Asymptotics and Recurrences

1 Asymptotic Analysis

It is often useful to express the work done by an algorithm as a function of the size of its input, for example,

$$W(n) = 42n^2 + 3n\lg n + 101\sqrt{n} + \frac{5}{\lg n} + 3.1415.$$

As you have probably seen in previous courses, this function is in $O(n^2)$. Big-O notation is useful in computer science because it removes the portions of functions that don't substantially affect the behavior of the function as n grows larger. In this way, asymptotic analysis is a useful abstraction because it simplifies expressions. By removing the lower-order terms, it also removes the details of things that might later change, such as the machine, program model, and compiler, which allows us to focus on the algorithm without having to worry about the implementation details. More specifically, it focuses on the impactful parts of the algorithm, while abstracting away parts of the algorithm that only change the runtime by a trivial amount.

We define asymptotic analysis in terms of domination:

Definition: Asymptotic Domination

A function $f(n)$ is said to *asymptotically dominate* another function $g(n)$ if there exist constants $c, n_0 \geq 0$ such that $g(n) \leq c \cdot f(n)$ for all $n \geq n_0$.

Example: Asymptotic Domination

- If $f(n) = 2n$ and $g(n) = n$, then $f(n)$ dominates $g(n)$. We can show that this holds by finding a satisfactory c and n_0 – in this case, $c = 1$ and $n_0 = 0$ works because $n \leq 1 \cdot 2n$ for all $n \geq 0$.
- $f(n) = n$ dominates $g(n) = 2n$ if we pick $c = 2$ and $n_0 = 0$, since $2n \leq 2 \cdot n$ for all $n \geq 0$.
- $f(n) = n\lg n$ dominates $g(n) = n$, if we pick $c = 1$ and $n_0 = 2$. In this case, picking n_0 was a bit harder, since we had to make sure that the $\lg n$ term did not make $f(n)$ smaller than $g(n)$.
- $f(n) = 2^n$ does not asymptotically dominate $2^{1.1n}$. To prove that this is the case, we have to show that for any c and n_0, there will be some $n \geq n_0$ where $g(n) > c \cdot f(n)$, which is a bit harder.

We can now define Big-O (along with its sisters big-Θ and big-Ω) in terms of domination.

Definition: Big-O, Big-Ω, Big-Θ

For any function $f(n)$, we can define three sets of functions:

$O(f(n))$ is defined as $\{g, \text{such that } f \text{ asymptotically dominates } g\}$.

$\Omega(f(n))$ is defined as $\{g, \text{such that } g \text{ asymptotically dominates } f\}$.

$\Theta(f(n))$ can then be defined as $O(f(n)) \cap \Omega(f(n))$.

We might think of $O(f(n))$ as the set of functions that are "asymptotically smaller than or equivalent to" $f(n)$, and $\Omega(f(n))$ as the set of functions that are "asymptotically bigger than or equivalent to" $f(n)$. It follows that $\Theta(f(n))$ consists of functions that are "asymptotically equivalent" to $f(n)$.

These definitions are kind of abstract. In this course the usual context in which we apply them will be to prove that for a specific pair of functions f and g that $f(n) = \Theta(g(n))$. And to do so we just show that f asymptotically dominates g (by finding an n_0 and c) and separately proving that g asymptotically dominates f (by finding another pair n_0' and c'). The process is similar for for Big-O and Big-Θ.

Example: Proving that $\log(n!) = \Theta(n \log(n))$

Proof: First note that here and throughout this course when we write "log" without specifying the base, the base is assumed to be 2.

