CS 124 Homework 1: Spring 2024

Your name: Anusha Murali

Collaborators:

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

Homework is due Wednesday Jan 31 at 11:59pm ET. You are allowed up to **twelve** late days, but the number of late days you take on each assignment must be a nonnegative integer at most **two**.

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 your papers to turn them in.

Collaboration Policy: You may collaborate on this (and all problem sets) only with other students currently enrolled in the class, and of course you may talk to the Teaching Staff or use Ed. You may also consult the recommended books for the course and course notes linked from the timetable. You may not use Generative AI or large language models, or search the web for solutions, or post the questions on chat forums. Furthermore, you must follow the "one-hour rule" on collaboration. You may not write anything that you will submit within one hour of collaborating with other students or using notes from such sources. That is, whatever you submit must first have been in your head alone, or notes produced by you alone, for an hour. Subject to that, you can collaborate with other students, e.g. in brainstorming and thinking through approaches to problem-solving.

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 no credit. Again, try to make your answers clear and concise.

There is a (short) programming problem on this assignment; you **should *NOT* code** with others on this problem like you will for the “major” programming assignments later in the course. (You may talk about the problem, as you can for other problems.)

Problems

Please note that Question 4 will use concepts covered on Monday, 1/29 (as well as in sections). Students are advised to begin this question until after they learn it in a formal setting.

1. In class we saw a simple proof that Euclid’s algorithm for n-bit integers terminates in $O(n)$ steps. In this problem, you will provide a tighter bound on the constant factor in $O(n)$.

   Given positive integers A, B satisfying $A \geq B$, let $(A', B') = (B, A \bmod B)$, denote the result of one step of Euclid’s algorithm.

(a) (**5 points**) Prove that $A' + B' \leq \frac{2}{3}(A+B)$. Conclude that starting from (A,B), Euclid's algorithm terminates after at most $\log_{3/2}(A+B)$ steps.

Since $A' = B$ and $B' = A \bmod B = A - BQ$, we find

$$A' + B' = B + (A - BQ)$$
$$A' + B' = \frac{2}{3}(A+B) + \left(\frac{A}{3} + \frac{B}{3} - BQ\right) \tag{1}$$

The second expression on the RHS is ≤ 0. We can easily see this by re-writing the expression as follows:

$$\left(\frac{A}{3} + \frac{B}{3} - BQ\right) = \frac{A}{3} - \frac{2BQ}{3} + \frac{B}{3} - \frac{BQ}{3} = \left(\frac{A-2BQ}{3}\right) + \left(\frac{B-BQ}{3}\right) \tag{2}$$

Since $A - BQ = B'$, $B' \leq A'$ and $A' = B$, we note that $A - BQ \leq B$ or $A - BQ - B \leq 0$. But $BQ \geq B$. Therefore, $A - 2BQ \leq 0$.
Similarly, we note that $B - BQ \leq 0$ as $BQ \geq B$.
Therefore,

$$\left(\frac{A}{3} + \frac{B}{3} - BQ\right) = \left(\frac{A-2BQ}{3}\right) + \left(\frac{B-BQ}{3}\right) \leq 0.$$

Plugging this in Equation 1, we find,

$$A' + B' = \frac{2}{3}(A+B) + \left(\frac{A}{3} + \frac{B}{3} - BQ\right)$$
$$A' + B' \leq \frac{2}{3}(A+B).$$

So we find that $A' + B' \leq \frac{2}{3}(A+B)$. □

Now we show that Euclid's algorithm terminates after at most $\log_{3/2}(A+B)$ steps.
Suppose that Euclid's algorithm makes k recursive calls. At the final recursive call we must have $A_k \geq 1$ and $B_k = 0$, which implies $A_k + B_k \geq 1$.
However, as we showed above, $A_k + B_k$ shrinks by a factor of $\left(\frac{2}{3}\right)$ during each iteration. Therefore, we have $A_k + B_k \leq \left(\frac{2}{3}\right)^k (A+B)$. Hence we conclude that we must have $1 \leq \left(\frac{2}{3}\right)^k (A+B)$.

