UMM AL-QURA UNIVERSITY

College of Computer and Information Systems

Computer Sciences Department

جامعة أم القرى
UMM AL-QURA UNIVERSITY

Algorithms Project

14012402-4 Algorithms

Median finding

By

Hana Shaikh Ali Haji
ID: 441017896

Areej Saud Al Qurashi
ID: 442001962

Maram Hussein Alfham
ID: 442008028

Raghad Omar Bazaraah
ID: 442002867

Maram Abdullah Khoaj
ID:442009326

Supervisor(s):

Dr. Areej Othman Alsini

UMM AL-QURA UNIVERSITY - February 2023

Contents

Introduction

This project is about finding the median using sorting algorithms, an algorithm like QuickSort, SelectionSort, and Midenfinding. Comparisons are made between these algorithms based on the time it takes to execute each algorithm. And a comparison of the data structures used for each algorithm.

The NumPy library was used to read, store and work with this large amount of data.

The main goal of the project is to compare the Sorting algorithms in finding the median.

Group work report

Name	Part
Hana Shaikh Ali Haji	Quick sort algorithm code - report - readme file
Areej Saud Al Qurashi	Selection Sort algorithm code - Comparison of algorithms
Maram Hussein Alfham	Selection Sort algorithm code - Analysis
Raghad Omar Bazaraah	Median finding algorithm code - Pseudocode
Maram Abdullah Khoaj	Median finding algorithm code - Time complexity

User computer specifications and data

- **computer specifications**
  - **Name**: Lenovo 81wb
  - **Processor:** Intel Core i7-10510U (10th Gen)
  - **RAM:** 16GB
  - **Disk:** 512 GB PCIe NVMe M.2 SSD

- **Data**

  The dataset was provided by the Mexican government (Found on Kaggle). This dataset contains an enormous number of anonymized patient-related information including pre-conditions. The raw dataset consists of 21 unique features and 1,048,576 unique patients.
  The dataset used in this project contains only the ages of 99999 patients.

Data structure

In our project, we chose a different data structure (Array, heap, linked list, B tree) for the following algorithms: Median finding, Selection Sort, and quick sort.

Why did we choose these data structures?

the most important thing to consider when choosing The appropriate data structure for the algorithm is the time complexity and the way the data works

- **Array**

the ease to deal with it and to implement it in the algorithm, as it takes O(n), and this is in its worst case, and also its size is limited, and it only stores data of the same type

- **Heap**

Heap sort is one of the fastest sorting algorithms with time complexity of O (n log(n)), and it's easy to implement too.

- **Tree B**

B-Tree facilitates ordered sequential access, thus it works efficiently compared to hash table, a tree can have millions of items stored in it, and with its flat structure, it facilitates easy and efficient traversal of data, for the time taken by this data is O(log n) Good and fast time

- **Linked List**

No memory wastage: In the linked list, efficient memory utilization can be achieved because the size of the linked list increases or decreases at runtime so that there is no memory waste and there is no need to allocate memory in advance

Implementation: Linear data structures such as stacks and queues are often easily implemented using a linked list. The time it takes to execute in the worst case is O(n).

Selection Sort algorithm

- **Discerption /Definition**
  Selection Sort to sorts the data, The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.

- **Time complexity**
  The time complexity of Selection Sort is $O(N^2)$ as there are two nested loops:
  - One loop to select an element of Array one by one = O(N)
  - Another loop to compare that element with every other Array element = O(N)

- **Pseudocode**

```
RSelect( Array[], p,r, i)

if p == r
     return A[p]
q = RandomPartition(Array[],p,r)
k = q - p + 1
if i == k // case that the pivot is the answer
return Array[q]

else if i<k
    return RSelect(Array,p, q-1,i)
else
    return RSelect(Array, q+1, r, i-k)
```

- **Code**

```
def selectionSort(array, size):


   for ind in range(size):
        min_index = ind
        for j in range(ind + 1, size):
            # select the minimum element in every iteration
            if array[j] < array[min_index]:
                min_index = j
         # swapping the elements to sort the array

        (array[ind], array[min_index]) = (array[min_index], array[ind])
```

Quick sort algorithm

- **Discerption /Definition**
  Quicksort is a fast sorting algorithm that takes a divide-and-conquer approach to sorting lists. While sorting is a simple concept, it is a basic principle used in complex programs such as file search, data compression, and pathfinding. Running time is an important thing to consider when selecting a sorting algorithm since efficiency is often thought of in terms of speed. Quicksort has a very slow worst-case running time, but a fast average and best-case running time

