CS 124 Homework 1: Spring 2024

Collaborators: Office hours

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 not 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.

I'll prove the claim by considering two exhaustive cases relating A and B. Assume for simplicity that $A \geq B$. If not, Euclid's algorithm will swap them and resume that invariant.

Consider when $A \geq 2B$. By the definition above, observe the following:

$$\begin{aligned} & A' + B' && \text{Initial} \\ = B + A \mod B & && \text{Substitute by definitions} \\ \leq B + B \mod B & && \text{Because } A \mod B < B \\ = 2B & && \text{Simplify} \end{aligned}$$

Similarly, because $A \geq 2B$ by the case assumption, we know $A + B \geq 2B + B = 3B$. Join these into an inequality and simplify:

$$\begin{aligned} A' + B' \leq 2B \leq 3B \leq A + B & && \text{Inequalities determined above} \\ A' + B' \leq 2B \leq \frac{2}{3} \cdot 3B \leq \frac{2}{3}(A + B) & && \text{Scale right two terms} \\ A' + B' \leq 2B = 2B \leq \frac{2}{3}(A + B) & && \text{Simplify} \\ A' + B' \leq \frac{2}{3}(A + B) & && \text{Remove intermediate terms} \end{aligned}$$

Thus, when $A \geq 2B$, it holds that $A' + B' \leq \frac{2}{3}(A + B)$.

Next, consider then $A < 2B$. Because A is less than $2B$ from the case assertion and $A \geq B$ from the simpifying assumption, it must hold that $A \mod B = A - B$. Also, because $A < 2B$, it's also true that $\frac{1}{2}A < B$ and $\frac{1}{3}A < \frac{2}{3}B$. Apply these observations:

$$\begin{aligned} & A' + B' && \text{Initial} \\ = B + A \mod B & && \text{Substitute by definitions} \\ = B + A - B & && \text{Asserted above} \\ = A & && \text{Simplify} \\ = \frac{2}{3}A + \frac{1}{3}A & && \text{Break apart } A \\ < \frac{2}{3}A + \frac{2}{3}B & && \text{Asserted above} \\ = \frac{2}{3}(A + B) & && \text{Factor} \end{aligned}$$

Thus, when $A < 2B$, it also holds that $A' + B' \leq \frac{2}{3}(A + B)$.

These cases together prove that $A' + B' \leq \frac{2}{3}(A + B)$.

Next, consider why Euclid's algorithm starting from (A, B) must conclude after at most $\log_{3/2}(A + B)$ steps.

Having concluded that recursive inputs to Euclid's algorithm shrink by at least $\frac{2}{3}$ per iteration, we expect the at step i the sum $A_i + B_i \leq (\frac{2}{3})^i (A+B)$ where $A_i + B_i > 1$. Because $A_k + B_k \leq 1$ indicates the base case, solve for the maximum i for which that point is reached:

$$
\begin{aligned}
1 &= (\frac{2}{3})^i (A+B) && \text{Initial} \\
\log_{\frac{3}{2}} 1 &= \log_{\frac{3}{2}} ((\frac{2}{3})^i (A+B)) && \text{Apply log} \\
0 &= \log_{\frac{3}{2}} (\frac{2}{3})^i + \log_{\frac{3}{2}} (A+B)) && \text{Log rule for multiplication} \\
0 &= i \log_{\frac{3}{2}} (\frac{2}{3}) + \log_{\frac{3}{2}} (A+B)) && \text{Log rule for exponents} \\
0 &= -i + \log_{\frac{3}{2}} (A+B)) && \text{Simplify} \\
i &= \log_{\frac{3}{2}} (A+B)) && \text{Add } i
\end{aligned}
$$

Thus, the value i representing the maximum bound for the number of iterations of Euclid's algorithm on inputs A and B is $\log_{\frac{3}{2}} (A+B))$.

