Scientific Programming Practical 15

Introduction

Luca Bianco - Academic Year 2018-19
luca.bianco@fmach.it

Sorting algorithms

Given an input sequence (U) of un-sorted elements **$U=u1,u2,...,un$** produce a new sequence **$S=s1,s2,...,sn$** which is a **permutation of the elements in U** such that $s1\leq s2,...,\leq sn$.

Sorting algorithms

Given an input sequence (U) of un-sorted elements ***U*=*u*1,*u*2,...,*un*** produce a new sequence ***S*=*s*1,*s*2,...,*sn*** which is a **permutation of the elements in U** such that *s*1≤*s*2,...,≤*sn.*

Today we will practice:

Merge sort

Quick sort

Divide and conquer algorithms

D*ivide et impera* algorithms (*divide and conquer* in English) work by:

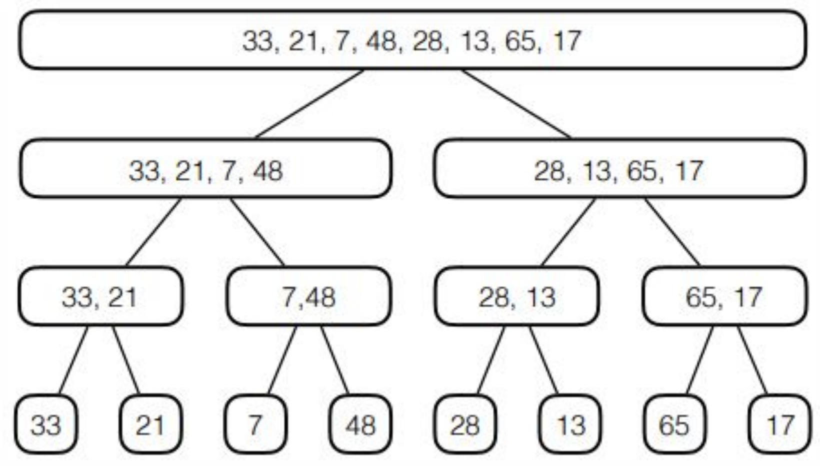
1. **dividing** the original problem in smaller problems (based on some parameters like the size of the input list);
2. **recursively solving the smaller problems** (recursively splitting them until the minimum unit – the base case – is reached and solved);
3. **combining** the partial results in the final solution.

Merge sort

The idea of **merge sort** is that given an unsorted list ***U=u_1,u_2,...,u_n*** the ***MergeSort*** procedure:

1. breaks the list *U* in two similarly sized lists (if the size is odd, the same list -- e.g. the first -- is always one element bigger than the other);
2. **calls *MergeSort* recursively** on the two sublists **until they are one element only → one element lists ARE sorted by definition**;
3. **merges two already sorted sublists in a sorted** (bigger) **list**.

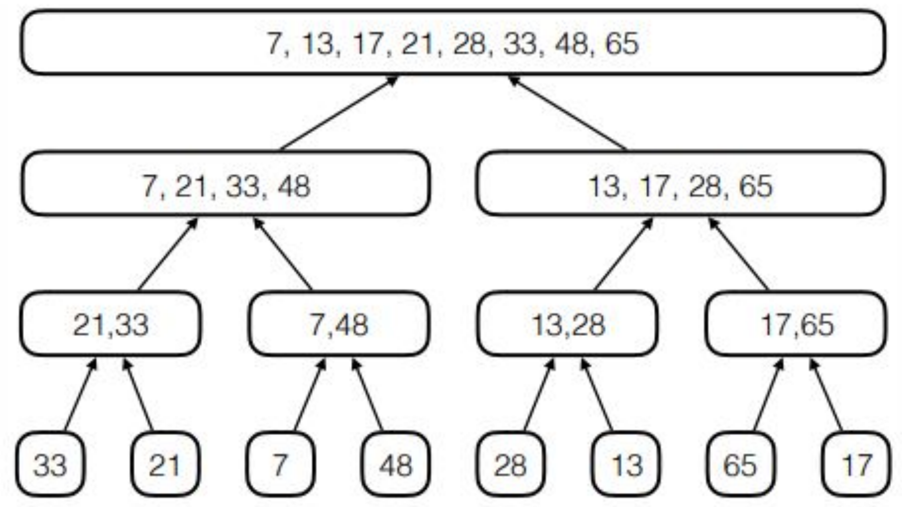
Merge sort



Keep dividing

1 sized lists are **ordered**!

