CS 124 Homework 1: Spring 2021

Your name: Sean Ty

Collaborators: David Zhang, Brandon Tang, Rodrigo Chaname

No. of late days used on previous psets: 0
No. of late days used after including this pset: 0

Homework is due Wednesday at midnight ET. You are allowed up to **twelve** (college)/**forty** (extension school) late days through the semester, but the number of late days you take on each assignment must be a nonnegative integer at most **two** (college)/**four** (extension school).

Try to make your answers as clear and concise as possible; style will count in your grades. Be sure to read and know the collaboration policy in the course syllabus. Assignments must be submitted in pdf format on Gradescope. If you do assignments by hand, you will need to scan in your results to turn them in.

For all homework problems where you are asked to give an algorithm, you must prove the correctness of your algorithm and establish the best upper bound that you can give for the running time. Generally better running times will get better credit; generally exponential time algorithms (unless specifically asked for) will receive no or little credit. You should always write a clear informal description of your algorithm in English. You may also write pseudocode if you feel your informal explanation requires more precision and detail, but keep in mind pseudocode does NOT substitute for an explanation. Answers that consist solely of pseudocode will receive little or not credit. Again, try to make your answers clear and concise.

There is a (short) programming problem on this assignment; you DO NOT WORK with others on this problem (e.g., write code together) like you will for the "major" programming assignments. (You may talk about the problem, as you can for other problems.)

1. Suppose you are given a six-sided die that might be biased in an unknown way.

   (a) **(10 points)** Explain how to use rolls of that die to generate unbiased coin flips. Using your scheme, determine the expected number of die rolls until a coin flip is generated, in terms of the (unknown) probabilities p_1, p_2, ..., p_6 that the die roll is 1, 2, ..., 6.

   (b) **(10 points)** Now suppose you want to generate unbiased die rolls (from a six-sided die) given your potentially biased die. Explain how to do this, and again determine the expected number of biased die rolls until an unbiased die roll is generated.

   For both problems, you need not give the most efficient solution; however, your solution should be reasonable, and exceptional solutions will receive exceptional scores.

   (a) We first simulate a fair coin with a biased die. Suppose that the probability that face i shows up is p_i. We attempt to generate a bit as follows: roll the die twice, and let the numbers shown be a, b respectively. If $a < b$, we say that this corresponds to a heads. If $a > b$, we say that this corresponds to a tails. Otherwise $a = b$ and we retry. Now note that by symmetry, it follows that the probability of a heads equals the probability of a tails. In particular, conditioning on the event $a \neq b$, this is a fair coin flip. So this algorithm works.

   Now we find the expected number of rolls. Let this be E. Then by Adam's law E must satisfy the equation $E = (E+2)p + (1-p)\cdot 2$, where $p = \sum_{i=1}^{6} p_i^2$ is the probability of rolling a pair. Solving, this yields $E = \frac{2}{1-p} = \frac{1}{\sum_{1\leq i<j\leq 6} p_i p_j}$, where we used the fact that $1 = (p_1 + \cdots + p_6)^2$.

   (b) We now simulate a fair die. We use a similar idea as before: roll a die three times. If the rolls are not all different, we reroll. Otherwise, suppose that the rolls are $A < B < C$ and look at the possibilities in lexicographical order, forming a bijection between the permutations and the set $\{1, 2, \ldots 6\}$ (e.g. ABC is 1, ACB is 2, BAC is 3, etc.). Then again by symmetry, each number is equally likely to appear conditional on the rolls being pairwise distinct, so this works.

   For the expected number of rolls, let this be E. Then E must satisfy $E = 3\cdot(1-p) + (E+3)\cdot p$, where $p = \sum_{i=1}^{6} p_i^3 + 3\sum_{1\leq i<j\leq 6} p_i^2 p_j$ is the probability of having duplicates. Solving, this yields $E = \frac{3}{1-p}$.

2. On a platform of your choice, implement the three different methods for computing the Fibonacci numbers (recursive, iterative, and matrix) discussed in lecture. Use integer variables. (You do not need to submit your source code with your assignment.)

