Eyu Chen

Nov 8

CS124

# Project4 Report

In project4, we will use the dataset from the previous work and go further with sorting algorithms. The dataset is the list of the fall uvm courses in 2021, and there are 5422 records in total. In this project, we will try to sort all the records by the maximum enrollment number. There are 6 algorithms that would be used for sorting – bubble sort, selection sort, insertion sort, merge sort, heap sort, quick sort and in addition, a two-sort algorithm which is to sort the objects by 2 algorithms. In our case, selection sort is the first method to sort and then sort with insertion sort.

## Data

The data file used is about all the fall courses in University of Vermont in 2021, the source is from the Office of Register https://serval.uvm.edu/~rgweb/batch/curr_enroll_fall.html, and was modified a little bit. There are 9 attributes in the .csv file and then become the fields for the class, some naming follows abbreviations used in CSV file:

- rowId: the row ID for ordering
- subject: the subject that the course belongs to
- title: the title of the course
- CRN: the registration number for this course
- college: the college code that the class belongs to
- maxEnrollment: the maximum enrollment of the course
- currEnrollment: the current enrollment of this course by now
- instructor: the professor's name of the course, in the format of FirstName, LastName
- email: the uvm email address of the professor

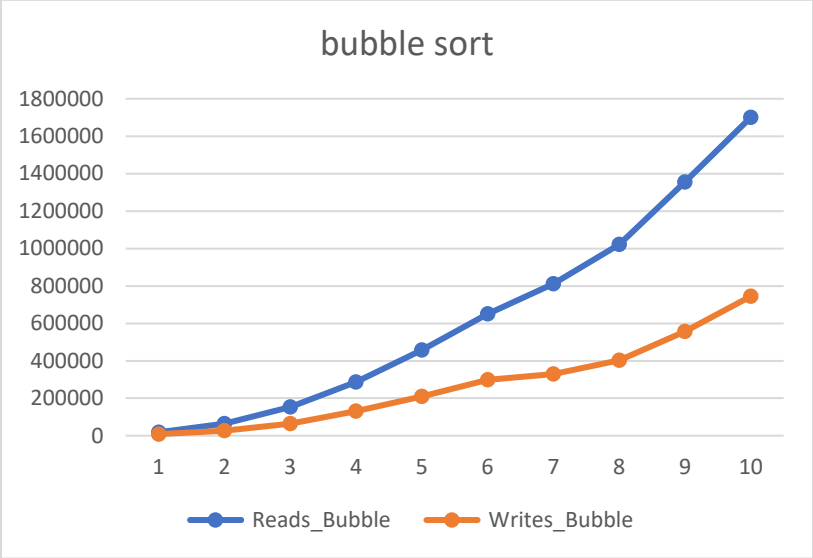
The original dataset is ordered by title over alphabet, and a column named rowId was added in the first place from 1 to 5422, and now the dataset is ordered by this. But this time we want it to be sorted by the field of "maxEnrollment".

## Citation

The method to set timer are from https://en.cppreference.com/w/cpp/chrono/duration

## Graph Analysis & Comparison

- Bubble sort:

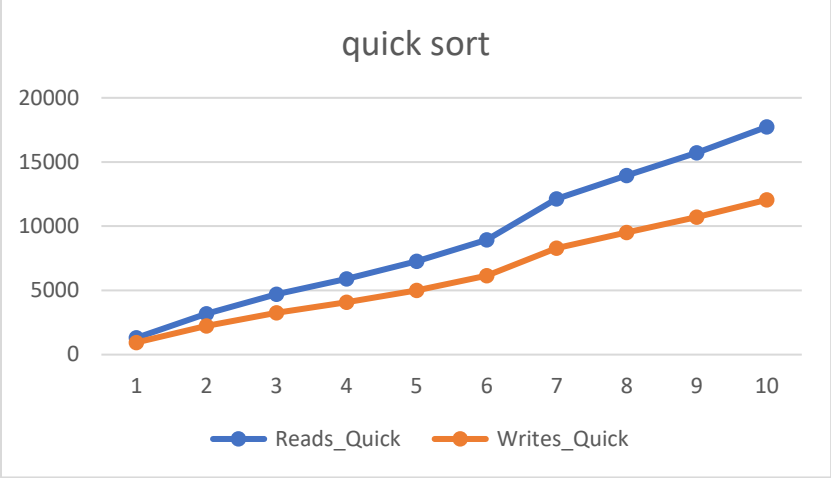



Bubble sort algorithm will firstly swap the largest item with the smaller one in each step and move it to the correct place. It will read through the unsorted part to find the biggest one and swap. According to the graph, it is clearly that the numbers of reads and writes grow linearly with the increasing size value.

