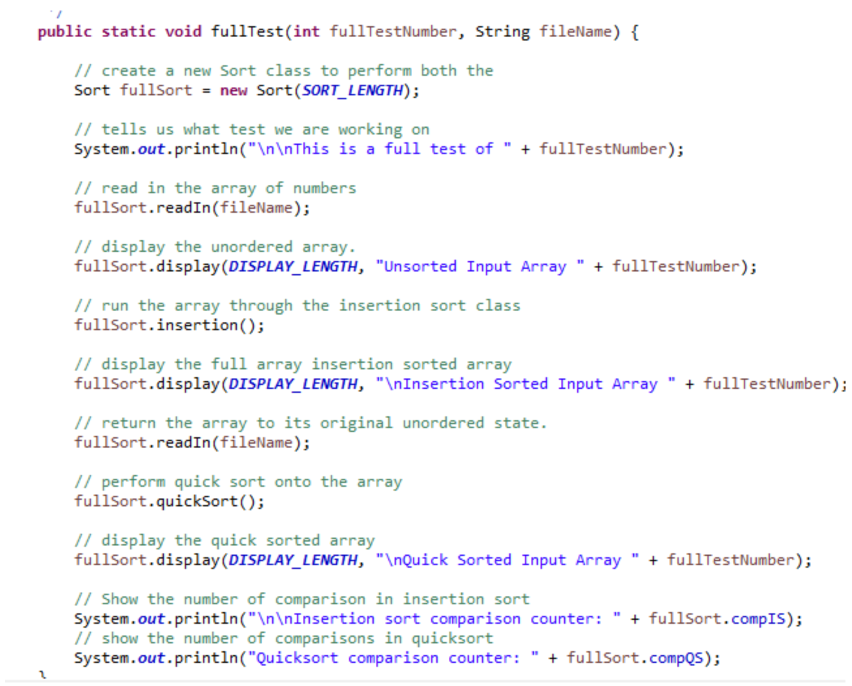
Callum Simpson
B6030326

Testing and Evaluation

This is my testing documentation for the CSC2023: Algorithm Design and Analysis Assignment 1: Implementing and Comparing Sorting Algorithms.

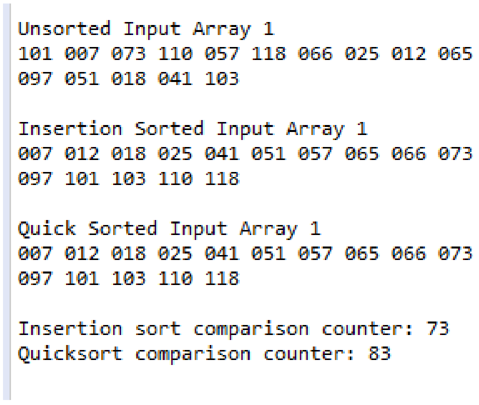
A note about testing



```
public static void fullTest(int fullTestNumber, String fileName) {

    // create a new Sort class to perform both the
    Sort fullSort = new Sort(SORT_LENGTH);

    // tells us what test we are working on
    System.out.println("\n\nThis is a full test of " + fullTestNumber);

    // read in the array of numbers
    fullSort.readIn(fileName);

    // display the unordered array.
    fullSort.display(DISPLAY_LENGTH, "Unsorted Input Array " + fullTestNumber);

    // run the array through the insertion sort class
    fullSort.insertion();

    // display the full array insertion sorted array
    fullSort.display(DISPLAY_LENGTH, "\nInsertion Sorted Input Array " + fullTestNumber);

    // return the array to its original unordered state.
    fullSort.readIn(fileName);

    // perform quick sort onto the array
    fullSort.quickSort();

    // display the quick sorted array
    fullSort.display(DISPLAY_LENGTH, "\nQuick Sorted Input Array " + fullTestNumber);

    // Show the number of comparison in insertion sort
    System.out.println("\n\nInsertion sort comparison counter: " + fullSort.compIS);
    // show the number of comparisons in quicksort
    System.out.println("Quicksort comparison counter: " + fullSort.compQS);
}
```

The code I use for my testing will follow the same structure as this. I will read in the text file and then display the unorganised input array in columns of 10 (that is what I set DISPLAY_LENGTH to be equal to). After that, I perform the insertion sort method with the read in values and then display the insertion sorted the array. The text file will be read in again, this is to make sure that the array returns to its original unsorted state. After that, I will perform the quick sort method/algorithm onto the unsorted array and then display the quick sorted array. When that is done, my code will show the Insertion sort comparison and then the Quicksort comparison.

