CSCI 2720

Data Structures

Project 4

Sorting Algorithms

Spring 2017

**Team Members**

1. Chris Callahan 811264461 26490
2. Johnathan Kulovitz 810641087 26490
3. **Introduction**

In this project, I will be implementing 6 different sorting algorithms and testing them with varying size/ordered arrays of integers. Before testing, I will discuss the theoretical analysis of each sorting algorithm on varying input values. The test driver program is called sortingDr. It will take in user input for a file containing a list of integers in some type of order (inorder, reverse, and random), as well as the desired size of the array and type of sorting algorithm (Selection, Bubble, Insertion, Merge, Quick or Heap sort). The purpose of the tests are to gather information about execution times and comparison counts and make some conclusions about how each algorithm is affected by the size and order of the data.

1. **Theoretical Analysis of Sorting algorithm**

Bubble Sort

Bubble sort is the simplest sorting algorithm. It works by starting at the first index and crawling towards the last index. It compares the element in front of the current location and makes a swap if the first number is larger. It then moves to the next index. When it gets to the last index, it starts over again. This process continues until it does a full pass through the array with no swaps.

The best case for this process is that the list is already in order. On the first pass, it will make zero swaps and end the sort giving it a complexity of O(n). The runtime for this process will be quick. The worst-case scenario is that the list is in the complete opposite order. Instead of doing only one pass in the ordered list, the reverse list will have to do n passes. This gives the worst case scenario a complexity of O(n2). The runtime will be absolutely terrible.

We believe the average runtime for this algorithm using the randomly generated test input will be close to the reverse ordered test.

Insertion Sort

Insertion sort is analogous to a rolling mist, slowly enveloping everything within it. With each iteration, the sort adds one more index of the array to be sorted. The first run iteration will sort the first two indexes. The next iteration will sort the third index in regards to the previous two indexes. This increases until it has iterated n times and the entire array is sorted.

If the array is already sorted, the algorithm will only have to make n comparisons as it moves through the array. Because of this, the best case complexity must be O(n). The worst case, similarly to bubble sort and other simple sorting algorithms, is when the array is reversely ordered. The maximum number of comparisons on each iteration must take place, as it has to shift each element ‘i’ (how many iterations into the sort it is) times.

Selection Sort

Selection sort works similarly to insertion sort in that it splits the array into a sorted section and a non-sorted section. Every loop increases the size of the sorted section by one. Instead of sorting the number downward in the sorted section, it looks for the minimum value in the unsorted section. That minimum is then added to the sorted section, where it is automatically in the correct spot.

The analysis of Selection sort easy because it is the same regardless of how the data is presented. The algorithm searches every single point in the unsorted section of the array each time. This means it searches n, then n-1, then n-2 etc… times. This can be simplified to:

Comparisons

This falls under the O(n2) complexity class. Selection sort is one of the most simple algorithms but also struggles with large data sets. The execution time will be very slow; we believe it will be similar to bubble sort.

Heap Sort

Heap sort works by creating a heap out of the array. At this point, either the smallest or largest element will be at the top of the heap. Whether or not the element is the largest or smallest depends on if you use a down-sift function or up-sift function to fill the heap. It then removes the largest element of the heap, adds it to the array, and replaces it with the smallest element. The heap is rebalanced and the process continues until the heap is size 0 and only the smallest element remains unsorted. That element is added to the smallest index of the array and the array is sorted.

Filling the heap and refilling the array has a complexity of 2 O(n). We also expect that the sort will act in O(log n) time. When adding these two complexities together, we get a final complexity of O(n log n). Making a heap out of the array will take the same time regardless of order so we think the number of comparisons will be similar for all test inputs.

Merge Sort

Divide and Conquer is a common problem solving technique across many fields of study. This also applies to merge sort. This method breaks the array into equal parts (as n allows). It then breaks the arrays further the same way. This continues until they are individual elements. At this point, the broken arrays merge in the same manor they were broken down but in reverse. When they are merged back together, they are sorted. By the time the pieces are put back together, they array is fully sorted.

The number of merges that occur can be found using the recursive function below.

This recursive function enables merge sort to be a very stable and efficient sorting algorithm because there is a relatively similar amount of merges regardless of input size. We think this formula will run in O(n log n) time because the merge functions alone take up O(n) time. By breaking down the array and dividing the work, the actual sorting part of the algorithm cannot take exponential time. We think this method will have the least amount of comparisons as the input size reaches the larger sizes.

Quick Sort

This method is a highly efficient. Firstly is shuffles the array. Then it picks one element and puts it in its correct final spot. It puts no elements that are smaller than it in higher indexes and no bigger elements in smaller indexes. This recursively continues on the sub-arrays (The parts of the arrays above and below the partition) until the list is sorted.

