Chapter 4
Complexity Analysis

4.1 Asymptotic Runtime

In this class, we will focus a lot on the *efficiency* of algorithms. However, what exactly is efficiency, and how do we measure it? Upon first glance, it might seem appropriate to measure the efficiency of an algorithm by measuring its runtime. Suppose that two students are given two algorithms, A and B, that do the exact same thing, and they want to determine which algorithm is more time-efficient. Student 1 prepares algorithm A on their machine, and student 2 prepares algorithm B on their machine. Both students begin running their algorithms at the same time, and whichever algorithm finishes first is the one that is deemed more efficient.

However, this approach does not work because there are *too many confounding factors* that could affect the runtime of the algorithms. What if student 2 had a faster computer than student 1? Even if algorithm A was truly better, the fact that student 2 had a faster machine may lead us to incorrectly conclude that algorithm B is superior.

What if we remove this confounding variable of machine performance by limiting the experiment to a single machine? Suppose student 1 runs both algorithm A and algorithm B on their machine, one after another, and uses the runtime results to determine which algorithm is better. While this approach removes the discrepancy across machines, it is still not perfect. What if more intensive processes were running in the background when the student was running function A? There is still a chance that the inferior algorithm can come up on top.

Thus, while it may seem intuitive to base the time-efficiency of a program off its runtime, *pure runtime is actually an imperfect metric!* This is because the runtime of an algorithm can be confounded by factors other than the efficiency of the algorithm itself. Variables such as CPU speed, the compiler and compiler flags used, and other programs running in parallel can influence the runtime of a program, even if these factors are not relevant to the algorithm itself. As a result, simply using program runtime to measure efficiency does not fully work! We want to measure efficiency in a way that removes these confounds and better reflects the efficiency of a program on its own.

To do this, we will be using **input size** instead of actual runtime to determine the relative efficiency of a program. That is, instead of asking ourselves *"How long does this program take to run?"* when measuring efficiency, we ask ourselves *"How does the program's runtime scale with input size?"*. If we double the size of the input, does the runtime also double? Does it quadruple? Or does it stay the same? Given two algorithms that perform the same task, if the runtime of one algorithm grows much slower than the other as input size increases, then that algorithm is considered to be better than the other when it comes to time efficiency.

Why is this approach better? As mentioned before, the actual runtime of a program may be confounded by factors such as CPU speed, compiler flags, and processes running the background. ***However, for input sizes that are large enough, the rate of growth of runtime with respect to input size is independent of most external factors!*** Regardless of how fast the CPU is, which compiler flags are used, or what programs are running in the background, the rate of growth of an algorithm should stay relatively consistent as you increase its input size. This idea forms the basis of **asymptotic runtime**, which describes how the runtime of an algorithm grows as the size of its input grows. We use asymptotic runtime to measure and compare the time-efficiency of algorithms.

Asymptotic runtime and actual runtime are not the same thing! For example, if program A completes in 2.81 seconds, the number 2.81 represents the *runtime* of the program in seconds. However, if program A's runtime doubles when input size doubles, triples when input size triples, and quadruples when input size quadruples, etc., we say there is a linear relationship between the input size and the runtime of the program. This linear relationship describes the *asymptotic runtime* of the program. Unlike runtime, which is expressed as a unit of time (e.g., seconds), the asymptotic runtime of an algorithm describes the relationship between input size and runtime, and it is expressed using something known as **big-O notation**. This will be covered in the next section.

4.2 Complexity Classes and Big-O Notation

In the previous section, we introduced the concept of asymptotic runtime. In this section, we will go over how the asymptotic runtime of an algorithm can be computed. To do so, we will first need to conceptualize how we want to measure our input size.

What counts as "one unit" of input? Ultimately, it depends on what the algorithm is designed to do: you want to select a unit of measurement that is related to the functionality of your algorithm. For example, if your algorithm sorts an array of integers, it would make sense to treat the input size as the number of integers in the array.

On a similar vein, if you had a graph with V vertices and E edges, the way you define the input size should be related to what your algorithm does. If your algorithm only deals with the number of vertices in the graph (and ignores the edges), it would be sufficient to define input size in terms of the number of vertices V. However, if your algorithm does work on *both* the vertices and edges of the graph, it may be better to define input size in terms of both V and E. The more information you include when defining your input size, the more information you can potentially get out from time complexity analysis (for example, measuring asymptotic runtime using both V and E can give you an idea of which variable has a greater impact on the performance of an algorithm).

After deciding what counts as a unit of input, you can vary the amount of input you feed into the algorithm, measure the amount of time it takes to run the algorithm, and plot out the points on a graph to visualize a relationship between input size and runtime. *To ensure that this relationship can be clearly determined by the results, make sure that the input sizes you experiment with are large enough!* If the input size is too small, your results may be obfuscated by variance, and you may not be able to clearly see how runtime scales with the input size.

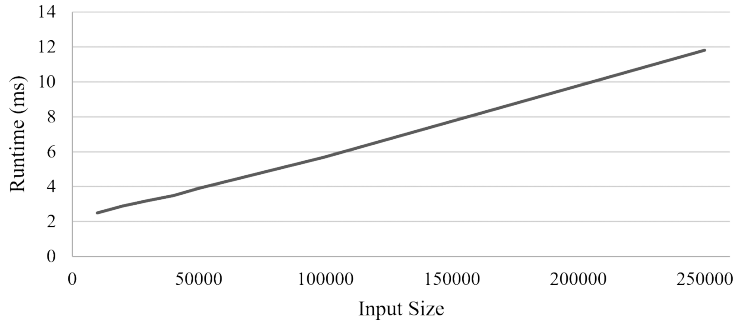
Example 4.1 Consider the following algorithm, which sums up all the numbers in an array of integers. Determine the asymptotic runtime complexity of this algorithm by measuring runtime with respect to input size.

```
1  int32_t sum_array(int32_t arr[], size_t size) {
2    int32_t total = 0;
3    for (size_t i = 0; i < size; ++i) {
4      total += arr[i];
5    } // for i
6    return total;
7  } // sum_array()
```

Our first step is to determine how we want to measure our input size. Since we are summing an array, it would make sense for input size to be the size of the input array. Now that we know our unit of input, let's measure the runtime of the program with different input sizes. The following data was collected by running the `sum_array()` function for array sizes of 10000, 20000, 30000, 40000, 50000, 100000, and 250000.

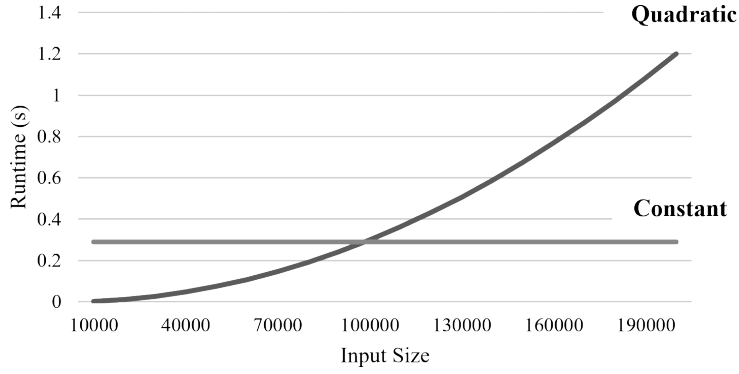
Input Size	Runtime (ms)
10000	2.5
20000	2.9
30000	3.2
40000	3.5
50000	3.9
100000	5.7
250000	11.8

Graphing this out, we can see that there is a *linear* relationship between the input size and the runtime. That is, as the size of the input size increases, the runtime increases linearly (in a straight line).



The **time complexity** of an algorithm is a way to express the relationship between runtime and input size. If there is a linear relationship between runtime and input size, then the algorithm has a *linear time complexity*.

Linear time is just one of many time complexity classes that an algorithm can exhibit. If there exists a quadratic relationship between input size and runtime, the algorithm has a *quadratic time complexity*. If the runtime stays relatively constant regardless of the input size, then the algorithm has a *constant time complexity*. Examples of these two time complexity classes are shown below:



Big-O notation can be used to express these time complexities mathematically. Instead of describing time complexity classes using words like constant, linear, and quadratic, we describe them using a notation that follows the form $O(f(\ldots))$, where $f(\ldots)$ is a function that depends on an algorithm's input size variables.

Let's revisit the graph of the `sum_array()` function, shown previously in example 4.1. How can we express its asymptotic runtime using big-O notation? To do so, we would need to determine a function $f(n)$, where n is the size of the array we want to sum. We can determine $f(n)$ by following these steps:

1. Express the runtime as a function of the input size variables.
2. Identify the fastest growing terms in the equation calculated in step 1 and drop all lower order terms.
3. Drop the coefficients of the fastest growing terms.

After doing this, you will be left with the term that is used for the big-O notation. Let's try an example:

Example 4.2 Express the time complexity class of the `sum_array()` function using big-O notation.

To do this, we can follow the three steps above.

1. **Express the runtime as a function of the input size variables.** We can use the points shown on the previous page to determine the line of best fit. By doing so, we can express the runtime $T(n)$ as the function of the input size n using the equation:

$$T(n) = (4 \times 10^{-5})n + 2$$

2. **Identify the fastest growing terms in the equation calculated in step 1 (and drop all lower order terms).** The fastest growing term is the term that grows the fastest as the input size (in this case, n) increases without bound. In the equation above, the constant term 2 does not grow at all as n increases, while the term $(4 \times 10^{-5})n$ grows linearly as n increases. The fastest growing term is thus $4 \times 10^{-5}n$.

   In many cases, the fastest growing term is the term with the largest exponent. For example, if the relationship between input size and runtime were $T(n) = 5n^3 + 2n^7 + 16n^6 + 9n^2 + 55n^4 + 23$, the fastest growing term would be $2n^7$, since this is the term with the largest exponent, 7. See the following page for orders of growth.

3. **Drop the coefficients of the fastest growing terms.** Since the fastest growing term is $4 \times 10^{-5}n$, we can remove the coefficient (4×10^{-5}). We are left with the term n, which is our value of $f(n)$.

The time complexity of `sum_array()` is thus $O(n)$. Let us consider another example.

