Competitive Programming DeCal

Lecture #3 on 2/6/23 - Greedy Algorithms

Welcome Back!

- Today we'll be covering greedy algorithms
- First we'll go over solutions to problem C and D from last week's problemset
    - Give those problems a read if you haven't already
- This week's problemset is up!
    - Feel free to sign up for it, we'll be going over the first two problems today

Announcements!

- Issues with viewing test cases :(
    - Problem B (Fence) will no longer be graded on correctness

Announcements!

- Issues with viewing test cases :(
  - Problem B (Fence) will no longer be graded on correctness
- For this week there will be hints rather than code!
  - They will be spoiler text placed in announcements
- Fast Input/Output
- Problems are generally ordered in difficulty

Solution To Problem C (Lecture Sleep)

- https://codeforces.com/problemset/problem/961/B

Example

input
6 3 1 3 5 2 5 4 1 1 0 1 0 0
output
16

Your friend Mishka and you attend a calculus lecture. Lecture lasts n minutes. Lecturer tells a_i theorems during the i-th minute.

Mishka is really interested in calculus, though it is so hard to stay awake for all the time of lecture. You are given an array t of Mishka's behavior. If Mishka is asleep during the i-th minute of the lecture then t_i will be equal to 0, otherwise it will be equal to 1. When Mishka is awake he writes down all the theorems he is being told — a_i during the i-th minute. Otherwise he writes nothing.

You know some secret technique to keep Mishka awake for k minutes straight. However you can use it **only once**. You can start using it at the beginning of any minute between 1 and $n - k + 1$. If you use it on some minute i then Mishka will be awake during minutes j such that $j \in [i, i + k - 1]$ and will write down all the theorems lecturer tells.

You task is to calculate the maximum number of theorems Mishka will be able to write down if you use your technique **only once** to wake him up.

Input

The first line of the input contains two integer numbers n and k ($1 \le k \le n \le 10^5$) — the duration of the lecture in minutes and the number of minutes you can keep Mishka awake.

The second line of the input contains n integer numbers $a_1, a_2, \dots a_n$ ($1 \le a_i \le 10^4$) — the number of theorems lecturer tells during the i-th minute.

The third line of the input contains n integer numbers $t_1, t_2, \dots t_n$ ($0 \le t_i \le 1$) — type of Mishka's behavior at the i-th minute of the lecture.

Output

Print only one integer — the maximum number of theorems Mishka will be able to write down if you use your technique **only once** to wake him up.

Solution To Problem C (Lecture Sleep)

- https://codeforces.com/problemset/problem/961/B
- Solution:
  - First, sum the number of theorems he writes when he is awake, then set them to zero
  - Create a prefix sum on the array
    - This prefix sum array is only built on the amount of theorems **WHILE HE IS ASLEEP**
  - Now find a range of length ***k*** with the greatest sum
  - The answer is ***awake_sum*** + ***largest_range_len_k***

Example

input
6 3 1 3 5 2 5 4 1 1 0 1 0 0
output
16

Solution to Problem D (Karen and Coffee)

- https://codeforces.com/problemset/problem/816/B

Karen, a coffee aficionado, wants to know the optimal temperature for brewing the perfect cup of coffee. Indeed, she has spent some time reading several recipe books, including the universally acclaimed "The Art of the Covfefe".

She knows n coffee recipes. The i-th recipe suggests that coffee should be brewed between l_i and r_i degrees, inclusive, to achieve the optimal taste.

Karen thinks that a temperature is *admissible* if at least k recipes recommend it.

Karen has a rather fickle mind, and so she asks q questions. In each question, given that she only wants to prepare coffee with a temperature between a and b, inclusive, can you tell her how many admissible integer temperatures fall within the range?

Input

The first line of input contains three integers, n, k ($1 \le k \le n \le 200000$), and q ($1 \le q \le 200000$), the number of recipes, the minimum number of recipes a certain temperature must be recommended by to be admissible, and the number of questions Karen has, respectively.

The next n lines describe the recipes. Specifically, the i-th line among these contains two integers l_i and r_i ($1 \le l_i \le r_i \le 200000$), describing that the i-th recipe suggests that the coffee be brewed between l_i and r_i degrees, inclusive.