Test 1

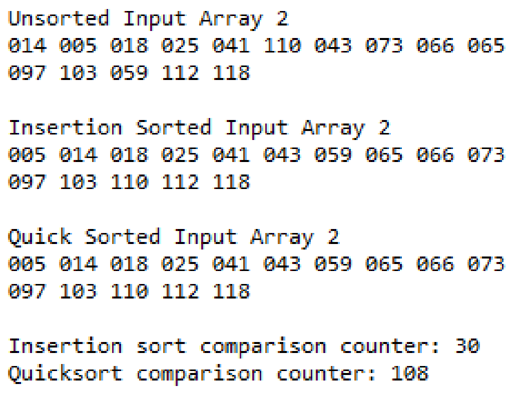


```
Unsorted Input Array 1
101 007 073 110 057 118 066 025 012 065
097 051 018 041 103

Insertion Sorted Input Array 1
007 012 018 025 041 051 057 065 066 073
097 101 103 110 118

Quick Sorted Input Array 1
007 012 018 025 041 051 057 065 066 073
097 101 103 110 118

Insertion sort comparison counter: 73
Quicksort comparison counter: 83
```

Callum Simpson
B6030326

This is the results for the unsorted input array 1. As you can see this is a small and clearly a very unordered array. As you can see in Insertion sort there were only 73 comparisons whereas with Quicksort there were 83 comparisons.

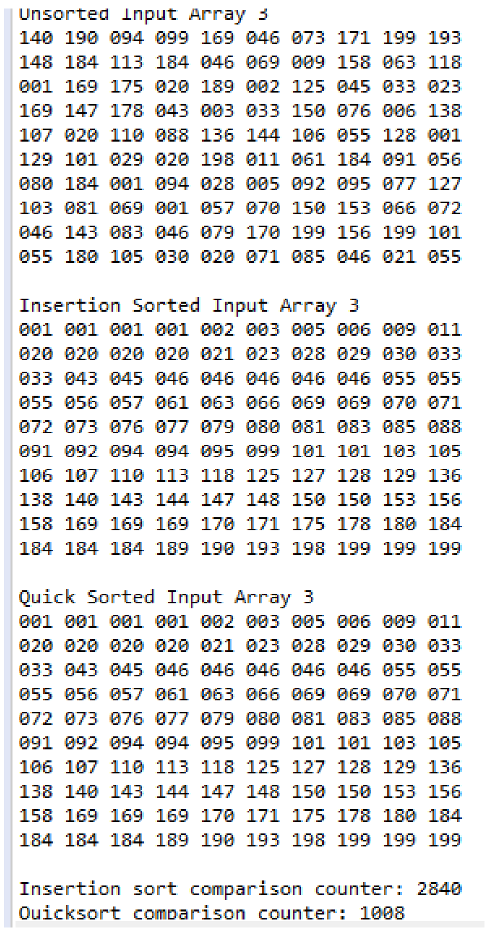
Test 2



```
Unsorted Input Array 2
014 005 018 025 041 110 043 073 066 065
097 103 059 112 118

Insertion Sorted Input Array 2
005 014 018 025 041 043 059 065 066 073
097 103 110 112 118

Quick Sorted Input Array 2
005 014 018 025 041 043 059 065 066 073
097 103 110 112 118

Insertion sort comparison counter: 30
Quicksort comparison counter: 108
```

These results are from unsorted input array 2. As you can see this is small, however, this array is close to being sorted, with a few values already being in the correct place. Insertion sort had 30 comparisons whereas Quick sort had 108 comparisons.

