**Data Structures & Algorithm Development - C**

**PROG20799**

**Final Project**

**Instructor’s Name: Muhammad Mohinuddin**

**Submitted By:**

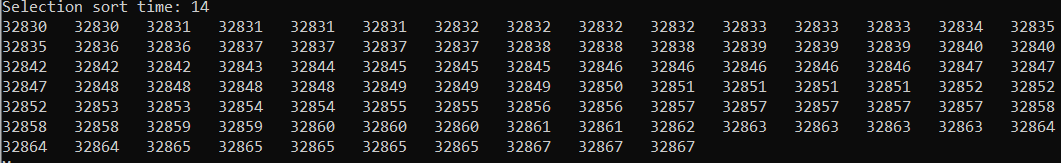
**Prabhdaman Singh Kareer (991478152)**

**Inderpreet Singh Matharu (991491804)**

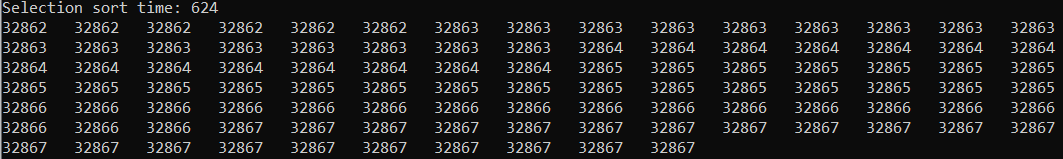
**Gurcharan Singh (991485723)**

**Selection Sort**

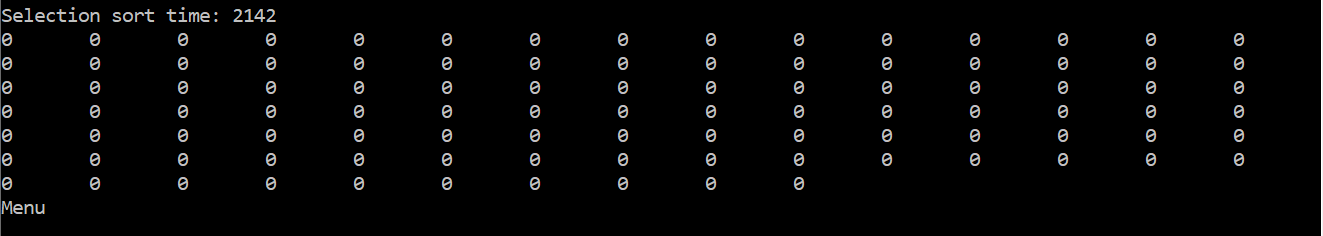
**Selection sort** is a simple sorting algorithm. This sorting algorithm is in-place comparison-based algorithm. It has [**O**](https://en.wikipedia.org/wiki/Big_O_notation)**(*n*2)** [time complexity](https://en.wikipedia.org/wiki/Time_complexity), making it in-efficient on large lists. Selection sort is mentioned for its simplicity, and has overall performance benefits over extra complex algorithms in certain situations, particularly where [auxiliary memory](https://en.wikipedia.org/wiki/Auxiliary_memory) is constrained. The selection sort algorithm sorts an array by repeatedly locating the minimum element from unsorted part and putting it at the start. The algorithm maintains two subarrays in a given array.



**Figure 1.1 – Selection Sort with 100000 records (Small)**

****

**Figure 1.2 – Selection Sort with 100000 records (Medium)**

****

**Figure 1.2 – Selection Sort with 1000000 records (Large)**

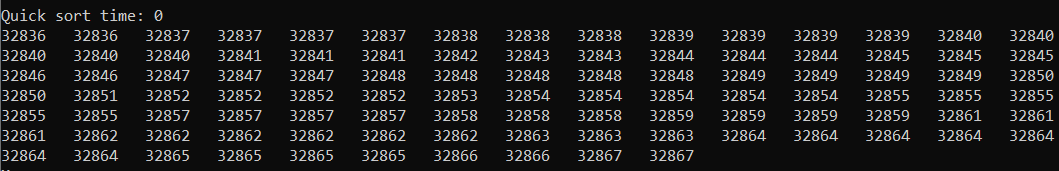
**Note:** *In this case, implementation of linked list was used instead of an array and it was observed that it took drastically longer time i.e. 35 minutes because all the data is stored on heap instead of stack.*

