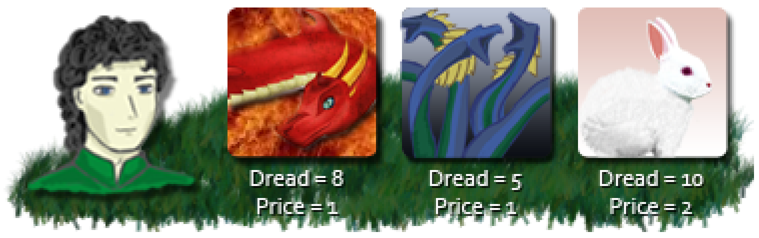
Problem Statement

Manao is traversing a valley inhabited by monsters. During his journey, he will encounter several monsters one by one. The scariness of each monster is a positive integer. Some monsters may be scarier than others. The i-th (0-based index) monster Manao will meet has scariness equal to **dread[i]**.

Manao is not going to fight the monsters. Instead, he will bribe some of them and make them join him. To bribe the i-th monster, Manao needs **price[i]** gold coins. The monsters are not too greedy, therefore each value in **price** will be either 1 or 2.

At the beginning, Manao travels alone. Each time he meets a monster, he first has the option to bribe it, and then the monster may decide to attack him. A monster will attack Manao if and only if he did not bribe it and its scariness is strictly greater than the total scariness of all monsters in Manao's party. In other words, whenever Manao encounters a monster that would attack him, he has to bribe it. If he encounters a monster that would not attack him, he may either bribe it, or simply walk past the monster.



Consider this example: Manao is traversing the valley inhabited by the Dragon, the Hydra and the Killer Rabbit. When he encounters the Dragon, he has no choice but to bribe him, spending 1 gold coin (in each test case, Manao has to bribe the first monster he meets, because when he travels alone, the total scariness of monsters in his party is zero). When they come by the Hydra, Manao can either pass or bribe her. In the end, he needs to get past the Killer Rabbit. If Manao bribed the Hydra, the total scariness of his party exceeds the Rabbit's, so they will pass. Otherwise, the Rabbit has to be bribed for two gold coins. Therefore, the optimal choice is to bribe the Hydra and then to walk past the Killer Rabbit. The total cost of getting through the valley this way is 2 gold coins.

You are given the int[]s **dread** and **price**. Compute the minimum price Manao will pay to safely pass the valley.

Definition

Class:	MonstersValley2
Method:	minimumPrice
Parameters:	int[], int[]
Returns:	int

Method signature: int minimumPrice(int[] dread, int[] price)
(be sure your method is public)

Constraints

- **dread** will contain between 1 and 20 elements, inclusive.
- Each element of **dread** will be between 1 and 2,000,000,000, inclusive.
- **price** will contain between the same number of elements as **dread**.
- Each element of **price** will be either 1 or 2.

Examples

0)

```
{8, 5, 10}
{1, 1, 2}
Returns: 2
```

The example from the problem statement.

1)

```
{1, 2, 4, 1000000000}
{1, 1, 1, 2}
Returns: 5
```

Manao has to bribe all monsters in the valley.

2)

```
{200, 107, 105, 206, 307, 400}
{1, 2, 1, 1, 1, 2}
Returns: 2
```

Manao can bribe monsters 0 and 3.

3)

```
{5216, 12512, 613, 1256, 66, 17202, 30000, 23512, 2125, 33333}
{2, 2, 1, 1, 1, 1, 2, 1, 2, 1}
Returns: 5
```

Bribing monsters 0, 1 and 5 is sufficient to pass safely.

Problem Statement

Given a string of digits, find the minimum number of additions required for the string to equal some target number. Each addition is the equivalent of inserting a plus sign somewhere into the string of digits. After all plus signs are inserted, evaluate the sum as usual. For example, consider the string "12" (quotes for clarity). With zero additions, we can achieve the number 12. If we insert one plus sign into the string, we get "1+2", which evaluates to 3. So, in that case, given "12", a minimum of 1 addition is required to get the number 3. As another example, consider "303" and a target sum of 6. The best strategy is not "3+0+3", but "3+03". You can do this because leading zeros do not change the result.

Write a class QuickSums that contains the method minSums, which takes a String **numbers** and an int **sum**. The method should calculate and return the minimum number of additions required to create an expression from **numbers** that evaluates to **sum**. If this is impossible, return -1.

