Lab Assignment #5
Quick Sort

Full Name: Shreyas Srinivasa
BlazerId: SSRINIVA

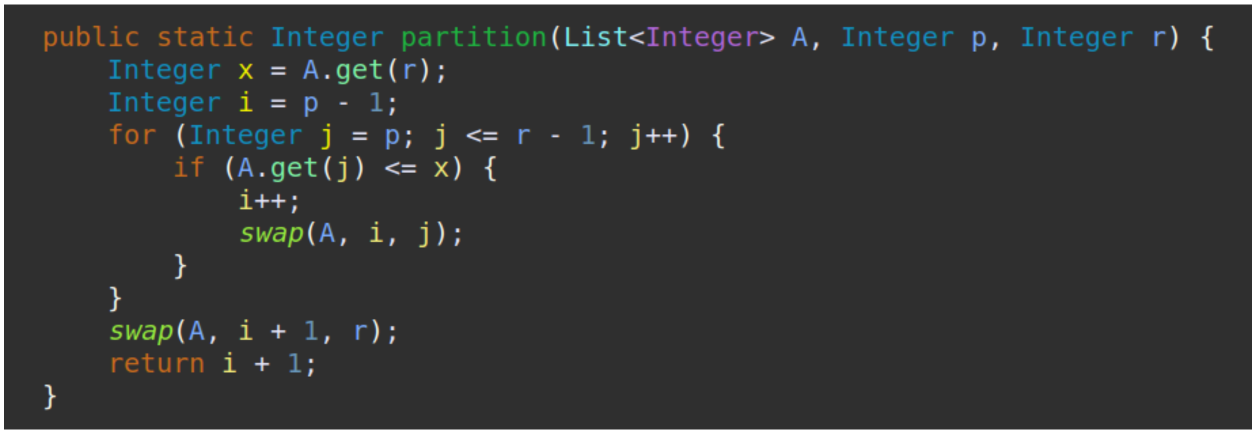
CS 303 Algorithms and Data Structures

Spring 2023

1. Problem Specification

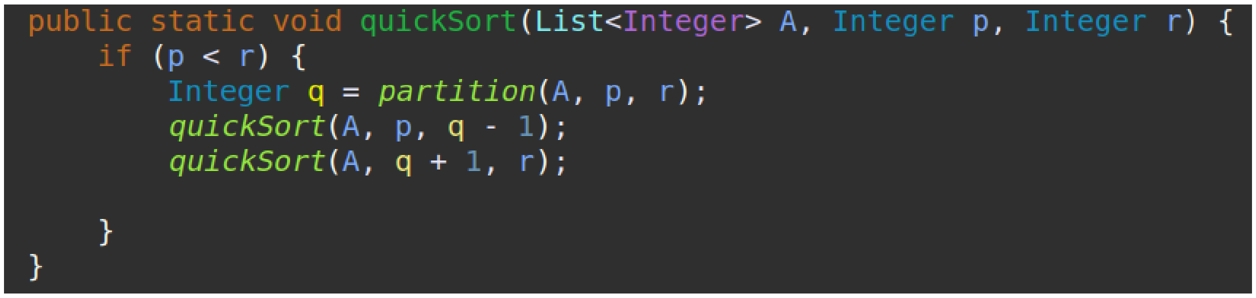
The purpose of this project was to implement an algorithm for quick sort sort and measure its execution time with a number of datasets of progressively increasing size. This execution time was used to study the connection between real time performance and theoretical complexity analyses. The algorithm was also tested with sorted, reverse sorted and unsorted datasets. We enhance the performance of quick sort by creating a version which selects the median of the dataset as the pivot. The data collected for quick sort was further compared to the data collected for heap, insertion sort and merge sort and all were studied together to determine which algorithm performs best.