   (a) **(10 points)** How fast does each method appear to be? Give precise timings if possible. (This is deliberately open-ended; give what you feel is a reasonable answer. You will need to figure out how to time processes on the system you are using, if you do not already know.)

   First tried computing $n = 30$ (n is the index). Recursive took 283.864 ms, iterative took 0.013 ms, and matrix took 0.024 ms. For $n = 31$, recursive took 473.886 ms, iterative took 0.012 ms, and matrix took 0.022ms. For $n = 32$, recursive took 3066.943 ms, iterative took 0.015 ms, and matrix took 0.023 ms. From here, the recursive method is definitely the slowest by a large margin, and the run time looks exponential. For these small values of n, it looks like iterative is faster than the matrix method (about twice as fast).

   Now we compare iterative and matrix. From before, for small n iterative wins. For $n = 100$, we got a run time of 0.087 ms for iterative and a run time of 0.025 ms for matrix, so matrix now wins, which checks out with what we discussed in class. We suspect that for larger n, matrix method will be much faster, and that it was slower because it had a lot of constant time operations to do (e.g. multiplying the matrices, initialization of more variables, etc.) (code below just in case).

   (b) **(4 points)** What's the first Fibonacci number that's at least 2^{31}? (If you're using C longs, this is where you hit integer overflow.)

   We used Python, which does not have integer overflow. Instead, we used the iterative method and repeatedly check whether or not the most recent Fibonacci number is greater than 2^{31} (code below just in case), and we got a result of $n = 47$.

   (c) **(10 points)** Since you should reach "integer overflow" with the faster methods quite quickly, modify your programs so that they return the Fibonacci numbers modulo $65536 = 2^{16}$. (In other words, make all of your arithmetic modulo 2^{16}—this will avoid overflow! You must do this regardless of whether or not your system overflows.) For each method, what is the largest value of k such that you can compute the kth Fibonacci number (modulo 65536) in one minute of machine time?

   For the recursive method, trial and error showed that $n = 41$ was the largest that could be computed in machine time. For the iterative method, trial and error (guessing using big O, i.e. the fact that it was linear time) yields an upper bound of $n = 6 \cdot 10^8$. For the matrix method, even until $n = 10^{14}$ the process only took 0.157 ms, although for larger values of n (say $n > 1e15$) the output remained the same–perhaps due to an issue with python and the numbers growing too large. The maximum n that could be computed within a minute was around $1e250000$. The matrix method then is about $O(\log n)$.

   Code below:

```
## for part a

import time
```

```
n = 30
MOD = 65536

## define for fib
def recursiveFib(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    return (recursiveFib(n-1)+recursiveFib(n-2)) % MOD

start = time.time()
print(recursiveFib(n))
end = time.time()
print("recursive until n=30 took " + str(round(1000*(end-start),3)) + " ms")

def iterativeFib(n):
    if n == 0:
        return 0
    counter = 1
    prev = 0
    curr = 1
    while counter < n:
        temp = curr
        curr = (curr + prev) % MOD
        prev = temp
        counter += 1
    return curr

start = time.time()
print(iterativeFib(n))
end = time.time()
print("iterative until n=30 took " + str(round(1000*(end-start),3)) + " ms")

## define for matrix
def matrixFib(n):
    ## multiply two 2x2 matrices
    def product(a,b):
        x1 = (a[0]*b[0]+a[1]*b[2]) % MOD
        x2 = (a[0]*b[1]+a[1]*b[3]) % MOD
        x3 = (a[2]*b[0]+a[3]*b[2]) % MOD
        x4 = (a[2]*b[1]+a[3]*b[3]) % MOD

        return [x1,x2,x3,x4]
```

```
    # yada yada
    temp = n
    ## identity as result first
    res = [1, 0, 0, 1]
    ## store matrix of recursion
    matr = [0, 1, 1, 1]
    while temp > 0:
        if temp % 2 == 0:
            matr = product(matr, matr)
            temp = temp//2
        else:
            res = product(res, matr)
            matr = product(matr, matr)
            temp = temp//2

    return res[1]

start = time.time()
print(matrixFib(n))
end = time.time()
print("matrix until n=30 took " + str(round(1000*(end-start),3)) + " ms")

## first find first k such that fib(k) > 2^31
## for part b
a = 0
b = 1
k = 1
while b < 2**31:
    temp = b
    b += a
    a = temp
    k +=1
print(k)
# get 47

## now find 1 min limit

#%%
start = time.time()
print(matrixFib(10**100000))
end = time.time()
print("took " + str(round(end-start,3))+ " seconds")
```