**Quick Sort**

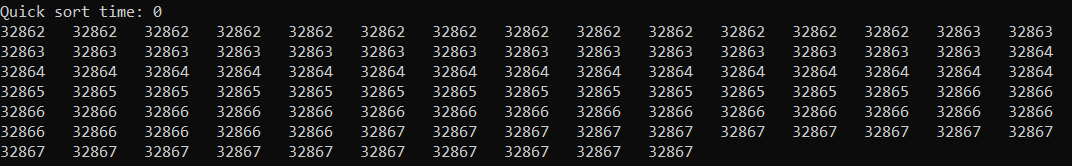
**Quicksort** is a [sorting algorithm](https://en.wikipedia.org/wiki/Sorting_algorithm), serving as a scientific approach for placing the elements of an [array](https://en.wikipedia.org/wiki/Array_data_structure) in order. Quick Sort is a **Divide** and **Conquer** algorithm. Quicksort is an evaluation kind, meaning that it is able to sort objects of any type for which a less-than relation is described. Quicksort can carry out in-area on an array, requiring small extra quantities of memory to perform the sorting. Quicksort first divides a massive array into two smaller sub-arrays: the low elements and the high factors. Quicksort can then recursively sort the sub-arrays.

**Best-case performance - O(n log n)**

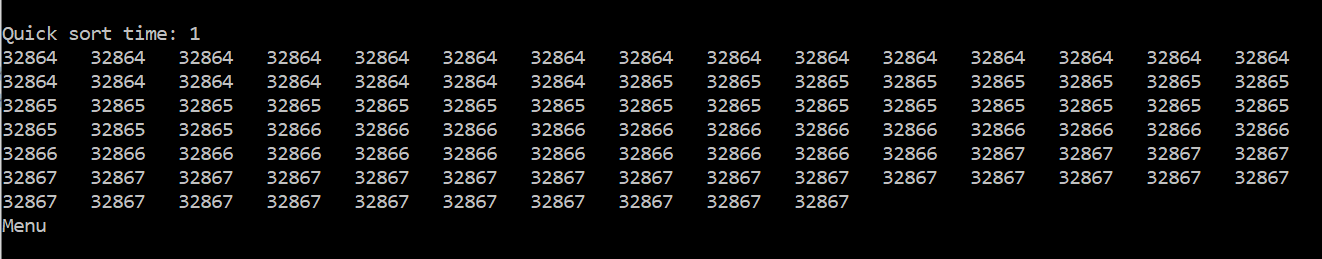
**Worst-case performance – O(n2)**

****

**Figure 2.1 – Quick Sort with 100000 records (Small)**

****

**Figure 2.2 – Quick Sort with 500000 records (Medium)**

****

**Figure 2.2 – Quick Sort with 1000000 records (Large)**

**Insertion Sort**

**Insertion sort** is a simple [sorting algorithm](https://en.wikipedia.org/wiki/Sorting_algorithm) that builds the final [sorted array](https://en.wikipedia.org/wiki/Sorted_array) one item at a time. It is much less efficient on large lists than more superior algorithms such as [quicksort](https://en.wikipedia.org/wiki/Quicksort), [heapsort](https://en.wikipedia.org/wiki/Heapsort), or [merge sort](https://en.wikipedia.org/wiki/Merge_sort).

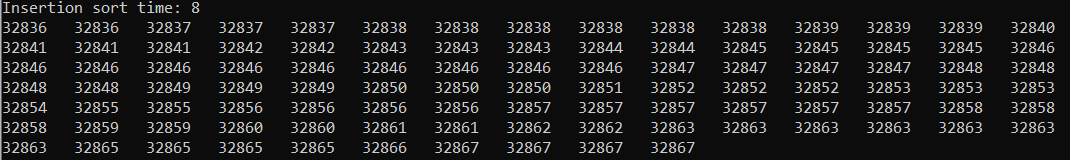
Insertion Sort advantages:

* Simple Implementation
* Efficient for small data sets and data sets that are already substantially sorted
* Stable which means does not change the relative order of elements with equal keys.

