Design And Analysis of Algorithms

Project Title: Sorting Algorithms

PROJECT REPORT

Submitted by

SOWJANYA SADASHIVA

STUDENT ID: 1001898874

MASTER OF SCIENCE, COMPUTER SCIENCE

UNIVERSITY OF TEXAS, ARLINGTON
701 S Nedderman Dr, Arlington, TX 76019

DATE: NOV 21ST, 2021

Table of Contents

1. ***Description***

I have implemented seven sorting algorithms in this project including, Merge sort, Heap sort, Quick sort regular, Quick sort using median, Insertion sort, Selection sort, Bubble sort. All the algorithms are implemented in python.

This project has two python files,

Sorting_algorithms.py has a class named sorting_algorithms with all the sorting algorithms listed above defined in the class. Each algorithm is implemented separately which takes number of elements and list of elements from the user through GUI, sorts the elements and gives the sorted list back to the gui to show it on the GUI.

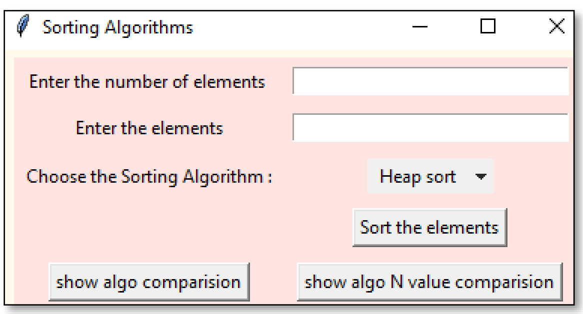
Sorting_algorithms_gui.py has simple gui using Tkinter GUI toolkit. It has one entry for number of elements, one for list of elements and it has a drop-down menu for the user to select the sorting algorithm, once button 'sort the elements' Is hit the elements will be sorted and show it to the user with the average time taken to sort those elements. We can also see 'show algo comparison button' which gives the comparison graph of average time taken by each algorithm to rum the elements entered by the user, and 'show algo N value comparison' gives the run time comparison of all the algorithms to sort N randomly generated values.

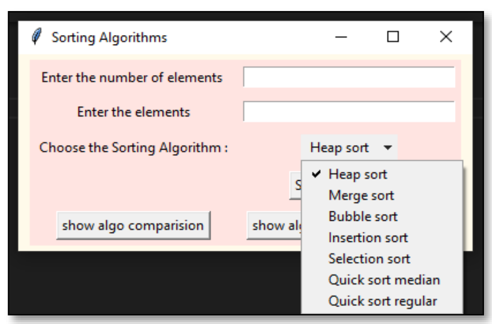
I have used,

```
from tkinter import *
from tkinter import ttk
```

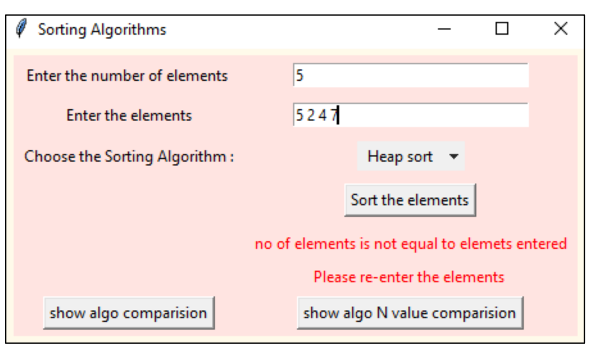
to build GUI.

Design of GUI,





If the number of elements does not match the list of elements enter it throughs an error,



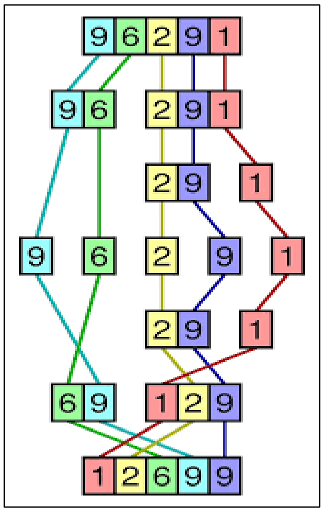
2. Sorting Algorithms

a. Merge sort

It is divide-and-conquer algorithm, we divide a list of elements into multiple subsets and sort those subsets of lists and add them together to get one sorted list.