Definition

Class:	QuickSums
Method:	minSums
Parameters:	String, int
Returns:	int
Method signature:	int minSums(String numbers, int sum)

(be sure your method is public)

Constraints

- **numbers** will contain between 1 and 10 characters, inclusive.
- Each character in **numbers** will be a digit.
- **sum** will be between 0 and 100, inclusive.

Examples

0)

```
"99999"
45
Returns: 4
```

In this case, the only way to achieve 45 is to add 9+9+9+9+9. This requires 4 additions.

1)

```
"1110"
3
Returns: 3
```

Be careful with zeros. 1+1+1+0=3 and requires 3 additions.

2)

```
"0123456789"
45
Returns: 8
```

3)

```
"99999"
100
Returns: -1
```

4)

```
"382834"
100
Returns: 2
```

There are 3 ways to get 100. They are 38+28+34, 3+8+2+83+4 and 3+82+8+3+4. The minimum required is 2.

5)

```
"9230560001"
71
Returns: 4
```

Problem Statement

We start with an integer and create a sequence of successors using the following procedure: First split the decimal representation of the given number into several (at least two) parts, and multiply the parts to get a possible successor. With the selected successor, we repeat this procedure to get a third number, and so on, until we reach a single-digit number.

For example, let's say we start with the number 234. The possible successors are:

- 23 * 4 = 92,
- 2 * 34 = 68 and
- 2 * 3 * 4 = 24.

If we select 68 as the successor, we then generate 6 * 8 = 48 (the only possibility), from this we generate 4 * 8 = 32 and finally 3 * 2 = 6. With this selection, we have generated a sequence of 5 integers (234, 68, 48, 32, 6).

Given the starting number, **start**, return the length of the longest sequence that can be generated with this procedure. In the example, the given sequence would be the longest one since the other selections in the first step would give the sequences: (234, 92, 18, 8) and (234, 24, 8), which are both shorter than (234, 68, 48, 32, 6).

Definition

Class:	NumberSplit
Method:	longestSequence
Parameters:	int
Returns:	int
Method signature:	int longestSequence(int start)

(be sure your method is public)

Notes

- There can not exist an infinite sequence.
- Although we use no leading zeros in the decimal representation of the number we start with, the parts we get by splitting the number may have leading zeros (e.g. from 3021 we can get 3 * 021 = 63).

Constraints

- **start** is between 1 and 100000, inclusive.

Examples

0)

```
6
```

```
Returns: 1
```

A single-digit number is already the last number.

1)

```
97
```

```
Returns: 4
```

For two-digit numbers, there is always only one possible sequence. Here: 97 -> 63 -> 18 -> 8 (4 numbers).

2)

```
234
```

```
Returns: 5
```

The example from the problem statement.

3)

```
876
```

```
Returns: 7
```

Here, it is optimal to make a three way split in the first step - i.e. use 8*7*6=336 as the first successor. Although a two way split would lead to a larger number (87*6=522 or 8*76=608), both these choices would lead in the end to a shorter sequence. The optimal sequence is: 876, 8*7*6=336, 33*6=198, 19*8=152, 1*52=52, 5*2=10, 1*0=0.

4)

```
99999
```

```
Returns: 29
```

Problem Statement

Your teacher has given you some problems to solve. You must first solve problem 0. After solving each problem i, you must either move on to problem i+1 or skip ahead to problem i+2. You are not allowed to skip more than one problem. For example, {0, 2, 3, 5} is a valid order, but {0, 2, 4, 7} is not because the skip from 4 to 7 is too long.

You are given a int[] **pleasantness**, where **pleasantness**[i] indicates how much you like problem i. The teacher will let you stop solving problems once the range of pleasantness you've encountered reaches a certain threshold. Specifically, you may stop once the difference between the maximum and minimum pleasantness of the problems you've solved is greater than or equal to the int **variety**. If this never happens, you must solve all the problems. Return the minimum number of problems you must solve to satisfy the teacher's requirements.

Definition

Class:	ProblemsToSolve
Method:	minNumber
Parameters:	int[], int
Returns:	int
Method signature:	int minNumber(int[] pleasantness, int variety)

(be sure your method is public)

Constraints

- **pleasantness** will contain between 1 and 50 elements, inclusive.
- Each element of **pleasantness** will be between 0 and 1000, inclusive.
- **variety** will be between 1 and 1000, inclusive.

