CS 124 Homework 1: Spring 2024

Collaborators: Ryan Jiang, Ossimi Ziv, Alex Gong, Charlie Chen, Sophie Zhu, Denny Cao, Kathryn Harper

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.

Proof. Let $A = QB + R$, where Q is the integer quotient and R is the integer remainder when you divide A by B. Since A is defined to be greater than B, it follows by properties of division that $Q \geq 1$. Moreover, we know that $R \leq B$ by the properties of a remainder. Basic arithmetic operations yield $0 \leq B - R$ and $0 \leq \frac{1}{3}(B-R)$. Then, if we add $B+R$ on both sides of the inequality, we have $B + R \leq \frac{4}{3}B + \frac{2}{3}R$. Splitting the $\frac{4}{3}B$ term, we are left with $B + R \leq \frac{2}{3}B + \frac{2}{3}(B+R)$. Upon further examination, we can then see that $\frac{2}{3}(B+R) \leq \frac{2}{3}(QB+R)$, since $Q \geq 1$. Furthermore, since $A = QB + R$, it follows by substitution that $\frac{2}{3}(B+R) \leq \frac{2}{3}A$. With this fact, we now have $\frac{2}{3}B + \frac{2}{3}(B+R) \leq \frac{2}{3}B + \frac{2}{3}A = \frac{2}{3}(A+B)$. Using the definition of a remainder and the given definitions of A', B', we know $A' = B$ and $B' = A \mod B = R$. Thus from $B + R \leq \frac{2}{3}B + \frac{2}{3}(B+R)$, it follows that $A' + B' \leq \frac{2}{3}B + \frac{2}{3}(B+R) \leq \frac{2}{3}(A+B)$. Therefore, by the transitive property of inequalities, we have proved that $A' + B' \leq \frac{2}{3}(A+B)$.

Now, suppose that Euclid's algorithm makes p recursive calls. That is, at the final recursive call we must have $A_p > 0$ and $B_p = 0$. Thus, $A_p + B_p \geq 1$. Furthermore, we have proved above that $A' + B' \leq \frac{2}{3}(A+B)$. It follows that $A_{i+1} + B_{i+1} \leq \frac{2}{3}(A_i + B_i) \leq (\frac{2}{3})(\frac{2}{3})(A_{i-1} + B_{i-1}) \leq \ldots \leq (\frac{2}{3})^{i+1}(A+B)$ for some integer i. We can then determine that $1 \leq A_p + B_p \leq (\frac{2}{3})^p(A+B)$. If we take logs, we get $0 = \log_2(1) \leq p\log_2(2/3) + \log_2(A+B)$. If we use the fact that $\log_2(2/3) = -\log_2(3/2)$, we can rearrange this inequality such that $p \leq \frac{\log_2(A+B)}{\log_2(3/2)}$. Simplifying, we conclude that $p \leq \log_{3/2}(A+B)$. Thus, we have proved that Euclid's algorithm will terminate after at most $\log_{3/2}(A+B)$ recursive calls. □

(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

Answer: Let $g(A,B) = A + \beta B$ for $\beta > 0$. Let $A = QB + R$, where R is the remainder and Q is the quotient when dividing A by B. Since A is defined to be greater than B, it follows by properties of a quotient that $Q \geq 1$. From the problem description we have $(A',B') = (B, A \bmod B)$. Thus, if we would like to find $L' = g(A',B')$, we can plug in the definitions of A', B' into g. Thus, we have $g(A',B') = B + \beta(A \mod B)$. However, since $A = QB + R$, then $A \mod B$ yields $A - QB = R$. Thus, $L' = g(A',B') = B + \beta R$.

Next, we will find a lower bound L of $g(A,B)$. Since we know $Q \geq 1$, we have $g(A,B) = g(QB+R,B) = QB + R + \beta B \geq B + R + \beta B$. Therefore, $L = B + R + \beta B$.