- Selection sort:

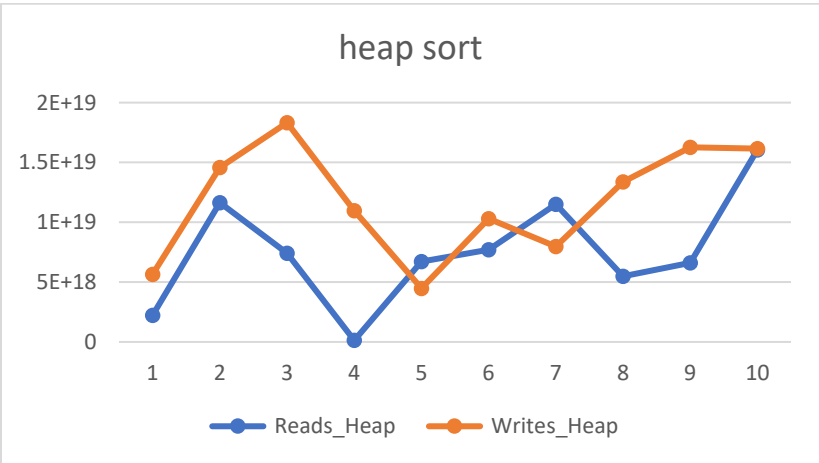
Selection sort is aimed at moving the smallest item to the first place of the unsorted part. The line of reads number is similar to the that of bubble sort because both of them will read through the unsorted sequence in each iteration. However, the number of writes is much smaller than that in bubble sort, the reason is that selection sort only swap items when the smallest item is found, while bubble sort will swap once the two items compared need to be sorted.

- Quick sort:




Quick sort requires a pivot first in each iteration, read through all the rest items and move the smaller values to the left and the right space is for the bigger values. It is clear that the numbers of reads and writes are much smaller than the previous 2 algorithms (especially bubble sort), and meanwhile the lines imply a linear relationship between reads/writes and data size.

- Heap sort:

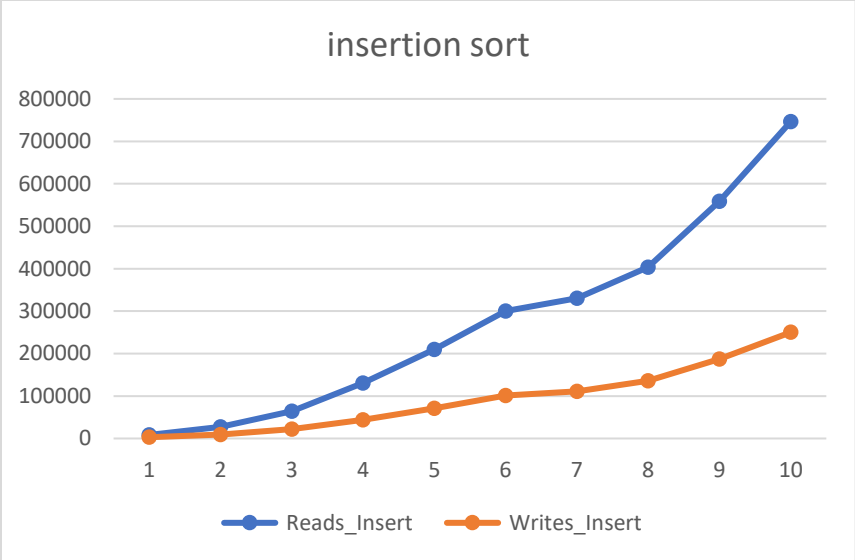


Heap sort uses a heap tree to store the items and the heap properties to sort. What the algorithm does is to heapify the items and build a maximum or minimum heap tree (in our project, we use maximum heap tree), and then remove the root in each step and rebalance the tree to get new root and remove as before. Since we have three complex operations for this algorithm – heapify, removing and rebalancing, the number of reads and writes could be extremely large. In the programing, a data type of unsigned double long integer would be used for number of reads and writes. Also, unlike the previous algorithms, there is no obvious rule for the relationship between reads/writes and data size.

