DS & Algorithms

1. Big O notation:

 Define "time complexity" and "space complexity"

* Timing our code:
  + Check which one is better,

Graphical user interface, text, application, website

Description automatically generated

* + What does better mean?
    - Faster?
    - Less memory-intensive?
    - More readable?
  + Mostly we focus on the first two. Readable is also important but first two are most.
  + We cannot time our code to find the best algorithms, because if we time our code, everytime runs it will give different value even in the same machine.
  + Refer below image,
  + Graphical user interface, text

    Description automatically generated
  + The Problem with Time
    - Different machines will record different times
    - The same machine will record different times!
    - For fast algorithms, speed measurements may not be precise enough?
  + If not time, then what?
    - Rather than counting seconds, which are so variable...
    - Let's count the number of simple operations the computer has to perform!

Diagram

Description automatically generated

A picture containing diagram

Description automatically generated

* + Counting is hard!
    - Depending on what we count, the number of operations can be as low as 2n or as high as 5n + 2
    - But regardless of the exact number, the number of operations grows roughly proportionally with n
    - If n doubles, the number of operations will also roughly double
  + Big O Definition
    - We say that an algorithm is O(f(n)) if the number of simple operations the computer has to do is eventually less than a constant times f(n), as n increases
    - f(n) could be linear (f(n) = n)
    - f(n) could be quadratic (f(n) = n )
    - f(n) could be constant (f(n) = 1)
    - f(n) could be something entirely different!
  + We just count the number of operations with respect to n. don’t need to count the number of n. lets say 5n = n. Refer below

A picture containing graphical user interface

Description automatically generated

* Simplifying the Big O
* A picture containing pie chart

  Description automatically generated

A picture containing table

Description automatically generated

* Big O Shorthands
  + Arithmetic operations are constant
  + Variable assignment is constant
  + Accessing elements in an array (by index) or object (by key) is constant
  + In a loop, the the complexity is the length of the loop times the complexity of whatever happens inside of the loop
  + A picture containing timeline

    Description automatically generated
  + A picture containing line chart

    Description automatically generated
* Space Complexity
* So far, we've been focusing on time complexity: how can we analyze the runtime of an algorithm as the size of the inputs increases?
* We can also use big O notation to analyze space complexity: how much additional memory do we need to allocate in order to run the code in our algorithm?
* What about the inputs?
  + Sometimes you'll hear the term **auxiliary space complexity** to refer to space required by the algorithm, not including space taken up by the inputs.
  + Unless otherwise noted, when we talk about space complexity, technically **we'll be talking about auxiliary space complexity**.
* Space Complexity in JS - Rules of Thumb
  + Most primitives (booleans, numbers, undefined, null) are constant space
  + Strings require O(n) space (where n is the string length)
  + Reference types are generally O( n), where n is the length (for arrays) or the number of keys (for objects)
* Example 1 –
  + refer below one, we have two variable assignment so O(2) so it become O(1). We don’t need to consider **total +=** which is inside the loop, because its just adding numbers. Numbers are constant space in JS
  + Diagram

    Description automatically generated with medium confidence
  + Example 2
    - Refer below, two variable assignment i=0 and the newArr.
    - newArr length will be directly proportional to given input.
    - So O(n + 1) = O(n)
  + Text

    Description automatically generated with medium confidence
* Logarithms
  + We've encountered some of the most common complexities: O(1), O(n), O(n^2)
  + Sometimes big O expressions involve more complex mathematical expressions
  + One that appears more often than you might like is the logarithm!
  + Diagram

    Description automatically generated
  + so here's a rule of thumb.
  + The logarithm of a number roughly measures the number of times you can divide that number by 2 before you get a value that's less than or equal to one.
  + A picture containing diagram

    Description automatically generated
  + A picture containing chart

    Description automatically generated
  + Who Cares?
    - Certain searching algorithms have logarithmic time complexity.
    - Efficient sorting algorithms involve logarithms.
    - Recursion sometimes involves logarithmic space complexity.
    - ...and more!
* **Recap**
  + To analyze the performance of an algorithm, we use Big O Notation
  + Big O Notation can give us a high level understanding of the time or space complexity of an algorithm
  + Big O Notation doesn't care about precision, only about general trends (linear? quadratic? constant?)
  + The time or space complexity (as measured by Big O) depends only on the algorithm, not the hardware used to run the algorithm
  + Big O Notation is everywhere, so get lots of practice!

1. Analyzing the performance of arrays & objects:
   * Objects:
     + When to use objects
       - When you don't need order
       - When you need fast access / insertion and removal
     + Big O of Object
       - Insertion - O(1)
       - Removal - O(1)
       - Searching - O(N)
       - Access - O(1)
       - When you don't need any ordering, objects are an excellent choice!
     + Big O of Object Methods
       - Object.keys -   O(N)
       - Object.values -   O(N)
       - Object.entries -   O(N)
       - hasOwnProperty -   O(1)
   * Arrays:
     + WHEN TO USE ARRAYS
       - When you need order
       - When you need fast access / insertion and removal (sort of....)
     + Big O of Arrays
       - Insertion - It depends....
       - Removal - It depends....
       - Searching - O(N)
       - Access - O(1)
2. Problem solving approach:
   * PROBLEM SOLVING
     + Understand the Problem
     + Explore Concrete Examples
     + Break It Down
     + Solve/Simplify
     + Look Back and Refactor
   * UNDERSTAND THE PROBLEM
     + Can I restate the problem in my own words?
     + What are the inputs that go into the problem?
     + What are the outputs that should come from the solution to the problem?
     + Can the outputs be determined from the inputs? In other words, do I have enough information to solve the problem? (You may not be able to answer this question until you set about solving the problem. That's okay; it's still worth considering the question at this early stage.)
     + How should I label the important pieces of data that are a part of the problem?
   * EXPLORE EXAMPLES
     + Coming up with examples can help you understand the problem better
     + Examples also provide sanity checks that your eventual solution works how it should
   * EXPLORE EXAMPLES
     + Start with Simple Examples
     + Progress to More Complex Examples
     + Explore Examples with Empty Inputs
     + Explore Examples with Invalid Inputs

(Refer section 4, no.20 for real time example.)

