TECHNICAL UNIVERSITY OF MOLDOVA

FACULTY OF COMPUTERS, INFORMATICS AND MICROELECTRONICS

DEPARTMENT: **SOFTWARE ENGINEERING AND AUTOMATION**

CC - LABORATORY WORK NR 2

Sorting Algorithms

Ciuş Iurie FAF-203

Chişinău 2022

Contents

1 Algorithm Analysis

Algorithm analysis is an important part of computational complexity theory, which provides theoretical estimation for the required resources of an algorithm to solve a specific computational problem. Analysis of algorithms is the determination of the amount of time and space resources required to execute it.

1.1 Introduction

The current paper work represents my implementation of four sorting algorithms.

A Sorting Algorithm is used to rearrange a given array or list elements according to a comparison operator on the elements. The comparison operator is used to decide the new order of element in the respective data structure.

From the beginning of computing, the sorting problem has attracted a great deal of research, perhaps due to the complexity of solving it efficiently despite its simple, familiar statement. Among the authors of early sorting algorithms around 1951 was Betty Holberton, who worked on **ENIAC** and **UNIVAC**. Bubble sort was analyzed as early as 1956. Asymptotically optimal algorithms have been known since the mid-20th century - new algorithms are still being invented, with the widely used Timsort dating to 2002, and the library sort being first published in 2006.

Comparison sorting algorithms have a fundamental requirement of $\Omega(nlogn)$ comparisons (some input sequences will require a multiple of n log n comparisons, where n is the number of elements in the array to be sorted). Algorithms not based on comparisons, such as counting sort, can have better performance.

Sorting algorithms are prevalent in introductory computer science classes, where the abundance of algorithms for the problem provides a gentle introduction to a variety of core algorithm concepts, such as big O notation, divide and conquer algorithms, data structures such as heaps and binary trees, randomized algorithms, best, worst and average case analysis, time-space tradeoffs, and upper and lower bounds.

Sorting small arrays optimally (in fewest comparisons and swaps) or fast (i.e. taking into account machine specific details) is still an open research problem, with solutions only known for very small arrays (¡20 elements). Similarly optimal (by various definitions) sorting on a parallel machine is an open research topic.

1.2 Objectives

- Study and empirical analysis of sorting algorithms. QuickSort, mergeSort, heapSort analysis, (one of your choice).
- Establish the properties of the input data in relation to which the analysis is made.
- Choose the metric for comparing algorithms.
- Perform empirical analysis of the proposed algorithms.
- Make a conclusion on the work done.

1.3 Theoretical Notes

An algorithm is correct if for any instance of it ends with a correct output which is the solution to the calculation problem solved by the respective algorithm. Some algorithms, however, are incorrect In the sense that they do not lead to any finite time solution or lead to Wrong solutions, they can be useful as long as the errors they produce can be controlled.

The behavior of the same algorithm may be different depending on input data. That is why great attention must be paid to them the latter. A variable present in the input of an algorithm can identify an array or object of a composite data and it will be treated as a pointer to the elements of the painting, respectively to the fields (attributes) of the corresponding object.

How to define the size of the input data depends on the calculated calculation problem. It can be expressed by:

- the number of items contained in the input data (for example, the size of an array of integers);
- the total number of bits in the binary representation of the data entry;
- two natural numbers

For each algorithm that solves a certain problem P it is necessary to specify how the input data size is expressed.

If there is an algorithm that solves the problem P it doesn't mean he is unique.

For example, there are algorithms like: **QuickSort**, **MergeSort**, **TreeSort**, sorting by selection and insertion etc. which are used for the same purpose.

Therefore, there is a need to choose an algorithm from the class of algorithms that solve the problem P corresponding to some requirements. The algorithm depends on the application, implementation, environment, frequency of use etc. Comparing algorithms is a process subtle that has several aspects in mind. Next, we'll look at some ways to implement a few of the most popular sorting algorithms.

