A run-time analysis of insertion sort and counting sort

Jack Furby
CM2303
February 6, 2018

1 Introduction

In this report I aim to compare the theoretical run-time with the actual run-time of two sorting algorithms: insertion sort and counting sort. Insertion sort has the best theoretical run-time complexity of $o(n)$ (n is the size of the input array) which will have the respective input of a sorted array in ascending order. Both the average and worst theoretical run-time complexity is $o(n^2)$ with a random array and an array in descending order as an input respectively. Counting sort has a good theoretical run-time complexity of $o(n)$ achieved when the value of k (where $k-1$ is the maximum value in the input array) is the same or less than the value of n. For a bad theoretical run-time complexity for counting sort it will be $o(n+k)$. This will have an input of k larger than the value of n. With counting sort the run-time complexity is always $0(n+k)$ but if k is the same as n this would work out as $o(n)$.

2 Pseudocode and implementation

2.1 Pseudocode

2.1.1 Insertion sort

Insertion sort takes in an array A of n integers and the value of n itself. In the algorithm it will go ever each element in ascending order of index and compare it to other elements who's current index is lower than itself. It will start at the previous element and continue to descend the array by index until it finds an element with a lower value. If there are elements with larger values then the element in focus will be moved so it then comes before such elements. Because of this after each pass of the array from index zero to the current index being compared will be sorted, although may still change on future iterations.

```
Algorithm insertionSort(A, n)
Input: an array $A$ storing $n$ integers
Output: array $A$ sorted in non descending order
for $i \leftarrow 1$ to $(n-1)$ do
    $item \leftarrow A[i]$
    $j \leftarrow i-1$
```

```
        **while** $j \geq 0$ and $A[i] > item$ **do**
            $A[j+1] \leftarrow A[j]$
            $j \leftarrow j - 1$
        **end while**
        $A[j+1] \leftarrow item$
    **end for**
```

2.1.2 Counting sort

Insertion sort takes in an array A of n integers, the value of n and the value of k where $k-1$ is the maximum element value. B is the sorted output array. Counting sort is made of 4 loops with the first making an array C of 0's k in size, the second filling that array with the count of numbers in input array A in their respective indexes (e.g. an element with the value 2 in the input array would add 1 to the value currently at index 2 in array C), The 3rd loop adds all the elements in array C together such that element $C[i] = C[i] + C[i-1]$ starting at index 1 and working up till the end. Finally the last loop goes over each element in input array A and adds the current element to output array B using array C to find the index. Due to half of the loops in this sorting algorithm having a run-time complexity of $o(n)$ and the other half being $o(k)$ the run-time complexity is $o(n+k)$. If k is the same as n then this would result in the run-time complexity of $o(n)$.

```
Algorithm countingSort(A, B, n, k)
Input: array $A$, with $n$ elements, each with value from 0 to $k-1$
Output: sorted array $B$
**for** $i \leftarrow 0$ to $k-1$ **do**
    $C[i] \leftarrow 0$
**end for**
**for** $j \leftarrow 0$ to $n-1$ **do**
    $C[A[j]] \leftarrow C[A[j]] + 1$
**end for**
**for** $i \leftarrow 1$ to $k-1$ **do**
    $C[i] \leftarrow C[i] + C[i-1]$
**end for**
**for** $j \leftarrow n-1$ downto 0 **do**
    $B[C[A[j]] - 1] \leftarrow A[j]$
    $C[A[j]] \leftarrow C[A[j]] - 1$
**end for**
```

2.2 Implementation

The function getRunTime() is called to analyse runtime for the algorithms. It is passed the array sizes to test, sort type, the array type (best, worst, average, good or bad) and finally how many times each array size should be tested. It will return an array containing the average run time for each of the array sizes provided as an input.

```java
double testRunTimeBad[] = getRunTime(a, "countingSort", "bad",
   5500);

// returns an array containing the run-time analysis for
   specified array sizes
public static double[] getRunTime(int arraySize[], String
   sortType, String arrayType, int loops) {
  int warmUp = 1500; // removes high run-times before JIT
     improves the performance
  double runtime[] = new double[arraySize.length];
  for (int i = 0; i < arraySize.length; i++) { // For each
     array length to test
    long currentTime = 0L;
    for (int j = 0; j < loops; j++) { // Loop over the
       specified number of times (used to find average)
      if (sortType == "insertionSort") {

        int a[] = insertionSortArray(arraySize[i], arrayType);
           // Gets an array to sort in a specified format (best
           , worst, average)
        int n = a.length;
        long timeDone = insertionSort(a, n); // Sorts array and
            gets run-time
        if (j > warmUp) { // Discards run times before JIT
           kicks in
          currentTime += timeDone;
        }

      } else if (sortType == "countingSort") {

        int a[] = countingSortArray(arraySize[i], arrayType);
           // Gets an array to sort in a specified format (good
           , bad)
        int n = a.length;
        int k = n;
        if (arrayType == "bad") { // If running a bad example k
            = n squared
          k = n * n;
        }
        long timeDone = countingSort(a, n, k); // Sorts array
           and gets run-time
        if (j > warmUp) { // Discards run times before JIT
           kicks in
          currentTime += timeDone;
        }
      }
      if (j % 1000 == 0) { // Prints current status
        System.out.println(sortType + "␣-␣i:␣" + i + ",␣j:␣" + j);
      }

    }
    runtime[i] = currentTime / (loops - (warmUp + 1)); // Works out average run time
  }
  return runtime;
}
```

