

Assignment 1

Algorithm Design and Analysis

Student: Doriela Grabocka

Contents

[**Performance and simulation** 2](#_Toc38922743)

[**Input and Output** 2](#_Toc38922744)

[**Sorting Algorithms** 2](#_Toc38922745)

[**Heapsort Algorithm and Pseudocode** 2](#_Toc38922746)

[**Mergesort Algorithm and Pseudocode** 5](#_Toc38922747)

[**Quicksort Algorithm and Pseudocode** 8](#_Toc38922748)

[**Recorded Data** 11](#_Toc38922749)

[**InsMergesort Algorithm and Pseudocode** 14](#_Toc38922750)

[Actual value of M and number of comparisons 16](#_Toc38922751)

[General solution for value of M and time complexity of the algorithm 17](#_Toc38922752)

[Correctness of this algorithm 19](#_Toc38922753)

[**Bibliography** 20](#_Toc38922754)

# **Performance and simulation:**

## **Input and Output:**

On the tables below, ([Data Recorded Section](#_Recorded_Data:)), I have provided all the outputs of my algorithms.  
My domain was made of 65 (13x5 files per input size) files in total with sizes varying from 10 to 100000 array elements.  
More precisely my domain of input sizes was:

D={10,20,50,100,200,500,1000,2000,5000,10000,20000,50000,100000}

For each of the input sizes I have created 5 different files containing random integers using a SecureRandom object and its method nextInt() in order to produce cryptographically strong numbers.

At the end, I output the total number of comparisons performed in each algorithm.

# **Sorting Algorithms**

For each of the problems I used the algorithms described on the books and lecture notes.

**Heapsort Algorithm and Pseudocode[[1]](#footnote-1):**

My implementation is in-place. I have based my algorithm and implementation on the Heapsort algorithm provided in “*Introduction to Algorithms*” book (Cormen, Heapsort, 2009).It time complexity is O(nlog2n) I build a maximum heap data structure in order to be able to sort the elements in ascending order. I build the heap first and then delete the maximums repeatedly using a for loop. I used an in-place implementation by putting the deleted maximums at the end of the array whose maximums I am deleting. I built the heap by starting from the middle of the array index and moving up to the start position to check for all the nodes its children. I start from the middle index since in a heap data structure all elements from middle index until the end will be leaves. For a perfect binary heap, the last level will contain half of the elements. Therefore, for each element accessed, I call the insert method starting from the current index and providing the size of the array as the heap size in order to check all children. Inside the insert method, the algorithm first checks if the item at the current index is smaller than its left child, if yes than the largest position will become that of the left child, if not than the largest position will become the current index. Then the algorithm will check the right child against the array element at the largest index (can be the left child or the current element). If the right child is greater than we update again the largest position by setting it to the position of the right child, if not we leave the largest position as it is. Then the algorithm will check whether the largest position is different from the current position that we are accessing. If yes this means that we have found an element larger than our current element and thus we need a recursive call to do the same for the subtree rooted at element largest now. This will go until we wither reach the end of the array or the elements do not violate any of the properties (position==larger).

After building the heap, we need to delete the maximums so that we end up with an array sorted in ascending order. To do so, the algorithm again makes use of the insertion method, but this time it will call it with different argument. For each call, the position argument supplied to the insert method will be 0. This mean that we will start the comparisons from the beginning and going down possibly log2n levels. However, before calling insertion method we have to swap the first element (position 0) that is the maximum, with the last element (position length-1). After doing this we call insertion and supply to it the array, the heap size that will be length – 1 – i (because the space where the children reside will shrink each time we do a delete and elements length-1-i until length-1 will be sorted) and the last argument will be 0 so that we start comparisons from the beginning.

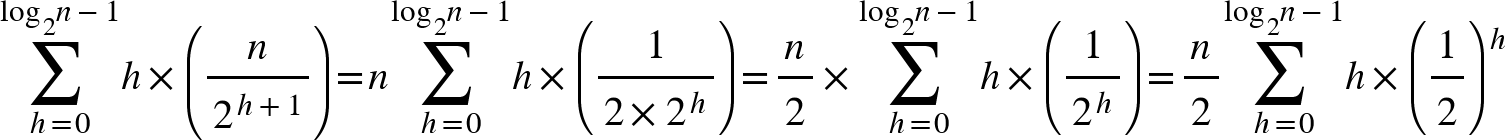
In terms of comparisons made, in delete we do at most 2 comparisons for every element, after checking both the right and the left children. For each delete, we have to check at most log2 n levels and this will happen n times until all maximums are deleted. Therefore, in total there will be 2nlog2n comparisons worst case. Contrary the number of the comparisons made in the procedure are linear. (Heapsort, 2009) When building the heap, we compare each element with all its own children. This means that the number of comparisons for each node at a specific height h[[2]](#footnote-2) will be equivalent to h. So in order to a have a more realistic bound than nlog2n for the number of comparisons made in building a heap, we have to sum up all comparisons made at each height of the heap. The height of H of a heap is:

**H=Depth-1= ceil (log2 (n+1))-1≈ log2n -1=floor (log2n)**

So for a heap of size n=15, the height will be 3. We know that the last level will have at most half of the elements. The last level will have height=0 as well. So generally there will be:

<math xmlns="http://www.w3.org/1998/Math/MathML"><mi>h</mi><mi>e</mi><mi>i</mi><mi>g</mi><mi>h</mi><mi>t</mi><mo>=</mo><mn>0</mn><mo>&#xA0;</mo><mo>-</mo><mo>&gt;</mo><mo>&#xA0;</mo><mi>m</mi><mi>a</mi><mi>x</mi><mo>.</mo><mo>&#xA0;</mo><mi>n</mi><mi>o</mi><mi>d</mi><mi>e</mi><mi>s</mi><mo>=</mo><mfrac><mi>n</mi><mn>2</mn></mfrac><mo>=</mo><mfrac><mi>n</mi><msup><mn>2</mn><mrow><mi>h</mi><mo>+</mo><mn>1</mn></mrow></msup></mfrac><mspace linebreak="newline"/><mi>h</mi><mi>e</mi><mi>i</mi><mi>g</mi><mi>h</mi><mi>t</mi><mo>=</mo><mn>1</mn><mo>&#xA0;</mo><mo>-</mo><mo>&gt;</mo><mi>m</mi><mi>a</mi><mi>x</mi><mo>.</mo><mo>&#xA0;</mo><mi>n</mi><mi>o</mi><mi>d</mi><mi>e</mi><mi>s</mi><mo>=</mo><mfrac><mi>n</mi><mn>4</mn></mfrac><mo>=</mo><mfrac><mi>n</mi><msup><mn>2</mn><mrow><mi>h</mi><mo>+</mo><mn>1</mn></mrow></msup></mfrac><mspace linebreak="newline"/><mi>h</mi><mi>e</mi><mi>i</mi><mi>g</mi><mi>h</mi><mi>t</mi><mo>=</mo><mn>2</mn><mo>&#xA0;</mo><mo>-</mo><mo>&gt;</mo><mi>m</mi><mi>a</mi><mi>x</mi><mo>.</mo><mo>&#xA0;</mo><mi>n</mi><mi>o</mi><mi>d</mi><mi>e</mi><mi>s</mi><mo>=</mo><mfrac><mi>n</mi><mn>8</mn></mfrac><mo>=</mo><mfrac><mi>n</mi><msup><mn>2</mn><mrow><mi>h</mi><mo>+</mo><mn>1</mn></mrow></msup></mfrac><mspace linebreak="newline"/><mo>.</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>.</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>.</mo><mspace linebreak="newline"/><mo>.</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>.</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>.</mo><mspace linebreak="newline"/><mi>h</mi><mi>e</mi><mi>i</mi><mi>g</mi><mi>h</mi><mi>t</mi><mo>=</mo><msub><mi>log</mi><mn>2</mn></msub><mi>n</mi><mo>-</mo><mn>1</mn><mo>&#xA0;</mo><mo>-</mo><mo>&gt;</mo><mo>&#xA0;</mo><mi>m</mi><mi>a</mi><mi>x</mi><mo>.</mo><mi>n</mi><mi>o</mi><mi>d</mi><mi>e</mi><mi>s</mi><mo>=</mo><mpadded lspace="+1px"><mi>c</mi><mi>e</mi><mi>i</mi><mi>l</mi><mfenced><mfrac><mi>n</mi><msup><mn>2</mn><mrow><mi>h</mi><mo>+</mo><mn>1</mn></mrow></msup></mfrac></mfenced></mpadded><mo>=</mo><mi>c</mi><mi>e</mi><mi>i</mi><mi>l</mi><mfenced><mfrac><mi>n</mi><msup><mn>2</mn><mrow><msub><mi>log</mi><mn>2</mn></msub><mi>n</mi><mo>&#xA0;</mo><mo>-</mo><mn>1</mn><mo>+</mo><mn>1</mn></mrow></msup></mfrac></mfenced><mo>=</mo><mi>c</mi><mi>e</mi><mi>i</mi><mi>l</mi><mfenced><mfrac><mi>n</mi><mi>n</mi></mfrac></mfenced><mo>=</mo><mn>1</mn><mo>&#xA0;</mo><mi>r</mi><mi>o</mi><mi>o</mi><mi>t</mi></math>