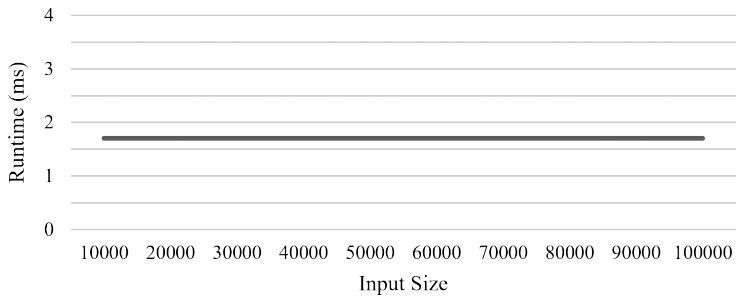
Example 4.3 You are given the following function. Express the time complexity of the `return_zero()` function using big-O notation:

```
1  int32_t return_zero(int32_t arr[], size_t size) {
2    int32_t zero = 0;
3    return zero;
4  } // return_zero()
```

After running this function with arrays of many different sizes and measuring its runtime, we get the following:

Input Size	Runtime (ms)
10000	1.7
25000	1.7
50000	1.7
75000	1.7
100000	1.7

The runtime of the function is plotted below:



We can use this information to follow the three steps on the previous page.

1. **Express the runtime as a function of the input size variables.** We can express the runtime $T(n)$ as a function of the input size variable n using the equation:
$$T(n) = 1.7$$
2. **Identify the fastest growing terms in the equation calculated in step 1 (and drop all lower order terms).** There is only one term here (1.7), so it is by default the fastest growing term.
3. **Drop the coefficients of the fastest growing terms.** Although it may not be obvious, the term 1.7 can actually be written as $1.7n^0$. Thus, the coefficient of the fastest growing term is 1.7 — removing this, we end up with just n^0, or 1.

The time complexity of `return_zero()` is thus $O(1)$.

Why are we allowed to drop coefficients and lower order terms when dealing with time complexities? Recall that the time complexity of an algorithm only deals with how its runtime *scales* with respect to input size. Thus, it does not matter what the coefficients and lower order terms are, as they are not directly responsible for an algorithm's rate of growth. For example, the functions $T(n) = 0.5n^2 + 13n + 5$ and $T(n) = 14n^2 + 3n + 9$ both experience the same quadratic growth with respect to input size, even though they have different coefficients and lower order terms; it would therefore be more meaningful to express both functions with a complexity of $O(n^2)$.

Remark: Just because coefficients are dropped does *not* mean that they are not important! An algorithm whose runtime can be expressed using the function $T(n) = 999n^2$ is slower than an algorithm whose runtime can be expressed using $T(n) = 0.01n^3$ for most reasonable input sizes n, even if it technically has a better time complexity.

This is where knowing the difference between runtime and asymptotic runtime is crucial. When dealing with time complexities, we only work with the rate of growth and not the actual runtime itself. Because of this, *a better time complexity does not guarantee a faster runtime.* As shown with the example above, it is entirely possible for an algorithm with a better asymptotic time complexity to run slower — this is because time complexity only indicates how fast runtime *scales with input size*, not the actual runtime itself!

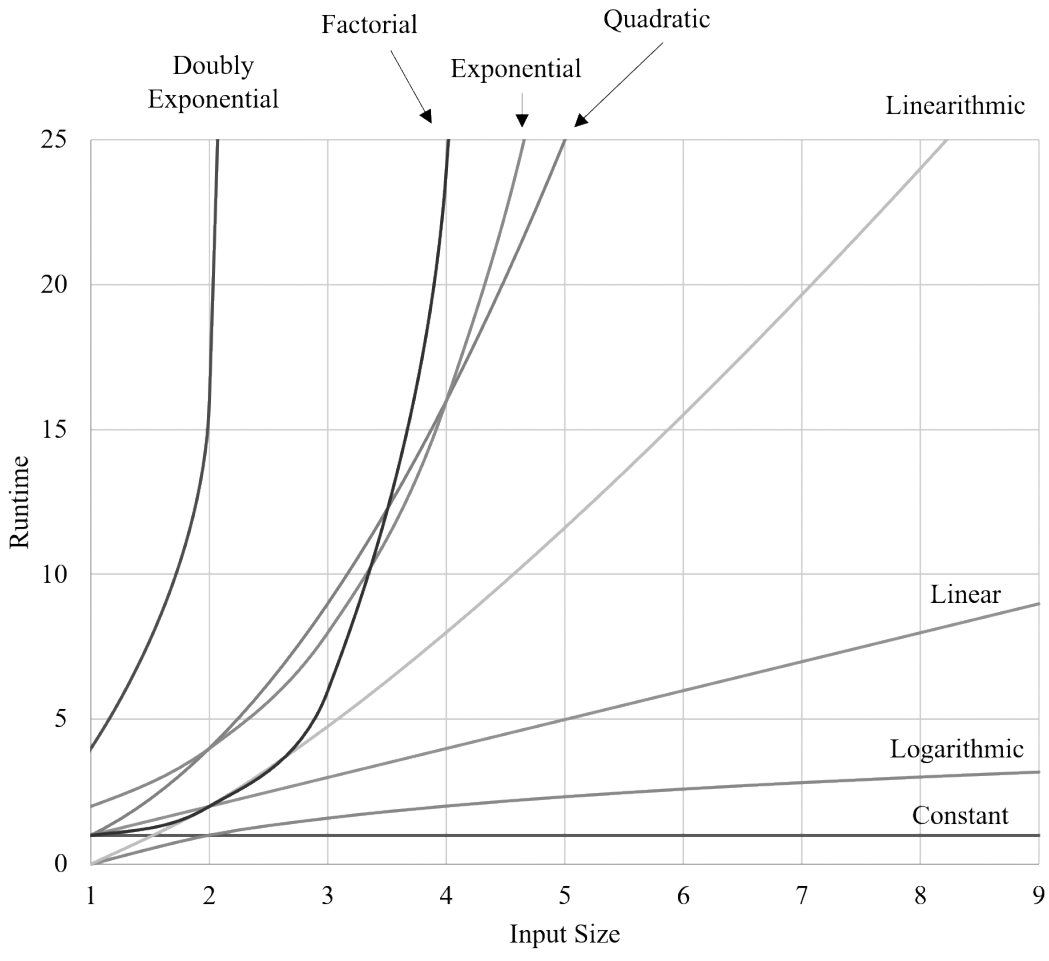
The following table lists a few common complexity classes, in order of rate of growth:

Name of Complexity Class	Big-O Notation
Constant	$O(1)$
Logarithmic	$O(\log(n))$
Linear	$O(n)$
Loglinear/Linearithmic	$O(n\log(n))$
Quadratic	$O(n^2)$
Cubic	$O(n^3)$
Polynomial (includes quadratic and cubic)	$O(n^d)$ where d is a constant
Exponential	$O(c^n)$ where c is a constant
Factorial	$O(n!)$
Doubly Exponential	$O(2^{2^n})$

A visualization of each of these complexity classes is provided on the graph at the top of the next page.

To summarize this section so far, we introduced the concept of big-O notation and how we can derive the time complexity of an algorithm using experimentation. However, if the chapter ended up here, you should not be satisfied! First of all, experimentation is not always feasible (for example, you will not be able to run code on an exam!). And, in the cases where experimentation can be done, it is not always easy. Even with data, how can we derive an equation that best relates runtime to input size? How large will our input size have to be to derive a precise estimation that is not subject to noise and variation? Furthermore, if we have multiple variables at play (such as the number of vertices (V) and edges (E) in a graph), how can we take all of these variables into account and distinguish among them in our measurements?

The good news is that you do not need a computer or any experimentation to determine the time complexity of an algorithm; simply looking at the code itself is often more than enough to calculate its time complexity. This can be done by analyzing the steps that the algorithm takes and measuring how the *number of steps taken* (rather than its runtime) scales with the input size.



4.3 Measuring Complexity by Counting Steps

In many cases, you will not be able to measure the runtime of an algorithm via experimentation. Luckily, experimentation is not the only way to calculate an algorithm's time complexity. An alternative approach is to count the *number of steps* that an algorithm takes and express the result as a function of the input size. This is the approach you will be expected to know on exams.

Why does this work? As long as the runtime of a single step is independent of input size and can be done in constant time, a function that expresses the relationship between the number of steps and the input size should have the *same order of growth* as a function that expresses the relationship between runtime and input size. In other words, the number of steps that an algorithm takes is directly related to its runtime! For example, suppose that there is a linear relationship between the number of steps an algorithm takes and its input size. In this case, increasing the input size causes a linear growth in the step count needed to complete the algorithm. Since each step takes constant time, increasing the step count linearly would also increase the runtime linearly. Similarly, if there is a quadratic relationship between the number of steps that an algorithm takes and its input size, a quadratic blowup in the number of steps would also lead to a quadratic blowup in runtime.

What counts as a step in a program? In this class, we will consider the following operations as a single step. These operations all take a constant amount of time and are *not* dependent on the size of the input. For example, the time it takes to multiply two fixed-size numbers in an array should remain roughly the same regardless of whether the input array has a size of ten or ten million.

1. variable assignment (e.g. $a = 5$)
2. arithmetic operation (e.g. $a + b$)
3. comparison operation (e.g. $a < b$)
4. array indexing (e.g. `arr[i]`)
5. pointer reference (e.g. `*ptr`)
6. function call
7. function return

With this in mind, we can use the number of steps an algorithm takes to determine its time complexity. The following procedure can be used to complete this process (this is very similar to the procedure outlined in section 4.2):

1. Express the *step count* as a function of the input size variables.
2. Identify the fastest growing terms in the equation calculated in step 1 and drop all lower order terms.
3. Drop the coefficients of the fastest growing terms.

Let's consider the example in the previous section, this time using a step-counting approach:

Example 4.4 Express the time complexity of the `sum_array()` function using big-O notation. Let input size n be the size of the array.

```
1  int32_t sum_array(int32_t arr[], size_t size) {
2    int32_t total = 0;
3    for (size_t i = 0; i < size; ++i) {
4      total += arr[i];
5    } // for i
6    return total;
7  } // sum_array()
```

In this example, we will count the number of steps needed to complete this function. The input size n is the value of the variable `size`, so we will use both `size` and n interchangeably. Let's look at the function line-by-line and analyze the number of steps required.

- On line 1, we have the function call definition. For our examples, we will ignore this line and attribute the work required to call the function to the caller of the `sum_array()` function instead of the `sum_array()` function itself. In other words, if `main()` calls `sum_array()`, then line 1 executes — however, the work for line 1 will be counted for `main()`, and not `sum_array()`.
- Line 2 is where the implementation of `sum_array()` begins. We have a variable assignment, which takes one step.
- Lines 3 and 4 contain a `for` loop. Line 3 itself has an initialization assignment (`i = 0`) which is done once, a comparison that is done $n+1$ times (one comparison for every time the body of the loop runs, plus one additional comparison for the final check to determine that the loop should end), and an arithmetic operation (`++i`) that is run n times (once for each time the body of the loop runs).
- Line 4 contains an arithmetic operation, which takes one step. However, since this is run inside a `for` loop that runs n times, the total number of steps taken is n.
- Line 6 returns from the function, which takes one step.