The next q lines describe the questions. Each of these lines contains a and b, ($1 \le a \le b \le 200000$), describing that she wants to know the number of admissible integer temperatures between a and b degrees, inclusive.

Output

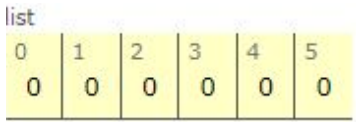
For each question, output a single integer on a line by itself, the number of admissible integer temperatures between a and b degrees, inclusive.

Solution to Problem D (Karen and Coffee)

- https://codeforces.com/problemset/problem/816/B
- Solution:
  - Observe that the ranges can take values between 1 and 200000
  - We can make an array a where a_i = number of recipes with that can be at temperature i
  - This array a can be quickly computed with the fast range update trick
    - Trick: Make a new array p. For a recipe with range [a, b], add one to $p[a]$ and subtract one from $p[b+1]$
  - Now array a is the prefix sum array of p

Demo

- Let’s say we have two coffees with temperature ranges [1, 4] and [2, 3]



We can make an array a where a_i = number of recipes with that can be at temperature i

This array a can be quickly computed with the fast range update trick

- Trick: Make a new array p. For a recipe with range [a, b], add one to $p[a]$ and subtract one from $p[b+1]$

Now array a is the prefix sum array of p

Demo

- Let's say we have two coffees with temperature ranges [1, 4] and [2, 3]
- Adding the first coffee

p

list

0	1	2	3	4	5
0	1	0	0	0	0

a

list

0	1	2	3	4	5
0	1	1	1	1	1

We can make an array a where a_i = number of recipes with that can be at temperature i

This array a can be quickly computed with the fast range update trick

- Trick: Make a new array p. For a recipe with range [a, b], add one to $p[a]$ and subtract one from $p[b+1]$

Now array a is the prefix sum array of p

Demo

- Let's say we have two coffees with temperature ranges [1, 4] and [2, 3]
- Adding the first coffee

p

list

0	1	2	3	4	5
0	1	0	0	0	-1

a

list

0	1	2	3	4	5
0	1	1	1	1	0

We can make an array a where a_i = number of recipes with that can be at temperature i

This array a can be quickly computed with the fast range update trick

- Trick: Make a new array p. For a recipe with range [a, b], add one to $p[a]$ and subtract one from $p[b+1]$

Now array a is the prefix sum array of p

Demo

- Let's say we have two coffees with temperature ranges [1, 4] and [2, 3]
- Adding the second coffee

p

list

0	1	2	3	4	5
0	1	1	0	-1	-1

a

list

0	1	2	3	4	5
0	1	2	2	1	0

Solution to Problem D (Karen and Coffee)

- https://codeforces.com/problemset/problem/816/B
- Solution:
  - We now have array a where a_i = number of recipes with that are valid at temperature i
  - Make a prefix sum array on a but instead of adding a_i to your running sum, add 1 if $a_i >= k$
  - Now we can solve the problem using a traditional prefix sum range query on our array!

What are greedy algorithms?

- A **greedy algorithm** will always take the option that appears to be the best at the given moment
  - We never look back or undo this decision at a later choice
  - Formally, a **greedy algorithm** assumes that a **locally optimal solution** will always contribute to the **globally optimal solution**

What are greedy algorithms?

- A **greedy algorithm** will always take the option that appears to be the best at the given moment
  - We never look back or undo this decision at a later choice
  - Formally, a **greedy algorithm** assumes that a **locally optimal solution** will always contribute to the **globally optimal solution**
- Example:
  - Given that you have infinite of each U.S. coin **{1, 5, 10, 25}**, what is the **least amount of coins** needed to make change for ***x*** cents?

What are greedy algorithms?

- A **greedy algorithm** will always take the option that appears to be the best at the given moment
  - We never look back or undo this decision at a later choice
  - Formally, a **greedy algorithm** assumes that a **locally optimal solution** will always contribute to the **globally optimal solution**
- Example:
  - Given that you have infinite of each U.S. coin **{1, 5, 10, 25}**, what is the **least amount of coins** needed to make change for ***x*** cents?
- The greedy algorithm for the coin change problem above?
  - Take as many of the biggest coin as possible first. Then move down to the next coin and repeat

Quick Brain Teaser

- What if we weren't dealing with the American coin system and instead had other values for coins?
- Would it still work?