To generate the appropriate array for insertion sort I created the function insertionSortArray(). This takes in the array size and type before returning the selected array. For best array type this will be an array with elements in ascending order from 1 to the size of the array minus 1. Worst will be the same but in descending order. Average will have elements with a value on 0 to 100 but in no particular order.

```
public static int[] insertionSortArray(int size, String type) {
  Random randomGenerator = new Random();
  int currentArray[] = new int[size];
  for (int i = 0; i < size; i++) {
    if (type == "best") { // adds elements to the array for a best run-time
      currentArray[i] = i;
    } else if (type == "worst") { //Adds elements to the array for a worst run-time
      currentArray[size - 1 - i] = i;
    } else if (type == "average") { //Adds elements to the array for a average run-time
      currentArray[i] = randomGenerator.nextInt(100);
    }
  }
  return currentArray;
}
```

As for counting sort this used a function called countingSortArray(). This takes in the array size and type before returning the selected array. For the good array type this will be an array with elements up to the value of the size of the array. For a bad array type elements can be at max n squared in value.

```
public static int[] countingSortArray(int size, String type) {
  Random randomGenerator = new Random();
```

```java
  int currentArray[] = new int[size];
  for (int i = 0; i < size; i++) {
    if (type == "good") { // Adds elements to the array for a
       good run-time
      currentArray[i] = randomGenerator.nextInt(size);
    } else if (type == "bad") { // Adds elements to the array
       for a bad run-time
      currentArray[i] = randomGenerator.nextInt(size * size);
    }
  }
  return currentArray;
}
```

For running the insertion sort algorithm and recording time I am using a the function insertionSort() for insertion sort. this takes the input of an array and the size of the array. The return value will be the runtime of the algorithm. Nanotime is used to record time in order to be as accurate as possible. It starts just before the algorithm starts and ends as soon as it finishes. runtime is then calculated and returned.

```java
public static long insertionSort(int a[], int n) {
  long startTime = System.nanoTime(); // Get start time
  for(int i = 1; i < n; i++) {
    int item = a[i];
    int j = i - 1;
    while (j >= 0 && a[j] > item) {
      a[j + 1] = a[j];
      j = j - 1;
    }
    a[j + 1] = item;
  }
  long endTime = System.nanoTime(); // Get end time
  long currentTime = endTime - startTime; // Work out run-time
  return currentTime;
}
```

For counting sort I have made the function countingSort(). This takes the input of an array, the size of the array and the maximum value in the array . The return value will be the runtime of the algorithm. Nanotime is used to record time in order to be as accurate as possible with it starting just before the algorithm starts and ends as soon as it finishes. runtime is then calculated and returned.

```java
public static long countingSort(int a[], int n, int k) {
  long startTime = System.nanoTime(); // Get start time
  int c[] = new int[k];
  int b[] = new int[n];
  for(int i = 0; i < k-1; i++) {
    c[i] = 0;
  }
  for(int j = 0; j < n; j++) {
    c[a[j]] = c[a[j]] + 1;
  }
  for(int i = 1; i < k; i++) {
    c[i] = c[i] + c[i - 1];
  }
  for(int j = n - 1; j >= 0; j--) {
    b[c[a[j]] - 1] = a[j];
    c[a[j]] = c[a[j]] - 1;
  }
  long endTime = System.nanoTime(); // Get end time
  long currentTime = endTime - startTime; // Work out run-time
  return currentTime;
}
```

3 Testing

3.1 Insertion sort

For insertion sort I will be conducting my testing with 3 array inputs. These are as follows:
Best: $[1, 3, 5, 7, 9, 11, 13, 15]$
Worst: $[14, 12, 10, 8, 6, 4, 2, 0]$
Average: $[2, 5, 3, 0, 2, 3, 0, 3]$

I selected these as they will demonstrate the 3 different conditions that will affect the run-time of the algorithm.

3.1.1 Best

Start: $[1, 3, 5, 7, 9, 11, 13, 15]$, $n = 8$
First pass: $[1, 3, 5, 7, 9, 11, 13, 15]$
Result: $[1, 3, 5, 7, 9, 11, 13, 15]$
Fourth pass: $[1, 3, 5, 7, 9, 11, 13, 15]$
Result: $[1, 3, 5, 7, 9, 11, 13, 15]$
Seventh pass (Final): $[1, 3, 5, 7, 9, 11, 13, 15]$
Result: $[1, 3, 5, 7, 9, 11, 13, 15]$

3.1.2 Worst

Start: $[14, 12, 10, 8, 6, 4, 2, 0]$
First pass: $[14, 12, 10, 8, 6, 4, 2, 0]$

Result: $[12, 14, 10, 8, 6, 4, 2, 0]$
Fourth pass: $[8, 10, 12, 14, 6, 4, 2, 0]$
Result: $[6, 8, 10, 12, 14, 4, 2, 0]$
Seventh pass (Final): $[2, 4, 6, 8, 10, 12, 14, 0]$
Result: $[0, 2, 4, 6, 8, 10, 12, 14]$

3.1.3 Average

Start: $[2, 5, 3, 0, 2, 3, 0, 3]$
First pass: $[2, 5, 3, 0, 2, 3, 0, 3]$
Result: $[2, 5, 3, 0, 2, 3, 0, 3]$
Fourth pass: $[0, 2, 3, 5, 2, 3, 0, 3]$
Result: $[0, 2, 2, 3, 5, 3, 0, 3]$
Seventh pass (Final): $[0, 0, 2, 2, 3, 3, 5, 3]$
Result: $[0, 0, 2, 2, 3, 3, 3, 5]$

3.2 Counting sort

For counting sort I will be conducting my testing with 3 array inputs and values of k. These are as follows:
Good: $[7, 0, 5, 7, 2, 4, 1, 6], k = 8$
Bad: $[14, 12, 10, 1, 5, 15, 2, 0], k = 16$
Small k: $[1, 0, 0, 0, 1, 1, 0, 1], k = 2$

I selected these as they will demonstrate the 2 different conditions that will affect the run-time of the algorithm.

