Algorithms Project

Finding median

Author(s):

Shayma'a Saeed Almalki 439011029
Razan Sami Al-Refaey 441005995
Manar Ahmed Alabdli 438007643
Heba Osama Jan 441008803

Supervisor(s):

Areej Othman Alsini

Introduction

This project is about finding the median using sorting algorithms, an algorithm such as QuickSort, SelectionSort, medianOfMedians and radixSort were chosen, to measure and compare between algorithms in finding the medians that are better in performance and faster in extracting the result for sorted and unsorted data. It is in a program to determine the time for the execution of the algorithm, and the graphics library was used tKinter to create the interface of the program with specifying the type of data if it was sorted or unsorted due to the natural time difference between sorted and unsorted data. The data was read through a file with a range of 10000 data.

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

Chapter 1

Algorithms Discerption/Definition, Time Complexity and Code

1.1 Finding Median with Median_Selection_Sort

1.1.1 Discerption/Definition

The Algorithm will use Selection Sort to sorts the data and then finding the median which is the middle of them all.

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.

1.1.2 Time Complexity

The time complexity of this algorithm is $O(n^2)$, Because of calculating the median part takes $O(1)$, What remain is the time complexity of the Selection Sort Algorithm part, which is $O(n^2)$.

1.1.3 Code

```python
# This funaction will sort the data and find the median with Slection Sort Algorithm
def Median_Selection_Sort(A):
    import math
    Len = len(A)# O(1)
    # Traverse through all array elements
    for i in range(Len-1):# O(n)
        # Find the minimum element in remaining
        # unsorted array
        min_i = i# O(1)
        for j in range(i+1, Len): # O(n)
            if(A[j] < A[min_i]): # O(1)
                A[j], A[min_i] = A[min_i], A[j] # O(1)
        # Swap the found minimum element with
        # the first element
        if(min_i != i): # O(1)
            A[min_i], A[i] = A[i], A[min_i] # O(1)
 # Calculating median O(1)
    # if the data have an even number of elements we have to get the average of the 2 elements in the middle
    if( (Len % 2) == 0 ): # O(1)
        m1 = math.floor((Len)/2) # O(1)
        m2 = math.floor((Len-1)/2) # O(1)
        med = (A[m1] + A[m2])/2 # O(1)
    # otherwise we get the mid of them all
    else:
        med = A[math.floor((Len)/2)] # O(1)
    return med
```

Figure 1 Median_Selection_Sort Algorithm code

1.2 Finding Median with medianOfMedians

1.2.1 Discerption/Definition

It uses a divide and conquer strategy to efficiently compute the i_th smallest number in an unsorted list of size n. Selection algorithms are often used as part of other algorithms; for example, they are used to help select a pivot in quicksort.
The median-finding algorithm can find the i_th smallest element in a list in O(n) time.

How it works:

The median-of-medians algorithm is a deterministic linear-time selection algorithm. The algorithm works by dividing a list into sublists and then determines the approximate median in each of the sublists. Then, it takes those medians and puts them into a list and finds the median of that list. It uses that median value as a pivot and compares other elements of the list against the pivot. If an element is less than the pivot value, the element is placed to the left of the pivot, and if the element has a value greater than the pivot, it is placed to the right. The algorithm recurses on the list, until found the value it is looking for.

1.2.2 Time Complexity

This algorithm runs in O(n) time. n is divided into n/5 sublists of five elements each. If M is the list of all of the medians from these sublists, then M has n/5 median for each of the n/5 sublists. Let's call the median of this list (the median of the medians) p. Half of the n/5 elements in M are less than p.

Half of n/5 = n/10 For each of these n/10 elements, there are two elements that are smaller than it (since these elements were medians in lists of five elements two elements were smaller and two elements were larger). Therefore, there are 3n/10 < p and, in the worst case, the algorithm may have to recurse on the remaining 7n/10 elements. The time for dividing lists, finding the medians of the sublists, and partitioning takes

T(n) = T(n/5) + O(n) time, and with the recursion factored in, the overall recurrence to describe the median-of-medians algorithm is.

