[Notes] Cracking the Coding Interview

Xu, Bangyao
*bangyaoxu*0321@*gmail.com*

January 24, 2023

1 Introduction

1.1 Big O

- $O \rightarrow$ upper bound; $\Omega \rightarrow$ lower bound
- $\Theta \rightarrow$ tight bound (both O and Ω): $\Theta(N)$ if it's both $O(N)$ and $\Omega(N)$
- Best/Worst/Expected cases
- Stack space in recursive calls takes up actual memory
- Big O drops the constants and non-dominant terms
- When the array hits capacity, **ArrayList** class will create a new array with double capacity and copy all elements over to the new array
- Amortized time: $X + \frac{X}{2} + \frac{X}{4} + ... + 1 \approx 2X \rightarrow O(X)$
- X adds take $O(X)$ time; Amortized time for each adding is $O(1)$
- The number of elements in the problem space gets halved each time $\rightarrow O(logN)$ e.g. binary search
- Recursive function making multiple calls $\rightarrow O(branches^{depth})$, where the *branches* is the number of times each recursive call branches

2 Data Structure

2.1 Arrays and Strings

- Array questions and string questions are often interchangeable

- An array of linked lists + A hash code function → Hash tables
- Number of collisions: from $O(1)$ to $O(N)$, where N is number of keys
- Dynamically resizable arrays/lists → resizing factor e.g. 2 in Java
- **StringBuilder** creates a resizable array of all the strings, copying them back to a string only with necessary

2.2 Linked Lists

- Iterate through K elements if want to find the K^{th} element in the list
- Benefit of linked list is that you can add and remove items from the beginning of the list in constant time
- Deleting a node from a singly linked list → 1) check null pointer; 2) update head/tail pointer as necessary (C/C++ needs memory management)
- The 'runner' technique → iterate through the linked list with 2 pointers simultaneously, with one ahead of the other e.g. 'fast' node is fixed amount/hopping multiple nodes ahead of 'slow' node
- A number of linked list problems reply on recursions

2.3 Stacks and Queues

- A stack (LIFO) allows constant-time adds and removes
- Stacks are often useful in recursive algorithms → push temporary data onto a stack as you recurse but then remove them as you backtrack
- A queue (FIFO) can also be implemented with a linked list
- Queues are often used in breadth-first search or implementing a cache

2.4 Trees and Graphs

- The tree cannot contain cycles
- A *binary tree* is a tree which each node has up to 2 children
- A node is called a 'leaf' node if it has no children
- A *binary search tree* is a binary tree in which every node fits specific ordering property: $all\ left\ descendants \leq n < all\ right\ descendants$

- A 'balanced' tree means not terribly imbalanced → balanced enough to ensure $O(logN)$ times for **insert** and **find**
- A *complete binary tree* is a binary tree in which every level of the tree is fully filled, except for perhaps the last level (filled left to right)
- A *full binary tree* is a binary tree in which every node has either 0 or 2 children i.e. no nodes have only one child
- A *perfect binary tree* is one where all interior nodes have 2 children and all leaf nodes are at the same level ($2^k - 1$ nodes, where k is the number of levels)
- Binary tree traversal: 1) in-order (visiting nodes in ascending order); 2) pre-order (root is first visited); 3) post-order (root is last visited)
- A *min-heap* is a complete binary tree where each node is smaller than its children → the root is the minimum element in the tree
- **Insert** on a min-heap: insert it at the bottom looking left to right → bubble up by swapping with its parent ($O(logN)$, where N is the number of nodes in the heap)
- **Extract minimum element** on a min-heap: find it at the top → remove it and swap it with the last element → bubble down by swapping with its smaller child ($O(logN)$)
- A *max-heap* is equivalent to a min-heap, but the elements are in descending order rather than ascending order
- A *trie* (prefix tree) is a variant of an n-ary tree in which characters are stored at each node → Each path down the tree represents a word
- There are 2 common ways to represent a graph: 1) adjacency list (every vertex/node stores a list of adjacent vertices); 2) adjacency matrices
- Depth-First-Search(DFS): go deep (children) before go wide (neighbours)
- Breadth-First-Search(BFS): go wide before go deep
- DFS is preferred if wanting to visit every node in the graph
- BFS is better for finding the shortest path between 2 nodes
- Tree traversal are a form of DFS (must check if the node has been visited in the graph to avoid infinite loop)

- BFS is ***not*** recursive → Iterative solution involving a queue works best
- *Bidirectional Search* is used to find the shortest path between a source and destination → run 2 simultaneous BFS until their searches collide
- Traditional BFS: $O(k^d)$ v.s. Bidirectional search: $O(k^{\frac{d}{2}})$, where k is the number of most adjacent nodes and d is the length in between

3 Concepts and Algorithms

3.1 Bit Manipulation

- Binary representation of -K as a N-bit number is **concat**(1, $2^{N-1} - K$)
- Logical right shift (>>>) → put a 0 in the significant bit
- Arithmetic right shift (>>) → fill the new bits with sign bit
- Common bit tasks: get/set/clear/update

3.2 Math and Logic Puzzles

- Every positive integer can be decomposed into a product of primes
- **gcd**(x,y) × **lcm**(x,y) = $x \times y$
- Check for primality → iterate from 2 through $\sqrt{n}$
- The Sieve of Eratosthenes → all non-prime numbers are divisible by a prime number

3.3 Object-Oriented Design

- How to approach: 1) Handle ambiguity; 2) Define the core objects; 3) Analyze relationships; 4) Investigate actions
- Widely used design patterns: Singleton class & Factory Method

3.4 Recursion and Dynamic Programming

- Recursive solutions are built of solutions to subproblems
- 3 most common approaches to divide a problem into subproblems: 1) bottom-up; 2) top-down; 3) half-and-half (e.g. data set)

- Recursive (space inefficient) v.s. Iterative (complex) → trade-off
- Dynamic Programming → taking recursive algorithm and finding the overlapping subproblems (cache results for future recursive calls)

3.5 System Design and Scalability

- System design steps: 1) Scope the problem; 2) Make reasonable assumptions; 3) Draw the major components; 4) Identify the key issues; 5) Redesign for the key issues
- Vertical scaling → increasing the resources of a specific node
- Horizontal scaling → increasing the number of nodes
- Load balancer → allow a system to distribute the load evenly so that one server doesn't crash and take down the whole system
- Denationalization means adding redundant information into a database to speed up reads → Or go with a NoSQL database
- Common ways of partitioning: vertical/key(hash)-based/directory-based
- Slow operations should ideally be done asynchronously → pre-process
- MapReduce → parallel processing → scalable
- Considerations when designing a system: 1) Failures; 2) Availability & Reliability; 3) Read-heavy v.s. Write-heavy; 4) Security

3.6 Sorting and Searching

- Bubble sort and Selection sort → $O(n^2)$ runtime + $O(1)$ memory
- Merge sort: divide, sort, merge → $O(nlog(n))$ runtime
- Quick sort: random pick, partition, sort → $O(nlog(n))$ average and $O(n^2)$ worst runtime + $O(log(n))$ memory
- Radix sort: group integers by each digit → $O(kn)$ runtime
- Binary search → compare to the midpoint

3.7 Testing

- Black box testing → given the software as-is and need to test it
- White box testing → additional access to individual functions
- Types of test cases: 1) normal cases; 2) extremes; 3) nulls & illegal inputs; 4) strange inputs
- Troubleshooting: 1) Understand the scenario; 2) Break down the problem; 3) Create specific, manageable tests

■