(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)

Proof. From part (b), let $L = B + R + \beta B$ and $L' = B + \beta R$. The ratio $\frac{L'}{L}$ is then $\frac{L'}{L} = \frac{B+\beta R}{B+R+\beta B}$. Factoring the denominator, we get $\frac{L'}{L} = \frac{B+\beta R}{R+(\beta+1)B}$. In order for the ratio $\frac{L'}{L}$ to remain the same for any B, R, the ratio of the coefficients of B and R in the numerator and denominator must be the same. That is, if $\frac{L'}{L} = \frac{(1)B+\beta R}{(1)R+(\beta+1)B}$ is constant, then $\frac{1}{\beta+1} = \frac{\beta}{1}$ must be true. After cross multiplying and simplifying, we have $\beta(\beta+1) = 1$ and $\beta^2 + \beta - 1 = 0$. Solving for β, we get $\beta = \frac{-1+\sqrt{5}}{2}$ and $\beta = \frac{-1-\sqrt{5}}{2}$. However, since we are given $\beta > 0$ in part (b), $\beta = \frac{-1+\sqrt{5}}{2}$ is our only solution. To verify our solution, we can plug $\beta = \frac{-1+\sqrt{5}}{2}$ back into $\frac{L'}{L}$, yielding $\frac{L'}{L} = \frac{-1+\sqrt{5}}{2}$. Therefore, a choice of β for which the ratio L'/L is always the same regardless of B, R is $\beta = \frac{-1+\sqrt{5}}{2}$. □

(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).

Proof. Suppose that Euclid's algorithm makes p recursive calls. Thus, at the final recursive call we must have $A_p > 0$ and $B_p = 0$. In addition, in part (c), we proved that a choice of $\beta = \frac{-1+\sqrt{5}}{2}$ implies $L'/L = \frac{-1+\sqrt{5}}{2} = \beta$, where $L = B + R + \beta B$ and $L' = B + \beta R$. We also have shown in part (b) that L is the lower bound of $g(A, B)$ and thus $L \leq g(A, B)$.

It follows that $L = \frac{L'}{\beta}$ and $\frac{L'}{\beta} \leq g(A, B)$. Furthermore, since $\beta > 0$ (from part (b)), we have $L' \leq \beta g(A, B)$. Substitution yields $A' + \beta B' \leq \beta(A + \beta B)$. It follows that $A_{i+1} + \beta B_{i+1} \leq \beta(A_i + \beta B_i) \leq (\beta)(\beta)(A_{i-1} + \beta B_{i-1}) \leq ... \leq (\beta)^{i+1}(A + \beta B)$ for some integer i. In addition, since $A_p > 0$ and $B_p = 0$, then $A_p + \beta B_p \geq 1$. Therefore, we have $1 \leq A_p + \beta B_p \leq (\beta)^p (A + \beta B)$. Taking logs, we get $0 = \log_2(1) \leq p \log_2(\beta) + \log_2(A + \beta B)$. If we use the fact that $\log_2(\beta) = -\log_2(\frac{1}{\beta})$, we can rearrange this inequality such that $p \leq \frac{\log_2(A+\beta B)}{\log_2(\frac{1}{\beta})}$. Simplifying, we conclude that $p \leq \log_{\frac{1}{\beta}}(A + \beta B)$. However, $\frac{1}{\beta} = \frac{1+\sqrt{5}}{2} = \phi$. Hence, we have $p \leq \log_\phi(A + \frac{B}{\phi})$. Thus, we have proved that Euclid's algorithm will terminate after at most $\log_\phi(A + \frac{B}{\phi})$, or $\log_\phi(g(A, B))$ recursive calls, where $\phi = \frac{1+\sqrt{5}}{2}$. □

(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.