Test 3

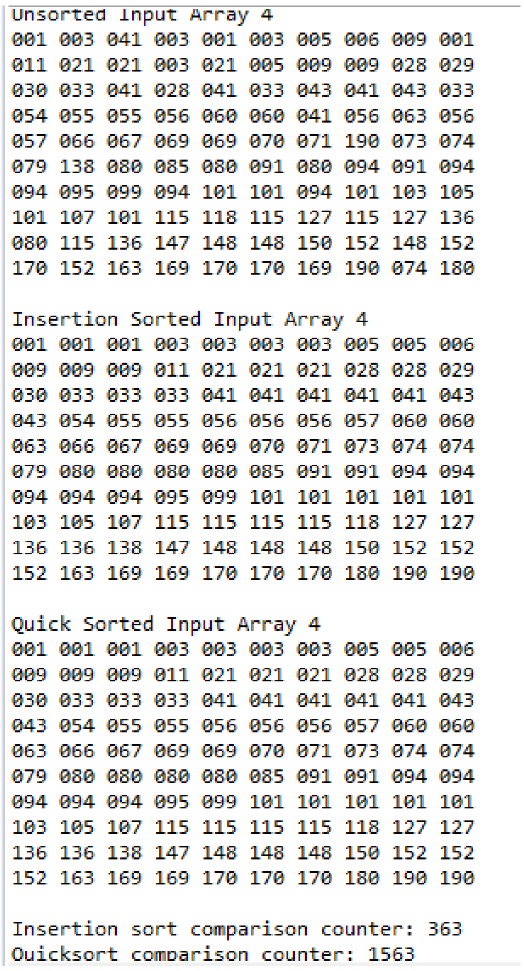


```
Unsorted Input Array 3
140 190 094 099 169 046 073 171 199 193
148 184 113 184 046 069 009 158 063 118
001 169 175 020 189 002 125 045 033 023
169 147 178 043 003 033 150 076 006 138
107 020 110 088 136 144 106 055 128 001
129 101 029 020 198 011 061 184 091 056
080 184 001 094 028 005 092 095 077 127
103 081 069 001 057 070 150 153 066 072
046 143 083 046 079 170 199 156 199 101
055 180 105 030 020 071 085 046 021 055

Insertion Sorted Input Array 3
001 001 001 001 002 003 005 006 009 011
020 020 020 020 021 023 028 029 030 033
033 043 045 046 046 046 046 046 055 055
055 056 057 061 063 066 069 069 070 071
072 073 076 077 079 080 081 083 085 088
091 092 094 094 095 099 101 101 103 105
106 107 110 113 118 125 127 128 129 136
138 140 143 144 147 148 150 150 153 156
158 169 169 169 170 171 175 178 180 184
184 184 184 189 190 193 198 199 199 199

Quick Sorted Input Array 3
001 001 001 001 002 003 005 006 009 011
020 020 020 020 021 023 028 029 030 033
033 043 045 046 046 046 046 046 055 055
055 056 057 061 063 066 069 069 070 071
072 073 076 077 079 080 081 083 085 088
091 092 094 094 095 099 101 101 103 105
106 107 110 113 118 125 127 128 129 136
138 140 143 144 147 148 150 150 153 156
158 169 169 169 170 171 175 178 180 184
184 184 184 189 190 193 198 199 199 199

Insertion sort comparison counter: 2840
Quicksort comparison counter: 1008
```

Callum Simpson
B6030326

These results are from unsorted input array 3. This array is a lot larger than the last two sets of numbers. Also, it would be safe to say that input array 3 is unordered. During the Insertion sort there were 2840 comparisons and during Quicksort, there were 1008 comparisons.

Test 4



```
Unsorted Input Array 4
001 003 041 003 001 003 005 006 009 001
011 021 021 003 021 005 009 009 028 029
030 033 041 028 041 033 043 041 043 033
054 055 055 056 060 060 041 056 063 056
057 066 067 069 069 070 071 190 073 074
079 138 080 085 080 091 080 094 091 094
094 095 099 094 101 101 094 101 103 105
101 107 101 115 118 115 127 115 127 136
080 115 136 147 148 148 150 152 148 152
170 152 163 169 170 170 169 190 074 180

Insertion Sorted Input Array 4
001 001 001 003 003 003 003 005 005 006
009 009 009 011 021 021 021 028 028 029
030 033 033 033 041 041 041 041 041 043
043 054 055 055 056 056 056 057 060 060
063 066 067 069 069 070 071 073 074 074
079 080 080 080 080 085 091 091 094 094
094 094 094 095 099 101 101 101 101 101
103 105 107 115 115 115 115 118 127 127
136 136 138 147 148 148 148 150 152 152
152 163 169 169 170 170 170 180 190 190

Quick Sorted Input Array 4
001 001 001 003 003 003 003 005 005 006
009 009 009 011 021 021 021 028 028 029
030 033 033 033 041 041 041 041 041 043
043 054 055 055 056 056 056 057 060 060
063 066 067 069 069 070 071 073 074 074
079 080 080 080 080 085 091 091 094 094
094 094 094 095 099 101 101 101 101 101
103 105 107 115 115 115 115 118 127 127
136 136 138 147 148 148 148 150 152 152
152 163 169 169 170 170 170 180 190 190

Insertion sort comparison counter: 363
Quicksort comparison counter: 1563
```