$T(n) \leq T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + O(n).$

1.2.3 Code

```python
#this function do the partitioning depends on pivot value
def partition(data, pivot):
    j = 0 # O(1)
    right = len(data) - 1 # O(1)
    i = 0 # O(1)
    while i <= right:  # O(n)
        if data[i] == pivot: # O(1)
            i += 1 # O(1)
        elif data[i] < pivot: # O(1)
            data[j], data[i] = data[i], data[j] # O(1)
            j += 1 # O(1)
            i += 1# O(1)
        else:
            data[right], data[i] = data[i], data[right] # O(1)
            right -= 1 # O(1)
    return j # O(1)

# the complexity of this function reach to # O(n)

#===========================================================================

#this is the main function of median_of_medians algorithm
# it is spliting the data array to lists of 5 element
#and find the median of each list and the the median of them
#then assign it as a pivot to da the partitioning
#then decide if the pivot is the median or decide the direction
#of the next call

def selectPivot(data, k):

    sublists = [data[i : i+5] for i in range(0, len(data), 5)]    # O(n/5) this is gathering each 5 element to one list

    sortedSublists = [sorted(sub) for sub in sublists]    # O(n/5) it sort each sublist created from previous step
    medians = [sub[len(sub) // 2] for sub in sortedSublists]     #O(n/5) it retrieve the median of each sublist

    # if # of sublists > 5 we will have recursive call with median list to get the median of them
    #otherwise we will choose the median of medians directly

    #so selectPivot function may take #O(n/5) = #O(n)

    if len(medians) <= 5: #O(1)
        pivot = sorted(medians)[len(medians) // 2] #O(1)
    else:
        pivot = selectPivot(medians, len(medians) // 2) #O(7n/10)

    res = partition(data, pivot) #as we see it may takes O(n)

    if k == res: #O(1)
        return pivot #O(1)
    if k < res: #O(1)
        return selectPivot(data[0:res], k)  #O(7n/10)
    else:
        return selectPivot(data[res+1:len(data)], k - res - 1) #O(7n/10)

    #the max complexity in this section = O(7n/10)
#===========================================================================

#here is the main call to implement the algorithm

def medianOfMedians(data, size):
    #if there is no data return none
    if data is None or len(data) == 0: #O(1)
        return None #O(1)

    return selectPivot(data, size) #O(7n/10)
#===========================================================================

#so the complexity we may get is:
#spliting + select pivot + partitioning
# O(n/5) + O(7n/10) + O(n)  = O(n)
```

Figure 2 medianOfMedians Algorithm code

1.3 Finding Median with quickSort

1.3.1 Discerption/Definition

Quick sort is based on the divide-and-conquer approach based on the idea of choosing one element as a pivot element and partitioning the array around it such that: Left side of pivot contains all the elements that are less than the pivot element Right side contains all elements greater than the pivot.

1.3.2 Time Complexity

The time complexity of this algorithm is $O(n^2)$.

1.3.3 Code

```python
# -*- coding: utf-8 -*-
"""
Created on Tue May 17 08:41:57 2022

@author: Manar
"""
def partition(arr, low, high):
    i = (low-1)      # index of smaller element constant-time
    pivot = arr[high]    # pivot constant-time

    for j in range(low, high):
        # If current element is smaller than or
        # equal to pivot
        if arr[j] <= pivot: #O(n)
            # increment index of smaller element
            i = i+1 #constant-time
            arr[i], arr[j] = arr[j], arr[i]
    arr[i+1], arr[high] = arr[high], arr[i+1] #constant-time
    return (i+1) #constant-time
#O(n)



def quickSort(arr, low, high):
    if len(arr) == 1:
        return arr
    if low < high: #constant-time
        # pi is partitioning index, arr[p] is now
        # at right place
        pi = partition(arr, low, high) #O(n)
        quickSort(arr, low, pi-1) # Sort left of partitioned # n-1
        quickSort(arr, pi+1, high) # Sort right  of partitioned  #n


# = O(n-1)+O(n)
# Worst case O(n^2)
```

Figure 3 quickSort Algorithm code

1.4 Finding Median with radixSort

1.4.1 Discerption/Definition

Radix sort is a sorting algorithm that sorts the elements by first grouping the individual digits of the same place value. Then, sort the elements according to their increasing/decreasing order. First, we will sort elements based on the value of the unit place. Then, we will sort elements based on the value of the tenth place. This process goes on until the last significant place.

1.4.2 Time Complexity

The radix sort that uses counting sort as an intermediate stable sort, the time complexity is $O(d(n+k))$.

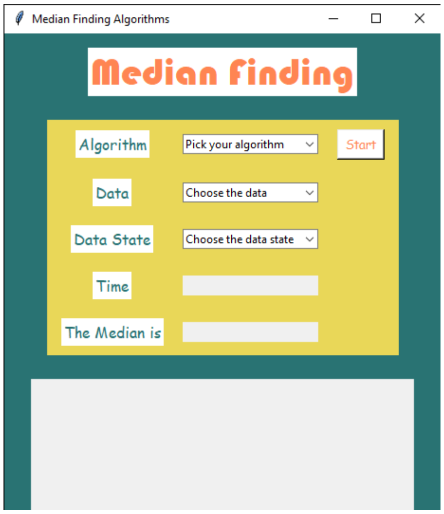
1.4.3 Code

```
# Radix sort

# Using counting sort to sort the elements in the basis of significant places
def countingSort(array, place):
    #length of data
    #O(1)
    size = len(array)

    #auxiliary array for assigning sorted data
    #O(1)
    output = [0] * size

    #This array is used for storing the count of the elements in the array.
    #O(1)
    count = [0] * 10

    # Calculate count of elements
    # digit position: ones / tens /hundred and so on
    #O(max)
    for i in range(0, size):
        index = array[i] // place
        count[index % 10] += 1

    # Calculate cumulative count
    #O(size)
    for i in range(1, 10):
        count[i] += count[i - 1]


    # Find the index of each element of the original array in count array
    # place the elements in output array

    #O(max)
    i = size - 1 #placing integers to it correcponding place in array in backward
    while i >= 0:
        index = array[i] // place
        output[count[index % 10] - 1] = array[i]
        count[index % 10] -= 1 #decrease count by 1
        i -= 1 #deacrement by 1


    #filling original array with sorted data
     #O(size)
    for i in range(0, size):
        array[i] = output[i]


# Main function to implement radix sort
def radixSort(array):

    # Get maximum element in data
    ##O(1)
    max_element = max(array)

    # Apply counting sort to sort elements based on place value.
    #O(maxDigit)
    place = 1
    while max_element // place > 0:
        countingSort(array, place)
        place *= 10 #increasse the place by 10 each loop
```