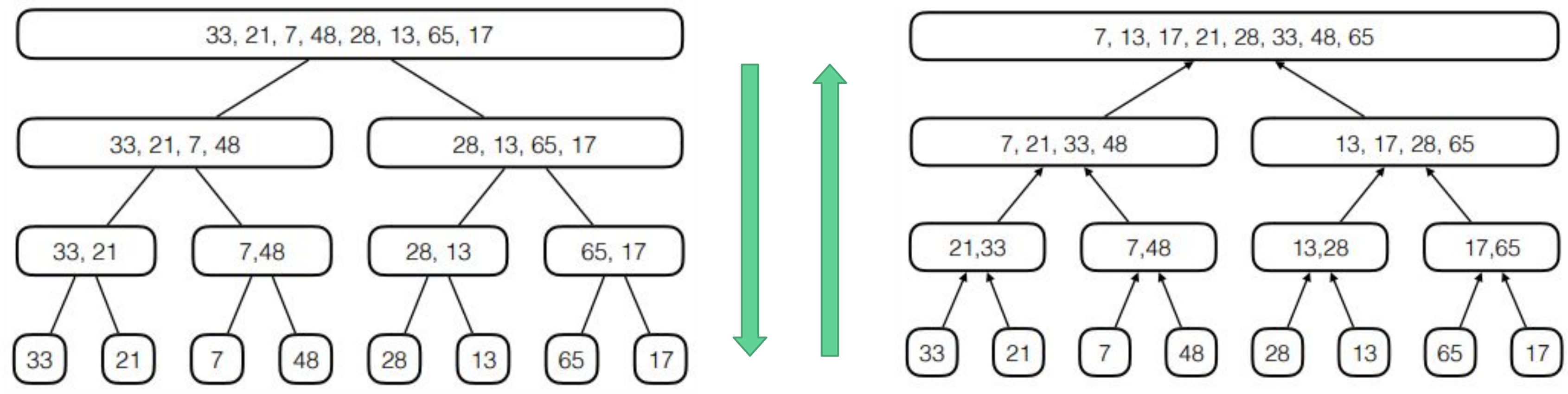
Merge sort



1 List only: solution

merge sorted lists
into bigger sorted
lists

Merge sort



33, 21, 7, 48, 28, 13, 65, 17

33, 21, 7, 48

28, 13, 65, 17

33, 21

7,48

28, 13

65, 17

33

21

7

48

28

13

65

17

7, 13, 17, 21, 28, 33, 48, 65

7, 21, 33, 48

13, 17, 28, 65

21,33

7,48

13,28

17,65

33

21

7

48

28

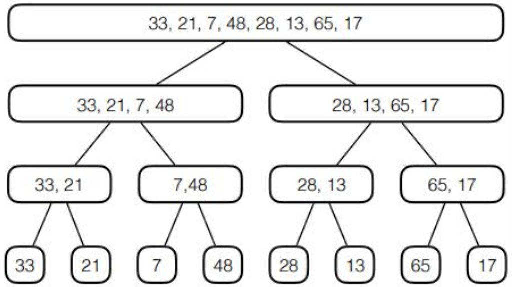
13

65

17

A good implementation of this sorting algorithm has complexity ***O*(*nlogn*)** where *n* is the number of elements in the list to sort

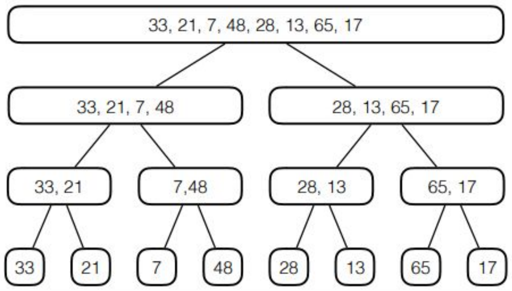
Merge sort: implementation



Merge sort requires three methods:

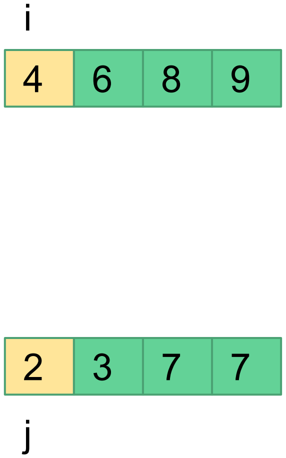
1. (`merge`): **gets two sorted lists and produces a sorted list** with all the elements. Builds the return list by getting the minimum element of the two lists, "removing" it from the corresponding list and appending it to the list with the result. "removal" can be done by using two indexes pointing to the smallest elements of each of the two (sub)lists and incrementing the index of the minimum of the two (i.e. the element that is also copied to the result list);
2. (`recursiveMergeSort`): gets **an unordered (sub)list**, the the **index of the beginning** of the list and the index of the end of the list and recursively **splits it in two halves until it reaches lists with length 0 or 1**, at that point **it starts merging pairs of sorted lists to build the result (with `merge`)**;
3. (`mergeSort`) gets an **unordered list and applies the** `recursiveMergeSort` method to it starting **from position 0 to *len*−1**..