3. (a) **(10 points)** Sort the following functions from asymptotically least to greatest: that is, make a series of a statements like "$f_3 = o(f_1)$, $f_1 = \Theta(f_4)$, $f_4 = o(f_5)$, $f_5 = o(f_9)$, ..., $f_8 = \Theta(f_7)$", where each function is either o or Θ of the next. All logs are base 2 unless otherwise specified.

i. $f_1 = (\log n)^{\log n}$
ii. $f_2 = n(\log\log n)^2/\log n$
iii. $f_3 = \log_3 n$
iv. $f_4 = (\log_4 n)^4$
v. $f_5 = \log_5(n^5)$
vi. $f_6 = 6^{\sqrt[6]{n}}$
vii. $f_7 = (7^n)/(n^{\log n})$
viii. $f_8 = n/(8\log n)$
ix. $f_9 = (9n)^{\log\log n}$

We list them in "ascending order." We first have $f_3 = \Theta(f_5)$, since $f_5(n) = 5\log_5(3)\log_3(n) = c \cdot f_3(n)$.

Next, we have $f_3 = o(f_4)$ since after accounting for the log constant term (use the fact that $\log_a(b) = \log(b)/\log(a)$) this is equivalent to $\log n = o((\log n)^4)$, which is true (we can see this more clearly by say, letting $x = \log(n)$, so it becomes $x = o(x^4)$).

Next, we have $f_4 = o(f_8)$. We can ignore the constant $\frac{1}{8}$ factor in f_8 and take logs. Then it suffices to prove that for any $M > 0$, we can find N large enough such that for all $n \geq N$ we have $\log n - \log\log n > 4\log\log n + M$, which is clearly true (choose $N > 2^{2^{2^M}}$ for example).

Next, we have $f_8 = o(f_2)$. We can again ignore constant factors and take logs. Then proving this is equivalent to proving that for any $M > 0$, there exists N such that for all $n \geq N$ we have $\log n + \log\log\log n - \log\log n > M + \log n - \log\log n$, which is true by say considering $N = 2^{2^{2^M}} + 69$.

Next, we have $f_2 = o(f_1)$. We can again take logs, so it suffices to prove that for any $M > 0$ there exists N such that for all $n \geq N$ we have $\log n \log\log n > M + \log n + \log\log\log n - \log\log n$, which is true since $\log\log n$ grows much faster than $\log\log\log n$, so we can take say $N = 2^{2^{2^{2^M}}} + 69$ and that would work.

Next, we have $f_1 = o(f_9)$. This is because we have $f_9(n) = \exp(\log(9n)\log\log n) = (\log n)^{\log(9n)} > (\log n)^3(\log n)^{\log n}$ since $\log(9) > 3$ (we are using base 2), so if M is a positive real number then setting $N = 2^{\sqrt[3]{M}}$ will work, since then we obtain $f_9(n) > M \cdot f_1(n)$ for all $n \geq N$.

Next, we have $f_9 = o(f_6)$. To prove this, we once again take logs, so that it it suffices to prove that for all $M > 0$, there exists N such that for all $n \geq N$, we have $\sqrt[6]{n}\log 6 > M + \log\log n(\log 9 + \log n)$, which is true since we know that n^a (i.e. powers of n) always grows faster than log transforms of n for any $a > 0$.

Finally, we prove that $f_6 = o(f_7)$. We again take logs, which reduces the problem to proving that for any $M > 0$, there exists N such that for all $n \geq N$, we have $n\log 7 - (\log n)^2 > M + \sqrt[6]{n}\log 6$, which is true since n grows faster than $\sqrt[6]{n}$.

(b) **(5 points)** Give an example of a function g that would *not* fit into the order above: that is, one for which, for some i, $f_i \neq \Theta(g)$, $f_i \neq o(g)$, and $g \neq o(f_i)$.