With this information, we can express the number of steps as a function of input size.

1. **Express the step count as a function of the input size variables.** The total number of steps taken is:
   - 1 step on line 2
   - $1 + (n + 1) + n$ steps on line 3
   - n steps on line 4
   - 1 step on line 6

   which totals to $1+1+n+1+n+n+1 = 3n+4$ steps.

   Thus, our equation that expresses the step count $S(n)$ as a function of input size n is:

   $$S(n) = 3n+4$$

2. **Identify the fastest growing terms in the equation calculated in step 1 (and drop all lower order terms).** The fastest growing term in the expression is $3n$.

3. **Drop the coefficients of the fastest growing terms.** Removing the coefficeint from $3n$ leaves us with n.

The time complexity of `sum_array()` is thus $O(n)$.

We got the same answer that we did when we measured the complexity using the more tedious method of experimentation. However, we are still doing a bit more work than necessary. For example, consider the += operator on line 4. In the example, we listed it as one step. However, += is actually two separate steps: an addition followed by an assignment. A similar argument can be made for the `++i` operation on line 3: the ++ operator first increments, then assigns, resulting in two steps rather than one. Thus, the equation $3n+4$ is not fully accurate.

So, why did we provide an example with an "inaccurate" analysis? It turns out that debating whether += counts as one step or two steps does not matter! Since coefficients and lower order terms are removed anyway, small differences like these will not affect the time complexity class. If we had treated += and ++ as two steps instead of one in the previous example, we would have gotten the equation $S(n) = 5n+4$. However, the growth of this function is still linear! The difference between the 3 and the 5 does not change the overall time complexity class fo the algorithm.

This little detail actually makes our job easier. Why? It does not matter if a line of code takes 1, 2, 3, or even 999 steps; as long as the number is a constant, the difference in steps does not affect the answer. Because these details do not matter, we do not need to calculate the exact number of steps when deriving the complexity class of an algorithm. Instead, it would suffice to just determine the *asymptotic complexity* of each line instead — any coefficients or lower order terms would eventually be dropped regardless.

In other words, if a line of code involves a constant number of steps, we assign the line with the complexity $O(1)$, regardless of what the constant is. Similarly, if the number of steps required to execute a line of code has a linear relationship with the input size, we would assign it with the complexity $O(n)$. This leads us to the following procedure for determining the time complexity of an algorithm, given its code:

Calculating the Time Complexity of a Function By Counting Steps

1. Start at the first line of the function, not including the function header. Walk through the code line by line, assigning each line with a value using the following rules (it may be easy to jot it down to the side).
   a. If the line is the definition of a loop (e.g. `for(...)` or `while(...)`, etc.), assign it with the number of times the loop executes (you may ignore coefficients and lower order terms).
   b. If the line is not a loop definition, assign it with the time complexity required to execute the line.
2. After all lines have been assigned a value, identify the loops in the function. For non-nested loops, *add* all time complexities in the body of the loop, and then *multiply* this result with the number of times the loop runs. That is, to determine the overall time complexity of a loop, multiply the time complexity of its body with the number of times the loop runs.
3. For nested loops (loops within a loop), start from the innermost loop and work outwards, calculating the total complexity at each step. This can be done by starting with the innermost loop and running the procedure from step 2 on that loop. Then, use the value you get to repeat step 2 on the next innermost loop, repeating until you reach the outermost loop.
4. After all the loops are taken care of, *add* the remaining complexities to get the final time complexity. Remember to remove coefficients and lower order terms.

Let's try a few examples.

Example 4.5 Express the time complexity of the `sum_array()` function using big-O notation. Let input size n be the size of the array.

```
1  int32_t sum_array(int32_t arr[], size_t size) {
2    int32_t total = 0;
3    for (size_t i = 0; i < size; ++i) {
4      total += arr[i];
5    } // for i
6    return total;
7  } // sum_array()
```

To approach this problem, walk through each line of the function code and assign it with the time required to execute it:

- **Line 2:** this is an assignment, which takes constant time, so line 2 gets assigned $O(1)$.
- **Line 3:** this is a loop definition, so we assign it with the number of times it runs. This loop runs `size` times. Since `size` is the input size n, we can substitute `size` with n, so line 3 gets assigned $O(n)$.
- **Line 4:** this is an addition and an assignment, which takes constant time, so line 4 gets assigned $O(1)$.
- **Line 6:** this is a return statement, which takes constant time, so line 5 gets assigned $O(1)$.

At this point, we should have the following (the box represents the scope of the `for` loop):

```
1  int32_t sum_array(int32_t arr[], size_t size) {      1
2    int32_t total = 0;                                 2  O(1)
3    for (size_t i = 0; i < size; ++i) {                3  O(n):
4      total += arr[i];                                 4    O(1)
5    } // for i                                         5
6    return total;                                      6  O(1)
7  } // sum_array()                                     7
```

Now that we have listed out all the time complexities, we will now combine the loops. Here, we only have one loop defined on line 3. To determine the overall time complexity of running this loop, we can multiply the body of the loop (which takes $O(1)$ time, as indicated on line 4) with the number of times the loop runs: $O(n) \times O(1) = O(n)$.

```
1  int32_t sum_array(int32_t arr[], size_t size) {      1
2    int32_t total = 0;                                 2  O(1)
3    for (size_t i = 0; i < size; ++i) {                3  O(n)   <-- the for loop runs a O(1) operation O(n) times, so the overall time complexity of the loop is O(n) × O(1), or O(n).
4      total += arr[i];                                 4
5    } // for i                                         5
6    return total;                                      6  O(1)
7  } // sum_array()                                     7
```

Now that we have dealt with all the loops, we can sum up the remaining time complexities to get the overall time complexity of the function:

$$T(n) = O(1) + O(n) + O(1)$$

The highest order term is $O(n)$, so the time complexity of the `sum_array()` function is $O(n)$.

Example 4.6 Express the time complexity of the `foo()` function using big-O notation. Let the input size n be the size of the array. The `sum_array()` function is the same as the one in the previous example.

```
1   void foo(int32_t arr[], size_t size) {
2     int32_t num = static_cast<int32_t>(size);
3     while (num > 0) {
4       for (size_t j = 0; j < size; ++j) {
5         for (size_t k = size - 1; k > (size / 2); --k) {
6           std::cout << arr[k] << '\n';
7           sum_array(arr, size);
8         } // for k
9       } // for j
10      --num;
11    } // while
12    for (size_t i = 0; i < size; ++i) {
13      std::cout << arr[i] << '\n';
14    } // for i
15    return;
16  } // foo()
```

Let's walk through each line of this function and assign it with the time required to execute the line:

- **Line 2:** this is an assignment, which takes constant time, so line 2 gets assigned $O(1)$.
- **Line 3:** this is a loop definition, so we assign it with the number of times it runs. Notice how this loop runs as long as `num` is greater than 0, where `num` is initialized to `size` and is decremented after each run of the loop. Thus, the `while` loop runs `size`, or n times.
- **Line 4:** this is a loop definition, so we assign it with the number of times it runs. Because this loop runs from 0 to `size` and increments by 1 per iteration, it runs n times.
- **Line 5:** this is a loop definition, so we assign it with the number of times it runs. This loop runs from $n-1$ to $n/2$, or roughly $n/2$ times. This can be represented using the complexity class $O(n)$.
- **Line 6:** this is a print statement with array indexing — the runtime of this operation is independent of input size, so this line gets assigned constant time, or $O(1)$.
- **Line 7:** the `sum_array()` function runs in linear $O(n)$ time, from the previous example. Thus, this line gets assigned with $O(n)$.
- **Line 10:** this is a constant time operation, so it gets assigned $O(1)$.
- **Line 12:** this is a loop definition, so we assign it with the number of times it runs. This loop runs from 0 to `size` and increments by 1 per iteration, so it runs n times.
- **Line 13:** this is a constant time operation, so it gets assigned $O(1)$.
- **Line 15:** this is a constant time operation, so it gets assigned $O(1)$.

At this point, we have the following (the boxes represent the scope of the loops):

```
1   void foo(int32_t arr[], size_t size) {
2     int32_t num = static_cast<int32_t>(size);
3     while (num > 0) {
4       for (size_t j = 0; j < size; ++j) {
5         for (size_t k = size - 1; k > (size / 2); --k) {
6           std::cout << arr[k] << '\n';
7           sum_array(arr, size);
8         } // for k
9       } // for j
10      --num;
11    } // while
12    for (size_t i = 0; i < size; ++i) {
13      std::cout << arr[i] << '\n';
14    } // for i
15    return;
16  } // foo()
```

```
1
2   O(1)
3   O(n):
4     O(n):
5       O(n):
6         O(1)
7         O(n)
8
9
10    O(1)
11
12  O(n):
13    O(1)
14
15  O(1)
16
```

Now, we will walk through the code and condense the loops. The first loop we encounter is a triple nested loop spanning lines 3-11, consisting of one `while` loop and two `for` loops. Since this loop is nested, we start from the innermost loop and work outwards. We will first begin by analyzing the `for` loop defined on line 5 (the innermost loop):

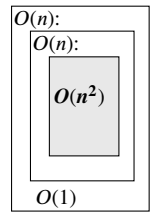
```
3     while (num > 0) {
4       for (size_t j = 0; j < size; ++j) {
5         for (size_t k = size - 1; k > (size / 2); --k) {
6           std::cout << arr[k] << '\n';
7           sum_array(arr, size);
8         } // for k
9       } // for j
10      --num;
11    } // while
```

```
3   O(n):
4     O(n):
5       O(n):
6         O(1)
7         O(n)
8
9
10    O(1)
11
```

When dealing with a non-nested loop, we first compute the complexity class of the loop body (lines 6-7). The time complexity of lines 6 and 7 is $O(1)+O(n)$, which can be simplified to $O(n)$ after dropping lower order terms. Thus, the body of the loop (lines 6 and 7) runs in $O(n)$ time. Since lines 6-7 run $O(n)$ times inside the innermost `for` loop defined on line 5, the time complexity of the innermost loop thus is $O(n) \times O(n) = O(n^2)$. After condensing this loop into a single complexity class $O(n^2)$, we get the following:

```
3     while (num > 0) {
4       for (size_t j = 0; j < size; ++j) {
5         for (size_t k = size - 1; k > (size / 2); --k) {
6           std::cout << arr[k] << '\n';
7           sum_array(arr, size);
8         } // for k
9       } // for j
10      --num;
11    } // while
```

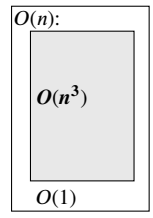
running this entire loop takes $O(n^2)$ time



