Final Project

ENSF 594

Ziad Chemali

UCID: 10109966

Date: 17/08/2020

# I] Exercise 1:

### 1) Experimental method:

For each of the orders ascending, descending, and random order different sizes of arrays were uses (10, 100, 1000, 10,000, 100,000, and 1,000,000), then all four of the sorting algorithms (bubble ,insertion, merge, and quick sort) were tested and the result time is written to an output file

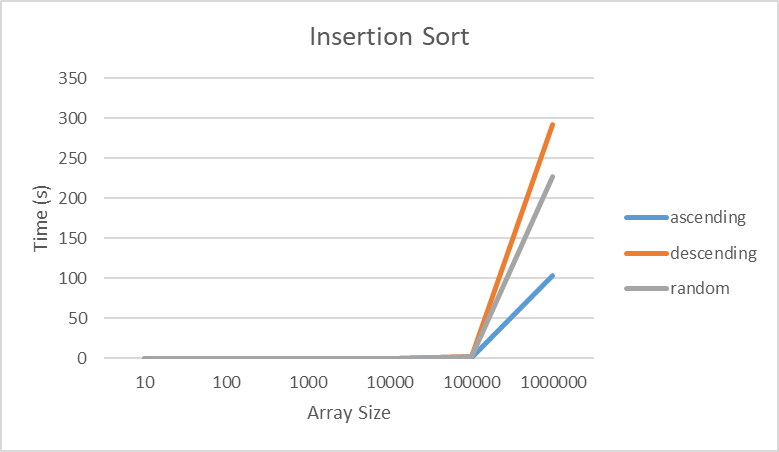
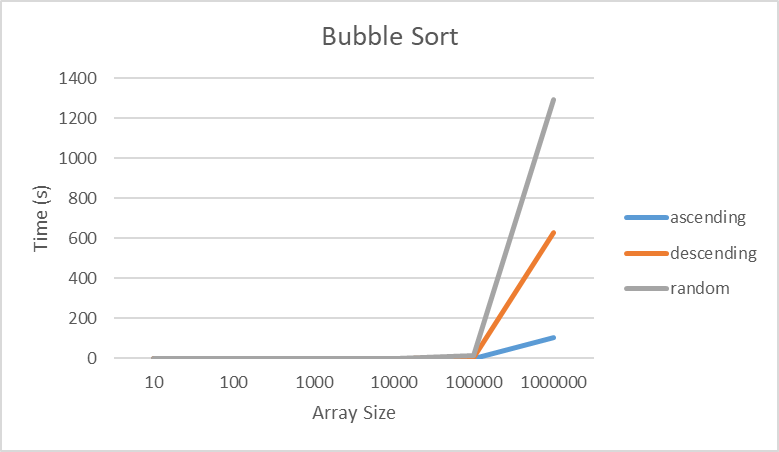
### 2) Data Collected:

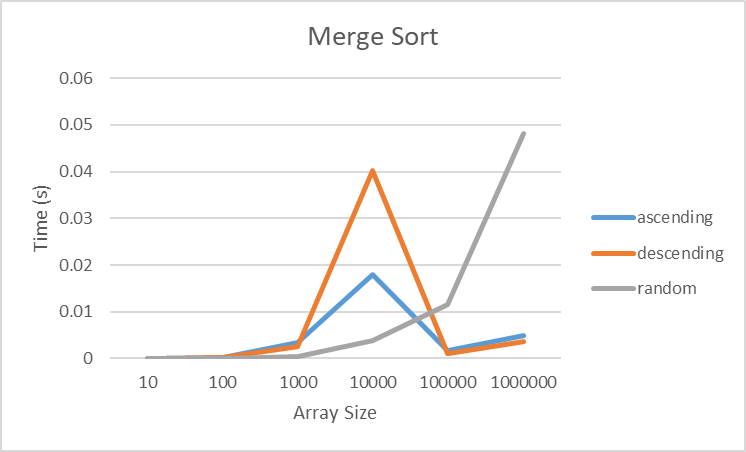
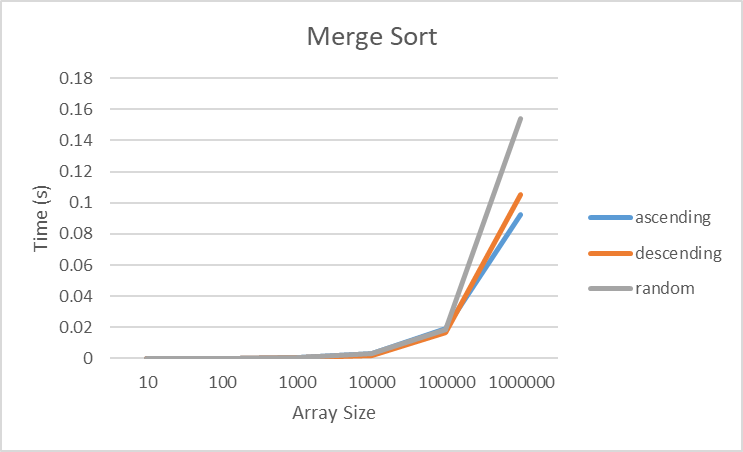
All the cases in experimental method were studied and the following table is generated:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Size | Algorithm | Ascending | Descending | Random |
| 10 | bubble | 8.30E-06 | 3.70E-06 | 3.60E-06 |
| 10 | insertion | 3.00E-06 | 3.50E-06 | 3.50E-06 |
| 10 | merge | 8.40E-06 | 8.30E-06 | 9.00E-06 |
| 10 | quick | 1.46E-06 | 6.00E-06 | 5.80E-06 |
| 100 | bubble | 1.07E-04 | 2.42E-04 | 1.35E-04 |
| 100 | insertion | 1.17E-04 | 1.25E-04 | 7.26E-05 |
| 100 | merge | 5.52E-05 | 5.33E-05 | 5.88E-05 |
| 100 | quick | 1.87E-04 | 1.74E-04 | 3.74E-05 |
| 1000 | bubble | 2.28E-03 | 3.39E-03 | 9.90E-03 |
| 1000 | insertion | 1.74E-03 | 5.01E-03 | 2.35E-03 |
| 1000 | merge | 5.83E-04 | 5.80E-04 | 6.53E-04 |
| 1000 | quick | 3.44E-03 | 2.45E-03 | 3.44E-04 |
| 10,000 | bubble | 1.45E-02 | 7.74E-02 | 1.13E-01 |
| 10,000 | insertion | 1.27E-02 | 3.49E-02 | 3.04E-02 |
| 10,000 | merge | 3.05E-03 | 2.00E-03 | 3.28E-03 |
| 10,000 | quick | 1.79E-02 | 4.01E-02 | 3.78E-03 |
| 100,000 | bubble | 1.07E+00 | 6.26E+00 | 1.28E+01 |
| 100,000 | insertion | 9.64E-01 | 2.92E+00 | 2.34E+00 |
| 100,000 | merge | 1.92E-02 | 1.66E-02 | 1.86E-02 |
| 100,000 | quick | 1.67E-03 | 1.09E-03 | 1.15E-02 |
| 1,000,000 | bubble | 1.05E+02 | 6.28E+02 | 1.29E+03 |
| 1,000,000 | insertion | 1.04E+02 | 2.92E+02 | 2.28E+02 |
| 1,000,000 | merge | 9.25E-02 | 1.06E-01 | 1.54E-01 |
| 1,000,000 | quick | 4.79E-03 | 3.60E-03 | 4.81E-02 |

### 3) Data Analysis:

The following graphs are obtained from the table above:





**Observation 1:**

The first observation is that as the size of array increase the running time of bubble sort drastically increase (see graph 1). For an array size of 1million the running time using bubble can take up to 22 min (random order)

The insertion sort seems to work somewhat well for array size of 1 million. It took ~5min to sort (Descending order)

Even though the complexity of bubble sort and insertion sort is the same O(n^2) this does not mean that they have the same running time.

**Observation 2:**

Another observation, all sorting algorithm were doing well for an array size less than or equal to **100**,**000,** it is safe to use any of the four algorithms is array size is less that or equal to **100,000**

**Observation 3:**

Merge/Quick sort are best used for arrays with sizes greater than **100,000**

### 4) Complexity Analysis:

#### i) Bubble Sort:

|  |  |
| --- | --- |
| Cost | Times |
| C1 | 1 |
| C2 | n(outer loop) |
| C3 | n\*n (nested) |
| C4 | n(inner loop) |
| C5 | n(inner loop) |
| C6 | n(inner loop) |
| C7 | n(inner loop) |
|  | |
| C8 | n(outer loop) |

C8 n

public void bubbleSort(int[] array) {

int n = array.length;--------------------🡪

while (n > 0) {--------------------------🡪

for (int i = 0; i < n - 1; i++) {-🡪

if (array[i] > array[i + 1]) {

int temp = array[i];

array[i] = array[i + 1];

array[i + 1] = temp;