Examples

0)

```
{1, 2, 3}
2
Returns: 2
```

Solve the 0-th problem, and the 2-nd after it.

1)

```
{1, 2, 3, 4, 5}
4
Returns: 3
```

Obviously, the first and the last problems should be solved. Skip a problem ahead twice in a row.

2)

```
{10, 1, 12, 101}
```

```
100
```

```
Returns: 3
```

3)

```
{10, 1}
9
```

```
Returns: 2
```

4)

```
{6, 2, 6, 2, 6, 3, 3, 3, 7}
4
```

```
Returns: 2
```

You can stop after solving the first 2 problems.

Problem Statement

DNA testing is one of the most popular methods of establishing paternity. In such a test, you compare samples of DNA from the man, the child, and the child's mother. For the purposes of this problem, assume that each sample is represented as a String of uppercase letters ('A'-'Z'). If half of the characters in the child's sample match the characters in the corresponding positions in the man's sample, and the remaining characters in the child's sample match the characters in the corresponding positions in the mother's sample, then the man is most likely the father. On the other hand, if it is impossible to partition the child's sample such that half of the characters match the man's and the other half match the mother's, then the man is definitely ruled out as the father.

For example, suppose the child's sample is "ABCD" and the mother's sample is "AXCY" (all quotes for clarity only). The 'A' and 'C' in the child's sample must have come from the mother, so the 'B' and 'D' must have come from the father. If the man's sample is "SBTD", then he is most likely the father, but if the man's sample is "QRCD", then he is definitely not the father. Note in the latter case that the man was definitely ruled out even though half of his sample (the 'C' and 'D') did in fact match the child's.

Your method will take samples from the child and the mother, as well as samples from several men, and return the indices of the men who cannot be ruled out as the father, in increasing order.

Definition

Class:	PaternityTest
Method:	possibleFathers
Parameters:	String, String, String[]
Returns:	int[]
Method signature:	int[] possibleFathers(String child, String mother, String[] men)

(be sure your method is public)

Notes

- You may assume that the identity of the mother is not in question.

Constraints

- **men** contains between 1 and 5 elements, inclusive.
- **child**, **mother**, and all elements of **men** contain the same number of characters, which is even and between 2 and 20, inclusive.
- **child**, **mother**, and all elements of **men** contain only uppercase letters ('A'-'Z').
- At least half of the characters in **mother** match the characters in the corresponding positions in **child**.

Examples

0)

```
"ABCD"
"AXCY"
{ "SBTD", "QRCD" }
```

```
Returns: { 0 }
```

The example above.

1)

```
"ABCD"
"ABCX"
{ "ABCY", "ASTD", "QBCD" }
```

```
Returns: { 1,  2 }
```

"ABCY" could not be the father. "ASTD" could be the father, with the 'A' and 'D' coming from the father and the 'B' and 'C' coming from the mother. "QBCD" could also be the father, with either the 'B' and 'D' or the 'C' and 'D' coming from the father.

2)

```
"ABABAB"
"ABABAB"
{ "ABABAB", "ABABCC", "ABCCDD", "CCDDEE" }
```

```
Returns: { 0,  1 }
```

3)

```
"YZGLSYQT"
"YUQRWYQT"
{"YZQLDPWT", "BZELSWQM", "OZGPSFKT", "GZTKFYQT", "WQJLSMQT"}
```

```
Returns: { }
```

4)

```
"WXETPYCHUWSQEMKKYNVP"
"AXQTUQVAUOSQEEKCYNVP"
{ "WNELPYCHXWXPCMNKDDXD",
  "WFEEPYCHFWDNPMKKALIW",
  "WSEFPYCHEWEFGMPKIQCK",
  "WAEXPYCHAWEQXMSKYARN",
  "WKEXPYCHYWLLFMGKKFBB" }
```

```
Returns: { 1,  3 }
```

Problem Statement

In this problem, a left single arrow is defined as a "less than" character ('<') immediately followed by zero or more consecutive hyphen characters ('-'). A left double arrow is a "less than" character ('<') immediately followed by zero or more consecutive equals characters ('='). A right single arrow is zero or more hyphen characters ('-') immediately followed by a "greater than" character ('>'). A right double arrow is zero or more equals characters ('=') immediately followed by a "greater than" character ('>'). For example, the following are arrows (quotes for clarity only): "==>", "<-", "<", "<===", "--->", ">". The length of an arrow is the number of characters it contains.

