CS:3330 Homework 6, Fall 2015

Due in class on Thu, Dec 10

Note: This homework is longer than usual, but is not harder than usual.

1. Suppose that we want to compute the value of the expression

$$a^d \bmod n$$

for positive integers a, d, and n, where a and d are guaranteed to belong to $\{1, 2, \ldots, n-1\}$. Since a and d are both guaranteed to be less than n, we know that the input size is $\Theta(\log n)$. Therefore, an efficient (i.e., polynomial time) algorithm for the problem is one that runs in $O(\log^c n)$ for some constant c. In this problem you are required to design *and implement* an $O(\log^3 n)$-time algorithm. In practical terms, I want your function to be able to run very fast (essentially instantaneously) on pretty large numbers, e.g., numbers with roughly 100 digits. You can use your favorite high level language (e.g., `Java`, `C`, `C++`, `Python`, `Scala`, etc.). Specifically, I want you to implement a function – let us call it `bigPowerMod` – that takes as arguments a, d, and n and returns $a^d \bmod n$.

There are two efficiency-related issues to pay attention to.

(i) It is possible to compute a^d by performing $d-1$ multiplications. But, this is too many because $d-1$ can be as large as $n-2$ and therefore performing so many multiplications would take $\Theta(n)$ time. For another way of seeing how inefficient this would be suppose that n is a 100-digit number, then your program would be performing, roughly 10^{100} multiplications, which would literally take forever!

(ii) While your final answer is an integer in the range $[0, n-1]$ (because of the $\bmod n$), you have to watch out for the size of intermediate answers. When we mutiply a number with b bits with another number with b' bits, the answer can have answer many as $b+b'$ bits. This means that if a has $\log_2 n$ bits, then a^2 can have $2\log_2 n$ bits, a^3 can have $3\log_2 n$ bits, and so on. Thus a^d can have $d \log n$ bits, which is $\Theta(n \log n)$ in the worst case. Again, if n is a 100-digit number, this amounts to more that 10^{100} bits (which is more than the number of atoms in the universe!). Of course, carrying such large intermediate answers around also means that each multiplication will take forever.

You can use *divide-and-conquer* to solve the first problem. Here is a hint: to compute a^d, you can (recursively) compute $a^{d/2}$ and after that it takes one or two multiplications, depending on whether d is even or odd, to compute a^d. This approach allows you to compute $a^d \bmod n$ using $O(\log d)$ multiplications. To solve the second problem, you should note that you can perform $\bmod n$ as soon as you get intermediate answers rather than wait to compute $\bmod n$ at the very end. This is because of the following property of $\bmod$: $(a \cdot b \cdot c) \bmod n = ((a \cdot b) \bmod n) \bmod n$.

(a) Print and submit your code. No matter what programming language you use, I expect that your function will be no more than 10 lines long. Make sure your code is well documented.

(b) Let n be the following 100-digit number.

```
2908511952812557872434704820397229928450530253990 1
5899055073199101184657163562102578687988156181498 9
```

Use the function `bigPowerMod` to compute $(n-100)^{n-1} \bmod n$. Report the answer you get.

(c) Provide an argument for why the running time of your algorithm/code is $O(\log^3 n)$.

2. The algorithm for the *Closest Point Pair* problem (that we discussed in class) is careful to ensure that it does $O(n)$ work outside of the recursive calls. In this problem, I want to investigate the consequences of not being this careful, on the running time of algorithm for his problem. Specifically, suppose that instead of sorting the points initially (outside the recursion), we sort the points as needed (by x-coordinate and by y-coordinate) inside the recursive calls.

   (a) Write down the recurrence relation that characterizes the running time of this new algorithm.

   (b) Solve this recurrence to obtain the running time of this new algorithm.

3. Like `mergeSort`, `quickSort` is a sorting algorithm based on the divide-and-conquer paradigm. As we have seen, the worst case running time of `mergeSort` is $\Theta(n \log n)$, as was shown by solving the `mergeSort` recurrence relation. The situation with the running time of `quickSort` is a bit murkier, and more interesting. Here is Python code for `quickSort` that I wrote a few years ago:

```
def partition(L, first, last):
    # Pick L[first] as the "pivot" around which we partition the list
    p = first

    for current in range(p+1, last+1):
        if L[current] < L[p]:
            swap(L, current, p+1)
            swap(L, p, p+1)
            p = p + 1

    return p

def generalQuickSort(L, first, last):
    # Base case: if first == last, there is nothing to do

    # Recursive case: 2 or more elements in the slice L[first..last]
    if first < last:
        # Divide step
        p = partition(L, first, last)

        # Conquer step
        generalQuickSort(L, first, p-1)
        generalQuickSort(L, p+1, last)

        # Combine step: there is nothing left to do!
```

   Suppose that L is an already-sorted list (in increasing order) of length n. This problem asks you to figure out the worst case running time of the function call `generalQuickSort(L, 0,` $n-1$`)`. Start by writing a recurrence relation and then solve it.
   Hint: The complicatio with `quickSort` is that the choice of the "pivot" in the `partition` function affects the sizes of the two subproblems that are solved by the recursive calls. This in turn affects the overall running time quite significantly.

