From our analysis, we determined the following:

- The quad partition algorithm runs in approximately $\Theta(n^{1.585})$ time.
- The binary partition with linear search algorithm runs in $\Theta(n\log(n))$ time.
- The binary partition with binary search algorithm runs in $\Theta(n)$ time.

Thus, the third algorithm, the binary partition with binary search, is the one that is asymptotically the fastest. From a performance perspective, this algorithm should be chosen to solve the problem. This indeed turns out to be the right choice. Here is the runtime of each algorithm for 1 million searches on a 100x100 matrix. The quad partition algorithm is the slowest, the binary partition with linear search algorithm is in the middle, and the binary partition with binary search algorithm is the fastest:

Algorithm	Complexity	Runtime
Quad Partition	$\Theta(n^{1.585})$	17.33 seconds
Binary Partition with Linear Search	$\Theta(n\log(n))$	10.93 seconds
Binary Partition with Binary Search	$\Theta(n)$	6.56 seconds

Just by looking at these three recursive algorithms, it may not seem apparent that the runtime saved by removing a single recursive call is worth an extra search down the middle column. However, from our analysis, we discovered that this extra work is worthwhile, as the removal of this one recursive call dropped our complexity class from polynomial to linear. Even doing an inefficient $\Theta(n)$ linear search down the middle column is more efficient than making a third recursive call.

This is the power of big-O analysis; we can use it to analyze different approaches to a problem before we start implementation! However, expectations do not always mirror reality — even if the correct algorithm is chosen, there may be factors that cause our runtimes to differ from what we expect. Performance tools like `perf`, which can identify the percentage of total time you are spending on an operation, can be used to debug these issues. Furthermore, big-O only dictates how runtime scales, not actual runtime itself. An algorithm that takes n steps will be twice as fast as an algorithm that takes $2n$ steps, even if both algorithms are $\Theta(n)$. And lastly, finding an efficient algorithm is only half the battle; you must also implement it optimally, using the correct choice of data structures. A suboptimal choice of data structure can worsen the performance of an otherwise efficient algorithm! We will begin exploring different data structures in the following chapters.

Remark: EECS 281 is not a math class! All of the fancy math equations you see in this chapter, especially for iterative substitution problems, are provided for completeness. You do *not* need to remember any summation formulas or identities for this class; everything you need should be given to you, unless otherwise specified.

Chapter 5 Practice Exercises

Disclaimer: These practice questions are not official and may not have been vetted for course material. You may use these for practice, but you should prioritize official resources from current staff for a more accurate depiction of what you need to know for assignments and exams.

1. Which of the following statements on recursion is/are **TRUE**?

   I. A tail recursive function is a linear recursive function where the recursive call is the final instruction of the function.
   II. Since tail recursive functions do not need to store local variables on a stack frame, all tail recursive functions use $\Theta(1)$ auxiliary space.
   III. The auxiliary space of a non-tail recursive function will always be the same as its time complexity, since all the recursive calls that are evaluated must be placed on the program stack.

   A) I only
   B) II only
   C) I and II only
   D) I and III only
   E) I, II, and III

2. What is the auxiliary space complexity of the function `foo()`, with respect to the input size n?

```
int32_t foo(int32_t n) {
  if (n <= 1) {
    return 1;
  } // if
  return n + foo(n / 2);
} // foo()
```

   A) $\Theta(1)$
   B) $\Theta(\log(n))$
   C) $\Theta(n)$
   D) $\Theta(n\log(n))$
   E) $\Theta(n^2)$

3. What is the auxiliary space complexity of the function `foo()`, with respect to the input size n?

```
int32_t foo(int32_t n) {
  if (n <= 1) {
    return 1;
  } // if
  return foo(n / 2);
} // foo()
```

A) $\Theta(1)$
B) $\Theta(\log(n))$
C) $\Theta(n)$
D) $\Theta(n\log(n))$
E) $\Theta(n^2)$

