Chapter 4 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 provides an explanation for why it may be better to measure a program's relative efficiency by analyzing how its runtime scales by input size rather than its pure runtime?
   - **I.** If you compare two algorithms based on pure runtime, the superior algorithm may end up being measured as less efficient if more processes were running in the background when it was run.
   - **II.** For input sizes that are large enough, the rate of growth of a program's runtime with respect to input size is independent of most external factors, making it a consistent benchmark for comparing the growth of different algorithms.
   - **III.** Runtime measured with respect to input size can be expressed mathematically using big-O notation, which makes it easier to categorize and compare different algorithms with different orders of growth.

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

2. Consider the following four statements regarding algorithm complexities:
   - **I.** An algorithm with a $\Theta(n^2)$ time complexity will always run faster than an algorithm with a $\Theta(n\log(n))$ time complexity.
   - **II.** An algorithm with a $\Theta(n\log(n))$ time complexity will always run faster than an algorithm with a $\Theta(n^2)$ time complexity.
   - **III.** An algorithm with a $\Theta(n^2)$ time complexity will always run faster than an algorithm with a $\Theta(n!)$ time complexity.
   - **IV.** An algorithm with a $\Theta(n!)$ time complexity will always run faster than an algorithm with a $\Theta(n^2)$ time complexity.

   How many of these statements is/are **TRUE**?
   A) 0
   B) 1
   C) 2
   D) 3
   E) 4

3. Given a function f, which of the following best defines $O(f)$ using big-O notation?
   A) $O(f)$ is the set of functions that grow at least as slowly as f
   B) $O(f)$ is the set of functions that grow at least as quickly as f
   C) $O(f)$ is the set of functions that grow exactly as quickly as f
   D) $O(f)$ is the set of functions that do not grow more slowly than f
   E) $O(f)$ is the set of functions that do not grow more quickly than f

4. You are measuring the asymptotic time complexity of a function *foo* by counting steps. Suppose you determine that the total number of steps taken by *foo* with respect to input size n can be expressed as:

$$T(n) = 53\log(n) + 79n + 12$$

   What of the following statements is **TRUE**?
   A) *foo* has a time complexity of $\Theta(53\log(n))$
   B) *foo* has a time complexity of $\Theta(\log(n))$
   C) *foo* has a time complexity of $\Theta(n)$
   D) *foo* has a time complexity of $\Theta(n\log(n))$
   E) More than one of the above

5. You are given five different algorithms, each with a different time complexity:
   - **I.** Algorithm A has a time complexity of $\Theta(n!)$
   - **II.** Algorithm B has a time complexity of $\Theta(2^n)$
   - **III.** Algorithm C has a time complexity of $\Theta(n^3)$
   - **IV.** Algorithm D has a time complexity of $\Theta(n^3\log(n))$
   - **V.** Algorithm E has a time complexity of $\Theta(n^n)$

   Which of the following correctly ranks these algorithms in order of increasing time complexity?
   A) D, C, B, A, E
   B) B, C, D, A, E
   C) B, D, C, A, E
   D) C, D, B, A, E
   E) C, D, B, E, A

6. You have n billiard balls, all of which have the same weight except for one that is heavier than all the others. Which of the following is a $\Theta(\log(n))$ solution for finding the billiard ball with a heavier weight using only a single balance?

   A) Randomly pick two balls at a time and place them on each side of the balance until you find an imbalance
   B) Place one ball on one side of the balance, and compare its weight with each of the other balls using the other side of the balance
   C) Split the balls into two halves, place one half on one side of the balance and the other half on the other, and check to see which half is heavier; continue this process with the balls on the heavier side until the target ball is found
   D) Split the balls into three groups, place one third on one side and another third on the other, and check which group the heavier ball is in (if the two groups on the scale have the same weight, then the heavier ball must be in the group not on the scale); continue this process with the group that includes the heavier ball until the target ball is found
   E) More than one of the above

7. You just completed a project after a long day of work, and good news: your code appears to run correctly with the provided test files! However, when you plot out runtimes with your own test files, you notice that there exists a $\Theta(n)$ relationship even though you had implemented a $\Theta(\log(n))$ algorithm. Which of the following could be a reason for this discrepancy?
   A) There were too many programs running in parallel with your program
   B) $\Theta(n)$ algorithms can also have a time complexity of $\Theta(\log(n))$
   C) The tests that you used were too large for the actual relationship to be revealed
   D) The tests that you used were too small for the actual relationship to be revealed
   E) None of the above

8. Your friend is working on a project, which they implemented using a $\Theta(n^2)$ algorithm. However, when they ran a test file on their program, they noticed that the program instead ran in $\Theta(n)$ time. All else equal, which of the following is **NOT** a possible reason for this discrepancy?
   A) Your friend accidentally ran the test case with a different algorithm
   B) Your friend incorrectly analyzed the time complexity of their algorithm
   C) Your friend's test inputs exposed worst-case behavior
   D) Your friend's test case input size was too small
   E) None of the above

9. Suppose you have three functions f, g, and h. Consider the following statements:
   I. f must either be $O(g)$, $\Theta(g)$, or $\Omega(g)$.
   II. If $f+g$ is $O(h)$, then either f or g must also be $O(h)$.
   III. If $f \times g$ is $O(h)$, then either f or g must also be $O(h)$.
   IV. There exists no f such that f is $\Omega(f \times g)$, where $f \times g$ is the product of two functions f and g.