1.4 Algorithms

1.4.1 QuickSort

QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

1. Always pick first element as pivot.
2. Always pick last element as pivot.
3. Pick a random element as pivot.
4. Pick median as pivot (implemented below).

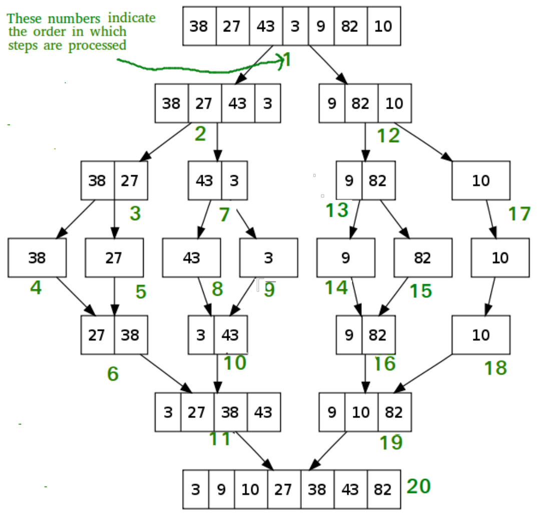
```
/* low --> Starting index, high --> Ending index */
quickSort(arr[], low, high)
{
   if (low < high)
   {
      /* pi is partitioning index, arr[pi] is now
        at right place */
      pi = partition(arr, low, high);

      quickSort(arr, low, pi - 1); // Before pi
      quickSort(arr, pi + 1, high); // After pi
   }
}
```

1.4.2 Merge Sort

Like QuickSort, Merge Sort is a Divide and Conquer algorithm. It divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves. The merge() function is used for merging two halves. The merge(arr, l, m, r) is a key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one.

```
MergeSort(arr[], l, r)
If r > l
   1. Find the middle point to divide the array into two halves:
         middle m = l+ (r-l)/2
   2. Call mergeSort for first half:
         Call mergeSort(arr, l, m)
   3. Call mergeSort for second half:
         Call mergeSort(arr, m+1, r)
   4. Merge the two halves sorted in step 2 and 3:
         Call merge(arr, l, m, r)
```

The following diagram from wikipedia shows the complete merge sort process for an example array 38, 27, 43, 3, 9, 82, 10. If we take a closer look at the diagram, we can see that the array is recursively divided into two halves till the size becomes 1. Once the size becomes 1, the merge processes come into action and start merging arrays back till the complete array is merged.



1.4.3 Heap Sort

Heap sort is a comparison-based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the minimum element and place the minimum element at the beginning. We repeat the same process for the remaining elements.

What is Binary Heap?

A **binary heap** is a heap, i.e, a tree which obeys the property that the root of any tree is greater than or equal to (or smaller than or equal to) all its children (heap property). The primary use of such a data structure is to implement a priority queue.

How to "heapify" a tree?

The process of reshaping a binary tree into a Heap data structure is known as **heapify**. A binary tree is a tree data structure that has two child nodes at max. If a node's children nodes are **heapified**, then only **heapify** process can be applied over that node. A heap should always be a complete binary tree.

Starting from a complete binary tree, we can modify it to become a Max-Heap by running a function called **heapify** on all the non-leaf elements of the heap. i.e. **heapify** uses recursion.

```
heapify(array)
Root = array[0]
Largest = largest( array[0] , array [2 * 0 + 1]. array[2 * 0 + 2])
if(Root != Largest)
   Swap(Root, Largest)
```

1.4.4 Cocktail Sort

Cocktail Sort is a variation of Bubble sort. Bubble sort algorithm always traverses elements from left and moves the largest element to its correct position in first iteration and second largest in second iteration and so on. Cocktail Sort traverses through a given array in both directions alternatively.

Each iteration of the algorithm is broken up into 2 stages:

1. The first stage loops through the array from left to right, just like the Bubble Sort. During the loop, adjacent items are compared and if value on the left is greater than the value on the right, then values are swapped. At the end of first iteration, largest number will reside at the end of the array.

