Kickstart 2019 B (20 Apr)

<https://codingcompetitions.withgoogle.com/kickstart/round/0000000000050eda>

# Building Palindromes

### **Problem**

Anna has a row of **N** blocks, each with exactly one letter from A to Z written on it. The blocks are numbered 1, 2, ..., **N** from left to right.

Today, she is learning about palindromes. A palindrome is a string that is the same written forwards and backwards. For example, ANNA, RACECAR, AAA and X are all palindromes, while AB, FROG and YOYO are not.

Bob wants to test how well Anna understands palindromes, and will ask her **Q** questions. The i-th question is: can Anna use all of the blocks numbered from **Li** to **Ri**, inclusive, rearranging them if necessary, to form a palindrome? After each question, Anna puts the blocks back in their original positions.

Please help Anna by finding out how many of Bob's questions she can answer "yes" to.

### **Input**

The first line of the input gives the number of test cases, **T**. **T** test cases follow. Each test case starts with a line containing the two integers **N** and **Q**, the number of blocks and the number of questions, respectively. Then, another line follows, containing a string of **N** uppercase characters (A to Z). Then, **Q** lines follow. The i-th line contains the two integers **Li** to **Ri**, describing the i-th question.

### **Output**

For each test case, output one line containing Case #x: y, where x is the test case number (starting from 1) and y is the number of questions Anna can answer "yes" to.

### **Limits**

Time limit: 30 seconds per test set.  
Memory limit: 1GB.  
1 ≤ **T** ≤ 100. 1 ≤ **Li** ≤ **Ri** ≤ **N**.

#### **Test set 1 (Visible)**

1 ≤ **N** ≤ 20.  
1 ≤ **Q** ≤ 20.

#### **Test set 2 (Hidden)**

1 ≤ **N** ≤ 105.  
1 ≤ **Q** ≤ 105.

### **Sample**

|  |  |
| --- | --- |
| Input | Output |
| 2  7 5  ABAACCA  3 6  4 4  2 5  6 7  3 7  3 5  XYZ  1 3  1 3  1 3  1 3  1 3 | Case #1: 3  Case #2: 0 |

In Sample Case #1, **N** = 7 and **Q** = 5.

* For the first question, Anna must use the blocks AACC. She can rearrange these blocks into the palindrome ACCA (or CAAC).
* For the second question, Anna must use the blocks A. This is already a palindrome, so she does not need to rearrange them.
* For the third question, Anna must use the blocks BAAC. These blocks cannot be rearranged into a palindrome.
* For the fourth question, Anna must use the blocks CA. These blocks cannot be rearranged into a palindrome.
* For the fifth question, Anna must use the blocks AACCA. She can rearrange these blocks to form the palindrome ACACA (or CAAAC).

In total, she is able to answer "yes" to 3 of Bob's questions, so the answer is 3.

In Sample Case #2, **N** = 3 and **Q** = 5. For the first question, Anna must use the blocks XYZ to create a palindrome. This is impossible, and since the rest of Bob's questions are the same as the first one, the answer is 0.

#### **Analysis**

For each query[**L**, **R**], we have to figure out if it is possible to make a palindrome from substring[**L**, **R**] of the original block string.   
For any substring, we can create all possible permutations of it and check if any of those is a palindrome. For a substring of length *l*, in the worst case there would be [Factorial(*l*)](https://en.wikipedia.org/wiki/Factorial) permutations, and to check if a string of length *l* is palindrome is O(*l*). So the cost of each query would be O(**N**\*Factorial(**N**)) and thus wouldn't fit within the time limit.   
One important observation is that in a palindrome, at most one character can appear odd number of times. If more than one character appears odd number of times in a string, it is impossible to rearrange that string to form a palindrome.   
If all characters in a string are present even times, we can just divide the characters into two identical sets. Then we can make two strings with those two sets such that one string is the reverse of the other one. Finally concatenate them to get a palindrome.   
If we have only one character let's say *x*, which is present odd number of times, we can set aside one *x*. Then we get a set of character where all characters are present even number of times, and we can construct a palindrome in somewhat similar way as described above, this time we can just put the *x* in between those two strings.

### **Test set 1 (Visible)**

For each query, we can count the frequencies of all characters in the substring and decide if it is possible to make a palindrome or not. The complexity of solving each query in this approach is the length of the substring, which is O(**N**). Total complexity of this approach is O(**N** × **Q**), which will be sufficient for test set 1.

