CSP2348.3 - Data Structures

Assignment 2 - Algorithm implementation and analysis

Student ID: 10636908

Name: Poorav Sharma

Due Date: 05/05/2023

Table of Contents

Introduction

This report showcases the design, analysis, implementation, and performances of the array-based sorting algorithms. It answers three questions which requires the application of mathematical theory to algorithm analysis and analyse complexity and performance of the algorithms associated with it. The first question is answered by using algorithm design, algorithm analysis using O-notation, and implementing the algorithm in Python. The second question is based on theoretical algorithm analysis. It contains a table of array sorting algorithms and its complexities. The third question shows an experimental study on several array-based sorting algorithms.

Question 1: Array sorting algorithm: design, implementation, and analysis

Question 1.1: Bubble sort algorithm

a) Pseudocode

sorting array in ascending order

n = number of elements in array

```
FOR repeat until i equals n
        FOR repeat until j equals n-1
                IF array[j] is bigger than array [j+1]
                        swap (array[j], [j+1])
                END IF
        END FOR
END FOR
```

b) Analysis

The algorithm will compare the element beside each other until the largest number is at the end of the array. The comparison will happen n-1 time in inner loop because that is the maximum number for comparison that can be made in one iteration. The algorithm will repeat iteration n number of times. Therefore, the number of comparisons made in this algorithm will be n*(n-1). The O-notation is $O(n^2)$ because $n*(n-1) = n^2-n$.

As for the number of copying operation it will depend on the how unsorted the array is. In the best case where the array is already sorted then the number of copying operation will be 0. In the worst-case scenario where the array is arranged from biggest to smallest (descending) order the number of copying operation will equal n*(n-1)/2. The average at worst case will be half of the equation which is n2-n/4. The O-notation is O(n^2) for worst case and average because (n*(n-1))/2 = (n^2-n)/2 and (n*(n-1))/2 * ½ = (n^2-n)/4.

c) Testing algorithm

Image 1.1 below shows the array with 20 elements when it is unsorted. Afterwards it shows the array when it has been sorted by the algorithm created.

```
Question 1.1 Bubble sort algorithm

Unsorted Arrays:  [-89, 25, 57, 17, 44, -34, 61, 20, -48, 98, -30, -80, -93, -21, -7, -96, -66, -67, -71, -68]

Sorted Array:  [-96, -93, -89, -80, -71, -68, -67, -66, -48, -34, -30, -21, -7, 17, 20, 25, 44, 57, 61, 98]

Number of comparisons: 380

Number of copies(swaps): 129
```

Below it also shows the number of comparisons it made, and the number of copies (swaps) made.

Image 1.1 Test result of bubble algorithms.

Question 1.2: Improvement to the bubble sort algorithm

Question 1.2.1: Observation 1

a) Pseudocode

sorting array in ascending order

n = number of elements in array

FOR repeat until i equals n

    FOR repeat until j equal n-i-1

        IF array[j] bigger than array [j+1]

            swap (array[j], [j+1])

        END IF

    END FOR

END FOR

b) Analysis

The algorithm will compare the element beside each other until the largest number is at the end of the array. The comparison will happen n-1-i time in inner loop because the elements that have been sorted doesn't have to be compared again. The algorithm will repeat iteration n number of times. Therefore, the maximum number of comparisons made in this algorithm will be halved making n*(n-1) = (n*(n-1))/2. The O-notation is O(n^2) because (n*(n-1))/2 = (n^2-n)/2.

As for the number of copying operation it will depend on the how unsorted the array is. In the best case where the array is already sorted then the number of copying operation will be 0. In the worst-case scenario where the array is arranged from biggest to smallest (descending) order the number of copying operation will equal (n*(n-1))/2. The average at worst case will be half of the equation which is (n2-n)/4. The O-notation is O(n^2) for worst case and average because (n*(n-1))/2 = (n^2-n)/2 and (n*(n-1))/2 * ½ = (n^2-n)/4.

c) Testing algorithm

Image 1.2.1 below shows the array with 20 elements when it is unsorted. Afterwards it shows the array when it has been sorted by the new observation 1 algorithm created. Below it also shows the number of comparisons it made have been halved compared to the original algorithm. It also shows the number of copies (swaps) made.

```
Question 1.2.1 Observation 1-BubbleSort algorithm

Unsorted Arrays:  [86, -92, 77, 31, 83, -68, -50, 78, 37, 91, -15, -51, -57, -90, -12, 51, -37, -32, 8, -54]

Sorted Array:  [-92, -90, -68, -57, -54, -51, -50, -37, -32, -15, -12, 8, 31, 37, 51, 77, 78, 83, 86, 91]

Number of comparisons: 190

Number of copies(swaps): 112
```

