# Chapter 5
## *Recursion and Recurrence Relations*

---

### 5.1 Recursion

In the previous chapter, we introduced the concept of time complexity and several strategies that can be used to compute the time complexity of a given algorithm. One such strategy is to count the number of steps a program takes with respect to input size. However, this is not always easy! With an iterative algorithm, we can analyze loops to get an accurate estimate for time complexity in most cases. However, this becomes much trickier if recursion is involved.

A function is recursive if it calls itself in its own implementation. A recursive function is *linear recursive* if each invocation of the function can only make at most one recursive call, and *tree recursive* if each invocation can make more than one recursive call. Recursion is useful if the solution to a smaller subproblem (that is of the same type as the original problem) can be used to solve for the solution to a larger subproblem. To solve problems using recursion, we

1. assume that our recursive function already works and make a recursive call to a smaller subproblem to get its solution (this is known as the *recursive leap of faith*).
2. then, use the solution for this smaller subproblem to solve our original problem.

The soundness of recursion as an algorithmic approach rests on the assumption that, if a function calls itself over and over again using smaller and smaller input sizes, it will eventually reach an input size that is small enough for the problem to be solved trivially. This case is known as the **base case** of a recursive algorithm, and it allows the solution to build back up to the original input size.

The concept of recursion can be difficult to understand at first, so let's use the following real life example. Suppose you are waiting in a very long line, and you want to know what position you are in. Since you cannot leave the line without losing your position, you make the assumption that the person in front of you already knows what position they are in, and you ask them for their position. The person in front of you turns out does not know either, so they make the same assumption that the people in front of them know their positions, and they ask the same question to the person in front of them.

This continues until the first person in line is asked. Since the first person in line clearly knows that they are the first in line, they tell this to the second person. The second person now knows they are second in line, so they tell this information to the third person. The third person now knows they are third in line, so they tell this information to the fourth person. This continues all the way down the line, where each person just adds one to the previous person's answer to get their own position. Eventually, the person in front of you will know what their position is. They tell you their position, and you can add one to this number to get your own position in line. Your initial question has now been answered!

Recursion works in the same way, but this time you are a function call instead of a person waiting in line. The process of assuming the person in front of you knows their own position is analogous to the recursive leap of faith. Asking the person in front of you for the answer to your question is analogous to making a recursive call with a smaller input size. Lastly, the very first person in line is analogous to the base case, since their position number is small enough that they can come up with an answer to your initial question trivially. If you were to implement this process as a recursive function, it would look something like this:

```
1  int32_t get_line_position(int32_t person) {
2    // if first person, return one (base case)
3    if (person == 1) {
4      return 1;
5    } // if
6    // otherwise, return one plus the position of the person before you
7    return 1 + get_line_position(person - 1);
8  } // get_line_position()
```

As another example, let's apply recursion to solve a math problem. Suppose you wanted to implement a function that calculates the factorial of a number — that is, given a non-negative integer $n$, the function returns $n!$, or $n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1$. By definition, $0! = 1$. We can define a factorial in terms of itself as follows:

$$n! = \begin{cases} 1, & \text{if } n = 0 \\ n \times (n-1)!, & \text{if } n > 0 \end{cases}$$

Let's convert this definition into a function that can calculate a factorial when passed in an integer $n$. First, we need to consider the base case: if $n = 0$, we immediately know that the function should return 1. Thus, we can start off the function with the following code:

```
1  int32_t factorial(int32_t n) {
2    if (n == 0) {
3      return 1;   // base case
4    } // if
5    ...
6  } // factorial()
```

Now, what if we are not given the base case? To calculate the value of $n!$ for $n \neq 0$, we would have to calculate $n \times (n-1)!$. We know the value of $n$, but what is the value of $(n-1)!$? We do not know, but we can use recursion and pass $(n-1)$ into our own factorial function to get its value. Similar to the people waiting in line, each recursive call is going to calculate $(n-1)!$ for smaller values of $n$. Eventually, $n$ will reach 0, and we would be able to return 1 using our base case. This information will allow us to solve for 1!, 2!, 3!, …, all the way back up to $n!$. The completed code is shown below:

```
1  int32_t factorial(int32_t n) {
2    if (n == 0) {
3      return 1;   // base case
4    } // if
5    return n * factorial(n - 1);
6  } // factorial()
```

What happens when this code runs? For example, suppose we pass in the number 10 as the value of $n$. Line 5 would run, and the function would return `10 * factorial(9)`. However, this requires a recursive call, so the factorial function is run again with input $n = 9$. The recursive call with input 9 would return `9 * factorial(8)`, which triggers a recursive call with input $n = 8$. This process continues, where `factorial(8)` calls `factorial(7)`, which itself calls `factorial(6)`, which itself calls `factorial(5)`, and so on. The recursive calls stop when the input reaches 0, as this would trigger the base case and immediately return without any additional recursive calls.

There is one important thing we have to note, however. When we first called `factorial(10)`, we wanted to return the value `10 * factorial(9)`. This requires a recursive call to `factorial(9)`. However, after calling `factorial(9)`, we still have to multiply the result we get by `10`, which is stored in the variable `n`. Because we still need to reference the variable `n` after the recursive call to `factorial(9)` completes, we have to save it somewhere in memory so that it can be retrieved after the necessary recursive calls are done. This storage location is known as the **program stack**.

## 5.2 The Program Stack

Suppose we have two functions, A and B, and function A calls function B. Before function B can begin running after the function call, we have to store the state of function A so that we can return to it after function B is done. In other words, we have to remember all the local variables of function A so that, after function B runs to completion, we can return back to function A and continue where we left off.

For example, in the following code, `func_A()` calls `func_B()` on line 3. However, we have to remember that the value of `num` is `281` and that we are currently on line 3 of `func_A()` before we start running `func_B()`. That way, when `func_B()` is done, we can return back to line 3, restore the values of the local variables of `func_A()` (in this case, `num`), and continue the execution of `func_A()`.

```
1  int32_t func_A() {
2    int32_t num = 281;
3    int32_t val = func_B();
4    int32_t res = num + val;
5    return res;
6  } // func_A()
```

To remember this information, we use **stack frames**, which are stored on the program stack in memory. Recall that the stack is a portion of memory set aside for local variables and bookkeeping data (unlike the heap, which is reserved for dynamic memory). Information that is placed on a stack is accessed in last-in first-out (LIFO) order: the most recently allocated block of information is also the next block to be removed.

Each time a function call is made, the local variables that belong to the caller of the function gets stored on the program stack. Then, the arguments of the function call are pushed onto the stack. Control is then transferred from the caller of the function to the function that has been called. The new function then pops the function arguments off the stack and begins running.

When a function returns, the return value is pushed onto the program stack. The caller of the function that just returned then resumes execution, popping the return value off the program stack and restoring its local variables.

Let's analyze the stack frames that are allocated for the following snippet of code:

```
1  int32_t factorial(int32_t n) {
2    if (n == 0) {
3      return 1;
4    } // if
5    return n * factorial(n - 1);
6  } // factorial()
7
8  int main() {
9    int32_t res = factorial(3);
10   std::cout << "3! = " << res << std::endl;
11   return 0;
12 } // main()
```

What happens when this code runs? Let's walk through the code one step at a time and analyze the contents of the stack frame.

| # | *Stack Frame* | *Behavior* |
|---|---|---|
| **Program Begins Running** | | |
| **A Stack Frame is Allocated for `main()`** | | |
| (1) | `main()` | When the program begins running, the `main()` function is called. Information for the `main()` function is stored in a block of memory on the stack (called the *activation record*; this stores the local variables of `main()`, temporary objects, the return address, and other information that is needed by the function). |
| | | Line 9 creates an instance of the integer `res`, which starts off uninitialized. To assign the value of `res`, a recursive call is made with an input of `3`. |
| | | The current state of `main()` is stored in the `main()` function's stack frame before `factorial(3)` begins running. The argument value `3` is pushed onto the stack. |
| **A Stack Frame is Allocated for `factorial(3)`** | | |
| (2) | `factorial(3)` | The `factorial()` function pops the argument `3` off the stack and sets it to the value of `n`. It then begins running. |
| | | Since the value of `n` is not `0`, the `if` statement does not run. Line 5 makes a recursive call with an input size of `2` (the value of `n - 1`). |
| | | The current state of the `factorial(3)` function call is stored in a stack frame before the recursive call to `factorial(2)` begins running. The value of the argument, `2`, is pushed onto the stack. |
| **A Stack Frame is Allocated for `factorial(2)`** | | |
| (3) | `factorial(2)` | The `factorial()` function pops the argument `2` off the stack and sets it to the value of `n`. Because this is a brand new function call, the value of `n = 2` is local to the `factorial(2)` function call. The function begins running. |
| | | Since the value of `n` is not `0`, the `if` statement does not run. Line 5 makes a recursive call with an input size of `1` (the value of `n - 1`). |
| | | The current state of the `factorial(2)` function call is stored in a stack frame before the recursive call to `factorial(1)` begins running. The value of the argument, `1`, is pushed onto the stack. |
| **A Stack Frame is Allocated for `factorial(1)`** | | |

The table below continues from the previous page.

| | | |
|---|---|---|
| **A Stack Frame is Allocated for `factorial(1)`** | | |
| (4) | `factorial(1)` | The `factorial()` function pops the argument `1` off the stack and sets it to the value of `n`. It then begins running. |
| | | Since the value of `n` is not `0`, the `if` statement does not run. Line 5 makes a recursive call with an input size of `0` (the value of `n - 1`). |
| | | The current state of the `factorial(1)` function call is stored in a stack frame before the recursive call to `factorial(0)` begins running. The value of the argument, `0`, is pushed onto the stack. |
| **A Stack Frame is Allocated for `factorial(0)`** | | |
| (5) | `factorial(0)` | The `factorial()` function pops the argument `0` off the stack and sets it to the value of `n`. It then begins running. |
| | | Since the value of `n` is `0`, the base case (line 3) runs, and the return value is set to 1. |
| | | The return value of `1` is pushed onto the stack, and the `factorial(0)` function call exits. |
| **The Stack Frame for `factorial(0)` is Deallocated** | | |
| (6) | `factorial(1)` | The most recent stack frame is `factorial(1)`, so `factorial(1)` resumes execution on line 5. Local variables for this function call are restored, and the return value of `factorial(0)` is multiplied by `n` to get `1 * 1 = 1`. |
| | | The return value of `1` is pushed onto the stack, and the `factorial(1)` function call exits. |
| **The Stack Frame for `factorial(1)` is Deallocated** | | |
| (7) | `factorial(2)` | The most recent stack frame is `factorial(2)`, so `factorial(2)` resumes execution on line 5. Local variables for this function call are restored, and the return value of `factorial(1)` is multiplied by `n` to get `1 * 2 = 2`. |
| | | The return value of `2` is pushed onto the stack, and the `factorial(2)` function call exits. |
| **The Stack Frame for `factorial(2)` is Deallocated** | | |
| (8) | `factorial(3)` | The most recent stack frame is `factorial(3)`, so `factorial(3)` resumes execution on line 5. Local variables for this function call are restored, and the return value of `factorial(2)` is multiplied by `n` to get `2 * 3 = 6`. |
| | | The return value of `6` is pushed onto the stack, and the `factorial(3)` function call exits. |
| **The Stack Frame for `factorial(3)` is Deallocated** | | |
| (9) | `main()` | The most recent stack frame is `main()`, so `main()` resumes execution on line 9. Local variables are restored, and the return value of `6` is set to `res`. |
| | | Line 10 prints out `"3! = 6"`. |
| | | Line 11 returns `0`, which is pushed onto the stack. Returning `0` from `main()` indicates that the program completed successfully (the return value of `main()` is known as the *exit status* of the program). |
| **The Stack Frame for `main()` is Deallocated** | | |
| **Program Completes** | | |

A visual representation of this process is shown on the next page. The numbers on the figure (above the stack frames) correspond with the numbers in the first column of the above table.

Unfortunately, the program stack is limited, so you could exhaust all the available stack space if you recurse too deeply. If you run out of available stack space, you would end up with an error known as a **stack overflow**. One potential strategy for avoiding stack overflows is to use a process known as tail recursion, which is discussed in the following section.