}

}

n--;-----------------------------🡪

}

}

T(n)=cn2+cn+1 which is equivalent to O(n2)

#### ii) Insertion Sort:

public void insertionSort(int[] array) {

|  |  |
| --- | --- |
| Cost | Times |
| C1 | n(outer loop) |
| C2 | n(outer loop) |
| C3 | n\*n (nested) |
| C4 | n(inner loop) |
| C5 | n(inner loop) |
| C6 | n(inner loop) |
|  |  |
|  |  |

C8 n

for (int i = 1; i < array.length; i++) {-🡪

int temp = array[i];--------------🡪

for (int j = i - 1; j >= 0; j--) {------🡪

if (array[j] > temp) {

array[j + 1] = array[j];

array[j] = temp;

}

}

}

}

T(n)=cn2+cn which is equivalent to O(n2)

#### ii) Merge Sort:

The merge sort has two function, merge () and mergeSort (). mergeSort () invokes merge () so first analyze the complexity of merge ().

|  |  |
| --- | --- |
| Cost | Times |
| C1 | 1 |
| C2 | 1 |
|  |  |
| C3 | n |
| C4 | n |
|  |  |
| C5 | n |
| C6 | N |
|  |  |
|  |  |
| C7 | 1 |
| C8 | 1 |
|  |  |
| C9 | n |
| C10 | n |
| C11 | n |
| C12 | n |
|  |  |
| C13 | n |
| C14 | n |
| C16 | n |

public void merge(int[] array, int l, int m, int r) {

int[] leftArray = new int[m - l + 1];------------🡪

int[] rightArray = new int[r - m];---------------🡪

// Filling left arary

for (int i = 0; i < leftArray.length; i++) {---🡪

leftArray[i] = array[l + i];

}

for (int i = 0; i < rightArray.length; i++) {

rightArray[i] = array[m + 1 + i];

}

// compare and merge

int i = 0, j = 0;-------------------------------🡪

int k = l;// actual array

while (i < leftArray.length && j < rightArray.length) {

if (leftArray[i] <= rightArray[j]) {----🡪

array[k] = leftArray[i];----------🡪

i++;-----------------------------🡪

} else {

array[k] = rightArray[j];---------🡪

j++;

}

k++;

}

The merge function has complexity T(n)=cn+c ⬄ O(n)

|  |  |
| --- | --- |
| Cost | Times |
|  |  |
| C1 | 1 |
| C2 | 1 |
| C3 | T(n/2) |
| C4 | T(n/2) |
| C5 | O(n) |

public void mergeSort(int[] array, int l, int r) {

if (r > l) {------------------------🡪

int m = (l + r) / 2;---------🡪

mergeSort(array, l, m -----🡪

mergeSort(array, m + 1, r);

merge(array, l, m, r);

}

}

T(n)=2T(n/2)+O(n)+c

Big O using master theorem is O(nlogn)

#### iv) Quick Sort:

Quick sort has two functions quicksort() and partition(). quicksort () invokes partition , so first analyze the complexity of partition function

|  |  |
| --- | --- |
| Cost | Times |
| C1 | 1 |
| C2 | 1 |
| C3 | n |
| C4 | n |
| C5 | n |
| C6 | n |
| C7 | n |
| C8 | n |

public int partition(int[] array, int low, int high) {

double pivot = array[high];--------------🡪

int i = low;-----------------------------🡪

for (int j = low; j < high; j++) {

if (array[j] <= pivot) {--------🡪

int temp = array[j];------🡪

array[j] = array[i];------🡪

array[i] = temp;----------🡪

i++;----------------------🡪

}

}

The partition function has complexity T(n)=cn+c ⬄ O(n)

|  |  |
| --- | --- |
| Cost | Times |
|  |  |
| C1 | 1 |
| C2 | T(n) |
| C3 | 1 |
|  |  |
| C5 | T(n/2) |
| C6 | T(n/2) |

public void quickSort(int[] array, int low, int high) {

if (low < high) {--------------------------🡪

int index = partition(array, low, high);

if (index > 0 && (index) < high) {--🡪

quickSort(array, low, index - 1);

quickSort(array, index + 1, high);

}

}

}

T(n)=2T(n/2)+T(n)=2T(n/2)+O(n)+c

So, the complexity of the quick sort using master theorem is O(nlogn)