2. The second stage loops through the array in opposite direction- starting from the item just before the most recently sorted item, and moving back to the start of the array. Here also, adjacent items are compared and are swapped if required.

2 Code

2.1 Implementation

```python

class quicksort():
    def __init__(self) -> None:
        pass

    def sort(self, array: list = None, start: "int >= 0" = 0, end:
        "int > 1" = 2) -> list:
        '''
        :param array: list
            The given array that should be sorted.
        :param start: int > 0
            Starting index.
        :param end: int > 0
            Ending index.
        '''

        low, high = start, end
        pivot = array[int((low + high) / 2)]

        while (low <= high):

            while array[low] < pivot:
                low += 1

            while array[high] > pivot:
                high -= 1

            if (low <= high):
                array[low], array[high] = array[high], array[low]
                low += 1
                high -= 1

        if (high > start):
            self.sort(array, start, high)

        if (low < end):
            self.sort(array, low, end)


class mergesort():
    def __init__(self) -> None:
        pass

    def sort(self, array: list = None) -> list:
        '''
        :param array: list
            The given array that should be sorted.
        '''

        if len(array) > 1:
            middle = len(array) // 2
            left, right = array[:middle], array[middle:]

            self.sort(left)
            self.sort(right)

            i = j = k = 0

            while i < len(left) and j < len(right):
                if left[i] < right[j]:
                    array[k], i = left[i], i + 1
                else:
                    array[k], j = right[j], j + 1
                k = k + 1

            while i < len(left):
                array[k] = left[i]
                i, k = i + 1, k + 1

            while j < len(right):
                array[k] = right[j]
                j, k = j + 1, k + 1


class heapsort():
    def __init__(self) -> None:
        pass

    def __heapify(self, arr, size, idx):
        # Initialize largest as root at index i, left = 2 * i + 1, right = 2 * i + 2.
        largest, left, right = idx, 2 * idx + 1, 2 * idx + 2

        if left < size and arr[largest] < arr[left]:
            largest = left

        if right < size and arr[largest] < arr[right]:
            largest = right

        if largest != idx:
            arr[idx], arr[largest] = arr[largest], arr[idx]

            self.__heapify(arr, size, largest)

    def sort(self, array: list = None) -> list:
        '''
        :param array: list
            The given array that should be sorted.
        '''

        length = len(array)

        # Build a maxheap.
        for i in range(length // 2 - 1, -1, -1):
            self.__heapify(array, length, i)

        # One by one extract elements
        for i in range(length - 1, 0, -1):
            # Perform the swapping
            array[i], array[0] = array[0], array[i]
            self.__heapify(array, i, 0)


class cocktailsort():
    def __init__(self) -> None:
        pass

    def sort(self, array: list = None) -> list:
        '''
        :param array: list
            The given array that should be sorted.
        '''

        length = len(array)
        swapped = True
        start, end = 0, length - 1

        while swapped == True:
```

```
      # reset the swapped flag on entering the loop,
      # because it might be true from a previous
      # iteration.
      swapped = False

      # loop from left to right same as the bubble
      # sort
      for i in range(start, end):
         if array[i] > array[i + 1]:
            array[i], array[i + 1] = array[i + 1], array[i]
            swapped = True

      # if nothing moved, then array is sorted.
      if (swapped == False):
         break

      # otherwise, reset the swapped flag so that it
      # can be used in the next stage
      swapped = False
      end -= 1

      # from right to left, doing the same
      # comparison as in the previous stage
      for i in range(end - 1, start - 1, -1):
         if array[i] > array[i + 1]:
            array[i], array[i + 1] = array[i + 1], array[i]
            swapped = True

      # increase the starting point, because
      # the last stage would have moved the next
      # smallest number to its rightful spot.
      start += 1
```

