CS5800 Algorithms *Out: September 13, 2019*

Solutions for Problem Set 1

Name: Instructor's solutions *Due: September 20, 2019*

Problem 1 (Proof by Contradiction) 10 points

P is a given program which takes an input string and always returns 1. (That is, program P computes the function $f(n) = 1$, for every $n \in \mathbb{N} = \{1, 2, ...\}$.) Prove that there is no equality checking program which when given another program Q on input, can conclusively determine whether $P = Q$ (i.e. in terms of behavior Q computes the same function as P).

Solution:

Assume there is a checking program CHECK, which consumes an input Q, and returns true if P = Q. Now we suppose we have a program Q as follows[1]:

Algorithm 1 Q

```
1: procedure Q
2:     if $CHECK(Q)$ then return -1
3:     else return $P(n)$
4:     end if
5: end procedure
```

We note the following possibilities:

1. If CHECK(Q) is true, it means $P = Q$, and $Q(n)$ should return 1. However, it returns -1, which means $Q \neq P$ although CHECK thinks $Q = P$.
2. If CHECK(Q) is false, it means $Q \neq P$, and the output of Q should not equal $P(n)$. However, it returns $P(n)$, which means $Q = P$.

In either case, we will have a contradiction. Q both equals and doesn't equal P and whatever answer CHECK returns is wrong. Thus we proved that there exists at least one program Q, in which no such $CHECK$ program can work. Therefore, there can be no equality checking program that conclusively determines whether $P = Q$. ■

Problem 2 (Proof by Induction) 20 points

1. For all $n \geq 1$ prove that the following summation formula is correct:

$$\frac{1}{1 \times 3} + \frac{1}{3 \times 5} + \frac{1}{5 \times 7} + \cdots + \frac{1}{(2n-1)(2n+1)} = \frac{n}{2n+1}$$

[1]Note that although the algorithms body is in terms of itself by a call to CHECK(Q), it is known that it is possible for a Turing machine to access a string description of itself

2. Prove that $2^n + 1$ is divisible by 3 for all odd natural numbers n.
3. Let n and k be non-negative integers with $n \geq k$. Prove $\sum_{i=k}^{n} \binom{i}{k} = \binom{n+1}{k+1}$.
4. Let $f_0 = 0, f_1 = f_2 = 1$ and define $f_n = f_{n-1} + f_{n-2}$ for all $n \geq 1$. These are called the Fibonacci numbers. Prove that for all $n \geq 1$, the following holds:

$$f_{n+1} < \left(\frac{7}{4}\right)^n$$

Solution:

1. Base case: $k = 1 \implies \frac{1}{1\times 3} = \frac{k}{2k+1}$.

   Inductive Hypothesis: Assume that the statement is true until k, so

$$\frac{1}{1\times 3} + \frac{1}{3\times 5} + \frac{1}{5\times 7} + \cdots + \frac{1}{(2k-1)(2n+1)} = \frac{k}{2k+1}$$

   Inductive Step: We want to show that

$$\frac{1}{1\times 3} + \frac{1}{3\times 5} + \frac{1}{5\times 7} + \cdots + \frac{1}{(2k-1)(2n+1)} + \frac{1}{(2k+1)(2k+3)} = \frac{k+1}{2k+3}$$

   Note that by the inductive hypothesis,

$$\frac{1}{1\times 3} + \frac{1}{3\times 5} + \frac{1}{5\times 7} + \cdots + \frac{1}{(2k-1)(2n+1)} + \frac{1}{(2k+1)(2k+3)} = \frac{k}{2k+1} + \frac{1}{(2k+1)(2k+3)}$$

   Doing some algebra, we get the result we wanted to show so we are done

$$\frac{k(2k+3)}{(2k+1)(2k+3)} + \frac{1}{(2k+1)(2k+3)} = \frac{(k+1)(2k+1)}{(2k+1)(2k+3)} = \frac{k+1}{2k+3}$$

2. Base case: $k = 1 \implies 2^1 + 1 = 3$ is divisible by 3.

   Inductive Hypothesis: Assume that $k = 2l + 1$ be an odd number and that $2^k + 1 = 2^{2l+1} + 1$ be divisible by 3. In other words, $2^{2l+1} + 1 = 3m$ for some m.

