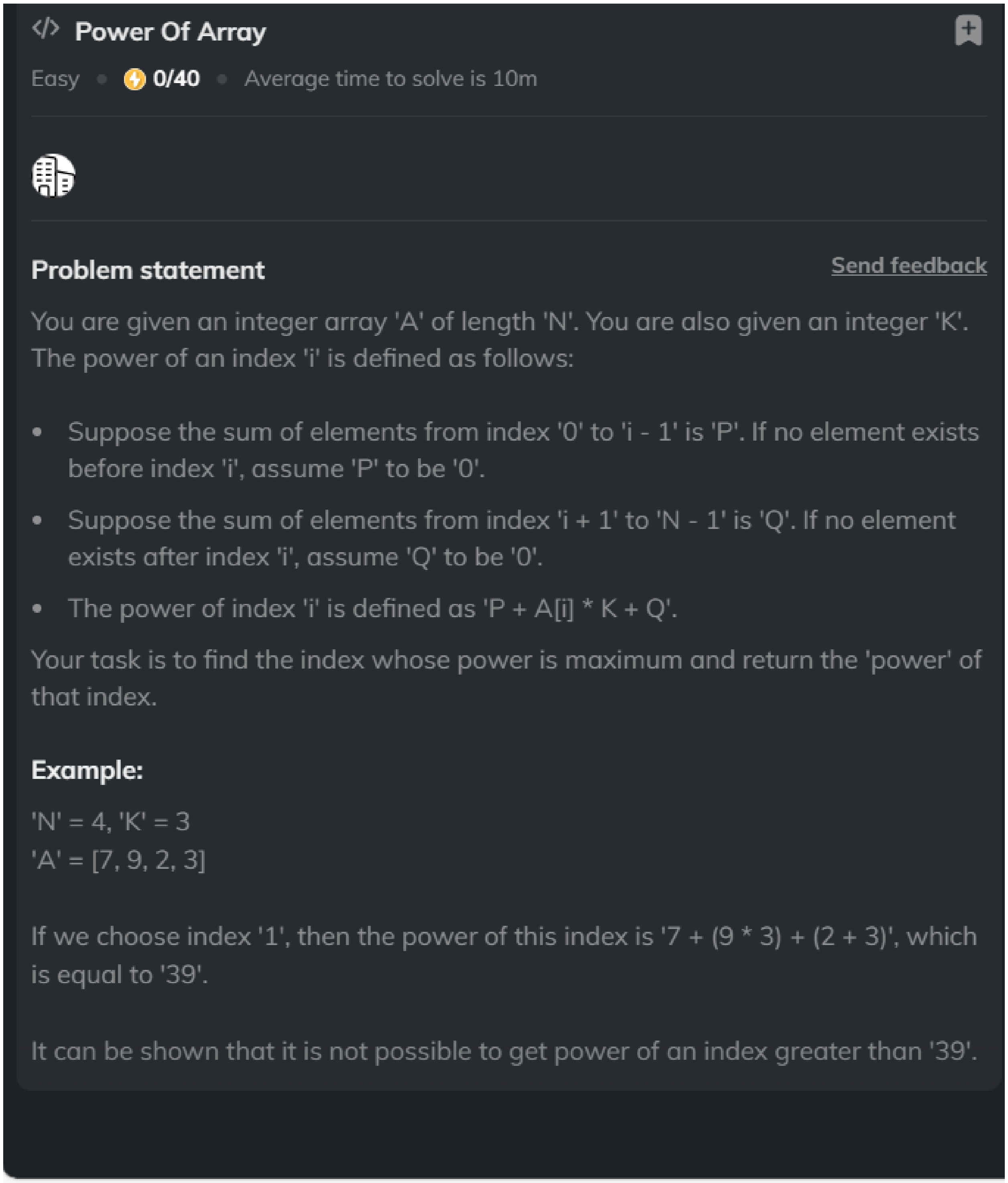


Power Of Array

Easy • 0/40 • Average time to solve is 10m

Problem statement

Send feedback

You are given an integer array 'A' of length 'N'. You are also given an integer 'K'. The power of an index 'i' is defined as follows:

- Suppose the sum of elements from index '0' to 'i - 1' is 'P'. If no element exists before index 'i', assume 'P' to be '0'.
- Suppose the sum of elements from index 'i + 1' to 'N - 1' is 'Q'. If no element exists after index 'i', assume 'Q' to be '0'.
- The power of index 'i' is defined as 'P + A[i] * K + Q'.