   How many of these statements is/are **TRUE**?
   A) 0
   B) 1
   C) 2
   D) 3
   E) 4

10. Which of the following statements is **FALSE**?
   A) $\log(n^3 9^n)$ is $\Theta(9^n)$
   B) $\log(n^5 4^{3n})$ is $\Theta(n)$
   C) $\log(n!)$ is $\Theta(n\log(n))$
   D) $\log_3(n)$ is $\Theta(\log_2(n))$
   E) $\log(n^2)$ is $\Theta(\log(n))$

11. Which of the following strategies can be used to improve the worst-case *asymptotic* time complexity of an algorithm?
   A) Reducing the size of the input that is passed into the algorithm
   B) Increasing the size of the input that is passed into the algorithm
   C) Running the algorithm on a faster machine
   D) Compiling the algorithm using the `-O3` compiler flag
   E) None of the above

12. Consider the following function implementation:

```
1  int32_t foo(int32_t n) {
2    int32_t a = helper_a(n);
3    int32_t b = helper_b(n);
4    int32_t c = helper_c(n);
5    return a + b + c;
6  } // foo()
```

For an input size of n, the `helper_a()` function has an asymptotic time complexity of $\Theta(n^2)$, the `helper_b()` function has an asymptotic time complexity of $\Theta(n)$, and the `helper_c()` function has an asymptotic time complexity of $\Theta(n\log(n))$. Which of the following modifications would *change* the asymptotic time complexity class of the function `foo()`?

A) Improving the asymptotic time complexity of `helper_a()` from $\Theta(n^2)$ to $\Theta(n\log(n))$
B) Improving the asymptotic time complexity of `helper_a()` from $\Theta(n^2)$ to $\Theta(0.5n^2)$
C) Improving the asymptotic time complexity of `helper_b()` from $\Theta(n)$ to $\Theta(1)$
D) Improving the asymptotic time complexity of `helper_c()` from $\Theta(n\log(n))$ to $\Theta(\log(n))$
E) More than one of the above

13. You are given three algorithms A, B, and C, each with runtimes that can be modeled using the functions $A(n)$, $B(n)$, $C(n)$ for input size n. You know that $A(n) = O(B(n))$ and $A(n) = \Omega(C(n))$. Which of the following statements **CANNOT** be true?

A) $B(n) = \Theta(C(n))$
B) $C(n) = \Omega(A(n))$
C) $B(n) = \Omega(C(n))$
D) $B(n) = O(C(n))$
E) None of the above (i.e., all of the above statements can be true)

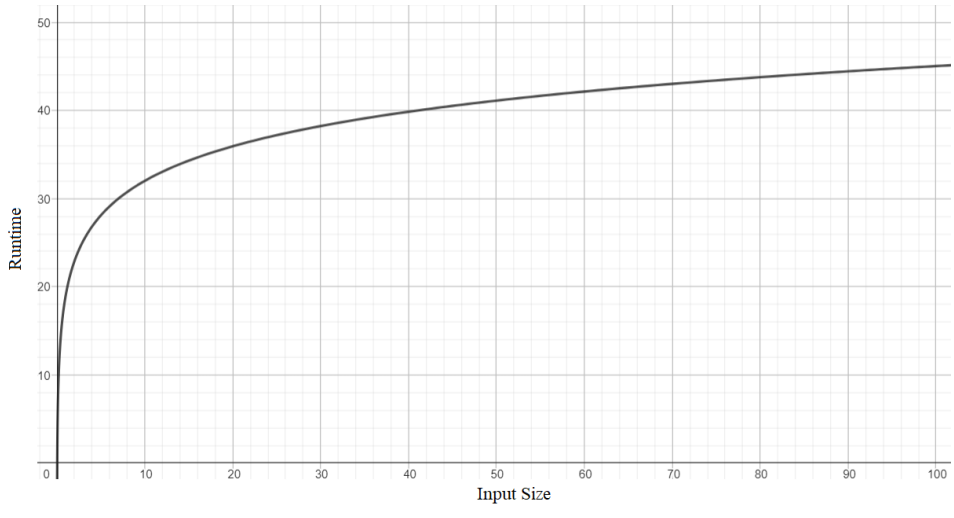
14. You have four algorithms that accomplish the same task, and you want to know which algorithm to use. Memory is not an issue, so you want to make your decision based on time efficiency alone. To make your decision, you run each algorithm once on the same machine at the exact same time. The same input size is used for all the algorithms. You obtain the following results from this experiment:

Algorithm	Runtime (ms)
A	0.011
B	0.015
C	0.006
D	0.019

Using just the information collected from this experiment, which algorithm should you select?

A) Algorithm A
B) Algorithm B
C) Algorithm C
D) Algorithm D
E) The algorithm you should select should not be determined using the results of this experiment

15. You are given a function `foo()`, and you graph its runtime with respect to different input sizes. After running multiple experiments with differing inputs, you end with the following relationship between runtime and input size:



What is the asymptotic time complexity of `foo()` with respect to input size n?

