1st Annual

Brookfield Computer Programming Challenge

2017

Problem Set

Problems:

1.	Passing Bills [Bills.java]	- Teal
2.	Apple Pen [Applepen.java]	- Red
3.	The Halting Problem [Halting.java]	- Yellow
4.	One [One.java]*	- Blue
5.	Negligent Norbert [Negligence.java]*	- Brown
6.	Calculatus Eliminatus [Calculatus.java]*	- Pink
7.	Bike Race [Race.java]*	- White

*For a program to be correct, it must output the correct answer within 5 seconds of being called *for any input that is within the given constraints*. A computer can do about 10^8 iterations in 5 seconds.
**If you use packages, you will get all the problems wrong. Please don't.
***The word "Integer" refers to the mathematical definition of the word, not the data type [so it isn't necessarily less than 2^31, for example]. *Be very careful of the input constraints.*

Passing Bills [Bills.java] - Teal

The Bi-Cameral Progress Committee loves making progress by passing bills. Since the committee is determined to make as much progress as possible, all votes are required to be of unanimous approval of the bill at hand. *That is, if the committee votes on whether to pass a bill, it will definitely be passed.*

Because these committee members are all political science majors [instead of computer science majors], they much prefer reading the flavortext of unnecessarily convoluted problem statements to using logic or algebra, and assume you prefer the same. They could have asked you for the shortest path of a bill to touch each member given the position of all committee members. Or they could have asked you the maximum value of any passable combination of bills, given the total political capital of the committee and political cost and value of each potential bill. Or they could have asked whether there exist are any bill whose text is of the complexity that can be read in polynomial time but cannot be written in polynomial time.

Instead, they need your help figuring out how many bills the committee will pass given the number of bills the committee will vote on. **Help them figure out how many bills will be passed given the *number of bills that will be voted on and the total number of french fries eaten since 1970*.**

Input:

- The input will contain two, space-separated integers: **n**, the number of bills to be voted on this session, and **f**, the total number of french fries eaten since 1970 [in billions of fries, rounded to the nearest integer].

0 ≤ **n** ≤ 99
0 ≤ **f** ≤ 20

Output:

- Output a single integer: the number of bills that will be passed this session.

Example:
Input:
7 20

Output:
7

Explanation:
The committee will vote 7 times, and each time chose to pass the bill, leading to 7 bills being passed (See paragraph 1). So we output 7. Also, as of the time of the bill being passed, 20 billion french fries will have been eaten since 1970.

Example 2:
Input:
98 13

Output 2:
98

Apple-Pen (Applepen.java) - Red

After visiting the Apple store and playing with the new iPad Pro, many are hopelessly attracted to having their own Apple Pencil. However, because Apple Pencils have been brutally trademarked and are obscenely expensive, such visitors must settle for a compromise: making an Apple-Pen! Apple-Pens can be created by uh-ing a Pen with an Apple (uh-ing an Apple with a Pen would create a Pen-Apple, which, of course, is not nearly as cool).

It has recently been discovered that this uh function can be applied to things other than apples and pens! Uhing item A with item B will create a new item: B-A. Given 2***n** items, ***what will result when each pair of items are uhed?***

Input:

- The first line will contain a single integer **n**, the number of times to perform the uh operation.
- 2***n** lines follow. Each line will contain a sentence of the form "I have a <item>" where <item> is replaced by a single word of upper case letters, lower case letters, and hyphens (-), but not spaces.

$1 \le \mathbf{n} \le 1000$
$1 \le \text{charactersPerItemName} \le 1000$

Output:

- Output n lines each of the form "Uh! <item b>-<item a>!" where <item b> and <item a> are replaced by their respective items.

Example:

Input:

```
2
I have a Pen                        //← Test case 1
I have a Apple
I have a Apple-Pen                  //← Test case 2
I have a Pen-Pineapple
```

Output:

```
Uh! Apple-Pen!
Uh! Pen-Pineapple-Apple-Pen!
```

Explanation:

There are two test cases. In the first, a Pen is uhed with an Apple to create an Apple-Pen.

The Halting Problem [Halting.java] - Yellow

In the deterministic land of Babbalon, founded by Charles Babbage, of course, all cars have been replaced by fully-autonomous, Turing-complete driving machines. For safety reasons, the two driving machines will never be in the same intersection at the same time, and a machine will halt before an intersection if there is another machine in that intersection. Babbage, late to his afternoon meeting with Mrs. Lovelace, would like to know whether any of the driving machines will stop before going through an intersection.

