SOFTWARE IMPLEMENTATION OF RADIX SORT AND QUICK SORT ALGORITHMS IN PYTHON WITH AN OBJECT-ORIENTED APPROACH

Gachechiladze Lela
Ph.D., Associate Professor
of the Faculty of Informatics and Control Systems
Georgian Technical University

Margvelashvili Ana
PhD student
of the Faculty of Informatics and Control Systems
Georgian Technical University

Sorting of data according to a certain order according to the increase or decrease in its essence belongs to the class of combinatorial tasks. According to specialists, approximately 25% of computer time is spent on systematic sorting tasks. Therefore, the mentioned algorithms deserve special attention.

As usual, every organization sorts this or that data, and in many cases, it is necessary to sort a significant amount of data.

Based on the above, we will consider the well-known Radix Sort and Quick Sort algorithms and their implementation in the Python programming language with an object-oriented approach, which involves developing a user class and methods, is placing them in a user module and importing the latter one. At the end of the article, we present a brief analysis of the above-mentioned algorithms.

Radix Sort is a unique sorting algorithm that works on the basic principle that numbers are sets of numbers.

The Radix Sort algorithm is used to sort integers in ascending or descending order, because integers have only one mathematical component - digits.

First of all, the mentioned algorithm provides for determining the largest value in the integer array, then counting the number of sequences in the number of this maximum value and sorting the numbers first according to the digits of the smallest sequence (units) (say, in ascending order), then tens, hundreds, etc. depending on the number of coefficients in the number with the maximum value [1].

We will implement the algorithm on list elements, because this data structure has a lot in common with arrays.

Thus, the software implementation of the Radix Sort algorithm with an object-oriented approach in the Python programming language using the Pycharm editor is presented below, and the result of the program execution is shown in the first picture.

```
class MyClass:
def method1(self, place):
    list_size=len(mylist)
    x=[0]*list_size
```

```
        count=[0]*10
        for i in range(0, list_size):
            ind=mylist[i]//place
            count[ind%10]+=1
        for i in range(1, 10):
            count[i]=count[i] + count[i-1]
        i=list_size-1
        while i>=0:
            ind=mylist[i]//place
            x[count[ind%10]-1]=mylist[i]
            count[ind % 10] -= 1
            i-=1
        for i in range(0,list_size):
            mylist[i]=x[i]
    def radix_sort(self):
        ob=MyClass()
        max_item=max(mylist)
        place=1
        while max_item//place>0:
            ob.method1(place)
            place*=10
        print(mylist)
ob1=MyClass()
mylist = [123,24,345,6,567,678,76,44,357,10,234,555,767,1,15]
print("Current List:")
print(mylist)
print("\nSorted List by ASC:")
ob1.radix_sort()
```

```
C:\Python34\python.exe C:/Users/Pc/PycharmProjects/sortiing/radix.py
Current List:
[123, 24, 345, 6, 567, 678, 76, 44, 357, 10, 234, 555, 767, 1, 15]

Sorted List by ASC:
[1, 6, 10, 15, 24, 44, 76, 123, 234, 345, 357, 555, 567, 678, 767]
```

fig. 1 The result of program execution using the Radix Sort algorithm

We will sort now the elements of the same list in ascending order based on the Quick Sort algorithm.

The quick sort algorithm is based on the "disconnect and conquer" principle. The array, in the role of which we will again use a data structure - a list, is divided into sub-lists by selecting the supporting element. As a result of dividing the list, the supporting element should be placed in such a way that the elements of less value than it is on the left side of the list, and the elements of greater value than it is on the right

side of it. The left and right sublists are split again using the same approach, and this process continues until each sublist is found to contain one element [2].

This time, the software implementation of the Quick Sort algorithm with an object-oriented approach in the Python programming language using the Pycharm editor has the form presented below, and the result of the program execution, since we have not changed the initial values in the code, will obviously remain the same as shown in the first picture.

```
class MyClass:
    def method2(self, mylist, low, high):
        pivot=mylist[high]
        i=low-1
        for k in range(low, high):
            if(mylist[k]<=pivot):
                i+=1
                (mylist[i], mylist[k])=(mylist[k], mylist[i])
        (mylist[i+1], mylist[high])=(mylist[high], mylist[i+1])
        return i+1
    def quick_sort(self, mylist, low, high):
        ob=MyClass()
        if low<high:
            x = ob.method2(mylist, low, high)
            ob.quick_sort(mylist, low, x - 1)
            ob.quick_sort(mylist, x+1, high)
ob1=MyClass()
mylist = [123, 24, 345, 6, 567, 678, 76, 44, 357, 10, 234, 555, 767, 1, 15]
print("Current List:")
print(mylist)
size=len(mylist)
ob1.quick_sort(mylist, 0, size-1)
print("\nSorted List by ASC:")
print(mylist)
```

So, the results of the implementation of both algorithms are the same, because the initial list, in the case of both algorithms, is the same.

