CS 124 Homework 1: Spring 2022

Your name: Jamin Liu

Collaborators: Brad Campbell

No. of late days used on previous psets: 0
No. of late days used after including this pset: 1

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.)

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.

   Solution:

   Let us consider a sequence of two rolls of the biased die. Let r_1 be the value of the first roll and r_2 be the value of the second roll. Then we may decompose our strategy into two separate cases.

   Case 1: $r_1 \neq r_2$

If $r_1 \neq r_2$, then if $r_1 < r_2$, let us call this a "heads" and if $r_1 > r_2$, let us call this a "tails".

$$r_1 < r_2 \rightarrow \text{heads}$$
$$r_1 > r_2 \rightarrow \text{tails}$$

For every combination of any two distinct values rolled, there are two permutations of the combination that occur with equal likelihood. As such, by assigning one of these permutations to "heads" and the other to "tails", for every possible combination of distinct values, we simulate an unbiased coin flip.

Case 2: $r_1 = r_2$

If $r_1 = r_2$, we disregard the result and initiate another sequence of two rolls.

Expectation:

Next, let us calculate the expected number of rolls until a coin flip is generated. Let t be the expected number of rolls until a coin flip is generated, let f be the number of rolls in a round, and let e be the probability of generating a flip each round. If we succeed in the first round, then we only use f rolls. If we do not, then we have completed f rolls and have to start over again, so the expected number of rolls remaining is still t. As such, we have the equation:

$$t = ef + (1-e)(f+t)$$
$$t = ef + f - ef + t - et$$
$$et = f$$
$$t = f/e$$

Each round consists of 2 rolls, so $f = 2$. The probability that a round generates a flip is the same as the probability that the value of two rolls are not the same, so $e = 1 - \sum_{i=1}^{6} p_i^2$. Thus we have:

$$t = f/e = \frac{2}{1 - \sum_{i=1}^{6} p_i^2}$$

As such the expected number of die rolls to generate an unbiased flip is: $\frac{2}{1-\sum_{i=1}^{6} p_i^2}$.

(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.

Let us consider a sequence of two rolls of the biased die. Let r_1 be the value of the first roll, r_2 be the value of the second roll, and r_3 be the value of the third roll. Then we may decompose our strategy into three separate cases.

Case 1: (Three distinct values) $r_1 \neq r_2 \neq r_3$

If we roll three distinct values, or $r_1 \neq r_2 \neq r_3$, then there exists a smallest value, median value, and largest value rolled. There are 6 unique and equally likely permutations of these three values among r_1, r_2, and r_3. As such, by assigning each of these permutations a die face, we have the following:

$$r_1 < r_2 < r_3 \to 1$$
$$r_1 < r_3 < r_2 \to 2$$
$$r_2 < r_1 < r_3 \to 3$$
$$r_2 < r_3 < r_1 \to 4$$
$$r_3 < r_1 < r_2 \to 5$$
$$r_3 < r_2 < r_1 \to 6$$

For each combination of any three distinct values rolled, there are six permutations of the sequence with equal likelihood. As such, by mapping each of these permutations to the face of a die, we are able to simulate an unbiased die roll for every possible combination of three distinct values.

Case 2: (Exactly two values equal) $r_1 = r_2 \neq r_3 \vee r_1 = r_3 \neq r_2 \vee r_1 \neq r_2 = r_3$

If our sequence of three rolls yields exactly two values that are equal, then there are three unique and equally likely permutations that these three values can be arranged among r_1, r_2, and r_3. We may assign each of these permutations a group of two die faces in the following manner:

$$r_1 = r_2 \neq r_3 \to 1, 2$$
$$r_1 = r_3 \neq r_2 \to 3, 4$$
$$r_1 \neq r_2 = r_3 \to 5, 6$$

Next, once we have determined a group, we may use our strategy from part (a) for simulating an unbiased coin flip to choose between the two die faces in the group. If the result is heads, we select the larger number, and if the result is tails, we select the smaller number. As such, by selecting one of three groups with equal probability and then one of two die faces within a group with equal probability, we are able to simulate an unbiased die roll in any case where exactly two values within a sequence are equal.

Case 3: (All values are equal) $r_1 = r_2 = r_3$

If $r_1 = r_2 = r_3$, we disregard the result and initiate another sequence of three rolls.

