605.202 Data Structures

Author: Ivan Sheng

Lab 4 Analysis

Due Date: 11/27/2019

Date Turned In: 11/26/2019

Design:

There are five classes used to sort with natural merge sort, and the four quicksort variations.

- The NaturalMerge class is the driver that executes the code for the natural merge sort. It is implemented with linked lists and utilizes the QueueList class to store the input data and it will help to store the partitions.
- The QuickSortFirstPivot is the class driver that executes the code for the quicksort with first item pivot. It is implemented with arrays to store the input data. It also utilizes the StackList class to store the start and end points of the partitions for backtracking purposes.
- The QuickSort3Mpivot class is the driver that executes the code for quicksort with the median of three as the pivot. It utilizes the same structures as the QuickSortFirstPivot.
- The QuickInsertionSort50 class is the driver that executes the code for the hybrid quicksort/insertion sort for a stopping case of a partition size of 50. It utilizes the same structures as the QuickSortFirstPivot.
- The QuickInsertionSort100 class is the driver that executes the code for the hybrid quicksort/insertion sort for a stopping case of a partition size of 100. It utilizes the same structures as the QuickSortFirstPivot.

Iteration vs Recursion:

My design implemented both algorithms as the iterative versions. One major difference between the two versions is that the recursive version requires overhead room to hold the stack of recursive calls. However, the recursive code of both algorithms would be better in terms of readability and length than the iterative counterpart. While we don't need to account for recursive overhead via iteration, algorithms like quicksort could only have been done by implementing a stack to handle the partitions. Similarly, in natural merge sort, linked lists were used to store the partitions of each iteration.

Simulation Procedure:

For the natural merge sort algorithm, it attempts to partition the input list into pairs of lists that will be sorted and then combined together. This partition and sorting strategy continues until the list is fully sorted. The reasoning behind this strategy is that it'd be easier to sort a partially sorted list than an unsorted random list.

For the quick sort algorithm, a pivot is selected, and the list is partitioned into two branches - one that contains items larger than the pivot, and another that contains the items that are smaller than the pivot. This pivot can be selected in many ways, but for this lab specifically, the first item of the list and the median of the first, middle, and last items were used as pivots. As the algorithm runs, the start and end indices of each partition are pushed into a stack to simulate the backtracking recursion stack. So long as the algorithm hasn't reached it's stopping case or the end of the pair stack, then it will keep sorting.

There were four variations of quicksort for lab #4:

1. First item pivot, 1 and 2 partition stopping case
2. Median of Three pivot, 1 and 2 partition stopping case
3. First item pivot, 50 partition stopping case
4. First item pivot, 100 partition stopping case

Quicksort Efficiency:

All four variations ran for a random order, ascending order, and reverse order list in the sizes of 50, 1000, 2000, 5000, and 10000.

Comparing variations one and two, expectations are that more often than not, the median of three pivot would perform better than the first item pivot. The reason being is that although we take up time to calculate the pivot, we can achieve an overall efficiency of O(nlogn) for the best, worst, and average case scenarios. This prevents us from experiencing a worst-case scenario of $O(n^2)$, but will prevent us from achieving the best-case scenario or O(n). Looking at the resulting timed executions, we can see that the average runtimes for the random order list are comparable. However, when we got to a file size of 20,000, the runtime of median of three pivot doubled in comparison to the first item pivot. This could have been due to variance or possibly that the random order favored the first item versus the median of three result as the average throughout all six runs are close.

Random time (ns)		
File Size	QuickSort First 1&2	QuickSort Med 1&2
50	44,700	40,200
1000	1,059,900	1,096,700
2000	1,611,100	1,275,700
5000	2,649,000	1,984,600
10000	5,140,800	3,511,900
20000	6,419,300	7,116,700
Average	**2,820,800**	**2,504,300**

However, moving on to the reverse and ascending order lists, quicksort experiences the worst-case scenario of $O(n^2)$: when the pivot is an extreme - either the smallest or largest value of the list. This is when the median of three pivot significantly trumps the first item pivot: it maintains O(nlogn) efficiency. When comparing the runtimes of the two differing pivots, experimentally, the median of three pivot outperforms the first item pivot by over a factor of ten on both sorts. The runtime of median of three when sorting larger numbers still maintains its behavior of doubling.