(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

Derive L':

$$
\begin{aligned}
L' &= g(A' + B') && \text{Initial} \\
&= A' + \beta B' && \text{Definition of } g \\
&= B + \beta (A \mod B) && \text{From transition relation} \\
&= B + \beta ((QB + R) \mod B) && \text{Definition above} \\
&= B + \beta R && \text{Definition of remainder}
\end{aligned}
$$

Thus, $L' = B + \beta R$.

Derive L. Before doing so, notice that because $A \geq B$ and $A + B \geq 1$, a minimum bound for Q can always be defined as $Q \geq 1$.

$$
\begin{aligned}
L &= g(A+B) && \text{Initial} \\
&= A + \beta B && \text{Definition of } g \\
&= QB + R + \beta B && \text{Definition above} \\
&\geq B + R + \beta B && \text{Asserted above} \\
&= B(1+\beta) + R && \text{Distribute}
\end{aligned}
$$

Thus, $L = B(1+\beta) + R$.

(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)

Consider the ratio:

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

The goal is to express this ratio in terms of some constant c. That yields this equation: $B + \beta R = c(B + B\beta + R)$. It happens that β itself is a suitable candidate for c. Simplify this equation where $c = \beta$. The derivation below assumes that B is nonzero.

$$\begin{aligned}
B + \beta R &= c(B + \beta B + R) && \text{Initial} \\
B + \beta R &= \beta B + \beta^2 B + \beta R && \text{Substitue } \beta = c \\
B &= \beta B + \beta^2 B && \text{Subtract } \beta R \\
0 &= B\beta^2 + B\beta - B && \text{Rearrange into polynomial} \\
0 &= \beta^2 + \beta - 1 && \text{Divide by } B \\
0 &= \frac{-1 \pm \sqrt{1 - 4(1)(-1)}}{2} && \text{Quadratic formula} \\
0 &= \frac{-1 \pm \sqrt{5}}{2} && \text{Simplify}
\end{aligned}$$

Because an algorithm's lower bound is logically never negative, choose the positive variant for $\beta = \frac{-1+\sqrt{5}}{2}$.

Prove that this assignment of β satisfies the claim by demonstrating that $\frac{L'}{L} = \beta$ for any values of B and R:

$$\begin{aligned}
\frac{L'}{L} &= \beta && \text{Initial} \\
\frac{B + \beta R}{B(1+\beta) + R} &= \beta && \text{Defined above} \\
\frac{B + (\frac{-1+\sqrt{5}}{2})R}{B(1 + (\frac{-1+\sqrt{5}}{2})) + R} &= \frac{-1+\sqrt{5}}{2} && \text{Definition of } \beta \\
2(B + \frac{-1}{2}R + \frac{\sqrt{5}}{2}R) &= (-1+\sqrt{5})(B - \frac{1}{2}B + \frac{\sqrt{5}}{2}B + R) && \text{Cross multiply} \\
2B - R + \sqrt{5}R &= -B + \frac{1}{2}B - \frac{\sqrt{5}}{2}B - R + \sqrt{5}B - \frac{\sqrt{5}}{2}B + \frac{5}{2}B + \sqrt{5}R && \text{Distribute} \\
2B - R + \sqrt{5}R &= -B + 3B - sqrt5B + \sqrt{5}B - R + \sqrt{5}R && \text{Simplify} \\
2B - R + \sqrt{5}R &= 2B - R + \sqrt{5}R && \text{Simplify}
\end{aligned}$$

Because the equality above holds, it must be true that $\frac{L'}{L} = \beta$ for any values of b and R.

(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).