Quick Brain Teaser

- What if we weren't dealing with the American coin system and instead had other values for coins?
- Would it still work?
- Nope!
- Consider the case for coin = {1, 7, 10} and x = 14
  - Greedy solution would give 5 coins (10 + 1 + 1 + 1 + 1)
  - Actual solution would be using two 7 cent coins
  - To make it work with any coins we would use dynamic programming (covered in a few weeks!)
- Even bigger brain teaser: what property of {1, 5, 10, 25} makes the greedy algorithm viable that {1, 7, 10} does not have?

Sorting

- Often, it can be useful to sort your input data before running a greedy algorithm
- No need to remember any of the complicated sorting algorithms from 61B!
    - Use the built-in sort for your programming language

Sorting

- Often, it can be useful to sort your input data before running a greedy algorithm
- No need to remember any of the complicated sorting algorithms from 61B!
    - Use the built-in sort for your programming language

Sorting

- Often, it can be useful to sort your input data before running a greedy algorithm
- No need to remember any of the complicated sorting algorithms from 61B!
    - Use the built-in sort for your programming language
- A sorting algorithm arranges elements of a list in a specific order
    - **{4, 3, 2, 1, 5} → {1, 2, 3, 4, 5}** increasing order
    - **{2, 3, 5, 1, 4} → {5, 4, 3, 2, 1}** decreasing order
    - **{'B', 'F', 'Z', 'A'} → {'A', 'B', 'F', 'Z'}** we can also sort alphabetically

Sorting

- Often, it can be useful to sort your input data before running a greedy algorithm
- No need to remember any of the complicated sorting algorithms from 61B!
    - Use the built-in sort for your programming language
- A sorting algorithm arranges elements of a list in a specific order
    - **{4, 3, 2, 1, 5} → {1, 2, 3, 4, 5}** increasing order
    - **{2, 3, 5, 1, 4} → {5, 4, 3, 2, 1}** decreasing order
    - **{'B', 'F', 'Z', 'A'} → {'A', 'B', 'F', 'Z'}** we can also sort alphabetically
- You can assume it works in O(n log n)

Sorting

- Given an array or list named ***a*** of size ***n***
- Python:
  - a.sort()
- C++:
  - sort(a, a + n)
- Java:
  - Arrays.sort(a)

A. GamingForces

time limit per test: 2 seconds
memory limit per test: 256 megabytes
input: standard input
output: standard output

Monocarp is playing a computer game. He's going to kill n monsters, the i-th of them has h_i health.

Monocarp's character has two spells, either of which he can cast an arbitrary number of times (possibly, zero) and in an arbitrary order:

- choose exactly two alive monsters and decrease their health by 1;
- choose a single monster and kill it.

When a monster's health becomes 0, it dies.

What's the minimum number of spell casts Monocarp should perform in order to kill all monsters?

Input

The first line contains a single integer t ($1 \le t \le 10^4$) — the number of testcases.

The first line of each testcase contains a single integer n ($1 \le n \le 100$) — the number of monsters.

The second line contains n integers $h_1, h_2, \dots, h_n$ ($1 \le h_i \le 100$) — the health of each monster.

The sum of n over all testcases doesn't exceed $2 \cdot 10^4$.

Output

For each testcase, print a single integer — the minimum number of spell casts Monocarp should perform in order to kill all monsters.

Example

input Copy

```
3
4
1 2 1 2
3
2 4 2
5
1 2 3 4 5
```

output Copy

```
3
3
5
```

Note

In the first testcase, the initial health list is $[1, 2, 1, 2]$. Three spells are casted:

- the first spell on monsters 1 and 2 — monster 1 dies, monster 2 has now health 1, new health list is $[0, 1, 1, 2]$;
- the first spell on monsters 3 and 4 — monster 3 dies, monster 4 has now health 1, new health list is $[0, 1, 0, 1]$;
- the first spell on monsters 2 and 4 — both monsters 2 and 4 die.

