**Computer Science Thinking:  
Recursion, Searching, Sorting and   
Big O**

* Introduction

11.1 Q1: Which of the following statements is false?

a. Recursive functions (or methods) call themselves, either directly or indirectly through other functions (or methods).

b. Recursion can often help you solve problems more naturally when an iterative solution is not apparent.

c. Sorting unique values is a fascinating problem, because no matter what algorithms you use, the final result is the same. So you’ll want to choose algorithms that perform "the best"—usually, the ones that run the fastest or use the most memory.

d. All of the above statements are true.

Answer: c. Sorting unique values is a fascinating problem, because no matter what algorithms you use, the final result is the same. So you’ll want to choose algorithms that perform "the best"—usually, the ones that run the fastest or use the most memory.

Actually, you’ll want to choose the algorithms that run the fastest and use the least memory.

11.1 Q2: Which of the following statements is false?

a. For big data applications, you’ll also want to choose algorithms that are easy to sequentialize—that will enable you to put lots of processors to work simultaneously.

b. The simplest and most apparent algorithms often perform poorly—developing more sophisticated algorithms can lead to superior performance.

c. Big O notation concisely classifies algorithms by how hard they have to work to get the job done—it helps you compare the efficiency of algorithms.

d. All of the above statements are true.

Answer: a. For big data applications, you’ll also want to choose algorithms that are easy to sequentialize—that will enable you to put lots of processors to work simultaneously. Actually, for big data applications, you’ll also want to choose algorithms that are easy to parallelize—that will enable you to put lots of processors to work simultaneously.

* Factorials

11.2 Q1: Which of the following statements is false?

a. The factorial of a positive integer n is written n! and pronounced “n factorial.”

b. n! is the product

n · (n – 1) · (n – 2) · … · 1

with 1! equal to 1 and 0! defined to be 1.

c. 4! is the product 4 · 3 · 2 · 1, which is equal to 24.

d. All of the above statements are true.

Answer: d. All of the above statements are *true*.

11.2 Q2: What should the question mark (?) in the following for statement be replaced with, so that the statements will calculate 5!:

In [1]: factorial = 1

In [2]: for number in range(5, 0, ?):

...: factorial \*= number

...:

In [3]: factorial

Out[3]: 120

a. 1

b. 0

c. -1

d. None of the above.

Answer: c. -1.

* Recursive Factorial Example

11.3 Q1: Which of the following statements is false?

a. When you call a recursive function to solve a problem, it’s actually capable of solving only the simplest case(s), or base case(s). If you call the recursive function with a base case, it immediately returns a result.

b. The recursion step executes while the original function call is still active (i.e., it has not finished executing).

c. For the recursion to eventually terminate, each time the function calls itself with a simpler version of the original problem, the sequence of smaller and smaller problems must converge on a recursion step.

d. All of the above statements are true.

Answer: c. For the recursion to eventually terminate, each time the function calls itself with a simpler version of the original problem, the sequence of smaller and smaller problems must converge on a recursion step. Actually, for the recursion to eventually terminate, each time the function calls itself with a simpler version of the original problem, the sequence of smaller and smaller problems must converge on a *base case*.

11.3 Q2: Which of the following statements is false?

a. The amount of memory in a computer is finite, so only a certain amount of memory can be used to store activation records on the function-call stack.

b. If more recursive function calls occur than can have their activation records stored on the stack, a fatal error known as an infinite loop occurs.

c. This typically is the result of infinite recursion, which can be caused by omitting the base case or writing the recursion step incorrectly so that it does not converge on the base case.

d. All of the above statements are true.

Answer: b. If more recursive function calls occur than can have their activation records stored on the stack, a fatal error known as an infinite loop occurs. Actually, if more recursive function calls occur than can have their activation records stored on the stack, a fatal error known as *stack overflow* occurs.

