**Huong Duong Mai**
**COT 4400 - Analysis of Algorithm**
**Fall 2022**

# ASYMPTOTIC ANALYSIS PROJECT

The given table below includes the computed ratios as well as the behavior of the algorithms. The valid algorithms include Selection Sort (S), Insertion Sort (I), Merge Sort (M), and Quick Sort (Q). The valid input types are sorted (S), random (R), and constant (C) where 'random' is an unsorted array, 'sorted' is a sorted array, and 'constant' is an array with all of the elements are identical.

| | **t_max/t_min** | **n ratio** | **nln(n) ratio** | **n^2 ratio** | **Behavior** |
|---|---|---|---|---|---|
| **SC** | 9075.941176 | 97.56097561 | 147.1483417 | 9518.143962 | Quadratic |
| **SS** | 12223.84375 | 112.5 | 171.6210194 | 12656.25 | Quadratic |
| **SR** | 11269.87097 | 113.3333333 | 173.4168226 | 12844.44444 | Quadratic |
| **IC** | 63.40625 | 60.0000024 | 74.77310058 | 3600.000288 | Linear |
| **IS** | 57.61111111 | 53.28947396 | 65.94000025 | 2839.768035 | Linear |
| **IR** | 6794.235294 | 81.00445525 | 119.7063169 | 6561.72177 | Quadratic |
| **MC** | 2458.935484 | 1612.903226 | 2506.068272 | 2601456.816 | nlgn |
| **MS** | 2664.16129 | 1636.363934 | 2545.27648 | 2677686.924 | nlgn |
| **MR** | 5388.333333 | 3333.333333 | 5477.329897 | 11111111.11 | nlgn |
| **QC** | 123.9142857 | 11.22082585 | 14.20366629 | 125.9069328 | Quadratic |
| **QS** | 1646.848485 | 1282.051282 | 1958.295363 | 1643655.49 | nlgn |
| **QR** | 4380.945946 | 2857.142857 | 4638.161997 | 8163265.306 | nlgn |

The chart below is the theoretical analysis for four algorithms:

| | Best-case complexity | Average-case complexity | Worst-case complexity |
|---|---|---|---|
| SelectionSort | $\Theta(n^2)$ | $\Theta(n^2)$ | $\Theta(n^2)$ |
| InsertionSort | $\Omega(n)$ | $\Theta(n^2)$ | $O(n^2)$ |
| MergeSort | $\Theta(n \lg n)$ | $\Theta(n \lg n)$ | $\Theta(n \lg n)$ |
| QuickSort | $\Omega(n \lg n)$ | $\Theta(n \lg n)$ | $O(n^2)$ |

## SELECTION SORT

**Constant Input:**
As we can see from the table above, the behavior for selection sort with constant input is quadratic $O(n^2)$, which is similar to the theoretical analysis for the selection sort algorithm.

It makes sense because, with constant inputs, all of the elements in the input array are the same. Selection Sort will take O(n) times to loop through the array. Specifically, the time complexity for iteration $i^{th}$ will be O(n-i-1) = O(n) (this is the time complexity for the inner loop). This algorithm is used to search the minimum element O(n) and swap it with the leftmost unsorted element in the array O(1). With n inputs, there will have O(n) iterations (for the outer loop). Therefore, we time complexity is $O(n)*O(n) = O(n^2)$ - quadratic

**Sorted Input:**
As we can see from the table above, the behavior for selection sort with sorted input is still quadratic $O(n^2)$, which is similar to the theoretical analysis for the best case in the selection sort algorithm.

Similarly, although with sorted inputs, which means all of the elements in the array are in increasing order, selection sort sill loop through the array O(n) time to search for the smallest element and then swap it with the leftmost unsorted element in the array O(n). However, even though it is already an ascending-order array and no swap will be executed (best case), the selection sort still has O(n) iterations. Then, the time complexity is still $O(n)*O(n) = O(n^2)$ - quadratic

**Random Input:**
In the same way, the behavior for selection sort with random input is similar to the theoretical analysis for the selection sort algorithm, quadratic, $O(n^2)$.

It still makes sense because when the array has random elements, the selection sort will still loop through the array with O(n) times. Selection sort will find the minimum element in the array and then change the positions with the leftmost unsorted elements in the array. There are O(n) iterations in the selection sort. Therefore, the time complexity will be $O(n)*O(n) = O(n^2)$.

## INSERTION SORT

### Constant Input

It can be seen from the table that the time complexity for insertion sort with constant inputs is linear O(n). The result is similar to the theoretical analysis for the best-case complexity insertion sort algorithm.