In the second testcase, the initial health list is $[2, 4, 2]$. Three spells are casted:

- the first spell on monsters 1 and 3 — both monsters have health 1 now, new health list is $[1, 4, 1]$;
- the second spell on monster 2 — monster 2 dies, new health list is $[1, 0, 1]$;
- the first spell on monsters 1 and 3 — both monsters 1 and 3 die.

In the third testcase, the initial health list is $[1, 2, 3, 4, 5]$. Five spells are casted. The i-th of them kills the i-th monster with the second spell. Health list sequence: $[1, 2, 3, 4, 5] \rightarrow [0, 2, 3, 4, 5] \rightarrow [0, 0, 3, 4, 5] \rightarrow [0, 0, 0, 4, 5] \rightarrow [0, 0, 0, 0, 5] \rightarrow [0, 0, 0, 0, 0]$.

GamingForces Solution

- Consider the first operation where we can choose two alive monsters and decrease their health by one
- If the health of either monster is greater than one, it would take at least two operations
- We can simply eliminate both monsters with two of the second operation
    - Always better unless both monsters have health = 1

GamingForces Solution

- Consider the first operation where we can choose two alive monsters and decrease their health by one
- If the health of either monster is greater than one, it would take at least two operations
- We can simply eliminate both monsters with two of the second operation
  - Always better unless both monsters have health = 1
- Solution:
  - Find ***cnt***, the number of monsters with health equal to 1
  - The answer is **ceil(*cnt*/2) + rest_of_the_monsters**

Event Scheduling

Given ***n*** events with their starting and ending times, find a schedule that includes as many events as possible. If you select an event, you must stay for the **entire duration**.

event	starting time	ending time
A	1	3
B	2	5
C	3	9
D	6	8

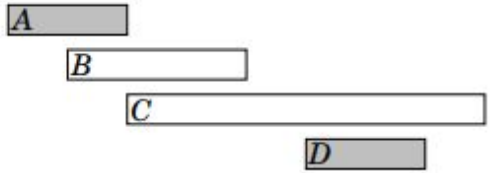
In this case the maximum number of events is two. For example, we can select events B and D as follows:

Images/example taken from Competitive Programmer's Handbook (supplemental book for the course)

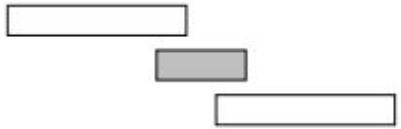
Event Scheduling

Algorithm 1

The first idea is to select as *short* events as possible. In the example case this algorithm selects the following events:



However, selecting short events is not always a correct strategy. For example, the algorithm fails in the following case:



Images/example taken from Competitive Programmer's Handbook (supplemental book for the course)

Event Scheduling

Algorithm 2

Another idea is to always select the next possible event that *begins* as *early* as possible. This algorithm selects the following events:

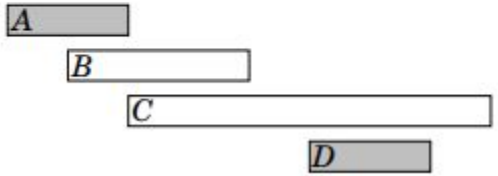
However, we can find a counterexample also for this algorithm. For example, in the following case, the algorithm only selects one event:

Images/example taken from Competitive Programmer's Handbook (supplemental book for the course)

Event Scheduling

Algorithm 3

The third idea is to always select the next possible event that *ends* as *early* as possible. This algorithm selects the following events:



Images/example taken from Competitive Programmer's Handbook (supplemental book for the course)

Why does this work?

- Intuition: the earlier you finish a task, the earlier you can start on a new task!
  - Note that the opposite isn't true, starting a task earlier may not necessarily mean anything!
- We can rigorously prove this with an *exchange argument*
  - Suppose there's an algorithm O that's **not our greedy algorithm**, and **also does better than greedy**
  - Suppose greedy picks a sequence of events G = $\{g_1, g_2, g_3, \ldots\}$ and O picks a sequence O = $\{o_1, o_2, o_3, \ldots\}$
  - At any point in time, there is a set of events options that can still be picked by each algorithm, E \ O and E \ G.
  - G always picks the events with the earliest end time. Since O is different, there must be some point where O does *not* pick the event with the earliest end time.
  - However, picking events with later end times can only remove event options later, as time will have passed.
  - This means that throughout the entire run, G's set of potential future events will always be a superset of O's potential future events, so |E \ G| >= |E \ O| and thus |G| >= |O|, which is a contradiction because we assumed |O| > |G|
  - Thus, greedy will always perform *at least as well* as the optimal algorithm. Greedy is optimal!

