CS 124 Homework 1: Spring 2022

Your name: Lev Kruglyak

Collaborators: Swati Goel, Katrina Brown

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 assignme nt must be a nonnegative integer at most **two** (college)/**four** (extension school).

Try to make your answers as clear and concise as possible; style may count in your grades. 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.

You can collaborate with other students that are currently enrolled in this course in brainstorming and thinking through approaches to solutions but you should write the solutions on your own: you must wait one hour after any collaboration or use of notes from collaboration before any writing in your own solutions that you will submit.

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 CODE with others on this problem like you will for the "major" programming assignments. (You may talk about the problem, as you can for other problems.)

Problem 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 $p_1, p_2, \ldots, p_6$ that the die roll is $1, 2, \ldots, 6$.

(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; your solution should need at most 1000 times as many die rolls as an optimal solution does no matter what $p_1, \ldots, p_6$ are; and exceptional solutions will receive exceptional scores.

(a) Suppose we use the simple strategy which can be described as follows: Roll the die twice. If the first die rolled strictly less than the second die, output a heads. If the second die rolled strictly less than the first die, output a tails. Otherwise, if the two dies are equal, start over. This clearly produces an unbiased coin, since the odds of rolling a particular side before the other is equal to the probability of rolling those same sides but in the reverse order.

To calculate it's efficiency first we'll calculate the expected number of die rolls required to obtain a single unbiased flip X_c. Note that

$$\mathbb{E}(X_c) = 2\sum_{k=0}^{\infty} kP_k$$

where P_k is the probability that the first k pairs of dice rolls were all identical. It is clear that $P_k = (p_1^2 + p_2^2 + p_3^2 + p_4^2 + p_5^2 + p_6^2)^k$, so using a simple geometric series we get

$$\mathbb{E}(X_c) = \frac{2}{1 - (p_1^2 + p_2^2 + p_3^2 + p_4^2 + p_5^2 + p_6^2)}.$$

It is clear that this algorithm will work even if all but two of the probabilities will be zero, so it is robust against even extremely biased dies.

This algorithm is far from optimal however. For context, this algorithm produces approximately $\frac{N}{\mathbb{E}(X_c)}$ bits of output for every N rolls of input. Using the Shannon entropy formula, we can obtain the theoretical maximum amount of bits which can be extracted from a (sequence) of die rolls:

$$H(X_c) = -\sum_{i=1}^{6} p_i \log_2(p_i).$$

Thus, the efficiency of our algorithm can be characterized by

$$P(X_c) = \frac{1}{\mathbb{E}(X_c)} \cdot \frac{1}{H(X_c)} = -\frac{1 - (p_1^2 + p_2^2 + p_3^2 + p_4^2 + p_5^2 + p_6^2)}{2\sum_{i=1}^{6} p_i \log_2(p_i)}.$$

For example, when $p_1 = p_2 = \cdots = p_6$, the efficiency is $P(X_c) \approx 16\%$, whereas if $p_1 = p_2$ and $p_3 = p_4 = p_5 = p_6 = 0$, the algorithm becomes identical to the Von Neumann algorithm for coin flips, and the efficiency confirms this, giving us $P(X_c) = 25\%$. In theory, there should be

an algorithm which extracts the maximum amount of entropy from this system, giving us up to 2.58 coin flips per die roll.

(b) Consider the following die roll strategy: Roll the die 3 times. If the three rolls were distinct, there are 6 possibilities for their relative orderings: $123, 132, 213, 231, 312$, and 321. Each of these events clearly occurs with the same probability (3 independent events happening in different orders) so we can match them to outcomes for the die roll. Next, if two of the rolls are the same and one is different, there are three possibilities, $112, 121$, and 211, each happening with equal probability. If we also roll an unbiased die, generated by the algorithm from (a), we have 6 events happening with equal probabilities. We throw out the case when all three are the same, completing the algorithm.