If the bound in the part (d) is asymptotically tight, then the equation $\beta(A + \beta B) = A' + \beta B'$ must be true. Thus, by rearranging terms, we get:

$$\beta A + \beta^2 B = A' + B'\beta$$

$$A + \beta B = \frac{A'}{\beta} + B'$$

$$A + \beta B = \phi B + B'$$

$$A = (\phi - \beta)B + B'$$

$$A = B + B'$$

The last line defines the Fibonacci sequence. Therefore, a sequence of increasing inputs (A,B) for which the bound is asymptotically tight is the Fibonacci sequence ((1,1), (2,1), (3,2), (5,3)...).

(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.

Answer: Out of the three methods of computing Fibonacci numbers, the recursive method appears to be the slowest. The next fastest method is the iterative method, followed by the matrix exponentiation method. However, it is important to note that the difference in speed between the recursive method and the iterative method is extremely large while the difference in speed between the iterative method and the matrix method is smaller until extremely large n. The following computations were done in VSCode using python and numpy.

I computed the 40th Fibonacci number (F_{40}) ten times using the three algorithms (recursive, iterative, matrix) and averaged the run times of each computational method. The recursive method averaged a real-world runtime of 10.63 seconds per computation of F_{40}. Meanwhile, the iterative method averaged a real-world runtime of $3.64 * 10^{-6}$ seconds (0.00000364s) per computation of F_{40}. Finally, the matrix method averaged a real-world runtime of $3.63 * 10^{-5}$ seconds (0.0000356s) per computation of F_{40}. The difference between the recursive method is vast; the iterative method and matrix method complete the computation in fractions of a second while the recursive method is more than 10^6 times slower. In addition, the iterative method is around 10 times faster than the matrix approach.

Next, I computed the 10th Fibonacci number (F_{10}) ten times using the three algorithms (recursive, iterative, matrix) and averaged the run times of each computational method. The recursive method averaged a real-world runtime of $2.05 * 10^{-5}$ seconds (0.0000205s) per computation of F_{10}. Meanwhile, the iterative method averaged a real-world runtime of $3.78 * 10^{-6}$ seconds (0.00000378s) per computation of F_{10}. Finally, the matrix method averaged a real-world runtime of $2.15 * 10^{-5}$ seconds (0.0000215s)

per computation of F_{10}. This time, when computing smaller fibonacci numbers, the difference between the recursive method and the other two methods is much smaller, with all three methods computing F_{10} in fractions of a second. It is important to note that the recursive method actually was faster than the matrix method for $n = 10$. The iterative method still remained around 10 times faster than the other two algorithms.

Finally, I computed the 1000-th Fibonacci number (F_{1000}) ten times using just the iterative and matrix algorithms and averaged the run times of each computational method. I excluded the recursive method since the algorithm was too slow and timed out before computing F_{1000}. The iterative method averaged a real-world runtime of $4.03 * 10^{-4}$ seconds (0.000403s) per computation of F_{1000}. Finally, the matrix method averaged a real-world runtime of $4.06 * 10^{-5}$ seconds (0.0000406s) per computation of F_{10}. This time, when computing extremely large fibonacci numbers, the matrix method is around 10 times faster than the iterative method.

With these measurements, I hypothesize that the running time of the recursive method is exponentially greater than the running times of the iterative and matrix method. This is because when computing smaller Fibonacci numbers, the recursive algorithm performs relatively similar to the other two algorithms while the difference in its runtime grows increasingly large as n increases. I believe that the slow running time of the recursive method is due to computing the smaller Fibonacci numbers multiple times: the algorithm must compute F_{n-1} once, F_{n-2} twice, F_{n-3} three times, and so on, resulting in inefficiency. I also noticed that as n reaches extremely large values, the matrix method is much faster than the iterative method while at small n the iterative method is faster. I believe this is attributed to the matrix method using a python numpy object which could incur a small running time cost to initialize. Thus, at large n, the advantage in runtime speed of the matrix method would compensate for the numpy running time cost.

(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.)

Answer: The first Fibonacci number that is at least 2^{31} is $F_{47} = 2971215073$, where F_n denotes the $n-th$ Fibonacci number.

(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.