$$1 \leq \left(\frac{2}{3}\right)^k (A+B).$$

Taking logarithm on both sides in base (3/2), we obtain,

$$\log_{3/2}(1) \leq k \log_{3/2}\left(\frac{2}{3}\right) + \log_{3/2}(A+B)$$
$$0 \leq -k + \log_{3/2}(A+B)$$
$$k \leq \log_{3/2}(A+B).$$

Hence, we conclude that Euclid's algorithm terminates after at most $\log_{3/2}(A+B)$ steps. □

(b) (**3 points**) In lieu of $A+B$, consider a more general function $g(A,B) = A+\beta B$ for $\beta > 0$. Let $A = QB+R$, where R is the remainder and Q is the quotient when dividing A by B. Give an exact expression for $L' = g(A',B')$ solely in terms of B,R,β, and give a lower bound L for $g(A,B)$ solely in terms of B,R,β. Briefly justify your answer.

From the Euclid algorithm, using the input (A,B), after one step, $A' = B$ and $B' = A \mod B = R$. Therefore,

$$L' = g(A',B') = g(B,R) = B+\beta R. \quad \square$$

We proceed to find the lower bound for $g(A,B)$ as follows.

$$\begin{aligned} g(A,B) &= A+\beta B \\ &= (QB+R)+\beta B \\ &= (Q+\beta)B+R. \\ L &= (1+\beta)B+R. \quad \square \end{aligned}$$

The lower bound L for $g(A,B)$ occurs when $Q = 1$. This is because, for $A > B$, only when the decrease $A-B$ is the smallest, which occurs when the integer quotient $Q = 1$, $g(A,B)$ can attain the lowest value. This is when $A \mod B = A-B$ or when $Q = 1$. $\square$

(c) (**5 points**) Define a choice of β for which the ratio L'/L in the previous question is always the same regardless of B,R. Prove your answer. (Note: β and this ratio will be irrational numbers)

In the previous part, we found $L = (1+\beta)B+R$ and $L' = B+\beta R$. Therefore, the ratio L'/L is,

$$\frac{L'}{L} = \frac{B+\beta R}{(1+\beta)B+R}$$

We want the ratio L'/L to remain the same regardless of B and R. Factoring out β from the numerator, we obtain,

$$\frac{L'}{L} = \frac{B+\beta R}{(1+\beta)B+R} = \frac{\beta\left(\frac{1}{\beta}\cdot B+R\right)}{(1+\beta)B+R}$$

We note that when $\frac{1}{\beta} = 1+\beta$, the ratio $L'/L = \beta$, which is independent of B and R. Hence, solving $\frac{1}{\beta} = 1+\beta$, and noting $\beta > 0$, we find,

$$\begin{aligned} \beta^2+\beta-1 &= 0 \\ \beta = \frac{-1\pm\sqrt{1+4}}{2} &= \frac{\sqrt{5}-1}{2}. \quad \square \end{aligned}$$

Plugging this value of β into the expression for the ratio L'/L, we confirm that $\frac{L'}{L} = \beta = \frac{\sqrt{5}-1}{2}$.

$$\frac{L'}{L} = \frac{B+\beta R}{(1+\beta)B+R} = \frac{B+\left(\frac{\sqrt{5}-1}{2}\right)\cdot R}{\left(1+\frac{\sqrt{5}-1}{2}\right)\cdot B+R} = \frac{\sqrt{5}-1}{2}.$$

(d) (**5 points**) Use this choice of β to prove an improved upper bound on the number of steps of Euclid's algorithm starting from (A, B).