Given that $\frac{L'}{L}$ gives the ratio of inputs from one iteration to the next, we wish to know how that ratio progresses. Consider the following: $L' = \beta L$. Similarly, $L'' = \beta\beta L$. Etc. Thus, $L^i = \beta^i L = \beta^i(B + \beta B + R)$. Solve this for i where $L^i = 1$, the minimum input size:

$$1 = \beta^i(B + \beta B + R) \qquad \text{Initial}$$

$$\frac{1}{B + \beta B + R} = \beta^i \qquad \text{Divide}$$

$$\log_\beta \frac{1}{B + \beta B + R} = log_\beta \beta^i \qquad \text{Apply log}$$

$$\log_{\frac{-1+\sqrt{5}}{2}} \frac{1}{B + \frac{-1+\sqrt{5}}{2} B + A \mod B} = i \qquad \text{Expand beta, R}$$

Thus, an improved upper bound on the number of steps in Euclid's Algorithm starting from (A, B) is $\log_{\frac{-1+\sqrt{5}}{2}} \frac{1}{B + \frac{-1+\sqrt{5}}{2} B + A \mod B}$.

(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.

Consider this sequence:

- A = 21, B = 13
- A = 13, B = 8
- A = 8, B = 5
- A = 5, B = 3
- A = 3, B = 2
- A = 2, B = 1
- A = 1, B = 0

In increasing order, that's 0, 1, 2, 3, 5, 8, 13, 21, etc. The bound stated above is essentially a logarithmic inversion of the Fibonacci numbers. That relationship is revealed by the value of β and it's closeness to the Golden Number. Further, the Fibonacci numbers collapse almost perfectly within this algorithm, making them a tight bound to the algorithm.

(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.

I ran the three algorithms in release-mode Rust using $v1.75.0$ on an M1 Macbook Pro. The recursive and matrix algorithms are identical to the lecture notes. I modified the iterative algorithm to just keep a pointer to the previous two numbers to avoid all the unnecessary heap allocations in a vector. So, it might be a bit faster than otherwise. All three algorithms were tested against each-other for correctness in the range $n = 1$ to $n = 20$.

My benchmark was essentially this:

Algorithm 1 Test fib

```
for 16 seconds do
  for n = 1 do
    fibonacci(n)
    n += 1
  end for
end for
```

It's a race to see how high of n can be reached in less than 16 seconds (the last run that went over was not counted). After three passes, these were my average results:

- Recursive: n = 47
- Iterative: n = 225,248
- Matrix: n = 196,768,930

Based on these benchmarks, the matrix method appears much faster.

Edit: After seeing part c, I realize my test might not look very creative. If you'd like actual times, computing the 32nd Fibonacci number takes the following times:

- Recursive: 15.812041ms
- Iterative: 167ns
- Matrix: 83ns

Edit 2: Part c also required writing my own matrix exponentiation (to accomodate the modding). My own multiplication uses basic n^3 multiplication and doesn't use SIMD or any serious optimizations, so I was curious to see how different it was from the previously-used Linear Algebra library. Here are the new results (Edit 3: these were taken before I added timeouts mentioned in part c):

- Recursive: n = 47
- Iterative: n = 202,663
- Matrix (my implementation): n = 55,309,376
- Matrix (nalgebra crate): n = 198,068,230

And new times. My matrix implementation and iterative are the same here, but the difference is at the nanosecond level, so asymptotic behavior is much more effectively approximated by the 16 second tests.

- Recursive: 15.025ms
- Iterative: 125ns
- Matrix (my implementation): n = 125ns

- Matrix (nalgebra crate): n = 83ns

(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.)

(c) The **47th** Fibonacci number is 2,971,215,073, which exceeds $2^{31} = 2,147,483,648$.

(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.

I changed my code substantially this problem. Now, the algorithms perform modulo arithmetic on intermediate Fibonacci numbers, and they carry timers so that long-running functions can interrupt themselves. Polling the timers seems to meaningfully slow down each recursive frame and iterative pass. I also wrote my own matrix exponentiation (instead of the graphics library from part *a*), and I added a function that binary-searches for the greatest input to an algorithm that halts within 60 seconds.

