**Algorithms**

**Time & Space Complexity**

Time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of input.

Space complexity of an algorithm quantifies the amount of space/memory taken by an algorithm to run as a function of the length of input.

E.G – 1.1 – sort lowest to highest

[10, 7, 40, 99, 22]

// loop through the data – create a nested loop to compare and put array in order

This is referred to as n^2 (n squared) or quadratic. As the number of elements grow, the amount of work increases at that rate. So, for 5 elements – 5^2 = 25; 100 elements – 100^2 = 10,000.

E.G – 1.2 – find min and max

[7, 12, 3, 40]

// only one loop needed as we store the current loop in appropriate variable as it iterates through and checks if the current lowest is lower than the iterated element – if not then the value is swapped.

This is referred to as 2n or linear. Imagine a straight line on a line graph. There are less comparisons when compared to quadratic, therefore is quicker. 4 elements – 4 x 2 = 8; 100 elements – 100 x 2 = 200.

E.G – 1.3 min max in sorted array

[12, 20, 30, 99];

// we know arr[0] is the lowest, and arr[arr.length – 1] is the max.

This is referred to as constant. As data increases there is always the same number of operations needed.

**Big O Notation**

Big O notation is the language used to describe the complexity of an algorithm

|  |  |  |
| --- | --- | --- |
| **Big O name** | **Number of operations** | **Algorithm** |
| O (n^2), quadratic | n^2 | Compare all elements with one another (i.e. nested loop) |
| O(n), linear | 2n | Compare all elements once |
| O(1) constant | 2 | Number of operations do not change as data set grows/shrinks |

*Super Fast Super Slooooww*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Name** | Constant | Logarithmic | Linear | Quadratic | Exponential |
| **Notation** | O(1) | O(log n) | O(n) | O(n^2) | O(k^n) |

A good rule of thumb is if the data set is small, focus on readability and don’t worry too much about performance. If the data set is large then it may be worth looking to improve performance with these algorithms.

Complexity of common operations

|  |  |
| --- | --- |
| **Complexity** | **Operation eg** |
| O(1) | Value look up in an array |
| O(log n) | Loop that cuts problem by fraction every time |
| O(n) | Loop |
| O(n^2) | Nested loop |
| O(n^3) | Triple nested loop |

Native JS

|  |  |
| --- | --- |
| **Method / exp** | **Algorithm complexity** |
| For loop | Linear O(n) // 2n |
| Pop() | Constant O(1) |
| Arr[0] | Constant |
| Obj.prop | Constant |
| Arithmetic e.g. 1+2 | Constant |
| Shift/unshift | Linear – as all elements in array have to shift across |

**Optimisation with caching**

Cache Algorithm

E.G – 2.1 – check if array is unique

Const isUnique = arr => {

Let result = true; // is unique

For (let i = 0; I < arr.length; i++) {

For (let j = 0; j < arr.length; j++) {

If (i !== j && arr[i] === arr[j]) {

Result = false;

}

}

}

Return result;

}

// Quadratic as double nested loop – O(n^2)

E.G – 2.2 – improved 2.1

Const isUnique = arr => {

Const obj = {};

Let result = true

For (let i = 0; i < arr.length; i++) {

If (obj[arr[i]]) { // check if prop currently exists on obj

Result = false;

} else {

Obj[arr[i]] = true; // cache result in obj

}

}

Return result;

}

// Linear O(n)

E.G – 2.3 – sort array desc -> asc

Const uniqueSort = arr => {

Const cache = {};

Const result = [];

For (let i = 0; i < arr.length; i++) {

If (!cache[arr[i]]) {

Result.push(arr[i]);

Cache[arr[i]] = true;

}

}

Return result.sort((a-b) => a – b);

}

uniqueSort([5, 6, 6, 2, 1, 1, 7]); // [1, 2, 5, 6, 7]

// here we are sacrificing space for time. We are creating a linear space complexity as we are caching rows in a temporary object (cache), in order to get more speed. As opposed to saving space and doing a quadratic O(n^2) time complexity.

// We are creating a new array to push values to which is sacrificing space also.

**Memoization**

Memoization is an optimization technique use primarily to speed up computer programs by storing the results of expensive function calls and return the cached result when the same inputs occur again.

E.G – 3.1 with closure

Const memoizedClosureTimes10 = () => {

Let cache = {}; // not in global scope

Return (n) => {

If (n in cache) {

Return cache[n];

}

Let result = n \* 10;

Cache[n] = result;

Return result;

}

}

Const memoClosureTimes10 = memoizedClosureTimes10(); // returns closure function

Console.log(memoClosureTimes10(9)); // calculated

Console.log(memoClosureTimes10(9)); // cached

**Recursion**

Recursion is when a function calls itself. In most cases, a loop will be quicker as no function call expense. There is always a loop alternative for recursion, however, in certain data structures, recursion can be easier to implement (more readable) such as nested data structures, e.g. trees and graphs.