2. Program Design



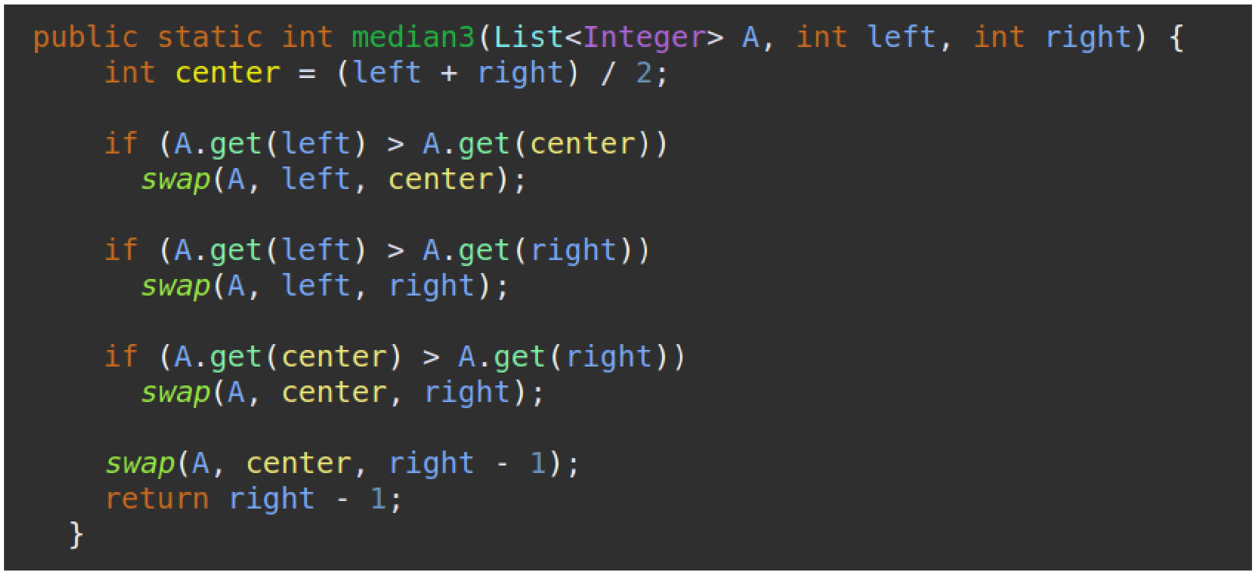
```java
public static Integer partition(List<Integer> A, Integer p, Integer r) {
    Integer x = A.get(r);
    Integer i = p - 1;
    for (Integer j = p; j <= r - 1; j++) {
        if (A.get(j) <= x) {
            i++;
            swap(A, i, j);
        }
    }
    swap(A, i + 1, r);
    return i + 1;
}
```

The core algorithm of quick sort, which is used recursively on partitions divided by the selected pivot element. The rearrangement eventually leads to a fully sorted sequence.



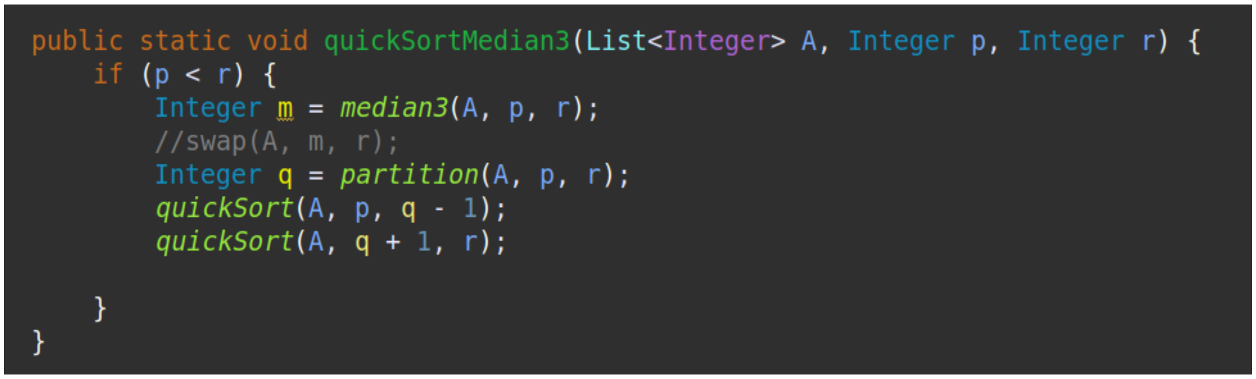
```java
public static void quickSort(List<Integer> A, Integer p, Integer r) {
    if (p < r) {
        Integer q = partition(A, p, r);
        quickSort(A, p, q - 1);
        quickSort(A, q + 1, r);

    }
}
```