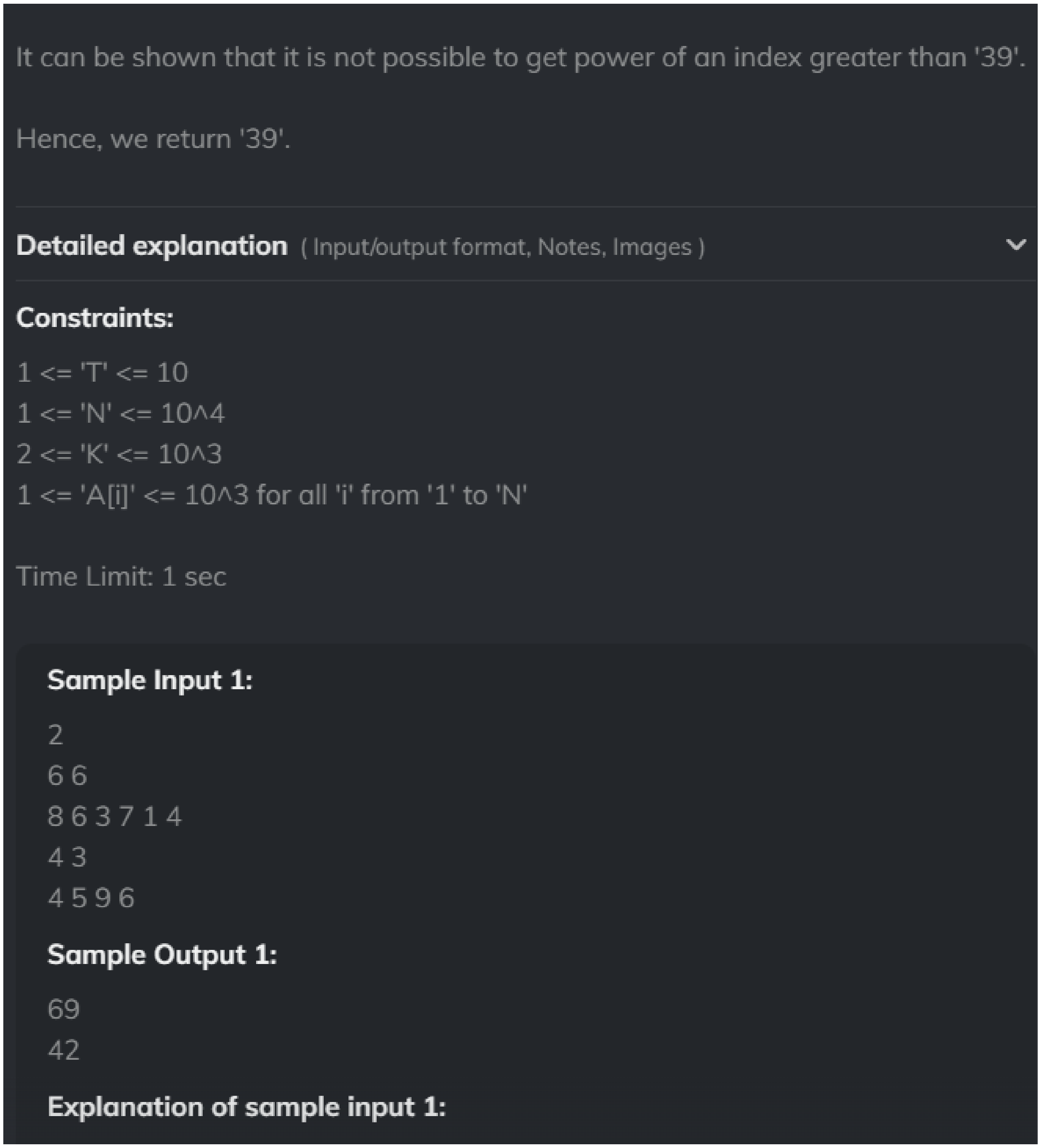
Your task is to find the index whose power is maximum and return the 'power' of that index.

Example:

'N' = 4, 'K' = 3
'A' = [7, 9, 2, 3]

If we choose index '1', then the power of this index is '7 + (9 * 3) + (2 + 3)', which is equal to '39'.

It can be shown that it is not possible to get power of an index greater than '39'.


It can be shown that it is not possible to get power of an index greater than '39'.

Hence, we return '39'.

Detailed explanation (Input/output format, Notes, Images)

Constraints:

1 <= 'T' <= 10
1 <= 'N' <= 10^4
2 <= 'K' <= 10^3
1 <= 'A[i]' <= 10^3 for all 'i' from '1' to 'N'

