VIETNAM NATIONAL UNIVERSITY
HO CHI MINH CITY

UNIVERSITY OF SCIENCE

FACULTY OF INFORMATION TECHNOLOGY

Project Report

Topic: SORTING ALGORITHMS

Subject: Data Structures and Algorithms

Students:
Phan Huynh Minh Khoi
CLC07 - 23127072

Lecturers:
LT. Bui Huy Thong
LT. Nguyen Ngoc Thao

Friday 5^{th} July, 2024

Contents

List of Algorithms

Full Name	Student ID	Email
Le Thien Phu	23127xxx	email
Vo Tran Quoc Duy	23127xxx	email
Nguyen Huy Quan	23127xxx	email
Phan Huynh Minh Khoi	23127072	phmkhoi23@clc.fitus.edu.vn

1 Algorithms Presentation

1.1 Quick Sort

1.1.1 Ideas

Quick Sort is a sorting algorithm based on the Divide and Conquer algorithm. It picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.

1.1.2 Descriptions

Using these following steps to perform the Quick Sort:

- Step 1: Choosing the pivot of the array. To choose the pivot, this code below uses the median-with-three method. Specifically, consider three elements (the first, middle, and last elements of the array), then determine the median value among those three elements to become a pivot. After that, swap the pivot element with the first element of the array.
- Step 2: Sequentially comparing each element in the array with the pivot value and separating the remaining array into two parts - smaller and greater than or equal to the pivot value. Then, swap the pivot element with the last element in a smaller part.
- Step 3: Partitioning the array into two smaller sub-arrays - before and after the pivot, then go back to Step 1.

Algorithm 1 Quick Sort

```
function SWAP(a, b)
    temp ← a
    a ← b
    b ← temp
end function

function PARTITION(array, low, high)
    mid ← (low + high)/2
    median ← array[low]+array[high]+array[mid]−max(array[low], array[high], array[mid])
    −min(array[low], array[high], array[mid])

    pos ← low
    if median = a[high] then
        pos ← high
    end if
    if median = a[mid] then
        pos ← mid
    end if

    pivot ← low
    last_S1 ← low
    first_unknown ← low + 1

```

```
    while first_unknown <= high do
        if array[first_unknown] < array[pivot] then
            Swap(array[last_S1 + 1], array[first_unknown])
            last_S1 ← last_S1 + 1
        end if
        first_unknown ← first_unknown + 1
    end while

    Swap(array[pivot], array[last_S1])
    return last_S1 //(the pivot position)
end function

function quickSortRecursion(array, low, high)
    if low < high then
        pivot ← Partition(array, low, high)

        if pivot > low + 1 then
            quickSortRecursion(array, low, pivot − 1)
        end if

        if pivot < high − 1 then
            quickSortRecursion(array, pivot + 1, high)
        end if
    end if
end function

function quickSort(array, n)
    quickSortRecursion(array, 0, n − 1)
end function
```

1.1.3 Time Complexity

The problem size is: n - the number of elements of the array. In the Partitioning Step, this sort chooses the pivot, and then compares each remaining element in the array with the pivot. So the time complexity for this step is $O(n)$. After that, the array will be divided into two sub-arrays. These steps will be repeated until the array is sorted. In short, the time complexity of Quick Sort is $O(nlogn)$ for Average and Best case. In the Worst case, specifically when the array is divided into two parts, one part consisting of N-1 elements and the other and so on, the time complexity is $O(n^2)$. **Time complexity:**

- **Best case:** $O(nlogn)$
- **Average case:** $O(nlogn)$
- **Worst case:** $O(n^2)$

1.1.4 Space Complexity

Since Quick Sort is an in-place sorting algorithm, it does not require additional storage.
Space complexity: $O(n)$

1.2 Shaker Sort

1.2.1 Ideas

Shaker Sort (or Cocktail Sort) is a variation of Bubble Sort. Shaker Sort traverses through a given array in both directions alternatively. Shaker sort does not go through the unnecessary iteration making it efficient for large arrays.

