5/27/2019

CSP 2348 Assignment 2

A Mini Team Project

Samuel Morris – 10456767

Riegard Lemmer - 10424383

Sam Morris and Riegard Lemmer

ECU JOONDALUP – CSP2348 – Data structures – assignment 2

# Executive Summary

Table of Contents

[Executive Summary 1](#_Toc9788277)

[Introduction 3](#_Toc9788278)

[Bubble and Heap Sort Algorithm Analysis 4](#_Toc9788279)

[Bubble sort algorithm analysis 4](#_Toc9788280)

[Pseudocode 4](#_Toc9788281)

[Heap sort algorithm analysis 4](#_Toc9788282)

[Pseudocode 4](#_Toc9788283)

[Summary of Sort Algorithm Complexities 5](#_Toc9788284)

[Algorithm Analysis by Experimental Studies 6](#_Toc9788285)

[Sequencing Array Sorting Algorithms Based on their Complexity 7](#_Toc9788286)

[Bubble Sort 7](#_Toc9788287)

[Insertion Sort 7](#_Toc9788288)

[Selection Sort 7](#_Toc9788289)

[. Quick Sort 8](#_Toc9788290)

[Merge Sort 8](#_Toc9788291)

[Heap Sort 8](#_Toc9788292)

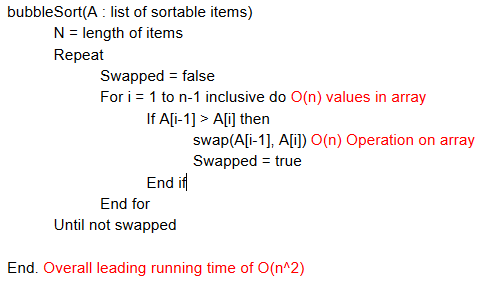
# Introduction

A sorting algorithm is employed to re-arrange any given array or list of elements to improve organization and efficiency. The way in which the array or elements are organised is based on the algorithm; a procedure that sorts the data in a given amount of time. The difference between sorting algorithms are their methods of sorting and efficiencies. O-notation is used to measure algorithm efficiency. It can be calculated based on the number of comparisons or run time of the algorithm (Nazeer and Sebastian 2009). This report will explore theoretical and practical analysis of sorting algorithms through design, implementation and testing. The set of sorting algorithms will include Bubble, Heap, Insertion, Selection, Merge and quick sort. These shall be implemented though arrays, singly linked lists and binary tree data structures.

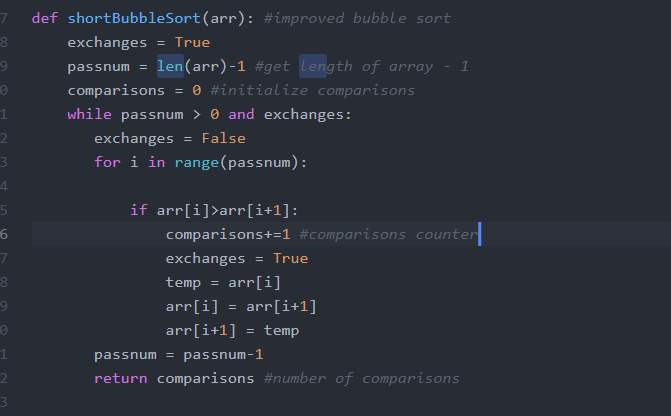
# Bubble and Heap Sort Algorithm Analysis

## Bubble sort algorithm analysis

### Pseudocode



## Improved bubble sort algorithm analysis

Bubblesortshot(array: List of elements)