If **n** driving machines each take **s** seconds to pass through an intersection, each one arriving at time $\mathbf{t_i}$, ***will any machines have to stop before crossing the intersection?***

The times the driving machines will arrive at the intersection will not necessarily be inputted in chronological order. A machine will not halt if it reaches the intersection at the exact same that time another leaves it.

Input:

- The first line will contain a single integer **T**, the number of test cases
- The first line of each test case will contain two integers, **n** and **s**, the number of driving machines and the number of seconds it takes any driving machine to get through the intersection.
- **n** lines follow, each containing a single integer, $\mathbf{t_i}$, the time in seconds that the i^{th} car arrives at the intersection

$0 \le \mathbf{T} \le 40$
$0 \le \mathbf{n}, \mathbf{s} \le 100$
$0 \le \mathbf{t_i} \le 1000$

Output:

Output **T** lines, each containing either “YES” if at least one driving machine will have to halt, or “NO” otherwise.

Example:

Input:

```
2
3 3
1
4
10
2 5
4
0
```

Output:

```
NO
YES
```

Explanation:

For the first test case, driving machines will arrive at times 1, 4, and 10, and take 3 seconds to get through. No machine will have to wait for its turn, so we output "NO".

For the second test case, a driving machine that arrives at time 0 will occupy the intersection up to, but not including, time 5. When another driving machine arrives at time 4, it will have to wait, so we output "YES".

One (One.java) - Blue

Hooray! It's time for the obligatory problem concerning the random mathematical properties of an arbitrary number! And since today is the 1st Brookfield Computer Programming Challenge, this year's number is 1!

One is a pretty unique number. Of course, it is the multiplicative identity (that is x*1 = x), as well as the smallest natural number. It is "the loneliest number," the maximum value of Random.nextDouble(), the magnitude of any unit vector, and the only positive number that is a valid digit in any base, as well as power of 2 itself (2^0==1, oh joy!). It is also both the second and third Fibonacci number (0, **1**, **1**, 2, 3, 5, 8...). In fact, it is the only number to appear a prime number of times in the Fibonacci sequence.

Speaking of primes, whether or not 1 is a prime number has been discussed quite fervently among mathematicians. A prime is a number with exactly two unique natural factors. A composite number is a number with more than two unique natural factors. 7, for example, has factors of 1 and 7, but no others, making it prime. 9 has factors of 1, 3, and 9, making it composite. 1, however, is only divisible by 1, and is therefore neither prime nor composite. Interestingly, 1 is the only number with this property.

Given **n** natural numbers, ***decide whether each of them is prime, composite, or neither***. Be careful that your program finishes execution within 5 seconds for any valid input.

Input:

- The first line will contain an integer **n**, the number of natural numbers to compute.
- **n** lines follow, each containing a single integer $\mathbf{q}_i$

1 ≤ **n** ≤ 100
1 ≤ $\mathbf{q}_i$ ≤ 2*10^9

Output:

- Output **n** lines, each containing either "Prime", "Composite", or "Neither", depending on the state of the number

Example:

Input:
```
4
9
1
2017
1000000007
```

Output:
```
Composite
Neither
Prime
Prime
```

Explanation:
9 has 3 factors: 1,3, and 9, and is therefore composite.
1 is discussed in the problem statement.
2017 only has factors of 1 and itself, and is therefore prime.
1000000007 also happens to be prime for the same reason.

Negligent Norbert (Negligence.java) - Brown

Naughty Negligent Norbert has been given a notoriously repetitive responsibility: answering the clarification requests for a programming competition! Norbert realizes that since all the questions are perfectly unambiguous, he doesn't really have to do his job, and the only reason he is responding to requests at all is so he can get service hours. The savvy programmers[1], however, have implemented a system to prevent Norbert from simply giving the same answer to all clarification requests.

Having seen the source code for the competition, Norbert knows that he will lose all his service hours for not doing his job correctly if any two of his responses are identical. Negligent Norbert decides that for each of his **T** competitions, he will respond to the **q** clarification requests using as few characters as possible. Also, because Norbert is very diverse and speaks many languages, he will answer all questions in a given competition using a language with **n** characters. ***What is the fewest number of keystrokes Norbert can type in each competition to still get his service hours?***

Each response must have at least 1 character, and responses do not necessarily all have to be the same length. Norbert will click the "submit" button with his mouse instead of pressing the return key, *so the total number of keystrokes is the sum of the lengths of each response.*

Input:

- The first line will contain an integer **T.**
- **T** lines follow, each containing two space-separated integers, **q** and **n**, the number of clarification requests and the number of characters in the language for the current competition.

