**Sorting Algorithms Analysis and Comparison**

**Purpose**

The aim of this experiment is to analyze and compare three different sorting algorithms based on their efficiency (or complexity). The complexity is measured using the runtime of each of the algorithms on the same unsorted array. The number of swaps and iterations made is also computed to give an idea of the total number of operations each algorithm makes before it can sort the array.

**Procedure**

The three different sorting algorithms that were compared are Insertion sort, Quicksort, and Counting sort. Insertion sort was chosen because it is an improved version of the naïve Bubble sort and it would be interesting to see how it compares against the mighty Quicksort that employs the divide-and-conquer paradigm and uses recursion. Counting sort is a non-comparison based algorithm. Hence it always runs in linear time.

In Insertion sort, we loop over the entire array times. When we loop over the array the th time, the first elements are sorted. And, at the end of the th iteration, the first elements of the array are sorted. Thus, when we finish looping over the array times, all the elements are sorted in the array. Clearly, this outperforms the naïve Bubble sort but is still not as powerful as other algorithms.

Quicksort is a divide-and-conquer algorithm that partitions the array into two smaller sub-arrays. The middle element is chosen as a pivot and after comparing it to other elements, a swap is made based on the result of the comparisons. Then Quicksort is recursively called on the two sub-arrays until all the sub-arrays are sorted.

Counting sort is a non-comparison based algorithm that works only when it is known that the input array consists only of positive numbers with an upper bound. First a histogram is made of the keys that occur in the input array. In the second loop, the prefix-sum is computed. Finally, the correct position is assigned to each of the input key into a new array. We can then either copy the values from the temporary array into the original array or simple use the new sorted array.

Now that we’ve seen a brief overview of the algorithms we’ve implemented, note that Insertion sort is the slowest of the three. Quicksort uses an algorithmic approach called divide-and-conquer. Counting does not do any comparison and thus loops over the array at least 3 times. Since it takes this method, a pure in-place implementation of Counting sort is difficult to achieve. Hence Quicksort has an edge over Counting sort in terms of memory requirement; however, they must perform at almost the same levels.

To evaluate and compare the performance of these three algorithms, we will need a measure of the performance. For this metric, I will be primarily using the runtime of the algorithms. Along with this, counting the number of operations made (iterations and swaps) will also give an interesting perspective to the problem. Since, Counting sort has no direct swapping of two elements, I will count a copying operation from the temporary array to the original array as a swap operation.

**Data**

The runtime of each of the three algorithms will be tested against arrays of the following sizes:

The arrays were generated using the random\_device from <random> library. It is a uniformly-distributed integer random number generator that produces non-deterministic random numbers. *Mersenne Twister* pseudo-random number generator was used in conjuncture with this. The rand() from <cstdlib> library creates a bias towards the lower end of the integers when generating numbers within a confined range. This limitation was overcome by using uniform\_int\_distribution<int> that guarantees unbiased numbers given a range.

Once an array was generated using this process, it was copied into two separate arrays. Thus, we have three identical, randomly generated arrays to be tested against three different algorithms.

**Result**

Table 1 Number of Swaps

|  |  |  |  |
| --- | --- | --- | --- |
| **Array Size** | **Insertion Sort** | **Quicksort** | **Counting sort** |
| 10 | 16 | 9 | 10 |
| 100 | 2498 | 181 | 100 |
| 500 | 60493 | 1191 | 500 |
| 1000 | 249053 | 2560 | 1000 |
| 5000 | 6309548 | 15376 | 5000 |
| 10000 | 25246025 | 33210 | 10000 |
| 50000 | 624496303 | 195973 | 50000 |
| 100000 | 2497811749 | 415558 | 100000 |
| 500000 | 62375174049 | 2365823 | 500000 |
| 1000000 | - | 4987030 | 1000000 |
| 5000000 | - | 28804556 | 5000000 |
| 10000000 | - | 61548675 | 10000000 |
| 50000000 | - | 360546020 | 50000000 |

Table 2 Number of Iterations

|  |  |  |  |
| --- | --- | --- | --- |
| **Array Size** | **Insertion Sort** | **Quicksort** | **Counting sort** |
| 10 | 26 | 30 | 100000030 |
| 100 | 2598 | 685 | 100000300 |
| 500 | 60993 | 4515 | 100001500 |
| 1000 | 250053 | 10422 | 100003000 |
| 5000 | 6314548 | 72839 | 100015000 |
| 10000 | 25256025 | 153932 | 100030000 |
| 50000 | 624546303 | 819547 | 100150000 |
| 100000 | 2497911749 | 1744521 | 100300000 |
| 500000 | 62375674049 | 10086047 | 101500000 |
| 1000000 | - | 21577722 | 103000000 |
| 5000000 | - | 115492919 | 115000000 |
| 10000000 | - | 252160734 | 130000000 |
| 50000000 | - | 1256715538 | 250000000 |

Table 3 Runtime in milliseconds

|  |  |  |  |
| --- | --- | --- | --- |
| **Array Size** | **Insertion Sort** | **Quicksort** | **Counting sort** |
| 10 | 0 | 0 | 0 |
| 100 | 0 | 0 | 0 |
| 500 | 0 | 0 | 0 |
| 1000 | 0 | 0 | 0 |
| 5000 | 0 | 0 | 0 |
| 10000 | 0 | 0 | 0 |
| 50000 | 3000 | 0 | 0 |
| 100000 | 14000 | 0 | 0 |
| 500000 | 364000 | 0 | 0 |
| 1000000 | - | 0 | 0 |
| 5000000 | - | 0 | 1000 |
| 10000000 | - | 1000 | 2000 |
| 50000000 | - | 10000 | 9000 |

As expected from the discussion, Insertion sort performed very poorly for arrays larger than 100,000 (around 13 seconds). At 500,000, it took more than 6 minutes to sort the array. And at 1,000,000, it could not complete the sorting process even after 30 minutes.

Counting sort and Quicksort were much better. However, Counting sort has an edge over quick sort when the input range is not much bigger than the input array size. At the size of 5 million, Counting sort took 1 second to complete, while Quicksort took 0 milliseconds to complete. At the size of 50 million, Counting sort took 9 seconds while Quicksort completed the operation in 10 seconds.

Comparing the number of operations made by each algorithm also leads to a similar conclusion. From the data, we observe that the number of operations made by Counting sort is great when the input array size is much smaller than the input range. However, for large arrays, Counting sort takes fewer number of operations to sort the array than Quicksort. We can therefore conclude and say that Counting sort is the best out of the three algorithms when we are dealing with large databases. The one drawback in Counting sort is that the keys should be confined to a range. Quicksort performs well and can be used when the database is very small or we do not know that the array keys are integers confined to a range. And Insertion Sort is the poorest of the three algorithms and took more than 6 minutes to sort a 100,000 array. Therefore, on large scale datasets, Quicksort or Counting sort is preferred, with Counting sort performing better having fewer operations made.

**Sources**

* Counting Sort pseudocode - <http://www.albany.edu/~csi503/pdfs/handout_9.1.pdf>

Retrieved on March 18 2017, 2:20 PM (GMT-0600)

* Quicksort and Insertion Sort - Recitation 9 starter code