CS/SE 2XB3 Lab 4 Report
Enrolled in CSL02

Wang, Mingzhe
400316660
wangm235@mcmaster.ca

Li, Xing
400292346
li64@mcmaster.ca

Moon, Hyosik
400295620
moonh8@mcmaster.ca

February 19, 2021

Contents

1 Bottom-up

We implement a helper function merge_bottom(L, start, mid, end) to merge the sub-array L[start:mid] and L[mid: end]. We follow the python convention to include the left end and exclude the right end.

No recursion are used. We use only while loop and for loop. To deal with the issue that the list length may not be the power of 2, we use min() function to choose the minimum element between end index and n. We also compare middle index and n. If middle index is already greater than n, no bottom-up sort is needed.

To compare the bottom-up implementation to the original top-down algorithm in "lab4.py," we run both algorithm for 5 times and take the average run time as the base to compare. Then we run both algorithm with different lengths of list from 100 to 10,000, with a step of 100. There is convincing improvement on the run time with an average speed-up of 19%.

As the merge part is the same in both algorithm, we guess that the difference may come from the lower space complexity in the bottom-up algorithm. The additional space required starts from 1 and doubles after each round of merge. Therefore, the average additional space usage is only $(\sum_{i=0}^{log_2 n} 2^i)/log_2 n \approx 2n/log_2 n$. In comparison, the merge sort space complexity is O(n). Less space requirement means less memory access, and this can result in the better performance in our case.

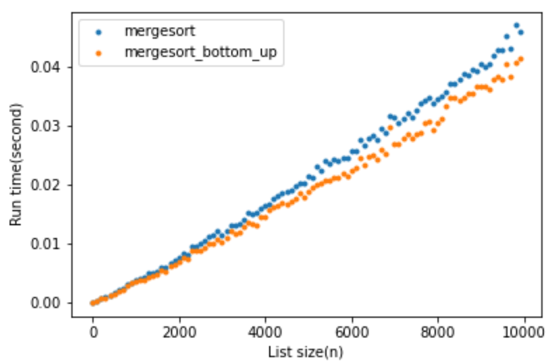


Figure 1: Bottom_up vs Top_down

2 Three-Way Mergesort

2.1 Prediction Before the Experiment

Before starting testing `mergesort_three` and `merge_three`, we predict that `mergesort_three` will perform no better or worse than `mergesort`. The reason is that we can define a recurrence $T(n) = 3 * T(n/3) + O(n)$ for calculating the time complexity of `merge_three`. By master theorem, the time complexity of `merge_three` is $T(n) = O(n \log_3 n) = O(n \log n)$, which is the same as the time complexity of `mergesort` $T(n) = O(n \log n)$.

2.2 Compare the performance

For comparing the performance of `mergesort_three` and `mergesort`, our testing method is consistent with the previous. Specifically, the list size n is from 0 to 20000 with an interval 200, and for each n, we run the test 5 times to decrease the noise from other factors.

Our observation is that in average `mergesort_three` is 16.28% faster than `mergesort`. In addition, we observe that `mergesort_three` is relatively more faster than `mergesort` when the list size is around $15000 - 20000$. The test result is showed in Figure 2.

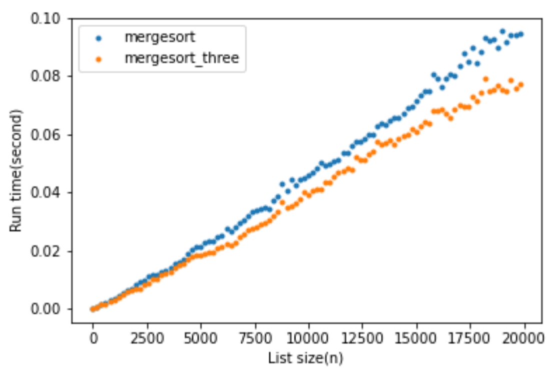


Figure 2: Mergesort vs Mergesort_three

2.3 Result Explanation

The experiment result shows that our prediction is wrong – `mergesort_three` actually performs better than `mergesort`. To explain this observation, we analyze these two algorithms, some of our perspectives are as the following:

Because the experiment result is the average time complexity, we cannot simply use Big-O notation to predict it. The actually running time of `mergesort_three` should meet this recurrence $T(n) = 3 * T(n/3) + T'(n) = T'(n) \log_3 n$, where $T'(n)$ is the actual time complexity of `merge_three`.

We assume that for each recursive call of `merge_three`, the time consumed at the beginning (i.e. things like `l = len(L) // 3`) can be omitted because they are not corresponding to list size. In addition, we assume all operations need constant time t.

