Lauren
Lyons

1) The worst case running time is n^2. This is not a linear-time algorithm because the worst case is n^2, so the Big-Oh complexity is $O(n^2)$, especially since we are dealing with an nxn array.

2) A strategy that can find the missing integer in A is that you will do the summation $\sum_{i=0}^{n-1} i$, and then you sum all of the elements in the array A, and finally you subtract the array's sum from the summation. An example is when n=6, so the array A could be [0, 1, 2, 3, 5]. The array's sum is 11. The summation $\sum_{i=0}^{6-1} i = 15$. 15-11=4 which is the missing element from that range.

3) A strategy to convert a string of n digits in the range 0 to 9 into the integer it represents is to take the substrings of just length 1 and parse that individual integer, whether through a hardcoded "0"-"9" or a single integer parser, and then multiply it by 10^{n-i-1} where n is the length of the string and i is the index of a single integer. Then you add the result from each index and total up in order to get the integer that the string represents.

3 cont) An example is "465" where 4 is at index 0, 6 is at index 1, and 5 is at index 2. n = 3.
$(4 \times 10^{3-0-1}) + (6 \times 10^{3-1-1}) + (5 \times 10^{3-2-1}) = 400 + 60 + 5 = 465.$

4)

Algorithm A1	Algorithm A2	Algorithm A3
$O(n)$	$O(n \log n)$	$O(n^2)$
$200,000,000$	$2 \times 10^7 \log(2 \times 10^9)$	$(2 \times 10^8)^2$
2×10^8 steps	5.515×10^9 steps	4×10^{16} steps
$\div 2 \times 10^7$ steps/sec	$\div 2 \times 10^7$ steps/sec	$\div 2 \times 10^7$ steps/sec
10 seconds	275.75 seconds	2×10^9 seconds

5) Inputs: - Campus building from user selection
- Users location (latitude, longitude)

Outputs: - Google Map API's turn-by-turn directions
- Lot name
- (optional) number of parking spots available in lot

Data Representations: String: campus building name and parking lot name
Float: user's location (latitude, longitude)
Int: (optional) number of parking spots available in lot
Queue: turn-by-turn directions

6) - A correct and efficient strategy that checks the max difference between any pair of numbers in an array of n numbers starts by assigning the first element of the array to the two variables highest and lowest to be used in the final output. You then iterate through the array and compare each element to the two variables and determine whether the current element will take place of either variable.
- Assuming the pairs don't repeat ([1,2] is the same as [2,1]), the total number of number pairs computed will be $\sum_{i=1}^{n-1} i$ so that the result is $\frac{n(n-1)}{2}$.

7) a) This algorithm outputs the max sum from all consecutive sub-arrays. In this simulation, n=3, A=[-4, 8, 9]. At beginning, m=0, j=1 on first iteration, then k=1 on first iteration, s=0 and i=1 on its first/only iteration where it goes to k=1. So s=s+A[i]=0+A[1]=-4. s is not greater than m, so m stays equal to 0. Then k iterates to 2 and s=0, i=1, s=s+A[i]=0+A[1]=-4. i iterates to 2, s=s+A[i]=-4+A[2] =-4+8=4. s is greater than m so m=s=4. Then k iterates to 3. s=0, i=1, s=s+A[i]=0+A[1]=-4. i iterates to 2, s=s+A[i]=-4+A[2]=4, i iterates to 3.

7 cont) $s = s + A[i] = 4 + A[3] = 4 + 9 = 13$, $s = 13 > m = 4$ so $m = 13$ now. Then j iterates to 2. $k = 2$, $s = 0$, $i = 2$ for one iteration where $s = s + A[i] = 0 + A[2] = 8$. k iterates to 3, $s = 0$, $i = 2$, $s = s + A[i] = 0 + A[2] = 8$. i iterates to 3, $s = s + A[i] = 8 + A[3] = 8 + 9 = 17$. $s = 17 > m = 13$ so $m = 17$. j iterates to 3 for the last iteration. $s = s + A[i] = 0 + A[3] = 9$. The for loops all terminate and m is returned as 17, the largest sum from the consecutive subarray [8, 9]

b) The approximate time complexity is $O(n^3)$ based on the 3 nested loops.

c) The following time complexity improves to $O(n^2)$ instead. It changes strategy from going through every combination to taking the first step and putting it into the sum array. The algorithm's first for loop adds the consecutive numbers where index 0 is none added, index 1 is 1 added, index 2 is 2 consecutive numbers added, etc. From there it takes one level of n tries away to make it $O(n^2)$.