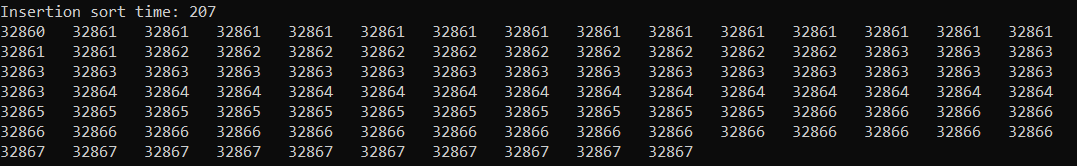
Insertion sort [iterates](https://en.wikipedia.org/wiki/Iteration), consuming one enter element each repetition, and growing a sorted output list. At every iteration, insertion sort eliminates one element from the input data, finds the location it belongs within the sorted list, and inserts it there. It repeats until no input elements remain.

**Best-case performance – O(n2)**

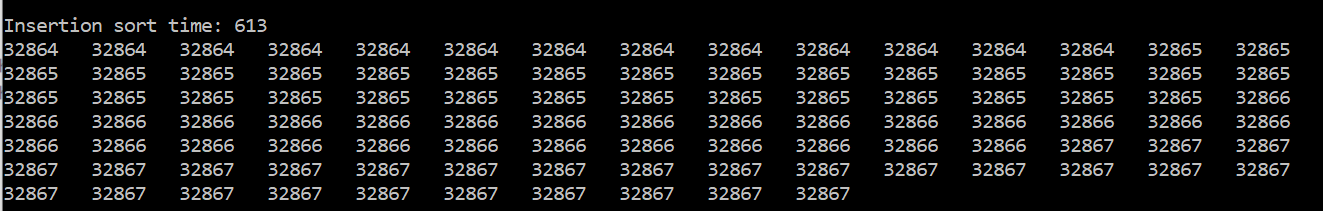
**Worst-case performance – O(n)**

****

**Figure 3.1 – Insertion Sort with 100000 records (Small)**

****

**Figure 3.2 – Insertion Sort with 500000 records (Medium)**



**Figure 3.3 – Insertion Sort with 1000000 records (Large)**

**Comparison**

**Selection Sort VS Insertion Sort**

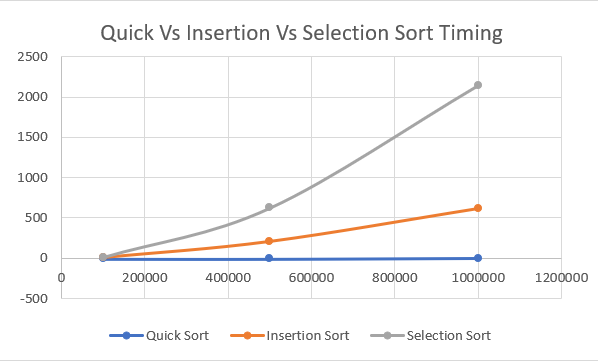
* If there is live data incoming, insertion sort can easily sort it. Selection sort needs all the data to be present in advance.
* If you care approximately the order of similar elements, then insertion sort keeps the ordering stable whereas there may be no such guarantee in selection sort.
* The best case for insertion sort is **O(n)** whereas for selection sort it is **O(n2)**

**Insertion Sort VS Quick Sort**

Insertion sort is faster for small numbers because Quick Sort has more overhead from the recursive function calls. Insertion sort is likewise more stable than Quick sort and calls for less memory.

**Selection Sort VS Quick Sort**

Quick sort is better for large quantity of data than selection sort. Selection sort would perform better in cases where the test data has a larger set of sorted data within. Selection sort looks through every number to find the smallest number and putting it at the front, which is going to take a lot time.