4. I want you to now remember the following partitioning problem we considered a while ago. You are given a list L of length n and asked to partition the elements of L into two sublists

L_1 and L_2 such that (i) $n/3 \leq |L_1|, |L_2| \leq 2n/3$ and (ii) all elements in L_1 are less than or equal to all elements in L_2.

We designed a simple, randomized (Las Vegas) algorithm for this problem that ran in $O(n)$ expected time. Let us replace the call to `partition` in the code for `quickSort` given above by a call this Las Vegas algorithm. After we obtain a partition (L_1, L_2) by calling this Las Vegas algorithm, we can simply call `quickSort` on L_1 and then on L_2. Now we have a randomized (Las Vegas) version of `quickSort`. I would like to analyze the expected running time of this algorithm.

(a) Write down a recurrence relation for the expected running time of this randomized version of `quickSort`.

(b) Solve this recurrence to obtain an upper bound on the expected running time of this randomized version of `quickSort`.

5. In one of the earlier homeworks, you considered a primality-testing algorithm running in $O(\sqrt{n})$ time. Since the input size for the problem is $\Theta(\log n)$, this algorithm (whose running time is exponential in the input size) is extremely inefficient. On an input with 100s of digits this algorithm could, in the worst case, take more time than the entire lifetime of the universe! This problem will introduce you to a method for designing extremely fast primality testing algorithms.

   One of the oldest *fast* primality testing algorithm, called the *Miller-Rabin Algorithm* is a randomized, Monte Carlo algorithm dating back to the late 70s. If implemented correctly, the Miller-Rabin Algorithm can easily test integers with 100s of digits, for primality. The Miller-Rabin algorithm is extremely fast, not just because it is randomized, but also because it is allowed to produce an incorrect answer! If the input is a prime, the Miller-Rabin algorithm will correctly figure this out; however, if the input is a composite, then the algorithm may, with a very tiny probability, make an error and report the number as a prime.

   In this homework, you are asked to implement a randomized primality testing algorithm that is simpler than the Miller-Rabin Algorithm and is called the *Fermat's Little Theorem Primality Test (FLTP Test)*. The well-known encryption program PGP uses the FLTP Test in its algorithms. As the name suggests, the FLTP Test depends on *Fermat's Little Theorem*. This is an old mathematical result, first stated by Pierre de Fermat in 1640 and it says this:

   > If p is a prime then for all integers a, $1 \leq a < p$, a^{p-1} mod p equals 1.

   In other words, if p is a prime then you can pick any integer a between 1 and $p-1$ (inclusive of 1 and $p-1$) and compute a^{p-1}, divide this by p and the remainder will be 1.

   Example. Suppose $p = 7$. Then Fermat's Little Theorem is saying that for $a = 1, 2, \ldots, 6$, a^6 mod 7 equals 1. This is easy to check.

$1^6 = 1$	$1 \bmod 7 = 1$
$2^6 = 64$	$64 \bmod 7 = 1$
$3^6 = 729$	$729 \bmod 7 = 1$
$4^6 = 4096$	$4096 \bmod 7 = 1$
$5^6 = 15625$	$15625 \bmod 7 = 1$
$6^6 = 46656$	$46656 \bmod 7 = 1$

   Fermat's Little Theorem suggests the following simple algorithm for primality testing:

   > Given an integer $n > 1$, compute a^{n-1} mod n for each $a = 1, 2, \ldots n-1$. If for any of the a's that were considered, a^{n-1} mod $n \neq 1$ then output `composite`; otherwise output `prime`.

While this algorithm is correct, it is not any faster than the naive primality testing algorithms we have already implemented. Notice that the above algorithm simply runs through all a's between 1 and $n-1$.

To speed up up this algorithm we use a different mathematical fact. Before we can state this fact, we need to define two pieces of terminology.

(i) For a composite n, we call an integer a, $1 \leq a < n$, a *Fermat witness* if $a^{n-1} \bmod n \neq 1$. Thus a Fermat witness is a "witness" to the compositeness of n.