4. Consider the following two functions, `foo()` and `bar()`:

```
int32_t foo(int32_t n) {
  if (n <= 1) {
    return n;
  } // if
  return foo(n - 2) + foo(n - 1);
} // foo()

int32_t bar(int32_t n, int32_t prev = 0, int32_t curr = 1) {
  if (n == 0) {
    return prev;
  } // if
  if (n == 1) {
    return curr;
  } // if
  return bar(n - 1, curr, prev + curr);
} // bar()
```

Which of the following statements is/are **TRUE** regarding these two functions?
I. Both `foo(n)` and `bar(n)` would return the same value for all positive integer values of `n`.
II. `bar()` is tail recursive, while `foo()` is not.
III. The auxiliary space used by `foo()` is $\Theta(n)$, while the auxiliary space used by `bar()` is $\Theta(1)$.

A) I only
B) II only
C) I and II only
D) II and III only
E) I, II, and III

5. Given the function below, calculate the recurrence relation. Assume that `bar(n)` runs in $\log(n)$ time.

```
void foo(int32_t n) {
  if (n == 1) {
    return;
  } // if

  foo(n / 7);
  int32_t sq = n * n;

  for (int32_t i = 0; i < sq; ++i) {
    for (int32_t j = 0; j < n; ++j) {
      bar(n);
    } // for j
  } // for i

  for (int32_t k = 0; k < n; ++k) {
    foo(n / 3);
  } // for k

  bar(sq * sq);
} // foo()
```

A) $T(n) = T\left(\frac{n}{7}\right) + n^2\log(n) + nT\left(\frac{n}{3}\right) + \log(n)$
B) $T(n) = T\left(\frac{n}{7}\right) + n^2\log(n) + nT\left(\frac{n}{3}\right) + 2\log(n)$
C) $T(n) = T\left(\frac{n}{7}\right) + n^3\log(n) + nT\left(\frac{n}{3}\right) + \log(n)$
D) $T(n) = T\left(\frac{n}{7}\right) + n^3\log(n) + nT\left(\frac{n}{3}\right) + 2\log(n)$
E) $T(n) = T\left(\frac{n}{7}\right) + n^3\log(n) + nT\left(\frac{n}{3}\right) + 4\log(n)$

6. Consider the recurrence relation $T(n) = T(n-2) + 3^n$. If we solve this recurrence using the substitution process, which of the following is a valid first step in the expansion?
   A) $T(n) = T(n-2) + 3^n = T(n-4) + 3^{2n} = T(n-6) + 3^{3n} = \dots$
   B) $T(n) = T(n-2) + 3^n = T(n-4) + 3^n + 3^n = T(n-6) + 3^n + 3^n + 3^n = \dots$
   C) $T(n) = T(n-2) + 3^n = T(n-4) + 3^n + 3^{n-2} = T(n-6) + 3^n + 3^{n-2} + 3^{n-4} = \dots$
   D) $T(n) = T(n-2) + 3^n = T(n-2) + T(n-4) + 3^n = T(n-2) + T(n-4) + T(n-6) + 3^n + \dots$
   E) None of the above

7. Which of the following recurrence relations can one solve by applying the Master Theorem?
   A) $T(n) = nT\left(\frac{n}{3}\right) + \Theta(n^2)$
   B) $T(n) = 24T\left(\frac{n}{6}\right) + 32T\left(\frac{n}{8}\right) + \Theta(n^2)$
   C) $T(n) = 11T\left(\frac{n}{13}\right) + \Theta(\sqrt{n})$
   D) $T(n) = \frac{1}{2}T\left(\frac{n}{3}\right) + \Theta(n^2)$
   E) $T(n) = 2T(n-1) + \Theta(n^3)$

8. Which of the following recurrence relations has the closed form expression $T(n) = \Theta(n^2)$? Assume that $T(1) = 1$ for all the choices.
   A) $T(n) = 2T\left(\frac{n}{4}\right) + n^5$
   B) $T(n) = T\left(\frac{n}{2}\right) + \left(\frac{n}{2}\right)$
   C) $T(n) = 2T\left(\frac{n}{4}\right) + \sqrt{n}$
   D) $T(n) = T(n-1) + n + 4$
   E) None of the above