The proof is in two parts. First we prove that $\log(n!) = O(n \log(n))$, then we prove that $\log(n!) = \Omega(n \log(n))$. For the first part expand the left hand side:

$$\begin{aligned} \log(n!) &= \log(1) + \log(2) + \cdots + \log(n) \\ &\leq \log(n) + \cdots \log(n) \\ &= n \log(n) \end{aligned}$$

This completes the first part. For the second part, just keep the terms of the right hand side involving $\log(a)$ where $a \geq \lceil \frac{n}{2} \rceil$. This gives us the following lower bound on $\log(n!)$.

$$\begin{aligned} \log(n!) &\geq \log\lceil \tfrac{n}{2} \rceil + \log\lceil \tfrac{n}{2} \rceil + \cdots + \log\lceil \tfrac{n}{2} \rceil = \tfrac{n}{2} \log\lceil \tfrac{n}{2} \rceil \\ &\geq \tfrac{n}{2} \log \tfrac{n}{2} \\ &= \tfrac{n}{2} (\log(n) - 1) \end{aligned}$$

But for $n \geq 4$ it's easy to see that $\log(n) - 1 \geq \frac{1}{2} \log n$. Thus if we pick $n_0 = 4$ and $c = 4$ then $n \log(n) \leq c \log(n!)$ for all $n \geq n_0$. QED

Sometimes it is useful to denote a form of strict domination of one function over another.

Definition: Little-o and Little-ω

The set of functions $o(f(n))$ is defined as $O(f(n)) \setminus \Theta(f(n))$, that is, the functions in $O(f(n))$ that are not in $\Theta(f(n))$.

$\omega(f(n))$ is similarly defined as $\Omega(f(n)) \setminus \Theta(f(n))$.

$o(f(n))$ is then the functions "asymptotically strictly smaller than" $f(n)$, which is also the set of functions in $O(f(n))$ for which $f(n)$ is not asymptotically tight. The same relationship is true between $\omega(f(n))$ and $\Omega(f(n))$.

Clearly if $f(n) = o(g(n))$ it follows that $f(n) = O(g(n))$. The converse is not necessarily true. It is often useful to make use of limits to prove these bounds, because of the following theorem.

Theorem: Limit Theorem for Little-o and Little-ω

For positive functions f and g, the following are equivalent:

$$\begin{aligned} f(n) &= o(g(n)) \\ g(n) &= \omega(f(n)) \\ \lim_{n\to\infty} \frac{f(n)}{g(n)} &= 0 \end{aligned}$$

Often, in this course, and indeed everywhere in the algorithms teaching and research world, we write notation such as:

$$n = O(n^2)$$

This doesn't really make sense, since n is a function and $O(n^2)$ is a set of functions! When we see this, we interpret this as meaning that $n \in O(n^2)$. Likewise, we might write:

$$f(n) = g(n) + O(n^2)$$

Interpreting this is a bit trickier. We interpret this as meaning that $f(n)$ is a set of functions, which is equal to the set of functions that can be created by adding $g(n)$ to functions in the set $O(n^2)$. This second form of notation abuse again is common all throughout the algorithms world and will show up throughout the course as we look at recurrences.

2 Recurrences

A **recurrence relation** is a recursively defined generic function – that is, a mathematical function that is defined in terms of itself. One common example of a recursive function is the Fibonacci numbers:

$$f(n) = \begin{cases} 1 & \text{if } n = 0, 1 \\ f(n-1) + f(n-2) & \text{if } n \geq 2 \end{cases}$$

By looking at the recursive definition of the Fibonacci numbers, we can see some of features of a recurrence:

- Recurrences tend to have base cases and recursive cases.
- Recurrences are good for modeling recursive functions – the recursive call(s) in the algorithm easily corresponds to a recursive term in the recurrence.
- Recurrences do not tell us about the values of the function easily, so we ideally want to convert them to a closed (non-recursive) form to evaluate them on larger inputs.

Example: Merge Sort

Consider the following code for mergesort:

```
fun mergesort(S : sequence<T>) -> sequence<T>:
  if |S| <= 1: return S
  else:
    L, R = split_mid(S)
    sL, sR = parallel (mergesort(L), mergesort(R))
    return merge(sL, sR)
```

We can define the function $W_{\text{msort}}(n)$ to be the work done by the mergesort function on an input of size n. The "if" case of the conditional takes constant time, and it occurs when $n \leq 1$. The "else" case happens otherwise. When this happens, we make two recursive calls to mergesort, each with size $n/2$, and then we need to merge the two results together. Putting this all together, we get:

$$W_{\text{msort}}(n) = \begin{cases} c_b & n \leq 1 \\ 2W_{\text{msort}}\left(\frac{n}{2}\right) + c_m * n & \text{otherwise} \end{cases}$$

Here c_b represents the cost of the base case, and $c_m * n$ represents the cost of doing the split (of a list of length n) and doing the merge (of two lists of length $\frac{n}{2}$).

