WEEK 1 NOTES

About the Course, Intro

- Course is not language specific and uses pseudo code
- Intent is to give understanding of algorithms, data structures, big O notation, which is invaluable for all software engineers and especially in the coding interview
- Provides an example of long form multiplication being an algorithm
  - Not only one way to do it, an alternative is Karatsuba Multiplication
- Most important point of algorithms:
  - ALWAYS ASK: CAN WE DO BETTER?

Merge Sort

- Merge Sort is a good algorithm to introduce divide & conquer
  - An improvement over Selection sort, Insertion sort, and Bubble sort
- How merge sort works
  - Take a list / array of numbers, split that in half
  - Then take each half and run recursive sorting calls on each, sorting them
  - Then merge the two halves together
- Pseudocode for Merge sort
  - For k = 1 to n
    - If A(I) < B(j)
      - C(k) = A(i)
      - i++
    - Else [B(j) < A(i)]
      - C(k) = B(j)
      - j++
  - End
- Running Time of Merge (Big O)
  - 6n log2(n) + 6n operations

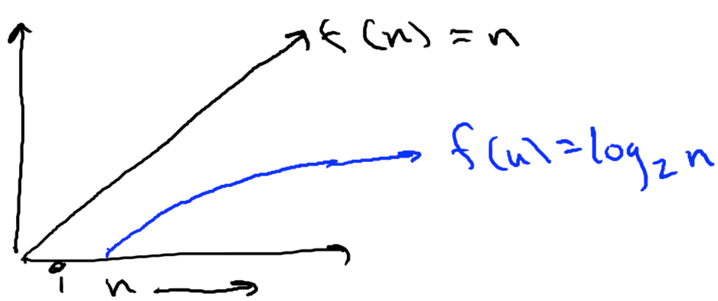
Running Time of Merge Sort

Claim : Merge Sort requires

$\leq 6n \log_2 n + 6n$ operations

to sort n numbers.