Steps:

a. Divide the unsorted list of N elements into N/2 sub lists,
   - Repeat this process till each sub list contain one element
b. Take the adjacent singleton list sort and merge.
   - Repeat this process of sorting and merging till we get one single sorted list.



Implementation of merge sort algorithm,

a. One to divide the list of elements into multiple sub lists with left partition and right partition till there is one element in each subset.
b. Second to sort and merge the sub lists.

This algorithm is designed to take number of elements and list of elements from the user and sort those elements by applying merge sort.

The run-time complexity of merge sort: Big – O Notation is a statistical measure, used to describe the complexity of the algorithm.

Time Complexity

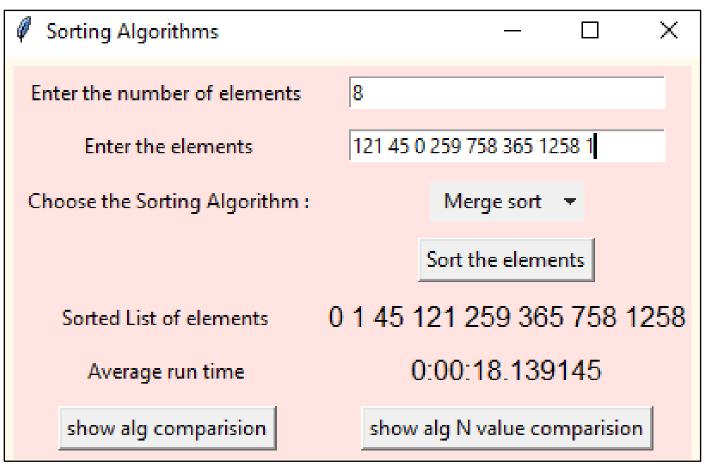
- Best case: O (n log n)
- Worst case: O (n log n)
- Average case: O (n log n)

Space complexity: n

Algorithm

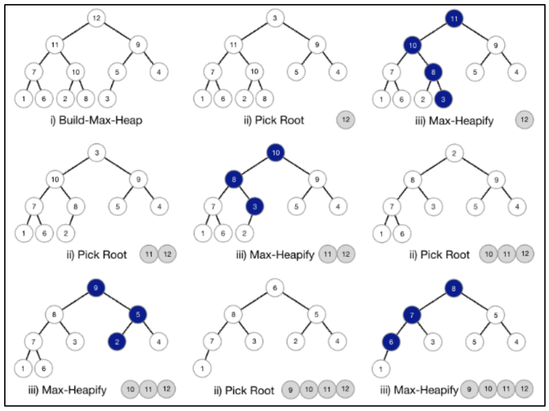
```
def merge_sort(self, arr):
    no_of_elements = len(arr)
    if no_of_elements == 1:
        return arr

    middle_element = no_of_elements // 2

    left_partition = self.merge_sort(arr[:middle_element])
    right_partition = self.merge_sort(arr[middle_element:])

    return self.merge(left_partition, right_partition)

def merge(self,left, right):
    sorted_arr = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            sorted_arr.append(left[i])
            i += 1
        else:
            sorted_arr.append(right[j])
            j += 1

    sorted_arr.extend(left[i:])
    sorted_arr.extend(right[j:])
    return sorted_arr
```

Time taken to sort n elements



b. Heap sort

It is a comparison-based sorting algorithm. It's a binary tree data structure. We are sorting the algorithms in ascending order. It takes in the list of elements from the user into the heap_sort method, which passes it to heapify method. In heapify method the elements are sorted by max heap steps. If the ith value is less than current largest value, then ith elements go to left else it goes to right side. This process is repeated through the levels of list till all the elements are sorted.



