 Answers to review questions from Chapter 10

1. The simplest recursive implementation of the Fibonacci function is considerably less efficient than the iterative version. Does this fact allow you to make any general conclusions about the relative efficiency of recursive and iterative solutions?

**No. Recursion is a technique that can be applied in many different ways, some of which are efficient and some of which are inefficient. It is possible to write a recursive implementation of the fib function that runs in *O*(log *N*) time, as discussed in exercise 12.**

2. What is the sorting problem?

**The *sorting problem* consists of rearranging the elements of an array or vector so that the elements appear in some well‑defined order.**

3. The implementation of **sort** shown in Figure 10‑1 runs through the code to exchange the values at positions **lh** and **rh** even if these values happen to be the same. If you change the program so that it checks to make sure **lh** and **rh** are different before making the exchange, it is likely to run more slowly than the original algorithm. Why might this be so?

**The algorithm would need to perform this check on every cycle, which takes a certain amount of time. In a small percentage of those cases, it could save the small amount of work involved in swapping two elements, but the savings would almost certainly be less than the additional cost.**

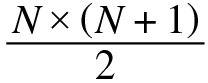
4. Suppose that you are using the selection sort algorithm to sort a vector of 250 values and find that it takes 50 milliseconds to complete the operation. What would you expect the running time to be if you used the same algorithm to sort a vector of 1000 values on the same machine?

**The number of values has gone up by a factor of 4, which means that you would expect the running time of selection sort to go up by a factor of 16. The expected running time would therefore be on the order of 800 milliseconds.**

5. What is the closed‑form expression that computes the sum of the series

*N* + *N*–1 + *N*–2 + ... + 3 + 2 + 1

**The closed form is**



6. In your own words, define the concept of computational complexity.

***Computational complexity* is a qualitative measure that relates the size of a problem (generally denoted as *N*) to the running time of the algorithm.**

7. True or false: Big‑O notation was invented as a means of expressing computational complexity.

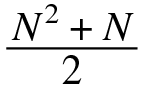
**False. Big-O notation was developed as a means of expressing the quality of an approximation.**

8. What are the two rules presented in this chapter for simplifying big‑O notation?

**1. *Eliminate any term whose contribution to the total ceases to be significant as N becomes large.***

**2. *Eliminate any constant factors.***

9. Is it technically correct to say that selection sort runs in

*O*()

time? What, if anything, is wrong with expressing computational complexity in this form?

**Although it is technically correct to use this specification, doing so fails to convey a useful qualitative sense of the running time because the formula offers too much detail. You can simplify the formula by eliminating the less significant terms involving *N* and removing the constant factor.**

10. Is it technically correct to say that selection sort runs in *O*(*N*3) time? Again, what, if anything, is wrong with characterizing selection sort in this way?

**Once again, saying that selection sort runs in *O*(*N*3) time is technically correct because anything that is bounded by *O*(*N*2) is certainly bounded by *O*(*N*3). The disadvantage of doing so is that the bound is not as tight as possible.**

11. Why is it customary to omit the base of the logarithm in big‑O expressions such as *O*(*N* log *N*)?

**Logarithms that use different bases differ from one another by a constant factor. Since constant factors are usually eliminated from big‑O expressions, there is no need to specify the logarithmic base.**

12. What is the computational complexity of the following function:

int mystery1(int n) {

int sum = 0;

for (int i = 0; i < n; i++) {

for (int j = 0; j < i; j++) {

sum += i \* j;

}

}

return sum;

}

***O*(*N*2)**

13. What is the computational complexity of this function:

int mystery2(int n) {

int sum = 0;

for (int i = 0; i < 10; i++) {

for (int j = 0; j < i; j++) {

sum += j \* n;

}

}

return sum;

}

***O*(1)**

14. Explain the difference between worst‑case and average‑case complexity. In general, which of these measures is harder to compute?

**As the names imply, worst‑case complexity indicates the complexity in the worst possible situation, and average‑case complexity indicates the expected complexity averaged over all possible cases. Average‑case complexity is generally much more difficult to compute.**

15. Explain the roles of the constants *C* and *N*0 in the formal definition of big‑O.

**The formal definition of big‑O requires you to select two constants—*C* and *N*0—that ensure that the specified formula is within a factor of *C* of the actual value for all values on *N* ≥ *N*0.**

16. In your own words, explain why the **merge** function runs in linear time.

**The loops in the merge function run for a total of *N* cycles, one to fill each of the elements of the vector. On each cycle, the code compares the current elements of v1 and v2, which requires a single comparison operation and then an assignment of the smaller value. Each cycle therefore takes constant time, and repeating that operation *N* times gives rise to the linear time bound.**

17. The last two lines of the **merge** function are

while (p1 < n1) vec.add(v1[p1++]);

while (p2 < n2) vec.add(v2[p2++]);

Would it matter if these two lines were reversed? Why or why not?

**The order of these loops doesn’t matter. At this point, only one of the vectors still contains elements so that only one of these loops will be executed at all.**

18. What are the seven complexity classes identified in this chapter as the most common classes encountered in practice?

**Constant, logarithmic, linear, *N* log *N,* quadratic, cubic, and exponential.**

19. What does the term *polynomial algorithm* mean?

**A *polynomial algorithm* is one whose running time is bounded by a polynomial function in *N,* which means that its computational complexity is *O*(*Nk*) for some constant *k.***

20. What criterion do computer scientists use to differentiate tractable and intractable problems?

**A *tractable problem* is one that can be solved in polynomial time. An *intractable problem* requires a larger time bound, typically exponential.**

21. In the Quicksort algorithm, what conditions must be true at the conclusion of the partitioning step?

**At the conclusion of the partitioning step, there must be a position in the vector that contains an element chosen to be the *pivot;* that position is called the *boundary.* In addition, all elements less that or equal to the pivot must appear to the left of the boundary and all larger elements must appear to the right.**

22. What are the worst‑ and average‑case complexities for Quicksort?

**In the worst case, Quicksort runs in *O*(*N*2) time; on average, it runs in *O*(*N* log *N*) time.**

23. Describe the two steps involved in a proof by mathematical induction.

**1. *Prove the base case.* In this step, you establish that the formula holds for the basis value, which is typically 0 or 1.**

**2. *Prove the inductive case.* In this step, you show that whenever the formula holds for *N,* it also holds for *N* + 1.**

24. In your own words, describe the relationship between recursion and mathematical induction.

**In a way, mathematical induction and recursion look at the same process from opposite directions. In mathematical induction, you start by proving simple cases and then use your existing knowledge to prove that the same principle applies for successively larger cases. In a recursive solution, you start with a complex problem and the break it down into successively simpler cases until you obtain problems that are simple enough to solve immediately.**