My matrix exponentiation algorithm calculates the (mod 2^{16}) Fibonacci number for Rust's u64::MAX = $2^{64} - 1$ = 18,446,744,073,709,551,615 value in much less than a second. This surprised me a lot. I wrote a test to ensure recursive, iterative, and matrix all produce the same values up to the 20th Fibonacci number and a similar test for just matrix and iterative up to the 2^{12}th Fibonacci number. I really don't think my matrix implementation is incorrect. Because exponentiation is essentially $\lg n$ and $\lg 2^{64} = 64$, my conclusion is that computing the maximum non-overflow Fibonacci number on my computer in microseconds when a low Fibonacci number takes nanoseconds might make sense.

- Recursive: Computed $k = 45$ in less than one minute. (This is slower than the 16 seconds test from *a*, but checking the timer at every frame had a big impact)
- Iterative: Computed $k = 2,464,538,623$ in less than one minute.
- Matrix: Computed $k = 18446744073709551615$ (maximum unsigned 64-bit integer) in 1.208 microseconds.

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

- $f_i \in O(f_1), f_i \in \Omega(f_1)$
- $f_1 \in \omega(f_2), f_1 \in \Omega(f_2)$
- $f_1 \in \omega(f_3), f_1 \in \Omega(f_3)$
- $f_1 \in O(f_4), f_1 \in \Omega(f_4)$
- $f_1 \in o(f_5), f_1 \in O(f_5)$

• *ii*
- $f_2 \in O(f_2), f_2 \in \Omega(f_2)$
- $f_2 \in \omega(f_3), f_2 \in \Omega(f_3)$
- $f_2 \in o(f_4), f_2 \in O(f_4)$
- $f_2 \in o(f_5), f_2 \in O(f_5)$

• *iii*
- $f_3 \in O(f_3), f_3 \in \Omega(f_3)$
- $f_3 \in o(f_4), f_3 \in O(f_4)$
- $f_3 \in o(f_5), f_3 \in O(f_5)$

• *iv*
- $f_4 \in O(f_4), f_4 \in \Omega(f_4)$
- $f_4 \in o(f_5), f_4 \in O(f_5)$

• *v*
- $f_5 \in O(f_5), f_5 \in \Omega(f_5)$

(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 $i \in \{1,2,3,4,5\}$.

$f_6 = 2^{\frac{1}{\sin n}}$. My reasoning is that this function oscillates wildly between zero and infinity as it increases and thus doesn't have an asymptotically clear bound. That makes it impossible to compare using order notation.

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

Derivation:

i) $T(1) = C$

$T(2) = T(1) + 2^2 - 2 = C + 4 - 2 = 2 + C$ | $k=2,\ (k^2-k) = 2$ | $1 = \frac{k+1}{3} = \frac{3}{3}$

$T(3) = T(2) + 3^2 - 3 = 2 + C + 9 - 3 = 8 + C$ | $k=3,\ (k^2-k) = 6$ | $\frac{4}{3} = \frac{k+1}{3} = \frac{4}{3}$

$T(4) = T(3) + 4^2 - 4 = 8 + C + 16 - 4 = 20 + C$ | $k=4,\ (k^2-k) = 12$ | $\frac{5}{3} = \frac{k+1}{5} = \frac{5}{3}$

$T(5) = T(4) + 5^2 - 5 = 20 + C + 25 - 5 = 40 + C$ | $k=5,\ (k^2-k) = 20$ | $2 = \frac{k+1}{3} = \frac{6}{3}$

$T(6) = T(5) + 6^2 - 6 = 40 + C + 36 - 6 = 70 + C$ | $k=6,\ (k^2-k) = 30$ | $\frac{7}{3} = \frac{k+1}{3} = \frac{7}{3}$

$(k^2 - k)$ $\quad$ $k(k-1)\left(\frac{k+1}{3}\right)$

$k(k-1)$ $\quad$ $\frac{1}{3}k(k-1)(k+1)$

$\frac{1}{3}k(k^2-1) + C$

Because $C=1$, $f(k) = \frac{1}{3}k(k^2-1) + 1$

Claim: the closed form is $f(n) = \frac{1}{3}n(n^2-1)+1$.