In previous classes, you have probably seen that $W_{\text{msort}}(n)$ has a closed form of $O(n \log n)$. It's not obvious that this recurrence solves to $O(n \log n)$, however. Our next goal will therefore be to somehow convert recurrences to their closed forms.

As with asymptotic analysis, there are a few notational notes first:

- Instead of writing an algorithm out, we instead sometimes write its work function as a mathematical function.
- Formally, we would write $W(n) = \{\ldots$ and write out both the base cases and recursive cases. In our case, we drop any base cases that are constant time.
- We sometimes abuse notation, as before, and write things like $W(n) = 2W\left(\frac{n}{2}\right) + O(n)$. The right-hand side is, as before, a set of functions which $W(n)$ then belongs to.

3 The Tree Method

The **tree method** is a way of solving recurrences that involves visually breaking down the structure of calls and subcalls to functions, while allowing us to understand where the costs lie. To execute the tree method, we start by unfolding the recurrence level by level in a tree.

Example: Tree Method

Say we have the recurrence

$$W(n) = 2W\left(\frac{n}{2}\right) + O(n).$$

We can rewrite this recurrence as

$$W(n) = W\left(\frac{n}{2}\right) + W\left(\frac{n}{2}\right) + c_1 n + c_2.$$

It becomes clear from this that there are two recursive calls being made from the parent call, each with size $n/2$. The amount of work done at the parent is $O(n)$, which we express as $c_1 n + c_2$. We can then use this recurrence to draw a tree, where each node of the tree represents a call to the function. The root has a size of n, and has two child calls each with size $n/2$. Likewise, each of the children has two child calls of size $n/4$, and so on and so forth.

The total work done is then the sum of the work at each node of the tree. Thus, to find a closed form for this recurrence, we begin with finding the work on each level, and we will then sum up all the levels to find our answer.

At the root, we have a single function call of size n, so we get that the root level does $c_1 n + c_2$ work. The level below it has two calls of size $n/2$, so it does

$$2\left(c_1 \cdot \frac{n}{2} + c_2\right) = c_1 n + 2c_2$$

work. The level below that does $c_1 n + 4c_2$ work, and so on. In general, at a non-leaf level i, we will have $c_1 n + 2^i c_2$. There are going to be $\lg n$ levels in this tree. (Note that the leaves will have cost equal to the base case, so the leaf layer will have $2^{\lg n} \cdot c_b = n c_b$ work, where c_b is the work of the base case.)

To analyze the overall work, we sum up the work per level over all the levels:

$$\sum_{i=0}^{\lg n} c_1 n + 2^i c_2 + n c_b = c_1 n \lg n + c_2(n-1) + n c_b \in O(n \lg n)$$

The tree method therefore consists of three steps:

- Draw out the recursion tree with the work at each node.
- Calculate the work of level i in terms of i, as well as the number of levels.

- Sum the work over all levels to get the final result.

Example: More Tree Method

Consider the recurrence

$$W(n) = 2W\left(\frac{n}{2}\right) + n^2.$$

The tree has the same structure as the tree from the earlier example. The root layer (layer 0) has local cost n^2. The next layer has local cost

$$\frac{n^2}{4} + \frac{n^2}{4} = \frac{n^2}{2}$$

since it contains two calls of size $n/2$. More generally, the cost of each layer decreases by a factor of 2 each time we go down the tree by one layer.