* + BREAK IT DOWN
    - Explicitly write out the steps you need to take.
    - This forces you to think about the code you'll write before you write it, and helps you catch any lingering conceptual issues or misunderstandings before you dive in and have to worry about details (e.g. language syntax) as well.
    - Writing comments to break down the problem is very important, even cannot finish the problem, but our steps show how we are approaching the problem.
    - Refer below example
      * Write a function which takes in a string and returns counts of each character in the string.(Refer section 4, no.20 for real time example.)

(Refer section 4, no.21 for this example problem.)

* + SIMPLIFY
    - Find the core difficulty in what you're trying to do
    - Temporarily ignore that difficulty
    - Write a simplified solution
    - Then incorporate that difficulty back in
    - (Refer section 5, Step.4 - for this example problem.)
  + Refactor questions to ask once completed:
    - Can you check the result?
    - Can you derive the result differently?
    - Can you understand it at a glance?
    - Can you use the result or method for some other problem?
    - Can you improve the performance of your solution?
    - Can you think of other ways to refactor?
    - How have other people solved this problem?

1. Problem solving patterns:
   * There are many patterns. Some patterns are
     + Frequency Counter
     + Multiple Pointers
     + Sliding Window
     + Divide and Conquer
     + Dynamic Programming
     + Greedy Algorithms
     + Backtracking
     + Many more!
   * Frequency Counter Pattern:
     + This pattern uses objects or sets to collect values/frequencies of values. This can often avoid the need for nested loops or O(N^2) operations with arrays / strings
     + When compare two string or array, instead of doing the nested loop, try to break down each string/array into object, then compare two object which will be more efficient.
     + **Any time you have multiple pieces of data and you need to compare them in particular, if you need to see if they consist of the same individual pieces, whether it's anagrams or you're trying to see if they're if one array is equal to another array squared, each individual piece, etc. If you need to tell if numbers consist of the same digits just in a different order. Then we can use this frequency counter pattern.**
     + Refer the *codingproblem* repo ***FrequencyCounterPattern*** directory files file for implementation.
   * Multiple Pointer Pattern:
     + Creating pointers or values that correspond to an index or position and move towards the beginning, end or middle based on a certain condition. Very efficient for solving problems with minimal space complexity as well
     + When need to find something in single array/string which need to compare each element with the other elements in the string/array then we can use this pattern.
     + Refer the *codingproblem* repo ***MultiplePointerPattern*** directory files file for implementation.
       - **Note: sumZero.js file solution is only works for sorted array.**
   * Sliding Window Pattern:
     + This pattern involves creating a window which can either be an array or number from one position to another. Depending on a certain condition, the window either increases or closes (and a new window is created). **Very useful for keeping track of a subset of data in an array/string etc.**
     + When need to check the consecutive/subset of items in the array or string, instead of adding/chekcing again and again, we simply remove the first one and add the next item in the window. Just move the window.
     + Refer the udemy video - Section 5 -> 33.Sliding window pattern, maxSubarraySum problem.
   * Divide & Conquer:
     + This pattern involves dividing a data set into smaller chunks and then repeating a process with a subset of data.
     + This pattern can tremendously decrease time complexity.
     + This pattern is commonly used for really large and complex data. Not used for a simple pattern like finding a unique character or something.
2. Recursion:

* A process (a function in our case) that calls itself.
* Bonus point :
  + The tool does JavaScript use to manage function invocations is Call Stack.
  + It’s a stack data structure
* How recursive functions work:
  + Invoke the same function with a different input until you reach your base case!
  + Base Case: The condition when the recursion ends. This is the most important concept to understand
* Two essential parts of a recursive function!
  + Base Case – Condition when recursion ends.
  + Different Input – Every time, should call the function with different input.
* Where things go wrong
  + No base case
  + Forgetting to return or returning the wrong thing!
  + Stack overflow – Meaning stacks are never-ending; keep going.
* Helper method recursion
  + If want to collect the list into an array, then can use the helper method recursion. It’s not the only way, but its straight forward.
  + Refer to the git repo **Recursion&Loop** directory ***collectOddNumbersInArray***.***js*** file
* 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
  + Remember that 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
* Practice all the problems in the recursion problem sets sections.
* Recap
  + A recursive function is a function that invokes itself
  + Your recursive functions should always have a base case and be invoked with different input each time
  + When using recursion, it's often essential to return values from one function to another to extract data from each function call
  + Helper method recursion is an alternative that allows us to use an external scope in our recursive functions
  + Pure recursion eliminates the need for helper method recursion, but can be trickier to understand at first

1. Searching Algorithms:
   * Objective of this section:
     + Describe what a searching algorithm is
     + Implement linear search on arrays
     + Implement binary search on sorted arrays
     + Implement a naive string searching algorithm
     + Implement the KMP string searching algorithm
   * Linear Search:
     + Given an array, the simplest way to search for an value is to look at every element in the array and check if it's the value we want.
     + Big O(n)
       - Best case O(1) – if the value is found in the very first place.
       - Worst case O(n) – if the value is found in the last place.
       - Average O(n) – average also O(n) since we are omitting constants.
   * Binary Search:
     + Binary search is a much faster form of search
     + Rather than eliminating one element at a time, you can eliminate half of the remaining elements at a time
     + Binary search only works on sorted arrays!
     + Binary Search Pseudocode
       - This function accepts a sorted array and a value
       - Create a left pointer at the start of the array, and a right pointer at the end of the array
       - Edge case
         * if the array is empty
         * Since this is sorted array arr[left] > num || arr[right] < num -> return -1
       - While the left pointer comes before the right pointer:
         * Create a pointer in the middle
         * If you find the value you want, return the index
         * If the value is too small, move the left pointer up
         * If the value is too large, move the right pointer down
       - If you never find the value, return -1
     + Big O of Binary search
       - O(log n) - Worst and Average Case
       - O(1) - Best Case
       - O(log n) is very good than O(n).
       - When we look at the chart below, log n is very close to O(1).
       - A picture containing line chart