3.2.1 Good

Start: $[7, 0, 5, 7, 2, 4, 1, 6]$
After the first for loop: $C = [0, 0, 0, 0, 0, 0, 0, 0]$
After the second for loop: $C = [1, 1, 1, 0, 1, 1, 1, 2]$
After the third for loop: $C = [1, 2, 3, 3, 4, 5, 6, 8]$
First pass of forth loop:
$A = [7, 0, 5, 7, 2, 4, 1, 6]$
$B = [0, 0, 0, 0, 0, 0, 0, 0]$
$C = [1, 2, 3, 3, 4, 5, 6, 8]$
Result:
$B = [0, 0, 0, 0, 0, 6, 0, 0]$
$C = [1, 2, 3, 3, 4, 5, 5, 8]$
Forth pass of forth loop:
$A = [7, 0, 5, 7, 2, 4, 1, 6]$

$B = [0, 1, 0, 4, 0, 6, 0, 0]$
$C = [1, 1, 3, 3, 3, 5, 5, 8]$
Result:
$B = [0, 1, 2, 4, 0, 6, 0, 0]$
$C = [1, 1, 2, 3, 3, 5, 5, 8]$
Eighth pass of forth loop:
$A = [7, 0, 5, 7, 2, 4, 1, 6]$
$B = [0, 1, 2, 4, 5, 6, 0, 7]$
$C = [0, 1, 2, 3, 3, 4, 5, 7]$
Result:
$B = [0, 1, 2, 4, 5, 6, 7, 7]$
$C = [0, 1, 2, 3, 3, 4, 5, 6]$

3.2.2 Bad

Start: $[14, 12, 10, 1, 5, 15, 2, 0]$
After the first for loop: $C = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]$
After the second for loop: $C = [1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 1]$
After the third for loop: $C = [1, 2, 3, 3, 3, 4, 4, 4, 4, 4, 5, 5, 6, 6, 7, 8]$
First pass of forth loop:
$A = [14, 12, 10, 1, 5, 15, 2, 0]$
$B = [0, 0, 0, 0, 0, 0, 0, 0]$
$C = [1, 2, 3, 3, 3, 4, 4, 4, 4, 4, 5, 5, 6, 6, 7, 8]$
Result:
$B = [0, 0, 0, 0, 0, 0, 0, 0]$
$C = [0, 2, 3, 3, 3, 4, 4, 4, 4, 4, 5, 5, 6, 6, 7, 8]$
Forth pass of forth loop:
$A = [14, 12, 10, 1, 5, 15, 2, 0]$
$B = [0, 0, 2, 0, 0, 0, 0, 15]$
$C = [0, 2, 2, 3, 3, 4, 4, 4, 4, 4, 5, 5, 6, 6, 7, 7]$
Result:
$B = [0, 0, 2, 5, 0, 0, 0, 15]$
$C = [0, 2, 2, 3, 3, 3, 4, 4, 4, 4, 5, 5, 6, 6, 7, 7]$
Eighth pass of forth loop:
$A = [14, 12, 10, 1, 5, 15, 2, 0]$
$B = [0, 1, 2, 5, 10, 12, 0, 15]$
$C = [0, 1, 2, 3, 3, 3, 4, 4, 4, 4, 4, 5, 5, 6, 7, 7]$
Result:
$B = [0, 1, 2, 5, 10, 12, 14, 15]$
$C = [0, 1, 2, 3, 3, 3, 4, 4, 4, 4, 4, 5, 5, 6, 6, 7]$

3.2.3 Small k

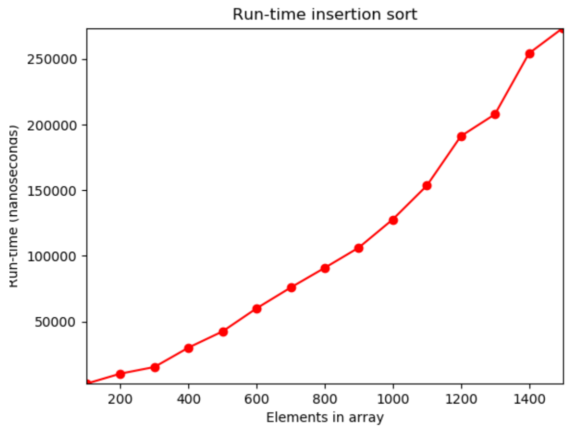
Start: $[1, 0, 0, 0, 1, 1, 0, 1]$
After the first for loop: $C = [0, 0]$
After the second for loop: $C = [4, 4]$
After the third for loop: $C = [4, 8]$
First pass of forth loop:
$A = [1, 0, 0, 0, 1, 1, 0, 1]$
$B = [0, 0, 0, 0, 0, 0, 0, 0]$
$C = [4, 8]$
Result:
$B = [0, 0, 0, 0, 0, 0, 0, 1]$
$C = [4, 7]$
Forth pass of forth loop:
$A = [1, 0, 0, 0, 1, 1, 0, 1]$
$B = [0, 0, 0, 0, 0, 0, 1, 1]$
$C = [3, 6]$
Result:
$B = [0, 0, 0, 0, 0, 1, 1, 1]$
$C = [3, 5]$
Eighth pass of forth loop:
$A = [1, 0, 0, 0, 1, 1, 0, 1]$
$B = [0, 0, 0, 0, 0, 1, 1, 1]$
$C = [0, 5]$
Result:
$B = [0, 0, 0, 0, 1, 1, 1, 1]$
$C = [0, 4]$

