Faculty of Engineering and Technology

Department of Computer Science

Data Structures

COMP2321

Project No. 4: **Sorting Algorithms**

Done By: Munther Anati
#ID: 1182028

Instructor: Dr. Radi Jarrar
Section: 2

Date: 12\Jan\2021

Abstract

This report discusses different types of sorting algorithms that vary in their properties. Different cases were tested to illustrate the difference between the algorithms used (in terms of time), and what makes a certain one more useful and better to use in some specific case.

Contents

1 Introduction

1.1: Sorting Algorithms

Sorting refers to arranging data in a particular format. Sorting algorithm specifies the way to arrange data in a particular order. Most common orders are in numerical or lexicographical order.

The importance of sorting lies in the fact that data searching can be optimized to a very high level, if data is stored in a sorted manner. Sorting is also used to represent data in more readable formats [1].

1.2: Algorithms Complexity

The complexity of an algorithm is a function describing the efficiency of the algorithm in terms of the amount of data the algorithm must process. There are two main complexity measures of the efficiency of an algorithm:

- **Time complexity** is a function describing the amount of time an algorithm takes in terms of the amount of input to the algorithm.
- **Space complexity** is a function describing the amount of memory (space) an algorithm takes in terms of the amount of input to the algorithm [2].

Hence, the efficiency of an algorithm is indicated depending on both time and space.

1.3: Big O Notation

Big O notation, is a symbolism used in complexity theory, computer science, and mathematics to describe the asymptotic behavior of functions. It tells how fast a function grows or declines [3]. This notation is used to indicate the efficiency of sorting algorithms.

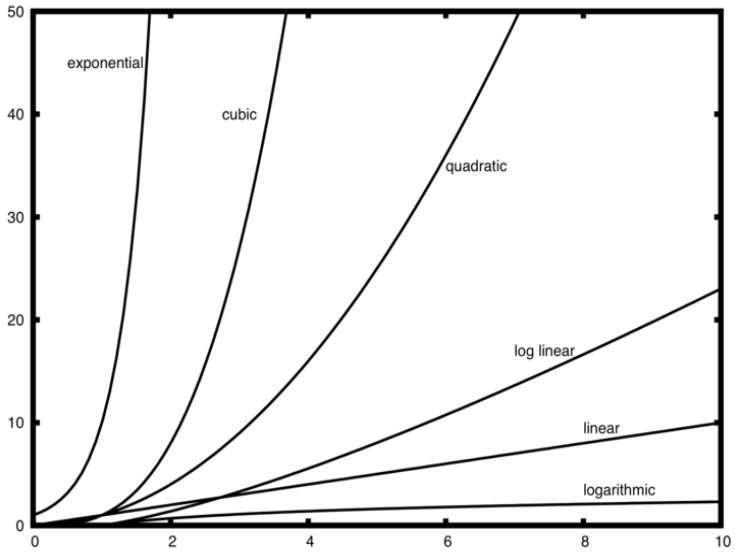

Figure 1: Function's Big O Notation Graph

2 Procedure

To test the given cases (arrays of different sizes and states), the algorithms down below were used, (codes attached in appendix):

2.1: Selection Sort

- An algorithm that maintains two subarrays in a given array, it sorts an array by repeatedly finding the minimum element -from the unsorted subarray- and putting it at the beginning.
- **Time Complexity:** $O(n^2)$ for all cases.
- Motivation of using: Showing the big difference between such an inefficient algorithm and other algorithms that are faster.

2.2: Heap Sort

- This algorithm turns the array of data to be sorted into a heap, deletes the minimum element and inserts it into a sorted array until the heap is empty.
- **Time Complexity:** O(n log n) for all cases.
- Motivation of using: Experiencing a bit- challenging implementation for a quite fast algorithm.

2.3: Merge Sort

- A divide and conquer technique that recursively sort each half of the given array, then it merges the two halves.
- **Time Complexity:** O(n log n) for all cases.
- Motivation of using: Suitable for the very large arrays to be tested since it's a very fast recursive algorithm.

2.4: Pigeonhole Sort

- Also known as "count sort". It's a non-comparative sorting technique that creates holes (an array) and inserts items into each hole depending on the key, then puts the elements back into the original array, sorted, depending on that key.
- **Time Complexity:** O(n) for all cases.
- Motivation of using: a fast algorithm that is suitable for all integer data with a known range.

3 Results

3.1: Sorted Arrays

It's the best case tested, as the arrays given to the sorting algorithms are already sorted. The table down below concludes the results reported for each sorting algorithm sunning time (in seconds), for different array sizes: (the following table was also reported to a file "Generate Sorted.txt")

Array Size	Selection Sort	Heap Sort	Merge Sort	Pigeonhole Sort
1,000	0.040000000	0.000000	0.002000	0.000000
10,000	0.569000000	0.005000	0.009000	0.001000
100,000	17.53200000	0.056000	0.071000	0.004000
1,000,000	1308.773000	0.264000	0.331000	0.038000
5,000,000	29318.37600	1.434000	1.754000	0.085000

3.2: Randomly Sorted Arrays

This is the average case, the data was inserted to the arrays completely randomly, using the built in function rand(), the following table reports the run-time (in seconds) for each algorithm, results were also reported to a file called "Generate Randomly.txt":