For each of these nodes we will do h comparisons and everything will become:



This is quite similar to the geometric series with ration ½. Therefore, we can solve this using knowledge from Calculus II:

<math xmlns="http://www.w3.org/1998/Math/MathML"><mfrac><mn>1</mn><mfenced><mrow><mn>1</mn><mo>-</mo><mi>x</mi></mrow></mfenced></mfrac><mo>=</mo><munderover><mo>&#x2211;</mo><mrow><mi>r</mi><mo>=</mo><mn>0</mn></mrow><mo>&#x221E;</mo></munderover><msup><mi>x</mi><mi>r</mi></msup><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mi>d</mi><mi>i</mi><mi>f</mi><mi>f</mi><mi>e</mi><mi>r</mi><mi>e</mi><mi>n</mi><mi>t</mi><mi>i</mi><mi>a</mi><mi>t</mi><mi>e</mi><mo>&#xA0;</mo><mi>b</mi><mi>o</mi><mi>t</mi><mi>h</mi><mo>&#xA0;</mo><mi>s</mi><mi>i</mi><mi>d</mi><mi>e</mi><mi>s</mi><mo>&#xA0;</mo><mi>a</mi><mi>n</mi><mi>d</mi><mo>&#xA0;</mo><mi>t</mi><mi>h</mi><mi>e</mi><mi>n</mi><mo>&#xA0;</mo><mi>m</mi><mi>u</mi><mi>l</mi><mi>t</mi><mi>i</mi><mi>p</mi><mi>l</mi><mi>y</mi><mo>&#xA0;</mo><mi>b</mi><mi>o</mi><mi>t</mi><mi>h</mi><mo>&#xA0;</mo><mi>s</mi><mi>i</mi><mi>d</mi><mi>e</mi><mi>s</mi><mo>&#xA0;</mo><mi>b</mi><mi>y</mi><mo>&#xA0;</mo><mi>x</mi><mspace linebreak="newline"/><mfrac><mi>x</mi><msup><mfenced><mrow><mn>1</mn><mo>-</mo><mi>x</mi></mrow></mfenced><mn>2</mn></msup></mfrac><mo>=</mo><munderover><mrow><mo>&#x2211;</mo><mi>r</mi></mrow><mrow><mi>r</mi><mo>=</mo><mn>0</mn></mrow><mo>&#x221E;</mo></munderover><msup><mi>x</mi><mi>r</mi></msup><mspace linebreak="newline"/><mi>T</mi><mi>h</mi><mi>i</mi><mi>s</mi><mo>&#xA0;</mo><mi>i</mi><mi>s</mi><mo>&#xA0;</mo><mi>s</mi><mi>i</mi><mi>m</mi><mi>i</mi><mi>l</mi><mi>i</mi><mi>a</mi><mi>r</mi><mo>&#xA0;</mo><mi>t</mi><mi>o</mi><mo>&#xA0;</mo><mi>o</mi><mi>u</mi><mi>r</mi><mo>&#xA0;</mo><mi>s</mi><mi>u</mi><mi>m</mi><mo>&#xA0;</mo><mo>:</mo><mspace linebreak="newline"/><mfrac><mi>n</mi><mn>2</mn></mfrac><munderover><mo>&#x2211;</mo><mrow><mi>h</mi><mo>=</mo><mn>0</mn></mrow><mrow><msub><mi>log</mi><mn>2</mn></msub><mi>n</mi><mo>-</mo><mn>1</mn></mrow></munderover><mi>h</mi><msup><mfenced><mfrac><mn>1</mn><mn>2</mn></mfrac></mfenced><mi>h</mi></msup><mo>&#xA0;</mo><mi>w</mi><mi>h</mi><mi>e</mi><mi>r</mi><mi>e</mi><mo>&#xA0;</mo><mi>x</mi><mo>=</mo><mfrac><mn>1</mn><mn>2</mn></mfrac><mo>&#xA0;</mo><mi>a</mi><mi>n</mi><mi>d</mi><mo>&#xA0;</mo><mi>r</mi><mo>=</mo><mi>h</mi><mspace linebreak="newline"/><mi>U</mi><mi>sin</mi><mi>g</mi><mo>&#xA0;</mo><mi>t</mi><mi>h</mi><mi>e</mi><mo>&#xA0;</mo><mi>a</mi><mi>b</mi><mi>o</mi><mi>v</mi><mi>e</mi><mo>&#xA0;</mo><mi>f</mi><mi>o</mi><mi>r</mi><mi>m</mi><mi>u</mi><mi>l</mi><mi>a</mi><mo>&#xA0;</mo><mi>w</mi><mi>e</mi><mo>&#xA0;</mo><mi>h</mi><mi>a</mi><mi>v</mi><mi>e</mi><mo>:</mo><mspace linebreak="newline"/><mfrac><mi>n</mi><mn>2</mn></mfrac><mo>&#xD7;</mo><mfrac><mstyle displaystyle="true"><mfrac><mn>1</mn><mn>2</mn></mfrac></mstyle><msup><mfenced><mrow><mn>1</mn><mo>-</mo><mstyle displaystyle="true"><mfrac><mn>1</mn><mn>2</mn></mfrac></mstyle></mrow></mfenced><mn>2</mn></msup></mfrac><mo>=</mo><mfrac><mi>n</mi><mn>2</mn></mfrac><mo>&#xD7;</mo><mfrac><mn>1</mn><mn>2</mn></mfrac><mo>&#xD7;</mo><mfrac><mn>4</mn><mn>1</mn></mfrac><mo>=</mo><mi>n</mi></math>

This means that we can bound the number of comparisons in building the heap by n. So in total our algorithm will make:

**n+2nlog2n**

worst case comparisons. This is the worst case since here I assume that full comparisons among all possible elements are made. All the data in my table below are bounded from above by n+2nlog2n.

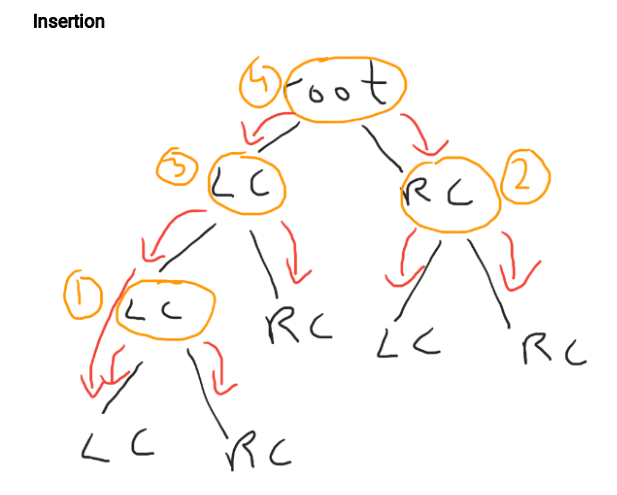
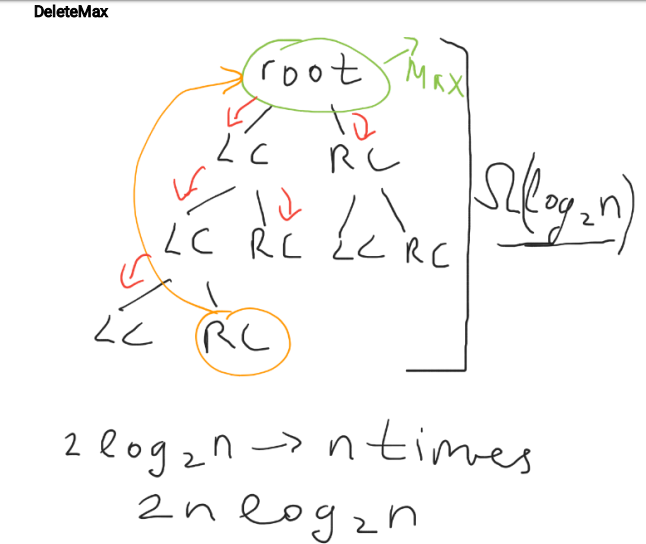


Figure : Demo of delete (left)-Demo of insert (right)

Below is my algorithm in pseudocode. I have not included the points where comparisons are made since this is just the pseudocode. For the implementation, I was based on the Heapsort algorithm of Introduction to Algorithms book. (Heapsort, 2009, pp. 151-161)

heapsort(int[] array){

//building the heap here. It starts from length/2 since this part is made only of leaves

for( i from array.length/2 to 0){   
 insert(array, array.length, i);  
 }//order and size property are not violated  
 //deleting the maximums here  
 for(int i=array.length-1; i>0;i--){//0 is the first element and the max  
 if(array[i]<array[0]){  
 swap maximum at position 0 with the last element  
 insert(array,i-1,0);//now do the insertion  
 }  
 }  
}