If the elements are already sorted the algorithm takes O(n) time to run, it has no swaps to perform since the list is already sorted.

Time Complexity

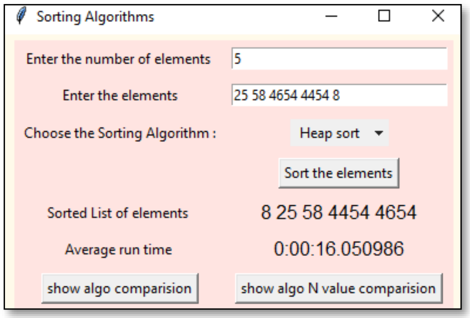
- Best case: O (n)
- Worst case: O (n log n)
- Average case: O (n log n)

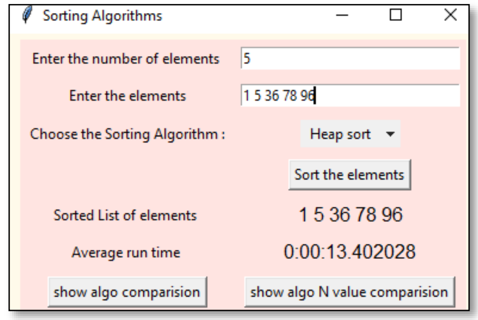
Space complexity: O(n)

Algorithm

```
def heapify(arr, n, i):
  largest_value = i
  left = 2 * i + 1
  right = 2 * i + 2
  if left < n and arr[largest_value] < arr[left]:
    largest_value = left
  if right < n and arr[largest_value] < arr[right]:
    largest_value = right
  if largest_value != i:
    arr[i], arr[largest_value] = arr[largest_value], arr[i]
    heapify(arr, n, largest_value)

def heap_sort(arr):
  no_of_elements = len(arr)
  for i in range(no_of_elements // 2 - 1, -1, -1):
    heapify(arr, no_of_elements, i)
  for i in range(no_of_elements - 1, 0, -1):
    arr[i], arr[0] = arr[0], arr[i]
    heapify(arr, i, 0)
  return arr
```

Time taken to sort n elements

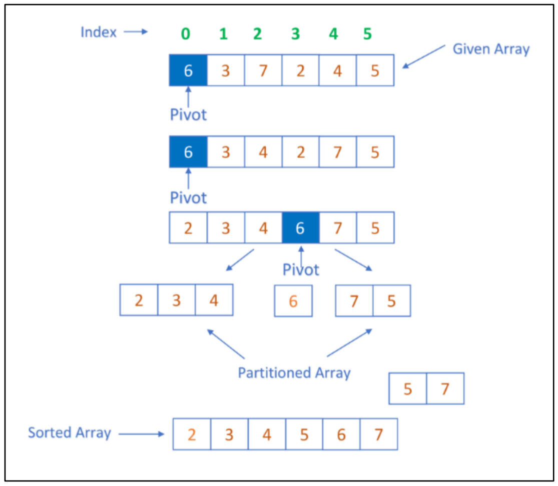




c. Quick Sort

Quick sort is an in-place algorithm. It is one of quickest sorting algorithms. We first select a pivot point and then start comparing and sorting the elements. Pivot point can be first element, last elements, median or random element. Each pivot point selected has its own advantage and disadvantage.

In this project we have implemented two ways, one with median as a pivot point and the other with left or first element as pivot. We'll compare see which is quick-er. Once we select the pivot point, elements less than pivot point go to left and greater than pivot point goes to right side of it.



Time Complexity

- Best case: O (n log n) when the pivot point is median.
- Average Case: O (n log n)
- Worst Case: O (n^2)
    - Worst case occurs when the elements are already sorted, and we select first or last element as pivot point and the algorithm must compare with each element.

