# Algorithms

## Search Algorithms

**Linear Search**

- Involves searching one location after another until searched for value is found.

**Binary Search**

- Binary search works by dividing the list in two each time until we find the item being searched for.

- List must be in order beforehand.

## Sorting Algorithm

**Bubble Sort**

- Bubble sort is one of the easiest sorting algorithms to implement. It essentially starts by comparing the first pair of values and swaps them if necessary. It then compares the second and third values and does the same. It does this for all pairs until the last pair of values in the list is reached. It will then cycle back to the start of the list and repeat the process.

- Bubble sort is finished when it does a full pass through a list without making a change.

**Insertion Sort**

- Insertion sort works by dividing a list into two parts; sorted and unsorted. Elements are inserted one by one into their correct position in the sorted section.

**Merge Sort**

- To understand merge sort you need to understand how we **merge** lists. If we have two lists in order we can merge them into one list by taking the first item of each list depending on which has the lowest and adding it to the start of the new list. Until both lists are empty and we have a merged sorted list.

- To **merge sort** you split the list of n items into n lists of 1 item. While there are more than 1 list recursively pair up the lists and merge each pair into a single list twice the size.

**Quicksort**

- Another common divide and conquer sorting algorithm.

- Take the first item and make it a list one item big and call it the pivot.

- Split the remainder of the list into two sub lists. Those less than or equal to the pivot and those greater than.

- Recursively do this until all sub-lists are pivots.

- The pivots can now be combined to form a sorted list.

## **Complexity**

- We can evaluate algorithms in terms of how long they take to execute and how much memory they use. Often speed can be increased at the expense of using more memory.

- Whilst knowing how long an algorithm to execute can be of use computing power is rapidly changing (doubling in power roughly ever 18 months).

- A more useful way to compare algorithms is the complexity. Complexity doesn’t show how fast an algorithm performs but rather how well it scales given larger data sets to act on.

- We can use Big-O notation to note an algorithm’s complexity. Its called Big-O because it is written O(x) where x is the worst case complexity of the algorithm.

- **Constant Complexity O(1) -** Algorithms have constant complexity and take the same time to run regardless of the data set.

- **Linear Complexity O(n)-** Increase at the same rate as the input size increases. Increase is proportional to input size.

- **Polynomial Complexity** **O(n^k) k>=0**- Time taken can be expressed as n^k where k is a constant value and n is the input size.

- **Exponential Complexity O(k^n) where k>1** - Does not scale well at all. Exponential increase means as n gets larger time taken increases at a rate of k^n where n is constant.

- **Logarithmic Complexity O(logn)** - Scale extremely well. As n gets larger time taken increases at a rate of log(n)

## Shortest Path Algorithms

**Dijkstra’s Algorithm**

- You know this one. Just practice it.

**A\* Algorithm**

- Alternative that uses heuristics to be faster. A heuristic is a rule of thumb or estimate of the remaining distance but one that can never make an overestimate.

- Instead of adding the distance from start as the value of a node instead you add this to the heuristic estimate of the remaining distance.