“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.” + Nishesh Kalakheti+ 110017507

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.

**Answer:**

The script file name is “test.java”

**public** **static** **long** generate\_random\_numbers()

This method returns random number.

**public** **static** **long** merge\_sort\_analysis(Long[] merge\_list, **int** j)

This method takes in the merge\_list as one of the parameters which contains 100000 random numbers. The parameter j tracks the iteration number which goes up to 100.

This method sorts the merge\_list array by using the merge sort algorithm and returns the time taken to sort the 100000 random numbers.

**public** **static** **long** heap\_sort\_analysis(Long[] heap\_list, **int** j)

This method takes in the heap\_list as one of the parameters which contains 100000 random numbers. The parameter j tracks the iteration number which goes up to 100.

This method sorts the heap\_list array by using the heap sort algorithm and returns the time taken to sort the 100000 random numbers.

**public** **static** **long** dual\_piv\_sort\_analysis(Long[] dual\_list, **int** j)

This method takes in the dual\_list as one of the parameters which contains 100000 random numbers. The parameter j tracks the iteration number which goes up to 100.

This method sorts the dual\_list array by using the dual pivot quick sort algorithm and returns the time taken to sort the 100000 random numbers.

**public** **static** **long** quick\_sort\_analysis(Long[] quick\_list, **int** j)

This method takes in the quick\_list as one of the parameters which contains 100000 random numbers. The parameter j tracks the iteration number which goes up to 100.

This method sorts the quick\_list array by using the quick sort algorithm and returnsthe time taken to sort the 100000 random numbers.

