Execution and Comparison of Various Sorting Algorithms

Gurpreet Kaur, Purnasree Saha, Yamini Pathuri, Koundinya Raghava Nerella
Department of Computer Science and Information Technology
Montclair State University
Montclair, NJ, USA

Abstract: *Different types of sorting algorithms are used to rearrange values in a list or in an array. Based on the type of sorting algorithm that is used, values are going to be arranged differently either in ascending or descending order since the algorithms execute differently. Comparison operators are used to compare and rearrange elements in the sequence. The sequence or list could either be alphabetical or numerical. Duplicate values could also be found in a given array or list. Time complexity and space complexity are what determines the efficiencies of the sorting algorithms.*

Keywords- Algorithm, Mergesort, Quicksort, Bubble Sort, Shell Sort, Comparison, Quick Sort, Time Complexity, Divide and Conquer, Polynomial Time

1. Introduction

When there is a sequence or a list of elements that are given, it is going to be unsorted, so we need to determine which sorting algorithm technique will be used to put the elements in the correct order. The list or array will continue to execute over and over again until all the elements are fully sorted. Since data is increasing each day, sorting methods need to be efficient and quick. There are ways in which some of the sorting algorithms work the same way as a binary tree search algorithm does. Distances from shortest to longest are also determined by different sorting algorithms. Searching, selection, duplicates, and distribution are the main methods in sorting. When a list is given, the value will be searched, but if the elements are already sorted in the list, the elements will be found rapidly. While searching for elements, the elements will be selected based on the relationship of the sequence if it's in increasing or decreasing

order. Comparison operator is used to swap elements in the list.

2. Related Work

The article called Sorting Algorithms which is written by Neelam Yadav and Sangeeta Kumari explains the different format of how sorting algorithms work and the pros and cons of each algorithm. There are also ways in which the sorting algorithms are classified differently. The working procedure for bubble sort and shell sort work differently. Bubble sort is considered to be the simplest comparison based sort. Shell sort is efficient and fastest in sorting smaller data sets. [2]

The article called Comparison of parallel sorting algorithms written by Darko Bozidar and Tomaz Dobravec introduces the processes of how various sequential and parallel sorting algorithms work. Merge sort refers to the divide and conquer since it splits the array or list into multiple subsequences. Quick sort is kind of similar to merge sort since it's based on the divide and conquer method. [3]

The article called Threshold Analysis and Comparison of Sequential and Parallel Divide and Conquer Sorting Algorithms written by Tinku Singh and Durgesh Kumar Srivastava talks about how the steps and techniques of sorting algorithms need to be fast. It also introduces how parallelism is applied to the sorting algorithms. [1]

3. Algorithms and Execution

Sorting algorithms are used to put elements in a list in a certain order. We will be exploring Mergesort, Quicksort, Bubble sort and Shell Sort. The example we will be using is sorting a list of cities in ascending order. We will use these algorithms to sort the list of cities and record the time it took to run it. They each have different approaches to sorting but they all reach the same resulting list.

3.1 Mergesort

Merge Sort is an algorithm technique that involves the divide and conquer technique since the input array is divided into 2 equal parts. If more than one element is in the array, then the array is going to be divided into two equal parts and then the merge sort algorithm will be called recursively. When the unsorted list is divided, it will be taken in as two smaller arrays. The arrays will be solved recursively and then be merged into a sorted list.

```
36  #mergesort pseudocode
37  MergeSort(A, p, r):
38      if p > r
39          return
40      q = (p+r)/2
41      mergeSort(A, p, q)
42      mergeSort(A, q+1, r)
43      merge(A, p, q, r)
```

Figure 1. Merge Sort Pseudocode

3.2 Quicksort

Quicksort is one of the most efficient sorting algorithms and it uses the divide and conquer approach. It can be faster than the merge sort and two to three times faster than the remaining algorithms. It acts as an internal sorting method, where in the internal memory the data will be sorted.
It selects the pivot element from the array, it can be any element in the array and “partition” the array into two sub-arrays. The average case time complexity for Quicksort is O(N log(N)).

```
quickSort(array, leftmostIndex, rightmostIndex)
  if (leftmostIndex < rightmostIndex)
    pivotIndex <- partition(array,leftmostIndex, rightmostIndex)
    quickSort(array, leftmostIndex, pivotIndex - 1)
    quickSort(array, pivotIndex, rightmostIndex)

partition(array, leftmostIndex, rightmostIndex)
  set rightmostIndex a pivotIndex
  storeIndex <- leftmostIndex - 1
  for i <- leftmostIndex + 1 to rightmostIndex
  if element[i] < pivotElement
    swap element[i] and element[storeIndex]
    storeIndex++
  swap pivotElement and element[storeIndex+1]
return storeIndex + 1
```

Figure 2. Quicksort Pseudocode

3.3 Bubble Sort

Bubble sort or also known as sinking sort is a simple sorting algorithm which compares adjacent elements and swaps if they are not in the right order. The pass through the list is repeated until the list is sorted. The first pass sorts the heaviest element in the list, the second pass sorts the second heaviest element in the list and so on. The maximum number of passes repeated for 'n' element list is 'n-1'. The time complexity of the bubble sort is O(n2).

```
# Bubble sort pseudocode
bubbleSort(array)
  for i <- 1 to indexOfLastUnsortedElement-1
    if leftElement > rightElement
      swap leftElement and rightElement
end bubbleSort
```

