**605.202 DATA STRUCUTRES**

**GUAN YUE WANG**

**JHU ID: gwang39**

**LAB 4 ANALYSIS**

**AUGUST 16, 2017**

**General Program Design**

The program conducts seven sorting methods on data sets with different size and natures in order to compare their relative performances. Sorting methods include insertion sort, heap sort, and five shell sorts with different increments based on various sequences such as Knuth's sequence, Marcin Ciura's sequence, and Hibbard sequence.

The overall program is controlled by a while loop to read input text file number by number. After running this while loop for the first round to figure out the file size, the program runs it again to take each number and add them to array so they can be sorted later. When seven data sets are prepared and ready for seven sorting methods, the program runs those sorts one by one and records sort time. At the end of each round, the sorted results and run time are output to corresponding output file based on output path provided.

The program consists of the main class and three other classes for each of the insertion sort, heap sort, and shell sort. All sorts are constructed using iterative approach plus array implementation if needed. The main entry point can be ran with or without command line arguments ([input file] [output file]). If the user choose not to do so, the program console would prompt user to input I/O file locations.

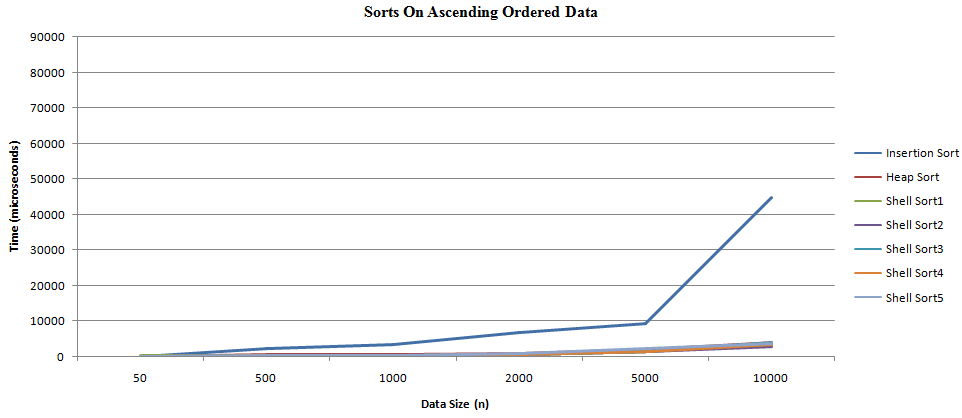
**Data Structure Implementation And Appropriateness**

Array is used as the major data structure to store data and support sorting in the program. It is considered a good fit with our program based on several reasons. To begin with, the major part of our program is to use insertion sort or selection sort; therefore, array implementation gives us random access to the data and makes it easier for us to construct the sort and build iterative logics. To be specific, insertion sort requires iteration through the data and conduct comparison along the process whereas selection sort iteratively move data from unsorted region to sorted region so array implementation provides a better support for both. Furthermore, as we can easily figure out the size of the file, the memory allocation to the array can be accurately defined without any waste of resource.

**Sorting Performances**

**Performance Overview**

Overall, we can see heap sort and shell sorts are generally better in terms of efficiency. The only case insertion sort can be better than the rest is when data size is small which is proved our test case n = 50. This actually makes sense as insertion has a best case n but average and worse case n^2 whereas heap sort has a consistent complexity nlogn and shell sort has complexity ranging from nlogn (best case) to n(logn)^2 (average and worse case). In terms of space complexity, all sorts has memory requirement 1 as all of them involve in-place comparison so there are all equal. Tables and Graphs below shows overview of the performance of all sorts with respect to different data size and data nature (ascending, random, reversed).





