CSE 250: Data Structures

Course Reference

Contents

1. Introduction

2. Math Refresher

3. Asymptotic Runtime Complexity

Data Structures are the fundamentals of algorithms: How efficient an algorithm is depends on how the data is organized. Think about your workspace: If you know exactly where everything is, you can get to it much faster than if everything is piled up randomly. Data structures are the same: If we organize data in a way that meets the need of an algorithm, the algorithm will run much faster (remember the array vs linked list comparison from earlier)?

Since the point of knowing data structures is to make your algorithms faster, one of the things we'll need to talk about is how fast the data structure (and algorithm) is for specific tasks. Unfortunately, "how fast" is a bit of a nuanced comparison. I could time how long algorithms **A** and **B** take to run, but what makes a fair comparison depends on a lot of factors:

- How big is the data that the algorithm is running on?
  - **A** might be faster on small inputs, while **B** might be faster on big inputs.
- What computer is running the algorithm?
  - **A** might be much faster on one computer, **B** might be much faster on a network of computers.
  - **A** might be especially tailored to Intel x86 CPUs, while **B** might be tailored to the non-uniform memory latencies of AMD x86 CPUs.
- How optimized is the code?
  - Hand-coded optimizations can account for multiple orders of magnitude in algorithm performance.

In short, comparing two algorithms requires a lot of careful analysis and experimentation. This is important, but as computer scientists, it can also help to take a more abstract view. We would like to have a shorthand that we can use to quickly convey the 50,000-ft view of "how fast" the algorithm is going to be. That shorthand is asymptotic runtime complexity.

3.1. Some examples of asymptotic runtime complexity

Look at the documentation for data structures in your favorite language's standard library. You'll see things like:

- The cost of appending to a Linked List is $O(1)$
- The cost of finding an element in a Linked List is $O(N)$
- The cost of appending to an Array List is $O(N)$, but amortized $O(1)$
- The cost of inserting into a Tree Set is $O(\log N)$
- The cost of inserting into a Hash Map is Expected $O(1)$, but worst case $O(N)$
- The cost of retrieving an element from a Cuckoo Hash is always $O(1)$

These are all examples of asymptotic runtimes, and they give you a quick at-a-glance idea of how well the data structure handles specific operations. Knowing these facts about the data structures involved can help you plan out the algorithms you're writing, and avoid picking a data structure that tanks the performance of your algorithm.

3.2. Runtime Growth Functions

Let's start by talking about **Runtime Growth Functions**. A runtime growth function looks like this:

T(N)

Here, T is the name of the function (We usually use T for runtime growth functions, and N is the *size* of the input.

You can think of this function as telling us "For an input of size N, this algorithm will take $T(N)$ seconds to run" This is a little bit of an inexact statement, since the actual number of seconds it takes depends on the type of computer, implementation details, and more. We'll eventually generalize, but for now, you can assume that we're talking about a specific implementation, on a specific computer (like e.g., your computer).

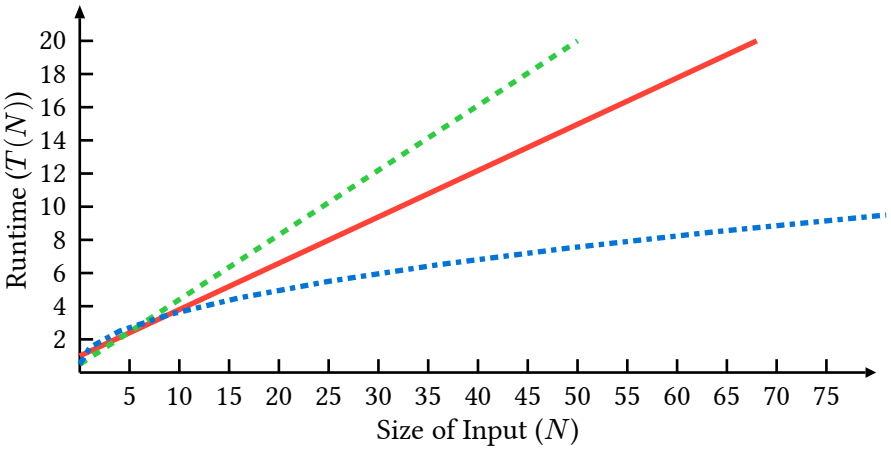
We call this a growth function because it generally has to follow a few rules:

- For all $N \geq 0$, it must be the case that $T(N) \geq 0$
  - The algorithm can't take negative time to run.
- For all $N \geq N'$, it must be the case that $T(N) \geq T(N')$
  - It shouldn't be the case that the algorithm runs faster on a bigger input[1].

3.3. Complexity Classes

Before we define the idea of asymptotic runtimes precisely, let's start with an intuitive idea. We're going to take algorithms (including algorithms that perform specific operations on a data structure), and group them into what we call **Complexity Classes**[2].

Graph 1 shows three different runtime growth functions: Green (dashed), Red (solid), and Blue (dash-dotted). For an input of size $N = 2$, the green dashed function appears to be the best, while the blue dash-dotted function appears the worst. By the time we get to $N = 10$, the roles have reversed, and the blue dash-dotted function is the best.



Graph 1: Different types of growth functions

So let's talk about these lines and what we can say about them. First, in this book, we're going to ignore what happens for "small" inputs. This isn't always the right thing to do, but we're taking the 50,000 ft view of algorithm performance. From this perspective, the blue dot-dashed line is the "best".

But why is it better? If we look closely, both the green dashed and the red solid line are straight lines. The blue dot-dashed line starts going up faster than both the other two lines, but bends downward. In short, the blue dot-dashed line draws a function of the form $a \log(N) + b$, while the other two lines draw functions of the form . For "big enough" values of N, any function of the form $a \log(N) + b$ will

[1] In practice, this is not actually the case. We'll see a few examples of functions who's runtime can sometimes be faster on a bigger input. Still, for now, it's a useful simplification.