It makes sense when the time complexity is O(n). The insertion sort loops through the array in O(n) time and comparing each pair of values in the array is also taking O(n) time. If the values are not in the same order, they will be swapped. However, because all of the elements in the array are the same, therefore the inner loop will not be executed (which also means it will take O(1) time. Insertion sort with n constant inputs will lead to O(n) iteration. As a result, the time complexity will be O(n) * O(1) = O(n)

### Sorted Input

As can be seen from the table, the behavior for insertion sort with sorted input is also linear O(n). This is similar to the theoretical analysis for the best-case complexity insertion sort algorithms.

Similarly to constant input, it takes O(n) time for the insertion sort to loop through the array and O(n) time to check if each pair of values is in order. However, the inputs are already sorted so swapping will not be swapped, which means the inner loop will not be executed, which leads to O(1). Therefore, the time complexity will be equal to the number of iterations, O(n) with n inputs

### Random Input

The behavior for insertion sort with random input is quadratic O($n^2$), which is similar to the average-case and worst-case complexity in theoretical analysis for insertion sort.

It makes sense because, in insertion sort, it will take O(n) time for the algorithm to loop through the array and another O(n) time to check if each pair of values is out of order and swap them. As a result, with O(n) iterations from n inputs, and O(n) time from the inner loop because the inputs are random, the time complexity for the random input will be O(n) * O(n) = O($n^2$)

## MERGE SORT

**Constant Input**
From the result table, the merge sort algorithm with constant inputs has time complexity as O(n lgn), which is similar to the three-case complexity theoretical analysis for merge sort

With identical elements in the array, the array will be divided into smaller subarrays with one element, which take O(lgn). Then, the two adjacent subarrays will be combined into bigger sorted subarrays and the process will be repeated until all of them are merged into one sorted array. This will also take O(n). Therefore, the time complexity will be O(lgn)*O(n) = O(n lgn). On the other way, when we write Merge Sort under Master Theorem recurrence relation, it will be T(n) = 2T(n/2) + O(n). Using the master theorem, we will also find the time complexity that will take O(nlgn).

**Sorted Input**
As can be seen from the result table, the time complexity for sorted inputs is also O(n lgn), which is similar to the merge sort complexity in theoretical analysis.

With sorted inputs, the array is already in ascending order. Again, the merge sort will start with dividing the array into one-element subarrays, which will take O(lgn) time. Then, it will take O(n) to merge all of the subarrays into a bigger sorted array and repeat until all of the subarrays become one big sorted array. Therefore, the time complexity will be O(lgn) * O(n) = O(n lgn)

**Random Input**
Time complexity for merge sort with random inputs is also O(n lgn). Compared this result to the time complexity for merge sort in theoretical analysis, they are the same: O(n lgn)

With the same process as constant or sorted input, will merge sort use divide and conquer rule. It will divide the given array into one-element arrays, which take O(lgn) time and O(n) to merge all of the subarrays into a bigger sorted array and repeat until all of the subarrays become one big sorted array. The time complexity that will take  O(lgn) * O(n) = O(nlgn).

## QUICK SORT

**Constant Input**

As we can see from the table above, the time complexity for quick sort with constant inputs is quadratic O($n^2$). Compared to the theoretical analysis chart, it can be stated that the result is similar to the worst-case complexity for quick sort.

It makes sense because when all the elements are identical, the pivot will have the same value as the rest of the array. As a result, two subarrays with the size n-1 and size 1 will be returned. With n input, we will have n-1 iterations. The time complexity will be calculated by adding the time complexity of each partition: O(n) + O(n-1) + O(n-2) + ,,, + 2 = O(n(n+1)/2 - 1) = O($n^2$).

**Sorted Input**

It makes sense that the time complexity for quick sort with sorted input is O(n lgn) which is the same as the best-case and average-case complexity for quick sort in theoretical analysis.

With sorted inputs, when the pivot is chosen from the array, it is a small chance that the pivot will be the first or last element (which is the worst case). Therefore, after the quick sort picking of the pivot, the smaller elements will be moved to the left and the bigger ones to the right. Then, the process will be repeated and combined until the whole array is sorted. The algorithm will take O(lgn) to divide the array into halves. And the partition has O(n) time complexity. Then, the time complexity for quick sort with sorted input is O(lgn) * O(n) = O(n lg n)

**Random Input**

It can be seen that the time complexity for the quick sort with random input is O(n lgn), which is similar to theoretical analysis best-case and average-case complexity for the quick sort.

In this case, the pivot is picked from the array will be a random number. Similarly to the sorted input, it is difficult to pick the smallest or the biggest element as a pivot. Therefore, after choosing a pivot, the array will be divided into two, the left one is used for elements that are smaller than the pivot and the right one is used for elements that are bigger than the pivot. It takes O(lg n) to divide the array into half. The process will be repeated until all elements are sorted. With n elements, the algorithm will take O(n) time. As a result , the time complexity for quick sort with random input is O(lgn) * O(n) = O(n lg n)