**Sorting algorithms.**

In this report we will go over multiple different sorting algorithms to verify their time complexity, and how they compare between each other, the algorithms to review are the following:

1. Bubble sort.

2. Insertion sort.

3. Selection sort.

4. Merge sort.

5. Default sort (standard *sorted* method in python)

**Bubble sort:**

The bubble sort is perhaps the simplest of all sorting algorithms, it iterates through the array and checks if the element on k is bigger than the kth + 1 element on the array, if that is the case, it swaps their values, meaning the value on k + 1 will now be on k, and the value on k will be the one there was on k + 1 before updating it.

Code:

def BubbleSort(arr):

ordered = True

while True:

#Iterate through the array

for i in range(len(arr)):

if i < len(arr) - 1:

#CheckIndex if the positions are ordered

if arr[i] > arr[i + 1]:

#Swap

temp = arr[i]

arr[i] = arr[i + 1]

arr[i + 1] = temp

ordered = False

if ordered:

break

else:

ordered = True

Time complexity:

As the algorithm will always have to scan through the list at least once to verify that all the elements are properly sorted, the best-case scenario for its time complexity will be when the given list is already sorted. As for its worst-case scenario, the algorithm will have to traverse through the list times to compare the elements times while verifying each one of the elements, thus this case’s time complexity will be .

**Insertion sort:**

A slightly more efficient algorithm that compares a given element with the ones previous to its index, so an element on will be continuously compared with this with the recursive function , this will be done until the current value reaches the lower bounds of the array, or it is found that the element is already sorted with respect to its predecessor, as if this is true, we will not need to compare it with the rest of the array.

Code:

def InsertionSort(arr):

for i in range(1, len(arr)):

#Check if it is smaller than the previous one

InsertBackIndex(arr, i)

def InsertBackIndex(arr, index):

for i in range(index, 0, -1):

if arr[i] >= arr[i - 1]:

return

#Swap the elements

temp = arr[i]

arr[i] = arr[i - 1]

arr[i - 1] = temp

Time complexity:

In the best case scenario (already sorted list), the time complexity will be , as its worst-case scenario is concerned, we will have to go through the list times while iterating times, thus, the worst case is .

**Selection sort:**

Here we start to see major improvements to the ways we can sort numbers, selection sort is a very intuitive algorithm that takes the given list, and compares each element on the array with the rest of them (iterating through the array from to to check which number in the subarray is the smallest of them with respect to , if there is a smaller number than the initial point of comparison, we swap the values so that the value on will now be the value on , and the value on will be the value on , this is very similar to the insertion sort, but here we check each position of the array each time we iterate through a new element (we’ll see how this affects time complexity).

Code:

def SelectionSort(arr):

#Get the subarray

for i in range(len(arr)):

currMin = i

for j in range(i + 1, len(arr)):

if arr[currMin] > arr[j]:

currMin = j

#Swap

temp = arr[i]

arr[i] = arr[currMin]

arr[currMin] = temp

Time complexity:

Since we always iterate through the list in a nested fashion, both, worst case and best case scenario for this algorithm is .

**Merge sort:**

Merge sort is a “divide and conquer” algorithm, which means it takes the array, divides it, then processes it by sending the parts of the array to another instance of itself to further divide, once we cannot divide the array any further, we begin sorting, by merging the two parts of the array together, so if we were to have a list that looks like this: , the algorithm splits the array into two equal parts like so: , then it compares the elements on the list and swaps their position if necessary, after this is done for each subarray, we combine the two of them by performing the same comparisons and swapping.

Code:

def MergeSort(arr):

if len(arr) > 1:

mid = len(arr) // 2

left = arr[:mid]

right = arr[mid:]

MergeSort(left)

MergeSort(right)

i = 0

j = 0

# Iterator for the main list

elementIndex = 0

while i < len(left) and j < len(right):

if left[i] < right[j]:

# The value from the left half has been used

arr[elementIndex] = left[i]

# Move the iterator forward

i += 1

else:

arr[elementIndex] = right[j]

j += 1

elementIndex += 1

while i < len(left):

arr[elementIndex] = left[i]

i += 1

elementIndex += 1

while j < len(right):

arr[elementIndex]=right[j]

j += 1

elementIndex += 1

Time complexity:

Both worst and best case scenario for this algorithm is as the division will help us eliminate cases, avoiding it being as many of the algorithms we’ve reviewed.