### **Test set 2 (Hidden)**

We need to count the frequencies of each characters in a query substring, but can we make the counting of frequencies faster?   
Notice that, if we calculate prefix sum of frequencies for each character on the whole input string, we can get the frequency of any character in any given substring in O(1) time.   
In this approach, the time required to compute the prefix sum of the frequencies for each character is O(**N**) and hence O(**N** × |character set|) for all. And for each query substring, we can check parity of the frequency for each of the characters in O(|character set|) time. So for **Q** queries, the overall complexity of this approach is O((**N** + **Q**) × |character set|), which is sufficient for test set 2.

# Energy Stones

### **Problem**

Duda the rock monster lives in the enchanted forest and has collected **N** *energy stones* for lunch. Since he has a small mouth, he eats energy stones one at a time. Some stones are tougher than others! The i-th stone takes him **Si** seconds to eat.

Duda eats energy stones to get *energy*. Different stones give him different amounts of energy. Furthermore, the stones lose energy over time. The i-th stone initially contains **Ei** units of energy and will lose **Li** units of energy each second. When Duda starts to eat a stone, he will receive all the energy the stone contains immediately (no matter how much time it takes to actually finish eating the stone). The stone's energy stops decreasing once it hits zero.

What is the largest amount of energy Duda could receive from eating his stones?

### **Input**

The first line of the input gives the number of test cases, **T**. **T** test cases follow. Each test case starts with a line containing the integer **N**, the number of energy stones Duda has. Then, there are **N** more lines, the i-th of which contains the three integers **Si**, **Ei** and **Li**, as described above.

### **Output**

For each test case, output one line containing Case #x: y, where x is the test case number (starting from 1) and y is the maximum amount of energy Duda could receive from eating stones.

### **Limits**

Time limit: 30 seconds per test set.  
Memory limit: 1GB.  
1 ≤ **T** ≤ 100.  
1 ≤ **N** ≤ 100.  
1 ≤ **Si** ≤ 100.  
1 ≤ **Ei** ≤ 105.  
0 ≤ **Li** ≤ 105.

#### **Test set 1 (Visible)**

All stones take the same amount of time to eat. That is: **Si** = **Sj** for all i and j.

#### **Test set 2 (Hidden)**

There are no additional constraints beyond the general Limits.

### **Sample**

|  |  |
| --- | --- |
| Input | Output |
| 3  4  20 10 1  5 30 5  100 30 1  5 80 60  3  10 4 1000  10 3 1000  10 8 1000  2  12 300 50  5 200 0 | Case #1: 105  Case #2: 8  Case #3: 500 |

In Sample Case #1, there are **N** = 4 stones. One possible order Duda can eat stones is:

* Eat the fourth stone. This takes 5 seconds and gives him 80 units of energy.
* Eat the second stone. This takes 5 more seconds and gives him 5 units of energy (the second stone started with 30 energy, and over 5 seconds, has lost 25 units of energy).
* Eat the third stone. This takes 100 more seconds and gives him 20 units of energy (the third stone started with 30 energy, and over 10 seconds, has lost 10 units of energy).
* Eat the first stone. This takes 20 more seconds and gives him 0 units of energy (the first stone started with 10 units of energy, and over 110 seconds, has lost all of its energy).

This gives him 105 units of energy, which is the best he can do. So the answer is 105.

In Sample Case #2, there are **N** = 3 stones. No matter which stone Duda eats, the other two will have no energy left once he is done eating. So he should eat the third stone, giving him 8 units of energy.

In Sample Case #3, there are **N** = 2 stones. Duda can:

* Eat the first stone. This takes 12 seconds and gives him 300 units of energy.
* Eat the second stone. This takes 5 seconds and gives him 200 units of energy (the second stone does not lose any energy over time!).

So the answer is 500.

Note that Sample Cases #1 and #3 will not appear in the Visible dataset. Consequently, sample input is not included as a test case for your submissions, and hence any attempts that fail to produce correct output will accrue a penalty. We recommend you to run a manual test in the editor before submitting the attempt.

