DESIGNING ALGORITHMS

Thinking like a Machine

HEURISTICS VS ALGORITHMS

- Heuristics
  - “Rule of thumb”
  - Gets you an approximate answer easily
  - As humans, we tend to think like this!
- Algorithms
  - Procedure for generating an exact solution every time
  - Sometimes tedious, with every edge case accounted for
  - This is how computers think!

THINKING LIKE A MACHINE

- For job interviews, many major tech companies favor white-boarding algorithms over writing actual code.
- Why do you think that is?
- Communication Ability >= Implicit knowledge

THINK PAIR SHARE

- Find a partner
- We will show you a mock interview question
- *THINK* about your approach
- *PAIR* off with your partner and discuss (~3 mins)
- *SHARE* your ideas with the group (~3 mins)

THINK PAIR SHARE

- When encountering a new problem, how can we attempt to solve it algorithmically?
- What makes a certain approach “better” than others?
- When do we want to use iteration (loops?)
- When do we want to use recursion?

THINK PAIR SHARE

- Think of some of the Data Structure's we've learned so far. What is a good use-case for:
  - Hash Tables
  - Linked Lists
  - Binary Search Trees

THINK PAIR SHARE

- Think about the following terms. What do you think they mean in regards to problem solving?
  - *Brute Force*
  - *Bottoms-up Problem Solving*
  - *Dynamic Programming*
  - *Memoization*

BIG O

WHAT IS IT?

- "Order of Magnitude"
  - Gets at efficiency of an algorithm / performance at scale.
- Complexity of algorithm
  - Time
  - Space
- Big O refers to an upper bound
  - Worst case scenario

WHAT IT IS NOT?

- Best Case or Average Case scenario.
    - Best Case - Big Omega (Ω)
    - Average Case - Big Theta (Θ)
- Not useful when algorithm is small in size.
- Doesn’t provide real units of time/space just their magnitude.

HOW DO WE CALCULATE IT?

- **Count the steps**
  - **Anytime you hit a loop, multiply the number of times we iterate by the max complexity of each loop iteration.**
    - **Stuff inside loop** multiplies
    - **Nested Loops**: Multiply
    - **Sibling Loops:** Add
- **Drop the constants**
- **Drop less significant terms**

COMMON RUNTIMES

- **O(1):** constant
- **O(log n):** logarthmic complexity
- **O(n):** linear
- **O(n*log n):** log-linear
- **O(n^2)**: quadratic
- **O(n^3), O(n^2), O(n^4):** polynomial
- **O(2^n)**: exponential complexity
- **O(n!)**: factorial

SPACE COMPLEXITY

- Really: memory
- Process is similar to deriving time complexity, except you count the space.
- Things to look for:
  - # Recursive calls
  - Size of iterables (strings, arrays, linked-lists, etc)
  - New data structures created during course of also

EXAMPLE

- Let’s diagram a function ‘nthFib’ that consumes a positive integer.
- This function should return the nth term in the Fibonacci Sequence
- Fibonacci sequence: ’0, 1, 1, 2, 3, 5, 8, 13, 21…’

WORKSHOP: SEARCH && !DESTROY

- Practice algorithmic problem solving with a few classic interview problems
- Diagram your solution using pen && paper/whiteboard
- Identify any edge cases (and feel free to test them!)
- Step through your solution BEFORE attempting to code it
- Emphasis on communication NOT getting the answer quickly (this is what Google/Amazon/etc. will grade you on)