

Run Time Analysis of Insertion Sort and Counting Sort

In this report I will be analysing the run time of two different sorting algorithms. Insertion sort and counting sort. Insertion sort is expected to take different amounts of time depending on the data being sorted.

The best case for insertion is when the data is already pre-sorted, this means that the algorithm only looks at each element once to ascertain that it is in the correct order and therefore we would expect to find this to be of the order $O(n)$.

The worst case scenario is where the data is in reverse order. The algorithm will have to loop through the entire data set to find the smallest number and then the whole way through again to find the next before the smallest can then be inserted in the correct place, this means that we would expect this to be of order $O(n^2)$.

In the average case where numbers are in random order it can also be very time consuming looping through the data set to find the next number and so therefore we also expect an order of $O(n^2)$.

The insertion sort algorithm is known to be very quick for small data sets but increases massively when the data sample increases.

Counting sort is known to be a fast and efficient method of sorting so we expect this to be of order $O(n)$ in all cases.

When designing this experiment I used the provided code for reading and writing files in java. I modified the writing file to output an array of random numbers, as well as a sorted and unsorted arrays that can be used to test each scenario. I then further modified this code to produce multiple files of varying sizes to be used for the experiment.

For the sorting algorithm I used the provided code to read a file and added functions that I obtained online for the actual algorithms.

Insertion Sort Pseudocode

1. Initialize array
2. Select first number to be sorted
3. Compare against the next number in the sequence
4. Check If number to be sorted is smaller than next number
5. If it is smaller, insert number in correct place
6. If it is not smaller then keep looking until a smaller number is found and insert there
7. Select next number to be sorted and repeat

Counting Sort Pseudocode

1. Find max and min value
2. Count cumulative occurrences of each number
3. Loop through array and place each number in its correct position
4. After number is used decrease count for that number

Insertion sort algorithm

```
42
43     long startTime = System.nanoTime();
44     for(int t = 0; t < 999; t++) {
45         insertionSort(num);
46         //System.out.println("\nAfter Sorting: ");
47         //printArray(num);
48     }
49     long endTime = System.nanoTime();
50
51     long duration = (endTime - startTime);
52     System.out.println();
53     System.out.println("\nAverage Time Taken: " + (duration/1000));
54
55
56 }
57
58 private static void insertionSort(int[] arr) {
59     for (int i = 1; i < arr.length; i++) {           //loop through array
60         int valueToSort = arr[i];                   //select value
61         int j = i;
62         while (j > 0 && arr[j - 1] > valueToSort) {   // check value of selected num
63             arr[j] = arr[j - 1];
64             j--;
65         }
66         arr[j] = valueToSort;                       //Place in correct position
67     }
68 }
69
70 public static void printArray(int[] B) {
71     System.out.println(Arrays.toString(B));
72 }
73 }
```

Adapted from <http://www.journaldev.com/585/insertion-sort-in-java-algorithm-and-code-with-example>