Consider

$$f(n) = \begin{cases} (69n!^{420})^{420^{1337^n}} + 1337^{1337} & \text{if } n \text{ even} \\ -1 & \text{otherwise} \end{cases}$$

Then note that $\lim_{n\to\infty} \frac{f(2n)}{f_i(2n)} = \infty$ but $\lim_{n\to\infty} \frac{f(2n+1)}{f_i(2n+1)} = 0$, so we cannot have $f = o(f_i), \Theta(f_i)$ (because of the former), or $f_i = o(f)$ (because of the latter).

4. In each of the problems below, all functions map positive integers to positive integers.

   (a) **(5 points)** Find (with proof) a function f_1 such that $f_1(n+1) \in O(f_1(n))$.
   Consider $f_1(n) = n$ for all positive integers n. Then $f_1(n+1) = n+1 \in O(n)$, since we have for $n > 69$, $n+1 < 69 \cdot n$ (i.e. can use 69 as bounding constant, or really any positive integer and $n > 1$).

   (b) **(10 points)** Prove that there does not exist any function f such that $f(n+1) \in o(f(n))$.
   Suppose for the sake of contradiction that there did exist a function $f : \mathbb{N} \to \mathbb{N}$ satisfying $f(n+1) \in o(f(n))$. Then there exists N such that for all $n \geq N$, we have $f(n+1) < \frac{1}{69} f(n)$. In particular, this means that for all $m \in \mathbb{N}$ we have $f(N+m) < \frac{1}{69^m} f(N)$. Let $f(N) = M$. Then for $m > \log_{69}(M)$, we have $f(N+m) < 1$, which is a contradiction since f must output positive integers (so it must always be at least 1).

   (c) **(5 points)** Find (with proof) a function f_2 such that $f_2(n+1) \notin O(f_2(n))$.
   Consider $f_2(n) = n!$ for all positive integers n. Suppose for the sake of contradiction that $f_2(n+1) \in O(f_2(n))$. Then there exists positive M, N such that for all $n \geq N$, we have $(n+1)! \leq M \cdot n!$, which is absurd, since we can choose $n = \lceil M \rceil + 69$ to get $(n+1)! = (\lceil M \rceil + 70) \cdot n! > M \cdot n!$, contradiction. So this example works.

5. Buffy and Willow are facing an evil demon named Stooge, living inside Willow's computer. In an effort to slow the Scooby Gang's computing power to a crawl, the demon has replaced Willow's hand-designed super- fast sorting routine with the following recursive sorting algorithm, known as StoogeSort. For simplicity, we think of Stoogesort as running on a list of distinct numbers. StoogeSort runs in three phases. In the first phase, the first 2/3 of the list is (recursively) sorted. In the second phase, the final 2/3 of the list is (recursively) sorted. Finally, in the third phase, the first 2/3 of the list is (recursively) sorted again. Willow notices some sluggishness in her system, but doesn't notice any errors from the sorting routine.

   (a) **(5 points)** We didn't specify what StoogeSort does if the number of items to be sorted is not divisible by 3. Specify what StoogeSort does in those cases in such a way that StoogeSort terminates and correctly sorts.

   It must be the first/last $\lceil \frac{2n}{3} \rceil$ items. If it was the first/last $\lfloor \frac{2n}{3} \rfloor$ items, then for $n = 4$ for example the list $[1, 4, 3, 2]$ will never be sorted since after Stoogesort it becomes $[1, 4, 2, 3]$. This is because $\lfloor \frac{2n}{3} \rfloor = 2$, so the first half will always be "disjoint" from the second half.

   (b) **(15 points)** Prove rigorously that StoogeSort correctly sorts. (You may not assume all numbers to be sorted are distinct.)

   We prove that Stoogesort works by inducting on the array size n. Specifically, we claim that for each positive integer n, Stoogesort correctly sorts any array A with size n (with all elements distinct). We prove this claim by strong induction on n. The base cases $n = 1$ and $n = 2$ are trivial.