//building a max heap and deleting maximums

/\*

@params array: the array which will be bulit as a maxHeap  
 @params heapSize:array length-for bulding the heap , changes for deleting the max  
 @params pos: the position to check if it is in place   
\*/

private static long insert(int[] array, int heapSize, int pos){

l=2\*pos+1; //left child position  
 r=2\*pos+2; //right child position

if (l< heapSize and array[l]>array[pos])

largest=l;//setting the position of the largest to child since it is greater

}

else{ //left child not greater than element at pos  
 largest=pos;  
 }

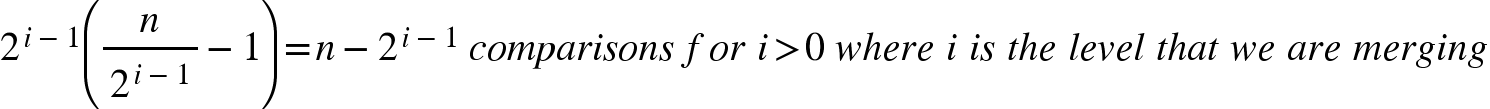
if(r<heapSize and array[r]>array[largest]){  
 largest=r;//setting the position of the largest  
 }

if(pos!= largest){ //if the largest is one of the children then exchange it with our element  
 swap array the largest element with the element at our position  
 insert(array, heapSize, largest); //now we compare the new largest with it children

}  
 }

## **Mergesort Algorithm and Pseudocode:**

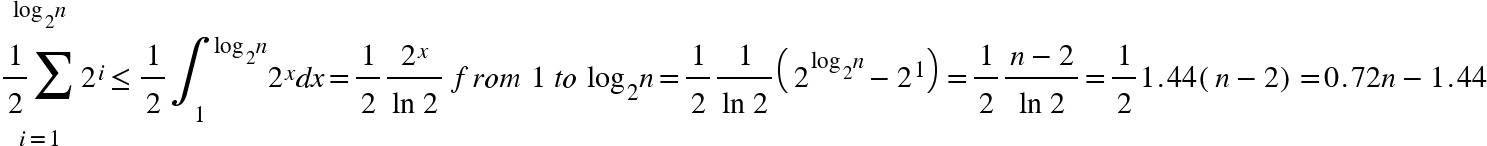
My implementation of merge sort algorithm is out of place. It splits the array of size n into 2 arrays of size n/2 at each level. When we reach the base case: that is an array of either size 0 or 1 we return from recursion and merge the subarrays by simultaneously comparing elements in both subarrays and putting them inside the larger array that will contain sorted input at each point of time that we get out of merge. In terms of running time, this algorithm sorts elements in O(nlog2n) time. It is based on the idea of a binary search tree, whose depth is log2n. So we will go log2n levels down and when merging we will make linear amount of work O(n). So we end up with the following recurrence:   
 T(n)=T(n/2)+O(n)  
Using Master’s Theorem to solve this we get: T(n)=O(nlog2n) as solution.  
The difference with Heapsort is on the number of comparisons made. As we can see from the tables, there is a huge difference between the number of comparisons for the same inputs. There is no exact formula about the number of comparisons in merge sort but a general version can be:  
 **nlog2n** (Merge sort, n.d.)

This happens due to the number of comparisons that we perform on the merging phase. Here I provide my proof to the above assumption.  
For each input size n, when merging we will make at most n-1 comparisons. So for level 0, we will make 2n/2-1=n-1 comparisons, for level 1 we will make 2(n/2-1) =n-2 since we will have 4 arrays of size n/4. The comparisons for one n/2 array will be n/4+n/4-1=n/2-1. Since we have 2 such groups, we end up with n-2 comparisons at most. Continuing like this we will get to the last level (log2n) where all arrays will be of size 1, if we assume the original size n to be a power of 2. At this point, we will have exactly n size 1 arrays. To merge them into n/2 arrays of size 2 we would need n/2 comparisons at most (1+1-1=1 comparisons for each). In general, we would need   
  


In general, we would have:

<math xmlns="http://www.w3.org/1998/Math/MathML"><mi>i</mi><mo>=</mo><mn>1</mn><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>-</mo><mo>&gt;</mo><mo>&#xA0;</mo><mi>n</mi><mo>-</mo><mn>1</mn><mo>&#xA0;</mo><mi>c</mi><mi>o</mi><mi>m</mi><mi>p</mi><mi>a</mi><mi>r</mi><mi>i</mi><mi>o</mi><mi>n</mi><mi>s</mi><mspace linebreak="newline"/><mi>i</mi><mo>=</mo><mn>2</mn><mo>&#xA0;</mo><mo>-</mo><mo>&gt;</mo><mo>&#xA0;</mo><mi>n</mi><mo>-</mo><mn>2</mn><mo>&#xA0;</mo><mi>c</mi><mi>o</mi><mi>m</mi><mi>p</mi><mi>a</mi><mi>r</mi><mi>i</mi><mi>o</mi><mi>n</mi><mi>s</mi><mspace linebreak="newline"/><mi>i</mi><mo>=</mo><mn>3</mn><mo>&#xA0;</mo><mo>-</mo><mo>&gt;</mo><mo>&#xA0;</mo><mi>n</mi><mo>-</mo><mn>4</mn><mo>&#xA0;</mo><mi>c</mi><mi>o</mi><mi>m</mi><mi>p</mi><mi>a</mi><mi>r</mi><mi>i</mi><mi>o</mi><mi>n</mi><mi>s</mi><mspace linebreak="newline"/><mo>.</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>.</mo><mspace linebreak="newline"/><mo>.</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>.</mo><mspace linebreak="newline"/><mo>.</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>&#xA0;</mo><mo>.</mo><mspace linebreak="newline"/><mi>i</mi><mo>=</mo><msub><mi>log</mi><mn>2</mn></msub><mi>n</mi><mo>&#xA0;</mo><mo>-</mo><mo>&gt;</mo><mfrac><mi>n</mi><mn>2</mn></mfrac><mo>&#xA0;</mo><mi>c</mi><mi>o</mi><mi>m</mi><mi>p</mi><mi>a</mi><mi>r</mi><mi>i</mi><mi>s</mi><mi>o</mi><mi>n</mi><mi>s</mi><mspace linebreak="newline"/><mspace linebreak="newline"/><munderover><mo>&#x2211;</mo><mrow><mi>i</mi><mo>=</mo><mn>1</mn></mrow><mrow><msub><mi>log</mi><mn>2</mn></msub><mi>n</mi></mrow></munderover><mo>&#xA0;</mo><mi>n</mi><mo>-</mo><msup><mn>2</mn><mrow><mi>i</mi><mo>-</mo><mn>1</mn></mrow></msup><mo>=</mo><mi>n</mi><msub><mi>log</mi><mn>2</mn></msub><mi>n</mi><mo>-</mo><munderover><mo>&#x2211;</mo><mrow><mi>i</mi><mo>=</mo><mn>1</mn></mrow><mrow><msub><mi>log</mi><mn>2</mn></msub><mi>n</mi></mrow></munderover><mo>&#xA0;</mo><mfrac><mn>1</mn><mn>2</mn></mfrac><msup><mn>2</mn><mi>i</mi></msup><mo>=</mo><mi>n</mi><msub><mi>log</mi><mn>2</mn></msub><mi>n</mi><mo>&#xA0;</mo><mo>-</mo><mfrac><mn>1</mn><mn>2</mn></mfrac><munderover><mo>&#x2211;</mo><mrow><mi>i</mi><mo>=</mo><mn>1</mn></mrow><mrow><msub><mi>log</mi><mn>2</mn></msub><mi>n</mi></mrow></munderover><msup><mn>2</mn><mi>i</mi></msup></math>

Using integrals, I found that the summation is bounded by



The summation result will be less than nlog2n since log2n will always be greater than 0.72n for our input that is greater than 10. Therefore, in general this will determine the number of comparisons. However, this is an approximation taking into account that the number of the original size of the array n is a power of 2. So approximately the number of comparisons will be:   
 **nlog2n**

Below is my algorithm in pseudocode. I have not included the points where comparisons are made since this is just the pseudocode.

mergesort(int[] array){

If(array.length>1){  
 al <-populate left subarray  
 ar <-populate right subarray  
 mergesort(al);  
 mergesort(ar);  
 merge(array,al,ar);  
}

}

merge(int[]array, int[] left, int[]right){  
 int a=0;  
 int b=0;  
 int c=0;  
 while(a<left.length and b<right.length){  
 if(left[a]<=right[b]){  
 array[c]=left[a];  
 c++; a++;  
 }

else{  
 array[c]=right[b];  
 c++; b++;   
 }  
 }//end of while  
   
 if(a=leftArray.length and b<rightArray.length){//we reached the end of left array but not of right  
 while(b<rightArray.length){  
 arraySorted[c]=rightArray[b];  
 b++;c++;  
 }

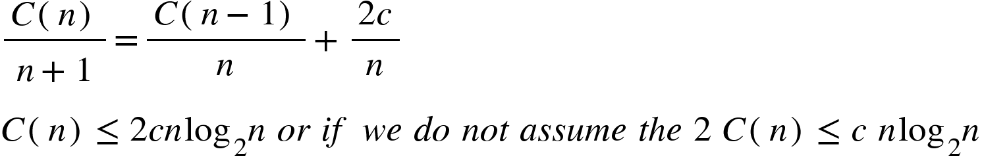
}//end of if

else if(a<leftArray.length && b==rightArray.length){  
 //we reached the end of right array but not of left  
 while(a<leftArray.length){  
 arraySorted[c]=leftArray[a];  
 a++; c++;  
 }  
 }  
}//end of merge

**Quicksort Algorithm and Pseudocode:**

For quicksort, I have provided an in-place implementation. It is a comparison based algorithm that runs in O(nlog2n) time where n is the original size of the inputted array. It uses a pivot value to split the original array into 2 subarrays with size T(i) and T(n-i-1). The size of the problem depends on the choice of the pivot value. The ideal case will be when the pivot is the median of the dataset, however finding the median during each recursive step will increase the time complexity. In this case, when the pivot is the median we will have as many elements in the left subarray as in the right one. Therefore, we will end up with 2 sub problems each of size n/2. Moreover, we do O(n) work on each recursive step. So we will have T(n)=2T(n/2)+O(n) where we end up with O(nlog2 n) time complexity using the Master’s theorem. The worst case happens when the pivot is the minimum or maximum, since one of our sub problems will have size one while the other n-i-1, so we will get a new sub problem that will be 1 less than the original one. In this case, the algorithm will require O(n2) time to sort the dataset. While on average we will need O(nlog2 n) time to sort the dataset assuming that the probability that each value on the dataset being the pivot is equal and that the pivot value is chosen randomly (Introduction to Algorithms, 2009). I used this approach in selecting the pivot value so that it would be harder for the pivot to be either the minimum or maximum of the dataset.