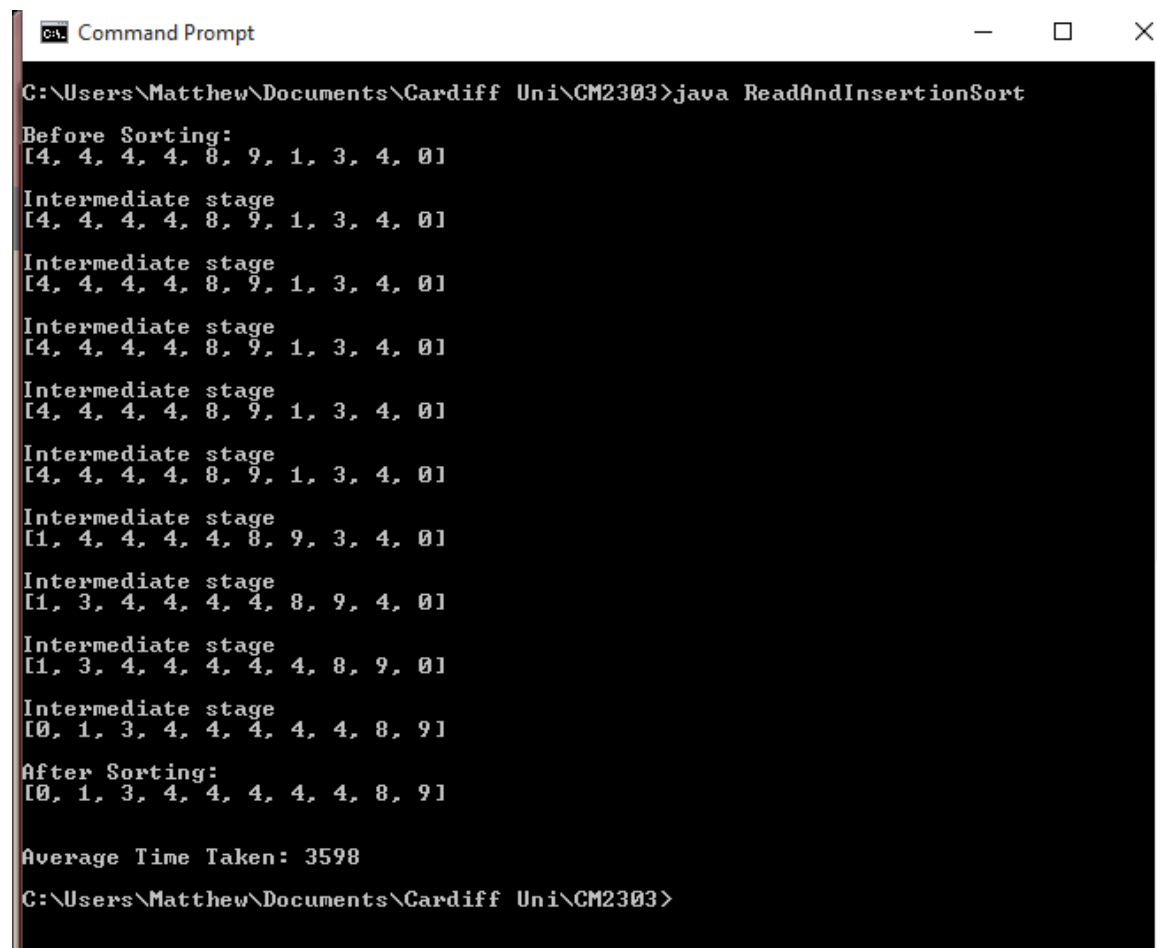
Counting sort algorithm

```
48     // printArray(num);
49 }
50 long endTime = System.nanoTime(); // stop timing
51
52 long duration = (endTime - startTime); //calculate duration
53 System.out.println();
54 System.out.println("\nAverage Time Taken: " + (duration/1000)); // work out average du
55
56 }
57
58 private static void countingSort(int[] arr){
59     int min=arr[0];
60     int max=arr[0];
61     for(int ii=0;ii<arr.length;ii++){
62         if(arr[ii]<min){
63             min=arr[ii];
64         }
65         else if(arr[ii]>max){
66             max=arr[ii];
67         }
68     }
69     int[] count= new int[max - min + 1]; //Create empty array to count values
70     for(int number : arr){
71         count[number - min]++;
72     }
73     int z= 0;
74     for(int i= min;i <= max;i++){ //Create array to hold sorted values
75         while(count[i - min] > 0){
76             arr[z]= i;
77             z++;
78             count[i - min]--; //Decrease count once value is used
79         }
80     }
81 }
82
83
84 public static void printArray(int[] B) {
85     System.out.println(Arrays.toString(B));
86 }
87 }
```

Adapted from http://rosettacode.org/wiki/Sorting_algorithms/Counting_sort#Java

As stated previously I adapted some provided code to output various files of different sizes containing lists of numbers. I have created five files for each case that will be tested, sorted, reverse sorted and random. The number of values (n) of these files starts at 10 and increases by a factor of 10 up to 100000. I decided to measure the elapsed time using the `system.nanoTime()` function. The following are some examples of the code functioning on the file with 10 values.