[2] To be pedantic, what we'll be describing is called "simple complexity classes", but throughout this book, we'll refer to them as just complexity classes.

always be smaller than any function of the form $a \cdot N + b$. On the other hand, the value of any two functions of the form $a \cdot N + b$ will always "look" the same. No matter how far we zoom out, those functions will always be a constant factor different.

Our 50,000 foot view of the runtime of a function (in terms of N, the size of its input) will be to look at the "shape" of the curve as we plot it against N.

3.3.1. Example

Try the following code in python:

```
from random import randrange
from datetime import datetime
N = 10000
TRIALS = 1000
data = []
for x in range(N):
    data += [x]
data = list(data)

contained = 0
start_time = datetime.now()
for x in range(TRIALS):
    if randrange(N) in data:
        contained += 1
end_time = datetime.now()

time = (end_time - start_time).total_seconds() / TRIALS

print(f"For N = {N}, that took {time} seconds per lookup")
```

This code creates a list of `N` elements, and then does `TRIALS` checks to see if a randomly selected value is somewhere in the list. This is a toy example, but see what happens as you increase the value of `N`. In most versions of python, you'll find that every time you multiply `N` by a factor of, for example 10, the total time taken per lookup grows by the same amount.

Now try something else. Modify the code so that the `data` variable is initialized as:

```
data = []
for x in range(N):
    data += [x]
data = set(data)
```

You'll find that now, as you increase `N`, the time taken **per lookup** grows at a much smaller rate. Depending on the implementation of python you're using, this will either grow as $\log N$ or only a tiny bit. The `set` data structure is much faster at checking whether an element is present than the `list` data structure.

Complexity classes are a language that we can use to capture this intuition. We might say that `set`'s implementation of the `in` operator belongs to the **logarithmic** complexity class, while `list`'s implementation of the operator belongs to the **linear** complexity class. Just saying this one fact about the two implementations makes it clear that, in general, `set`'s version of `in` is much better than `list`'s.

3.3.2. Formal Notation

Sometimes it's convenient to have a shorthand for writing down that a runtime belongs in a complexity class. We write:

$g(N) \in \Theta(f(n))$

... to mean that the mathematical function $g(N)$ belongs to the same **asymptotic complexity class** as $f(N)$. You may also see this written as an equality. For example

$T(N) = \Theta(N)$

... means that the runtime function $T(N)$ belongs to the **linear** complexity class. Continuing the example above, we would use our new shorthand to describe the two implementations of Python's `in` operator as:

- $T_{\text{set}} \in \Theta(\log N)$
- $T_{\text{list}} \in \Theta(N)$

Formalism: A little more formally, $\Theta(f(N))$ is the **set** of all mathematical functions $g(N)$ that belong to the same complexity class as $f(N)$. So, writing $g(N) \in \Theta(f(N))$ is saying that $g(N)$ is in ($\in$) the set of all mathematical functions in the same complexity class as $f(N)$[3].

Here are some of the more common complexity classes that we'll encounter throughout the book:

- **Constant**: $\Theta(1)$
- **Logarithmic**: $\Theta(\log N)$
- **Linear**: $\Theta(N)$
- **Loglinear**: $\Theta(N \log N)$
- **Quadratic**: $\Theta(N^2)$
- **Cubic**: $\Theta(N^3)$
- **Exponential** $\Theta(2^N)$

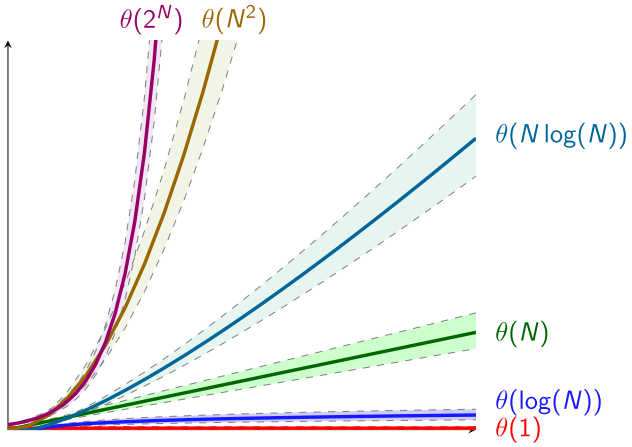


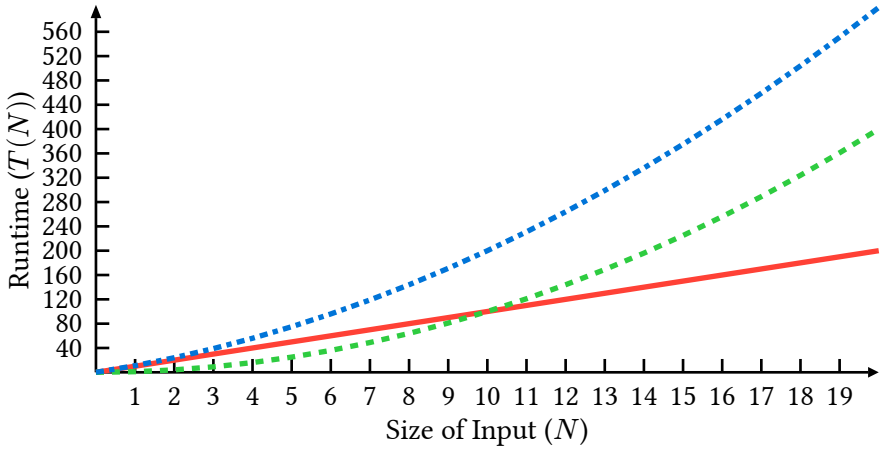
Figure 1: $\Theta(f(N))$ is the set of all mathematical functions including $f(N)$ and everything that has the same "shape", represented in the chart above as a highlighted region around each line.

[3]We are sweeping something under the rug here: We haven't precisely defined what it means for two functions to be in the same complexity class yet. We'll get to that shortly, after we introduce the concept of complexity bounds.

Complexity classes are given in order. The later they appear in the list, the faster they grow. Any function that has a **linear** shape, will always be smaller (for big enough values of N) than a function with a **loglinear** shape.

