1 A numerical tug of war

The worst-case number of elementary steps of an algorithm is a precise function of its parameters. For instance, sorting \( n \) numbers with a particular sorting program has a worst-case number of elementary steps for each value of \( n \), so it is a legitimate function.

It would be hard to figure out what this exact function is, as that would depend on the compiler and the computer you are running it on. The problem is even more difficult when we are talking about an algorithm in abstract terms without yet giving a program that implements it or specifying the computer.

Big-O analysis, also known as asymptotic analysis, allows us to get into the “ballpark”. It is an analysis that groups large sets of functions into ballparks, and when two functions aren’t in the same ballpark, it tells which one is decisively faster growing. The ballparks are known as equivalence classes.

It’s hard to figure out the exact number of steps that an algorithm takes in the worst case. If we try, we get an ugly polynomial, such as the one on page 27 of the text, and even then, the constants in the expression reflect engineering considerations that haven’t been settled until we write out a full program and specify the compiler and the machine. It’s often easy to figure out what equivalence class it falls into, however. When the running times of two algorithms for a problem are in different equivalence classes, it gives us a decisive reason for claiming that one algorithm is superior, independently of implementation details, at least for large instances of the problem.

To compare two functions \( f(n) \) and \( g(n) \), we will look at what happens to \( f(n)/g(n) \) as \( n \) goes to infinity. We can think of \( f(n) \) as trying to pull the ratio to infinity and \( g(n) \) as trying to pull it to zero.

We’ll say that functions \( f(n) \) and \( g(n) \) are in the same equivalence class if, as \( n \) gets large, the ratio \( f(n)/g(n) \) settles down to a region of the y axis that lies between two positive constants, \( c_1 \leq c_2 \). In class, I used the notation \( f(n) =\) \( g(n) \) to express this.

**Example:** \( f(n) = n^2 \), \( g(n) = 3n^2 - 7n + 5 \). The ratio \( f(n)/g(n) \) gets closer and closer to 1/3 as \( n \) gets large. For \( n = 100 \), \( f(n) = 10,000 \) and \( g(n) = 30,000 - 700 + 5 \). The ratio is 0.34124. From then on out, it never gets any farther from 1/3. So we can pick, say, \( c_1 = 0.1 \) and \( c_2 = 5 \), and claim that for \( n > 100 \), \( f(n)/g(n) \) is always between \( c_1 \) and \( c_2 \). Therefore, \( f(n) =\) \( g(n) \). They are in the same equivalence class.

We could say that \( f(n) <\) \( g(n) \) if the limit of \( f(n)/g(n) \) as \( n \) goes to infinity is 0. That means that no matter how small a positive constant we choose for \( c_1 \), the ratio eventually drops under \( c_1 \) and stays under it, defeating any attempt to show that they are in the same equivalence class.
• **Example:** \( f(n) = 100n, g(n) = n^3 \). Then the limit of \( f(n)/g(n) = 100/n^2 \) is 0. If we pick \( c_1 \) to be 1/10,000, then for \( n > 1000 \), the ratio will lie below \( c_1 \). No matter how small a positive constant we pick for \( c_1 \), the ratio will eventually go below it and stay below it.

We could will say that \( f(n) \sim g(n) \) if the limit of \( f(n)/g(n) \) as \( n \) goes to infinity is infinity. That means that no matter how large a \( c_2 \) we pick, the ratio will eventually go above \( c_2 \) and stay there, defeating any effort to show they are in the same equivalence class.

• **Example:** \( f(n) = n^3, g(n) = 100n \). Notice that we have swapped the roles of \( f(n) \) and \( g(n) \) from the previous example, so, since \( f(n) \) was “less than” \( g(n) \) in the previous example, it should now be “greater than” \( g(n) \). The ratio is \( n^2/100 \). If we pick \( c_2 \) to be something large, such as 10,000, then for \( n > 1000 \), the ratio is greater than 10,000. No matter how large a constant we pick for \( c_2 \), it will suffer the same fate, defeating any attempt to show that they are in the same equivalence class.