Proof by induction on n:
Base case: For $n = 1$, $T(1) = 1$ is given by the definition, and $f(1) = \frac{1}{3}(1)(1^2-1)+1 = 1$. So, the base case holds.

Inductive hypothesis: Assume the claim is true for all k from $1 \le k \le n$.

Inductive step: Consider $T(n+1)$:

$$\begin{aligned} T(n+1) &= T(n+1-1) + (n+1)^2 - (n-1) && \text{Initial} \\ &= T(n) + n^2 + n && \text{Simplify} \\ &= \frac{1}{3}n(n^2-1) + 1 + n^2 + n && \text{Apply inductive hypothesis} \\ &= \frac{1}{3}n^3 + n^2 + \frac{2}{3}n + 1 && \text{Simplify} \end{aligned}$$

Similarly, consider $f(n+1)$:

$$\begin{aligned} f(n+1) &= \frac{1}{3}(n+1)((n+1)^2-1)+1 && \text{Initial} \\ &= (\frac{1}{3}n + \frac{1}{3})(n^2+2n)+1 && \text{Distribute and FOIL} \\ &= \frac{1}{3}n^3 + n^2 + \frac{2}{3}n + 1 && \text{Multiply and combine} \end{aligned}$$

Thus, $T(n+1) = f(n+1)$, proving the inductive step. By induction, the closed form of the recurrence must be correct.

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

Derivation:

ii) $T(1) = C$

$T(2) = 3T(1) - 2 + 1 = 3C - 2 + 1 = 3C - 1 = 2$

$T(3) = 3T(2) - 3 + 1 = 3(3C-1) - 2 = 9C - 3 - 2 = 9C - 5 = 4$

$T(4) = 3T(3) - 4 + 1 = 3(9C-5) - 3 = 27C - 15 - 3 = 27C - 18 = 9$

$T(5) = 3T(4) - 5 + 1 = 3(27C-18) - 4 = 81C - 54 - 4 = 81C - 58 = 23$

$T(1) = C$

$T(2) = 3T(1) - 2 + 1 = 3C - 2 + 1$

$T(3) = 3T(2) - 3 + 1 = 3(3C - 2 + 1) - 3 + 1$

$T(4) = 3T(3) - 4 + 1 = 3(3(3C - 2 + 1) - 3 + 1) - 4 + 1$

$T(5) = 3T(4) - 5 + 1 = 3(3(3(3C - 2 + 1) - 3 + 1) - 4 + 1) - 5 + 1$

$3^{k-1}C + \cancel{3^3} + \cancel{3^2} + 3 + 1 - \cancel{2 \cdot 3^3} = \cancel{3 \cdot 3^2} - \cancel{4 \cdot 3} - 5$

$+ 3^3 - 2 \cdot 3^3 + 3^2 - 3 \cdot 3^2 + 3 - 4 \cdot 3 + 3^0 - 5 \cdot 3^0$

$-3^3 - 2 \cdot 3^2 - 3 \cdot 3^1 - 4 \cdot 3^0$

$3^{k-1}C + \sum_{i=1}^{k-1} -(k-i) \cdot 3^{i-1} = 3^{k-1}C - \sum_{i=1}^{k-1} k3^{i-1} - i3^{i-1} = 3^{k-1}C - k\left[\sum_{i=1}^{k-1} 3^{i-1}\right] + \left[\sum_{i=1}^{k-1} i3^{i-1}\right]$

Geometric series: $S = a + ar + ar^2 + \ldots + ar^{n-1} = \frac{a(r^n - 1)}{r - 1}$

$\sum_{i=1}^{k-1} 3^{i-1} = 3^0 + 3^1 + 3^2 + \ldots + 3^{k-1} = \frac{3^{k-1} - 1}{3 - 1} = \frac{3^{k-1} - 1}{2}$

$3^{k-1}C - k\frac{3^{k-1} - 1}{2} + \left[\sum_{i=1}^{k-1} i3^{i-1}\right]$

Use derivative variant of closed form $f(x) = \frac{x^n - 1}{x - 1}$, $f'(x) = \frac{d}{dx}\left(\frac{x^n - 1}{x - 1}\right) = \frac{-nx^n + nx^{n+1} - x^n + 1}{(x-1)^2}$