Recall : = $\log_2 n$ is the # of times you divide by 2 until you get down to 1



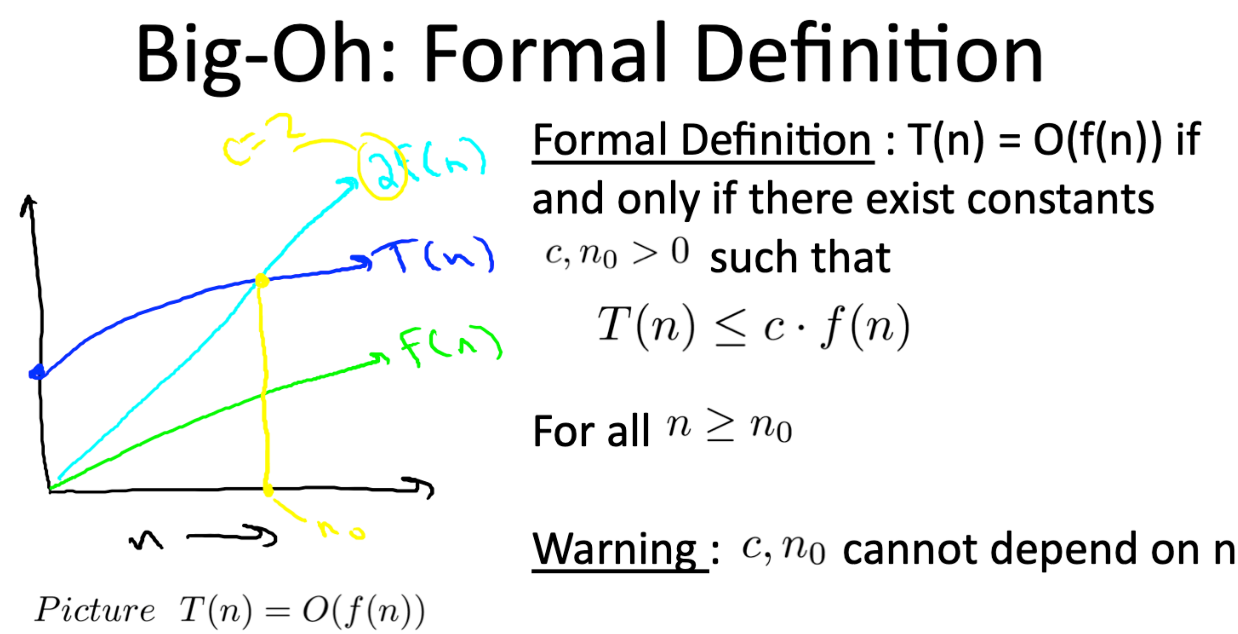
Guiding Principles

1. We will be using "Worst case analysis" for big O runtime analysis
   A. This is best for general purpose without knowing anything about the underlying data or use case
   B. In the real world, if this is a critical algorithm for your business, then you will also want to analyze the underlying data and use case, but for our purpose in this course it's not required
2. We won't pay much attention to constant factors, lower order terms
   A. Similarly, this is easier to manage and we lose very little predictive power. But for real life critical algorithms, you will want to also look into the constant terms
3. Asymptotic Analysis: focus on running time for large input sizes of n vs small input sizes
   A. Smaller input sizes are usually trivial for modern computers to handle, so we should really only be optimizing for large inputs

- What is a fast algorithm?
  - As close to O(1) as possible
    - O(1) = 1 step
    - O (logn) = grows at the log of n (less than linear)
    - O(n) = linear
    - O(n^2) = exponential

Asymptotic Analysis

- The Gist
  - Asymptotic Analysis = "Big Oh" notation
  - Provides a standardized vocabulary analysis of performance of algorithms
  - Allows to differentiate between better and worse approaches to sorting, searching, etc.
  - High-level idea
    - Ignores constant factors and lower order terms
    - As data grows in size, constants and lower order terms become more and more irrelevant
    - Also simplifies things for rapid analysis
  - Example - searching array for an integer
- Big-Oh notation
  - Formal definition:
    - T(n) = O(f(n)) if and only if there exist constants c, n0 > 0 such that
      - T(n) <= c * f(n)
      - for all n >= n0
      - Warning: c, n0 cannot depend on n



- **Big-Oh basic examples**

Example #1

Claim : if $T(n) = a_k n^k + ... + a_1 n + a_0$ then

$$T(n) = O(n^k)$$

Proof : Choose $n_0 = 1$ and $c = |a_k| + |a_{k-1}| + .. + |a_1| + |a_0|$

Need to show that $\forall n \geq 1, T(n) \leq c \cdot n^k$

We have, for every $n \geq 1$,

$$\begin{aligned} T(n) &\leq |a_k| n^k + ... + |a_1| n + |a_0| \\ &\leq |a_k| n^k + ... + |a_1| n^k + |a_0| n^k \\ &= c \cdot n^k \end{aligned}$$

Example #2

Claim : for every $k \geq 1$, n^k is not $O(n^{k-1})$

Proof : by contradiction. Suppose $n^k = O(n^{k-1})$

Then there exist constants c, n_0 such that

$$n^k \leq c \cdot n^{k-1} \quad \forall n \geq n_0$$

But then [cancelling n^{k-1} from both sides]:

$$n \leq c \quad \forall n \geq n_0$$

Which is clearly False [contradiction].

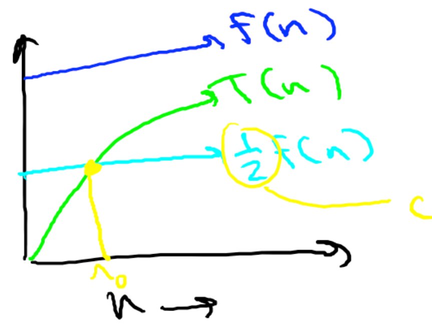
- **Big Omega and Theta**
  - Close relatives but less popular than Big-Oh notation
  - Omega = upper bound
  - Theta = lower bound
- Little-Oh notation
  - Not used often
  - Simply says that one function grows faster than another

Omega Notation

Definition : $T(n) = \Omega(f(n))$
If and only if there exist constants c, n_0 such that

$$T(n) \geq c \cdot f(n) \quad \forall n \geq n_0 \, .$$

Picture



$$T(n) = \Omega(f(n))$$

Theta Notation

Definition : $T(n) = \theta(f(n))$ if and only if
$T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$

Equivalent : there exist constants c_1, c_2, n_0 such that

$$c_1 f(n) \leq T(n) \leq c_2 f(n)$$

$$\forall n \geq n_0$$