**Default sort:**

Timsort is a hybrid stable sorting algorithm, derived from merge sort and insertion sort, designed to perform well on many kinds of real-world data.

Code:

randNums = sorted(randNums)

Time complexity:

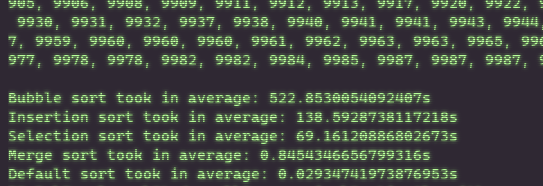
As a derivate of merge sort, this algorithm has an average time complexity of

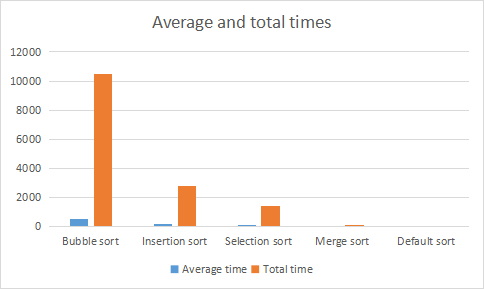
**Results:**

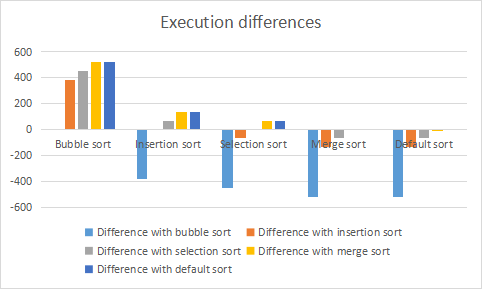
Before presenting the results, we ran into an issue with the amount of random numbers generated, since 100,000 numbers would’ve taken absurd amount of time to sort 20 times (specially with our least efficient algorithm, the bubble sort) in the python programming language, mainly because of its interpreted nature, and overall disregard for memory allocation and cpu usage optimizations, thus we ran the tests under a different load, 10,000, and 30,000 elements over 20 iterations, this means that in total, in Python, each algorithm will have to sort 600000 elements, each array will be different every iteration to ensure the results are as unbiased as possible.

Now, as for sorting 100,000 elements, we did indeed complete this task as well, although the algorithms had to be implemented in a much faster programming language, running in a much more optimized environment (CLR), we’re talking about C#, C# is a compiled programming language that is capable of some amazing performance feats, so that’s what we’ll be using to sort the full list of 100,000 random numbers, the rules are still the same on C#, and the algorithms were implemented in the same way the Python algorithms were to better reflect the real performance of said methods.

Python output with 10,000 integers:

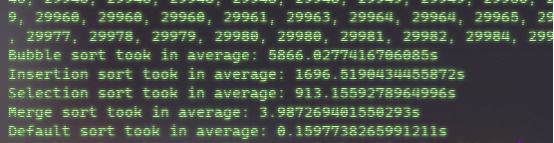


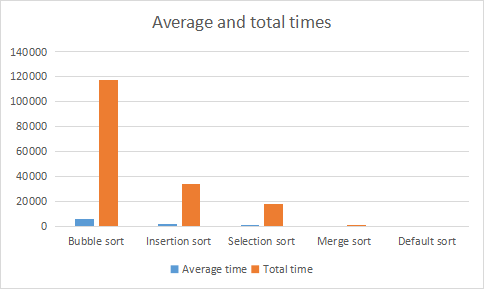


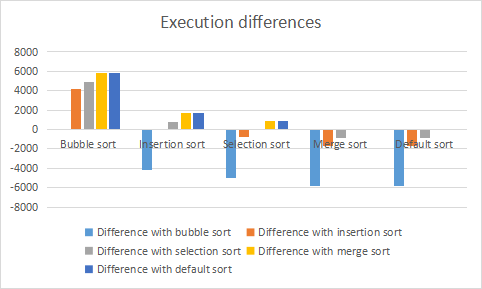


|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Bubble sort | Insertion sort | Selection sort | Merge sort | Default sort |
| Average time | 522.853 | 138.592874 | 69.161201 | 0.84543466 | 0.02934742 |
| Total time | 10457.06 | 2771.85748 | 1383.22402 | 16.9086932 | 0.5869484 |
| Difference with bubble sort | 0 | -384.260126 | -453.691799 | -522.0075653 | -522.8236526 |
| Difference with insertion sort | 384.260126 | 0 | -69.431673 | -137.7474393 | -138.5635266 |
| Difference with selection sort | 453.691799 | 69.431673 | 0 | -68.31576634 | -69.13185358 |
| Difference with merge sort | 522.0075653 | 137.7474393 | 68.31576634 | 0 | -0.81608724 |
| Difference with default sort | 522.8236526 | 138.5635266 | 69.13185358 | 0.81608724 | 0 |

Python output with 30,000 integers:

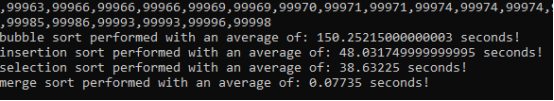


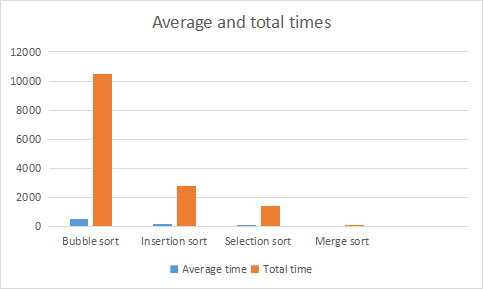


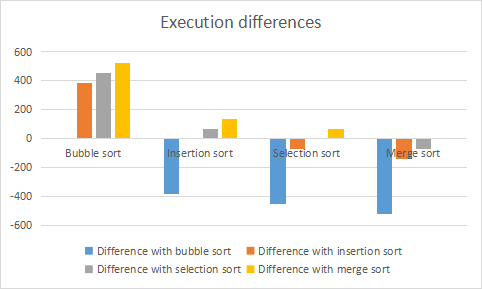


|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Bubble sort | Insertion sort | Selection sort | Merge sort | Default sort |
| Average time | 5866.027417 | 1696.51904 | 913.1559279 | 3.987269 | 0.159773 |
| Total time | 117320.5483 | 33930.3808 | 18263.11856 | 79.74538 | 3.19546 |
| Difference with bubble sort | 0 | -4169.508377 | -4952.871489 | -5862.040148 | -5865.867644 |
| Difference with insertion sort | 4169.508377 | 0 | -783.3631121 | -1692.531771 | -1696.359267 |
| Difference with selection sort | 4952.871489 | 783.3631121 | 0 | -909.1686589 | -912.9961549 |
| Difference with merge sort | 5862.040148 | 1692.531771 | 909.1686589 | 0 | -3.827496 |
| Difference with default sort | 5865.867644 | 1696.359267 | 912.9961549 | 3.827496 | 0 |

C# output with 100,000 integers:







|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Bubble sort | Insertion sort | Selection sort | Merge sort | Default sort |
| Average time | 522.853 | 138.592874 | 69.161201 | 0.84543466 | 0.02934742 |
| Total time | 10457.06 | 2771.85748 | 1383.22402 | 16.9086932 | 0.5869484 |
| Difference with bubble sort | 0 | -384.260126 | -453.691799 | -522.0075653 | -522.8236526 |
| Difference with insertion sort | 384.260126 | 0 | -69.431673 | -137.7474393 | -138.5635266 |
| Difference with selection sort | 453.691799 | 69.431673 | 0 | -68.31576634 | -69.13185358 |
| Difference with merge sort | 522.0075653 | 137.7474393 | 68.31576634 | 0 | -0.81608724 |
| Difference with default sort | 522.8236526 | 138.5635266 | 69.13185358 | 0.81608724 | 0 |