As before it is clear that this strategy still works even if only two of the probabilities are nonzero. To measure how well it works, let's do the same expected value / entropy analysis as in (a). Let $P_1 = p_1^3 + p_2^3 + \cdots + p_6^3$ be the probability that we rolled only one distinct value 3 times, $P_2 = p_1^2(1-p_1) + p_2^2(1-p_2) + \cdots + p_6^2(1-p_6)$ be the probability that we rolled two distinct values out of 3 times, and $P_3 = 1 - P_1 - P_2$ be the probability that we rolled three distinct values in 3 rolls. To calculate the expected value for the number of times it takes to roll an unbiased die (X_d), we'll use the following recurrence:

$$\begin{aligned}
\mathbb{E}(X_d) &= 3P_3 + (3 + \mathbb{E}(X_c))P_2 + (3 + \mathbb{E}(X_d))P_1 \\
&= 3 - 3P_1 - 3P_2 + 3P_2 + \mathbb{E}(X_c)P_2 + 3P_1 + \mathbb{E}(X_d)P_1 \\
&= \frac{3 + \mathbb{E}(X_c)P_2}{1 - P_1}
\end{aligned}$$

where $\mathbb{E}(X_c)$ is the expectation from (a). Now we can do a similar analysis for the algorithms efficiency as we did in (a). Since our output has 6 possible states, we'll switch our logarithms to base 6, so the entropy function becomes

$$H(X_d) = -\sum_{i=1}^{6} p_i \log_6(p_i).$$

As before, our efficiency function is then

$$P(X_d) = \frac{1}{\mathbb{E}(X_d)} \cdot \frac{1}{H(X_d)} = -\frac{1 - P_1}{\sum_{i=1}^{6} p_i \log_6(p_i)(3 + \mathbb{E}(X_c)P_2)}.$$

Assuming the die is unbiased, the efficiency is around $P(X_d) \approx 25\%$, which isn't too bad. If all but two of the probabilities are zero, and the remaining two are equal probability, the efficiency is around $P(X_d) \approx 32\%$.

Problem 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.

(a) **(15 points)** How fast does each method appear to be? Give precise timings if possible.

(b) **(4 points)** What's the first Fibonacci number that's at least 2^{31}?

(c) **(15 points)** Since you should reach "integer overflow" with the faster methods quite quickly, modify your programs so that they return the Fibonacci numbers modulo $65536 = 2^{16}$. For each method, what is the largest value of k such that you can compute the k-th Fibonacci number (modulo 65536) in one minute of machine time?

(a) The recursive algorithm is the slowest by far, staying under 1ms for the first 30 Fibonacci numbers but quickly surpassing 1s after 40 Fibonacci numbers. The iterative algorithm is about $O(n)$ in its runtime, averaging 2.2×10^{-4} ms per small Fibonacci numbers. Similarly, the matrix algorithm averages around 3×10^{-4} ms per Fibonacci number, although it's significantly faster than the iterative algorithm for very large Fibonacci numbers.

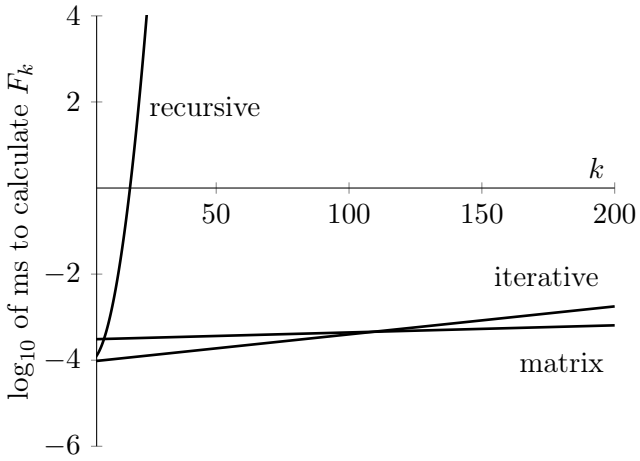

Figure 1: Log graph of various algorithm times

(b) The first Fibonacci number over 2^{31} is $F_{47} = 2,971,215,073$.

(c) The recursive algorithm only reaches 48, the iterative algorithm reaches approximately 25 billion, and the matrix algorithm reaches the staggering $2^{5 \times 10^9}$.

Problem 3.

(a) **(10 points)** Make all true statements of the form $f_i \in o(f_j)$, $f_i \in O(f_j)$, $f_i \in \omega(f_j)$, and $f_i \in \Omega(f_j)$ that hold for $i \leq j$, where $i, j \in \{1, 2, 3, 4\}$ for the following functions. No proof is necessary. All logs are base 2 unless otherwise specified.

(i) $f_1 = (\log n)^{\log n}$

(ii) $f_2 = n^2$

(iii) $f_3 = n(\log n)^3$

(iv) $f_4 = \frac{n!}{4^n}$.