9. Solve the following recurrence.

$$T(n) = \begin{cases} 1, & \text{if } n = 0 \\ T(n-1) + 2, & \text{if } n > 0 \end{cases}$$

   A) $T(n) = n$
   B) $T(n) = 2n + 1$
   C) $T(n) = 2n - 1$
   D) $T(n) = 2n$
   E) $T(n) = 2$

10. What is the time complexity of the following recurrence relation?

$$T(n) = \begin{cases} 1, & \text{if } n = 1 \\ 3T(n-1) + 1, & \text{if } n > 1 \end{cases}$$

   A) $\Theta(n)$
   B) $\Theta(n^2)$
   C) $\Theta(n^3)$
   D) $\Theta(2^n)$
   E) $\Theta(3^n)$

11. What is the time complexity of the following recurrence relation?

$$T(n) = \begin{cases} 1, & \text{if } n = 1 \\ 4T\left(\frac{n}{2}\right) + 16n + n^2 + 1, & \text{if } n > 1 \end{cases}$$

   A) $\Theta(n)$
   B) $\Theta(n\log(n))$
   C) $\Theta(n^2)$
   D) $\Theta(n^2\log(n))$
   E) $\Theta(n^4)$

12. What is the time complexity of the following recurrence relation?

$$T(n) = \begin{cases} 1, & \text{if } n = 1 \\ 5T\left(\frac{n}{25}\right) + 5\sqrt{n} + 1, & \text{if } n > 1 \end{cases}$$

A) $\Theta(\sqrt{n})$
B) $\Theta(\sqrt{n}\log(n))$
C) $\Theta(n)$
D) $\Theta(n^5 \log(n))$
E) $\Theta(n^5)$

13. What is the time complexity of the following recurrence relation?

$$T(n) = \begin{cases} 1, & \text{if } n = 1 \\ 729T\left(\frac{n}{9}\right) + 3n\sqrt[3]{n} + 81n + 1, & \text{if } n > 1 \end{cases}$$

A) $\Theta(\sqrt[3]{n}\log(n))$
B) $\Theta(\sqrt[3]{n})$
C) $\Theta(n)$
D) $\Theta(n\sqrt[3]{n})$
E) $\Theta(n^3)$

14. Consider the following recurrence:

$$T(n) = \begin{cases} 1, & \text{if } n = 1 \\ aT\left(\frac{n}{b}\right) + 19\sqrt{n} + 6n, & \text{if } n > 1 \end{cases}$$

You are told that the time complexity of $T(n)$ is $\Theta(n)$. Which of the following must be true?
A) $a = 1$
B) $a = b$
C) $a < b$
D) $a > b$
E) None of the above

15. Suppose you are given the following incomplete recurrence relation for a recursive function:

$$T(n) = T(?) + 281n$$

You are told that this recurrence is **directly solvable** by the Master Theorem. Knowing this information, which of the following statements must be true?
A) $T(n) \to \Theta(1)$
B) $T(n) \to \Theta(n)$
C) $T(n) \to \Theta(n\log(n))$
D) $T(n) \to \Theta(n^2)$
E) The final complexity of $T(n)$ cannot be determined from the information given

16. Consider the following function `foo()` below.

```cpp
int32_t foo(int32_t n) {
  if (n <= 0) {
    return 0;
  } // if

  int32_t count = 0;
  for (int32_t i = 0; i < n; ++i) {
    for (int32_t j = 0; j < n; ++j) {
      ++count;
    } // for j
  } // for i

  for (int32_t k = 0; k < 8; ++k) {
    count += foo(n / 2);
  } // for k

  return count;
} // foo()
```

What is the time complexity of `foo()`, in terms of the input size n?
A) $\Theta(n)$
B) $\Theta(n^2)$
C) $\Theta(n^2\log(n)))$
D) $\Theta(n^3)$
E) $\Theta(n^3\log(n))$

17. Consider the following function `foo()` below.

```cpp
int32_t foo(int32_t n) {
  if (n <= 0) {
    return 0;
  } // if

  int32_t count = 0;
  for (int32_t i = 0; i < n; i *= 2) {
    ++count;
  } // for i

  count += foo(n - 1);
  return count;
} // foo()
```