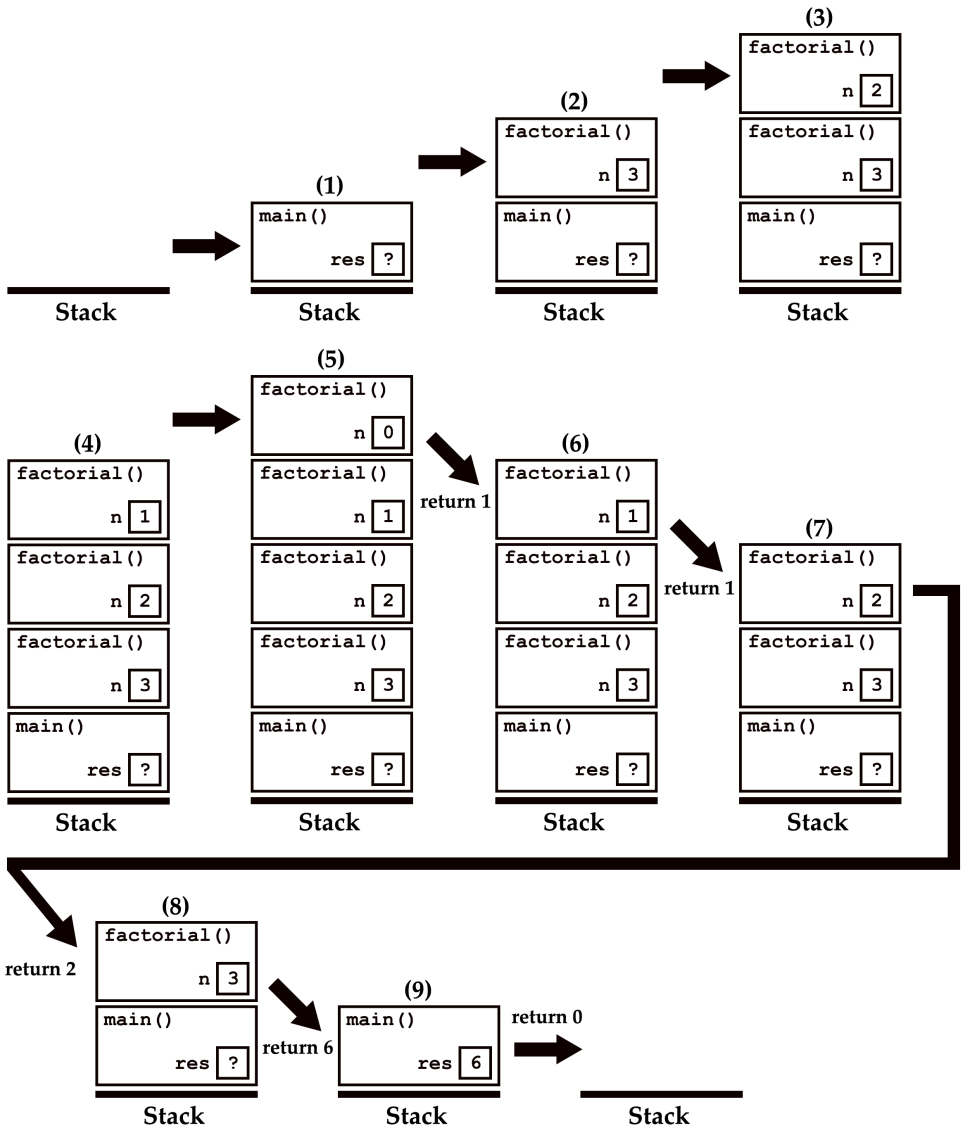
## 5.3 Tail Recursion

In the previous example, we allocated a stack frame each time we made a recursive call. This is because we needed the the values of local variables *after* all the recursive calls were complete (e.g., multiplying the recursion result with the local variable `n`). Thus, stack frames were needed to store the values of local variables so that their values could be retrieved after the recursive calls were done. However, there are certain situations where a stack frame does *not* need to be allocated with each recursive call. Such cases happen when a function is **tail recursive**.

What is the difference between a tail recursive function and a non-tail recursive function? A tail recursive function is a *linear* recursive function where *the recursive call is the **final** instruction of the function*. Because the recursive call is the final step of a function, there is no need to remember the function's local variables on a stack frame, as they will never be needed later. As a result, the compiler is able to optimize memory and reuse stack frames when a tail recursive function call is made. For instance, if the factorial function we defined earlier were tail recursive, we could just use the *same* stack frame for each factorial call (instead of having to allocate a separate stack frame for each value of `n`). Let's look at the factorial function we defined earlier:

```
1  int32_t factorial(int32_t n) {
2    if (n == 0) {
3      return 1;
4    } // if
5    return n * factorial(n - 1);
6  } // factorial()
```

Here, the result of the recursive call on line 5 must be multiplied with the value of `n` after the recursive call completes, so this function is **not** tail recursive. Since we need to remember the value of `n` before making the recursive call (so that we can use its value later), we cannot reuse the same stack frame for each recursive call.




However, it turns out that we can optimize memory by converting this non-tail recursive implementation into a tail recursive one. In our non-tail recursive implementation, we needed to remember the value of `n` before each recursive call so that we could multiply it with `factorial(n - 1)` *after* the recursive call completes. This is why each recursive call needed its own stack frame to keep track of the value of `n`. Thus, for our factorial function to be tail recursive, we would need a way to remember the value of `n` and multiply it with the result of the recursive call *before the recursive call returns*, so that we do not need to reference it later. That way, we can guarantee that the recursive call is the last instruction of the function that called it, and that nothing else needs to be done once the recursive call returns.

In order to complete the multiplication before the recursive call returns, we can add something called an *accumulator argument* as a parameter of each recursive call. This accumulator argument allows us to complete the multiplication *in the argument* of the recursive call, before the recursive call returns! The code for this is shown below:

```
1  int32_t factorial(int32_t n, int32_t res = 1) {
2    if (n == 0) {
3      return res;
4    } // if
5    return factorial(n - 1, n * res);
6  } // factorial()
```

Since we are multiplying `n` with the result of `factorial(n - 1)` in an argument of the function, we no longer have to do any additional work once the recursive call completes. Thus, our factorial function is now tail recursive, and the same stack frame can be reused for multiple recursive calls.

**Remark:** On line 1, the function definition above sets `res = 1`. This is known as a *default argument*. By doing this, we are specifying that, if the `factorial()` function is called without the `res` argument (e.g., `factorial(3)`), the value of `res` for that function call is default set to a value of 1.

Iteration and recursion are two methods that can both be used to achieve repetition in a program. Although you will not have to know how to convert between the two, any iterative function can in fact be converted to a recursive function. This is because iteration is simply a special case of tail recursion. Furthermore, any recursive function can be converted to an iterative function by simulating the program stack (e.g., you could keep track of a separate container, such as a `std::stack<>`, that behaves just like the program stack would with recursive calls).

## 5.4 Stack Frames and Space Complexity

As shown in the previous section, if a recursive function is not tail recursive, each additional recursive call allocates a stack frame on the program stack. Because stack frames take up memory, you will need to take this memory into account when calculating the auxiliary space used by a recursive function.

In general, the auxiliary space required by a non-tail recursive call is determined by its **recursion depth**, or the number of return statements that must be executed until the base case is reached. The non-tail recursive implementation of the factorial function, for example, has a recursion depth of $n$, since a call to `factorial(n)` has to execute $n$ return statements before it reaches the base case of `factorial(0)`.

When assessing the auxiliary space of a recursive function, we look at the number of return statements until the base case, not the total number of recursive function calls that are made. This is because stack frames for each recursive call may not all exist in memory at the same time. Consider the following example:

**Example 5.1** Express the auxiliary space used by `foo()` using big-O notation, in terms of the input size $n$.

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

Here, two recursive calls are made each time `foo()` is invoked. Thus, it may be tempting to say that the function uses $\Theta(2^n)$ space, since $\Theta(2^n)$ recursive calls are made. However, these recursive calls do not all exist on the stack frame at the exact same time. To visualize this, suppose you call `foo(4)`. A stack frame for `foo(4)` is allocated on the program stack:

```
foo(4) stack frame
```

The function call to `foo(4)` then makes its first recursive call to `foo(3)`, and the stack frame now looks like:

```
foo(4) stack frame
  foo(3) stack frame
```

`foo(3)` then makes its first recursive call to `foo(2)`, which makes its first recursive call to `foo(1)`. At this point, we end up at the base case, and the stack frame looks like this:

```
foo(4) stack frame
  foo(3) stack frame
    foo(2) stack frame
      foo(1) stack frame
```

At this point, `foo(1)` returns 1, and its stack frame is deallocated. `foo(2)` then calls `foo(1)` a second time, and a new stack frame is allocated. However, notice that this stack frame *does not coexist* with the stack frame allocated for the first call to `foo(1)`, since that call had to complete before the second call could be made. As a result, even though $2^4 = 16$ recursive calls to `foo()` are made when `foo(4)` is invoked, at most 4 stack frames are actually needed at any point in time.

Given an initial input size of $n$, only $n$ return statements are executed before the base case is reached, since $n$ is decremented with each recursive call. As a result, the auxiliary space required for stack frames is also $\Theta(n)$.

In general, if the recursion depth of a non-tail recursive function is $n$, the auxiliary space it uses for stack frames must also be $\Theta(n)$. Note that the auxiliary space used by the function outside of stack frames must also be considered! For instance, if a recursive algorithm requires $\Theta(n)$ stack frames, but also initializes a container that uses $\Theta(n^2)$ space, the overall auxiliary space of the algorithm is $\Theta(n + n^2) = \Theta(n^2)$.

In a tail recursive implementation, the stack frame is reused with each recursive call. Hence, the number of stack frames does not increase as the recursion depth increases, and the auxiliary space used by stack frames is $\Theta(1)$.

**Example 5.2** Express the auxiliary space used by `print_to_n()` using big-O notation, in terms of the input size $n$.

```
1  void print_to_n(int32_t n) {
2    if (n < 1) {
3      return;
4    } // if
5    std::cout << n << " ";
6    print_to_n(n - 1);
7  } // print_to_n()
```

In this example, we have a recursive call at the very end. Since the input size $n$ is decremented by 1 with each recursive call from $n$ to 1, the recursion depth is $n$ (note that the `return` statement for a `void` function is implicit if it is not specified). However, the recursive call to `print_to_n()` is the very last thing that is done! As a result, the function is tail recursive, and the compiler reuses the same stack frame for each recursive call. Since the remaining memory usage of this function does not depend on input size, and the function is tail recursive, the auxiliary space used by the function is $\Theta(1)$.

**Example 5.3** Express the auxiliary space used by `bar()` using big-O notation, in terms of the input size $n$.

```
1  int32_t bar(int32_t n) {
2    if (n < 1) {
3      return;
4    } // if
5    return n + bar(n / 2);
6  } // bar()
```

Since the input size of the recursive call is halved each time, the recursion depth is $\log(n)$. This is because $\log(n)$ recursive calls are needed for the input size to go from the initial value $n$ to the base case of 1. Notice here that the recursive call is *not* the last thing that is done, since its value must still be added to $n$ after the function returns. Thus the function is not tail recursive, and stack frames cannot be reused. The number of stack frames needed is equal to the recursion depth, so the auxiliary space required by the stack frames is $\Theta(\log(n))$.

---

To summarize: when dealing with space complexity, it is important to analyze sources of memory usage for both the stack and the heap. A non-tail recursive function that does not explicitly allocate memory may still allocate additional stack frames as the input size grows, as long as the recursion depth also increases with the size of the input.

## 5.5 Identifying Recurrence Relations

In the previous chapter, we discussed how the time complexity of an algorithm can be determined by counting steps. However, what if recursion is involved? If a function calls itself, how do we measure its time complexity?

In the following sections, we will discuss methods that can be used to find the time complexities of recursive algorithms. Previously, we defined the runtime of an algorithm as a function $T(n)$ of its input size $n$. We will still do this when dealing with recursive functions. However, because a recursive algorithm calls itself with a smaller input size, we will write $T(n)$ as a **recurrence relation**, or an equation that defines the runtime of a problem in terms of the runtime of a recursive call on smaller input. As an example, let's revisit the non-tail recursive factorial function defined at the beginning of section 5.3. In this function, we return 1 if $n == 0$ and make a recursive call to `factorial(n - 1)` otherwise. This can be converted into the following recurrence relation:

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

Here, $T(n)$ represents the runtime of `factorial(n)`, $T(n-1)$ represents the runtime of the recursive call to `factorial(n - 1)`, and the additional $\Theta(1)$ term represents the constant work we do outside the recursive call. Since the recurrence relation will be used to calculate time complexities, having an exact number for this constant term does not matter.

On the next few pages, we will look at several recursive functions and define them as recurrence relations, where runtime is expressed as a function of input size. This procedure is similar to the one used for an iterative function: operations that follow one another have their complexities added, while operations that are nested within loops have their complexities multiplied. However, when a recursive call is reached, we add a term that defines $T(n)$ in terms of the input size of that recursive call.