Time Limit: 1 sec

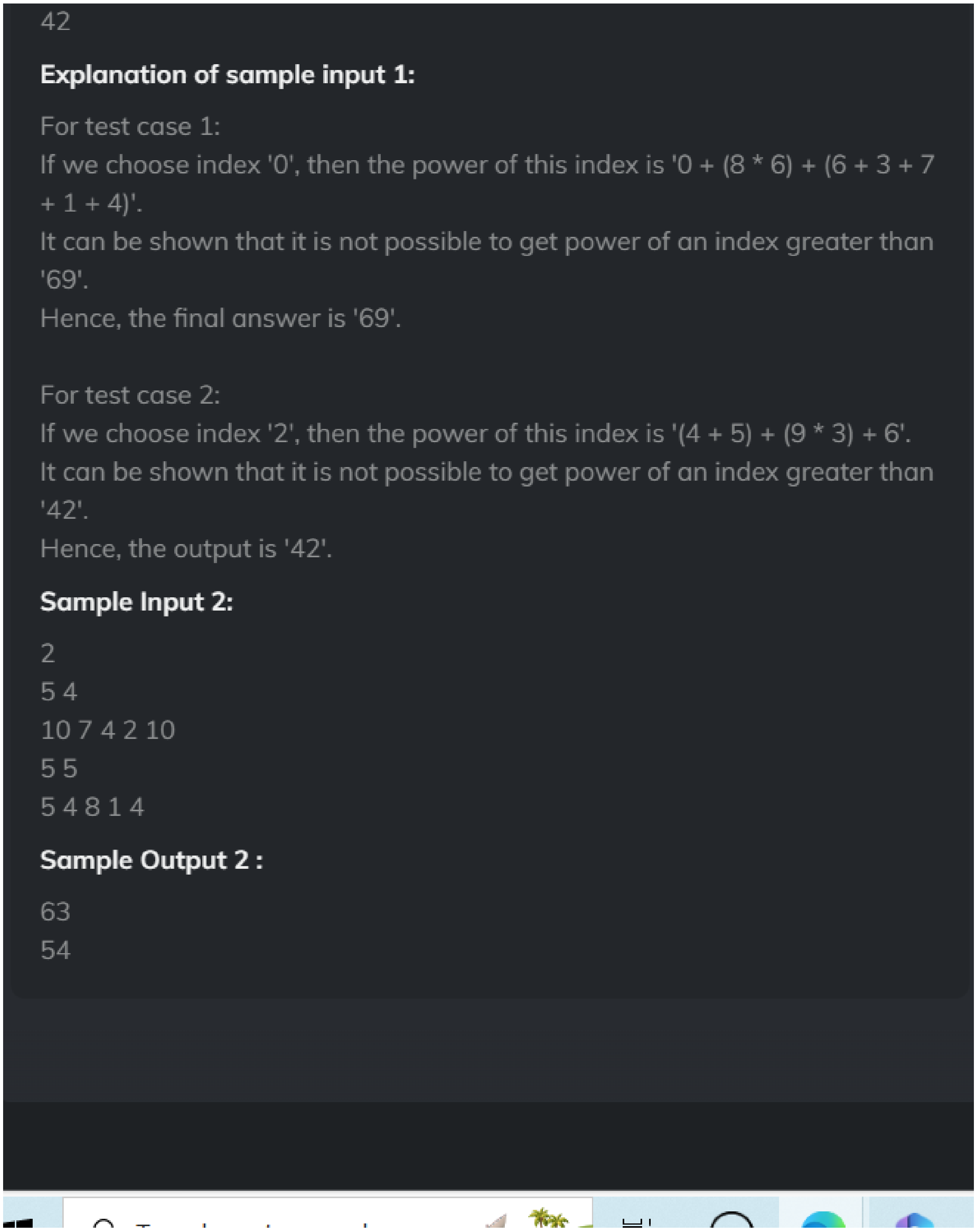
Sample Input 1:

```
2
6 6
8 6 3 7 1 4
4 3
4 5 9 6
```

Sample Output 1:

```
69
42
```

Explanation of sample input 1:



42

Explanation of sample input 1:

For test case 1:
If we choose index '0', then the power of this index is '0 + (8 * 6) + (6 + 3 + 7 + 1 + 4)'.
It can be shown that it is not possible to get power of an index greater than '69'.
Hence, the final answer is '69'.

For test case 2:
If we choose index '2', then the power of this index is '(4 + 5) + (9 * 3) + 6'.
It can be shown that it is not possible to get power of an index greater than '42'.
Hence, the output is '42'.

Sample Input 2:

```
2
5 4
10 7 4 2 10
5 5
5 4 8 1 4
```

Sample Output 2 :

```
63
54
```

Approach 1

Greedy

Approach:

- The problem can be changed to 'choose one number from the array, multiply it by 'K', and add the rest of the elements to it'.
- The element that should be multiplied by 'K' must be the maximum element of the array 'A'.
    - Suppose we have an array 'A', in which we consider two elements 'A1' and 'A2'.
    - Suppose that 'A1' is the maximum element of the array 'A' while 'A2' is not.
    - So, we can say that 'S1' = 'Sum of elements of array 'A' (other than 'A1' and 'A2') + A1 * K + A2'.
    - And 'S2' = 'Sum of elements of array 'A' (other than 'A1' and 'A2') + A2 * K + A1'.
    - If we subtract 'S2 'from 'S1', we get '(K - 1) * A1 - (K - 1) * A2'.
    - Which can be written as '(K - 1) * (A1 - A2)'.
    - We know that (K - 1) is greater than '1' and so is (A1 - A2).
    - This means that 'S1' is greater than 'S2', and hence, it's better to multiply 'K' with the maximum element.