What is the time complexity of `foo()`, in terms of the input size n?
A) $\Theta(\log(n))$
B) $\Theta(n)$
C) $\Theta(n\log(n)))$
D) $\Theta(n^2)$
E) $\Theta(n^2\log(n))$

18. Consider the following function `foo()` below.

```cpp
uint64_t foo(int32_t n) {
  uint64_t count = 1;
  if (n == 0) {
    return count;
  } // if

  for (uint64_t i = 1; i <= n; ++i) {
    count += foo(n - 1);
  } // for i

  return count;
} // foo()
```

What is the time complexity of `foo()`, in terms of the input size n?
A) $\Theta(n)$
B) $\Theta(n^2)$
C) $\Theta(2^n)$
D) $\Theta(n!)$
E) $\Theta(n^n)$

19. Consider the following function `foo()` below.

```
int32_t foo(int32_t n) {
  if (n <= 1) {
    return 1;
  } // if
  return foo(n - 1) + foo(n - 1);
} // foo()
```

What is the time complexity of `foo()`, in terms of the input size n?
A) $\Theta(n)$
B) $\Theta(n^2)$
C) $\Theta(2^n)$
D) $\Theta(n!)$
E) $\Theta(n^n)$

20. Consider the following function `foo()` below.

```
int32_t foo(int32_t n) {
  if (n <= 1) {
    return 1;
  } // if
  return foo(n / 2) + foo(n / 2);
} // foo()
```

What is the time complexity of `foo()`, in terms of the input size n?
A) $\Theta(n)$
B) $\Theta(n^2)$
C) $\Theta(2^n)$
D) $\Theta(n!)$
E) $\Theta(n^n)$

21. Consider the following function `foo()` below.

```
int32_t foo(int32_t n) {
  if (n <= 1) {
    return 1;
  } // if

  int32_t count = 0;
  count += foo(n / 4);

  for (int32_t i = 0; i < n; ++i) {
    for (int32_t j = 0; j < n; ++j) {
      ++count;
    } // for j
  } // for i

  count += foo(n / 2);

  for (int32_t k = 0; k < n; ++k) {
    ++count;
  } // for k

  return count + foo(n / 2);
} // foo()
```

What is the time complexity of `foo()`, in terms of the input size n?
A) $\Theta(n)$
B) $\Theta(n\log(n))$
C) $\Theta(n^2)$
D) $\Theta(n^2\log(n))$
E) $\Theta(n^3)$

22. Consider the following function `search_for_life()` below.

```
void search_for_life_helper(int32_t arr[], size_t n, size_t idx) {
  if (idx < 0 || idx >= n) {
    return;
  } // if

  if (arr[idx] == 42) {
    std::cout << "Found the answer to life\n";
  } // if
  else {
    std::cout << "Failure\n";
  } // else

  search_for_life_helper(arr, n, idx + 1);
} // search_for_life_helper()

void search_for_life(int32_t arr[], size_t n) {
  for (size_t i = 1; i < n; i *= 2) {
    search_for_life_helper(arr, n, i);
  } // for i
} // search_for_life()
```

What is the time complexity of `foo()`, in terms of the input size n?

A) $\Theta(n)$
B) $\Theta(n\log(n))$
C) $\Theta(n^2)$
D) $\Theta(n^2\log(n))$
E) $\Theta(2^n)$

23. Consider the following functions below.

```
void foo(int32_t s) {
  for (int32_t i = s; i > 0; i /= 2) {
    std::cout << "EECS 281\n";
  } // for i
} // foo()

void bar(int32_t k) {
  if (k <= 1) {
    return;
  } // if

  foo(k);
  bar(k - 1);
} // bar()

void baz(int32_t n) {
  for (int32_t i = 0; i < n; ++i) {
    bar(n);
  } // for i

  if (n > 281) {
    foo(n);
  } // if
  else {
    bar(n - 1);
  } // else
} // baz()
```

What is the time complexity of `baz()`, in terms of the input size n?

