Algorithms Project 2 – Analyzing Sorting Algorithms

**Abstract:**

This report details the way in which we ran and analyzed four different sorts: heap sort, insertion sort, merge sort, and quick sort. The results indicated that for a large array, insertion and merge sort performed roughly equally well, followed by heap sort, then quick sort which performed significantly worse than the rest. Insertion sort performed exceptionally well in the best case, and quick sort performed about just as poorly in the worst case as in the average or best case scenario. However, when dealing with small arrays, quick sort performed the best, though it was by an insignificant margin.

**Introduction:**

This program analyzes the above-mentioned sorts by randomly generating arrays of incrementing sizes (sizes 10, 100, and 1000) and passing them through the sorts. As the best case for insertion sort was a different run time than the worst or average cases, we used an already sorted array to trigger the best case for analysis. Similarly, the worst case for quick sort was different than the best or average cases, so we passed an already sorted array to this one as well to guarantee the highest value would be chosen as the pivot and trigger a worst case run time.

**Group Contributions**

* Taylor Boling: Gathering of sorting algorithms, file naming/writing, mainline code
* Justin Lee: Best/Worst case sorting, Time stamping, debugging, implementing counter
* Luke Simpson: Writing to file, Outline/Report, debugging

**Background:**

Insertion Sort:

insertionSort()

for i = 0 to arraySize - 1

current = A[i]

j = i – 1

while j >=0 && A[j] > current

A[j+1] = A[j]

j = j -1

A[j + 1] = current

Heap Sort:































Merge Sort:











True































True

Quick Sort:

quickSort(left, right)

if right - left

return;

q = partition(A, p, r);

quickSort(A, p, q - 1);

quickSort(A, q + 1, r);

return;

**Theoretical Analysis:**

* Insertion Sort – Insertion sort runs O(n) as its best case when the data is already sorted. Otherwise, for unsorted data, the algorithm runs at a time complexity of O(n2). The worst case occurs when the data is sorted in reverse order, but it still runs at a complexity of O(n2). This is because the algorithm builds a sequence of sorted values from the beginning of the list, and if it runs in to an element that does belong at the end of the list, will compare the value to the sorted sequence until it finds a place for it. This means that an already sorted list will have virtually no comparisons to be made, but a reverse sorted one will have to compare every element.
* Merge sort – Merge sort runs O(n log n) for every case: best, worst, and average. Regardless of how the list is ordered, the algorithm will check it recursively, causing it to be O(n log n).
* Heap sort – Heap sort also runs O(n log n) for every case. This is because no matter how the input data is arranged, the algorithm builds the “heap” as a sort of binary tree and therefore relies on division.
* Quick sort – Quick sort runs at O(n log n) for its best and average case scenarios, and O(n2) for it’s worst case. For the best and average case, the recursive nature of quick sort causes the time complexity to be O(n log n). However, the worst case occurs when the greatest or smallest element is selected as the pivot, meaning all elements being compared are either smaller or greater than the pivot and causing the time complexity to become O(n2).

**Design of experiments:**

For input data sets, we populated array sizes of 10, 100, and 1000 with random elements and passed every size of array into each sort. To trigger the best case of insertion sort and worst case of quick sort, we used arrays that were already sorted. For performance indicators, we measured time in microseconds, as seconds or milliseconds were too large to capture the time it took to run small array sizes through the sorts. We also used a counter variable that would increment by one each time a comparison was performed in each sort.