- **Time Complexity:**
  Dividing the list into parts less than and greater than the pivot takes O(n) time because the algorithm needs to scan through the list, which has O(n) elements. During this step, for each element, the algorithm performs a constant number of comparisons. It determines if the element is greater than or less than the pivot.
- **Pseudocode**

```
quicksort (array){
  if (array.length > 1){
    choose a pivot;
    while (there are items left in array){
      if (item < pivot)
        put item into subarray1;
      else
        put item into subarray2;
    }
    quicksort(subarray1);
    quicksort(subarray2);
  }
}
```

- **Code**

```
def partition(arr, low, high):
    i = (low-1)         # index of smaller element
    pivot = arr[high]   # pivot
    for j in range(low, high):
        if arr[j] <= pivot:
            i = i+1
            arr[i], arr[j] = arr[j], arr[i]
    arr[i+1], arr[high] = arr[high], arr[i+1]
    return (i+1)

def quickSort(arr, low, high):
    if len(arr) == 1:
        return arr
    if low < high:
        pi = partition(arr, low, high)
        quickSort(arr, low, pi-1)
        quickSort(arr, pi+1, high)
```

Median finding algorithm

Definition and description

Median-finding algorithms (also called linear-time selection algorithms) use a divide and conquer strategy to efficiently compute the I *thi*th smallest number in an unsorted list of size *n*, where I *i*is an integer between *1*1 and I *n*. Selection algorithms are often used as part of other algorithms; for example, they are used to help select a pivot in quicksort and also used to determine *thi*th-order statistics such as the maximum, minimum, and median elements in a list.

Time complexity

This algorithm runs in O(n) linear time complexity, we traverse the list once to find medians in sub lists and another time to find the true median to be used as a pivot.

Pseudocode

```
medianOfMedians(arr[1...n])
    if(n < 5 return Select(arr[1...n], n/2))
    Let M be an empty list
    For i from 0 to n/5 - 1:
        Let m = Select(arr[5i + 1...5i+5], 3)
        Add m to M
    Return QuickSelect(M[1...n/5], n/10)
End medianOfMedians
```

Code

```
def findMedianArray(a, n):
    if n % 2 != 0:
        return float(a[int(n / 2)])
    return float((a[int((n - 1) / 2)] + a[int(n / 2)]) / 2.0)
```

Analysis

- **Comparison of algorithms**

Median finding algorithm	
Median finding for Selection Sort algorithm for Array	2877.67923
	2809.8792
	2778.8226
	Average =2812.3872
Median finding QuickSort for BST	2973.421875
	2873.35688
	2987.33567
	Average =2944.7048
Median finding QuickSort for LinkedList	2976.046875
	2966.045688
	2987.567777
	Average = 2955.5689
Selection Sort algorithm	
Selection Sort algorithm for Array	2976.671875
	2989. 187566
	2901.08187
	Average =2847.933144
Selection Sort algorithm for heap	13.4375
	15.796875
	18.203124
	Average =15.8123998
Selection Sort algorithm for LinkedList	58.5679
	59.8979
	59.9822
	Average =59.8923
Quick sort algorithm	
QuickSort for BST	2641.84375
	2551.85367
	2556.94556
	Average =2528.8792
QuickSort for LinkedList.	57.15635
	59,8765
	55.67943
	Average =56.66777
QuickSort for Array	2495.390625
	2595.390625
	2599.490625
	Average =254

- **Algorithm analysis**

In this project, we relied on two sorting algorithms: the Quicksort and the Selection Sort algorithms.

The Selection Sort is considered as one of the worst sorting algorithms, because of its time complexity in all cases “terms of input size”= O(n^2). While Quicksort is better in terms of Time complexity = O(n)

And based on the size of the data that was used in this project “one hundred thousand.”

the Selection Sort will clearly have a longer runtime compared to the Quicksort in general But with more than one implementation of these algorithms and more than one data structure, there are certainly some differences that will be compared and determined whether their impact is strong on improving the performance of the algorithms or not

Three data Structures were used for the Quicksort, which are: Array, Linked List, and Binary Search tree. As for the selection Sort, was used: Array, Linked List and Heap

Let's take a quick look at the data structure used, and which one considers the best regarding To searching and accessing unordered elements.

The binary search tree is considered the best in terms of the search process among these data structures and takes Time complexity = O(log n). Heap is the second-best data structure in terms of searching. Time complexity will take = O (log n). While Array is also good in terms of searching and accessing elements, it takes Time complexity = O(n). Linked list is also good and takes Time complexity equal to Array in the searching = O(n).

So, since the sorting process is a process that needs to search for the minimum elements in order to arrange them ascendingly, this data Structure will be the best for Sorting.

Now we will get deep by Discuss the results that appeared during the runtime.

