Studying for a tech interview sucks. Here's an open source cheat sheet to help

🕙 **34** commits	🖇 **2** branches	📦 **0** packages	🏷 **0** releases	👥 **5** contributors	⚖ MIT

Branch: master ▾   New pull request

Find file   Clone or download ▾

🖼 TSiege Merge pull request #8 from d-m-u/patch-1 ...		Latest commit 2bf4b5b 16 hours ago	
📁 challenges	improve link		2 months ago
📄 .gitignore	basic set up for ts/js challenges		2 months ago
📄 LICENSE	license		6 months ago
📄 README.md	Update README.md		2 days ago

📖 **README.md**

Tech Interview Cheat Sheet

This list is meant to be both a quick guide and reference for further research into these topics. It's basically a summary of that comp sci course you never took or forgot about, so there's no way it can cover everything in depth.

Contributing

This is an open source, community project, and I am grateful for all the help I can get. If you find a mistake make a PR and please have a source so I can confirm the correction. If you have any suggestions feel free to open an issue.

Challenges

This project now has actual code challenges! This challenges are meant to cover the topics you'll read below. Maybe you'll see them in an interview and maybe you won't. Either way you'll probably learn something new. Click here for more

Table of Content

- Asymptotic Notation
- Data Structures
  - Array
  - Linked List
  - Hash Table or Hash Map
  - Binary Tree
- Algorithms
  - Algorithm Basics
  - Search Algorithms
    - Breadth First Search
    - Depth First Search

Asymptotic Notation

Definition:

Asymptotic Notation is the hardware independent notation used to tell the time and space complexity of an algorithm. Meaning it's a standardized way of measuring how much memory an algorithm uses or how long it runs for given an input.
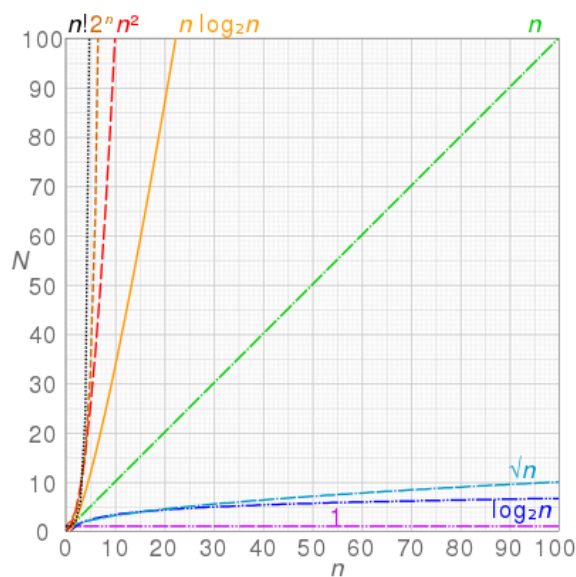
Complexities

The following are the Asymptotic rates of growth from best to worst:

- constant growth - `O(1)` Runtime is constant and does not grow with `n`
- logarithmic growth – `O(log n)` Runtime grows logarithmically in proportion to `n`
- linear growth – `O(n)` Runtime grows directly in proportion to `n`
- superlinear growth – `O(n log n)` Runtime grows in proportion *and* logarithmically to `n`
- polynomial growth – `O(n^c)` Runtime grows quicker than previous all based on `n`
- exponential growth – `O(c^n)` Runtime grows even faster than polynomial growth based on `n`
- factorial growth – `O(n!)` Runtime grows the fastest and becomes quickly unusable for even small values of `n`

(source: Soumyadeep Debnath, *Analysis of Algorithms | Big-O analysis*)

Visualized below; the x-axis representing input size and the y-axis representing complexity:

(source: Wikipedia, *Computational Complexity of Mathematical Operations*)

Big-O notation

Big-O refers to the upper bound of time or space complexity of an algorithm, meaning it worst case runtime scenario. An easy way to think of it is that runtime could be better than Big-O but it will never be worse.

Big-Ω (Big-Omega) notation

Big-Omega refers to the lower bound of time or space complexity of an algorithm, meaning it is the best runtime scenario. Or runtime could worse than Big-Omega, but it will never be better.

Big-θ (Big-Theta) notation

Big-Theta refers to the tight bound of time or space complexity of an algorithm. Another way to think of it is the intersection of Big-O and Big-Omega, or more simply runtime is guaranteed to be a given complexity, such as `n log n`.

