Homework 2: Loop Invariants, Modular Arithmetic, and Primality

Instructions: **Due 01/24/25 11:59pm**

- Please type your solutions using LaTex or any other software. Handwritten solutions will not be accepted.
- Please try to write concise responses.
- You should not use pseudocode to describe your algorithms.
- Unless otherwise stated, saying log means base 2

Q1 Let p be the polynomial defined by $p(x) = a_0 + a_1x + a_2x^2 + \cdots + a_nx^n$. Suppose we want to evaluate $p(x)$ for a given real number x

Algorithm: Polynomial Evaluation

EvaluatePolynomial$(a_0, a_1, \ldots, a_n, x)$**:**

```
z := a_n
i := n
while i > 0 do
    i := i − 1
    z := z · x + a_i
return z
```

(a) Briefly explain why the algorithm works (without using loop invariants).

We have $z = zx + a_i$. We also have z initialized as a_n. So we see that after the algorithm is over we have something like,

$$x(\ldots x(x(xa_n + a_{n-1}) + a_{n-2}) + \ldots) + a_0$$

We see that for every n term we multiply the term with x, n times giving us the term a_nx^n in our final result. That is, for any given $i = n$ for all loop iterations $0 < i < n$. We're multiplying a_i with x, n times as we're multiplying z with x in each of those iterations.

(b) Show that $z = \sum_{j=i}^{n} a_j x^{j-i}$ is an invariant that the algorithm maintains.

First we check if the invariance is maintained before the loop begins. So we have,

$$z = a_n = a_n x^0 = \sum_{j=n}^{n} a_j x^{j-i}$$

Hence the invariance is maintained before the loop begins. Now we show by induction that if the invariance holds for any arbitrary $i = k$ then it must hold for $i = k - 1$.

So assume it holds for $i = k$ so we have,

$$z = \sum_{j=k}^{n} a_j x^{j-k}$$

Now in the next iteration we have,

$$\begin{aligned} z &= zx + a_i \\ &= x \sum_{j=k}^{n} a_j x^{j-k} + a_i \\ &= \sum_{j=k}^{n} a_j x^{j-k+1} + a_i \\ &= \sum_{j=k}^{n} a_j x^{j-(k-1)} + a_i \\ &= \sum_{j=k-1}^{n} a_j x^{j-(k-1)} \end{aligned}$$

Which is the invariant for when $i = k - 1$.

Hence by induction as we established the base case for when $i = n$ we can conclude that it is true for all $0 \leq i < n$

So our invariance is also true after the loop for when $i = 0$

Q2 Consider the following *iterative* algorithm to compute $X^Y \bmod N$.

Algorithm: Modular Exponentiation

ModExp(X, Y, N)**:**

```
x, y, z := X, Y, 1
while y ≠ 0 do
    if y is even then
        x = x^2 mod N
        y = ⌊y/2⌋
    if y is odd then
        z = x · z mod N
        y = y − 1
return z
```

(a) Identify a loop invariant for your algorithm and show that the algorithm maintains the invariant.

Hint: You must show the invariant is maintained before, during, and after the loop.

The loop invariance for this algorithm is $X^Y \bmod N = z \cdot x^y \bmod N$
Consider before the loop, we have $x, y, z = X, Y, 1$. So,

$$z \cdot x^y \bmod N = X^Y \bmod N$$

So our invariance is maintained.

Now consider within the loop. If y is even we see that z does not change. But we have,

$$x = x^2 \bmod N \text{ and } y = y/2$$

After the update we have,

$$(x^2 \bmod N)^{y/2} \bmod N$$

We know because of the mod rules that this is equal to,

$$(x^2 \bmod N)^{y/2} \bmod N = x^y \bmod N$$

This means our invariance is maintained.

Now if y is odd we have,

$$z = x \cdot z \bmod N \text{ and } y = y - 1$$

So before update we have,

$$X^Y \bmod N = z \cdot x^y \bmod N$$

We can rewrite $x^y = xx^{y-1}$ such that,

$$\begin{aligned} X^Y \bmod N &= z \cdot xx^{y-1} \bmod N \\ &= (z \cdot x \bmod N)(x^{y-1} \bmod N) \end{aligned}$$

We see that our invariance holds after the update. So we've shown that in either case within the loop our in variance holds.

Now consider after the loop. We have $y = 0$. So,

$$X^Y \bmod N = z \cdot x^0 \bmod N = z$$

So z has the final solution which is what we want hence its maintained.

(b) Let X, Y, and N, be n-bit integers. What is the time and space complexity of this algorithm with respect to n?

First we see that we divide y by 2 whenever its even so its complexity would be $O(\log(Y))$. However we know that Y has n bits such that $\log(Y) = n$ so we have $O(n)$. Now we see that within each iteration of the loop we're performing multiplication and/or division. We know that these would have a complexity of $O(n \log n)$ if we use the optimized approach. Hence the total complexity with respect to the number of bits n would be $O(n^2 \log(n))$ however if we use the direct approach we know the complexity would be $O(n^2)$ for multiplication and div ion. Hence it would be $O(n^3)$

The algorithm uses x, y, z, N to store the information. We see that N is constant but x, y, z are updated in each iteration. In every iteration we do not create new instances of these variables but we see we use the previous value to update the current value of these variables. So no new space is created because of a new iteration. Because of this the space complexity would be dpeendent only on the size of the variables themselves. We know that this these integers are n-bit integers. Hence as we only store a constant number of n bit integers we have a space complexity of $O(n)$

Questions (3), (4), and (5) require you to read Chapter 1.3 of the textbook.