Now, we move on to the next innermost loop, defined on line 4. This loop runs n times, and the body of the loop runs in $O(n^2)$ time. Thus, the combined time complexity of the two loops is $O(n) \times O(n^2) = O(n^3)$. Lines 4-9 have now been condensed into a single time complexity class:

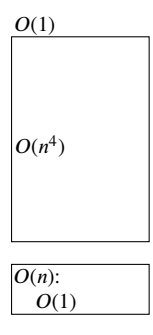
```
3     while (num > 0) {
4       for (size_t j = 0; j < size; ++j) {
5         for (size_t k = size - 1; k > (size / 2); --k) {
6           std::cout << arr[k] << '\n';
7           sum_array(arr, size);
8         } // for k
9       } // for j
10      --num;
11    } // while
```

running this entire loop takes $O(n^3)$ time



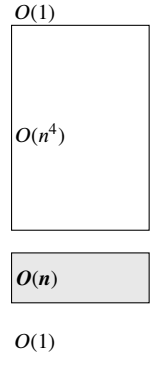
We have one loop left in this nested loop (the `while` loop defined on line 3). The body of this loop has an overall time complexity of $O(n^3)+O(1) = O(n^3)$, and it is run a total of $O(n)$ times. Thus, the overall time complexity of the `while` loop on line 3 is $O(n) \times O(n^3) = O(n^4)$. The nested loop on lines 3-11 has been condensed into a single time complexity of $O(n^4)$, so now we can move on to the next loop on line 12.

```
1   void foo(int32_t arr[], size_t size) {
2     int32_t num = static_cast<int32_t>(size);
3     while (num > 0) {
4       for (size_t j = 0; j < size; ++j) {
5         for (size_t k = size - 1; k > (size / 2); --k) {
6           std::cout << arr[k] << '\n';
7           sum_array(arr, size);
8         } // for k
9       } // for j
10      --num;
11    } // while
12    for (size_t i = 0; i < size; ++i) {
13      std::cout << arr[i] << '\n';
14    } // for i
15    return;
16  } // foo()
```



The loop on line 12 runs a $O(1)$ operation $O(n)$ times, so the complexity of this loop is $O(n) \times O(1) = O(n)$.

```
1   void foo(int32_t arr[], size_t size) {
2     int32_t num = static_cast<int32_t>(size);
3     while (num > 0) {
4       for (size_t j = 0; j < size; ++j) {
5         for (size_t k = size - 1; k > (size / 2); --k) {
6           std::cout << arr[k] << '\n';
7           sum_array(arr, size);
8         } // for k
9       } // for j
10      --num;
11    } // while
12    for (size_t i = 0; i < size; ++i) {
13      std::cout << arr[i] << '\n';
14    } // for i
15    return;
16  } // foo()
```



After dealing with all the loops, we have managed to condense this function into four sequential steps: an assignment that takes $O(1)$ time, a triple-nested `while` loop that runs in $O(n^4)$ time, and `for` loop that runs in $O(n)$ time, and a return statement that takes $O(1)$ time. Since these steps happen one after another, we can add their complexities together to obtain the overall time complexity of the `foo()` function: $O(1)+O(n^4)+O(n)+O(1) = O(n^4)$.

Note that lower order terms can be ignored, and that the time complexity of the entire `foo()` function is governed by the term with the largest time complexity class. Here, the nested loop that spans lines 3-11 runs in $O(n^4)$ time, which dominates all the other components of the function. As a result, this loop is a bottleneck that determines the rate of growth for the entire `foo()` function.

4.4 Logarithmic Runtime

※ 4.4.1 *Analyzing Logarithmic Runtime*

In the previous two examples, we dealt with loops that ran n iterations, where n is the input size. An example of such a loop is shown below.

```
for (int32_t i = 1; i < input_size; ++i) { ... }
```

However, not all loops behave like this. What if we changed the update condition so that the loop counter is *doubled* rather than incremented?

```
for (int32_t i = 1; i < input_size; i *= 2) { ... }
```

How many times does the loop run now? On the first iteration, the loop counter `i` is 1. Then, it becomes 2, then 4, then 8, then 16, doubling with each iteration until it reaches the value of `input_size`, or n. Thus, the number of times the loop runs is equal to the number of times we can double `i` before it hits `input_size`. In mathematical terms, this can be represented as:

$$2^k = n$$

where k represents the number of times the loop can run before `i` exceeds `input_size` or n. Solving for k, we get:

$$k = \log_2(n)$$

Thus, the second `for` loop (where `i` is doubled with each iteration) runs on the order of $\log_2(n)$ times, or $O(\log_2(n))$ times. For example, suppose the input size is 64, and the loop counter `i` doubles with each iteration. The loop would run six times: when `i` equals 1, 2, 4, 8, 16, and 32. After the sixth iteration runs, `i` gets doubled to 64, and the loop exits. This makes sense with our analysis, since $\log_2(64)$ is indeed 6. If the input size were a non-power of two like 65, the loop would run seven times, which is still on the order of $\log_2(n)$.

Division works similarly. Consider the following loop:

```
for (int32_t i = input_size; i > 1; i /= 2) { ... }
```

Here, the loop counter `i` is halved at each iteration of the loop. Now, we want to calculate the number of times we can halve the input size before we hit 1, since this is the number of times the loop runs. Once again, we can write this in mathematical terms as:

$$n\left(\frac{1}{2}\right)^k = 1$$

Solving for k, we can rewrite this as:

$$k = \log_2(n)$$

Division gives us the same result as multiplication. This leads us to an interesting conclusion: if multiplication or division is involved in the update of a loop, there is a good chance that a logarithmic term is in play. Furthermore, if the search space of a problem gets cut in half (or by some other factor) with each iteration, a logarithmic term is likely involved.

Here is an example of a logarithmic algorithm in practice: suppose you wanted to find someone in a phone book, where the phone book has a total of n pages. If you looked through every page of the phone book until you find the person you want, that would be an $O(n)$ algorithm. However, if the phone book is alphabetical by last name, you can use this fact to reduce the work you have to do! For example, if you wanted to find a last name starting with the letter "P", you could completely ignore the first half of the book since you know that the letter "P" must be in the second half. You can continue this process by flipping to the middle page of the second half and checking if "P" comes before or after. This eliminates half of the remaining pages at each step (similar to the `i /= 2` update of the loop), leading to a logarithmic algorithm with complexity $O(\log_2(n))$. In other words, you would only need to check at most $O(\log_2(n))$ pages of a phone book with n pages to find any person you want!

Example 4.7 Express the time complexity of the `foo()` function using big-O notation. Let the input size n be the size of the array.

```
1  int32_t foo(int32_t arr[], size_t size) {
2    int32_t total = 0;
3    for (size_t i = 1; i < size; i *= 2) {
4      total += arr[i];
5    } // for i
6    return total;
7  } // foo()
```

Walking through the code and assigning each line with its complexity results in the following:

```
1  int32_t foo(int32_t arr[], size_t size) {          1
2    int32_t total = 0;                               2  O(1)
3    for (size_t i = 1; i < size; i *= 2) {           3  O(log2(n)):
4      total += arr[i];                               4    O(1)
5    } // for i                                       5
6    return total;                                    6  O(1)
7  } // foo()                                         7
```

Here, the loop counter `i` is multiplied by 2 with each iteration, so the number of times the loop runs is $O(\log_2(n))$. Since the body of the loop runs in $O(1)$ time, the total time complexity of the `for` loop on line 3 is $O(\log_2(n)) \times O(1) = O(\log_2(n))$.

```
1  int32_t foo(int32_t arr[], size_t size) {
2    int32_t total = 0;
3    for (size_t i = 1; i < size; i *= 2) {
4      total += arr[i];
5    } // for i
6    return total;
7  } // foo()
```

```
1
2  O(1)
3
4  O(log2(n))
5
6  O(1)
7
```

The overall time complexity of the `foo()` function is thus $O(1) + O(\log_2(n)) + O(1)$, or simply $O(\log_2(n))$ after dropping lower order terms.

Example 4.8 Express the time complexity of the `foo()` function using big-O notation, in terms of the input value n.

```
1  void foo(int32_t n) {
2    for (int32_t i = 1; i < n; ++i) {
3      for (int32_t j = 1; j < log2(n); j *= 2) {
4        for (int32_t k = 1; k < n; k *= 2) {
5          std::cout << "Potato!\n";
6        } // for k
7      } // for j
8    } // for i
9  } // foo()
```

As before, we will walk through each line of this function and assign it with the time required to execute the line:

- **Line 2:** this loop runs from 1 to n, incrementing the counter by one with each iteration, so it gets assigned with $O(n)$.
- **Line 3:** this loop runs from 1 to $\log_2(n)$, multiplying the counter by two with each iteration. Thus, the number of times this loop runs is equal to k in the following equation:

$$2^k = \log_2(n)$$
$$k = \log_2(\log_2(n))$$

- **Line 4:** this loop runs from 1 to n, multiplying the counter by two with each iteration. The number of times this loop runs is $O(\log_2(n))$.
- **Line 5:** this is a constant time operation, so it gets assigned $O(1)$.

```
1  void foo(int32_t n) {
2    for (int32_t i = 1; i < n; ++i) {
3      for (int32_t j = 1; j < log2(n); j *= 2) {
4        for (int32_t k = 1; k < n; k *= 2) {
5          std::cout << "Potato!\n";
6        } // for k
7      } // for j
8    } // for i
9  } // foo()
```

```
1
2  O(n):
3    O(log2(log2(n))):
4      O(log2(n)):
5        O(1)
6
7
8
9
```

The innermost loop on line 4 runs a $O(1)$ operation $O(\log_2(n))$ times, so the overall time complexity of this loop is $O(\log_2(n)) \times O(1) = O(\log_2(n))$.

```
1  void foo(int32_t n) {
2    for (int32_t i = 1; i < n; ++i) {
3      for (int32_t j = 1; j < log2(n); j *= 2) {
4        for (int32_t k = 1; k < n; k *= 2) {
5          std::cout << "Potato!\n";
6        } // for k
7      } // for j
8    } // for i
9  } // foo()
```

```
1
2  O(n):
3    O(log2(log2(n))):
4      O(log2(n))
5
6
7
8
9
```

The loop on line 3 runs the loop on line 4 $O(\log_2(\log_2(n)))$ times, so its complexity is $O(\log_2(\log_2(n)) \times O(\log_2(n)) = O(\log_2(\log_2(n)) \times \log_2(n))$.

```
1  void foo(int32_t n) {
2    for (int32_t i = 1; i < n; ++i) {
3      for (int32_t j = 1; j < log2(n); j *= 2) {
4        for (int32_t k = 1; k < n; k *= 2) {
5          std::cout << "Potato!\n";
6        } // for k
7      } // for j
8    } // for i
9  } // foo()
```

```
1
2  O(n):
3
4
5    O(log2(log2(n)) × log2(n))
6
7
8
9
```

The loop on line 2 runs the loop on line 3 $O(n)$ times, so its complexity is $O(n) \times O(\log_2(\log_2(n)) \times \log_2(n)) = O(n \times \log_2(\log_2(n)) \times \log_2(n))$. This is also the overall time complexity of the `foo()` function: $O(n \times \log_2(\log_2(n)) \times \log_2(n))$.

