Divide and Conquer Sorting Algorithms

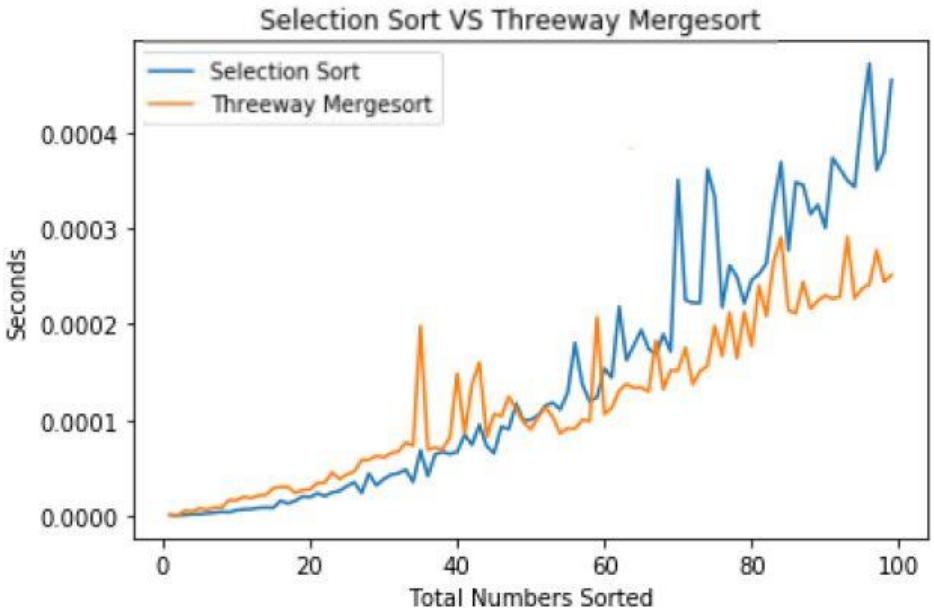
Usama Puri

Minerva Schools at KGI

Divide and Conquer Sorting Algorithms

Justification for the Threshold value

The justification for the threshold value of 40 that I used in this assignment for the extended merge sort algorithm can bee seen in the graph below



In this graph we can see that when the function has to sort more than 50 numbers it is better to do it with a three-way merge sort than a selection sort as it takes more time to sort that list. This can be explained by considering that selection sort requires less time to sort the same array as long as the length of the array is approximately 50. After that, we can see in the graph that between the values 40 and 60 the selection sort graph gets steeper and crosses the merge sort graph. This means that after 40 value it is more efficient to use merge sort because it requires less time to sort the same number of integers. But we cannot generalize this by looking into one example only, we also have to take into account the best or worst-case scenarios for these sorting algorithms which I will discuss in the later section.

Complexity Analysis for Merge Sort, Three-way Merge Sort and Extended Merge Sort

Merge Sort:

A regular merge sort algorithm applies a divide and conquer approach. This rule can be explained into three parts:

1. **Divide:**

This part of the approach divides the array from the midpoint into two further subarrays. This takes a total computational time of O(1).

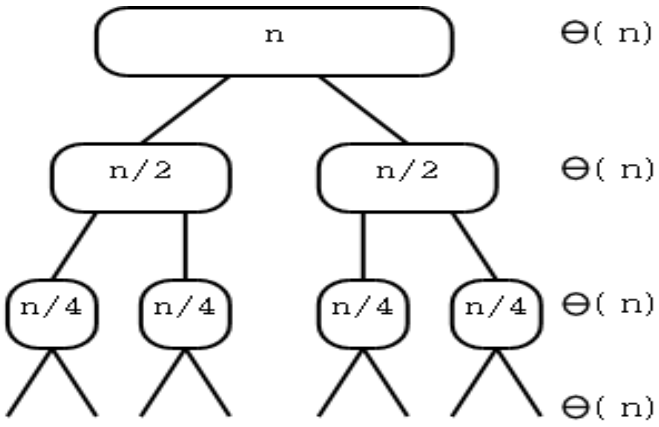
2. **Conquer:**

This step recursively sorts the subarrays that were made after division, each of size n/2, where n is the total number of elements in the array. This takes the total run time 2T(n/2).

3. **Merge:**

The is the last step of the algorithm and finally merges the N number of elements back into the parent array. This takes a total run time of O(n).

After combining all these steps, we can create a recurrence equation for merge sort. Which is $T(n) = 2T(\ n\ 2\ ) + \theta(n)$. We can use this recurrence equation to build a recurrence tree and then analyze how many time these steps are executed by the algorithm.



This graph has been copied from Google Images

From top to bottom, the array and subarrays are divided further into two parts ($cn/2$). Here if we analyze level 2 the complexity is $2x(cn/2) = cn$, at level 3 it is $4x(cn/4) = cn$ and this will go on depending on the number of elements in the array.

Then we will analyze the height of the tree. We can see that it is increasing according to the expression $log2n$. The total height of the tree then can be expressed by the following expression $log2n + 1$ for a given n. Therefore, we can compute the overall complexity of merge sort to be $(log2n + 1)xcn$. This gives $O(nlog2n)$.