Ascending		
time (ns)		
File Size	QuickSort First 1&2	QuickSort Med 1&2
50	114,700	37,800
1000	3,938,600	824,800
2000	7,451,300	1,090,800
5000	17,943,400	2,003,800
10000	28,144,600	3,973,600
20000	80,817,200	6,105,400
Average	**23,068,300**	**2,339,367**

Reverse		
time (ns)		
File Size	QuickSort First 1&2	QuickSort Med 1&2
50	63,600	39,100
1000	4,112,300	932,700
2000	8,666,100	1,089,600
5000	16,932,000	1,675,000
10000	49,186,400	2,984,200
20000	177,064,400	6,221,700
Average	**42,670,800**	**2,157,050**

Hybrid Sorting:

Moving on to the QuickSort with Insertion sort after a partition of size 50 versus a partition of size 100, we observe that the reverse order list is the worst performing sort. This is because both quicksort and insertion sort run into their worst-case scenarios of $O(n^2)$ since the pivot is the largest extreme, and the order is reversed - maximizing the number of comparisons for insertion sort. This would also result in significantly worse performance than Quicksort with a first item or median of three pivot.

Reverse				
time (ns)				
File Size	QuickSort First 1&2	QuickSort Med 1&2	QuickSort First 50 Insertion	QuickSort First Insertion 100
50	63,600	39,100	63,200	70,100
1000	4,112,300	932,700	4,174,800	4,098,600
2000	8,666,100	1,089,600	6,776,600	11,872,900
5000	16,932,000	1,675,000	20,227,100	18,906,500
10000	49,186,400	2,984,200	67,435,400	67,488,700
20000	177,064,400	6,221,700	284,537,700	275,408,500
Average	**42,670,800**	**2,157,050**	**63,869,133**	**62,974,217**

For the random list: insertion sort would on average perform at$O(n^2)$, and Quicksort's average performance would be O(nlogn). This should result in better performance in comparison to the reverse order list, but still worse performance than Quicksort by itself, since insertion sort's average performance is the same as its worst case.

Random				
time (ns)				
File Size	QuickSort First 1&2	QuickSort Med 1&2	QuickSort First 50 Insertion	QuickSort First Insertion 100
50	44,700	40,200	45,500	52,900
1000	1,059,900	1,096,700	2,820,500	3,028,400
2000	1,611,100	1,275,700	4,239,300	5,759,800
5000	2,649,000	1,984,600	7,873,700	11,190,800
10000	5,140,800	3,511,900	14,790,400	13,649,200
20000	6,419,300	7,116,700	45,890,500	48,612,200
Average	**2,820,800**	**2,504,300**	**12,609,983**	**13,715,550**

Moving on to the ascending list, the hybrid sorting strategy significantly outperforms both Quicksort counterparts. This performance is due to insertion sort performing at its best-case scenario of O(n) since the list is already sorted in ascending order. In comparison to the first item pivot, which suffers from a worst-case of $O(n^2)$ due to the extreme pivot, the hybrid strategy mitigates some of Quicksort's poor performance. The same reasoning applies to the comparison to the median of three pivot and hybrid sorting comparison, albeit the better pivot choice results in a better performance of O(nlogn).

Ascending time (ns)				
File Size	QuickSort First 1&2	QuickSort Med 1&2	QuickSort First 50 Insertion	QuickSort First Insertion 100
50	114,700	37,800	7,100	6,800
1000	3,938,600	824,800	49,500	49,000
2000	7,451,300	1,090,800	89,600	88,100
5000	17,943,400	2,003,800	203,700	203,500
10000	28,144,600	3,973,600	476,400	399,600
20000	80,817,200	6,105,400	857,700	817,200
Average	**23,068,300**	**2,339,367**	**280,667**	**260,700**

When comparing between the partition stopping cases of size 50 and 100, we can see that in the best-case scenario of an already ordered list, the larger partition size performs better because insertion sort takes over sooner than in the smaller partition size. This is best highlighted when observing the execution on the ascending list.