**Example 5.4** Express the runtime of the function `foo()` as a recurrence relation. You may assume that the function `bar()` runs in $\Theta(\log(n))$ time and that $n \geq 1$.

```
1  void foo(int32_t n) {
2    if (n == 1) {
3      return;
4    } // if
5    foo(n - 1);
6    int k = n * n;
7    for (int i = 0; i < k; ++i) {
8      for (int j = 0; j < n; ++j) {
9        bar(n);
10     } // for j
11   } // for i
12   for (int i = 0; i < n; ++i) {
13     foo(n / 2);
14   } // for i
15   bar(k);
16 } // foo()
```

To approach this problem, we will use the same approach as in the previous chapter. Walk through each line and determine its time complexity. However, if you reach a recursive call, express its contribution toward the runtime recursively in terms of the input size of that recursive call.

- **Lines 2-3:** this is a constant $\Theta(1)$ operation, but it only runs if $n = 1$.
- **Line 5:** this is a recursive call with an input size of $n-1$, which we assign a recurrence term of $T(n-1)$.
- **Line 6:** this is a constant $\Theta(1)$ calculation.
- **Lines 7-9:** `bar()` runs in $\Theta(\log(n))$ time, the inner `for` loop runs $n$ times, and the outer `for` loop runs $n^2$ times (since `k = n * n`). The total time complexity of this entire nested loop is thus $\Theta(\log(n) \times n \times n^2) = \Theta(n^3 \log(n))$.
- **Lines 12-13:** line 13 makes a recursive call with an input size of $n/2$, so this line gets assigned a recurrence term of $T(n/2)$. However, since this recursive call is made $n$ times in the `for` loop defined on line 12, the overall contribution of the entire loop is $nT(n/2)$.
- **Line 15:** since `bar()` has a time complexity of $\Theta(\log(n))$ and is passed in an input size of $n^2$, the time complexity of this line is $\Theta(\log(n^2))$, which is equivalent to $\Theta(2\log(n))$ using logarithm identities.

There are two different cases that can happen. If the input size $n$ is 1, only line 3 runs, so $T(1) = \Theta(1)$ (i.e., constant time). However, if $n > 1$, we can express the runtime of `foo()` using the following recurrence relation:

$$T(n) = T(n-1) + n^3 \log(n) + nT(n/2) + 2\log(n) + \Theta(1)$$

The full recurrence relation is therefore as follows:

$$T(n) = \begin{cases} \Theta(1), & \text{if } n = 1 \\ T(n-1) + n^3 \log(n) + nT(n/2) + 2\log(n) + \Theta(1), & \text{if } n > 1 \end{cases}$$

## 5.6 The Iterative Substitution Method

After we convert a recursive algorithm into a recurrence relation, we can use that recurrence relation to determine its time complexity. In this class, we will be looking at two methods that can be used to accomplish this: the *iterative substitution method* and the *Master Theorem*. In this section, we will discuss the iterative substitution method (also known as the iteration method). To start, let's look at the recursive function defined below, which takes in a positive integer $n$ as its input size:

```
1  int32_t foo(int32_t n) {
2    if (n == 1) {
3      return 1;
4    } // if
5    return foo(n - 1) + foo(n - 1) + 1;
6  } // foo()
```

The runtime of this function is constant when $n = 1$. Otherwise, it makes two recursive calls with input size $n-1$ and performs constant time math operations. We can express the runtime of this function using the following recurrence relation:

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

Just by looking at this recurrence relation alone, there is not much we can immediately conclude about its time complexity. If $T(n)$ had been equal to something like $5n^2 + 6n + 7$, we'd be able to easily tell that the runtime complexity is $\Theta(n^2)$. However, in this example, we have to determine the runtime complexity of $2T(n-1) + \Theta(1)$. How do we go about doing that?

We do not know anything yet about the time complexity of the $T(n-1)$ recursive call. However, we do know the time complexity of the base case: $T(1)$ performs a constant time operation. If we are somehow able to convert $2T(n-1) + \Theta(1)$ into a form that only contains the base case $T(1)$, we could substitute $T(1)$ with a constant value of 1 to get rid of all the recursive terms in our $T(n)$ equation.[1] Such a form is known as a *closed form solution*, which makes it easier to deduce a recurrence relation's complexity class.

This is the idea behind the **iterative substitution method**, which finds an explicit formula for a recursively defined sequence. After you have a recurrence relation, the steps of the iterative substitution method are as follows:

1. Write out the recursive terms ($T(n-1)$, $T(n-2)$, etc.), as their own recurrence relations and substitute their equations into the original $T(n)$ formula at each step.
2. Look for a pattern that describes $T(n)$ at the $k^{\text{th}}$ step (for any arbitrary $k$), and express it using a summation formula.
3. Solve for $k$ such that the base case is the only recursive term that is present on the right-hand side of the equation for $T(n)$. Determine the closed form solution by replacing instances of the base case with its value (e.g., replacing $T(1)$ with 1 if the base case is $T(1) = \Theta(1)$).

Let's walk through some examples:

**Example 5.5** Express the time complexity of the `foo()` function using big-O notation, in terms of the input size $n$. Assume $n \geq 1$.

```
1  int32_t foo(int32_t n) {
2    if (n == 1) {
3      return 1;
4    } // if
5    return foo(n - 1) + foo(n - 1) + 1;
6  } // foo()
```

You may find the following identity helpful with your calculations:

$$\sum_{i=m}^{n} ar^i = \frac{a(r^m - r^{n+1})}{1 - r}$$

[1] Since $T(1)$ runs in constant time, we can actually substitute it with any constant, and our big-O result would not change. However, replacing any constant work with 1 in the recurrence relation makes the math a lot easier to work with.

To solve this problem, first convert the function into a recurrence relation. This recurrence relation is:

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

From now on, we will substitute $\Theta(1)$ with the constant 1. This simplifies our math without changing the result of our analysis.

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

Now, let's follow the procedure specified above.

*Step 1: Write out the recursive terms ($T(n-1)$, $T(n-2)$, etc.) as their own recurrence relations and substitute their equations into the original $T(n)$ formula at each step.*

Here, the base case is $T(1) = 1$, so we want to convert $2T(n-1)$ into a form that only contains the recursive term $T(1)$. Assuming that $n$ is not already 1, we know from our recurrence relation that

$$T(n) \;=\; 2T(n-1) + 1$$

Using the definition of our recurrence relation, we can replace $n$ with $n-1$ to get an expression for $T(n-1)$.

$$T(n-1) \;=\; 2T((n-1)-1) + 1 \;=\; 2T(n-2) + 1$$

Plugging $2T(n-2)+1$ for $T(n-1)$ into the first equation gives us

$$T(n) \;=\; 2T(n-1) + 1 \;=\; 2 \times \underbrace{[2T(n-2)+1]}_{T(n-1)} + 1 \;=\; 2 \times 2 \times T(n-2) + 2 + 1$$

Now, we have written $T(n)$ in terms of $T(n-2)$ instead of $T(n-1)$. This process is then repeated to decrement the input size of the recursive call until the base case of $T(1)$ is reached. Let's do one more step. Using the definition of our recurrence relation, we can write $T(n-2)$ as

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

Plugging this into our current recurrence relation, we get

$$T(n) \;=\; 2 \times 2 \times T(n-2) + 2 + 1 \;=\; 2 \times 2 \times \underbrace{[2T(n-3)+1]}_{T(n-2)} + 2 + 1 \;=\; 2 \times 2 \times 2 \times T(n-3) + 4 + 2 + 1$$

Repeating this substitution process for the first four steps will give you the following results:

| ***Step #*** | ***Recurrence Equation*** |
|---|---|
| 1 | $T(n) = 2 \times T(n-1) + 1$ |
| 2 | Rewrite $T(n-1)$ as $2 \times T(n-2) + 1$ |
| | Substitute $2 \times T(n-2) + 1$ in for $T(n-1)$ in the previous recurrence equation |
| | $T(n) = 2 \times T(n-1) + 1$<br>$\rightarrow T(n) = 2 \times 2 \times T(n-2) + 2 + 1$ |
| 3 | Rewrite $T(n-2)$ as $2 \times T(n-3) + 1$ |
| | Substitute $2 \times T(n-3) + 1$ in for $T(n-2)$ in the previous recurrence equation |
| | $T(n) = 2 \times 2 \times T(n-2) + 2 + 1$<br>$\rightarrow T(n) = 2 \times 2 \times 2 \times T(n-3) + 4 + 2 + 1$ |
| 4 | Rewrite $T(n-3)$ as $2 \times T(n-4) + 1$ |
| | Substitute $2 \times T(n-4) + 1$ in for $T(n-3)$ in the previous recurrence equation |
| | $T(n) = 2 \times 2 \times 2 \times T(n-3) + 4 + 2 + 1$<br>$\rightarrow T(n) = 2 \times 2 \times 2 \times 2 \times T(n-4) + 8 + 4 + 2 + 1$ |

After substituting for a few iterations, we can move on to the next step.

*Step 2: Look for a pattern that describes $T(n)$ at the $k^{th}$ step, and express it using a summation formula.*

With the first 4 steps, we have enough information to describe a pattern for $T(n)$.

- At the first step, $T(n) = 2 \times T(n-1) + 1$
- At the second step, $T(n) = 2 \times 2 \times T(n-2) + 2 + 1$
- At the third step, $T(n) = 2 \times 2 \times 2 \times T(n-3) + 4 + 2 + 1$
- At the fourth step, $T(n) = 2 \times 2 \times 2 \times 2 \times T(n-4) + 8 + 4 + 2 + 1$

We can generalize this and derive the following formula for $T(n)$ at the $k^{\text{th}}$ step:

$$T(n) = 2^k T(n-k) + \sum_{i=0}^{k-1} 2^i$$

This equation states that the recurrence relation $T(n) = 2T(n-1)+1$ is identical to $T(n) = 2^k T(n-k) + \sum_{i=0}^{k-1} 2^i$ at the $k^{\text{th}}$ step of substitution.

*Step 3: Solve for k such that the base case is the only recursive term that is present on the right-hand side of the equation for $T(n)$. Determine the closed form solution by replacing instances of the base case with the value of the base case.*

We determined that $T(n) = 2T(n-1)+1$ can be rewritten as $T(n) = 2^k T(n-k) + \sum_{i=0}^{k-1} 2^i$ at the $k^{\text{th}}$ step of substitution. Since we want the base case to be the only recursive term present on the right-hand side of the equation, we want to select a $k$ such that $T(n-k)$ equals the base case of $T(1)$. This happens when $n-k=1$, or when $k=n-1$. Substituting $n-1$ for $k$ in our equation gives us an expression for $T(n)$ where $T(1)$ is the only recursive term on the right-hand side.

$$\begin{aligned}
T(n) &= 2^k T(n-k) + \sum_{i=0}^{k-1} 2^i \quad \text{(substitute } n-1 \text{ for } k\text{)} \\
&= 2^{n-1} T(n-(n-1)) + \sum_{i=0}^{(n-1)-1} 2^i \\
&= 2^{n-1} T(1) + \sum_{i=0}^{n-2} 2^i
\end{aligned}$$

We know from the base case that $T(1) = 1$. Therefore, we can replace $T(1)$ in the equation with 1, as follows:

$$\begin{aligned}
T(n) &= 2^{n-1} T(1) + \sum_{i=0}^{n-2} 2^i \\
&= 2^{n-1} \times 1 + \sum_{i=0}^{n-2} 2^i \\
&= 2^{n-1} + \sum_{i=0}^{n-2} 2^i
\end{aligned}$$

We now have a closed form solution to the original recurrence relation without any recursive terms on the right-hand side! Using the provided summation identity,

$$\sum_{i=m}^{n} ar^i = \frac{a(r^m - r^{n+1})}{1-r}$$

where $a=1$, $r=2$, $m=0$, and $n=n-2$, we can show that

$$\sum_{i=0}^{n-2} 2^i = \frac{1(2^0 - 2^{n-1})}{1-2} = 2^{n-1} - 1$$

Plugging this back in our equation for $T(n)$ gives us

$$\begin{aligned}
T(n) &= 2^{n-1} + \sum_{i=0}^{n-2} 2^i \\
&= 2^{n-1} + 2^{n-1} - 1 \\
&= 2 \times 2^{n-1} - 1 \\
&= 2^n - 1
\end{aligned}$$

Since $T(n) = 2^n - 1$, the time complexity of the function `foo()` is therefore $\Theta(2^n)$.

**Example 5.6** Express the time complexity of the `foo()` function using big-O notation, in terms of input size $n$. Assume $n \geq 0$.

```
1  void foo(int32_t n) {
2    if (n == 0) {
3      return;
4    } // if
5    for (int32_t i = 1; i < n; i *= 2) {
6      std::cout << "281" << std::endl;
7    } // for i
8    foo(n - 1);
9  } // foo()
```

Just by looking at this code, you may recognize that the time complexity is $\Theta(n\log(n))$, since the `for` loop runs $\log(n)$ times, and a total of $n$ recursive calls are made. However, let's do some substitution to prove that this is indeed the case.

