Ngoc Tran 113318919

Extra Credit: Yes

PROJECT 3: SORTING ALGORITHMS REPORT

Introduction

Sorting algorithms rearrange the array or list elements in ascending order by applying comparison operator on the elements. The comparison method determines the order of element in the data. In sorting categories, the two main types include internal sorting and external sorting. Internal sorting is conducted when the number of objects is small enough to fit into the main memory while external sorting is applied when the number of objects should put in the external storage during the sort. In this study, four main sorting algorithms are conducted: Bubble Sort, Quick Sort, Shell Sort for internal sorting and Adaptive Sort. Note that these sorting methods are built on comparison-based algorithm.

Bubble Sort and Quick Sort algorithms perform sorting by exchanges when the two elements are out of the order in each step. In Bubble Sort, a loop is run through all elements of an array from the first position to check whether two adjacent elements are out of order. The out-of-order elements are then switched. In the first phase, all adjacent elements are compared until the looping is finished. The largest element is swapped with the adjacent elements unil it "bubbles up" to the top position. The second phase restarts from the previous phase and continues comparing and exchanging the last element. This results in the correct location of the second-largest element. As for Quick Sort, pivot value choice and partition step are the three crucial steps. The pivot value choice step is either the first key value or the median of set element, which contributes as the main comparing agent. The second step, called partition, exchanges of key values in the array. If the key arrays are less than or equal to pivot values, they are set to the left side of the array. Otherwise, the keys are set to its right in array if they are greater than the pivot value. This partition ensures the correct position of pivot value in the array. The recursive step conducts quicksort on elements which are either left or right to the pivot value. In general, bubble sort is regarded as the simplest and most basic sorting algorithm; however, it fails to process properly with large number of input (or memory) since complexity depends on $O(n^2)$. Therefore, the importance of sorting time in the large problem should be analyzed. On the other hand, Quick Sort performs much faster than other algorithms regardless of array size with $O(n)$. However, the selection of pivot value would cause drastic slow-down if values are always the smallest or the largest value.

Shell Sort algorithm is developed from insertion sort. This algorithm performs sorting based on diminishing increments, by breaking the array (or list) to smaller sub-array (or sub-list). Then an increment, or "gap", is employed to create the array/ list which elements are apart by gap increment which is also gap-sorted. The algorithm of shell sorting allows the exchange of far item. Then the increment keeps reducing to 1, which means more sub-array keeps breaking. The array of gap-sorted is sorted once all sub-array with gap numbers of elements are sorted. Since no undo of previous work is involved, Shell Sorted is effective since sorting with one increment will not reverse or rescind the previous sorting which implemented in a different increment. The complexity of this method can overcome the worst-case running time (i.e. better than $O(n^2)$).

Another improvement of comparison-based sorting is Adaptive Sort when the given array/ list is relative sorted (i.e. "some-what sort"). The relatively sort term is defined based on out-of-order and the estimation of how array is sorted closely by p-sortedness. The 2 elements Ki and Kj at position i and j in the array are out-of-order when Ki > Kj and $1 \leq i \leq j \leq n$. The array is considered p-sorted when for some value of positive p, all of pairs of keys in position i and j are out-of-order, meaning that $j-i \leq p$. The algorithm involves merge sort with different splitting strategies. In each recursive step, the array is split into 2 lists of odd and even position. After splitting, a sub-list is tested whether it is sorted. A merge array is employed to merge the sub-lists when the sub-lists are sorted. Otherwise, a recursive step of adaptive sort is called for unsorted list. With the complexity of O(log p), this algorithm operates well in the worst-case and improves the existing order in the input array.

Note that sorting helps standardizing the data and turns data into readable products. The complexity of the sorting algorithm is a crucial factor when an analysis is conducted on how fast the array can sort from a rando set. The efficiency of algorithms is optimized through the efficient sorting. Evaluating the speed of algorithm will eliminate the processing time. In this report, a comparative analysis of processing time with different sizes is examined to evaluate the efficiency of sorting algorithm. Therefore, we can implement to different types of data with modification.