Figure 4 radixSort Algorithm code

1.5 tkinter

1.5.1 Interface Window



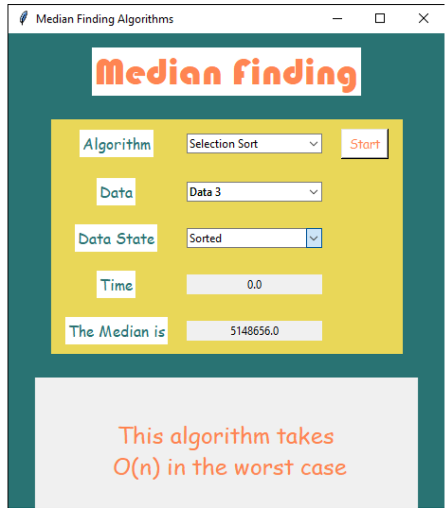


Figure 5 and 6 program interface window

1.6 Read/Write data

1.6.1 Read data

Reading the data from the file that has been chosen by the user.

```
def readData():
    select = dataChoice.get() # get the data that have been chosen
    global data, n
    if select == "Data 1":
        with open ('data1.txt', 'r') as d:
            data0 = d.readlines()    #to read the content of the data file
            data= [ x.strip() for x in data0]   #to delete the new line character from the data
            for i in range (len(data)):
                data[i] = int(data[i])
    elif select == "Data 2":
        with open ('data2.txt', 'r') as d:
            data0 = d.readlines()    #to read the content of the data file
            data= [ x.strip() for x in data0]   #to delete the new line character from the data
            for i in range (len(data)):
                data[i] = int(data[i])
    elif select == "Data 3":
        with open ('data3.txt', 'r') as d:
            data0 = d.readlines()    #to read the content of the data file
            data= [ x.strip() for x in data0]   #to delete the new line character from the data
            for i in range (len(data)):
                data[i] = int(data[i])
    n = len(data) # reset the value of n
    state = dataState.get() # get the state of data
    if state == "Sorted": # apply sorting if the state eaual to 'Sorted'
        data.sort()
```

Figure 7 reading the data from chosen file code

1.6.2 Write data

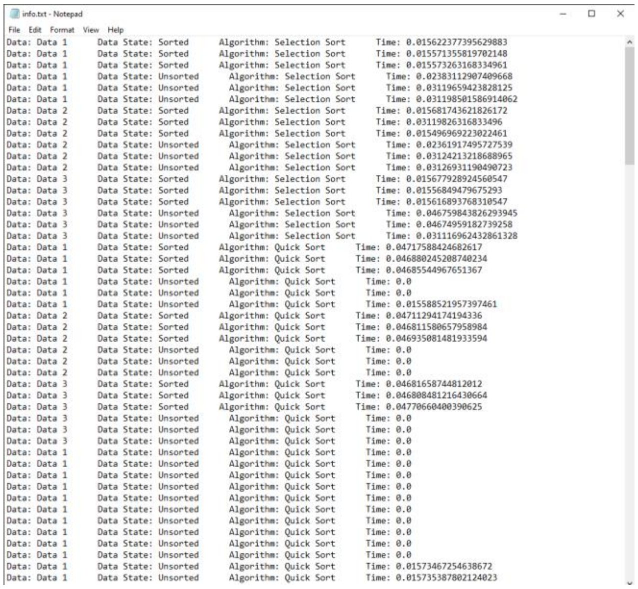
Write down (store it) in the info.txt file.

```
def write_to_file(): # Store the information of the analysis
    outfile = open('info.txt', 'a')
    outfile.write('Data: '+dataChoice.get()+'      Data State: '+dataState.get()+'      Algorithm: '
                  + choice.get()+'      Time: ' +total.get()+ '\n')
    outfile.close()
```

Figure 8 write(store) the algorithm name, chosen data, state of data and total time to .txt file

