JAYADITYA KAKARALA

STUDENT ID: 110026622

"I confirm that I will keep the content of this assignment confidential. I confirm that I have not received any unauthorized assistance in preparing for or writing this assignment. I acknowledge that a mark of 0 may be assigned for copied work." JAYADITYA KAKARALA, 110026622

1. **Use class Sort.java provided in class, the dual-pivot Quicksort of Java 8 (Arrays.sort), and RadixSort.java provided in class**

   Imported **Sort.java** class (which comprises of Mergesort, Quicksort, Heapsort) and **RadixSort.java** from the assets gave and utilized the predefined **dual-pivot Quicksort** in the java (Arrays.sort) technique into the **test.java** file to utilize those sort strategies.

2. **Do the following for Mergesort, Quicksort, Heapsort and dual-pivot Quicksort:**

   a. **Create 100,000 random keys (of type long) and sort them. Repeat this 100 times.**

   Location of the code: C:\Users\jayad\eclipse-workspace\Assignment2_110026622\src
   Reference of the code: Advanced Computing Concepts Lab Source Code
   Java files: test.java, Sort.java
   Method Name: sort (int n);
   Variables used: long avg, sumM, sumQ, sumH, sumDQ, start1, start2;
   int size, number, l=100000, num, i, j, k;
   long string1; // A Long type variable used to generate a random key
   Methods used: mergeSort(), quicksort(), heapsort(), Arrays.sort(), copy();
   Procedure: Initialized and defined 2 **arrays a**, **b** of type **long** and **size 100000** at the beginning, One to store those randomly generated numbers and afterward it is sorted by utilizing one of the sorting technique mentioned above and we measure that time utilizing **start1** and **start2** variables in **nano seconds** the and the other array has the duplicate of the randomly generated numbers which is utilized to copy those strings into the first array after the sorting is finished then the first array is again ready for sorting by utilizing another sorting procedure. This Entire methodology is done for multiple times and afterward the average time for all the arranging procedure is determined and changed over it into **milli seconds**.

   b. **Compute the average CPU time taken to sort the keys for the four methods.**

   Average time can be calculated by dividing the **sum** of each time took to sort the 100000 random generated numbers for 100 times to the **number of times** it had been repeated i.e. **100**. To calculate sum there are 4 different variables of type long for the four different sort methods which adds the previous sum to current time took to sort the numbers.

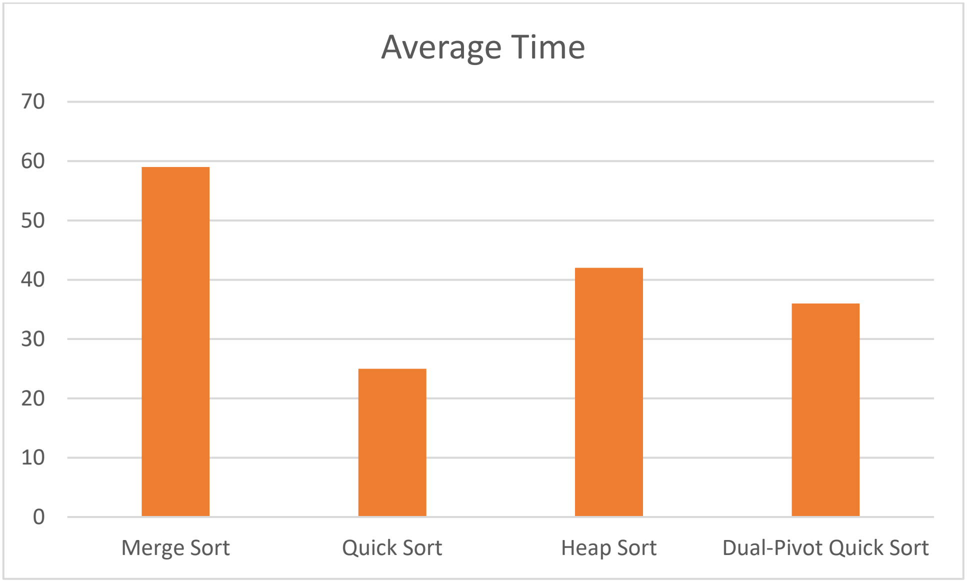
Table 1.1: Average Time calculated for sorting 100000 keys of 100 times in Milli Seconds

Types of Sorting	Merge Sort	Quick Sort	Heap Sort	Dual-Pivot Quick Sort
Average Time	59	25	42	36

c. Comment on the results and compare them to the average-case complexities discussed in class.