Expectation:

Next, let us calculate the expected number of rolls until an unbiased die roll is generated. Let t be the expected number of rolls until an unbiased die roll is generated and let f be the number of rolls in a round. If we have a sequence in which all three values are equal, the we have completed f rolls and have to start over again, so the expected number of rolls remaining is still t. Let the probability that all three values in a sequence are equal be e_1. Next, if we have a sequence in which exactly two values are equal, then we have completed f rolls and successfully selected a group of die faces; however, we still must generate an unbiased coin flip to select a die face, so the expected number of rolls remaining is t_a, where t_a is the expected number of rolls to generate an unbiased coin flip from part (a). Let the probability a sequence has exactly two values that are equal be e_2. Finally, if we have sequence of three distinct values, we have completed f rolls and are finished. The probability that a sequence has three distinct values is $1 - e_1 - e_2$ as the three cases cover ever possible sequence. From here, we may derive an equation for t:

$$\begin{aligned}
t &= e_1(f + t) + e_2(f + t_a) + (1 - e_1 - e_2)(f) \\
t &= e_1 f + e_1 t + e_2 f + e_2 t_a + f - e_1 f - e_2 f \\
t &= e_1 t + f + e_2 t_a \\
t - e_1 t &= f + e_2 t_a \\
t &= \frac{f + e_2 t_a}{1 - e_1}
\end{aligned}$$

Each round consists of 3 rolls, so $f = 3$. e_1, or the probability that all three values in a sequence are equal can be written a $e_1 = \sum_{i=1}^{6} p_i^3$ as the probability of getting a value i three times is p_i^3. e_2, or the probability that exactly two values in a sequence are equal can be written a $e_2 = 3\sum_{i=1}^{6} p_i^2(1 - p_i)^2$, as the probability of getting a value i twice is p_i^2 and the probability of not getting i is $(1 - p_i)$. We multiply by three as there are three possible permutations for a sequence where exactly two values are equal. From part (a) we have that $t_a = \frac{2}{1-\sum_{i=1}^{6} p_i^2}$.

Finally we have:

$$\begin{aligned}
t &= \frac{f + e_2 t_a}{1 - e_1} \\
t &= \frac{3 + \frac{6\sum_{i=1}^{6} p_i^2(1-p_i)^2}{1-\sum_{i=1}^{6} p_i^2}}{1 - \sum_{i=1}^{6} p_i^3}
\end{aligned}$$

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.

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) **(15 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.)

Solution:

Using the python *time*() function, I recorded the amount of time each Fibonacci method took to compute the 50th Fibonacci number. For each method, the time to compute the 50th Fibonacci number follows:

Recursive: 232376.131604 milliseconds

Iterative: 0.01311302185 milliseconds

Matrix: 0.01592037048 milliseconds

These results make sense as in class we showed that the Recursive method has $O(2^n)$ run time while the Iterative and Matrix methods both have $O(n^2)$ running time.

(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.)

Solution:

The first Fibonacci number that exceeds integer overflow is 2,971,215,073.

(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 $65536 = 2^{16}$. (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?

Solution:

For each method, the maximum kth Fibonacci number (modulo 65536) in one minute of machine time follows:

Recursive: 43rd Fibonacci number

Iterative: $230,087,529$th Fibonacci number

Matrix: $2,147,483,647$th Fibonacci number was calculated in roughly 1 millisecond. The integer overflow limit for the input k of the matrix method was exceeded in well

under a minute.

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

   Solution:

   f_1: $f_1 \in \omega(f_2)$, $f_1 \in \Omega(f_2)$, $f_1 \in \omega(f_3)$, $f_1 \in \Omega(f_3)$, $f_1 \in o(f_4)$, $f_1 \in O(f_4)$
   f_2: $f_2 \in \omega(f_3)$, $f_2 \in \Omega(f_3)$, $f_2 \in o(f_4)$, $f_1 \in O(f_4)$
   f_3: $f_3 \in o(f_4)$, $f_3 \in O(f_4)$

   (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 $i \in \{1, 2, 3, 4\}$.

