**Computational Thinking with Algorithms Project**

**Sorting Algorithms**

**Bubble Sort**

The bubble sort is a simple sorting algorithm that repeatedly steps through a list, compares adjacent elements (from left to right) and swaps their positions if they are in the wrong order. [1] It does this through the entire list until it cannot find any two elements that are required to be swapped and the list is completely sorted. It is a comparison based sort due to the way it compares the element adjacent to it and determines if they need to be swapped or not. It also uses in place sorting therefore uses a constant amount of additional working space in addition to the memory required for the input. [2]

It was first used by Iverson in 1962 but it has been analysed from as early as 1956. It received its name because the biggest elements in the list always ‘bubble’ to the end or top of the list [2].

As mentioned above, a comparison based algorithm complexity cannot do no better than n log n performance in the average or worst cases. Bubble sort is a comparison-based algorithm so that applies analysis will apply when reviewing benchmarking results.

The bubble sort is very impractical for large datasets and has an average and worst complexity of Ο(n2). It can achieve n performance in the Best Case but that is unlikely to ever be achieved using random pseudocode as per this project. It is much more likely that worst or average case will be achieve where it will compare every element in the list with every other one. This will result in Ο(n2) or exponential to the size of the input.

**How Bubble Sort Works**

Irrespective of the nature of input, the number of passes to be made is n – 1. [4] By the end of every pass, at least one element is placed in its right position.

As per figure below, input [a] has 9 elements in unsorted positions.

**Step 1**: Pass 1, compare 54 and 26. 54 is greater than 26 so the elements are swapped.

**Step 2:** Pass 1, compare 54 and 93. 54 is smaller than 93 so the elements are not swapped

**Step 3:** Pass 1, compare 93 and 17. 93 is greater than 17 so the elements are swapped.

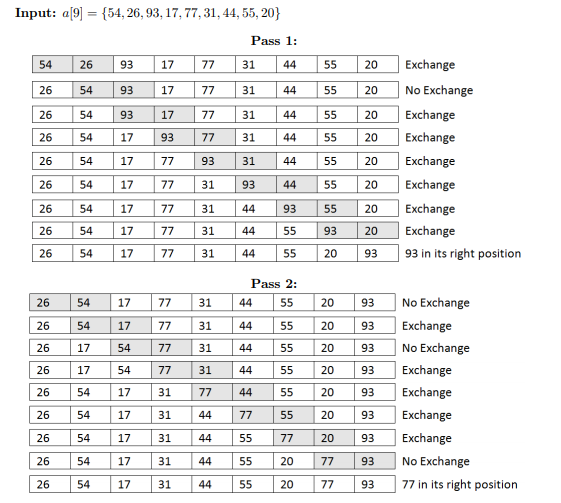
 Pass 1 continues in this process until the largest element is at the end, in this case its 93. Then, Pass 2 completes and so on until the list/array is sorted. There will be 8 passes in the example because there are 9 elements.

Diagram Source [4]

**Selection Sort**

Selection sort is an in place comparison based sorting algorithm which is simple to implement. It divides the list into two separate parts, the sorted list on the left and the unsorted list on the right. It is based on the process of finding the minimum and maximum element of an unsorted array and placing it in its correct position [5].

The algorithm starts at element 0 and works its way through until n-1. It selects the smallest element and puts that element in position 0 (sorted sub array). It then goes to element 1, sorts through n-1 to find the smallest element on the right side (in the unsorted sub array) and swaps it to position 1. There are now 2 elements in the sorted array on the left. The algorithm continues in this way until there is nothing left to search and sort. [2]

Selection sort, while performs better than the Bubble Sort, is still an impractical sorting algorithm for real world data. It can be beneficial on small datasets [2]. Time complexity of selection sort in best, worst and average case is n2. It does not matter the state in which the list is in if selection sort is used. If it is already sorted or in reverse order, O(n2) is returned. This quadratic because of the number of comparisons required to be completed. At every step, you have to find the minimum element and put it in the right place. The minimum element is not known until the end of the array is reached. [6]

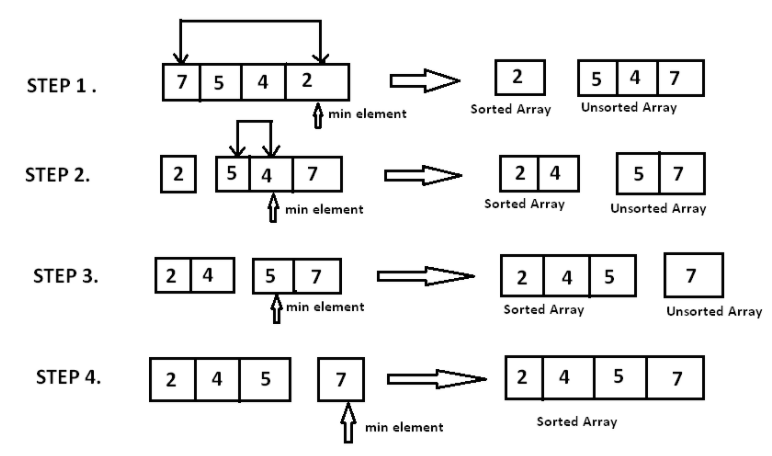
**How Selection Sort Works**

**Step 1:** The minimum value is identified, in this case 7. 2 is then swapped into position 0. There are now 2 sub arrays. A sorted sub array and an unsorted subarray.