8)

Step	Big-Oh complexity
1	$O(1)$
2	$O(1) \rightarrow O(n)$
3	$O(1) \rightarrow O(n) \rightarrow O(n^2)$
4	$O(1) \rightarrow O(n) \rightarrow O(n^2)$
5	$O(1) \rightarrow O(n) \rightarrow O(n^2) \rightarrow O(n^3)$
6	$O(1) \rightarrow O(n) \rightarrow O(n^2) \rightarrow O(n^3)$
7	$O(1) \rightarrow O(n) \rightarrow O(n^2)$
8	$O(1) \rightarrow O(n) \rightarrow O(n^2)$
9	$O(1)$

Overall Big-Oh Time complexity: $O(n^3)$

9)

Step	Cost	# of times
1	1	1
2	1	$n+1$
3	2	$\frac{n(n+1)}{2}$
4	1	$\frac{n(n+1)}{2}$
5	3	$\frac{n(n+1)(n-1)}{2}$
6	6	$\frac{n(n+1)(n-1)}{2}$
7	3	$\frac{n(n+1)}{2}$
8	2	$\frac{n(n+1)}{2}$
9	2	1

$T(n) = \frac{9}{2}n^3 + 4n^2 + \frac{1}{2}n + 4$

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

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

$= 4 + \frac{1}{2}n - 4n^2 + \frac{9}{2}n^3$

10) The algorithm is incorrect. The example they give us and the expected output does not match the actual output from the algorithm. (I shortened the variable count to c)

Proof of wrong output

c: 0	c: 0	c: 1	c: 1	c: 1	c: 2	c: 2	c: 3	c: 3
i: ~~0~~ →	i: 1 →	i: 3 →	i: 3 →	i: 3 →	i: 6 →	i: 7 →	i: 8 →	i: 8
j: ~~0~~	j: 2	j: 3	j: 4	j: 5	j: 6	j: 8	j: 8	j: ∅

Final return: c=3

- The algorithm uses read-next-integer(f) too often, iterating over all the numbers without the right comparisons.

11) [4] In the algorithm, it has base cases where k=0 and k=1 in the form of if statements. So when n=0, it returns 1, which is correct. Then when n=1, it also returns 1, which is also correct. The 0th number of a Fibonacci sequence is 1 and the 1st number is 1. So in both cases, the algorithm returns the correct answer.

[6] When n=k and k>1, we then assign 1 to last and current, and steps 3-9 will be executed.

[8] If k=3, steps 5-8 will be executed twice. In the first iteration where those lines are

11 (cont) executed, current = $F_0 + F_1 = F_2$, $F_2 = F_0 + F_1$, and last = current = F_1. Now in the second iteration on step 6, temp = last + current = $F_1 + F_2$. Then step 7 updates last = current = F_2. Then step 8 sets current = $F_1 + F_2 = F_3$ and then it returns in step 9. $F_1 + F_2$ is equal to F_3, which is the correct answer. So the k for which the algorithm fails must be greater than 3.

9 But if $k=4$, steps 5-8 will be executed 3 times, and after the second iteration, current = $F_1 + F_2 = F_3$ and last = F_2. Step 6, temp = last + current = $F_2 + F_3$, then step 7, last = F_3, and step 8 current = temp = $F_2 + F_3 = F_4$ and F_4 is returned in step 9, which is correct. So the k for which the algorithm fails must be greater than 4.

10 The above argument can be repeated to show that for all k, the algorithm will return $F_k = F_{k-1} + F_{k-2}$ since temp will equal $F_{k-1} + F_{k-2}$, last = F_{k-1}, and current will equal $F_{k-1} + F_{k-2}$ which is then returned to equal F_k. So, the k for which the algorithm fails must be greater than k.

12) The ReverseString function will have the string passed as an array of characters, the base case is covered by an if statement (where its an empty string), then it will swap(A[i], A[j]) where p is the first element in the character array and q is the last. Then theres an if statement to handle the case if the total string is an uneven length. If p+1 is equal to q-1 or p is equal to q, then the character array is uneven and you just return the string. As an example, for a string of length 3, it could be "abc", and if you swap "a" and "c", there is no moving "b", so you can just return that string, "cba". If the if statement is true, you just return the string. Then if it is even length, it passes the if statement and returns the first letter of the current charArray (that has already been swapped by swap(A[p], A[q]) plus ReverseString(A[p+1...q-1]) plus the last character of the current charArray. For A[p+1,...q-1], the charArray will go from [abcd] → [bc] → [].