Algorithm:

- Declare two integer variables 'sum' and 'mx' both assigned to '0' and '-1', respectively.
- Iterate from '0' to ‘N-1’ using the variable 'i':
    - Assign 'mx' to the maximum of 'A[i]' and 'mx'.
    - Add 'A[i]' to 'sum'.
- Declare an integer variable 'ans'.
- Assign 'ans' to 'mx * K + (sum - mx)'.
- Return 'ans'.

Time Complexity

O(N), where 'N' is the length of the array 'A'.

We will iterate over the array 'A' one time.

Thus, the overall time complexity is O(N).

Space Complexity

O(1).

We are using constant variables to store the answer.

Thus, the overall space complexity is O(1).

```
/*
    Time complexity: O( N )
    Space complexity: O( 1 )

    where 'N' is the number of elements in the array 'A'.
*/

int powerOfArray(int n, int k, vector<int>& a) {

    // Declare integer variables 'mx' and 'sum'.
    int sum = 0, mx = -1;
    for (int i = 0; i < n; ++i) {
        sum += a[i];

        // Maximizing 'mx'.
        mx = max(mx, a[i]);
    }

    return sum - mx + mx * k;
}
```

APProach 2

The problem is straightforward, centering around the identification of the index with the maximum element in the array. The solution entails computing the sum of all array elements and subsequently subtracting the value of the maximum element from the calculated sum. Code:

```
int powerOfArray(int n, int k, vector<int> &a) {

  // P + A[i] * K + Q
  int index = 0;

  // Find the largest element index
  for (int i = 0; i < n; i++) {
    if (a[i] > a[index])
      index = i;
  }

  // Find the sum of the whole array
  int s = accumulate(a.begin(), a.end(),0);
  int p = a[index] * k;


  // Subbtract the largest element
  return s + p - a[index];
}
```

Hard Sort

Moderate • 0/80

Problem statement

Send feedback

You are given an array 'A' consisting of 'N' integers from '1' to 'N'. You want to sort this array in non-decreasing order by performing a series of operations.

Here's how an operation works:

- You start with an integer 'X', initially set to '1'.
- In each operation, you select an index 'i' (where '0' <= 'i' < 'N') such that the value of 'A[i]' equals 'X'.
- You then increment the value at 'A[i]' by 'X', and subsequently increment 'X' by '1'.

You can perform any number of operations you want.

Note: you can not perform more operations if there is no element with a value equal to 'X' in the array.

Your task is to return '1' if it is possible to sort the array in non-decreasing order. Otherwise, return '0'.

Example:

N = 4
A = [1, 2, 4, 3]
Initially, 'X' is equal to '1'.
In the first operation, choose 'i' equal to '0', So after performing an operation, 'A' and 'X' are ['2', '2', '4', '3'] and '2', respectively.
In the second operation, choose 'i' equal to '1', So after performing an operation, 'A' and 'X' are ['2', '4', '4', '3'] and '3', respectively.
In the third operation, choose 'i' equal to '3', So after performing an operation, 'A' and 'X' are ['2', '4', '4', '6'] and '4', respectively.
We can see that array 'A' is sorted in non-decreasing order.
So, the answer for this case is '1'.

Detailed explanation (Input/output format, Notes, Images)

Constraints:

1 <= 'N' <= 10^5
1 <= 'A[i]' <= N

Time limit: 1 sec

Sample input 1:

```
2
4
1 3 1 4
4
4 4 4 2
```

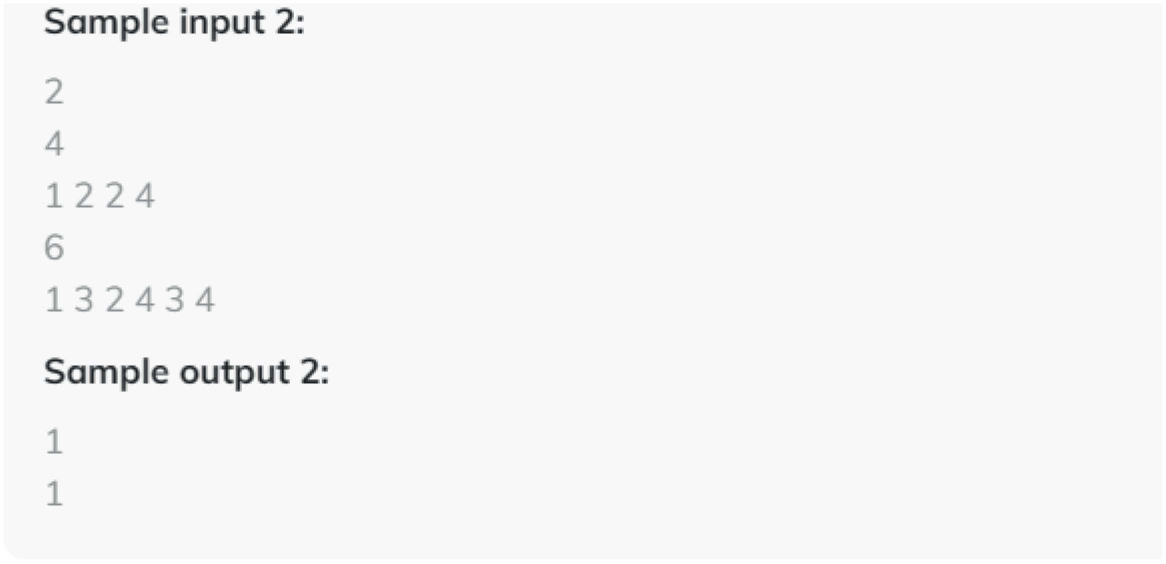
Sample output 1:

```
1
0
```

Explanation of sample input 1:

For the test case 1:
Initially, 'X' is equal to '1'.
In the first operation, choose 'i' equal to '2', So after performing an operation, 'A' and 'X' are ['1', '3', '2', '4'] and '2', respectively.
In the second operation, choose 'i' equal to '2', So after performing an operation, 'A' and 'X' are ['1', '3', '4', '4'] and '3', respectively.
We can see that array 'A' is sorted in non-decreasing order.
So, the answer for this case is '1'.

For the test case 2:
Initially, 'X' is equal to '1'.
We can see that we cannot perform any operation as no element is equal to 'X' in the array, and the array is not sorted in non-decreasing order.
So, the answer for this case is '0'.



Sample input 2:

```
2
4
1 2 2 4
6
1 3 2 4 3 4
```

Sample output 2:

```
1
1
```

Approach 1

Greedy

Approach:

- When multiple elements are equal to 'X' in the array, we must always choose the last element to perform the operation. The proof for the same is given below.
- Suppose that we are operating on index 'i'. So before performing an operation 'A[i]' is equal to 'X'.
- Suppose that there is also an index 'last' greater than 'i' such that 'A[last]' is also equal to 'X'.
- After performing the operation:
    - 'A[i]' and 'A[last] are 'X * 2' and 'X', respectively. We can see that now 'A[last]' will always be smaller than 'A[i]' because the value of 'A[last]' can no longer be updated because the value of 'X' has now become 'X+1'.
    - Hence it is impossible to sort an array if we update an index 'i' less than 'last'.
- Now, our task is easy as we are clear about the index on which we are required to perform an operation.
- To find the last occurrence of 'X' every time, we can store the last index for each value from '1' to 'N'. We also need to maintain the last index while performing the operations.
- Also, we need to check whether the array is sorted after every operation.
- We can do that by maintaining the count of indices 'i' from '1' to 'N - 1' such that 'A[i]' >= 'A[i - 1]'. The array is sorted when the count is 'N - 1'.
- Also, note that we can perform operations at most 'N + 1' times because all the elements are less than or equal to 'N', and there can not be any odd element greater than 'N' even after performing the first 'N' operations.

- If we will find the array sorted while performing the first 'N + 1' operations, then the answer is '1'. Otherwise, the answer is '0'.

Algorithm:

```
● Initialize a vector of integer 'lastIndex' with all elements equal to '-1', which will store the
  last index for each value.
● Initialize a vector of boolean 'isGoodIndex' with all elements equal to 'true', which
  maintains good indices.
● Initialize an integer 'goodCount' equal to 'N - 1' to store the number of good indices.
● For 'i' in the range from '0' to 'N - 1':
    ○ If i is greater than '0' and 'A[i]' is less than 'A[i - 1]':
        ■ Update 'isGoodIndex[i]' to 'false'.
        ■ Decrement 'goodCount' by '1'.
    ○ Update 'lastIndex[A[i]]' to 'i'.
● For 'X' in the range from '1' to 'N + 1':
    ○ Initialize an integer 'index' equal to 'lastIndex[X]'.
    ○ If 'index' is equal to '-1':
        ■ Break the loop.
    ○ Increment 'A[index]' to 'X'.
    ○ If  '2 * X' is less than or equal to 'N + 1':
        ■ Update 'lastIndex[2 * X]' to maximum of 'lastIndex[2 * X]' and 'index'.
    ○ If 'index' is greater than '0', 'A[index]' is greater than or equal to 'A[index - 1]', and
      'isGoodIndex[index]' is 'false':
        ■ Increment 'goodCount' by '1'.
        ■ Update 'isGoodIndex[index]' to 'true'.
    ○ If 'index + 1' is less than 'N', 'A[index + 1]' is less than 'A[index]', and
      'isGoodIndex[index + 1]' is 'true':
        ■ Decrement 'goodCount' by '1'.
        ■ Update 'isGoodIndex[index + 1]' to 'false'.
    ○ If 'goodCount' is equal to 'N - 1':
        ■ Return '1'.
● Return '0'.
```

Time Complexity

O(N), where 'N' denotes the length of the array 'A'.

