Comparing Different Common Sorting Algorithms

Shahriar Ahmad Fahim

Tokyo Institute of Technology

February 11, 2021

1 Introduction

This paper explains written code for project 2 to compare different sorting algorithms. The goal of this project is to gain a good idea on writing the code and time complexity of different sorting algorithms by analysing them on chart.

2 Sorting algorithms in this project

The algorithms presented in this project are mentioned in the following table with their time complexities:

Table 1: Algorithms discussed in this project with their time complexities (Best, Average and Worst)

No.	Name of algorithm	Best	Average	Worst
1	Insertion sort	$\Omega(n)$	$\theta((n^2)$	$O(n^2)$
2	Quick sort	$\Omega(n\log(n))$	$\theta((n\log(n))$	$O(n^2)$
3	Merge sort	$\Omega(n\log(n))$	$\theta(n\log(n))$	$O(n\log(n))$
4	Bubble sort	$\Omega(n)$	$\theta((n^2)$	$O(n^2)$
5	Selection sort	$\Omega(n^2)$	$\theta((n^2)$	$O(n^2)$
6	Pigeonhole sort	$\Omega(n+k)$	$\theta((n+k)$	$O(n+k)$
7	Python Builtin sort	-	-	-

3 Explanation of code

For this project, the code for insertion sort, quick sort, and a portion for merge sort were already given. We wrote the code for the rest of the algorithms except Python Builtin Sort (it is already in Python library). So any modifications done to the given skeleton of code along with these newly written code are explained:

3.1 Improving the swap function

The swap function was already given to us. However, we improved this function by not using an extra variable in terms of space complexity. It works fine to swap a number i.e. enough for this project. The main change is shown:

```
    array[i] = array[i] + array[j]
    array[j] = array[i] - array[j]
    array[i] = array[i] - array[j]
```

The new function is named as swapImproved in the program.

3.2 Merge Sort

We write the code for the mergeSort function through the recursive algorithm.

```
1 def mergeSort(array):
2     if len(array)==1: #base case
3             return array
4           else: #recursive case
5             return mergeRec(mergeSort(array[:len(array)//2]),
              ↪  mergeSort(array[len(array)//2:]))
```

For the base case when the array has only one element, we return the array. For the recursive case when the array has more than one element, we divide the array in half, for example , using array[:len(array)//2]. Then we call mergeSort again recursively on it. Similarly, we do the same for the other half of the array - array[len(array)//2:]. Finally, the mergeSort applied on these two halves are passed to mergeRec function for merging.

3.3 Bubble Sort

```
1 def bubbleSort(array):
2     res = array.copy()
3     for i in range(0,len(res)-1):
4         for j in range(0,len(res)-1-i):
5             if res[j]>res[j+1]:
6                 swapImproved(res,j,j+1)
7     return res
```

First, we copy the array to another variable named res in line 2. Then we iterate (i) through each element in res. Again for each iteration (j) under each i (value of i at first is 0), we check if an element is greater than the next element. If this is true (line 5), we swap them i.e. the idea si to send the bigger number to the latter position. In this way, we send the maximum number at the end of the array. It is easy to understand that we do not need to check this equality (line 5) for the last position of the array because there is no element after that. All the iterations of j ends.

Correspondingly, i becomes 1 for the next iteration. Now, we do the similar process again. We have to to send the second largest element to the second last position. So we run the iteration of j until len(res)-1-1 = len(res)-2. So, checking the equality (line 5) this time ends one position earlier, which makes sense.

In this technique, biggest numbers from larger to smaller order go the end of the list like a bubble, as the name suggests.

3.4 Selection Sort

```
1  def selectionSort(array):
2      sorted = []
3      unsorted = deepcopy(array)
4      T = len(unsorted)
5      for c in range(T):
6          min = 0
7          for i in range(len(unsorted)):
8              if unsorted[i]<unsorted[min]:
9                  min = i
10         sorted.append(unsorted[min])
11         del unsorted[min]
12     return sorted
```

We declare two lists- one empty named sorted another named unsorted with deepcopy of the given array variable. We use deepcopy so that the any change does not impact the original array.