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

16. You are given three strictly increasing functions $A(n)$, $B(n)$, and $C(n)$ for input size n. Suppose you know that $A(n)$ is $O(B(n))$ and $A(n)$ is $\Omega(C(n))$. Which of the following statements must also be **TRUE**?
    A) $B(n)$ is $O(C(n))$
    B) $B(n)$ is $\Omega(C(n))$
    C) $B(n)$ is $\Theta(C(n))$
    D) $B(n)$ is $\Theta(A(n))$
    E) None of the above

17. Which of the following correctly orders these complexity classes in order from slowest to fastest growth?
    I. $\Theta\left(\sqrt{\log(n)}\right)$
    II. $\Theta(\log(n))$
    III. $\Theta(\log^2(n))$
    IV. $\Theta(\log(\log(n)))$

    A) I, IV, II, III
    B) I, IV, III, II
    C) I, II, IV, III
    D) IV, I, II, III
    E) IV, I, III, II

18. Suppose you are given a function $f(n)$ that you know is $\Theta(n^2 \log(n))$. Which of the following statements must also be **TRUE** about $f(n)$?
    A) $f(n)$ is $\Theta(n^2)$
    B) $f(n)$ is $\Theta(n^2 + n)$
    C) $f(n)$ is $\Omega(n^2 + \log(n))$
    D) $f(n)$ is $O(n^2)$
    E) More than one of the above

19. For which of the following pairs of $f(n)$ and $g(n)$ is $f(n) = \Theta(g(n))$? Note: e is a mathematical constant with approximate value 2.71828…
    I. $f(n) = 2^{2n}$
    $g(n) = 2^n$
    II. $f(n) = 2^{n \log_2(n)}$
    $g(n) = n^n$
    III. $f(n) = 2e^{n/2}$
    $g(n) = e^n$

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

20. Which one of the following statements is **TRUE**?
    A) $\log(n+m)$ is $O(\log(n) + \log(m))$
    B) $n!$ is $O((n-1)!)$
    C) 2^{n+m} is $O(2^n + 2^m)$
    D) $\sin(n)$ is $O(\cos(n))$
    E) More than one of the above

21. If $f(n)$ and $g(n)$ are real-valued functions such that $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$, which of the following conclusions are valid?
    A) $f(n)$ is $O(g(n))$
    B) $f(n)$ is $\Theta(g(n))$
    C) $f(n)$ is $\Omega(g(n))$
    D) More than one of the above
    E) None of the above conclusions can be made from the given information

22. You are given an algorithm whose runtime can be expressed in the form $f(n) = C^n$ for input size n and some unknown constant C. From analysis, you discover that it takes T time for this algorithm to solve a problem with input size S, and $9T$ time for this algorithm to solve a problem with input size $S+6$. Knowing this, what is the value of the constant C?
    A) $\sqrt[3]{3}$
    B) $\sqrt{3}$
    C) $3/2$
    D) 2
    E) 3

23. You are given an algorithm whose runtime can be expressed in the form $f(n) = \log_2(n)$ for input size n. If it takes T time to solve a problem with input size S, a problem of what input size can be solved in time $T+1$?
    A) $\log_2(S)$
    B) $S + \log_2(S)$
    C) $2S$
    D) $2\log_2(S)$
    E) $2S + 2\log_2(S)$

24. You are given an algorithm whose runtime can be expressed in the form $f(n) = n^2$ for input size n. If it takes T time to solve a problem with input size S, a problem of what input size can be solved in time $8T$?
    A) $2S$
    B) $(2\sqrt{2})S$
    C) $3S$
    D) $4S$
    E) $(4\sqrt{2})S$

25. Consider the following function implementation:

```
1  void foo(int32_t m, int32_t n) {
2    for (int32_t i = 0; i < m; ++i) {
3      std::cout << "EECS 281\n";
4    } // for i
5
6    for (int32_t j = 0; j < n; ++j) {
7      std::cout << "EECS 281\n";
8    } // for j
9  } // foo()
```

    What is the time complexity of this function in terms of m and n?
    A) $\Theta(m)$
    B) $\Theta(n)$
    C) $\Theta(m+n)$
    D) $\Theta(mn)$
    E) None of the above

26. Consider the following function implementation:

```
1  void foo(int32_t m, int32_t n) {
2    for (int32_t i = 0; i < m; ++i) {
3      for (int32_t j = 0; j < n; ++j) {
4        std::cout << "EECS 281\n";
5      } // for j
6    } // for i
7  } // foo()
```

    What is the time complexity of this function in terms of m and n?
    A) $\Theta(m)$
    B) $\Theta(n)$
    C) $\Theta(m+n)$
    D) $\Theta(mn)$
    E) None of the above

27. Consider the following function implementation:

```
1  void foo(int32_t m, int32_t n, int32_t p) {
2    for (int32_t k = 0; k < p; ++k) {
3      for (int32_t i = 0; i < m; ++i) {
4        std::cout << "EECS 281\n";
5      } // for i
6
7      for (int32_t j = 0; j < n; ++j) {
8        std::cout << "EECS 281\n";
9      } // for j
10   } // for k
11 } // foo()
```

What is the time complexity of this function in terms of m, n, and p?

A) $\Theta(mp)$
B) $\Theta(np)$
C) $\Theta(mp+np)$
D) $\Theta(mnp)$
E) None of the above

28. Consider the following function implementation:

```
1  int32_t foo(int32_t n) {
2    int32_t count = 0;
3    for (int32_t i = 0; i < n; ++i) {
4      for (int32_t j = n; j > 0; --j) {
5        ++count;
6      } // for j
7    } // for i
8
9    return count;
10 } // foo()
```

What is the time complexity of this function in terms of n?

A) $\Theta(\log(n))$
B) $\Theta(n)$
C) $\Theta(n\log(n))$
D) $\Theta(n^2)$
E) None of the above

29. Consider the following function implementation:

```
1  int32_t foo(int32_t n) {
2    int32_t count = 0;
3    int32_t root = static_cast<int32_t>(std::floor(std::sqrt(n)));
4    for (int32_t i = n / 2; i < n; ++i) {
5      for (int32_t j = 1; j < n; j *= 2) {
6        for (int32_t k = 0; k < n; k += root) {
7          ++count;
8        } // for k
9      } // for j
10   } // for i
11
12   return count;
13 } // foo()
```

What is the time complexity of this function in terms of n? Note that `std::floor(x)` computes the largest integer value not greater than `x`, and `std::sqrt(x)` computes the square root of `x`.