4 Experimental setup and results

4.1 Setup

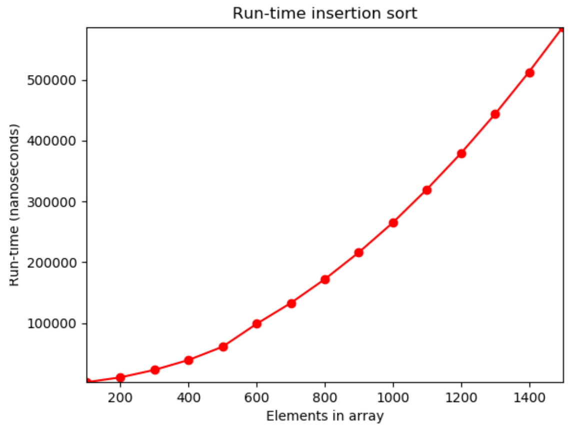
4.1.1 Array size

For my array size I tested a number of different options. This started with 15 arrays starting will 100 and going up in steps of 100. This proved to be partially accurate but it also required a large numbers of iterations to get any usable data as seen in figure 1 and figure 2. Moving on from here I jumped to starting at 1000 and increased my arrays in steps of 1000. This continue to use 15 arrays. With larger steps and larger arrays my results were more consistent from run to run as seen in figure 3. As these array sizes worked well I stayed with this for my final testing for $o(n)$ run-time complexity. When I tested for $o(n^2)$ run-time complexity I changed my array sizes to start at 1000 and have steps of $\sqrt{4000000}$. Figure 9 displays the result from using these array sizes. This like with steps of 1000 yielded good results.



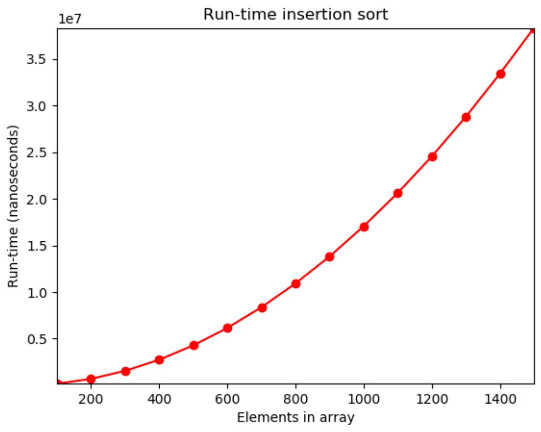
- Average case insertion sort

Figure 1: Insertion sort with 500 iterations and steps of 100 between array sizes



- Average case insertion sort

Figure 2: Insertion sort with 92500 iterations and steps of 100 between array sizes

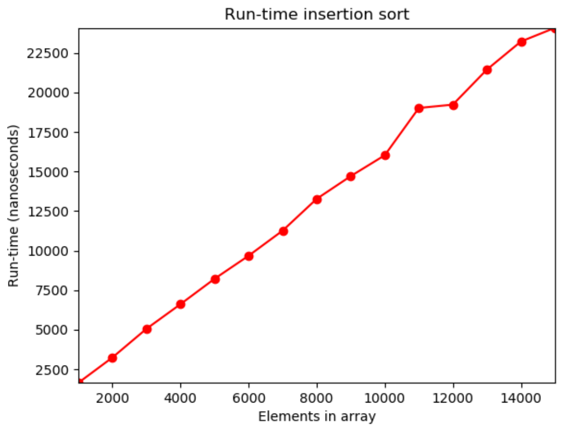


- Average case insertion sort

Figure 3: Insertion sort with 2500 iterations and steps of 1000 between array sizes

4.1.2 Iterations

As I could not guarantee my computer to always run an algorithm in the same time I opted to run each algorithm multiple times for each array size and test case. From here I would sum together all the run times and divide them by how many iterations I had run. This would achieve finding the average. For my arrays going up in steps of 100 I would have to run 100,000 iterations to get semi accurate results (figure 1 and figure 2) whereas with arrays of step 1000 this was reduced to around 2500 iterations (figure 4 and figure 5).



- Best case insertion sort

Figure 4: Insertion sort with 500 iterations and steps of 1000 between array sizes

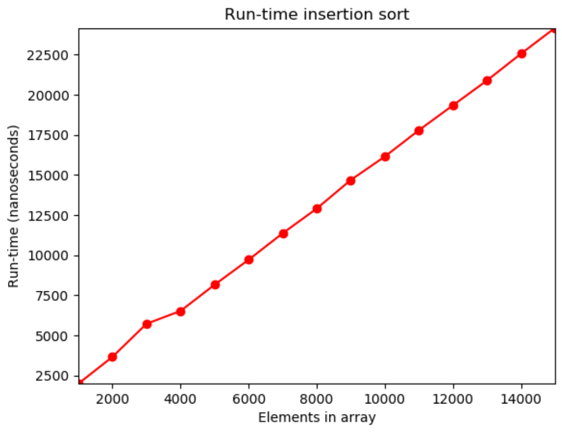


Figure 5: Insertion sort with 2500 iterations and steps of 1000 between array sizes

4.1.3 Just-in-time (JIT) compilation

Java has a feature called Just-in-time compilation (JIT) which will create bytecode for sections of code at run time. This would mean the first few run of the algorithms will be slower than subsequent runs. Because of this my smaller arrays would end up with a higher run time than larger arrays considering I ran my smaller arrays before the larger ones. To combat this I discarded some results from the earliest runs. For my array sizes this worked out to be approximately 1000 runs.