We are traversing the array to store the last index for each value, which is of order O(N).

Also, we are performing 'N + 1' operations in the worst case, and each operation is performed in constant time. The time complexity for the same is of order O(N).

So, the overall time complexity is of order O(N).

Space Complexity

O(N), where 'N' denotes the length of the array 'A'.

We are storing the last index for each value and also storing boolean values for each index from '0' to 'N - 1', which is of order O(N).

So, the overall space complexity is of order O(N).

Magic Portals

Hard • 0/120

Problem statement

Send feedback

Alice was bored these days, so she went to Magic Street to make herself happy. Magic Street can be represented as a one-dimensional coordinate system.

Initially, Alice was standing on the coordinate '0'. In one second, she can move to the next or previous coordinate of the coordinate at which she is standing. In other words, if she is on coordinate 'X', she can move to the coordinates 'X + 1' or 'X - 1' in one second.

She found 'N' two-directional magic portals in the street in different places, numbered from '0' to 'N-1'. Portal 'i' can be represented by two integers: 'portals[i][0]' and 'portals[i][1]'. She can use the portal 'i' to move from the coordinate 'portals[i][0]' to 'portals[i][1]' or 'portals[i][1]' to 'portals[i][0]' instantly.

She wants to explore all the portals by using them exactly once in less time, as she is late.
Your task is to determine the minimum time required in seconds so that Alice can use all the portals exactly once.

Example:

N = 3
portals = [[1, 5], [2, 10], [3, 6]]
Initially, Alice is standing on coordinate '0'.

In the first second, she will move from coordinate '0' to '1'.
Then she will use the portal '0' and instantly move from coordinate '1' to '5'.

In the next second, she will move from coordinate '5' to '6'.
Then she will use the portal '2' and instantly move from coordinate '6' to '3'.

In the next second, she will move from coordinate '3' to '2'.
Then she will use the portal '1' and instantly move from coordinate '2' to '10'.

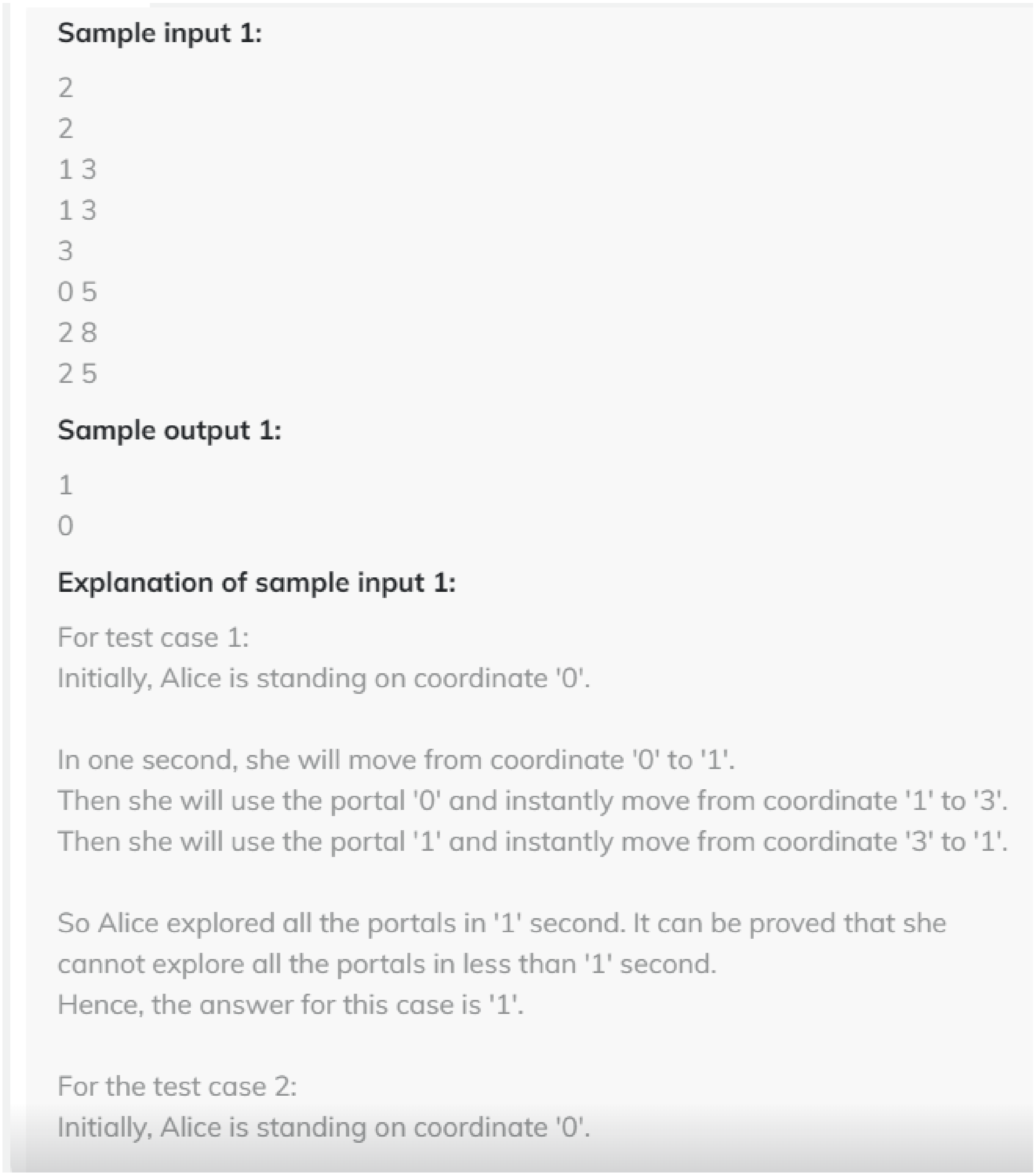
So Alice explored all three portals in '3' seconds. It can be proved that she cannot explore all the portals in less than '3' seconds.
Hence, the answer for this case is '3'.

