CSC 495 – Problem Set 1 – Due Tuesday, January 17

Problem 1.R1: How Many Bits?

Required Problem
Points: 50 points

Background

When a number is stored in a primitive type, like an `int` or `long` variable, it always uses the same number of bits (32 or 64, typically). But if we don't need to store the leading 0 bits, we might want to know how many bits are actually required to represent a number. For example, the number 13 in binary is `1101`, which requires only 4 bits to write down. Note: For a useful example of when you need to know how many bits are required to store a number or a counter, see problem 6 in Column 1 of *Programming Pearls* (or Problem 1.C2 in this problem set!).

The Problem to Solve

You are to read a sequence of numbers, and for each one output how many bits are required to store that number. The numbers will all be in the range $1, \ldots, 10^{18}$. You are required to *compute* the answer using only basic boolean operations — you may not just call a library function!

Hints and Techniques

The allowable range for input numbers is important — pay attention to it, and make sure you can handle all numbers in that range!

Elegance Considerations

Avoid hard-coding numbers or making ugly special-case code. Your goal is a very clean and simple solution!

Input and Output

The input will consist of one line with the count of how many numbers follow (call this number n), and then n lines each containing a single number in the range $1, \ldots, 10^{18}$. For each of the n numbers you should print the number of bits required to represent that number.

Sample Input

```
4
13
5
40000
5555555555555
```

Sample Output

```
4
3
16
43
```

Problem 1.R2: Counting Prime Numbers

Required Problem
Points: 50 points
Ninja Points: 5 extra ninja points to the fastest solution, and any within 5% of the fastest time.

Background

An important and interesting area of Mathematics studies the distribution and density of prime numbers. In number theory, they use the function $\pi(x)$ to represent "The number of prime numbers less than or equal to x" (this is not related in any way to the geometric constant π – it just happens to use the same Greek letter). As an example, $\pi(15) = 6$ since there are 6 prime numbers less than or equal to 15 (specifically, 2, 3, 5, 7, 11, and 13). There is no known way to compute $\pi(x)$ efficiently for large x, so people consider techniques for approximating $\pi(x)$. It turns out that we could approximate $\pi(x)$ with pretty high accuracy if an unproven Mathematical hypothesis, known as the Riemann Hypothesis, is true. Settling this unsolved Mathematics problem is in fact one of the Clay Institute "Millennium Prize Problems," a list from 2000 of what the Clay Institute considered to be the seven most important unsolved Mathematics problems. Anyone who solves this problem will be awarded a million dollars by the Clay Institute. Fortunately, you do not need to solve the Riemann Hypothesis for this problem (however, 1,000,000 ninja points and an automatic T-shirt if you do!).

The Problem to Solve

In order to consider approximations for $\pi(x)$, it is important to know what the exact value is for different values of x. You are to write a program that computes $\pi(1,000,000,000)$ – the number of prime numbers less than or equal to a billion.

Hints and Techniques

Consider using a boolean array with indices from 1 to 1,000,000,000, called `isPrime[]`. We will set the entry `isPrime[x]` to `true` if and only if `x` is a prime number. If you can fill in this array, then you can simply count the number of `true` entries to see how many prime numbers there are.

To fill in the array, generalize the following procedure: first, assume everything ≥ 2 is prime (so set all entries to `true`). Then starting at 4 (which is 2*2), step through the boolean array and mark every other entry false, since these are the multiples of 2 (this is sometimes referred to as going through the array with "stride" 2). Next, starting at 6 (which is 2*3), walk through with stride 3, setting all multiples of 3 to `false`. Now continue looking from 3 and find the next `true` entry — remember that we set 4 to false, so the next `true` entry is 5, and you can mark out all multiples of 5. You can repeat this, looking for the next larger prime and crossing off its multiples, until you have marked out all non-prime values in the array.

One question to ask, for efficiency: When can you stop? In other words, if an iteration is crossing off all multiples of prime p, how large does p have to get before you stop? To rephrase this question a little bit, consider an arbitrary non-prime $x \leq 1,000,000,000$: Can you fill in m in the statement "x must have at least one prime factor $\leq m$"? If you can do that, then you can stop your loop once your prime "stride" becomes larger than m.

Elegance Considerations

There are several ways to implement this, and as long as your code can correctly compute $\pi(1,000,000,000)$ in less than 10 minutes on `linux.uncg.edu` you will get full correctness points. However, to get full elegance points, you will need to code things efficiently so that your program runs in less than two minutes and uses less than 150 megabytes of memory. To do this in a space-efficient manner, you should use techniques similar to those described in Column 1 of *Programming Pearls*.

Input and Output

There is no input for this problem, and the required output is the number of prime numbers that are $\leq 1,000,000,000$. This number is fairly easy to find on the Internet, but you get no correctness credit just for printing that number. It must be *computed* correctly!