* Recursive Fibonacci Series Example

11.4 Q1: Which of the following statements is false?

a. The Fibonacci series,

0, 1, 1, 2, 3, 5, 8, 13, 21, …

begins with 0 and 1 and has the property that each subsequent Fibonacci number is the sum of the previous two.

b. This series occurs in nature and describes a form of spiral.

c. The ratio of successive Fibonacci numbers converges on a constant value of 1.618…, a number that has been called the golden ratio or the golden mean. Humans tend to find the golden mean aesthetically pleasing.

d. The Fibonacci series may be defined recursively as follows:

Base case: fibonacci(0) is defined to be 0   
 Recursion step: fibonacci(n) = fibonacci(n – 1) + fibonacci(n – 2)

Answer: d. The Fibonacci series may be defined recursively as follows:

Base case: fibonacci(0) is defined to be 0   
 Recursion step: fibonacci(*n*) = fibonacci(*n* – 1) + fibonacci(*n* – 2)

Actually, There are two base cases—the second one is "fibonacci(1) is defined to be 1".

11.4 Q2: Function fibonacci calculates the nth Fibonacci number recursively:

In [1]: def fibonacci(n):

...: if n in (0, 1): # base cases

...: return n

...: else:

...: return fibonacci(n - 1) + fibonacci(n - 2)

...:

Which of the following statements is false?

a. Because fibonacci is a recursive function, all calls to fibonacci are recursive.

b. Each time you call fibonacci, it immediately tests for the base cases—n equal to 0 or n equal to 1, which we test simply by checking whether n is in the tuple (0, 1).

c. If a base case is detected, fibonacci simply returns n, because fibonacci(0) is 0 and fibonacci(1) is 1.

d. Interestingly, if n is greater than 1, the recursion step generates two recursive calls, each for a slightly smaller problem than the original call to fibonacci.

Answer: a. Because fibonacci is a recursive function, all calls to fibonacci are recursive. Actually, the initial call to function fibonacci is not a recursive call, but all subsequent calls to fibonacci performed from function fibonacci’s block *are* recursive because at that point the calls are initiated by the function itself.

* Recursion vs. Iteration

11.5 Q1: Which of the following statements is false?

a. Both iteration and recursion are based on a control statement: Iteration uses an iteration statement (e.g., for or while), whereas recursion uses a selection statement (e.g., if or if…else or if…elif…else):

b. Iteration and recursion each involve a termination test: Iteration terminates when the loop-continuation condition fails, whereas recursion terminates when a base case is reached.

c. Iteration with counter-controlled iteration and recursion both gradually approach termination: Iteration keeps modifying a counter until the counter assumes a value that makes the loop-continuation condition fail, whereas recursion keeps producing smaller versions of the original problem until the base case is reached.

d. Only iteration can occur infinitely: An infinite loop occurs with iteration if the loop-continuation test never becomes false.

Answer: d. Only iteration can occur infinitely: An infinite loop occurs with iteration if the loop-continuation test never becomes false. Actually, both iteration and recursion can occur infinitely: An infinite loop occurs with iteration if the loop-continuation test never becomes false, whereas infinite recursion occurs if the recursion step does not reduce the problem each time in a manner that converges on a base case, or if a base case is mistakenly not tested.

* Searching and Sorting

11.6 Q1: Which of the following statements is false?

a. Searching data involves determining whether a value (referred to as the search key) is present in the data and, if so, finding its location.

b. Two popular search algorithms are the simple binary search and the faster but more complex linear search.

c. Sorting places data in ascending or descending order, based on one or more sort keys.

d. Each of the above statements is true.

Answer: b. Two popular search algorithms are the simple binary search and the faster but more complex linear search. Actually, two popular search algorithms are the simple *linear* search and the faster but more complex *binary* search.

* Linear Search

11.7 Q1: Which of the following statements is false?

a. The linear search algorithm searches each element in an array sequentially.

b. If the search key does not match an element in the array, the algorithm informs the user that the search key is not present.

c. If the search key is in the array, the algorithm tests each element until it finds one that matches the search key and returns the index of that element.

d. All of the above statements are true.

Answer: d. All of the above statements are *true*.

11.7 Q2: The following code implements a simple linear search.

In [1]: def linear\_search(data, search\_key):

...: for index, value in enumerate(data)::

...: if value == search\_key:

...: return ?