Let us consider $g(A,B) = A + \beta B$. The lower bound of $g(A,B)$ is L. In one step of the Euclid's algorithm, from the results of the previous part, we obtain $g(A', B') = L' = A' + \beta B' = \beta L$, where L is the lower bound on $g(A,B)$. Hence, we find,

$$\begin{aligned} L' &= \beta L \\ &\leq \beta \cdot g(A,B) \qquad \text{(Because } L \text{ is the lower bound on } g(A,B).) \end{aligned}$$

Suppose that Euclid's algorithm makes k recursive calls before terminating. At the final recursive call, we must have that $A_k \geq 1$ and $B_k = 0$, which implies that $A_k + \beta B_k \geq 1$. Let $L^{(k)} = A_k + \beta B_k$ be the value of L' after k recursive calls. Therefore, we find,

$$\begin{aligned} L^{(k)} &\leq \beta^k \cdot g(A,B) \\ A_k + \beta B_k &\leq \beta^k \cdot (A + \beta B) \\ 1 &\leq \beta^k \cdot (A + \beta B). \end{aligned}$$

Since $\beta = \dfrac{\sqrt{5}-1}{2}$, it is the inverse of the Golden Ratio, $\phi = \dfrac{\sqrt{5}+1}{2}$. Hence taking logarithm on both sides in base ϕ, we obtain,

$$\begin{aligned} \log_\phi(1) &\leq k \log_\phi \beta + \log_\phi(A + \beta B) \\ 0 &\leq k \log_\phi \left(\frac{1}{\phi}\right) + \log_\phi(A + \beta B) \\ k &\leq \log_\phi(A + \beta B) < \log_\phi(A + B). \end{aligned}$$

Hence the upper bound on the number of steps of Euclid's algorithm starting from (A, B) is $k = \log_\phi(A + B)$, where $\phi = \dfrac{\sqrt{5}+1}{2}$, is the Golden Ratio.

Note that a stricter upper bound is, $k \leq \log_\phi(A + \beta B) = \log_\phi\left(A + \dfrac{B}{\phi}\right)$. □

(e) (**2 points**) Construct an increasing sequence of inputs (A, B) for which the bound in the previous question is asymptotically tight. Informally justify why the sequence you constructed is tight in 1-2 sentences.

The sequence of inputs (A, B) for which the bound in the previous question is asymptotically tight is the sequence of pairs of consecutive Fibonacci numbers (F_{k+1}, F_k), for $k > 1$.
Therefore, we can construct an increasing sequence of input using the Fibonacci numbers,

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$$

and the sequence of inputs (A, B) that we are interested in are,

$$(1,1), (2,1), (3,2), (5,3), (8,5), \ldots, (55,34), \ldots$$

The reason why the above sequence provides the desired asymptotically tight bound is any two consecutive Fibonacci numbers are relatively prime. Secondly, each Fibonacci number is constructed by adding the previous two Fibonacci numbers. Therefore in the Euclid algorithm, we find that $A \mod B = F_{k+2} \mod F_{k+1} = F_{k+1} \mod (F_{k+2} - F_{k+1}) = F_{k+1} \mod F_k$. Hence, it will take k steps before the Euclid algorithm terminates with finding the GCD as 1.

(Note; Part (e) above may be attempted independently before attempting other parts and the answer may give a hint to solving Parts (b)-(d).)

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? (This is deliberately open-ended; part of the problem is to decide what constitutes a reasonable answer.) Include precise timings if possible—you will need to figure out how to time processes on the system you are using, if you do not already know.

   The timing, averaged over 5 runs, on my MacBook for the three algorithms are given in the following table:

Algorithm	Runtime for F_{20}	Runtime for F_{40}	Runtime for F_{50}	Runtime for F_{100}
Recursive	0.051 s	10.691 s	1325.560 s	Did not terminate
Iterative	0.051 s	0.051 s	0.051 s	0.052 s
Matrix Method	0.010 s	0.028 s	0.041 s	0.043 s

   Recursive algorithm

   When we express the running time of the recursive algorithm, we can express it as a recurrence relation on input n. When we simply count the number of addition operations of this algorithm, we can write,

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

   which is analogous to the Fibonacci equation. Hence, we note that the number of addition operation is growing exponentially on n or $T(n)$ is in the order of $2^{\Theta(n)}$. Since each addition operation involves adding n bit integers, which requires $O(n)$ time, we conclude that $T(n) = n \cdot 2^{O(n)}$. This is consistent with my experiments. It took more than 22 minutes to compute F_{50} on my MacBook. The estimation for $T(50) \approx 5.6 \times 10^{16}$ and $T(100) > 1.3 \times 10^{32}$. Notably, I could not find F_{100} on my MacBook using the recursive method.