Image 1.2.1 Test result of Observation 1-BubbleSort algorithm

Question 1.2.2: Observation 2

a) Pseudocode

sorting array in ascending order

n = number of elements in array

FOR repeat until i equals n

```
Set swapped to False.
    FOR repeat until j equal n-1
        IF array[j] bigger than array [j+1]
            swap (array[j], [j+1])
            Set swapped to True.
        END IF
    END FOR
    IF swapped is not True
        End loop
    END IF
END FOR
```

b) Analysis

The algorithm will compare the element beside each other until the largest number is at the end of the array. The comparison will happen n-1 time in inner loop. The algorithm will repeat iteration n number of times. If the loop was in the worst case (array in descending order), the maximum number of comparisons made in this algorithm will be n*(n-1). The O-notation for worst case is O(n^2) because n*(n-1) = (n^2-n). In the best case where the array is already sorted then the algorithm will only iterate n-1 times in the inner loop and terminate. This is because it no swapped is made the Boolean and if statement placed will active and break away from the loop and stop unwanted comparisons. The O-notation for best case is O(n) because n-1 = n.

As for the number of copying operation it will depend on the how unsorted the array is. In the best case where the array is already sorted then the number of copying operation will be 0. In the worst-case scenario where the array is arranged from biggest to smallest (descending) order the number of copying operation will equal (n*(n-1))/2. The average at worst case will be half of the equation which is (n2-n)/4. The O-notation is O(n^2) for worst case and average because (n*(n-1))/2 = (n^2-n)/2 and (n*(n-1))/2 * ½ = (n^2-n)/4.

c) Testing algorithm

Image 1.2.2 below shows the array with 20 elements when it is unsorted. Afterwards it shows the array when it has been sorted by the new observation 2 algorithm created. Below it also shows the number of comparisons and the

number of copies (swaps) made. In the second test a sorted array is put into new sorting algorithm to test whether the algorithm terminates as required if there are no elements being swapped.

```
Question 1.2.2 Observation 2-BubbleSort algorithm

Unsorted Arrays:  [-2, -55, -7, 14, -52, 29, -84, 62, 50, -59, 96, 54, 82, 41, -38, 88, 55, 37, -19, 4]

Sorted Array:  [-84, -59, -55, -52, -38, -19, -7, -2, 4, 14, 29, 37, 41, 50, 54, 55, 62, 82, 88, 96]

Number of comparisons: 266

Number of copies(swaps): 75

===== RESTART: H:\Assessment\Computer Science Bachlore\Data Structures\Assessment 2\Code\Obs2-BubbleSort algorithm.p
Question 1.2.2 Observation 2-BubbleSort algorithm

Unsorted Arrays:  [-84, -59, -55, -52, -38, -19, -7, -2, 4, 14, 29, 37, 41, 50, 54, 55, 62, 82, 88, 96]

Sorted Array:  [-84, -59, -55, -52, -38, -19, -7, -2, 4, 14, 29, 37, 41, 50, 54, 55, 62, 82, 88, 96]

Number of comparisons: 19

Number of copies(swaps): 0
```

Image 1.2.2 Test result of Observation 2-BubbleSort algorithm

Question 1.2.3: Observation 3

a) Pseudocode

```
sorting array in ascending order
n  = number of elements in array
FOR repeat until i equals n
Set swapped to False.
    FOR repeat until j equal n-1-i
        IF array[j] bigger than array [j+1]
            swap (array[j], [j+1])
            Set swapped to True.
        END IF
    END FOR
    IF swapped is not True
        End loop
    END IF
END FOR
```

b) Analysis

The algorithm will compare the element beside each other until the largest number is at the end of the array. The comparison will happen n-1-i times in inner loop because the elements that have been sorted doesn't have to be compared again. The algorithm will repeat iteration n number of times. If the loop was in the worst case (array in descending order), the maximum number of comparisons made in this algorithm will be n*(n-1)/2. The O-notation for worst case is O(n^2) because (n*(n-1))/2 = (n^2-n)/2. In the best case where the array is already sorted then the algorithm will only iterate n-1 times in the inner loop and terminate. This is because it no swapped is made the Boolean and if statement placed will active and break away from the loop and stop unwanted comparisons. The O-notation for best case is O(n) because n-1 = n.