         Description automatically generated
       - Why log n ?
         * If we take the array length of 16 elements, it will take 4 step to find. Refer to the below screenshot  
           Graphical user interface, text

           Description automatically generated
         * If we double the size of array, if we provide array length of 32, function takes 5 steps to complete. Refer below,  
           A picture containing table

           Description automatically generated
         * So, the function takes only one extra step even if we double the array size. That’s why big O is log n.
   * String search:
     + Naïve search:
       - Suppose you want to count the number of times a smaller string appears in a longer string
       - A straightforward approach involves checking pairs of characters individually.
       - Big O – O(mn) – quadradic time complexity.
     + KMP:
       - Reference link - <https://www.youtube.com/watch?v=GTJr8OvyEVQ&t=53s>
       - The Knutt-Morris-Pratt algorithm offers an improvement over the naive approach
       - Published in 1977
       - This algorithm more intelligently traverses the longer string to reduce the amount of redundant searching
       - Prefixes and Suffixes
         * In order to determine how far we can shift the shorter string, we can pre-compute the length of the longest (proper) suffix that matches a (proper) prefix
         * This tabulation should happen before you start looking for the short string in the long string
       - Pseudo code:
         * Create function accepting two param
         * Edgecase - if length of any string is 0 then return.
         * Step 1:
         * Create a temp array from the pattern string.
         * while loop with two 2 points i & j
         * i start with 1. j start with 0
         * temp array 0th index is always 0.
         * compare i & j,

if there is a match get the current index of j and add 1, store this in the ith place the temp array. and increase 1 in j & i

if there is no match

j != 0, get the previous index value of j and j should jump to that place and compare again. This will continue until the i goes to last element.

j = 0, then set the tempArray[i]=0, increase i to 1

* + - * + Step 2:
        + while loop - with two pointer textPointer & patternpointer.
        + Now compare the long string with the pattern.
        + if there is match then increase both pointer by 1.
        + if there is no match

if patternPointer = 0 then increase the textPointer

if patternPointer != 0, then get patternPointer's previous index of tempArray. tempArray[patternPointer-1] and move the patternpointer to that value and loop again

* + - * Big O of KMP is O(n+m). Much better than brut*al* force or naïve solution.
  + **Big O of Search Algorithms**
    - Linear Search - O(n)
    - Binary Search - O(log n)
    - Naive String Search - O(nm)
    - KMP - O(n + m) time, O(m) space
  + Recap
    - Searching is a very common task that we often take for granted
    - When searching through an unsorted collection, linear search is the best we can do
    - When searching through a sorted collection, we can find things very quickly with binary search
    - KMP provides a linear time algorithm for searches in strings

1. Sorting Algorithms – Bubble Sorting:
   * What is sorting?
     + Sorting is the process of rearranging the items in a collection (e.g. an array) so that the items are in some kind of order.
     + Examples
     + Sorting numbers from smallest to largest
     + Sorting names alphabetically
     + Sorting movies based on release year
     + Sorting movies based on revenue
   * Why do we need to learn this?
     + Sorting is an incredibly common task, so it's good to know how it works
     + There are many different ways to sort things, and different techniques have their own advantages and disadvantages
     + Sorting sometimes has quirks, so it's good to understand how to navigate them
   * Pseudocode :
     + Start looping from with a variable called i the end of the array towards the beginning
     + Start an inner loop with a variable called j from the beginning until i - 1
     + If arr[j] is greater than arr[j+1], swap those two values!
     + Return the sorted array
   * Some sorting algorithms are faster than others in a particular case/based on the data set. Refer to this link for <https://www.toptal.com/developers/sorting-algorithms>
   * **All the algorithms are doing the roughly same thing, which is sorting. Refer to the link above; in the interview, based on the data set/use case, we should select the correct sorting algorithms**.
   * Bubble Sorting - Big O(n)
     + O(nsquare) since its comparing each values to others.
     + **If the array is almost sorted, may be with the swap check (check the coding bubbleSorting.js), this would be ok to use, but don’t use this unless really wanted.**
     + **Mostly its not used in the real world**
2. Selection Sort:
   * Comparing each element in the array with all the other element to find the lowest value and then swap that lowest value to the first index and start over the loop again.
   * Like this compares each element and find the lowest value every time, and store in the first index.
   * Pseudo code:
     + Function with accept array of numbers.
     + Store the first element is the smallest one.
     + Compare the smallest element with all the other elements, until finding= the smallest element.
     + If smallest is found then swap this into first index.
     + If the first element is the smallest of others then no need to swap at the end.
     + Then start loop from the second index, and do the comparison again to find the next smalles one and store it in the second index. And repeat again.
     + Return the array.
   * Big O of selection sort:
     + O(n²)
     + I**ts similar to the bubble sort, only advantage here is the swaps are lesser compared to the bubble sort.**
3. Insertion Sorting:
   * Graphical user interface, application

     Description automatically generated
   * Pseudo Code:
     + Start by picking the second element in the array
     + Now compare the second element with the one before it and swap if necessary.
     + Continue to the next element and if it is in the incorrect order, iterate through the sorted portion (i.e. the left side) to place the element in the correct place.
     + Repeat until the array is sorted.
   * Big O(n)
     + O(n²) – worst case. If the array is almost sorted then time complexity will be reduce
     + I**ts similar to the bubble & selection sort. This algorithm will be useful when we receiving the value one at a time into an array and we need to put this in a correct sorting place, in this secenario it will be useful. Because all the values in the left side are already sorted, so this algorithm will be quite faster.**
4. Big O of Sorting Algorithms (Bubble, Insertion, Selection):
   * Table