3.3.3. Polynomials and Dominant Terms

What complexity class does $10N + N^2$ fall into? Let's plot it and see:



Graph 2: Comparing N (solid red), N^2 (dashed green), and $10N + N^2$ (dash-dotted blue)

Graph 2 compares these three functions. Observe that the dash-dotted blue line starts off very similar to the solid red line. However, as N grows, its shape soon starts resembling the dashed green N^2 more than the solid red $10N$.

This is a general pattern. In any polynomial (a sum of mathematical expressions), for really big values of N, the complexity class of the "biggest" term starts to win out once we get to really big values of N.

In general, for any sum of mathematical functions:

$g(N) = f_1(N) + f_2(N) + ... + f_{k(N)}$

The complexity class of $g(N)$ is the greatest complexity class of any $f_{i(N)}$

For example:

- $10N + N^2 \in \Theta(N^2)$
- $2^N + 4N \in \Theta(2^N)$
- $1000 \cdot N\log(N) + 5N \in \Theta(N\log(N))$

3.3.4. Θ in mathematical formulas

Sometimes we'll write $\Theta(g(N))$ in regular mathematical formulas. For example, we could write:

$\Theta(N) + 5N$

You should interpret this as meaning any function that has the form:

$f(N) + 5N$

... where $f(N) \in \Theta(N)$.

3.3.5. Code Complexity

Let's see a few examples of how we can figure out the runtime complexity class of a piece of code.

```
def userFullName(users: List[User], id: int) -> str:
    user = users[id]
    fullName = user.firstName + " " + user.lastName
    return fullName
```

The `userFullName` function takes a list of users, and retrieves the `id`th element of the list and generates a full name from the user's first and last names. For now, we'll assume that looking up any element of any array (`users[id]`), string concatenation (`user.firstName + " " + user.lastName`), assignment (`user = ...`, and `fullName`), and returns are all constant-time operations[4].

Under these assumptions, the first, second, and third lines can each be evaluated in constant time $\Theta(1)$. The total runtime of the function is the time required to run each line, one at a time, or:

$$T_{\text{userFullName}}(N) = \Theta(1) + \Theta(1) + \Theta(1)$$

Recall above, that $\Theta(1)$ in the middle of an arithmetic expression can be interpreted as $f(N)$ where $f(N) \in \Theta(1)$ (it is a constant). That is, $f(N) = c$. So, the above expression can be rewritten as[5]:

$$T_{\text{userFullName}}(N) = c_1 + c_2 + c_3$$

Adding three constant values together (even without knowing what they are, exactly) always gets us another constant value. So, we can say that $T_{\text{userFullName}}(N) \in \Theta(1)$.

```
def updateUsers(users: List[User]) -> None:
    x = 1
    for user in users:
        user.id = x
        x += 1
```

The `updateUsers` function takes a list of users and assigns each user a unique id. For now, we'll assume that the assignment operations (`x = 1` and `user.id`), and the increment operation (`x += 1`) all take constant ($\Theta(1)$) time. So, we can model the total time taken by the function as:

$$T_{\text{updateUsers}}(N) = O(1) + \sum_{\text{user}}(O(1) + O(1))$$

Simplifying as above, we get

$$T_{\text{updateUsers}}(N) = c_1 + \sum_{\text{user}}(c_2 + c_3)$$

Recalling the rule for summation of a constant, using N as the total number of users, and then the rule for sums of terms, we get:

$$T_{\text{updateUsers}}(N) = c_1 + N \cdot (c_2 + c_3) = \Theta(N)$$

3.4. Complexity Bounds

Not every mathematical function fits neatly into a single complexity class. Let's go to our python code example above. The `in` operator tests to see whether a particular value is present in our data. If `data` is a `list`, then the implementation checks every position in the list, in order. Internally, Python implements the expression `target in data` with something like:

---

[4] Array lookups being constant-time is a huge simplification, called the RAM model, that we'll roll back at the end of the book.

[5] There's another simplification here. Technically, $f(N)$ is always within a bounded factor of a constant c_1, and likewise for $g(N)$, but we'll clarify this when we get to complexity bounds below.

```
def __in__(data, target):
    N = len(data)
    for i in range(N):
        if data[i] == target:
            return True
    return False
```

In the best case, the value we're looking for happens to be at the first position `data[0]`, and the code returns after a single check. In the worst case, the value we're looking for is at the last position `data[N-1]` or is not in the list at all, and we have to check every one of the `N` positions. Put another way, the **best case** behavior of the function is constant (exactly one check), while the **worst case** behavior is linear (N checks). We can write the resulting runtime using a case distinction:

$$T_{\text{in}}(N) = \begin{cases} a \cdot 1 + b & \text{if } \texttt{data[0]} = \texttt{target} \\ a \cdot 2 + b & \text{if } \texttt{data[1]} = \texttt{target} \\ \cdots & \\ a \cdot (N-1) + b & \text{if } \texttt{data[N-2]} = \texttt{target} \\ a \cdot N + b & \text{if } \texttt{data[N-1]} = \texttt{target} \end{cases}$$

We don't know the runtime exactly, as it is based on the computer and version of python we are using. However, we can model it, in general in terms of some upfront cost b (e.g., for computing `N = len(data)`), and some additional cost a for every time we go through the loop (e.g., for computing `data[i] == target`). Since we don't know where the target is, exactly,

