A picture containing text, clipart

Description automatically generated

Algorithms Project

Youssef Mohamed, Mohab Ashraf

211777 – 210439

Faculty of Computer Science

Group A

Dr. Zeinab Abd El Halim

December 29, 2022

TABLE OF CONTENTS

1.0 DESCRIPTION…………………………………..…………………………………………...3

1.1 Chosen Problem………………………………………3

1.2 Selected Algorithm.…………………………………...3

1.3 Designed Algorithm.…………………………………..3

2.0 Implementation Details………………………………………………………………………..3

3.0 Analysis of Algorithms………………………………………………………………………..4

3.1 Empirical ………………………………………………..4 & 6

3.2 Theoretical……………………………………………….5 & 7

4.0 Discussion…………………………………………………………………………………….8

1.0 Description

1.1 Chosen Problem

We Chose the Dutch National Flag problem, It’s a classic sorting problem, proposed by Edsger Dijkstra. It requires ordering a hypothetical group of colors in this case, RGB, which will be represented as 0,1,2 accordingly. The problem makes use of array data structure. Random inputs will be added say [1,0,1,2,2,0] and it should be sorted back to [0,0,1,1,2,2] at the lowest cost in terms of both time and space complexities.

1.2 Selected Predecessor Algorithm

We chose the best and quickest algorithm, Quick Sort. This is by far the best implementation method for solving such problem.

1.3 Designed Algorithm

The algorithm we wrote was the classic Insertion Sort, fast to implement, and produces fast results with small inputs.

2.0 Implementation

Link to video: shorturl.at/agikY (If link is dead, alternative link in txt file)

We decided to use Python as our language, on a Visual Studio Code as the IDE. We also used Lists as our main data structure to represent the problem.

3.0 Analysis of Both Algorithms

Insertion Sort Algorithm

Insertion sort is a simple sorting algorithm that works by iterating through the elements of an array and inserting each element into its correct position in the array. It works in a similar way to how you might sort a hand of playing cards.

Assume the given testcase:

Arr *=* [5, 2, 4, 6, 1, 3]

1. [2, 5, 4, 6, 1, 3]

2. [2, 4, 5, 6, 1, 3]

3. [2, 4, 5, 6, 1, 3]

4. [1, 2, 4, 5, 6, 3]

5. [1, 2, 3, 4, 5, 6]

Sorted\_Arr *=* [1, 2, 3, 4, 5, 6]

At each iteration, the algorithm takes the next element in the input array and inserts it into its correct position in the sorted portion of the array (the left side of the array). It does this by shifting elements to the right as necessary to make room for the new element.

3.1 Empirical Analysis

*def* InsertionSort(*Algo\_List*):

*for* i *in* range(len(Algo\_List)):

        temp *=* Algo\_List[i]

        j *=* i - 1

*while* j *>=* 0 *and* temp *<* Algo\_List[j]:

            Algo\_List[j + 1] *=* Algo\_List[j]

            j *-=* 1

            Algo\_List[j + 1] *=* temp

    print("Insertion Sorted: ",Algo\_List)

Best case: O(N) at testcase example Arr = [0,1,2,3 - 50]

Execution Time: 0.000382 ms (Avg. of 10 runs)

Average case: O(N2) at testcase example Arr = [0,2,1,3, RAND,50]

Execution Time: 0.000443 ms (Avg. of 10 runs)

Worst Case: O(N2) at testcase example Arr = [50 ,49, 48, 47 … 0]

Execution Time: 0.000482 ms (Avg. of 10 runs)

3.2 Theoretical Analysis

Time Complexity:

In the best case, the input array is already sorted, so the algorithm only needs to make a single pass through the array to verify that it is already sorted. This gives it a time complexity of O(n).

In the average case, the input array is randomly shuffled, so the algorithm must make multiple passes through the array to sort it. The time complexity in this case is O(n^2), since the algorithm must make n-1 passes through the array (one for each element) and do O(n) work on each pass.

In the worst case, the input array is sorted in the reverse order, so the algorithm must make n-1 passes through the array and do O(n) work on each pass. This gives it a time complexity of O(n^2).

Space Complexity:

The space complexity of the insertion sort algorithm is O(1), meaning it requires only a constant amount of additional memory beyond the input array itself. This is because the algorithm sorts the input array in place, meaning it does not create any additional arrays or data structures to store the sorted elements.

Quick Sort Algorithm

Quicksort is a divide and conquer sorting algorithm that works by partitioning an array into two smaller subarrays, sorting the subarrays, and then merging them back together. It is one of the most widely used sorting algorithms and is known for its efficiency and simplicity.