Swapped = true

Variable = length of array(arr) - 1

Comparison counter = 0

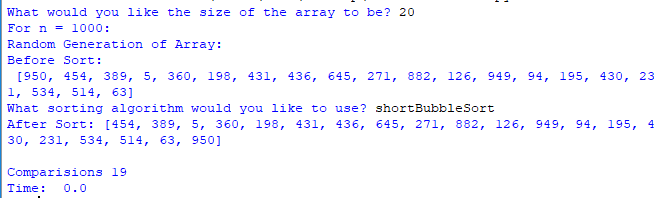
While the length of the array is greater than 0

set swapped to false

For the value “i” in the whole array O(n) values in array

If the variable i in the array is greater than variable i +1

Count comparison counter +1

 Set exchange to true

Set a variable to the current value in the array

Make the current position of i = to the next position O(n)swap

Set the next position to the temporary variable

Size of array - 1

Return the comparisons

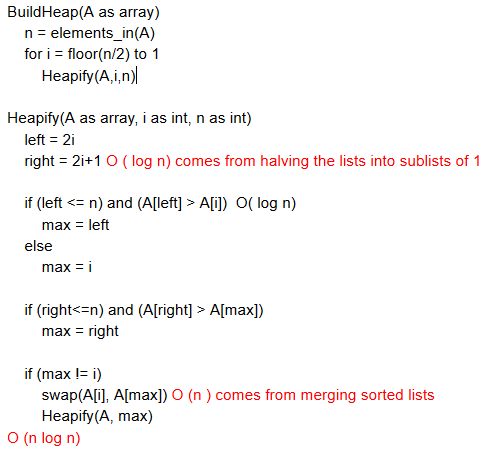
O(N^2)

## Justification

The unimproved bubble sort was chosen for analysis as it was the original version and other algorithms also have improved versions that testing is not done for in this implementation.

## Heap sort algorithm analysis

### Pseudocode



# Summary of Sort Algorithm Complexities

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Sorting Algorithm | No. of comparisons | No. of  Copies | Time complexity | Space complexity |
| Bubble | ~ O(n2) | BC:0  WC: n(n-1)/2 | BC: O(N)  WC: O(n2) | O (1) |
| Selection | *~* | *~ 2n* | *BC:* O(n2)  WC: O(n2) | *O (1)* |
| Insertion | *~* n2 */ 4* | *~*n2 */ 4* | *BC:O(N)*  *WC:* O(n2) | *O (1)* |
| Merge | *~n log2n* | *~ 2n log2n* | *BC: O (n log2n)*  WC:*O (n log2n)* | *O(n)* |
| Quick | *\*BC: ~n log2n*  *\*WC: ~* n2 */2* | *BC: ~2n/3 log2n*  *WC: 0* | *BC: O (n log2n)*  *WC:* O (n2) | *\*BC: O ()*  *\*WC: O(n)* |
| Heap | O () | BC: O (1)  WC:O () | BC: O ()  WC: O () | BC; O (n)  WC: O (1) |

*Notes \*: BC – Best case, WC Worst case*

# Algorithm Analysis by Experimental Studies

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Comparison | N = 200 | N = 400 | N = 800 | N = 1000 | N = 2000 |
| Bubble | 19900 | 79800 | 319600 | 499500 | 1999000 |
| Selection | 20099 | 80199 | 320399 | 500499 | 2000999 |
| Insertion | 10412 | 39838 | 160893 | 254158 | 1901844 |
| Merge | 1279 | 2963 | 6721 | 8705 | 19417 |
| Quick | 1585 | 3608 | 8721 | 11202 | 25191 |
| Heap | 1021 | 2433 | 5638 | 7359 | 16671 |

*Table measuring the number of comparisons for various algorithms with varying n values*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Time | N = 200 | N = 400 | N = 800 | N = 1000 | N = 2000 |
| Bubble | 9 | 401 | 172 | 273 | 1.134 |
| Selection | 9 | 24 | 91 | 154 | 0.534 |
| Insertion | 5 | 20 | 94 | 146 | 0.592 |
| Merge | 2 | 3 | 7 | 9 |  |
| Quick | 1 | 2 | 4 | 6 | 4 |
| Heap | 2 | 5 | 11 | 11 | 34 |

*Table measuring the run time for each algorithm in Ms, rounded to the nearest millisecond*