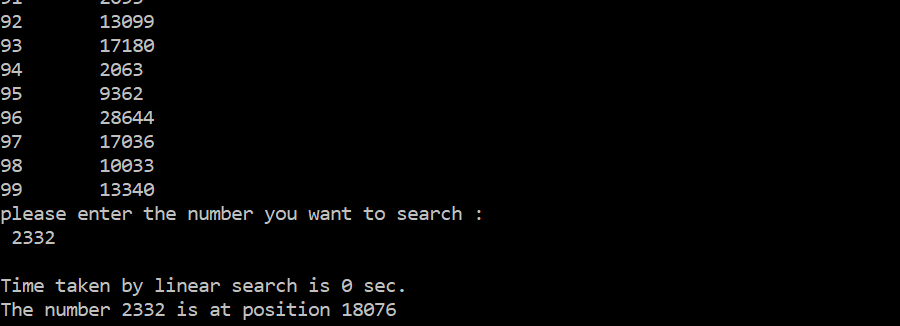


**Linear Search**

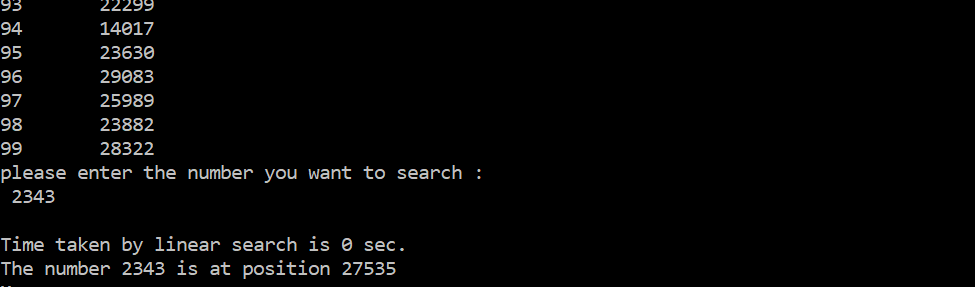
**Linear Search** additionally called **Sequential Search** is a technique for finding an element within a [list](https://en.wikipedia.org/wiki/List_(computing)). A linear search sequentially checks each element of the list until it finds an element that matches the target value. If the program reaches the end of the list, the search terminates unsuccessfully. The time complexity of the linear search is **O(N)** as each element in the array is compared only once. Working of linear search:

* Start from the leftmost element of array and compare one by one the element being searched with each element of the array.
* If a match is found between the element being searched and element of array, the index is returned and if no match is found return -1.

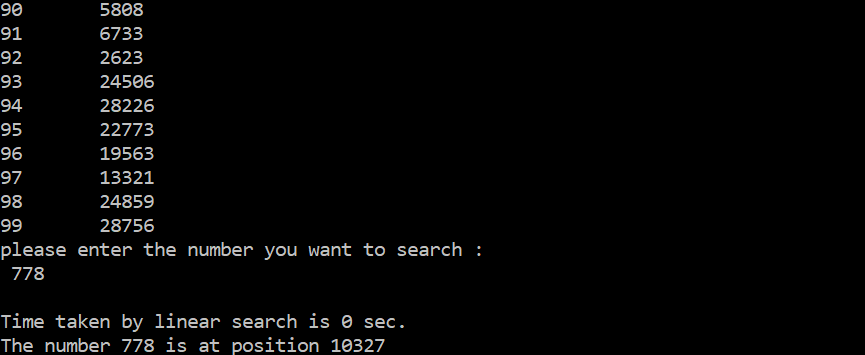
**Best-case performance – O(1) Worst-case performance – O(n)**

****

**Figure 4.1 – Linear Search with 100000 records (Small)**

****

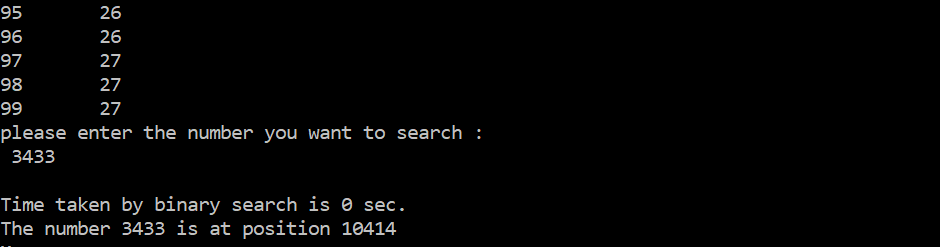
**Figure 4.2 – Insertion Sort with 500000 records (Medium)**