**Step 2:** The minimum value in the unsorted sub array is identified as 4. It is swapped into position 1 in the sorted sub array

**Step 3:** 5 is identified as the minimum value in the unsorted subarray and moved into position 2 in the sorted array.

**Step 4:** 7 is the last number in the unsorted array and therefore the minimum value. It is swapped into position 3 of the sorted array. Sort complete.

Diagram Source: [5]

**Insertion Sort**

Insertion sort is a simple comparison based algorithm that builds the sorted list one item at a time. It works similar to the way a person would sort playing cards in your hand. That is because if you have 5 cards in your hand that are in order and you pick up another card, you go through the deck 1 by 1 comparing the new card to the cards you already have in order and you insert it into the correct position [7][8]. The Insertion sort algorithm follows a similar process.

Insertion sort can be easy to implement, it is stable, follows in place sorting and can be effective and efficient on small lists or lists already nearly sorted. However, similar to bubble sort and selection sort, it is not inefficient on large random datasets [2].

It is an iterative sorting algorithm and works by splitting a list of input size n into sorted and unsorted sub lists (similar to selection sort). The algorithm starts at the left of the unsorted array and sets a key at index 1. The key value then compares itself to the element to its left. If the key value is greater than the element to its left, nothing swaps but if the element to its left is greater, then the element moves right by one position. Set the key at index 2. This keeps iterating for index n-1 until the list is sorted.

Insertion sort can achieve O(n), linear, in the best case. That is because, if it receives an already sorted list, it will still execute the outer for loop, thereby requiring n steps to sort an already sorted array/list of n elements, which makes its O(n) complexity. [9]. Average and worst case for insertion sort is O(n2). The algorithm takes an element to compare with every other element: n x n comparisons, quadratic.

**How Insertion Sort Works**

Array = [9, 6, 4, 1]

**Step 1:** Nothing to compare to the left of index 0, therefore add key value to index 1

|  |  |  |  |
| --- | --- | --- | --- |
| 9 | 6 (key) | 4 | 1 |

**Step 2:** Compare Key (6) with element to its left (9). 9 > 6, therefore they swap position. 5 is now the key as it gets assigned to the next element to the right.

|  |  |  |  |
| --- | --- | --- | --- |
| 9 | 6 (key) | 4 | 1 |

|  |  |  |  |
| --- | --- | --- | --- |
| 6 | 9 | 4 (key) | 1 |

**Step 3:** 4 is now the key. Compare 4 to 9. 9 > 4, swap position. Compare 4 to 6. 6 > 4. 4 is now in index 0 and 1 is the key value

|  |  |  |  |
| --- | --- | --- | --- |
| 4 | 6 | 9 | 1(key) |

|  |  |  |  |
| --- | --- | --- | --- |
| 6 | 9 | 4 (key) | 1 |

**Step 4:** 1 is now the key value. Compare 1 to 9, 9 > 1. Compare 1 to 6, 6 > 1. Compare 1 to 4, 4 > 1. All elements are now sorted in the correct order and the algorithm is complete.

|  |  |  |  |
| --- | --- | --- | --- |
| 4 | 6 | 9 | 1(key) |

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 4 | 6 | 9 |

**Quick Sort**

Quicksort is a recursive divide and conquer algorithm. The approach is based on choosing one element as the pivot value and then partitioning the array (into two sub arrays) around that pivot element. The left side of the pivot contains elements that are less than the pivot value and elements to the right are greater than the pivot value. [10]. The sub arrays are sorted recursively and is completed in place.

It was developed by Tony Hoare in 1959 and when implemented well can be very efficient and quick to sort as its name suggests. [11]. It is a comparison based sorting algorithm such as bubble, selection and insertion sort but it is much more efficient than any of those mentioned algorithms.

The partitioning process can determine the running time of Quick sort. In the best and average case, Quick sort will run in 𝑛 log 𝑛 but if a bad pivot is continually chosen, running time performance will be O(n2) in the worst case. Worst case can occur if too many of the elements are in the sub array < the pivot or in the sub array > pivot. [2]. There are many ways to choose the pivot: the 1st or last element in the array, the median element of the array or choose the pivot randomly. Worst case (quadratic) time would most only likely occur when the 1st or last elements are selected. By selecting the median element, there is a better chance of getting balanced sub arrays and therefore an average running time of O(n log n).