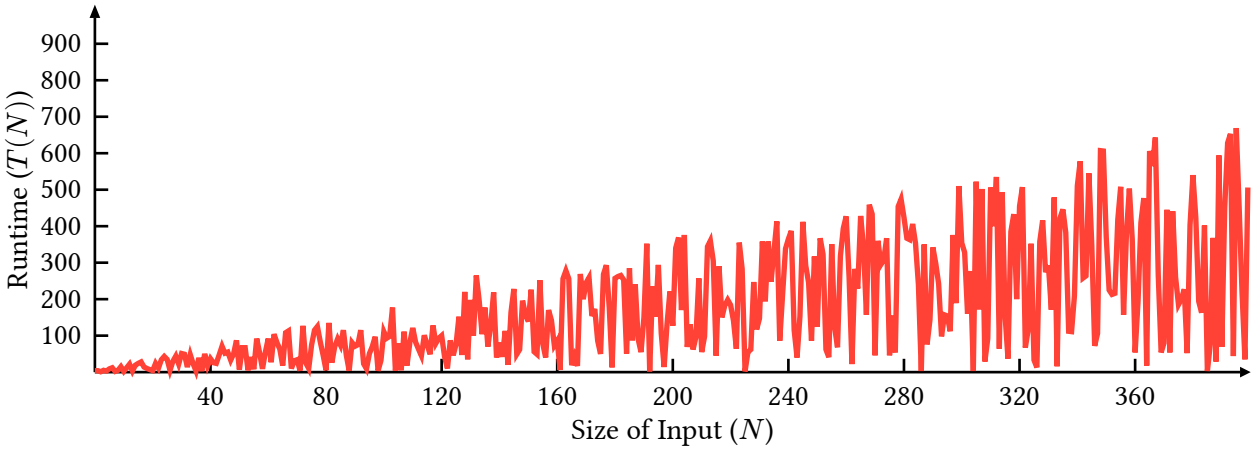
Let's do a quick experiment. The code below is like our example above, but measures the time for one lookup at a time. Each point it prints out is the runtime of a single lookup as the `list` gets bigger and bigger.

```
from random import randrange
from datetime import datetime

N = 100000
TRIALS = 400
STEP = int(N/TRIALS)

data = list()

for i in range(TRIALS):
    # Increase the list size by STEP
    for j in range(STEP):
        data += [i * STEP + j]
    start = datetime.now()
    # Measure how long it takes to look up a random element
    if randrange(i * STEP + STEP) in data:
        pass
    end = datetime.now()
    # Print out the total time in microseconds
    microseconds = (end - start).total_seconds() * 1000000
    print(f"{i}, {microseconds}")
```

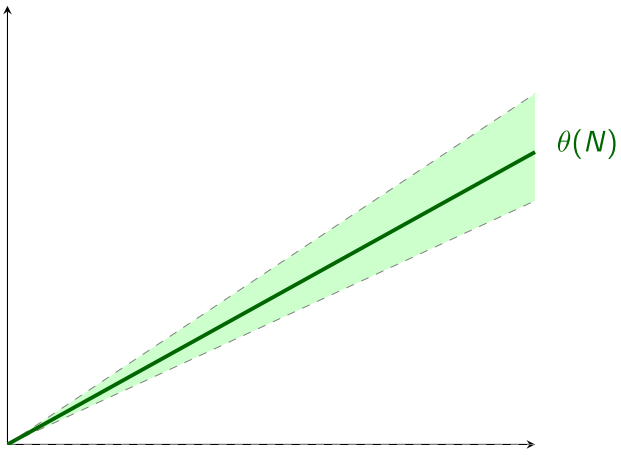


Graph 3: Scaling the `list` lookup.

Graph 3 shows the output of one run of the code above. You can see that it looks a lot like a triangle. The **worst case** (top of the triangle) looks a lot like the **linear** complexity class ($\Theta(N)$, or an angled line), but the **best case** (bottom of the triangle) looks a lot more like a flat line, or the **constant** complexity class ($\Theta(1)$, or a flat line). The runtime is *at least* constant, and *at most* linear: We can **bound** the runtime of the function between two complexity classes.

3.4.1. Big-O and Big-Ω

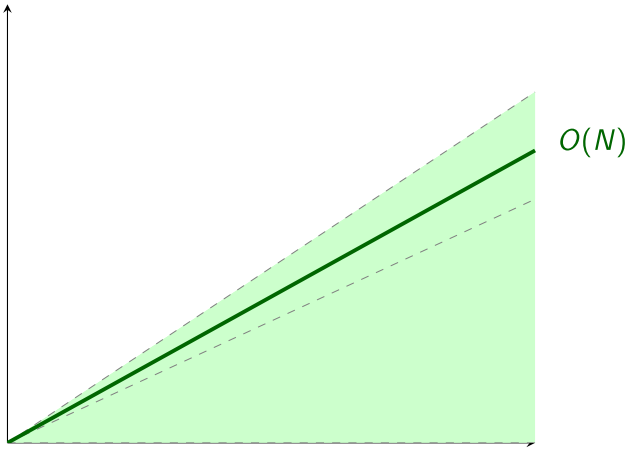
We capture this intuition of bounded runtime by introducing two new concepts: Worst-case (upper, or Big-O) and Best-case (lower, or Big-Ω) bounds. To see these in practice, let's take the linear complexity class as an example:



We write $O(N)$ to mean the set of all mathematical functions that are **no worse than $\Theta(N)$**. This includes:

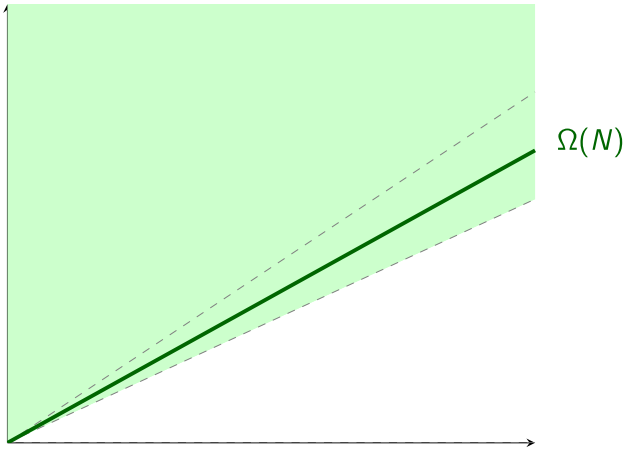
- all mathematical functions in $\Theta(N)$ itself
- all mathematical functions in lesser (slower-growing) complexity classes (e.g., $\Theta(1)$)
- any mathematical function that never grows faster than $O(N)$ (e.g., the runtime of each individual lookup in our Python example above)

The figure below illustrates $\Theta(N)$ (the dotted region) and all lesser complexity classes. Note the similarity to Graph 3.