If we put all the methods of the two codes shown above in the MyClass class we present, and we format it as a user module file with a .py extension, which does not include method calls, and we import the class from our module, then we will be able to separate (encapsulate) the implementation of the class) possibility and we will be able to call the methods defined in the class quite easily from the module using the class object and dot operator.

However, if both algorithms do not sort the elements of the list in ascending order, one of, say, the quicksort algorithm, will sort the elements according to their descending values. In this case, we will have to make only one change in method2,

namely: replace the condition if(mylist[k]<=pivot) with the preceding one: if(mylist[k]>=pivot).

Thus, our module is ready and its import code is presented below, and the program execution results are shown in Figure 2. mymodule here is the name of our module, which includes the class we created named MyClass and its four methods.

```
from mymodule import MyClass
ob1=MyClass()
mylist = [123, 24, 345, 6, 567, 678, 76, 44, 357, 10, 234, 555, 767, 1, 15]
print("Current List:")
print(mylist)
print("\nSorted List by ASC:")
ob1.radix_sort()
size=len(mylist)
ob1.quick_sort(mylist, 0, size-1)
print("\nSorted List by DESC:")
print(mylist)
```

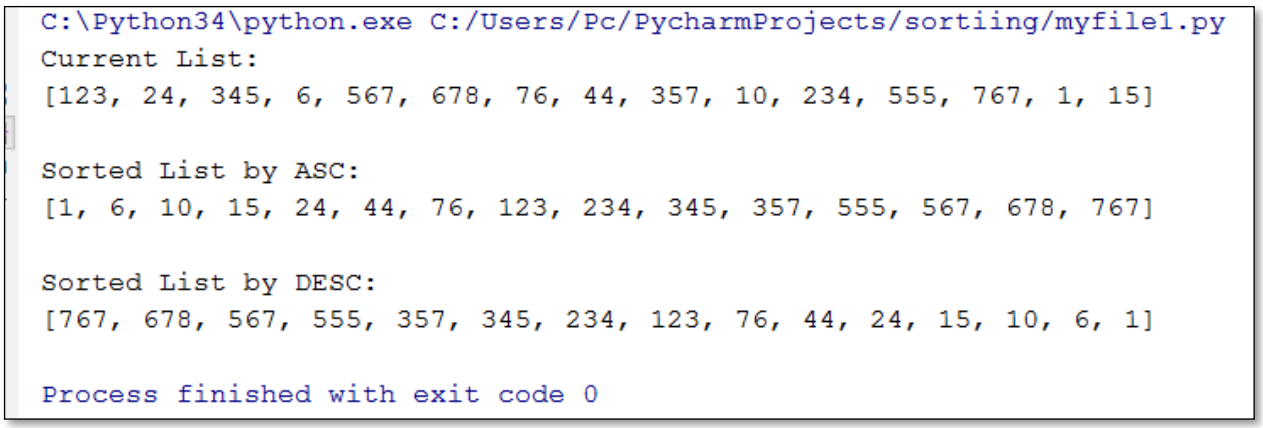

fig. 2 Program execution results using class import from module

As for the difficulties of the above-mentioned algorithms [3]. The time complexity of the Quick Sort algorithm is in the worst case [Big-O]: O(n2) and it occurs when the element with the largest or smallest value is considered as the base element.

Best complexity [big omega]: O(n*log n) is obtained when the support element is the middle element or is close to the middle element.

The Radix Sort algorithm, because it is not a comparison algorithm, has certain advantages. In particular, the time complexity is O(d(n+k)), where d is the cycle number, and O(n+k) is the time complexity. It turns out that the Radix Sort algorithm has linear time complexity. If we take a fairly large number of data, the algorithm will run in linear time, but intermediate sorting will take up a lot of space. This will make the sorting space inefficient.s

The Quick Sort algorithm is used when the programming language allows the use of recursion and the time complexity is insignificant [4].

The Radix Sort algorithm is used in cases where the numerical range of the right-hand data is large.

References:

1. Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, Charles E. Leiserson, Introduction to Algorithms, The MIT Press (3rd Edition), 2009. Pp 149-152.
2. Michael T. Goodrich, Roberto Tamassia, Michael H. Goldwasser. Data Structures and Algorithms in Python. John Wiley & Sons, Inc. 2013. Pp 550-561.
3. Erickson J. Algorithms. 1st edition. 2019. Pp 30-31.
4. Bullinaria J., Kerber M. Lecture Notes for Data Structures and Algorithms. School of Computer Science University of Birmingham Birmingham, UK. 2019. Pp 25-29.