ain't nobody got time fo' dat

- In competitive programming, you often won't have the luxury to write out full proofs
  - Just wing it lmao
- If you can convince yourself it's correct, that's *probably* good enough
- For more practice in formally proving greedy algorithms with exchange arguments, take CS 170

Implementation of Event Scheduling

- Sort the array of events by their end times
  - Just Google how to do this in your programming language of choice lol
- Maintain a list of selected events, and the current time
- Loop through all events
  - If the starting time of this event is after the current time, add the current event and update the current time
  - Else, do nothing! We can't take this event because it already started
- Return the list of selected events

```
def schedule(events):
  sorted_events = sorted(events, key=lambda i: i[1])
  curr_time = 0
  selected = []
  for event in sorted_events:
    start, end = event
    if start > curr_time:
      selected.append(event)
      curr_time = end
  return selected
```

The Nature of Greedy Algorithms

- Often very fast!
- Usually also easy to implement!
- Hard part is seeing *the trick*, and everything becomes easy after that
- Can be challenging to prove, even if the idea is simple.
    - Don’t worry about it for the sake of competitive programming!
    - Just make sure you can convince yourself that it’s correct

Potential Refreshers

- Algorithms with greedy properties from 61B
  - Kruskal's algorithm for minimum spanning tree
    - Keep picking the lightest edge that **immediately** connects two trees!
  - Dijkstra's algorithm for single source shortest path
    - Keep exploring from the **closest known** vertex so far!

Check in!

- Scan this QR code
- https://tinyurl.com/4a3vjz9x

Problem 4: Water Bottles
4+3=7 Point(s)

Problem ID: `bottles`
Rank: `2+3`

Introduction

Berkeley students living in some parts of the Foothill residential complex source most potable water from a communal water dispenser. This problem is inspired by an everyday technique we use to reduce the awkwardness of delaying people behind us in line! The real dispenser outputs 2 liters per minute, but we'll say 1 per minute for this problem to make things simpler.

Problem Statement

$\mathbf{N}$ students numbered 1, ..., $\mathbf{N}$ are lined up at a water dispenser that dispenses water at a constant rate of 1 liter per minute. The i^{th} student has an empty bottle with capacity $\mathbf{C}_i$ liters that they begin to fill immediately after the previous student has finished (formally, the i^{th} student begins refilling when all j^{th} students for which $j < i$ finish refilling).

We define the *wait time* of a student as the **total** time they have to wait until their bottle **finishes** refilling. The students ask you to reorder the line into a new permutation $(a_1, ..., a_{\mathbf{N}})$ of the students' numbers such that the students' total wait time is minimized.

Input Format

The first line of the input contains an integer $\mathbf{T}$, denoting the number of test cases that follow.
For each test case:

- The first line contains a positive integer $\mathbf{N}$ denoting the number of students in line.
- The second line contains a sequence of $\mathbf{N}$ positive integers $\mathbf{C}_1, ..., \mathbf{C}_{\mathbf{N}}$, denoting the bottle capacities in liters.

Output Format

For each test case, output the following two lines:

- On the first line, output the minimum total wait time in minutes.
- On the second line, output $\mathbf{N}$ integers $a_1, ..., a_{\mathbf{N}}$ ($1 \le a_i \le \mathbf{N}$) where a_i is the new index of the i^{th} student from the front of the original line. If there are multiple permutations that satisfy all criteria, you may output any.

Constraints

$1 \le \mathbf{T} \le 100$
$1 \le \mathbf{N} \le 10^3$
The sum of $\mathbf{N}$ across all test cases in a test file does not exceed 10^3.
$1 \le \mathbf{C}_i \le 10^3$
All capacities $\mathbf{C}_i$ **are distinct**.
There is guaranteed to exist exactly one optimal permutation for each test case.