There is a fourth case, which is none of the above three, illustrated by the exotic example of \( f(n) = n \) and \( g(n) = n^{1+\sin n} \). We could call this a “forfeit.” It’s caused by one of the players in the contest behaving in an erratic way that we won’t ever see in the worst-case performance of an algorithm. That is the last time in the course that we’ll see the fourth case.

### 2 Notation

For historical reasons, we use will use the following notation, which we have not choice but to memorize.

- \( f(n) \sim g(n) \) is denoted \( f(n) = o(g(n)) \)
- \( f(n) \ll g(n) \) is denoted \( f(n) = \Theta(g(n)) \)
- \( f(n) \sim g(n) \) is denoted \( f(n) = \omega(g(n)) \)
- \( f(n) \leq g(n) \) is denoted \( f(n) = O(g(n)) \). This says, “\( f(n) \) is “not faster growing” than \( g(n) \). It might be in the same equivalence class or it might be slower growing; we aren’t taking a stand.
- \( f(n) \geq g(n) \) is denoted \( f(n) = \Omega(g(n)) \). It might be in the same equivalence class or it might be faster growing.

Notice that \( f(n) = o(g(n)) \) if and only if \( g(n) = \omega(f(n)) \), \( f(n) = \Theta(g(n)) \) if and only if \( g(n) = \Theta(f(n)) \), and \( f(n) = O(g(n)) \) if and only if \( g(n) = \Omega(f(n)) \).

**Examples:**

- \( 3n^2 - 5n + 24 = O(n^3) \);
- \( 3n^2 - 5n + 24 = \Omega(n^2) \);
- \( 3n^2 - 5n + 24 = \Theta(n^2) \);
• $3n^2 - 5n + 24 = o(n^3)$;
• $3n^2 = 5n + 24 \neq \omega(n^2)$;
• $3n^2 - 5n + 24 \neq o(n^2)$;

The usual convention is to put a complicated function, which is often the worst-case running time of an algorithm, on the left, and a simple expression inside the $O(\cdot)$, $\Omega(\cdot)$, $\Theta(\cdot)$, $o(\cdot)$, or $\omega(\cdot)$ expression.

**Example:** The worst-case running time of Mergesort, when run on a list of size $n$, is $\Theta(n \log_2 n)$.

The expression $f(n) = \Theta(g(n))$ is the most specific of these relationships. Why not always use it? Sometimes, we can prove that $f(n) = O(g(n))$, but we don’t know how to prove $f(n) = \Omega(g(n))$, hence we can’t claim $f(n) = \Theta(g(n))$. Sometimes, we are only interested in showing upper (big-O) bounds on the running time of an algorithm, and we don’t want to bother with showing lower (\Omega()) bounds.

### 3 Tricks we can get away with if we are doing asymptotic analysis

When comparing $f(n)$ and $g(n)$ in a class in algebra, you can only change the way you express functions if it doesn’t change their value anywhere. That’s because standard algebra is concerned with the exact values of things. For example, $f(n) = (n + 2)(n + 1)$ can be rewritten as $f(n) = n^2 + 3n + 2$, but not as $f(n) = n^2$. There is no notion of “ballparks.” The algebraic identities tell you what you can get away with.

When comparing $f(n)$ and $g(n)$ in our big-$\Theta$ analysis, it is legitimate to transform them in any way that can’t affect their big-$\Theta$ bounds. This allows us a lot of tricks for simplifying them that wouldn’t be allowed if we were doing an exact analysis. They would get you in trouble with your teacher in middle school, who would think you are lazy and sloppy. They make life really easy, yet they don’t allow people to cheat when deriving a time bound.

Here are some examples. Convince yourself that they can’t affect the outcome of the tug of war if we only apply them a constant number of times.

• Increase $f(n)$ by an additive constant. Same with $g(n)$.
• Multiply $f(n)$ or $g(n)$ by a constant that does not depend on $n$.
• When $f(n)$ has two terms, $f'(n)$ and $f''(n)$ and $f'(n) = O(f''(n))$, we can throw out the $f'(n)$ term. Same with $g(n)$.