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

**Answer:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Iteration | Heap Sort(ns) | Merge Sort(ns) | Quick Sort( ns) | Dual pivot quick sort(ns) |
| 1 | 151003400 | 121392300 | 65409800 | 77282200 |
| 2 | 52899300 | 116885800 | 85776200 | 196925200 |
| 3 | 24704600 | 46094700 | 40419500 | 71555400 |
| 4 | 21168900 | 21678300 | 26402200 | 30058000 |
| 5 | 20607000 | 21350400 | 11361500 | 37129100 |
| 6 | 41946700 | 29776800 | 19054000 | 93417400 |
| 7 | 20841100 | 21955300 | 11260700 | 37622900 |
| 8 | 22280100 | 21812100 | 11843900 | 23867400 |
| 9 | 20615000 | 22093100 | 11845800 | 22019200 |
| 10 | 26431700 | 22423400 | 14826700 | 21788000 |
| 11 | 35268900 | 35860900 | 20449100 | 22931000 |
| 12 | 24105600 | 26214200 | 14404700 | 20237100 |
| 13 | 24329900 | 26642400 | 14248100 | 20154900 |
| 14 | 25536800 | 26175700 | 14215900 | 20628200 |
| 15 | 25624200 | 26446800 | 14858900 | 20265400 |
| 16 | 26461300 | 27137800 | 14846400 | 20636500 |
| 17 | 20468100 | 38511100 | 11376900 | 16342500 |
| 18 | 20296400 | 22030700 | 11014100 | 15742200 |
| 19 | 27600400 | 24604700 | 14648600 | 21266500 |
| 20 | 21793700 | 29715300 | 15798400 | 16484400 |
| 21 | 20846500 | 21545400 | 11227600 | 16241700 |
| 22 | 20035300 | 21387400 | 11913500 | 15896500 |
| 23 | 21711300 | 22015300 | 11381500 | 15983200 |
| 24 | 20615200 | 21187900 | 11361800 | 16132300 |
| 25 | 19843300 | 20979200 | 11586500 | 15283600 |
| 26 | 19797700 | 21755500 | 11208400 | 16746700 |
| 27 | 22981800 | 21909100 | 11709100 | 23658600 |
| 28 | 36307400 | 31837200 | 17237900 | 40449100 |
| 29 | 27302600 | 44608700 | 16201000 | 22375600 |
| 30 | 25320700 | 28341600 | 15159200 | 20743000 |
| 31 | 28013600 | 29036300 | 15287700 | 23320200 |
| 32 | 33481300 | 33955700 | 19641100 | 21715600 |
| 33 | 41023200 | 34582500 | 22988400 | 34106000 |
| 34 | 32197000 | 37733400 | 20587900 | 20234700 |
| 35 | 24318200 | 26228000 | 14173300 | 20398200 |
| 36 | 29299300 | 30253700 | 17305600 | 24021700 |
| 37 | 53970500 | 33308700 | 17807300 | 16253000 |
| 38 | 26915600 | 20802100 | 14766700 | 20693500 |
| 39 | 25586100 | 27210500 | 15156500 | 19665700 |
| 40 | 23777200 | 27350200 | 15779600 | 15956800 |
| 41 | 20439900 | 21999600 | 12263900 | 26548800 |
| 42 | 33741200 | 36981800 | 20617300 | 19796200 |
| 43 | 24835100 | 24801900 | 12688200 | 28479000 |
| 44 | 27334000 | 29562500 | 15586200 | 23949500 |
| 45 | 33926600 | 33984400 | 19513100 | 22007600 |
| 46 | 27615900 | 29916700 | 16446500 | 21433100 |
| 47 | 27401900 | 28850900 | 15502300 | 21433600 |
| 48 | 56430900 | 29756400 | 23015000 | 53846200 |
| 49 | 31733400 | 54184300 | 29337300 | 22248100 |
| 50 | 28399300 | 29096300 | 16933700 | 21012900 |
| 51 | 19922400 | 21241700 | 11686100 | 15836400 |
| 52 | 20090400 | 21228300 | 12022400 | 15264900 |
| 53 | 30811300 | 22864200 | 14468900 | 26586800 |
| 54 | 30157400 | 35917100 | 19623300 | 20670400 |
| 55 | 20529400 | 25034400 | 11443600 | 15944500 |
| 56 | 21178500 | 20923300 | 11258100 | 16969800 |
| 57 | 25877600 | 21294300 | 13663800 | 21572300 |
| 58 | 31340900 | 28342900 | 16362900 | 24707200 |
| 59 | 26180600 | 32722700 | 17940000 | 20403800 |
| 60 | 26252200 | 27640100 | 15290100 | 19757200 |
| 61 | 27253100 | 28087100 | 14596600 | 20530400 |
| 62 | 25624000 | 27675300 | 15003400 | 20308800 |
| 63 | 29696200 | 26860200 | 14369600 | 28863500 |
| 64 | 39125500 | 34688100 | 19849100 | 29470100 |
| 65 | 26826700 | 30428000 | 15085700 | 20740400 |
| 66 | 20404700 | 26201500 | 13259400 | 15824200 |
| 67 | 20249000 | 21072000 | 11092400 | 15744200 |
| 68 | 19496200 | 20646300 | 11432200 | 15321900 |
| 69 | 20088100 | 23490200 | 11595800 | 15845400 |
| 70 | 20158200 | 21203100 | 11288500 | 15479800 |
| 71 | 28268900 | 21681400 | 12205800 | 23235100 |
| 72 | 27779400 | 31596400 | 17993600 | 17932500 |
| 73 | 25985200 | 26900100 | 15302100 | 19634700 |
| 74 | 25984300 | 26972300 | 14643600 | 19153300 |
| 75 | 21834300 | 26993900 | 14769500 | 15848400 |
| 76 | 19919400 | 21315500 | 11454800 | 15955600 |
| 77 | 25907400 | 21127000 | 13848900 | 19731700 |
| 78 | 25387900 | 28053000 | 14674000 | 19679300 |
| 79 | 24909800 | 26598400 | 14364400 | 19297700 |
| 80 | 25944600 | 27541100 | 15054200 | 21225800 |
| 81 | 39936300 | 41995500 | 25571900 | 20251600 |
| 82 | 27128400 | 27919100 | 16076700 | 19327100 |
| 83 | 38965400 | 39716700 | 24694600 | 22701600 |
| 84 | 28179500 | 28299600 | 17540100 | 21868000 |
| 85 | 29049600 | 28020100 | 16212900 | 20791200 |
| 86 | 20846400 | 21534600 | 11747700 | 16806600 |
| 87 | 20366900 | 20973400 | 11583700 | 16951600 |
| 88 | 34844900 | 20674900 | 12339200 | 25770700 |
| 89 | 28882100 | 34436300 | 16298600 | 21797500 |
| 90 | 55082400 | 28806800 | 19153600 | 36915000 |
| 91 | 20737100 | 20520400 | 11415700 | 16317300 |
| 92 | 21835300 | 21604700 | 11233400 | 15455100 |
| 93 | 23928900 | 21220300 | 12231600 | 21100500 |
| 94 | 29629900 | 27686500 | 15999200 | 22366500 |
| 95 | 37852600 | 35142600 | 20950400 | 22412700 |
| 96 | 37323900 | 43928700 | 19377000 | 28739500 |
| 97 | 38880400 | 34730800 | 21884600 | 28085500 |
| 98 | 30092100 | 36433100 | 16695700 | 20312200 |
| 99 | 27960200 | 27216500 | 15828300 | 30377000 |
| 100 | 47416400 | 54941900 | 30028700 | 15910000 |