First, let's express this function as a recurrence relation. If $n = 0$, the base case executes, which takes constant time. Otherwise, the `for` loop and recursive call run. Since the value of `i` is doubled with each iteration of the loop, the loop executes the print statement $\log(n)$ times. The recursive call can be denoted as $T(n-1)$, since its input size is $n-1$. Thus, the recurrence relation for this function is:

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

Now, we will use the same procedure as above to compute the complexity of this recurrence relation. *Note: Technically, it is more accurate to describe the runtime of $T(n)$ for $n > 0$ as $T(n-1) + \log(n) + 1$, since there is still some constant time work that is done (e.g., multiplication, assignment, etc.). However, since the constant term is clearly a lower-order term that is dominated by the* $\log(n)$ *work, it is okay to ignore this constant term altogether since it will not make a difference in the final complexity class.*

*Step 1: Write out the recursive terms ($T(n-1)$, $T(n-2)$, etc.) as their own recurrence relations and substitute their equations into the original $T(n)$ formula at each step.*

| ***Step #*** | ***Recurrence Equation*** |
|---|---|
| 1 | $T(n) = T(n-1) + \log(n)$ |
| 2 | Rewrite $T(n-1)$ as $T(n-2) + \log(n-1)$ |
| | Substitute $T(n-2) + \log(n-1)$ in for $T(n-1)$ in the previous recurrence equation |
| | $T(n) = T(n-1) + \log(n)$<br>$\rightarrow T(n) = T(n-2) + \log(n-1) + \log(n)$ |
| 3 | Rewrite $T(n-2)$ as $T(n-3) + \log(n-2)$ |
| | Substitute $T(n-3) + \log(n-2)$ in for $T(n-2)$ in the previous recurrence equation |
| | $T(n) = T(n-2) + \log(n-1) + \log(n)$<br>$\rightarrow T(n) = T(n-3) + \log(n-2) + \log(n-1) + \log(n)$ |

**Note: If you want to write a recurrence relation for $T(n-1)$, you must replace ALL instances of $n$ with $(n-1)$, even the terms outside the recursive call!** A common mistake is only substituting the $n$ in the recursive term.

**Correct**: $T(n-1) = T(n-2) + \log(n-1)$
**Incorrect**: $T(n-1) = T(n-2) + \log(n)$

Once you are able to recognize a pattern, you can continue to the next step.

*Step 2: Look for a pattern that describes $T(n)$ at the $k^{th}$ step, and express it using a summation formula.*

With the first few steps, we have enough information to describe a pattern for $T(n)$.

- At the first step, $T(n) = T(n-1) + \log(n)$
- At the second step, $T(n) = T(n-2) + \log(n-1) + \log(n)$
- At the third step, $T(n) = T(n-3) + \log(n-2) + \log(n-1) + \log(n)$
- At the fourth step, $T(n) = T(n-4) + \log(n-3) + \log(n-2) + \log(n-1) + \log(n)$

We can generalize this and derive the following formula for $T(n)$ at the $k^{\text{th}}$ step:

$$T(n) = T(n-k) + \sum_{i=0}^{k-1} \log(n-i)$$

*Step 3: Solve for k such that the base case is the only recursive term that is present on the right-hand side of the equation for $T(n)$. Determine the closed form solution by replacing instances of the base case with the value of the base case.*

We determined that $T(n) = T(n-1) + \log(n)$ can be rewritten as $T(n) = T(n-k) + \sum_{i=0}^{k-1} \log(n-i)$ at the $k^{\text{th}}$ step of substitution. Since we want the base case to be the only recursive term present on the right-hand side of the equation, we want to select a $k$ such that $T(n-k)$ equals the base case of $T(0)$. This happens when $n-k = 0$, or when $k = n$. Substituting $n$ for $k$ in our equation gives us an expression for $T(n)$ where $T(0)$ is the only recursive term on the right-hand side.

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

Since we know that $T(0) = 1$, we can plug this into the above equation.

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

Using log rules, we know that the sum of log terms is equal to the log of the product of the terms; that is, $\log(a) + \log(b) + \ldots + \log(z) = \log(a \times b \times \ldots \times z)$.

$$\begin{aligned} T(n) &= 1 + \log(n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1) \\ &= 1 + \log(n!) \end{aligned}$$

After dropping lower order terms, we can conclude that $T(n) = \Theta(\log(n!))$. We proved in chapter 4 that $\log(n!) = \Theta(n\log(n))$, so the time complexity of the function `foo()` is indeed $\Theta(n\log(n))$.

**Example 5.7** Express the time complexity of the `foo()` function using big-O notation, in terms of input size $n$. Assume $n \geq 0$.

```
1   int64_t foo(int64_t n) {
2     int64_t s = 1;
3     if (n == 0) {
4       return s;
5     } // if
6     for (int64_t i = 1; i <= n; ++i) {
7       s += foo(n - 1);
8     } // for i
9     return s;
10  } // foo()
```

You may find the following identity helpful with your calculations:

$$\sum_{i=1}^{n} \frac{n!}{(n-i)!} = \lfloor n!e \rfloor - 1, \text{ where } e \approx 2.71828\ldots$$

First, convert this function to a recurrence relation. If $n = 0$, we have a base case that runs in constant time. If $n > 0$, we run the `for` loop, which runs a recursive call to `foo(n - 1)` and performs some constant time work $n$ times. The recurrence relation is therefore

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

*Step 1: Write out the recursive terms ($T(n-1)$, $T(n-2)$, etc.) as their own recurrence relations and substitute their equations into the original $T(n)$ formula at each step.*

| ***Step #*** | ***Recurrence Equation*** |
|---|---|
| 1 | $T(n) = nT(n-1) + n$ |
| 2 | Rewrite $T(n-1)$ as $(n-1)T(n-2) + (n-1)$ |
| | Substitute $(n-1)T(n-2) + (n-1)$ in for $T(n-1)$ in the previous recurrence equation |
| | $T(n) = nT(n-1) + n$<br>$\rightarrow T(n) = n[(n-1)T(n-2) + (n-1)] + n$<br>$\rightarrow T(n) = n(n-1)T(n-2) + n(n-1) + n$ |
| 3 | Rewrite $T(n-2)$ as $(n-2)T(n-3) + (n-2)$ |
| | Substitute $(n-2)T(n-3) + (n-2)$ in for $T(n-2)$ in the previous recurrence equation |
| | $T(n) = n(n-1)T(n-2) + n(n-1) + n$<br>$\rightarrow T(n) = n(n-1)[(n-2)T(n-3) + (n-2)] + n(n-1) + n$<br>$\rightarrow T(n) = n(n-1)(n-2)T(n-3) + n(n-1)(n-2) + n(n-1) + n$ |

*Step 2: Look for a pattern that describes $T(n)$ at the $k^{th}$ step, and express it using a summation formula.*

With the first few steps, we have enough information to describe a pattern for $T(n)$.

- At the first step, $T(n) = nT(n-1) + n$
- At the second step, $T(n) = n(n-1)T(n-2) + n(n-1) + n$
- At the third step, $T(n) = n(n-1)(n-2)T(n-3) + n(n-1)(n-2) + n(n-1) + n$

We can generalize this and derive the following formula for $T(n)$ at the $k^{\text{th}}$ step:

$$T(n) = \frac{n!}{(n-k)!}T(n-k) + \sum_{i=1}^{k} \frac{n!}{(n-i)!}$$

*Step 3: Solve for k such that the base case is the only recursive term that is present on the right-hand side of the equation for $T(n)$. Determine the closed form solution by replacing instances of the base case with the value of the base case.*

Since we want the base case to be the only recursive term present on the right-hand side of the equation, we want to select a $k$ such that $T(n-k)$ equals the base case of $T(0)$. This happens when $n-k=0$, or $k=n$. Substituting $n$ for $k$ gives us an expression for $T(n)$ where $T(0)$ is the only recursive term on the right-hand side. We can then plug in $T(0)=1$ into the equation and use the provided identity to simplify the equation.

$$T(n) = \frac{n!}{(n-n)!}T(n-n)+\sum_{i=1}^{n}\frac{n!}{(n-i)!} = n!T(0)+\sum_{i=1}^{n}\frac{n!}{(n-i)!} = n!+\sum_{i=1}^{n}\frac{n!}{(n-i)!} = \lfloor n!(e+1)\rfloor-1$$

Since $T(n)=\lfloor n!(e+1)\rfloor-1\approx 3.71828n!-1$, the time complexity of the function `foo()` is $\Theta(n!)$.

**Example 5.8** Express the time complexity of the `foo()` function using big-O notation, in terms of input size $n$. Assume $n\geq 1$.

```
1  void foo(int32_t n) {
2    if (n == 1) {
3      return;
4    } // if
5    foo(n / 2);
6    for (int32_t i = 0; i < n; ++i) {
7      std::cout << "281" << std::endl;
8    } // for i
9    foo(n / 2);
10 } // foo()
```

First, convert this function to a recurrence relation. If $n=1$, we have a base case that runs in constant time. If $n>1$, we make two recursive calls and print "281" $n$ times. The recurrence relation is therefore

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

*Step 1: Write out the recursive terms as recurrence relations and substitute their equations into the original $T(n)$ formula at each step.*

| ***Step #*** | ***Recurrence Equation*** |
|---|---|
| 1 | $T(n)=2T(n/2)+n$ |
| 2 | Rewrite $T(n/2)$ as $2T(n/4)+n/2$ |
| | Substitute $2T(n/4)+n/2$ in for $T(n/2)$ in the previous recurrence equation |
| | $T(n)=2T(n/2)+n$<br>$\rightarrow T(n)=2(2T(n/4)+n/2)+n$<br>$\rightarrow T(n)=4T(n/4)+2n$ |
| 3 | Rewrite $T(n/4)$ as $2T(n/8)+n/4$ |
| | Substitute $2T(n/8)+n/4$ in for $T(n/4)$ in the previous recurrence equation |
| | $T(n)=4T(n/4)+2n$<br>$\rightarrow T(n)=4(2T(n/8)+n/4)+2n$<br>$\rightarrow T(n)=8T(n/8)+3n$ |

*Step 2: Look for a pattern that describes $T(n)$ at the $k^{th}$ step, and express it using a summation formula.*

With the first few steps, we have enough information to describe a pattern for $T(n)$.

- At the first step, $T(n)=2T(n/2)+n$
- At the second step, $T(n)=4T(n/4)+2n$
- At the third step, $T(n)=8T(n/8)+3n$

We can generalize this and derive the following formula for $T(n)$ at the $k^{\text{th}}$ step:

$$T(n) = 2^kT\left(\frac{n}{2^k}\right)+kn$$

*Step 3: Solve for k such that the base case is the only recursive term that is present on the right-hand side of the equation for $T(n)$. Determine the closed form solution by replacing instances of the base case with the value of the base case.*

Since we want the base case to be the only recursive term present on the right-hand side of the equation, we want to select a $k$ such that $T(n/2^k)$ equals the base case of $T(1)$. This happens when $2^k=n$, or $k=\log_2(n)$. Substituting $\log_2(n)$ for $k$ in our equation gives us an expression for $T(n)$ where $T(1)$ is the only recursive term on the right-hand side. We can then plug in $T(1)=1$ into the equation.

$$T(n) = 2^{\log_2(n)}T\left(\frac{n}{2^{\log_2(n)}}\right)+n\log_2(n) = nT(1)+n\log_2(n) = n+n\log_2(n)$$

Since $n\log_2(n)$ is the higher-order term, the time complexity of the function is $\Theta(n\log(n))$.

**Remark:** By convention, we only define $T$ with integer arguments, so $T(n/2)$ usually represents $T(\lfloor n/2\rfloor)$ if $n$ does not divide evenly. However, if we only care about finding a big-O bound for a recurrence relation, this rounding should not make a difference. As a result, the floor operation is usually implicitly assumed if division occurs in the recursive term.

## 5.7 The Master Theorem

### ※ 5.7.1 *Applying the Master Theorem*

The **Master Theorem** provides a template that can be used to compute the time complexity of any recurrence relation that follows a given format. The theorem is defined below:

**The Master Theorem:**

If the recurrence relation of an algorithm is of the form:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

where $a \geq 1$, $b > 1$, and $f(n)$ is a positive function that is $\Theta(n^c)$, then the following is true:

- If $a > b^c$, the complexity of $T(n)$ is $\Theta\left(n^{\log_b(a)}\right)$.
- If $a = b^c$, the complexity of $T(n)$ is $\Theta(n^c \log(n))$.
- If $a < b^c$, the complexity of $T(n)$ is $\Theta(n^c)$.