Methods

A program is written in C++ to implement the four sorting algorithms. The packages iostream, cstdlib, math.h and ctime are implemented. A sortElements class is created with 2 private fields: number of elements (N) and the elements themselves (pointers to elements) which is an array of integer.

A pseudoradom number generation is also employed. The numbers are randomly generated with a give seed (srand()). The seed ensures the same set of pseudorandom number. The random number ranges within the input upper and lower values. Therefore, each element is calculated by rand() % (upper number-lower number +1) + lower number.

The public modifiers of sortElements class include constructors (default and non-default), destructor, random generator, accessor to get either elements of array or the number of elements and display the entire array. The method for 4 sorting algorithms (Bubble Sort, Quick Sort, Shell Sort, and Adaptive Sorting) are also written in the public fields.

Other necessary methods for each sorting algorithm are also set in public modifiers. A swap method helps changing the position of element x and element y. It is employed in Bubble Sort and Partition. A partition method exchanges key values by comparing assigned pivot value. A size method returns the size of array dynamically. And a merge array merges the sorted sub-list or sub-array in the Adaptive Sort method.

The execution time for each algorithm is recorded for comparative analysis. Package "ctime" is implemented for measuring time. Using clock_t, a begin clock is activated before the sorting algorithm is called and the end clock is also marked after the sorting algorithm. The difference between 2 clock indicates the number of clicks which ends up being divided by

CLOCKS_PER_SECOND to obtain the seconds. The execution time is recorded in different numbers of elements.

Results

Table 1 and Figure 1 below indicates the time taken in seconds for Bubble Sort, Quick Sort, Shell Sort and Adaptive Sort in different numbers of elements in the array.

Table 1: Time taken (in seconds) for Bubble Sort, Quick Sort, Shell Sort and Adaptive Sort with 4 different numbers of elements (N= 5000, 10000, 50000,100000)

N / Sorting	5000	10000	50000	100000
Bubble Sort	0.162	0.642	16.25	65.003
Quick Sort	0.001	0.003	0.014	0.034
Shell Sort	0.001	0.002	0.009	0.017
Adaptive Sort	0.002	0.005	0.02	0.040