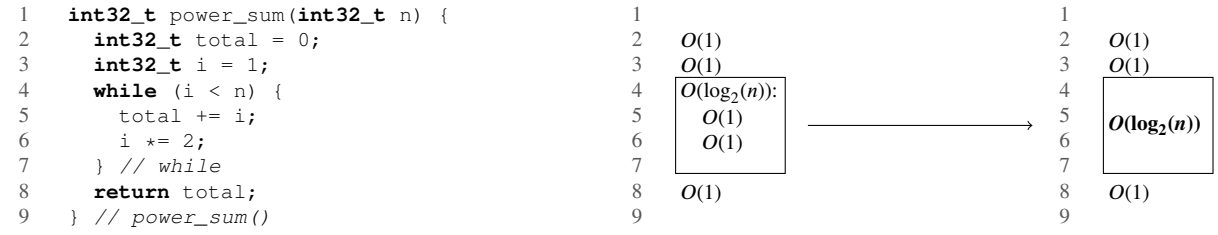
```
1  void foo(int32_t n) {
2    for (int32_t i = 1; i < n; ++i) {
3      for (int32_t j = 1; j < log2(n); j *= 2) {
4        for (int32_t k = 1; k < n; k *= 2) {
5          std::cout << "Potato!\n";
6        } // for k
7      } // for j
8    } // for i
9  } // foo()
```

```
1
2
3
4
5  O(n × log2(log2(n)) × log2(n))
6
7
8
9
```

Example 4.9 Express the time complexity of the `power_sum()` function using big-O notation, in terms of the input value n.

```
1  int32_t power_sum(int32_t n) {
2    int32_t total = 0;
3    int32_t i = 1;
4    while (i < n) {
5      total += i;
6      i *= 2;
7    } // while
8    return total;
9  } // power_sum()
```

Lines 2, 3, 5, 6, and 8 are all constant time operations. Thus, the time complexity of this function will be determined by the number of times the `while` loop on line 4 runs. The loop runs as long as `i` is less than `n`, and `i` is doubled with each iteration of the loop. Thus, the number of times the `while` loop runs is $O(\log_2(n))$. Since the body of the loop runs in $O(1)$ time, the time it takes for the entire `while` loop to execute is $O(\log_2(n))$ iterations $\times$ $O(1)$ work per iteration $= O(\log_2(n))$.



The overall time complexity of the `power_sum()` function is $O(1)+O(1)+O(\log_2(n))+O(1) = O(\log_2(n))$.

※ 4.4.2 *Does the Base of a Logarithm Matter?*

When dealing with logarithmic complexity terms, we often ignore the base of the logarithm. It turns out that it does not matter whether an algorithm is $O(\log_2(n))$ or $O(\log_3(n))$; we treat both as part of the same complexity class $O(\log(n))$. This is because we can always convert a log with one base to a log with another base by multiplying it with a constant, thanks to the *change of base formula* for logarithms:

$$\log_b(n) = \frac{\log_a(n)}{\log_a(b)}$$

To change a logarithmic expression with base a to an identical expression with base b, we simply have to multiply the base-a log with the constant $\frac{1}{\log_a(b)}$. Because of this identity, the base does not matter! We will go over an example of this in the following example.

Example 4.10 Show that $\log_2(n)$ is both $O(\log_2(n))$ and $O(\log_3(n))$.

$f(n)$ is by definition also $O(f(n))$, so $\log_2(n)$ must be $O(\log_2(n))$. To show that $\log_2(n)$ is also $O(\log_3(n))$, we will use the change of base formula:

$$\log_2(n) = \frac{\log_3(n)}{\log_3(2)}$$

In other words, $\log_2(n)$ can be rewritten as $\frac{\log_3(n)}{\log_3(2)}$. This means that

$$\log_2(n) = O(\log_2(n)) = O\left(\frac{\log_3(n)}{\log_3(2)}\right) = O\left(\frac{1}{\log_3(2)} \times \log_3(n)\right)$$

Since we can ignore constants when expressing asymptotic time complexity, we can ignore the $\frac{1}{\log_3(2)}$ term, since it is a constant (≈ 1.58):

$$\log_2(n) = O\left(\frac{1}{\log_3(2)} \times \log_3(n)\right) = O(\log_3(n))$$

Thus, we have shown that $\log_2(n)$ is also $O(\log_3(n))$, and that $O(\log_2(n))$ and $O(\log_3(n))$ must be a part of the same complexity class! The same applies for any base. Because of the change of base formula, two logarithms with different bases only differ by a *constant* factor. As shown above, the difference between $\log_3(n)$ and $\log_2(n)$ is a constant factor of approximately 1.58. As a result, if you are in a logarithmic complexity class, it does not matter if the base is 2 or 3 or 999. While a $\log_3(n)$ algorithm does run faster than a $\log_2(n)$ algorithm for large enough input sizes, the speed difference should only differ by a constant amount that *does not grow as the input size grows*. Therefore, the base of the logarithm does not affect the growth of an algorithm, and thus is ignored in big-O notation.

Note that this does **NOT** apply to exponents! The base of an exponent **does** matter. For example, $O(2^n)$ and $O(8^n)$ are not part of the same complexity class. This is because $8^n = 2^n \times 2^{2n}$, and 2^{2n} is not a constant factor!

※ 4.4.3 *Logarithmic and Power Identities*

The following identities may be useful for working with time complexities that involve logarithms:

Log Identities	Power Identities
1) $\log_a(xy) = \log_a(x) + \log_a(y)$	1) $a^{n+m} = a^n a^m$
2) $\log_a(\frac{x}{y}) = \log_a(x) - \log_a(y)$	2) $a^{n-m} = \frac{a^n}{a^m}$
3) $\log_a(x^r) = r\log_a(x)$	3) $(a^n)^m = a^{nm}$
4) $\log_a\left(\frac{1}{x}\right) = -\log_a(x)$	4) $a^{-n} = \frac{1}{a^n}$
5) $\log_a(x) = \frac{\log(x)}{\log(a)}$	
6) $\log_a(a) = 1$	
7) $\log_a(1) = 0$	

Example 4.11 An algorithm has a runtime that can be expressed using the following equation. Express the asymptotic time complexity of this algorithm using big-O notation.

$$T(n) = \log(n) + \log(n^{17} \times 3^{n+5})$$

We will use the identities above to simplify the expression. First, using log identity 1, we can simplify $\log(n^{17} \times 3^{n+5})$ to $\log(n^{17}) + \log(3^{n+5})$. This leaves us with

$$T(n) = \log(n) + \log(n^{17}) + \log(3^{n+5})$$

Using log identity 3, we can further simplify $\log(n^{17})$ to $17\log(n)$. This leaves us with

$$T(n) = \log(n) + 17\log(n) + \log(3^{n+5})$$

Using power identity 1, we can simplify $\log(3^{n+5})$ to $\log(3^n \times 3^5)$. We can then use log identity 1 again to simplify $\log(3^n \times 3^5)$ to $\log(3^n) + \log(3^5)$. This leaves us with

$$T(n) = \log(n) + 17\log(n) + \log(3^n) + \log(3^5)$$

We can use log identity 3 again to simplify $\log(3^n)$ into $n\log(3)$. This leaves us with

$$T(n) = \log(n) + 17\log(n) + n\log(3) + \log(3^5) = 18\log(n) + n\log(3) + \log(243)$$

After dropping all coefficients and lower order terms in this equation, we can see that the highest order term here is n (from the $n\log(3)$, but $\log(3)$ is a constant coefficient, so it is dropped). Thus, $T(n) = O(n)$.

Example 4.12 Express the time complexity of the `foo()` function using big-O notation, in terms of m and n, where m and n are positive integers greater than 1.

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

Lines 2, 4, 6, and 9 are all constant time operations. We can tell that the inner `for` loop on line 5 runs a total of $\log_3(n)$ times since `i` is tripled with each iteration. However, how many times does the outer `while` loop on line 3 run, in terms of m and n?

To solve this, first imagine that the `while` loop condition were $m > 1$ instead of $m > n$. How many times would the `while` loop run? Since m is halved with each iteration, the outer `while` loop would run $\log_2(m)$ times.

However, we are stopping the loop once m reaches n, not when it reaches 1. As a result, we do not fully run the loop $\log_2(m)$ times. Instead, we *lose* $\log_2(n)$ iterations if the loop is stopped when m reaches n instead of when m reaches 1. Let's use an example to visualize this. Suppose m starts off with a value of 256. If we allowed the loop to run as long as $m > 1$, the loop would run a total of $\log_2(256) = 8$ times:

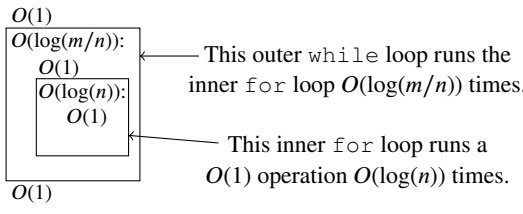
256 128 64 32 16 8 4 2

However, let's choose the value of n to be 32. How many iterations does the `while` loop run now? In this case, we only end up running the first three iterations, since the loop now terminates when m reaches 32:

256 **128** **64** 32 16 8 4 2

We lost $\log_2(32) = 5$ iterations of the loop when we set $n = 32$, and only $\log_2(256) - \log_2(32)$ loop iterations were run. In fact, for any m and n, the total number of times the `while` loop runs is equal to $\log_2(m) - \log_2(n)$, which can be rewritten as $\log_2(m/n)$ using log identities.

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



Since the inner loop executes in $O(\log(n))$ time, and the outer loop runs the inner loop $O(\log(m/n))$ times, the overall time complexity of the `foo()` function is $O(\log(m/n) \times \log(n))$.

Remark: A more direct way to solve this problem is to notice that m is halved at each iteration in the outer loop, which means that the value of m at the k^{th} iteration of this loop must be $m \times (1/2)^k$. Since our loop stops once m becomes equivalent to n, the total number of iterations run by the `while` loop can be expressed as k in the following equation:

$$m \times (1/2)^k = n$$

Solving for the number of iterations k, we get $k = \log_2(m/n)$. Therefore, the outer `while` loop runs the inner `for` loop $\log_2(m/n)$ times. Since the complexity of the inner loop is $\Theta(\log(n))$, the overall time complexity of the function is $O(\log(m/n) \times \log(n))$.

