# INTERVIEW PR3P

•••

~kryddaður~

# Newsflash

- Apple Confirms it is a Google Cloud Customer (link)
- Six Degrees of Wikipedia (link)

# Some terms

- In computer science, we are concerned with the efficiency in which we use resources, which are oftentimes scarce.
  - Time complexity: resource = time
  - Space complexity: resource = memory, disk space (includes call stack)
- O(X) means the operation you are conducting takes up, *at most*, c * X resources, where c is some constant number (whereas X can be asymptotically yuge)

- O(1): Constant, not dependent on size of input data (n)
- O(n): Linear, every time n doubles, resource usage doubles
- $O(\log_k n)$: Logarithmic, generally involves some kind of partitioning of the data
- $O(n^2)$: Quadratic, every time n doubles, resource usage quadruples
- $O(n^k)$ for some constant k: Polynomial
- $O(k^n)$ for some constant k: Exponential, every time n increases by 1, resource usage **is multiplied by k**, *avoid at all costs*

# Collections

# Arrays

- Most rudimentary collection data structure
- Association between key-value pairs, where keys must be integers

- O(1) indexed element access
- O(log n) search **if sorted**, O(n) otherwise
- O(n) insertion/deletion

- O(n) space

Pros:

- Good cache locality
- Easy to make, minimal additional information other than your data

Cons:

- Any unused indices between 0 - [max index] is wasted space
- Layout does not give additional information about the contents of the data

# Linked Lists

- No indexed element access, instead they can be iterated through by following pointers between them.
- Can augment pointing structure
  - e.g. doubly-linked, circular, etc.

- O(n) search
- O(1) insertion/deletion

- O(n) space

Pro:

- Simple insertion and deletion

Con:

- Pointers is small space overhead
- Can’t perform binary search

# Hash Sets & Tables

- Map any arbitrary *key* to some associated *value*
- Values are transformed into a key, which is used to index and find given value
- Generally ok to take their functionality for granted during coding problem, but be prepared to explain its inner workings
    - Probing, hash function, load factor, etc.

- O(1) insertion/deletion
- O(1) search *if searching by key*

- O(n) space

Pros:

- Most languages provide an easy-to-use built-in hash set/table
- Easy to use

Cons:

- Generally tricky to *implement* right
    - Delete is hard
- Many assumptions need to be met for time complexity to be true
- Higher overhead per-element (not good for small collections)

# Practice Problems

1. Write a function that takes an input string and returns the minimum substring which contains every letter of the alphabet set at least once.
2. Given two arrays representing two individuals' rankings of restaurants, find the mutually highest ranked restaurant, or return None if no such restaurant is found.
3. You are given an int array of size n with unique values ranging from 0 to (n - 1). Suppose one element is replaced with a -1. Find the value of the original element. Explore ways to do this in-place, with an extra array, and an extra hash table.

# Recursive things

# Trees

- Every “node” has information, but also points to other nodes in an acyclical manner
- Binary Search Trees are most common, in which every “parent” node is greater than its left “child” and less than its right “child”
- BSTs can be “augmented” to ensure balanced-ness
  - Red-black
  - AVL
  - Treap

Pros:

- Lots of flexibility by being able to augment nodes
- Keep searching as fast as with arrays, but with less maintenance work

Cons:

- Rotations can be a pain

- O(log n) insert/search/delete **if balanced**, O(n) otherwise
- O(n) space

# Recursion

- Call a function within the function itself.
- Helpful on problems that require solving a smaller problem first.

Pro:

- Work on complex, multi-part problems without taking up memory space
- Often a really nice way to model problems

Con:

- Can cause a function call “stack overflow"
- Slow if not tail recursion

# Stacks & Queues

- Can only access the first and/or last elements

- O(1) insertion/deletion
- Generally can't iterate/search

Pro:

- Can simulate recursion without fear of stack overflow

Con:

- Searching is expensive and needs external structure to avoid deleting data

# Heaps

- Tree structure, but with different relationship rules
  - Parents are greater (or smaller) than *both* children
- Has a sorting algorithm based off of it

- O(1) find min
- O(log n) insertion/deletion/search
- O(n) build

Pro:

- Obtain minimum super fast
- Can be represented through special indexing on an array

Con:

- Occasionally too complicated for the problem at hand
- Better get upHeap() right!

# Practice Problems

1. Find the nth Fibonacci number, once using recursion, again with a stack or queue.
2. Given pointers to two nodes of a BST (nodes have parent pointers but you are not given a root), find the common ancestor (if it exists).
3. Given a two dimensional array of pixels, and a pixel that the user clicked on, implement a “bucket-filling” operation.
4. Print a binary tree in “zig-zag” order, in which the tree is printed level-by-level, where odd levels are read left to right and even levels are right to left.

# Graphs

- Great for modeling relationships between entities
- Have nodes, edges
- Edges can be undirected, directed
- Edges can be weighted
- Can have cycles
- Many representations
  - Edge list
  - Adjacency matrix
  - Adjacency lists
- BFS, DFS

# Practice Problems

1. Given a directed graph, design an algorithm to find out whether there is a route between two nodes.
2. Given a function that returns a list of URLs that can be visited from a given URL, write a function that returns all distinct pages on a website.
3. You are given a function that returns all friends of a given user id. We define a user as “recommendable” to another user if they are second degree connections. Write a function that returns an array of user id’s recommendable to a given user id, sorted by the number of mutual friends the two have.

# Data layout & search

# Sorting

- Lower bound of O(n log n) *for comparison-based sorts*
- Some freaky shit happens with radix sort and it ends up being O(dn)
- Good to know when to use bubble, merge, quick, insertion sorts & their respective time complexities

- What is the space complexity of a traditional merge sort?

# Searching

- How can you best lay out your data such that it is searchable in some key way?
- Often a trade-off between initial build time and search time

# Practice problems

**Given a log like:**

T1 requested A @ 1

T2 requested B @ 3

T1 requested C @ 4

T1 freed A @ 6

Write a function that can determine who owned a given server S at time T.

How fast can you do this?

How do you index your data?