...: return -1

...:

...:

In the statement return ?, what should the ? be?

a. data

b. search\_key

c. index

d. None of the above  
Answer: c. index

* Efficiency of Algorithms: Big O

11.8 Q1: Suppose an algorithm is designed to test whether the first element of an array is equal to the second. If the array has 10 elements, this algorithm requires one comparison. If the array has 1000 elements, it still requires one comparison. Which of the following statements is false:

a. The algorithm is completely independent of the number of elements in the array.

b. This algorithm is said to have a constant run time, which is represented in Big O notation as O(1) and pronounced as “order one.”

c. An algorithm that’s O(1) does not necessarily require only one comparison. O(1) means that the number of comparisons is constant—it does not grow as the size of the array increases.

d. An algorithm that tests whether the first element of an array is equal to any of the next three elements is O(3).

Answer: d. An algorithm that tests whether the first element of an array is equal to any of the next three elements is *O*(3). Actually, an algorithm that tests whether the first element of an array is equal to any of the next three elements is still *O*(1) even though it requires three comparisons.

11.8 Q2: An algorithm that tests whether the first array element is equal to any of the other array elements requires at most n – 1 comparisons, where n is the number of array elements. Which of the following statements is false?

a. If the array has 10 elements, this algorithm requires up to nine comparisons.

b. If the array has 1000 elements, it requires up to 999 comparisons.

c. As n grows larger, the n part of the expression n – 1 “dominates,” and subtracting 1 becomes inconsequential. Big O is designed to highlight these dominant terms and ignore terms that become unimportant as n grows.

d. An algorithm that requires a total of n – 1 comparisons is said to be O(n – 1) and is said to have a linear run time.

Answer: d. An algorithm that requires a total of *n* – 1 comparisons is said to be O(*n* – 1) and is said to have a linear run time. Actually, An algorithm that requires a total of *n* – 1 comparisons is said to be *O(n)* and is said to have a linear run time.

11.8 Q3: Suppose you have an algorithm that tests whether any element of an array is duplicated elsewhere in the array. The first element must be compared with every other element in the array. The second element must be compared with every other element except the first (it was already compared to the first). The third element must be compared with every other element except the first two. In the end, this algorithm makes (n – 1) + (n – 2) + … + 2 + 1 or n2/2 – n/2 comparisons. Which of the following statements is false?

a. As n increases, the n*2* term dominates, and the n term becomes inconsequential. O(n2) notation highlights the n2 term, ignoring n/2. O(n2) is referred to as quadratic run time and pronounced “on the order of n-squared” or more simply “order n-squared.”

b. Big O is concerned with how an algorithm’s run time grows in relation to the number of items, n, processed. When n is small, O(n2) algorithms (on today’s super fast computers) will not noticeably affect performance, but as n grows, you’ll start to notice performance degradation.

c. An O(n2) algorithm operating on a billion-element array (not unusual in today’s big-data applications) would require a quintillion operations, which on today’s desktop computer could take approximately 13.3 years to complete! O(n2) algorithms, unfortunately, are easy to write.

d. All of the above statements are true.

Answer: d. All of the above statements are *true*.

11.8 Q4: Which of the following statements is false?

a. The linear search algorithm runs in O(n) time.

b. The worst case in this algorithm is that every element must be checked to determine whether the search item exists in the array. If the size of the array is doubled, the number of comparisons that the algorithm must perform is quadrupled.

c. Linear search can provide outstanding performance if the element matching the search key happens to be at or near the front of the array.

d. Linear search is easy to program, but it can be slow compared to other search algorithms. If a program needs to perform many searches on large arrays, it’s better to implement a more efficient algorithm, such as the binary search.

Answer: b. The worst case in this algorithm is that every element must be checked to determine whether the search item exists in the array. If the size of the array is doubled, the number of comparisons that the algorithm must perform is quadrupled. Actually, if the size of the array is doubled, the number of comparisons that the algorithm must perform is also *doubled*.

* Binary Search

11.9 Q1: Which of the following statements about binary search of an array in ascending order is false?

a. The binary search algorithm is more efficient than linear search, but the linear search requires a sorted array.

b. The first iteration of this algorithm tests the middle element in the array. If this matches the search key, the algorithm ends.

c. If the search key is less than the middle element, it cannot match any element in the second half of the array so the algorithm continues with only the first half of the array (i.e., the first element up to, but not including, the middle element).

d. If the search key is greater than the middle element, it cannot match any element in the first half of the array so the algorithm continues with only the second half of the array (i.e., the element after the middle element through the last element).