   Solution:

   Consider the function f_5:

$$f_5 = \begin{cases} 0 & \text{if } n \bmod 2 = 0 \\ n^n & \text{if } n \bmod 2 = 1 \end{cases}$$

   The behaviour of this function is undefined as x approaches infinity. As such, $\lim_{x\to\infty} \frac{f_5}{f_i}$ and $\lim_{x\to\infty} \frac{f_i}{f_5}$ are both indeterminate for all f_i where $i \in \{1, 2, 3, 4\}$ from part (a). As such, the statements $f_i \in o(f_5)$, $f_i \in O(f_5)$, $f_i \in \omega(f_5)$, and $f_i \in \Omega(f_5)$ are not true for $i \in \{1, 2, 3, 4\}$.

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

   Solution:

   Consider the constant function $f_1(x) = 1$. By definition, we may say $f_1(n^2) \in O(f_1(n))$ if there exists $c, N \in \mathbb{N}$ such that $f_1(n^2) \leq c\dot{f}_1(n)$ for all $n > N$. Accordingly, in order to prove $f_1(n^2) \in O(f_1(n))$, we must prove there exists $c, N \in \mathbb{N}$ such that $f_1(n^2) \leq c\dot{f}_1(n)$ holds for all $x > N$.

   Let us consider the values $c = 2$ and $N = 1$. Since f_1 is a constant function, for

all x, $f_1(x) = 1$. Thus, we have:

$$\begin{aligned} f_1(n^2) &\leq 2f_1(n) \\ 1 &\leq 2(1) \\ 1 &\leq 2 \end{aligned}$$

As such, we have shown that there exists $c, N \in \mathbb{N}$ such $f_1(n^2) \leq c\dot{f}_1(n)$ for all $n > N$, namely $c = 2$ and $N = 1$ (though $f_1(n^2) \leq c\dot{f}_1(n)$ is true for any $c, N \in \mathbb{N}$). Thus, $f_1(n^2) \in O(f_1(n))$.

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

Solution:

Consider the identity function $f_2(x) = x$. Let us assume towards contradiction that $f_2(n^2) \in O(f_2(n))$. Then by definition, there exists $c, N \in \mathbb{N}$ such that $f_2(n^2) \leq c\dot{f}_2(n)$ for all $n > N$. Expanding this we have:

$$\begin{aligned} f_2(n^2) &\leq cf_2(n) \\ n^2 &\leq cn \end{aligned}$$

Since we have that f_2 only maps positive integers to positive integers, we may divide both sides by n without altering the inequality. As such, we have:

$$\begin{aligned} n^2 &\leq cn \\ n &\leq c \end{aligned}$$

Accordingly, we have that if $f_2(n^2) \in O(f_2(n))$, there exists $c, N \in \mathbb{N}$ such that $n \leq c$ for all $n > N$. Next, since $c, N \in \mathbb{N}$, let $n = c + N$ such that $n > N$. Then, we have:

$$\begin{aligned} c + N &\leq c \\ N &\leq 0 \end{aligned}$$

But this cannot be true because $N \in \mathbb{N}$. Thus, by contradiction, $f_2(n^2) \notin O(f_2(n))$.

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

Solution:

Let us assume towards contradiction that there exists a function f such that $f(n^2) \in o(f(n))$. Then by definition, there exists $c, N \in \mathbb{N}$ such that $f(n^2) \leq c\dot{f}(n)$ for all $n > N$. Without loss of generality, let $c = 1$. Then we have $f(n^2) < f(n)$. Similarly, we may observe $f(n^4) < f(n^2)$, $f(n^8) < f(n^4)$, and so on. More generally, we may summarize this observation as:

$$\forall k \in \mathbb{N}, f(n^{2^{k+1}}) < f(n^{2^k})$$

Next, we may observe that since f only maps positive integers to positive integers, if $\forall k \in \mathbb{N}, f(n^{2^{k+1}}) < f(n^{2^k})$, then every $f(n^{2^k})$ must be greater than every $f(n^{2^{k+1}})$ by at least 1. More generally, we may summarize this observation as:

$$\forall k \in \mathbb{N}, f(n^{2^{k+1}}) < f(n) - k$$

Since this statement holds for all $k \in \mathbb{N}$, let us consider the case in which $f(n) = k$. Then we have:

$$\begin{aligned} f(n^{2^{k+1}}) &< f(n) - k \\ f(n^{2^{k+1}}) &< k - k \\ f(n^{2^{k+1}}) &< 0 \end{aligned}$$

But this cannot be true because f only maps positive integers to positive integers. Thus, by contradiction, there cannot exist a function such that $f(n^2) \in o(f(n))$.

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.