Despite the fact that all the sorts for example Merge sort, Quick sort, Heap sort and Dual-Pivot Quick sort has a similar time-complexity in normal case i.e. O(n log n). By observing the underneath outline, we can obviously say that Quick Sort sets aside less effort to sort those 100000 irregular keys for multiple times.

Figure 1.1: Average Time calculated for sorting 100000 keys of 100 times in Milli Seconds



In the Above figure, Merge sort sets aside more effort to sort the 100000 random generated numbers as a result of recursion of sort technique which needs to sort each and every single number for numerous number of times due to divide and conquer rule which consists of sub arrays and the Heap sort also sets more effort to sort since it has to allocate heaps for every number and again re arrange the heaps based on the sorting. Whereas Quick Sort takes less time since there is no compelling reason to sort a similar component various number of times as Merge sort does and there is no compelling reason to designate stacks like heap sort.

3. Do the following for the four sorting methods of #2, and for Radix sort:

a. Create 100,000 random strings of length 4 and sort them using the five sorting methods.

Location of the code: C:\Users\jayad\eclipse-workspace\Assignment2_110026622\src
Reference of the code: Advanced Computing Concepts Lab Source Code

Java files:	test.java, Sort.java, RadixSort.java
Method Name:	sort (int n);
Variables used:	long avg, sumM, sumQ, sumH, sumDQ, start1, start2;
	int size, number, l=100000, num, i, j, k;
	String generatedString; // A String type variable used to generate a random key
Methods used:	mergeSort(), quicksort(), heapsort(), Arrays.sort(), radixSortA(), copy();

Procedure: Initialized and defined 2 **arrays c**, **d** of type **String** and **size 100000** at the beginning, One to store those randomly generated strings and afterward it is sorted by utilizing one of the sorting technique mentioned above and we measure that time utilizing **start1** and **start2** variables in **nano seconds** the and the other array has the duplicate of the randomly generated strings which is utilized to copy those strings into the first array after the sorting is finished then the first array is again ready for sorting by utilizing another sorting procedure. This Entire methodology is done for multiple times and afterward the average time for all the arranging procedure is determined and changed over it into **milli seconds**.

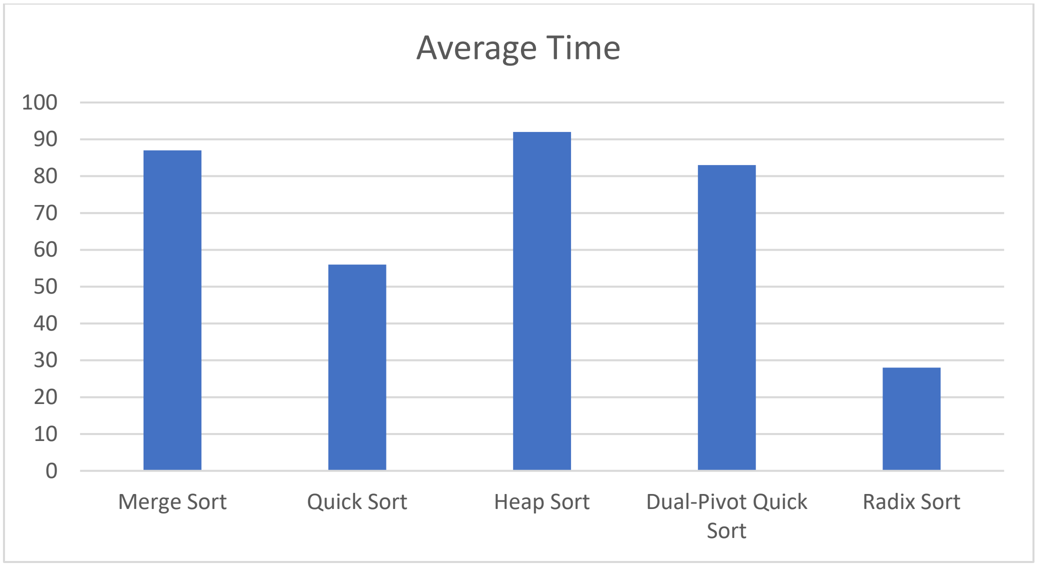
b. Repeat (a) 10 times and compute the average CPU time that takes to sort the keys for the five methods.

Average time can be calculated by dividing the **sum** of each time took to sort the 100000 random generated numbers for 10 times to the **number of times** it had been repeated i.e. **10**. To calculate sum there are 5 different variables of type long for the five different sort methods which adds the previous sum to current time took to sort the numbers.