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A/N | 200 T | 200 C | 400 T | 400 C | 800 T | 800 C | 1000 T | | 1000 C | 2000 T | 2000 C |
| B | 6 | 6 | 6 | 6 | 6 | 6 | | 6 | 6 | 6 | 6 |
| I | 5 | 5 | 5 | 5 | 4 | 5 | | 5 | 5 | 5 | 5 |
| S | 4 | 1 | 4 | 1 | 5 | 1 | | 4 | 1 | 4 | 1 |
| H | 2 | 2 | 3 | 2 | 3 | 2 | | 3 | 2 | 2 | 2 |
| Q | 1 | 4 | 1 | 4 | 1 | 4 | | 1 | 4 | 1 | 4 |
| M | 3 | 3 | 2 | 3 | 2 | 3 | | 2 | 3 | 3 | 3 |

*A =algorithm, N = number, C = comparisons, T = Time, B = Bubble, I = Insertion, S = Selection, H =Heap, Q= quick, M = Merge,*

# Sequencing Array Sorting Algorithms Based on their Complexity

## Bubble Sort (unoptimized)

The bubble sort algorithm has an average performance of O(n2), the performance of this algorithm is relative to its size, so It would rarely be used to sort large data sets. Bubble sort works well on small sets of data where the inefficiency doesn’t affect the overall performance as much. The most efficient use of bubble sort would be for a list that is mostly sorted as time would be relative to the number of unsorted elements. One unsorted element would give 2n time, two would take 3n time and so on. In testing bubble ranked 6th in both time and comparisons making it the worst sorting algorithm.

## Insertion Sort

The insertion sort algorithm can be compared to an improved version of selection sort. It is quite efficient on smaller set of data and mostly sorted sets of data, because of this it is often used in conjunction with more complex algorithms. The downfall of insertion is that computationally expensive due to its shifting of all following elements when it shifts any element. In testing, Insertion ranked 5th in both time and comparisons making it the 2nd worst sorting algorithm.

## Selection Sort

The selection sort algorithm has O(n2) complexity, which is like the bubble sort as its performance is relative to its size. Selection sort is notable for how simple it is, and under specific circumstances has better performance than algorithms with greater complexity. It generally performs worse than much alike insertion sort. In testing, Selection ranks 4th for time and 1st every time for comparisons.

## . Quick Sort

The quick sort algorithm users a partition and pivot to sort elements and this gives it great linear time efficiency. Quick sort can be quite a complex algorithm but in practice it is quite fast. With its mediocre space usage of O (log n) all round quick sort is one of the top algorithms, although poor pivot selection can alter the performance significantly. In testing, Quick ranks 1st in time and 4th in comparisons.

## Merge Sort

The merge sort algorithms worst run time is O (n log n), this means that it scales well when sorting large sets of data. Merge sorts efficiency stems from when it combines list as this is one if its main operations. The O (n log n) run time is enables it to scale well with large sets of data. In testing, Merge ranks 2nd for lower values in time and 3rd for higher values, for comparisons it ranks 3rd every time.

## Heap Sort

Heap sort is a more advanced selection sorting algorithm and operates by determining the largest value, then allocating it to the end of the list while continuing with the rest of the data. The heap sort accomplishes the sorting task through a binary tree which operates through the elements at an O (n log n) time which makes it a very efficient sorting algorithm for sorting large sets of data compared to selection sort which operates at O(n). In testing, heap sort performed the second best in the number of comparisons in all instances, while attaining a second best for a 200 and 2000 element count while being third for a 400, 800 and 1000 element count.

## Recommendation

According to the data obtained above, the best algorithm to use for the sorting of data would be the quick sort algorithm. Although quicksort has a rather high comparison count, it does have the best run time compared to the other sorting algorithms. Compared to the other algorithms quicksort comes third for the number of comparisons only beaten by Merge (second) and Heap (first) sort. The time taken to sort through data from sizes 1000 to 2000 does go up exponentially, it is still faster than the other algorithms.

The best approach to have when organising data would be to combine two sorting algorithms to assure that the best aspects of both the algorithms. For example the use of quick to organise the chunks while the data is merged as per merge sort (Sorting Algorithms).

