test case, output n integers representing the elements of the permutation satisfying array a.

If there are multiple solutions, you may output any of them.

input
7 3 1 1 -1 5 1 -1 1 2 1 8 3 1 2 1 -1 1 1 2 7 1 1 1 -1 1 1 1 5 1 1 1 1 -1 5 -1 1 1 1 1 5 -1 1 2 1 2

output
3 2 1 4 3 5 1 2 6 7 2 4 3 8 5 1 6 5 2 1 3 4 7 5 4 3 2 1 1 2 3 4 5 4 5 2 3 1

In the first test case, operations will be applied to permutation $[3, 2, 1]$ as follows:

1. The only local minimum in $[3, 2, 1]$ is 1. Hence, elements 3 and 2 are removed. There is only one remaining element; hence the process terminates.

This satisfies array $a = [1, 1, -1]$ as both p_1 and p_2 were removed on iteration number 1, while p_3 was not removed.

In the second test case, operations will be applied to permutation $p = [4, 3, 5, 1, 2]$ as follows:

1. The local minima in $[4, 3, 5, 1, 2]$ are 3 and 1. Hence, elements 4, 5, and 2 are removed.
2. The only local maximum in $[3, 1]$ is 3. Hence, element 1 is removed. There is only one remaining element; hence the process terminates.

This satisfies array $a = [1, -1, 1, 2, 1]$ as elements $p_1 = 4$, $p_3 = 5$, and $p_5 = 2$ were removed on iteration 1, element $p_4 = 1$ was removed on iteration 2, and element $p_2 = 3$ was not removed.

In the third test case, operations will be applied on permutation $[6, 7, 2, 4, 3, 8, 5, 1]$ as follows:

1. The local minima in $[6, 7, 2, 4, 3, 8, 5, 1]$ are 6, 2, 3, and 1. Hence, elements 7, 4, 8, and 5 are removed.
2. The local maxima in $[6, 2, 3, 1]$ are 6 and 3. Hence, elements 2 and 1 are removed.
3. The only local minimum in $[6, 3]$ is 3. Hence, element 6 is removed. There is only one remaining element; hence the process terminates.

In the fourth test case, one permutation satisfying the constraints is $[6, 5, 2, 1, 3, 4, 7]$. 1 is the only local minimum, so only it will stay after the first iteration. Note that there are other valid permutations; for example, $[6, 4, 3, 1, 2, 5, 7]$ would also be considered correct.

E. Keep the Sum

2.5 seconds, 256 megabytes

You are given an integer k and an array a of length n, where each element satisfies $0 \le a_i \le k$ for all $1 \le i \le n$. You can perform the following operation on the array:

- Choose two distinct indices i and j ($1 \le i, j \le n$ and $i \ne j$) such that $a_i + a_j = k$.
- Select an integer x satisfying $-a_j \le x \le a_i$.
- Decrease a_i by x and increase a_j by x. In other words, update $a_i := a_i - x$ and $a_j := a_j + x$.

Note that the constraints on x ensure that all elements of array a remain between 0 and k throughout the operations.

Your task is to determine whether it is possible to make the array a non-decreasing* using the above operation. If it is possible, find a sequence of at most $3n$ operations that transforms the array into a non-decreasing one.

It can be proven that if it is possible to make the array non-decreasing using the above operation, there exists a solution that uses at most $3n$ operations.

* An array $a_1, a_2, \ldots, a_n$ is said to be non-decreasing if for all $1 \le i \le n - 1$, it holds that $a_i \le a_{i+1}$.

Input

Each test contains multiple test cases. The first line contains the number of test cases t ($1 \le t \le 10^4$). The description of the test cases follows.

The first line of each test case contains two integers, n and k ($4 \le n \le 2 \cdot 10^5$, $1 \le k \le 10^9$) — the length of the array a and the required sum for the operation.

The second line of each test case contains n integers $a_1, a_2, \ldots, a_n$ ($0 \le a_i \le k$) — the elements of array a.

It is guaranteed that the sum of n over all test cases does not exceed $2 \cdot 10^5$.

Output

For each test case, output -1 if it is not possible to make the array non-decreasing using the operation.

Otherwise, output the number of operations m ($0 \le m \le 3n$). On each of the next m lines, output three integers i, j, and x representing an operation where a_i is decreased by x and a_j is increased by x.

Note that you are **not** required to minimize the number of operations. If there are multiple solutions requiring at most $3n$ operations, you may output any.

input
4 5 100 1 2 3 4 5 5 6 1 2 3 5 4 5 7 7 1 5 3 1 10 10 2 5 3 2 7 3 1 8 4 0

output

```
0
1
4 1 1
-1
6
1 8 2
3 5 2
5 7 3
5 9 3
8 10 5
2 10 4
```

In the first test case, the array is already non-decreasing, so we do not need to perform any operations.

In the second test case, we can perform an operation with $i = 4$, $j = 1$, and $x = 1$. a_4 decreases by 1 to become $5 - 1 = 4$ while a_1 increases by 1 to become $1 + 1 = 2$. After the operation, the array becomes $[2, 2, 3, 4, 4]$, which is non-decreasing.