   Now suppose that the result is true for all positive integers $k < n$, where $n \geq 3$. It suffices to prove that the claim is true for n. Let the list be originally

$$A = [a_1, a_2, \ldots, a_n]$$

   in that order. After Stoogesorting the first $\lceil \frac{2n}{3} \rceil$ elements of A, suppose that we end up with

$$B = [b_1, b_2, \ldots, b_n],$$

   and that after Stoogesorting the last $\lceil \frac{2n}{3} \rceil$ elements of B, we end up with

$$C = [c_1, c_2, \ldots c_n].$$

   Finally, we Stoogesort the first $\lceil \frac{2n}{3} \rceil$ elements of C to obtain

$$D = [d_1, d_2, \ldots, d_n].$$

   Then D is the result of Stoogesorting the entire list A, and so it suffices to prove that D is correctly sorted. Notice that by definition, we know that

$$b_1 \leq b_2 \leq \cdots \leq b_{\lceil \frac{2n}{3} \rceil},$$

   and that

$$c_1 \leq c_2 \leq \cdots \leq c_{\lfloor \frac{n}{3} \rfloor}, \text{ and } c_{\lfloor \frac{n}{3} \rfloor + 1} \leq c_{\lfloor \frac{n}{3} \rfloor + 2} \leq \cdots \leq c_n.$$

Now since we got D by Stoogesorting the first $\lceil \frac{2n}{3} \rceil$ elements of C, it follows that

$$d_1 \leq d_2 \leq \cdots \leq d_{\lceil \frac{2n}{3} \rceil}, \text{ and } d_{\lceil \frac{2n}{3} \rceil + 1} \leq d_{\lceil \frac{2n}{3} \rceil + 2} \leq \cdots \leq d_n.$$

Thus, in order to prove that D is correctly sorted, it suffices to show that $d_{\lceil \frac{2n}{3} \rceil} \leq d_{\lceil \frac{2n}{3} \rceil + 1}$. But notice that by definition, we have $d_{\lceil \frac{2n}{3} \rceil + 1} = c_{\lceil \frac{2n}{3} \rceil + 1}$, and $d_{\lceil \frac{2n}{3} \rceil} = \max(c_{\lfloor \frac{n}{3} \rfloor}, c_{\lceil \frac{2n}{3} \rceil})$. Now note from the induction hypothesis that $c_{\lceil \frac{2n}{3} \rceil} \leq c_{\lceil \frac{2n}{3} \rceil + 1}$, and so it suffices to prove that $c_{\lceil \frac{2n}{3} \rceil + 1} \geq c_{\lfloor \frac{n}{3} \rfloor}$.

Assume for the sake of contradiction that what we have to prove is false. Then $c_{\lceil \frac{2n}{3} \rceil + 1} < c_{\lfloor \frac{n}{3} \rfloor}$, and so there are at most $n - \lceil \frac{2n}{3} \rceil - 1$ elements of the last $\lceil \frac{2n}{3} \rceil$ elements of C that can be more than it (from $c_{\lceil \frac{2n}{3} \rceil + 2}, \ldots c_n$). However, since C is obtained by sorting the last $\lceil \frac{2n}{3} \rceil$ elements of B, it follows that there are at least $\lceil \frac{2n}{3} \rceil - \lfloor \frac{n}{3} \rfloor$ elements in $c_{\lfloor \frac{n}{3} \rfloor + 1} \leq c_{\lfloor \frac{n}{3} \rfloor + 2} \leq \cdots \leq c_n$ that are larger than $c_{\lfloor \frac{n}{3} \rfloor}$. It then follows that

$$2\lceil \frac{2n}{3} \rceil \leq n + \lfloor \frac{n}{3} \rfloor - 1.$$

This is a contradiction because we have

$$n + \lfloor \frac{n}{3} \rfloor - 1 \leq n + \frac{n}{3} - 1 < \frac{4n}{3} \leq 2\lceil \frac{2n}{3} \rceil,$$

absurd. Thus, D is sorted, which completes the inductive step. So by strong induction, we proved that Stoogesort correctly sorts.

(c) **(5 points)** Give a recurrence describing StoogeSort's running time, and, using that recurrence, give the asymptotic running time of Stoogesort.