   Solution:

   Let n be the number of items in the list sorted by StoogeSort. In cases where n is not divisible by 3, StoogeSort will sort the first $\lceil \frac{2}{3}n \rceil$ elements of the list, then the last $\lceil \frac{2}{3}n \rceil$ elements of the list, then the first $\lceil \frac{2}{3}n \rceil$ elements of the list again. We take the ceiling to ensure there is adequate overlap between the two portions of the list that are called by StoogeSort so that all elements are sorted correctly. If were to use $\lfloor \frac{2}{3}n \rfloor$ instead, there would be items in the middle of the list that would not be properly sorted by the algorithm.

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

   Solution:

   Claim: StoogeSort sorts correctly.

Proof by induction:

Let n be the number of elements in the list sorted by StoogeSort.

Base Case:

If $n = 0$, or $n = 1$, StoogeSort returns a correctly sorted list. If a list has a single element or no elements, it does not need to be sorted as it is always in correct order by default. When $n = 2$, $\lceil \frac{2}{3}n \rceil = 2$, so StoogeSort will sort the entire list all at once.

Inductive Hypothesis: Assume for some integer k, where k ¿ 1, StoogeSort correctly sorts all lists of length n where $n \leq k$.

Inductive Step: We seek to prove that if StoogeSort correctly sorts all lists of length n where $n \leq k$ then it correctly sorts a list of length $k + 1$. Following the algorithm, StoogeSort will sort the first $\lceil \frac{2}{3}(k+1) \rceil$ elements of the list, then the last $\lceil \frac{2}{3}(k+1) \rceil$ elements of the list, then the first $\lceil \frac{2}{3}(k+1) \rceil$ elements of the list again. From our inductive hypothesis, we know that StoogeSort can correctly sort a list of length $\lceil \frac{2}{3}(k+1) \rceil$ since $\lceil \frac{2}{3}(k+1) \rceil < k$. From part(a) above we know that sorting in sections of size $\lceil \frac{2}{3}(k+1) \rceil$ guarantees adequate overlap, meaning all $k + 1$ elements will be sorted at least once. By sorting the first $\lceil \frac{2}{3}(k+1) \rceil$ elements of the list, then the last $\lceil \frac{2}{3}(k+1) \rceil$ elements of the list, we ensure that the largest $\lceil \frac{1}{3}(k+1) \rceil$ elements are sorted correctly within the list. Next, the final call ensures the remaining smallest $\lceil \frac{2}{3}(k+1) \rceil$ is sorted correctly as StoogeSort can correctly sort a list of length $\lceil \frac{2}{3}(k+1) \rceil$. As such, StoogeSort correctly sorts a list of length $k + 1$ and thus, sorts correctly.

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

Solution:

The running time of StoogeSort can be described by the following recurrence:

$$T(n) = 3 \cdot T(\frac{2}{3}n) + c$$

Where c is a constant factor. Following the Master Recurrence theorem from class, we know that for $a > 0, b > 1, c > 0, k$ real, the solution to $T(n) = aT(\frac{n}{b}) + cn^k$ is given by $T(n) = \Theta(n^{\frac{\log a}{\log b}})$ if $a > b^k$. In this case, $a = 3$, $b = \frac{3}{2}$ and $k = 0$. Thus, $a > b^k$, so the asymptotic running time of StoogeSort can be written as:

$$\Theta(n^{\frac{\log a}{\log b}})$$
$$\Theta(n^{\frac{\log 3}{\log 3/2}})$$
$$\Theta(n^{2.71})$$

6. **(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.)

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

   Solution:

$$\begin{aligned} T(n) &= T(n-1) + 124n \\ &= T(n-2) + 124(n-1) + 124n \\ &= T(n-3) + 124(n-2) + 124(n-1) + 124n \\ &\cdots \\ &= T(1) + 124\sum_{i=2}^{n}(i) \\ &= 1 + 124\frac{(n-1)(n+2)}{2} \\ &= 62(n-1)(n+2) + 1 \end{aligned}$$

   Proof by induction:

   Claim:

   The recurrence $T(1) = 1$, $T(n) = T(n-1) + 124n$ is equivalent to $T(n) = 62(n-1)(n+2) + 1$.