- 2-sort:

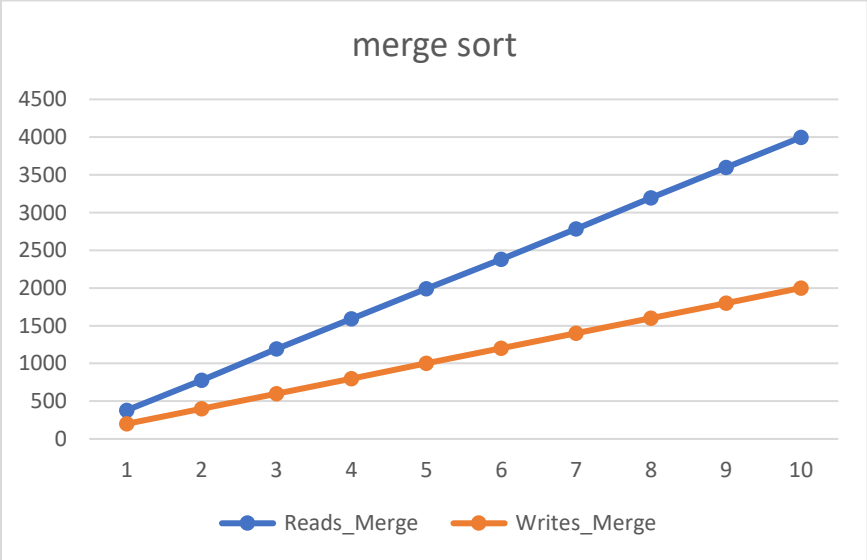

Two-sort algorithm sorts the dataset twice with different sorting algorithm and on distinct fields. We first use selection sort to sort items by the default field “maxEnrollment”, and then use stable insertion sort to sort items on “CRN” field. Comparing with others, the number of reads/writes is absolutely bigger because it actually sorts twice. In addition, the number of reads increases so quickly while the number of writes grows slowly.

- Insertion sort:




Insertion sort will pick one item from the unsorted part and compare this item with the ones in sorted part, find the correct place and insert. From the graph, the line of the number of reads increases quickly while the number of writes is at a slow pace.

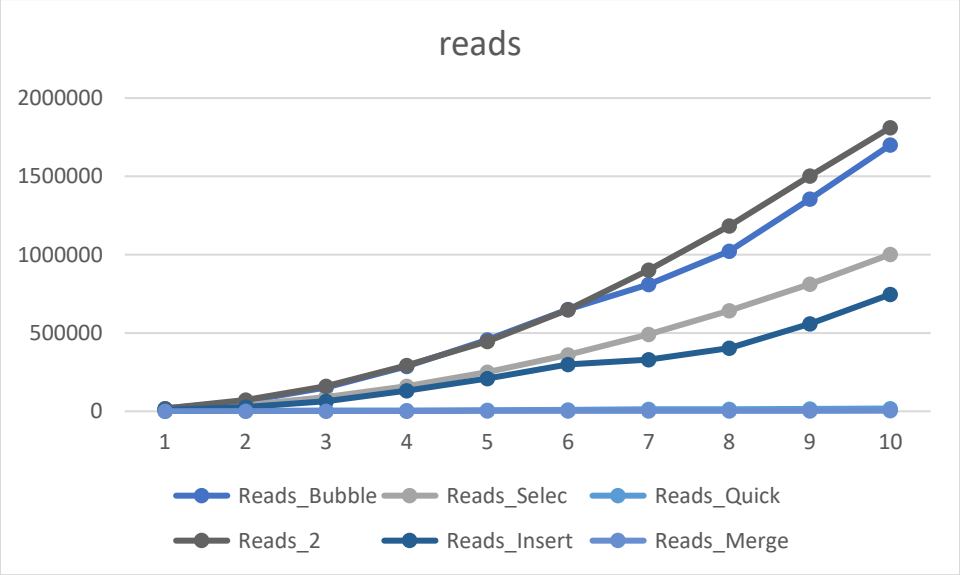
- Merge sort:



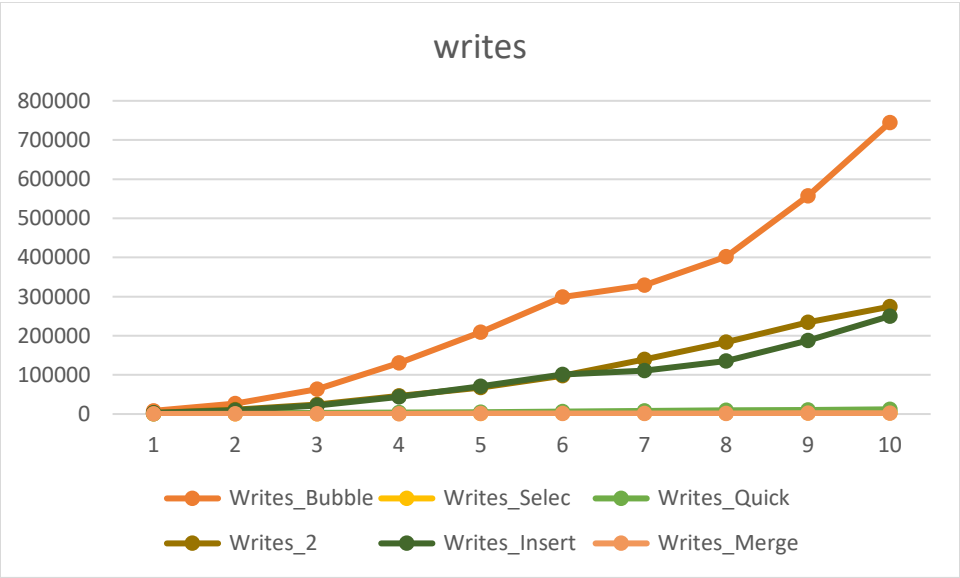

The key to merge sort is to partition the dataset into pieces of one item and then compare the partitions. According to the graph, both the number of reads and writes are surprisingly small and in a nearly perfect linear format. From the aspect of coding, the recursive part of merge algorithm is for partitioning and that could be one reason of the low number of reads/writes.

## Conclusion

The following two graphs compare the reads and writes of 6 algorithms in this project. We do not include the heap sort because the numbers are too huge.




With combining the lines together, we can easily see that the quick and merge sort gains a much smaller numbers and slower growing pace; selection and insertion sort share a similar trend while insertion sort is smaller than selection sort; two-sort is the biggest and fastest because it sorts twice, however, the bubble sort shows a similar pace with two-sort even though it just sort once.



In general, the number of writes of each algorithm is smaller than the number of reads. Obviously, bubble sort has the biggest and fastest line of writes because it swaps so many times even in just one iteration; then, the lines of two-sort and insertion sort are nearly the same while the values of other 3 algorithms are really small.

According to the graphs, we can know that bubble sort is not good in reducing the times of reads and writes in this case, but quick sort and merge sort have an excellent performance with lowing reads and writes; also, selection sort works better in saving writing operations even though it has a higher number of reads. In conclusion, if we believe that low number of operations is related to the efficiency, then the merge sort and quick sort would be more efficient in our case.

## Questions

- If you need to sort a contacts list on a mobile app, which sorting algorithm(s) would you use and why?
  Stable algorithm such as merge sort, insertion sort and bubble sort could be available for a contacts list. In a contacts list, it is possible to have many duplicate values and we need a stable algorithm to ensure that the original order in these values would not be in a mess. Also, since the amount of data of a contacts list on a mobile phone usually is small, which means that we do not need to worry about performing millions of operations.

- What about if you need to sort a database of 20 million client files that are stored in a datacenter in the cloud?
  Merge sort could be a good choice. To sort a sorted data sequence, stable algorithms are necessary. Also, with the huge amount of data, we do not want the algorithms that operate billions of times because the number of operations has some relationship to the time and space we used for this algorithm.
  Also, heap sort might be useful in this case because it seems like the number of reads/writes have no relationship with the data size, therefore it could be used with huge dataset when other algorithms reach to a high number of reads/writes.

## Complexity of Algorithms

- bubble sort
  time complexity: O(n^2)
  auxiliary complexity: O(1)
  space complexity: O(n)

- selection sort
  time complexity: O(n^2)
  auxiliary complexity: O(1)
  space complexity: O(1)

- insertion sort
  time complexity: O(n^2)
  auxiliary complexity: O(1)
  space complexity: O(n)

- quick sort
  time complexity: O(n log n)
  auxiliary complexity: O(log n)
  space complexity: O(log n)

- merge sort
  time complexity: O(n log n)
  auxiliary complexity: O(n)
  space complexity: O(n)

- heap sort
  time complexity: O(n log n)
  auxiliary complexity: O(1)
  space complexity: O(n)

## Timer

In this project, we use methods and functions from chrono to set timer and count the duration of each sorting algorithm. The code should return the duration in microseconds of each algorithm with different data sizes and show message in the command line.