Note that there are other ways to make the array non-decreasing, all of which would be considered correct as long as they do not use more than $3 \cdot n = 15$ operations.

In the third test case, it is not possible to make the array non-decreasing. This is because there are no distinct pairs of indices i and j where $a_i + a_j = 7$, so no operation can be done on the array.

In the fourth test case, the array is transformed as follows:

1. $[\mathbf{0}, 5, 3, 2, 7, 3, 1, \mathbf{10}, 4, 0]$
2. $[0, 5, \mathbf{1}, 2, \mathbf{9}, 3, 1, 10, 4, 0]$
3. $[0, 5, 1, 2, \mathbf{6}, 3, \mathbf{4}, 10, 4, 0]$
4. $[0, 5, 1, 2, \mathbf{3}, 3, 4, 10, \mathbf{7}, 0]$
5. $[0, 5, 1, 2, 3, 3, 4, \mathbf{5}, 7, \mathbf{5}]$
6. $[0, \mathbf{1}, 1, 2, 3, 3, 4, 5, 7, \mathbf{9}]$

F. Maximize Nor

4 seconds, 1024 megabytes

The bitwise nor* of an array of k-bit integers $b_1, b_2, \ldots, b_m$ can be computed by calculating the bitwise nor cumulatively from left to right. More formally, $\operatorname{nor}(b_1, b_2, \ldots, b_m) = \operatorname{nor}(\operatorname{nor}(b_1, b_2, \ldots, b_{m-1}), b_m)$ for $m \geq 2$, and $\operatorname{nor}(b_1) = b_1$.

You are given an array of k-bit integers $a_1, a_2, \ldots, a_n$. For each index i ($1 \leq i \leq n$), find the maximum bitwise nor among all subarrays† of a containing index i. In other words, for each index i, find the maximum value of $\operatorname{nor}(a_l, a_{l+1}, \ldots, a_r)$ among all $1 \leq l \leq i \leq r \leq n$.

* The logical nor of two boolean values is 1 if both values are 0, and 0 otherwise. The bitwise nor of two k-bit integers is calculated by performing the logical nor operation on each pair of the corresponding bits.

For example, let us compute $\operatorname{nor}(2, 6)$ when they are represented as 4-bit numbers. In binary, 2=0010_2 and $6 = 0110_2$. Therefore, $\operatorname{nor}(2, 6) = 1001_2 = 9$ as by performing the logical nor operations from left to right, we have:

- $\operatorname{nor}(0, 0) = 1$
- $\operatorname{nor}(0, 1) = 0$
- $\operatorname{nor}(1, 0) = 0$
- $\operatorname{nor}(1, 1) = 0$

Note that if 2 and 6 were represented as 3-bit integers instead, then $\operatorname{nor}(2, 6) = 1$.

† An array x is a subarray of an array y if x can be obtained from y by the deletion of several (possibly, zero or all) elements from the beginning and several (possibly, zero or all) elements from the end.

Input

Each test contains multiple test cases. The first line contains the number of test cases t ($1 \leq t \leq 10^4$). The description of the test cases follows.

The first line of each test case contains two integers n and k ($1 \leq n \leq 10^5$, $1 \leq k \leq 17$) — the number of elements in the array and the number of bits of the array elements.

The second line of each test case contains n integers $a_1, a_2, \ldots, a_n$ ($0 \leq a_i \leq 2^k - 1$) — the elements of array a.

It is guaranteed that the sum of n over all test cases does not exceed 10^5.

Output

For each test case, output n integers, the i-th of which is the maximum bitwise nor among all subarrays of a containing index i.

input

```
2
2 2
1 3
5 3
1 7 4 6 2
```

output

```
1 3
5 7 5 6 5
```

In the first test case, subarrays that have index 1 are $[1]$ and $[1, 3]$. The values of their bitwise nor are 1 and 0 respectively. Hence, the answer for index 1 is 1. Subarrays that have index 2 are $[3]$ and $[1, 3]$. The values of their bitwise nor are 3 and 0 respectively. Hence, the answer for index 2 is 3.

In the second test case:

- For $i = 1$, the subarray with maximum bitwise nor is $[a_1, a_2, a_3, a_4, a_5] = [1, 7, 4, 6, 2]$, $\operatorname{nor}(1, 7, 4, 6, 2) = 5$
- For $i = 2$, the subarray with maximum bitwise nor is $[a_2] = [7]$, $\operatorname{nor}(7) = 7$
- For $i = 3$, the subarray with maximum bitwise nor is $[a_1, a_2, a_3, a_4, a_5] = [1, 7, 4, 6, 2]$, $\operatorname{nor}(1, 7, 4, 6, 2) = 5$
- For $i = 4$, the subarray with maximum bitwise nor is $[a_4] = [6]$, $\operatorname{nor}(6) = 6$
- For $i = 5$, the subarray with maximum bitwise nor is $[a_1, a_2, a_3, a_4, a_5] = [1, 7, 4, 6, 2]$, $\operatorname{nor}(1, 7, 4, 6, 2) = 5$