     Description automatically generated
   * These three are sometimes called as quadratic algorithms or elementary algorithms.
   * Worst case – O(n²)
   * **If the array is nearly sorted, then the Bubble & Insertion sorting time complexity is O(n).**
   * **Insertion algorithm will be useful when we receiving the value one at a time into an array and we need to put this in a correct sorting place, in this secenario it will be useful. Because all the values in the left side are already sorted, so this algorithm will be quite faster compared to other two.**
   * <https://www.toptal.com/developers/sorting-algorithms> - comparing all the algorithms speed.
   * **These three algorithms are good for small set of data like 20/30 items in the array, but it don’t scale well. It will take quite some time if array contains thousands & thousands of items.**
5. **Merge Sorting**
   * WHY LEARN THIS?
     + The sorting algorithms we've learned so far don't scale well
     + Try out bubble sort on an array of 100000 elements, it will take quite some time!
     + We need to be able to sort large arrays more quickly
     + There is a family of sorting algorithms that can improve time complexity from O(n ) to O(n log n)
     + There's a tradeoff between efficiency and simplicity
     + The more efficient algorithms are much less simple, and generally take longer to understand
   * What is MERGE SORT:
     + It's a combination of two things - **merging and sorting**!
     + Exploits the fact that arrays of 0 or 1 element are always sorted
     + Works by decomposing an array into smaller arrays of 0 or 1 elements, then building up a new sorted array.

A picture containing text

Description automatically generated

* + Merging Arrays:
    - In order to implement merge sort, it's useful to first implement a function responsible for merging two sorted arrays
    - Given two arrays which are sorted, this helper function should create a new array which is also sorted, and consists of all of the elements in the two input arrays
    - This function should run in **O(n + m)** time and **O(n + m)** space and **should not** modify the parameters passed to it.
  + Merging Arrays Pseudocode:
    - Create an empty array, take a look at the smallest values in each input array
    - While there are still values we haven't looked at...
    - If the value in the first array is smaller than the value in the second array, push the value in the first array into our results and move on to the next value in the first array
    - If the value in the first array is larger than the value in the second array, push the value in the second array into our results and move on to the next value in the second array
    - Once we exhaust one array, push in all remaining values from the other array
  + **MergeSort Pseudocode**
    - Break up the array into halves until you have arrays that are empty or have one element
    - Once you have smaller sorted arrays, merge those arrays with other sorted arrays until you are back at the full length of the array
    - Once the array has been merged back together, return the merged (and sorted!) array
  + Graphical user interface, application

    Description automatically generated
  + Big O of MergeSort:
    - Table

      Description automatically generated
    - How the O(nlogn) is derived?
      * There are two operations in this algorithm, Splitting & Merging.
      * For splitting, if the array length is 8(8->4->2->1)( 2 power of 3), then it need to be split into a single item array in 3 steps. If the array length is 32, then it need to be split to single item array in 5 steps(32->16->8->4->2->1)(2 power of 5). So always, Refer to the pic below
      * Text

        Description automatically generated with medium confidence
      * There are three splits for 8. From below log screenshot, log(8) = 2³.
      * Diagram

        Description automatically generated
      * That’s how this logn is derived
      * For split*ting* its logn.
      * But for the merging, algorithm needs to compare each element in the array to find the minimum value to push into the new array, so its O(n) time complexity.
      * Text

        Description automatically generated
      * **So, finally O(nlogn) time complexity.**
      * **Space complexisty is O(n).** Because creating new array to store the sorted value while doing the merging. So its depends on the array length.
      * **But in the Bubble, Insertion & Selection sorting, the space complexity is O(1), if in any situation, the time complexity is not a problem but a space is, then can consider the one of these elementary/quadratic sorting algorithm.**
      * **Best possible time complexity can do for sorting is O(nlogn)**

1. **Quick Sort**
   * Like merge sort, exploits the fact that arrays of 0 or 1 element are always sorted
   * Works by selecting one element (called the "pivot") and finding the index where the pivot should end up in the sorted array
   * Once the pivot is positioned appropriately, quick sort can be applied on either side of the pivot
   * The recursion technique is used.
   * There are two parts: one is the decomposition of array and sorting.
   * Create one helper function to select the pivot and find the correct index after being placed in the correct position.
   * Pivot Helper:
     + In order to implement merge sort, it's useful to first implement a function responsible arranging elements in an array on either side of a pivot
     + Given an array, this helper function should designate an element as the pivot
     + It should then rearrange elements in the array so that all values less than the pivot are moved to the left of the pivot, and all values greater than the pivot are moved to the right of the pivot
     + The order of elements on either side of the pivot doesn't matter!
     + The helper should do this **in place**, that is, it should not create a new array
     + When complete, the helper should return the index of the pivot
   * Picking a pivot
     + The runtime of quick sort depends in part on how one selects the pivot
     + Ideally, the pivot should be chosen so that it's roughly the median value in the data set you're sorting
     + For simplicity, we'll always choose the pivot to be the first element (we'll talk about consequences of this later)
   * Pivot Pseudocode
     + It will help to accept three arguments: an array, a start index, and an end index (these can default to 0 and the array length minus 1, respectively)
     + Grab the pivot from the start of the array
     + Store the current pivot index in a variable (this will keep track of where the pivot should end up)
     + Loop through the array from the start until the end
     + If the pivot is greater than the current element, increment the pivot index variable and then swap the current element with the element at the pivot index
     + Swap the starting element (i.e. the pivot) with the pivot index
     + Return the pivot index
   * Quicksort Pseudocode
     + Call the pivot helper on the array
     + When the helper returns to you the updated pivot index, recursively call the pivot helper on the subarray to the left of that index, and the subarray to the right of that index
     + Your base case occurs when you consider a subarray with less than 2 elements
   * Big O of QuickSort:
     + Table

       Description automatically generated
     + **For best & average its same as the mergesort.**
     + Text