Through the runtime table below, we note that the worst runtime for large data is the selection sort with Linked list Data Structure, and this is due to the fact that each of the algorithms and data Structure used are not the best. So, the runtime becomes longer, and the performance is weaker.

We note that it took approximately 59.8 seconds. Contrasting the best performance of the Quicksort algorithm with the binary search tree, which took an average runtime = 2528.8 part of a second.

So, as it is clear, the use of improper data structure affects the performance of the algorithm and may weaken it and reduce its speed in executing the arrangement of big data. (Like the Quicksort in Linked List Data Structure). Although the Quicksort is a good sorting algorithm, the use of ineffective data Structure led to an increase the runtime and poor performance.

We move on to the median finding algorithm, which was implemented with three different data structures as well: Array, Binary Search Tree and Heap. And the best, by a slight difference, is Array, due to the speed of direct access to the elements.

It is also noted that the algorithm may affect performance if it is not effective, such as the selection Sort, which is considered the worst in terms of implementation, even with a better data structure such as Heap, as the average runtime is 15.8 seconds.

The rest of the data structure and algorithms provided a similar performance in terms of runtime "parts of a second" as shown in the table.

Conclusion

The main objective of the project is to implement sort algorithms and compare them with different data structures to find the best sort algorithm for finding the data mediator.

As we mentioned in detail in the analysis part, the results showed that the most efficient sort algorithm is the Quicksort algorithm with binary search data structure, in terms of running time and Time complexity and the worst data structure is Linked list With Selection Sort algorithm. Also, in this project, we were able to read huge amounts of data and reduce the size of the pipeline according to the capabilities of the devices available.

Through research, implementation, modification, and observation, it was possible to learn more about the sort algorithms and their different implementation, and to determine which is the most convenient for the objective of our project. In conclusion, it was confirmed that the data structure may somehow affect the performance, due to how this huge data is stored and the way to access it. Also, the algorithm itself may affect performance if it is not efficient as well .

Another Pseudocode for algorithms with Data Structure

Array quickSort :

```
FUNCTION partition(arr, low, high):
SET i = (low-1)   # index of
smaller element
SET pivot = arr[high]  # pivot
FOR j = low to high:
IF arr[j] <= pivot:
INCREMENT i by 1
SWAP arr[i] and arr[j]
SWAP arr[i+1] and arr[high]
RETURN (i+1)
FUNCTION quickSort(arr, low, high):
IF length of arr is 1:
RETURN arr
IF low < high:
SET pi = partition(arr, low,
high)
CALL quickSort(arr, low, pi-1)
CALL quickSort(arr, pi+1, high)
```

Heap selectionSort:

```
FUNCTION heapSort(arr):
n = length of arr
FOR i FROM n to 0 step -1:
heapifyy(arr, n, i)
FOR i FROM n-1 to 0 step -1:
[•]SWAP arr[i] and arr
heapifyy(arr, i, 0)
FUNCTION heapifyy(arr, n, i):
largest = i
left = 2*i + 1
right = 2*i + 2
IF left < n and arr[i] < arr[left]:
largest = l
IF right < n and arr[largest] <
arr[right]:
largest = r
IF largest != i:
SWAP arr[i] AND arr[largest]
heapifyy(arr, n, largest)
FUNCTION heapSort(arr):
n = LENGTH OF arr
FOR i = (n//2) - 1 DOWN TO 0:
heapify(arr, n, i)
FOR i = n - 1 DOWN TO 0:
[•]SWAP arr[i] WITH arr
heapify(arr, i, 0)
```

Array selectionSort :

```
:FUNCTION selectionSort(array, size)
:FOR ind FROM 0 TO size
SET min_index = ind
:FOR j FROM ind + 1 TO size
:IF array[j] < array[min_index]
SET min_index = j
SWAP array[ind] WITH
array[min_index]
```

References

- Available at: https://www.geeksforgeeks.org/find-median-bst-time-o1-space/ (Accessed: January 20, 2023).
- Available at: https://www.geeksforgeeks.org/heap-sort/amp/ (Accessed: January 26, 2023).
- Available at: https://www.javatpoint.com/heap-sort (Accessed: January 25, 2023).
- Available at: https://www.scaler.com/topics/median-of-array/ (Accessed: February 1, 2023).
- Available: https://www.youtube.com/watch?v=6eo6CXdqfd0 (Accessed: February 2, 2023).
- Available at: https://iq.opengenus.org/quick-sort-on-linked-list/ (Accessed: February 3, 2023).
- Available at: https://www.programiz.com/dsa/b-plus-tree (Accessed: February 3, 2023).
- Available at: https://www.geeksforgeeks.org/selection-sort/ (Accessed: February 1, 2023).