   Inductive Step: For the next odd number $k + 2 = 2l + 3$, we have:

$$2^{2l+3} + 1 = 4 \cdot 2^{2l+1} + 1 = 4 \cdot 2^{2l+1} + 4 - 3 = 4(\cdot 2^{2l+1} + 1) - 3 = 4(3m) - 3 = 3(4m-1)$$

   Hence $2^{k+2} + 1$ is also divisible by 3 so we are done.

   Note: In one step above, we added and subtracted the same quantity (which doesn't change anything) to help us get things to the form that we want (multiple of 3). This is a general strategy that is very useful. You can add/subtract, multiply/divide, take log/exponent (this comes in handy in the asymptotics section), etc. so that you can change the form of the equation to be more favorable without changing the actual value.

3. Base Case: When $n = k$, $\sum_{i=k}^{n} \binom{i}{k} = \binom{n}{n} = 1 = \binom{n+1}{n+1}$.

   IMPORTANT: Note that our base case is $n = k$, for all integers, not just $n = k = 0$. You can think of this as doing one induction proof for each $k \in \{0, 1, 2, \dots\}$, and the base case being $n = k$. We do this because we have two parameters here. We could set the base case to $n = k = 0$ but then we'd have to induct on both n, k. By doing a base case for all (k, k) pairs instead of just the $(0, 0)$ pair we can induct just on n, since to "reach" an arbitrary (n, k) pair that we wish to prove this formula for, we start at the point (k, k) which is valid by the base case, and then we can increase n however we need using the inductive step. In a sense this is strong induction, but with an infinitely large base case. But it's valid because we can show it's true for every case.

   Inductive Hypothesis: Suppose that $\sum_{i=k}^{n} \binom{i}{k} = \binom{n+1}{k+1}$ holds for some $n \geq k$.

   Inductive Step: Then for $n + 1$ we want to show that

$$\sum_{i=k}^{n+1} \binom{i}{k} = \binom{n+2}{k+1}$$

   By the inductive step

$$\sum_{i=k}^{n+1} \binom{i}{k} = \sum_{i=k}^{n} \binom{i}{k} + \binom{n+1}{k} = \binom{n+1}{k+1} + \binom{n+1}{k}$$

   By Pascal's Identity (Recitation 2, Problem 4 part 2) this is equal to

$$\binom{n+2}{k+1}$$

   So we are done.

4. Base Case: $f_1 = f_2 = 1 < \frac{7}{4}$. Note, we need to show the base case for the first two terms, because we need to assume the formula is true for the two smallers terms f_k, f_{k-1} in the induction to prove it for f_{k+1}. (This is similar to Recitation 2, Problem 3 part 4.)

   Inductive Hypothesis: Now assume that $f_k < \left(\frac{7}{4}\right)^{k-1}$ and that $f_{k-1} < \left(\frac{7}{4}\right)^{k-2}$
   Inductive Step: For $k + 1$ we want to show that

$$f_{k+1} < \left(\frac{7}{4}\right)^{k}$$

   By the inductive hypothesis,

$$f_{k+1} = f_k + f_{k-1} < \left(\frac{7}{4}\right)^{k-1} + \left(\frac{7}{4}\right)^{k-2} < \left(\frac{7}{4}\right)^{k-2} \left(\frac{7}{4} + 1\right) < \left(\frac{7}{4}\right)^{k-2} \left(\frac{7}{4}\right)^{2} < \left(\frac{7}{4}\right)^{k}$$

   By principle of Induction we've proven the required statement.