   Iterative algorithm

   The iterative algorithm does an n-bit integer addition in the for loop, which iterates n times, for a total runtime of $O(n^2)$. Therefore, the estimated runtime for computing F_{100} is $T(100) \approx 10^4$, is many orders of magnitude smaller than the recursive algorithm, which as indicated above, required an estimated runtime of $T(100) > 1.3 \times 10^{32}$. This is consistent with my experiments as it only took a mere 0.052 seconds to compute F_{100} on my MacBook using the iterative method.

   Matrix exponentiation algorithm

   The matrix exponentiation algorithm on the average was found to be faster than the iterative algorithm for computing very large Fibonacci numbers ($n > 100$). This is because, the computation of Fibonacci numbers using the matrix exponentiation approach involves exponentiating a 2×2 matrix to the n-th power.

$$\begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n \begin{pmatrix} F_1 \\ F_0 \end{pmatrix}.$$

   In order to raise a matrix A to the n-th power, we compute $A^{n/2}$ and then square the results, which leads us to the recurrence equation, $T(n) = T(n/2) + 1$. Hence $T(n) = \log n$. The total

runtime of the matrix exponentiation algorithm is $O(n \log n)$, which is consistent with my experiment. For small $n < 100$, I could not see noticeable differences between the iterative and the matrix approaches.

(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.)

The first Fibonacci number that's at least 2^{31} is $F_{47} = 2971215073$.

(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 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 k^{th} Fibonacci number (modulo 65536) in one minute of machine time? If that value of k would be too big to handle (e.g. if you'd get integer overflow on k itself) but you can still calculate F_k quickly, you may report the largest value k_{max} of k you can handle and the amount of time the calculation of $F_{k_{\text{max}}}$ takes.

The maximum value of k such that I can compute F_k (modulo 65536) under various elapsed times are listed in the table below. I have highlighted the row in red for the k_{max} that takes approximately 1 minute of machine time on my MacBook.

Algorithm	k_{max}	**Elapsed Time (s)**
Recursive	38	57.40
Recursive	39	92.71
Iterative	$2^{26} = 67,108,864$	58.57
Iterative	$2^{28} = 268,435,456$	231.89
Iterative	$2^{30} = 1,073,741,824$	986.31
Matrix Method	$2^{32} = 4,294,967,296$	0.018
Matrix Method	2^{64}	0.047
Matrix Method	$2^{1,000,000}$	1.59
Matrix Method	$2^{40,000,000}$	59.48
Matrix Method	$2^{50,000,000}$	75.09
Matrix Method	$2^{100,000,000}$	147.65
Matrix Method	$2^{500,000,000}$	746.64
Matrix Method	$2^{1,000,000,000}$	1538.011

For the recursive algorithm, when I converted all arithmetic operations to modulo 2^{16}, I did not see any performance improvement. In fact, due to the additional modulo operations in the recursive calls, the overall runtime noticeably increased. It took more than 92 seconds to compute F_{39}, whereas I was able to compute F_{40} in Part (a) in nearly 11 seconds.

For the matrix exponentiation algorithm, in order to compute F_n, where $n = 2^m$, I used Python's `numpy.linalg.matrix_power()` method to square the 2×2 matrix $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$ m times, before finally multiplying it with $\begin{pmatrix} F_1 \\ F_0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$. This approach yields an $O(n \log n)$ runtime, which is consistent with my experimental results tabulated above. The table above shows that the runtime

of the matrix method increases in $\log_2(2^k) = k$ time as (n mod 65536) is always < 65536. In only 1538 seconds (25 minutes), I was able to compute the $2^{1,000,000,000}$-th Fibonacci number using modulo 65536!

