# CSCI 220 - Data Structures and Algorithms: **Project 2: K-Way Merge Sort Analysis**

Patrick Maes

November 15, 2022

## Part A

In order to analyze the k-way merge sort algorithm, start by finding the complexity of merging k vectors. In order to merge k vectors, we need k comparisons to find minimum element, assuming the vectors are sorted. This is because we only need to compare the unpushed elements farthest to the left in each vector, since they are the minimum of each.

Likewise, we also need to find the minimum element n times before all the elements are placed into the final sorted vector. It is easier to see if we say each vector has $\frac{n}{k}$ elements and we have k vectors total, and we have to move all the elements.

Therefore, if T(n) can be represented as $T(n) =< subproblemCount > T(subproblemSize)+ < mergeComplexity >$, then

$$T(n) = kT(\frac{n}{k}) + O(kn) \tag{1}$$

where k is the number of vectors the original is split into, and n is the size of the original vector. The number of subproblems is always k since this is how many vectors we are splitting and merging. The size of the subproblem is $\frac{n}{k}$ since each of the split vectors will be of this approximate size. These vectors may not be of size 1, so the T() represents running the recursive function again until the base case. Finally, the merge complexity of my algorithm is $kn$.

I know that my merge complexity is $kn$ since I iterated over the two dimensional vector container and found the maximum element n times. In each case, we are only comparing the elements at the ends of the vector (depends on if you find minimum or maximum element). There are always k vectors in the 2 dimensional vector, so we need k comparisons to find the maximum element. Since we need to find the maximum element n times with k comparisons each time, the total comparisons is kn.

Therefore, the complexity of my merge algorithm is $kn$. I use this in the tree as the cost of each level of the total merges. We can prove that the cost is constant since we are performing the merge k times on arrays of size $\frac{n}{k}$. This means the number of arrays increases by a factor of k while the size decreases by a factor of k. This means we just have to multiply the cost of merging, $kn$, by the number of levels in the tree. Since we split the vector in k parts, we need to split $log_k(n)$ times before we can start merging. The total complexity can be written as

$$C_{total}(n, k) = knlog_k(n) \tag{2}$$

If we were to choose values of k that were large relative to our list size, it probably would not be more beneficial. As $k \to n$, the complexity would essentially be the complexity of merging the elements in the split vector, since the log term would approach 1. As a result, we would just have to compare k elements n times to get the final vector. However, as $k \to n$ the complexity would approach $n^2$ which is not better than $knlog_k(n)$

Based on the answer as $k \to n$, I think that my algorithm is most efficient when k is either 2 or 3. This is because we need to minimize the growthrate of $k$, since it grows faster than $log_k(n)$. In order to find an optimal value of k, i first started by taking the derivative of our complexity.

$$\frac{\partial}{\partial k}(knlog_k(n)) = nln(n)\frac{\partial}{\partial k}(\frac{k}{ln(k)})$$
$$\frac{\partial}{\partial k}(\frac{k}{ln(k)}) = \frac{\frac{d}{dk}(k)ln(k) - k\frac{d}{dk}(ln(k))}{ln^2(k)}$$
$$\frac{\partial}{\partial k}(\frac{k^2}{ln(k)}) = \frac{ln(k)-1}{ln^2(k)}$$

Substitute this back into the original to get the derivative of the total complexity.

$$\frac{\partial}{\partial k}(k^2nlog_k(n)) = \frac{nln(n)(ln(k)-1))}{ln^2(k)}$$

Now we can see that the complexity will be minimized when the function $\frac{ln(k)-1}{ln^2(k)}$ is at a 0. No matter the value of n, a value of zero from this function means that there is a slope of zero at this location of k. If the slope also goes from negative to positive, this relative point is a minimum.

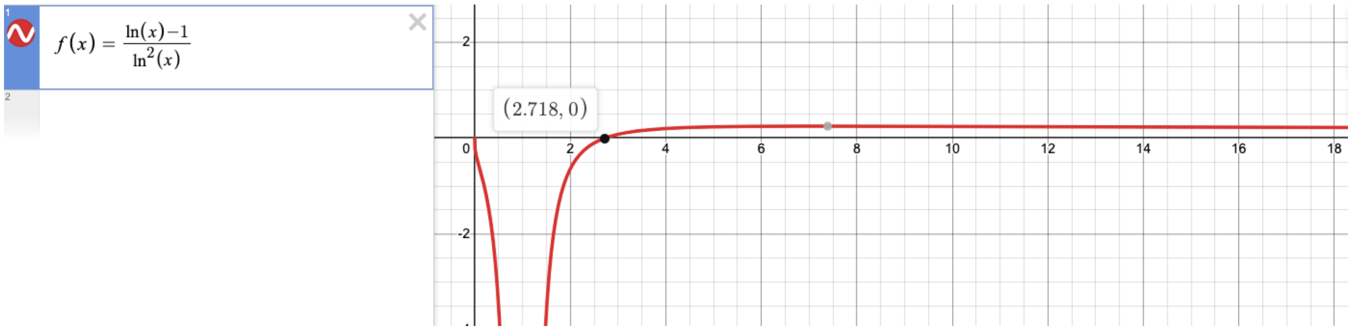


Figure 1: Plot of the derivative of complexity only as a function of k.