## Ranking of Algorithms

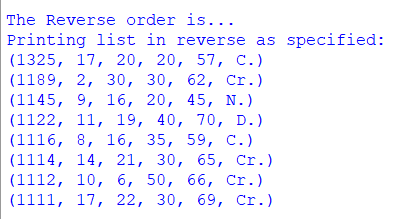
## 

|  |  |  |
| --- | --- | --- |
| A | Time (Ms) | Comparisons (Number) |
| B | 6th every instance | 6th every instance |
| I | 5th every instance  Except 3th Worst at 800 | 5th every instance |
| S | 4th most instances  Except 5th at 800, | 1st every instance |
| H | 2nd or 3rd every instance  2nd at 200,2000, 3rd at 400,800,1000 | 2nd Best every instance |
| Q | 1st every instance | 4th every instance |
| M | 2nd or 3rd every instance  2nd at 400,800,1000 3rd at 200,2000 | 3rd every instance |

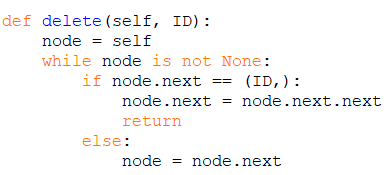
*B = Bubble, I = Insertion, S = Selection, H =Heap, Q= quick, M = Merge,*

# Linked-list programming

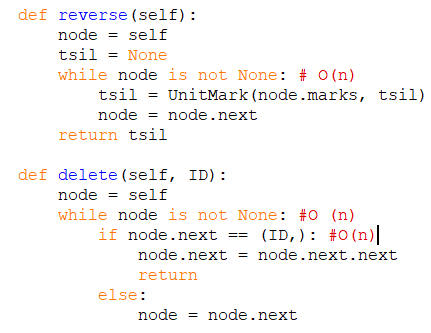
a)



b)

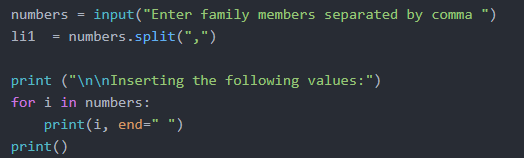


2)

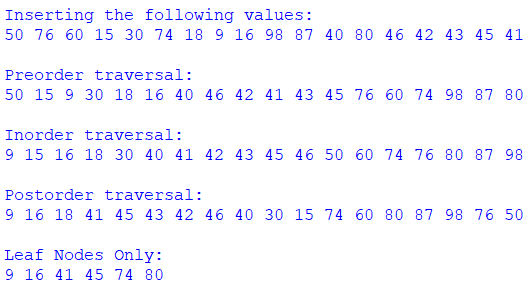


# Binary Tree Traversal Application

a)



c)



# Conclusion

The experimental analysis that was conducted highlights how theoretical analysis give’s quite an accurate approximation of number of comparisons. However, when it comes to runtime, the time can vary greatly. This is due to individual nuances in algorithms and varying set of data. Data structures are becoming more prevalent in society today due to the rapid advancements of technology. As more research is done into sorting algorithms, more efficient methods shall be found; whether it be due to a new implementation or hybrid combination.

# References

A Sort of All Sorting Algorithms. (2019). Retrieved from <https://levelup.gitconnected.com/a-sort-of-all-sorting-algorithms-506cbc76d47?gi=feec654f610f>

A tour of the top 5 sorting algorithms. (2019). Retrieved from <https://medium.com/@george.seif94/a-tour-of-the-top-5-sorting-algorithms-with-python-code-43ea9aa02889>

Analysis of different sorting techniques - GeeksforGeeks. (2019). Retrieved from <https://www.geeksforgeeks.org/analysis-of-different-sorting-techniques/>

Bubble Sort | Brilliant Math & Science Wiki. (2019). Retrieved from <https://brilliant.org/wiki/bubble-sort/>

Heap sort - Algorithmist. (2019). Retrieved from <https://www.algorithmist.com/index.php/Heap_sort>

Insertion Sort - GeeksforGeeks. (2019). Retrieved from <https://www.geeksforgeeks.org/insertion-sort/>

Nazeer, K. A. and M. Sebastian (2009). Improving the Accuracy and Efficiency of the k-means Clustering Algorithm. Proceedings of the world congress on engineering, Association of Engineers London.