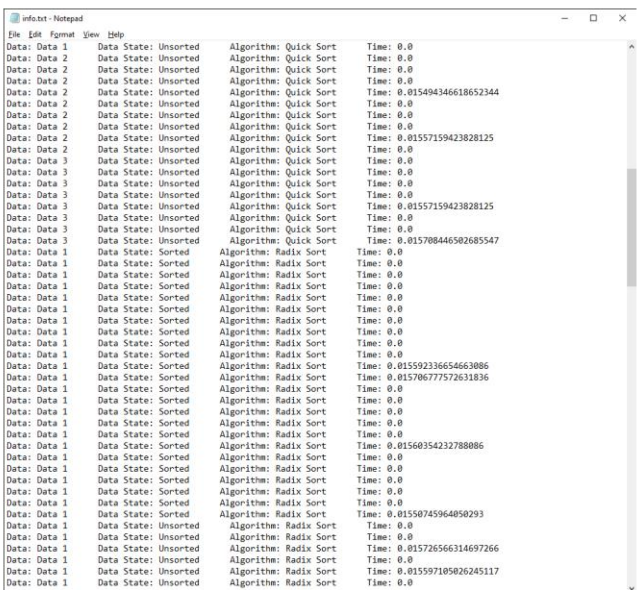
167.2 .txt file

Some of data capture from info.txt file.



```
info.txt - Notepad
File Edit Format View Help
Data: Data 1	Data State: Sorted	Algorithm: Selection Sort	Time: 0.015622377395629883
Data: Data 1	Data State: Sorted	Algorithm: Selection Sort	Time: 0.015571355819702148
Data: Data 1	Data State: Sorted	Algorithm: Selection Sort	Time: 0.015573263168334961
Data: Data 1	Data State: Unsorted	Algorithm: Selection Sort	Time: 0.02383112907409668
Data: Data 1	Data State: Unsorted	Algorithm: Selection Sort	Time: 0.03119659423828125
Data: Data 1	Data State: Unsorted	Algorithm: Selection Sort	Time: 0.031198501586914062
Data: Data 2	Data State: Sorted	Algorithm: Selection Sort	Time: 0.015681743621826172
Data: Data 2	Data State: Sorted	Algorithm: Selection Sort	Time: 0.03119826316833496
Data: Data 2	Data State: Sorted	Algorithm: Selection Sort	Time: 0.015496969223022461
Data: Data 2	Data State: Unsorted	Algorithm: Selection Sort	Time: 0.02361917495727539
Data: Data 2	Data State: Unsorted	Algorithm: Selection Sort	Time: 0.03124213218688965
Data: Data 2	Data State: Unsorted	Algorithm: Selection Sort	Time: 0.03126931190490723
Data: Data 3	Data State: Sorted	Algorithm: Selection Sort	Time: 0.015677928924560547
Data: Data 3	Data State: Sorted	Algorithm: Selection Sort	Time: 0.01556849479675293
Data: Data 3	Data State: Sorted	Algorithm: Selection Sort	Time: 0.015616893768310547
Data: Data 3	Data State: Unsorted	Algorithm: Selection Sort	Time: 0.046759843826293945
Data: Data 3	Data State: Unsorted	Algorithm: Selection Sort	Time: 0.04674959182739258
Data: Data 3	Data State: Unsorted	Algorithm: Selection Sort	Time: 0.031116962432861328
Data: Data 1	Data State: Sorted	Algorithm: Quick Sort	Time: 0.04717588424682617
Data: Data 1	Data State: Sorted	Algorithm: Quick Sort	Time: 0.046880245208740234
Data: Data 1	Data State: Sorted	Algorithm: Quick Sort	Time: 0.04685544967651367
Data: Data 1	Data State: Unsorted	Algorithm: Quick Sort	Time: 0.0
Data: Data 1	Data State: Unsorted	Algorithm: Quick Sort	Time: 0.0
Data: Data 1	Data State: Unsorted	Algorithm: Quick Sort	Time: 0.015588521957397461
Data: Data 2	Data State: Sorted	Algorithm: Quick Sort	Time: 0.04711294174194336
Data: Data 2	Data State: Sorted	Algorithm: Quick Sort	Time: 0.046811580657958984
Data: Data 2	Data State: Sorted	Algorithm: Quick Sort	Time: 0.046935081481933594
Data: Data 2	Data State: Unsorted	Algorithm: Quick Sort	Time: 0.0
Data: Data 2	Data State: Unsorted	Algorithm: Quick Sort	Time: 0.0
Data: Data 2	Data State: Unsorted	Algorithm: Quick Sort	Time: 0.0
Data: Data 3	Data State: Sorted	Algorithm: Quick Sort	Time: 0.04681658744812012
Data: Data 3	Data State: Sorted	Algorithm: Quick Sort	Time: 0.046808481216430664
Data: Data 3	Data State: Sorted	Algorithm: Quick Sort	Time: 0.04770660400390625
Data: Data 3	Data State: Unsorted	Algorithm: Quick Sort	Time: 0.0
Data: Data 3	Data State: Unsorted	Algorithm: Quick Sort	Time: 0.0
Data: Data 3	Data State: Unsorted	Algorithm: Quick Sort	Time: 0.0
Data: Data 1	Data State: Unsorted	Algorithm: Quick Sort	Time: 0.0
Data: Data 1	Data State: Unsorted	Algorithm: Quick Sort	Time: 0.0
Data: Data 1	Data State: Unsorted	Algorithm: Quick Sort	Time: 0.0
Data: Data 1	Data State: Unsorted	Algorithm: Quick Sort	Time: 0.0
Data: Data 1	Data State: Unsorted	Algorithm: Quick Sort	Time: 0.0
Data: Data 1	Data State: Unsorted	Algorithm: Quick Sort	Time: 0.0
Data: Data 1	Data State: Unsorted	Algorithm: Quick Sort	Time: 0.0
Data: Data 1	Data State: Unsorted	Algorithm: Quick Sort	Time: 0.0
Data: Data 1	Data State: Unsorted	Algorithm: Quick Sort	Time: 0.0
Data: Data 1	Data State: Unsorted	Algorithm: Quick Sort	Time: 0.0
Data: Data 1	Data State: Unsorted	Algorithm: Quick Sort	Time: 0.01573467254638672
Data: Data 1	Data State: Unsorted	Algorithm: Quick Sort	Time: 0.015735387802124023
```