As for the number of copying operation it will depend on the how unsorted the array is. In the best case where the array is already sorted, the number of copying operation will be 0. In the worst-case scenario where the array is arranged from biggest to smallest (descending) order the number of copying operation will equal (n*(n-1))/2. The average worst case will be half of the equation which is (n2-n)/4. The O-notation is O(n^2) for worst case and average because (n*(n-1))/2 = (n^2-n)/2 and (n*(n-1))/2 * ½ = (n^2-n)/4.

c) Testing algorithm

Image 1.2.3 below shows the array with 20 elements when it is unsorted. Afterwards it shows the array when it has been sorted by the new observation 3 algorithm created which utilises new components used in observation 1 and 2. The test also shows the number of comparisons and the number of copies (swaps) made. In the second test a sorted array is put into new sorting algorithm to test whether the algorithm terminates as required if there are no elements being swapped. The last test shows the worst case (array with element in descending order). It shows the maximum number of comparisons and copies(swaps) the algorithm needed to sort the array. ((n*(n-1)/2) = (20*(20-19))/2 = (20*19)/2 = 380/2 = 190.

```
Question 1.2.3 Observation 3-BubbleSort algorithm

Unsorted Arrays:  [-94, -92, 15, 37, 62, 77, -87, -34, 83, 96, -73, 18, -95, -2, -7, 28, 59, 53, 46, 61]

Sorted Array:  [-95, -94, -92, -87, -73, -34, -7, -2, 15, 18, 28, 37, 46, 53, 59, 61, 62, 77, 83, 96]

Number of comparisons: 169

Number of copies(swaps): 71

====== RESTART: H:/Assessment/Computer Science Bachlore/Data Structures/Assessment 2/Code/Obs3-BubbleSort algorit
Question 1.2.3 Observation 3-BubbleSort algorithm

Unsorted Arrays:  [-95, -94, -92, -87, -73, -34, -7, -2, 15, 18, 28, 37, 46, 53, 59, 61, 62, 77, 83, 96]

Sorted Array:  [-95, -94, -92, -87, -73, -34, -7, -2, 15, 18, 28, 37, 46, 53, 59, 61, 62, 77, 83, 96]

Number of comparisons: 19

Number of copies(swaps): 0

====== RESTART: H:/Assessment/Computer Science Bachlore/Data Structures/Assessment 2/Code/Obs3-BubbleSort algorit
Question 1.2.3 Observation 3-BubbleSort algorithm

Unsorted Arrays:  [96, 83, 77, 62, 61, 59, 53, 46, 37, 28, 18, 15, -2, -7, -34, -73, -87, -92, -94, -95]

Sorted Array:  [-95, -94, -92, -87, -73, -34, -7, -2, 15, 18, 28, 37, 46, 53, 59, 61, 62, 77, 83, 96]

Number of comparisons: 190

Number of copies(swaps): 190
```

Image 1.2.3 Test result of Observation 3-BubbleSort algorithm

Question 1.3: Variation of the bubble algorithm

Question 1.3.1: Sink-down sort algorithm

a) Pseudocode

sorting array in ascending order

n = number of elements in array

FOR repeat until i equals n

Iterate i until it equals n

Set swapped to False.

    FOR repeat until j equal i

    Set j as n-1 loop until equals j equals i

        IF array[j] bigger than array [j-1]

            swap (array[j], [j-1])

            Set swapped to True.

        END IF

    END FOR

    IF swapped is not True

End loop

END IF

END FOR

b) Analysis

This new variation of bubble sort is called sink sort does the exact same thing as a bubble sort except for the order in which the array is sorted in. Unlike bubble sort which sorts the array in ascending order, sink sort algorithm sorts the arrays in descending order. The sink sort applies strategies used in observation 3. Therefore, when the array is already sorted the number of comparisons made is n-1. The O-notion for comparisons for best case is O(n) because n-1 = n. In the worst case (array is in ascending order) the maximum comparisons are (n*(n-1))/2. The O-notation for worst case is O(n^2) because (n*(n-1))/2 = (n^2-n)/2 .