From this plot, we can see that the most efficient value of k is at 2.718. This is where the slope of the complexity is zero. Mathematically, I would usually have to test the endpoints near the critical point on either side to determine what kind of maxima it is. However, from this plot it is easy to see that the slope changes from negative to positive, indicating a relative minimum. The plot also validates my answer for larger values of k where k = n. A large k would not be better since you can see the slope is always positive from k = 2.718 to k = $\infty$. Since the slope is positive from the relative minimum, I don't think the complexity would be better than the relative minimum. Instead, the most optimal value of k is around 2 or 3, which is the closest integer to the most efficient value of k. In order to verify, I also plotted $\frac{k}{ln(k)}$:

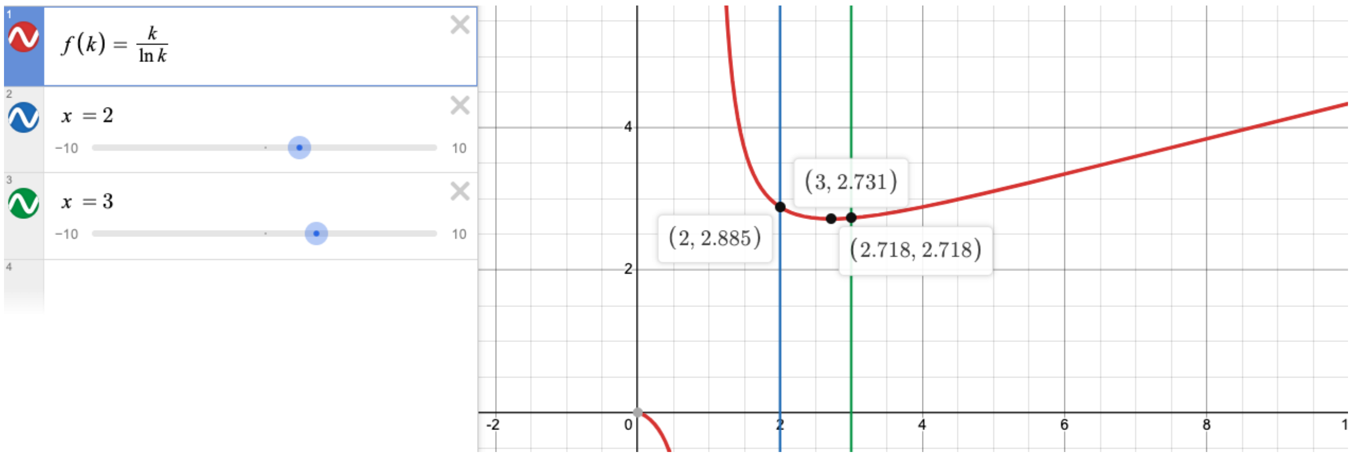


Figure 2: Plot of the complexity only as a function of k.

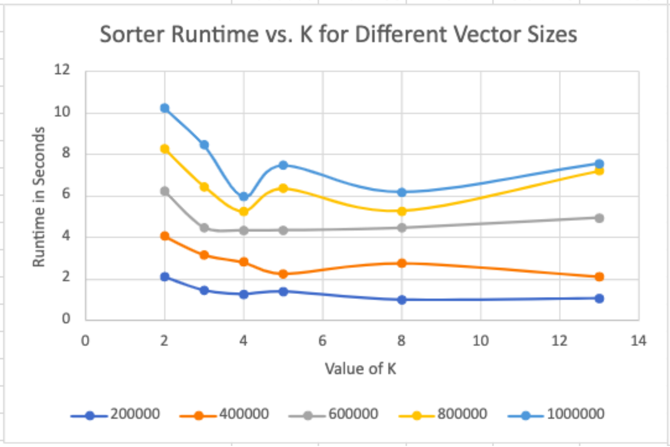
This appears to be the only minimum for a merge complexity of $kn$. Since the derivative is always positive after the relative minimum of 2.718, the complexity is always increasing after this point. This supports my hypothesis that the complexity would grow to $n^2$ when $k \to n$. The true minimum would be k=3 since it's complexity is less than k=2, which are the closest integers to the decimal minimum.

## Part B

In order to attempt to verify my results from part A, I timed the k-way merge sort algorithm for different vector sizes and values of k. Unfortunately, my results at first contradicted my analysis. I used the vector sizes of 20,000 to 100,000 at first. However when I would run the code, the times became faster as k increased in size. As written in part A, the complexity should be greater as k increases. However even with very large input sizes, I am still not quite sure why the results are not really noticeable.