More formally,

$$T(n) = \begin{cases} \Theta\left(n^{\log_b(a)}\right), & \text{if } a > b^c \\ \Theta(n^c \log(n)), & \text{if } a = b^c \\ \Theta(n^c), & \text{if } a < b^c \end{cases}$$

Notes:

- There must exist a base case that is solvable in constant time.
- The value $a$ represents the number of times a recursive call is made, the value $b$ represents the factor that the input is divided by with each recursive call, and the value $c$ represents the highest power of the polynomial term $f(n)$.
- The values of $a$ and $b$ cannot depend on $n$.
- The Master Theorem can only be used if **all** these conditions are met.

The steps for using the Master Theorem are as follows:

1. Determine the values of $a$, $b$, and $c$.
2. Make sure that the Master Theorem can be used on the recurrence relation.
   - The coefficient of the recursive call, $a$, must be at least one.
   - The argument of the recursive call must be divided by some number, $b$, that is larger than one.
   - The function $f(n)$ must be an asymptotically positive function with complexity $\Theta(n^c)$.
   - There exists a base case that can be solved in constant time.
3. Compare the values of $a$ and $b^c$ to determine which case of the Master Theorem should be used.

**Example 5.9** The runtime of an algorithm can be expressed using the following recurrence relation. Can the Master Theorem be used, and if so, what is the time complexity of this algorithm?

$$T(n) = \begin{cases} 1, & \text{if } n = 1 \\ 3T(n/2) + 5n + 13, & \text{if } n > 1 \end{cases}$$

**1. Determine the values of $a$, $b$, and $c$.**

The value of $a$ is 3 because that is the coefficient of the recursive term. The value of $b$ is 2 because that is the denominator of the recursive term. The value of $c$ is 1 because $f(n) = 5n + 13 = \Theta(n^1)$.

**2. Make sure that the Master Theorem can be used on the recurrence relation.**

$a = 3 \geq 1$, $b = 2 > 1$, and $5n + 13$ is part of a polynomial complexity class. The structure is correct, and there exists a base case that can be solved in constant time. Thus, the Master Theorem can be used on the recurrence relation.

**3. Compare the values of $a$ and $b^c$ to determine which case of the Master Theorem should be used.**

The value of $a$ is 3. The value of $b^c$ is $2^1 = 2$. Because $a > b^c$, the time complexity of the algorithm is $\Theta\left(n^{\log_b(a)}\right) = \Theta\left(n^{\log_2(3)}\right) \approx \Theta\left(n^{1.58}\right)$.

**Example 5.10** The runtime of an algorithm can be expressed using the following recurrence relation. Can the Master Theorem be used, and if so, what is the time complexity of this algorithm?

$$T(n) = \begin{cases} 1, & \text{if } n = 1 \\ 4T(n/2) + 5n + 13n^2, & \text{if } n > 1 \end{cases}$$

**1. Determine the values of $a$, $b$, and $c$.**

The value of $a$ is 4 because that is the coefficient of the recursive term. The value of $b$ is 2 because that is the denominator of the recursive term. The value of $c$ is 2 because $f(n) = 5n + 13n^2 = \Theta(n^2)$.

**2. Make sure that the Master Theorem can be used on the recurrence relation.**

$a = 4 \geq 1$, $b = 2 > 1$, and $5n + 13n^2$ is part of a polynomial complexity class. The structure is correct, and there exists a base case that can be solved in constant time. Thus, the Master Theorem can be used on the recurrence relation.

**3. Compare the values of $a$ and $b^c$ to determine which case of the Master Theorem should be used.**

The value of $a$ is 4. The value of $b^c$ is $2^2 = 4$. Because $a = b^c$, the time complexity of the algorithm is $\Theta(n^c \log(n)) = \Theta\left(n^2 \log(n)\right)$.

**Example 5.11** The runtime of an algorithm can be expressed using the following recurrence relation. Can the Master Theorem be used, and if so, what is the time complexity of this algorithm?

$$T(n) = \begin{cases} 1, & \text{if } n = 1 \\ 6T(n/2) + 5n + 2^n, & \text{if } n > 1 \end{cases}$$

Here, $f(n) = 5n + 2^n = \Theta(2^n)$. This is not a polynomial complexity class, so the Master Theorem cannot be used to compute the complexity of this recurrence relation.

**Example 5.12** The runtime of an algorithm can be expressed using the following recurrence relation. Can the Master Theorem be used, and if so, what is the time complexity of this algorithm?

$$T(n) = \begin{cases} 1, & \text{if } n = 1 \\ 6T(n/2) + 3T(n/5) + 4n + 3, & \text{if } n > 1 \end{cases}$$

No, this recurrence relation is not in the correct format. The Master Theorem cannot be used on a recurrence relation with two different recursive calls with differing values of $b$.

**Example 5.13** The runtime of an algorithm can be expressed using the following recurrence relation. Can the Master Theorem be used, and if so, what is the time complexity of this algorithm?

$$T(n) = \begin{cases} 1, & \text{if } n = 1 \\ T(n/4) + n\sqrt{n}, & \text{if } n > 1 \end{cases}$$

**1. Determine the values of $a$, $b$, and $c$.**

The value of $a$ is 1 because that is the coefficient of the recursive term. The value of $b$ is 4 because that is the denominator of the recursive term. The value of $c$ is 3/2 because $f(n) = n\sqrt{n} = n(n^{1/2}) = \Theta(n^{3/2})$.

**2. Make sure that the Master Theorem can be used on the recurrence relation.**

$a = 1 \geq 1$, $b = 4 > 1$, and $n^{3/2}$ is part of a polynomial complexity class. The format is correct, and there exists a base case that can be solved in constant time. Thus, the Master Theorem can be used on the recurrence relation.

**3. Compare the values of $a$ and $b^c$ to determine which case of the Master Theorem should be used.**

The value of $a$ is 1. The value of $b^c$ is $4^{3/2} = 8$. Because $a < b^c$, the time complexity of the algorithm is $\Theta(n^c) = \Theta(n^{3/2})$.

**Example 5.14** The runtime of an algorithm can be expressed using the following recurrence relation. Can the Master Theorem be used, and if so, what is the time complexity of this algorithm?

$$T(n) = \begin{cases} 1, & \text{if } n = 1 \\ 5T(n/8) + 15, & \text{if } n > 1 \end{cases}$$

**1. Determine the values of $a$, $b$, and $c$.**

The value of $a$ is 5 because that is the coefficient of the recursive term. The value of $b$ is 8 because that is the denominator of the recursive term. The value of $c$ is 0 because $f(n) = 15 = 15n^0 = \Theta(n^0) = \Theta(1)$.

**2. Make sure that the Master Theorem can be used on the recurrence relation.**

$a = 5 \geq 1$, $b = 8 > 1$, and $n^0$ is part of a polynomial complexity class. The structure is correct, and there exists a base case that can be solved in constant time. Thus, the Master Theorem can be used on the recurrence relation.

**3. Compare the values of $a$ and $b^c$ to determine which case of the Master Theorem should be used.**

The value of $a$ is 5. The value of $b^c$ is $8^0 = 1$. Because $a > b^c$, the time complexity of the algorithm is $\Theta\left(n^{\log_b(a)}\right) = \Theta\left(n^{\log_8(5)}\right) \approx \Theta\left(n^{0.774}\right)$.

**Example 5.15** The runtime of an algorithm can be expressed using the following recurrence relation. Can the Master Theorem be used, and if so, what is the time complexity of this algorithm?

$$T(n) = \begin{cases} 1, & \text{if } n = 1 \\ 7T(10n/11) + 9n, & \text{if } n > 1 \end{cases}$$

**1. Determine the values of $a$, $b$, and $c$.**

The value of $a$ is 7 because that is the coefficient of the recursive term. The value of $b$ is 11/10 because that is the denominator of the recursive term (since $10n/11$ can be written as $n/(11/10)$). The value of $c$ is 1 because $f(n) = 9n = \Theta(n^1)$.

**2. Make sure that the Master Theorem can be used on the recurrence relation.**

$a = 7 \geq 1$, $b = 1.1 > 1$, and $n^1$ is part of a polynomial complexity class. The structure is correct, and there exists a base case that can be solved in constant time. Thus, the Master Theorem can be used on the recurrence relation.

**3. Compare the values of $a$ and $b^c$ to determine which case of the Master Theorem should be used.**

The value of $a$ is 7. The value of $b^c$ is $1.1^1 = 1.1$. Because $a > b^c$, the time complexity of the algorithm is $\Theta\left(n^{\log_b(a)}\right) = \Theta\left(n^{\log_{1.1}(7)}\right) \approx \Theta\left(n^{20.417}\right)$.

**Example 5.16** The *Karatsuba algorithm* is a fast multiplication algorithm that can be used to multiply two $n$-digit numbers faster than the classical multiplication algorithm often taught in grade school. This algorithm is recursive and can be expressed using the following recurrence relation. What is the time complexity of the Karatsuba algorithm?

$$T(n) = \begin{cases} 1, & \text{if } n = 1 \\ 3T(n/2) + \Theta(n), & \text{if } n > 1 \end{cases}$$

**1. Determine the values of $a$, $b$, and $c$.**

The value of $a$ is 3 because that is the coefficient of the recursive term. The value of $b$ is 2 because that is the denominator of the recursive term. The value of $c$ is 1 because $f(n) = \Theta(n^1)$.

**2. Make sure that the Master Theorem can be used on the recurrence relation.**

$a = 3 \geq 1$, $b = 2 > 1$, and $n^1$ is part of a polynomial complexity class. The structure is correct, and there exists a base case that can be solved in constant time. Thus, the Master Theorem can be used on the recurrence relation.

**3. Compare the values of $a$ and $b^c$ to determine which case of the Master Theorem should be used.**

The value of $a$ is 3. The value of $b^c$ is $2^1 = 2$. Because $a > b^c$, the time complexity of the algorithm is $\Theta\left(n^{\log_b(a)}\right) = \Theta\left(n^{\log_2(3)}\right) \approx \Theta\left(n^{1.585}\right)$.

**Example 5.17** In linear algebra, the *Strassen algorithm* is an algorithm that can be used to multiply two matrices of size $n \times n$ faster than the standard matrix multiplication algorithm. This algorithm is recursive and can be expressed using the following recurrence relation. What is the time complexity of the Strassen algorithm?

$$T(n) = \begin{cases} 1, & \text{if } n = 1 \\ 7T(n/2) + \Theta(n^2), & \text{if } n > 1 \end{cases}$$

**1. Determine the values of $a$, $b$, and $c$.**

The value of $a$ is 7 because that is the coefficient of the recursive term. The value of $b$ is 2 because that is the denominator of the recursive term. The value of $c$ is 2 because $f(n) = \Theta(n^2)$.

**2. Make sure that the Master Theorem can be used on the recurrence relation.**

$a = 7 \geq 1$, $b = 2 > 1$, and $n^2$ is part of a polynomial complexity class. The structure is correct, and there exists a base case that can be solved in constant time. Thus, the Master Theorem can be used on the recurrence relation.

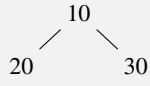
**3. Compare the values of $a$ and $b^c$ to determine which case of the Master Theorem should be used.**

The value of $a$ is 7. The value of $b^c$ is $2^2 = 4$. Because $a > b^c$, the time complexity of the algorithm is $\Theta\left(n^{\log_b(a)}\right) = \Theta\left(n^{\log_2(7)}\right) \approx \Theta\left(n^{2.807}\right)$.

### ※ 5.7.2 *Deriving the Master Theorem* (✻)

To provide intuition into why the Master Theorem works, we can visualize the total work done by a recurrence relation using a recursion tree. A **recursion tree** is a tree can be used to illustrate the work done by a recursive algorithm, where each branch represents a recursive call, and each node represents the total amount of additional work that is done at that recursive call.

**Remark:** Recursion trees gives us a way to visualize the total work that is done by a recursive algorthm. Each node of a recurrence tree represents a single recursive call, and it stores the work that is done at that recursive call. For example, if a recursive algorithm does 10 units of work and then makes two recursive calls, one that does 20 units of work, and one that does 30 units of work, we can visualize the total work done using the following recurrence tree:

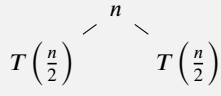



The sum of all the nodes in the recursion tree represents the total work that is completed by the recursive algorithm. In the example above, a total of $10 + 20 + 30 = 60$ units of work are done after the recursive algorithm runs to completion.