Similarly, we write $\Omega(N)$ to mean the set of all mathematical functions that are **no better than $\Theta(N)$**. This includes:

- all functions in $\Theta(N)$ itself
- all greater (faster-growing) complexity classes (e.g., $\Theta(N^2)$), and anything in between.



To summarize, we write:

- $f(N) \in O(g(N))$ to say that $f(N)$ is in $\Theta(g(N))$ or a lesser complexity class.
- $f(N) \in \Omega(g(N))$ to say that $f(N)$ is in $\Theta(g(N))$ or a greater complexity class.

3.4.2. Formalizing Big-O

Before we formalize our bounds, let's first figure out what we want out of that formalism.

Let's start with the basic premise we outlined above: For a function $f(N)$ to be in the set $O(g(N))$, we want there to be some function in $\Theta(g(N))$ that is always bigger than $f(N)$.

The first problem we run into with this formalism is that we haven't really defined what exactly $\Theta(g(N))$ is yet, so we need to pin down something first. Let's start with the same assumption we made earlier: we can scale $g(N)$ by any constant value without changing its complexity class. Formally:

Formally: $\forall c : c \cdot g(N) \in \Theta(g(N))$

That is, for any constant c ($\forall$ means 'for all'), the product $c \cdot g(N)$ is in $\Theta(g(N))$ (remember that $\in$ means is in). This isn't meant to be all-inclusive: There are many more functions in $\Theta(g(N))$, but this gives us a pretty good range of functions that, at least intuitively, belong in $g(N)$'s complexity class.

Now we have a basis for formalizing Big-O: We can say that $f(N) \in O(g(N))$ if there is **some** multiple of $g(N)$ that is always bigger than $f(N)$. Formally:

$\exists c > 0, \forall N : f(N) \leq c \cdot g(N)$

That is, there exists ($\exists$ means there exists) some positive constant c, such that for each value of N, the value of $f(N)$ is smaller than the corresponding value of $c \cdot g(N)$.

Let's look at some examples:

Example: $f(N) = 2N$ vs $g(N) = N$

Can we find a c and show that for this c, for all values of N, the Big-O inequality holds for f and g?

- $f(N) \leq c \cdot g(N)$
- $2N \leq c \cdot N$
- $2 \leq c$

We start with the basic inequality, substitute in the values of $f(N)$ and $g(N)$, and then divide both sides by N. So, the inequality is always true for any value of $c \geq 2$.

Example: $f(N) = 100N^2$ vs $g(N) = N^2$

Can we find a c and show that for this c, for all values of N, the Big-O inequality holds for f and g?

- $f(N) \leq c \cdot g(N)$
- $100N^2 \leq c \cdot N^2$
- $100 \leq c$

We start with the basic inequality, substitute in the values of $f(N)$ and $g(N)$, and then divide both sides by N. So, the inequality is always true for any value of $c \geq 100$.

Example: $f(N) = N$ vs $g(N) = N^2$

Can we find a c and show that for this c, for all values of N, the Big-O inequality holds for f and g?

- $f(N) \leq c \cdot g(N)$
- $N \leq c \cdot N^2$
- $1 \leq c \cdot N$

Uh-oh! For $N = 0$, there is no possible value of c that we can plug into that inequality to make it true ($0 \cdot c$ is never bigger than 1 for any c).

Attempt 2: So what went wrong? Well, we mainly care about how $f(N)$ behaves for really big values of N. In fact, for the example, for any $N \geq 1$ (and $c \geq 1$), the inequality is satisfied! It's just that pesky $N = 0$![6].

So, we need to add one more thing to the formalism: the idea that we only care about "big" values of N. Of course, that leaves the question of how big is "big"? Now, we could pick specific cutoff values, but

[6]Recall that we're only allowing non-negative input sizes (i.e., $N \geq 0$), so negative values of N aren't a problem.

any specific cutoff we picked would be entirely arbitrary. So, instead, we just make the cutoff definition part of the proof: When proving that $f(N) \in O(g(N))$, we just need to show that **some** cutoff exists, beyond which $f(N) \leq c \cdot g(N)$. **The formal definition of Big-O is**:

$f(N) \in O(g(N)) \Leftrightarrow \exists c > 0, N_0 \geq 0 : \forall N \geq N_0 : f(N) \leq c \cdot g(N)$

This is the same as our first attempt, with only one thing added: N_0. In other words, $f(N) \in O(g(N))$ if we can pick some cutoff value ($\exists N_0 \geq 0$) so that for every bigger value of N ($N \geq N_0$), $f(N)$ is smaller than $c \cdot g(N)$.

3.4.2.1. Proving a function has a specific Big-O bound

To show that a mathematical function is **in** $O(g(N))$, we need to find a c and an N_0 for which we can prove the Big-O inequality. A generally useful strategy is:

1. Write out the Big-O inequality
2. "plug in" the values of $f(N)$ and $g(N)$
3. "Solve for" c, putting it on one side of the inequality, with everything else on the other side.
4. Try a safe default of $N_0 = 1$.
5. Use the $A \leq B$ and $B \leq C$ imply $A \leq C$ trick (transitivity of inequality) to replace any term involving N with N_0
5. Use the resulting inequality to find a lower bound on c

Continuing the above example:

- $f(N) \leq c \cdot g(N)$
- $N \leq c \cdot N^2$
- $\frac{1}{N} \leq c$

For that last step, we have $N_0 \leq N$, or $1 \leq N$, so dividing both sides by N, we get $\frac{1}{N} \leq 1$. So, if we pick $1 \leq c$, then $\frac{1}{N} \leq 1 \leq c$, and $\frac{1}{N} \leq c$.

3.4.2.2. Proving a function does not have a specific Big-O bound

To show that a mathematical function is **not in** $O(g(N))$, we need to prove that there can be **no** c or N_0 for which we can prove the Big-O inequality. A generally useful strategy is:

1. Write out the Big-O inequality
2. "plug in" the values of $f(N)$ and $g(N)$
3. "Solve for" c, putting it on one side of the inequality, with everything else on the other side.
4. Simplify the equation on the opposite side and show that it is strictly growing. Generally, this means that the right-hand-side is in a complexity class at least N.