What you need to know

- Big-O and Big-Theta are the most common and helpful notations
- Big-O does *not* mean Worst Case Scenario, Big-Theta does *not* mean average case, and Big-Omega does *not* mean Best Case Scenario. They only connote the algorithm's performance for a particular scenario, and all three can be used for any scenario.
- Worst Case means given an unideal input, Average Case means given a typical input, Best case means a ideal input. Ex. Worst case means given an input the algorithm performs particularly bad, or best case an already sorted array for a sorting algorithm.
- Best Case and Big Omega are generally not helpful since Best Cases are rare in the real world and lower bound might be very different than an upper bound.
- Big-O isn't everything. On paper merge sort is faster than quick sort, but in practice quick sort is superior.

Data Structures

Array

Definition

- Stores data elements based on an sequential, most commonly 0 based, index.
- Based on tuples from set theory.
- They are one of the oldest, most commonly used data structures.

What you need to know

- Optimal for indexing; bad at searching, inserting, and deleting (except at the end).
- **Linear arrays**, or one dimensional arrays, are the most basic.
  - Are static in size, meaning that they are declared with a fixed size.
- **Dynamic arrays** are like one dimensional arrays, but have reserved space for additional elements.
  - If a dynamic array is full, it copies its contents to a larger array.
- **Multi dimensional arrays** nested arrays that allow for multiple dimensions such as an array of arrays providing a 2 dimensional spacial representation via x, y coordinates.

Time Complexity

- Indexing: Linear array: `O(1)`, Dynamic array: `O(1)`
- Search: Linear array: `O(n)`, Dynamic array: `O(n)`
- Optimized Search: Linear array: `O(log n)`, Dynamic array: `O(log n)`
- Insertion: Linear array: n/a, Dynamic array: `O(n)`

Linked List

Definition

- Stores data with **nodes** that point to other nodes.
  - Nodes, at its most basic it has one datum and one reference (another node).
  - A linked list *chains* nodes together by pointing one node's reference towards another node.

What you need to know

- Designed to optimize insertion and deletion, slow at indexing and searching.
- **Doubly linked list** has nodes that also reference the previous node.
- **Circularly linked list** is simple linked list whose **tail**, the last node, references the **head**, the first node.
- **Stack**, commonly implemented with linked lists but can be made from arrays too.
  - Stacks are **last in, first out** (LIFO) data structures.
  - Made with a linked list by having the head be the only place for insertion and removal.
- **Queues**, too can be implemented with a linked list or an array.

- Queues are a **first in, first out** (FIFO) data structure.
  - Made with a doubly linked list that only removes from head and adds to tail.

Time Complexity

- Indexing: Linked Lists: `O(n)`
- Search: Linked Lists: `O(n)`
- Optimized Search: Linked Lists: `O(n)`
- Insertion: Linked Lists: `O(1)`

Hash Table or Hash Map

Definition

- Stores data with key value pairs.
- **Hash functions** accept a key and return an output unique only to that specific key.
  - This is known as **hashing**, which is the concept that an input and an output have a one-to-one correspondence to map information.
  - Hash functions return a unique address in memory for that data.

What you need to know

- Designed to optimize searching, insertion, and deletion.
- **Hash collisions** are when a hash function returns the same output for two distinct inputs.
  - All hash functions have this problem.
  - This is often accommodated for by having the hash tables be very large.
- Hashes are important for associative arrays and database indexing.

Time Complexity

- Indexing: Hash Tables: `O(1)`
- Search: Hash Tables: `O(1)`
- Insertion: Hash Tables: `O(1)`

Binary Tree

Definition

- Is a tree like data structure where every node has at most two children.
  - There is one left and right child node.

What you need to know

- Designed to optimize searching and sorting.
- A **degenerate tree** is an unbalanced tree, which if entirely one-sided, is essentially a linked list.
- They are comparably simple to implement than other data structures.
- Used to make **binary search trees**.
  - A binary tree that uses comparable keys to assign which direction a child is.
  - Left child has a key smaller than its parent node.
  - Right child has a key greater than its parent node.
  - There can be no duplicate node.
  - Because of the above it is more likely to be used as a data structure than a binary tree.

Time Complexity

- Indexing: Binary Search Tree: `O(log n)`
- Search: Binary Search Tree: `O(log n)`
- Insertion: Binary Search Tree: `O(log n)`

Algorithms

Algorithm Basics

Recursive Algorithms

Definition

- An algorithm that calls itself in its definition.
  - **Recursive case** a conditional statement that is used to trigger the recursion.
  - **Base case** a conditional statement that is used to break the recursion.

What you need to know

- **Stack level too deep** and **stack overflow**.
  - If you've seen either of these from a recursive algorithm, you messed up.
  - It means that your base case was never triggered because it was faulty or the problem was so massive you ran out of alloted memory.
  - Knowing whether or not you will reach a base case is integral to correctly using recursion.
  - Often used in Depth First Search

Iterative Algorithms

Definition

- An algorithm that is called repeatedly but for a finite number of times, each time being a single iteration.
  - Often used to move incrementally through a data set.

