IE531: Algorithms for Data Analytics
Spring, 2023
Homework 1: Recursion
Due Date: February 3, 2023

Instructions

1. You will submit a PDF-version of your answers on Canvas on-or-before midnight of the due date.

Instructions

1. (*10 points*) (**Recursive GCD Computation**) The *greatest common divisor* (GCD) of two positive numbers m and n is the largest number k that divides m and n. That is, $\frac{m}{k}$ and $\frac{n}{k}$ leaves no remainder. Consider the Python code shown in figure 1. Using the fact that the GCD of two numbers does not change if the smaller number is subtracted from the larger number, show that the recursive function gcd(m, n) in the code shown in figure 1 implements the GCD computation.

   (**Note**: Please do not give me an illustrative example as a "proof" of that the code does what it is supposed to do. *I am looking for at least a semi-rigorous attempt at showing the correctness of the code for any pair of numbers*.)

```
def gcd(x , y):
    if y == 0:
        return x
    else:
        return gcd(y, x % y)
```

Figure 1: Function gcd(m, n) for Problem 1.

2. (*40 points*) (**Modular Exponentiation**) Suppose $x, n \in \mathcal{N}$, where n is very very large, we can use the recursive procedure of *Repeated Squaring* to compute x^n in $O(logn)$ time. Oftentimes, we need to compute

$$(x^n)_{mod\ m},$$

   where both the exponent $n \in \mathcal{N}$ and the modulus $m \in \mathcal{N}$ are very large[1].

   (a) (*10 points*) Show that

$$(x^n)_{mod\ m} = \left\{ \underbrace{(x_{mod\ m}) \times (x_{mod\ m}) \times \cdots \times (x_{mod\ m})}_{n \text{ times}} \right\}_{mod\ m} = \{(x_{mod\ m})^n\}_{mod\ m}$$

[1] In Public-Key Cryptography, for example.

(b) (*10 points*) Present a pseudo-code sample that uses *Repeated Squaring* to implement the observation in problem 2a. It might help to write some Python code that correctly implements this algorithm. This way you will make sure you have covered all the subtleties (and trust me, there are a few!) in your answer.

(c) (*10 points*) (**Chinese Remainder Theorem**): As a follow-on to problems 2a and 2b, let us suppose that the modulus m can be factored as $m = p_1 \times p_2 \times \cdots \times p_k$, where $\forall i, j, gcd(p_i, p_j) = 1$. That is, we have a pairwise coprime factorization of the modulus m. Show that the system of k-many equations

$$\begin{aligned}(x^n)_{mod\ p_1} &= a_1\\ (x^n)_{mod\ p_2} &= a_2\\ &\cdots\\ (x^n)_{mod\ p_k} &= a_k\end{aligned}$$

has a unique solution for $(x^n)_{mod\ m}$.

The next problem is about computing the *Prime Factorization* of any number, which together with the *Chinese Remainder Theorem* and the *Repeated Squaring Algorithm*, presents a very efficient algorithm to compute $(x^n)_{mod\ m}$ for large n and m.

(d) (*10 points*) (**Prime Factorization**) A Prime number is a positive integer, greater than 1, that cannot be divided by any other integer other than itself and 1. The Fundamental Theorem of Arithmetic says every number $n \in \mathcal{N}$ has a unique *Prime Factorization*. For example,

$$\begin{aligned}123456789 &= 3^2 \times 3607 \times 3803\\ 987654321 &= 3^2 \times 17^2 \times 379721,\end{aligned}$$

where the numbers $\{3, 17, 3607, 3803, 379721\}$ are all prime numbers. Consider the Python code shown in figure 2. Present a cogent argument that this code will generate the prime factorization of any number n.

```
In [1]: def primeFactors(n):
            c = 2
            while(n > 1):
                if(n % c == 0):
                    print(c, end=" ")
                    n = n / c
                else:
                    c = c + 1

In [2]: primeFactors(987654321)
        3 3 17 17 379721

In [3]: primeFactors(123456789)
        3 3 3607 3803
```

Figure 2: Python Code to compute the Prime Factors of a number.

In principle, we can compute the *Prime Factors* of any modulus m, and then compute the values of $a_1, a_2, \ldots, a_k$ using the Modular Exponentiation with Repeated Squaring. Following this, we can use the efficient algorithms (that you can discover yourself on the web) to solve the equations of the Chinese Remainder Theorem. All this will result in an extremely efficient procedure to compute $(x^n)_{mod\ m}$ for very large n and m.

3. (50 points) For the *Randomized Selection Algorithm* we picked a random member of the array as the *pivot* p (as opposed to p being the *Median-of-Medians*, when we did the *Deterministic Selection Algorithm*). Assume there is only one such p in the array – we have to compare with $(n-1)$ many others to split the original array into three arrays. This takes $c*(n-1)$ time. In the worst case, we will recurse on the longer of two arrays (i.e. the array that contains numbers smaller than p, and the array that contains numbers larger than p[2]. The average value of the number of elements in the longer array cannot be larger than $\frac{3n}{4}$. This means the average running time of the algorithm must satisfy the recursion

$$A(n) \leq c(n-1) + A\left(\frac{3n}{4}\right) \leq cn + A\left(\frac{3n}{4}\right) \Rightarrow A(n) \leq 4cn.$$

That is, the average running time of the *Randomized Selection Algorithm* must be linear in n.

To see the fact that the average value of the longer array is bounded by $\frac{3n}{4}$, we used an analogy of splitting a candy-bar of length L into two pieces by picking a uniformly distributed random variable in the interval $[0, L]$. It is not hard to see that the length of the longer-piece is uniformly distributed in the interval $[L/2, L]$, which means the average-value of the length of the longer-piece is $\frac{3L}{4}$.

In class we discussed the possibility of using multiple pivots, and improving the average running time of the Randomized Selection Algorithm. For example, if we had two (random) pivots p_1 and p_2. Using p_1, after $(n-1)$-many comparisons, we split the original array into three parts; this is followed by identifying the appropriate sub-array that contains the k-th smallest elements (that we wish to find). A similar process can be done with the other pivot p_2 – after another $(n-1)$-many comparisons – we identify yet another sub-array that contains the k-th smallest element (that we wish to find). The logical thing to do, would be to recurse on the smaller of these two sub-arrays that contains the k-th smallest element.

We would like to get an estimate of the average running time of the Randomized Selection Algorithm in the presence of m-many randomly-chosen pivots. We will do this in multiple steps in this homework.

   (a) (*25 points*) You have m-many candy-bars, each of them has a length L. Each candy bar is broken into two pieces using a uniformly distributed random variable in the interval $x \in [0, L]$ where the two pieces have a

[2]For the worst-case running-time we can glibly ignore the array that contains of elements in the original array that are equal to p , why?

length of x and $L - x$, respectively. We throw away the shorter of these two pieces and keep the longer one. In the end, we will have m-many longer-pieces of candy-bars. Show that the average value of the smallest of these m-many longer-pieces of candy-bars is

$$\frac{L(m+2)}{2(m+1)}$$

(b) (*25 Points*) Show that the version of the Randomized Selection Algorithm that uses m-many pivots to split the original array into m-many sub-arrays, which is then followed by a recursion on the array of smallest-size that contains the k-th smallest element will have an average running time

$$A_m(n) \leq \frac{2(m+1)}{m} \times c \times n.$$