Figure 9 Stored data in the .txt file



```
info.txt - Notepad
File Edit Format View Help
Data: Data 1	Data State: Unsorted	Algorithm: Quick Sort	Time: 0.0
Data: Data 2	Data State: Unsorted	Algorithm: Quick Sort	Time: 0.0
Data: Data 2	Data State: Unsorted	Algorithm: Quick Sort	Time: 0.0
Data: Data 2	Data State: Unsorted	Algorithm: Quick Sort	Time: 0.0
Data: Data 2	Data State: Unsorted	Algorithm: Quick Sort	Time: 0.015494346618652344
Data: Data 2	Data State: Unsorted	Algorithm: Quick Sort	Time: 0.0
Data: Data 2	Data State: Unsorted	Algorithm: Quick Sort	Time: 0.0
Data: Data 2	Data State: Unsorted	Algorithm: Quick Sort	Time: 0.0
Data: Data 2	Data State: Unsorted	Algorithm: Quick Sort	Time: 0.01557159423828125
Data: Data 2	Data State: Unsorted	Algorithm: Quick Sort	Time: 0.0
Data: Data 3	Data State: Unsorted	Algorithm: Quick Sort	Time: 0.0
Data: Data 3	Data State: Unsorted	Algorithm: Quick Sort	Time: 0.0
Data: Data 3	Data State: Unsorted	Algorithm: Quick Sort	Time: 0.0
Data: Data 3	Data State: Unsorted	Algorithm: Quick Sort	Time: 0.0
Data: Data 3	Data State: Unsorted	Algorithm: Quick Sort	Time: 0.01557159423828125
Data: Data 3	Data State: Unsorted	Algorithm: Quick Sort	Time: 0.0
Data: Data 3	Data State: Unsorted	Algorithm: Quick Sort	Time: 0.0
Data: Data 3	Data State: Unsorted	Algorithm: Quick Sort	Time: 0.015708446502685547
Data: Data 1	Data State: Sorted	Algorithm: Radix Sort	Time: 0.0
Data: Data 1	Data State: Sorted	Algorithm: Radix Sort	Time: 0.0
Data: Data 1	Data State: Sorted	Algorithm: Radix Sort	Time: 0.0
Data: Data 1	Data State: Sorted	Algorithm: Radix Sort	Time: 0.0
Data: Data 1	Data State: Sorted	Algorithm: Radix Sort	Time: 0.0
Data: Data 1	Data State: Sorted	Algorithm: Radix Sort	Time: 0.0
Data: Data 1	Data State: Sorted	Algorithm: Radix Sort	Time: 0.0
Data: Data 1	Data State: Sorted	Algorithm: Radix Sort	Time: 0.0
Data: Data 1	Data State: Sorted	Algorithm: Radix Sort	Time: 0.0
Data: Data 1	Data State: Sorted	Algorithm: Radix Sort	Time: 0.0
Data: Data 1	Data State: Sorted	Algorithm: Radix Sort	Time: 0.015592336654663086
Data: Data 1	Data State: Sorted	Algorithm: Radix Sort	Time: 0.015706777572631836
Data: Data 1	Data State: Sorted	Algorithm: Radix Sort	Time: 0.0
Data: Data 1	Data State: Sorted	Algorithm: Radix Sort	Time: 0.0
Data: Data 1	Data State: Sorted	Algorithm: Radix Sort	Time: 0.0
Data: Data 1	Data State: Sorted	Algorithm: Radix Sort	Time: 0.0
Data: Data 1	Data State: Sorted	Algorithm: Radix Sort	Time: 0.0
Data: Data 1	Data State: Sorted	Algorithm: Radix Sort	Time: 0.01560354232788086
Data: Data 1	Data State: Sorted	Algorithm: Radix Sort	Time: 0.0
Data: Data 1	Data State: Sorted	Algorithm: Radix Sort	Time: 0.0
Data: Data 1	Data State: Sorted	Algorithm: Radix Sort	Time: 0.0
Data: Data 1	Data State: Sorted	Algorithm: Radix Sort	Time: 0.0
Data: Data 1	Data State: Sorted	Algorithm: Radix Sort	Time: 0.0
Data: Data 1	Data State: Sorted	Algorithm: Radix Sort	Time: 0.01550745964050293
Data: Data 1	Data State: Unsorted	Algorithm: Radix Sort	Time: 0.0
Data: Data 1	Data State: Unsorted	Algorithm: Radix Sort	Time: 0.0
Data: Data 1	Data State: Unsorted	Algorithm: Radix Sort	Time: 0.015726566314697266
Data: Data 1	Data State: Unsorted	Algorithm: Radix Sort	Time: 0.0
Data: Data 1	Data State: Unsorted	Algorithm: Radix Sort	Time: 0.015597105026245117
Data: Data 1	Data State: Unsorted	Algorithm: Radix Sort	Time: 0.0
```

