**Data Structures & Algorithms**

**Algorithms**

* Algorithms – steps needed to complete a task
* A problem can be solved using various algorithms.
* An algorithm can have many implementations

**Big O Notation / Time Complexity:**

Time Complexity – The amount of time taken by an algorithm to run.

Memory Complexity – The amount of memory taken by an algorithm to run. (not as important now because memory is cheap).

Table

Description automatically generated

Listed from best to worst.

Chart, line chart

Description automatically generated

Constant O(1) : As the number of items increases the number of steps remain the same (constant). This is the best-case scenario.

As you go down the table the number of steps drastically increase as the number of items increase.

**Arrays**

* Size of an array is static
* Contiguous block in memory
  + Memory is allocated on declaration of the size of an array.
  + Every element occupies the same amount of space in memory.
  + In the case of non-primitive types (objects) the array contains references to those types of same size.

Benefits:

* Useful when you know the indices of the element you want to retrieve
  + O(1) constant time complexity
* Memory Efficient

Cons:

* When indices of element is unknown it becomes:
  + O(n) linear time complexity

Table

Description automatically generated

**Unstable vs Stable Algorithms:**

Table, calendar

Description automatically generated

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

Graphical user interface, application, table

Description automatically generated

Stable Algorithms are usually preferred in some cases such as sorting objects.

**Sorting Algorithms**

**Bubble Sort Theory**

* Performance degrades quickly as the items needed to sort grows.
* In-place algorithm (sorted items occupy the same storage as the original items)
* O(n2) – quadratic time complexity
* Stable algorithm
* Grows right to left
* Traversing through the unsorted partition
* Uses two variables
  + Unsorted partition index
  + Traverse index

Graphical user interface, application

Description automatically generated

Diagram, table

Description automatically generated with medium confidence

**Bubble Sort Implementation**

Text

Description automatically generated

* In general, the time complexity can be determined by looking at the amount of “for” loops the algorithm has as they correspond to “n”.
  + 2 for loops (quadratic algorithms)

**Selection Sort Theory**

* In-place algorithm
* O(n2) – quadratic time complexity
* Less swapping than bubble sort
* Unstable algorithm
* Grows right to left
* Traversing through the unsorted partition
* Uses 3 variables
  + Unsorted partition index
  + Traverse index
  + Largest value index

Graphical user interface, application

Description automatically generated

**Selection Implementation**

Text

Description automatically generated

**Insertion Sort (Theory)**

* In place algorithm
* O(n2) – quadratic time complexity
* Stable
* Grows from left to right
* Work with the sorted partition
* Uses 3 variables
  + Unsorted partition index
  + Traverse index
  + New Element – value we want to insert

Graphical user interface, application

Description automatically generated

**Insertion Sort (Implementation)**

Text

Description automatically generated

**Shell Sort (Theory)**

* In place algorithm
* Time complexity is dependent on gap
  + Worst case: O(n2)
* Unstable
* Variation of Insertion Sort
* Shell Sort starts out using a larger gap value
  + Insertion sort chooses which element to insert using a gap of 1
  + Goal is to reduce amount of shifting required
* As the algorithm runs the gap is reduced
  + Last gap value is always 1, which is ultimately an insertion short

Table

Description automatically generated

* Gap is calculated using (3k-1) / 2
* k is set based on the length of the array
* We want the gap to be as close as possible to the length of the array, without being greater than the length.

Graphical user interface, application

Description automatically generated with medium confidence

**Shell Sort (Implementation)**

Text

Description automatically generated

**Merge Sort (Theory)**

* Not in place
* O (n log n)
* Stable
* Divide and conquer algorithm (recursion)
* Two phases:
  + Splitting
  + Merging
* Splitting leads to faster sorting during the merging phase
* Splitting is logical, not creating new arrays

**Splitting Phase**

* Start with an unsorted array
* Divide the array into 2 arrays, which are also unsorted
  + First array – left array
  + Second array – right array
* Keeping splitting the subarrays until they become arrays of one element
* The one element arrays are considered sorted

Diagram

Description automatically generated

**Merging Phase**

* 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 arrays (left/right) siblings to end up with a bunch of 4-element sorted arrays
* Repeat until you have a single sorted array
* Not in-place. (uses temporary arrays).

Diagram

Description automatically generated