The array must be shuffled because quick sort works on an average case basis. The worst case occurs when the partition is made at either the smallest or largest element, making two sub-arrays sizes 0 and n-1. This means the recursive sort will be called n times. Given that partition function has a complexity of O(n) and that it would be called n times, this gives quick sort a worst case complexity of O(n2). It is important to note that this is extremely rare and has a probability of 2/n. The best case is that the partition makes two equal sized sub-arrays. On an average basis, quick sort is able to make two relatively close sized arrays. This means the best case and average case must share the same complexity. We believe this to be O( n log n).

We believe quick sort will have a faster run time and use less memory than merge sort, but will still have more comparisons.

1. **Experimental Setup**

Testing Machine Specifications:

Lenovo Z70

Core i7 Processor

8 gigabytes of RAM

C++ GCC 5.2.0

Red Hat UNIX

Test inputs were generated by entering 200,000 integers (INT) into three separate “.dat” files. Each file had a unique purpose. One file was randomly generated with numbers. Another was filled with ordered numbers up to 199,999. The final file was the reverse order of the second test input. These files were each made because they represent the most common practices of storing one-dimensional sets of data. Because they are already in the correct sequence, we can easily fill the testing arrays. We were able to easily see how the different algorithms were affected by input size as the sheer quantity of numbers in the testing files allowed us to look at large sets of numbers. Because these files contained so many numbers, we could test varying sizes of inputs to look at how the algorithms are affected by the changes. The input sizes we chose were 10, 100, 1000, 10000, 20000, 100,000, 200,000.

Testing times were taken using a timer function that returned accurate timestamps of when the function was called. This was called before the sorting algorithm was started and directly after. The first timestamp was subtracted from the second one. That difference was the execution time of the test. We did this three times for each input size and input file to find an average.

There was only one case of extra memory, which was from the Merge Sort algorithm. This sorting method calls a function “merge()” to literally merge each “broken down” array into the final sorted size n array. The values held within the array “data[]” were stored temporarily in “c[]” where they were correctly sorted until they were merged back into the main array.

1. **Experimental Results**

In this section, for each algorithm, you should compare its theoretical performance to the actual performance (Execution time). Then you will compare them to one another in terms of actual performance and the number of comparisons done.

**4.1 O(n2) Sorting Algorithms**

SELECTION:

Compared to the theoretical performance, the data we collected for the execution time of sorting less than 1000 elements seemed faster than expected. We expected the execution time to suffer for large sets of data, which it did. As you can see in the plot below, the execution time suffered greatly when sorting sets of 100,000 or more elements. During testing, the long execution was very noticeable for sets greater than 100,000. As expected, the order of the data did not affect the execution time greatly.

INSERTION:

The theoretical performance of insertion sort predicted it’s best case to be a list already in order, and the results prove that this theory is correct. The worst case for insertion sort was a list in reverse order, and again the results show that this is correct. Our results show that insertion sort proves to be a slightly better algorithm than selection sort and bubble sort for large sets of data, but not by much (as shown by the plot below).

BUBBLE:

Our findings show that bubble sort is a great algorithm, but only in certain situations. Theoretically, bubble sort is a O(n^2) algorithm, and with our added Boolean value O(n) in the best case (list is in order). As you can see in the plot, on the average case bubble sort climbs to higher execution time faster than the rest of the O(n^2) algorithms, but in a scenario where a sorted list is common, this algorithm can be useful.

COMPARISON ANALYSIS:

When comparing the average case for each of the O(n^2) algorithms, Insertion sort has far less comparisons than it’s counterparts. This can be seen in the plot below which shows the number of comparisons for each algorithm. For data that is already in order, bubble sort is the best performing algorithm due to its added Boolean value to check if a list is already in order. For reverse order lists, bubble sort and selection sort have similar numbers of comparisons. Selection sort has the most consistant numbers, because its algorithm must go through the same amount of comparisons for each type of data.

**4.2 O (n log n) Sorting Algorithms**

Compare O (n log n) sorting algorithms (Heap, merge and quick) in terms of execution time and number of comparisons. Show a table of your results and 2 plots similar to previous subsection 4.1.

MERGE:

Merge sort is an nlogn algorithm, so its execution time was expected to be much better than the O(n^2) algorithms. A key finding from our research was that its execution time was slightly better for inorder and reverse order cases.

QUICK:

Quick sort was by far the fastest algorithm for all types of data in the O(nlogn) category. This was expected due to our research on the theoretical analysis. Its best performance was with inorder data, with a slight increase (but not by much) on random data. The reverse order data had similar execution times to the inorder data.