Sample Test Cases

Main Sample Input *Download*

```
1
3
5 1 2
```

Main Sample Output *Download*

```
12
2 3 1
```

Main Sample Explanations

The optimal permutation rearranges the line into the order (2, 3, 1).

1. Student 2 is first in the new line. They have a 1 L bottle and spend 1 minute refilling it.
2. Student 3 is second. They have to wait 1 minute and then spend 2 minutes refilling their own bottle, finishing after 3 minutes.
3. Student 1 is third. They have to wait 3 minutes and then spend 5 minutes refilling their own bottle, finishing after 8 minutes.

The total wait time is (1 + 3 + 8) min = 12 minutes, and it can be shown that no other permutation results in a total less than or equal to than 12 minutes.

CSES Problem Set

Stick Lengths

TASK | SUBMIT | RESULTS | STATISTICS | HACKING

Time limit: 1.00 s **Memory limit:** 512 MB

There are n sticks with some lengths. Your task is to modify the sticks so that each stick has the same length.

You can either lengthen and shorten each stick. Both operations cost x where x is the difference between the new and original length.

What is the minimum total cost?

Input

The first input line contains an integer n: the number of sticks.

Then there are n integers: $p_1, p_2, \ldots, p_n$: the lengths of the sticks.

Output

Print one integer: the minimum total cost.

Constraints

- $1 \le n \le 2 \cdot 10^5$
- $1 \le p_i \le 10^9$

Example

Input:

```
5
2 3 1 5 2
```

Output:

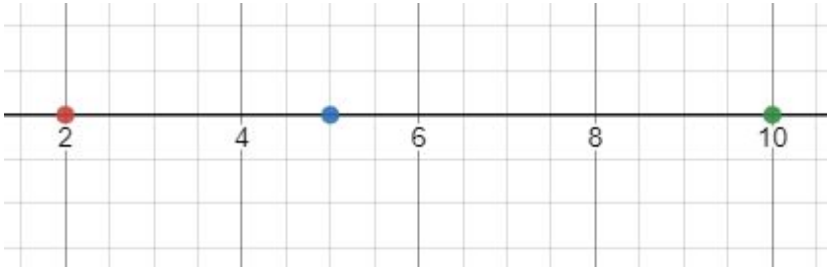
```
5
```

Stick Lengths Solution

- What length should we set all the sticks to?
  - The median!
- Why not the mean?
  - Counter example [1, 2, 1005]
    - Mean gives answer of 1338
    - Median gives answer of 1004

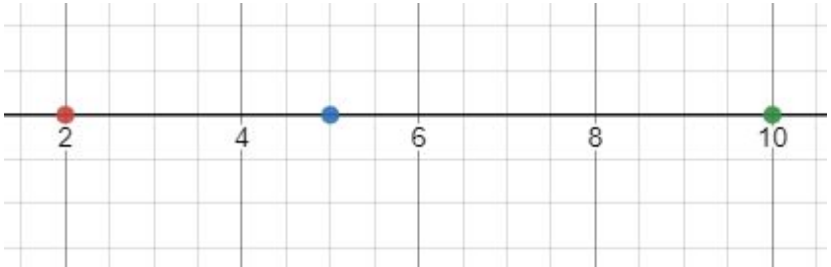
Stick Lengths Solution

- What length should we set all the sticks to?
  - The median!
- Choose arbitrary lengths ***A***, ***B***, and ***C***
  - Using the median yields an answer of abs(***A*** - ***B***) + abs(***B*** - ***B)*** + abs(***B*** - ***C***) = ***C*** - ***A***
  - Using the mean ***X*** = (***A*** + ***B*** + ***C***) / 3 gives and answer of (***C*** - ***A***) + abs(***B*** - ***X***)



Stick Lengths Solution

- What length should we set all the sticks to?
  - The median!
- Choose arbitrary lengths ***A***, ***B***, and ***C***
  - Using the median yields an answer of abs(***A*** - ***B***) + abs(***B*** - ***B)*** + abs(***B*** - ***C***) = ***C*** - ***A***
  - Using the mean ***X*** = (***A*** + ***B*** + ***C***) / 3 gives and answer of (***C*** - ***A***) + abs(***B*** - ***X***)
- The final answer becomes the distance of all sticks to the median length