Table 2.1: Average Time taken to sort 100000 random strings of length 4 for 10 times

Type of Sorting	Merge Sort	Quick Sort	Heap Sort	Dual-Pivot Quick Sort	Radix Sort
Average Time	87	56	92	83	28

Figure 2.1: Average Time taken to sort 100000 random strings of length 4 for 10 times



c. Repeat (a) and (b) with strings of length 6, 8, 10.

To generate the random strings, I didn't give any predefined value as i took the string size value from the user itself. So, the strings of size 6, 8 and 10 are easily generated with the same code instead of writing any other code. By using the same code, the loop executes a greater number of times as per the string size.

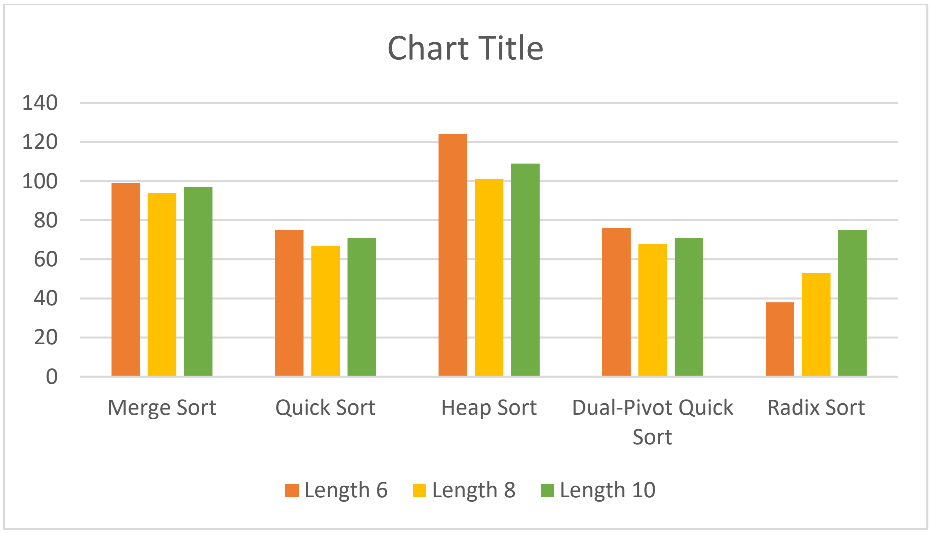
d. Create a table with the results and compare the times with the average-case and worst-case complexities as studied in class.

Below Table is created by measuring the average time took to sort 100000 random generated strings of different sizes like 6, 8, 10 for 10 times.

Table 2.2: Average Time taken to sort 100000 random strings of length 6,8,10 for 10 times

Type of Sorting	Merge Sort	Quick Sort	Heap Sort	Dual-Pivot Quick Sort	Radix Sort
Length 6	99	75	124	76	38
Length 8	94	67	101	68	53
Length 10	97	71	109	71	75

Figure 2.2: Average Time taken to sort 100000 random strings of length 6,8,10 for 10 times



By watching the above blueprint, we can say that if the string size is less Radix Sort puts aside less exertion to sort those 100000 unpredictable keys for various occasions and the time taken to sort those strings continues expanding with the expansion in string size as we probably are aware that for normal case, the time complexity for all the above sorts are same i.e. O(n log n) while coming to most pessimistic scenario Quick Sort sets aside more effort to sort the components, i.e. O(n^2) where as different sorts have a similar time intricacy as in the average-case.

4. Comment on: which sorting method will you use in your applications? in which case? Why?

I generally choose **Quick Sort** among all the sorts in my applications because This popular sorting has an average-case performance of O(*n* log*n*), which contributes to making it a very fast algorithm in practice. But given a worst-case input, its performance degrades to O(n^2). Also, when implemented with the "shortest first" policy, the worst-case space complexity is instead bounded by O(log*n*) whereas **Heapsort** has O(n) time when all elements are the same. Heapify takes O(n) time and then removing elements from the heap is O(1) time for each of the n elements. The run time grows to O(n logn) if all elements must be distinct.

Merge sort is of merging effectively sorted lists into another sorted list. It begins by looking at each two components and swapping them. This is the main that scales well to large lists, whereas its most pessimistic scenario running time is O(n log n). it has extra O(n) space complexity, and includes more copies in direct usage so I don't prefer merge sort in real time applications whereas **Dual-Pivot Quick Sort** is a predefined sorting technique provided in java dual pivot quick sort will take two pivots, one in the left end of the array and the second, in the right end of the array. The left pivot must be less than or equal to the right pivot, so we swap them if necessary although it is a little bit faster than the original single pivot quicksort. But still, the worst case remains same O(n^2) when the array is already sorted in an increasing or decreasing order.