Flipping the above example:

- $f(N) \leq c \cdot g(N)$
- $N^2 \leq c \cdot N$
- $N \leq c$

N is strictly growing: for bigger values of N, it gets bigger. There is no constant that can upper bound the mathematical function N.

3.4.3. Formalizing Big-Ω

Now that we've formalized Big-O (the upper bound), we can formalize Big-Ω (the lower bound) in exactly the same way:

$f(N) \in O(g(N)) \Leftrightarrow \exists c > 0, N_0 \geq 0 : \forall N \geq N_0 : f(N) \geq c \cdot g(N)$

The only difference is the direction of the inequality: To prove that a function exists in Big-Ω, we need to show that $f(N)$ is bigger than some constant multiple of $g(N)$.

3.4.4. Formalizing Big-Θ

Although we started with an intuition for Big-Θ, we haven't yet formalized it. To understand why, let's take a look at the following runtime:

$T(N) = 10N + \text{rand}(10)$

Here $\text{rand}(10)$ means a randomly generated number between 0 and 10 for each call. If the function were **just** $10N$, we'd be fine in using our intuitive definition of $\Theta(N)$ being all multiples of N. However, this function still "behaves like" $g(N) = N$... just with a little random noise added in. For big values of N (e.g., 10^{10}), the random noise is barely perceptible. Although we can't say that $T(N)$ is **equal to** some multiple $c \cdot N$, we can say that it is **close to** that multiple (in fact, it's always between $10N$ and $10N + 10$). In other words, we can bound it from both above and below!

Let's try proving this with the tricks we developed for Big-O and Big-Ω:

- $T(N) \leq c_{\text{upper}} \cdot N$
- $10N + \text{rand}(10) \leq c_{\text{upper}} \cdot N$ (plug in $T(N)$)
- $10 + \frac{\text{rand}(10)}{N} \leq c_{\text{upper}}$ (divide by N)

Looking at this formula, we can make a few quick observations. First, by definition $\text{rand}(10)$ is never bigger than 10. Second, if $N_0 = 1$, $\frac{1}{N}$ can never be bigger than 1. In other words, $\frac{\text{rand}(10)}{N}$ can not be bigger than 10. Let's prove that to ourselves.

Taking the default $N_0 = 1$ we get:
- $1 \leq N$ (plug in N_0)
- $10 \leq 10N$ (multiply by 10)
- $\text{rand}(10) \leq 10 \leq 10N$ (transitivity with $\text{rand}(10) \leq 10$)
- $\frac{\text{rand}(10)}{N} \leq 10$ (divide by N)
- $10 + \frac{\text{rand}(10)}{N} \leq 10 + 10$ (add 10)

So, if we pick $c_{\text{upper}} \geq 20$, we can show (again, by transitivity):

$10 + \frac{\text{rand}(10)}{N} \leq c_{\text{upper}}$

Which gets us $T(N) \leq c_{\text{upper}} \cdot N$ for all $N > N_0$.

Now let's try proving a lower (Big-Ω) bound:

- $T(N) \geq c_{\text{lower}} \cdot N$
- $10N + \text{rand}(10) \geq c_{\text{lower}} \cdot N$ (plug in $T(N)$)
- $10 + \frac{\text{rand}(10)}{N} \geq c_{\text{lower}}$ (divide by N)
- $10 \geq c_{\text{lower}}$ (By transitivity: $10 \geq 10 + \frac{\text{rand}(10)}{N}$)

This inequality holds for any $10 \geq c_{\text{lower}} > 0$ (recall that c has to be strictly bigger than zero).

So, we've shown that $T(N) \in O(N)$ **and** $T(N) \in \Omega(N)$. The new thing is that we've shown that the upper and lower bounds **are the same**. That is, we've shown that $T(N) \in O(g(N))$ and $T(N) \in \Omega(g(N))$ **for the same mathematical function** g. If we can prove that an upper and lower bound for some mathematical function $f(N)$ that is the same mathematical function $g(N)$, we say that $f(N)$ and $g(N)$ are in the same complexity class. Formally, $f(N) \in \Theta(g(N))$ if and only if $f(N) \in O(g(N))$ **and** $f(N) \in \Omega(g(N))$.

3.4.5. Tight Bounds

In the example above, we said that $\text{rand}(10) \leq 10$. We could have just as easily said that $\text{rand}(10) \leq 100$. The latter inequality is just as true, but somehow less satisfying; yes, the random number will always be less than 100, but we can come up with a "tighter" bound (i.e., 10).

Similarly Big-O and Big-Ω are bounds. We can say that $N \in O(N^2)$ (i.e., N is no worse than N^2). On the other hand, this bound is just as unsatisfying as $\text{rand}(10) \leq 100$, we can do better.

If it is not possible to obtain a better Big-O or Big-Ω bound, we say that the bound is **tight**. For example:

- $10N^2 \in O(N^2)$ and $10N^2 \in \Omega(N^2)$ are tight bounds.
- $10N^2 \in O(2^N)$ is correct, but **not** a tight bound.
- $10N^2 \in \Omega(N)$ is correct, but **not** a tight bound.

Note that since we define Big-Θ as the intersection of Big-O and Big-Ω, all Big-Θ bounds are, by definition tight. As a result, we sometimes call Big-Θ bounds "tight bounds".

3.5. Summary

We defined three ways of describing runtimes (or any other mathematical function):

- Big-O: The worst-case complexity:
  - $T(N) \in O(g(N))$ means that the runtime $T(N)$ scales **no worse than** the complexity class of $g(N)$
- Big-Ω: The best-case complexity
  - $T(N) \in \Omega(g(N))$ means that the runtime $T(N)$ scales **no better than** the complexity class of $g(N)$
- Big-Θ: The tight complexity
  - $T(N) \in \Theta(g(N))$ means that the runtime $T(N)$ scales **exactly as** the complexity class of $g(N)$

We'll introduce amortized and expected runtime bounds later on in the book; Since these bounds are given without qualifiers, and so are sometimes called the **Unqualified** runtimes.