For $x = 3$, $\sum_{i=1}^{k-1} i3^{i-1} = \frac{2 \cdot 3^k k - 3^{k+1} + 3}{12}$

$3^{k-1}C - k\frac{3^{k-1} - 1}{2} + \frac{2 \cdot 3^k k - 3^{k+1} + 3}{12} = \frac{4 \cdot 3^k C}{12} - \frac{2 \cdot k 3^k - 6k}{12} + \frac{2 \cdot 3^k k - 3^{k+1} + 3}{12}$

$\frac{\cancel{4 \cdot 3^k C} = \cancel{2 \cdot k 3^k} + 6k + \cancel{2 \cdot 3^k k} - 3^{k+1} + 3}{12} = \frac{4C \cdot 3^k - 3^{k+1} + 6k + 3}{12}$

Given $C = 1$, closed form is $\frac{-3^{k+1} + 4 \cdot 3^k + 6k + 3}{12}$

Claim: the closed form is $f(n) = \frac{-3^{n+1} + 4 \cdot 3^n + 6n + 3}{12}$.

Proof by induction on n:

Base case: For $n = 1$, $T(1) = 1$ is given by the definition, and $f(1) = \frac{-3^2 + 4 \cdot 3 + 6 + 3}{12} = \frac{-9 + 21}{12} = 1$. So, the base case holds.

Inductive hypothesis: Assume the claim is true for all k from $1 \leq k \leq n$.

Inductive step: Consider $T(n+1)$:

$$\begin{aligned}
T(n+1) &= 3T(n+1-1) - (n+1) + 1 && \text{Initial} \\
&= 3T(n) - n && \text{Simplify} \\
&= 3 \cdot \frac{-3^{n+1} + 4 \cdot 3^n + 6n + 3}{12} - n && \text{Apply inductive hypothesis} \\
&= \frac{-3 \cdot 3^{n+1} + 4 \cdot 3 \cdot 3^n + 18n + 9 - 12n}{12} && \text{Distribute, join terms} \\
&= \frac{-3^{n+2} + 4 \cdot 3^{n+1} + 6n + 9}{12} && \text{Merge 3s into exponents}
\end{aligned}$$

Similarly, consider $f(n+1)$:

$$\begin{aligned}
f(n+1) &= \frac{-3^{n+2} + 4 \cdot 3^{n+1} + 6(n+1) + 3}{12} && \text{Initial} \\
&= \frac{-3^{n+2} + 4 \cdot 3^{n+1} + 6n + 9}{12} && \text{Distribute}
\end{aligned}$$

Thus, $T(n+1) = f(n+1)$, proving the inductive step. By induction, the closed form of the recurrence must be correct.

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

$T(n) = \Theta(n^2 \cdot \lg n)$

Proof by case II of the Master Theorem where $a = 9$, $b = 3$, and $f(n) = n^2 + 3n$:

$O(n^{log_3 9} \cdot (\lg n)^k)$ where $k = 0$ simplifies to $O(n^2)$. Because $f(n) = n^2 + 3n = O(n^2)$, the upper bound in the second case of the Master Theorem is satisfied.

$\Omega(n^{log_3 9} \cdot (\lg n)^k)$ similarly simplifies to $\Omega(n^2)$ when $k = 0$. Because $f(n) = n^2 + 3n = \Omega(n^2)$, the lower bound in the second case of the Master Theorem is satisfied. Assert from *CLRS* page 102 that the asymptotic bounds provided by the Master Theorem are not affected by floor or ceiling rounding, so this bound also applies to the given equation.

Thus, by the second case of the Master Theorem, assert that $T(n) = \Theta(n^{\log_3 9} \cdot (\lg n)^1)$, which simplifies to $T(n) = \Theta(n^2 \cdot \lg n)$.