Then, in our implementation of `merge_three`, for each position of the merged list consisting of three subarrays, in the worst case, it needs 4 comparisons (i.e. `i >= len(left)` or `left[i] <= right[k]`), 1 `append` (i.e. `L.append(left[i])`), and 1 increment (i.e. `i += 1`). Based on this, the actual running time `mergesort_three` needs is $6n \log_3 n = \frac{6}{\ln 3} n \ln n$. Similarly, because in the worst case there are 3 comparisons, 1 `append` and 1 increment in `mergesort`, the actual running time of `mergesort` is $5n \log_2 n = \frac{5}{\ln 2} n \ln n$. Therefore, due to the fact that $\frac{6}{\ln 3} \approx 5.46 < 7.21 \approx \frac{5}{\ln 2}$, the actual running time of `merge_three` should be faster than that of `merge`, which verifies our observation.

2.4 Another Consideration

We have also noticed that the professor's version of `mergesort` is not the "traditional" version of `mergesort`. Its running time is not optimal because in the `merge` method this version of implementation performs redundant comparisons when one of the subarray being merged is exhausted. And if we improve the professor's version of `mergesort` to the real "traditional" version of `mergesort`, the performance of `mergesort` (improved prof's version) and `merge_three` can be relatively similar as showed in Figure 3.

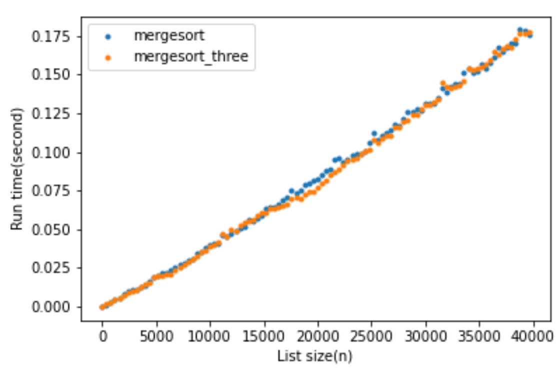


Figure 3: Mergesort vs Mergesort_three

2.5 A Fun Fact

A fun fact is that as showed in both Figure 2 and Figure 3, `merge_three` perform even better when the size list is around $15000-20000$, for which we need to do more researches to provide a reasonable explanation in the future.

3 Worst Case

3.1 Best mergesort

Based on the previous experimental facts we thought that `mergesort_three_bottom_up` will be the best.

Fact 1. `mergesort_bottom_up` is better than `mergesort_top_down`. (Figure 1)
Fact 2. `mergesort_three_top_down` is better than `mergesort_top_down`. (Figure 2)
Fact 3. `mergesort_bottom_up` is better than `mergesort_three_top_down`. (Figure 4a)

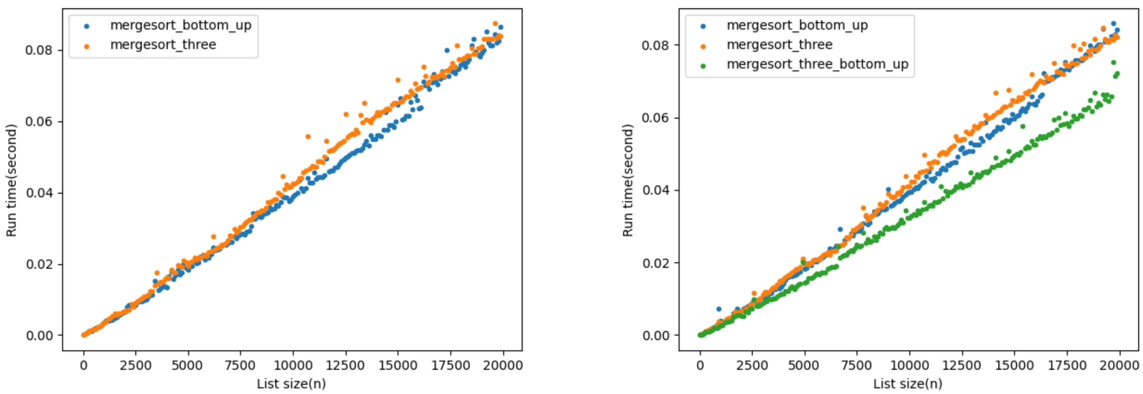
Figure 4b is the result comparing three algorithms, and as we expected `mergesort_three_bottom_up` showed the best performance. We ran the algorithm 5 times and took the average run time with different lengths of list from 100 to 10,000, increasing the step of 100 each time. There is convincing improvement on the run time with an average speed-up of 22.346%.

This is because `mergesort_bottom_up` has a time complexity of $O(nlog_2n)$. On the other hand, `mergesort_three_bottom_up` has a time complexity of $O(nlog_3n)$, which is faster than $O(nlog_2n)$. This means that `mergesort_three_top_down` can improve the performance by $log_2 3 - 1 \approx 58\%$ in terms of the append and assignment (app_ass) operation.