3.5.1. Formal Definitions

For any two functions $f(N)$ and $g(N)$ we say that:

- $f(N) \in O(g(N))$ if and only if $\exists c > 0, N_0 \geq 0 : \forall N > N_0 : f(N) \leq c \cdot g(N)$
- $f(N) \in \Omega(g(N))$ if and only if $\exists c > 0, N_0 \geq 0 : \forall N > N_0 : f(N) \geq c \cdot g(N)$
- $f(N) \in \Theta(g(N))$ if and only if $f(N) \in O(g(N))$ and $f(N) \in \Omega(g(N))$

Note that a simple $\Theta(g(N))$ may not exist for a given $f(N)$, specifically when the tight Big-O and Big-Ω bounds are different.

3.5.2. Simple Complexity Classes

We will refer to the following specific complexity classes:

- **Constant**: $\Theta(1)$
- **Logarithmic**: $\Theta(\log N)$
- **Linear**: $\Theta(N)$
- **Loglinear**: $\Theta(N \log N)$
- **Quadratic**: $\Theta(N^2)$
- **Cubic**: $\Theta(N^3)$
- **Exponential** $\Theta(2^N)$

These complexity classes are listed in order.

3.5.3. Dominant Terms

In general, any function that is a sum of simpler functions will be dominated by one of its terms. That is:

$$f(N) = f_1(N) + f_2(N) + ... + f_{k(N)}$$

The asymptotic complexity of $f(N)$ (i.e., its Big-O and Big-Ω bounds, and its Big-Θ bound, if it exists) will be the **greatest** complexity of any individual term $f_{i(N)}$.

4. The Sequence and List ADTs

Now that we have the right language to talk about algorithm runtimes (asymptotic complexity), we can start talking about actual data structures that these algorithms can use. Since the choice of data structure can have a huge impact on the complexity of an algorithm, it's often useful to group specific data structures together by the roles that they can fulfill in an algorithm. We refer to such a grouping as an **abstract data type** or ADT.

4.1. What is an ADT?

An ADT is, informally, a contract that states what we can expect from a data structure. Take, for example, a Java `interface` like the following:

```
public interface Narf<T>
{
  public void foo(T poit, int zort);
  public T bar(int zort);
  public int baz();
}
```

This `interface` states that any class that implements `Moof` must provide `foo`, `bar`, and `baz` methods, with arguments and return values as listed above. This is not especially helpful, since it doesn't give us any idea of what these methods are supposed to do. Contrast `Moof` with the following interface:

```
public interface MutableSequence<T>
{
  public void update(T value, int index);
  public T get(int index);
  public int size();
}
```

`Sequence` is semantically identical to `Narf`, but far more helpful. Just by reading the names of these methods, you get some idea of what each method does, and how it interacts with the state represented by a `Sequence`. You can build a mental model of what a Sequence is from these names.

An Abstract Data Type is:
1. A mental model of some sort of data (a data type)
2. One or more operations for accessing or modifying that state (an interface)
3. Any other rules for the state (constraints)

For most ADTs discussed in this book, the state modeled by the ADT is some sort of collection of elements.

A data structure is a **specific** strategy for organizing data. We say that the data structure implements (or conforms to) an ADT if:
1. The data structure stores the same type of data as the ADT
2. Each of the operations required by the ADT can be implemented on the data structure
3. The operation implementations are guaranteed to respect the rules for the state.

4.2. The Sequence ADT

A very common example of an ADT is a sequence. Examples include:

- The English alphabet ('A', 'B', ..., 'Z')

- The Fibbonacci Sequence (1, 1, 2, 3, 5, 8, ...)
- An arbitrary collection of numbers (42, 19, 86, 23, 19)

What are some commonalities in these examples?

- Every sequence is a collection of elements of some type `T` (integer, character, etc...)
- The same element may appear multiple times.
- Every sequence has a size N (which may be infinite).
- Every element (or occurrence of an element) is assigned to an index: $0 \leq \text{index} < N$.
- Every index in the range $0 \leq \text{index} < N$ has exactly one element assigned to it.

What kind of operations can we do on a sequence?

```
public interface Sequence<T>
{
  /** Retrieve the element at a specific index */
  public T get(int index);
  /** Obtain the size of the sequence */
  public int size();
}
```

4.2.1. Sequences by Rule

For some of the example sequences listed above, we can implement this interface directly. For example, for the English alphabet:

```
public class Alphabet implements Sequence<Character>
{
  /** Retrieve the element at a specific index */
  public Character get(int index)
  {
    if(index == 0){ return 'A'; }
    if(index == 1){ return 'B'; }
    /* ... */
    if(index == 25){ return 'Z'; }
    throw IndexOutOfBoundsException("No character at index "+index)
  }
  /** Obtain the size of the sequence */
  public int size()
  { return 26; }
}
```

Similarly, we can implement the Fibonacci sequence according to the Fibonacci rule $\text{Fib}(x) = \text{Fib}(x-1) + \text{Fib}(x-2)$:

```
public class Fibonacci implements Sequence<Int>
{
  /** Retrieve the element at a specific index */
  public Int get(int index)
  {
    if(index < 0){
      throw IndexOutOfBoundsException("Invalid index: "+index)
    }
    if(index == 0){ return 0; }
```

```
    if(index == 1){ return 1; }
    return get(index-1) + get(index-2);
  }
  /** Obtain the size of the sequence */
  public int size()
  { return INFINITY; }
}
```

4.2.2. Arbitrary Sequences

Often, however, we want to represent an sequence of **arbitrary** elements. We won't have simple rule we can use to translate the index into a value. Instead, we need to store the elements somewhere if we want to be able to retrieve them. We need an **array**.

