1 Big-Oh Notation

1.1 Definition

Big-Oh notation is a way to describe the rate of growth of functions. In CS, we use it to describe properties of algorithms (number of steps to compute or amount of memory required) as the size of the inputs to the algorithm increase. The idea is that we want something that is impervious to constant factors; for example, if your new laptop is twice as fast as your old one, you don't want to have to redo all your analysis.

$f(n)$ is $O(g(n))$	if there exist c, N such that $f(n) \leq c \cdot g(n)$ for all $n \geq N$.	f "$\leq$" g
$f(n)$ is $o(g(n))$	if $\lim_{n\to\infty} f(n)/g(n) = 0$.	f "$<$" g
$f(n)$ is $\Theta(g(n))$	if $f(n)$ is $O(g(n))$ and $g(n)$ is $O(f(n))$.	f "$=$" g
$f(n)$ is $\Omega(g(n))$	if there exist c, N such that $f(n) \geq c \cdot g(n)$ for all $n \geq N$.	f "$\geq$" g
$f(n)$ is $\omega(g(n))$	if $\lim_{n\to\infty} g(n)/f(n) = 0$.	f "$>$" g

1.2 Notes on notation

When we use asymptotic notation within an expression, the asymptotic notation is shorthand for an unspecified function satisfying the relation:

- $n^{O(1)}$ means $n^{f(n)}$ for some function $f(n)$ such that $f(n) = O(1)$.
- $n^2 + \Omega(n)$ means $n^2 + g(n)$ for some function $g(n)$ such that $g(n) = \Omega(n)$.
- $2^{(1-o(1))n}$ means $2^{(1-\epsilon(n))\cdot g(n)}$ for some function $\epsilon(n)$ such that $\epsilon(n) \to 0$ as $n \to \infty$.

When we use asymptotic notation on both sides of an equation, it means that for all choices of the unspecified functions in the left-hand side, we get a valid asymptotic relation:

- $n^2/2 + O(n) = \Omega(n^2)$ because for all f such that $f(n) = O(n)$, we have $n^2/2 + f(n) = \Omega(n^2)$.
- But it is not true that $\Omega(n^2) = n^2/2 + O(n)$ (e.g. $n^2 \neq n^2/2 + O(n)$).

1.3 Exercises

Exercise. *Using Big-Oh notation, describe the rate of growth of*

- n^{124} *vs.* 1.24^n
- $\sqrt[124]{n}$ *vs.* $(\log n)^{124}$
- $n \log n$ *vs.* $n^{\log n}$
- $\sqrt{n}$ *vs.* $2^{\sqrt{\log n}}$
- $\sqrt{n}$ *vs.* $n^{\sin n}$
- $(n + \log n)^2$ *vs.* $n^2 + n \log n$

Solution.
All the bounds below can be made "looser" if desired:

- n^{124} is $o(1.24^n)$
- $\sqrt[124]{n}$ is $\omega((\log n)^{124})$
- $n \log n$ is $o(n^{\log n})$
- $\sqrt{n}$ is $\omega(2^{\sqrt{\log n}})$
- $\sqrt{n}$ and $n^{\sin n}$ aren't related. In particular, $n^{\sin n}$ is bounded by 0 and n, but oscillates between the two
- $(n + \log n)^2$ is $\Theta(n^2 + n \log n)$

Exercise. *Which of the following hold:*

- $\omega(2^n) = \Omega(n^2)$
- $\log_2(n)\Theta(2^n) = o(n^2)$

Solution.
We have:

- True. Let $f(n) = \omega(2^n)$. Then, we have that $f(n)/2^n \to \infty$ as $n \to \infty$. But then for all $c > 0$, there exists N such that $f(n) \geq c2^n$ for $n \geq N$. Pick $c = 1$ and the appropriate N. Then, $f(n) \geq n^2 \leq 2^n$ for all $n \geq \max(2, N)$, so $f(n) = \Omega(n^2)$.
- False; for example, $\log_2(n)(2^n) \neq o(n^2)$ but $2^n = \Theta(2^n)$.

Exercise (Challenge)**.** *Let $f(n) = 1^k + 2^k + \ldots + n^k$ for some constant $k \in \mathbb{N}$; find a 'simple' function $g : \mathbb{N} \to \mathbb{N}$ such that $f(n) = \Theta(g(n))$. Find a constant c such that $\frac{f(n)}{cg(n)} \to 1$ as $n \to \infty$.*

Solution.
Observe that (plot the function x^k, and represent the sum $f(n)$ in two ways as some area on the plot - one for each direction!)

$$\int_0^n x^k \, dx < 1^k + 2^k + \ldots + n^k < \int_0^{n+1} x^k \, dx$$

from which we conclude that

$$\frac{n^{k+1}}{k+1} < f(n) < \frac{(n+1)^{k+1}}{k+1}$$

Now it's easy to conclude that $f(n) = \Theta(n^{k+1})$ and the hidden constant is $\frac{1}{k+1}$.