The number of copies made depends on how unsorted the array is. In the best case where the array is already sorted, the number of copying operation will be 0. In the worst-case scenario where the array is arranged from smallest to biggest (ascending) order the number of copying operation will equal (n*(n-1))/2. The average worst case will be half of the equation which is (n2-n)/4. The O-notation is O(n^2) for worst case and average because (n*(n-1))/2 = (n^2-n)/2 and (n*(n-1))/2 * ½ = (n^2-n)/4.

c) Testing algorithm

Image 1.3.1 below shows the array with 20 elements when it is unsorted. Afterwards it shows the array when it has been sorted by using the sink sort algorithm which utilises strategies used in observation 1 and 2. The test also shows the number of comparisons and the number of copies (swaps) made. In the second test a sorted array is put into sink sort algorithm to test whether the algorithm terminates as required if there are no elements being swapped. The last test shows the worst case (array with element in ascending order). It shows the maximum number of comparisons and copies(swaps) the algorithm needed to sort the array. ((n*(n-1)/2) = (20*(20-19))/2 = (20*19)/2 = 380/2 = 190.

```
Question 1.3.1 SinkSort algorithm

Unsorted Arrays:  [51, -39, -88, 39, -1, 98, -46, -27, 84, -82, 52, -23, 53, -66, 93, 12, -11, 86, 48, 24]

Sorted Array:  [98, 93, 86, 84, 53, 52, 51, 48, 39, 24, 12, -1, -11, -23, -27, -39, -46, -66, -82, -88]

Number of comparisons: 189

Number of copies(swaps): 110

= RESTART: H:\Assessment\Computer Science Bachlore\Data Structures\Assessment 2\Code\Question1.31Sink Sort.py
Question 1.3.1 SinkSort algorithm

Unsorted Arrays:  [98, 93, 86, 84, 53, 52, 51, 48, 39, 24, 12, -1, -11, -23, -27, -39, -46, -66, -82, -88]

Sorted Array:  [98, 93, 86, 84, 53, 52, 51, 48, 39, 24, 12, -1, -11, -23, -27, -39, -46, -66, -82, -88]

Number of comparisons: 19

Number of copies(swaps): 0

= RESTART: H:\Assessment\Computer Science Bachlore\Data Structures\Assessment 2\Code\Question1.31Sink Sort.py
Question 1.3.1 SinkSort algorithm

Unsorted Arrays:  [-88, -82, -66, -46, -39, -27, -23, -11, -1, 12, 24, 39, 48, 51, 52, 53, 84, 86, 93, 98]

Sorted Array:  [98, 93, 86, 84, 53, 52, 51, 48, 39, 24, 12, -1, -11, -23, -27, -39, -46, -66, -82, -88]

Number of comparisons: 190

Number of copies(swaps): 190
```

Image 1.3.1 Test result of Sink Sort algorithm

Question 1.3.2: Bi-directional-bubble sort (BDBS) algorithm

a) Pseudocode

sorting array in ascending order

n = number of elements in array

FOR repeat until i equals n

Iterate i until it equals n

Set swapped to False.

    Set j as i

    FOR repeat until j equal n-1-i

        IF array[j] bigger than array [j+1]

            swap (array[j], [j+1])

            Set swapped to True.

        END IF

    END FOR

    IF swapped is not True

        End loop

```
        END IF


Set swapped to False.
        FOR repeat until k equal i
        Set k as n-1 loop until equals k equals i
                IF array[k] bigger than array [k-1]
                        swap (array[k], [k-1])
                        Set swapped to True.
                END IF
        END FOR
        IF swapped is not True
                End loop
        END IF
END FOR
```

b) Analysis

This new variation of bubble sort algorithm is called Bi-directional-bubble sort (BDBS) algorithm. This sort does the exact same thing as a bubble sort except but this time it also utilises sink sort algorithm in its inner loop. This means in the inner look of this algorithm there are two for loop. One for loop scans the array from left to right and places the largest element on right (end). The second loop scans the array from right to left and places the smallest element on the left (beginning) of the array. The number of comparisons made is the same as sink sort and observation 3 bubble sort. This is because every time an element is sorted in the beginning or end using the two loops they are removed from the scan in the future loop. It uses the Boolean swapped to end the loop if there were no swaps in either of the inner loops. When the array is already sorted (best case - array is in ascending order) the number of comparisons made is n-1. The O-notion for comparisons for best case is O(n) because n-1 = n. In the worst case (array is in ascending order) the maximum comparisons are (n*(n-1))/2. The O-notation for worst case is $O(n^2)$ because (n*(n-1))/2 = (n^2-n)/2 .