■

Problem 3 (Asymptotics) 20 points

Arrange the following functions in ascending order of growth rate. Also give reasoning for the order.

$$g_{01}(n) = n^{\frac{101}{100}} \qquad g_{02}(n) = n2^{n+1}$$
$$g_{03}(n) = n(\log n)^3 \qquad g_{04}(n) = n^{\log\log n}$$
$$g_{05}(n) = \log(n^{2n}) \qquad g_{06}(n) = n!$$
$$g_{07}(n) = 2^{\sqrt{\log n}} \qquad g_{08}(n) = 2^{2^{n+1}}$$
$$g_{09}(n) = \log(n!) \qquad g_{10}(n) = \lceil \log(n) \rceil!$$
$$g_{11}(n) = 2^{\log\sqrt{n}} \qquad g_{12}(n) = \sqrt{2}^{\log n}$$

Solution:

$g_{07} \ll g_{11} \equiv g_{12} \ll g_{09} \equiv g_{05} \ll g_{03} \ll g_{01} \ll g_{10} \ll g_{04} \ll g_{02} \ll g_{06} \ll g_{08}$

You'll need a few ideas to simplify the functions into forms that you can easily compare. You can use the fact that $1 \ll \log^a n \ll n^b \ll c^n \ll n^n$. It is often useful to rewrite n as $a^{\log_a n}$ so that the base in an exponentiation is some constant that we like and not a function of n. It is very useful to rewrite factorials using Stirling's approximation $n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$.

Using the above we can simplify the functions to make them easier to compare.

$$g_{07} = 2^{\sqrt{\log n}} = \left(2^{\sqrt{\log n}}\right)^{\frac{\log n}{\log n}} = (n)^{\frac{1}{\sqrt{\log n}}}$$

$$g_{11} = g_{12} = \sqrt{n} = n^{1/2}$$

$$g_{09} = n \log n \qquad \text{(Stirling's approximation)}$$

$$g_{05} = 2n\log(n)$$

$$g_{03} = n\log(n) \cdot \log^2(n)$$

$$g_{01} = n \cdot n^{\frac{1}{100}}$$

Let $x = \log n$, then $x \le \lceil x \rceil \le x + 1$ and,

$$g_{10} = \lceil \log(n) \rceil! \approx \sqrt{x+1}\left(\frac{x+1}{e}\right)^{x+1} = \sqrt{x+1}\left(\frac{2^{\log(x+1)}}{2^{\log e}}\right)^{x+1} = 2^{(x+1)\cdot(\log(\frac{x+1}{e})) + \frac{\log(x+1)}{2}}$$

$$g_{04} = n^{\log\log n} = 2^{\log n \cdot \log\log n}$$

$$g_{02} = n2^{n+1} \qquad \text{(Exponent has polynomial compared to logarithm)}$$

$$g_{06} = n! = \left(\frac{n}{e}\right)^n = \frac{e^{n\ln n}}{e^n} = e^{n\ln n - n}$$

$$g_{08} = 2^{2^{n+1}} \qquad \text{(The double exponent grows exponentially faster than a single exponent)}$$

Comparing g_{10} and g_{04} is the hardest comparison to make even after bringing them to similar forms. If you compare the two exponents (by taking their difference, not ratio) then you'll see that the difference of exponent of g_{04} and that of g_{10} is always positive and growing for large enough value of $x = \log n$. Thus, g_{04} grows faster. ■

Problem 4 (More Asymptotics) 20 points

For a given two functions f, g and h. Decide whether each of the following statements are correct and give a proof for each part.