These results are from unsorted input array 4. This is a large array that is relatively sorted. During the Insertion sort there were 363 comparisons and during Quicksort, there were 153 comparisons.

Callum Simpson
B6030326

Over-all conclusion

Array	Noticeable features	Insertion	Quicksort
1	small unorder	73	83
2	small, relatively ordered	30	108
3	Large unorder	2840	1008
4	Large relatively ordered	363	1563

From these results, we can conclude a few things.

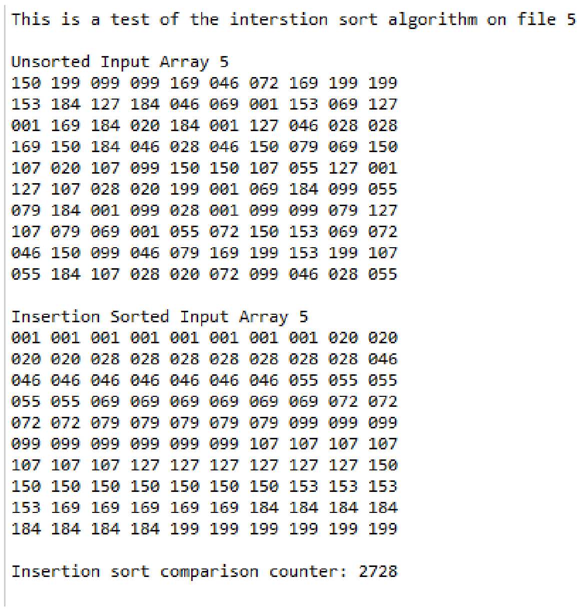
1. If an array is nearly ordered, then Insertion sort would be the best option. We can see this in tests 2 and test 4. These arrays where mostly sorted or needed very little work done on them. As you can see Insertion completed the search in a lot fewer comparisons than quicksort, this is shown practically well in the results for test 4.
2. When it comes to large unordered arrays Quicksort is better. This can be shown with the results from Array 3. However, this is not true if the array is relatively ordered, in that case, Insertion sort will be the better/faster option.
3. It would be safe to say that Insertion arrays work a lot better than quicksort when it comes to arrays that are quite small. This is shown in test 1 and 2, both are small arrays in which the insertion sort has outperformed quicksort.
4. On the flip side of point 3, it would also be safe to say that quicksort is most likely not the best option for small arrays, no matter if it's near sorted or totally random. We can see this in the results of test 1 and test 2. In both quicksort took longer to sort the array than insertion sort did.
5. From what we can see the best case and worst case big oh vary wildly for Insertion due to the huge differences in between the comparison values. In quicksort big oh for worst case and best case do not have that big of a split.

Continue down for further testing (task 5)