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 $i, j \in \{1,2,3,4,5\}$ for the following functions. No proof is necessary. All logs are base 2 unless otherwise specified.

   i. $f_1 = (\log n)^{\log n}$
   ii. $f_2 = 2^{\sqrt{\log n}}$
   iii. $f_3 = 2^{2^{\sqrt[3]{\log\log\log n}}}$
   iv. $f_4 = n^{\log\log n}$
   v. $f_5 = (\log\log n)^n$

   i. $f_1 \in O(f_1)$, $f_1 \in \Omega(f_1)$, $f_1 \in \Omega(f_2)$, $f_1 \in \omega(f_2)$, $f_1 \in \Omega(f_3)$, $f_1 \in \omega(f_3)$,$f_1 \in O(f_4)$, $f_1 \in \Omega(f_4)$, $f_1 \in O(f_5)$, $f_1 \in o(f_5)$.
   ii. $f_2 \in O(f_1)$, $f_2 \in o(f_1)$, $f_2 \in O(f_2)$, $f_2 \in \Omega(f_2)$, $f_2 \in O(f_3)$, $f_2 \in o(f_3)$, $f_2 \in O(f_4)$, $f_2 \in o(f_4)$, $f_2 \in O(f_5)$, $f_2 \in o(f_5)$.
   iii. $f_3 \in O(f_1)$, $f_3 \in o(f_1)$, $f_3 \in \Omega(f_2)$, $f_3 \in \omega(f_2)$, $f_3 \in O(f_3)$, $f_3 \in \Omega(f_3)$, $f_3 \in O(f_4)$, $f_3 \in o(f_4)$, $f_3 \in O(f_5)$, $f_3 \in o(f_5)$.
   iv. $f_4 \in O(f_1)$, $f_4 \in \Omega(f_1)$, $f_4 \in \Omega(f_2)$, $f_4 \in \omega(f_2)$, $f_4 \in \Omega(f_3)$, $f_4 \in \omega(f_3)$, $f_4 \in O(f_4)$, $f_4 \in \Omega(f_4)$, $f_4 \in O(f_5)$, $f_4 \in o(f_5)$.
   v. $f_5 \in \Omega(f_1)$, $f_5 \in \omega(f_1)$, $f_5 \in \Omega(f_2)$, $f_5 \in \omega(f_2)$, $f_5 \in \Omega(f_3)$, $f_5 \in \omega(f_3)$, $f_5 \in \Omega(f_4)$, $f_5 \in \omega(f_4)$, $f_5 \in O(f_5)$, $f_5 \in \Omega(f_5)$.

   The above relationships are summarized in the table below.

	f_1	f_2	f_3	f_4	f_5
f_1	O, Ω	Ω, ω	Ω, ω	O, Ω	O, o
f_2	O, o	O, Ω	O, o	O, o	O, o
f_3	O, o	Ω, ω	O, Ω	O, o	O, o
f_4	O, Ω	Ω, ω	Ω, ω	O, Ω	O, o
f_5	Ω, ω	Ω, ω	Ω, ω	Ω, ω	O, Ω

   (b) **(5 points)** Give an example of a function $f_6 : \mathbb{N} \to \mathbb{R}^+$ for which *none* of the four statements $f_i \in o(f_6)$, $f_i \in O(f_6)$, $f_i \in \omega(f_6)$, and $f_i \in \Omega(f_6)$ is true for any $i \in \{1,2,3,4,5\}$.

   One possible function for $f_6 : \mathbb{N} \to \mathbb{R}^+$ is,

$$f_6 = e^{n\sin(n)}.$$

   As n cycles through the four quadrants of a circle, $\sin(n)$ will oscillates in the range of $-1 \leq \sin(n) \leq 1$, which makes $n\sin(n)$ to become either very large negative or very large positive numbers as n grows large. Hence $e^{n\sin(n)}$ oscillates between extremely small positive real numbers and extremely large positive real numbers. Hence none of the statements, $f_i \in o(f_6)$, $f_i \in O(f_6)$, $f_i \in \omega(f_6)$, and $f_i \in \Omega(f_6)$ is true for any $i \in \{1,2,3,4,5\}$. □

4. (a) Solve the following recurrences exactly, and then prove your solutions are correct:

i. **(5 points)** $T(1) = 1,\ T(n) = T(n-1) + n^2 - n$

We get the following values for $T(n)$ for the first few values of n:

$$T(1) = 1, T(2) = 3, T(3) = 9, T(4) = 21, T(5) = 41, T(6) = 71, \ldots$$

Guessing a closed form expression, we claim $T(n) = \dfrac{n^3 - n}{3} + 1$.