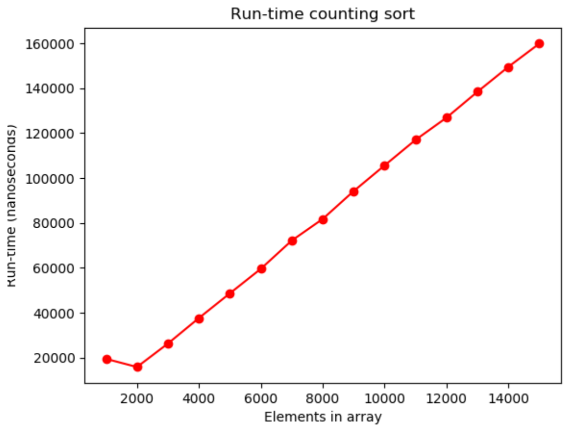
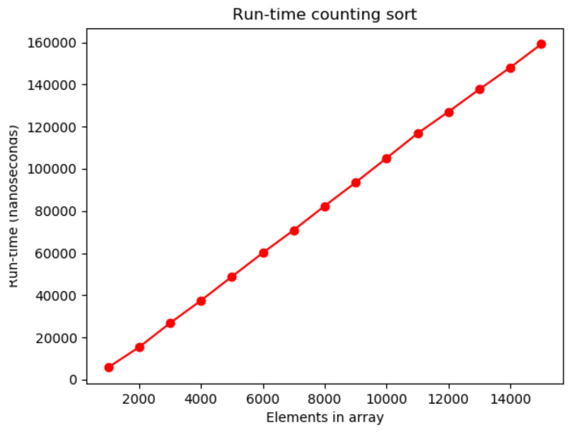


Figure 6: Counting sort with 1000 iterations without letting the JIT compiler create bytecode first



- Good case counting sort

Figure 7: Counting sort with 1000 iterations to allow the JIT compiler create bytecode

4.1.4 Computer setup

My setup was critical to getting accurate results. Making the correct arrangements would minimalize the affect of other processes on my results.

I settled on using Windows 10 with all applications closed and processes ended that did not affect the stability of the system. Unfortunately it was not possible to close all processes as applications such as Cortana would reboot whenever I closed it. To get round issues such as this however I could disable their functionality so they would not process anything (or significantly limit it). In addition to this I also disconnected my computer from the internet. This would mean windows update would not start and / or other applications running checks such as my email client. Finally during testing I would not use the computer.

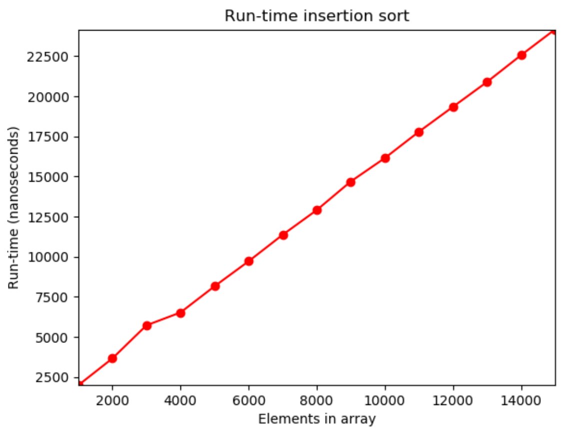
4.2 Insertion sort

4.2.1 Best case

To test insertion sort with an input of an sorted array I used 15 arrays starting at 1000 elements in size and going up by 1000 each time. I selected these array sizes as the best case should have $o(n)$ run-time complexity. This would mean there is an even distribution of point along the x axis. In addition to the array sizes I settled on 2500 iterations of each array size which was then used to take an average and an additional 1000 iterations which were discarded so JIT could take affect. Table 1 and figure 8 shows my results using the parameters mentioned.

Array size	**Run-time (nanoseconds)**
1000	1992
2000	3661
3000	5723
4000	6523
5000	8157
6000	9714
7000	11364
8000	12902
9000	14685
10000	16144
11000	17785
12000	19336
13000	20885
14000	22554
15000	24164

Table 1: Insertion sort best case results



- Best case insertion sort

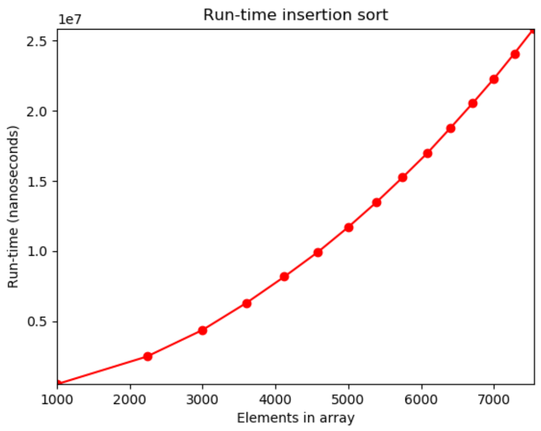
Figure 8: Insertion sort with 2500 iterations and 1000 iterations for JIT

4.2.2 Worst case

To test insertion sort with an input of an array with elements sorted in descending order I used 15 arrays starting at 1000 elements in size and going up each time such that the increase between each element squared (n^2) is approximately 4000000. I selected these data points so when plotting the results I can represent them with n^2 along the x axis which should show a straight line if the run-time analysis is correct. In addition to the array sizes I settled on 2500 iterations of each array size which was then used to take an average and an additional 1000 iterations which were discarded so JIT could take affect. Table 2 and figure 9 shows my results using the parameters mentioned.