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

30. Consider the following function implementation:

```
1  int32_t foo(int32_t n) {
2    int32_t count = 0;
3    while (n > 1) {
4      n /= 2;
5      ++count;
6    }  // while
7
8    return count;
9  } // foo()
```

What is the time complexity of this function in terms of n?

A) $\Theta(\log(n))$
B) $\Theta(n)$
C) $\Theta(n\log(n))$
D) $\Theta(n^2)$
E) None of the above

31. Consider the following function implementation:

```
1  void foo(int32_t n) {
2    int32_t k = 0;
3    while (n > 1) {
4      for (int32_t i = 0; i <= n; ++i) {
5        ++k;
6      } // for i
7
8      n /= 2;
9    } // while
10 } // foo()
```

What is the time complexity of this function in terms of n? Note: you may find the equation for the sum of a geometric series helpful, where a is the first term and r is the common ratio:

$$S_n = \frac{a(1-r^n)}{1-r}, r \neq 1$$

A) $\Theta(\log^2(n))$
B) $\Theta(n)$
C) $\Theta(n\log(n))$
D) $\Theta(n^2)$
E) None of the above

32. Consider the following function implementation:

```
1  void foo(int32_t m, int32_t n) {
2    int32_t k = 0;
3    while (n > m) {
4      for (int32_t i = 0; i < m; i *= 3) {
5        ++k;
6      } // for i
7
8      n /= 2;
9    } // while
10 } // foo()
```

What is the time complexity of this function in terms of m and n?

A) $\Theta(\log(m/n)\log(n))$
B) $\Theta(\log(n/m)\log(m))$
C) $\Theta(n\log(m))$
D) $\Theta(\log(m)\log(n))$
E) None of the above

33. Consider the following function, which identifies whether an array has any duplicate values. Let n represent the size of the array.

```
1  bool has_duplicates(int32_t arr[], size_t n) {
2    for (size_t i = 0; i < n; ++i) {
3      for (size_t j = i + 1; j < n ; ++j) {
4        if (arr[i] == arr[j]) {
5          return true;
6        } // if
7      } // for j
8    } // for i
9
10   return false;
11 } // has_duplicates()
```

What is the best and worst-case time complexity of this function, in terms of n? Note: you may find the following identity useful:

$$\sum_{x=1}^{n} x = 1+2+3+4+\ldots+n = \frac{n(n+1)}{2}$$

A) Best-Case: $\Theta(1)$, Worst-Case: $\Theta(n)$
B) Best-Case: $\Theta(n)$, Worst-Case: $\Theta(n)$
C) Best-Case: $\Theta(1)$, Worst-Case: $\Theta(n^2)$
D) Best-Case: $\Theta(n)$, Worst-Case: $\Theta(n^2)$
E) Best-Case: $\Theta(n^2)$, Worst-Case: $\Theta(n^2)$

34. Consider the following function implementation:

```
1  int32_t foo(int32_t n) {
2    int32_t count = 0;
3    for (int32_t i = 1; i < n; ++i) {
4      for (int32_t j = 1; j < i; ++j) {
5        ++count;
6      } // for j
7    } // for i
8
9    return count;
10 } // foo()
```

What is the time complexity of the this function in terms of n? Note: you may find the following identity useful:

$$\sum_{x=1}^{n} x = 1+2+3+4+\ldots+n = \frac{n(n+1)}{2}$$

A) $\Theta(n)$
B) $\Theta(n\log(n))$
C) $\Theta(n^2)$
D) $\Theta(n^3)$
E) None of the above

35. Consider the following function implementation:

```
1  int32_t foo(int32_t n) {
2    int32_t count = 0;
3    for (int32_t i = 1; i < n; ++i) {
4      for (int32_t j = 1; j < i; j *= 2) {
5        ++count;
6      } // for j
7    } // for i
8
9    return count;
10 } // foo()
```

What is the time complexity of the this function in terms of n?

A) $\Theta(\log(n))$
B) $\Theta(n)$
C) $\Theta(n\log(n))$
D) $\Theta(n^2)$
E) None of the above

36. Consider the following function implementation:

```
int32_t foo(int32_t n) {
  int32_t count = 0;
  for (int32_t i = 1; i < n; i *= 2) {
    for (int32_t j = 1; j <= i; ++j) {
      ++count;
    } // for j
  } // for i

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

What is the time complexity of the this function in terms of n? Note: you may find the equation for the sum of a geometric series helpful, where a is the first term and r is the common ratio:

$$S_n = \frac{a(1-r^n)}{1-r}, r \neq 1$$

A) $\Theta(\log(n))$
B) $\Theta(n)$
C) $\Theta(n\log(n))$
D) $\Theta(n^2)$
E) None of the above

37. Consider the following function implementation:

```
int32_t foo(int32_t n) {
  int32_t count = 0;
  for (int32_t i = 0; i < n; ++i) {
    for (int32_t j = n; j > i; j /= 2) {
      for (int32_t k = j / 2; k < j; ++k) {
        for (int32_t m = 1; m < n; m *= 2) {
          ++count;
        } // for m
      } // for k
    } // for j
  } // for i

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

What is the time complexity of this function in terms of n? Note: you may find the following identity useful:

$$\sum_{x=1}^{n} \frac{1}{x} = \left(1+\frac{1}{2}+\frac{1}{3}+\frac{1}{4}+\ldots+\frac{1}{n}\right) = \Theta(\log(n))$$