Assume the given testcase:

Arr *=* [5, 2, 4, 6, 1, 3]

1. Select pivot: 5

2. Partition array: [2, 4, 1, 3], [6]

3. Recursively sort left subarray: [1, 2, 3, 4]

4. Recursively sort right subarray: [6]

Sorted\_Array *=* [1, 2, 3, 4, 5, 6]

The algorithm chooses a pivot element from the input array and partitions the array into two subarrays, a subarray of elements less than the pivot, and a subarray of elements greater than or equal to the pivot. Then we Recursively sort the subarrays and merge the sorted subarrays back together.

3.1 Empirical Analysis

*def* QuickSort(*Algo\_List*):

*def* partition(*Algo\_List*, *low*, *high*):

        pivot *=* Algo\_List[high]

        i *=* low - 1

*for* j *in* range(low, high):

*if* Algo\_List[j] *<=* pivot:

                i *=* i + 1

                (Algo\_List[i], Algo\_List[j]) *=* (Algo\_List[j], Algo\_List[i])

        (Algo\_List[i + 1], Algo\_List[high]) *=* (Algo\_List[high], Algo\_List[i + 1])

*return* i + 1

*def* Sort\_Part(*Algo\_List*, *low*, *high*):

*if* low *<* high:

            pi *=* partition(Algo\_List, low, high)

            Sort\_Part(Algo\_List, low, pi - 1)

            Sort\_Part(Algo\_List, pi + 1, high)

    size *=* len(Algo\_List)

    Sort\_Part(Algo\_List, 0, size - 1)

    print("Quick-Sorted List: ", Algo\_List)

Best case: O(N \* log N) at testcase example Arr = [0,1,2,3 - 50]

Execution Time: 0.000322 ms (Avg. of 10 runs)

Average case: O(N \* log N) at testcase example Arr = [0,2,1,3, RAND,50]

Execution Time: 0.000343 ms (Avg. of 10 runs)

Worst Case: O(N2) at testcase example Arr = [50 ,49, 48, 47 … 0]

Execution Time: 0.000396 ms (Avg. of 10 runs)

3.2 Theoretical Analysis

Time Complexity:

In the best case, the input array is already sorted, so the algorithm only needs to make a single pass through the array to verify that it is already sorted. This gives it a time complexity of O(n \* log(n)).

In the average case, the input array is randomly shuffled, so the algorithm must make multiple passes through the array to sort it. The time complexity in this case is O(n \* log(n)), since the algorithm divides the input array into two smaller subarrays on each pass and the size of the input decreases by a factor of 2 on each pass.

In the worst case, the input array is sorted in the reverse order, so the algorithm must make n-1 passes through the array and do O(n) work on each pass. This gives it a time complexity of O(n^2).

Space Complexity:

The space complexity of the quicksort algorithm is O(log(n)) in the average case and O(n) in the worst case. This is because the algorithm uses recursive function calls to sort the input array, and each recursive call requires a small amount of memory to store the function's local variables and return address.

The average case space complexity of O(log(n)) occurs when the pivot element is chosen in such a way that the input array is evenly divided into two subarrays on each recursive call. In this case, the space complexity is determined by the maximum depth of the recursion, which is log(n) for a fully sorted array.

The worst-case space complexity of O(n) occurs when the pivot element is chosen in such a way that one subarray is empty and the other is the same size as the original array. In this case, the space complexity is determined by the number of recursive calls, which is equal to the size of the input array.

Overall, the quicksort algorithm is considered to decently low space complexity, making it a good choice for sorting large arrays in memory-constrained environments.

4.0 Discussion

In terms of time complexity, quicksort is generally considered to be more efficient than insertion sort. Quicksort has a time complexity of O(n \* log(n)) in the average case and O(n^2) in the worst case, while insertion sort has a time complexity of O(n^2) in the worst case and O(n) in the best case. This means that quicksort is generally faster than insertion sort, especially for large inputs.

In terms of time complexity, Insertion sort has a constant O(1) across a space complexity cases since it sorts the input array in place and does not create any additional arrays or data structures. Quicksort on the other hand has a space complexity of O(log(n)) in the average case and O(n) in the worst case, due to the use of recursive function calls. Overall, both quicksort and insertion sort are efficient sorting algorithms with low space complexity. However, insertion sort may be a better choice if you are working with a small input or need a simple and easy-to-implement sorting algorithm, while quicksort may be a better choice if you are working with a larger input or need a more efficient sorting algorithm.

Overall, quicksort is generally considered to be a more efficient sorting algorithm than insertion sort, but only if you favor speed over storage when dealing with large data. However, insertion sort may be a good choice in certain situations, such as when the input array is small or nearly sorted, or when you need a simple and easy-to-implement sorting algorithm.