**Example:** $f(n) = 2n^3 + 3n^3 + 4n^2 + n + 5$, we can throw out all but the first term.

**Other key facts:**

• $\log_b n = \Theta(\log_c n)$ for all $b, c > 1$. This follows from the fourth identify of 3.15 on page 56: $\log_b n = \log_c n / \log_c b$, and $\log_c b$ is a constant factor. This means that we don’t need to specify the base of a log when we put it inside a big-O expression, big-$\Theta$ expression, little-o expression, etc.
• $n^k = o(c^n)$ for all $k \geq 0, c > 1$. 

3
\begin{itemize}
\item $\log^k n = o(n^\epsilon)$ for all $k, \epsilon > 0$. It doesn’t matter how large $k$ is or how small $\epsilon$ is; the latter will eventually overtake the former and leave it in the dust.

Here is a proof: let $x = \log_2 n$. Then $\log^k n = x^k$, and $n^\epsilon = (2^{\log_2 n})^\epsilon = 2^{x \log_2 n} = 2^x$. (See the identities on pages 55-56.) Since $x$ goes to infinity as $n$ does, the outcome of the comparison between $\log^k n$ and $n^\epsilon$ will be the same as the outcome of the comparison between $x^k$ and $2^x$, and $x^k = o(2^x)$ by the previous fact.
\end{itemize}

\section{Summations}
A function is expressed in \textit{closed form} if it is not expressed recursively or as a summation with a variable number of terms. If we want to compare one running time to others, it is convenient to get it into closed form.

\subsection{4.1 Geometric series}
When $f(n) = \sum_{i=0}^{n} r^i$, where $0 < r$ and $r \neq 1$, the summation is called a \textit{geometric series}. The ratio of each pair of consecutive terms is a constant, $r$.

\begin{itemize}
\item Here’s a convenient trick: for bounding a geometric series, we can throw out all but the largest term of the summation and put a big-$\Theta$ around it.
\end{itemize}

\textbf{Example:} $1 + 2 + 4 + 8 + 16 + 32 + 64 = 2 \times 64 - 1$. $1 + 2 + 4 + 8 + 16 + 32 + 64 + 128 = 2 \times 128 - 1$. The summation is always just shy of two times the largest term. Throwing out all but the largest term, $2^n$, of the summation $\sum_{i=1}^{n} 2^i$ diminishes it by less than a factor of two, so it doesn’t change its equivalence class. The summation is $\Theta(2^n)$.

\textbf{Example:} $1 + 1/2 + 1/4 + 1/8 + \ldots + 1/2^n < 2$, so throwing out all but the largest term, 1, diminishes the summation by less than a factor of two. The summation is $\Theta(1)$.

\textbf{Proof:} We will first show using mathematical induction that $f(n) = \sum_{i=0}^{n} r^i = (r^{n+1} - 1)/(r - 1)$. If $n = 0$ the sum is 1 and $(r^{0+1} - 1)/(r - 1) = 1$. Therefore, if the equality fails for some $n$, it must fail at a smallest $n$. Let $k$ be this $n$. Because the equality holds at $n = 0$, $k > 0$. $\sum_{i=0}^{k-1} r^i = \sum_{i=0}^{k} r^i = \sum_{i=0}^{k-1} r^i + r^k$. Since $k$ is the smallest value where the identity fails, $\sum_{i=0}^{k-1} r^i = (r^{k-1+1} - 1)/(r - 1) = (r^k - 1)/(r - 1)$. Therefore, $f(n) = (r^k - 1)/(r - 1) + r^k = (r^k - 1 + r^{k+1} - r^k)/(r - 1) = (r^{n+1} - 1)/(r - 1)$, contradicting our supposition that there is a smallest $n$ where equality doesn’t hold.