Figure 3. Bubble Sort Pseudocode

3.4 Shell Sort

Shell sort is a very efficient algorithm and is an improvised or more efficient version of insertion sort. The Shell sort overcomes the large shifts/swaps as in case of insertion sort. Especially if the smaller value is to the far right and has to be moved to the far left, the shell sort helps in bringing the smaller values to the left in the starting few passes. The shell sort compares the elements in a certain distance/gap in the initial pass and swaps the elements if required. After each swap the algorithm compares the swapped element with the previous element on the left which is at the same gap. In case there is no element at the left with the same gap or no swapping required, the algorithm moves to the next element and compares it with the element on the right with the same gap and proceeds till it reaches the last element. Once the algorithm reaches the last element, the pass ends and the gap value is reduced. The same steps are carried out with the reduced gap values until the gap reaches 1. Once the gap between elements reaches 1, the Shell sort works the same as an insertion sort but by the time gap reaches 1, all the smaller values to the farther right would have been moved to the farther left, thus reducing the number of swap operations. There are many ways in deciding the gap, however one popular method is to divide the number elements by 2 and consider the base value of the result i.e., n/2. The gap value is divided by 2 after each pass making the value as n/4, n/8 etc., The time complexity of shell sort while following the gap = n/2 method is O(n2).

```

# Shell sort Pseudocode

ShellSort(array, size)
  for interval i <- size/2n down to 1
    for each interval "i" in array
      sort all the elements at interval "i"
end shellSort
```

Figure 4. Shell Sort Pseudocode

4. Experimentation and Observations

To test the sorting algorithms, we ran 2 lists that contained the name of cities around the world. One list contains 14 cities while the other list has 385 cities. We ran the algorithms five times for each list and calculated the average time it took to sort each list. Here are the following average times:

	Time	
Algorithm	Size: 14	Size: 385
Mergesort	4.3e-5 sec.	.0022 secs
Quicksort	2.8e-5 sec.	.0011 secs.
Bubble Sort	3.3e-5 sec.	.0247 secs
Shell Sort	1.8e-5 sec.	.0021 secs.

Table 1: Calculation for average times

For the short list, Shell sort has the fastest time of 1.8e–05 seconds and Mergesort has the slowest time of 4.3e-05 seconds. For the list of size 385, Quicksort has the fastest time while Bubble sort has the slowest time.

5. Complexity and Tractability

5.1 Complexity of Sorting Algorithms

It can be quite difficult to choose which sorting algorithm to use when they all do the job efficiently. Comparing the average case, the worst case, the space complexity, and the stability is important to finding the right algorithm to sort. When there are n elements given in quick sort the number of comparisons will be O(n log2n) in average case and O(n2) comparisons will be made in the worst case. The run time and average case for the merge sort algorithm is O(n log2n). Shell sort has the average case of O(n log n) and worst case of O(n^2). Bubble sort has the average case of O(n) and worst case of O(n^2). Bubble sort seems to be the least efficient out of the four algorithms.

Algorithm	**Average Case**	**Worst Case**	**Space Complexity**	**Stability**
Mergesort	O(n log n)	O(n log n)	O(n)	Yes
Quicksort	O(n log n)	O(n^2)	O(log n)	No
Bubble Sort	O(n)	O(n^2)	O(n)	Yes
Shell Sort	O(n log n)	O(n^2)	O(1)	No

Table 2: Complexity for each sort

5.2 Determining NP

Sorting algorithms are Polynomial Time (P) and they are not NP - Complete. P is when a problem or algorithm can be solved in polynomial time and is efficient. P is the subset of NP (nondeterministic polynomial time) which is the set of decision problems that can be solved in polynomial time. Sorting algorithms are not NP-Complete because there is an efficient solution found in each sorting algorithm. NP-Complete refers to the problem that has yet to find a solution in polynomial time.

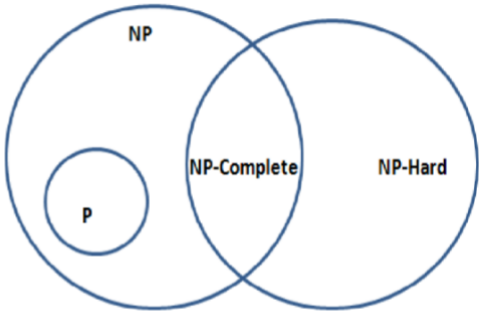

Figure 5: General NP Venn Diagram

6. Conclusion

This paper discusses the four sorting algorithms and their efficiency. Each algorithm was tested to sort a list of cities in ascending order. We found that shell sort ran the fastests while mergesort was the slowest.

We also found that bubble sort runtime increases as the size increases.

Acknowledgements

Assistance in this project was provided by Dr. Aparna Varde of Montclair State University with assisted materials and general guidance through this project. Python was used to implement the sorting algorithms.

References:

1.Singh Tinku and Srivastava Durgesh Kumar (July 2016) Threshold Analysis and Comparison of Sequential and Parallel Divide and Conquer Sorting Algorithms, International Journal of Computer Applications (0975 – 8887) Volume 145 – No.1.

2. Yadav Neelam and Kumari Sangeeta (Feb 2016) Sorting Algorithm, International Research Journal of Engineering and Technology (IRJET) Volume: 03 Issue: 02.

3.Bozidar Dark and Dobravez Tomaz (Nov 2015) Comparison of parallel sorting algorithms Faculty of Computer and Information Science, University of Ljubljana, Slovenia.