These k-values were computed in VSCode using Python.
Recursive: The largest value of k such that I could compute the k^{th} Fibonacci number (modulo 65536) in one minute of machine time was $k = 43$ for the recursive method.
Iterative: The largest value of k such that I could compute the k^{th} Fibonacci number (modulo 65536) in one minute of machine time was $k = 770{,}000$ for the iterative method.
Matrix: The largest value of k such that I could compute the k^{th} Fibonacci number (modulo 65536) in one minute of machine time was $k = 10^{308001}$ for the matrix method.

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. $f_5 = (\log\log n)^n$

Answer: $f_3 < f_2 < f_1 = f_4 < f_5$

The rows of the table are i and the columns are j. Therefore, row 1 column 1 in the table is equivalent to $f_1 \in O(f_1)$, $f_1 \in \Omega(f_1)$.

i,j	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, Ω	ω,Ω	o,O	o,O
f_3			O, Ω	o,O	o,O
f_4				O, Ω	o,O
f_5					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 $i \in \{1,2,3,4,5\}$.

Answer: If f_6 has the property that *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 $i \in \{1,2,3,4,5\}$, then we must make sure that three properties are true: f_6 has to grow faster than f_5, f_6 codomain is $\mathbb{R}^+$, and f_6 fluctuates between being greater than $f_1 \ldots f_5$ and less than $f_1 \ldots f_5$ as $n \to \infty$. Thus, a possible function f_6 could be

$$f_6 = (\log n)^n(\cos(n) + 1) + 0.1$$

Since $f_5 = (\log\log n)^n$, we know that the $(\log n)^n$ component of our function will grow faster than f_5 since we take one less log (also, $(\log n)^n$ will always be defined on our domain of natural numbers). In addition, since $\cos(n) + 1$ fluctuates between 0 and 2, inclusive, then our function will fluctuate between being greater than f_5 and 0. However, we need $f_6 : \mathbb{N} \to \mathbb{R}^+$ so we will add 0.1 so that the least possible value of f_6 is $0.1 \in \mathbb{R}^+$.

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

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

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

i: $T(1) = 1, T(n) = T(n-1) + n^2 - n$
$T(2) = 1 + 2^2 - 2 = 3$, $T(3) = 3 + 3^2 - 3 = 9$, $T(4) = 9 + 4^2 - 4 = 21$

After calculating different T(n) values, I would guess that the form of a solution is

$$T(n) = \frac{n(n-1)(n+1)+3}{3}$$

Proof. We will prove this solution correct with induction.

Base Case: Our base case is when $n = 1$. We have $T(1) = \frac{1(0)(2)+3}{3} = 1$. Thus, our base case is proved correct.

Inductive Step: Assume $T(n) = \frac{n(n-1)(n+1)+3}{3}$. We wish to prove that given $T(n) = \frac{n(n-1)(n+1)+3}{3}$, then $T(n+1) = \frac{(n)(n+1)(n+2)+3}{3}$. It follows:

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

$$T(n+1) = \frac{n(n-1)(n+1)+3}{3} + (n+1)^2 - (n+1)$$

$$T(n+1) = \frac{n(n-1)(n+1)+3+3(n+1)^2-3(n+1)}{3}$$

$$T(n+1) = \frac{n^3+3n^2+2n+3}{3}$$

$$T(n+1) = \frac{n(n+1)(n+2)+3}{3}$$

Therefore, by induction, $T(n) = \frac{n(n-1)(n+1)+3}{3}$. □

ii: $T(1) = 1$, $T(n) = 3T(n-1) - n + 1$
$T(2) = 3(1) - 2 + 1 = 2, T(3) = 3(2) - 3 + 1 = 4, T(4) = 3(4) - 4 + 1 = 9, T(5) = 3(9) - 5 + 1 = 23, T(6) = 3(23) - 5 + 1 = 65$

After calculating different T(n) values, I would guess that the form of a solution is

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

Proof. We will prove this solution correct with induction.