The entry point of quick sort, which recursively calls itself on each of the two pieces formed by the partition



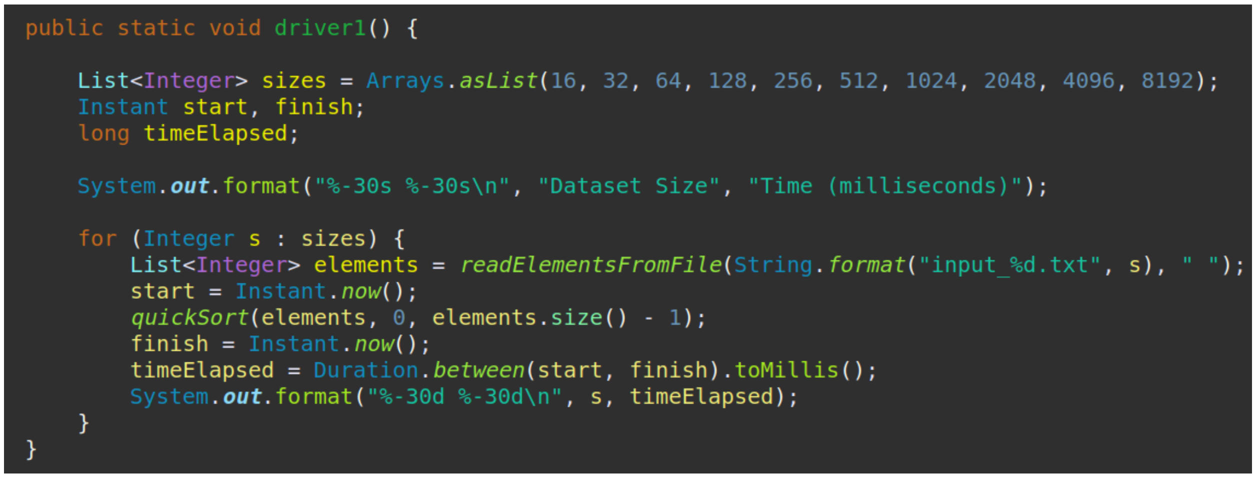
```java
public static int median3(List<Integer> A, int left, int right) {
    int center = (left + right) / 2;

    if (A.get(left) > A.get(center))
      swap(A, left, center);

    if (A.get(left) > A.get(right))
      swap(A, left, right);

    if (A.get(center) > A.get(right))
      swap(A, center, right);

    swap(A, center, right - 1);
    return right - 1;
  }
```

This function is used to optimize the algorithm by shifting the median of the values located in the indexes left, right and center to the right index

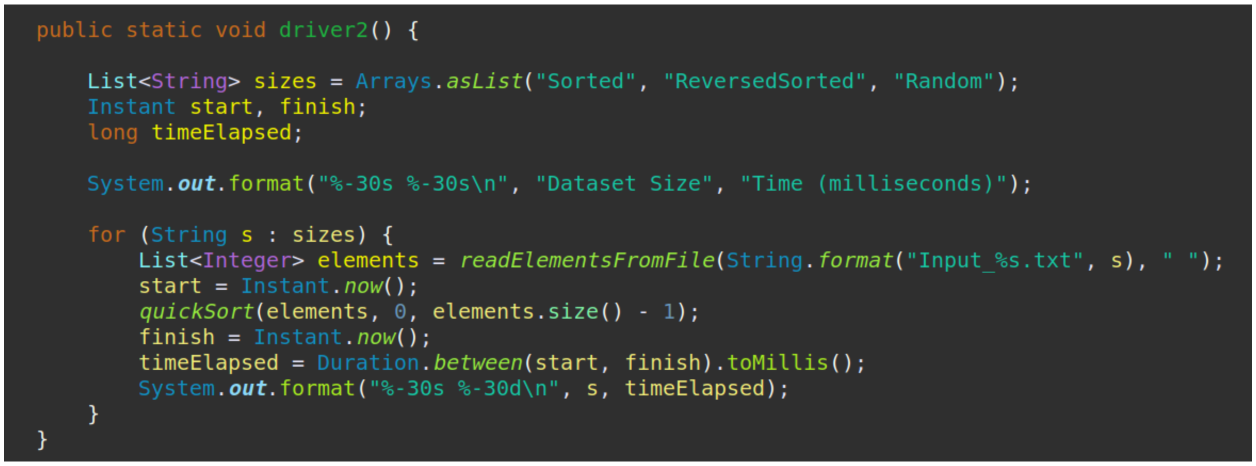


```java
public static void quickSortMedian3(List<Integer> A, Integer p, Integer r) {
    if (p < r) {
        Integer m = median3(A, p, r);
        //swap(A, m, r);
        Integer q = partition(A, p, r);
        quickSort(A, p, q - 1);
        quickSort(A, q + 1, r);

    }
}
```