For the comparison operation, `mergesort_three_bottom_up` requires two comparison to merge three sub-arrays, while `mergesort_bottom_up` only requires one comparison to merge two sub-arrays. In total, `mergesort_three_bottom_up` has a time complexity of $O(2nlog_3n)$, and `mergesort_bottom_up` has a time complexity of $O(nlog_2n)$. This means that the performance of `mergesort_three_bottom_up` is $1 - 0.5 * log_2 3 \approx 21\%$ worse in terms of append operation.

Overall, we have a time saving of $(nlog_2n - nlog_3n) * T_{app\_ass} - (2nlog_3n - nlog_2n) * T_{compare}$.

As long as the time of app_ass operation is $\geq \frac{2ln2-ln3}{ln3-ln2} \approx 0.71$ of the time of compare operation, the overall performance of `mergesort_three_bottom_up` will be better than `mergesort_bottom_up`.



(a) Bottom_up vs Three_Top_down

(b) Bottom_up vs Three_Top_down vs Three_bottom_up

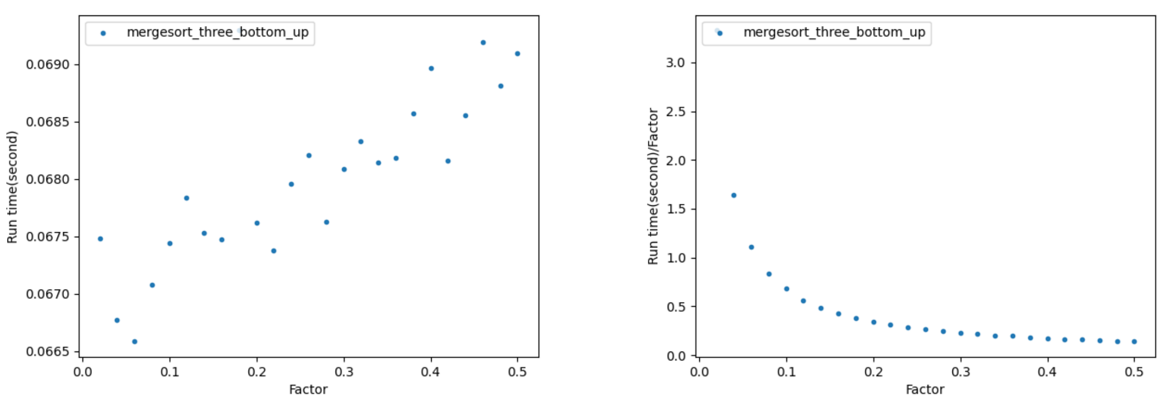
Figure 4: Comparison for best algorithms

3.2 Best mergesort vs Factors(sorted)

Under different sorted fators we obtained the average run times. We ran the algorithm 30 times with the length 20,000 list and took the average run time with different factors from 0.02 to 0.5, increasing the step of 0.02 each time. In Figure 5a, it seems to be a linear function, but when we divide the run time with factor, we can see that it is a power function, which is $0.0691x^{-0.99}$ (Figure 6). It means that regarding sorted factors the `mergesort_three_top_down` algorithm with the fixed length of 20,000 has a time comlexity of $0.0691x^{0.01}$, which is a irrational function.

We interpreted it as due to the time complexity of the comparison part of the algorithm. But the result was slightly different from our expectation. Because we thought that when a list is more sorted, it will increase the run time because the relatively complicated comparison part of the algorithm will be implemented more frequently.

However, as the time complexity of the algorithm is $0.0691x^{0.01}$, which is extremly small, we can conclude that the mergesort is not affacted by the sorted fatocrs. It means that the worst case time complexity of mergesort is $O(nlogn)$.



(a) Runtime
(n = 20,000, factor = 0 to 0.5)

(b) Runtime/factor
(n = 20,000, factor = 0 to 0.5)

Figure 5: Comparison under different sorted factors

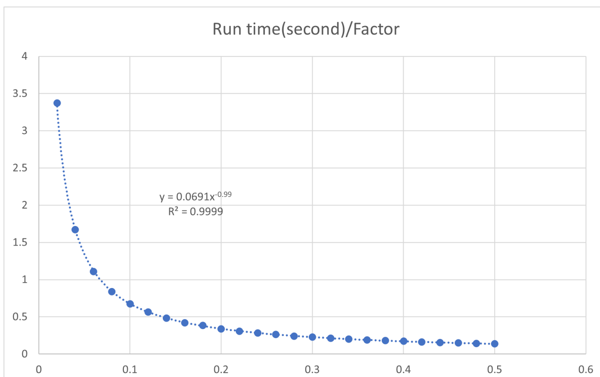


Figure 6: Runtime/factor
(n = 20,000, factor = 0 to 0.5)