E.G – 4.1 accumulator

Function joinElements(array, joinStr) {

Function recurse(index, acc) {

If (index === array.length – 1) {

Return acc;

}

Return recurse(index + 1, acc + joinStr);

}

Return recurse(0, ‘’);

}

joinElements([‘s’, ‘cr’, ‘t cod’, ‘:) :)’], ‘e’); // secret code :) :)

**Divide & Conquer**

Divide & conquer is an algorithm design paradigm based on multi-branched recursion. It works by recursively breaking down a problem in to two or more sub problems until these become simple enough to be solved directly.

Binary search

Binary search is a search algorithm that finds the position of a target value within a *sorted* array. An alternative approach to a binary search is to do a linear search which has a time complexity of O(n), however the binary search approach has a time complexity of O(log n), i.e. logarithmic and is therefore quicker in most circumstances.

Const arr = [10, 40, 70, 90, 100, 200, 500];

// aim: find 200

// cut the array in half

[10, 40, 70, **90**, 100, 200, 500];

// as 200 is greater than 90 we know we can ignore the first half of the array

[90, 100, 200, 500];

// cut array in half again

[90, **100**, 200, 500];

// 200 still greater than 100 so we can ignore anything left of the pointer again

[100, **200**, 500];

// we have found the position!

E.G – 5.1 binary search

Function binarySearch(list, item) {

Var min = 0;

Var max = list.length – 1;

Var middle;

While (min <= max) {

Middle = Math.floor((min + max) / 2);

If (list[middle] === item) {

Return middle;

} else {

If (list[middle] < item) {

Min = middle + 1;

} else {

Max = middle – 1;

}

}

}

Return -1; // if not found

}

binarySearch([10, 40, 60, 90, 113], 90); // 3

**Sorting**

1. **Naïve sorts**: Keep looping and comparing values until the list is sorted. E.g. bubble sort, insertion sort, selection sort. These are quadratic time complexities (i.e. nested loops)
2. **Divide & Conquer sorts**: Recursively divide list and sort smaller parts of list until entire list is sorted. E.g. merge sort, quick sort. These are logarithmic time complexities O(n log n). These are quicker than naïve sorts.

Merge sort

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 38, 27, 43, 3, 9, 82, 10 | | | | | | |
| 38, 27, 43, 3 | | | | 9, 82, 10 | | |
| 38, 27 | | 43, 3 | | 9, 82 | | 10 |
| 38 | 27 | 43 | 3 | 9 | 82 | 10 |
| 27, 38 | | 3, 43 | | 9, 82 | | 10 |
| 3, 27, 38, 43 | | | | 9, 10, 82 | | |
| 3, 9, 10, 27, 38, 43, 82 | | | | | | |

// rows 1-4 are the divide phase of divide and conquer

// rows 5+ are the conquer phase

During the pointer phase, imagine 2 pointers, 1 pointing to the beginning of each half of the array. The pointers compare their values against each other and slices off the lowest number (if sorting desc -> asc) before moving along to the next iteration. E.g. row 6:

[**3**, 27, 38, 43], [**9**, 10, 82]

// compare 3 and 9 – put 3 on to new array as lower and move pointer along, but keep pointer on 2nd array where it is

[**27**, 38, 43], [**9**, 10, 82] // result so far [3]

// compare 27 and 9

[**27**, 38, 43], [**10**, 82] // result so far [3, 9]

// compare 27 and 10 etc…

[**27**, 38, 43], [82] // result so far [3, 9, 10]

[**38**, 43], [82] // result so far [3, 9, 10, 27]

[**43**], [**82**] // result so far [3, 9, 10, 27, 38]

[**82**] // result so far [3, 9, 10, 27, 38, 43]

// result [3, 9, 10, 27, 38, 43, 82]

E.G – 6.1 merge sort

Function mergeSort(arr) {

If (arr.length === 1) {

Return arr;

}

Const middle = (Math.floor(arr.length / 2));

Const left = arr.slice(0, middle);

Const right = arr.slice(middle);

Return merge(mergeSort(left), mergeSort(right));

}

Function merge(left, right) {

Let result = [];

Let indexLeft = 0;

Let indexRight = 0;

While (indexLeft < left.length && indexRight < right.length) {

If (left[indexLeft] < right[indexRight]) {

Result.push(left[indexLeft]);

indexLeft++;

} else {

Result.push(right[indexRight]);

indexRight++;

}

}

Return result.concat(left.slice(indexLeft)).concat(right.slice[indexRight]);

}

**TEST**

**Part 1**

1. Define time and space complexity
2. What are the main 4 big O notation names and describe them from slowest to fastest in terms of time complexity
3. Using a caching algorithm (i.e. keeping algorithm linear) – return the result of an array which only includes unique elements from the array passed in. Don’t worry about sorting the array
4. Using memoization and caching techniques – create a function/closure which like example 3.1
5. When should you use recursion?
6. Create a recursive function/closure which multiplies each element in an array by the element prior.

**Part 2**

1. What is the divide and conquer approach?
2. What is a binary search?
3. Create a binary search function (on a sorted array), which returns the index of a value in an array (i.e. binarySearch(arr, value))
4. What are the time complexities of:
   1. Naïve sorts
   2. Divide and conquer sorts
5. How does a merge sort work?
6. Implement a merge sort