Detailed explanation (Input/output format, Notes, Images)

Constraints:

2 <= 'N' <= 16
0 <= 'portals[i][0]' < 'portals[i][1]' <= 10^5

Time limit: 1 sec



Sample input 1:

```
2
2
1 3
1 3
3
0 5
2 8
2 5
```

Sample output 1:

```
1
0
```

Explanation of sample input 1:

For test case 1:
Initially, Alice is standing on coordinate '0'.

In one second, she will move from coordinate '0' to '1'.
Then she will use the portal '0' and instantly move from coordinate '1' to '3'.
Then she will use the portal '1' and instantly move from coordinate '3' to '1'.

So Alice explored all the portals in '1' second. It can be proved that she cannot explore all the portals in less than '1' second.
Hence, the answer for this case is '1'.

For the test case 2:
Initially, Alice is standing on coordinate '0'.

She will use the portal '0' and instantly move from coordinate '0' to '5'.
Then she will use the portal '2' and instantly move from coordinate '5' to '2'.
Then she will use the portal '1' and instantly move from coordinate '2' to '8'.

As Alice explored all three portals instantly, the answer for this case is '0'.

Sample input 2:

```
2
4
0 2
1 3
3 5
0 4
3
0 7
1 4
1 5
```

Sample output 2:

```
2
2
```

Approach 1

Bitmask, Dynamic programmingkeyboard_arrow_up

Approach:

- We can use the following three-dimensional dynamic programming approach with bitmask.
- First, let's see what 'mask' represents. 'mask' is an integer between '0' and '2 ^ N'. We can use this integer to store a unique subset of 'N' portals.
  - If the 'i-th' bit is '1' in the mask, portal 'i' is present in the subset.
- dp[mask][lastPortal][direction] represents the minimum time required to use all the portals defined by 'mask' such that 'lastPortal' is the last used portal.

    - 'direction' can have the value '0' or '1' representing the direction in which the last portal is used.
    - If it is '1', then the last portal is used to move from 'portals[lastPortal][0]' to 'portals[lastPortal][1]' and vice versa.
- Now let's see how to calculate the 'dp[mask][lastPortal][direction]'.
    - Suppose that 'previousPortal' is the portal used before 'lastPortal' and 'previousDirection' is the direction in which 'previousPortal' is used.
    - So 'dp[mask][lastPortal][direction]' can be 'dp[previousMask][previousPortal][previousDirection]' + 'extra time for movements between lastPortal and previousPortal'. 'previousMask' is the same as 'mask' except one bit representing 'lastPortal' which is '0'.
    - So we will brute force on all possible pairs of 'previousPortal' and 'previousDirection' and find the minimum value for 'dp[mask][lastPortal][direction]'.
    - Note that we will not consider a portal that is not in the subset defined by 'mask' as the last portal or previous portal. The DP state for the same will have a large value denoting infinity.
- After calculating all the states, the final answer will be the minimum of 'dp[((2 ^ N) - 1)][lastPortal][direction]' where 'lastPortal' and 'direction can have any possible value.
    - '(2 ^ N) - 1' is the integer having first 'N' bits equal to '1', which denotes that all the portals are used.