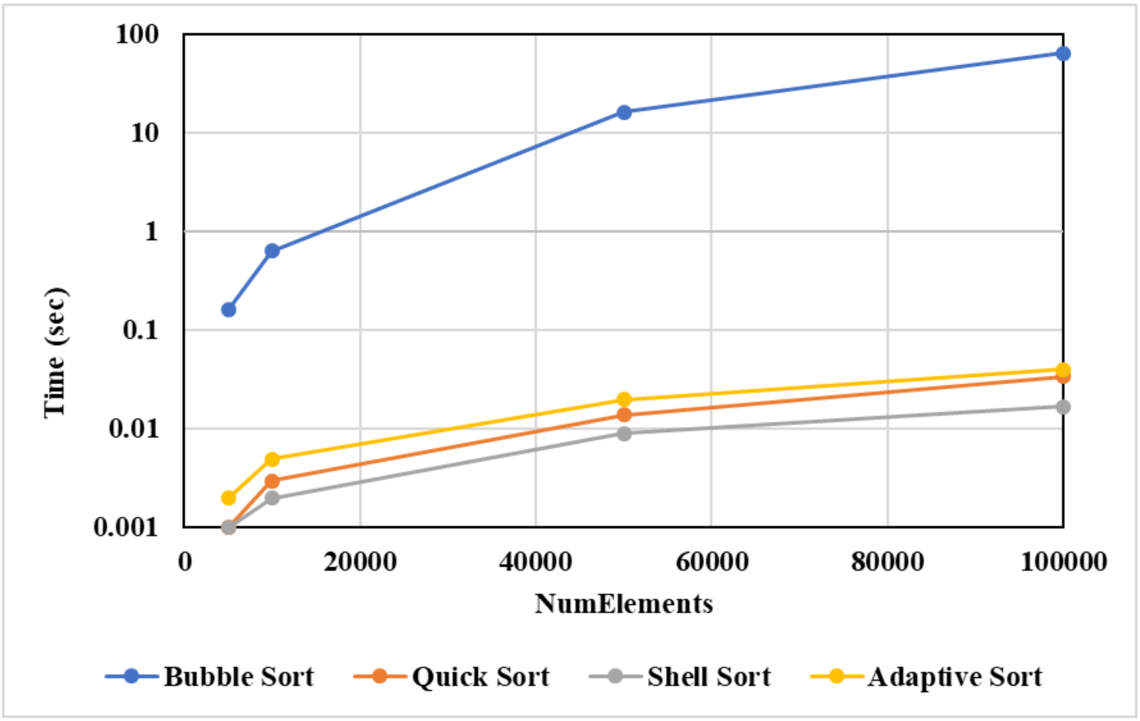

Figure 1. Plots of Time Taken for 4 sortings with different numbers of elements

Discussion

In general, the increasing numbers of elements result in more time execution to process sort algorithms. These 4 algorithm are all slow for the larger inputs. Thus, the complexity of algorithms greatly depends on the memory storage of the data.

From Table 1 and Figure 1, it is obvious that Bubble Sort takes most time to process its algorithm than the remaining sortings. As the number of elements reach 100000, the time is extremely slow down to 65.003 seconds in Bubble Sort. This happens because the complexity of Bubble Sort algorithm is $O(n^2)$ for the worst-case and average-case scenarios with n-1 phase. Bubble Sort algorithm is highly regarded for its simplicty, assuring both being stable and inplace. However, the effiency of Bubble Sort decreases significantly as the array is sized up.

Time taken for Adaptive Sorting is relatively slower than Quick Sort's and Shell Sort's yet faster than Bubble Sort's. The comparison performance of adaptive sort is improved by identifying out-of-order and evaluating how nearly array can be sorted. The complexity of Adaptive Sorting is O(nlogn) for worst-case scenario and O(nlogp) for average-case scenarios (if array is p-sorted). Therefore, its complexity indicates a lower degree the Bubble Sort. However, the splitting of elements with even and odd position and p-sortedness causes the instable in sort processing as the complexities in both worst and average scenarios show discepancy.

Quick Sort takes less time to process its algorithm than Bubble Sort and Adaptive Sort because it has 2 sub-array of half size at most (after comparing with pivot table). Eventhough it might take $O(n^2)$ algorithm, Quick Sort did not reach to its worst-case input. In average case, Quick Sort perform faster with complexity of O(nlogn). The advanatge of this algorithm is that it processes sorting quicker and faster than the other 2 methods. Quick Sort achieves inplace but its instability hinders the effiency of quick sort. Since Quick Sort and Adaptive Sort are relatively similar throughout different sizes, it should consider other factors, modifications and case scenario to select Quick Sort over Adaptive Sort or vice versa.

Shell Sort performs much faster than the other 3 sortins as the number of elements keep increasing. It tooks 0.001 sec with 5000 elements and 0.017 sec with 100000 elements. As mentioned in the introduction, this sorting achieves better efficiency due to the fact that no undo of previous work in different increments is involved when one new increment is applied.. Therefore, it saved much more time for the process. Following division by three approach, Shell Sort can reach to $O(n^{1.5})$ for average-case scenario, which overcome $O(n^2)$ complexity for the worst case. Eventhough Shell Sort takes shorted time and is inplace, it fails to achieve the stability when average case complexity has lower degree than the worst case complexity.

The selection of sorting algorithms depends on various applications and data types. Besides space complexities, the time complexities for different case scenarios also account for the sorting algorithm decision. In this report, Shell Sort perform much faster yet it has an issue of stability while the Bubble Sort reveals a significantly low performance for the large input but it indicates both stability and inplace during the process. Therefore, there is no definite answer to determine the "best" algorithm.

References

Fryar, Terry D. (1988). "Analysis of Sorting Algorithms". Honors Theses. Paper 230.

Radhakrishnan, S., Wise, L., Sekharan, C. (2013). Data Structures Featuring C++ a Programmer's Perspective.