

## Section 2: Arrays and Big-O Notation

- Time complexity: (In order of speed, fastest to slowest)
  - Constant  $O(1)$
  - Logarithmic  $O(\log N)^{***}$
  - Linear  $O(N)$
  - N Log-Star N  $O(N \log N)^{***}$
  - Quadratic  $O(N^2)$

**\*\*\*These are Log base 2, NOT 10**

Big-O gives a way of comparing time of algorithms in a hardware independent manner.

- Arrays in Memory
  - Arrays are stored contiguously in memory, not like linked lists that can hop around memory. We denote array size at allocation so the computer knows how much space it needs to free up in a row.
  - Each array element is the same size in terms of number of bits.
  - An array of objects holds the references to each instance, like a string array or random type of Object that you created.
  - Because the array is contiguous and the elements are the same size, we can easily index array elements, and get something at a given index instantly.
    - The formula is  $\text{element}[x] = \text{address of element 0} + (x * \text{size of each element in bits})$

## Section 3: Sort Algorithms

1. Bubble Sort
2. Selection Sort
3. Insertion Sort
4. Shell Sort
5. Merge Sort
6. Quick Sort
7. Counting Sort
8. Radix Sort

- **Bubble Sort**
  - It splits the array into a sorted and unsorted partition
  - Swaps adjacent elements and goes through the array as many times as it takes to get them all in order,
  - For loop nested in another, first decrements the end of the sorted partition, second for loop increments the elements and stops at the end of the sorted partition
  - **$O(N^2)$  (an approximation kinda)**
  - **In-place algorithm**
  - **Degrades pretty quickly**
  - **Stable sort algorithm**

- **Stable Vs Unstable Sort Algorithms**

- Comes into play when there's duplicate values in an array- What happens when they're next to each other?
- Unstable sort - they are not in the same relative order WRT each other
- Stable sort - the relative positions WRT each other is preserved
- If sorting objects it gets complicated, integers not so much


- **Selection Sort**

- Traverse array and find largest/smallest and swap it with the last element, then traverse the unsorted partition again and repeat
- **$O(N^2)$  quadratic**
- **In-place algorithm**
- **Only swaps once a traversal**
- **Unstable sort algorithm (swaps largest element in unsorted partition so it doesn't account for duplicates)**

- **Insertion Sort**






- Partition the array into sorted, unsorted
- Grows the sorted partition from the front of the array
- Compare value to be sorted to those in the sorted array, moving right to left
- **$O(N^2)$  quadratic**
- **In-place algorithm**
- **Stable sort algorithm**

- **Shell Sort**

- Variation of insertion sort
- Insertion sort does a gap of 1 to slide things over, shell sort starts with a larger gap and shrinks it as it goes, reducing the number of needed shifts
  - When the gap value hits 1, it's an insertion sort. This will always happen at the end of the sorting process
- Instead of comparing values to a neighbor, it compares them to a group of values within the gap size
- By the time we get to an insertion sort stage, most of the array is sorted
- A common way to find initial gap value is using the knuth sequence
-  Shell Sort Algorithm Example
- **Time complexity depends on the gap calculation**
- **In-place algorithm**
- **Unstable sort algorithm**

- **Merge Sort**

- Divide and conquer array (splitting array into smaller ones)
- Usually written recursively
- First phase is splitting, then merging phase which is where sorting is done

- Splitting leads to faster sorting, is only logical and is not actually done using new array objects
-  Merge sort 
- **$O(N \log N)$  Logarithmic, Quasilinear time**
- **Not in-place, space could be an issue**
- **Stable Algorithm**
- **Usually written recursively with a helper method**
- **Quick Sort**
  - Divide and conquer
  - Recursive
  - Uses a pivot element to split the array
    - Less than the pivot goes to the left, greater than to the right of the pivot element
    - At the end, the pivot is in the right place but everything around it is not sorted with respect to each other. Then we repeat the process.
  -  Quick sort 
  - **In-place**
  - **$O(N \log N)$  Logarithmic, Quasilinear time**
  - **Unstable**
- **Counting Sort**
  - No comparisons
  - Keeps a count of distinct values
  - Only works with non-negative discrete values of integers
  - Values must be within a specific range
  - You count how many of each digit there is, and then you determine each digit's starting position by counting how many cells are taken up by the digits before it
  -  Counting Sort Explained and Implemented with Examples in Java | Sorting ...
  - **Not in-place**
  - **$O(N)$**
  - **Could be stable**
- **Radix Sort**
  - Radix is the number of unique values in a numbering system of alphabet
  - Data has to have the same radix and width
  - Same logic as counting sort, but accounts for bigger range of values
  - LSD vs MSD Radix sort just starts on different significant digits of the given value
  - Do a counting sort on each digit in the value
  - Can't be used on floats
  - **Not in-place**
  - **$O(N)$**
  - **Stable**

## **Section 4: Lists**

- **Abstract Data Types**
  - Usually an interface
  - Is the data walking around with methods or behavior of that data
- **Vectors and Array Lists**
  - Pretty much the same except vectors are synchronized
- **Linked List Basic Operations**
  - Insertion
    - addFirst
    - addLast
    - Insert
  - Deletion
    - removeFirst
    - removeLast
    - Remove
  - Search
  - Update
    - Setters
  - Sorting

## **Section 5: Stacks**

- LIFO
  - No random access allowed!
- Push - adds to top of stack
- Pull - removes from top of stack
- Peak - get the top item in the stack
- Ideal implementation is using linked lists