Figure 10 Stored data in the .txt file

Chapter 2

Data Analysis

2.1 Analyse the data and compare

2.1.1 Analyse and Compare the data

Data	Data State	Algorithm	Time
Data 1	Sorted	Selection Sort	0.015622377395629883
Data 1	Sorted	Selection Sort	0.015571355819702148
Data 1	Sorted	Selection Sort	0.015573263168334961
AVG: 0.015588998794555664			
Data 1	Unsorted	Selection Sort	0.02383112907409668
Data 1	Unsorted	Selection Sort	0.03119659423828125
Data 1	Unsorted	Selection Sort	0.031198501586914062
AVG: 0.028742074966430664			
Data 2	Sorted	Selection Sort	0.015681743621826172
Data 2	Sorted	Selection Sort	0.03119826316833496
Data 2	Sorted	Selection Sort	0.015496969223022461
AVG: 0.020792325337727863			
Data 2	Unsorted	Selection Sort	0.02361917495727539
Data 2	Unsorted	Selection Sort	0.03124213218688965
Data 2	Unsorted	Selection Sort	0.03126931190490723
AVG: 0.028710206349690754			
Data 3	Sorted	Selection Sort	0.015677928924560547
Data 3	Sorted	Selection Sort	0.01556849479675293
Data 3	Sorted	Selection Sort	0.015616893768310547
AVG: 0.015621105829874674			
Data 3	Unsorted	Selection Sort	0.046759843826293945
Data 3	Unsorted	Selection Sort	0.04674959182739258
Data 3	Unsorted	Selection Sort	0.031116962432861328
AVG: 0.04154213269551595			

Data 1	Sorted	Quick Sort	0.04717588424682617
Data 1	Sorted	Quick Sort	0.046880245208740234
Data 1	Sorted	Quick Sort	0.04685544967651367
AVG: 0.046970526377360024			
Data 1	Unsorted	Quick Sort	0.015588521957397461
Data 1	Unsorted	Quick Sort	0.01573467254638672
Data 1	Unsorted	Quick Sort	0.015735387802124023
AVG: 0.015686194101969402			
Data 2	Sorted	Quick Sort	0.04711294174194336
Data 2	Sorted	Quick Sort	0.046811580657958984
Data 2	Sorted	Quick Sort	0.046935081481933594

AVG: 0.04695320129394531			
Data 2	Unsorted	Quick Sort	0.015494346618652344
Data 2	Unsorted	Quick Sort	0.01557159423828125
Data 2	Unsorted	Quick Sort	0.0009958744049072266
AVG: 0.01068727175394694			
Data 3	Sorted	Quick Sort	0.04681658744812012
Data 3	Sorted	Quick Sort	0.046808481216430664
Data 3	Sorted	Quick Sort	0.04770660400390625
AVG: 0.047110557556152344			
Data 3	Unsorted	Quick Sort	0.01557159423828125
Data 3	Unsorted	Quick Sort	0.015708446502685547
Data 3	Unsorted	Quick Sort	0.005028963088989258
AVG: 0.012103001276652018			

Data 1	Sorted	Radix Sort	0.015592336654663086
Data 1	Sorted	Radix Sort	0.015706777572631836
Data 1	Sorted	Radix Sort	0.01560354232788086
AVG 0.01563421885172526			
Data 1	Unsorted	Radix Sort	0.015726566314697266
Data 1	Unsorted	Radix Sort	0.015597105026245117
Data 1	Unsorted	Radix Sort	0.015553951263427734
AVG 0.015625874201456707			
Data 2	Sorted	Radix Sort	0.008651256561279297
Data 2	Sorted	Radix Sort	0.002687692642211914
Data 2	Sorted	Radix Sort	0.00099945068359375
AVG 0.004112799962361653			
Data 2	Unsorted	Radix Sort	0.0009958744049072266
Data 2	Unsorted	Radix Sort	0.0019981861114501953
Data 2	Unsorted	Radix Sort	0.0027408599853515625
AVG 0.0019116401672363281			
Data 3	Sorted	Radix Sort	0.015564918518066406
Data 3	Sorted	Radix Sort	0.015635013580322266
Data 3	Sorted	Radix Sort	0.015699148178100586
AVG 0.015633026758829754			
Data 3	Unsorted	Radix Sort	0.015572071075439453
Data 3	Unsorted	Radix Sort	0.015583276748657227
Data 3	Unsorted	Radix Sort	0.015585899353027344
AVG 0.015580415725708008			

Data 1	Sorted	Median of medians	0.008489370346069336
Data 1	Sorted	Median of medians	0.015681743621826172
Data 1	Sorted	Median of medians	0.015584945678710938
AVG 0.013252019882202148			
Data 1	Unsorted	Median of medians	0.015589237213134766