Arranging The Sheep

https://codeforces.com/contest/1520/problem/E

E. Arranging The Sheep

time limit per test: 2 seconds
memory limit per test: 256 megabytes
input: standard input
output: standard output

You are playing the game "Arranging The Sheep". The goal of this game is to make the sheep line up. The level in the game is described by a string of length n, consisting of the characters '.' (empty space) and '*' (sheep). In one move, you can move any sheep one square to the left or one square to the right, if the corresponding square **exists and is empty**. The game ends as soon as the sheep are lined up, that is, there should be no empty cells between any sheep.

For example, if $n = 6$ and the level is described by the string "**.*..", then the following game scenario is possible:

- the sheep at the 4 position moves to the right, the state of the level: "**..*.";
- the sheep at the 2 position moves to the right, the state of the level: "*.*.*.";
- the sheep at the 1 position moves to the right, the state of the level: ".**.*.";
- the sheep at the 3 position moves to the right, the state of the level: ".*.**.";
- the sheep at the 2 position moves to the right, the state of the level: "..***.";
- the sheep are lined up and the game ends.

For a given level, determine the minimum number of moves you need to make to complete the level.

Input

The first line contains one integer t ($1 \le t \le 10^4$). Then t test cases follow.

The first line of each test case contains one integer n ($1 \le n \le 10^6$).

The second line of each test case contains a string of length n, consisting of the characters '.' (empty space) and '*' (sheep) — the description of the level.

It is guaranteed that the sum of n over all test cases does not exceed 10^6.

Output

For each test case output the minimum number of moves you need to make to complete the level.

Example

input

```
5
6
**.*..
5
*****
3
.*.
3
...
10
*.*...*.**
```

output

```
1
0
0
0
9
```

You are playing the game "Arranging The Sheep". The goal of this game is to make the sheep line up. The level in the game is described by a string of length n, consisting of the characters '`.`' (empty space) and '`*`' (sheep). In one move, you can move any sheep one square to the left or one square to the right, if the corresponding square **exists and is empty**. The game ends as soon as the sheep are lined up, that is, there should be no empty cells between any sheep.

For example, if $n = 6$ and the level is described by the string "`**.*..`", then the following game scenario is possible:

- the sheep at the 4 position moves to the right, the state of the level: "`**..*.`";
- the sheep at the 2 position moves to the right, the state of the level: "`*.*.*.`";
- the sheep at the 1 position moves to the right, the state of the level: "`.**.*.`";
- the sheep at the 3 position moves to the right, the state of the level: "`.*.**.`";
- the sheep at the 2 position moves to the right, the state of the level: "`..***.`";
- the sheep are lined up and the game ends.

For a given level, determine the minimum number of moves you need to make to complete the level.

Input

The first line contains one integer t ($1 \le t \le 10^4$). Then t test cases follow.

The first line of each test case contains one integer n ($1 \le n \le 10^6$).

The second line of each test case contains a string of length n, consisting of the characters '`.`' (empty space) and '`*`' (sheep) — the description of the level.

It is guaranteed that the sum of n over all test cases does not exceed 10^6.

Output

For each test case output the minimum number of moves you need to make to complete the level.

Example

input

```
5
6
**.*..
5
*****
3
.*.
3
...
10
*.*...*.**
```

output

```
1
0
0
0
9
```

Solution

- Almost the same problem as stick lengths!
- You want the sheep in the middle to stay in the same place and move the others towards it

Given an array of n integers, your task is to find the maximum sum of values in a contiguous, nonempty subarray.

Input

The first input line has an integer n: the size of the array.

The second line has n integers $x_1, x_2, \ldots, x_n$: the array values.

Output

Print one integer: the maximum subarray sum.

Constraints

- $1 \leq n \leq 2 \cdot 10^5$
- $-10^9 \leq x_i \leq 10^9$

Example

Input:

```
8
-1 3 -2 5 3 -5 2 2
```

Output:

```
9
```

Check in!

- Scan this QR code
- https://tinyurl.com/4a3vjz9x