You will be given a String **s**. Return the length of the longest arrow in **s**, or -1 if it does not contain any arrows.

Definition

Class:	Arrows
Method:	longestArrow
Parameters:	String
Returns:	int
Method signature:	int longestArrow(String s)

(be sure your method is public)

Notes

- Arrows may overlap. See examples for further clarifications.

Constraints

- **s** will contain between 1 and 50 characters, inclusive.
- Each character in **s** will be one of '<', '>', '-' or '='.

Examples

0)

```
"<--->--==>"
Returns: 4
```

The arrows contained in **s** in this case are, by order of appearance:

"<", "<-", "<--", "<---", "--->", "-->", "->", ">", "==>", "=>" and ">".

Note that many of these arrows, including some pairs with one left arrow and one right arrow, are overlapping.

1)

```
"<<<<<<<<<<"
```

```
Returns: 1
```

All arrows are of length 1. Note that "<" is both a left single arrow and a left double arrow according to the definition.

2)

```
"----==-"
```

```
Returns: -1
```

No arrows contained.

3)

```
"<----=====>"
```

```
Returns: 6
```

Problem Statement

Often the maintenance staff of a large building wants to verify that the elevator in the building is being used appropriately. The manufacturer designates safety limits regarding the maximum number of people that should be on the elevator at the same time. Also, there is a physical limitation as to how many people can actually be in the elevator simultaneously. The building in question has very primitive sensing equipment, and only has data telling how many people entered and exited on a particular stop. That is, the data does not explicitly state how many people were on the elevator at any given point in time.

You are to write a class ElevatorLimit, with a method getRange, which takes as parameters a int[] **enter**, a int[] **exit**, and an int **physicalLimit**, where the *i*th element in each int[] indicates the number of people that entered and exited, respectively, the elevator on the *i*th stop. At each stop people exit the elevator first, and then people enter. **physicalLimit** is how many people are physically capable of being in the elevator simultaneously. You are to determine the maximum and minimum numbers of people that could have been on the elevator before the simulation begins, and return these values in a int[] containing exactly two elements. The first element of the return value is the minimum, and the second element is the maximum number.

If the data entered yields an impossible situation, you are to return an empty int[].

Definition

Class:	ElevatorLimit
Method:	getRange
Parameters:	int[], int[], int
Returns:	int[]
Method signature:	int[] getRange(int[] enter, int[] exit, int physicalLimit)

(be sure your method is public)

Constraints

- **enter** and **exit** will have between 1 and 50 elements, inclusive.
- **enter** and **exit** will have the same number of elements.
- Each element of **enter** and **exit** will be between 0 and 1000, inclusive.
- **physicalLimit** will be between 1 and 1000, inclusive.

Examples

0)

```
{1,0}
{0,1}
1
Returns: { 0,  0 }
```

With a **physicalLimit** of one person on the elevator at a time, there had to be 0 to start with.

1)

```
{1,0}
{0,1}
2
Returns: { 0,  1 }
```

With a **physicalLimit** of 2 people, the elevator could have started with 0 or 1 people on it.

2)

```
{0,1}
{1,0}
1
Returns: { 1,  1 }
```

3)

```
{0,2}
{1,0}
1
Returns: { }
```

Since there is only a maximum of 1 person, 2 people cannot get on at the second stop. Therefore, there is no possible solution.

4)

```
{6, 85, 106, 1, 199, 76, 162, 141}
{38, 68, 62, 83, 170, 12, 61, 114}
668
Returns: { 223,  500 }
```

5)

```
{179, 135, 104, 90, 97, 186, 187, 47, 152, 100, 119, 28, 193, 11, 103, 100,
 179, 11, 80, 163, 50, 131, 103, 50, 142, 51, 112, 62, 69, 72, 88, 3, 162,
 93, 190, 85, 79, 86, 146, 71, 65, 131, 179, 119, 66, 111}
{134, 81, 178, 168, 86, 128, 1, 165, 62, 46, 188, 70, 104, 111, 3, 47, 144,
 69, 163, 21, 101, 126, 169, 84, 146, 165, 198, 1, 65, 181, 135, 99, 100,
 195, 171, 47, 16, 54, 79, 69, 6, 97, 154, 80, 151, 76}
954
Returns: { 453,  659 }
```

6)

```
{2}
{3}
2
Returns: { }
```

Problem Statement

There are two types of egg cartons. One type contains 6 eggs and the other type contains 8 eggs. John wants to buy exactly **n** eggs. Return the minimal number of egg cartons he must buy. If it's impossible to buy exactly **n** eggs, return -1.