Example of Insertion Algorithm



```
C:\Users\Matthew\Documents\Cardiff Uni\CM2303>java ReadAndInsertionSort
Before Sorting:
[4, 4, 4, 4, 8, 9, 1, 3, 4, 0]
Intermediate stage
[4, 4, 4, 4, 8, 9, 1, 3, 4, 0]
Intermediate stage
[4, 4, 4, 4, 8, 9, 1, 3, 4, 0]
Intermediate stage
[4, 4, 4, 4, 8, 9, 1, 3, 4, 0]
Intermediate stage
[4, 4, 4, 4, 8, 9, 1, 3, 4, 0]
Intermediate stage
[4, 4, 4, 4, 8, 9, 1, 3, 4, 0]
Intermediate stage
[1, 4, 4, 4, 4, 8, 9, 3, 4, 0]
Intermediate stage
[1, 3, 4, 4, 4, 4, 8, 9, 4, 0]
Intermediate stage
[1, 3, 4, 4, 4, 4, 4, 8, 9, 0]
Intermediate stage
[0, 1, 3, 4, 4, 4, 4, 4, 8, 9]
After Sorting:
[0, 1, 3, 4, 4, 4, 4, 4, 8, 9]

Average Time Taken: 3598
C:\Users\Matthew\Documents\Cardiff Uni\CM2303>
```

Example of Counting Algorithm

```
Command Prompt
C:\Users\Matthew\Documents\Cardiff Uni\CM2303>java ReadAndCountingSort
Before Sorting:
[4, 4, 4, 4, 8, 9, 1, 3, 4, 0]
Intermediate stage
[0, 4, 4, 4, 8, 9, 1, 3, 4, 0]
Intermediate stage
[0, 1, 4, 4, 8, 9, 1, 3, 4, 0]
Intermediate stage
[0, 1, 3, 4, 8, 9, 1, 3, 4, 0]
Intermediate stage
[0, 1, 3, 4, 8, 9, 1, 3, 4, 0]
Intermediate stage
[0, 1, 3, 4, 4, 9, 1, 3, 4, 0]
Intermediate stage
[0, 1, 3, 4, 4, 4, 1, 3, 4, 0]
Intermediate stage
[0, 1, 3, 4, 4, 4, 4, 3, 4, 0]
Intermediate stage
[0, 1, 3, 4, 4, 4, 4, 4, 4, 0]
Intermediate stage
[0, 1, 3, 4, 4, 4, 4, 4, 8, 0]
Intermediate stage
[0, 1, 3, 4, 4, 4, 4, 4, 8, 9]
After Sorting:
[0, 1, 3, 4, 4, 4, 4, 4, 8, 9]
Average Time Taken: 2415
C:\Users\Matthew\Documents\Cardiff Uni\CM2303>_
```