#### **Analysis**

### **Test set 1 (Visible)**

For this test set, it is guaranteed that **S**i = **S**j for all i, j. For simplicity, we will assume that we never eat a stone with zero energy. Consider two energy stones i and j that will be eaten back-to-back. If **L**i > **L**jthen we should eat i before j. This is because stone i loses energy faster than j, so taking it first will result in a smaller overall loss of energy.

Thus, no matter which set of energy stones are eaten, that set should be eaten in non-increasing value of **L**i. So we should first sort the stones by **L**i and then the only decision to be made is which stones should be eaten and which should not be eaten. This reduces the problem to a [0/1 Knapsack](https://en.wikipedia.org/wiki/Knapsack_problem)problem. This can be solved with [dynamic programming](https://en.wikipedia.org/wiki/Dynamic_programming).

Define max\_energy(time, i) as the maximum total energy that can be achieved given the current time and considering only the suffix of energy stones sorted in decreasing **L**i from i to **N**. The recurrence relation for this function considers two cases. Either take the i-th energy stone (with its energy adjusted by the time), or do not take it. So, max\_energy(time, i) is the maximum of:

* max\_energy(time+**S**i, i+1) +max(0,**E**i-**L**i\*time)
* max\_energy(time,i+1)

The maximum possible time is the sum of all **S**i because an optimal strategy might eat all the stones and will not use any time waiting. Call this sum(**S**). The time complexity of this approach can be described as O(**N**\*sum(**S**)). This is fast enough for both test sets. However, sorting energy stones by **L**i is incorrect for Test set 2.

### **Test set 2 (Hidden)**

We will need to find a different way to order the energy stones to solve Test set 2. As before, consider two energy stones i and j assuming that we can take both i and j without either going to zero energy. We know that **S**i might not equal **S**j. However, there is an ordering for taking both i and j that is always optimal. Observe that **S**i\***L**j is the total loss of energy if i is used first. Likewise, **S**j\***L**i is the loss if j is used first. Thus, if **S**i\***L**j < **S**j\***L**i then taking i first leads to a smaller overall loss of energy. It may not be obvious that we should always take i before j even if it leads to a smaller loss of energy. This is because there may be other stones between i and j in some potential ordering. However, if i and j are adjacent in some ordering, then we will achieve more energy by swapping them if **S**i\***L**j > **S**j\***L**i. Applying this rule iteratively will eventually sort the stones. Therefore, this rule defines an ordering on our energy stones. If our assumption about neither stone going to zero energy is not true, then the order of i and j does not matter. In that case, the only decision is what stones we eat. Thus, we can use the dynamic programming solution from Test set 1 to solve Test set 2 with the same time complexity.

The reader may have noticed that this sort order is equivalent to comparing fractions; it is the same as sorting by **S**i/**L**i. However, one must be careful when **L**i=0.

# Diverse Subarrays

### **Problem**

Vanity has **N** trinkets on her shelf, numbered 1, 2, ..., **N** from left to right. Trinkets come in different types, which are denoted by positive integers. The i-th trinket on her shelf is of type **Ai**.

She is going to see her family overseas today and would like to bring as many trinkets as she can. However, since she is in a hurry, Vanity must take a consecutive interval of trinkets. Formally, Vanity selects two indices, *l* and *r*, and takes all of the trinkets numbered *l*, *l*+1, ..., *r*-1, *r*. Also, due to tax rules, airport security will throw away *all* trinkets of a type if Vanity has more than **S** of that type in the chosen interval.

For example, suppose that **S** = 2, and Vanity brings six trinkets: one of type 0, two of type 1, and three of type 2. She will be allowed to keep the trinket of type 0 and both trinkets of type 1, but she will lose*all* of the trinkets of type 2!

Vanity needs to choose *l* and *r* such that she can take the maximum number of trinkets for her family. What is the maximum number of trinkets she can bring?

### **Input**

The first line of the input gives the number of test cases, **T**. **T** test cases follow. The first line of each test case contains the two integers **N** and **S**, the number of trinkets, and the maximum number of trinkets allowed of a single type, respectively. The second line contains **N** integers. The i-th integer gives **Ai**, the type of the i-th trinket.

### **Output**

For each test case, output one line containing Case #x: y, where x is the test case number (starting from 1) and y is the maximum number of trinkets that Vanity can bring to her family.