Average time for sorting 10000 random numbers using heap sort: 29113074 ns

Average time for sorting 10000 random numbers using merge sort: 29961074 ns

Average time for sorting 10000 random numbers using quick sort: 17009519 ns

Average time for sorting 10000 random numbers using dual pivot quick sort: 25187775 ns

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

**Answer:**

* As per the results obtained, quick sort has produced the least average CPU time by a significant amount.
* While it is said that dual pivot quick sort algorithm typically performs better than single pivot quick sort, but as per the results obtained that’s not the case. However, for some iterations, dual pivot has the least CPU time out of all the algorithms (if we look at the graph above).
* The other two algorithms heap sort and merge has almost same average CPU time. Even if we look at the graph, the CPU time for sorting by these two algorithms is overlapping for most of the iterations.
* The average case complexity for all the algorithms is O(nlog(n)).

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.

**Answer**

The script file name is “test2.java”

**public** **static** **long** random\_string\_generate()

This method returns random strings.

**public** **static** **long** merge\_sort\_analysis(String[] merge\_list, **int** LEN)

This method takes in the merge\_list as one of the parameters which contains 100000 random strings. The parameter LEN is the length of string.

This method sorts the merge\_list array by using the merge sort algorithm and returns the time taken to sort the 100000 random strings.

**public** **static** **long** heap\_sort\_analysis(Long[] heap\_list, **int** LEN)

This method takes in the heap\_list as one of the parameters which contains 100000 random strings. The parameter LEN is the length of string.

This method sorts the heap\_list array by using the heap sort algorithm and returns the time taken to sort the 100000 random strings.

**public** **static** **long** dual\_piv\_sort\_analysis(Long[] dual\_list, **int** LEN)

This method takes in the dual\_list as one of the parameters which contains 100000 random strings. The parameter LEN is the length of string.

This method sorts the dual\_list array by using the dual pivot quick sort algorithm and returns the time taken to sort the 100000 random strings.

**public** **static** **long** quick\_sort\_analysis(Long[] quick\_list, **int** LEN)

This method takes in the quick\_list as one of the parameters which contains 100000 random strings. The parameter LEN is the length of string.

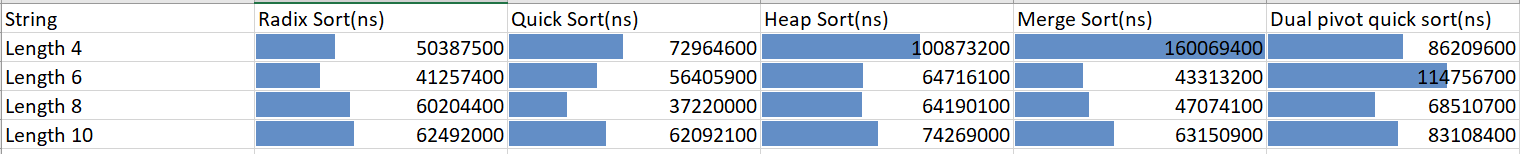
This method sorts the quick\_list array by using the quick sort algorithm and returns the time taken to sort the 100000 random strings.