Base case: For $n = 1$, $T(1) = 1$ and $\dfrac{n^3 - n}{3} + 1 = 1$. Therefore the closed form expression holds for $n = 1$.

Inductive hypothesis: Assume that the above closed form expression for $T(n)$ holds for all positive integers $n \leq k-1, k \in \mathbb{N}$, where k is arbitrarily positive integer with $k > 2$. Therefore, we have,

$$T(k-1) = \frac{(k-1)^3 - (k-1)}{3} + 1 = \frac{k^3 - 3k^2 + 2k}{3} + 1. \tag{3}$$

Inductive step: Assuming inductive hypothesis holds for $n \leq k-1$, using the given recurrence equation, for $n = k$,

$$\begin{aligned} T(k) &= T(k-1) + k^2 - k \\ &= \left(\frac{k^3 - 3k^2 + 2k}{3} + 1\right) + k^2 - k \qquad \text{(Substituting for } T(k-1) \text{ from Equation 3)} \\ &= \frac{k^3 - k}{3} + 1. \end{aligned}$$

Therefore, if the closed form expression for $T(n)$ is true for all positive integers $n \leq k-1$, then it is true for $n = k$, where k is an arbitrarily positive integer. Since it is true for the base case $n = 1$, it follows that $T(n) = \dfrac{n^3 - n}{3} + 1$ is true for all $n \in \mathbb{N}$. □

ii. **(5 points)** $T(1) = 1,\ T(n) = 3T(n-1) - n + 1$

We get the following values for $T(n)$ for the first few values of n:

$$T(1) = 1, T(2) = 2, T(3) = 4, T(4) = 9, T(5) = 23, T(6) = 64, \ldots$$

Guessing a closed form solution, we claim $T(n) = \dfrac{3^{n-1} + 2n + 1}{4}$.

Base case: For $n = 1$, $T(1) = 1$ and $\dfrac{3^{n-1} + 2n + 1}{4} = 1$. Therefore the closed form expression holds for $n = 1$.

Inductive hypothesis: Assume that the above closed form expression for $T(n)$ holds for all positive integers $n \leq k-1, k \in \mathbb{N}$, where k is arbitrarily positive integer with $k > 2$. Therefore,

we have,

$$T(k-1) = \frac{3^{(k-1)-1} + 2(k-1) + 1}{4} = \frac{3^{k-2} + 2k - 1}{4}. \tag{4}$$

Inductive step: Assuming inductive hypothesis holds for all positive integers $n \leq k-1$, using the given recurrence equation, for $n = k$,

$$\begin{aligned} T(k) &= 3T(k-1) - k + 1 \\ &= 3\left(\frac{3^{k-2} + 2k - 1}{4}\right) - k + 1 \qquad \text{(Substituting for } T(k-1) \text{ from Equation 4)} \\ &= \frac{3^{k-1} + 2k + 1}{4}. \end{aligned}$$

Therefore, if the closed form expression for $T(n)$ is true for all positive integers $n \leq k-1$, then it is true for $n = k$, where k is an arbitrarily positive integer. Since it is true for the base case $n = 1$, it follows that $T(n) = \frac{3^{n-1} + 2n + 1}{4}$ is true for all $n \in \mathbb{N}$. □

(Hint: Calculate values and guess the form of a solution. Then prove that your guess is correct by induction.)