Callum Simpson
B6030326

Further testing

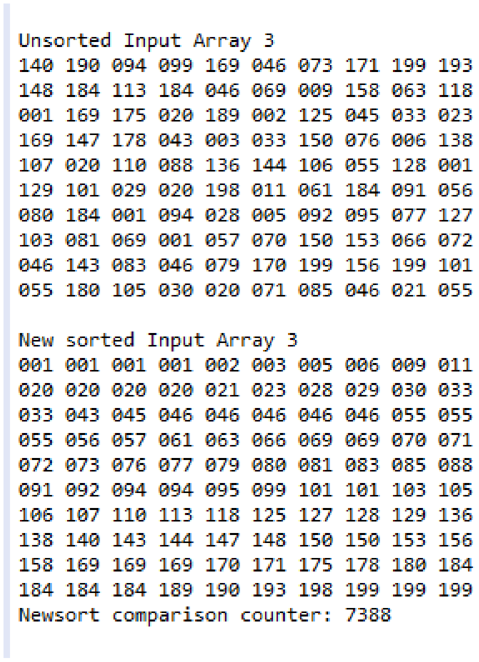
First, I need to perform integration search on the test5.



```
This is a test of the interstion sort algorithm on file 5

Unsorted Input Array 5
150 199 099 099 169 046 072 169 199 199
153 184 127 184 046 069 001 153 069 127
001 169 184 020 184 001 127 046 028 028
169 150 184 046 028 046 150 079 069 150
107 020 107 099 150 150 107 055 127 001
127 107 028 020 199 001 069 184 099 055
079 184 001 099 028 001 099 099 079 127
107 079 069 001 055 072 150 153 069 072
046 150 099 046 079 169 199 153 199 107
055 184 107 028 020 072 099 046 028 055

Insertion Sorted Input Array 5
001 001 001 001 001 001 001 001 020 020
020 020 028 028 028 028 028 028 028 046
046 046 046 046 046 046 046 055 055 055
055 055 069 069 069 069 069 069 072 072
072 072 079 079 079 079 079 099 099 099
099 099 099 099 099 099 107 107 107 107
107 107 107 127 127 127 127 127 127 150
150 150 150 150 150 150 150 153 153 153
153 169 169 169 169 169 184 184 184 184
184 184 184 184 199 199 199 199 199 199

Insertion sort comparison counter: 2728
```

It took insertion sort 2728 comparisons to sort the array. What should be noted by test5 is that it is a large array containing many duplicate numbers.

New sort of test3

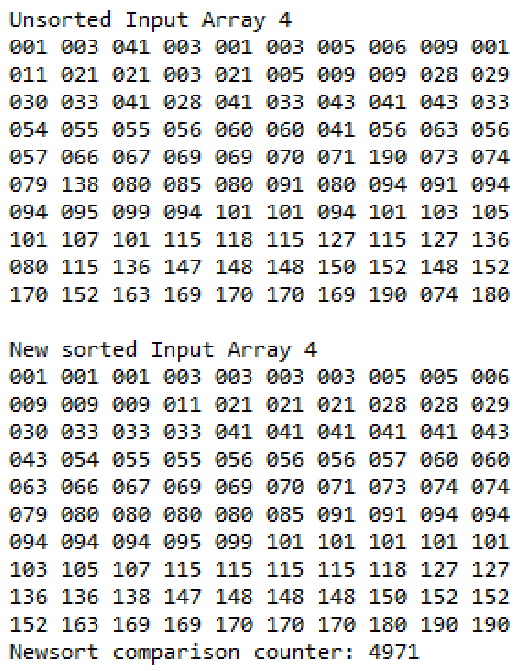


```
Unsorted Input Array 3
140 190 094 099 169 046 073 171 199 193
148 184 113 184 046 069 009 158 063 118
001 169 175 020 189 002 125 045 033 023
169 147 178 043 003 033 150 076 006 138
107 020 110 088 136 144 106 055 128 001
129 101 029 020 198 011 061 184 091 056
080 184 001 094 028 005 092 095 077 127
103 081 069 001 057 070 150 153 066 072
046 143 083 046 079 170 199 156 199 101
055 180 105 030 020 071 085 046 021 055

New sorted Input Array 3
001 001 001 001 002 003 005 006 009 011
020 020 020 020 021 023 028 029 030 033
033 043 045 046 046 046 046 046 055 055
055 056 057 061 063 066 069 069 070 071
072 073 076 077 079 080 081 083 085 088
091 092 094 094 095 099 101 101 103 105
106 107 110 113 118 125 127 128 129 136
138 140 143 144 147 148 150 150 153 156
158 169 169 169 170 171 175 178 180 184
184 184 184 189 190 193 198 199 199 199
Newsort comparison counter: 7388
```

This is the results I get when I run test3.txt through the New sort algorithm. There was a total of 7388 comparisons.