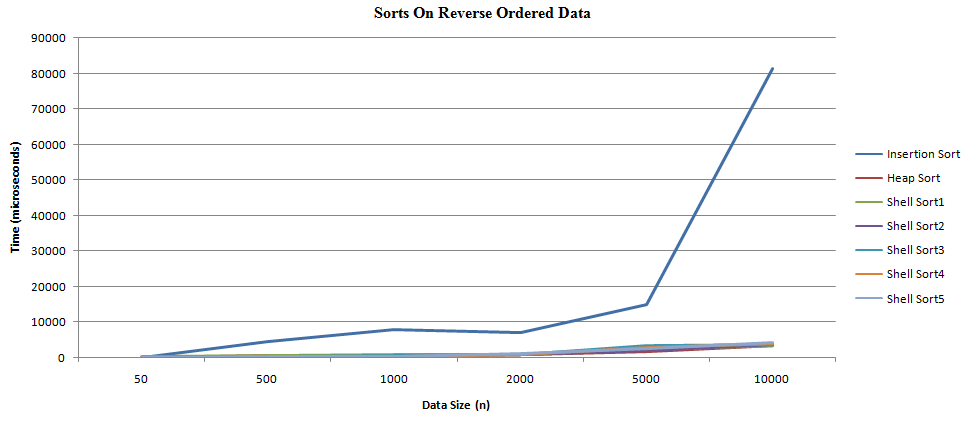
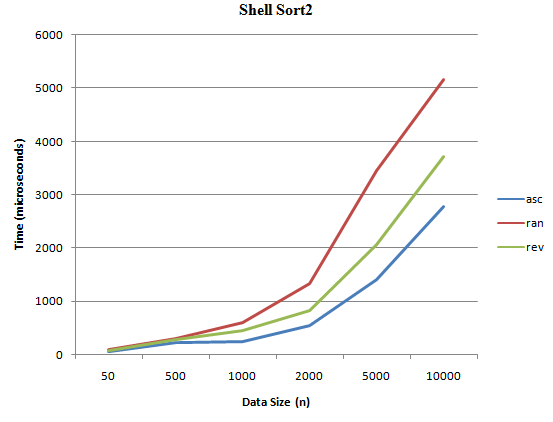
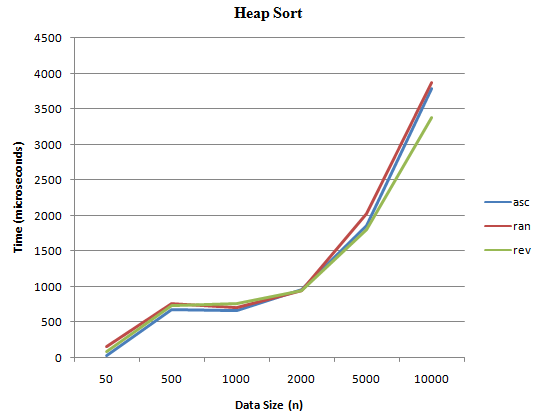


Chart - Sorting Time On Different Data (time measured in microseconds)

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **n** | **data** | **Insertion Sort** | **Heap Sort** | **Shell Sort1** | **Shell Sort2** | **Shell Sort3** | **Shell Sort4** | **Shell Sort5** |
| 50 | asc | 27 | 22 | 132 | 61 | 72 | 87 | 82 |
| 50 | ran | 129 | 154 | 155 | 90 | 87 | 103 | 118 |
| 50 | rev | 50 | 82 | 121 | 65 | 78 | 73 | 114 |
| 500 | asc | 2141 | 675 | 283 | 236 | 202 | 253 | 251 |
| 500 | ran | 3735 | 758 | 371 | 291 | 516 | 338 | 323 |
| 500 | rev | 4475 | 735 | 349 | 277 | 273 | 276 | 286 |
| 1000 | asc | 3454 | 653 | 326 | 254 | 268 | 336 | 355 |
| 1000 | ran | 4155 | 704 | 653 | 591 | 1111 | 579 | 630 |
| 1000 | rev | 7811 | 760 | 626 | 444 | 529 | 420 | 501 |
| 2000 | asc | 6731 | 951 | 572 | 551 | 540 | 618 | 868 |
| 2000 | ran | 6852 | 950 | 1332 | 1327 | 2974 | 1350 | 1477 |
| 2000 | rev | 6983 | 949 | 804 | 818 | 955 | 866 | 1094 |
| 5000 | asc | 9284 | 1864 | 1411 | 1421 | 1410 | 1411 | 2098 |
| 5000 | ran | 19909 | 2028 | 3572 | 3460 | 4303 | 3570 | 3574 |
| 5000 | rev | 14868 | 1805 | 2078 | 2061 | 3308 | 2735 | 2654 |
| 10000 | asc | 44628 | 3791 | 3387 | 2787 | 3828 | 3314 | 3715 |
| 10000 | ran | 80169 | 3877 | 4804 | 5166 | 7276 | 4550 | 4774 |
| 10000 | rev | 81284 | 3383 | 3304 | 3715 | 3726 | 3827 | 4214 |

**Heap Sort vs. Shell Sort**

As we can see insertion sort is much worse in terms of performance when n gets larger despite the nature of the data, we will focus our performance comparison and discussion around heap sort and shell sort.

From graph above, we can see heap sort provides a more consistent performance across all different kinds of data whereas shell sort shows a difference in efficiency in handling different type of data. To be specific, while heap sort has a almost the same performance on ascending, random, and descending data, Shell sort works the best on ascending data, then reverse ordered data, and performs the worst on random data. This actually aligns with the big O we learnt in class where heap sort always give nlogn performance whereas shell sort's performance can range from nlogn (best case) to n(logn)^2 (average and worst case). Based on the time we recorded, we can see heap sort performs better when the data is random; otherwise, shell sort provides a better run time on ascending and reversed data.

