1 More analysis terminology

We know that $O(n \rightarrow n)$ is the set of all functions from $\mathbb{N}$ to $\mathbb{N}$, and we can write $f \in O(g)$ to indicate that the function $f$ grows no faster than $g$ (at least up to constant multiples). But while the function $n \mapsto 1$ grows no faster than the function $n \mapsto n$, we have the sense that it’s substantially smaller, not just “off by a factor of 2 or 3,” for instance. That’s confirmed by our knowledge that $n \mapsto n \notin O(n \mapsto 1)$. Fortunately, just as big-O captures the idea that one function grows slower than another, there’s another notation that means the opposite: we define $\Omega(f)$ as the set of all functions that eventually dominate $f$, up to constants. In symbols,

$$f \in O(g) \iff g \in \Omega(f).$$

So because $n \mapsto 1 \in O(n \mapsto n)$, we can write

$$n \mapsto n \in \Omega(n \mapsto 1).$$
Informally, $\Omega(f)$ consists of functions that grow at least as fast as $f$.

By contrast, the functions $n \mapsto n$ and $n \mapsto 2n + 1$ really do seem to be “the same size”; that’s reflected by the fact that $O(n \mapsto n) = O(n \mapsto 2n + 1)$. There’s a notation for that as well. We write

$$f \in \Omega(g)$$

to mean that $f \in O(g)$ and $f \in \Omega(g)$, or, equivalently, that $f \in O(g)$ and $g \in O(f)$, or, also equivalently, that $O(f) = O(g)$. So, $f$ is eventually less than $g$ up to constants, and $g$ is eventually less that $f$ up to constants. More informally, $\Theta(f)$ consists of functions that grow the same way as $f$.

Most of the analysis we will be doing in this class will be proving our functions lie in Big-O of something, not Big-$\Omega$ or Big-$\Theta$. This is because, it is quite hard to accurately say how much time a function will “at least” take to run. Take for example, the contains17? procedure. In the case where 17 is the first item in the list, we are done in constant time! So Big-$\Omega$ will just be $\Omega(1)$.

Next consider the length procedure. Our conjecture, though operation counting and plug and chug was:

$$L(n) = B(n) + A$$

We can prove that $L \in O(n \mapsto n)$ by choosing $M = 1, c = A + B$. It is also true that the function $n \mapsto n$ is in $O(L)$, by choosing $M = 1, c = \frac{1}{B}$. So the runtime of the procedure is not just “at worst linear time”, but is in fact “linear time”! This is true for the length procedure because no matter what the list is, we need to go through every item to find the length. So we can say $L \in \Theta(n \mapsto n)$.

People often speak of big-O pretty informally, and say things like “Sorting is $O(n \mapsto n \log n)$”, when what they really mean is “A reasonably efficient sort algorithm can be written that takes time $n \mapsto n \log n$, and you can’t find one that runs any faster than that.” In short, they mean that the op-counting function for a decent sorting algorithm is in $\Theta(n \mapsto n \log n)$. Beware of this usage. It’s sloppy and wrong and misleading to others.

## 2 Theorems for proving Big-O class

We will now prove some useful theorems that will help you find the Big-O category of your conjectures.

### 2.1 Basic Theorems

Firstly, let us prove that any function $f : \mathbb{N} \mapsto \mathbb{N}$, is in its own Big-O category.

**Theorem 1.** For any function $f : \mathbb{N} \mapsto \mathbb{N}, f \in O(f)$

**Proof.** Pick $M = 1, c = 1$

Suppose that $n > M$. [completely unnecessary in this case]

I claim that $f(n) \leq cf(n)$ for $n > M$

Rewrite: $f(n) \leq 1f(n)$ for $n > M$

Therefore, our claim is true. $\diamond$
Next, we prove that a constant function with value 0 is in the Big-O class of any function \( f : \mathbb{N} \mapsto \mathbb{N} \).

**Theorem 2.** For any function \( f : \mathbb{N} \mapsto \mathbb{N} \) and \( Z : \mathbb{N} \mapsto \mathbb{N} : n \mapsto 0, Z \in O(f) \)

**Proof.** Pick \( M = 1, c = 1 \)
Suppose that \( n > M \). [again, unnecessary in this case]
I claim that \( Z(n) \leq cf(n) \) for \( n > M \)
Rewrite: \( 0 \leq 1f(n) \) for \( n > M \)
Therefore, our claim is true. \( \diamond \)

We prove that for a function \( f : \mathbb{N} \mapsto \mathbb{N}, 2f \) is in the Big-O class of the function \( f \).