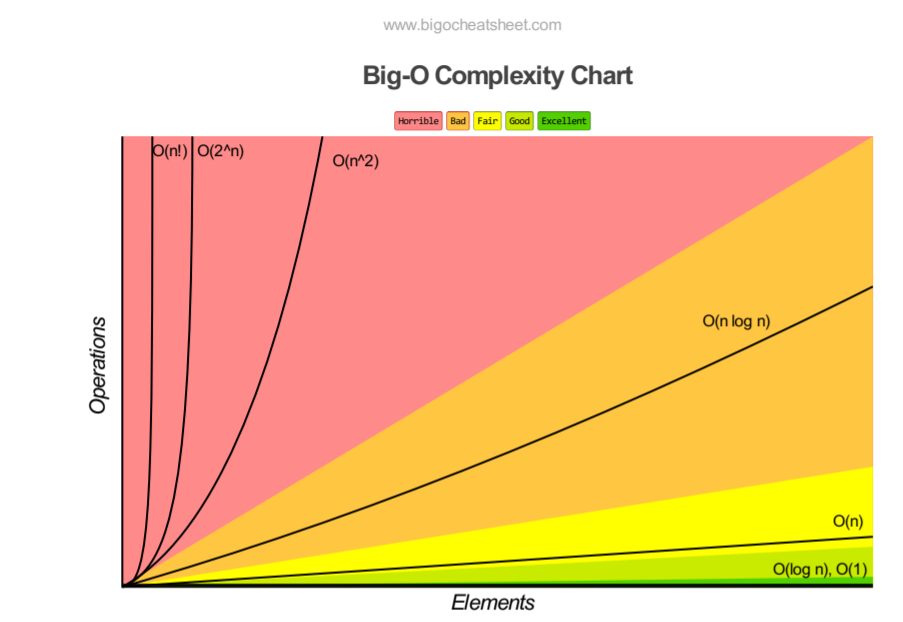
### 5) Interpretation:

**Visualizing the time complexity:**

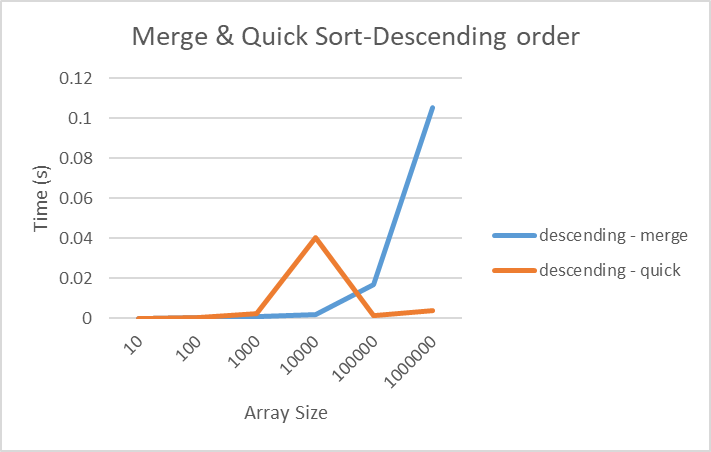
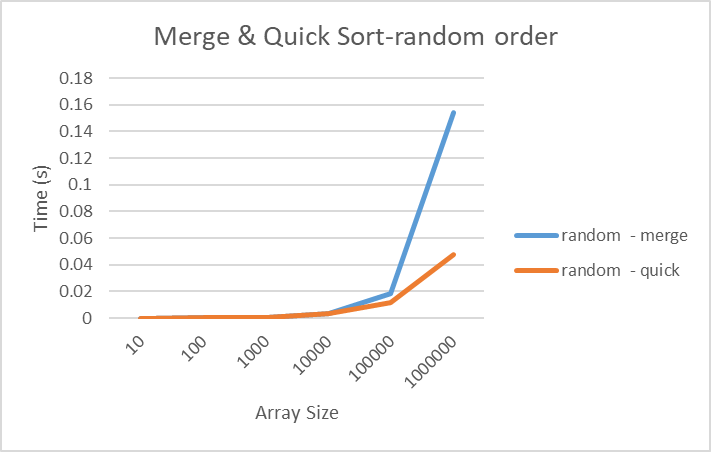
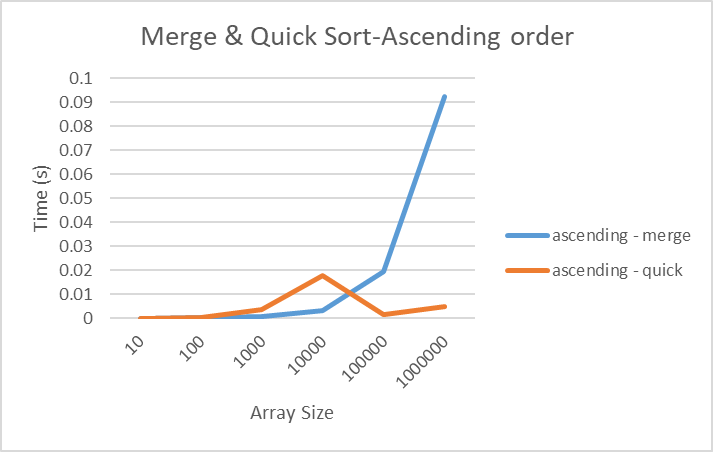
Both Quick &merge Sort has O(nlogn) complexity