A) $\Theta(n\log(n))$
B) $\Theta(n^2\log(n))$
C) $\Theta(n\log^2(n))$
D) $\Theta(n^2\log^2(n))$
E) None of the above

38. Consider the following function implementation:

```
int32_t sum(int32_t arr[], size_t n) {
  int32_t sum = 0;
  for (int32_t i = 0; i < n; ++i) {
    sum += arr[i];
  } // for i

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

What is the *space complexity* and *auxiliary space* used by this function, in terms of the size of `arr`, n?

A) Space Complexity: $\Theta(1)$, Auxiliary Space: $\Theta(1)$
B) Space Complexity: $\Theta(n)$, Auxiliary Space: $\Theta(1)$
C) Space Complexity: $\Theta(1)$, Auxiliary Space: $\Theta(n)$
D) Space Complexity: $\Theta(n)$, Auxiliary Space: $\Theta(n)$
E) None of the above

Chapter 4 Exercise Solutions

1. **The correct answer is (E).** All of the statements are true. For statement I, pure runtime is an imperfect metric for comparing two algorithms due to the presence of confounding factors like background processes and CPU usage. For statement II, a program's runtime with respect to input size is indeed independent of most external factors, congruent with the claim in statement I. For statement III, when we express runtime in terms of input size, we can more easily categorize and compare them using big-O notation (e.g., its easier to see that an $\Theta(n)$ algorithm has a lower order of growth than an $\Theta(n^2)$ algorithm).

2. **The correct answer is (A).** None of the statements are true. The asymptotic efficiency of algorithms deals with how runtime increases with the size of the input as the size of the input increases bound. For instance, as the size of the input grows, we can reason that a $\Theta(n^2)$ algorithm will see its runtime increase much more drastically than a $\Theta(n\log(n))$ algorithm. However, it does not guarantee that an algorithm with a $\Theta(n^2)$ time complexity will always run faster than an algorithm with a $\Theta(n\log(n))$ time complexity, since there are other factors at play (e.g., the algorithm with the larger complexity class may have been run in a best-case scenario or may have better constants and lower-order terms that make it faster for a given input size).

3. **The correct answer is (E).** Big-O represents an asymptotic upper bound, where $a(n)$ is $O(b(n))$ if $b(n)$ serves as an upper bound for $a(n)$. The set of functions that comprise $O(f)$ for a given function f are the set of functions that are bounded above by f. This matches option (E) — for a function to be bounded above by f, it cannot grow quicker than f.

4. **The correct answer is (D).** Lower order terms can be ignored in big-O, and the term with the highest order is $79n$. Coefficients can be ignored as well, so the complexity of *foo* can be simplified to $\Theta(n)$.

5. **The correct answer is (D).** A $\Theta(n^3\log(n))$ algorithm grows faster than a $\Theta(n^3)$ algorithm due to the presence of an additional $\log(n)$ term that is multiplied with n^3. A $\Theta(n!)$ algorithm grows faster than a $\Theta(2^n)$ algorithm, and a $\Theta(n^n)$ algorithm grows faster than both a $\Theta(2^n)$ and a $\Theta(n!)$ algorithm. You can also see this by testing a few values of n, as shown below:

n	2	4	8
2^n	4	16	256
$n!$	2	24	40,320
n^n	4	256	16,777,216

6. **The correct answer is (E).** Algorithms involving $\log(n)$ complexities often involve splitting work over and over again and doing a constant number of operations for each split. This is true for both (C) and (D).

7. **The correct answer is (D).** If the input size is too small, the relationship may not be revealed. Plotting out runtimes is most revealing when the input size is large.

8. **The correct answer is (C).** The program ran better than expected, so it could not have exposed worst-case behavior if all else went to plan.

9. **The correct answer is (A).** None of the statements are true. The best way to disprove these statements is to use trigonometric functions, which do not adhere to many of the rules that are associated with traditional functions. Statement I is false when $f = \sin(x)$ and $g = \cos(x)$. Statement II is false when $f = \sin(x)$, $g = \cos(x)$, and $h = \sin(x) + \cos(x)$. Statement III is false when $f = \sin(x)$, $g = \cos(x)$, and $h = \sin(x)\cos(x)$. Statement IV is false when $f = x^2$ and $g = 1/x$.

10. **The correct answer is (A).** Using log rules, $\log(n^3 9^n)$ can be rewritten as $\log(n^3) + \log(9^n)$. We can move the exponents in front of the log to get $3\log(n) + n\log(9)$. Both 3 and $\log(9)$ are constants, so they can be ignored, leaving us with $\log(n) + n$. The lower order term of $\log(n)$ can also be dropped, so the final complexity is $\Theta(n)$, not $\Theta(9^n)$. (A) is false. The other options are true. For (B), $\log(n^5 4^{3n})$ can be rewritten as $\log(n^5) + \log(4^{3n}) = 5\log(n) + 3n\log(4) = \Theta(n)$. For (C), $\log(n!)$ is $\Theta(n\log(n))$ (see example 4.15). For (D), the base of a log does not matter when working with big-O notation, so $\log_3(n)$ is $\Theta(\log_2(n))$. For (E), $\log(n^2)$ is $2\log(n)$ using log rules, which is $\Theta(\log(n))$.