Data 1	Unsorted	Median of medians	0.015616893768310547
Data 1	Unsorted	Median of medians	0.015712499618530273
AVG 0.015639543533325195			
Data 2	Sorted	Median of medians	0.015593290328979492
Data 2	Sorted	Median of medians	0.016228199005126953
Data 2	Sorted	Median of medians	0.01569533348083496
AVG 0.0158389409383138			
Data 2	Unsorted	Median of medians	0.015612602233886719
Data 2	Unsorted	Median of medians	0.015568733215332031
Data 2	Unsorted	Median of medians	0.015567541122436523
AVG 0.015582958857218424			
Data 3	Sorted	Median of medians	0.015494585037231445
Data 3	Sorted	Median of medians	0.01562809944152832
Data 3	Sorted	Median of medians	0.0157470703125
AVG 0.015623251597086588			
Data 3	Unsorted	Median of medians	0.01550436019897461
Data 3	Unsorted	Median of medians	0.015582561492919922
Data 3	Unsorted	Median of medians	0.015591144561767578
AVG: 0.01555935541788737			

3 set of different data of size n = 1000

Selection:

AVG sorted data = 0.0173341433207194.

AVG unsorted data=0.03299813800387912.

Quicksort:

AVG sorted data = 0.04701142840915256.

AVG unsorted data=0.012825489044189453.

Radix:

AVG sorted data = 0. 011793348524305558.

AVG unsorted data=0.011039310031467015

Median of medians:

AVG sorted data = 0. 01490473747253418.