Now that we have our summation in closed form, let’s simplify. Suppose $r > 1$. Let’s mess with $(r^{n+1} - 1)/(r - 1)$ in ways that are allowed in asymptotic analysis. We can multiply it by the constant $(r - 1)$ to change it to $r^{n+1} - 1$. We can add the constant 1 to change it to $r^{n+1}$. We can divide it by the constant $r$ to change it to $r^n$, which is equal to the biggest term. If it ties with the biggest term after these changes, it must be that it tied initially with it. Therefore, $\sum_{i=0}^{n} r^i = (r^{n+1} - 1)/(r - 1) = \Theta(r^n)$, and the claim holds.

Suppose $0 < r < 1$. Let’s get both the numerator and denominator positive, and rewrite it as $(1 - r^{n+1})/(1 - r)$ This gets nearer and nearer to $1/(1 - r)$ as $n$ gets large, since $r^{n+1}$ gets smaller and smaller. We can now multiply this by the constant $(1 - r)$, and it is 1. If it was $\Theta(1)$ after these steps, it must be that $\sum_{i=0}^{n} r^i = (r^{n+1} - 1)/(r - 1) = \Theta(1)$, and 1 is the largest term of the summation.
4.2 Hurting your case

The summation \( f(n) = \sum_{i=1}^{n} i \) is not a geometric series, since the ratio of consecutive terms is not a constant \( r \). The difference between consecutive terms is a constant. This is called an arithmetic series. We can’t throw away all but the biggest term without changing its equivalence class.

Here’s a proof that it’s \( \Theta(n^2) \). Our strategy is to show first that it’s \( O(n^2) \) and then show that it’s \( \Omega(n^2) \). These two facts together show that it’s \( \Theta(n^2) \), just as \( i \geq 5 \) and \( i \leq 5 \) imply that \( i = 5 \).

To show it’s \( O(n^2) \), we can transform it into a larger function, since this can only hurt our case. A skeptic won’t object. If we can show that it is still \( O(n^2) \) after the transformation, then we’ve proved that the original is \( O(n^2) \). The trick is not to hurt our case so much that we change its equivalence class. Let’s boost up all the terms of \( \sum_{i=1}^{n} i \) to \( n \). We now have \( n \) terms, each of which is \( n \), so our new sum is \( O(n^2) \). We showed the bound in spite of hurting our case. It must be that the original is \( O(n^2) \).

To show it’s \( \Omega(n^2) \), we can transform it into a smaller function, since this can only hurt our case. Let’s throw away the first \( \lceil n/2 \rceil \) terms. There are at least \( n/2 \) remaining terms, each of which is at least \( n/2 \). Then let’s cut each of the remaining terms down to \( n/2 \). No term has been increased, so this can only hurt our case. What is left is at least \( n/2 \times n/2 = n^2/4 = \Omega(n^2) \).

- We have shown that \( \sum_{i=1}^{n} i = \Theta(n^2) \).

**Another example:** \( f(n) = \sum_{i=1}^{n} \log_2 i \). The largest term is \( \log_2 n \). For an upper bound, nobody will object if we change all the terms to \( \log_2 n \). That could only hurt our case when trying to show an upper bound. It becomes \( \sum_{i=1}^{n} \log_2 n = n \log_2 n \). Therefore, \( f(n) = O(n \log n) \).

For a lower bound, suppose \( n \) is even, since nobody would object if we throw out the last term and \( n' = n + 1 \). \( n' = \Theta(n') \) for \( n' = n + 1 \). (We can get away with a lot of simplifying steps when we’re working toward a big-\( \Theta \) bound.) Nobody will object if we throw out the terms that are smaller than \( \log_2 (n/2) \). That could only hurt our efforts to still get our lower bound.

Our summation is now \( \sum_{i=n/2}^{n} \log_2 i \). The smallest of the terms in this expression is \( \log_2 (n/2) = (\log_2 n) − 1 \). Nobody will object if we change all of the terms to be equal to the smallest. This gives \( \sum_{i=n/2}^{n} ((\log_2 n) − 1) = (n/2 + 1)((\log_2 n) − 1) = \Omega(n \log n) \).

- We have shown that \( f(n) = O(n \log n) \) and \( f(n) = \Omega(n \log n) \), so \( f(n) = \Theta(n \log n) \).