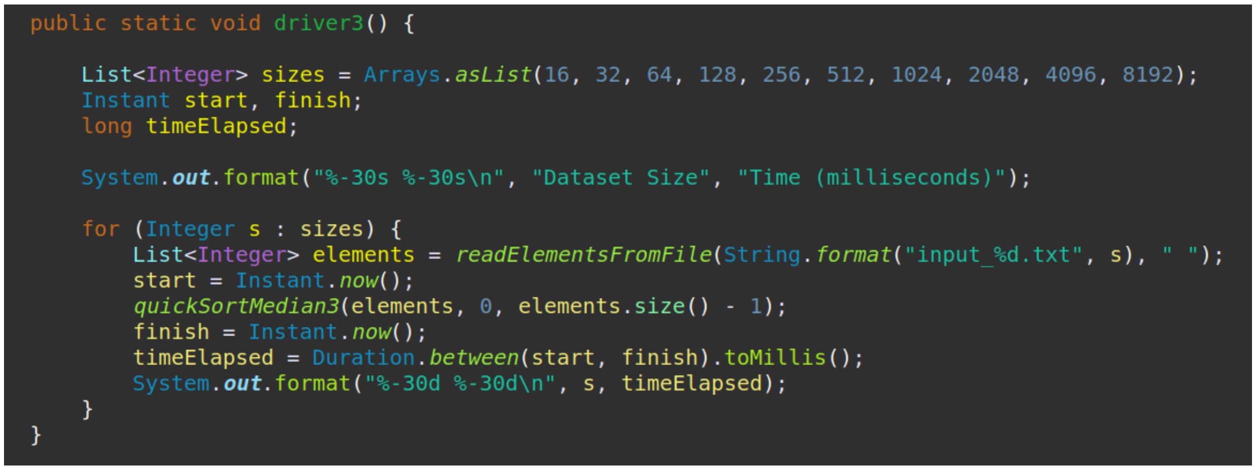
The entry point for our optimized version of quick sort



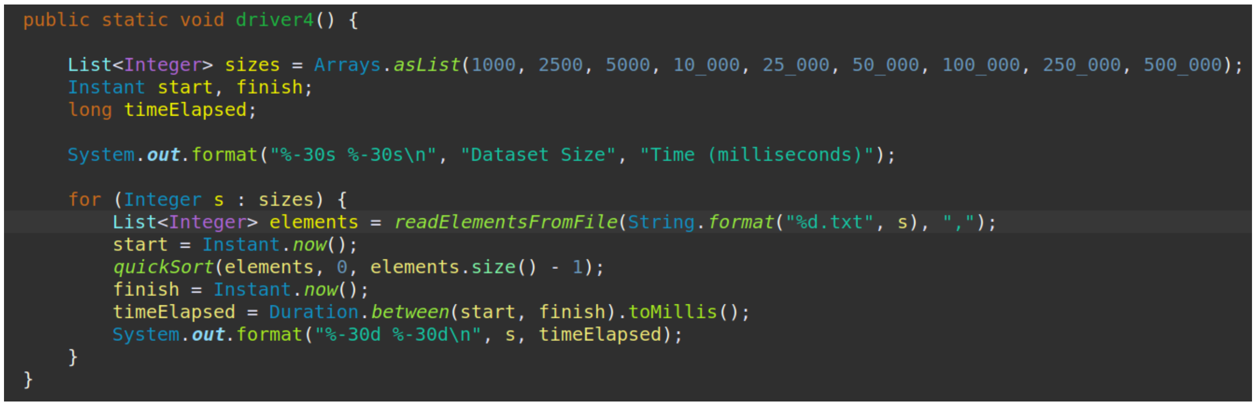
```java
public static void driver1() {

    List<Integer> sizes = Arrays.asList(16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192);
    Instant start, finish;
    long timeElapsed;

    System.out.format("%-30s %-30s\n", "Dataset Size", "Time (milliseconds)");

    for (Integer s : sizes) {
        List<Integer> elements = readElementsFromFile(String.format("input_%d.txt", s), " ");
        start = Instant.now();
        quickSort(elements, 0, elements.size() - 1);
        finish = Instant.now();
        timeElapsed = Duration.between(start, finish).toMillis();
        System.out.format("%-30d %-30d\n", s, timeElapsed);
    }
}
```



```java
public static void driver2() {

    List<String> sizes = Arrays.asList("Sorted", "ReversedSorted", "Random");
    Instant start, finish;
    long timeElapsed;

    System.out.format("%-30s %-30s\n", "Dataset Size", "Time (milliseconds)");

    for (String s : sizes) {
        List<Integer> elements = readElementsFromFile(String.format("Input_%s.txt", s), " ");
        start = Instant.now();
        quickSort(elements, 0, elements.size() - 1);
        finish = Instant.now();
        timeElapsed = Duration.between(start, finish).toMillis();
        System.out.format("%-30s %-30d\n", s, timeElapsed);
    }
}
```