AVG unsorted data=0.015593952602810329

2.1.2 Conclusion

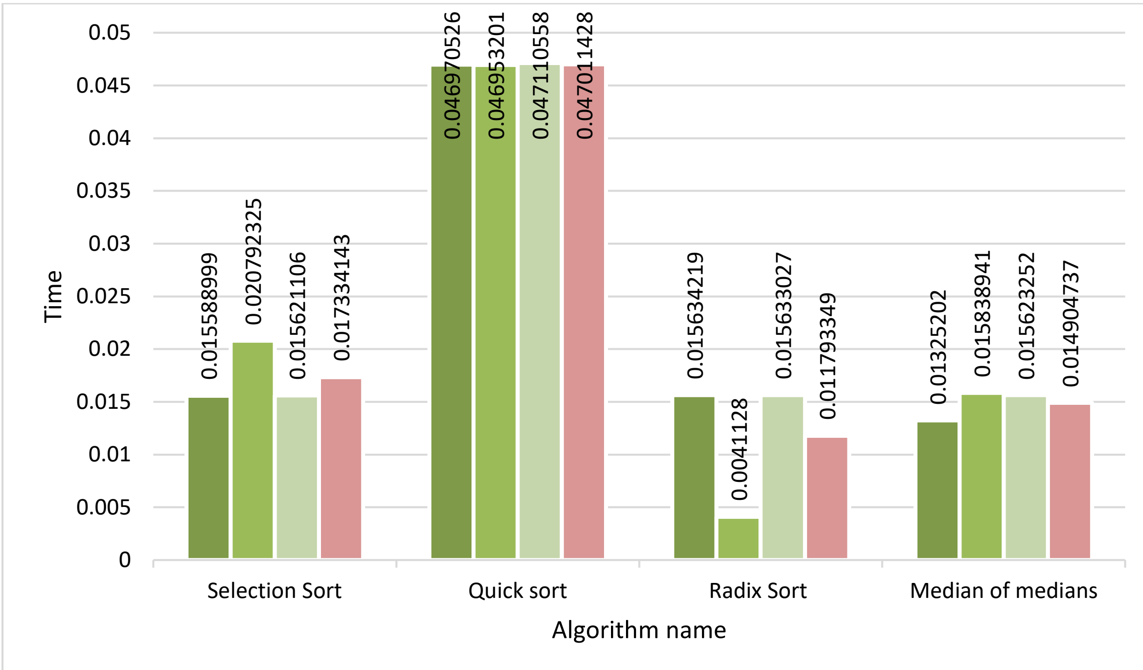


Figure 11 Chart of Sorted data

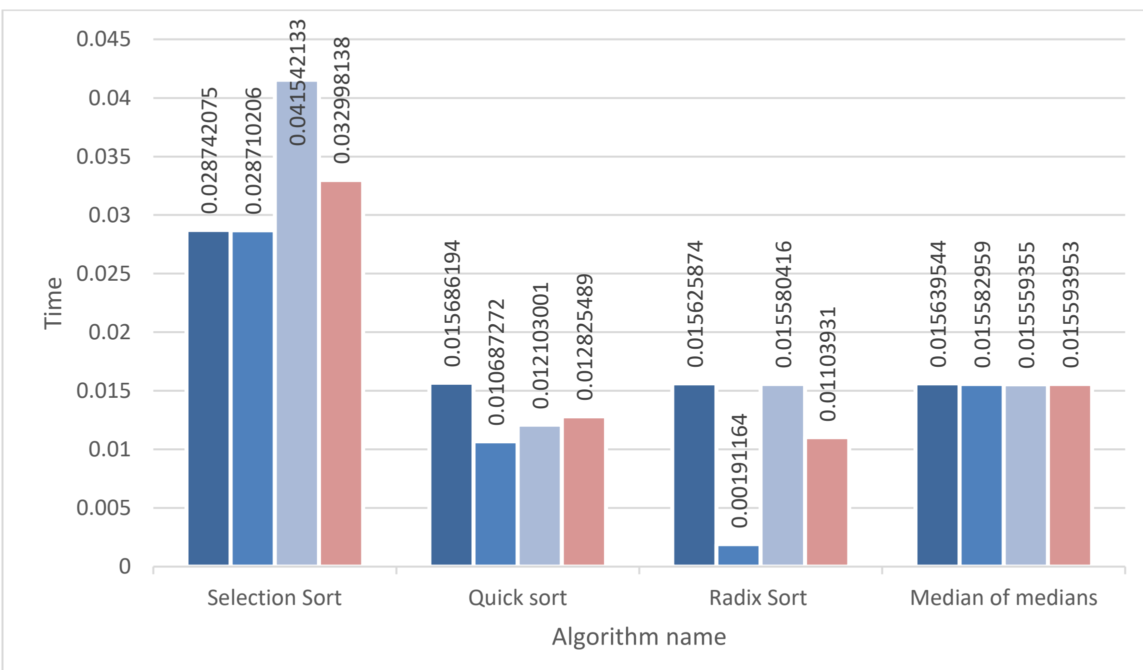


Figure 12 Chart of Unsorted data

A comparison of algorithms has been studied by a data size of 1000 divided into 3 data files was used for measurement and analysis to find the median. The sample was tested once for each datafile and once for its state Stored and Unsorted, then it's identified the average of each datafile of each algorithm, After that the average for each sorted and unsorted data of each algorithm. We noticed through the graph that Quicksort has different time complexity depend on unsorted or sorted data, it shows that the Quicksort algorithm If the data was sorted in the process of finding the median in the it takes 0.047011428 seconds, and if it is unsorted it takes the time 0.012825489044189453 seconds, this is a large time difference compared to other algorithms. On the other hand, Radix sort has a small difference between the time in the case of sorted 0.011793348524305558 seconds, and unsorted 0.011039310031467015 seconds. In a same condition, the Selection Sort have a big difference in the case of the data is unsorted 0.03299813800387912 again, and if it is sorted 0.0173341433207194 second. As for the median of medians, they have the same time, but varying in the case that they are unsorted 0.015593952602810329 seconds, and sorted 0.01490473747253418 seconds, it can be because the algorithm is special to find the median, so it has a convergent and varying time, unlike Quicksort.

Closure

depending on what we studied and learned and what we saw when we test these algorithms and analyzed them, we can confirm that some of them take linear time such as Radix sort and Median of medians algorithms and they could be better choice than others to find the median in large data sets, on the other hand there are some of the algorithms take more than O(n) such as Quick sort and Selection sort and they're not the best choice for large data sets.

Reference:

[Online]. Available: https://www.youtube.com/watch?v=6eo6CXdqfd0 . [Accessed: April 2022].

[Online]. Available: https://www.youtube.com/watch?v=sNtu2oGDRvU&t=10s . [Accessed: April 2022].

[Online]. Available: https://www.youtube.com/watch?v=PBu2QgU6Qfk . [Accessed: April 2022].

[Online]. Available: https://brilliant.org/wiki/median-finding-algorithm/ . [Accessed: April 2022].

[Online]. Available: https://www.codegrepper.com/profile/vastemonde . [Accessed: April 2022].

[Online]. Available: https://www.geeksforgeeks.org/selection-sort/ . [Accessed: April, May 2022].

[Online]. Available: . [Accessed: https://www.youtube.com/watch?v=7h1s2SojIRw . April 2022].

[Online]. Available: . [Accessed: https://www.youtube.com/watch?v=3Bbm3Prd5Fo&t=935s . April 2022].

[Online]. Available: https://interviewalgorithm.wordpress.com/sortingordering/median-of-medians-algorithm/ . [Accessed: May 2022].

[Online]. Available: https://ltcconline.net/greenl/courses/CAHSEE/StatisticsProb/CAHSEEMeanMedianMode.htm . [Accessed: May 2022].

[Online]. Available: https://www.youtube.com/watch?v=0C2405R-uGk&t=316s . [Accessed: May 2022].

[Online]. Available: https://www.programiz.com/dsa/radix-sort . [Accessed: May 2022].

[Online]. Available: https://youtu.be/UwUWEL58kVY . [Accessed: May 2022].

[Online]. Available: https://youtu.be/4ungd6NXFYI . [Accessed: May 2022].

[Online]. Available: https://youtu.be/BVGRgTALQ44 . [Accessed: May 2022].

[Online]. Available: https://youtu.be/kMoqmJ3TUFE . [Accessed: May 2022].

[Online]. Available: https://www.programiz.com/dsa/selection-sort . [Accessed: May 2022].