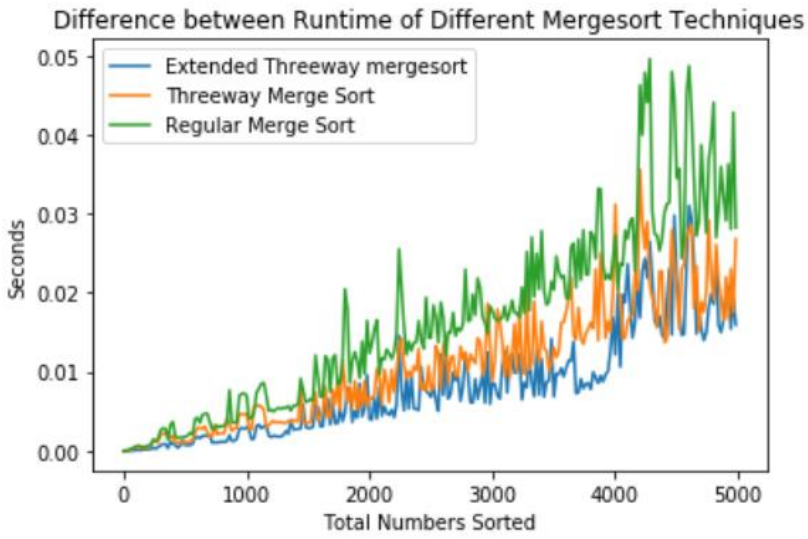
Threeway Merge Sort.

We will now analyze the threeway merge sort the same way we did the regular merge sort. The difference is that a threeway merge sort divides the array into three equal parts instead of two. Its tree is similar to the one we used for regular but each node is divided into further 3 nodes at every level the subarray. So, we can compute the recursion equation for this algorithm the same way we calculated the equation for regular merge sort.

The recursion equation for this algorithm will be $T(n) = 3T(n/3) + O(n)$. And by solving the equation the same way we did in regular merge sort we arrive at the time complexity of of $O(nlog3n)$. It is important to note that three-way merge sort has a higher space complexity because of the division of subarrays into 3 parts.

Extended Merge Sort.

For analyzing the extended merge sort algorithm, it is helpful analyze the graph of its difference with respect to the 2-way and three-way merging algorithms.



Difference between runtime of different Merge sort techniques.

This graph shows the difference of algorithms run-time with respect to the total numbers in the list that are being sorted. We can see that Extended merge sort works better than 3- way merge sort which works better than regular (Two-way) merge sort. This topic is also discussed in the above section and we can see that selection sort works better than merge sort for lower values of n. In this case, that value is set to 50.

The extended merge sort runs selection sort for small values of n because it is faster than merge sort. So we divide the main array into smaller subarrays such that it reaches the threshold value i.e. 40, and, on those subarrays, we implement selection sort. And then we merge the sorted lists back. We used three-way merge sort and selection sort together.

We already know that the time complexity of three-way merge sort is $O(nlog3n)$. For selection sort, it has two scenarios, Best Case and Worst Case. In the best case scenario the time complexity of selection sort is $O(n)$ and in the worst case scenario, it is$O(n\ 2\ )$. That gives us a total complexity of:

Best Case: $O(n + nlog\ n\ x\ )$.

Here n is the main array or list and x our threshold value

Worst Case: $O(\text{n}x + nlog\ n\ x\ )$

Here n is the main array again and x again is our threshold value.

With our specific case. We are using threshold value = 40. This makes the selection sort cost as O (40^2) in worst case scenario not O(n^2). Therefore, as n continues to increase the computational cost of selection sort remains constant. And instead of having a recurrence for ($nlog3n$ + 1) we are now saving x levels, which is our threshold value. All these factors combined gives us a higher efficiency from the extended merge sort function where extended merge sort has the least steep line.

Bucket Sort

Bucket sort is kind of different form other sorting algorithms. In this algorithm we generate a list of n number of lists and call them as our buckets. We then append elements in those buckets using a certain range. These list now contain some elements in one range. After all the elements have been distributed to buckets. We person Selection Sort with worst time complexity O(n^2) and sort the elements within the bucket. After they are sorted we append all these small lists into a big results that is sorted through bucket sort

Bucket Sort assumes that the element is drawnfrom a uniform distribution. The average time complexity for bucket sort involves the number of buckets being created. This bucket sort is very fast because of the way elements are are assigned to the buckets. The average time complexity of bucket sort is O(n+k). Whereas, the worst-case time complexity is O(n^2) in case all elements end up in same bucket

HC Used:

#breakitdown:

The merge sort algorithm works on the principles of this HC. However, the extended merge sort helped me understand how we don't always have to condense the problem to the base case, instead setting a threshold and breaking it down to that threshold can help make our solution more optimized.

#dataviz:

The graph comparing merge sort and selection sort makes it more understandable for us when the threshold value is reaches and how both the function behaves with increasing values of inputs. This helps us better make the decision and explain which method is better for specific cases.

#testability:

In the code by plotting more than one graphs and similarly in the assignment as well by reiterating the results helped me confirm the durability of my code. Since I used random number generator and there are so many nuances involved in the sorting algorithms, there always used to be one list that didn't give the right sorted output. After number of reiterations I was able to generate a code without any bugs and will sort any universal array.