Let's take a brief diversion into the guts of a computer. Most computers store data in a component called RAM[7]. You can think of RAM as being a ginormous sequence of bits (0/1 values). We've set up some rules for how bits can be used to encode other information. The details of how that works are not immediately relevant to us, but two things are useful to know:

1. Most basic data values (integers, decimals, characters, dates) can be represented using a fixed number of bits (e.g., 8 bits per character This is another blatant lie. Characters can sometimes take 16 bits, and for special characters like emoji, they can consist of a semi-random group of 16-bit sequences. String encodings are a real mess. Similarly, integers and decimals are only representable up to a fixed upper bound/precision. Don't get us started on dates. For most purposes though, this is a reasonable assumption.)
2. The number of bits required in most cases are multiples of 8, so we usually talk about

] the number of bytes required (1 byte = 8 bits).

For example, the ASCII character encoding forms the basis for most character representations we use today, and assigns each of the letters of the English alphabet (upper and lower case, and a few other symbols) to a particular sequence of 8 bits (one byte). Let's say we wanted to store the sequence 'H', 'e', 'l', 'l', 'o'. The ASCII code for 'H' is hexadecimal 0x48, or binary 01001000. We would store that byte at one position in RAM, then the ASCII encoding of 'e' at the next byte, and then the encoding of 'l' at the next byte, and so forth.

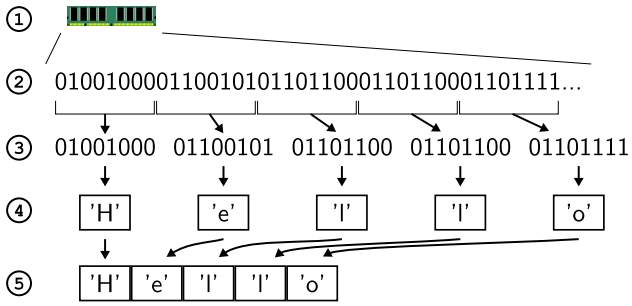


Figure 5: RAM (1) can be viewed as a sequence of bits (2). We can break these bits up into fixed size chunks (3). Each fixed size chunk can be used to identify some value, like for example a character (4). Thus a sequence of characters can be stored somewhere in RAM (5).

[7]This is a pretty blatant lie... but a useful simplification for now. We'll walk the lie back slightly later in the book.

This arrangement, where we just concatenate elements side by side in memory is really useful, *if each element always uses up the same number of bytes*. Specifically, if we want to retrieve the ith element of the sequence, we need only two things:

1. The position of the 0th element in RAM (Let's call this S)
2. The number of bytes that each element takes up (Let's call this E).

The the ith element's E bytes will always start at byte $S + i \cdot E$ (with 0 being the first element).

4.2.2.1. Array Runtime Analysis

In order to retrieve the i'th element of an array, we need one multiplication, and one addition. This is true regardless of how big the array is. If the array has a thousand, or a million, or a trillion elements, we can still obtain the ith element with one multiplication and one addition. In other words, the cost of retrieving an element is constant (i.e., $\Theta(1)$)[8].

4.2.2.2. Arrays In Java

Java provides a convenient shorthand for creating and accessing arrays using square brackets. To instantiate an array, use `new`:

```
int[] myArray = new int[100];
```

Note that the type of an integer array is `int[]`[9], and the number of elements in the array must be specified upfront (`100` in the example above). To access an element of an array use `[index]`.

```
myArray[10] = 29;
int myValue = myArray[10];
assert(myValue == 29);
```

Java arrays are **bounds-checked**. That is, Java stores the size of the array as part of the array itself (the first 4 bytes of the array stores the number of elements). Whenever you access an array element, Java checks to make sure that the index is greater than or equal to zero and less than the size. If not, it throws an `IndexOutOfBounds` exception otherwise. As a convenient benefit, bounds checking means we can get the array size from Java.

```
int size = myArray.size
assert(size == 100)
```

To prove to ourselves that the `Array` implements the `Sequence` ADT, we can implement its methods:

```
public class ArraySequence<T> implements Sequence<T>
{
  T[] data;

  public ArraySequence(T[] data){ this.data = data; }
  public T get(int index) { return data[index]; }
```

[8]The attentive reader might note that we still need to retrieve E bytes, and so technically the cost of an array access is $\Theta(E)$. However, we're mainly interested in how runtime scales with the size of the collection, as E is fixed upfront, and usually very small. So, more precisely, **with respect to N**, the cost of retrieving an element from an array is $\Theta(1)$. The even more attentive reader might be aware of things like caches, which we're going to ignore until much later in the book.

[9]Java sort of allows you to store variable size objects in an array. For example, `String[]` is a valid array type. The trick to this is that Java allocates the actual string *somewhere else in RAM*. What it stores in the actual array is the address of the string (also called a pointer).

```
  public int size() { return data.size; }
}
```

Since there's no way to modify the array through the `Sequence` interface, the class initializer allows us to import a pre-constructed array.

4.2.3. Mutable Sequences

The Fibonacci sequence and the English alphabet are examples of **immutable** sequences: Sequences that are pre-defined and can not be changed. However, there's technically nothing stopping us from just modifying the bytes of an array's element. We can make our `Sequence` ADT a little more general by adding a way to modify its contents. We call the resulting ADT a `MutableSequence`.

```
public interface MutableSequence<T> extends Sequence<T>
{
  public void update(T value, int index)
}
```

The `extends` keyword in java can be used to indicate that an interface takes in all of the methods of the extended interface, and potentially adds more of its own. In this case `MutableSequence` has all of the methods of `Sequence`, plus its own `update` method.

To prove to ourselves that the `Array` implements the `MutaleSequence` ADT, we can implement its methods:

```
public class MutableArraySequence<T> implements MutableSequence<T>
{
  T[] data;

  public ArraySequence(int size){ this.data = new T[size]; }
  public void update(T value, int index) { data[index] = value; }
  public T get(int index) { return data[index]; }
  public int size() { return data.size; }
}
```

Note how we initialize the `Array` to a specified size instead of bringing in an existing array.

4.2.4. Array Summary

- Retrieving an element of an Array (i.e., `array[index]`) is $\Theta(1)$
- Updating an element of an array (i.e., `array[index] = ...`) is $\Theta(1)$
- Retrieving the array's size (i.e., `array.size`) is $\Theta(1)$

4.3. The List ADT