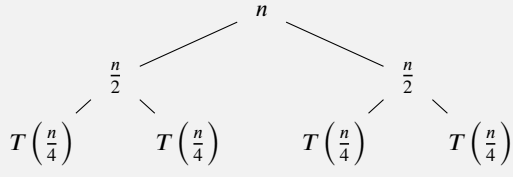
Given a recurrence relation, we can build a recursion tree by repeating the following steps until we reach the base case:

1. At each level, each node is assigned the total amount of work that is done outside the recursive call(s).
2. Each recursive call creates a branch to the next level of the tree.

For example, consider the recurrence relation $T(n) = 2T(n/2) + n$. This recurrence relation calls itself twice with half the input size and does an additional $n$ work. With a recursion tree approach, we can visualize the first recursive call as follows:




How much work do these recursive calls to $T(n/2)$ do? We know from our recurrence relation that $T(n/2) = 2T(n/4) + n/2$, where a recursive call with an input size of $n/2$ does $n/2$ work and makes two recursive calls, each with input size $n/4$. We can use this to extend our tree down another level.

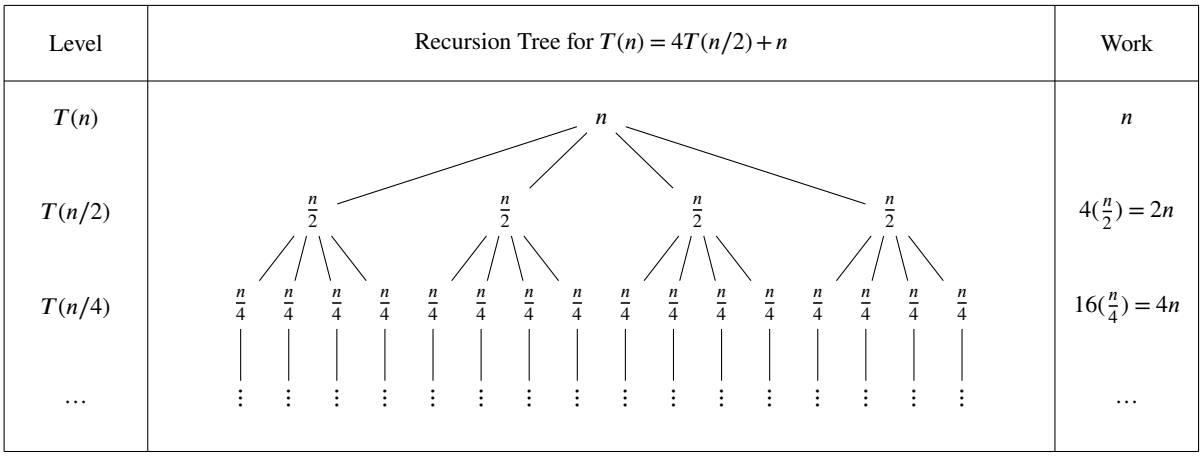



By repeating this process to the base case, we can construct the full recursion tree.

Let's look at the following three recurrence relations, each of which falls into a Master Theorem category:

1. $T(n) = 4T(n/2) + n$ (condition where $a > b^c$)
2. $T(n) = 2T(n/2) + n^2$ (condition where $a < b^c$)
3. $T(n) = 4T(n/2) + n^2$ (condition where $a = b^c$)

In the first recurrence relation, the algorithm calls itself four times with an input size of $n/2$ and does an additional $n$ work. We can represent the total work done by this recurrence relation using the following tree:




Here, "level" represents the recursion depth, and "work" represents the total work that is done at each level.

You might notice here that the amount of work increases at each recursion level. We call this tree "bottom-heavy" or "leaf-heavy," since the amount of work spent on the recursive subproblems (the $aT(n/b)$ term of the recurrence relation) dwarfs the amount of work spent on splitting and recombining the results of the recursive subproblems (the additional work represented by $f(n)$ in the recurrence relation). Because of this, asymptotically all of the work of this recurrence relation is done on the lowest level of the tree (i.e., the leaves)! From the perspective of the Master Theorem, the value of $a$ is so large that the amount of work attributed to the sheer number of recursive calls we are making dominates over the additional $f(n)$ work we have to do at each recursive call. This dominance can be shown by the fact that the total work at each level increases even as the work attributed to $f(n)$ decreases as the input size is cut in half at each level (i.e., $n < \boldsymbol{4}(\frac{n}{2}) < \boldsymbol{16}(\frac{n}{4}) < \ldots$).

How much work is done at the last level? We know that the last level represents calls to the base case, so every node at the last level must do a constant amount of work. Thus, the total amount of work that is done at the lowest level can be determined using the number of nodes at that level. Since we are making four recursive calls at each level, the last level of the tree has $4^i$ nodes, where $i$ is the total number of levels in the tree. We also know that the number of levels in the tree is equal to the number of recursive calls we need to make to go from $n$ to our base case: since the input size is cut in half at each level (from $n$ to $n/2$ to $n/4\ldots$), a total of $\log_2(n)$ levels are needed before the base case is reached. Thus, the number of nodes at the last level is

$$4^i = 4^{\log_2(n)}$$

since $i$, or the number of levels in the tree, is approximately equal to $\log_2(n)$.

This can actually be generalized to any value of $a$ and $b$. The number of nodes at the last level of a recursion tree for a recurrence relation of the form $aT(n/b) + f(n)$ can be approximated as

$$a^{\log_b(n)}$$

We can use log rules to manipulate this expression into something we are more familiar with. Using the following identity:

$$a = n^{\log_n(a)}$$

we can substitute $a$ with $n^{\log_n(a)}$ in the original equation:

$$a^{\log_b(n)} = \left(n^{\log_n(a)}\right)^{\log_b(n)} = n^{\log_n(a)\log_b(n)}$$

Using the change of base formula, we can simplify the exponent term:

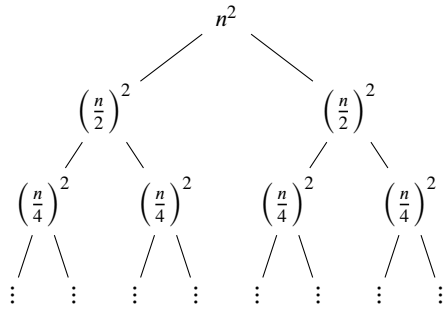
$$\log_n(a)\log_b(n) = \frac{\log(a)}{\log(n)} \times \frac{\log(n)}{\log(b)} = \frac{\log(a)}{\log(b)} = \log_b(a)$$

Therefore, the number of nodes at the last level of the tree can be expressed as

$$n^{\log_n(a)\log_b(n)} = n^{\log_b(a)}$$

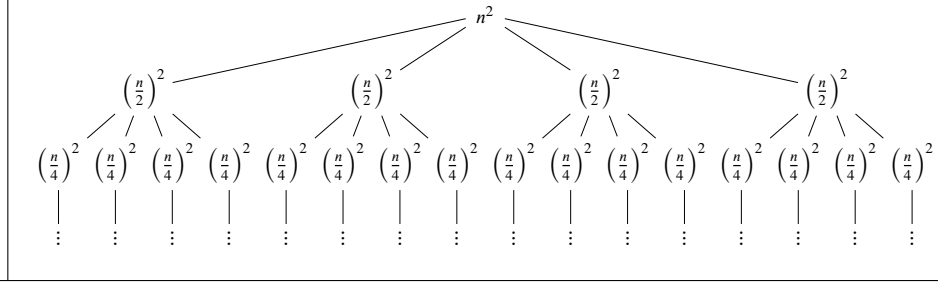
Since each node at the bottom level of the tree is a base case that does a constant amount of work, the total amount of work attributable to the bottom level is $n^{\log_b(a)} \times c$ for some constant $c$, or $\Theta\left(n^{\log_b(a)}\right)$. Because the work in a bottom-heavy tree is asymptotically performed at the lowest level, the overall complexity of the recurrence relation must also be $\Theta\left(n^{\log_b(a)}\right)$. This matches the result of the Master Theorem.

Now, let's look at the case where $a < b^c$ by drawing out the recursion tree of $T(n) = 2T(n/2) + n^2$. In this recurrence relation, the algorithm calls itself twice with an input size of $n/2$ and does an additional $n^2$ work.

| Level | Recursion Tree for $T(n) = 2T(n/2) + n^2$ | Work |
|---|---|---|
| $T(n)$ |  | $n^2$ |
| $T(n/2)$ | | $2(\frac{n^2}{4}) = \frac{n^2}{2}$ |
| $T(n/4)$ | | $4(\frac{n^2}{16}) = \frac{n^2}{4}$ |
| … | | … |

Here, we can see that the amount of work decreases at each recursion level. We call this tree "top-heavy" or "root-heavy," since the amount of work spent on splitting and recombining the results of the recursive subproblems (the additional work represented by $f(n)$ in the recurrence relation) dwarfs the amount of work spent on the recursive subproblems (the $aT(n/b)$ term of the recurrence relation). Because of this, asymptotically all of the work that is done at the top (or root) of the tree! From the Master Theorem perspective, the contribution of $f(n)$ is so large that it dominates over the contribution of the recursive calls. Because $f(n)$ dominates the runtime of the recurrence relation, the time complexity of the entire recurrence relation must also be $\Theta(f(n))$, or $\Theta(n^c)$. This matches the result of the Master Theorem.

There exists a point where the work spent on the recursive subproblems balances out the additional work spent on splitting and recombining the results of the recursive subproblems. This happens in the case where $a = b^c$. Let's look at the example of $T(n) = 4T(n/2) + n^2$:

| Level | Recursion Tree for $T(n) = 4T(n/2) + n^2$ | Work |
|---|---|---|
| $T(n)$ |  | $n^2$ |
| $T(n/2)$ | | $4(\frac{n^2}{4}) = n^2$ |
| $T(n/4)$ | | $16(\frac{n^2}{16}) = n^2$ |
| … | | … |

Here, the amount of work done at every level of the tree is always $\Theta(n^c)$. The total number of levels in the tree is $\log_2(n)$, since $n$ can be halved a total of $\log_2(n)$ times before the base case is reached. Because each level does $\Theta(n^c)$ work, the overall time complexity of the entire recurrence relation is $\Theta(n^c \log_2(n))$, or just $\Theta(n^c \log(n))$. This is also the result of the Master Theorem when $a = b^c$.

To summarize, the total amount of work that a recurrence relation of the form $aT(n/b) + f(n)$ does — where $f(n)$ is a polynomial that is $\Theta(n^c)$ — depends on the values of $a$ and $b^c$.

- If $a > b^c$, the work attributed to the number of recursive calls dominates over the additional $f(n)$ work at each recursive call. As a result, the work is concentrated at the bottom of the recursion tree — since there are $n^{\log_b(a)}$ nodes at this bottom level, and each node does a constant amount of work, the complexity of a recurrence where $a > b^c$ is $\Theta\left(n^{\log_b(a)}\right)$.
- If $a < b^c$, the additional $f(n)$ work done with each recursive call dominates over the work attributed to the number of recursive calls, and the work is concentrated at the top of the recursion tree — since the top level does $f(n)$ work, the overall complexity of a recurrence where $a < b^c$ is $\Theta(f(n)) = \Theta(n^c)$.
- If $a = b^c$, the work attributed to the number of recursive calls is comparable to the additional $f(n)$ work done with each recursive call. Thus, the amount of work done at every level of the recursion tree is $\Theta(n^c)$, and since there are a total of $\log(n)$ levels in the tree, the complexity of the overall recurrence is $\Theta(n^c \log(n))$.

### ※ 5.7.3 *The Extended Master Theorem for Polylogarithmic Functions* (✻)

There is actually an extension of the Master Theorem that can be used if $f(n)$ is a polylogarithmic function, but you will *not* need to know this for the class. Given a recurrence relation of the form $T(n) = aT(n/b) + f(n)$, where $f(n)$ is of the form $\Theta(n^c \log^k(n))$, the following are true:

- If $a > b^c$, then $T(n)$ is $\Theta\left(n^{\log_b(a)}\right)$.
- If $a = b^c$, then
  - If $k > -1$, then $T(n)$ is $\Theta\left(n^{\log_b(a)} \log^{k+1}(n)\right)$.
  - If $k = -1$, then $T(n)$ is $\Theta\left(n^{\log_b(a)} \log(\log(n))\right)$.
  - If $k < -1$, then $T(n)$ is $\Theta\left(n^{\log_b(a)}\right)$.
- If $a < b^c$, then
  - If $k \geq 0$, then $T(n)$ is $\Theta(n^c \log^k(n))$.
  - If $k < 0$, then $T(n)$ is $\Theta(n^c)$.