A) $\Theta(n\log(n))$
B) $\Theta(n^2)$
C) $\Theta(n^2\log(n))$
D) $\Theta(n^3)$
E) $\Theta(n^3\log(n))$

Chapter 5 Exercise Solutions

1. **The correct answer is (A).** Only statement I is true. Statement II is false because a tail recursive function may still use additional memory that is not part of the stack frames. Statement III is false because not all recursive calls in a non-tail recursive function will need to be evaluated at once (and thus do not need to be stored on the program stack together).

2. **The correct answer is (B).** This function is not tail recursive, so additional stack frames will be needed for the recursive call on line 5. How many stack frames are needed? The number of stack frames needed is equal to the number of recursive calls we will encounter before reaching the base case; since we are halving n with each recursive call, this comes out to $\Theta(\log(n))$. The function uses constant auxiliary space outside the recursive call, so we can conclude that the auxiliary space usage of the entire function is also $\Theta(\log(n))$.

3. **The correct answer is (A).** This function is tail recursive, so the number of stack frames needed does not depend on the input size n.

4. **The correct answer is (E).** Both functions can be used to calculate the n^{th} Fibonacci number; the main difference is that `bar()` is tail recursive (via an accumulator argument) and `foo()` is not. This indicates that statements I and II are true. Statement II is also true because the tail recursiveness of `bar()` allows it to use constant auxiliary space rather than the linear number of stack frames required by `foo()`.

5. **The correct answer is (E).** On line 6, we perform a recursive call with input size $n/7$, which adds a $T(n/7)$ term to our recurrence. On lines 9-13, we perform a loop that executes `bar(n)` a total of n^3 times; since we are given that `bar(n)` runs in $\log(n)$ time, the total contribution of this loop can be expressed as $n^3\log(n)$. On lines 15-17, we perform a loop that makes a recursive call with input size $n/3$ a total of n times, for a total contribution of $nT(n/3)$. Lastly, on line 19, we call `bar()` with an input size of $n^2 \times n^2 = n^4$, which contributes $\log(n^4) = 4\log(n)$ work. Adding all of these terms together, we get the recurrence relation matching option (E).

6. **The correct answer is (C).** We know that $T(n-2) = T(n-4) + 3^{n-2}$, so the substitution after $T(n-2) + 3^n$ is $T(n-4) + 3^n + 3^{n-2}$. When substituting, make sure to substitute all instances of n in the expression (e.g., $T(n-2)$ is $T(n-4) + 3^{n-2}$ and not $T(n-4) + 3^n$).

7. **The correct answer is (C).** Option (A) does not work because the value of a cannot depend on the input size n. Option (B) does not work because it is not in the correct form (two different recursive calls that split the input into different sizes). Option (D) does not work because $a < 1$. Option (E) does not work because it is also not in the correct form (the input size must be divided). Only option (C) works, where $a = 11$, $b = 13$, and $c = 1/2$.

8. **The correct answer is (D).** For option (A), we can use the Master Theorem for $a = 2$, $b = 4$, and $c = 5$ to conclude that the recurrence has a closed form that is $\Theta(n^5)$. For option (B), we can use the Master Theorem for $a = 1$, $b = 2$, and $c = 1$ to conclude that the recurrence has a closed form that is $\Theta(n)$. For option (C), we can use the Master Theorem for $a = 2$, $c = 4$, and $c = 1/2$ to conclude that the recurrence has a closed form that is $\Theta(\sqrt{n}\log(n))$. This leaves choice (D), which we can prove is $\Theta(n^2)$ using iterative substitution:

$$\begin{aligned} T(n) &= T(n-1) + n + 4 = [T(n-2) + (n-1) + 4] + n + 4 = [[T(n-3) + (n-2) + 4] + (n-1) + 4] + n + 4 \\ &= \dots \\ &= T(n-k) + kn + 4k - \sum_{i=0}^{k-1} i \end{aligned}$$

The base case happens when $n - k = 1$, so we can set $k = n - 1$ to solve for the overall time complexity of the recurrence:

$$T(n) = T(1) + (n-1)n + 4(n-1) - \sum_{i=0}^{n-2} i = 1 + n^2 - n + 4n - 4 - \frac{(n-1)(n-2)}{2} = \frac{1}{2}n^2 + \frac{3}{2}n - 2 = \Theta(n^2)$$