Moreover, I implemented another algorithm of mine for choosing the pivot, which makes it impossible for the pivot to be either the minimum or maximum of the dataset. I iterate twice over the dataset. Once to find the minimum and maximum vales. After doing this I find their mean. In addition, the second time I iterate through the dataset to find the value that is nearest to the mean. This algorithm does not guarantee us that the pivot will be the mean or median of the dataset however; it makes it impossible for the pivot to be the minimum and/or the maximum value. Moreover, for random values in the dataset that are very spread it will choose the median or a value that is near the median, while for high polarised datasets(so that the data on one of the bounds are very dense) it will probably choose a pivot that will shrink the size of the sub problem by a small amount. In all the cases there is no chance that the maximum or minimum will be picked. However, a big drawback of this method is it huge number of comparisons. I have included it in the below table but just for comparing it with the other algorithms, however my strategy for selecting the pivot is using randomization.   
The number of comparisons for quicksort varies based on which case we are. If we are on the best case, worst case or neither. If we are on the best case then we are going to make C(n)=2C(n/2)+n-1. n is the number of comparisons performed during each partition since we have to compare every element with the pivot. If we solve this using master theorem we are going to get C(n)=nlog2n comparisons. On average the algorithm will make C(n)=C(i)+C(n-i-1)-n-1 comparisons where i is the number of elements less than the pivot. Using equation 14 from lecture notes on quicksort, we reach to the point where:



So as we can see from this, the average number of comparisons that quicksort makes is cnlog2n, which means that it is some times greater than merge sort.

Below is my algorithm in pseudocode. I have not included the points where comparisons are made since this is just the pseudocode.

realQuicksort(int[] array, int start, int end){  
 if(start<end){  
 int pivot=partition(array, start, end);//setting the pivot  
 realQuicksort(array, start, pivot-1);  
 realQuicksort(array, pivot+1, end);  
 }

}

partition(int[] array,int start,int end){

long comp=0;//setting the comparisons to0 at the beginning  
 int pivot= randomChoosePivot(start, end); //this is the random pivot

//int pivot=myChoosePivot(array,start,end);//this is the position of pivot using my method

//here we exchange the last element with the pivot so that the algorithm will proceed as normal  
 exchange pivot with last element  
 //normal algorithm of the book starts here

int x=array[end];//this is the pivot element  
 int i=start-1;

//now starts the real partition

for(int j=start;j<end; j++){

if(array[j]<=x){  
 i=i+1;  
 //exchanging a[i] with a[j];  
 int temp1=array[i];  
 array[i]=array[j];  
 array[j]=temp1;  
 }

}//end of for

//exchange the pivot value with A[i+1]

int temp1=array[i+1];

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

array[end]=temp1;

return i+1;//returning the pivot position to be used in the next partition

}

Here is the pseudocode for choosing a random pivot value.

randomChoosePivot(int start, int end){

int numberOfElements=end-start+1;//number of elements in the current subarray  
 Random ran= new Random();  
 //here we select a random position from start to end and put it as the pivot value  
 int pos=start+ran.nextInt(numberOfElements);//will produce an integer between start-end

return pos;  
}

Here is the pseudocode for my algorithm for choosing the pivot using the mean of the largest and smallest elements of the dataset.

myChoosePivot(int[] array, int start, int end){  
 long comp=0;  
 int max=array[start];  
 int min=array[start];

iterate through the array and find the minimum and the maximum

mean=min+max/2;//finding the mean of the highest and lowest element

int dif=max;//the difference between the mean and an element of the array

int pos=0;//position of the supposed median of our array

//now we need to find the value in the array that is the closest to the mean

for(int i=start;i<=end;i++){

if(Math.abs(array[i]-mean)<dif){

dif=Math.abs(array[i]-mean);//setting the new difference

pos=i;//setting the position of our supposed pivot

}

}//end of for  
return pos;

}

Analysis of InsMergesort is on the next section. Below there are the tables containing the data of the experiments.

# **Recorded Data:**

Table : Number of comparisons for each input size for each algorithm

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Algorithm | Array Size/Input | a | b | c | d | e | Mean |
| Heapsort | 10 | 67 | 69 | 71 | 67 | 67 | 68.2 |
| 20 | 179 | 171 | 173 | 165 | 179 | 173.4 |
| 50 | 579 | 573 | 579 | 573 | 565 | 573.8 |
| 100 | 1361 | 1357 | 1363 | 1373 | 1359 | 1362.6 |
| 200 | 3145 | 3165 | 3079 | 3125 | 3097 | 3122.2 |
| 500 | 9017 | 9079 | 9033 | 9085 | 9107 | 9064.2 |
| 1000 | 20253 | 20075 | 20117 | 20101 | 20131 | 20135.4 |
| 2000 | 44321 | 44243 | 44221 | 44375 | 44299 | 44291.8 |
| 5000 | 124221 | 124131 | 124141 | 124095 | 124199 | 124157.4 |
| 10000 | 268273 | 268307 | 268533 | 268251 | 268333 | 268339.4 |
| 20000 | 576755 | 576595 | 576745 | 576619 | 576603 | 576663.4 |
| 50000 | 1574925 | 1574777 | 1574327 | 1575089 | 1574113 | 1574646 |
| 100000 | 3350083 | 3349763 | 3350947 | 3350427 | 3348647 | 3349973 |
| Mergesort | 10 | 21 | 23 | 21 | 24 | 23 | 22.4 |
| 20 | 62 | 63 | 61 | 67 | 65 | 63.6 |
| 50 | 216 | 223 | 225 | 221 | 220 | 221 |
| 100 | 543 | 539 | 544 | 543 | 543 | 542.4 |
| 200 | 1282 | 1280 | 1282 | 1279 | 1281 | 1280.8 |
| 500 | 3859 | 3842 | 3853 | 3852 | 3867 | 3854.6 |
| 1000 | 8703 | 8712 | 8721 | 8716 | 8723 | 8715 |
| 2000 | 19402 | 19456 | 19398 | 19428 | 19449 | 19426.6 |
| 5000 | 55191 | 55277 | 55297 | 55189 | 55213 | 55233.4 |
| 10000 | 120420 | 120515 | 120539 | 120464 | 120402 | 120468 |
| 20000 | 260806 | 260902 | 261010 | 260802 | 260794 | 260862.8 |
| 50000 | 718306 | 718100 | 718126 | 718033 | 717940 | 718101 |
| 100000 | 1536512 | 1536373 | 1536083 | 1536283 | 1536211 | 1536292 |
| Quicksort[[3]](#footnote-3) | 10 | 24 | 25 | 20 | 26 | 31 | 25.2 |
| 20 | 70 | 60 | 72 | 80 | 62 | 68.8 |
| 50 | 244 | 256 | 294 | 253 | 249 | 259.2 |
| 100 | 718 | 659 | 604 | 646 | 659 | 657.2 |
| 200 | 1654 | 1523 | 1581 | 1461 | 1595 | 1562.8 |
| 500 | 4578 | 5205 | 5180 | 4487 | 5007 | 4891.4 |
| 1000 | 10984 | 11860 | 10607 | 9778 | 10554 | 10756.6 |
| 2000 | 25193 | 25244 | 26201 | 23466 | 25298 | 25080.4 |
| 5000 | 66543 | 67342 | 68836 | 67074 | 68300 | 67619 |
| 10000 | 157546 | 148528 | 152970 | 151956 | 160224 | 154244.8 |
| 20000 | 328965 | 349816 | 340937 | 328568 | 331725 | 336002.2 |
| 50000 | 920909 | 909429 | 941543 | 898956 | 948990 | 923965.4 |
| 100000 | 1965742 | 1987465 | 2004258 | 1989766 | 2010942 | 1991635 |