Natural Merge Sort:

Natural merge sort's best-case scenario is when there is a sequential pattern to the order, because prior to merge sorting, it checks for any naturally occurring sequences to avoid needless comparisons. The reasoning behind this technique is that even though it takes time to check the list, it's more efficient to perform merge sort on sorted sub-files. For the ascending ordered lists, merge sort would take advantage of the fact that it's already in a specific order. Looking at the execution times versus the other four sorting algorithms, its observed that natural merge sort beats out both quicksorts, but loses to the hybrid sorts. This might be because natural merge needs to first check for patterns in its first pass, which takes up some time. However, both the insertion sort within the hybrid sort and natural merge sort should perform at O(n) under best case scenarios.

Ascending time (ns)					
File Size	QuickSort First 1&2	QuickSort Med 1&2	QuickSort First 50 Insertion	QuickSort First Insertion 100	Nat Merge
50	114,700	37,800	7,100	6,800	24,600
1000	3,938,600	824,800	49,500	49,000	380,900
2000	7,451,300	1,090,800	89,600	88,100	478,200
5000	17,943,400	2,003,800	203,700	203,500	1,485,000
10000	28,144,600	3,973,600	476,400	399,600	1,612,900
20000	80,817,200	6,105,400	857,700	817,200	3,057,400
Average	**23,068,300**	**2,339,367**	**280,667**	**260,700**	**1,173,167**

Looking both the random and reverse sorting times, natural merge sort runs comparably the same amongst these two. This is expected because natural merge sort has a worst and average case of O(nlogn). In comparison to quicksort for the randomly ordered list, it outperforms the hybrid sorting algorithms, but fails to be as fast as both the first item pivot and median of three pivot. This is somewhat expected as there isn't a pattern for natural merge to take advantage of.

Random time (ns)					
File Size	QuickSort First 1&2	QuickSort Med 1&2	QuickSort First 50 Insertion	QuickSort First Insertion 100	Nat Merge
50	44,700	40,200	45,500	52,900	182,800
1000	1,059,900	1,096,700	2,820,500	3,028,400	1,451,400
2000	1,611,100	1,275,700	4,239,300	5,759,800	2,919,600
5000	2,649,000	1,984,600	7,873,700	11,190,800	7,985,200
10000	5,140,800	3,511,900	14,790,400	13,649,200	11,854,000
20000	6,419,300	7,116,700	45,890,500	48,612,200	15,393,100
Average	**2,820,800**	**2,504,300**	**12,609,983**	**13,715,550**	**6,631,017**

Moving on to the reverse ordered list, we see that natural merge sort is faster than the first item pivot quicksort. This is because extreme pivots are the worst-case scenario for quicksort. However, it isn't as fast as the median of three quicksort, even though both algorithms have an efficiency of O(nlogn). This variance might be due to inefficient programming for the natural merge algorithm.

Reverse time (ns)					
File Size	QuickSort First 1&2	QuickSort Med 1&2	QuickSort First 50 Insertion	QuickSort First Insertion 100	Nat Merge
50	63,600	39,100	63,200	70,100	155,400
1000	4,112,300	932,700	4,174,800	4,098,600	1,588,000
2000	8,666,100	1,089,600	6,776,600	11,872,900	3,588,100
5000	16,932,000	1,675,000	20,227,100	18,906,500	7,765,200
10000	49,186,400	2,984,200	67,435,400	67,488,700	13,517,500
20000	177,064,400	6,221,700	284,537,700	275,408,500	16,962,100
Average	**42,670,800**	**2,157,050**	**63,869,133**	**62,974,217**	**7,262,717**

Natural Merge Sort vs Straight Merge Sort:

When comparing straight merge sort with its natural variant, the key difference is that under best case scenarios, natural merge sort performs better than straight at an efficiency of O(n) versus O(nlogn). It would be expected that if straight merge sort was ran on the ascending list, it'd perform worse than natural merge sort.