We find the minimum of the unsorted list using the for loop from line 7 to line 9. We append that element to the empty array (line 10) and remove that element from the unsorted array (line 11). In this way, we select the minimum value each time and send them to the new list (sorted). We repeat this iteration (line 5) for all the elements in unsorted list, eventually sorting all the numbers in the new sorted list and return it.

3.5 Pigeonhole Sort

```
1  def pigeonholeSort(array):
2      min = array[0]
3      max = array[0]
4      for c in array:
5          if max<c:
6              max = c
7          if min>c:
8              min = c
9      count_array = [0]*(max-min+1)
10     for c in array:
11         count_array[c-min] += 1
12     sorted = []
13     for i in range(len(count_array)):
14         if count_array[i]==1:
15             sorted.append(i+min)
16     return sorted
```

First we find the maximum and minimum in the array (line 2 to 8). Then we declare count_array with (maximum-minimum+1) number of zeros (line 9).

The idea is, the minimum and the maximum hold the first and the last position or index in the count_array. So, we can imagine there is a place for each integer number between the minimum and maximum whether it is in the given array or not. Now we iterate through the array again (line 10) and for each number we find, we make 0 to 1 with corresponding index position (c-min) in the count_array. For example, if minimum is 3, its index position becomes c-min = 3-3 = 0 in the count_array. Finally, we iterate through the count_array again to find out which indices have value 1 to get our sorted list.

4 Sample Output

A sample output or the chart generated through our program is attached below:

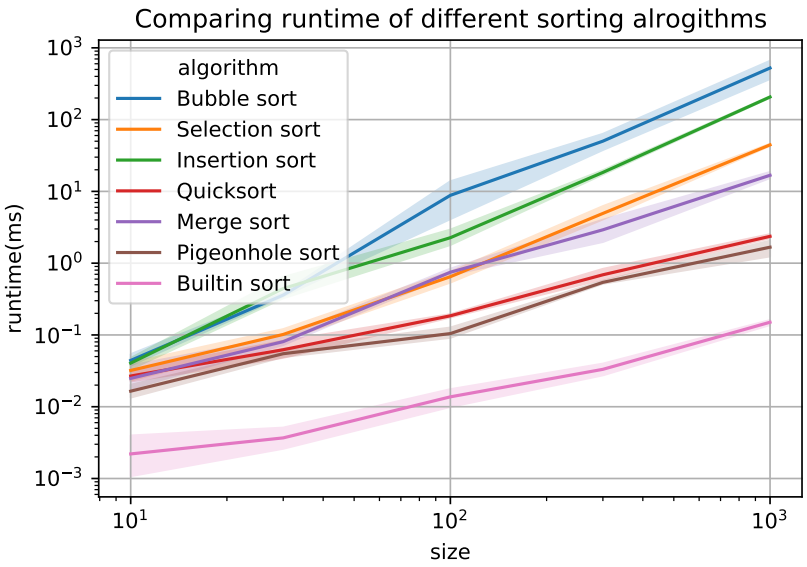

Figure 1: Sample chart generated showing the runtime of different algorithms

The test size of data was [10, 30, 100, 300, 1000] with repetitions = 5. The graph shows us run-time in milliseconds in the vertical axis and test data/number size in the horizontal axis. Both the axes are in logarithmic scale. We can see that quick sort and pigeonhole sort remain the most efficient among our algorithms implemented in this project. And the slowest algorithm is the bubble sort.

5 Conclusion

In this project, we gained some idea on the sorting algorithms, their time complexity, and using data frame in Python. We can conclude that none of our algorithms can beat the Python built in sort which is obvious. So it may be advisable to use the builtin sort normally. Otherwise, we may choose the algorithm with least runtime compared to the size of data. Also, depending on the space complexity, we may choose which type of sort we should use. But this discussion is out of scope for this project. Studying sorting algorithms is a good way to learn algorithm and grow interest in computer science.