Radix sort sorts numbers by individual digits. Time took to sort n numbers comprising of k digits is O(nk). Radix sort process digits of each number either beginning from the least significant digit (LSD) or beginning from the most significant digit (MSD). The LSD algorithm first sorts the list by the least significant digit. While the LSD radix sort requires the utilization of a stable sort, the MSD radix sort algorithm doesn't. Though Radix Sort takes less time, it goes on increasing with the increase in the length of the string.

So, I prefer to use Quick Sort more than any other sorting techniques provided while coming to the real time applications.

5. Use the edit distance (class Sequences.java) implementation provided in the source code:

a. Generate 1,000 pairs of random words of lengths 10, 20, 50 and 100.

Location of the code: C:\Users\jayad\eclipse-workspace\Assignment2_110026622\src
Reference of the code: Advanced Computing Concepts Lab Source Code
Java files: test.java, Sequences.java
Method Name: sequence();
Variables used:

```
long avg, sum, start1, start2;
int size, pairs=1000, num, i, j,d,string1,string2;
int left=65;                                  // ASCII value for A
int right=90;                                 // ASCII value for Z
String generatedString1, generatedString2; // to store 2 random strings
```

Methods used: editDistance();
Procedure: String size is taken from the user at start then two generated random strings of range from beginning **A** to end **Z** of user given size are produced and put away in **generatedString1** and **generatedString2** then the separation between these two strings is determined and put away in the variable **d** then the time taken to figure the separation is determined by utilizing two variables **start1** and **start2** and the difference between these 2 variables gives the time in **nano seconds** and afterward the Average time taken for all the 1000 pairs is determined and it is imprinted in the proportion of **nano seconds**

b. Compute the edit distance for all words and find the average CPU time for each pair.

Edit distance between the pair of words for all the 1000 pairs can be calculated by using the method **editDistance**(**generatedString1**, **generatedString2**) which was defined in the source code **sequence.java** by calling this method in the **test.java** file the distance between the words.

Table 3.1: Average Time to compute edit distance of 1000 pair of strings for multiple lengths

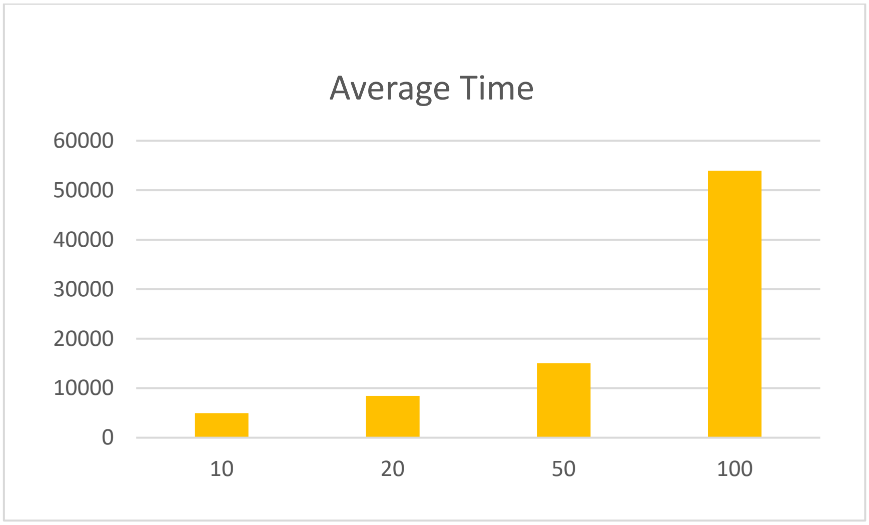
Length of the Strings	10	20	50	100
Average Time	4973	8458	15062	53930

c. **Compare the CPU times obtained for each word length with the running times of the edit distance algorithm.**

The Average time calculated and noted in the above table 3.1 is in **nano seconds** and stored in a variable **avg.** It can be calculated by dividing the **sum** of each time took to find the edit distance between the pair of random generated strings for 1000 pairs to the **number of pairs** it had been calculated i.e. **1000**. To calculate sum there is a variable **sum** of type long which adds the previous sum to current time took to calculate the edit distance.

The Average time to find edit distance is increasing while increasing the size of the string because while increasing the size of the string there will be increase in the changes i.e. steps to substitution, insertion and deletion to transform one string to another string. So, the Average time taken to calculate the edit distance for the string size 100 is far more than the one which has the size 10.

Figure 3.1: Average Time to compute edit distance of 1000 pair of strings for multiple lengths


Average Time