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

$T(n) = \Theta((\lg n)^2)$

Because proving this bound requires a change of variable, my analysis determines an upper bound (and the bound itself) using the Master Theorem and proves the lower bound using substitution.

In both parts, let $k = \lg n$.

Upper bound: Let t be T without any floor function. Due to the floor function, a complete upper bound on this recurrence is $t(n) = 4t(\sqrt{n}) + \lg n$. Because $2^k = n$, let $t(2^k) = 4t(\sqrt{2^k}) + \lg 2^k = 4t(2^{\frac{1}{2}k}) + k$.

Let $R(k) = t(2^k)$. By this definition, $R(\lg n) = t(2^{\lg n}) = t(n)$. Similarly, $R(k) = 4R(\frac{1}{2}k) + k$.

Consider case I of the Master Theorem on $R(k)$ where $a = 4$, $b = 2$, $f(k) = k$, and $\lg 4 = 2$. Because $f(k) = k = O(k^{2-\epsilon})$ where $\epsilon \leq 1$, the case is satisfied and the Master Theorem asserts that $R(k) = \Theta(k^2)$.

De-substituting this yields the following:

- $R(k) = \Theta(k^2)$
- $R(\lg n) = \Theta((\lg n)^2)$
- $T(n) = \Theta((\lg n)^2)$

Thus, by the Master Theorem and substitutions above, assert that the upper bound on $T(n)$ indicated is $O((\lg n)^2)$.

Lower bound: Let $R(k) = T(2^k)$. Using the same definition of k, we have $T(n) = 4T(\lfloor\sqrt{n}\rfloor) + \log n$ and $T(k) = 4T(\lfloor 2^{\frac{1}{2}k}\rfloor) + k$. This yields $R(k) = 4R(\lg(\lfloor 2^{\frac{1}{2}k}\rfloor)) + k$.

Assert that $\frac{k}{2} - 1 \leq \lg(\lfloor 2^{\frac{1}{2}k}\rfloor)$ for all $k > 0$.

Under that assertion, $R(k) = 4R(\lg(\lfloor 2^{\frac{1}{2}k}\rfloor)) + k \geq H(k)$ where $H(k) = 4R(\frac{k}{2} - 1) + k$ for all $k > 0$.

I'll then prove by substitution that $H(k) = \Omega(k^2)$ for all k. Definition of Ω specifies that $H(k)$ must be greater than $c \cdot k^2$ for $c > 0$ and sufficiently large $k_0 \leq k$. Here, let $c = 0.1$ and $k_0 = 1$. Prove by induction on k:

Base case: At $k = 0$, $H(1) = 4 \cdot H(\frac{1}{2} - 1) + 1 \geq 0.1(1^2) = 0.1$, assuming the output of H is always nonnegative.

Inductive hypothesis: Assume that $ck^2 \leq H(k)$ for all $k_0 \leq k$.

Inductive step: Consider the following.

$$
\begin{aligned}
H(k+1) &= 4H(\frac{k+1}{2} - 1) + k + 1 && \text{Initial} \\
&= 4H(\frac{1}{2}k - \frac{1}{2}) + k + 1 && \text{Simplify fraction} \\
&\geq 4c \cdot (\frac{1}{2}k - \frac{1}{2})^2 + k + 1 && \text{By inductive hypothesis} \\
&= 4c \cdot (\frac{1}{4}k^2 - \frac{1}{2}k + \frac{1}{4}) + k + 1 && \text{FOIL} \\
&= ck^2 - 2ck + c + k + 1 && \text{Distribute} \\
&= 0.1k^2 + 0.8k + 1.1 && \text{Substitute } c = 0.1 \\
&\geq c(k+1)^2 && \text{Greater than assumed bound} \\
&= 0.1k^2 + 0.2k + 0.1 && \text{Expanded for clarity}
\end{aligned}
$$

Because $c(k+1)^2 \leq H(k+1)$, the claim holds by induction. This proves that $H(k) = \Omega(k^2)$. Because $H(k) \leq R(k)$ for all $k \geq 1$, it must also hold that $R(k) = \Omega(k^2)$. Substituting in terms of n yields $R(\lg n) = \Omega((\lg n)^2)$. Finally, substituting functions produces $T(n) = \Omega((\lg n)^2)$.