The number of copies made depends on how unsorted the array is. In the best case where the array is already sorted, the number of copying operation will be

0. In the worst-case scenario where the array is arranged from biggest to smallest (descending) order the number of copying operation will equal (n*(n-1))/2. The average worst case will be half of the equation which is (n2-n)/4. The O-notation is O(n^2) for worst case and average because (n*(n-1))/2 = (n^2-n)/2 and (n*(n-1))/2 * ½ = (n2-n)/4.

c) Testing algorithm

Image 1.3.2 below shows the array with 20 elements when it is unsorted. Afterwards it shows the array when it has been sorted by using the sink sort algorithm which utilises strategies used in observation 1 and 2. The test also shows the number of comparisons and the number of copies (swaps) made. In the second test a sorted array is put into sink sort algorithm to test whether the algorithm terminates as required if there are no elements being swapped. The last test shows the worst case (array with element in ascending order). It shows the maximum number of comparisons the algorithm needed to sort the array. ((n*(n-1)/2) = (20*(20-19))/2 = (20*19)/2 = 380/2 = 190. The maximum number of copies is one less because the array has a element that repeats twice.

```
Question 1.3.2 Bi-directional-bubble sort (BDBS) algorithm

Unsorted Arrays:  [95, 100, 70, 98, -18, 20, -99, -43, 20, -30, 69, -40, -17, -37, 66, 24, -60, 96, -29, -23]

Sorted Array:  [-99, -60, -43, -40, -37, -30, -29, -23, -18, -17, 20, 20, 24, 66, 69, 70, 95, 96, 98, 100]

Number of comparisons: 169

Number of copies(swaps): 116
>>
= RESTART: H:/Assessment/Computer Science Bachlore/Data Structures/Assessment 2/Code/Question1.3.2 Bi-Directional -BubbleSort algor
ithm.py
Question 1.3.2 Bi-directional-bubble sort (BDBS) algorithm

Unsorted Arrays:  [-99, -60, -43, -40, -37, -30, -29, -23, -18, -17, 20, 20, 24, 66, 69, 70, 95, 96, 98, 100]

Sorted Array:  [-99, -60, -43, -40, -37, -30, -29, -23, -18, -17, 20, 20, 24, 66, 69, 70, 95, 96, 98, 100]

Number of comparisons: 19

Number of copies(swaps): 0
>>
= RESTART: H:/Assessment/Computer Science Bachlore/Data Structures/Assessment 2/Code/Question1.3.2 Bi-Directional -BubbleSort algor
ithm.py
Question 1.3.2 Bi-directional-bubble sort (BDBS) algorithm

Unsorted Arrays:  [100, 98, 96, 95, 70, 69, 66, 24, 20, 20, -17, -18, -23, -29, -30, -37, -40, -43, -60, -99]

Sorted Array:  [-99, -60, -43, -40, -37, -30, -29, -23, -18, -17, 20, 20, 24, 66, 69, 70, 95, 96, 98, 100]

Number of comparisons: 190

Number of copies(swaps): 189
>>
```

Image 1.3.2 Test result of Bi-directional-bubble sort (BDBS) algorithm

Question 2: Summary of theoretical analysis of array-based sorting algorithms

Note that in **Table 1** only quick sort has the same average complexity as its best case. As for all the other sorting algorithm with worst-cases its worst- case time complexity is the same as their average O-notation.

Sorting algorithm	**Number of comparisons**	**Number of copies**	**Time complexity**	**Space complexity**
Selection	~ $n^2/2$	~ 2n	$O(n^2)$	O(1)
Insertion	~ $n^2/4$	~ $n^2/4$	$O(n^2)$	O(1)
Merge	~ n $\log_2 n$	~ 2n $\log_2 n$	O(n $\log_2 n$)	O(n)
Quick	*BC: ~ n $\log_2 n$ *WC: ~ $n^2/2$	*BC: ~ 2n/3 $\log_2 n$ *WC: 0	*BC: O(n $\log_2 n$) *WC: $O(n^2)$	*BC: O($\log_2 n$) *WC: O(n)
Heap	~2*(n $\log_2 n$)	~ n $\log_2 n$	O(n $\log_2 n$)	O(1)
Bubble	n(n-1)	BC: 0 WC: n(n-1)/2	$O(n^2)$	O(1)

Obs1-Bubble	n(n-1)/2	BC:0 WC: n(n-1)/2	$O(n^2)$	O(1)
Obs2-Bubble	BC: n-1 WC: n(n-1)	BC:0 WC: n(n-1)/2	BC: O(n) WC: $O(n^2)$	O(1)
Obs3-Bubble	BC: n-1 WC: n(n-1)/2	BC:0 WC: n(n-1)/2	BC: O(n) WC: $O(n^2)$	O(1)
Sink-down	BC: n-1 WC: n(n-1)/2	BC:0 WC: n(n-1)/2	BC: O(n) WC: $O(n^2)$	O(1)
Bi-directional	BC: n-1 WC: n(n-1)/2	BC:0 WC: n(n-1)/2	BC: O(n) WC: $O(n^2)$	O(1)

Notes: *: BC – Best Case; WC- Worst Case

Table 1: Array sorting algorithm complexity (by Theoretical analysis)

Question 3: Algorithm analysis by experimental study

Question 3.1: Developing a system to test various cases of sorting algorithms.