Array Size	Selection Sort	Heap Sort	Merge Sort	Pigeonhole Sort
1,000	0.01100000	0.0000000	0.001000	0.0000000
10,000	0.39900000	0.0020000	0.002000	0.0000000
100,000	23.4650000	0.0210000	0.009000	0.0020000
1,000,000	1366.96800	0.2480000	0.013000	0.0110000
5,000,000	32231.8810	1.3450000	0.035000	0.0520000

3.3: Reversed Sorted Arrays

This is the worst tested case since the data of the arrays was reversed in the indices, so it's expected for the running time to be the highest. The table down below illustrates the results in seconds, this data was also reported to a file called "Generate Reversed.txt":

Array Size	Selection Sort	Heap Sort	Merge Sort	Pigeonhole Sort
1,000	0.006000	0.000000	0.001000	0.000000
10,000	0.206000	0.002000	0.002000	0.001000
100,000	16.914000	0.021000	0.024000	0.001000
1,000,000	1845.754000	0.253000	0.261000	0.012000
5,000,000	44713.349000	1.438000	1.406000	0.060000

3.4: Questions

3.4.1: Best algorithm for small size arrays.

3.4.2: Best algorithm for large size arrays.

As we can see in above tables the best algorithm in general is the pigeonhole sort 'Count sort', and it is the best for all cases.

4 Appendices

4.1: Selection Sort Algorithm:

```
void SelectionSort(int arr[], int n) {
    int i, j, temp;
    for (i = 0; i < n - 1; i++) {
        for (j = i + 1; j < n; j++) {
            if (arr[j] <
                arr[i]) {
                temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }

    }
    return;
```

4.2: Heap Sorting Algorithm:

```
void swap(int *a, int *b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

void max_heap(int array[], int size, int i) {
    int max = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < size && array[left] > array[max])
        max = left;

    if (right < size &&
        array[right] > array[max]
        max = right;

    if (max != i) {
        swap(&array[max], &array[i]);
        max_heap(array, size, max);
    }
}

void Heap_sort(int array[], int size) {
    int j = size / 2 - 1;
    for (int i = j; i >= 0; i--)
        max_heap(array, size, i);

    for (int i = size - 1; i >= 0; i--)
        swap(&array[0], &array[i]);
        max_heap(array, i, 0);
    }
}
```

4.3: Merge Sort Algorithm:

```
void merge(int arr[], int left, int mid, int right) {

    int l1 = mid - left + 1;
    int l2 = right - mid;

    static int *leftArray;
    static int *rightArray;
    rightArray = calloc(l2, sizeof(int));
    leftArray = calloc(l1, sizeof(int));


    for (int i = 0; i < l1; i
        leftArray[i] = arr[left + i];
    for (int j = 0; j < l2; j++)
        rightArray[j] = arr[mid + 1 + j];

    int i, j, k;
    i = 0;
    j = 0;
    k = left;
    while (i < l1 && j < l2) {
        if (leftArray[i] <= rightArray[j]) {
            arr[k] = leftArray[i];
            i++;
        } else {
            arr[k] = rightArray[j];
            j++;
        }
        k++;
}
    for (i; i < l1; i++, k++) {
        arr[k] = leftArray[i];
    }
    for (j; j < l2; j++, k++) {
        arr[k] = rightArray[j];
    }
}
void mergeSort(int array[], int left, int right) {
    if (left < right) {

        int mid = left + (right - left) / 2;
        mergeSort(array, left, mid);
        mergeSort(array, mid + 1, right);
        merge(array, left, mid, right);
    }
}
```

4.4: Pigeonhole Sort Algorithm:

```
void pigeonhole_sort(int arr[], int size, int min, int max
{
    if (min == 0 && max == 0) {
        max = find_max(arr, size
        min = find_min(arr, size);
    }

    int length, i, j, k, count;
    length = max - min + 1;
    static int *holes;
    holes = calloc(length, sizeof(int));

    for (i = 0; i < length; i++)
    {
        holes[i] = 0;
    }

    for (j = 0; j < size; j++)
        holes[arr[j] - min]++;
    k = 0;
    for (count = 0; count < length; count++)
        while ((holes[count]--) > 0) {
            arr[k] = count + min;
            k++;
        }
}
```

5 References

[1]: https://www.tutorialspoint.com/data_structures_algorithms/sorting_algorithms.htm
Accessed on 11th/Jan/2021 at 10:45PM

[2]: https://medium.com/@info.gildacademy/time-and-space-complexity-of-data-structure-and-sorting-algorithms-588a57edf495
Accessed on 11th/Jan/2021 at 10:54PM

[3]: http://web.mit.edu/16.070/www/lecture/big_o.pdf
Accessed on 11th/Jan/2021 at 11:10PM

[4]: Sorting Algorithms →

https://www.mygreatlearning.com/blog/heap-sort/.

https://www.mygreatlearning.com/blog/merge-sort/?highlight=shell%20sort

https://programology.wordpress.com/2015/08/28/c-program-for-pigeonhole-sort/

https://www.codingeek.com/algorithms/counting-sort-explanation-pseudocode-and-implementation/