(ii) A positive integer n is a *Carmichael number* if $a^{n-1} \bmod n = 1$ for all a's in the range $[1, n-1]$ *that are relatively prime to* n. Recall that two numbers are relatively prime if they have no common factors other than 1. For example, 4 and 9 are relatively prime.

Example. The smallest Carmichael number is 561. 561 is a composite because $3 \times 11 \times 17 = 561$. 561 is a Carmichael number because for every a in the range $[1, 560]$ that is relatively prime to 561, $a^{560} \bmod 561 = 1$. So what are some values of a that are relatively prime to 561? 1, 2, 4, 5, and 7 are the first 5 integers in the range $[1, 560]$ that are relatively prime to 561. For each value of $a = 1, 2, 4, 5, 7$, it is the case that $a^{560} \bmod 561 = 1$. For values of a not relatively prime to 561, $a^{560} \bmod 561$ may or may not be equal to 1.

Now the mathematical fact that helps us speed up primality testing is this:

> Every composite integer n is either a Carmichael number or at least 1/2 of the integers in $[1, n-1]$ are Fermat witnesses.

Thus the above fact is telling us that with the exception of Carmichael numbers, every composite n has lots of Fermat witnesses – at least 50% of the numbers in the range $[1, n-1]$ are Fermat witnesses.

Example. Let $n = 6$. The following table shows values of $a^5 \bmod 6$ for $a = 1, 2, 3, 4, 5$.

$1^5 = 1$	$1 \bmod 6 = 1$
$2^5 = 32$	$32 \bmod 6 = 2$
$3^5 = 243$	$243 \bmod 6 = 3$
$4^5 = 1024$	$1024 \bmod 6 = 4$
$5^5 = 3125$	$3125 \bmod 6 = 5$

From the table it is clear that 6 has 4 Fermat witnesses - thus more than 50% of the 5 possible values of a are Fermat witnesses.

This means that if the input is a non-Carmichael composite n, then we can pick an integer a at *random* from the range $[1, n-1]$ and expect that a will be a Fermat witness for the compositeness of n with probability at least 1/2. Thus we would have an at least 50% chance of correctly identifying n as a composite, just by performing one test. To improve the chances of getting the test right, we could just repeat the random choice of a a few times. Suppose we repeat the above process 10 times, independently picking a at random (from the range $[1, n-1]$) each time, then the probability of not finding a Fermat witness all 10 times would be under $1/2^{10} = 1/1024$. Thus the probability of incorrectly declaring that a non-Carmichael composite is a prime is less than 1/1000, even if we repeat the test only 10 times. This leads to the following simple *randomized* algorithm for primality testing:

Input: a positive integer n
Algorithm: FLTP TEST
repeat 10 times
  pick an integer a at random from $[1, n-1]$
  if $a^{n-1} \bmod n \neq 1$, output `composite` and exit the program.
output `prime`

This leaves just the vexing issue of Carmichael numbers. These may have only few Fermat witnesses and so the above algorithm may declare them as primes (even though they are composite) with a fairly high probability. For example, 561 has 240 Fermat witnesses, somewhat less than half of 560. In fact, this is the main reason that the Miller-Rabin Algorithm and not the FLTP test is used for primality testing in general. However, Carmichael numbers are not that common. The smallest Carmichael number is 561 and these numbers get rarer as we start considering larger numbers.

(a) Your task is to implement the FLTP TEST. Your program should prompt the user for a positive integer, larger than 1 and then output a message indicating the primality of the input. Notice that FLTP TEST involves computing $a^{n-1} \bmod n$, where n could be quite large. Of course, you should use the function `bigPowerMod` you implemented for Problem 1 in this homework.

Use your implementation to test the primality of the following numbers:

- 599181055463339651776702496758089432115 3
- 198222712543662401291126962480559035452916883102 93
- 271751460953412243574650375322181330929301452213 79
- 4702877858580764415667235078667510929270158248348819067635 07
- 693711969678975263512873427191894879124339838606362751311911118403883

(b) As you know from the above discussion, the FLTP TEST can incorrectly classify composites as primes. Write a program that runs the FLTP TEST on all integers in the range $[500, 100000]$ and reports all integers in this range that are *incorrectly* classified as primes.

To complete this task, your program would have to be able to correctly identify primes/composites and the easiest way to do this is to simply use a naive primality testing algorithm. Examine the output of your program and compare the output with the list of Carmichael numbers less than 100000 (there are not that many Carmichael numbers under 10,000). Are you seeing any non-Carmichael composites classified as primes? In general, what would be a simple way of improving the accuracy of the program with respect to non-Carmichael composites?