4.5 Loop Dependencies

Consider the following function. What is its time complexity? Assume that the input size n is the variable `size`.

```
1  int32_t foo(int32_t arr[], size_t size) {
2    int32_t count = 0;
3    for (size_t i = size; i > 1; i /= 2) {
4      for (size_t j = 0; j <= i; ++j) {
5        ++count;
6      } // for j
7    } // for i
8    return count;
9  } // foo()
```

This problem may look relatively straightforward upon first glance. You might be tempted to say that the time complexity is $O(n\log(n))$, since the outer loop appears to run $O(\log(n))$ times, and the inner loop appears to run $O(n)$ times. However, this is not accurate! **This is because the terminating condition of the *inner* loop depends on a value that is changing with the *outer* loop.** Notice that the value of `i` used to terminate the inner loop on line 4 depends on the current iteration of the outer `for` loop on line 3; as a result, the inner loop runs a different number of steps each time. Because of this, the interaction between the two loops cannot be determined if each loop is analyzed individually. To approach the problem, you must consider the entire loop dependency together and count the number of steps that occur with each iteration of the outermost loop in the dependency. This procedure is shown in the following examples.

Remark: For solving loop dependency problems, it may be useful to know the equation for the sum of a geometric series. Given a sequence where the number of terms is n, the first term is a, and the common ratio between terms is r, the sum of the first n terms can be expressed using the following equation:

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

Example 4.13 Express the time complexity of the `foo()` function using big-O notation. Let the input size n be the size of the array.

```
1  int32_t foo(int32_t arr[], size_t size) {
2    int32_t count = 0;
3    for (size_t i = size; i > 1; i /= 2) {
4      for (size_t j = 0; j <= i; ++j) {
5        ++count;
6      } // for j
7    } // for i
8    return count;
9  } // foo()
```

As mentioned previously, there is a loop dependency in this function: the inner `for` loop on line 4 depends on the value of `i` in the outer `for` loop on line 3. Thus, we will analyze both `for` loops together and count the total number of steps that occur with each iteration of the outer loop. On the first iteration of the outer loop, the value of `i` is n, or the input size. Therefore, the inner loop would also run n times on the first iteration of the outer loop. On the second iteration of the outer loop, the value of `i` is halved to $n/2$, so the inner loop will also run $n/2$ times.

We can see that the value of `i` is halved with each iteration, which also halves the number of times the inner loop is able to run. As a result, the runtime of the outer loop can be expressed as:

$$T(n) = n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \frac{n}{16} + \frac{n}{32} + \dots$$

since the inner loop runs n times on the first iteration, $n/2$ times on the second iteration, $n/4$ times on the third iteration, and so on.

Since the outer loop halves the value of `i` until it reaches a value of 1, the inner loop must run a total of $\log_2(n)$ times. After factoring out the n from our expression, we get the following:

$$T(n) = n\left(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \frac{1}{32} + \dots\right) = n\sum_{k=0}^{\log_2(n)} \left(\frac{1}{2}\right)^k$$

To solve this, we can use the equation for the sum of a geometric series.

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

Here, the number of terms is $\log_2(n)+1$, the common ratio is $1/2$, and the first term is 1:

$$T(n) = n\sum_{k=0}^{\log_2(n)} \left(\frac{1}{2}\right)^k \approx n\left[\frac{1\left(1-\left(\frac{1}{2}\right)^{\log_2(n)+1}\right)}{1-\frac{1}{2}}\right] = n\left[2\left(1-\left(\frac{1}{2}\right)^{\log_2(n)+1}\right)\right] = n\left[2-2\left(\left(\frac{1}{2}\right)^{\log_2(n)+1}\right)\right]$$

We can see that the coefficient $2-2((1/2)^{\log_2(n)+1})$ is a constant value that is roughly equal to 2, since the term that is subtracted approaches zero as n gets large. Thus, we can conclude that $n(2-2((1/2)^{\log_2(n)+1})) \approx 2n = O(n)$, and that the nested loop therefore takes $O(n)$ time.

```
1  int32_t foo(int32_t arr[], size_t size) {
2    int32_t count = 0;
3    for (size_t i = size; i > 1; i /= 2) {
4      for (size_t j = 0; j <= i; ++j) {
5        ++count;
6      } // for j
7    } // for i
8    return count;
9  } // foo()
```

```
1
2  O(1)
3
4
5  O(n)
6
7
8  O(1)
9
```

The number of steps that this nested loop takes can be expressed using the summation $n(1+1/2+1/4+1/8+1/16+...) \approx 2n$
Thus, the complexity of this nested loop is $O(n)$.

Since lines 2 and 6 take constant time, the $O(n)$ nested `for` loop on line 3 takes the most time and is thus the dominant term of the entire function. The time complexity of `foo()` is therefore $O(n)$.

Example 4.14 Express the time complexity of the `foo()` function using big-O notation, in terms of the input size n.

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

Here, we have a loop dependency on lines 3 and 4. The number of times the loop on line 4 runs depends on the value of `j`, which changes in the `for` loop on line 3. Thus, we will need to consider these two loops together.

On the first iteration of the line 3 loop, the line 4 loop runs 1 time. On the second iteration of the line 3 loop, the line 4 loop runs 3 times. On the third iteration of the line 3 loop, the line 4 loop runs 9 times. This continues until the value of `j` reaches n. From this, we can see that the line 3 loop runs the following number of iterations (using the geometric series formula, where $a = 1$, $r = 3$, and $n = \log_3(n)+1$):

$$1+3+9+27+\dots+n = \sum_{k=0}^{\log_3(n)} 3^k = \frac{1\left(1-3^{\log_3(n)+1}\right)}{1-3} = \frac{1-3\left(3^{\log_3(n)}\right)}{1-3} = \frac{1-3n}{1-3} = \frac{3n-1}{2} = O(n)$$

Thus, the loop on line 3 runs in $O(n)$ time:

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

```
1
2  O(n):
3
4
5  O(n)
6
7
8
9
```

The number of steps that this nested loop takes can be expressed using the summation $1+3+9+27+81+\dots+n \approx 1.5n$
Thus, the complexity of this nested loop is $O(n)$.

We can see that the loop on line 2 runs n times, so the loop on line 3 must be executed n times as well. Thus, the time complexity of this function is $O(n) \times O(n) = O(n^2)$, since the $O(n)$ loop on line 3 is run n times in the loop on line 2.

4.6 Big-O, Big-Θ, and Big-Ω

In the previous few sections, we measured asymptotic time complexities using big-O. In fact, this is the notation that you will likely see in industry. However, for this course and in academia, the letter "O" on its own may not be specific enough to express certain time complexities. In this section, we will expand upon the definition of big-O notation by introducing two new notations: big-Θ and big-Ω.

Even though we used big-O as our only notation in the previous sections, the O-notation is actually used to denote an **asymptotic upper bound**. The formal definition of big-O is as follows:

A function $f(n)$ is $O(g(n))$ if there exist positive constants c and n_0 such that $f(n) \leq cg(n)$ for all input sizes $n \geq n_0$.

Big-O can often be used to denote a worst-case time complexity. If an algorithm has a time complexity of $O(g(n))$, it means that the algorithm's runtime growth will be no worse than $g(n)$ for input size n. For example, a $O(n^2)$ algorithm will scale quadratically with respect to input size *at worst*. Big-O notation is similar to a *less than or equal to* relationship. If $T(n) = O(g(n))$, $T(n) \leq g(n)$ as input size n grows without bound.

In addition to big-O, we also have big-Ω, which represents an **asymptotic lower bound**. The formal definition of big-Ω is as follows:

A function $f(n)$ is $\Omega(g(n))$ if there exist positive constants c and n_0 such that $f(n) \geq cg(n)$ for all input sizes $n \geq n_0$.

Big-Ω can often be used to denote a best-case time complexity. If an algorithm has a time complexity of $\Omega(g(n))$, it means that the algorithm's runtime growth will be no better than $g(n)$ for input size n. For example, a $\Omega(n^2)$ algorithm will scale quadratically with respect to input size *at best*. Big-Ω notation is similar to a *greater than or equal to* relationship. If $T(n) = \Omega(g(n))$, $T(n) \geq g(n)$ as n grows without bound.

Notice that, while big-O and big-Ω are upper and lower bounds, they are *not always tight* bounds. If someone came to you and asked you how many students were enrolled in EECS 281 this semester, it would not be wrong to say "less than one million" or "more than five." The same thing applies to big-O and big-Ω. Because big-O specifies an upper bound, any algorithm that grows slower than $g(n)$ can be treated as a $O(g(n))$ algorithm. You could assign a constant time algorithm a time complexity of $O(2^n)$, and it would not be wrong because $O(1)$ grows slower than $O(2^n)$! However, much like the "less than one million" response, this does not provide any useful information. The same applies for the "more than five" response for big-Ω. An algorithm that runs in exponential time is technically also $\Omega(1)$, but saying that this algorithm will not run faster than constant time does not provide any meaningful analysis.

To address this issue, we use something called big-Θ to denote an **asymptotic tight bound**. Big-Θ is the tightest bound we can assign to an algorithm. For example, an algorithm with a $O(n^2)$ complexity will not experience anything worse than quadratic growth, and an algorithm with a $\Omega(n^2)$ complexity will not experience anything better than quadratic growth. However, if you are given a $\Theta(n^2)$ algorithm, you can expect its runtime to scale quadratically as the input size gets larger. The formal definition of big-Θ is defined as follows:

A function $f(n)$ is $\Theta(g(n))$ if there exist positive constants c_1, c_2 and n_0 such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all input sizes $n \geq n_0$.

Big-Θ is similar to an *equal to* relationship. If $T(n) = \Theta(g(n))$, we can expect $T(n)$ and $g(n)$ to have the same order of growth. **In this course, we will typically use the tightest possible bound to denote the asymptotic complexity of an algorithm.** (In fact, the complexities from the earlier examples could have been more accurately denoted using big-Θ rather than big-O.)

By definition, if a function $T(n)$ is both $O(g(n))$ and $\Omega(g(n))$, it is also $\Theta(g(n))$. Like with the $\leq$, $\geq$, and $=$ operations, if $T(n) \leq g(n)$ and $T(n) \geq g(n)$ are **both** true, then $T(n) = g(n)$ must also be true.

Complexity Rules: The following rules can be used when dealing with asymptotic time complexities.