Definition

Class:	EggCartons
Method:	minCartons
Parameters:	int
Returns:	int
Method signature:	int minCartons(int n)

(be sure your method is public)

Constraints

- **n** will be between 1 and 100, inclusive.

Examples

0)

```
20
Returns: 3
```

He should buy 2 cartons containing 6 eggs and 1 carton containing 8 eggs. In total, he buys 3 egg cartons.

1)

```
24
Returns: 3
```

There are two ways to buy 24 eggs: buy 4 cartons containing 6 eggs or buy 3 cartons containing 8 eggs. Minimize the number of cartons.

2)

```
15
Returns: -1
```

He can't buy an odd number of eggs.

3)

```
4
Returns: -1
```

Problem Statement

The Fibonacci sequence is defined as follows:

- F[0] = 0
- F[1] = 1
- for each i >= 2: F[i] = F[i-1] + F[i-2]

Thus, the Fibonacci sequence starts as follows: 0, 1, 1, 2, 3, 5, 8, 13, ... The elements of the Fibonacci sequence are called Fibonacci numbers.

You're given an int **N**. You want to change **N** into a Fibonacci number. This change will consist of zero or more steps. In each step, you can either increment or decrement the number you currently have. That is, in each step you can change your current number X either to X+1 or to X-1.

Return the smallest number of steps needed to change **N** into a Fibonacci number.

Definition

Class:	FibonacciDiv2
Method:	find
Parameters:	int
Returns:	int
Method signature:	int find(int N)
(be sure your method is public)	

Constraints

- **N** will be between 1 and 1,000,000, inclusive.

Examples

0)

```
1
Returns: 0
```

The number 1 is already a Fibonacci number. No changes are necessary.

1)

```
13
Returns: 0
```

The number 13 is also a Fibonacci number.

2)

```
15
```

```
Returns: 2
```

The best way to change 15 into a Fibonacci number is to decrement it twice in a row (15 -> 14 -> 13).

3)

```
19
```

```
Returns: 2
```

You can increase it by 2 to get 21.

4)

```
1000000
```

```
Returns: 167960
```

This is the biggest possible number that you can get as input.

Problem Statement

You are writing firmware for an exercise machine. Each second, a routine in your firmware is called which decides whether it should display the percentage of the workout completed. The display does not have any ability to show decimal points, so the routine should only display a percentage if the second it is called results in a whole percentage of the total workout.

Given a String **time** representing how long the workout lasts, in the format "hours:minutes:seconds", return the number of times a percentage will be displayed by the routine. The machine should never display 0% or 100%.

Definition

Class:	ExerciseMachine
Method:	getPercentages
Parameters:	String
Returns:	int
Method signature:	int getPercentages(String time)

(be sure your method is public)

Constraints

- **time** will be a String formatted as `"HH:MM:SS"`, HH = hours, MM = minutes, SS = seconds.
- The hours portion of **time** will be an integer with exactly two digits, with a value between 00 and 23, inclusive.
- The minutes portion of **time** will be an integer with exactly two digits, with a value between 00 and 59, inclusive.
- The seconds portion of **time** will be an integer with exactly two digits, with a value between 00 and 59, inclusive
- **time** will not be "00:00:00".

Examples

0)

```
"00:30:00"
Returns: 99
```

The standard 30 minute workout. Each one percent increment can be displayed every 18 seconds.

1)

```
"00:28:00"
Returns: 19
```

The 28 minute workout. The user completes 5 percent of the workout every 1 minute, 14 seconds.

2)

```
"23:59:59"
```

```
Returns: 0
```

This is the longest workout possible, given the constraints. No percentages are ever displayed on the screen.

3)

```
"00:14:10"
```

```
Returns: 49
```

4)

```
"00:19:16"
```

```
Returns: 3
```