9. **The correct answer is (B).** We can solve this recurrence using substitution:

$$\begin{aligned} T(n) &= T(n-1) + 2 = T(n-2) + 2 + 2 = T(n-3) + 2 + 2 + 2 \\ &= \dots \\ &= T(n-k) + 2k \end{aligned}$$

The base case happens when $n - k = 0$, so we can set $k = n$ to solve the overall recurrence:

$$T(n) = T(0) + 2n = 2n + 1$$

10. **The correct answer is (E).** We can solve this recurrence using iterative substitution:

$$\begin{aligned} T(n) &= 3T(n-1) + 1 = 3[3T(n-2) + 1] + 1 = 3^2T(n-2) + 3 + 1 = 3^2[3T(n-3) + 1] + 3 + 1 = 3^3T(n-3) + 9 + 3 + 1 \\ &= \dots \\ &= 3^kT(n-k) + \sum_{i=0}^{k-1} 3^i \end{aligned}$$

The base case happens when $n - k = 0$, so we can set $k = n$ to solve for the overall time complexity of the recurrence:

$$T(n) = 3^nT(0) + \sum_{i=0}^{n-1} 3^i = \Theta(3^n)$$

11. **The correct answer is (D).** We can use the Master Theorem for this recurrence. Here, a is 4, b is 2, and c is the exponent of the dominating term, or 2 in the case of n^2. Since $a = b^c$, we can conclude that $T(n) = \Theta(n^c \log(n)) = \Theta(n^2 \log(n))$.

12. **The correct answer is (B).** We can use the Master Theorem for this recurrence. Here, a is 5, b is 25, and c is 1/2. Since $a = b^c$, we can conclude that $T(n) = \Theta(n^c \log(n)) = \Theta(\sqrt{n}\log(n))$.

13. **The correct answer is (E).** We can use the Master Theorem for this recurrence. Here, a is 729, b is 9, and c is the exponent of the dominating term, or 4/3. Since $a > b^c$, we can conclude that $T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\log_9 729}) = \Theta(n^3)$.

14. **The correct answer is (C).** This is a problem we can solve using the Master Theorem. We know that c is equal to one since that is the exponent of the dominating term of $6n$. Thus, if $a = b$, then the complexity of $T(n)$ would have to be $\Theta(n^c \log(n)) = \Theta(n \log(n))$, which we can remove as a possibility. If $a < b$, the time complexity would be $\Theta(n^c) = \Theta(n)$, so this case would work. If $a > b$, the time complexity would be $\Theta(n^{\log_b a})$, which can only be $\Theta(n)$ if a and b were equal, so this case would not work. In conclusion, for the given recurrence to be $\Theta(n)$, then a must be less than b.

15. **The correct answer is (B).** In this case, we know that a and c are both 1, since we are told the recurrence is directly solvable by the Master Theorem. This actually limits the Master Theorem conditions we can possibly encounter. Since $b > 1$ is necessary for the Master Theorem to be used, we know for certain that a cannot be greater than or equal to b. This leaves us with the $a < b^c$ condition, which states that the complexity of the recurrence must be $\Theta(n^c) = \Theta(n)$.

16. **The correct answer is (D).** If we convert this function into a recurrence relation, we would get $T(n) = 8T(n/2) + n^2$ (from the eight recursive calls with half the input size on line 14, plus the quadratic work done in the nested loop on line 7). We can use the Master Theorem on this recurrence, with $a = 8$, $b = 2$, and $c = 2$. Since $a > b^c$, the time complexity of this function is $\Theta(n^{\log_2 8}) = \Theta(n^3)$.