Both Bubble& Insertion sort has O(n2) complexity

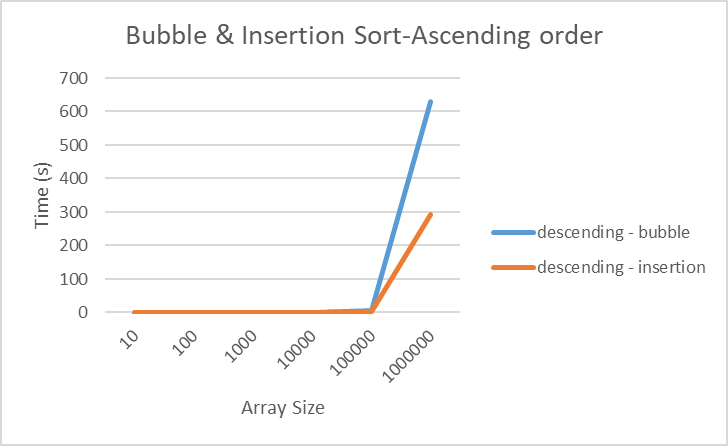
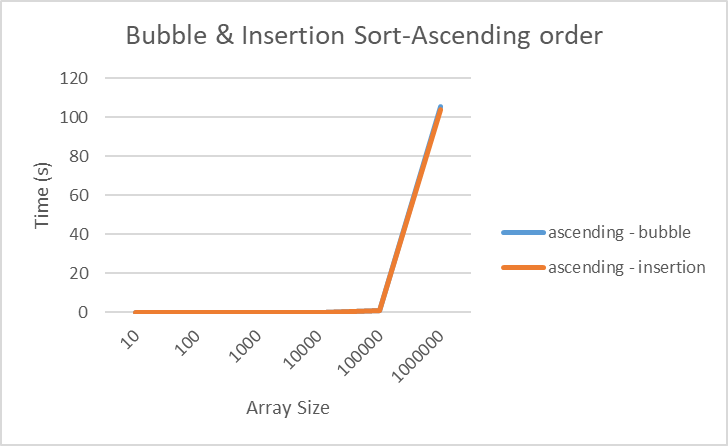
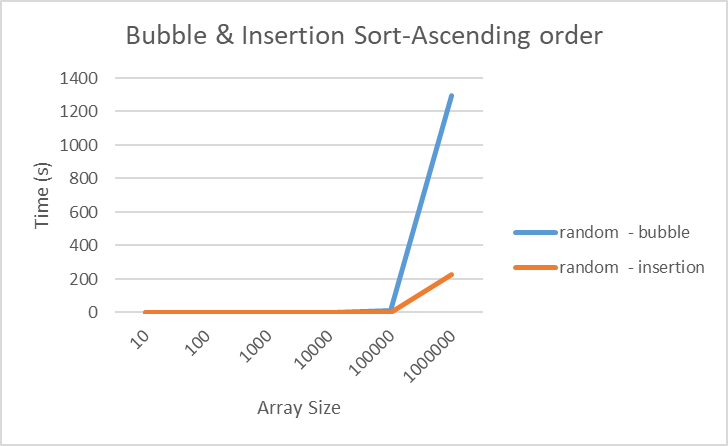
As seen in the complexity chart below O(nlogn) is much faster than O(n2)



**Comparing sort algorithms with same complexity:**



Both merge and quicksort have complexity of O(nlogn); however, it seems that quick sort is a bit faster than merge sort.



Both Bubble and insertion sort have time complexity of O(n2), insertion sort is faster than bubble sort even though they have same complexity.

**Order of input analysis:**

The worst case for each of the array sizes seem to differ, the heat map below shows in red which one has the longest run time per array size



So each algorithm at different array size has different worse case, in other words the order of input affects the running time of each algorithm but it is not obvious which filling order has the longest running time since it keeps changing as the array size changes.