2.2 Time Complexity

Quick Sort

- **Worst Case**: The worst case occurs when the partition process always picks greatest or smallest element as pivot. If we consider above partition strategy where last element is always picked as pivot, the worst case would occur when the array is already sorted in increasing or decreasing order. $\Omega(n^2)$.
- **Best Case**: The best case occurs when the partition process always picks the middle element as pivot. Following is recurrence for best case: $O(n \cdot log(n))$

Merge Sort

Time Complexity: Sorting arrays on different machines. Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.

$$T(n) = 2T\frac{n}{2} + O(n)$$

The above recurrence can be solved either using the Recurrence Tree method or the Master method. It falls in case II of Master Method and the solution of the recurrence is $O(n \cdot log(n))$. Time complexity of Merge Sort is $O(n \cdot log(n))$ in all 3 cases (worst, average and best) as merge sort always divides the array into two halves and takes linear time to merge two halves.

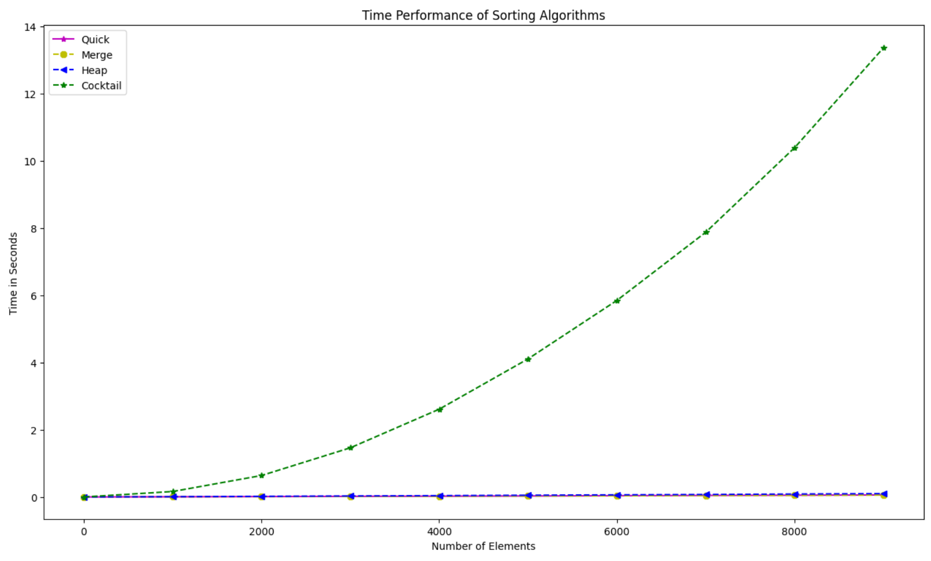
Heap Sort

Time complexity of heapify is $O(log(n))$. Time complexity of createAndBuildHeap() is O(n) and the overall time complexity of Heap Sort is $O(n \cdot log(n))$.

Cocktail Sort

- **Worst and Average Case Time Complexity:** $O(n^2)$.
- **Best Case Time Complexity:** $O(n)$. Best case occurs when array is already sorted.

2.3 Graphs



3 Conclusion

During this laboratory work, I studied and analyzed four of the most popular sorting algorithms, each with it's upsides and downsides. Efficient sorting is important for optimizing the efficiency of other algorithms (such as search and merge algorithms) that require input data to be in sorted lists. Sorting is also often useful for canonicalizing data and for producing human-readable output.

I have tested all the algorithm on 10 arrays within a range of 0 to 10000 elements, randomly generated from 0 to 200. Based on the results and the computing time, I can say that the **Cocktail Sort** performed the worst with a time complexity of *nlogn*. Meanwhile the rest of them performed too good, almost close to 0. Which is why I am unable to tell which is the fastest. But in my opinion, the **Merge Sort** was easier to implement.

3.1 References

- https://github.com/IuraCPersonal/FAF203-CC