1 ≤ **T** ≤ 100
1 ≤ **q** ≤ 10^11
2 ≤ **n** ≤ 10^5

Output:

- Output **T** lines, each containing a single integer representing the number of characters in all of Norbert's responses for a competition.

[1] One of such savvy programmers was Vim Trakas, who wrote all of the source code using his ~~namesake~~ favorite terminal-based text editor, nano.

Example:

Input:
```
3
7 3
5 26
14 3
```

Output:
```
11
5
27
```

Explanation:

For the first test case, Norbert is using a language with 3 characters, Norbert could answer, for example, AC, CA, A, BC, C, AA, and B, for a total character count of 2+2+1+2+1+2+1 = 11.

For the second test case, Norbert could answer each of the five clarification requests with a different character, using a total of 5 characters. So we output 5 on a new line.

For the third test case, Norbert will need to use all of the one-letter and two-letter words, and two more three-letter words, for a total of 27 characters.

Calculatus Eliminatus [Calculatus.java] - Pink

When you've mislaid a certain something, keep your cool and don't get hot.
Calculatus Eliminatus is the best friend that you've got!
Calculatus Eliminatus always helps an awful lot;
The way to find a missing something is to find out where it's not.

Mark the places it could be all from 1 up to **n**
--But careful now 'cuz **n** could be up to two billion ten!--
Then cross off all **u** ranges, all inclusive start and end,
And you'll know all of the places that the object hasn't been!

We just jot down all the places where it isn't and gee wiz--
Very shortly we will locate ***where the missing object is***!

Input:

- The first line contain two integers **n** and **u**
- **u** lines follow, each containing two integers representing the start and end of a range, respectively [ranges can be overlapping]

1 ≤ **n** ≤ 2,000,000,010
0 ≤ **u** ≤ 1,000

Output:

- Output a single integer representing the location of the missing object. There will only be one possible location it can be.

Example:

Input:
```
10 3
7 10
1 1
3 8
```

Output:
```
2
```

Explanation:
```
12345678910
??????????
      XXXX
X
  XXXXXX
=
X?XXXXXXXX
```

So, by using Calculatus Eliminatus, we have found that the missing object must be at position 2. (Notice that ranges can, but need not, overlap)

Example 2:

Input:
```
2000000010 2
1 8
10 2000000010
```

Output:
```
9
```

Explanation:

The missing object is at position 9. Your program should give the answer within 5 seconds (And not run out of heap space with default settings; if you run out of heap space, that means you can't make an array that big, so try a different approach).

Bike Race [Race.java] - White

Tyler, Daniel, and Billy, captains of the Doofenshmirtz biker-gang, decide to have an irresponsibly destructive race across the tri-state area that contains **n** intersections and **m** bidirectional roads. In the interest of having their unmuffled engines create the optimal amount of chaos, they want their race to pass through as many intersections as possible. However, since all three racers are logically infallible and bike optimally, they know that the others will race to the finish passing through the fewest intersections necessary. The racers agree that the race should both start and end at an intersection.

Can you help them find the ***length of the race*** that will pass through the greatest number of the **n** intersections in the tri-state area, if all three racers pass through the fewest intersections necessary?

It is guaranteed that all intersections will be directly or indirectly reachable from all others.

Input:

- The line contains a single integer **T**, the number of test cases.
- The first line of each test cases contains two integers **n**, the number of intersections, and **m**, the number of roads
- **m** lines follow, each containing two integers representing two intersections connected by a road. Intersections are 0-indexed (they are numbered 0, 1, 2, 3, … **n**-1). So "0 2" would mean that a road connects intersection 0 and intersection 2.

0 ≤ **T** ≤ 100
0 ≤ **n, m** ≤ 500

Output:

- Output **T** lines, with each containing a single integer representing the minimum number of intersections the racers will need to pass through in the longest possible race.

Example:
Input:

```
2
3 3          //← test case 1
1 2
0 1
2 0
5 4          //← test case 2
0 1
2 1
2 3
4 3
```

Output:

```
2
5
```

Explanation:

There are two test cases. The first test case consists of 3 intersections and 3 roads, with each intersection directly connect to each other intersection. If any two cities are picked as starting and ending locations, the racers will take a path that passes only through those two cities.

For the second test case, there are 5 cities, with each one connected to the two ones with adjacent numerical representations. The longest race possible would start at intersection 0 and end at intersection 4, passing through all 5 intersections.