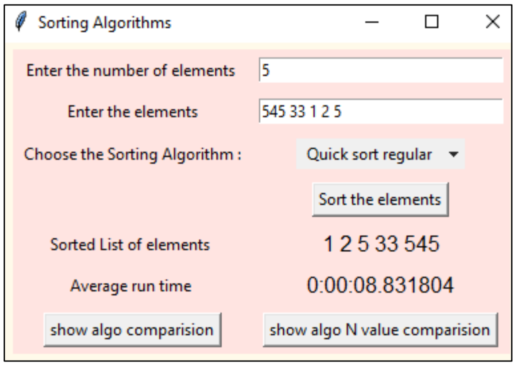
Space Complexity: O (log n)

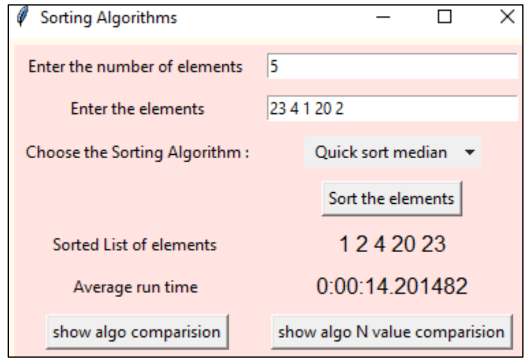
Algorithm

```python
def quick_sort_regular(arr, left=0, right=None):
    if right is None:
        right = len(arr) - 1

    def _quicksort(arr, left, right):
        if left >= right:
            return
        pivot = partition(arr, left, right)
        _quicksort(arr, left, pivot - 1)
        _quicksort(arr, pivot + 1, right)
    return _quicksort(arr, left, right)


def partition(arr, left, right):
    pivot_position = left
    for i in range(left + 1, right + 1):
        if arr[i] <= arr[left]:
            pivot_position += 1
            arr[i], arr[pivot_position] = arr[pivot_position], arr[i]
    arr[pivot_position], arr[left] = arr[left], arr[pivot_position]
    return pivot_position
```

```python
def quick_sort_median(arr):
  def quickSortMedian(left, right):
    if(left >= right):
      return

    medians = ( int )( ( left + right ) / 2 )
    if(
        (
            ( arr[left] <= arr[medians] ) and ( arr[medians] <= arr[right] ) )
             or ( ( arr[right] <= arr[medians] ) and ( arr[medians] <= arr[left] )
             )
        ):
      pass
    elif(
         (
             ( arr[medians] <= arr[left] ) and ( arr[left] <= arr[right] ) )
              or ( ( arr[right] <= arr[left] ) and ( arr[left] <= arr[medians] )
              )
        ):
      arr[left], arr[medians] = arr[medians], arr[left]
    elif(
         (
             ( arr[left] <= arr[right] ) and ( arr[right] <= arr[medians] ) )
             or ( ( arr[medians] <= arr[right] ) and ( arr[right] <= arr[left] )
             )
        ):
      arr[right], arr[medians] = arr[medians], arr[right]

    pivot = arr[ medians ]
    arr[ medians ] = arr[ right ]

    i = left
    j = right - 1
    while( i <= j ):
      while( i < right and arr[i] <= pivot ):
        i += 1
      while( j >= left and arr[j] >= pivot ):
        j -= 1

      if( i < j ):
        temp = arr[ i ]
        arr[ i ] = arr[ j ]
        arr[ j ] = temp
        i += 1
        j -= 1

    arr[ right ] = arr[ j + 1 ]
    arr[ j + 1 ] = pivot

    quickSortMedian(left, j)
    quickSortMedian(j + 2, right)
    return arr

  quickSortMedian(0, len(arr)-1)
  return arr
```

