2xc3 Lab Report 4

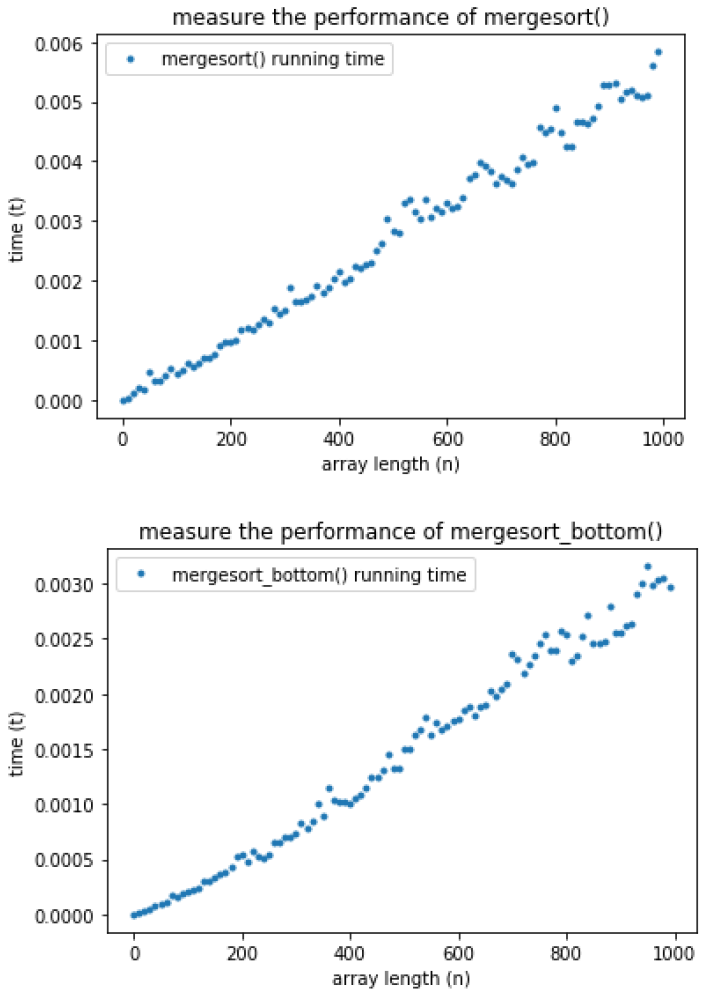
Team Information

Name: Sai Santhosh Thunga
Section: Lab 02
Student ID: 400343455
MacID: Thungas
Email: thungas@mcmaster.ca

Name: Reshmii Bondili
Section: Lab 02
Student ID:400323168
MacID: Bondilir
Email: bondilir@mcmaster.ca

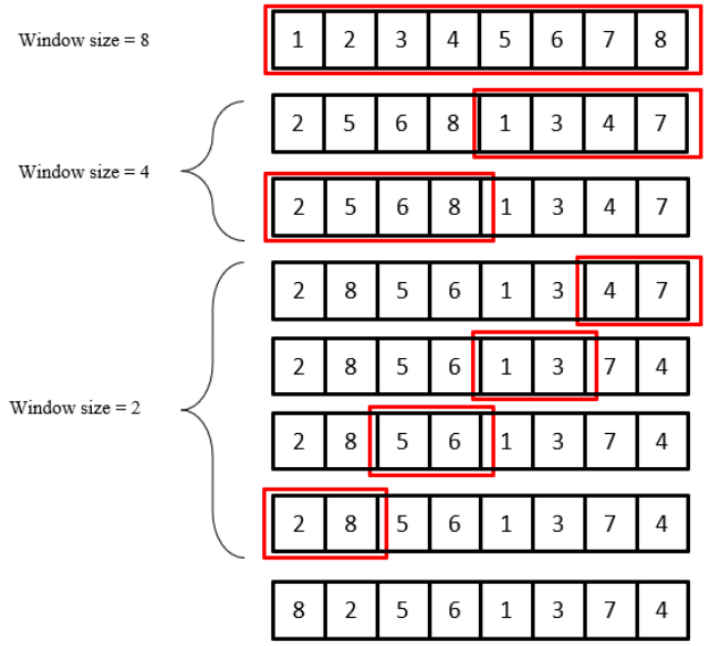
Name: Proyetei Akanda
Section:Lab 02
Student ID:400327972
MacID: Akandap
Email: akandap@mcmaster.ca

Bottom-up [60%]



The first graph is for the mergesort() that was given to us. The second graph is the one we programmed using a bottom-up strategy. As you can see in the bottom-up strategy, mergesort_bottom() is 2 times faster than the mergesort(). At 1000 elements, mergesort_bottom() takes 0.0030 seconds while mergesort() takes 0.0060 seconds.

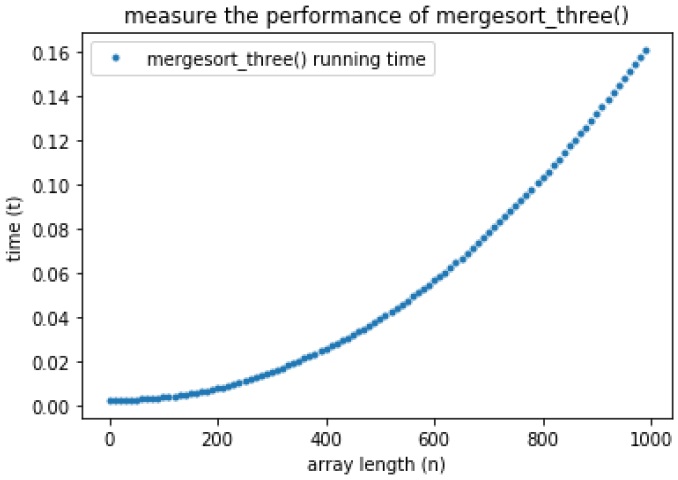
In the bottom-up implementation, we use an iterative approach rather than a recursive approach. We do this by sorting all the subarrays of 1 element then merging the resulting arrays into subarrays of 2 elements. Then, we merge it into subarrays of 4 elements and continue this process so on and so forth until the array is completely sorted. The picture below visualizes this process.

