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.* $n^2 + n \log n$

Exercise. *Which of the following hold:*

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

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 $n \to \infty$.*

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* $T(n) = 4T(n/9) + 7\sqrt{n}$.

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

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?*