• **DISCLAIMER:** These notes are not necessarily an accurate representation of what I said during the class. They are mostly what I intend to say, and have not been carefully edited.

• Main topics:
  - Intro logistics: evaluation, homeworks, tests, Piazza.
  - A few more examples of big-O notation.
  - Runtime recurrences.
  - Master Theorem for solving runtime recurrences.

## 1 Asymptotic Runtimes

We start with a few more concrete examples about big-O, specifically how to compare functions. Recall that the definition was $f(n) \leq O(g(n))$ if there exists some constant $c > 0$ such that $f(n) \leq c \cdot g(n)$ for all values of $n$.

Note that one consequence of this definition is that it’s multiplicative, that is if

$$f_1(n) \leq O(g_1(n)) \quad \text{and} \quad f_2(n) \leq O(g_2(n)),$$

we have

$$f_1(n) \cdot f_2(n) \leq O(g_1(n) \cdot g_2(n)).$$

And also (by definition of $\Omega$),

$$g_1(n) \cdot g_2(n) \geq \Omega(f_1(n) \cdot f_2(n)).$$

Two critical building blocks for proving runtime bounds are:

1. For any $a$ and $b > 1$, $n^a \leq O(b^n)$.
2. For any $a$ and $b > 0$, $(\log n)^a \leq O(n^b)$.

Composing these allows us to prove most statements involving $O(\cdot)$ and $\Omega(\cdot)$. For example, composing $n \leq O(n)$ with $\log n \leq O(n)$ gives $n \log n \leq O(n^2)$. 

1
2 Recursive Algorithms

The goal of the next few lectures is to discuss divide-and-conquer algorithms, which are like recursive algorithms, except one recurses onto problems that are significantly smaller. Consider two recursive algorithms that you’ve probably seen: sorting and binary search.

1. We can combine two sorted lists of length \(n/2\) into one sorted list of length \(n\) in \(O(n)\) time.

2. If we have a sorted list, and want to find number \(x\) in it, we can narrow down to half the list in \(O(1)\) time (by comparing against middle).

In each of these cases, we’re reducing solving a larger problem to several, or one, problems of smaller size.

1. Merge sort reduces sorting \(n\) numbers to sorting two lists of \(n/2\) numbers, with \(O(n)\) overhead. For example, given the array:

\[
\text{Arr}[1 \ldots 8] = 8, 7, 12, 3, 5, 11, 2, 9,
\]

we partition it into

\[
\text{Arr}[1 \ldots 4] = 8, 7, 12, 3; \quad \text{Arr}[5 \ldots 8] = 5, 11, 2, 9,
\]

sorting both (via recursive calls) gives:

\[
\text{Arr}[1 \ldots 4] = 3, 7, 8, 12; \quad \text{Arr}[5 \ldots 8] = 2, 5, 9, 11,
\]

upon which we can walk two pointers across the two arrays (in linear time) to get the sorted list

\[
\text{Arr}[1 \ldots 8] = 2, 3, 5, 7, 8, 9, 11, 12.
\]

An animation of how this merging step works is in the slides.

2. Binary search reduces finding something on a list of length \(n\) to finding something on a list of length \(n/2\), with \(O(1)\) overhead. Specifically, if we want to search for 6 in the above sorted array, we once again split it into two halves,

\[
\text{Arr}[1 \ldots 4] = 2, 3, 5, 7; \quad \text{Arr}[5 \ldots 8] = 8, 9, 11, 12,
\]

comparing 3 against 7 tells us that it’s in the left half, so the problem becomes searching for 6 on the list

\[
\text{Arr}[1 \ldots 4] = 2, 3, 5, 7,
\]

which we can split again into

\[
\text{Arr}[1 \ldots 2] = 2, 3; \quad \text{Arr}[3 \ldots 4] = 5, 7,
\]
which by comparing 3 against 6 gives that we should repeat again on the right side,

\[ Arr[3\ldots4] = 5, 7, \]

but here we realize that everything on the left half is < 6, while everything on the right side is > 6, so we stop.

One can also analyze routines such as quick sort / quick select similarly. However, those are randomized, so rigorously analyzing them fall outside of our scope.

What’s interesting here is that in each case, we’re using the same problem to solve itself. When viewed non-algorithmically, this is a chicken-and-egg problem: if solving a problem takes time \( T \), then two calls to it should take time \( 2T \), and we should be getting:

\[ T = 2T + O(n). \]

This is clearly not the case, and that is because we’re recursing onto problems whose sizes are smaller.

That is, we should parameterize running times via the size of the input, and instead define the function \( T(n) \), which is the cost of solving a problem of size \( n \). With this function present, we get the recurrence

1. \( T(n) = 2T(n/2) + O(n) \) for merge-sort,
2. \( T(n) = T(n/2) + O(1) \) for binary search.

In both cases, it’s reasonably easy to reason about what the solution looks like.

## 3 Analyzing Recursion Runtimes

There are also several more systematic methods for analyzing running times. The most adhoc is guess-and-check, which just uses induction and the definition of big-O notation. However, for now, we’ll use something more systematic that suffices for most runtime recurrences, the Master Theorem.

**Theorem 3.1.** If \( T(n) = aT(n/b) + O(n^d) \) for constants \( a > 0, b > 1, \) and \( d \geq 0 \), then

\[
T(n) = \begin{cases} 
O(n^d) & \text{if } d > \log_b a \\
O(n^d \log n) & \text{if } d = \log_b a \\
O(n^{\log_b a}) & \text{if } d < \log_b a.
\end{cases}
\]

1. Merge sort: \( T(n) = 2T(n/2) + O(n) \)
   (a) \( a = 2, \)
   (b) \( b = 2, \)
2. Binary search: $T(n) = T(n/2) + O(1)$
   (a) $a = 1$,
   (b) $b = 2$,
   (c) $d = 0$,
   (d) $\log_b a = 0$, same as $d$ (again), second case,
   (e) $O(n^d \log n) = O(n \log n)$.

3. Selection: $T(n) = T(n/2) + O(n)$:
   (a) $a = 1$,
   (b) $b = 2$,
   (c) $d = 1$,
   (d) $\log_b a = 0$, $< d$, first case,
   (e) $O(n^d) = O(n)$.

4. Integer multiplication (next time): $T(n) = 4T(n/2) + O(n)$:
   (a) $a = 4$,
   (b) $b = 2$,
   (c) $d = 1$,
   (d) $\log_b a = 2$, $> d$, third case,
   (e) $O(n^{\log ba}) = O(n^2)$. 