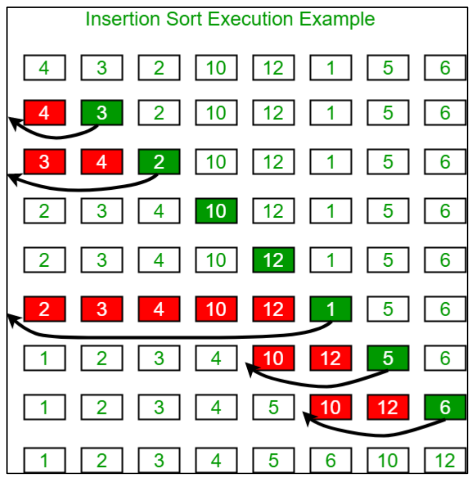
Time taken by the algorithm





d. Insertion Sort

It is a simple sorting algorithm to implement, it takes the input elements compares each element with the next element in the list, if the next element is greater than the current element no swap is done if the next element is less than the current element swaps those two values. This algorithm is used when the number of elements is small. It is also best algorithm when the elements are almost sorted.



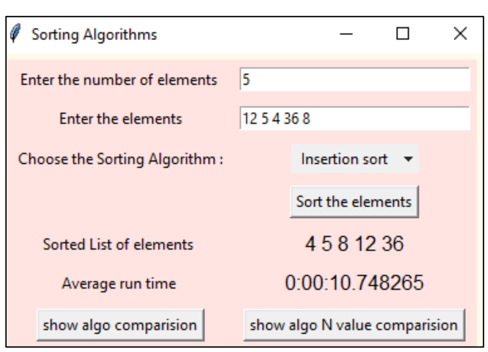
Time complexity

- Best case: O (n) when the elements are already sorted.
- Average Case: $O(n^2)$
- Worst case: $O(n^2)$ If the list is in reverse order the algorithm takes maximum time.

Space Complexity: O (1) constant

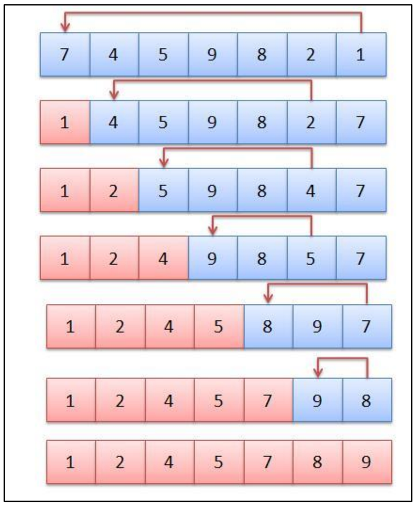
```
def insertion_sort(arr):
  for i in range(len(arr),):
    curr_value = arr[i]
    curr_position = i
    while curr_position > 0 and arr[curr_position - 1] >curr_value:
      arr[curr_position] = arr[curr_position - 1]
      curr_position = curr_position - 1
    arr[curr_position] = curr_value
  return arr
```

Time taken by the algorithm,



e. Selection sort

It is a simple sorting algorithm. It is an in-place algorithm in which the list is divided into two parts. We first consider the first element and compare it with each element next to it till we find the min value, if we find the min value, we swap the elements.



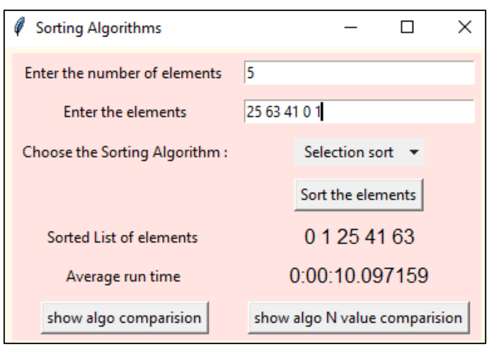
Time complexity

- Best case: $O(n^2)$ if the list is already sorted no swaps needed but we compare with each element.
- Average Case: $O(n^2)$
- Worst case: $O(n^2)$