- **Rules of transitivity:**
  1. $f(n) = \Theta(g(n))$ **AND** $g(n) = \Theta(h(n))$ **IMPLY** $f(n) = \Theta(h(n))$
     - If $f(n)$ grows at the same rate as $g(n)$, and $g(n)$ grows at the same rate of $h(n)$, then $f(n)$ must grow at the same rate as $h(n)$.
  2. $f(n) = O(g(n))$ **AND** $g(n) = O(h(n))$ **IMPLY** $f(n) = O(h(n))$
     - If $f(n)$ grows slower than or equal to $g(n)$, and $g(n)$ grows slower than or equal to $h(n)$, then $f(n)$ must grow slower than or equal to $h(n)$.
  3. $f(n) = \Omega(g(n))$ **AND** $g(n) = \Omega(h(n))$ **IMPLY** $f(n) = \Omega(h(n))$
     - If $f(n)$ grows faster than or equal to $g(n)$, and $g(n)$ grows faster than or equal to $h(n)$, then $f(n)$ must grow faster than or equal to $h(n)$.
- **Rules of reflexivity (i.e., a function is big-O of itself):**
  1. $f(n) = \Theta(f(n))$
  2. $f(n) = O(f(n))$
  3. $f(n) = \Omega(f(n))$
- **Rules of symmetry and transpose symmetry:**
  1. $f(n) = \Theta(g(n))$ **if and only if** $g(n) = \Theta(f(n))$
     - If $f(n)$ grows at the same rate as $g(n)$, then $g(n)$ must grow at the same rate as $f(n)$.
  2. $f(n) = O(g(n))$ **if and only if** $g(n) = \Omega(f(n))$
     - If $f(n)$ grows slower than or equal to $g(n)$, and $g(n)$ must grow faster than or equal to $f(n)$.

When working with complexities, you may also encounter two other notations: little-o and little-ω. These two notations are similar to their counterparts big-O and big-Ω, except that they are used to describe an upper or lower bound that is *not* tight (i.e., *strictly* less than or greater).

More formally:

- A function $f(n)$ is $o(g(n))$ if there exist positive constants c and n_0 such that $f(n) < cg(n)$ for all input sizes $n \geq n_0$.
- A function $f(n)$ is $\omega(g(n))$ if there exist positive constants c and n_0 such that $f(n) > cg(n)$ for all input sizes $n \geq n_0$.

However, you do not need to worry about these two notations for this class.

Example 4.15 Show that $\log(n!) = \Theta(n\log(n))$.

This is an interesting question, as it is not intuitive that taking the log of a factorial term would result in something with the same complexity class as $n\log(n)$. However, with an understanding of log identities, we can prove that this is in fact true. For our proof, we will show that $\log(n!)$ is both $O(n\log(n))$ and $\Omega(n\log(n))$ — if both of these are true, then by definition, $\log(n!)$ must also be $\Theta(n\log(n))$.

Before we begin, the *factorial* of a non-negative integer n (denoted as $n!$) is defined as the product of all positive integers less than or equal to n. In other words, $n! = n\times(n-1)\times(n-2)\times\ldots\times 1$. By definition, the special case of 0! is equal to 1.

First, we will show that $\log(n!) = O(n\log(n))$, or that $\log(n!)$ will never grow at a rate that exceeds a constant multiple of $n\log(n)$. To do this, we can separate the individual terms of the factorial, as follows:

$$\log(n!) \;=\; \log(n\times(n-1)\times\ldots\times 2\times 1)$$

Using our log identities, we can rewrite this as:

$$\log(n!) \;=\; \log(n)+\log(n-1)+\ldots+\log(2)+\log(1)$$

We know that this sum is less than $n\log(n)$, which proves that $\log(n!) = O(n\log(n))$.

$$\log(n)+\log(n-1)+\ldots+\log(2)+\log(1) \;\leq\; \log(n)+\log(n)+\ldots+\log(n)+\log(n) \;=\; n\log(n)$$

Now, we will show that $\log(n!) = \Omega(n\log(n))$, or that $\log(n!)$ will always grow at least as fast as a constant multiple of $n\log(n)$. Once again, we will start by using log identities to expand the factorial term:

$$\log(n!) \;=\; \log(n)+\log(n-1)+\ldots+\log(2)+\log(1)$$

However, this time we will remove the lower half of these terms from this sum. That is, we will remove all terms $\log(k)$ where $k < \frac{n}{2}$ (the terms that are grayed out below are removed):

$$\log(n!) \;=\; \log(n)+\log(n-1)+\ldots+\log\left(\tfrac{n}{2}+1\right)+\log\left(\tfrac{n}{2}\right)+\log\left(\tfrac{n}{2}-1\right)+\ldots+\log(2)+\log(1)$$
$$\log(n!) \;\geq\; \log(n)+\log(n-1)+\ldots+\log\left(\tfrac{n}{2}+1\right)+\log\left(\tfrac{n}{2}\right)+\log\left(\tfrac{n}{2}-1\right)+\ldots+\log(2)+\log(1)$$

We know that the other terms must be greater than or equal to $\frac{n}{2}\log\left(\frac{n}{2}\right)$, which is what we get if we replaced all remaining terms with $\log\left(\frac{n}{2}\right)$:

$$\log(n!) \;\geq\; \log(n)+\log(n-1)+\ldots+\log\left(\tfrac{n}{2}+1\right)+\log\left(\tfrac{n}{2}\right) \;\geq\; \log\left(\tfrac{n}{2}\right)+\log\left(\tfrac{n}{2}\right)+\ldots+\log\left(\tfrac{n}{2}\right)+\log\left(\tfrac{n}{2}\right) \;=\; \tfrac{n}{2}\log\left(\tfrac{n}{2}\right)$$

Thus, we have shown that $\log(n!) = \Omega\left(\frac{n}{2}\log\left(\frac{n}{2}\right)\right)$ Now, we will need to show that $\frac{n}{2}\log\left(\frac{n}{2}\right) = \Theta(n\log(n))$, which would prove that $\log(n!)$ is also $\Omega(n\log(n))$. To do so, we can rewrite $\log\left(\frac{n}{2}\right)$ as $\log(n)-\log(2)$ using our log identities. $\frac{n}{2}\log\left(\frac{n}{2}\right)$ can thereby be rewritten as:

$$\tfrac{n}{2}\log\left(\tfrac{n}{2}\right) \;=\; \tfrac{n}{2}(\log(n)-\log(2)) \;=\; \tfrac{n}{2}\log(n)-\tfrac{n}{2}\log(2) \;=\; \tfrac{1}{2}(n\log(n))-\tfrac{\log(2)}{2}(n)$$

Since coefficients and lower-order terms can be dropped, $\frac{1}{2}(n\log(n))-\frac{\log(2)}{2}(n) = \Theta(n\log(n))$.

Since $\log(n!) = \Omega\left(\frac{n}{2}\log\left(\frac{n}{2}\right)\right)$ and $\frac{n}{2}\log\left(\frac{n}{2}\right) = \Theta(n\log(n))$, we can conclude that $\log(n!) = \Omega(n\log(n))$.

We have successfully shown that $\log(n!)$ is both $O(n\log(n))$ and $\Omega(n\log(n))$. Therefore, $\log(n!) = \Theta(n\log(n))$.

4.7 Complexity Metrics

There are three primary ways we can express the complexity of an algorithm. Given an algorithm and a fixed input size n, we can either calculate its best, worst, or average-case time complexity.

The **best-case time complexity** is the time complexity of an algorithm in the very best case. This is the lower bound on the running time of an algorithm, and it describes the algorithm's behavior under optimal conditions.

The **worst-case time complexity** is the time complexity of an algorithm in the very worst case. This is the upper bound on the running time of an algorithm, and it describes the worst rate of runtime growth that could occur over all possible inputs. The worst-case time complexity is a common metric you will see, as it provides valuable insight into how bad an algorithm's runtime may be. However, this metric does not take into account the frequency of the worst-case scenario happening, so it is not always the only metric that should be considered. In cases where the worst-case occurs very rarely, the worst-case time complexity may not accurately reflect the true performance of an algorithm (this is addressed using a concept known as *amortized complexity*, which is discussed in chapter 12).

The **average-case time complexity** is the average time complexity of an algorithm over all possible inputs of size n. Much like finding an average in a given set of values, the average-case time complexity can be determined by summing up the resource usage of all possible inputs of size n and dividing this sum by the total number of inputs of size n.

Notice that the input size we feed into an algorithm is fixed for all three complexity metrics. When analyzing the best- and worst-case scenarios, we can only change the contents of the input and not its size. For example, it would be misleading to claim that the best-case scenario for a sorting algorithm occurs when the size of the array is set to 1, since that messes with the size of the input. It would be correct to claim, however, that the best case occurs when the input is already sorted, since that does not mess with the input size.

Example 4.16 You are given an array of n integers, and you want to find the position of a number that you know is in the array. You have an algorithm that conducts a linear search through this array to see if the number you want to find is at each position (i.e., it looks at the first element, then the second, then the third, and so on until it finds the number). What is the best-case, worst-case, and average-case time complexity of this algorithm in terms of n?

The best-case scenario occurs if the number you want to find is the very first number in the array. The algorithm would check the first element, see that it matches the number you want, and return. This would take constant time! Thus, the best-case time complexity of this algorithm is $\Theta(1)$. The following example illustrates the best-case scenario. Suppose you want to find the number 5 in this array — the linear search would find 5 on its very first try, so it only has to do a constant amount of work.

5	2	3	9	6	...	23	17	8	1
0	1	2	3	4	...	$n-4$	$n-3$	$n-2$	$n-1$

The worst-case scenario occurs if the number you want to find is the very last number in the array (or if it does not exist). This would force the algorithm to check every element in the array. Since there are n elements in total, the worst-case time complexity of this algorithm is $\Theta(n)$. The following example illustrates the worst-case scenario. Suppose you want to find the number 1 in this array (and assuming that 1 only appears once) — the linear search would have to check all n elements before it reaches 1, which takes $\Theta(n)$ time.

5	2	3	9	6	...	23	17	8	1
0	1	2	3	4	...	$n-4$	$n-3$	$n-2$	$n-1$

Now, let's find the average-case time complexity. To do so, we will compute the average time complexity over all possible cases when given an input size of n. For the sake of simplicity, we will assume that all cases are uniformly distributed (i.e., the likelihood of the target element ending up at index 0, index 1, …, index $n-1$ is equal).

First, we have the case where the element we want to find is the 1st element in the array. The number of elements our algorithm would have to check in this case is 1. Second, we have the case where the element we want to find is the 2nd element in the array. The number of elements our algorithm would have to check in this case is 2.