****

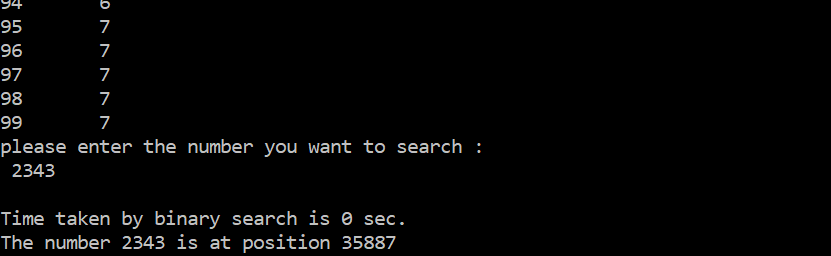
**Figure 4.3 – Linear Search with 1000000 records (Large)**

**Binary Search**

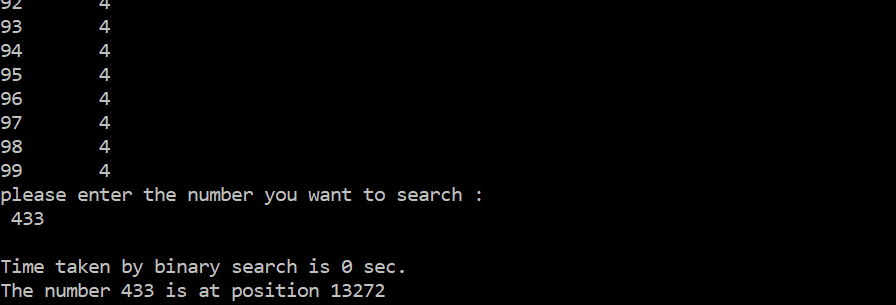
Binary search compares the goal value to the middle element of the array. If they do not match, the half in which the required element is not found is removed and the search continues on the remaining half, again taking the middle element to compare to the target value, and repeating this till the hunt is a success. If the search ends with the remaining half being empty, the element is not in the array. It is also known as **‘Half Interval Search’**. **Best-case performance – O(1) Worst-case performance – O(log n)**

****

**Figure 5.1 – Binary Search with 100000 records (Small)**

****

**Figure 5.1 – Binary Search with 500000 records (Medium)**

****

**Figure 5.1 – Binary Search with 1000000 records (Large)**

**Comparison**

**Linear Search VS Binary Search**

|  |  |  |
| --- | --- | --- |
| **Criteria** | **Linear Search** | **Binary Search** |
| Time Complexity | O(N) | O(log2N) |
| Worst case | N comparisons are required | Concludes after log2N comparisons |
| Insert Operation | Inserted easily at the end of list | Processing required to insert at proper place |
| Type of algorithm | Iterative | Divide and Conquer |
| Coding | Less | More |

**References**

* <https://www.geeksforgeeks.org/selection-sort/>
* <https://www.hackerearth.com/practice/algorithms/sorting/quick-sort/tutorial/>
* <https://en.wikipedia.org/wiki/Selection_sort>
* <https://www.geeksforgeeks.org/quick-sort/>
* <https://en.wikipedia.org/wiki/Quicksort>
* <https://en.wikipedia.org/wiki/Insertion_sort>
* <https://stackoverflow.com/questions/28129472/quicksort-or-selectionsort>
* <https://en.wikipedia.org/wiki/Linear_search>
* <https://www.hackerearth.com/practice/algorithms/searching/linear-search/tutorial/>
* <https://medium.com/karuna-sehgal/an-simplified-explanation-of-linear-search-5056942ba965>
* <https://techdifferences.com/difference-between-linear-search-and-binary-search.html>