Base Case: Our base case is when $n = 1$. We have $T(1) = \frac{3^0+2(1)+1}{4} = 1$. Thus, our base case is proved correct.

Inductive Step: Assume $T(n) = \frac{3^{n-1}+2n+1}{4}$. We wish to prove that given $T(n) = \frac{3^{n-1}+2n+1}{4}$, then $T(n+1) = \frac{3^n+2(n+1)+1}{4}$. It follows:

$$T(n+1) = 3T(n) - (n+1) + 1$$

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

$$T(n+1) = \frac{3*3^{n-1}+6n+3-4n-4+4}{4}$$

$$T(n+1) = \frac{3^n+2n+3}{4}$$

$$T(n+1) = \frac{3^n+2(n+1)+1}{4}$$

Therefore, by induction, $T(n) = \frac{3^{n-1}+2n+1}{4}$. □

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

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

i: According to Professor Sudan in an Ed post, the Master Theorem still holds with a floor function since it does not affect asymptotic behavior. Thus, to find tight asymptotic bounds for $T(n)$, we can apply the Master Theorem to the function $T(n) = 9T(n/3) + n^2 + 3n$. It follows that $a = 9, b = 3, k = 2$ and $a = b^k$. Therefore, by the Master Theorem, $T(n) = \Theta(n^2 \log n)$.

ii: As mentioned in part (i), the Master Theorem still holds with a floor function since it does not affect asymptotic behavior. Thus, we will ignore the floor function and consider $T(n) = 4T(\sqrt{n}) + \log n$. Now let $s = \log n$. It follows that $n = 2^s$. Therefore, $T(2^s) = 4T(2^{s/2}) + s$. Let us define another function $Q(s)$ such that $Q(s) = T(2^s)$. Then, we have $Q(s) = 4Q(s/2) + s$. Since $a = 4$, $b = 2$, $k = 1$, and $a > b^k$ then we can apply the Master Theorem to get $Q(s) = \Theta(s^{\log_2 4}) = \Theta(s^2)$. Substituting our change of variable, we get $Q(s) = T(2^s) = T(n) = \Theta((\log n)^2)$.

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

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

   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.,

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
```

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
```

describe at least one difference in the swaps made by the two algorithms.

Answer: One difference between the two methods is that BubbleSort compares and swaps elements that are in consecutive places in the array ("pairs" of elements) while TwistedBubbleSort compares and swaps elements that are non-adjacent in the array. Another difference between the methods is that Bubble Sort uses swaps to "bubble" the largest element to the end of the array, the second largest element to the second to last place of the array, and so on until the array is sorted. Meanwhile, TwistedBubbleSort uses swaps to move the largest element from the first index to the last index incrementally, sorting all the elements which come before the largest element.

(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.

Proof. We will prove this with induction.

Base Case: Our base case is when $i = 0$ (the first iteration of the outer loop). The algorithm's inner loop then iterates through the entire array (from index $j = 0$ to $j = n-1$) and compares the first element $A[0]$ with the element at $A[j]$. The algorithm swaps the two elements if $A[0]$ is less than $A[j]$. Let element $p = A[k]$ where $0 \leq k \leq n-1$ be the largest element in array A. Thus, when $j = k$, we know that the algorithm will swap the element at the 0-th index with p. Furthermore, for all $k < j \leq n-1$, the algorithm will not swap any element with the element p at the 0-th index since p is by definition the greatest value in the array. Thus, at the end of the 0-th iteration, the largest element p

will be at the 0-th index.

Inductive Step: Assume that after the i-th iteration of the outer loop, the largest element p is in the i-th index of array A. We wish to prove that this assumption implies that after the $(i+1)$-th iteration of the outer loop, the largest element p is in the $(i+1)$-th index of the array. In the $(i+1)$-th iteration, the algorithms inner loop iterates through the entire array (from index $j = 0$ to $j = n-1$). However, since the greatest element p is at the i-th index according to the inductive hypothesis, once the inner loop reaches the i-th index ($j = i$), then the algorithm swaps the $(i+1)$-th element with p at index i since $p > A[i+1]$ (by definition of the greatest element of an array). Moreover, for all $i+1 < j \leq n-1$, the algorithm will not swap any element with the element p at the $(i+1)$-th index since p is by definition the greatest value in the array. Thus, the largest element p will be at the $(i+1)$-th index after the $(i+1)$-th iteration of the outer loop of TwistedBubbleSort.