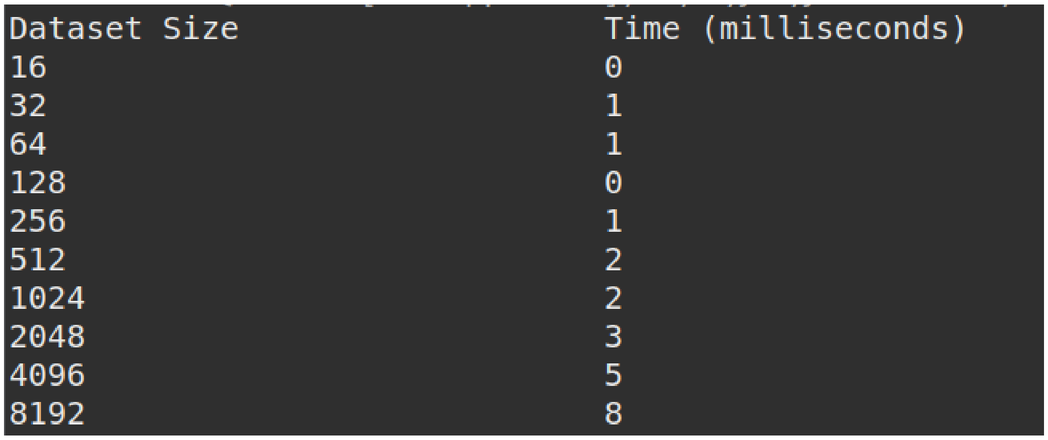
```java
public static void driver3() {

    List<Integer> sizes = Arrays.asList(16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192);
    Instant start, finish;
    long timeElapsed;

    System.out.format("%-30s %-30s\n", "Dataset Size", "Time (milliseconds)");

    for (Integer s : sizes) {
        List<Integer> elements = readElementsFromFile(String.format("input_%d.txt", s), " ");
        start = Instant.now();
        quickSortMedian3(elements, 0, elements.size() - 1);
        finish = Instant.now();
        timeElapsed = Duration.between(start, finish).toMillis();
        System.out.format("%-30d %-30d\n", s, timeElapsed);
    }
}
```



```java
public static void driver4() {

    List<Integer> sizes = Arrays.asList(1000, 2500, 5000, 10_000, 25_000, 50_000, 100_000, 250_000, 500_000);
    Instant start, finish;
    long timeElapsed;

    System.out.format("%-30s %-30s\n", "Dataset Size", "Time (milliseconds)");

    for (Integer s : sizes) {
        List<Integer> elements = readElementsFromFile(String.format("%d.txt", s), ",");
        start = Instant.now();
        quickSort(elements, 0, elements.size() - 1);
        finish = Instant.now();
        timeElapsed = Duration.between(start, finish).toMillis();
        System.out.format("%-30d %-30d\n", s, timeElapsed);
    }
}
```