       Description automatically generated with low confidence  
       A picture containing text

       Description automatically generated

A picture containing chart

Description automatically generated

Scatter chart

Description automatically generated with medium confidence

Chart

Description automatically generated

* + - **WORST CASE:**
    - **But for the worst case, it’s a quadradic time complexity, its because of choosing the pivot point.**
    - **Its not always correct way to pickup the first element as the pivot. Because if the array is already sorted, then if we pick the first element means, it’s a minimum number, so our code has to go through all the items in the array every time. So its quatradic time complexity. Refer to the below pic,**
    - **Text

      Description automatically generated**
    - **If the array is sorted, if we pick the first item, need to loop through every items.**
    - **A picture containing text

      Description automatically generated**
    - **Chart

      Description automatically generated**
    - **A picture containing chart

      Description automatically generated**
    - **Hence, mergeSort is better in all the cases than quicksort.**

1. Radix Sort:
   * Below are the comparison based sorting,
     + Bubble Sort - O(n^2)
     + Insertion Sort - O(n^2)
     + Selection Sort - O(n^2)
     + Quick Sort - O(n log (n))
     + Merge Sort - O(n log (n))
   * But Radix is not the comparison sorting; it doesn’t compare two items, which is larger and smaller, instead moving into buckets and regrouping and repeat this.
   * Its better than the other sorting.
   * **If the array contains +ve & -ve numbers, It doesn’t sort the -ve number. Consider everything as +ve numbers.**
   * **Radix Sort**:
     + Radix sort is a special sorting algorithm that works on lists of numbers.
     + It exploits the fact that information about the size of a number is encoded in the number of digits.
     + More digits mean a bigger number!
     + First need to find the maximum digits that the largest number has.
     + Loop through using that digit i.
     + Create a bucket with 10 empty subarrays.
     + Another loop for a given array.
     + Get the ith position of the each element in the array and put this item based on the ith position into the bucket array.
     + Then concat the buckets elements in the same order.
     + This will repeat until the i finishes.
   * To implement this RadixSort, we can implement two helper functions that will help to look our code clean.
   * Maximum Digit Helper – This is to find the maximum digits in the largest number in the array
     + Function accepts the array.
     + Add new variable with maxNumber = 0.
     + Loop through the array and find the digits in each element and compare with maxNumber.
   * Get ith position digit Helper – This is to find the ith position digit of the number.
     + Function accepts the integer, and the i
     + Get the ith position of the element from the given integer but starts from reverse order.
   * RadixSort Pseudocode:
     + Define a function that accepts list of numbers
     + Figure out how many digits the largest number has
     + Loop from *k* = 0 up to this largest number of digits
     + For each iteration of the loop:
     + Create buckets for each digit (0 to 9)
     + place each number in the corresponding bucket based on its *k*th digit
     + Replace our existing array with values in our buckets, starting with 0 and going up to 9
     + return list at the end!
   * Big O of RadixSort:
     + Table

       Description automatically generated
     + In the above screenshot, n is the length of the array and the k is the number of digits, so its better than the other sorting O(nlogn).
     + **If the digits in the number is really long, then we need to consider this k here, its not always constant. It will significantly impact the time.**
     + **But there are some controversial,**
     + **In the Wikipedia Radixsort page, under the efficiency section, there are two arguments, its based on how the numbers are stored in the memory of the computer, if the all the n values are distinct, then the k is alteast logn. So in this case, O(nlogn), then it’s a same as the other comparison algorithms.**
     + **But in the most of the cases, textbooks, cheatsheets, RadixSort is O(nk.)**

**Data Structure**

1. Singly Linked List:
   * A data structure that contains a head, tail and length property.
   * Linked Lists consist of nodes, and each node has a value and a pointer to another node or null
   * Visualize - <https://visualgo.net/en/list?slide=1>

Diagram

Description automatically generated with medium confidence

Graphical user interface, text, application

Description automatically generated

* + Push function:
    - Adding a new **node**to the end of the Linked List!
    - Pseudocode:
      * This function should accept a value
      * Create a new node using the value passed to the function
      * If there is no head property on the list, set the head and tail to be the newly created node
      * Otherwise, set the next property on the tail to be the new node and set the tail property on the list to be the newly created node
      * Increment the length by one
      * Return the linked list
  + Pop
    - Removing a **node**from the end of the Linked List!
    - Pseudocode:
      * If there are no nodes in the list, return undefined
      * Loop through the list until you reach the tail
      * Set the next property of the 2nd to last node to be null
      * Set the tail to be the 2nd to last node
      * Decrement the length of the list by 1
      * Return the value of the node removed
  + Shift
    - Removing a new **node**from the beginning of the Linked List!
    - Pseudocode
      * If there are no nodes, return undefined
      * Store the current head property in a variable
      * Set the head property to be the current head's next property
      * Decrement the length by 1
      * Return the value of the node removed.
  + Unshift
    - Adding a new **node**to the beginning of the Linked List!
    - Pseudocode:
      * This function should accept a value
      * Create a new node using the value passed to the function
      * If there is no head property on the list, set the head and tail to be the newly created node
      * Otherwise set the newly created node's next property to be the current head property on the list
      * Set the head property on the list to be that newly created node
      * Increment the length of the list by 1
      * Return the linked list
  + Get
    - Retrieving a **node**by it's position in the Linked List!
    - Pseudocode:
      * This function should accept an index
      * If the index is less than zero or greater than or equal to the length of the list, return null
      * Loop through the list until you reach the index and return the node at that specific index
  + Set
    - Changing the **value** of a node based on it's position in the Linked List
    - Pseudocode
      * This function should accept a value and an index
      * Use your get function to find the specific node.
      * If the node is not found, return false
      * If the node is found, set the value of that node to be the value passed to the function and return true
  + Insert
    - Adding a node to the Linked List at a **specific**position
    - Pseudocode
      * If the index is less than zero or greater than the length, return false
      * If the index is the same as the length, push a new node to the end of the list
      * If the index is 0, unshift a new node to the start of the list
      * Otherwise, using the get method, access the node at the index - 1
      * Set the next property on that node to be the new node
      * Set the next property on the new node to be the previous next
      * Increment the length
      * Return true
  + Remove
    - Removing a node from the Linked List at a **specific**position
    - Pseudocode
      * If the index is less than zero or greater than the length, return undefined
      * If the index is the same as the length-1, pop
      * If the index is 0, shift
      * Otherwise, using the get method, access the node at the index - 1
      * Set the next property on that node to be the next of the next node
      * Decrement the length
      * Return the value of the node removed
  + Reversing
    - Reversing the linked list **in place**. **Without duplicating it.**
    - Pseudocode
      * Set the tail as the current head.
      * Create a variable called prev and assign head to it.
      * Create a variable next and assign head.next.
      * Loop through the list.
      * Create a variable current and set this next.
      * Make next = current.nexxt
      * Set the current.next = previous -> reversing the list.
      * Set the prev = current for next loop.
      * After the loop, set the head = prev which is current.
      * Finally make tail.next = null
      * Return the list.
  + Big O of SinglyLinkedList:
    - Insertion - O(1)
    - Removal - It depends.... O(1) or O(N)
    - Searching - O(N)
    - Access - O(N)
  + **Recap**
    - **Singly Linked Lists are an excellent alternative to arrays when insertion and deletion at the beginning are frequently required**
    - **Arrays contain a built in index whereas Linked Lists do not**
    - **The idea of a list data structure that consists of nodes is the foundation for other data structures like Stacks and Queues**