Q3 Prove *Fermat's Little Theorem.* If p is prime, then for every $1 \leq a < p$,

$$a^{p-1} \equiv 1 \pmod{p}$$

Hint: The proof is in the book, we just need you to restate it in your own words.

First consider an arbitrary prime number p. Now let S be the set of the natural numbers smaller than p as follows,

$$S = \{1, 2, \ldots, p-1\}$$

Now consider an arbitrary $a \in S$ and indices $i, j \in S$. We show that for any distinct choice of $i, j \in S, a \cdot i \bmod p$ is distinct from $a \cdot j \bmod p$. Essentially that for a given a the function $f(i) = a \cdot i \bmod p$ is bijective. The bisecting implies that the product modulo p only reorders within the set S.

Let us assume the numbers are not distinct which means that,

$$a \cdot i \equiv a \cdot j \pmod{p}$$

However because a is coprime with p we can divide both sides by a to get,

$$i \equiv j \pmod{p}$$

$$i - j = kp \text{ for some k}$$

But we know that $i, j \in S$ which means that its smaller than p so the only choice of k is if its equal to 0. Hence we show that $i = j$ must be true then. This means that if $i \neq j$ then the function maps it to distinct elements in S.

So now we have,

$$S = \{1, 2, \ldots, p-1\} = \{a \cdot 1 \bmod p, \ldots, a \cdot (p-1) \bmod p\}$$

Multiplying the elements in each we have,

$$(p-1)! \equiv a^{p-1} \cdot (p-1)! \pmod{p}$$

However as p is prime all $p' \in S$ are not factors of p which means that $(p-1)!$ is relatively prime with p. Hence we can divide it from both sides to get,

$$a^{p-1} \equiv 1 \pmod{p}$$

Q4 Consider the following algorithms:

Algorithm: Trial Division

PrimalityOne(N)**:**

```
for i := 2, ..., √N do
    if N ≡ 0 (mod i) then
        return no
return yes
```

Algorithm: Randomized Primality Testing

PrimalityTwo(N)**:**

```
Pick positive integers a_1, a_2, ..., a_k < N at random
if a_i^{N-1} ≡ 1 (mod N) for all i = 1, 2, ..., k then
    return yes
else
    return no
```

(a) If we want to have a maximum probability of error $\delta \in (0, 1)$, how many positive integers k should we choose for running `PrimalityTwo`?

Assuming that N is not a carmichael number, for a given k the probability of error (the algo returns yes when N is composite) is smaller than $\frac{1}{2^k}$. So choose an arbitrary $\delta \in (0, 1)$. We need the error to be at most delta or,

$$\frac{1}{2^k} \leq \delta$$
$$\frac{1}{\delta} \leq 2^k$$
$$\log\left(\frac{1}{\delta}\right) \leq k$$

So for any $k \geq \log(\frac{1}{\delta})$ will be enough numbers such that the probability that our algorithm fails is less than δ

However if N is a carmichael number the probability of our random a not being coprime is very low for large N such that the probability of error is close to 1.

(b) In one sentence, what is an advantage of using `PrimalityOne` over `PrimalityTwo`?

PrimalaityOne is a deterministic algorithm and the probability of error is 0, however PrimalityTwo is a probabilistic algorithm whose probability of error is dependent on the number of a 's we choose.

(c) In one sentence, what is an advantage of using `PrimalityTwo` over `PrimalityOne`?

PrimailtyTwo is much faster than PrimailtyOne as it relies on tests done on a small random subset of numbers with the probability of error exponentially decreasing, where as PrimailtiyOne checks if every number less than equal to $\sqrt{N}$ is a factor or not which takes significantly longer.

Q5 Use `PrimalityTwo` to determine whether the following numbers are prime or composite with a confidence of **at least 99%**. Pick the first k numbers from the following list as a source of randomness: 89, 167, 242, 351, 386, 456, 503, 622, 682, 741 (where k is how much you need for 99% confidence).

You're allowed to use a calculator, and you can even write a program. Just show the result of each modulus calculation. If you decide to write a program, submit a screenshot of your code.

(a) 32767

To get at least 99% confidence we only require 7 random numbers as for $\delta = 0.01$,

$$\frac{1}{2^7} = \frac{1}{128} = 0.0078125 < 0.01 = \delta$$

```
❯ python3 ./PrimalityTest_fermats_little_theorem.py
Enter a number: 32767
a_0: 89
89^32766 mod 32767:  32740
False
```

(b) 743

To get at least 99% confidence we only require 7 random numbers as for $\delta = 0.01$,

$$\frac{1}{2^7} = \frac{1}{128} = 0.0078125 < 0.01 = \delta$$

```
❯ python3 ./PrimalityTest_fermats_little_theorem.py
Enter a number: 743
a_0: 89
89^742 mod 743:  1
a_1: 167
167^742 mod 743:  1
a_2: 242
242^742 mod 743:  1
a_3: 351
351^742 mod 743:  1
a_4: 386
386^742 mod 743:  1
a_5: 456
456^742 mod 743:  1
a_6: 503
503^742 mod 743:  1
True
```

Code

```python
p = int(input("Enter a number: "))
def PrimalityTwo(N):
    s = [89, 167, 242, 351, 386, 456, 503, 622, 682, 741]
    for a in range(0,7):
        num = s[a]
        print(f"a_{a}: {num}")

        pow = (num ** (N-1) )

        rem = pow % N
        print(f"{num}^{N-1} mod {N}: ", rem)
        if (rem != 1):
            return False
    return True
print(PrimalityTwo(p))
```