Let the running time be $T(n)$. Then it sorts three sublists of length $\lceil \frac{2n}{3} \rceil$, so we have $T(n) = 3T(n/3)$. Then we get $T(n) = 3^k T(\frac{2^k}{3^k} n)$ for all $k \in \mathbb{Z}^+$ by repeatedly substituting $n \to \frac{2n}{3}$, so we can pick $k = \log_{1.5}(n)$ to get that the asymptotic run time is $\Theta(3^{\log_{1.5} n}) \approx \Theta(n^{2.71})$.

6. (a) **(10 points)** Solve the following recurrences exactly, and then prove your solutions are correct. (Hint: Calculate values and guess the form of a solution: then prove that your guess is correct by induction.)

   i. $T(1) = 1$, $T(n) = T(n-1) + 4n - 4$
   We claim that $T(n) = 2n^2 - 2n + 1$. We prove this by induction on n, where the base case is clear: $T(1) = 2 - 2 + 1 = 1$. Now suppose that this is true for $n-1$ where $n \geq 2$ and we will prove it for n. And this is true, because from the recurrence and the induction hypothesis we obtain

   $$T(n) = 2(n-1)^2 - 2(n-1) + 1 + 4n - 4 = 2n^2 - 4n + 2 - 2n + 2 + 1 + 4n - 4 = 2n^2 - 2n + 1,$$

   which completes the inductive step, proving the claim.

   ii. $T(1) = 1$, $T(n) = 2T(n-1) + 2n - 1$
   We claim that $T(n) = 3 \cdot 2^n - (2n+3)$ for all n. We again prove this by induction on n, where the base case is clear: $T(1) = 6 - 5 = 1$. Now suppose that we have $T(n-1) = 3 \cdot 2^{n-1} - (2n+1)$ where $n \geq 2$. Then we need to prove that $T(n) = 2^{n+1} - (2n+1)$, which is true because from the recurrence and the inductive hypothesis we get

   $$T(n) = 3 \cdot 2^n - 4n - 2 + 2n - 1 = 3 \cdot 2^n - 2n - 3,$$

   as desired.

(b) **(10 points)** Give tight asymptotic bounds for $T(n)$ (i.e. $T(n) = \Theta(f(n))$ for some f) in each of the following recurrences.

   i. $T(n) = 4T(n/2) + n^3$
   By the master theorem, we have $a = 4$, $b = 2$ and $k = 3$ so $a < b^k$, meaning $T(n) = \Theta(n^3)$.

   ii. $T(n) = 17T(n/4) + n^2$
   By the master theorem, we have $a = 17$, $b = 4$, and $k = 2$ so $a > b^k$, implying $T(n) = \Theta(n^{\log_4(17)})$.

   iii. $T(n) = 9T(n/3) + n^2$
   By the master theorem, we have $a = 9$, $b = 3$, and $k = 2$, so $a = b^k$, implying $T(n) = \Theta(n^2 \log n)$.

   iv. $T(n) = T(\sqrt{n}) + 1$. (Hint: you may want to change variables somehow.)
   Let $n = 2^k$ to get $T(2^k) = T(2^{\frac{k}{2}}) + 1$. So if $f(k) = T(2^k)$, it follows that $f(k) = f(k/2) + 1$. This means that if we let $g(k) = f(2^k)$, then by setting $k = 2^k$ we get $g(k) = g(k-1) + 1$, so $g(k) = g(0) + k = k + c$ for some constant c. Then $f(2^k) = k + c$, so $T(2^{2^k}) = k + c$, implying $T(n) = \Theta(\log \log n)$.

7. **(0 points, optional)**[1] InsertionSort is a simple sorting algorithm that works as follows on input $A[0], \ldots, A[n-1]$.

Algorithm 1 InsertionSort

```
Input: A
for i = 1 to n − 1 do
  j = i
  while j > 0 and A[j − 1] > A[j] do
    swap A[j] and A[j − 1]
    j = j - 1
  end while
end for
```

Show that for every function $T(n) \in \Omega(n) \cap O(n^2)$ there is an infinite sequence of inputs $\{A_k\}_{k=1}^{\infty}$ such that A_k is an array of length k, and if $t(n)$ is the running time of InsertionSort on A_n, then $t(n) \in \Theta(T(n))$.

[1]This question will not be used for grades, but try it if you're interested. It may be used for recommendations or TF hiring.