Conclusion: The two cases above prove that the recurrence $T(n)$ is both $O((\lg n)^2)$ and $\Omega((\lg n)^2)$, proving that $T(n) = \Theta((\lg n)^2)$.

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

Algorithm 2 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 3 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.

Bubble sort operates by comparing and swapping neighbors. The upper index i exists to ensure enough such comparisons are made. In contrast, TwistedBubbleSort chooses an index i for a cycle of iteration and compares all numbers in the array to the value currently at i. At iteration i, it only performs swaps involving i, unlike BubbleSort, which performs operations on j and j's neighbor.

(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.

Let i be any valid index in the array. The proof below uses induction on j to prove that, after the inner j-loop completes, the largest element of A is in $A[i]$. Assume for simplicity that all elements are distinct and that the syntax $A[m..=n]$ denotes an inclusive subarray.

Below is a proof that after finishing step j, $0 \le j \le n-1$ of the j-loop, the value at $A[i]$ is greater than every value in $A[0..=j]$ excluding $A[i]$.

Base case: $j = 0$. If $A[i] < A[0]$, the value at $A[0]$ is swapped into $A[i]$. This results in $A[i]$ being greater than every element in $A[0..=0]$ excluding $A[i]$.

Inductive hypothesis: After completing step $j \ge 0$, assume the value at $A[i]$ is greater than all elements from $A[0..=j]$ excluding $A[i]$.

Inductive step: Consider step $j+1$ by cases.

- If $A[j+1] = A[i]$ (by the simplification assumption, only true if $i = j+1$), then the algorithm performs a trivial swap and, by the inductive hypothesis, the subarray $A[0..=j+1]$ excluding $A[i]$ still has no elements greater than $A[i]$.
- If $A[j+1] < A[i]$, then the subarray is now one element larger. By the inductive hypothesis, there are no elements in $A[0..=j]$ excluding $A[i]$ greater than $A[i]$, and the element at $A[j+1]$ is also not greater than $A[i]$. Thus, every element in the expanded subarray $A[0..=j+1]$ excluding $A[i]$ is still less than $A[i]$.
- If $A[j+1] > A[i]$, then the elements at index $j+1$ and i are swapped. After they're swapped, $A[j+1] < A[i]$, and the case above now applies.

Across all three cases, after step $j+1$, the value at $A[i]$ is still greater than every element in the subarray $A[0..=j+1]$ excluding $A[i]$. This proves the claim true by induction.

By the inductive proof above, for any index i, after iteration $j = n-1$, it must be true that the value at $A[i]$ is greater than every element in the (sub)array $A[0..=n-1]$, which is equivalent to saying the largest element of A is at index i.

(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.

Below is a proof by induction on i that the subarray $A[0..=i]$ is in sorted order after the ith iteration of the outer loop in TwistedBubbleSort. The subarray syntax is defined above, and sorting is least to greatest. I'm again making the simplifying assumption that every element in the array is distinct.

Base case: $i = 0$. Any number in the subarray $A[0..=0]$ is trivially sorted, so the claim is true for $i = 0$.

Inductive hypothesis: At index $0 \le i \le n-1$, the subarray $A[0..=i]$ is sorted from least to greatest.

Inductive step: At index $i+1$, consider the inner loop. By the proof above (question 5b), by the end of the j-loop, the greatest element in the array is at index $i+1$. Because $A[0..=i]$ is already in sorted order (inductive hypothesis) and $A[i+1]$ must be greater than all in $A[0..=i]$, it must be true that the expanded subarray $A[0..=i+1]$ is also now also in sorted order.

By induction, it must then be true that after the ith iteration, indices 0 to i are 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 4 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.