Space Complexity: O (1) constant

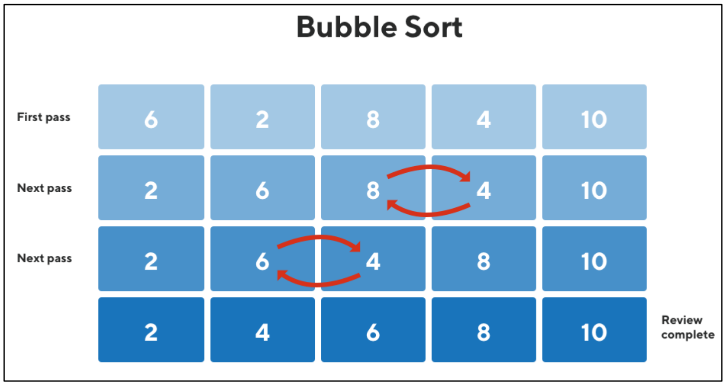
```python
def selection_sort(arr):
    for i in range(len(arr)):
        idx = i
        for j in range(i+1, len(arr)):
            if arr[j] < arr[idx]:
                idx = j
        arr[i], arr[idx] = arr[idx], arr[i]
    return arr
```

Time taken by the algorithm



f. Bubble sort

It is a simple comparison-based sorting algorithm, it is an in-place algorithm. it compares adjacent elements and swaps them if they are not in order. It is not suitable for large set of data.



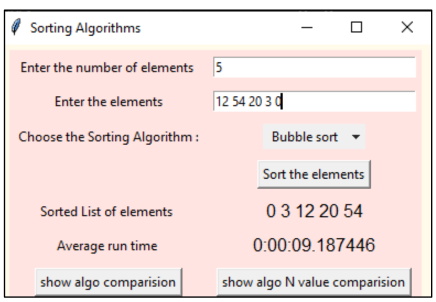
Time complexity

- Best case: O(n) if the list is already sorted.
- Average Case: $O(n^2)$
- Worst case: $O(n^2)$ if the list is in reverse order.

Space Complexity: O (1)

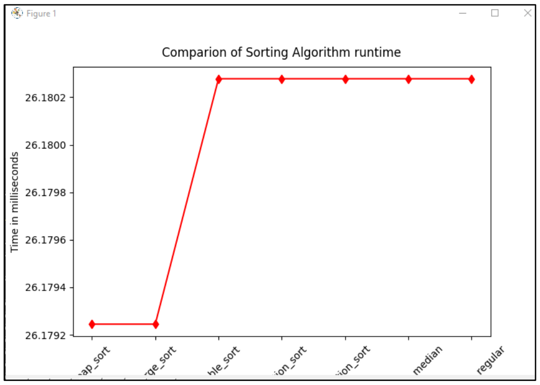
```
def bubble_sort(arr):
    for i in range(len(arr) - 1):
        for j in range(0, len(arr) - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
    return arr
```

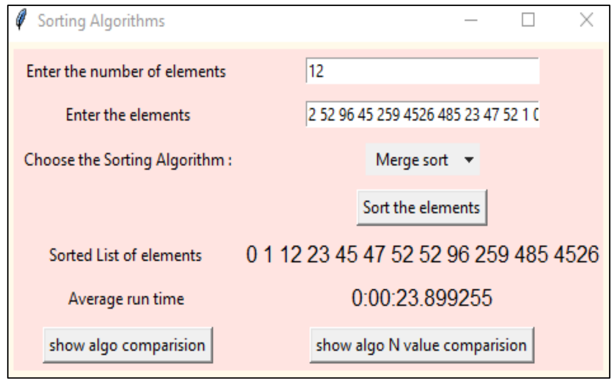
Time taken by the algorithm



3. Conclusion

Here we can see the comparison of time taken by all the algorithms for N elements given by the user, the time taken by each algorithm for small data set if almost the same, but it changes when the input data is large.





Comparison of algorithms for N random elements, here we can see how the algoerithms behave when we try to sort latge amount of data, I have used random function to generate random values to input to the sorting algortithm.