## **Section 6: Queues**

- FIFO
  - No random access allowed!
- Push/ Enqueue- adds to back of the queue
- Pull/ Dequeue - removes from end of the queue
- Peak - get the next item at the end of the queue
- Ideal implementation is using linked lists

## **Section 7: Hash Tables**

- A collection of key:value pairs
- Each pair is called an "entry"
- Key doesn't have to be an integer
- Give the hashtable the key, it gives you the value.
- Optimized for retrieval when you know the key
- AKA Dictionaries, Maps

- Under the hood, the keys are being converted to integers using a hashing function
  - So hashing functions maps keys to int
- A collision is when the hashing function maps two keys to the same integer and there's ways to work with this
- Load factor is the size/capacity
  - When load factor reaches a certain number, a reSize is done
- If the load factor is too low there will be a lot of empty space
- If the load factor is too high it can increase the likelihood of collisions
- Load factor plays a role in finding time complexity
- Methods
  - Put
  - Get
  - Remove
  - containsKey
  - Size
- **Bucket Sort Algorithm**
  - Uses hashing
  - A kind of generalized counting sort
  - $O(N)$  time
  - Performs best with less collisions or the data is evenly distributed.
  - Scatter Phase - puts things in buckets
  - Sort Phase - sort items in each bucket
  - Gathering Phase - merge th buckets
  - Values in bucket  $X$  must be greater than values in bucket  $X - 1$ , and less than values in bucket  $X + 1$ 
    - The hash function must meet this requirement
  - **Not in-place**
  - **$O(N)$  if one item in each bucket**
  - **Could be stable**

## Section 9: Trees

- **Trees**
  - Hierarchical data structure
  - Each node can only have one parent
  - Can have as many children as you want
  - The root is at the tip top and doesn't have a parent
  - Each class in java has one parent, but a class can have as many classes extending it as you want. All java classes have one root, the Object class.
  - File systems are just like trees. Each folder can only have one parent folder. Everything has the same root, the drive it's all located on.
  - Leaf nodes are the nodes without any children.
  - Every link to another node is called an edge, always goes from parent to child

- There are subtree: A node and all of its descendants.
- A path is the sequence of nodes required to go from one to another
- Height of a node is the number of edges between it and its furthest leaf.
- A node is an ancestor of another if it's in the other node's path.
- **Binary Tree**
  - Every node has 0, 1, or 2 children
  - Left and right child
  - Complete tree
    - Every level is filled (has 2 children) except the last level, and the leafs on the last levels are left justified as much as possible
  - Full tree
    - Every level is filled (has 2 children), even the last
- **Binary Search Tree**
  - Can perform insertions, deletions, retrievals in  $O(\log N)$  time
  - Left child is always smaller than parent
  - Right child is always larger than parent
  - Left Children < Root < Right Children
  - Duplicates usually aren't allowed if they are, counters can be used for values or you can store all equal values on one side of the parent so Left Children  $\leq$  Root < Right Children OR Left Children < Root  $\leq$  Right Children
  - Order of nodes is influenced by order of insertions
  - Max and Min values are on the right end and left ends, respectively
  - Inserting sorted data into an empty tree, you will make a linked list
- **Traversals**
  - Level: visit nodes on each level
  - Pre-order: visit the root of every subtree first
  - Post-order: visit the root of every subtree last. Go to the furthest leaf first then
  - In-order: Start with the left side's furthest leaf then move to the right.

## **Section 10: Heaps**

- A complete binary tree, must satisfy heap property
- Complete tree
  - Every level is filled (has 2 children) except the last level, and the leafs on the last levels are left justified as much as possible
- Heap Properties
  - Max heap: every parent  $\geq$  their children, max value at root
  - Min heap: every parent  $\leq$  their children, min value at root
- Usually implemented with arrays
- Children are added at each level from left to right
- Heapify: making a binary tree into a heap

- Usually done after an insertion or delete in a heap that changes the tree such that it now violates the heap property. Heapifying after this makes it a heap once again.
- In arrays:
  - To find children of Node at index  $i$ :
    - Left child:  $2*i+1$
    - Right child:  $2*i+2$
  - To find parent of Node at index  $i$ :
    - Parent:  $\text{floor}((i-1)/2)$
  - Always add new values to the end of the array, then fix heap (heapify)
    - Compare the new item with parent and swap if its greater, rinse and repeat
  - Deletion:
    - Must choose a replacement value, will take the rightmost leaf at bottom level's value so the tree remains a Complete Tree, then heapify
    - When replacement value is  $>$  parent, fix heap above, if not fix heap below
  - $O(\text{Log}N)$  to insert in worst case
  - $O(\text{Log}N)$  to delete in worst case
  - Heap isn't the best for random access
- **Priority Queue**
  - Rather than always remove in order they were added, we attach a priority to it and remove based on that priority
  - A value with the highest priority will be placed at the root, so removal is always instant at position 0.
  - Reorganizing the queue after removal moves the next highest priority item up to the root position 0.
- **Heap Sort Algorithm**
  - Swap element with last element in the array
  - Heapify the tree, excluding last node
  - After heapify, second largest element is at the root
  - Rinse and repeat
  - $O(N\text{Log}N)$
  - In-place
  - Using heapsort destroys the heap condition and operation speed on things will change
- **Set**
  - A collection with no duplicate values and up to one null element