Question 3.1.1: Testing individual sorting algorithm.

The images below are the test result for each sorting algorithm.

Selection Sort

```
Enter number to choose option: 1

Enter number to element you want in the array: 20
Unsorted Array:  [-367, -460, 169, 930, 616, -493, -575, 1189, 1941, -930, 1682, 256, 363, -1352, 304, 1218, -
948, -1850, -932, -1452]

Testing selection sort algorithm

Sorted Array:  [-1850, -1452, -1352, -948, -932, -930, -575, -493, -460, -367, 169, 256, 304, 363, 616, 930, 1
189, 1218, 1682, 1941]

Run time (ms):  0.0
Number of Comparisons:  190

1. Test an individual sorting algorithm

2. Test multiple sorting algorithms

3. Exit


Enter number to choose option:
```

Image 3.1.1 selection sort test

Iteration Sort

```
Enter number to choose option: 2

Enter number to element you want in the array: 20
Unsorted Array:  [1941, 946, 656, -1531, -1493, 1053, -1657, -1584, 1503, -495, 314, 335, 1984, 1721, -1702, -
261, 1651, 721, 47, 807]

Testing insertion sort algorithm

Sorted Array:  [-1702, -1657, -1584, -1531, -1493, -495, -261, 47, 314, 335, 656, 721, 807, 946, 1053, 1503, 1
651, 1721, 1941, 1984]

Run time (ms):  0.0
Number of Comparisons:  105
```

Image 3.1.2 Iteration sort test

Merge Sort

```
Enter number to choose option: 3

Enter number to element you want in the array: 20
Unsorted Array:  [1335, -381, -279, 1762, 758, 478, -809, -1187, 477, -1726, 20, -1209, -4, -815, 987, 1456, 1032, 41, 1323, -1531]

Testing merge sort algorithm

Sorted Array:  [-1726, -1531, -1209, -1187, -815, -809, -381, -279, -4, 20, 41, 477, 478, 758, 987, 1032, 1323, 1335, 1456, 1762]

Run time (ms):  0.0
Number of Comparisons:  63
```

Image 3.1.3 Merge sort test

Quick Sort

```
Enter number to choose option: 4

Enter number to element you want in the array: 20
Unsorted Array:  [-441, -1651, 1081, -1275, -208, 889, -302, -848, 447, 1841, 916, -1574, 714, 1781, 859, 1787, 227, 1040, -117, 1636]

Testing quick sort algorithm

Sorted Array:  [-1651, -1574, -1275, -848, -441, -302, -208, -117, 227, 447, 714, 859, 889, 916, 1040, 1081, 1636, 1781, 1787, 1841]

Run time (ms):  0.0
Number of Comparisons:  94
```

Image 3.1.4 Quick sort test

Heap Sort

```
Enter number to choose option: 5

Enter number to element you want in the array: 20
Unsorted Array:  [1678, 111, 324, 1473, 1545, 289, 1780, 1761, -688, 666, -59, -655, -187, 1541, 40, 302, -677, 868, 1113, 1379]

Testing heap sort algorithm

Sorted Array:  [-688, -677, -655, -187, -59, 40, 111, 289, 302, 324, 666, 868, 1113, 1379, 1473, 1541, 1545, 1678, 1761, 1780]

Run time (ms):  0.0
Number of Comparisons:  177
```

Image 3.1.5 Heap sort test

Bubble Sort

```
Enter number to choose option: 6

Enter number to element you want in the array: 20
Unsorted Array:  [1541, -1702, 937, -1060, -1454, 1670, -426, 938, 1128, -1212, 1807, 1298, -1747, -1136, -1646, -1970, -2000, 1416, -1803,
1914]

Testing bubble sort algorithm

Sorted Array:  [-2000, -1970, -1803, -1747, -1702, -1646, -1454, -1212, -1136, -1060, -426, 937, 938, 1128, 1298, 1416, 1541, 1670, 1807, 19
14]

Run time (ms):  0.0
Number of Comparisons:  380
```