1. DoublyLinkedList
   * **Almost** identical to Singly Linked Lists, except every node has **another**pointer, to the **previous** node!
   * **Comparison with SinglyLinkedList:**
     + **It takes more memory because one extra property called prev.**
     + More memory === More Flexibility
     + It's almost always a tradeoff!
   * Push
     + Adding a node to the **end** of the Doubly Linked List
     + Pseudocode:
       - Create a new node with the value passed to the function
       - If the head property is null set the head and tail to be the newly created node
       - If not, set the next property on the tail to be that node
       - Set the previous property on the newly created node to be the tail
       - Set the tail to be the newly created node
       - Increment the length
       - Return the Doubly Linked List
   * Pop
     + Removing a node from the **end** of the Doubly Linked List
     + Pseudocode
       - If there is no head, return undefined
       - Store the current tail in a variable to return later
       - If the length is 1, set the head and tail to be null
       - Update the tail to be the previous Node.
       - Set the newTail's next to null
       - Decrement the length
       - Return the value removed
   * Shift
     + Removing a node from the **beginning** of the Doubly Linked List
     + Pseudocode
       - If length is 0, return undefined
       - Store the current head property in a variable (we'll call it old head)
       - If the length is one
       - set the head to be null
       - set the tail to be null
       - Update the head to be the next of the old head
       - Set the head's prev property to null
       - Set the old head's next to null
       - Decrement the length
       - Return old head
   * Unshift
     + Adding a node to the **beginning** of the Doubly Linked List
     + Pseudocode
       - Create a new node with the value passed to the function
       - If the length is 0
         * Set the head to be the new node
         * Set the tail to be the new node
       - Otherwise
         * Set the prev property on the head of the list to be the new node
         * Set the next property on the new node to be the head property
       - Update the head to be the new node
       - Increment the length
       - Return the list
   * Get
     + Get the node from the DLL using index.
     + **Insert of always going from the start, check the index value, and decide from which side need to start the loop. From the beginning or the end. This improves the efficiency of the code.**
     + Pseudocode:
       - If the index is less than 0 or greater or equal to the length, return null
       - If the index is less than or equal to half the length of the list
         * Loop through the list starting from the head and loop towards the middle
         * Return the node once it is found
       - If the index is greater than half the length of the list
         * Loop through the list starting from the tail and loop towards the middle
         * Return the node once it is found
   * Set
     + Replacing the value of a node to the in a Doubly Linked List
     + Pseudocode:
       - Create a variable which is the result of the get method at the index passed to the function
         * If the get method returns a valid node, set the value of that node to be the value passed to the function
         * Return true
       - Otherwise, return false
   * Insert
     + Adding a node in a Doubly Linked List by a certain position
     + Pseudocode:
       - If the index is less than zero or greater than or equal to the length return false
       - If the index is 0, unshift
       - If the index is the same as the length, push
       - Use the get method to access the index -1
       - Set the next and prev properties on the correct nodes to link everything together
       - Increment the length
       - Return true
   * Remove
     + Removing a node in a Doubly Linked List by a certain position
     + Pseudocode
       - If the index is less than zero or greater than or equal to the length return undefined
       - If the index is 0, shift
       - If the index is the same as the length-1, pop
       - Use the get method to retrieve the item to be removed
       - Update the next and prev properties to remove the found node from the list
       - Set next and prev to null on the found node
       - Decrement the length
       - Return the removed node.
   * Reverse
     + Reversing a Doubly Linked List **in place!**
     + **Pseudocode:**
       - Create a variable called current and set it to be the head of the list
       - Create a variable called tail and set it to be the head of the list
       - Loop through the list and set the next property of the current node to be the prev property of the current node
       - If there is no next property, set the tail to be the head and the head to be the current variable
       - Return the list
   * Big O of DLL
     + Insertion - O(1)
     + Removal - O(1)
     + Searching - O(N) – Technically searching is O(N / 2), but that's still O(N)
     + Access - O(N)
   * Recap
     + Doubly Linked Lists are almost identical to Singly Linked Lists except there is an additional pointer to previous nodes
     + Better than Singly Linked Lists for finding nodes and can be done in half the time!
     + **However, they do take up more memory considering the extra pointer**
     + Doubly linked lists are used to implement other data structures and certain types of caches
2. Stack:

* + A **LIFO** data structure!
  + The last element added to the stack will be the first element removed from the stack
  + Where its used?
    - Managing function invocations
    - Undo / Redo
    - Routing (the history object) is treated like a stack!
  + There is more than one implantation for stacks, like an array, linked list, doubly linked list etc.
  + Stacks only need push & pop, so we don’t use Array because it has lot other helper methods.
  + We implement out own stack using Linked List.
  + In Singly Linked List, pop is not the constant time as need to loop thorough the entire list to remove the last item.
  + Stack data structure supposed to be **constant time**, So we use the shift and unshift methods as push & pop here.
  + **In the Doubly Linked List, push and pop are constant times. But here, we implement SLL for easier.**
  + Big O of Stack:
    - Stack is mainly used for insertion and removal, if we want to use accessing method a lot, then probably use the Array.
    - Insertion - O(1)
    - Removal - O(1)
    - Searching - O(N)
    - Access - O(N)
  + Recap:
    - Stacks are a LIFO data structure where the last value in is always the first one out.
    - Stacks are used to handle function invocations (the call stack), for operations like undo/redo, and for routing (remember pages you have visited and go back/forward) and much more!
    - They are not a built in data structure in JavaScript, but are relatively simple to implement
    - Insert and remove are both O(1)

1. Queue:
   * A FIFO data structure!
   * First In First Out
   * Similar to Stack, we can use the Array to implement the Queue. But adding or removing a item in the beginning of the array is O(n), so we use the Singly Linked List push and shift method.
   * Enqueue: Adding to the **beginning** of the Queue!
   * Dequeue: Removing from t*he* **beginning** of the Queue.
   * Big O of Queue:
     + Queue is mainly used for insertion and removal, if we want to use accessing method a lot, then we probably use the other data structure. Insertion & removal are constant time.
     + Insertion - O(1)
     + Removal - O(1)
     + Searching - O(N)
     + Access - O(N)
   * Recap:
     + Queues are a FIFO data structure, all elements are first in first out.
     + Queues are useful for processing tasks and are foundational for more complex data structures
     + Insertion and Removal can be done in O(1)
2. Binary Search Trees
   * A data structure that consists of nodes in a parent / child relationship
   * Tree Terminology:
     + **Root -**The top node in a tree.
     + **Child** -A node directly connected to another node when moving away from the Root.
     + **Parent** - The converse notion of a child.
     + **Siblings** -A group of nodes with the same parent.
     + **Leaf** - A node with no children.
     + **Edge** - The connection between one node and another.
   * Every parent node has at most two children
   * Every node to the left of a parent node is always less than the parent
   * Every node to the right of a parent node is always greater than the parent
   * Real-Time applications
     + HTML DOM
     + Network Routing
     + Abstract Syntax Tree
     + Artificial Intelligence
     + Folders in Operating Systems
     + Computer File Systems
   * Insert:
     + Inserting a node in the BST.
     + Pseudocode:
       - Create a new node
       - Starting at the root
         * Check if there is a root, if not - the root now becomes that new node!
         * If there is a root, check if the value of the new node is greater than or less than the value of the root
         * If it is greater

Check to see if there is a node to the right

If there is, move to that node and repeat these steps

If there is not, add that node as the right property

* + - * + If it is less

Check to see if there is a node to the left

If there is, move to that node and repeat these steps

If there is not, add that node as the left property

* + Find
    - Finding a item in the BST.
    - Pseudocode:
      * Starting at the root
        + Check if there is a root, if not - we're done searching!
        + If there is a root, check if the value of the new node is the value we are looking for. If we found it, we're done!
        + If not, check to see if the value is greater than or less than the value of the root
        + If it is greater

Check to see if there is a node to the right

If there is, move to that node and repeat these steps

If there is not, we're done searching!

* + - * + If it is less

Check to see if there is a node to the left

If there is, move to that node and repeat these steps

If there is not, we're done searching!

* + Big O of BST
    - Insertion - O(log n)
    - Searching - O(log n)
    - **NOT guaranteed!**
    - Whenever double the number of nodes, only increase one step to insert/find by 1, that’s why logn.
    - Refer to the below screenshot.
    - Diagram, shape

      Description automatically generated
    - **Chart, line chart

      Description automatically generated**
    - **But it’s not guaranteed, refer to the below screenshot, it’s a still valid BST, but its O(n)**
    - **Shape

      Description automatically generated with low confidence**

1. Tree Traversing
   * Two ways to traverse a tree
     + **Breadth-first Search**
     + **Depth-first Search**
   * **Breadth-first Tree**
     + Traversing a tree horizontally.
     + Pseudocode:
       - Create a queue (this can be an array) and a variable to store the values of nodes visited
       - Place the root node in the queue
       - Loop as long as there is anything in the queue
         * Dequeue a node from the queue and push the value of the node into the variable that stores the nodes
         * If there is a left property on the node dequeued - add it to the queue
         * If there is a right property on the node dequeued - add it to the queue
       - Return the variable that stores the values
   * **Depth-first Tree**
     + Traversing a tree vertically.
     + Three type
       - DFS – PreOrder
       - DFS - PostOrder
       - DFS – InOrder
     + DFS-PreOrder :
       - **Order is Root — Left — Right**
       - Starting from the root node:
       - First, access the current node value, to say, print it out.
       - Then, we do the preorder traversal on the left subtree.
       - And finally, do the same with the right subtree.
       - Basically, we print the parent node first and then the left node and lastly right node. Rinse and repeat, until we've covered the whole tree.
       - Start from the root, add it in the array, everytime visit the node, add it in the array and then visit all its left side of the node and add it in the array and then visit all itsright side of the node and add it.
       - **PreOrder – Add the node value into the array everytime when visit the node. But other two PostOrder n InOrder are different.**
       - Refer to the below screenshot.
       - Diagram