To prove only that $f(n) = \Theta(n^{k+1})$, note that it is also clear (without integrals) that:

- $1^k + 2^k + \ldots + n^k \leq n^k + n^k + \ldots + n^k = n^{k+1}$, so $f(n) = \Omega(n^{k+1})$.
- $1^k + 2^k + \ldots + n^k \geq \left(\frac{n}{2}\right)^k + \ldots + n^k \geq \frac{n}{2} \cdot \left(\frac{n}{2}\right)^k = \frac{n^{k+1}}{2^{k+1}}$, so $f(n) = O(n^{k+1})$.

1.4 Asymptotic notation as a set of functions*

If you're into formalities, you might be annoyed by 'equations' of the form $n = O(n^2)$. In what sense is the left side equal to the right side? One way not to abuse the equality sign (and coincidentally to be super careful about asymptotic notation) is to define $O(g(n))$ as a set of functions:

$$O(g(n)) := \{f : \mathbb{N} \to \mathbb{N} \mid \text{there exist } c, N \text{ such that } f(n) \leq c \cdot g(n) \text{ for all } n \geq N\}$$

and similarly for the other relations. In this new 'implementation' of asymptotic notation, our old $n = O(n^2)$ translates to $n \in O(n^2)$, and so on.

To see why this is useful, observe that the conventions from the previous subsection now feel a little more natural: expressions like $n^2 + \Omega(n)$ turn into $\{n^2 + h(n) \mid h \in \Omega(n)\}$, and equalities of the form $n^2/2 + O(n) = \Omega(n^2)$ turn into containment of the left side in the right side: $n^2 + O(n) \subset \Omega(n^2)$.

2 Recurrence relations

2.1 General guidelines

There is no single best method for solving recurrences. There's a lot of guesswork and intuition involved. That being said, here are some tips that will help you out most of the time:

- **Guess**! You can get surprisingly far by comparing recurrences to other recurrences you've seen, and by following your gut. Often quadratic, exponential, and logarithmic recurrences are easy to eyeball.
- **Graph it**. Try plugging in some values and graphing the result, alongside some familiar functions. This will often tell you the form of the solution.
- **Substitute**. If a recurrence looks particularly tricky, try substituting various parts of the expression until it begins to look familiar.
- **Solve for constants**. Using the previous methods, you can often determine the form of the solution, but not a specific function. However, there is often enough information in the recurrence to solve for the remainder of the information. For instance, let's say a recurrence looked quadratic. By substituting $T(n) = an^2 + bn + c$ for the recurrence and solving for a, b, and c, you'll likely have enough information to write the exact function.

2.2 The Master Theorem

The previous section stressed methods for finding the exact form of a recurrence, but usually when analyzing algorithms, we care more about the asymptotic form and aren't too picky about being exact. For example, if we have a recurrence $T(n) = n^2 \log n + 4n + 3\sqrt{n} + 2$, what matters most is that $T(n)$ is $\Theta(n^2 \log n)$. In cases like these, if you're only interested in proving Θ bounds on a recurrence, the Master Theorem is very useful. The solution to the recurrence relation $T(n) = aT(n/b) + cn^k$, where $a \geq 1$, $b \geq 2$ are integers, and c and k are positive constants, satisfies

$$T(n) \text{ is } \begin{cases} \Theta(n^{\log_b a}) & \text{if } a > b^k \\ \Theta(n^k \log n) & \text{if } a = b^k \\ \Theta(n^k) & \text{if } a < b^k \end{cases}$$

In general, the idea is that the relationship between a and b^k determines which term of the recurrence dominates; you can see this if you expand out a few layers of the recurrence. For more information about the proof, see the supplement posted with the section notes.

Exercise. *Use the Master Theorem to solve* $T(n) = 4T(n/9) + 7\sqrt{n}$.

Solution.
Plug and chug. $\Theta(n^{\log_9 4}) \approx \Theta(n^{0.631})$

3 Sorting

3.1 Getting our hands dirty with the mergesort recursion

When we were analyzing mergesort in class, we ignored the case when n might be an odd number, and this made the analysis really easy: we had the inequality

$$T(n) \leq 2T(n/2) + n$$

with $T(2) = 1$ and $n = 2^k$ for some k. This easily gives

$$\begin{aligned} T(n) &\leq 2T(n/2) + n \leq 2(2T(n/4) + n/2) + n = 4T(n/4) + 2n \\ &\leq \ldots \leq (n/2)T(2) + (k-1)n \leq n + (k-1)n = n\log_2 n \end{aligned}$$

For arbitrary n, we can let $2^k < n \leq 2^{k+1}$ for some k and use the above to say that $T(n) \leq T(2^{k+1}) = (k+1)2^{k+1}$. For n near 2^k this may not be very satisfactory (the bound is loose by at least a factor of 2). If we want sharper estimates, one way is to try to solve the general recursion:

Exercise. *Solve the recursion*

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n - 1$$

with $T(2) = 1, T(1) = 0$.

Solution.
How do we go about solving that recurrence? One way is to make a table of values and spot a relationship. So we compute some values:

n	1	2	3	4	5	6	7	8	9	10	11	12	...
$T(n)$	0	1	3	5	8	11	14	17	21	25	29	33	...