**Sequence Used In Shell Sort**

Based on project instruction and my search results, five different increments are used for shell sort as below

Shell Sort1- Knuth's sequence

Shell Sort2 - Knuth's sequence to the nearest prime number

Shell Sort3 - Knuth's sequence to the nearest composite

Shell Sort4 - Marcin Ciura's sequence

Shell Sort5 - Hibbard's sequence

**Alternative Approach**

The determinant calculation in this program leveraged recursion and Laplace expansion. This is an extremely inefficient approach for large matrix size considering it recursively calls minor itself and grows matrix operations. One of the alternative iteration solution can be Decomposition method. This includes lower upper decomposition and Cholesky decomposition which can decrease the complexity from O(n!) to O(n^3). For example, lower upper decomposition involves express matrix in terms of lower triangular matrix L and upper triangular matrix U and permutation matrix P. As the determinant of these 3 matrix can be easily calculated, determinant of the original matrix can also be quickly obtained by multiplying these 3 determinants together. Both lower and upper triangular matrix can be found using nested for loop to iterate through columns at per row level.

Regarding the data structure, linked implementation can be a better choice compared to array implementation when we have a large matrix size or we have sparse matrix. In the event of large matrix size, the combination of array and recursion can cause significant inefficiency as we need to allocate time and space to calculate minor matrix whereas linked implementation would provide a better big O by breaking and re-linking nodes together. On the other hand, if we have a sparse matrix, a lot of space in the array would be 0/null and thus get wasted. Instead, if we use linked implementation, the space can be better used by only storing non-0 values in the matrix. The linked structure can be implemented with three kinds of nodes (see Appendix 2 for illustration):

- Header node would tell total number of rows, columns and non-zero values.

- Row node would store current row number and point to two places: next column node in this row that stores non-zero value as well as next non-empty row.

- Column node contains column number, the actual data value, and points to next column node.

**Time And Space Efficiency**

As recursion is used in Laplace expansion to accommodate determinant calculation, each method call must evaluate n with determinant of size n-1. Consequently, total number of recursive call is n\*(n-1)\*(n-2)\*...\*2\*1 which leads to O(n!). Thus, the algorithm can become extremely slow when we have a large size matrix. Please see Appendix 3 for an demonstration of current time complexity using real run time records with different n size matrix. Therefore, as mentioned on the above, LU decomposition or Cholesky decomposition can be a better way to calculate determinant in order to decrease the time complexity to O(n^3) from O(n!).

From a space efficiency perspective, as the recursion requires keeps allocating space to store minor matrix for Laplace expansion, the space used is heavily wasted as we allocate space for data that we don't need in current operation. Thus, an alternate approach should be considered in order to achieve better space efficiency. Also, as mentioned above, a linked implementation would be better choice in the case of sparse matrix.

**Learning And Takeaway**

There are many takeaways from this lab. One of the most important ones is how the selection of data structure and algorithm can lead to huge impact on time and space complexity. It is surprised to see how Laplace expansion's O(n!) compared to Decompositions' O(3) as well as how space is affected by the matrix size and density. In summary, a combination of data structure and algorithm should be considered at the planning stage to evaluate the most efficient and suitable solution to program things out.

As mentioned above, I look forward to the next lab where I can use linked structure and decomposition method for determinant calculation. Although it's might take more time to figure out the structure and logic, it would be really interesting to see the complexity difference in the actual program. On the other hand, things I would do the same is the same is to keep the flexibility in the program I/O process. In this lab, both command line arguments and console input is available as options for user file to specify input and output locations. I believe all these user-friendly features would be valuable for user experience enhancement and adoption of the program.

**Enhancement**

There are several enhancements in the program to improve the process flow in the aspect of general design, user I/O, validations, and user display

To begin with, the general design of the program implements OOP concept by having a determinant calculator available to take the matrix, calculate determinant, and generate summary information. Compared to putting all methods in main class, it brings encapsulation to make sure the structure and creates the flexibility for people to modify calculation or display format later.

As a part of the design, there's also enhancement at user I/O level, it grants user the option to choose whether they want to provide input and output locations as command line arguments or not. To be more specific, they can state input and output file locations as command line arguments so the program gets the I/O information directly from there. Meanwhile, they can also run the program without providing any command line arguments and the program would prompt user for I/O locations in the console and read from there.

In addition to general design, the program also has an extra validation method to make sure the input file follows the correct matrix format (size N followed by N\*N matrix) so no processing power is wasted prior to we confirm the completeness of the data in the file. Besides, output is formatted in a way that matrix has border around it so people can easily tell whether they are looking at the matrix or output summary information.