# Data structures and algorithms

## Arrays

* Always the same length in memory.
* A continuous block stored in memory.
* All its elements are the same size.
* To get an element of an array knowing the index it is always the same time complexity since the formula is just the starting memory block plus the index of the element times the element size, meaning the time complexity would be O(1).
* If you don’t know the index of the element and you want to find a specific element you would have to iterate through the array to find it, so the time complexity would be O(n).
* Adding an element to a full array would be O(n) because you would need to create a new array, copy the original elements and adding the new one.
* Adding an element to the end of an array (with space) is O(1) since we have the index.
* Inserting, updating, or deleting an element at a specific index is also O(1).
* Deleting an element by shifting elements or if you don’t have the index, you would have O(n).

**Bubble Sort**

* We start with the unsortedPartitionIndex in the last index of the array
* And we have another index (i) in the start of the array. If the element in i is greater than the one on the right we swap the element, otherwise we leave them, and increment i+1.
* At the end of the iteration i = unsortedPartitionIndex.
* Then we pace the unsortedPartitionIndex in unsortedPartitionIndex – 1 since the last element is already ordered and i = 0. And we start all over again. We do this until unsortedPartitionIndex = 0.
* It’s an in-place algorithm, although we create extra variables, it does not depend on the number of elements we are sorting so it does not use any extra memory.
* It has an O( time complexity, so in terms of time it is not very effective.
* Stable sort algorithm.

**Stable vs Unstable sort:**

* If a sort is unstable, means that the duplicate order of repeated numbers will not be preserved.
* A stable sort is preferred, because when we’re sorting objects, the unstable sort might affect the outcome.

**Selection Sort**

* The lastUnsortedIndex will be equal to the array length – 1
* I will be equal to one and will be used to traverse the array from left to right.
* The largest element will be 0 (index of the largest element we know) and it always starts in the beginning of the array.
* If array[i] > array[largest] then we change largest to I and we increment i. If array[i] <= array[largest] we just increment i.
* When i = lastUnsortedIndex we have completed the first iteration of the array so we swap the largest element with the last Element from the unsorted partition.
* Then we reinitialize the values i = 1, largest = 0 and lastUnsortedIndex = lastUnsortedIndex- 1. We repeat this until lastUnsortedIndex = 0.
* In place algorithm.
* Time complexity of O(.
* It requires less swapping than bubble sort so it would usually perform better than bubble sort.
* Unstable algorithm.

**Insertion Sort**

* firstUnsortedIndex = 1
* i is used to traverse the sorted partition from right to left and is initialized in 0.
* New element is the value we want to insert into the sorted partition (array[firstUnsortedIndex]).
* If array[firstUnsortedIndex] > array[i] then we the firstUnsortedIndex will be added one and so will i.
* If the new element is < array[i] we move array[i] to the right (array[i=1]) to make room for the new number if the new element is greater than array[i-1] then we assign array[i] = new element else we keep moving the elements to the right until we reach the beginning of the array and we insert the new element in array[0].
* In place algorithm
* Time complexity of O(.
* Stable algorithm.

**Shell Sort**

* Variation of insertion sort
* Insertion sort chooses which element to insert using a gap of 1.
* It starts using a larger gap value that is reduced as it runs
* The goal is to reduce the amount of shifting required
* The last gap is 1
* A gap value of 1 is equivalent to insertion sort but at that point the array is more sorted than at the beginning so there is less shifting
* The gap value influences the time complexity.
* Knuth Sequence:
  + Gap is calculated using
  + K is based on the length of the array
  + We want the gap to be as close as possible to the length of the array without it being greater than the length
* In place algorithm
* Worst case time complexity is quadratic, but it depends on the gap value
* Does not require as much shifting
* Unstable algorithm

**Recursion**

* Factorial examples:
  + If a num is equal to 0 the factorial is 1
  + Otherwise, we set multiplier to 1
  + Set factorial to 1
  + While multiplier is not equal to num, do steps 5 and 6
  + Multiply factorial by multiplier and assign the result to factorial
  + Add 1 to multiplier
  + Stop. The result is factorial.
* A recursive method relies on different calls to the same function until the base condition is met and then all the calls are answered to get the final answer
* So it may imply a lot of processes
* Usually iterative implementation runs faster and uses less memory, however sometimes it is not understandable code.
* Without a base condition you get a stack overflow exception
* You may get a stack overflow exception with a base condition if the algorithm. Does not get to the base condition quick enough

**Merge sort**

* Divide and conquer algorithm
* Recursive algorithm
* Two phases: Splitting and Merging
* Splitting phase leads to faster sorting during the merging phase
* Splitting is logical
* Start with an unsorted array
* Divide the array into two arrays, which are unsorted. The first array is the left array, and the second array is the right array
* Split the left and the right arrays into two arrays each
* Keep splitting until all the arrays have only one element each – these arrays are sorted.
* Merge every left/right pair of sibling arrays into a sorted array
* After the first merge, we’ll have a bunch of 2-element sorted arrays
* Then merge those sorted arrays (left/right siblings) to end up with a bunch of 4-element sorted arrays
* Repeat until toy have a single sorted array
* Not in place, uses temporary arrays.
* When merging we merge left and right arrays
* We create a temporary array large enough to hold all the elements in the arrays we’re merging
* We set I to the first index of the left array, and j to the first index of the right array
* We compare left[i] to right[j], if left is smaller, we copy it to the temp array and increment I by 1, if right is smaller, we copy it to the temp array and increment j by 1.
* We repeat that until all elements in the two arrays have been processed
* At this point the temporary array contains the merged values in sorted order
* We then copy this temporary array back to the original input array, at the correct positions
* The time complexity is O(nlogn) – base 2. We’re repeatedly dividing the array in half during the splitting phase
* Stable algorithm