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

i. **(3 points)** $T(n) = 9T(\lfloor n/3 \rfloor) + n^2 + 3n$

Using the master theorem, we note that $f(n) = n^2 + 3n$ is $\Theta(n^{\log_3 9}) = \Theta(n^2)$ (which says that $f(n)$ grows at the same rate as the number of leaves). Therefore,

$$T(n) = \Theta(n^{\log_3 9} \log n) = \Theta(n^2 \log n). \quad \square$$

ii. **(7 points)** $T(n) = 4T(\lfloor \sqrt{n} \rfloor) + \log n$ (Hint: it may help to apply a change of variable)

Let us use the substitution $\sqrt{n} = \frac{m}{2}$. Therefore, the above recurrence becomes,

$$T(n) = 4T\left(\left\lfloor \frac{m}{2} \right\rfloor\right) + \log\left(\frac{m^2}{4}\right).$$

Using the master theorem, we note that since $f(m) = \log\left(\frac{m^2}{4}\right)$ grows more slowly than $m^{\log_2 4} = m^2$, we find the total work is dominated by the work done at the leaves. So,

$$T(n) = \Theta(m^2) = \Theta(n). \quad \square$$

5. One of the simplest algorithms for sorting is BubbleSort — see code below.

Algorithm 1 BubbleSort

```
Input: A[0], ..., A[n-1]
for i = 0 to n-1 do
  for j = 0 to n-2 do
    if A[j] > A[j+1] then
      Swap A[j] and A[j+1]
    end if
  end for
end for
```

In this problem we will study the behavior of a twisted version of BubbleSort, described below.

Algorithm 2 TwistedBubbleSort

```
Input: A[0], ..., A[n-1]
for i = 0 to n-1 do
  for j = 0 to n-1 do
    if A[i] < A[j] then
      Swap A[i] and A[j]
    end if
  end for
end for
```

Your task is to prove that TwistedBubbleSort also correctly sorts every array. (While not necessary, you may assume for simplicity that the elements of A are all distinct.)

(a) (**2 points**) Explain in plain English why TwistedBubbleSort is different from BubbleSort. I.e., describe at least one difference in the swaps made by the two algorithms.

In the original BubbleSort algorithm, at the end of the i-th iteration of the outer for loop, the last i elements will be already in their correct positions in the array, and they occupy the indices from $n-i$ to $n-1$. On the other hand, the TwistedBubbleSort algorithm, at the end of the i-th iteration, places the largest element of the array in the i-th index of the array, and the elements from indices 0 to i will be already in their correct positions in the array.

(b) (**5 points**) Prove that after the i-th iteration of the outer loop of TwistedBubbleSort, the largest element of A is at the i-th index.

We shall prove this using induction on i, the number of iterations of the outer loop.

Base case: When $i = 0$, the inner for loop runs n times from $j = 0$ to $j = n-1$, each time comparing $A[0]$ and $A[j]$ and swapping $A[0]$ wih $A[j]$ only when $A[0] < A[j]$. If $A[0]$ is initially the largest element in A, it will remain at index 0. Otherwise, regardless of the position of the largest element in the original array, it will get swapped with $A[0]$. Therefore at the end of the n iterations of the inner loop, $A[0]$ will contain the largest element of A.

Inductive hypothesis: Assume that after the $i = k$-th iteration of the outer loop of TwistedBubbleSort, the largest element of A is at the k-th index, where $k > 0, k \in \mathbb{N}$. In other words, after the $i = k$-th iteration of the outer loop, $A[k]$ is the largest element in A.

Inductive step: For $i = k+1$, the inner for loop is executed n times from $j = 0$ to $j = n-1$, each time comparing $A[k+1]$ and $A[j]$ and swapping $A[k+1]$ wih $A[j]$ only when $A[k+1] < A[j]$. From the inductive hypothesis, $A[k]$ is the largest element in A from the previous iteration of $i = k$. Therefore, when $j = k$, we find that $A[k+1] < A[k]$, so both elements will be swapped. Since $A[k+1]$ now contains the largest element of A, $A[k+1] > A[j]$ for all subsequent values of j for the remainder of the inner for loop. Consequently there will be no more swaps of elements involving $A[k+1]$, and $A[k+1]$ will remain the largest element of the array at the end of the $i = (k+1)$-st iteration.