4.3 Grouping terms into buckets

The summation \( \sum_{i=1}^{n} 1/i \) is the harmonic series. It turns out that this one is \( \Theta(\log n) \).

Here’s a proof. Let’s divide the terms into buckets, where the first term goes into the first bucket, the next two terms go into the second bucket, the next four terms go into the third, the next eight go into the fourth. That is the first bucket is \( \{1\} \), the second is \( \{1/2, 1/3\} \), the third is \( \{1/4, 1/5, 1/6, 1/7\} \), etc. We’ll end up with \( O(\log n) \) buckets, since we can double at most \( \lfloor \log_2 n \rfloor \) times before reaching \( n \).

To show that the sum is \( O(\log n) \), let’s observe that the largest element of the first bucket is at most 1, the largest element of the next bucket is 1/2, the largest of the next is 1/4, et. Let’s
replace all the elements in each bucket with the biggest element in the bucket. This can only hurt
our case. Now the elements in the first bucket sum to $1 \times 1 = 1$, the elements in the second bucket
sum to $2 \times 1/2 = 1$, the elements in the third sum to $4 \times 1/4 = 1$, etc. There are $O(\log n)$ buckets,
each of which sums to at most 1, so our terms all sum to $O(\log n)$.

Let’s now show that it is $\Omega(\log n)$. If there is an incomplete bucket at the end, let’s throw its
terms out, since this can only hurt our case. It is easy to see that last complete bucket is at least
$1/2 \times \log_2 n$. Therefore, the number of complete buckets is $\Omega(1/4 \log_2 n) = \Omega(\log n)$. If the last
element in a bucket is $1/i$, let’s replace all elements in the bucket with $1/(i + 1)$. This reduces the
size of all elements in the bucket, so it can only hurt our case.

The first bucket now has one term equal to $1/2$, so it sums to $1/2$. The second bucket has two
terms equal to $1/4$, so it sums to $1/2$. The third bucket has four terms equal to $1/8$, so it sums to
$1/2$. The sum is $1/2 \Omega(\log n) = \Omega(\log n)$. 
5 A peek at Python

Type the following into a file called mySort.py:

```python
print ('I’m starting.

def InsertionSort(l):
    if len(l) == 1:
        return l
    else:
        l2 = InsertionSort(l[:-1])
        t = l[-1]
        return [i for i in l2 if i <= t] + [t] + [i for i in l2 if i > t]

print ('I’m existing.

Then, from the Linux command line, type ipython -i mySort.py. The file will run the program, and the -i prompt tells python to give you the command line after it runs. The only thing the program does is to print two output statements and define a method. It doesn’t actually run the method; methods are defined with a command, def.

The -i option says to give you the Python command line after it runs. Type this to the Python command line: InsertionSort([5,3,6,4,2]). You will see that it returns a list that has the same elements, in sorted order.

Study the Python tutorial to find out about the non-Java-like statements you see. Find out how lists can be represented with the square-bracket syntax. Find out about the len operator. Find out what it means when l is a list and you write l[3:6]. Experiment by getting the Python command line and doing the following:

In [2]: l = [6,9,2,1,3,4,7,8]
In [3]: l[3:6]
Out[3]: [1, 3, 4]
In [4]: l[0:2]
Out[4]: [6, 9]
In [5]: l2 = [100,150]
In [6]: l + l2
Out[6]: [6, 9, 2, 1, 3, 4, 7, 8, 100, 150]
In [7]: l
Out[7]: [6, 9, 2, 1, 3, 4, 7, 8]
In [8]: l2
Out[8]: [100, 150]
```
In [5]: l2 = [100,150]
In [9]: [i for i in l if i >= 4]
Out[9]: [6, 9, 4, 7, 8]
In [10]: l
Out[10]: [6, 9, 2, 1, 3, 4, 7, 8]

Before long, you should study the entire tutorial, trying out all of the examples at the Python prompt as you go through it.