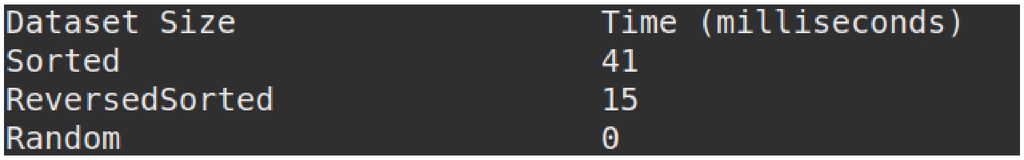
The driver functions which provides our results with different datasets and implementations

3. **Output**



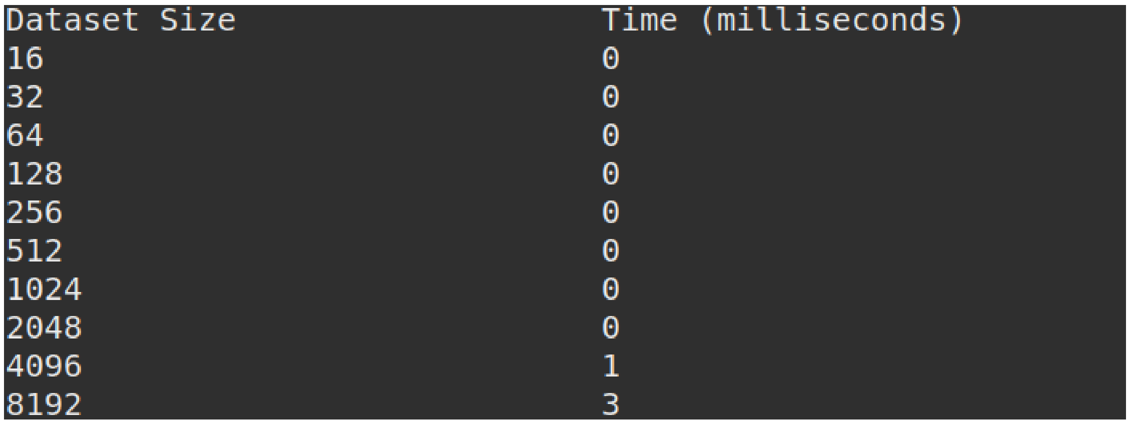
```
Dataset Size                   Time (milliseconds)
16                             0
32                             1
64                             1
128                            0
256                            1
512                            2
1024                           2
2048                           3
4096                           5
8192                           8
```

Output 1. Ordinary Quick Sort Results



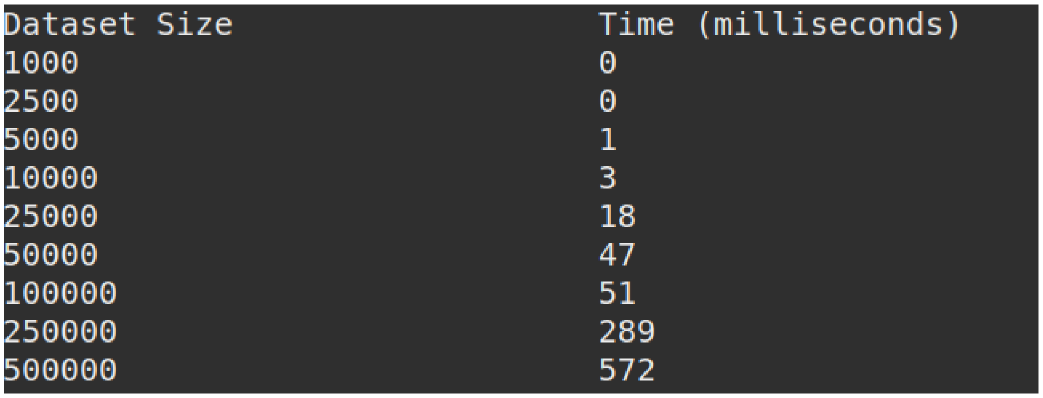
```
Dataset Size                   Time (milliseconds)
Sorted                         41
ReversedSorted                 15
Random                         0
```

Output 2. Quick Sort Results with Sorted, Reverse Sorted and Unsorted Datasets



```
Dataset Size                   Time (milliseconds)
16                             0
32                             0
64                             0
128                            0
256                            0
512                            0
1024                           0
2048                           0
4096                           1
8192                           3
```