(b) **(5 points)** Give an example of a function $f_5 : \mathbb{N} \to \mathbb{R}^+$ for which *none* of the four statements $f_i \in o(f_5)$, $f_i \in O(f_5)$, $f_i \in \omega(f_5)$, and $f_i \in \Omega(f_5)$ is true for any $i \in \{1, 2, 3, 4\}$.

(a) The rough asymptotic ordering of the functions is $f_3 < f_2 < f_1 < f_4$. From this, we can conclude that:

- $f_1, f_2, f_3, f_4 \in O(f_4)$, $f_1, f_2, f_3 \in o(f_4)$, $f_4 \in \Omega(f_4)$
- $f_3 \in O(f_3)$, $f_1, f_2, f_3 \in \Omega(f_3)$, and $f_1, f_2 \in \omega(f_3)$
- $f_2 \in O(f_2)$, $f_1, f_2 \in \Omega(f_2)$, and $f_1 \in \omega(f_2)$
- $f_1 \in O(f_1)$, $f_1 \in \Omega(f_1)$.

(b) Consider the function

$$f_5(n) = \begin{cases} f_4(n)! & n \text{ is prime} \\ 1 & \text{otherwise} \end{cases}.$$

This clearly satisfies the requirements by the infinitude of primes and by the fact that $f_i \in o(f_4!)$ for all $i \in \{1, 2, 3, 4\}$.

Problem 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 $f_1(n^2) \in O(f_1(n))$.

(b) **(5 points)** Find (with proof) a function f_2 such that $f_2(n^2) \notin O(f_2(n))$

(c) **(10 points)** Prove that there does not exist any function f such that $f(n^2) \in o(f(n))$.

(a) Let $f_1(n) = 1$. Then $f_1(n^2) = f_1(n)$ so clearly $f_1(n^2) \in O(f_1(n))$.

(b) If $f_1(n) = n$, then $f_1(n^2) = n^2$, yet clearly $n^2 \notin O(n)$.

(c) Suppose for the sake of contradiction that $f : \mathbb{N}^+ \to \mathbb{N}^+$ is a function satisfying $f(n^2) \in o(f(n))$. This means that for every $\epsilon > 0$, there exists some $n_0 \in \mathbb{N}^+$ such that for all $n \geq n_0$, $f(n^2) \leq \epsilon f(n)$. Letting $\epsilon = \frac{1}{2}$, we have the relation $f(n^2) \leq \frac{1}{2} f(n)$ for all $n > n_0$. Thus we have an infinite strictly descending sequence, $f(n), f(n^2), f(n^4), \cdots$ since $f(n^{2^k}) \leq \frac{1}{2^k} f(n)$. This is a contradiction to the well ordering principle because $f(n)$ takes on positive integer values.

Problem 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.

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

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

(a) To motivate our answer, suppose the number of items to be sorted is not divisible by 3, and we added one or two "infinite" elements to the array at the end. These elements would stay fixed during the sorting process, and we can remove them at the end. Since these elements didn't affect anything, we might have just as well shrunk the last third.

More formally, we let the first and second thirds of the array be of size $\lceil \frac{n}{3} \rceil$, and the last third to be of size $n - 2\lceil \frac{n}{3} \rceil$. Let's call the three segments of the array $A[1], A[2], A[3]$. Notably, $\#A[3] \leq \#A[1] = \#A[2]$.

(b) Assume inductively that the algorithm works for all sets of size less than the size of the current set. The base case of $n = 1, 2$ can be easily implemented, $n = 1$ doing nothing and $n = 2$ simply swapping the two elements. Now suppose we are given a set of size n.

Let $A_i[j]$ denote the set of elements in the j-th third of the array after stage i. Let $\circ$ be the array concatenation operation, so for example $A_1[1] \circ A_1[2]$ is sorted, $A_2[2] \circ A_2[3]$ is sorted, and $A_3[1] \circ A_3[2]$ is sorted. We'll say that $A_i[j] \geq A_k[m]$ if every element in $A_i[j]$ is greater than or equal to every element $A_k[m]$.