12 cont) ReverseString(A[p...q])

```
if (length of A is equal to 0)
    then return empty string
swap(A[p], A[q])
if (p+1 is equal to q-1 or p is equal to q)
    return A (the char Array)
return A[p] + ReverseString(A[p+1...q-1]) + A[q]
```

- Recursion Tree (ReverseString → RS)

RS("i<33270!") returns "!"+"07233<"+"i"="!07233<i" (Final output)

↓

RS("<33270") returns "0"+"7233"+"<"="07233<"

↓

RS("3327") returns "7"+"23"+"3"="7233"

↓

RS("32") returns "2"+""+"3"="23"

↓

RS("")

BASE CASE returns ""

13) Base Case Proof: n=0; when n=0, so it's g(0), the algorithm will be caught by an if statement and return(n) → return(0) → so g(0)=0, and $3^n-2^n=3^0-2^0=1-1=0$, so they are equal and therefore this base case is true. n=1; When n=1, so g(1), the algorithm will again be caught by an if statement and return(n) → return(1) → so g(1)=1 and $3^n-2^n=3^1-2^1=3-2=1$,

13cont) so they are equal and therefore this base case is also true.

Inductive Hypothesis: $g(n) = 3^n - 2^n$ for $0 \leq n \leq k$ for some k

Inductive step: we need to show that $g(k+1) = 3^{k+1} - 2^{k+1}$. We know that $g(k) = 3^k - 2^k$ and $g(k-1) = 3^{k-1} - 2^{k-1}$ from the inductive hypothesis. We also know that $g(k+1) = 5(g(k+1-1)) - 6(g(k+1-2)) = 5(g(k)) - 6(g(k-1))$ by the definition through the algorithm.

So $g(k+1) = 5(3^k - 2^k) - 6(3^{k-1} - 2^{k-1})$

$3^{k+1} - 2^{k+1} = 5(3^k - 2^k) - 6(3^k(1/3) - 2^k(1/2))$

$(3)3^k - (2)2^k = (5)3^k - (5)2^k - (2)3^k + (3)2^k$

$(3)3^k - (2)2^k = (3)3^k - (2)2^k$

$3^k - 2^k = 3^k - 2^k$

$0 = 0$ ✓ True! Done!

14) Loop Invariant: Before any execution of the for loop of line 5 in which the loop variable $i = k$, $2 \leq k \leq n$, the variable last will contain F_{k-2} and the variable current will contain F_{k-1}.

Initialization: Before the start of the loop where $i = 2$, we know that $F_0 = 1$ and $F_1 = 1$ and therefore we set last = 1 and current = 1 since last = F_0 and current = F_1 based on $i = 2$ and last = F_{i-2} and current = F_{i-1}.

ont) Maintenance: Before the start of the i^{th} iteration, last = F_{i-2} and current = F_{i-1}. During the i^{th} iteration, temp is computed as temp = last + current = $F_{i-2} + F_{i-1}$, last is computed as last = current = F_{i-1}, and current is computed as current = temp = $F_{i-2} + F_{i-1}$ where $F_{i-2} + F_{i-1} = F_i$ based off the definition of the Fibonacci sequence as $F_k = F_{k-1} + F_{k-2}$. Then on the next iteration, i+1, the proposal that last = $F_{i-2} = F_{(i+1)-2} = F_{i-1}$ and current = $F_{i-1} = F_{(i+1)-1} = F_i$ holds true because during the last iteration, last was updated to equal F_{i-1} and current was updated to equal F_i, so the LI holds true.

Termination: Since initialization shows that the LI will be true before the loop begins and Maintenance shows that it will continue to be true before each iteration of i, so it must be true before the last iteration, the n^{th} iteration. The loop ends after the n^{th} iteration, so before the $(n+1)^{th}$ iteration, the LI must be true. Before the i^{th} iteration, last will equal F_{i-2} and current will equal F_{i-1}. Therefore, before the $(n+1)^{th}$

14(cont) iteration, last will equal $F_{(n+1)-2} = F_{n-1}$ and current will equal $F_{(n+1)-1} = F_n$ which follows the Fibonacci sequence of $F_k = F_{k-1} + F_{k-2}$ → $\rightarrow F_{n+1} = F_n + F_{n-1}$. Since the algorithm terminates with the Fibonacci sequence definition, the algorithm works correctly.