**Theorem 3.** For any function \( f : \mathbb{N} \mapsto \mathbb{N}, 2f \in O(f) \)

**Proof.** Pick \( M = 1, c = 2 \)
Suppose that \( n > M \). [still unnecessary in this case]
I claim that \( (2f)(n) \leq cf(n) \) for \( n > M \)
Rewrite: \( (2f)(n) = 2f(n) \leq 2f(n) \) for \( n > M \)
Therefore, our claim is true. \( \diamond \)

Next, we prove that for a function \( f : \mathbb{N} \mapsto \mathbb{N}, \) any positive multiple of \( f \) is in the Big-O class of the function \( f \).

**Theorem 4.** For any function \( f : \mathbb{N} \mapsto \mathbb{N}, \) and a number \( a > 0, af \in O(f) \)

**Proof.** Pick \( M = 1, c = a \)
Suppose that \( n > M \). [still unnecessary in this case]
I claim that \( (af)(n) \leq cf(n) \) for \( n > M \)
Rewrite: \( (af)(n) = af(n) \leq af(n) \) for \( n > M \)
Therefore, our claim is true. \( \diamond \)

For our next theorem, we want to prove that if a function \( f \) is in the Big-O class of a function \( g \), which is in turn in the Big-O class of a function \( h \), then \( f \) is also in the Big-O class of function \( h \).

**Theorem 5.** If \( f \in O(g), g \in O(h), \) then \( f \in O(h) \)

Let us first try confirming this for a specific case with concrete values.
Suppose that (1), for \( n > 100 \), we know that \( f(n) \leq 22g(n) \).
And (2), for \( n > 33 \), we know that \( g(n) \leq 4h(n) \).
Then, from (2), we know that for \( n > 100 \), \( g(n) \leq 4h(n) \).
What can we say (for \( n > 100 \)), about the relationship of \( f(n) \) to \( h(n) \)?
\[
\begin{align*}
f(n) &\leq 22g(n) \\
g(n) &\leq 4h(n) \\
22g(n) &\leq 4 \times 22h(n) \\
f(n) &\leq 4 \times 22h(n)
\end{align*}
\]
And we have proved that our claim holds true for these values! Is it possible to generalize this proof to work for any values of \( n \)? If you observe carefully, our proof doesn’t really require our values to be these exact numbers and will also work in the general case!
Proof. For $n > M_1$, we have $f(n) \leq c_1 g(n)$.
For $n > M_2$, we know that $g(n) \leq c_2 h(n)$.
Then, for $n > \max(M_1, M_2)$, we have $f(n) \leq c_1 c_2 h(n)$.
Hence, $f \in O(h)$.
To summarize, if $[M_1, c_1]$ and $[M_2, c_2]$ show that $f \in O(g)$ and $g \in O(h)$, then $\max(M_1, M_2), c_1 c_2$ shows that $f \in O(h)$. ◦

Trying out a proof with concrete values as first can be a very useful technique in approaching these proofs. Often, some kind of pattern will emerge which will help you write the general case.

Now, let’s prove that for two functions $f$ and $g$, which are in $O(h)$, the function $f + g$ is also in $O(h)$.

**Theorem 6.** If $f \in O(h)$ and $g \in O(h)$, then $f + g \in O(h)$

Let’s try an example again,

For $n > 15$, assume $f(n) \leq 11 \times h(n)$.
For $n > 37$, assume $g(n) \leq 6 \times h(n)$.
Then for $n > \max(18, 37)$, $(f + g)(n) \leq 17h(n)$.
We can observe that here, $17 = 11 + 6 = c_1 + c_2$. We can now write the general proof,

Proof. For $n > M_1$, we have $f(n) \leq c_1 h(n)$.
For $n > M_2$, we know that $g(n) \leq c_2 h(n)$.
Then, for $n > \max(M_1, M_2)$, we have $f(n) \leq (c_1 + c_2)h(n)$.
Hence, $f + g \in O(h)$.
To summarize, if $[M_1, c_1]$ and $[M_2, c_2]$ show that $f \in O(h)$ and $g \in O(h)$, then $\max(M_1, M_2), c_1 + c_2$ shows that $f + g \in O(h)$. ◦

2.2 Bootstrapping lemma

We now use the following lemma, an important result which will be very useful in proving other interesting Big-O properties.

**Theorem 7.** Suppose $f \in O(g)$ and we define $F(n) = nf(n), G(n) = ng(n)$, then $F \in O(G)$.

Proof. Hint: use the same $[M, c]$ pair!
$f \in O(g)$ means for some $M$ and $c$, we have for $n > M$,
$f(n) \leq C \times g(n)$.
$n \times f(n) \leq C \times n \times g(n)$.
Hence, $F(n) \leq C \times G(n)$, and our claim is proved. ◦

Now we can use this lemma to prove other properties:
Let $U(n) = 1$ for all $n$.
Let $L(n) = n$ for all $n$.
Then $U \in O(L)$, which we can write as $(n \mapsto 1) \in O(n \mapsto n)$.
Bootstrapping tells us that $(n \mapsto n) \in O(n \mapsto n^2)$.
And also that $(n \mapsto n^2) \in O(n \mapsto n^3)$.
Hence, \((n \mapsto 1) \in O(n \mapsto n^3)\).
Continuing this, we see that \((n \mapsto n^k) \in O(n \mapsto n^p)\) for any natural numbers \(k, p\) with \(k \leq p\).