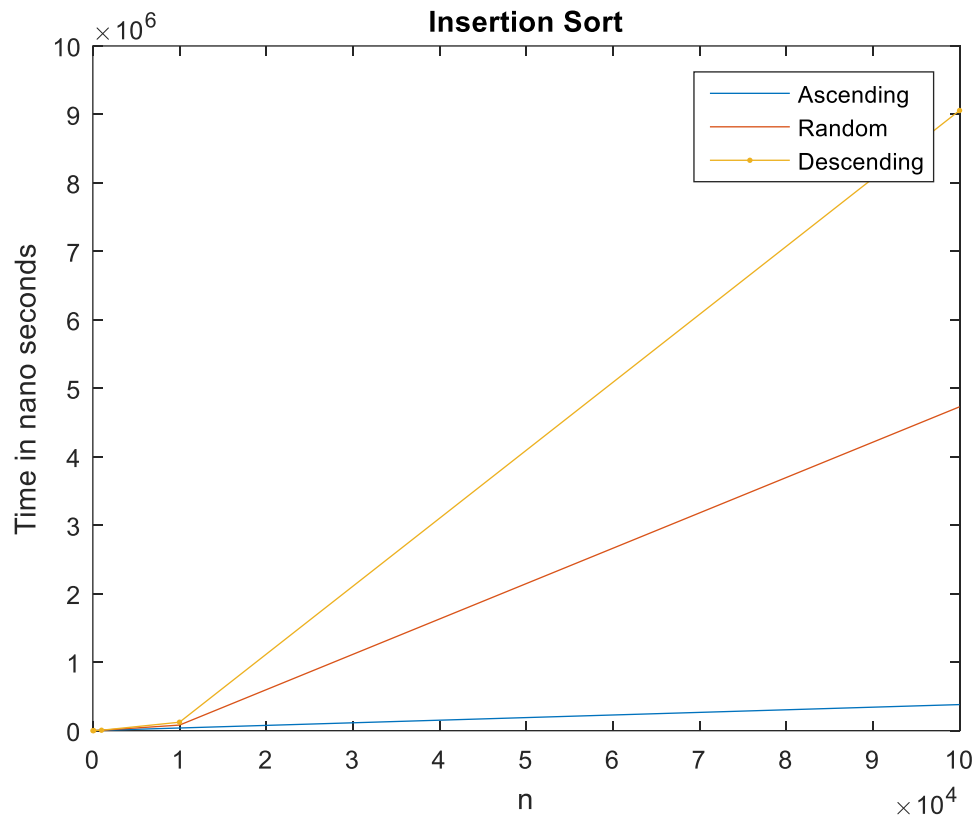
For the experiment itself I planned to run the code once for each file, so that was five files of increasing size for each scenario, giving a total of fifteen runs per algorithm. However when testing my code I discovered that results could vary from run to run and only running the code once was not an adequate way to obtain reliable results. I decided to add an extra loop to the code so that the sub routine that contained the algorithm was called 1000 times, the total time was then measured and divided by 1000 to obtain an average time. From the theory I was expecting to find that the best case scenario is a pre-sorted list, the average case would be a random list and the worst case would be a reverse sorted list. I also expected to find that the insertion sort algorithm would be fastest for very small files and the counting sort would become the quicker as file size move towards very large.

| Insertion sort | | | | | |
|----------------|-------------------------|------------|-------------------------|----------------|-------------------------|
| Random | | Pre-sorted | | Reverse sorted | |
| n | Time (s ⁻⁹) | n | Time (s ⁻⁹) | n | Time (s ⁻⁹) |
| 10 | 341 | 10 | 332 | 10 | 332 |
| 100 | 1707 | 100 | 1477 | 100 | 1489 |
| 1000 | 5988 | 1000 | 5251 | 1000 | 6664 |
| 10000 | 81786 | 10000 | 39221 | 10000 | 123816 |
| 100000 | 4730114 | 100000 | 381194 | 100000 | 9057776 |

