**Data Structures and Algorithms Mock Interview Study Guide**

**What is an algorithm?**

A clearly defined set of instructions to solve a problem

**How can algorithm performance be analyzed?**

Space or time complexity

Space complexity 🡪 the amount of physical memory that an algorithm requires to complete

🡪 for ex. stack of recursive function calls, each function in the stack takes up space in memory

**Recursion:** method of solving problems that involves calling itself, in each call it breaks down the problem into smaller and smaller subproblems

\*recursion is not a data structure or an algorithm, it is a concept

broken down into 2 parts: the base case (when to stop), and the general/recursive case (function calling itself)

\*forward phase: a **recursion** where it grows bigger with each **step**

\*backward phase: writing equations first for the final stage and then proceeding **backwards** to the first stage.

Pure recursion tips:

🡪 for arrays, use methods like slice, the spread operator, and concat that make copies of arrays so you do not mutate them

🡪strings are immutable so you will need to use methods like slice, substr, or substring to make copies of strings

🡪to make copies of objects use Object.assign, or the spread operator

**Big-O:** A way of describing the time complexity of an algorithm

🡪Time complexity: number of operations an algorithm requires to complete

\* Big O notation can help us understand how a given algorithm will perform in the best- case scenario, in the worst-case scenario, and on average.

Constant time **O(1)** : no matter the size of the input, the algorithm will take the same amount of time to complete

For ex.🡪 accessing an array item, basic arithmetic operations

Logarithmic Time **O(log(n))** : next best thing after constant time, does take longer with larger inputs, but the running time increases slowly, cut the problem size in half each round through

Linear Time **O(n)**: running times that are directly proportional to the size (n) of the input

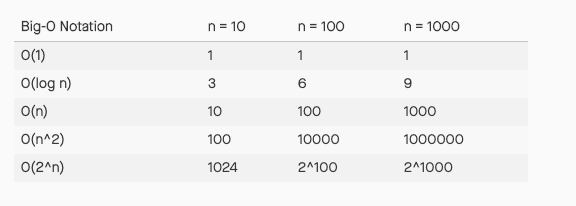
For ex.🡪 summing elements in an array, finding min/max value in an array

Polynomial time **O(n^k)**: has a running time that would be some input size n raised to some constant power k, easiest way to understand polynomial time complexity is with nested loops.

For ex. 🡪 An algorithm that requires two levels of looping over an input would be O(n^2), while one requiring 3 levels of looping would be O(n^3). Both have polynomial time complexity

Exponential time **O(2^n):** running times that grow rapidly with any increase in input size

For ex 🡪 For an input of size 2, an exponential time algorithm will take 2^2 = 4 time. With an input of size 10, the same algorithm will take 2^10 = 1024 time



**Arrays:** an ordered sequence of data, stored in contiguous memory

Pointers 🡪 variables containing memory addresses

**Linked Lists:** data structures that allocate blocks of memory on demand, these blocks of memory are connected by a mechanism called “linking”

Stored in non-contiguous memory

Node 🡪 unit of memory holding a single item in a linked list, a node can be allocate from anywhere in the memory, it does not have to be next to the previously allocated node

\*linked lists work by storing a series of nodes

(Singly Linked List)

\*each node consists of a value and a pointer to the next node in the sequence

(Doubly Linked List)

\*nodes contain a pointer to the previous node in addition to the next

**Stack:** a data structure similar to a list with access restricted to only 1 end, LIFO(last in first out) order

\*thought of as vertical data structures, unlike lists which are horizontal

\*the first and only directly accessible item on the stack is referred to as “top” of the stack

🡪 can be implemented using singly linked list

**Queue:** data structure that models a FIFO (first in first out) operation

\*a type of list where data is inserted at the end and is removed from the front

\*used to store data in the order in which they occur, as opposed to a stack, in which the last piece of data entered is the 1st element used for processing

🡪 can be implemented using singly, or doubly linked list

**Hash Maps:** unordered associations between keys and values

\*HashMap is a data structure that uses the concept of hashing (mapping a key to its location)

Hash table🡪 storage that holds the records (key and value), implemented internally using an array, hash maps require a hash table

Hash function🡪 maps keys to positions in the hash table, a good hash function attempts to distribute the keys as evenly as possible among slots in the hash table

Collision: the main difficulty with hashmaps, what happens if 2 unique keys hash to the same slot in the array?

2 ways to solve collisions🡪 open addressing, separate chaining

Open addressing: when you have a collision, you hash the key to the empty slot nearest to where it should live.

Separate chaining: uses linked lists to hash the keys that run into collision. The first slot contains the pointer to the head of the list. When a key collides with another, we use the next pointers to put the keys in a linked list.

**Binary Search Tree:** Trees are data structures that consist of nodes which are linked together in a certain way

\*Nodes in a tree have a parent-child relationship

Root Node- special node at the top of the tree, has no parent

Leaf Node- nodes without any children

binary tree🡪 is a tree with an additional limitation: each node can only have 0, 1, or 2 children (at most 2 children).

\*A branch in a tree signifies a decision path, a choice that connects 1 node to another.

\*A binary tree may have a left branch and a right branch.

\*A subtree is a mini tree within a binary tree, whose root can be any node and all of its descendants rooted at that node.

Binary Search Tree 🡪 All of the nodes in the left-hand branch of a node are guaranteed to have lower values than the node itself, and all of the nodes in the right-hand branch of a node are guaranteed to have a higher value than the node itself.