What you need to know

- Generally you will see iteration as loops, for, while, and until statements.
- Think of iteration as moving one at a time through a set.
- Often used to move through an array.

Recursion Vs. Iteration

- The differences between recursion and iteration can be confusing to distinguish since both can be used to implement the other. But know that,
  - Recursion is, usually, more expressive and easier to implement.
  - Iteration uses less memory.
- **Functional languages** tend to use recursion. (i.e. Haskell)
- **Imperative languages** tend to use iteration. (i.e. Ruby)
- Check out this Stack Overflow post for more info.

Pseudo Code of Moving Through an Array

```
Recursion                       | Iteration
--------------------------------|--------------------------------
recursive method (array, n)     | iterative method (array)
  if array[n] is not nil        |   for n from 0 to size of array
    print array[n]              |     print(array[n])
    recursive method(array, n+1)|
  else                          |
    exit loop                   |
```

Greedy Algorithms

Definition

- An algorithm that, while executing, selects only the information that meets a certain criteria.
- The general five components, taken from Wikipedia:
  - A candidate set, from which a solution is created.
  - A selection function, which chooses the best candidate to be added to the solution.
  - A feasibility function, that is used to determine if a candidate can be used to contribute to a solution.

- An objective function, which assigns a value to a solution, or a partial solution.
- A solution function, which will indicate when we have discovered a complete solution.

What you need to know

- Used to find the expedient, though non-optimal, solution for a given problem.
- Generally used on sets of data where only a small proportion of the information evaluated meets the desired result.
- Often a greedy algorithm can help reduce the Big O of an algorithm.

Pseudo Code of a Greedy Algorithm to Find Largest Difference of any Two Numbers in an Array.

```
greedy algorithm (array)
  var largest difference = 0
  var new difference = find next difference (array[n], array[n+1])
  largest difference = new difference if new difference is > largest difference
  repeat above two steps until all differences have been found
  return largest difference
```

This algorithm never needed to compare all the differences to one another, saving it an entire iteration.

Search Algorithms

Breadth First Search

Definition

- An algorithm that searches a tree (or graph) by searching levels of the tree first, starting at the root.
  - It finds every node on the same level, most often moving left to right.
  - While doing this it tracks the children nodes of the nodes on the current level.
  - When finished examining a level it moves to the left most node on the next level.
  - The bottom-right most node is evaluated last (the node that is deepest and is farthest right of it's level).

What you need to know

- Optimal for searching a tree that is wider than it is deep.
- Uses a queue to store information about the tree while it traverses a tree.
  - Because it uses a queue it is more memory intensive than **depth first search**.
  - The queue uses more memory because it needs to stores pointers

Time Complexity

- Search: Breadth First Search: O(V + E)
- E is number of edges
- V is number of vertices

Depth First Search

Definition

- An algorithm that searches a tree (or graph) by searching depth of the tree first, starting at the root.
  - It traverses left down a tree until it cannot go further.
  - Once it reaches the end of a branch it traverses back up trying the right child of nodes on that branch, and if possible left from the right children.
  - When finished examining a branch it moves to the node right of the root then tries to go left on all it's children until it reaches the bottom.
  - The right most node is evaluated last (the node that is right of all it's ancestors).

What you need to know

- Optimal for searching a tree that is deeper than it is wide.
- Uses a stack to push nodes onto.

- Because a stack is LIFO it does not need to keep track of the nodes pointers and is therefore less memory intensive than breadth first search.
  - Once it cannot go further left it begins evaluating the stack.

Time Complexity

- Search: Depth First Search: O(|E| + |V|)
- E is number of edges
- V is number of vertices

Breadth First Search Vs. Depth First Search

- The simple answer to this question is that it depends on the size and shape of the tree.
  - For wide, shallow trees use Breadth First Search
  - For deep, narrow trees use Depth First Search

Nuances

- Because BFS uses queues to store information about the nodes and its children, it could use more memory than is available on your computer. (But you probably won't have to worry about this.)
- If using a DFS on a tree that is very deep you might go unnecessarily deep in the search. See xkcd for more information.
- Breadth First Search tends to be a looping algorithm.
- Depth First Search tends to be a recursive algorithm.

Sorting Algorithms

Selection Sort

Definition

- A comparison based sorting algorithm.
  - Starts with the cursor on the left, iterating left to right
  - Compares the left side to the right, looking for the smallest known item
    - If the left is smaller than the item to the right it continues iterating
    - If the left is bigger than the item to the right, the item on the right becomes the known smallest number
    - Once it has checked all items, it moves the known smallest to the cursor and advances the cursor to the right and starts over
  - As the algorithm processes the data set, it builds a fully sorted left side of the data until the entire data set is sorted
- Changes the array in place.