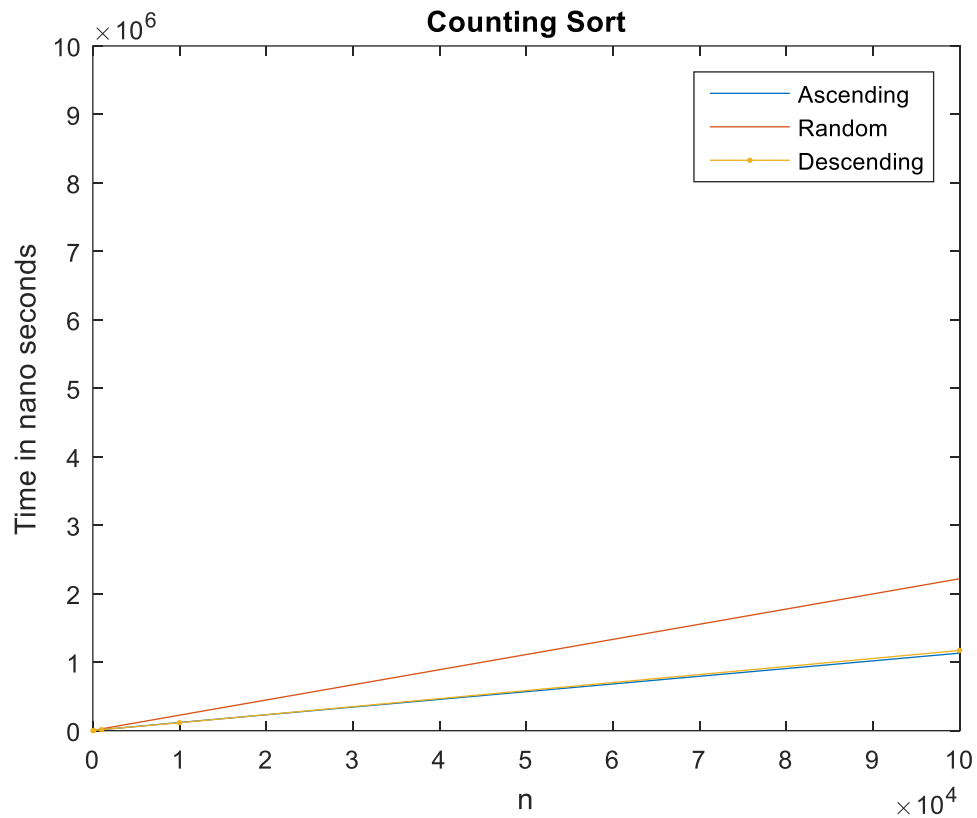
Here we can see that the predicted trend is followed, pre-sorted data was the quickest, random data was in the middle and reverse sorted data was the slowest, these tendencies became more pronounced as n became larger.

| Counting Sort | | | | | |
|---------------|-------------------------|------------|-------------------------|----------------|-------------------------|
| Random | | Pre-sorted | | Reverse sorted | |
| n | Time (s ⁻⁹) | n | Time (s ⁻⁹) | n | Time (s ⁻⁹) |
| 10 | 610 | 10 | 639 | 10 | 592 |
| 100 | 3322 | 100 | 2682 | 100 | 2673 |
| 1000 | 26391 | 1000 | 14747 | 1000 | 15422 |
| 10000 | 226773 | 10000 | 119707 | 10000 | 116711 |
| 100000 | 2218634 | 100000 | 1132298 | 100000 | 1171658 |

Again here the prediction is correct, there is less difference between the elapsed times regardless of the state of the data before it is sorted, however the values for random seem abnormally high compared to both pre-sorted and reverse sorted data. It is also true that for small values of n the insertion sort was quicker and as n became larger the counting sort then became the quicker method.



The graph here also confirms my predicted results, in this form however it is much easier to see the huge variation for each of the data types, and the rapid increase in gradient as n increases. This also seems to follow the prediction that a pre sorted list should be of order $O(n)$ and random and descending being of order $O(n^2)$.



The graph for the counting sort shows there is very little difference between the pre-sorted (ascending) and reverse sorted (descending) values. However the random files seem to have taken much longer, although still much quicker than using the insertion sort. The time for pre sorted and reverse sorted data follow the prediction that they are of order $O(n)$.

Overall my predicted results held true across the experiments. My graphs show that the elapsed time followed the theory from big O notation. You can also see quite clearly that for small values of n insertion sort was the quickest method but then the curve goes steeply upwards and as n increases the counting sort becomes the better option.

During my experiment there seems to have been some issues with the runtime calculations for random values using the counting sort algorithm. I believe this was due to the manner in which I generated my data. For both the pre sorted and reverse sorted data the numbers were generated in a linear fashion and have no repeated values, whereas the random numbers may contain repeats. This also means that the random numbers may contain a higher percentage of numbers with more digits which may have been a contributing factor to the longer runtime.

If I were to repeat this experiment I would be more careful when producing my test data, making sure to only use the same list of random numbers and pre-process it to obtain a pre-sorted and reverse sorted list. I would also like to take many more data points at smaller intervals to plot a much more accurate graph and hopefully see a more pronounced curve.

Despite the few discrepancies with the random data values for the counting sort, I believe that the experiment was a success. Both my data tables and my graphs clearly support my original predictions, showing that the theory held true when put into practice.