Problem 1.C1: No Carries Allowed

Challenge Problem
Ninja Points: This challenge problem is worth up to 20 base ninja points

Background

Adding numbers in binary is similar to the exclusive-OR, or XOR, operation, as you can see in the following table (here ^ denotes the XOR operation):

x	y	x^y	x+y
0	0	0	0
0	1	1	1
1	0	1	1
1	1	0	10

Note that the least significant bit of the sum is always equal to the XOR of the two input bits, but since the output of the XOR is a single bit it has no place to put the "carry bit" that you get in the sum.

When you use the XOR operator (^) in C, C++, or Java, it operates on all bits in parallel, and is called a "bitwise operator" since it works on each bit position independently. For some numbers, the XOR of the numbers is the same as the sum, and for others it isn't. To take two examples, 17 and 6 give the same result:

```
17+6 = 10001 + 110 = 10111
17^6 = 10001 ^ 110 = 10111
```

but 21 and 6 give different results:

```
21+6 = 10101 + 110 = 11011
21^6 = 10101 ^ 110 = 10011
```

The Problem to Solve

Given a integers x and y, you can consider the set $E = \{y \,|\, x + y = x\ XOR\ y\}$, which is the set of all values that produce the same results when added and XORed with x. We would like to be able to find the kth smallest element in the set E for any input x. x and k will be in the range $1, \ldots, 10^{18}$, and you are further guaranteed that the answer will be in the range $0, \ldots, 10^{18}$. Your program will be given at most 5 seconds to process each pair of numbers.

Hints and Techniques

The range of the input numbers is important! It's pretty easy to do this for small numbers, but once numbers get large (say around 10^{12}), if you don't use the correct algorithm then your program will not be be able to produce an answer in a reasonable amount of time. The correct algorithm will run in time proportional to the number of bits in the input numbers, not the values of the numbers. If you can't figure out the algorithm right away, write out the numbers in the sample in put and output in binary, and look for patterns for a clue.

If your program cannot process all numbers in the specified range in under 5 seconds for each pair, then it is not correct. However, some partial credit (but very little – at most 5 ninja points!) will be given if your program produces correct answers in under 5 seconds when the input values are limited to the range $1, \ldots, 10^6$.

Input and Output

The input will consist of one line with the count of how many pairs follow (call this number n), and then n lines each containing a pair of values x and k. For each pair, you should output the k-th smallest number y for which $x + y = x\hat{}y$.

Sample Input

```
5
1 8
7 1
17 3
21 4
23261 133
```

Sample Output

```
14
0
4
10
131328
```

Problem 1.C2: Most Popular BFF

Challenge Problem
Ninja Points: This challenge problem is worth up to 20 base ninja points
Special Note: The submission program will only run tests on your program on smaller inputs (a million items or less). The full test takes a while to run (just reading the file takes some time, since it is over 3 Gigabytes!). Therefore you will not get any feedback on whether your program is fast enough on the big data set — you need to be able to figure out on your own whether your program is fast enough. Submissions will each be tested a single time on the large input, after the due date when submissions are final.

Background

Sometimes you need to count a *lot* of items, requiring a lot of space, and so you need to use the most space-efficient representation possible. Imagine if we held a contest to see which person in the United States was "best friend" to the most other people. Forgetting all the privacy aspects of Social Security Numbers (SSNs), we will ask each person for the SSN of their best friend since SSNs are convenient and unique identifying numbers. How would we process this data to find which SSN was given most often?

The Problem to Solve

You are given a list of SSNs, each 9 digits long and without dashes (so the range is 100000000 to 999999999), in no particular order. You are guaranteed that no number appears more than 30 times (some people might be popular, but come on...). You are to find which number appears most often, and how often it occurs. Your program should be able to process around 300,000,000 SSN entries in under 15 minutes (but see the "special note" above).

Hints and Techniques

See problem 6 at the end of Column 1 in *Programming Pearls*.

Elegance Considerations

Note that to have an array with count value for each SSN, where each count is a 4-byte `int`, would require 4 Gigabytes of memory. While we are seeing more and more systems with 4 GB of RAM, this is too much to reliably count on. You should code this so that it takes no more than 1 GB of memory. There will be up to 310 million numbers provided (the approximate population of the United States).

Input and Output Format

The input will consist of one line with the count of how many lines follow (call this number n), and then n lines each containing a single SSN. The output of your program should be a single line of output, with two numbers on it: The most commonly occurring number first, and the count of how many times it occurs second. If there is more than one number with the same "most occurring count", output the smallest such number.

Note that the sample input given below is obviously much, much shorter than in the real data. In fact, the algorithm you would want to use for just 10 numbers would be very different from that used for 300,000,000 numbers, but the provided output is still correct!

Sample Input

```
10
735917530
753068294
544633009
735917530
544633009
735917530
544633009
735917530
341766943
753068294
```

Sample Output

```
735917530 4
```