Therefore, by induction, after the i-th iteration of the outer loop of TwistedBubbleSort, the largest element of A is at the i-th index. □

(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.

Proof. We will prove this with induction.

Base Case: Our base case is when $i = 0$ (the first iteration of the outer loop). Let p be the largest element of A. According to our proof in part (b), we know that the largest element p will be at the 0-th index after the 0-th iteration of the outer loop. Therefore, the indices 0 to 0 form a sub-array which contains a single element p. Since a sub-array with only 1 element is inherently sorted, then we can conclude that after the 0-th iteration of the outer loop of TwistedBubbleSort, the indices from 0 to 0 of the array are sorted.

Inductive Step: Assume that after the i-th iteration of the outer loop, the indices 0 to i are sorted. We wish to prove that this assumption implies that after the $(i+1)$-th iteration of the outer loop, the indices 0 to $i+1$ are sorted. To do this, we will consider three distinct stages of the inner loop j which together make up the $(i+1)$-th iteration. Let d be the element at the $(i+1)$-th index of the array. Let d be the m-th smallest value in the array.

Then, our first stage occurs when $0 \leq j \leq m-2$. Since our inductive hypothesis states that the array elements from 0 to i are all sorted, then there is a threshold index $(m-2)$ such that d is greater than all elements from 0 to $m-1$. Therefore, since d is by definition greater than the first $(m-2)$-th elements (due to the array being 0 indexed), then the algorithm will not perform any swaps and the array will remain unchanged.

The next stage is from $m-1 \leq j \leq i+1$. When $j = m-1$, then d at index $(i+1)$

will be swapepd with the element at index $(m-1)$ (this follows from our inductive hypothesis since the elements from indices $(m-1)$ to $(i+1)$ are sorted, the fact that d is by definition the m-th smallest element of the array, and the array is 0 indexed). Hence, d will be swapped to index $(m-1)$ and $A[m-1]$ will be swapped to index $(i+1)$. Then, as j increments one, the element at index $j = m$ is larger than the current element at index $(i+1)$ because our inductive hypothesis states that the elements from indices $0 \leq m-1 \leq m \leq i$ are all sorted. Therefore, $A[m]$ will be swapped to index $(i+1)$ and the element at index $(i+1)$ will be swapped to index m. Hence, the element which originally started out at index $m-1$ is now at index m. By the same logic, the element which originally started out at index m will be swapped to index $m+1$, and so on. Now, let us consider when the inner loop reaches index i (that is, when $j = i$). By our proof in part (b), we know that the largest element p of the array is at index i after the i-th iteration. It follows from the previous statements that p will be swapped to index $(i+1)$ at the end of this stage. Moreover, the array will remain sorted, since each element at index j will be shifted to index $(j+1)$, preserving the sorted order of our inductive hypothesis.

Our last stage occurs when $i+1 \leq j \leq n-1$. Since we previously showed that the largest element p will be at index $(i+1)$ at the end of the previous stage. Therefore, by definition of largest element, the algorithm will not swap any element from the indices $i+1$ to $n-1$ since no element will be greater than p at index $(i+1)$. It follows that no element smaller than previous elements will be swapped into index $(i+1)$ and cause the array from 0 to $(i+1)$ to be unsorted. Thus, the elements from indices 0 to $(i+1)$ will be sorted after the $(i+1)$-th iteration of the outer loop.

Therefore, by induction, we have proved that after the i-th iteration of the outer loop of TwistedBubbleSort, indices 0 to i of the array are sorted. □

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.