11. **The correct answer is (E).** Asymptotic runtime deals with how an algorithm's runtime scales with respect to input size; the rate of growth of an algorithm is not impacted by the input size, machine performance, or compiler flags.

12. **The correct answer is (A).** The overall time complexity of `foo()` is $\Theta(n^2 + n + n\log(n)) = \Theta(n^2)$, where the runtime of `helper_a()` contributes to the highest order term. Therefore, only improving `helper_b()` or `helper_c()` does not change the overall time complexity of `foo()`. This eliminates options (C) and (D). For option (B), the coefficient does not matter, so the time complexity of `foo()` would still be $\Theta(n^2)$. Only option (A) improves the overall time complexity of the `foo()` function from $\Theta(n^2)$ to $\Theta(n\log(n))$.

13. **The correct answer is (E).** All of statements can be true in the equal case, where $A(n)$, $B(n)$ and $C(n)$ have the same order of growth. Note that big-O, Θ, and Ω can also apply to a tight bound.

14. **The correct answer is (E).** This is not a good experiment for determining which algorithm is the most efficient due to the presence of confounding factors, especially since each algorithm is only timed once. Plus, the conditions of the experiment are not well defined enough to come to a reasonable conclusion (e.g., what input size are you running on, compilation flags used, etc.).

15. **The correct answer is (A).** This graph represents a logarithmic relationship between input size and runtime. None of options (B) through (E) are correct since they all grow at least as fast as linear, which is not true for the relationship in the provided graph.

16. **The correct answer is (B).** If $A(n)$ is bounded above by $B(n)$ and bounded below by $C(n)$, and all three functions are strictly increasing, then $B(n)$ must also be bounded below by $C(n)$ (otherwise, it could not be an upper bound to $A(n)$, which is also bounded below by $C(n)$). This matches option (B). Note that the other three options are all possible, but only (B) is true in all cases.

17. **The correct answer is (D).** Logarithmic functions grow slower than polynomial functions. To make this problem easier to understand, one strategy is to replace $\log(n)$ with another variable, let's say x. We know that $\Theta(\log(x)) < \Theta(x^{1/2}) < \Theta(x) < \Theta(x^2)$. Thus, it must also be true that $\Theta(\log(\log(n))) < \Theta(\log^{1/2}(n)) < \Theta(\log(n)) < \Theta(\log^2(n))$, which matches option (D).

18. **The correct answer is (C).** Options (A) and (B) are incorrect because $\Theta(n^2 \log(n))$ is part of a different complexity class as $\Theta(n^2)$. Option (D) is incorrect because $n^2 \log(n)$ grows faster than n^2, so n^2 cannot be an upper bound. Only option (C) is correct, since $n^2 + \log(n)$ is $\Theta(n^2)$, which does serve as a lower bound for $n^2 \log(n)$.

19. **The correct answer is (B).** Only II is true. Statement I is false because $f(n) = (2^n)^2 = g(n)^2$, so $f(n)$ and $g(n)$ are not part of the same complexity class. Statement III is false because $e^{n/2}$ and e^n are part of different complexity classes (one is the square root of the other, so they cannot grow at the same rate as input size grows). Statement II is true because $2^{n \log_2(n)} = (2^{\log_2(n)})^n = n^n$.

20. **The correct answer is (A).** Statement (A) is true because $\log(n) + \log(m) = \log(nm)$, which is an upper bound on $\log(n+m)$. Statement (B) is false because $(n-1)!$ is a factor of n smaller than $n!$, so it cannot be an upper bound on $n!$. Statement (C) is false because $2^{n+m} = 2^n 2^m$, which is not bounded above by $2^n + 2^m$. Statement (D) is false sine cannot be strictly bounded by cosine, and vice versa.

21. **The correct answer is (C).** If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$, that means that $f(n)$ must have a strictly larger rate of growth than $g(n)$. This means that $g(n)$ must be a lower bound on $f(n)$, which matches option (C). Note that you can also use the notation $\omega(g(n))$ to indicate that $f(n)$ is strictly larger than $g(n)$, but you do not need to worry about that here.

22. **The correct answer is (A).** The time to solve a problem with input size S can be expressed as C^S. The time to solve a problem with input size $S+6$ is C^{S+6}. From this, we know that $T = C^S$ and $9T = C^{S+6}$. Combining these equations, we can conclude that:

$$\frac{9T}{T} = \frac{C^{S+6}}{C^S}$$
$$9 = C^{S+6-S} = C^6$$
$$C = \sqrt[6]{9} = \sqrt[6]{3^2} = \sqrt[3]{3}$$

23. **The correct answer is (C).** The time to solve a problem with input size S can be expressed as $\log_2(S)$. Therefore, the input that can be solved in time $T+1$ is $1 + \log_2(S)$ (since $T = \log_2(S)$). Using log rules, we can express the following:

$$T + 1 = 1 + \log_2(S)$$
$$T + 1 = \log_2(2) + \log_2(S)$$
$$T + 1 = \log_2(2S)$$

$\log_2(2S)$ is the time needed to solve a problem of input size $2S$. Therefore, we can solve a problem of input size $2S$ in time $T+1$.

24. **The correct answer is (B).** The time T it takes to solve a problem of input size S is S^2. Thus, in terms of S, $8T$ can be equivalently expressed as $8S^2$. What input size can be solved in a time of $8S^2$? Let S' be the size of problems that can be solved in $8S^2$ time. Solving for S', we get a solution of $(2\sqrt{2})S$.