Answer: a. The binary search algorithm is more efficient than linear search, but the linear search requires a sorted array. Actually, the binary search algorithm is more efficient than linear search, but the binary search requires a sorted array.

* Binary Search Implementation

No questions.

* Big O of the Binary Search

11.9 Q2: Which of the following statements about the binary search of an array in ascending order is false?

a. In the worst-case scenario, searching a sorted array of 1023 elements takes only 10 comparisons when using a binary search. The number 1023 (which is 210 – 1) is divided by 2 only 10 times to get the value 0, which indicates that there are no more elements to test.

b. Dividing by 2 is equivalent to one comparison in the binary search algorithm. Thus, an array of 1,048,575 (220 – 1) elements takes a maximum of 20 comparisons to find the key, and an array of about one billion elements takes a maximum of 30 comparisons to find the key. This is a tremendous performance improvement over the linear search.

c. For a one-billion-element array, the increase in performance of a binary search over a linear search is a difference between an average of five million comparisons for the linear search and a maximum of only 30 comparisons for the binary search!

d. The binary search’s big O is O(log n), which is also known as logarithmic run time and pronounced as “order log n.” Of course, this assumes the array is sorted, though, which could take substantial time.

Answer: c. For a one-billion-element array, this is a difference between an average of five million comparisons for the linear search and a maximum of only 30 comparisons for the binary search! Actually, for a one-billion-element array, this is a difference between an average of *500 million* comparisons for the linear search and a maximum of only 30 comparisons for the binary search!

* Sorting Algorithms

11.10 Q1: Which of the following statements about sorting unique items is false?

a. Sorting data (i.e., placing the data in a particular order—ascending or descending—is one of the most important types of computing applications.

b. An important item to understand about sorting unique values is that the end result—the sorted array—will be the same no matter which algorithm you use to sort the array. The choice of algorithm affects only the run time of the program.

c. Selection sort and insertion sort—are relatively simple to program but inefficient. Merge sort—is much faster than selection sort and insertion sort but harder to program.

d. All of the above statements are true.

Answer: b. An important item to understand about sorting unique items is that the end result—the sorted array—will be the same no matter which algorithm you use to sort the array. The choice of algorithm affects only the run time of the program. Actually, The choice of algorithm affects the run time *and the memory use* of the program.

* Selection Sort

11.11 Q1: Which of the following statements about the selection sort sorting in increasing order is false?

a. Selection sort is a simple, efficient, sorting algorithm.

b. Its first iteration selects the smallest element in the array and swaps it with the first element. The second iteration selects the second-smallest item (which is the smallest item of the remaining elements) and swaps it with the second element.

c. The algorithm continues until the last iteration selects the second-largest element and swaps it with the second-to-last index, leaving the largest element in the last index.

d. After the ith iteration, the smallest i items of the array will be sorted into increasing order in the first i elements of the array.

Answer: a. Selection sort is a simple, efficient, sorting algorithm. Actually, selection sort is a simple, but *inefficient*, sorting algorithm.

* Selection Sort Implementation

No questions.

* Utility Function print\_pass

No questions.

* Big O of the Selection Sort

11.11 Q2: Which of the following statements is false?

a. The selection sort uses nested for loops. The outer loop iterates over the first n – 1 elements in the array, swapping the smallest remaining item into its sorted position.

b. The inner loop iterates over each item in the remaining array, searching for the smallest element. This loop executes n – 1 times during the first iteration of the outer loop, n – 2 times during the second iteration, then n – 3, …, 3, 2, 1.

c. This inner loop will iterate a total of n(n – 1)/2 or (n2 – n)/2. In Big O notation, smaller terms drop out, and constants are ignored, leaving a Big O of O(n2).

d. The selection sort algorithm iterates fewer times when the array’s elements are partially sorted than when they are randomly ordered.

Answer: d. The selection sort algorithm iterates fewer times when the array’s elements are partially sorted than when they are randomly ordered. Actually, the selection sort algorithm iterates the same number of times regardless of whether the array’s elements are randomly ordered, partially ordered or already sorted.