Callum Simpson
B6030326

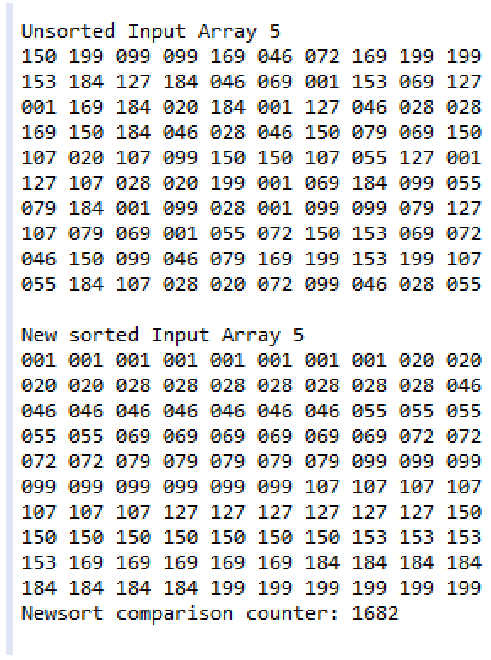
New sort of test4



```
Unsorted Input Array 4
001 003 041 003 001 003 005 006 009 001
011 021 021 003 021 005 009 009 028 029
030 033 041 028 041 033 043 041 043 033
054 055 055 056 060 060 041 056 063 056
057 066 067 069 069 070 071 190 073 074
079 138 080 085 080 091 080 094 091 094
094 095 099 094 101 101 094 101 103 105
101 107 101 115 118 115 127 115 127 136
080 115 136 147 148 148 150 152 148 152
170 152 163 169 170 170 169 190 074 180

New sorted Input Array 4
001 001 001 003 003 003 003 005 005 006
009 009 009 011 021 021 021 028 028 029
030 033 033 033 041 041 041 041 041 043
043 054 055 055 056 056 056 057 060 060
063 066 067 069 069 070 071 073 074 074
079 080 080 080 080 085 091 091 094 094
094 094 094 095 099 101 101 101 101 101
103 105 107 115 115 115 115 118 127 127
136 136 138 147 148 148 148 150 152 152
152 163 169 169 170 170 170 180 190 190
Newsort comparison counter: 4971
```

This is the results I get when I run test4.txt through the New sort algorithm. There was a total of 4971 comparisons.

New sort of test5



```
Unsorted Input Array 5
150 199 099 099 169 046 072 169 199 199
153 184 127 184 046 069 001 153 069 127
001 169 184 020 184 001 127 046 028 028
169 150 184 046 028 046 150 079 069 150
107 020 107 099 150 150 107 055 127 001
127 107 028 020 199 001 069 184 099 055
079 184 001 099 028 001 099 099 079 127
107 079 069 001 055 072 150 153 069 072
046 150 099 046 079 169 199 153 199 107
055 184 107 028 020 072 099 046 028 055

New sorted Input Array 5
001 001 001 001 001 001 001 001 020 020
020 020 028 028 028 028 028 028 028 046
046 046 046 046 046 046 046 055 055 055
055 055 069 069 069 069 069 069 072 072
072 072 079 079 079 079 079 099 099 099
099 099 099 099 099 099 107 107 107 107
107 107 107 127 127 127 127 127 127 150
150 150 150 150 150 150 150 153 153 153
153 169 169 169 169 169 184 184 184 184
184 184 184 184 199 199 199 199 199 199
Newsort comparison counter: 1682
```

This is the results I get when I run test5.txt through the New sort algorithm. There was a total of 1682 comparisons.

Callum Simpson
B6030326

Test	Noticeable features	Insertion	New sort
3	Large unorder	2840	7388
4	Large relatively ordered	363	4971
5	Large unorder many duplicates	2728	1682

a) In which of your test(s) does the new sorting algorithm perform better than insertion sort? State why you think this happens.

The only test where New sort performs better than Insertion sort is in test 5. As all the tests have an array size of 100 we can safely say that it has nothing to do with size. As both test 3 and test 5 are both unordered it's safe to say that it's not that either. That means it's likely that new sort is performing better in test 5 due to its high number of duplicates.

b) What is the worst case big-Oh performance for the new sorting algorithm and when does this occur?

The worst case for new sort is when there are no duplicates. This leads the big O being O(n^2), (n squared). This is because the Array must compare every value in the array to see if there are any duplicates in the array, which there won't be. I believe this occurs in the New sort class

c) What is the best case big-Oh performance for the new sorting algorithm and when does this occur?

The best case for this algorithm will be when every value in the array is the is the same value. For example, an array of seven 2s. This would mean the big O performance is 0(N). The reason for this is because every time I increased the number of values in the array by 1 the number of comparisons will increase by 2. As the comparison are increasing at a constant rate we can say that the performance is growing linearly, meaning it must be big O(N).