### 6) Conclusion:

For arrays of size less than 100,000 it is best to use Bubble and insertion sort algorithms, but for arrays greater than 100,000 it is best to use quick and merger sort algorithms. The input array order has an affect on the running time of the sorting algorithm; however, it is not obvious which order method (ascending, random, descending) might slow down the sorting algorithm.

Finally, even though some algorithms have same time complexity their running time might differ , this might be due to space complexity .

# I] Exercise 2:

### 1) Complexity Analysis:

In my program both checking for anagrams and reading input file happen in the same function readFile()

public void readFile(DynamicArray da, String inputfile) {

try {

BufferedReader reader = new BufferedReader(new FileReader(inputfile));

String word;----------------------------------🡪

System.out.println("Reading File/creating list of anagrams...");

while ((word = reader.readLine()) != null) {->

/// converting the word to a key

char[] temp = new char[word.length()];🡪

for (int i = 0; i < temp.length; i++) {->

temp[i] = word.charAt(i);-----🡪

}

insertionSort(temp);-------------🡪

StringBuffer sb = new StringBuffer();

for (int i = 0; i < temp.length; i++) {

sb.append(temp[i]);

}

String key = sb.toString();------------🡪

// checking if key exists in dynamic array

int i = da.linearSearch(key);--------🡪

// if key exist then append the word to the linked list

if (i >= 0) {-------------------------🡪

da.arrayLinked[i].insertAtEnd(new Node(word));

}

// if key doesnt exist then add it to the dynamicArray and create a new linked

// list

else {

LinkedList ll1 = new LinkedList(key);-🡪

ll1.insertAtEnd(new Node(word));->

da.add(ll1);------------------🡪

}

}

} catch (Exception e) {

System.err.println(e);

}

}

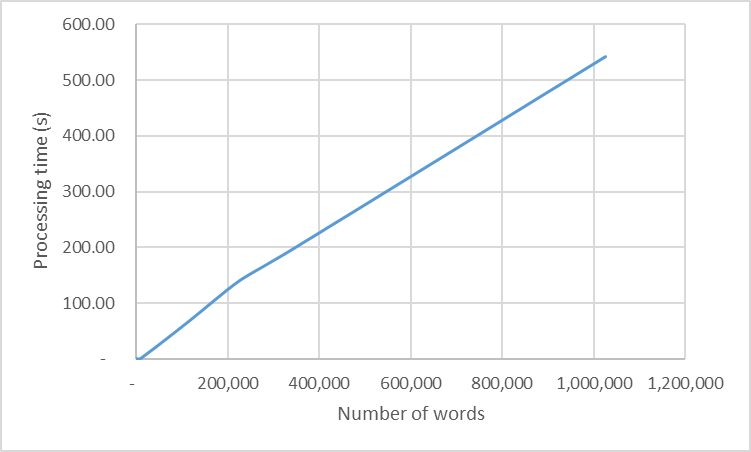
|  |  |
| --- | --- |
| Cost | Time |
|  |  |
|  |  |
| C1 | 1 |
| C2 | 1 |
| C3 | 1 |
| C4 | n |
| C5 | n |
| C6 | n\*L(nested) |
| C7 | n\*L(nested) |
| C8 | n\*L2(insertion sort of char list) |
| C9 | n |
| C10 | n\*L(nested) |
| C11 | n\*L(nested) |
|  |  |
| C12 | n |
|  |  |
|  |  |
| C13 | n\*n2(linear search of word in array) |
|  |  |
| C14 | n |
| C15 | n |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
| C16 | n |
| C17 | n |
| C18 | n\*n(add() has for loop) |
|  |  |

The time complexity of this function is in the form of T(n)=cn3+cnL2+cnL++cn2+ cn+c. that number of words has dominance then the Big O notation is O(n3), this is due to the linear search in the function embedded inside the while loop. A quick fix would be, use merge or quick sort instead of insertion sort then the complexity would decrease

### 2) Data Collected/Analysis:

Different number of words is used to measure the process time of readFile() function. The following table is obtained.

|  |  |
| --- | --- |
| Number of words in inputfile | Time to process (s) |
| 1,000 | 0.01 |
| 10,000 | 0.33 |
| 113,809 | 66.10 |
| 227,618 | 140.91 |
| 341,427 | 196.27 |
| 1,024,281 | 541.53 |



As the number of words in input file increase the running time starts to increase drastically which tell us that this complexity is O(n3)