Array size	**Run-time (nanoseconds)**
1000	507534
2236	2479180
3000	4374641
3606	6316950
4123	8174222
4583	9929671
5000	11709050
5385	13475794
5745	15248191
6083	16970999
6403	18779481
6708	20542092
7000	22289156
7280	24083769
7550	25875480

Table 2: Insertion sort worst case results



- Worst case insertion sort

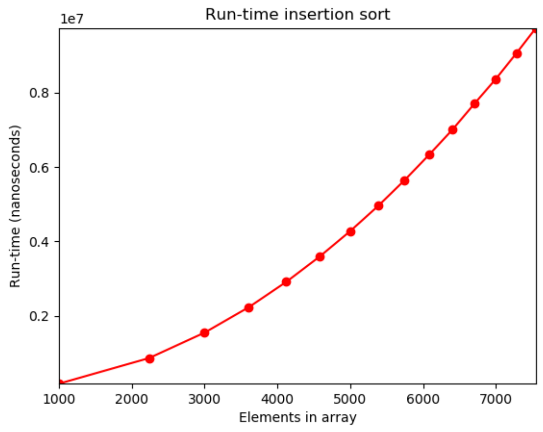
Figure 9: Insertion sort with 2500 iterations and 1000 iterations for JIT

4.2.3 Average case

Like with worst case the run-time complexity for average case should be $o(n^2)$ and therefore the same arrays can be used here. These are 15 arrays starting at 1000 elements in size and going up each time such that the increase between each element squared (n^2) is approximately 4000000. In addition to the array sizes I settled on 2500 iterations of each array size which was then used to take an average and an additional 1000 iterations which were discarded so JIT could take affect. Table 3 and figure 10 shows my results using the parameters mentioned.

Array size	**Run-time (nanoseconds)**
1000	176745
2236	863653
3000	1548924
3606	2231001
4123	2915075
4583	3598423
5000	4278139
5385	4957967
5745	5644644
6083	6330219
6403	7004524
6708	7709478
7000	8370063
7280	9055319
7550	9738642

Table 3: Insertion sort average case results



- Average case insertion sort

Figure 10: Insertion sort with 2500 iterations and 1000 iterations for JIT

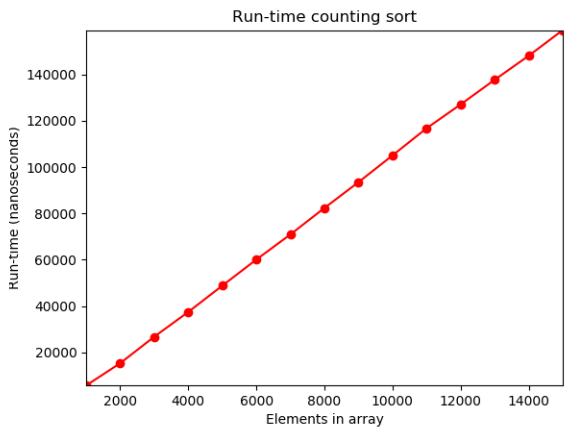
4.3 Counting sort

4.3.1 Good case

To test counting sort for a good run-time complexity I used 15 arrays starting at 1000 elements in size and going up by 1000 each time. I selected these array sizes as a good case should have $o(n)$ run-time complexity. This would mean there is an even distribution of point along the x axis. In addition to the array sizes I settled on 2500 iterations of each array size which was then used to take an average and an additional 1000 iterations which were discarded so JIT could take affect. Table 4 and figure 11 shows my results using the parameters mentioned.

Array size	**Run-time (nanoseconds)**
1000	5671
2000	15289
3000	26724
4000	37440
5000	48801
6000	60078
7000	70929
8000	82367
9000	93515
10000	105084
11000	116795
12000	127103
13000	137755
14000	148145
15000	159207

Table 4: Counting sort good case results



- Good case counting sort

Figure 11: Counting sort with 2500 iterations and 1000 iterations for JIT

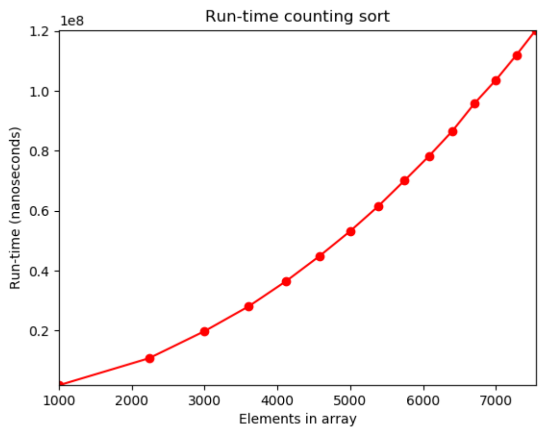
4.3.2 Bad case

A bad case for counting sort run-time complexity should be $o(n + k)$. For my testing $k = n^2$ making the run-time complexity $o(n^2)$ and therefore the arrays to test this will be the same as I have used for insertion sort worse and average case. These are 15 arrays starting at 1000 elements in size and going up each time such that the increase between each element squared

(n^2) is approximately 4000000. In addition to the array sizes I settled on 2500 iterations of each array size which was then used to take an average and an additional 1000 iterations which were discarded so JIT could take affect. Table 5 and figure 12 shows my results using the parameters mentioned.