1. If $f(n) = \Omega(g(n))$ and $g(n) = \Omega(f(n))$, then $f(n) = \Theta(g(n))$
2. If $f(n) = o(g(n))$, then $g(n) \notin O(f(n))$
3. If $f(n) = O(h(n))$ and $g(n) = O(h(n))$, then $f(n) + g(n) = O(h(n))$
4. If $f(n) = O(h(n))$ and $g(n) = O(h(n))$, then $f(n) \cdot g(n) = O(h(n))$
5. If $f(n) = O(g(n))$, then $2^{f(n)} = O(2^{g(n)})$

Solution:

1. **True.**

   $f(n) = \Omega(g(n)) \implies \exists n_1$ and $c_1 > 0$, such that $f(n) \geq c_1 g(n)$ when $n \geq n_1$ and $g(n) = \Omega(f(n)) \implies \exists n_2$ and $c_2 > 0$, such that $g(n) \geq c_2 f(n)$ is true when $n \geq n_2$. There exist constants $\exists c_3 = c_1 > 0, c_4 = \frac{1}{c_2} > 0$, such that when $n \geq max(n_1, n_2)$ the following holds $c_1 g(n) \leq f(n) \leq \frac{1}{c_2} g(n)$. By definition, $f(n) = \Theta(g(n))$.

2. **True.**

   Proof by contradiction. Suppose that $g(n) \in O(f(n))$. Then $\exists c_1, n_1 > 0$ such that $g(n) < c_1 f(n) \implies f(n) > \frac{1}{c_1} g(n)$ when $n > n_1$. However, by definition of $f(n) = o(g(n))$, we know that $\forall c, \exists n_c > 0$ such that $f(n) < cg(n)$ when $n > n_c$. So when $c = \frac{1}{c_1}$ and $n > \max(n_1, n_c)$ we have both $f(n) > \frac{1}{c_1} g(n)$ and $f(n) < \frac{1}{c_1} g(n)$, a contradiction. Hence our assumption that $g(n) \in O(f(n))$ must be wrong.

3. **True.** By definition, $f(n) = O(h(n)) \implies \exists n_1, c_1 > 0$, such that $f(n) \leq c_1 h(n)$ when $n \geq n_1$ and $g(n) = O(h(n)) \implies \exists n_2, c_2 > 0$, such that $g(n) \leq c_2 h(n)$ when $n \geq n_2$. Note that $f(n) + g(n) \leq c_1 h(n) + c_2 h(n) = (c_1 + c_2) h(n)$ would be true when $n \geq max(n_1, n_2)$. Therefore $\exists n_3 = max(n_1, n_2), c_3 = (c_1 + c_2) > 0$ such that $f(n) + g(n) \leq c_3 h(n)$ when $n \geq n_3$. Hence by definition, $f(n) + g(n) = O(h(n))$.

4. **False.**

   Consider $f(n) = g(n) = h(n) = n$. Then the conditions hold. But $f(n)g(n) = n^2 \notin O(h(n))$. In fact, in this case we have that $f(n)g(n) = \omega(h^2(n))$

5. **False.**

   Consider $f(n) = 2n, g(n) = n$, then it holds that $f(n) = O(g(n))$, but $2^{f(n)} = 2^{2n} \notin O(2^{g(n)}) = O(2^n)$. In fact, in this case we have that $2^{g(n)} = o(2^{f(n)})$.

■

Problem 5 (Programming: Modular exponentiation) 30 points

See Piazza for a post containing a link to the HackerRank page, where you will be submitting the assignment. In addition to the below description, it also contains more formal requirements for how your program should behave.
Description: Implement modular exponentiation in a way that outputs the intermediary steps of the algorithm.
Problem Statement: **IMPORTANT** You are NOT allowed to use built in language functions which trivialize the task of computing exponents. Any submissions which use this and avoid the task at hand will be given a 0.

In this problem, you will have to efficiently implement modular exponentiation. Recall that the problem of modular exponentiation is, given positive integers a and n, and a non-negative integer x, calculate $a^x \mod n$.

One way of doing this is exponentiation by squaring. It involves repeatedly squaring the base a and reducing it $\mod n$. Doing so yields the values a, $a^2 \mod n$, $a^4 \mod n$, $a^8 \mod n$, ... etc. By combining these in the correct way, and using the fact that every number has a binary representation, we can compute $a^x \mod n$ in time $O(\log x)$. Implement modular exponentiation by squaring, and output the intermediary values of $a^{2^i} \mod n$, as well as the final value $a^x \mod n$.