### **Limits**

Time limit: 30 seconds per test set.  
Memory limit: 1GB.  
1 ≤ **T** ≤ 100.  
1 ≤ **Ai** ≤ 105.  
1 ≤ **S** ≤ **N**.

#### **Test set 1 (Visible)**

1 ≤ **N** ≤ 1000.

#### **Test set 2 (Hidden)**

1 ≤ **N** ≤ 105.

### **Sample**

|  |  |
| --- | --- |
| Input | Output |
| 4  6 2  1 1 4 1 4 4  8 1  1 2 500 3 4 500 6 7  10 1  100 200 8 8 8 8 8 300 400 100  12 2  40 50 1 1 1 60 70 2 2 2 80 90 | Case #1: 4  Case #2: 6  Case #3: 4  Case #4: 6 |

In Sample Case #1, Vanity should choose *l* = 2 and *r* = 5. This allows her to take 4 trinkets to the airport of types 1, 4, 1 and 4. None of them are thrown away by airport security, so she is able to bring 4 trinkets to her family.

In Sample Case #2, Vanity should choose *l* = 1 and *r* = 8. This allows her to take all 8 trinkets to the airport. Her trinkets of type 500 are thrown away since she has more than **S** = 1 of them, so she is able to bring a total of 6 trinkets to her family.

In Sample Case #3, Vanity should choose *l* = 1 and *r* = 9. This allows her to take 9 trinkets to the airport of types 100, 200, 8, 8, 8, 8, 8, 300 and 400. Her trinkets of type 8 are thrown away since she has more than **S** = 1 of them, so she is able to bring a total of 4 trinkets to her family.

In Sample Case #4, Vanity should choose *l* = 1 and *r* = 12. This allows her to take all 12 trinkets to the airport. Her trinkets of type 1 and 2 are thrown away since she has more than **S** = 2 of each of them, so she is able to bring a total of 6 trinkets to her family.

**Note**: We do not recommend using interpreted/slower languages for this problem.

#### **Analysis**

Consider finding the optimal solution that is a prefix of the input. That is, let us only consider taking ranges of trinkets [1,r] for any r. This reduces the problem to finding prefix sum maximums. For example, consider this sequence of trinkets:   
*1, 1, 2, 1, 3, 2, 3, 2, 2*  
Assume that **S**=2.   
The first and second occurence of a type can be modeled as a +1 event since we can take the trinkets. The **S**+1th occurence can be a -**S** event, since now we cannot take any trinkets of that type. Finally, any further occurence of that type can be a +0 since we still cannot take them. Thus, we have the following events:   
*+1, +1, +1, -2, +1, +1, +1, -2, +0*  
A prefix sum ending at index r in our events corresponds to the score for taking the subarray [1,r]. This idea can be generalized to any value of **S**, since all events until the **S**+1th can be +1 events.

### **Test set 1 (Visible)**

A naive algorithm can be realized using the above strategy. It can be applied to each potential left point (instead of always starting at 1). This explicitly considers every possible range [l,r] and computes the number of trinkets. The events can be calculated by keeping track of how many times each type has occured while sweeping across the trinkets and keeping a running sum of events. The count of types can be handled with a frequency table. Any map from type to count can implement this, for example, an array or hashtable. The strategy uses O(**N**) time to process the solution starting at a particular left point. There are O(**N**) left points. Thus, the time complexity is O(**N**2). This is sufficient for Test set 1.

### **Test set 2 (Hidden)**

The previous algorithm will not be fast enough for the Test set 2. Observe that the events for the lth left point are closely related to those for the l+1th left point. In particular, only events corresponding to the type of the trinket at index l change. In fact, only the -**S** for that type changes. It moves over to the next +0 event for that type if it exists. The previous -**S** event becomes a +1. This means that the events only change at a constant number of places.

Using this observation, we can think about this as a data structure problem. We require a data structure that supports two operations:

1. Change the value at an index.
2. What is the maximum prefix sum?

This can be used to maintain our events and find the best solution for each left point respectively. These operations can be achieved using a modified [Segment Tree](https://en.wikipedia.org/wiki/Segment_tree). Each node in the segment tree can be modified to store the sum of the elements covered by that node. Also, each node should store the maximum sum of any prefix of elements that are covered by that node. Such a tree can support both operations in O(log(**N**)) time. We can achieve a final complexity of O(**N** log(**N**)).