Table 1: Number of comparisons for each input size for each algorithm

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Algorithm | Array Size/Input | a | b | c | d | e | Mean |
| Quicksort[[4]](#footnote-4) | 10 | 86 | 94 | 98 | 110 | 82 | 94 |
| 20 | 240 | 293 | 280 | 295 | 285 | 278.6 |
| 50 | 1187 | 1311 | 1366 | 1292 | 1300 | 1291.2 |
| 100 | 4416 | 4244 | 4628 | 4451 | 3948 | 4337.4 |
| 200 | 15338 | 16308 | 15532 | 14774 | 15320 | 15454.4 |
| 500 | 93091 | 91868 | 87505 | 85968 | 93961 | 90478.6 |
| 1000 | 354509 | 347487 | 347860 | 332314 | 360721 | 348578.2 |
| 2000 | 1368249 | 1351622 | 1334672 | 1334514 | 1419569 | 1361725 |
| 5000 | 8460661 | 8407120 | 8365452 | 8357690 | 8564459 | 8431076 |
| 10000 | 33672673 | 33473997 | 33574947 | 33201876 | 33406353 | 33465969 |
| 20000 | 100M+ | 100M+ | 100M+ | 100M+ | 100M+ | 100M+ |
| 50000 | 800M+ | 800M+ | 800M+ | 800M+ | 800M+ | 800M+ |
| 100000 | 1B+ | 1B+ | 1B+ | 1B+ | 1B+ | 1B+ |
| InsMergesort | 10 | 18 | 19 | 19 | 23 | 20 | 19.8 |
| 20 | 61 | 58 | 57 | 64 | 58 | 59.6 |
| 50 | 212 | 216 | 218 | 215 | 215 | 215.2 |
| 100 | 532 | 523 | 531 | 532 | 533 | 530.2 |
| 200 | 1258 | 1248 | 1254 | 1256 | 1259 | 1255 |
| 500 | 3793 | 3771 | 3771 | 3791 | 3796 | 3784.4 |
| 1000 | 8543 | 8554 | 8566 | 8571 | 8585 | 8563.8 |
| 2000 | 19109 | 19147 | 19085 | 19148 | 19185 | 19134.8 |
| 5000 | 54331 | 54396 | 54419 | 54387 | 54296 | 54365.8 |
| 10000 | 118677 | 118811 | 118797 | 118830 | 118623 | 118747.6 |
| 20000 | 257144 | 257439 | 257466 | 257588 | 257384 | 257404.2 |
| 50000 | 712646 | 712414 | 712458 | 712341 | 712463 | 712464.4 |
| 100000 | 1525131 | 1524995 | 1524811 | 1524870 | 1524999 | 1524961 |

Table 2: Algorithms comparison in terms of the average number of comparisons made

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Algorithms that we had to test | | | |
| Input Size/Algorithm | **Heapsort** | **Mergesort** | **Quicksort1** | **InsMergesort[[5]](#footnote-5)** | **Quicksort2** |
| **10** | 68.2 | 22.4 | 25.2 | 19.8 | 94 |
| **20** | 173.4 | 63.6 | 68.8 | 59.6 | 278.6 |
| **50** | 573.8 | 221 | 259.2 | 215.2 | 1291.2 |
| **100** | 1362.6 | 542.4 | 657.2 | 530.2 | 4337.4 |
| **200** | 3122.2 | 1280.8 | 1562.8 | 1255 | 15454.4 |
| **500** | 9064.2 | 3854.6 | 4891.4 | 3784.4 | 90478.6 |
| **1000** | 20135.4 | 8715 | 10756.6 | 8563.8 | 348578.2 |
| **2000** | 44291.8 | 19426.6 | 25080.4 | 19134.8 | 1361725 |
| **5000** | 124157.4 | 55233.4 | 67619 | 54365.8 | 8431076 |
| **10000** | 268339.4 | 120468 | 154244.8 | 118747.6 | 33465969 |
| **20000** | 576663.4 | 260862.8 | 336002.2 | 257404.2 | 100M+ |
| **50000** | 1574646 | 718101 | 923965.4 | 712464.4 | 800M+ |
| **100000** | 3349973 | 1536292 | 1991635 | 1524961 | 1B+ |

As we can see from Table 2, Mergesort outperforms both Quicksort and Heapsort in terms of comparisons. Although they all did nlog2 comparisons, in general the constant in their algorithms do matter a lot in practice. The number of comparisons on merge sort is nlog2 if we refer to the analysis above; the number of comparisons in heap sort is n+2nlog2n while in quicksort it is again cnlog2n. However, the constant in quicksort is greater than the constant in the merge sort. Estimated to be around 1.39nlog2n (Quicksort, n.d.) Which means that on average quicksort will make around 39% more comparisons than merge sort. Below there is a chart of the 4 algorithms, whose comparison number we had to compare.

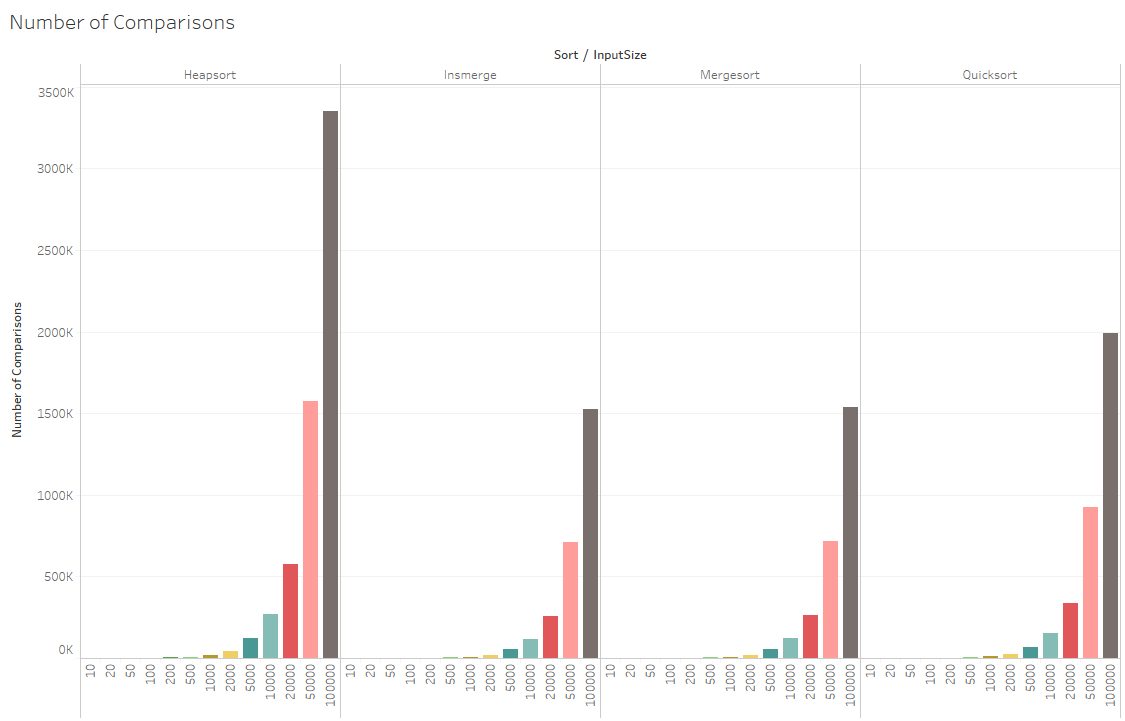


Figure : Comparing the number of comparisons for 4 of the algorithms

## **InsMergesort Algorithm and Pseudocode:**

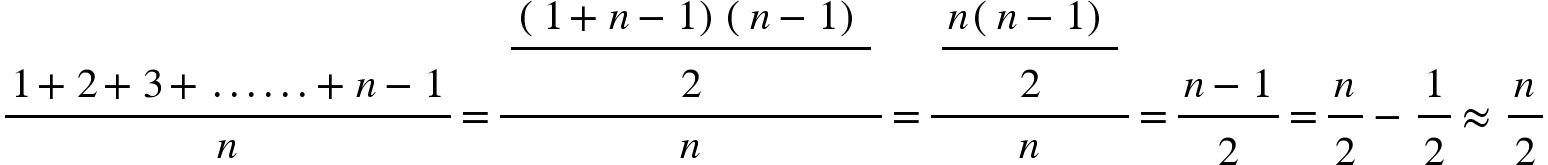
Here, as described I had to implement a version of merge sort combined with insertion sort. For this, the first job was to find a value of M such that it is better for array sizes n < M to use insertion sort rather than merge sort. To do this, I decided to compare the number of comparisons that each algorithm makes, and find an input size N such that for M<N it is better to use insertion sort in terms of comparisons made rather than merge sort.  
Here I include the pseudocode for insertion sort:

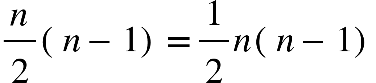
insertionsort(int[] array){  
 for(int i=1; i<array.length;i++){  
 value=array[i];  
 j=i;  
 while(j>0 && array[j-1]>value){

array[j]=array[j-1];//moving values to the right  
 j--;

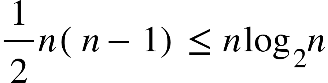
}//end of while  
 array[j]=value;//now we have found the place of value and we insert the value }//end of for   
}//end of insertion sort

According to the pseudocode above, we cannot deduce a formula about the number of comparisons made since it will depend on the order in which the items are in the unsorted array. If we take the best case, which would be an already sorted array, then for sure there, will be n-1 comparisons made, where n is the size of the array. If we take the worst case that would be an array sorted in reverse order, we are going to make, as many comparisons as there are elements on the left (already sorted part) of the array. This will happen n-1 times due to the outer loop. However, we need to know an approximation of the number of comparisons that made on each of the n-1 times. For this, I am going to find the average number of elements that the left part of the array will have:

  
 elements on the left list. If we multiply this by the number of passes to the array, we will get:



comparisons in the worst case. While the merge sort according to the logic explained above it will make nlog2n comparisons. So if we compare the worst case of comparison sort with merge sort we will find an M such that for sure for all n<M insertion sort will perform fewer comparisons than merge sort.