**Pseudocode:**

* Call function writeArray() that writes an array of size n to a file. Also creates file object for a given sort of a given size.
* Call function populateArray() that fills the array with random numbers that are unsorted.
* Call merge sort function merge(), merges 2 subarrays of array (arr).
* Sort the left and right index in function mergeSort()
* Pass mergeSort() into function callMergeSort()
  + Start time on mergeSort()
  + End time on mergeSort() and calculate the difference in time\_elapsed
  + Write data to file via writeArray()
* Call function insertionSort() for insertion sort algorithm
  + Pass array (arr) into function
  + Start time on high resolution clock
  + Do insertion sort algorithm (as explained in background section) using a comparison counter and incrementing it as it passes through the algorithm.
  + Get end time on high resolution clock and store time in time\_elapsed
  + Write data to file via writeArray()
* Call function to get the best case time for insertion sort (insertionSortBest())
  + Pass an already sorted array with a size of either 10, 100, or 1000
  + Print out data for the best case along with time.
* Call function heapSort() for heapsort algorithm
  + Start time on high resolution clock
  + Pass array into function
  + Perform Heap Sort with the heapSort() and subfunction heapify()
  + Stop clock and get the difference of stop and start time
  + Write data to file via writeArray()
* Call function callQuickSort()
  + Start time on high resolution clock
  + Pass array into function
  + Perform Quick Sort with quickSort() and subfunctions swap() and partition
  + Stop clock and get the difference of stop and start time
  + Write data to file via writeArray()
* Call function callQuickSortWorst()
  + Start time on high resolution clock
  + Pass a pre-determined “worst case” array into function
  + Perform Quick Sort with quickSortWorstCase() and subfunctions swap() and partitionWorstCase()
  + Stop clock and get the difference of stop and start time
  + Write data to file via writeArray()
* Call function executeSorts()
  + Fill an empty array with populateArray()
  + Perform each sort, resetting the array to its original value after execution:
    - insertionSort()
    - insertionSortBest()
    - callMergeSort()
    - heapSort()
    - callQuickSort()
    - callQuickSortWorst()
* Call driver function main()
  + Generate 3 arrays of 10^k size (10, 100, and 1000 respectively)
  + Upon generation of each array, perform executeSorts() and then delete generated array, so as to avoid memory leaks
  + End program.

**Implementation:**

For this project, we worked using Microsoft Visual Studio 2017 in the language C++. The program terminates after passing each array size through each sort one time and recording the results in output text files. Input data was randomly generated using the “rand()” function modulus 1000, giving us randomly generated values from 1 to 1000. Output data was collected through a function designed to open and write the analysis data to a text file each time a sort was run.

**Analyzing output data:**

Array Size 10:

|  |  |  |
| --- | --- | --- |
| **Sort** | **Time (microseconds)** | **Comparisons** |
| Insertion | 9 | 15 |
| Insertion (Best case) | 0 | 4 |
| Merge | 11 | 19 |
| Heap | 4 | 10 |
| Quick | 2 | 8 |
| Quick (Worst case) | 2 | 8 |

Array Size 100:

|  |  |  |
| --- | --- | --- |
| **Sort** | **Time (microseconds)** | **Comparisons** |
| Insertion | 7 | 2458 |
| Insertion (Best case) | 0 | 39 |
| Merge | 82 | 356 |
| Heap | 76 | 100 |
| Quick | 117 | 94 |
| Quick (Worst case) | 129 | 94 |

Array Size 1000:

|  |  |  |
| --- | --- | --- |
| **Sort** | **Time (microseconds)** | **Comparisons** |
| Insertion | 604 | 251,589 |
| Insertion (Best case) | 4 | 453 |
| Merge | 807 | 5044 |
| Heap | 1143 | 1000 |
| Quick | 7392 | 734 |
| Quick (Worst case) | 7454 | 734 |

**Summary:**

The results indicate that for small array sizes, in our case 10, the choice of sort does not have a noticeable effect on the time spent sorting, so an easily written or available sort is perfectly acceptable for applications that will never be dealing with large datasets. However, as datasets get larger, it becomes apparent that quick sort is the most inefficient of all the sorts we analyzed, taking much longer to run even in its best case. At an array size of 1000, insertion sort ran slightly faster than merge sort, which ran slightly but noticeably faster than heap sort, indicating that insertion or merge sort are the best choices for large datasets. However, when considering comparisons made, insertion sort made a remarkably high number of them, indicating that it requires more space and the speed may drop off significantly at particularly large datasets. Despite this, at it’s best case, insertion sort ran exceptionally fast even when dealing with a large dataset, but as this relies on the data being already sorted, it is not a practical choice for real-world applications where data must be ordered.