**CSC140 Algorithms and Paradigms**

**Spring 2018**

**Assignment 3 Report**

Student Name: Grant De La Campa Student ID: 215483906

**Part1: Algorithm Comparison**

Report the results of your sorting experiments in the tables below. All times should be in milliseconds.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | **10000** | **100000** | **1000000** |
| **Insertion Sort** | Sorted | 0.358324ms | 3.087479ms | 5.127171ms |
| Nearly Sorted | 0.47764ms | 0.889581ms | 3.833956ms |
| Reversely Sorted | 26.441613ms | 11807.851342ms | Freezes |
| Random | 17.807471ms | 2209.457344ms |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | **10000** | **100000** | **1000000** |
| **MergeSort** | Sorted | 3.160352ms | 17.342291ms | 97.171579ms |
| Nearly Sorted | 2.336093ms | 21.881191ms | 98.236358ms |
| Reversely Sorted | 3.082193ms | 12.43978ms | 86.971609ms |
| Random | 3.192825ms | 14.487779ms | 199.319297ms |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | **10000** | **100000** | **1000000** |
| **QuickSort** | Sorted | 8.646603ms | 19.585876ms | 82.029452ms |
| Nearly Sorted | 1.044766ms | 6.990156ms | 62.073546ms |
| Reversely Sorted | 0.721557ms | 7.430793ms | 71.696573ms |
| Random | 1.897344ms | 12.500949ms | 117.227544ms |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | **10000** | **100000** | **1000000** |
| **LibSort** | Sorted | 22.144365ms | 2.902465ms | 5.942746ms |
| Nearly Sorted | 1.058737ms | 14.865738ms | 69.522462ms |
| Reversely Sorted | 0.278277ms | 0.487457ms | 1.430276ms |
| Random | 0.839363ms | 45.685402ms | 97.748901ms |

The first basic implementation of quicksort causes a stack overflow error because you are not reducing the size of the problem on recursion thus causing the stack to grow past javas limits and overflow.

**Part2: Quicksort Focused Study**

Report the results of your focused study of Quicksort in the table below.

|  |  |  |
| --- | --- | --- |
|  | **Recursion Depth** | **Execution Time**  **(ms)** |
| **Simple Random** |  | 125.763138ms |
| **Median of Three** |  |  |
| **with InsertionSort** |  | 120.237619ms |
| **Your Best Result**  **(Extra Work)** |  |  |

**Discussion**

Discuss the following points:

1. For each data type (sorted, nearly sorted, reversely sorted, random), which algorithm has the best performance? Explain why?

Sorted: Insertion Sort has the best performance because in this case because its only comparing the elements and not shifting them.

Nearly Sorted: Insertion Sort in this case has the best performance again because the data is nearly sorted it requires a smaller number of shifts.

Reversely Sorted: LibSort is the fastest in this case because it uses a more advanced dual pivot quicksort for primitive types. Comparing the run time of quicksort to the other algorithms for reversely sorted lists it’s the fastest aside from LibSort.

Random: LibSort is the fastest again because for random numbers quicksort is the second fastest and so the LibSorts dual pivot quicksort algorithm will come out as faster than Quicksort.

1. How does the performance of each algorithm change when we change the input type? Explain why?

Insertion Sort: When that data is sorted, or nearly sorted Insertion sort performs very well. However, when the data is reversely sorted insertion sorts performance exponentially decays with increased input size. The reason behind this is that when the list is reversely sorted insert sort must shift each element to the opposite position of the array.

MergeSort: For small inputs of any type the merge sort performs consistently but with increased input size we see a decrease in performance for randomly sorted input. When the input is small the time difference for inputs are negligible in difference. However when the input is large and reversely sorted the second loop must execute the maximum number of times.

QuickSort: For small input quicksort performs best on anything not already sorted however as the input size increases for randomly sorted inputs quicksort’s performance decreases in comparison to other input types. This is due to the quicksort having to swap elements around in a far more random and erratic way when sorting such large inputs.

LibSort: for small input libSort performs poorly on sorted data but for larger inputs libSort begins to perform poorly for random input. Libsorts performance times are somewhat reflective of quicksort due to javas implementation of Libsorts. Since it uses a form of quicksort it suffers from the same issues with large random inputs where the swapping if occurring a lot more than when the input is sorted.

1. How does the performance of each algorithm change when we change the input size? Explain why?

Insertion Sort: For smaller inputs insertion sort is fast in all cases. When the input begins to increase in size we see that insertion sort becomes very slow for random and reversely sorted input due to its O(n^2) time complexity in the worst case.

MergeSort: This algorithm performs fairly consistently for smaller sized inputs but as it starts to approach input size of one million it begins to slow slightly for random input while still maintain a tight range of values. Merge sorts consistency is due to its O(nlogn) complexity.

QuickSort: Quick sorts performance for small input is fast excluding when the input is sorted because this is the o(n^2) case. For larger input quicksort is faster with a time complexity of o(nlogn)

LibSort: In this case dual pivot quicksort has a similar performance as quicksort where for small input it is quick If it isn’t sorted. For large inputs it is quick again with a O(nlogn) complexity.

1. Do you have any other observations or insights? Out of most of the algorithms insertion sort seems to be the least versatile and quicksort seems to be the most versatile when it comes to various data types.