$$(S')^2 = 8S^2$$
$$S' = \sqrt{8S^2}$$
$$S' = (2\sqrt{2})S$$

25. **The correct answer is (C).** The loop from lines 2-4 takes $\Theta(m)$ time, and the loop from lines 6-8 takes $\Theta(n)$ time. These loops are sequential (one happens directly after the other), so their time complexities should be added to get the overall complexity of the `foo()` function — this comes out to $\Theta(m+n)$.

26. **The correct answer is (D).** The loop from lines 3-5 takes $\Theta(n)$ time, and this loop is run m times in the outer loop on line 2. Since this is a nested loop, we multiply $\Theta(n)$ by m to get the overall time complexity of $\Theta(mn)$.

27. **The correct answer is (C).** The loop from lines 3-5 takes $\Theta(m)$ time, and the loop from lines 7-9 takes $\Theta(n)$ time. Since these two loops are sequential, the combined time complexity of lines 3-9 is $\Theta(m+n)$. These two loops are then run p times in an outer loop on line 2, so we multiply $\Theta(m+n)$ by p to get a final time complexity of $\Theta(mp+np)$ for the function `foo()`.

28. **The correct answer is (D).** The inner loop starting on line 4 runs in $\Theta(n)$ time, and it is run n times in the outer loop defined on line 3. All other work in the function takes constant time. Thus, the total time complexity is $n \times \Theta(n) = \Theta(n^2)$.

29. **The correct answer is (C).** The inner loop on line 6 iterates from 0 to n, incrementing by $\sqrt{n}$ each time. This means that $n/\sqrt{n} = \sqrt{n}$ iterations must be run in this innermost loop, each of which performs constant work. The middle loop on line 5 increments from 1 to n, doubling with each iteration — this runs $\log(n)$ iterations. Thus, the loops on line 5 and 6 have a combined time complexity of $\Theta(\sqrt{n}\log(n))$. Lastly, the outermost loop on line 4 runs from $n/2$ to n, which involves $n/2$ iterations. Each of these $n/2$ iterations performs a $\Theta(\sqrt{n}\log(n))$ nested loop, so the overall time complexity of the triple-nested loop is $(n/2) \times \Theta(\sqrt{n}\log(n)) = \Theta(n\sqrt{n}\log(n)))$ after dropping all coefficients. Since all other work takes constant time, which is a lower order term that can be dropped in our big-O notation, the overall time complexity of `foo()` is therefore $\Theta(n\sqrt{n}\log(n)))$.

30. **The correct answer is (A).** The dominant work within this function can be attributed to the `while` loop on line 3, which divides the input size n by half with every iteration, terminating when n reaches a value less than one. Thus, the time complexity of the loop is $\Theta(\log(n))$.

31. **The correct answer is (B).** There is a loop dependency here, where the number of times the `for` loop runs on line 4 is determined by the value of n, which is being halved within the outer loop defined on line 3. During the first iteration, the loop on line 4 runs n times; during the second iteration, the loop on line 4 runs $n/2$ times, and so on. Since each iteration of the inner loop takes a constant amount of work, the overall time complexity of the nested loop is $n + n/2 + n/4 + \ldots = \Theta(n)$ using the formula for a geometric series.

32. **The correct answer is (B).** See example 4.12.

33. **The correct answer is (C).** In the best case, the first two numbers in the array are duplicates, which would cause the function to return in constant time (since only two values would be checked). In the worst-case, there are no duplicates, which would require the entire nested loop to run. Using the provided identity, the time complexity in the worst case would be $\Theta(n^2)$.

34. **The correct answer is (C).** There is a loop dependency here, so we will consider both nested for loops in tandem. During the first iteration of the outer loop, the inner loop runs 0 times; during the second iteration of the outer loop, the inner loop runs 1 time; during the third iteration of the outer loop, the inner loop runs 2 times, and so on. Using the provided identity, the time complexity of the function comes out to $0+1+2+\ldots+(n-1) = \Theta(n^2)$.

35. **The correct answer is (C).** There is a loop dependency here, so we will consider both nested for loops in tandem. The analysis for this problem is similar to the previous problem, with the exception that the amount of work can be approximated as $0+\log(1)+\log(2)+\log(3)+\ldots+\log(n-1)$ instead of $0+1+2+3+\ldots+(n-1)$ because the value of `j` is doubled instead of incremented within the inner loop. Using log rules, we can conclude that the total work done by the nested loop is $\log(1\times2\times3\times\ldots\times(n-1)) = \log((n-1)!)$. Furthermore, we know that $\log(n!) = \Theta(n\log(n))$, which allows us to conclude that $\log((n-1)!) = \Theta(\log(n!/n)) = \Theta(\log(n!)-\log(n)) = \Theta(n\log(n)-\log(n)) = \Theta(n\log(n))$ after dropping lower order terms.