   Base case $n = 1$: $T(1) = 1$

$$\begin{aligned} T(1) &= 62(1-1)(1+2) + 1 \\ &= 62(0)(3) + 1 \\ &= 1 \end{aligned}$$

   Inductive hypothesis:

   Assume for all natural numbers less than or equal to n, $T(n) = 62(n-1)(n+2) + 1$

   Inductive step:

Consider $T(n+1)$.

$$\begin{aligned}
T(n+1) &= T(n) + 124(n+1) \\
&= 62(n-1)(n+2) + 1 + 124(n+1) \\
&= 62n^2 + 62n - 124 + 1 + 124n + 124 \\
&= 62n^2 + 186n + 1 \\
&= 62(n)(n+3) + 1 \\
&= 62((n+1)-1)((n+1)+2) + 1
\end{aligned}$$

Thus, the recurrence $T(1) = 1$, $T(n) = T(n-1) + 124n$ is equivalent to $T(n) = 62(n-1)(n+2) + 1$.

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

Solution:

$$\begin{aligned}
T(n) &= 2T(n-1) + 2n - 1 \\
&= 2(2T(n-2) + 2(n-1) - 1) + 2n - 1 \\
&= 2(2(2T(n-3) + 2(n-2) - 1) + 2(n-1) - 1) + 2n - 1 \\
&= (2^3 T(n-3) + 2^3(n-2) - 2^2) + (2^2(n-1) - 2) + (2n-1) \\
&= (2^{n-1}T(1) + 2^{n-1}(2) - 2^{n-2}) + \cdots + (2^3(n-2) - 2^2) + (2^2(n-1) - 2) + (2n-1) \\
&= 2^n + 2^{n-1} + 2 \cdot 2^{n-2} + 3 \cdot 2^{n-3} + \cdots + (n-2) \cdot 2^2 + (n-1) \cdot 2 - 1 \\
&= (2^n + 2^{n-1} + \cdots + 2) + (2^{n-1} + 2^{n-2} \cdots + 2) + \ldots (2^2 + 2) + (2) - 1 \\
&= (2^{n+1} - 2) + (2^{n-1} - 2) + \cdots (2^3 - 2) + (2^2 - 2) - 1 \\
&= (2^{n+1} - 2) + (2^n - 2) + (2^{n-1} - 2) + \cdots (2^3 - 2) + (2^2 - 2) - 1 - (2^n - 2) \\
&= (2^{n+1} + 2^n + 2^{n-1} + \cdots + 2^2) - 2n - 1 - (2^n - 2) \\
&= (2^{n+2} - 4) - 2n - 2^n + 1 \\
&= 2^n(2^2 - 1) - 2n - 3 \\
&= 3 \cdot 2^n - 2n - 3
\end{aligned}$$

Proof by induction:

Claim:

The recurrence $T(1) = 1$, $T(n) = 2T(n-1) + 2n - 1$ is equivalent to $T(n) = 3 \cdot 2^n - 2n - 3$.

Base case $n = 1$: $T(1) = 1$

$$\begin{aligned} T(1) &= 3 \cdot 2^1 - 2(1) - 3 \\ &= 3 \cdot 2 - 2 - 3 \\ &= 6 - 2 - 3 \\ &= 1 \end{aligned}$$

Inductive hypothesis:

Assume for all natural numbers less than or equal to n, $T(n) = 3 \cdot 2^n - 2n - 3$.

Inductive step:

Consider $T(n+1)$.

$$\begin{aligned} T(n+1) &= 2T(n) + 2(n+1) - 1 \\ &= 2(3 \cdot 2^n - 2n - 3) + 2(n+1) - 1 \\ &= 6 \cdot 2^n - 4n - 6 + 2n + 2 - 1 \\ &= 6 \cdot 2^n - 2n - 5 \\ &= 3 \cdot 2^{n+1} - 2(n+1) - 3 \end{aligned}$$

Thus, the recurrence $T(1) = 1$, $T(n) = 2T(n-1) + 2n - 1$ is equivalent to $T(n) = 3 \cdot 2^n - 2n - 3$.

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.

Solution 5a:

According to CRLS Textbook pg. 161, StoogeSort uses a value k, which is the number of items to be sorted divided by 3 rounded down (floored). If there n items to be sorted, StoogeSort will sort items 0 to $n - k - 1$, then k to $n - 1$, then 0 to $n - k - 1$. In essence, this rounds up what StoogeSort considers to be 2/3 of the nu mber of items, ensuring that each item is sorted against every other item.