1.2.2 Descriptions

Using these following steps to perform the Shaker Sort:

- Step 1: Sort the array from left to right by using Bubble Sort.
- Step 2: Sort the array in the opposite direction from the element just before the most recently sorted element by using Bubble Sort.

Repeat Step 2 and Step 3 until the array is sorted.

Algorithm 2 Shaker Sort

```
function SWAP(a, b)
    temp ← a
    a ← b
    b ← temp
end function
function SHAKERSORT(array, n)
    swapped ← true
    start ← 0, end ← n − 1
    while swapped == true do
        swapped ← false
        for i ← start to end − 1 do
            if array[i] > array[i + 1] then
                Swap(array[i], array[i + 1])
                Swapped ← true
            end if
        end for
        if swapped = true then
            break
        end if
        swapped ← false
        end ← end − 1
        for i ← end − 1 to start do
            if array[i] > array[i + 1] then
                Swap(array[i], array[i + 1])
                Swapped ← true
```

end if
end for

$start \leftarrow start + 1$
end while
end function

1.2.3 Time Complexity

The problem size is: n - the number of elements of the array.
The algorithm iterates through the array multiple times (in both directions) in Average and Worst case, so the time complexity in those case are $O(n^2)$. In the Best case, when the original array is already sorted, the time complexity is $O(n)$.
Time complexity:

- **Best case:** $O(n)$
- **Average case:** $O(n^2)$
- **Worst case:** $O(n^2)$

1.2.4 Space Complexity

Since Shaker Sort is an in-place sorting algorithm, it does not require additional storage.
Space complexity: $O(n)$

1.3 Shell Sort

1.3.1 Ideas

Shell Sort is mainly a variation of Insertion Sort. The method starts by sorting pairs of elements far apart, then progressively reducing the gap between elements to be compared. Starting with far-apart elements can move some out-of-place elements into the position faster than a simple nearest-neighbor exchange.

1.3.2 Descriptions

Using these following steps to perform the Shell Sort:

- Step 1: Initialize the value of the gap size, say h.
- Step 2: Divide the list into smaller sub-parts. Each must have equal intervals to h.
- Step 3: Sort these sub-lists using insertion sort.
- Step 4: Reducing the value of h, then go back to step 2 until h is equal to 1.

Algorithm 3 Shell Sort

function SHELLSORT($array$, n)
  $gap \leftarrow n/2$
  while $gap > 0$ **do**
    for $i \leftarrow gap$ **to** n **do**
      $temp \leftarrow array[i]$

$j \leftarrow i$
while $j >= gap$ && $array[j - gap] > temp$ **do**
$array[j] \leftarrow array[j - gap]$
$j \leftarrow j - gap$
end while
$array[j] \leftarrow temp$
end for
$gap \leftarrow gap/2$
end while
end function

1.3.3 Time Complexity

The problem size is: n - the number of elements of the array.
In the above implementation, the gap is reduced by half in every iteration. So, the time complexity of the above implementation of Shell sort is $O(nlogn)$. In the Worst case, consider the array where the odd and even elements are not compared until we reach the last increment of 1. So, the time complexity of this case if $O(n^2)$.
0000000 **Time complexity:**

- **Best case:** $O(nlog(n))$
- **Average case:** $O(nlog(n))$
- **Worst case:** $O(n^2)$

1.3.4 Space Complexity

Since the Shell Sort is an in-place sorting algorithm, it does not require additional storage.
Space complexity: $O(n)$

2 Project Organization

In this project, we have 16 files in total: 1 executable file, 5 .cpp files, 4 header files, and 6 txt files.

- **Executable file:** the main file to execute the request from the input commands.
- **"DataGenerator" files:** generating the data of the arrays into 4 types (sorted, nearly sorted, reversed, and randomized).
- **"Checker" files:** some converting and checking functions to support receiving the input and executing the data.
- **"Mode" files:** functions to read the commands and return the desired results.
- **"Sort" files:** all the sorts are implemented in this files. Each sort is installed in two types: count the comparisons and execute the running time.
- **txt files:** to store the data that has been generated (input) and the data after sorting (output).
- **"Main" file:** to read the commands and classify them into installed functions.