Assignment

By: Abdulaziz AlOwain

Theory

At an analytical level, Insertion Sort is a direct upgrade to Bubble Sort, it avoids rechecking values that it has already passed through, on a theoretical level however, Insertion Sort does not seem to bring that big of an upgrade, since its time complexity is very similar to that of Bubble Sort.

```
Insertion Sort (N):

for (i = 1 … n)                         n
  max = N[i]                            1
    for (j = i … 0)                     n · n
      if N[j] <= max: break             1
      else: N[j + 1] = N[j]             1
  N[j + 1] = max                        1
```

Code 1: since the 2nd n is nested in the first for-loop, it runs n^2 times.

When implementing Insertion Sort, your main challenge is to preserve its simplicity and at the same time mitigate any unneeded computation. My implementation of Insertion Sort seems to reach both objectives, shown above in the figure "Code 1" is an abstract pseudo-code version of my implementation of Insertion Sort (attached with the submission of the assignment is my implementation of Insertion Sort) you can however choose to view it on GitHub. The worst case complexity of Insertion Sort is $O(n^2)$. Unlike Bubble Sort its best case is O(n). Its space complexity can never exceed O(1).

Merge Sort on the other hand is a very fast and relatively simple, while I found its implementation a bit tedious, it is a good alternative to the more complicated and marginally faster, or the vastly slower and somewhat simpler options.

Implementation Analysis

	10	50	100	1,000	100,000
Insertion	$1.49 \cdot 10^{-7}$	$2.031 \cdot 10^{-7}$	$7.142 \cdot 10^{-6}$	$6.00291 \cdot 10^{-4}$	11.9538
Merge	$6.532 \cdot 10^{-6}$	$3.7386 \cdot 10^{-5}$	$7.695 \cdot 10^{-5}$	$6.611404 \cdot 10^{-4}$	0.088935

Given the above table; it is very clear that the growth rate of Insertion Sort is a lot larger than Merge Sort, taking more than 11 seconds to sort a 100,000 items[1]. While it is very important to note that the constant time spent on Merge Sort is a lot higher, which is why it took was slower than Insertion Sort even at 1,000 items.

1 I recognize that testing for 100,000 items is not a required part of the assignment I thought to include it as it shows clearly the importance of algorithmic analysis; under further curiosity I tried making it sort 1,000,000 items and it took over 19 minutes when Merge Sort finished in just 0.8 seconds! You can imagine how fast it can grow further.

Merge Sort

```
Merge Sort (N):
  L = N[0...n/2]                1
  R = N[n/2...n]                1

  if n > 2                      1
    L = Merge Sort (L)          n · log(n)
    R = Merge Sort (R)          n · log(n)

  for li...l OR ri...r          n / 2
    if L[li] < R[ri]            1
      N[ni] = L[li];            1
      nI, li += 1               1
    else                        1
      N[ni] = R[ri];            1
      ni, ri += 1               1

  for li...l                    n
    N[ni] = L[li];              1
    ni++                        1

  for ri..r                     n
    N[ni] = R[ri]               1
    ni++                        1
```

Code 2

While Merge Sort might seem complicated, breaking it down to its two main functions simplifies it a lot, at its root, Merge Sort divides the array into smaller and smaller arrays, each time getting divided into two halves; which then will be divided into two other halves, and so on till it eventually is broken down into only two elements, only then will the array be sorted and merged with another array that has been broken down into two halves as well.

Understanding Merge Sort is crucial to understanding recursive divide and conquer algorithms, which use Merge Sort as a stepping stone that allows you to grasp the initial required ideas to delve further into such algorithms.

Analyzing the time complexity of Merge Sort may seem complicated at first, through any Algorithm Analysis course you will learn that any call to a function would make the call itself be as complex as executing the function completely, which seems rational; but what about recursive functions?

Taking Merge Sort as an example, it calls itself, so to check its time complexity you go through its execution procedure and before finishing, you call Merge Sort again, and again, and so on... So where do you stop? In Merge Sort you stop when the size of the array given is less than two, when does the array's size reach two? Well, how is the size of the array affected each call; split in half, which means the array is divided by half, then divided by half and so on.. Which explains that it Merge Sort will call itself recursively $\log_2 n$ times, since dividing a number by half then dividing the result by half, is by definition a logarithmic function; as opposed to doubling a number, then doubling it again and so on being an exponential function.

Well then, since Merge Sort is called log(n) times, how much does each call cost; as you can see further down in the algorithm the cost of each execution of Merge Sort below the recursion is n, as such, so each of its recursive calls cost $n \cdot \log(n)$, giving Merge Sort a time complexity of $O(n{\cdot}\log n)$; which when compared with Insertion Sort explains the difference in speed!