         Description automatically generated
       - Pseudocode:
       - Create a variable to store the values of nodes visited
       - Store the root of the BST in a variable called current
       - Write a helper function which accepts a node
         * Push the value of the node to the variable that stores the values
         * If the node has a left property, call the helper function with the left property on the node
         * If the node has a right property, call the helper function with the right property on the node
       - Invoke the helper function with the current variable
       - Return the array of values
     + DFS – PostOrder:
       - Order is **Left —> Right —> Root**
       - First, traverse the left subtree recursively.
       - Then traverse the right subtree recursively.
       - Then finally, access the current node value and print it.
       - Its bit different from the PreOrder in terms of the order that we adding the node value in the array.
       - In PreOrder we add the value whenever visit the node. But here, we visit all the node first then add the value into the array.
       - Start from root, go all the way to the left and all the way to the right, then add the value.
       - Same code, but just the adding into array part will come after the visited logic.
       - Pseudocode:
         * Create a variable to store the values of nodes visited
         * Store the root of the BST in a variable called current
         * Write a helper function which accepts a node

If the node has a left property, call the helper function with the left property on the node

If the node has a right property, call the helper function with the right property on the node

Push the value of the node to the variable that stores the values

Invoke the helper function with the current variable

* + - * + Return the array of values
    - DFS – InOrder:
      * Order is Left —> Root —> Right.
      * We'll start from the root node and traverse the entire left subtree recursively.
      * Access the data value of the root node and do whatever you need to do with it. Adding that value into array.
      * Then traverse the entire right subtree recursively.
      * Diagram

        Description automatically generated
      * Pseudocode:
        + Create a variable to store the values of nodes visited
        + Store the root of the BST in a variable called current
        + Write a helper function which accepts a node

If the node has a left property, call the helper function with the left property on the node

Push the value of the node to the variable that stores the values

If the node has a right property, call the helper function with the right property on the node

* + - * + Invoke the helper function with the current variable
        + Return the array of values
  + BFS vs DFS – When to use?
    - **Time Complexity – Both types is O(n)**
    - **Space Complexity** 
      * **If the tree is wider,** 
        + **BFS, needs to store a lot of nodes in the queue on every level. Let’s say, if tree have 100s of node in one level. Then our queue have to store 100s of nodes at each level.**
        + **But in the DFS case, we are not using the queue, just using memory for call stack.**
        + **So, if the tree wider, DFS space complexity is lesser than the BFS.**
      * **If the tree is one-sided long/deep tree,**
        + **BFS, no need to store lot of nodes in the queue, since the nodes in each level is very less.**
        + **DFS, have to store many call stack in the memory, so space complexity is more than the BFS.**
    - **BFS – InOrder/PreOrder/PostOrder Usecase:**
      * **In the Inorder, the nodes are arrange in the sorted order in the Binary Search tree, if we want to store/fetch node in the sorted order then can use this Inorder.**
      * **PreOrder, Can be used to "export" a tree structure so that it is easily reconstructed or copied.**

|  |  |  |  |
| --- | --- | --- | --- |
| S. No. | Parameters | BFS | DFS |
| 1. | Stands for | BFS stands for Breadth First Search. | DFS stands for Depth First Search. |
| 2. | Data Structure | BFS(Breadth First Search) uses Queue data structure for finding the shortest path. | DFS(Depth First Search) uses Stack data structure. |
| 3. | Definition | BFS is a traversal approach in which we first walk through all nodes on the same level before moving on to the next level. | DFS is also a traversal approach in which the traverse begins at the root node and proceeds through the nodes as far as possible until we reach the node with no unvisited nearby nodes. |
| 4. | Technique | BFS can be used to find a single source shortest path in an unweighted graph because, in BFS, we reach a vertex with a minimum number of edges from a source vertex. | In DFS, we might traverse through more edges to reach a destination vertex from a source. |
| 5. | Conceptual Difference | BFS builds the tree level by level. | DFS builds the tree sub-tree by sub-tree. |
| 6. | Approach used | It works on the concept of FIFO (First In First Out). | It works on the concept of LIFO (Last In First Out). |
| 7. | Suitable for | BFS is more suitable for searching vertices closer to the given source. | DFS is more suitable when there are solutions away from source. |
| 8. | Suitable for Decision Treestheirwinning | BFS considers all neighbors first and therefore not suitable for decision-making trees used in games or puzzles. | DFS is more suitable for game or puzzle problems. We make a decision, and the then explore all paths through this decision. And if this decision leads to win situation, we stop. |
| 9. | Time Complexity | The Time complexity of BFS is O(V + E) when Adjacency List is used and O(V^2) when Adjacency Matrix is used, where V stands for vertices and E stands for edges. | The Time complexity of DFS is also O(V + E) when Adjacency List is used and O(V^2) when Adjacency Matrix is used, where V stands for vertices and E stands for edges. |
| 10. | Visiting of Siblings/ Children | Here, siblings are visited before the children. | Here, children are visited before the siblings. |
| 11. | Removal of Traversed Nodes | Nodes that are traversed several times are deleted from the queue. | The visited nodes are added to the stack and then removed when there are no more nodes to visit. |
| 12. | Backtracking | In BFS there is no concept of backtracking. | DFS algorithm is a recursive algorithm that uses the idea of backtracking |
| 13. | Applications | BFS is used in various applications such as bipartite graphs, shortest paths, etc. | DFS is used in various applications such as acyclic graphs and topological order etc. |
| 14. | Memory | BFS requires more memory. | DFS requires less memory. |
| 15. | Optimality | BFS is optimal for finding the shortest path. | DFS is not optimal for finding the shortest path. |
| 16. | Space complexity | In BFS, the space complexity is more critical as compared to time complexity. | DFS has lesser space complexity because at a time it needs to store only a single path from the root to the leaf node. |
| 17. | Speed | BFS is slow as compared to DFS. | DFS is fast as compared to BFS. |
| 18. | When to use? | When the target is close to the source, BFS performs better. | When the target is far from the source, DFS is preferable. |