Since the claim is true for $i = 0$, and as we have shown that it is true for $i = k+1$ assuming that the inductive hypothesis is true for $i = k$, it follows that the claim is true for all i from 0 to $n-1$. □

(c) (**10 points**) Prove that after the i-th iteration of the outer loop of TwistedBubbleSort, indices 0 to i of the array are sorted.

We shall prove this using strong induction on i, the number of iterations of the outer loop.

Base case: At the end of the iteration $i = 0$, the largest element of A will be found in $A[0]$, which is trivially sorted.

Inductive hypothesis: Assume that for all $i = 0, \ldots, i-1$, after the $(i-1)$-st iteration of the outer loop of TwistedBubbleSort, the subarray $A[0 \ldots i-1]$ is sorted.

Inductive step: From the results of Part (b), we know that after the $(i-1)$-st iteration of the outer loop, $A[i-1]$ will be the largest element of the subarray $A[0 \ldots i-1]$. We also know from the inductive hypothesis that at the beginning of the i-th iteration of the outer loop, $A[0 \ldots i-1]$ consists of the elements in sorted order with $A[i-1]$ being the largest element in A.

During the i-th iteration of the outer loop, the inner loop runs from $j = 0$ to $j = n-1$, comparing each $A[j]$ with $A[i]$ to see if $A[i] < A[j]$. There are three cases to consider:

(1) $0 \le j < i-1$: $A[0 \ldots i-1]$ consists of the elements in sorted order. There are two sub-cases to consider as follows:
   (a) $A[i]$ is larger than or equal to all the elements in the subarray $A[0 \ldots i-1]$. In other words, $A[i] \ge A[j]$, for $0 \le j < i-1$. In this case, no swaps will happen and therefore, the sorted property of the subarray $A[0 \ldots i-1]$ will be preserved.
   (b) There are one or more elements in the subarray $A[0 \ldots i-1]$ that are larger than $A[i]$. Consider the sequence of elements, $\ldots, < a < b <, \ldots$ in the subarray $A[0 \ldots i-1]$, where a is the first element in the subarray during the inner loop's iteration such that $a > A[i]$. Let $A[i] = k$. So, the elements a and k will be swapped and $A[i]$ will now be a and k will move to the index of a. Since $A[i]$ is now a, when the inner loop increments from j to $j+1$, again $A[i]$ (which is now a) will be compared with b. Since $a < b$, they will be swapped. This process repeats, resulting in the sequence of elements $\ldots, k < a < b <, \ldots$, thereby preserving the sorted property of the subarray $A[0 \ldots i-1]$.

(2) $j = i-1$: When $j = i-1$, $A[i] < A[j]$ because, from the results of Part (b), $A[j] = A[i-1]$ is the largest element of A. Hence, we swap $A[j]$ with $A[i]$, extending the length of the sorted subarray by 1. Therefore, the new subarray $A[0 \ldots i]$ is a sorted subarray with the last element $A[i]$ being the largest element in A.

(3) $i \le j \le n-1$: For $i \le j \le n-1$, the check $A[i] < A[j]$ fails as $A[i]$ is now the largest element in A, and consequently there will be no more swapping of elements during the current iteration of the outer loop. Hence the subarray $A[0 \ldots i]$ remains sorted.

Hence, at the end of the i-th iteration of the outer loop, the inductive hypothesis holds and the subarray $A[0 \dots i]$ remains sorted. Since the inductive hypothesis is true for $i = 0$, and assuming it is true for $i = 0, \dots, i-1$, it is true for i. Therefore, the subarray $A[0 \dots i]$ is sorted at the end of the i-th iteration for all $0 \leq i \leq n-1$. □

We also note that when $i = n-1$, the outer loop terminates with all the elements in A in sorted order. □

6. **(0 points, optional)**[1] InsertionSort is a simple sorting algorithm that works as follows on input $A[0]$, $\ldots$, $A[n-1]$.

Algorithm 3 InsertionSort

```
Input: A[0],...,A[n-1]
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.