## 5.8 Complexities of Common Recurrence Relations

In this section, we will use the tools covered in the previous sections to determine the complexity classes of common recurrence relations. These recurrence relations, along with their time complexities, are shown in the table below.

| Recurrence Type | Time Complexity |
|---|---|
| $T(n) = T(n/2) + 1$ | $\Theta(\log(n))$ |
| $T(n) = T(n-1) + 1$ | $\Theta(n)$ |
| $T(n) = 2T(n/2) + 1$ | $\Theta(n)$ |
| $T(n) = T(n-1) + n + 1$ | $\Theta(n^2)$ |
| $T(n) = 2T(n/2) + n + 1$ | $\Theta(n\log(n))$ |

**Example 5.18** Show that a recurrence relation of the form $T(n) = T(n/2) + 1$ has a time complexity of $\Theta(\log(n))$.

This recurrence relation matches the form $T(n) = aT(n/b) + f(n)$, where $a \geq 1$, $b > 1$, and $f(n)$ is part of a polynomial complexity class. Thus, the Master Theorem can be used on this recurrence relation. The value of $a$ is 1, the value of $b$ is 2, and the value of $c$ is 0. Since $1 = 2^0$, the complexity of this recurrence relation is $\Theta(n^c \log(n)) = \Theta(n^0 \log(n)) = \Theta(\log(n))$.

**Example 5.19** Show that a recurrence relation of the form $T(n) = T(n-1) + 1$ has a time complexity of $\Theta(n)$.

This recurrence relation is not in a form that allows us to use the Master Theorem. As a result, we will try using iterative substitution.

*Step 1: Write out the recursive terms as recurrence relations and substitute their equations into the original $T(n)$ formula at each step.*

| ***Step #*** | ***Recurrence Equation*** |
|---|---|
| 1 | $T(n) = T(n-1) + 1$ |
| 2 | Rewrite $T(n-1)$ as $T(n-2) + 1$ |
| | Substitute $T(n-2) + 1$ in for $T(n-1)$ in the previous recurrence equation |
| | $T(n) = T(n-1) + 1$<br>$\rightarrow T(n) = T(n-2) + 1 + 1$<br>$\rightarrow T(n) = T(n-2) + 2c$ |
| 3 | Rewrite $T(n-2)$ as $T(n-3) + 1$ |
| | Substitute $T(n-3) + 1$ in for $T(n-2)$ in the previous recurrence equation |
| | $T(n) = T(n-2) + 2$<br>$\rightarrow T(n) = T(n-3) + 1 + 2$<br>$\rightarrow T(n) = T(n-3) + 3$ |

*Step 2: Look for a pattern that describes $T(n)$ at the $k^{th}$ step, and express it using a summation formula.*

With the first few steps, we have enough information to describe a pattern for $T(n)$.

- At the first step, $T(n) = T(n-1) + 1$
- At the second step, $T(n) = T(n-2) + 2$
- At the third step, $T(n) = T(n-3) + 3$

We can generalize this and derive the following formula for $T(n)$ at the $k^{\text{th}}$ step:

$$T(n) = T(n-k) + k$$

*Step 3: Solve for k such that the base case is the only recursive term that is present on the right-hand side of the equation for $T(n)$. Determine the closed form solution by replacing instances of the base case with the value of the base case.*

Since we want the base case to be the only recursive term present on the right-hand side of the equation, we want to select a $k$ such that $T(n-k)$ equals the base case. We are not told what value of $n$ the base case happens at, so let's use an arbitrary constant $c$ to represent the base case. $T(n-k)$ equals the base case $T(c)$ when $n-k=c$, or $k=n-c$. Substituting $n-c$ for $k$ in our equation gives us an expression for $T(n)$ where $T(c)$ is the only recursive term on the right-hand side. We can then plug in $T(c) = 1$ into the equation (since the base case does constant work).

$$T(n) \;=\; T(n-k)+k \;=\; T(n-(n-c))+(n-c) \;=\; T(c)+(n-c) \;=\; 1+n-c$$

After dropping lower order terms (1 and $c$), we get that the complexity of the recurrence relation is $\Theta(n)$.

**Example 5.20** Show that a recurrence relation of the form $T(n) = 2T(n/2) + 1$ has a time complexity of $\Theta(n)$.

This recurrence relation matches the form $T(n) = aT(n/b) + f(n)$, where $a \geq 1$, $b > 1$, and $f(n)$ is part of a polynomial complexity class. Thus, the Master Theorem can be used on this recurrence relation. The value of $a$ is 2, the value of $b$ is 2, and the value of $c$ is 0. Since $2 > 2^0$, the complexity of this recurrence relation is $\Theta\left(n^{\log_b(a)}\right) = \Theta\left(n^{\log_2(2)}\right) = \Theta(n)$.

**Example 5.21** Show that a recurrence relation of the form $T(n) = T(n-1) + n + 1$ has a time complexity of $\Theta(n^2)$.

This recurrence relation is not in a form that allows us to use the Master Theorem. As a result, we will try using iterative substitution.

*Step 1: Write out the recursive terms as recurrence relations and substitute their equations into the original $T(n)$ formula at each step.*

| ***Step #*** | ***Recurrence Equation*** |
|---|---|
| 1 | $T(n) = T(n-1) + n + 1$ |
| 2 | Rewrite $T(n-1)$ as $T(n-2) + (n-1) + 1$ |
| | Substitute $T(n-2) + (n-1) + 1$ in for $T(n-1)$ in the previous recurrence equation |
| | $T(n) = T(n-1) + n + 1$<br>$\rightarrow T(n) = T(n-2) + (n-1) + 1 + n + 1$<br>$\rightarrow T(n) = T(n-2) + (n-1) + n + 2$ |
| 3 | Rewrite $T(n-2)$ as $T(n-3) + (n-2) + 1$ |
| | Substitute $T(n-3) + (n-2) + 1$ in for $T(n-2)$ in the previous recurrence equation |
| | $T(n) = T(n-2) + (n-1) + n + 2$<br>$\rightarrow T(n) = T(n-3) + (n-2) + 1 + (n-1) + n + 2$<br>$\rightarrow T(n) = T(n-3) + (n-2) + (n-1) + n + 3$ |

*Step 2: Look for a pattern that describes $T(n)$ at the $k^{th}$ step, and express it using a summation formula.*

With the first few steps, we have enough information to describe a pattern for $T(n)$.

- At the first step, $T(n) = T(n-1) + n + 1$
- At the second step, $T(n) = T(n-2) + (n-1) + n + 2$
- At the third step, $T(n) = T(n-3) + (n-2) + (n-1) + n + 3$

We can generalize this and derive the following formula for $T(n)$ at the $k^{\text{th}}$ step:

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

*Step 3: Solve for k such that the base case is the only recursive term that is present on the right-hand side of the equation for $T(n)$. Determine the closed form solution by replacing instances of the base case with the value of the base case.*

Since we want the base case to be the only recursive term present on the right-hand side of the equation, we want to select a $k$ such that $T(n-k)$ equals the base case. We are not told what value of $n$ the base case happens at, so let's use an arbitrary constant $c$ to represent the base case. $T(n-k)$ equals the base case $T(c)$ when $n-k=c$, or $k=n-c$. Substituting $n-c$ for $k$ in our equation gives us an expression for $T(n)$ where $T(c)$ is the only recursive term on the right-hand side. We can then plug in $T(c) = 1$ into the equation.

$$\begin{aligned} T(n) &= T(n-(n-c)) + \sum_{i=0}^{(n-c)-1} (n-i) + (n-c) \\ &= T(c) + \sum_{i=0}^{(n-c)-1} (n-i) + (n-c) \\ &= 1 + (n + (n-1) + (n-2) + \ldots + (c+1)) + (n-c) \end{aligned}$$

The sum of all terms in an arithmetic sequence can be calculated using the equation $n\left(\frac{a_1+a_n}{2}\right)$, where $n$ is the number of terms, $a_1$ is the value of the first term, and $a_n$ is the value of the $n^{\text{th}}$ term. Using this equation, we calculate the sum of all terms from $c+1$ to $n$ as $(n-c)\left(\frac{c+1+n}{2}\right) = \frac{n^2}{2} + \frac{n}{2} - \frac{c^2}{2} - \frac{c}{2}$. This allows us to rewrite $T(n)$ as

$$T(n) \;=\; 1 + (n + (n-1) + (n-2) + \ldots + (c+1)) + (n-c) \;=\; 1 + \left(\frac{n^2}{2} + \frac{n}{2} - \frac{c^2}{2} - \frac{c}{2}\right) + (n-c)$$

After dropping constants and lower order terms, we can see that $T(n)$ is $\Theta(n^2)$.

**Example 5.22** Show that a recurrence relation of the form $T(n) = 2T(n/2) + n + 1$ has a time complexity of $\Theta(n\log(n))$.

This recurrence relation matches the form $T(n) = aT(n/b) + f(n)$, where $a \geq 1$, $b > 1$, and $f(n)$ is part of a polynomial complexity class. Thus, the Master Theorem can be used on this recurrence relation. The value of $a$ is 2, the value of $b$ is 2, and the value of $c$ is 1. Since $2 = 2^1$, the complexity of this recurrence relation is $\Theta(n^c \log(n)) = \Theta(n^1 \log(n)) = \Theta(n\log(n))$.

## 5.9 Analysis of Recursive Algorithms

At this point, we have covered several different methods that can be used to measure the time complexity of an algorithm. In the last chapter, we focused on methods that can be used to determine the time complexity of an iterative algorithm. In this chapter, we expanded upon these methods by introducing techniques that can be used when recursion is involved. Time complexity analysis will serve as an important foundation for this class, as it allows us to compare different algorithms when trying to solve a problem. In this section, we will go through the algorithm analysis process using the following algorithm design question.

**Example 5.23** You are given a two-dimensional matrix with $m$ rows and $n$ columns, where $m \approx n$, represented using an array of arrays. This matrix is sorted along the rows and columns: for every element in the 2-D matrix, the value at index $[i][j]$ is less than the values at $[i+1][j]$ and $[i][j+1]$. You are given a target value that you want to find in the 2-D matrix, and you have to return whether the value exists. What are the time complexities of the following three recursive algorithms, and which one is asymptotically best for solving this problem?

You may find the following identity helpful with your calculations:

$$\sum_{i=0}^{n} i2^i = 2^{n+1}n - 2^{n+1} + 2$$

### ※ Algorithm #1: Quad Partition

The quad partition algorithm splits the 2-D matrix into four quadrants, compares the middle element (`arr[m/2][n/2]`) with the target value, and eliminates the quadrant that cannot contain the target value (since the matrix is sorted). The algorithm is then recursively called on the other three quadrants. An example is shown below.

You are asked to determine if 13 can be found in this 2-D matrix. The quad partition algorithm would

1. Compare the target value (13) with the middle element (9).
2. Eliminate the quadrant that cannot contain 13. In this case, because 13 is bigger than 9, we know that 13 must either be to the right of 9 or below 9. It is impossible for 13 to be both to the left and above 9, so the top left quadrant is eliminated.
3. Make three recursive calls, one for each of the three quadrants that were not eliminated.

| | | | | |
|---|---|---|---|---|
| 1 | 4 | 7 | 11 | 15 |
| 2 | 5 | 8 | 12 | 19 |
| 3 | 6 | 9 | 16 | 22 |
| 10 | 13 | 14 | 17 | 24 |
| 18 | 21 | 23 | 26 | 30 |

Attempt to find 13

| | | | | |
|---|---|---|---|---|
| 1 | 4 | 7 | 11 | 15 |
| 2 | 5 | 8 | 12 | 19 |
| 3 | 6 | 9 | 16 | 22 |
| 10 | 13 | 14 | 17 | 24 |
| 18 | 21 | 23 | 26 | 30 |

Compare 13 with middle element

| | | | | |
|---|---|---|---|---|
| 1 | 4 | 7 | 11 | 15 |
| 2 | 5 | 8 | 12 | 19 |
| 3 | 6 | 9 | 16 | 22 |
| 10 | 13 | 14 | 17 | 24 |
| 18 | 21 | 23 | 26 | 30 |

13 must be in one of these quadrants

### ※ Algorithm #2: Binary Partition with Linear Search

The binary partition algorithm with linear search also splits the 2-D matrix into four quadrants. However, instead of just looking at the middle value, the algorithm looks through the entire middle column using a *linear* search to determine where the target value should be if it were in that column. If the value is not in the middle column, the algorithm eliminates both the quadrant to the upper left and the quadrant to the lower right of where the target value should be. Two recursive calls are then made, one for each of the two quadrants that were not eliminated.

| | | | | |
|---|---|---|---|---|
| 1 | 4 | 7 | 11 | 15 |
| 2 | 5 | 8 | 12 | 19 |
| 3 | 6 | 9 | 16 | 22 |
| 10 | 13 | 14 | 17 | 24 |
| 18 | 21 | 23 | 26 | 30 |

Attempt to find 13

| | | | | |
|---|---|---|---|---|
| 1 | 4 | 7 | 11 | 15 |
| 2 | 5 | 8 | 12 | 19 |
| 3 | 6 | 9 | 16 | 22 |
| 10 | 13 | 14 | 17 | 24 |
| 18 | 21 | 23 | 26 | 30 |

Find 13's position in middle column

| | | | | |
|---|---|---|---|---|
| 1 | 4 | 7 | 11 | 15 |
| 2 | 5 | 8 | 12 | 19 |
| 3 | 6 | 9 | 16 | 22 |
| 10 | 13 | 14 | 17 | 24 |
| 18 | 21 | 23 | 26 | 30 |

13 must be in one of these quadrants

### ※ Algorithm #3: Binary Partition with Binary Search

The binary partition algorithm with binary search does the same thing as the algorithm with linear search. The only difference is that *binary* search is used to find where the target element should be in the middle column. For instance, using the example above, instead of checking 7, 8, 9, … , the search checks 9 first, compares 9 with 13, and then removes half of the search space (because the matrix is sorted, there is no need to check elements above 9). Since binary search cuts the search space in half at every iteration, it takes $\Theta(\log(n))$ time.