Image 3.1.6 Bubble sort test

Obs1bubble Sort

```
Enter number to choose option: 7

Enter number to element you want in the array: 20
Unsorted Array:  [-219, -313, -1792, -289, 1228, 1812, 1823, 403, 1891, -1623, 831, 1313, 1830, 234, -776, 248, -304, 447, -662, -617]

Testing Obs1-bubble sort algorithm

Sorted Array:  [-1792, -1623, -776, -662, -617, -313, -304, -289, -219, 234, 248, 403, 447, 831, 1228, 1313, 1812, 1823, 1830, 1891]

Run time (ms):  0.0
Number of Comparisons:  190
```

Image 3.1.7 Obs1bubble sort test

Obs2bubble Sort

```
Enter number to choose option: 8

Enter number to element you want in the array: 20
Unsorted Array:  [1878, 1745, -256, -1680, 468, -713, 340, 1162, 487, 916, 879, -1424, -741, -778, 663, 905, 805, -1690, -1713, -760]

Testing Obs2-bubble sort algorithm

Sorted Array:  [-1713, -1690, -1680, -1424, -778, -760, -741, -713, -256, 340, 468, 487, 663, 805, 879, 905, 916, 1162, 1745, 1878]

Run time (ms):  0.0
Number of Comparisons:  361
```

Image 3.1.8 Obs2bubble sort test

Obs3bubble Sort

```
Enter number to choose option: 9

Enter number to element you want in the array: 20
Unsorted Array:  [-197, -1734, 1538, 914, 936, -134, -565, 1512, 736, -967, -1478, -982, -513, 531, -221, -205, -1922, -1259, -331, -635]

Testing Obs3-bubble sort algorithm

Sorted Array:  [-1922, -1734, -1478, -1259, -982, -967, -635, -565, -513, -331, -221, -205, -197, -134, 531, 736, 914, 936, 1512, 1538]

Run time (ms):  0.0
Number of Comparisons:  187
```

Image 3.1.9 Obs3bubble sort test

Sink-down Sort

```
Enter number to choose option: 10

Enter number to element you want in the array: 20
Unsorted Array:  [1271, -249, -872, -781, -299, -1092, 1338, 1271, 223, 736, 993, -762, 1461, -1226, 41, -956, -1388, 594, -845, 835]

Testing Sink-down sort algorithm

Sorted Array:  [1461, 1338, 1271, 1271, 993, 835, 736, 594, 223, 41, -249, -299, -762, -781, -845, -872, -956, -1092, -1226, -1388]

Run time (ms):  0.0
Number of Comparisons:  175
```

Image 3.1.10 Sink-down sort test

Bi-directional Sort

```
Enter number to choose option: 11

Enter number to element you want in the array: 20
Unsorted Array:  [1026, 1444, -1579, 1300, 341, 1609, -346, -1243, -1755, 214, -1695, 200, 900, -211, -813, -1658, 1220, -451, -847, -304]

Testing Bi-directional sort algorithm

Sorted Array:  [-1755, -1695, -1658, -1579, -1243, -847, -813, -451, -346, -304, -211, 200, 214, 341, 900, 1026, 1220, 1300, 1444, 1609]

Run time (ms):  0.0
Number of Comparisons:  169
```

Image 3.1.11 Bi-directional sort test

Question 3.2.1: Testing multiple sorting algorithms.

```
|
Enter number to choose option: 2
Testing Multiple Sorting Algorithms

Enter number to element you want in the array: 20
Unsorted Array:  [578, -739, -152, -410, 1751, 1397, 883, -43, -1237, 1868, -298, 1069, 1610, 1214, 1388, -1000, 1401, -439, 1284, -272]

Sorted Array: (descending order)  [1868, 1751, 1610, 1401, 1397, 1388, 1284, 1214, 1069, 883, 578, -43, -152, -272, -298, -410, -439, -739,
-1000, -1237]

Sorted Array: (ascending order) [-1237, -1000, -739, -439, -410, -298, -272, -152, -43, 578, 883, 1069, 1214, 1284, 1388, 1397, 1401, 1610,
1751, 1868]
-----------------------------------------------------------------------------------------
| Sorting Algorithm Name  | Array Size  | Number of Comparisons  |   Run Time(in ms)    |
-----------------------------------------------------------------------------------------
| Selection Sort          |     20      |          190           |         0.0          |
-----------------------------------------------------------------------------------------
| Insertion Sort          |     20      |          106           |         0.0          |
-----------------------------------------------------------------------------------------
| Merge Sort              |     20      |          67            |         0.0          |
-----------------------------------------------------------------------------------------
| Quick Sort              |     20      |          81            |         0.0          |
-----------------------------------------------------------------------------------------
| Heap Sort               |     20      |          177           | 0.34689903259277344  |
-----------------------------------------------------------------------------------------
| Bubble Sort             |     20      |          380           |         0.0          |
-----------------------------------------------------------------------------------------
| Obs1-Bubble Sort        |     20      |          190           |         0.0          |
-----------------------------------------------------------------------------------------
| Obs2-Bubble Sort        |     20      |          285           |         0.0          |
-----------------------------------------------------------------------------------------
| Obs3-Bubble Sort        |     20      |          180           |         0.0          |
-----------------------------------------------------------------------------------------
| Sink-Down Sort          |     20      |          187           |         0.0          |
-----------------------------------------------------------------------------------------
| Bi-Directional Sort     |     20      |          154           |         0.0          |
-----------------------------------------------------------------------------------------
```