Output 3. Quick Sort Results with Merge 3 Optimized Sort



```
Dataset Size                   Time (milliseconds)
1000                           0
2500                           0
5000                           1
10000                          3
25000                          18
50000                          47
100000                         51
250000                         289
500000                         572
```

Output 4. Quick Sort Results with large datasets used for previous sort algorithms

Dataset Size	Time (milliseconds)
16	0
32	1
64	1
128	0
256	1
512	2
1024	2
2048	3
4096	5
8192	8

Table 1. Ordinary Quick Sort Results

Dataset Type	Time (milliseconds)
Sorted	41
Reverse Sorted	15
Random	0

Table 2. Quick Sort Results with Sorted, Reverse Sorted and Unsorted Datasets

Dataset Size	Time (milliseconds)
16	0
32	0
64	0
128	0
256	0
512	0
1024	0
2048	0
4096	1
8192	3

Table 3. Quick Sort Results with Merge3 Optimized Sort

Dataset Size	Time (milliseconds)
1000	0
2500	0
5000	1
10000	3
25000	18
50000	47
100000	51
250000	289
500000	572

Output 4. Quick Sort Results with large datasets used for previous sort algorithms

Dataset Size	Time (milliseconds)
1000	20
2500	17
5000	58
10000	185
25000	1435
50000	6813
100000	25776
250000	343166
500000	2426168
1000000	10414209

Output 5. Insertion Sort Results

Dataset Size	Time (milliseconds)
1000	13
2500	3
5000	8
10000	5
25000	8
50000	18
100000	44
250000	174
500000	379
1000000	838

Output 5. Merge Sort Results

Dataset Size	Time (milliseconds)
1000	6
2500	3
5000	2
10000	9
25000	18
50000	34
100000	84
250000	374
500000	806

Output 6. Heap Sort Results

4. Analysis and Conclusions

Analysis of Quicksort (GeeksForGeeks):-

Time taken by QuickSort, in general, can be written as follows.

$T(n) = T(k) + T(n\text{-}k\text{-}1) + \theta(n)$

The first two terms are for two recursive calls, the last term is for the partition process. k is the number of elements that are smaller than the pivot.
The time taken by QuickSort depends upon the input array and partition strategy. Following are three cases.

Worst Case:
The worst case occurs when the partition process always picks the greatest or smallest element as the pivot. If we consider the above partition strategy where the last element is always picked as a pivot, the worst case would occur when the array is already sorted in increasing or decreasing order. Following is recurrence for the worst case.

$T(n) = T(0) + T(n\text{-}1) + \theta(n)$*which is equivalent to* $T(n) = T(n\text{-}1) + \theta(n)$

The solution to the above recurrence is (n^2).

Best Case:
The best case occurs when the partition process always picks the middle element as the pivot. The following is recurrence for the best case.

$T(n) = 2T(n/2) + \theta(n)$

The solution for the above recurrence is (nlogn). It can be solved using case 2 of Master Theorem.

Average Case:
To do average case analysis, we need to consider all possible permutation of array and calculate time taken by every permutation which doesn't look easy.
We can get an idea of average case by considering the case when partition puts O(n/9) elements in one set and O(9n/10) elements in other set. Following is recurrence for this case.

$T(n) = T(n/9) + T(9n/10) + \theta(n)$

The solution of above recurrence is also O(nlogn)

Although the worst case time complexity of QuickSort is $O(n^2)$ which is more than many other sorting algorithms like Merge Sort and Heap Sort, QuickSort is faster in practice, because its inner loop can be efficiently implemented on most architectures, and in most real-world data. QuickSort can be implemented in different ways by changing the choice of pivot, so that the worst case rarely occurs for a given type of data. However, merge sort is generally considered better when data is huge and stored in external storage.

Why Quick Sort is preferred over MergeSort for sorting Arrays?

Quick Sort in its general form is an in-place sort (i.e. it doesn't require any extra storage) whereas merge sort requires **O(n)** extra storage, n denoting the array size which may be quite expensive. Allocating and de-allocating the extra space used for merge sort increases the running time of the algorithm. Comparing average complexity we find that both type of sorts have **O(nlogn)** average complexity but the constants differ. For arrays, merge sort loses due to the use of extra **O(n)** storage space.