Array size	**Run-time (nanoseconds)**
1000	1795145
2236	10791591
3000	19849868
3606	28132806
4123	36599559
4583	45004553
5000	53242759
5385	61562966
5745	70120127
6083	78323181
6403	86665508
6708	95918456
7000	103676127
7280	111964609
7550	120394019

Table 5: Counting sort bad case results



- Bad case counting sort

Figure 12: Counting sort with 2500 iterations and 1000 iterations for JIT

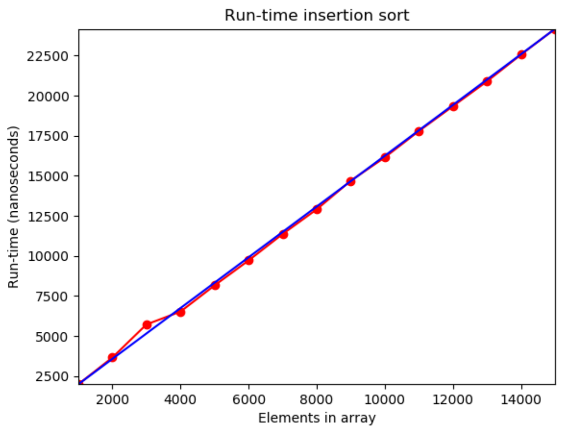
5 Conclusions and Discussion

In conclusion my experiments ended with the same result as the prediction. This can be seen with the following:

5.1 Insertion sort

5.1.1 Best case

Run-time complexity $o(n)$ is linear meaning as x increases y will also tend to increase. This is possible to see by applying a line of best fit to the result. This can be seen in figure 13 where almost all the points fall on or close the line of best fit (the blue line). In addition to this the correlation for the points is 0.99966058. As this is very close to 1 it shows the points have a high linear correlation and therefore represent $o(n)$ run-time complexity.

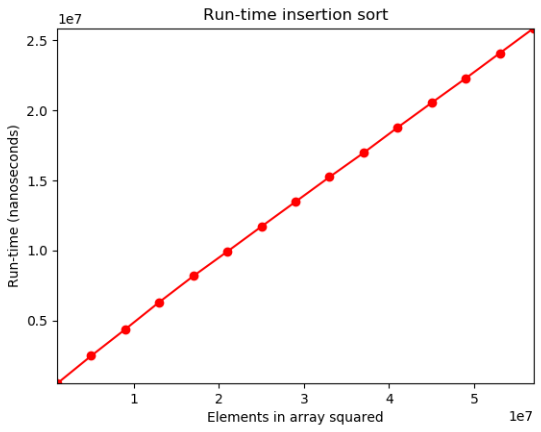


- Best case insertion sort
- $o(n)$ run-time complexity

Figure 13: Insertion sort with 2500 iterations and 1000 for JIT

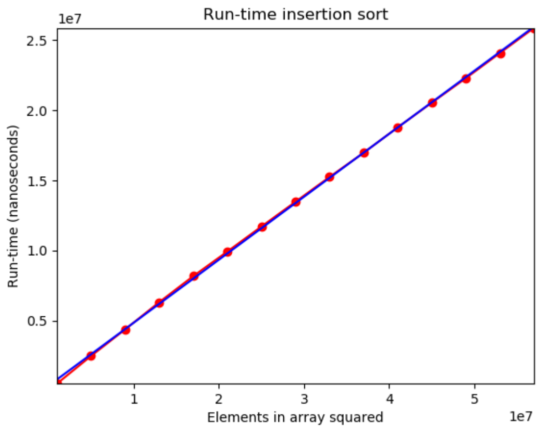
5.1.2 Worst case

To check weather an algorithm is running at $o(n^2)$ complexity I have put n^2 along the x axis. The result as seen in figure 14 shows what this looks like. This can now be checked like we have done for the best case. Figure 15 shows the line of best fit plotted in blue and the correlation between all of the points is 0.98110511. As this is close to 1 correlation is high and therefore run-time complexity is $o(n^2)$.



- Worst case insertion sort

Figure 14: Insertion sort with 2500 iterations and 1000 iterations for JIT

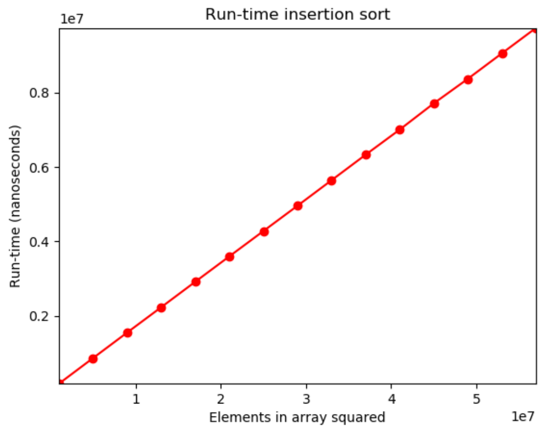


- Worst case insertion sort
- $o(n^2)$ run-time complexity

Figure 15: Insertion sort with 2500 iterations and 1000 iterations for JIT

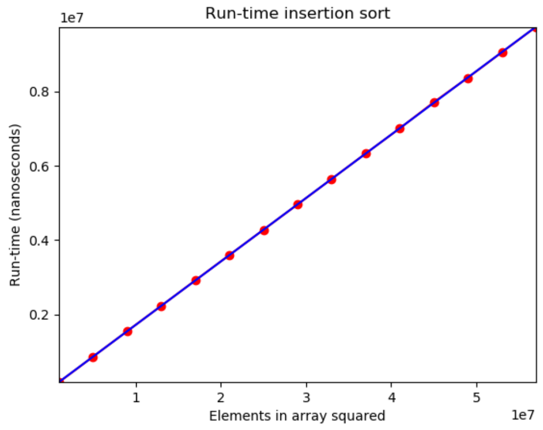
5.1.3 Average case

Again with the average case I have put n^2 along the x axis to check to see if the run-time complexity is $o(n^2)$. The result as seen in figure 16 shows what this looks like. This can now be checked like we have done for the best case. Figure 17 shows the line of best fit plotted in blue and the correlation between all of the points is 0.97824734. As this is close to 1 correlation is high and therefore run-time complexity is $o(n^2)$.