17. **The correct answer is (C).** If we convert this function into a recurrence relation, we would get $T(n) = T(n-1) + \log(n)$ (from the recursive with input size $n-1$ on line 11, plus the logarithmic work done in the loop on line 7). This recurrence can be solved using substitution (note that $\Theta(\log(n!))$ is $\Theta(n\log(n))$ from chapter 4):

$$\begin{aligned}T(n) &= T(n-1)+\log(n) = T(n-2)+\log(n-1)+\log(n) = T(n-3)+\log(n-2)+\log(n-1)+\log(n)\\ &= \dots\\ &= 1+\log(1)+\log(2)+\dots+\log(n-1)+\log(n) = 1+\log(n!) = \Theta(n\log(n))\end{aligned}$$

18. **The correct answer is (D).** See example 5.7.

19. **The correct answer is (C).** The recurrence relation for this function is $T(n) = 2T(n-1)+1$, which can be solved using substitution as $\Theta(2^n)$ (see problem 10 for a very similar process).

20. **The correct answer is (A).** The recurrence relation for this function is $T(n) = 2T(n/2)+1$, which can be solved using the Master Theorem for $a = 2$, $b = 2$, and $c = 0$. Since $a > b^c$, the time complexity of this function is $\Theta(n^{\log_b a}) = \Theta(n^{\log_2 2}) = \Theta(n^1)$.

21. **The correct answer is (C).** The recurrence relation for this function is $T(n) = 2T(n/2)+T(n/4)+n^2+n$, which cannot be solved using the Master Theorem. One alternative method to solve this could be to use a recurrence tree (an optional concept detailed in this chapter). However, there is an interesting detail we can notice that allows us to avoid having to use recurrence trees: we know that the function would do more work if the $n/4$ recursive call on line 7 had instead been a $n/2$ recursive call. In that situation, the recurrence would have become $2T(n/2)+T(n/2)+n^2+n$, or $3T(n/2)+n^2+n$. This recurrence *is* solvable using the Master Theorem, with $a = 3$, $b = 2$, and $c = 2$ — here, $a < b^c$, so the time complexity would be $\Theta(n^2)$. Therefore, we know that the time complexity of the original given function cannot exceed $\Theta(n^2)$, since that would be the time complexity if the $n/4$ recursive call were replaced with a larger input size of $n/2$. However, we also know that the time complexity cannot be better than $\Theta(n^2)$ either due to the presence of the $\Theta(n^2)$ loop on line 9. Since the function's time complexity cannot be worse than $\Theta(n^2)$ but also cannot be better, the overall time complexity of `foo()` must be $\Theta(n^2)$.

22. **The correct answer is (B).** For any index value i, the recurrence relation of the helper function can be expressed as:

$$T(i) = \begin{cases} 1, & \text{if } i < 0 \text{ or } i \ge n \\ T(i+1)+1, & \text{otherwise} \end{cases}$$

The helper function is initially called with an input size of 1. Using substitution, we see that the time complexity of this helper call is $\Theta(n)$:

$$T(1) = T(2)+1 = (T(3)+1)+1 = (T(4)+1)+1+1 = \dots = T(n)+(n-1) = 1+(n-1) = n$$

In fact, we can see that the total work performed by the helper for a given index i is $n+1-i$. Since the helper is invoked once for every power of two up to n, the total work of this function must be:

$$T(n) = n+1-1+n+1-2+n+1-4+\dots+n+1-\log_2(n) = \log_2(n)\times(n+1) - \sum_{i=0}^{\log_2(n)} 2^i$$

Using the equation for the sum of a geometric series, we can conclude that:

$$T(n) = \log_2(n)\times(n+1) - \left(\frac{1-2^{\log_2(n)}}{1-2}\right) = \Theta(n\log(n)-n) = \Theta(n\log(n))$$

23. **The correct answer is (C).** The time complexity of `foo()` is $\Theta(\log(s))$, since it performs a constant time operation in a loop that halves the input from s to 0. As a result, the time complexity of `bar()` can be expressed using the recurrence $T(k) = T(k-1)+\Theta(k-1)$, which is $\Theta(k\log(k))$ (see question 17 for the same recurrence). The `baz()` function invokes `bar()` a total of n times in the loop on line 17, so the overall complexity of this loop is $n \times \Theta(n\log(n)) = \Theta(n^2\log(n))$. Note that the function calls on line 22 and 25 both contribute to lower order terms, so the overall time complexity of `baz()` is $\Theta(n^2\log(n))$.