We can continue this process, keeping track of the number of values our algorithm would need to check if the element we want is the 3rd in the array, the 4th in the array, the 5th in the array, … , up until the n^{th} in the array. After doing this, we would find that:

- 1 comparison operation is needed in the case where the element we want is the 1st element in the array
- 2 comparison operations is needed in the case where the element we want is the 2nd element in the array
- …
- n comparison operations is needed in the case where the element we want is the n^{th} element in the array

To calculate the average-case time complexity, we would need find the average number of operations out of all of these possible cases. Using the definition of an average, we get:

$$\text{average number of operations} = \frac{\text{total number of operations over all possible cases}}{\text{number of possible cases}} = \frac{1+2+\ldots+n}{n} = \frac{\frac{n(n+1)}{2}}{n} = \frac{n+1}{2} = \Theta(n)$$

After carrying out the division, we find that the average number of operations scales with a complexity of $\Theta(n)$. Since each operation takes constant time, the runtime must also scale with a complexity of $\Theta(n)$. Thus, the average-case time complexity of this algorithm is $\Theta(n)$.

For those with a statistics background, here is a formal proof of the above calculation. We know that the input is uniformly distributed, and that the size of the array is n. If we define X as a random variable that represents the number of comparison operations needed before the target value is found, then the probability that we do x comparisons is $P(X = x) = \frac{1}{n}$ (since the distribution is uniform). Using the property of expectation, we can determine the expected number of comparison operations as follows:

$$E[X] = \sum_{x=1}^{n} xP(x) = \sum_{x=1}^{n} \frac{x}{n} = \frac{1}{n} + \frac{2}{n} + \ldots + \frac{n}{n} = \frac{n+1}{2} = \Theta(n)$$

In this class, however, you likely will not have to calculate the average-case time complexity from scratch. You will typically be given the average-case time complexities of important algorithms, without asking you to derive these results on your own. However, you should still understand what an average-case analysis measures and how to calculate it if you ever need to do so.

4.8 Additional Rules for Determining Complexity

It is possible for two functions to have neither a big-O, big-Θ, nor big-Ω relationship. A good example would be the trigonometric functions. For instance, if $f(n) = \sin(n)$ and $g(n) = \cos(n)$, then $f(n)$ is **NOT** $O(g(n)), \Theta(g(n))$, nor $\Omega(g(n))$. This is because there is no way to strictly bound one function by another, as shown by the graph on the next page. There is no constant that we can multiply $\cos(n)$ with to have it always grow faster or slower than $\sin(n)$ beyond some input size n_0. That being said, runtimes that follow a trigonometric pattern are relatively rare.

To show that a function $f(n)$ is $O(g(n))$, the following condition can be used. This condition is *sufficient* but *not necessary*. In other words, the following rule *cannot* be used to prove that $f(n) \neq O(g(n))$.

If $\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) = d < \infty$, then $f(n) = O(g(n))$. In other words, if the limit of $\frac{f(n)}{g(n)}$ as n grows toward infinity is a constant whose value is less than ∞, then $f(n)$ is $O(g(n))$.

In cases where n appears in both $f(n)$ and $g(n)$, L'Hopital's rule may have to be used:

L'Hopital's Rule: If $f(n)$ and $g(n)$ are differentiable functions and $\lim_{n\to c}\left(\frac{f(n)}{g(n)}\right) = \frac{0}{0}$ or $\frac{\infty}{\infty}$, then

$$\lim_{n\to c}\left(\frac{f(n)}{g(n)}\right) = \lim_{n\to c}\left(\frac{f'(n)}{g'(n)}\right)$$

where $f'(n)$ and $g'(n)$ represent the derivatives of $f(n)$ and $g(n)$, respectively.

Example 4.17 Prove that $\log_2(n)$ is $O(n)$ using the approach introduced in this section.

We can use the condition and L'Hopital's rule to complete this proof. Let $f(n) = \log_2(n)$ and $g(n) = n$. Using the limit definition, we have:

$$\lim_{n\to\infty}\left(\frac{\log_2(n)}{n}\right) = \frac{\infty}{\infty}$$

Since we get a result of $\frac{\infty}{\infty}$, we can use L'Hopital's rule. The derivative of $\log_2(n)$ is $\frac{1}{n\ln(2)}$, and the derivative of n is 1 (don't worry about calculating derivatives in this class — anything you might need will be provided, unless specified otherwise).

L'Hopital's rule gives us the following:

$$\lim_{n\to\infty}\left(\frac{\log_2(n)}{n}\right) = \lim_{n\to\infty}\left(\frac{\frac{d}{dn}\left[\log_2(n)\right]}{\frac{d}{dn}[n]}\right) = \lim_{n\to\infty}\left(\frac{1}{n\ln(2)}\right) = 0 < \infty$$

Since 0 is a constant that is less than ∞, $\log_2(n)$ must be $O(n)$.

Example 4.18 Prove that $\ln(n)$ is $O\left(\ln\left(\frac{n}{281}\right)\right)$ using the approach introduced in this section.

We can use the same process as above to complete this proof. Here, $f(n) = \ln(n)$ and $g(n) = \ln\left(\frac{n}{281}\right)$. We can simplify $\ln\left(\frac{n}{281}\right)$ to $\ln(n) - \ln(281)$ using logarithm identities. Using the limit definition, we have

$$\lim_{n\to\infty}\left(\frac{\ln(n)}{\ln(n)-\ln(281)}\right) = \frac{\infty}{\infty}$$

Since we get a result of $\frac{\infty}{\infty}$, we can use L'Hopital's rule. The derivative of $\ln(n)$ is $\frac{1}{n}$.

L'Hopital's rule gives us the following:

$$\lim_{n\to\infty}\left(\frac{\ln(n)}{\ln(n)-\ln(281)}\right) = \lim_{n\to\infty}\left(\frac{\frac{d}{dn}[\ln(n)]}{\frac{d}{dn}[\ln(n)]-\frac{d}{dn}[\ln(281)]}\right) = \lim_{n\to\infty}\left(\frac{\frac{d}{dn}[\ln(n)]}{\frac{d}{dn}[\ln(n)]-0}\right) = \lim_{n\to\infty}\left(\frac{1/n}{1/n}\right) = 1 < \infty$$

Since 1 is a constant that is less than ∞, $\ln(n)$ must be $O\left(\ln\left(\frac{n}{281}\right)\right)$.

4.9 Space Complexity and Auxiliary Space

In this chapter, we mainly focused on *time complexity*, which deals with how runtime scales with input size. However, runtime is not the only thing that can be expressed using big-O notation. In fact, we can use the notations introduced in this chapter to express the growth of *any computational resource* with respect to input size.

In this section, we will introduce the concept of space complexity (also known as memory complexity), which can be used to describe an algorithm's memory usage. The concept of complexity here is still the same — instead of measuring how runtime scales with input size, we will look at how memory usage scales with input size.

Space complexity is the amount of memory used by an algorithm to solve a problem, with respect to input size.

However, there is a caveat with this definition — since the space complexity looks the total memory usage of an algorithm, it also includes the memory used to store any input values themselves. This is not always something we want! If we do not consider the memory used for the input values themselves in our analysis, we are measuring the *auxiliary space* of the algorithm instead.

Auxiliary space is the temporary space allocated by an algorithm to solve a problem, with respect to input size. Auxiliary space only considers the additional memory used by the algorithm, and it does not include the memory used by the input values themselves.

For this class, we will mostly be dealing with *auxiliary* space. If an algorithm uses $\Theta(1)$ auxiliary space, the amount of additional memory needed to run the algorithm stays constant and does not depend on the input size. If an algorithm uses $\Theta(n)$ auxiliary space, the amount of additional memory needed to run the algorithm grows linearly with the size of the input. Similarly, if an algorithm uses $\Theta(n^2)$ auxiliary space, increasing the input size results in a quadratic increase in the amount of additional memory that is needed.

When dealing with space complexity, it is important to consider all possible memory sources. For example, if a function accepts an argument that is passed in by value (rather than by reference), a copy of that argument is made, which may contribute to auxiliary space. Furthermore, if an algorithm uses recursion, stack frames may be used with each recursive call. If the number of recursive calls depends on input size, the number of stack frames you need may depend on the input size as well! Because stack frames use up memory, an algorithm that does not explicitly allocate memory may still have an auxiliary space complexity worse than $\Theta(1)$.

Example 4.19 What is the auxiliary space complexity of the function `foo()`? The input size n is the size of the input array.

```
1  int32_t foo(int32_t arr[], size_t size) {
2    int32_t* bar = new int32_t[size];
3    int32_t counter = 0;
4    for (size_t i = 0; i < size; ++i) {
5      bar[i] = arr[size - 1 - i];
6    } // for i
7    for (size_t j = 0; j < size; ++j) {
8      counter += (bar[j] + arr[j]);
9    } // for j
10   delete[] bar;
11   return counter;
12 } // foo()
```

To determine the auxiliary space of this function, we need to identify where the function allocates memory and express this additional memory usage in terms of the input size. In this case, new memory is allocated on lines 2, 3, 4, and 7. The allocations on lines 3, 4, and 7 instantiate an integer, whose memory usage does not depend on the size of the input array. Thus, we say these lines use $\Theta(1)$ auxiliary space. The allocation on line 2, however, does depend on the size of the input, as the size of the `bar` array is set to the size of the input array. Hence, as input size grows, the amount of memory needed for `bar` also grows. This growth is linear, so the allocation on line 2 uses $\Theta(n)$ auxiliary space.

```
1  int32_t foo(int32_t arr[], size_t size) {          1
2    int32_t* bar = new int32_t[size];                2   Θ(n)
3    int32_t counter = 0;                             3   Θ(1)
4    for (size_t i = 0; i < size; ++i) {              4   Θ(1)
5      bar[i] = arr[size - 1 - i];                    5
6    } // for i                                       6
7    for (size_t j = 0; j < size; ++j) {              7   Θ(1)
8      counter += (bar[j] + arr[j]);                  8
9    } // for j                                       9
10   delete[] bar;                                    10
11   return counter;                                  11
12 } // foo()                                         12
```

Similar to before, we can remove coefficients and lower order terms when expressing space complexity. Because the $\Theta(n)$ memory allocation on line 2 dominates the $\Theta(1)$ allocations elsewhere in the function, the overall auxiliary space used by `foo()` is $\Theta(n)$.

Remark: If the size of `bar` did not depend on `size` (e.g., if the size of `bar` were set to 500 instead of `size` on line 2), then the auxiliary memory used by `bar` would be $500 = \Theta(1)$. If this were the case, the auxiliary space used by the entire `foo()` function would also be $\Theta(1)$, since the additional memory needed to execute the function would not depend on input size (i.e., would not change as the input size grows).