HEAP:

Heapsort proved to be the slowest of the nlogn algorithms as we expected. At about 20,000 data elements, it jumped to a higher execution time as you can see by the plot below. An interesting observation was that heap sort did slightly better in its reverse order cases.

COMPARISON ANALYSIS:

The O(nlogn) algorithms outdid the O(n^2) algorithms by a long shot, as you can see by the plot below. Merge sort had the most efficient comparison count for very large sets of data as we expected, with quick sort coming next and heap sort last. Quick sort, merge sort, and heap sort all performed better in their reverse order cases, with merge sort again being the winner for lease comparisons. The data for each only started to vary widely when the data sets reached 10,000 elements.

1. **Concluding Remarks**

Comparing the two different sets of algorithms (O(n^2) and O(nlogn) proved to be very interesting. Obviously, the number of comparisons as well as execution time was more efficient for nlogn algorithms, but that does not mean n^2 algorithms can be counted out. For small data sets, these algorithms work just fine. Especially an example where an efficient sorting algorithm and an algorithm with an efficient specific case are used in conjunction. One example would be a phone book or other type of database where after entering a long list of data, data is entered one at a time. After the initial data is entered, bubble sort(which is best in an already sorted state) could be used to add data one at a time.

We conclude that the best algorithm is merge sort. Although, with varying types of data, this could change. Overall, merge sort had the fastest and most efficient number of comparisons in all three types of data.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | **Number of elements** | | |  |  |  |  |  |
| **Sort Type** | **data type** | **10** | **100** | **500** | **1000** | **10000** | **20000** | **100000** | **200000** |
| Insertion | Inorder | 1 | 1 | 3.33 | 8.33 | 82.67 | 162.33 | 883 | 1711 |
| Insertion | Random | 1.33 | 44.33 | 384.67 | 1533 | 149577.5 | 600116.3 | 15104104 | 60569482 |
| Insertion | Reverse | 2.33 | 33.33 | 767.6 | 3006.66 | 297651.33 | 1250151 | 30816615 | 122067799 |
| Selection | Inorder | 1 | 45.33 | 595.67 | 2135.33 | 231747.33 | 942555 | 2346572 | 93881474 |
| Selection | Random | 1.33 | 54.33 | 626.33 | 2434.33 | 234747.3 | 942454.3 | 2346622 | 93891473 |
| Selection | Reverse | 1 | 55 | 629.33 | 2456.33 | 234848 | 943667.67 | 2356688 | 93892491 |
| Merge | Inorder | 4.3 | 14.33 | 76 | 114.33 | 2282 | 3014.33 | 16847.3 | 35420 |
| Merge | Random | 8.33 | 18 | 86.33 | 181.33 | 2479.67 | 5025.67 | 28205.33 | 59058 |
| Merge | Reverse | 7.3 | 19.5 | 88.67 | 123.33 | 2004 | 2986.67 | 16753.33 | 36323.6 |
| Quick | Inorder | 1 | 4.67 | 20 | 42.33 | 533.33 | 1149.34 | 6715 | 14182 |
| Quick | Random | 3 | 9.67 | 59 | 124 | 1628.33 | 3392.67 | 19685.67 | 39770 |
| Quick | Reverse | 2 | 5.3 | 22.2 | 45.33 | 545 | 1108.6 | 7064.3 | 15102 |
| Bubble | Inorder | 1 | 2.33 | 3.33 | 5.3 | 47 | 94.67 | 541 | 955 |
| Bubble | Random | 2.3 | 58 | 1343.67 | 5400 | 544976 | 2162716 | 54023240 | 216508014 |
| Bubble | Reverse | 1 | 64.3 | 1315.6 | 5483.33 | 545265.33 | 2162685 | 53428965 | 216453254 |
| Heap | Inorder | 1 | 19 | 125.33 | 289.33 | 4002.6 | 7865.6 | 53025.33 | 116258 |
| Heap | Random | 2 | 20.67 | 138.3 | 299.67 | 4198.33 | 8932.67 | 53457 | 116357.67 |
| Heap | Reverse | 1 | 18.3 | 122 | 260 | 3996.33 | 7859.67 | 48763.3 | 105266 |

Table 1: Running time vs. number of elements

**References:**

<http://cforbeginners.com/insertionsort.html>

<https://mathbits.com/MathBits/CompSci/Arrays/Selection.htm>

<http://www.cplusplus.com/forum/general/127295/>

<http://quiz.geeksforgeeks.org/merge-sort/>

<http://quiz.geeksforgeeks.org/heap-sort/>

<http://quiz.geeksforgeeks.org/quick-sort/>