I solved this inequality using trials for the value of n:

|  |  |  |
| --- | --- | --- |
| n | Insertion | Merge |
| 4 | 6 | 8 |
| **5** | **10** | **12** |
| 10 | 45 | 33.2 |
| 8 | 28 | 24 |
| 7 | 26 | 19.6 |
| 6 | 18 | 16 |

So I found that taking the worst case that an array is reversely sorted the insertion sort will perform less comparisons than the merge sort will. Therefore, in my algorithm, for all inputs that contain 5 elements or less I use insertion and for the others merge. Therefore, I break the entire array until size 5 end then use insertion sort to sort the pieces, which then will be merged using the merge method of merge sort.

Below I provide my pseudo code for this algorithm. No comparisons are provided since it is just a pseudocode.

mergesort(int[] array){

If(array.length>5){//do mergesort only for array sizes greater than 5  
 al <-populate left subarray  
 ar <-populate right subarray  
 mergesort(al);  
 mergesort(ar);  
 merge(array,al,ar);  
}

else{){//do insertionsort only for array sizes less than or equal to 6  
 insertionsort(array); //calling the pseudocode above  
}

}

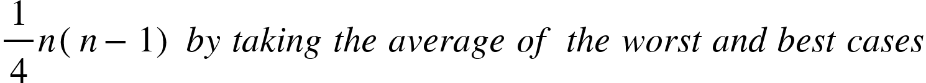
merge(int[]array, int[] left, int[]right){  
 int a=0;  
 int b=0;  
 int c=0;  
 while(a<left.length and b<right.length){  
 if(left[a]<=right[b]){  
 array[c]=left[a];  
 c++; a++;  
 }

else{  
 array[c]=right[b];  
 c++; b++;   
 }  
 }//end of while  
   
 if(a=leftArray.length and b<rightArray.length){//we reached the end of left array but not of right  
 while(b<rightArray.length){  
 arraySorted[c]=rightArray[b];  
 b++;c++;  
 }

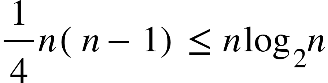
}//end of if

else if(a<leftArray.length && b==rightArray.length){  
 //we reached the end of right array but not of left  
 while(a<leftArray.length){  
 arraySorted[c]=leftArray[a];  
 a++; c++;  
 }  
 }  
}//end of merge

Actual value of M and number of comparisons

According to Watson e-book, I found out that on average insertion sort makes   
 (Insertion Sort, n.d.)

If I compare this to the number of comparisons performed in merge sort similarly to comparing the worst case of insertion with merge sort I will get:

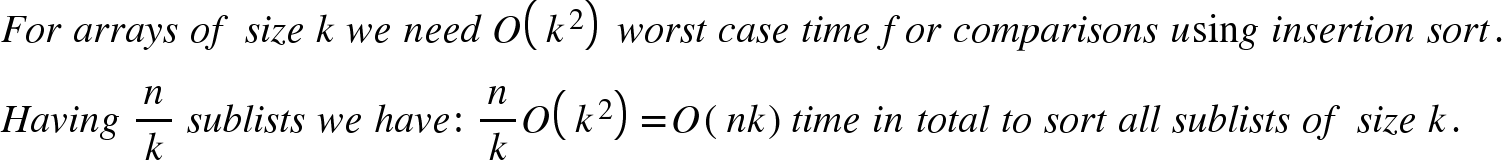


When solving this by trial as above, I find that for n<18 the number of comparisons performed by insertion sort would be less than the number performed by merge sort, but since this is on average for the insertion sort than we cannot be sure that this will provide fewer comparisons for all inputs. Therefore, I switched to the worst case and decided to use insertion sort for arrays of size less than 5. This guarantees me that ins-mergesort will work faster (at least in terms of time) than merge sort or insertion sort alone for every input.

General solution for value of M and time complexity of the algorithm

Moreover, in order to find a more general solution to the values of k and if we take into account the running times of the algorithms we would have :

(For the following reasoning, I have used Problem 2-1, Page 40 in Introduction to Algorithms book.)

If we suppose that our algorithm will use insertion sort for n/k arrays of size k then we would need O(kn) time to sort all of this arrays. This is because:   
 

Moreover, in order to merge all this sub lists using merge method of merge sort:

<math xmlns="http://www.w3.org/1998/Math/MathML"><mi>T</mi><mi>h</mi><mi>e</mi><mi>r</mi><mi>e</mi><mo>&#xA0;</mo><mi>w</mi><mi>i</mi><mi>l</mi><mi>l</mi><mo>&#xA0;</mo><mi>b</mi><mi>e</mi><mo>&#xA0;</mo><msub><mi>log</mi><mn>2</mn></msub><mi>n</mi><mo>-</mo><msub><mi>log</mi><mn>2</mn></msub><mi>k</mi><mo>=</mo><msub><mi>log</mi><mn>2</mn></msub><mfrac><mi>n</mi><mi>k</mi></mfrac><mo>&#xA0;</mo><mi>d</mi><mi>e</mi><mi>p</mi><mi>t</mi><mi>h</mi><mfenced><mrow><mi>n</mi><mi>u</mi><mi>m</mi><mi>b</mi><mi>e</mi><mi>r</mi><mo>&#xA0;</mo><mi>o</mi><mi>f</mi><mo>&#xA0;</mo><mi>l</mi><mi>e</mi><mi>v</mi><mi>e</mi><mi>l</mi><mi>s</mi></mrow></mfenced><mo>&#xA0;</mo><mi>i</mi><mi>n</mi><mo>&#xA0;</mo><mi>t</mi><mi>h</mi><mi>e</mi><mo>&#xA0;</mo><mi>t</mi><mi>r</mi><mi>e</mi><mi>e</mi><mo>&#xA0;</mo><mi>t</mi><mi>h</mi><mi>a</mi><mi>t</mi><mo>&#xA0;</mo><mi>w</mi><mi>e</mi><mo>&#xA0;</mo><mi>a</mi><mi>r</mi><mi>e</mi><mo>&#xA0;</mo><mi>g</mi><mi>o</mi><mi>i</mi><mi>n</mi><mi>g</mi><mo>&#xA0;</mo><mi>t</mi><mi>o</mi><mo>&#xA0;</mo><mi>m</mi><mi>e</mi><mi>r</mi><mi>g</mi><mi>e</mi><mo>&#xA0;</mo><mspace linebreak="newline"/><mi>S</mi><mi>i</mi><mi>n</mi><mi>c</mi><mi>e</mi><mo>&#xA0;</mo><mi>e</mi><mi>a</mi><mi>c</mi><mi>h</mi><mo>&#xA0;</mo><mi>l</mi><mi>e</mi><mi>v</mi><mi>e</mi><mi>l</mi><mo>&#xA0;</mo><mi>r</mi><mi>e</mi><mi>q</mi><mi>u</mi><mi>i</mi><mi>r</mi><mi>e</mi><mi>s</mi><mo>&#xA0;</mo><mi>c</mi><mi>n</mi><mo>&#xA0;</mo><mi>a</mi><mi>m</mi><mi>o</mi><mi>u</mi><mi>n</mi><mi>t</mi><mo>&#xA0;</mo><mi>o</mi><mi>f</mi><mo>&#xA0;</mo><mi>t</mi><mi>i</mi><mi>m</mi><mi>e</mi><mo>&#xA0;</mo><mi>f</mi><mi>o</mi><mi>r</mi><mo>&#xA0;</mo><mi>m</mi><mi>e</mi><mi>r</mi><mi>g</mi><mi>i</mi><mi>n</mi><mi>g</mi><mo>&#xA0;</mo><mi>i</mi><mi>n</mi><mo>&#xA0;</mo><mi>t</mi><mi>o</mi><mi>t</mi><mi>a</mi><mi>l</mi><mo>&#xA0;</mo><mi>w</mi><mi>e</mi><mo>&#xA0;</mo><mi>w</mi><mi>o</mi><mi>u</mi><mi>l</mi><mi>d</mi><mo>&#xA0;</mo><mi>r</mi><mi>e</mi><mi>q</mi><mi>u</mi><mi>i</mi><mi>r</mi><mi>e</mi><mo>&#xA0;</mo><mo>:</mo><mspace linebreak="newline"/><mi>c</mi><mi>n</mi><msub><mi>log</mi><mn>2</mn></msub><mfrac><mi>n</mi><mi>k</mi></mfrac><mo>&#xA0;</mo><mi>o</mi><mi>r</mi><mo>&#xA0;</mo><mi>u</mi><mi>sin</mi><mi>g</mi><mo>&#xA0;</mo><mo>&#xA0;</mo><mi>a</mi><mi>s</mi><mi>y</mi><mi>m</mi><mi>p</mi><mi>t</mi><mi>o</mi><mi>t</mi><mi>i</mi><mi>c</mi><mo>&#xA0;</mo><mi>n</mi><mi>o</mi><mi>t</mi><mi>a</mi><mi>t</mi><mi>i</mi><mi>o</mi><mi>n</mi><mo>&#xA0;</mo><mi>w</mi><mi>e</mi><mo>&#xA0;</mo><mi>w</mi><mi>o</mi><mi>u</mi><mi>l</mi><mi>d</mi><mo>&#xA0;</mo><mi>r</mi><mi>e</mi><mi>q</mi><mi>u</mi><mi>i</mi><mi>r</mi><mi>e</mi><mo>&#xA0;</mo><mi>O</mi><mfenced><mrow><mi>n</mi><msub><mi>log</mi><mn>2</mn></msub><mfrac><mi>n</mi><mi>k</mi></mfrac></mrow></mfenced></math>