Looking at how $T(n)$ increases, we see that the gaps are first 2, then 3, then 4,... and that moreover the changes happen at powers of 2. This immediately makes us guess that

$$T(n+1) - T(n) = k + 1 \ \spadesuit$$

where k is the unique integer such that $2^k \leq n < 2^{k+1}$ (i.e., the increment $T(n+1) - T(n)$ is the number of binary digits of n). Writing $n = 2^k + l$ for $l \in [0, 2^k)$, this gives

$$T(2^k + l) = T(2^k) + (k+1)l$$

It's easy to find $T(2^k)$ by expanding the recursion: it is $(k-1)2^k+1$ (exercise!). Recalling that $l = n-2^{\lfloor \log_2 n \rfloor}$ and $k = \lfloor \log_2 n \rfloor$, when we substitute into ♠ we get the conjecture

$$T(n) = n\lfloor \log_2 n \rfloor - 2^{\lfloor \log_2 n \rfloor + 1} + n + 1$$

This is indeed the true formula; our second way of attacking the problem will also serve as a proof of that.

The second idea is to try to use our approximate knowledge of what $T(n)$ should be; this approach illustrates a more systematic way to tackle recursions, but it also gets a little messy. The high-level description is this:

'get an understanding of some additive term in the final formula, and attempt to subtract it away to get something simpler.'

In particular, we expect that $T(n) \approx n \log_2 n$, maybe plus some more stuff. But $\log_2 n$ is not always an integer; so let's replace it by $\lfloor \log_2 n \rfloor$ and try to see what happens if we make the reasonable assumption that

$$T(n) = n\lfloor \log_2 n \rfloor + \text{stuff}$$

and let $F(n) = \text{stuff}$. Now we write the recurrence in terms of $F(n)$ with the hope that it will be simpler:

$$F(n) + n\lfloor \log_2 n \rfloor = F(\lceil n/2 \rceil) + \lceil n/2 \rceil \lfloor \log_2 \lceil n/2 \rceil \rfloor + F(\lfloor n/2 \rfloor) + \lfloor n/2 \rfloor \lfloor \log_2 \lfloor n/2 \rfloor \rfloor + n - 1$$

Maybe this looks very bad right now. But when we have logs base 2, and floors and ceilings of $n/2$ around, you should think binary representations. Indeed, observe that whenever $n+1$ is not a power of 2,

$$\lfloor \log_2 \lceil n/2 \rceil \rfloor = \lfloor \log_2 \lfloor n/2 \rfloor \rfloor = \lfloor \log_2 n \rfloor - 1$$

All this is saying is that $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$ have one fewer digit than n most of the time. Taking into account that $\lfloor n/2 \rfloor + \lceil n/2 \rceil = n$, if $n+1$ is not a power of 2, lots of terms cancel and we have

$$F(n) = F(\lceil n/2 \rceil) + F(\lfloor n/2 \rfloor) - 1$$

What if $n+1$ *is* a power of 2, say $n+1 = 2^{k+1}$? We can directly calculate that in this case we have

$$F(n) = F(\lceil n/2 \rceil) + F(\lfloor n/2 \rfloor) + 2^{\lfloor \log_2 n \rfloor} - 1$$

The recursion $h(n) = h(\lceil n/2 \rceil) + h(\lfloor n/2 \rfloor)$ admits one obvious solution: $h(n) = cn$. The additional term $2^{\lfloor \log_2 n \rfloor}$ we have in our case appears rarely; so we expect $F(n) \approx cn$. We hope to be able to find a constant c such that subtracting away cn from F will kill the linear term responsible for the $h(n) = h(\lceil n/2 \rceil) + h(\lfloor n/2 \rfloor)$ part.

Plotting shows that c shouldn't be too large in magnitude; choosing $c = 1$ the pattern becomes very obvious, and we can easily show by induction that $F(n) - n = 1 + 2^{\lfloor \log_2 n \rfloor + 1} = 1 + 2^{\#\text{binary digits of } n}$. Hint: because we go from n to roughly $n/2$ in the recursion, it will be convenient in your induction hypothesis should to assume that the statement is true for all numbers n with k binary digits, and use that to prove the statement for numbers with $k+1$ binary digits!

3.2 Counting sort with duplicates

In class, we went over the counting sort algorithm, but with the assumption that there were no duplicates in the array. Now, we address the issue of duplicates.

Exercise. *Give a modification of the counting sort algorithm from class which can handle arrays with duplicate values. What is the runtime of the modified algorithm?*

Solution.
Carry out the counting and prefix sum pieces of the algorithm as before. When walking back through the inputs, on an input a_i:

- If the sorted array already has an object in the count$[a_i]$ position, move one (this means we have already seen this a_i.
- Otherwise, copy a_i into the count$[a_i - 1] + 1$ through count$[a_i]$ positions, inclusive, in the sorted array.

To see why we're doing all this (and why it works), observe that after the first two operations, the number count$[a_i]$ gives us the position of the last copy of a_i in the sorted array.

This third step still takes $O(n)$ time, as before: in particular, the first case occurs at most n times, and the second results in at most one copy per cell in the sorted array. Thus, the runtime is $O(n + k)$, as before.