Which algorithm has the best time complexity? Since recursion is involved with all three algorithms, we will first convert them into recurrence relations. We will ask four questions to help us develop a recurrence relation:

1. Compared to the initial input size $n$, what input size is passed into the recursive call?
2. How many recursive calls are made?
3. What is the time complexity of the other operations outside the recursive calls?
4. What is the base case?

**Note:** In this problem, the input size $n$ represents the dimension size of the matrix, and *not* the total number of values in the matrix! This is because our algorithms recursively break up the problems by dimension size rather than the total number of elements.

### ※ Analysis of Algorithm #1: Quad Partition

Let's develop a recurrence relation for the quad partition algorithm:

**1. Compared to the initial input size $n$, what input size is passed into the recursive call?**

After splitting the input into four quadrants, the dimension of each quadrant is approximately $n/2$. Since we recurse into these quadrants, the input size of each recursive call is also $n/2$.

**2. How many recursive calls are made?**

After splitting the matrix into four quadrants, we eliminate one based on the value of the middle element. Since we make a recursive call for each of the remaining quadrants, 3 recursive calls are made.

**3. What is the time complexity of the other operations outside the recursive calls?**

Before making the recursive calls, the algorithm compares the middle value with the target value and decides which quadrant to eliminate. This is done in constant time.

**4. What is the base case?**

The base case happens when the dimensions of the input array is 1x1 (when $n$ is 1). If there is only one element in the array, it takes constant time to determine whether this element is the target value.

Using this information, we can derive the following recurrence relation:

$$T(n) = \begin{cases} 1, & \text{if } n = 1 \\ 3T(n/2) + 1, & \text{if } n > 1 \end{cases}$$

The Master Theorem can be used on this recurrence. Using $a = 3$, $b = 2$, and $c = 0$, we get

$$a > b^c \rightarrow \Theta\left(n^{\log_b(a)}\right) = \Theta\left(n^{\log_2(3)}\right) \approx \Theta(n^{1.585})$$

Thus, the time complexity of the quad partition algorithm is $\Theta\left(n^{\log_2(3)}\right)$, or approximately $\Theta(n^{1.585})$.

### ※ Analysis of Algorithm #2: Binary Partition with Linear Search

We will repeat this process for the binary partition with linear search:

**1. Compared to the initial input size $n$, what input size is passed into the recursive call?**

After splitting the input into four quadrants, the dimension of each quadrant is approximately $n/2$. Since we recurse into these quadrants, the input size of each recursive call is also $n/2$.

**2. How many recursive calls are made?**

After splitting the matrix into four quadrants, we eliminate two quadrants instead of one. Thus, two recursive calls are made.

**3. What is the time complexity of the other operations outside the recursive calls?**

Although we do eliminate one recursive call, we had to do some extra work in order to do so. This work is in the form of a linear search down the middle column. This takes linear time ($\Theta(n)$ since $m \approx n$).

**4. What is the base case?**

The base case happens when the dimensions of the input array is 1x1 (when $n$ is 1). If there is only one element in the array, it takes constant time to determine whether this element is the target value.

Using this information, we can derive the following recurrence relation:

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

The Master Theorem can be used on this recurrence. Using $a = 2$, $b = 2$, and $c = 1$, we get

$$a = b^c \rightarrow \Theta(n^c \log(n)) = \Theta(n \log(n))$$

Thus, the time complexity of the binary partition with linear search algorithm is $\Theta(n \log(n))$.

※ **Analysis of Algorithm #3:** Binary Partition with Binary Search

We will repeat this process for the binary partition with binary search:

**1. Compared to the initial input size *n*, what input size is passed into the recursive call?**

After splitting the input into four quadrants, the dimension of each quadrant is approximately $n/2$. Since we recurse into these quadrants, the input size of each recursive call is also $n/2$.

**2. How many recursive calls are made?**

Like with the binary partition with linear search algorithm, only 2 recursive calls are made.

**3. What is the time complexity of the other operations outside the recursive calls?**

Although we eliminate one recursive call, we had to do some extra work in order to do so. This work is in the form of a *binary* search down the middle column. This takes logarithmic time, or $\log_2(n)$. (If you want more a more in-depth coverage of binary search, see chapter 15.)

**4. What is the base case?**

The base case happens when the dimensions of the input array is 1x1 (when $n$ is 1). If there is only one element in the array, it takes constant time to determine whether this element is the target value.

Using this information, we can derive the following recurrence relation:

$$T(n) = \begin{cases} 1, & \text{if } n = 1 \\ 2T(n/2) + \log_2(n), & \text{if } n > 1 \end{cases}$$

The Master Theorem cannot be used on this recurrence relation since $f(n) = \log_2(n)$, which is not of the form $\Theta(n^c)$. As a result, we will need to use iterative substitution. (Even though the base-2 of the logarithm does not affect the ultimate complexity class, we will explicitly keep it for the math involved during the substitution process.)

**Remark:** This is a case where the extended Master Theorem (section 5.7.3) comes in handy.

- If $a > b^c$, then $T(n)$ is $\Theta\left(n^{\log_b(a)}\right)$.
- If $a = b^c$, then
  - If $k > -1$, then $T(n)$ is $\Theta\left(n^{\log_b(a)} \log^{k+1}(n)\right)$.
  - If $k = -1$, then $T(n)$ is $\Theta\left(n^{\log_b(a)} \log(\log(n))\right)$.
  - If $k < -1$, then $T(n)$ is $\Theta\left(n^{\log_b(a)}\right)$.
- If $a < b^c$, then
  - If $k \geq 0$, then $T(n)$ is $\Theta(n^c \log^k(n))$.
  - If $k < 0$, then $T(n)$ is $\Theta(n^c)$.

For the recurrence relation $T(n) = 2T(n/2) + \log_2(n)$, we can see that $f(n) = \log_2(n) = \Theta(n^0 \log^1(n))$. Since $a > b^c$, we can conclude that $T(n) = \Theta\left(n^{\log_b(a)}\right) = \Theta\left(n^{\log_2(2)}\right) = \Theta(n)$. However, since the extended Master Theorem is not required course material, we will still go through the process of iterative substitution to arrive at the same result.

*Step 1: Write out the recursive terms as recurrence relations and substitute their equations into the original $T(n)$ formula at each step.*

| ***Step #*** | ***Recurrence Equation*** |
|---|---|
| 1 | $T(n) = 2T(n/2) + \log_2(n)$ |
| 2 | Rewrite $T(n/2)$ as $2T(n/4) + \log_2(n/2)$ |
| | Substitute $2T(n/4) + \log_2(n/2)$ in for $T(n/2)$ in the previous recurrence equation |
| | $T(n) = 2T(n/2) + \log_2(n)$<br>$\rightarrow 2(2T(n/4) + \log_2(n/2)) + \log_2(n)$<br>$\rightarrow T(n) = 4T(n/4) + 2\log_2(n/2) + \log_2(n)$ |
| 3 | Rewrite $T(n/4)$ as $2T(n/8) + \log_2(n/4)$ |
| | Substitute $2T(n/8) + \log_2(n/4)$ in for $T(n/4)$ in the previous recurrence equation |
| | $T(n) = 4T(n/4) + 2\log_2(n/2) + \log_2(n)$<br>$\rightarrow T(n) = 4(2T(n/8) + \log_2(n/4)) + 2\log_2(n/2) + \log_2(n)$<br>$\rightarrow T(n) = 8T(n/8) + 4\log_2(n/4) + 2\log_2(n/2) + \log_2(n)$ |

*Step 2: Look for a pattern that describes $T(n)$ at the $k^{th}$ step, and express it using a summation formula.*

With the first few steps, we have enough information to describe a pattern for $T(n)$.

- At the first step, $T(n) = 2T(n/2) + \log_2(n)$
- At the second step, $T(n) = T(n) = 4T(n/4) + 2\log_2(n/2) + \log_2(n)$
- At the third step, $8T(n/8) + 4\log_2(n/4) + 2\log_2(n/2) + \log_2(n)$

We can generalize this and derive the following formula for $T(n)$ at the $k^{\text{th}}$ step:

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

*Step 3: Solve for k such that the base case is the only recursive term that is present on the right-hand side of the equation for $T(n)$. Determine the closed form solution by replacing instances of the base case with the value of the base case.*

Since we want the base case to be the only recursive term present on the right-hand side of the equation, we want to select a $k$ such that $T(n/2^k)$ equals the base case of $T(1)$. This happens when $n/2^k = 1$, or when $k = \log_2(n)$. Substituting $\log_2(n)$ for $k$ in our equation gives us an expression for $T(n)$ where $T(1)$ is the only recursive term on the right-hand side. We can then plug in $T(1) = 1$ into the equation.

$$\begin{aligned} T(n) &= 2^{\log_2(n)}T(1) + \sum_{i=0}^{\log_2(n)-1} 2^i \log_2\left(\frac{n}{2^i}\right) \\ &= nT(1) + \sum_{i=0}^{\log_2(n)-1} 2^i \log_2\left(\frac{n}{2^i}\right) \\ &= n + \sum_{i=0}^{\log_2(n)-1} 2^i \log_2\left(\frac{n}{2^i}\right) \end{aligned}$$

We can simplify the summation term using log rules:

$$\begin{aligned} T(n) &= n + \sum_{i=0}^{\log_2(n)-1} 2^i \log_2\left(\frac{n}{2^i}\right) \\ &= n + \sum_{i=0}^{\log_2(n)-1} 2^i \left(\log_2(n) - \log_2(2^i)\right) \\ &= n + \sum_{i=0}^{\log_2(n)-1} 2^i (\log_2(n) - i) \\ &= n + \left(\log_2(n) \sum_{i=0}^{\log_2(n)-1} 2^i\right) - \left(\sum_{i=0}^{\log_2(n)-1} i2^i\right) \end{aligned}$$

The first summation is the sum of a finite geometric series, which can be calculated using the summation formula:

$$S_n = \frac{a_1(1-r^n)}{1-r}$$

where $n$ is the number of terms, $a_1$ is the first term, and $r$ is the common ratio. Using $n = \log_2(n)$, $a_1 = 2^0 = 1$, and $r = 2$, we get

$$\log_2(n) \sum_{i=0}^{\log_2(n)-1} 2^i = \log_2(n)\left(\frac{\left(1-2^{\log_2(n)}\right)}{1-2}\right) = \log_2(n)(n-1) = n\log_2(n) - \log_2(n)$$

We can use the following identity (provided in the example) to solve the second summation in our original expression:

$$\sum_{i=0}^{n} i2^i = 2^{n+1}n - 2^{n+1} + 2$$

From this identity, we can simplify the second summation term as follows:

$$\begin{aligned} \sum_{i=0}^{\log(n)-1} i2^i &= 2^{\log_2(n)}(\log_2(n) - 1) - 2^{\log_2(n)} + 2 \\ &= n(\log(n) - 1) - n + 2 \\ &= n\log(n) - 2n + 2 \end{aligned}$$

Putting everything together, we get:

$$\begin{aligned} T(n) &= n + \left(\log_2(n) \sum_{i=0}^{\log_2(n)-1} 2^i\right) - \left(\sum_{i=0}^{\log_2(n)-1} i2^i\right) \\ &= n + (n\log_2(n) - \log_2(n)) - (n\log_2(n) - 2n + 2) \\ &= 3n - \log_2(n) - 2 \end{aligned}$$

By dropping coefficients and lower order terms, we see that the binary partition with binary search algorithm has a time complexity of $\Theta(n)$.

From our analysis, we determined the following:

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

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

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

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

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

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