**public** **static** **long** radix\_sort\_analysis(String[] radix\_list ,**int** LEN)

This method takes in the radix\_list as one of the parameters which contains 100000 random strings. The parameter LEN is the length of string.

This method sorts the radix\_list array by using the radix sort algorithm and returns the time taken to sort the 100000 random strings.

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



c. Repeat (a) and (b) with strings of length 6, 8, 10. d. Create a table with the results and compare the times with the average-case and worst-case complexities as studied in class.

**Answer**

* Results for sorting of the strings of length 6,8 and 10 is present just above this page. There are tables and graphs for the analysis.
* Radix sort has the least average CPU time for sorting random strings of any length. As the length of string increases, the average CPU time also increases.
* Quick sort stands the second best out of all the algorithms. Then comes merge sort, heap sort and dual pivot quick sort.
* The average time complexity for heap sort, quick sort, dual pivot quick sort and merge sort is O(nlog(n)). While the average time complexity for radix sort is O(d(n+b))

where d is the max element length

n is the number of words

b is the base (2 for binary, 10 for decimal values and so on)

* The worse time complexity for quick sort is O(n2). The worse complexity arises when we select a pivot that is either the highest value of the lowest value of the elements. The selection of the best pivot is the median of all the elements.
* The worse case for heap sort, dual pivot quick sort and merge sort is O(nlog(n)). The worse case for radix sort is same as its average case which is O(d(n+b))

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

**Answer:**

* I would use quick sort and dual pivot sort when the size of the input for large data sets. I did an experiment where I increased the value of random elements to 1M and 10M. The average CPU time for dual pivot quick sort stood the best for 1M elements and quick sort stood the best for the 10M elements.
* Also, the operations are in-place. So quick sort and dual pivot quick sort algorithm are memory efficient.
* But one needs to be very careful about selecting the pivot for the quick sort as unwise selection of pivot might lead to the worse case complexity of O(n^2). Also, another point to remember is quick sort is not a stable algorithm. This means the keys with the same values in input might appear in different order in the output.
* Merge sort and heap sort could also be used for huge amount of data. Merge sort is not in-place algorithm, hence requires more memory space to store the sorted elements. Merge sort proved to be faster than heap sort, but it comes with a trade off between memory and time complexity.
* I would use radix sorting depending on the largest element. As the time complexity for radix sort is O(d\*(n+b)) where ‘d’ is the length of the maximum element. If the value of ‘d’ increases, this would lead to worse time complexity.
* Also, one thing worth experimenting is to make the merge sort algorithm from being stable to not stable. In this case, merge sort could have better time complexity than quick sort and dual pivot quick sort.
* Overall, I found quick sort and dual pivot quick sort to be the best algorithm for sorting huge datasets while also being memory efficient. As per the research I made, it makes good utilization of the cache (cache works faster than RAM). However, if I need to use non-stable sorting algorithm, I would go with merge sorting.

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.

**Answer:**

The script file name is “test3.java”

**public** **static** String random\_string\_generate(**int** length)

This method returns random string with length of the string provided as a parameter.

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

**Answer:**

**public** **static** **void** main(String[] args)

This main method calculates the edit distance between two strings. Also it prints average CPU time to calculate the distance between two strings for lengths 10,20,50 and 100.

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

**Answer:**

|  |  |  |
| --- | --- | --- |
| String Length | Average Distance | Average Time (ns) |
| 10 | 8 | 2103 |
| 20 | 16 | 3873 |
| 50 | 39 | 18624 |
| 100 | 77 | 85047 |

The below graph is average distance vs average time(ns).