Note that in addition to the above, the challenge page also describes the input/output format, and gives a few examples.

Solution:

```
#include <iostream>
#include <string>
using namespace std;

string bin(unsigned long long i){
  if(i == 0)
    return "";
  if(i % 2 == 1)
    return bin(i/2) + "1";
  return bin(i/2) + "0";
}

typedef unsigned long long ull;

ull add_mod(ull a, ull b, ull n){
  a = a % n;
  b = b % n;
  if (a + b <= a){
    a = a - (n - b);
  }
  else
    a = (a+b) % n;
  return a;

}

unsigned long long correct_square(unsigned long long a, unsigned long long n){
  unsigned long long b, sum;
  sum = 0;
```

```cpp
  a = a % n;

  for(b = a; b != 0; b >>= 1){
    if(b & 1) {
      sum = add_mod(sum, a, n);
    }
    a = add_mod(a, a, n);
  }
  return sum;
}

unsigned long long multiply_mod(unsigned long long a, unsigned long long b, unsigned long
    long n){
  unsigned long long i, sum;
  sum = 0;
  a = a % n;
  b = b % n;

  for(i = b; i != 0; i >>= 1){
    if(i & 1) {
      sum = add_mod(sum, a, n);
    }
    a = add_mod(a, a, n);
  }
  return sum;
}

int main(){
  unsigned long long a, x, n;
  cin >> a >> x >> n;
  string bin_rep = bin(x);
  cout << bin_rep << endl;

  int d = bin_rep.size();
  a = a % n;
  unsigned long long result;
  result = 1;
  for(int i = d-1; i >= 0; i--){
    cout << a << endl;
    if(bin_rep[i] == '1')
      result = multiply_mod(result, a, n) % n;
    a = multiply_mod(a, a, n);
  }
  cout << result << endl;

}
```

Listing 1: C/C++ solution

```java
import java.math.BigInteger;
import java.util.Scanner;
public class SolutionExponentiation {

  public static void main(String[] args) {
    BigInteger a, x, n, final_ans = BigInteger.ONE;
    Scanner sc = new Scanner(System.in);
    a = new BigInteger(sc.next());
    x = new BigInteger(sc.next());
    n = new BigInteger(sc.next());
    String binary_representation = x.toString(2);
    System.out.println(binary_representation);

    int d = binary_representation.length();
    a = a.remainder(n);
    for (int i = d - 1; i >= 0; i--) {
      System.out.println(a);
      if (binary_representation.charAt(i) == '1') {
        final_ans = final_ans.multiply(a).remainder(n);
      }
      a = correct_square(a, n);
    }
    System.out.println(final_ans);
  }

  static BigInteger add_mod(BigInteger a, BigInteger b, BigInteger n) {
    a = a.remainder(n);
    b = b.remainder(n);
    if (a.add(b).compareTo(a) <= 0) {
      a = a.subtract(n.subtract(b));
    } else {
      a = a.add(b).remainder(n);
    }
    return a;
  }

  static BigInteger correct_square(BigInteger a, BigInteger n) {
    BigInteger b, sum = BigInteger.ZERO;
    a = a.remainder(n);
    for (b = a; b.compareTo(BigInteger.ZERO) != 0; b = b.shiftRight(1)) {
      if (b.and(BigInteger.ONE).compareTo(BigInteger.ONE) == 0) {
        sum = add_mod(sum, a, n);
      }
      a = add_mod(a, a, n);
    }
    return sum;
  }
}
```

Listing 2: Java solution

```python
a, x, n = map(int, raw_input().split())

bin_rep = bin(x)[2:]
print bin_rep
answer = 1
d = len(bin_rep)
curr_power = a % n
for i in range(d):
    print curr_power
    if bin_rep[::-1][i] == "1":
        answer = (answer * curr_power) % n
    curr_power = (curr_power**2) % n
print answer
```

Listing 3: Python solution

■