36. **The correct answer is (B).** There is a loop dependency here, so we will consider both nested for loops in tandem. The first iteration of the outer loop runs 1 time, the second iteration of the outer loop runs 2 times, the third iteration of the outer loop runs 4 times, the fourth iteration of the outer loop runs 8 times, and so on. This follows the form of a geometric series, where the iteration count is doubled with each iteration of the outer loop, and the outer loop runs $\log_2(n)$ times. Using the provided summation, we can conclude that:

$$1+2+4+8+\ldots+n = \sum_{k=0}^{\log_2(n)} 2^k = \frac{1\left(1-2^{\log_2(n)+1}\right)}{1-2} = \frac{1-2\left(2^{\log_2(n)}\right)}{1-2} = \frac{1-2n}{1-2} = \frac{2n-1}{1} = \Theta(n)$$

37. **The correct answer is (B).** This function involves a triply-nested loop dependency (where `j` depends on `i`, and `k` depends on `j`), so we will consider the three loops on lines 3-5 together. Notice that the innermost loop on line 6 always iterates from 1 to `n`, doubling with each iteration, so the time complexity of this innermost loop is always $\Theta(\log(n))$.
    - On the first iteration of the outermost for loop (`i = 0`):
        - The first iteration of the second nested loop (`j = n`) runs the third loop from `k = n/2` to `k = n`, which performs $\Theta(\log(n))$ work on each iteration. Since the `k` loop is run $n/2$ times, the total work done on this first iteration is $(n/2)\log(n)$.
        - The second iteration of the `j` loop runs the `k` loop from `n/4` to `n/2`, where $\Theta(\log(n))$ work is performed each iteration. Since the `k` loop is run $n/4$ times here, the total work done on this second iteration is $(n/4)\log(n)$.
        - The `j` loop halves `j` from `n` to `i= 0`, so the `j` loop runs $\log(n)$ times during this first iteration of the `i` loop. Continuing the previous pattern, the total work done on the first iteration of the `i` loop is $\frac{n}{2}\log(n)+\frac{n}{4}\log(n)+\frac{n}{8}\log(n)+\ldots+\frac{n}{2^{\log_2(n)}}\log(n)$.
    - On the second iteration of the outermost for loop (`i = 1`):
        - The first iteration of the `j` loop (`j = n`) runs the `k` loop $n/2$ times from `k = n` to `k = n/2`, which performs $\Theta(\log(n))$ work with each iteration. The total work done on this iteration is $(n/2)\log(n)$.
        - The second iteration of the `j` loop (`j = n/2`) runs the `k` loop $n/4$ times from `k = n/4` to `k = n/2`, which performs $\Theta(\log(n))$ work with each iteration. The total work done on this iteration is $(n/4)\log(n)$.
        - The only difference between the second and first iteration of the outer `i` loop is the number of times the `j` loop runs. During this iteration, `j` is halved from `n` to `1` instead of `0`, so the loop runs $\log_2(n-1)$ times instead of $\log_2(n)$ times, which means the total work done on the second iteration of the `i` loop is $\frac{n}{2}\log(n)+\frac{n}{4}\log(n)+\frac{n}{8}\log(n)+\ldots+\frac{n}{2^{\log_2(n-1)}}\log(n)$.
    - On the third iteration of the outermost for loop (`i = 2`), the total work done is $\frac{n}{2}\log(n)+\frac{n}{4}\log(n)+\frac{n}{8}\log(n)+\ldots+\frac{n}{2^{\log_2(n-2)}}\log(n)$.
    - This pattern continues, until the final iteration of the outermost for loop.

    Putting this all together, the total work done by the nested loop can be expressed as:

$$\begin{aligned}
T(n) &= \left(\frac{n}{2}\log(n)+\frac{n}{4}\log(n)+\ldots+\frac{n}{2^{\log_2(n)}}\log(n)\right)+\left(\frac{n}{2}\log(n)+\frac{n}{4}\log(n)+\ldots+\frac{n}{2^{\log_2(n-1)}}\log(n)\right)+\left(\ldots+\frac{n}{2^{\log_2(1)}}\log(n)\right)\\
&= \frac{1}{2}n\log(n)\left[\left(1+\frac{1}{2}+\frac{1}{4}+\ldots+\frac{1}{2^{\log_2(n)}}\right)+\left(1+\frac{1}{2}+\frac{1}{4}+\ldots+\frac{1}{2^{\log_2(n-1)}}\right)+\ldots+\left(\ldots+\frac{1}{2^{\log_2(1)}}\right)\right]\\
&= \frac{1}{2}n\log(n)\left[\sum_{y=0}^{\log_2(n)}\left(\frac{1}{2}\right)^y+\sum_{y=0}^{\log_2(n-1)}\left(\frac{1}{2}\right)^y+\ldots+\sum_{y=0}^{\log_2(1)}\left(\frac{1}{2}\right)^y\right] = \frac{1}{2}n\log(n)\sum_{x=1}^{n}\left(\sum_{y=0}^{\log_2(x)}\left(\frac{1}{2}\right)^y\right) = \frac{1}{2}n\log(n)\sum_{x=1}^{n}\left(\frac{1-\left(\frac{1}{2}\right)^{\log_2(x)}}{1-\frac{1}{2}}\right)\\
&= \frac{1}{2}n\log(n)\sum_{x=1}^{n}\left(2\left(1-\left(\frac{1}{2}\right)^{\log_2(x)}\right)\right) = n\log(n)\sum_{x=1}^{n}\left(1-\left(\frac{1}{2}\right)^{\log_2(x)}\right) = n\log(n)\left(\sum_{x=1}^{n}1-\sum_{x=1}^{n}\frac{1}{2^{\log_2(x)}}\right) = n\log(n)\left(n-\sum_{x=1}^{n}\frac{1}{x}\right)\\
&= n\log(n)\times(n-\Theta(\log(n))) = n\log(n)\times\Theta(n) = \Theta(n^2\log(n))
\end{aligned}$$

38. **The correct answer is (B).** The space complexity is the the amount of memory used by an algorithm to solve a problem with respect to input size. On the other hand, auxiliary space only considers the additional memory used by the algorithm but does not include the memory used by the input values themselves. In this example, the `sum()` function acts on an array of size n that is stored in memory, so the total space complexity is $\Theta(n)$. However, this array is an input, so although the function uses its memory, it does not actually allocate it. Beyond the memory passed in as an input, the `sum()` only initializes a constant amount of additional space, so its auxiliary space usage is $\Theta(1)$.