The total cost is then

$$n^2 + \frac{n^2}{2} + \frac{n^2}{4} + \ldots$$

If this series goes on to infinity, then the sum would be $2n^2$. However, we cannot have infinitely many layers, and must eventually reach a base case. Thus, the work is bounded by $2n^2$ plus the work of the base cases. From this, we can find that the actual sum is going to be at most $2n^2 + W(\text{base})$, and therefore is in $O(n^2)$. We can ignore the base cases here because they each do a small amount of work, and the number of base cases is asymptotically dominated by the $2n^2$ term.

4 Brick Method

In the last example, we used the fact that the cost per level is geometrically decreasing to find the total work, since we can easily find the sum of a geometric series. It turns out that if the work per level is decreasing by some factor $0 < \alpha < 1$ with the work at the root being some quantity w, then the total work will be:

$$w + w\alpha + w\alpha^2 + \cdots = w(1 + \alpha + \alpha^2 + \ldots)$$

Since each term is positive and the series $1 + \alpha + \alpha^2 + \ldots$ is not infinite, its sum must be smaller than that of the infinite series $1 + \alpha + \alpha^2 + \cdots = 1/(1-\alpha)$. This means that the cost of the root will dominate the closed form.

4.1 Root Dominated

In fact, the example we were covering is that of a **root dominated** recurrence, where the work per level goes down by a constant factor on each level. A recurrence is root dominated if the local cost of the children of a node v is at most the local cost of v times a constant $0 < \alpha < 1$ for all nodes v – that is, the children of any node do a constant factor less local work than the node itself.

If this is the case, then the asymptotic cost of the overall recurrence is the same as the asymptotic cost of the root. This is because the total cost is $\text{cost}(L_0) + \text{cost}(L_1) + \cdots \leq \text{cost}(L_0) + \alpha \cdot \text{cost}(L_0) + \ldots$. Using the properties of geometric series that were just mentioned, we can conclude that the total cost will be at most

$$\text{cost}(L_0) \cdot (1 + \alpha + \alpha^2 + \ldots) = \frac{\text{cost}(L_0)}{1-\alpha}.$$

Additionally, since α is a constant, we can conclude that $1/(1-\alpha)$ is also a constant, meaning the solution to this recurrence is in $O(\text{cost}(L_0))$ – namely, the asymptotic complexity of the whole function is going to be the same as the asymptotic complexity of the root.

4.2 Leaf Dominated

In a **leaf-dominated** recurrence, the cost of each level increases by a constant factor $\alpha > 1$ as you go down the tree, meaning the costs of the levels form an increasing geometric series. This means that the most costly level is the leaf level (or lowest level). This cost dominates the other costs, meaning that the closed form for this recurrence can be found by finding the total cost of the leaves.

Fortunately, we are able to assume that the cost of each leaf is constant in most cases. In other cases, the cost of a leaf node is typically stated in a base case. We then need to multiply the cost per leaf by the number of leaves to find the total cost, and thus the closed form of the recurrence.

Example: Leaf-Dominated Recurrences

Say we have a recurrence in the form

$$W(n) = a\,W\left(\frac{n}{b}\right) + \ldots$$

The recursion tree for this recurrence has a root node with size n, which has a child nodes of input size n/b. The constant a is called the **branching factor** of this recurrence.

There are going to be about $\log_b n$ levels in this tree. On level 1, there are a nodes. Level 2 has a^2 nodes, and in general, level i has a^i nodes. This means that the bottom layer has $a^{\log_b n}$ nodes, which is conveniently equal to $n^{\log_b a}$ by the below theorem.

This is a fairly common form of recurrence. If the non-recursive work on each level leads to the recurrence being leaf-dominated, then we know that the closed-form solution will have $n^{\log_b a}$ leaves. Multiplying this by the cost per leaf – which is often $O(1)$ – will give the final answer.

Theorem

For all real a, b, n where $a > 0, n > 0, b > 0, b \neq 1$,

$$a^{\log_b n} = n^{\log_b a}.$$

Proof. Take $\log_b$ of both sides. □

Unfortunately, not all leaf-dominated recurrences come in this form, and counting leaves can become quite difficult.

Example: Leaf-Dominated Recurrences II

Consider the recurrence $W(n) = W\left(\frac{n}{2}\right) + W\left(\frac{n}{3}\right) + O(\sqrt{n})$.