- Average case insertion sort

Figure 16: Insertion sort with 2500 iterations and 1000 iterations for JIT



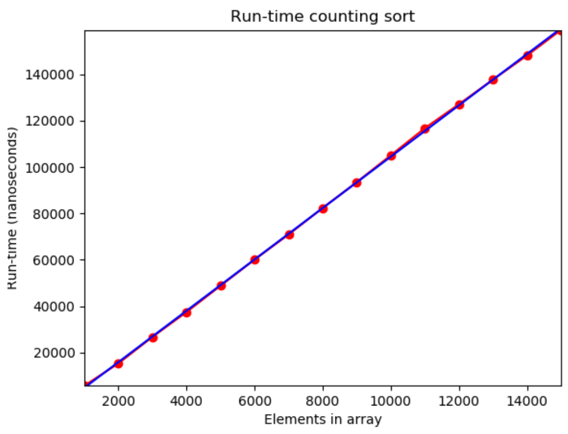
- Average case insertion sort
- $o(n^2)$ run-time complexity

Figure 17: Insertion sort with 2500 iterations and 1000 iterations for JIT

5.2 Counting sort

5.2.1 Good case

Counting with a good input (when k is less than or equal the size of the array to sort) will have a run-time complexity of $o(n)$. This as I've shown with insertion sort can be shown by finding the correlation between the points. In this case it is 0.99950304 which shows high correlation. Figure 18 shows the plot of run times in red with the line of best fit in blue.

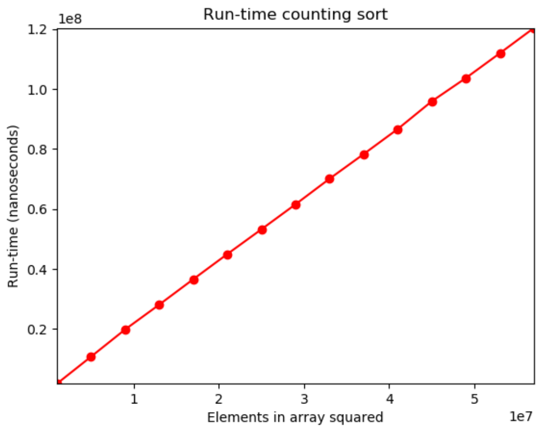


- Good case counting sort
- $o(n)$ run-time complexity

Figure 18: Counting sort with 2500 iterations and 1000 iterations for JIT

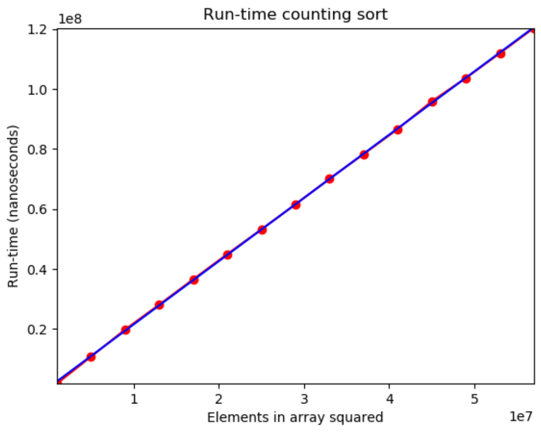
5.2.2 Bad case

In my testing I made the value of k $= n^2$. By doing this the run-time complexity ends up being $o(n^2)$. This still follows the theoretical run-time complexity of $o(n + k)$. Because I ended up with run-times which increased by what looks like n^2 I put n^2 on the x axis. This as shown in figure 19 ends up with a straight line and after adding the line of best fit (the blue line in figure 20) shows the points are plotted in a straight line. To confirm this is a straight line the correlation between the points is 0.97943326.



- Bad case counting sort

Figure 19: Counting sort with 2500 iterations and 1000 iterations for JIT



- Bad case counting sort
- $o(n^2)$ run-time complexity

Figure 20: Counting sort with 2500 iterations and 1000 iterations for JIT

5.3 Discrepancies

By looking at the correlation of each test you can see they are not perfect. If this was the case they would all be 1 (or -1 if I plotted larger arrays first). The reason for this is as much as I limited other processes from occurring some system processes would still take CPU time and therefore affect my results. This would be making use of context switching adding extra time to run-time without making any progress on the algorithms. In addition my CPU frequency could have changed during testing. If this did change during testing some loops my have been processed by the CPU faster as the frequency controls how fast operations are carried out. I managed to combat all of these issues reasonably well by disabling as much as I could in the operating system and BIOS. All of my correlations are within about 0.03 of 1 which is very good. Anything above 0.8 is considered high.

5.4 When to use each algorithm

Insertion sort is fastest with a sorted array. For this reason if you have an sorted array or close to sorted it is a very efficient algorithm. After this point though it will stop being efficient. As seen in the average case run-time complexity quickly reaches $o(n^2)$. With larger arrays this will be noticeable .

Counting sort is good to use if the value of k is close or less than the value of n. If like in my example k is significantly larger than n it becomes very inefficient to run. The other thing to take note with counting sort is it is not very space efficient. As much as it can take large arrays the amount of required cache and RAM if the arrays are large enough will be significant. After all of this however the arrangement of numbers in an array to sort will not affect run time allowing (if k is an appropriate value) the algorithm to sort the numbers with an efficient run-time.