Now if we combine them we will have total running time of:

<math xmlns="http://www.w3.org/1998/Math/MathML"><mi>O</mi><mfenced><mrow><mi>k</mi><mi>n</mi><mo>+</mo><mi>n</mi><msub><mi>log</mi><mn>2</mn></msub><mfrac><mi>n</mi><mi>k</mi></mfrac></mrow></mfenced><mo>=</mo><mi>O</mi><mfenced><mrow><mi>k</mi><mi>n</mi><mo>+</mo><mi>n</mi><msub><mi>log</mi><mn>2</mn></msub><mi>n</mi><mo>-</mo><mi>n</mi><msub><mi>log</mi><mn>2</mn></msub><mi>k</mi></mrow></mfenced><mspace linebreak="newline"/><mi>n</mi><mo>&gt;</mo><mi>k</mi><mo>&#xA0;</mo><mi>sin</mi><mi>c</mi><mi>e</mi><mo>&#xA0;</mo><mi>w</mi><mi>e</mi><mo>&#xA0;</mo><mi>a</mi><mi>r</mi><mi>e</mi><mo>&#xA0;</mo><mi>t</mi><mi>a</mi><mi>l</mi><mi>k</mi><mi>i</mi><mi>n</mi><mi>g</mi><mo>&#xA0;</mo><mi>a</mi><mi>b</mi><mi>o</mi><mi>u</mi><mi>t</mi><mo>&#xA0;</mo><mi>s</mi><mi>m</mi><mi>a</mi><mi>l</mi><mi>l</mi><mo>&#xA0;</mo><mi>v</mi><mi>a</mi><mi>l</mi><mi>u</mi><mi>e</mi><mi>s</mi><mo>&#xA0;</mo><mi>o</mi><mi>f</mi><mo>&#xA0;</mo><mi>k</mi><mo>&#xA0;</mo><mi>f</mi><mi>o</mi><mi>r</mi><mo>&#xA0;</mo><mo>&#xA0;</mo><mi>w</mi><mi>h</mi><mi>i</mi><mi>c</mi><mi>h</mi><mo>&#xA0;</mo><mi>t</mi><mi>o</mi><mo>&#xA0;</mo><mi>u</mi><mi>s</mi><mi>e</mi><mo>&#xA0;</mo><mi>i</mi><mi>n</mi><mi>s</mi><mi>e</mi><mi>r</mi><mi>t</mi><mi>i</mi><mi>o</mi><mi>n</mi><mo>&#xA0;</mo><mi>s</mi><mi>o</mi><mi>r</mi><mi>t</mi><mspace linebreak="newline"/><mi>S</mi><mi>o</mi><mo>&#xA0;</mo><mi>f</mi><mi>o</mi><mi>r</mi><mo>&#xA0;</mo><mi>s</mi><mi>u</mi><mi>r</mi><mi>e</mi><mo>&#xA0;</mo><mi>n</mi><msub><mi>log</mi><mn>2</mn></msub><mi>n</mi><mo>&gt;</mo><mi>n</mi><msub><mi>log</mi><mn>2</mn></msub><mi>k</mi><mo>&#xA0;</mo><mi>sin</mi><mi>c</mi><mi>e</mi><mo>&#xA0;</mo><mi>n</mi><mo>&#xA0;</mo><mi>i</mi><mi>s</mi><mo>&#xA0;</mo><mi>s</mi><mi>o</mi><mi>m</mi><mi>e</mi><mo>&#xA0;</mo><mi>p</mi><mi>o</mi><mi>w</mi><mi>e</mi><mi>r</mi><mi>s</mi><mo>&#xA0;</mo><mi>o</mi><mi>f</mi><mo>&#xA0;</mo><mn>2</mn><mo>&#xA0;</mo><mi>g</mi><mi>r</mi><mi>e</mi><mi>a</mi><mi>t</mi><mi>e</mi><mi>r</mi><mo>&#xA0;</mo><mi>t</mi><mi>h</mi><mi>a</mi><mi>n</mi><mo>&#xA0;</mo><mi>k</mi><mspace linebreak="newline"/><mi>T</mi><mi>h</mi><mi>e</mi><mo>&#xA0;</mo><mi>o</mi><mi>n</mi><mi>l</mi><mi>y</mi><mo>&#xA0;</mo><mi>t</mi><mi>h</mi><mi>i</mi><mi>n</mi><mi>g</mi><mo>&#xA0;</mo><mi>w</mi><mi>e</mi><mo>&#xA0;</mo><mi>h</mi><mi>a</mi><mi>v</mi><mi>e</mi><mo>&#xA0;</mo><mi>t</mi><mi>o</mi><mo>&#xA0;</mo><mi>c</mi><mi>o</mi><mi>m</mi><mi>p</mi><mi>a</mi><mi>r</mi><mi>e</mi><mo>&#xA0;</mo><mi>i</mi><mi>s</mi><mo>:</mo><mspace linebreak="newline"/><mo>&#xA0;</mo><mi>k</mi><mi>n</mi><mo>&#xA0;</mo><mi>w</mi><mi>i</mi><mi>t</mi><mi>h</mi><mo>&#xA0;</mo><mi>n</mi><msub><mi>log</mi><mn>2</mn></msub><mi>n</mi><mo>&#xA0;</mo><mi>f</mi><mi>o</mi><mi>r</mi><mo>&#xA0;</mo><mi>w</mi><mi>h</mi><mi>i</mi><mi>c</mi><mi>h</mi><mo>&#xA0;</mo><mi>k</mi><mi>n</mi><mo>&#xA0;</mo><mo>&lt;</mo><mo>&#xA0;</mo><mi>n</mi><msub><mi>log</mi><mn>2</mn></msub><mi>n</mi><mo>&#xA0;</mo><mi>i</mi><mi>n</mi><mo>&#xA0;</mo><mi>o</mi><mi>r</mi><mi>d</mi><mi>e</mi><mi>r</mi><mo>&#xA0;</mo><mi>f</mi><mi>o</mi><mi>r</mi><mo>&#xA0;</mo><mo>&#xA0;</mo><mi>n</mi><msub><mi>log</mi><mn>2</mn></msub><mi>n</mi><mo>&#xA0;</mo><mi>t</mi><mi>o</mi><mo>&#xA0;</mo><mi>b</mi><mi>e</mi><mo>&#xA0;</mo><mi>t</mi><mi>h</mi><mi>e</mi><mo>&#xA0;</mo><mi>m</mi><mi>o</mi><mi>s</mi><mi>t</mi><mo>&#xA0;</mo><mi>s</mi><mi>i</mi><mi>g</mi><mi>n</mi><mi>i</mi><mi>f</mi><mi>i</mi><mi>c</mi><mi>a</mi><mi>n</mi><mi>t</mi><mo>&#xA0;</mo><mi>t</mi><mi>e</mi><mi>r</mi><mi>m</mi><mo>&#xA0;</mo><mi>o</mi><mi>n</mi><mspace linebreak="newline"/><mi>e</mi><mi>x</mi><mi>p</mi><mi>r</mi><mi>e</mi><mi>s</mi><mi>s</mi><mi>s</mi><mi>i</mi><mi>o</mi><mi>n</mi><mo>&#xA0;</mo><mo>.</mo><mspace linebreak="newline"/><mi>U</mi><mi>sin</mi><mi>g</mi><mo>&#xA0;</mo><mi>t</mi><mi>h</mi><mi>i</mi><mi>s</mi><mo>&#xA0;</mo><mi>log</mi><mi>i</mi><mi>c</mi><mo>&#xA0;</mo><mi>w</mi><mi>e</mi><mo>&#xA0;</mo><mi>h</mi><mi>a</mi><mi>v</mi><mi>e</mi><mo>:</mo><mo>&#xA0;</mo><mspace linebreak="newline"/><mi>n</mi><mi>k</mi><mo>&lt;</mo><mo>&#xA0;</mo><mi>n</mi><msub><mi>log</mi><mn>2</mn></msub><mi>n</mi><mo>&#x21D2;</mo><mo>&#xA0;</mo><mi>k</mi><mo>&lt;</mo><msub><mi>log</mi><mn>2</mn></msub><mi>n</mi><mspace linebreak="newline"/><mspace linebreak="newline"/></math>

Therefore, for array sizes less than log2n we will end up with optimal time complexity of O(nlog2n).