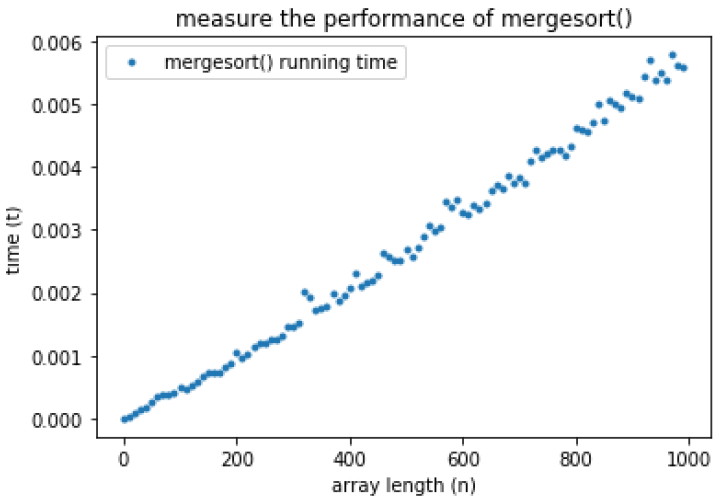


The top-down implementation is the implementation that uses recursion. It starts at the **top** of the tree and proceeds **downwards**, each time splitting the list into two and making recursive calls till you get to the bottom of the tree. Whereas the bottom-up implementation doesn't use recursion. It directly starts at the **bottom** of the tree and proceeds **upwards** by iterating over the pieces and merging them. The implementation of the two types of mergesort explains why top-down mergesort was slower than bottom-up since it uses recursion. Recursion can be slower than iteration in Python due to the fact that it allocates memory each time a recursive call occurs onto the recursive call stack frame which consumes memory making the process slower than bottom-up mergesort.

Three-Way Mergesort [20%]

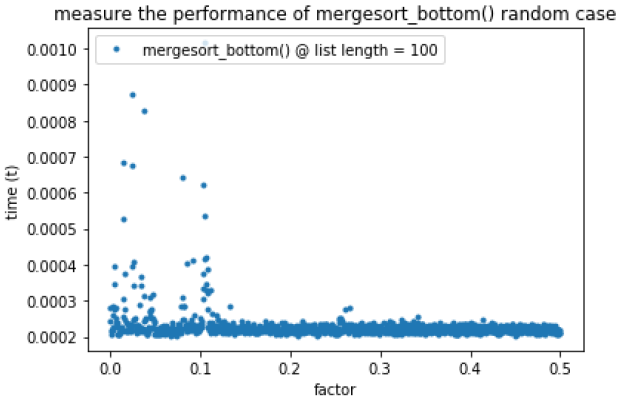
The traditional Mergesort is a recursive function that divides the list into two sub-list of equal parts and sorts them while merging them together to return the final list. A three-way mergesort in theory is supposed to do the same but this time split the list into three equal parts (or atleast close to equal parts) and join them to sort the list. We predicted that a 3 way merge sort (like a 3-pivot quicksort from the last lab) would run faster than a traditional mergesort, however, in practice it had a worse performance in comparison to traditional mergesort. This can be seen in the graph below.

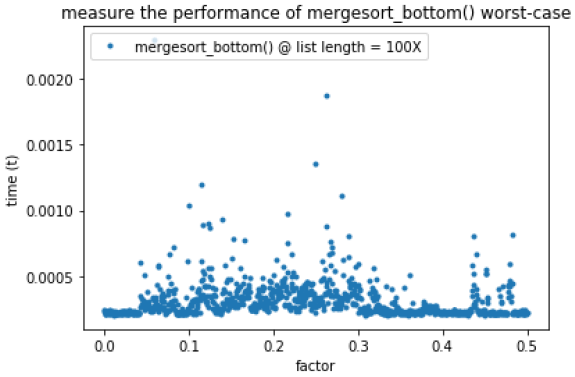




Theoretically, the traditional merge sort algorithm has an $O(n*\log_2(n))$ runtime, while the 3-way mergesort has an $O(n*\log_3(n))$ runtime. Mathematically, we can assert that the $n*\log_2(n)$ is a faster growing function, hence takes more time in comparison to $n*\log_3(n)$. However, in practice the mergesort_three() function is slower due to the increased number of comparisons [in each step] in comparison to the traditional mergesort() algorithm, hence runs slower.

Worst Case [20%]





The above graph was created by keeping the list length constant throughout the graph, but only changing the factor. We found that changing the factor had no real effect in time, and the time taken for various factors [from 0.0 - 0.5] was almost constant. This makes sense as the mergesort (or bottom-up mergesort) has the same the time-complexity for both its worst-case and best-case scenario which is $O(n*\log_2(n))$.

Therefore, providing a list when it’s nearly sorted or completely sorted will still give the same runtime as a random list.