This recurrence is going to be leaf-dominated, because a parent will do $\sqrt{n}$ work, while its children will do $\sqrt{n/2} + \sqrt{n/3}$ work which is more than $\sqrt{n}$.

If we draw out the tree, we unfortunately see that there isn't an easy way to count the leaves. They're not all on the same layer, which means that our previous counting method isn't going to be sufficient here. Instead, we're going to have to write a recurrence relation to represent the number of leaves in a tree when the root has size n:

$$L(n) = L\left(\frac{n}{2}\right) + L\left(\frac{n}{3}\right)$$

We assume that the base cases come when $n \leq 1$, so $L(n) = 1$ when $n \leq 1$. Now, we need to solve this recurrence, which looks somewhat like $W(n)$ without the $\sqrt{n}$ non-recursive term. To solve for $L(n)$, we need further techniques such as the substitution method.

4.3 Balanced

In a **balanced** recurrence, the cost of each level is approximately the same, and does not increase or decrease by a constant factor when going down the tree. It is therefore not sufficient to consider only one layer of the tree when finding the total cost, since every layer contributes an equal amount. Instead, what we must do when dealing with a balanced recurrence is first find the asymptotic cost of one layer, then multiply it by the total number of layers. This ensures that every layer of nodes is accounted for in our final sum.

Example: Balanced Recurrence

Consider the recurrence

$$W(n) = 3W\left(\frac{n}{3}\right) + n.$$

The work done by the parent on an input of size n is n. There are 3 children, each of which do $n/3$ work. Thus, the total work of the children is n. However, the parent's work of n and the childrens' work of n are not separated by an increasing or decreasing constant factor – thus, the recurrence is balanced!

As a result, we need to find the work per level and the number of levels, then multiply the two. The easiest way to find the work is to simply consider the work of the root layer, which is $O(n)$. The number of layers is $\log_3 n \in O(\log n)$, because the size of the input on each layer is divided by 3 as we go down, until a base case is reached. Thus, the solution ends up being $O(n \log n)$.

5 Substitution Method

The **substitution method** corresponds to a sort of guess-and-check method for recurrences. Similar to how guess-and-check works in mathematics, the substitution method operates by guessing a closed-form solution for a given recurrence, then proving that this solution actually holds by induction.

Example: Substitution Method

For the earlier recurrence

$$L(n) = L\left(\frac{n}{2}\right) + L\left(\frac{n}{3}\right),$$

we guess that the closed-form solution is of the form n^b for some constant b. We now show that this bound holds via induction:

Base case: $L(1) = 1 = 1^b$, which holds for any b.

Inductive case: $L(n) = L\left(\frac{n}{2}\right) + L\left(\frac{n}{3}\right)$. We proceed by strong induction, namely, that the solution $L(k) = k^b$ holds for all $k < n$. Then:

$$\begin{aligned} L(n) &= L\left(\frac{n}{2}\right) + L\left(\frac{n}{3}\right) \\ &= \left(\frac{n}{2}\right)^b + \left(\frac{n}{3}\right)^b \\ &= n^b\left[\left(\frac{1}{2}\right)^b + \left(\frac{1}{3}\right)^b\right] \end{aligned}$$

For this to be true, we need $(1/2)^b + (1/3)^b = 1$. It turns out that such a b exists, and using Wolfram Alpha or some other sort of calculator yields that b is approximately equal to 0.788, meaning that $L(n) \approx n^{0.788}$. Therefore, the previous recurrence solves to $W(n) = O\left(n^{0.788}\right)$.

Of course, the substitution method involves guessing, so what happens when we guess wrong? In that case, we will not be able to finish the inductive step, which means that any valid inductive proofs correspond to valid bounds on the solution to the recurrence. Guessing a very rapidly-rising function, like 2^n, can also lead to an easy inductive proof, but also prove a bound on the recurrence that isn't tight.

The substitution method is a valuable tool for solving more difficult recurrences that don't have a clear solution path using the brick method, and relies on mathematical induction to prove these bounds.