**How Quick Sort Works**

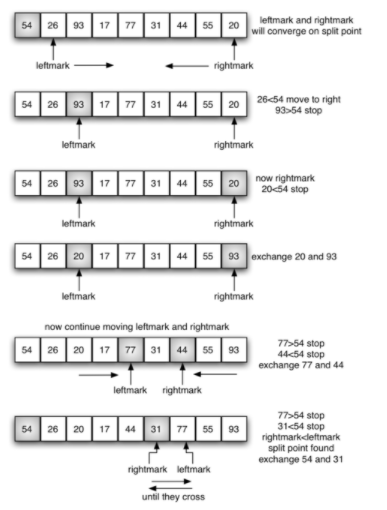
The pivot value taken in this example is the first element in the array 54.

Diagram Source [12]

**Step 1:** Pivot value identified as 54. The two position markers: left mark and right mark are identified. They are the most left and most right elements remaining in the list.

**Step 2:** Left mark is incremented right until a value greater than the pivot value is located. Then right mark is decremented until a value less than pivot value. This is evident between values, 93 (93 >54) and 20 (20 < 54). They are out of order so they can be swapped with each other. Then the left mark increments again.

**Step 3:** Neither 20 nor 17 are greater than the pivot value 54 therefore left mark increments until 77 which is greater than 54. The right mark starts to decrement because 55 is not less than 54. Right mark is not 44 which is less than 54. 77 and 44 can be swapped.

**Step 4:** Now right mark is less than left mark. The split point has been identified and the split point (rightmark) can be swapped with the pivot value. Now all the values less than 54 are to the left and all the values greater than 54 are to its right.

**Step 5:** The algorithm then recursively follows the above to sort the sub arrays until completely sorted.

**Complexity of sorting algorithms**

As discussed above and can be seen from table 1 below, all the simple comparison based sorting algorithms have an average and worst case time complexity of n2 (quadratic) and have a space complexity of 1. Bubble sort and insertion sort can achieve time complexity of n (linear) in the best case but the probability of that occurring in a real world occurrence are unrealistic. Using real word data or random data as per this project, it is expected to get an average /worst case and a time complexity of n2 will be seen. The 3 simple sorting algorithms all have a space complexity of 1 meaning there is no need extra space required to sort the list. In each case, the algorithm is only comparing 2 elements at a time and swapping them if required.

**Table 1. Overview of complexity**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Algorithm** | **Best Case** | **Worst Case** | **Average Case** | **Space Complexity** | **Stable** |
| Bubble Sort | n | n2 | n2 | 1 | Yes |
| Selection Sort | n2 | n2 | n2 | 1 | No |
| Insertion Sort | n | n2 | n2 | 1 | Yes |
| Quicksort | n log n | n2 | n log n | n (worst case) | Yes |
| Counting Sort | n + k | n + k | n + k | n + k | No (Unstable in standard form) |

**Size 100.000 250.000 500.000 750.000 1000.000 1250.000 2500.000 3570.000 5000.000 6250.000 7500.000 8500.000 10000.000**

**Bubble Sort 4.697 14.441 45.705 115.104 193.540 284.007 679.365 1505.593 2863.844 4494.123 6328.905 9014.407 12559.199**

**Selection Sort 10.831 12.789 26.087 48.598 109.395 143.616 328.415 642.019 1176.452 1865.908 2720.010 3894.067 5168.795**

**Insertion Sort 5.632 7.314 23.997 54.575 100.705 151.362 364.565 804.256 1555.408 2445.309 3761.383 5267.383 6973.977**

**Quick Sort 5.535 4.266 6.295 7.694 9.434 13.225 18.260 26.933 37.832 62.191 77.815 100.038 121.609**

**Counting Sort 10.565 5.932 5.288 5.465 5.851 6.159 7.477 10.103 12.934 18.486 23.229 26.365 28.949**

1. <https://en.wikipedia.org/wiki/Bubble_sort>
2. <https://learnonline.gmit.ie/pluginfile.php/191471/mod_resource/content/0/08%20Sorting%20Algorithms%20Part%202.pdf>
3. <https://www.tutorialspoint.com/data_structures_algorithms/bubble_sort_algorithm.htm>
4. <http://www.iiitdm.ac.in/old/Faculty_Teaching/Sadagopan/pdf/DAA/SortingAlgorithms.pdf>
5. <https://www.hackerearth.com/practice/algorithms/sorting/selection-sort/tutorial/>
6. <https://www.programiz.com/dsa/selection-sort>
7. <https://en.wikipedia.org/wiki/Insertion_sort>
8. <https://www.studytonight.com/data-structures/insertion-sorting>
9. <https://www.interviewbit.com/tutorial/insertion-sort-algorithm/>
10. <https://www.hackerearth.com/practice/algorithms/sorting/quick-sort/tutorial/>
11. <https://en.wikipedia.org/wiki/Quicksort>
12. <https://runestone.academy/runestone/books/published/pythonds/SortSearch/TheQuickSort.html>