Since $A_1[1] \circ A_1[2]$ is sorted, $A_1[2] \geq A_1[1]$. Now there are at most $\#A[3]$ elements in $A_1[2] \circ A_1[3]$ which are less than or equal to elements in $A_1[1]$. After $A_1[2] \circ A_1[3]$ gets sorted to become $A_2[2] \circ A_2[3]$, these elements must have moved into $A_2[2]$, since $\#A[3] \leq \#A[2]$ and every element in $A_2[2]$ is less than or equal to elements in $A_1[1]$. So it follows that $A_2[3] \geq A_2[1], A_2[2]$, and it is sorted so the $A[3]$ section is sorted after stage 2. After stage 3, $A_2[1] \circ A_2[2]$ is sorted, so the final array $A_3[1] \circ A_3[2] \circ A_3[3]$ is sorted and the algorithm concludes.

(c) A simple recurrence for StoogeSort's running time can be given by

$$T(n) = 3 \cdot T\left(\frac{2}{3}n\right).$$

Thus, by the master theorem of recurrences, $T(n) \in \Theta\left(n^{\log_{2/3} 3}\right)$.

Problem 6. (10 points) Solve the following recurrences exactly, and then prove your solutions are correct.

(a) $T(1) = 1$, $T(n) = T(n-1) + 124n$

(b) $T(1) = 1$, $T(n) = 2T(n-1) + 2n - 1$

(a) To solve this problem in a nice way, we'll use the Newton backward difference formula:

Proposition. Suppose $f : \mathbb{Z} \to \mathbb{Z}$ is some integral function. Define the *backward difference operator* ∇ as $\nabla^0 f(x) = f(x)$ and $\nabla^k f(x) = \nabla^{k-1} f(x) - \nabla^{k-1} f(x-1)$. Then for any $x \in \mathbb{Z}$, we have

$$f(x) = \nabla^0 f(0) + \nabla^1 f(0)x + \nabla^2 f(0)\frac{x(x+1)}{2!} + \cdots$$

In this case, $T(n) - T(n-1) = 124n$ so $\nabla T(n) = 124n$ and $\nabla^2 T(n) = 124$. All higher values are clearly zero. Since $T(1) = T(0) + 124$, $T(0) = -123$ so by the proposition we have

$$T(n) = -123 + 124\left(\frac{n(n+1)}{2}\right) = 62n^2 + 62n - 123.$$

A quick check confirms that this solution satisfies the recurrence.

$$\begin{aligned} 62n^2 + 62n - 123 &= 62(n-1)^2 + 62(n-1) - 123 + 124n \\ &= 62n^2 - 124n + 62 + 62n - 62 - 123 + 124n. \\ &= 62n^2 + 62n - 123 \end{aligned}$$

(b) This recurrence isn't a polynomial recurrence, so we can't use the Newton difference formula to find a closed form for it. Instead, we'll use a generating function. Let $A(x) = \sum_{k\geq 0} T(k)x^k$. Rewriting the recurrence relation as $T(n+1)x^n = 2T(n)x^n + (2n+1)x^n$, we can sum over both sides to obtain

$$\begin{aligned} \sum_{k\geq 0} T(k+1)x^k &= \sum_{k\geq 0} 2T(k)x^k + 2\sum_{k\geq 0} kx^k + \sum_{k\geq 0} x^k \\ \frac{A(x) - T(0)}{x} &= 2A(x) + 2\left(\frac{x}{(1-x)^2}\right) + \frac{1}{1-x} \quad . \\ A(x) &= \frac{x(x+1)}{(1-x)^2(1-2x)} \end{aligned}$$

Lastly, we perform a partial fraction decomposition to obtain

$$\begin{aligned} A(x) &= \frac{3}{1-2x} - \frac{1}{1-x} - \frac{2}{(1-x)^2} \\ \sum_{k\geq 0} T(k)x^k &= \sum_{k\geq 0} 3\cdot(2x)^k - \sum_{k\geq 0} x^k - \sum_{k\geq 0} 2(k+1)x^k \\ &\Downarrow \\ T(n) &= 3\cdot 2^n - 2n - 3. \end{aligned}$$

To verify that this solution satisfies the recurrence,

$$\begin{aligned} 3 \cdot 2^n - 2n - 3 &= 2\left(3 \cdot 2^{n-1} - 2(n-1) - 3\right) + 2n - 1 \\ &= 3 \cdot 2^n - 4n + 4 - 6 + 2n - 1 \\ &= 3 \cdot 2^n - 2n - 3 \end{aligned}$$

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

Algorithm 1 (Insertion Sort).

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 $t(n) \in \Theta(T(n))$.