* Insertion Sort

11.12 Q1: Which of the following statements about insertion sort sorting into ascending order is false?

a. Insertion sort is another simple, but inefficient, sorting algorithm.

b. The first iteration of this algorithm takes the second element in the array and, if it’s less than the first element, swaps it with the first element. The second iteration looks at the third element and inserts it into the correct position with respect to the first two, so all three elements are in order.

c. Using this algorithm, at the ith iteration, the first i elements of the original array are sorted, but they may not be in their final locations—smaller values may be located later in the array-.

d. All of the above statements are true.

Answer: d. All of the above statements are *true*.

* Insertion Sort Implementation

No questions.

* Big O of the Insertion Sort

No questions.

* Merge Sort

11.13 Q1: Which of the following statements is false?

a. Merge sort is an efficient sorting algorithm but is conceptually more complex than selection sort and insertion sort.

b. The merge sort algorithm sorts an array by splitting it into two equal-sized subarrays, sorting each subarray, then merging them into one larger array.

c. With an odd number of elements, the algorithm creates the two subarrays such that one has one more element than the other.

d. In the recursive merge sort, the base case is an array with two elements, which is, of course, sorted, so the merge sort immediately returns.

Answer: d. In the recursive merge sort, the base case is an array with two elements, which is, of course, sorted, so the merge sort immediately returns. Actually, in the recursive merge sort, the base case is an array with *one* element, which is, of course, sorted, so the merge sort immediately returns.

* Merge Sort Implementation

No questions.

* Big O of the Merge Sort

11.13 Q2: Which of the following statements a), b) or c) about the recursive merge sort is false?

a. Merge sort is far more efficient than insertion or selection sort.

b. At each level, O(n) comparisons are required to merge the subarrays.

c. Each level splits the arrays in half, so doubling the array size requires one more level. Quadrupling the array size requires two more levels.

d. An O(n2) algorithm is much faster than an O(n log n) algorithm.

Answer: d. An *O*(*n*2) algorithm is much faster than an *O*(*n* log *n*) algorithm. Actually, an *O*(*n*2) algorithm is much *slower* than an O(*n* log *n*) algorithm.

* Big O Summary for This Chapter’s Searching and Sorting Algorithms

11.14 Q1: Which of the following algorithm efficiencies is arranged in order from least efficient to most efficient?

a. O(log n), O(n2), O(n log n), O(n)

b. O(n), O(n log n), O(n2), O(log n)

c. O(n2), O(n log n), O(log n), O(n)

d. O(n log n), O(log n), O(n), O(n2)

Answer: c. *O*(*n*2), *O*(*n* log *n*), *O*(log *n*), *O*(*n*)

* Visualizing Algorithms

No questions.

* Generator Functions

11.15 Q1: Which of the following statements is false?

a. Generator expressions are similar to list comprehensions, but create generator objects that produce values on demand using lazy evaluation.

b. The yield and yield from statements, can make a function into a generator function.

c. Generator functions are eager—they return values immediately.

d. All of the above statements are true.

Answer: c. Generator functions are eager—they return values immediately. Actually, generator functions are lazy—they return values on demand.

11.15 Q2: Which of the following statements is false?

a. A generator function uses the yield keyword to return the next generated item, then its execution suspends until the program requests another item.

b. When the Python interpreter encounters a generator function call, it creates an iterable generator object that keeps track of the next value to generate. You must create a new generator object each time you wish to iterate through the generator’s values again.

c. The following generator function iterates through a sequence of values and returns the cube of each value on demand:

In [1]: def square\_generator(values):

...: for value in values:

...: return value \*\* 3

...:

d. When there are no more items to process, the generator raises a StopIteration exception, which is how a for statement knows when to stop iterating through any iterable object.

Answer: c. The following *generator function* iterates through a sequence of values and returns the cube of each value on demand:

In [1]: def square\_generator(values):

...: for value in values:

...: return value \*\* 3

...:

Actually, the function above returns one value immediately. The following *generator function* iterates through a sequence of values and returns the cube of each value on demand—note that return has been replaced by yield:

In [1]: def square\_generator(values):

...: for value in values:

...: yield value \*\* 3

...:

* Implementing the Selection Sort Animation

No questions.