6 Getting a recurrence for the running time of a recursive algorithm

Here are some facts about Python operations. Producing the concatenation of two lists takes time proportional to the sum of sizes of the lists. Creating a sublist of those elements of a list that meet a condition that takes $O(1)$ time to evaluate takes time proportional to the length of the list. To create a list of elements between two indices takes time proportional to the number of elements in this sublist.

Let’s think of a name for the worst-case running time of our Python method on a list of size $n$: we’ll call it $T(n)$. We would like to get a big-$\Theta$ bound for $T(n)$.

If $n > 1$, it makes a recursive call on $n - 1$ elements. We don’t know how long this takes, but we have a name for it: $T(n - 1)$. It also does a fixed number $k$ of operations that each take $\Theta(n)$ time. The total for these is $k\Theta(n) = \Theta(n)$ since $k$ is fixed. In the friendly world of big-$O$, we didn’t even need to bother counting what $k$ is!

If $n == 1$, it takes $\Theta(1)$ time.

We get the following recursive expression for the running time:

$$T(1) = \Theta(1); \quad T(n) = T(n - 1) + \Theta(n).$$

Since we’re doing big-$\Theta$ analysis, there is no harm in fudging this and just calling it this, since it can only affect $T()$ by a constant factor:

$$T(1) = 1; \quad T(n) = T(n - 1) + n.$$

If $n - 1 > 1$, we can substitute $T(n - 1) = T(n - 2) + n - 1$ into this expression, and get

$$T(n) = T(n - 2) + n - 1 + n.$$

If $n - 2 > 1$, we can substitute $T(n - 2) = T(n - 3) + n - 2$ into this expression, and get

$$T(n) = T(n - 3) + n - 2 + n - 1 + n.$$

Eventually, we will get down to $T(1)$, and we can substitute in $T(1) = 1$. We get $T(n) = 1 + 2 + 3 + \ldots + n$. We’ve turned our recursive expression for the running time into a summation. It’s still not in closed form, but it’s progress.

- We analyzed this summation above, and found out it was $\Theta(n^2)$. The running time of our Python program must be $\Theta(n^2)$.
7 Using the master theorem

Study how the book derives a recurrence for mergesort. It’s similar to what we have just done above.

The master theorem is a recipe that often works for getting a big-Θ bound on a recurrence of the form $T(n) = aT(n/b) + f(n)$. The recurrence for mergesort is similar.

The master theorem says that if the ratio $n^{\log_b a}/f(n)$ is $\Omega(n^\epsilon)$ for some $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$. If the inverse of this ratio is $\Omega(n^\epsilon)$, then $T(n) = \Theta(f(n) \log n) = \Theta(n^{\log_b a} \log n)$.

The second condition on case 3 of the master theorem applies to all recurrences we will see this semester. It is there as a disclaimer against skeptics who could otherwise show it is false by tripping it up with an exotic $f(n)$. I will explain what this is about in class.

Examples:

- $T(n) = 2T(n/2) + n$. We’re comparing $n^{\log_2 2} = n^1$ vs. $f(n) = n$. They’re big-Θ of each other. $T(n) = \Theta(n \log n)$.

- $T(n) = 4T(n/2) + n$. We’re comparing $n^{\log_2 4} = n^2$ vs. $f(n) = n$. The ratio is $n^\epsilon$ for $\epsilon = 1$. $T(n) = \Theta(n^2)$.

- $T(n) = 2T(n/2) + n^{1/2}$. We’re comparing $n^1$ vs $n^{1/2}$. The ratio is $n^\epsilon$ for $\epsilon = 1/2$. $T(n) = \Theta(n)$.

- $T(n) = 2T(n/2) + n^2$. We’re comparing $n^1$ vs. $n^2$. The ratio is $1/n^1$. $T(n) = \Theta(n^2)$.

- $T(n) = 2(n/2) = n \log n$. $n^1$ vs $n \log n$. The ratio is $1/\log n$. However, since $\log n = o(n^\epsilon)$ for all $\epsilon > 0$, the master theorem says you have to find another method. It punts. In this case, we have to expand the recurrence by iterative substitution, the way we did for the recurrence for insertion sort above.