What you need to know

- Inefficient for large data sets.
- Very simple to implement.

Time Complexity

- Best Case Sort: Merge Sort: `O(n^2)`
- Average Case Sort: Merge Sort: `O(n^2)`
- Worst Case Sort: Merge Sort: `O(n^2)`

Space Complexity

- Worst Case: `O(1)`

Visualization

Insertion Sort

Definition

- A comparison based sorting algorithm.
  - Iterates left to right comparing the current cursor to the previous item.
  - If the cursor is smaller than the item on the left it swaps positions and the cursor compares itself again to the left hand side until it is put in its sorted position.
  - As the algorithm processes the data set, the left side becomes increasingly sorted until it is fully sorted.
- Changes the array in place.

What you need to know

- Inefficient for large data sets, but can be faster for than other algorithms for small ones.
- Although it has an `O(n^2)` , in practice it slightly less since its comparison scheme only requires checking place if its smaller than its neighbor.

Time Complexity

- Best Case Sort: Merge Sort: `O(n)`
- Average Case Sort: Merge Sort: `O(n^2)`
- Worst Case Sort: Merge Sort: `O(n^2)`

Space Complexity

- Worst Case: `O(n)`

Visualization

6   5   3   1   8   7   2   4

Merge Sort

Definition

- A divide and conquer algorithm.

- Recursively divides entire array by half into subsets until the subset is one, the base case.
- Once the base case is reached results are returned and sorted ascending left to right.
- Recursive calls are returned and the sorts double in size until the entire array is sorted.

What you need to know

- This is one of the fundamental sorting algorithms.
- Know that it divides all the data into as small possible sets then compares them.
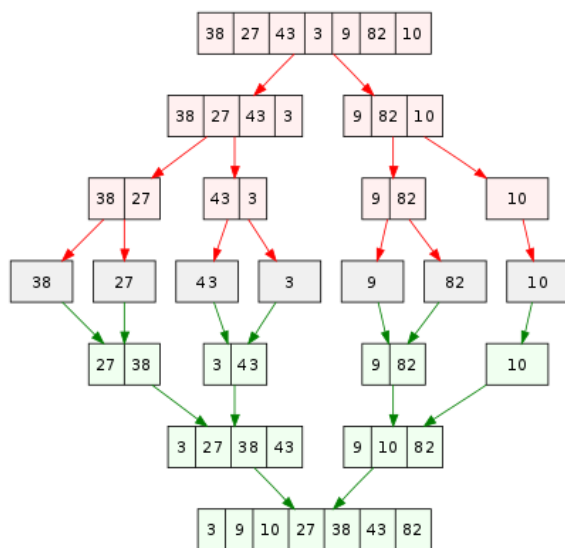
Time Complexity

- Worst Case: `O(n log n)`
- Average Case: `O(n log n)`
- Best Case: `O(n)`

Space Complexity

- Worst Case: `O(1)`

Visualization

(source: Wikipedia, *Merge Sort*)

Quicksort

🔗 **Definition**

- A divide and conquer algorithm
  - Partitions entire data set in half by selecting a random pivot element and putting all smaller elements to the left of the element and larger ones to the right.
  - It repeats this process on the left side until it is comparing only two elements at which point the left side is sorted.
  - When the left side is finished sorting it performs the same operation on the right side.
- Computer architecture favors the quicksort process.
- Changes the array in place.

What you need to know

- While it has the same Big O as (or worse in some cases) many other sorting algorithms it is often faster in practice than many other sorting algorithms, such as merge sort.
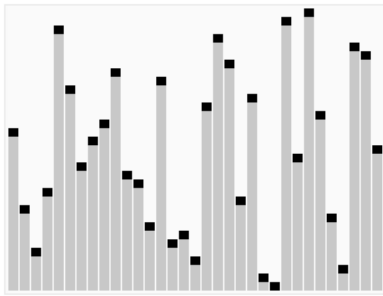
Time Complexity

- Worst Case: `O(n^2)`
- Average Case: `O(n log n)`
- Best Case: `O(n log n)`

Space Complexity

- Worst Case: `O(log n)`

Visualization

[(source: Wikipedia, *Quicksort*)](#)

Merge Sort Vs. Quicksort

- Quicksort is likely faster in practice, but merge sort is faster on paper.
- Merge Sort divides the set into the smallest possible groups immediately then reconstructs the incrementally as it sorts the groupings.
- Quicksort continually partitions the data set by a pivot, until the set is recursively sorted.

Additional Resources

[Khan Academy's Algorithm Course](#)