1. 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.

1.1. 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.

1.2. 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).

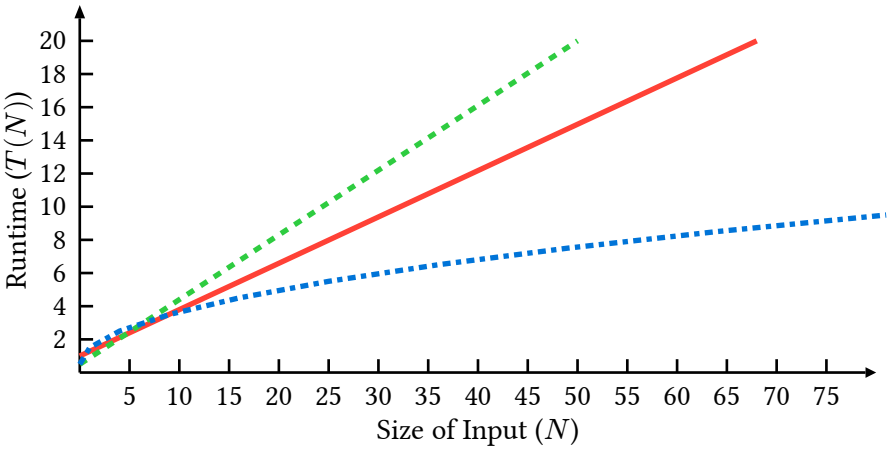
We call this a growth function because it generally has to follow a few rules:

- 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].

1.3. 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.

1.3.1. 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.

1.3.2. 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** $\Theta(2^N)$

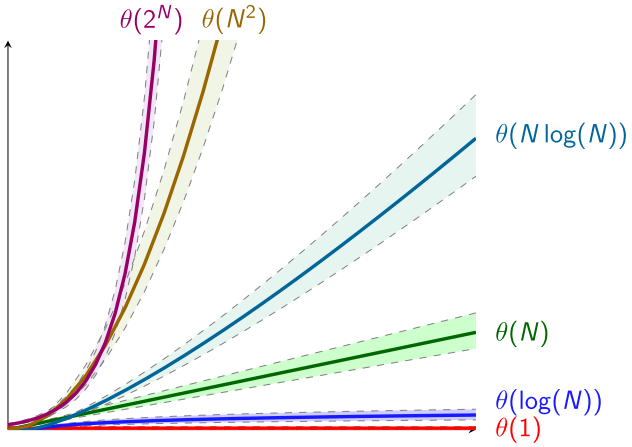


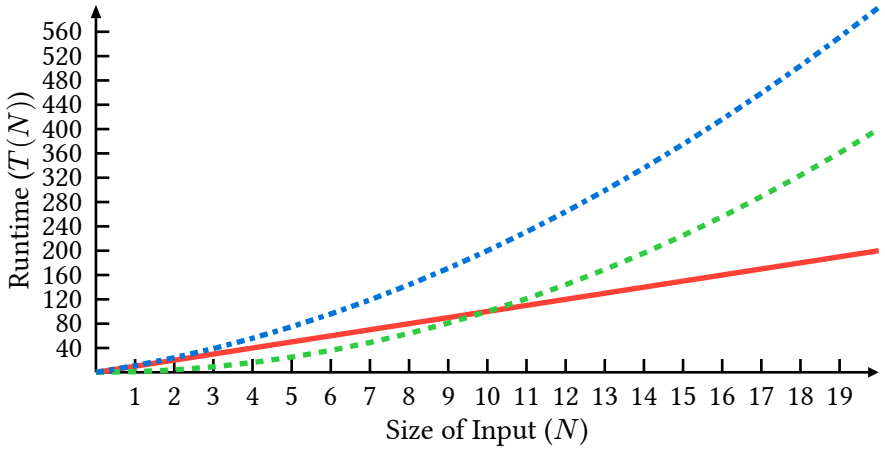
Figure 1: $\Theta(f(N))$ is the set of all mathematical functions including $f(N)$ and everything that has the same "shape", represented in the chart above as a highlighted region around each line.

[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.

1.3.3. 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:

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

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

1.3.5. 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:

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

1.4. 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
```

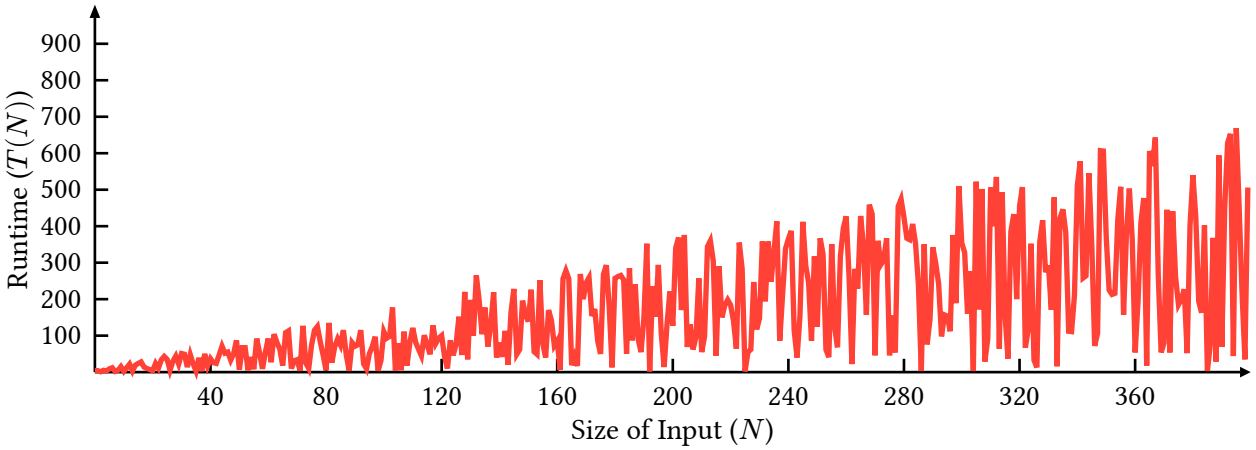
In the best case, the value we're looking for happens to be at the first position `data[0]`, and the code returns after a single check. In the worst case, the value we're looking for is at the last position `data[N-1]` or is not in the list at all, and we have to check every one of the `N` positions. 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.

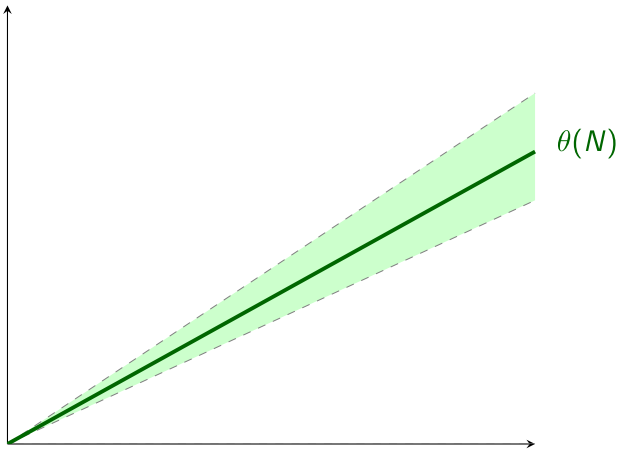
```
from random import randrange
from datetime import datetime

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}")
```



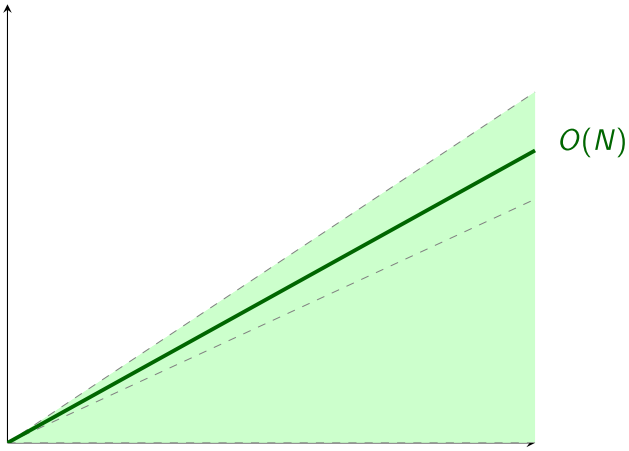
Graph 3: Scaling the `list` lookup.

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 ($O(N)$), but the **best case** (bottom of the triangle) looks a lot more like a flat line, or the **constant** complexity class. The runtime is *at least* constant, and *at most* linear: We can **bound** the runtime of the function between two complexity classes.

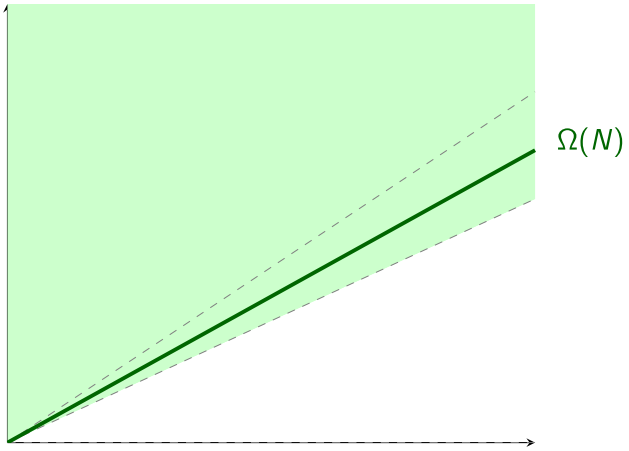
We capture this intuition of bounds through two additional concepts: the worst-case (Big-O) bound, and the best case (Big-Ω) bound. Take the linear complexity class as an example:



We write $O(N)$ to mean the set of all mathematical functions including those in $\Theta(N)$, as well as all lesser (slower-growing) complexity classes.



We write $\Omega(N)$ to mean the set of all mathematical functions including those in $\Theta(N)$, as well as all greater (faster-growing) complexity classes.



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.

1.4.1. Formalizing Bounds

1.4.2. Tight Bounds