I was advised to try increasing the vector sizes by a factor of 10, and add another k value that followed the given sequence. After this, the trend was slightly visible, but the results were still not clear. In the data table below, the left column is the number of elements in the vector we are sorting and the top row is the value of k. Each value in the table between these is the resulting runtime for the given combination in seconds.

| | 2 | 3 | 4 | 5 | 8 | 13 | 27 |
|---|---|---|---|---|---|---|---|
| 200000 | 2.0806 | 1.4382 | 1.255 | 1.385 | 0.988 | 1.054 | 0.9951 |
| 400000 | 4.0234 | 3 | 2.7766 | 2.2244 | 2.7335 | 2.0866 | 4.6242 |
| 600000 | 6.2113 | 4.4676 | 4.3346 | 4.3374 | 4.4511 | 4.935 | 5.6233 |
| 800000 | 8.2644 | 6.4453 | 5.236 | 6.3388 | 5.268 | 7.1839 | 6.5384 |
| 1000000 | 10.2188 | 8.422 | 5.94429 | 7.4567 | 6.1752 | 7.5552 | 7.5369 |



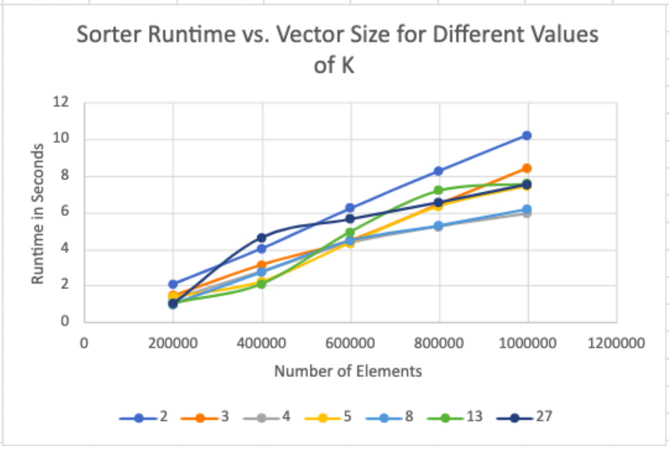


Figure 3: Plot of experimental runtime vs k for different vector sizes (left) and experimental runtime vs vector size for different values of k (right).

The above graphs represent the actual runtime of the program with two visualizations. It is still a little hard to tell the best value of k from these plots. Around k = 8 there are fluctuations in the complexity, mostly decreases. However, for some input sizes the larger values of k were not too bad, such as 200,000 elements (the bottom line, left graph). I did make sure that k=27 is included in these plots, but it seemed too far out of the domain for the left plot and would have made it hard to read. However on the right graph, you can see that k = 27 is not particularly good for all input sizes.

Judging just off the right graph, the better values of k are 4 and 8. These are the grey and light blue lines on the right plot. I mostly judged this off the greatest sized input, since we will probably see more consistant values for larger vectors. The complexity for k = 4 and 8 are the least for a vector of 1 million elements. However, these lines are also relatively good for the rest of the input sizes. Most of the other k values have at lease one place where the complexity is greater than the rest. An ideal value of k would need to be efficient for all the input sizes, including very large sizes. Based off the graphs alone, I would say the most efficient experimental value of k is 4. This is also very clear in the left graph since most of the input sizes have a dip at k=4.

## Part C

In order to sort queues, I don't think we would have to change the code too much. The main difference would be the merge step, where we would find the minimum value and move it to the next position in the final queue from the front.
Splitting the vectors would almost be exactly the same. I would make k queues, all to be stored in one large vector. While the original queue is not empty, then I would move one element to each of the k queues until we run out of elements. At the end we should have distributed as evenly as possible, similar to passing cards to 1 player at a time until the deck runs out. The queues are not guaranteed to hold the same number of elements. Then we recursively split the queues until they are of size 1.
After this, we can merge the queues once they only hold one element each. Queues can only add elements

to the back, and remove from the front. I would start by checking the front of all k queues to find the minimum. Then I would enqueue this element to the back of an empty queue. I would then continue finding the minimum and enqueing it to the final queue until the 2D vector of queues is empty. Then the final queue should be sorted since the minimum values were added in order to the back of the queue.

I think that the complexity of sorting the queues this way would be exactly the same. We are splitting the list into k queues, meaning we need $log_k(n)$ splits to reach the base case. Then, at each level we need to merge the queues together. This will still be kn complexity at each level since we need k comparisons to find the minimum element and we need to move n elements to the final queue.

Nothing about the operation complexity has changed either. Enqueue and dequeue are both O(1) operations, similar to $pop\_back()$ and $push\_back()$. Essentially when sorting vectors, the most efficient way to simultaneously merge k vectors is to use $pop\_back()$ after finding the greatest element. In the queue example, we can only remove from the front, so simply find the minimum value instead. Similarly, one way to split the vectors is to use $pop\_back()$ and just fill the number of vectors you need with the correct number of elements. We can do the same thing, except we don't know the size. Instead we would just implement a while loop to check if the original queue is empty or not.
Therefore, the complexity of sorting queues using k splits would be $O(knlog_k(n))$ where n is the size of the queue given.