Using the theorem that “If \(f \in O(h)\) and \(g \in O(h)\), then \(f + g \in O(h)\)” , we get,
\((n \mapsto n^2 + n) \in O(n \mapsto n^2)\), because \((n \mapsto n^2) \in O(n \mapsto n^2)\) and \((n \mapsto n) \in O(n \mapsto n^2)\)
In fact, for any polynomial \(p\) of degree \(k\) (where \(k \in \mathbb{N}\)) we have that \((n \mapsto p(n)) \in O(n \mapsto n^k)\).
Because of this, we seldom say that some procedure runs in \(O(n^N \mapsto n^2 + 5)\) time, but instead we say \(O(n \mapsto n^2)\) time. We will mostly be dealing with polynomials in our analysis, but a function like \(2^n\) is much worse than any of these polynomial functions, and increases in runtime extremely quickly.

### A generalized bootstrapping lemma

**Theorem 8.** Suppose \(f \in O(g)\) and \(h \in \Omega(n \mapsto 1)\), which means that \(h(n) \neq 0\) for large enough \(n\), we have

\(n \mapsto f(n)h(n) \in (n \mapsto g(n)h(n))\)

**Proof.** Hint: use the max of the \(M\) values and the \(c\) from \(f \in O(g)\) to prove this result! ∗

### Depth of Binary Trees

Informally, the depth of a binary tree is the number of steps needed to get to the bottom of the tree.
Now let’s consider, for a binary tree with \(n\) nodes, what could be the largest possible depth?

Take the example of a tree with 5 nodes. Such a tree would look like:

Thus the largest depth a binary tree with 5 nodes can have is 5.
Now consider, what could be the least possible depth of a binary tree with \( n \) nodes? The shallowest tree with 5 nodes would look like:

![Binary Tree Diagram]

This tree has depth 3. It is not possible to have a binary tree of 5 nodes with only depth 2 because there would only be room for 3 nodes.

Now, if a binary tree has depth \( k \), what’s the largest number of nodes it can have? Below is a table with the number of nodes for values of \( k \) from 0 to 6:

<table>
<thead>
<tr>
<th>( k )</th>
<th>nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>15</td>
</tr>
<tr>
<td>5</td>
<td>31</td>
</tr>
<tr>
<td>6</td>
<td>63</td>
</tr>
</tbody>
</table>

We can think of the number of nodes in a tree as being the root, plus number of nodes in the left subtree, plus the number in the right subtree. The left and right subtrees are equivalent to a tree with depth 1 lesser than our tree, and have the same number of nodes.

![Doubled Subtrees Diagram]

So, let \( Q(k) \) be the largest number of nodes in a tree of depth \( k \). We have,
$Q(0) = 0$
$Q(k) = 1 + 2Q(k - 1)$, because in a depth $k$ tree, the left and right subtrees can each contain at most $Q(k - 1)$ nodes, and their parent (the root) accounts for one more.

We can now use plug and chug on this recursive formula we have for largest number of nodes (like the table above) to get $Q(k) \leq 2^k - 1$.

Our next question is, if we want to put $n$ nodes in a tree, how deep must it be?
Well, we can put $\frac{n-1}{2}$ nodes in each subtree.
So $D(n)$, the maximum depth of an $n$ node tree satisfies $D(n) \geq D(\frac{n-1}{2} + 1)$

This is basically asking the question: how many times do I need to divide $n$ by 2 till I get to a number less than or equal to 1?
This is (informally) the definition of $\log_2(n)$! Or rather, $\lceil \log_2(n) \rceil$, since we are taking the ceiling of this number. More formally, a mathematical definition of log would be

$$\log_2(n) = x \text{ if } 2^x = n$$

However, for us, the intuitive understanding that the number of times you need to divide a number by 2 to reach 1 is $\log_2(n)$ is very effective in helping us analyze procedures!

4 Summary

Ideas

- Big-Theta and Big-Omega can also be used describe function growth, and are often confused with Big-O.

- The depth of a tree containing $n$ nodes is approximately $\log(n)$.

Skills

- Have a toolkit of theorems to use when analyzing your own programs and determining what Big-O class they belong to.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS17 document by filling out the anonymous feedback form: [http://cs.brown.edu/courses/cs017/feedback](http://cs.brown.edu/courses/cs017/feedback)