In regards to the randomly ordered list, on average, both variations of merge sort should run the same at O(nlogn). If there were patterns within the randomly ordered list, than the natural variant should edge out the straight variant.

When considering the worst-case scenario for both merge sorts, a reverse order list, their efficiencies are the same: O(nlogn). However, natural merge sort should run slower due to the fact that its first pass will be to check for a non-existing pattern.

Duplicates:

All algorithms were tested on lists that had increasing percentage of duplicates filling the rows lists. Looking at the results, it looks like natural merge sort is the most tolerant to duplicates, having maintained very consistent results and also performing faster than both quicksorts on average even without accounting for its performance at 100% duplicates. The two hybrid algorithms also performed well, although there was definitely greater variance. They both performed the fastest when the list was 100% duplicated, but that's primarily due to the fact that insertion sort performs at its best-case scenario in this situation. While natural merge sort should be taking advantage of the "pattern" of a fully duplicated file, its first pass needs to check for that pattern first, which slows it down in comparison to the hybrid algorithms.

Random Duplicate					
time (ns)					
Dup %	QuickSort First 1&2	QuickSort Med 1&2	QuickSort First 50 Insertion	QuickSort First Insertion 100	Nat Merge
20%	5,695,400	5,426,800	7,155,800	7,297,900	7,265,600
40%	9,173,100	9,910,300	7,424,000	6,735,500	7,283,500
60%	9,555,000	11,034,000	9,911,600	9,866,100	7,124,000
80%	15,430,700	14,754,500	7,402,200	8,462,300	6,544,800
100%	12,006,600	11,835,400	203,700	193,900	895,900
Average	**10,372,160**	**10,592,200**	**6,419,460**	**6,511,140**	**5,822,760**

Data Structures:

In the Quicksort algorithms, stacks were implemented in order to hold the start and end points of the partition branches - this allows for backtracking when we're done partitioning. This is similar to saving the recursive state when quicksort is implemented recursively. The stack data structure was implemented with linked lists. The reason the stack wasn't implemented with arrays is because the size of the stack is unknown. The number of partitions would be dependent on how the input file is sorted and because we have multiple randomly sorted files, the number of objects being pushed into the stack is unknown, especially with the larger files.

The natural merge algorithm was implemented with Queues via linked lists. A linked list implementation helps to prevent the space issues we could run into if we were to use arrays since traditional merge sort uses double the space. The queues were used to store the partitions, while decreasing the size of the main list without needing extra computation.

What I learned:

This lab gave me a better understanding of the differences between iterative and recursive solutions and how important foresight and planning is when trying to develop algorithms. I initially planned to program

both algorithms recursively, but I was having difficulty recursively programming natural merge sort. After struggling for a couple of days, I decided to switch to coding both algorithms iteratively, and had to crunch in order to make up for lost time.

Also, sticking to what you're comfortable with instead of picking what seemed like the more logical approach can be more effective in situations where you're crunched on time. I thought that both algorithms were naturally recursive so I jumped straight into trying to program them that way, however I couldn't figure out how to implement the partitioning aspect of natural merge sort recursively. I ended up defaulting back to the iterative implementation because I was running out of time and I was also more comfortable with iteration than recursion.

One thing I learned (or rather, experienced) was that execution times differed by computer. When I was originally gathering data, I was experiencing large variations in times. This was due to the fact that within the past week, I was executing the code on three different machines.

What I might do differently:

I probably should've planned out which implementation was the better one to program for the algorithms. I knew both used a divide and conquer method so recursion seemed like the sensible implementation, but I ran into too many difficulties when trying to recursively implement natural merge sort since I couldn't understand how to recursively partition. In the end, I implemented the iterative versions of both algorithms.