This is in line with my assumptions above regarding an exact value of k for which I implemented to be 5 in my algorithm. It may not be very significant for inputs of smaller sizes but for array sizes of 100+ it is very important and 5 will always fall in this range. This range will be extended up until n=17 using the analysis provided above regarding the average case of insertion sort. Therefore, the most optimal value will fall between 5 and 17, which satisfies our finding above: k<log2n.

Below I provide 2 graphs of comparing the number of comparisons of insertion sort and merge sort.

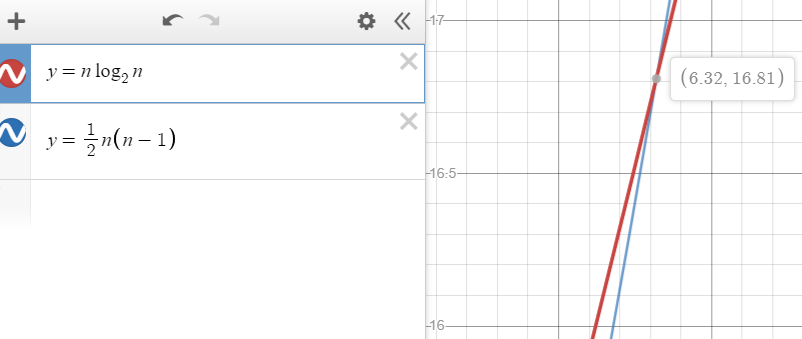
(Desmos, n.d.)

Figure : Worst Case of Insertion Sort vs Merge Sort Comparisons

For values of n<6 we see that the function representing the number of comparisons of insertion (blue) is smaller than the graph of merge sort (red).

While in the average case of insertion sort, we have:

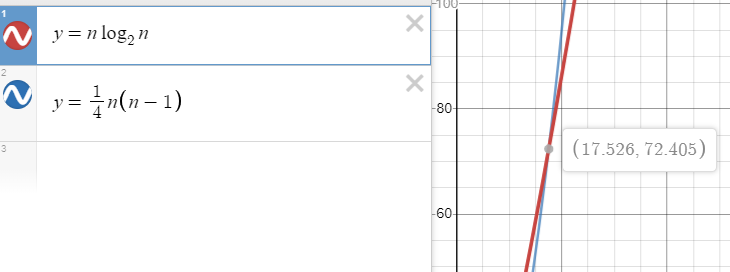


Figure : Average Case of Insertion Sort vs Merge Sort Comparisons

So on average for inputs less than 17 the graph representing the number of comparisons of insertion sort is grows slower.

The time complexity of this algorithm as explained above is **O(nlog2n)** if the value of M for which we use insertion sorts is less than log2n.

Correctness of this algorithm

In order to prove that this algorithm is correct we have to prove the correctness of merge sort and insertion sort separately. After we prove the correctness of insertion sort, we will provide already sorted arrays to the merge method of merge sort in order to merge them. So if we prove that merge will merge correctly arrays of any size then we are done.

In order for the insertion sort to sort its elements it two logical subarrays left and right. At any point of iteration, i the left subarray will have sorted elements and the right subarray unsorted elements. It will iterate n-1 times to find the correct place of its elements (outer loop). The inner loop will run until either it reaches the first position or we encounter an element that is greater than the value that we are currently seeing. What it does is constantly shifting greater elements to the right.

Using this logic, we have:

For an array of 2 elements a1 and a2 starting from position, 2 we will compare it with the element in the first position. If it is greater than we do nothing and the inner loop terminates. If it is less than, we shift a1 to position 2 and go out of the while loop. At this point, we insert the value at the empty space that was created by shifting a1, which in our case is the first position. So we end up with a sorted array of size 2.  
For input size of k, the outer loop will run k-1 times. We keep the value that we want to find its place in the left part of the list in a variable prior to entering the while loop so that we will not lose track of it. Then we use another counter to traverse the sorted part (left part) of the array. Each time that this loop encounter an element greater than our value it will shift the element to the right by one position. Therefore, if a[j-1] is greater than our value, it will be shifted to the jth position. In the first running of the while loop we will have element of position i-1 shifted to position i if it is greater than our value at a[i], otherwise the while loop will not run at all and the value is already at the correct place. While if we encounter values a[j], which are greater than our value, then we will iterate inside until we find a value that is greater than our value. When we break, we put our value into its correct position. At the end of any iteration, we are guaranteed to have a sorted array on the left side from index 0 to i. Moreover, at the start of each iteration on the while loop the elements in positions a[0]to a[j] will be less than or equal to our value and the elements from a[j+1] to a[i] will be greater than our value. This means that the movements of the values will not overlap and destroy each other values as long as we keep the element i in a variable and inserting it after the inner loop terminates. This happens inside one iteration of the outer loop. In addition, since in one iteration nothing of the data is lost or misplaced using the same logic everything will be sorted after n-1 iterations in the array.

So insertion sort will sort the elements of whatever the input size of an array by traversing the array from right to left in order to find the correct position of its value.

Now we have to make sure that merge will work for arrays of any size. The merge method will take two arrays end merge them in one. It assumes that these arrays are already sorted. The first while loop will run until we reach the end of one of the arrays. During each iteration, it will select the smallest element from both arrays. Therefore, at each iteration array C[k] will have elements less than the elements at A[a] and B[b] arrays since they have not yet been inserted. We suppose that A[k…l] and B[c….d] are the sorted arrays to be merged. a and b are the counters that we will iterate over A and B then: A[a+1…..l] and B[b+1…..d] are all sorted and each element on this subarrays is greater than the elements from C[0….k]. Moreover, after each iteration all elements in C[0….k] are sorted. We should make sure that this are maintained through all the iterations. At first when we start iterating the C array will be empty so it is sorted while the elements of both the left and right subarrays are A and B are sorted too. So all the properties above are maintained. Next on each other iteration, we will pick the smallest element from A[a] and B[b] to place at C[k]. This will guarantee us that C[k] will be sorted while the elements A[a+1….l] and B[b+1……d] are sorted and larger that every element on C. When we stop iterating after we either reach a=l or b=d, we will end up with C[0….k] being sorted, and one of the arrays traversed completely. The remaining part of the unfinished array will be sorted since we have not yet covered it. Therefore, we iterate all over it placing its elements as they are to the remaining positions of C. Since they are already sorted, and we insert them using their original order we end up with a sorted C array of size k=l+d at the end of each call to merge. This proves that merge will function correctly for any input size.

Now we will prove that merge sort works correctly assuming that merge works correctly as it was proven above. For this, I will use induction.

**Step 1:** Prove that it works correctly for input size=1. Since the array of size 1 represents a single element it is already sorted. And we do not even go to the recursive step.

**Step 2:** Assume that it works correctly for input sizes of less than n. Also, merge works fine.

**Step 3:** Prove that it works for any n. Here using the recursive step from an array of size n we will end up to 2 subarrays of size n/2. From the assumption that we made on step 2, merge sort will sort them correctly and merge will merge this two halves correctly too.

This means that merge sort will work fine for any input size n.

Since merge sort works for any input size n, and insertion sort works any input size n it means that when we sort some arrays of size k<log2n using insertion sort we will have sorted arrays of size k, which will serve as input to the merge procedure, which in turn will merge them correctly. Therefore, the whole ins-mergesort can sort arrays correctly. The role of mergesort is to recursively divide the problem of size n to 2 sub problems of size n/2. The role of insertion sort is to sort subarrays of size k and the role of merge is to merge this subarrays returned by insertion sort.

# Bibliography

Cormen, L. R. (2009). Heapsort. In *Introduction to Algorithms* (pp. 151-161). Cambridge,Massachutsetts: MIT Press.

Cormen, L. R. (2009). *Introduction to Algorithms.* Cambridge, Massachusetts: MIT Press.

*Desmos*. (n.d.). Retrieved from desmos.com

*Insertion Sort*. (n.d.). Retrieved from watson.latech.edu: http://watson.latech.edu/book/algorithms/algorithmsSorting2.html

*Merge sort*. (n.d.). Retrieved from http://watson.latech.edu/: http://watson.latech.edu/book/algorithms/algorithmsSorting3.html

*Quicksort*. (n.d.). Retrieved from Math.uic.edu: http://homepages.math.uic.edu/~leon/cs-mcs401-r07/handouts/quicksort-continued.pdf

1. For Heapsort, Mergesort and quicksort I have not provided proof of their correctness since it was covered in class. [↑](#footnote-ref-1)
2. Height h of a node- number of edges from it to the deepest leaf [↑](#footnote-ref-2)
3. For Random Quicksort I have run each file 3 times and took the median number of comparisons outputted.(Line 94 in Quicksort code) [↑](#footnote-ref-3)
4. This is a method of mine for implementing the quicksort. Above I have provided my explanation.(Line 96 in Quicksort class in project) [↑](#footnote-ref-4)
5. There is a small miscalculation in the number of comparisons that ins-merge performs in my code, however the real values are near those in the table. Ins-merge and merge are very near. [↑](#footnote-ref-5)