Algorithm:

- Initialize an integer 'minimumTime' equal to infinity to store the answer. (Infinity for the current context can be any value that is very large compared to the answer.)
- Initialize three-dimensional vector 'dp' with all the values equal to infinity.
- For 'mask' in the range from '1' to '(2 ^ N) - 1':
    - For 'last' in the range from '0' to 'N - 1':
        - If bit 'last' in the 'mask' is '0':
            - Continue to the next iteration.
        - For 'direction' equal to '0' and '1':
            - For 'previousLast' in the range from '0' to 'N - 1':
                - For 'previousDirection' equal to '0' and '1':
                    - Initialize an integer 'previousMask' equal to 'mask - (2 ^ last)'.
                    - Initialize an integer 'lastCoordinate' equal to 'portals[previousLast][previousDirection]'.

                - Initialize an integer 'currentCordinate' equal to 'portals[last][direction XOR 1]'. (XOR operation with '1' will change the value '1' to '0' and '0' to '1'.)
                - Initialize an integer 'time' equal to 'dp[previousMask][previousLast][previousDirection] + absolute difference between lastCoordinate and currentCoordinate'.
                - Update 'dp[mask][last][direction]' to the minimum from of 'dp[mask][last][direction]' and 'time'.
        - If 'mask' is equal to '2 ^ last':
            - Update 'dp[mask][last][direction]' equal to 'portals[last][direction XOR 1]'.
        - If 'mask' is equal to '(2 ^ N) - 1':
            - Update 'minimumTime' to a minimum of 'minimumTime' and 'dp[mask][last][direction]'.
- Return 'minimumTime'.

Time Complexity

O((2 ^ N) * (N^2)), where 'N' denotes the total number of portals.

We are using the five nested loops for state changes, in which the first loop is running from '1' to '2 ^ N', the second and fourth are running from '0' to 'N - 1', and the third and fifth are running for only '2' unique values, which is constant. So this is of order O((2 ^ N) * (N^2)).

So, the overall time complexity is of order O((2 ^ N) * (N^2)).

Space Complexity

O((2 ^ N) * N), where 'N' denotes the total number of portals.

We are using a three-dimensional vector to store dynamic programming states where the first and second parameters have '2 ^ N' and 'N' unique values, respectively. The third parameter has only '2' unique values, which is constant. So the complexity for the same is of order O((2 ^ N) * N).

So, the overall space complexity is of order O((2 ^ N) * N

```
/*
    Time Complexity: O((2 ^ n) * (n ^ 2))
    Space Complexity: O((2 ^ n) * n)

    Where 'n' denotes the total number of portals.
*/

int minimumTimeRequired(int n, vector<vector<int>> portals) {

    // Initialize an integer 'minimumTime' equal to infinity to store the answer.
    int minimumTime = 1e9;

    // Initialize three-dimensional vector 'dp' with all the values equal to infinity.
    vector<vector<vector<int>>> dp(1 << n, vector<vector<int>>(n, vector<int>(2, 1e9)));

    // Iterating through all possible triplets of 'mask', 'last', and 'direction'.
    for (int mask = 1; mask < (1 << n); mask++) {
        for (int last = 0; last < n; last++) {

            // Check if a bit 'last' in the 'mask' is '1' or not.
            if ((mask & (1 << last)) == 0) {
                continue;
            }
            for (int direction = 0; direction < 2; direction++) {

                // Iterate through all possible pairs of 'previousLast' and 'previousDirection'.
                for (int previousLast = 0; previousLast < n; previousLast++) {
                    for (int previousDirection = 0; previousDirection < 2; previousDirection++) {

                        // Calculate the previous mask, last coordinate of the previously used portal, and
initial coordinate of the last used portal.
```

```
                    int previousMask = mask - (1 << last);
                    int lastCoordinate = portals[previousLast][previousDirection];
                    int currentCoordinate = portals[last][direction ^ 1];

                    // Calculate the total required time.
                    int time = dp[previousMask][previousLast][previousDirection] +
abs(lastCoordinate - currentCoordinate);

                    // Update the 'dp[mask][last][direction]'.
                    dp[mask][last][direction] = min(dp[mask][last][direction], time);
                }
            }

            // Set the value of 'dp[mask][last][direction]' manually if the 'mask' has exactly one bit
equal to '1'.
            if (mask == (1 << last)) {
                dp[mask][last][direction] = portals[last][direction ^ 1];
            }

            // Update the answer if all the bits are '1' in the 'mask'.
            if (mask == ((1 << n) - 1)) {
                minimumTime = min(minimumTime, dp[mask][last][direction]);
            }
        }
    }
  }

  // Return the calculated answer.
  return minimumTime;
}
```