Image 3.2.1 Tests all sorting algorithms using the same array and displays it in table format.

Question 3.2: Conducting an experimental study.

Sorting Algorithm	*n=100*	*n=200*	*n=400*	*n=800*	*n=1000*	*n=2000*
Selection	4950	19900	79800	319600	499500	1999000
Insertion	2556	10236	39773	160925	252104	1003571
Merge	2717	6403	14818	33606	43551	97058
Quick	639	1551	3718	8487	11285	24829
Heap	1532	3665.1	8494	19289	25003	56006
Bubble	9900	3651	159600	639200	999000	3998000
Obs1-Bubble	4950	19900	79800	319600	499500	1999000
Obs2-Bubble	9098	37053	149784	617147	960838	3892652
Obs3-Bubble	4906	19721	79438	319002	498580	1997444
Sink-down	4883	19823	79438	318739	498410	1997403
Bi-directional	3857	15625	60602	245614	381180	1515287

Table 2: Experimental study: Average number of comparisons for sorting arrays of n integers

(Over 10 runs).

Sorting Algorithm	n=100	n=200	n=400	n=800	n=1000	n=2000
Selection	0.200	0.900	3.201	13.702	21.005	82.319
Insertion	0.546	0.600	2.300	10.202	16.904	67.615
Merge	0.100	0.200	0.800	1.200	1.100	3.001
Quick	0.000	0.200	0.300	0.700	1.000	2.701
Heap	0.200	0.200	0.801	1.400	2.000	4.201
Bubble	0.700	2.500	10.227	44.310	69.216	282.910
Obs1-Bubble	0.300	1.901	6.377	26.706	41.928	170.438
Obs2-Bubble	0.700	2.401	10.202	43.710	68.297	281.463
Obs3-Bubble	0.300	1.600	6.201	27.406	43.010	174.525
Sink-down	0.500	1.700	7.002	28.806	45.310	182.275
Bi-directional	0.200	1.500	5.601	24.206	37.508	150.484

Table 3: Experimental study: Average running time (in ms) for sorting arrays of n integers (Over 10 runs).

Conclusion

In this assignment I created the bubble sort algorithm in question. I further improved it according to the observation and created variant bubble sort such as sink down sort and bi-directional bubble sort. In the second question I analysed various sorting algorithms to fill in the table. I used my analysis from question one to fill in the bubble sort rows. For the third question I built a program that allows users to test all the sorting algorithms listed individually or as a group, so that I can compare them. By testing all the sorting algorithms and finding their average comparisons and runtime, I noticed that quick sort is the fastest. In the test with 100 elements in the array the quick sort algorithms was able to sort the array the fastest(program couldn't measure time so its time was 0 ms). It also had the fast time and least amount of comparison through out the tests. The normal bubble sort and the second observation have the longest runtime and comparisons because they don't have strategies that to reduce unnecessary comparisons. The tests done in question 3 further proves that the analysis done on the question 1 and 2 is correct because it is showing results that I was expecting when analysing the algorithms.

References

Programiz. (n.d.). Merge sort algorithm. Retrieved May 7, 2023, from https://www.programiz.com/dsa/merge-sort

Hussein, B. (2016, June 20). Introduction to data structures: 9. Merge sort [Video]. YouTube. https://www.youtube.com/watch?v=RvqmZLuCQrw

Code Studio. (n.d.). Www.codingninjas.com. https://www.codingninjas.com/codestudio/library/time-and-space-complexities-of-sorting-algorithms-explained

Dr Jitian XIAO, CSP2348_M3_Array Searching Algorithms and Analysis, Edit Cowan University, Data Structures - CSP2348.3

Dr Jitian XIAO, CSP2348_M7_Heaps and heap sorting, Edit Cowan University, Data Structures - CSP2348.3