Most practical implementations of Quick Sort use randomized version. The randomized version has expected time complexity of **O(nlogn)**. The worst case is possible in randomized version also, but worst case doesn't occur for a particular pattern (like sorted array) and randomized Quick Sort works well in practice.
Quick Sort is also a cache friendly sorting algorithm as it has good locality of reference when used for arrays.

Quick Sort is also tail recursive, therefore tail call optimizations is done.

Why MergeSort is preferred over QuickSort for Linked Lists ?

In case of linked lists the case is different mainly due to difference in memory allocation of arrays and linked lists. Unlike arrays, linked list nodes may not be adjacent in memory. Unlike array, in linked list, we can insert items in the middle in O(1) extra space and O(1) time. Therefore merge operation of merge sort can be implemented without extra space for linked lists. In arrays, we can do random access as elements are continuous in memory. Let us say we have an integer (4-byte) array A and let the address of A[0] be x then to access A[i], we can directly access the memory at (x + i*4). Unlike arrays, we can not do random access in linked list. Quick Sort requires a lot of this kind of access. In linked list to access i'th index, we have to travel each and every node from the head to ith node as we don't have continuous block of memory. Therefore, the overhead increases for quick sort. Merge sort accesses data sequentially and the need of random access is low.

Advantages and Disadvantages

Advantages of Quick Sort:
•It is a divide-and-conquer algorithm that makes it easier to solve problems.
•It is efficient on large data sets.
•It is a stable sort, meaning that if two elements have the same key, their relative order will be preserved in the sorted output.
•It has a low overhead, as it only requires a small amount of memory to function.

Disadvantages of Quick Sort:
•It has a worst-case time complexity of O(n^2), which occurs when the pivot is chosen poorly.
•It is not a good choice for small data sets.
•It can be sensitive to the choice of pivot.
•It is not cache-efficient.

Summary:
•Quick sort is a fast and efficient sorting algorithm with an average time complexity of **O(nlog n)**.

•It is a divide-and-conquer algorithm that breaks down the original problem into smaller subproblems that are easier to solve.
•It can be easily implemented in both iterative and recursive forms and it is efficient on large data sets, and can be used to sort data in-place.
•However, it also has some drawbacks such as worst case time complexity of **$O(n^2)$** which occurs when the pivot is chosen poorly.
•It is not a good choice for small data sets, it is not cache-efficient, and is sensitive to the choice of pivot.

Median of 3 Method

We take 3 values in positions low, high, (low+high)/2, find the median and place in the low or high position. This is an optimization technique which chooses the best possible pivot for the partition operation

As we already know from previous labs:-
The Time Complexity of an algorithm/code is **not** equal to the actual time required to execute a particular code, but the number of times a statement executes.

The **actual time required to execute code is machine-dependent**.

Instead of measuring actual time required in executing each statement in the code,Time Complexity considers how many times each statement executes.

Observations based on the lab experiment

- As we can see, even the ordinary quick sort algorithm is extremely efficient with small datasets
- This algorithm is sensitive to the choice of the pivot element, and its worst case and best case performance can be visualized clearly with the sorted, unsorted and reverse sorted datasets. The performance always suffers with sorted datasets and is the best for randomized, shuffled datasets

Median3 Performance

- There are always a myriad of techniques that can be used to optimize the performance of a sorting algorithm. We have already tried using insertion sort for very small partitions along with merge sort to enhance performance. Here we see a drastic increase in performance which is a result of choosing the median of the first, last and middle element of the partition. The time taken for sorting most of the datasets is virtually 0 ms.

Performance Compared to Heap Sort, Merge Sort and Insertion Sort

- The comparison clearly shows that merge sort and quick sort are the most practical algorithms for real world scenarios. The GeeksForGeeks analysis points out the factors which can determine which algorithm should be picked for which scenario. The quick sort performs poorly with linked lists and sorted sequences, but is space efficient.

5. References

- https://www.geeksforgeeks.org/quick-sort/
- https://www.cs.cornell.edu/courses/JavaAndDS/files/sort3Quicksort3.pdf

- https://www.cs.princeton.edu/courses/archive/fall12/cos226/lectures/23Quicksort.pdf
- http://www.java2s.com/Tutorial/Java/0140__Collections/Quicksortwithmedianofthreepartitioning.htm