Merge sort: implementation

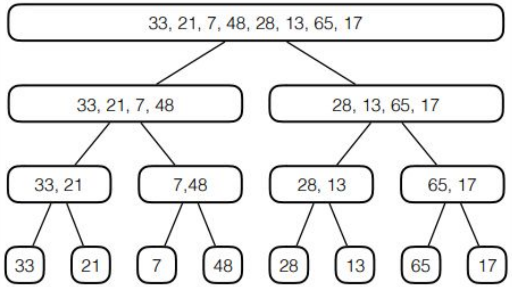


The Merge method:

gets two sorted lists and produces a sorted list with all the elements. Builds the return list by getting the minimum element of the two lists, “removing” it from the corresponding list and appending it to the list with the result (using two indexes pointing to the minimum of each list).

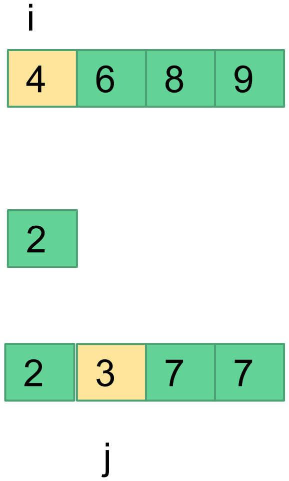


Merge sort: implementation

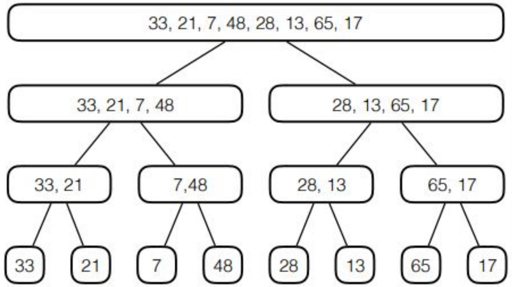


The Merge method:

gets two sorted lists and produces a sorted list with all the elements. Builds the return list by getting the minimum element of the two lists, “removing” it from the corresponding list and appending it to the list with the result (using two indexes pointing to the minimum of each list).

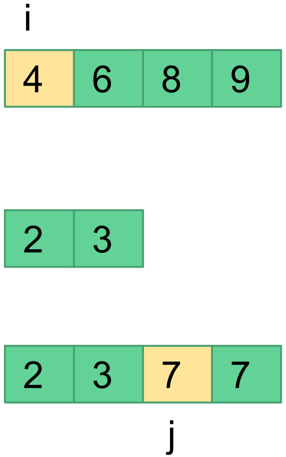


Merge sort: implementation

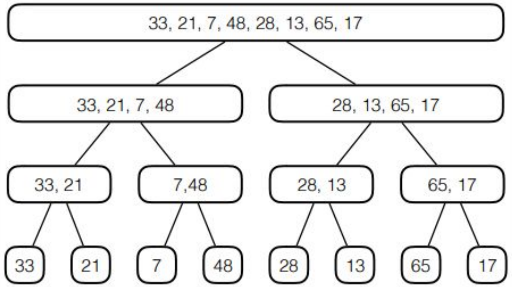


The Merge method:

gets two sorted lists and produces a sorted list with all the elements. Builds the return list by getting the minimum element of the two lists, “removing” it from the corresponding list and appending it to the list with the result (using two indexes pointing to the minimum of each list).

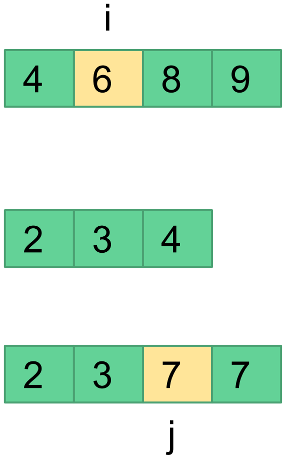


Merge sort: implementation

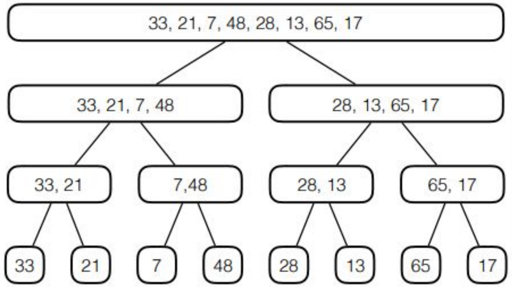


The Merge method:

gets two sorted lists and produces a sorted list with all the elements. Builds the return list by getting the minimum element of the two lists, “removing” it from the corresponding list and appending it to the list with the result (using two indexes pointing to the minimum of each list).

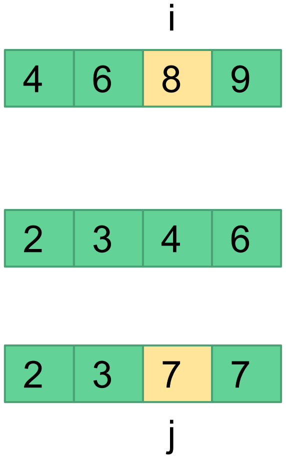


Merge sort: implementation

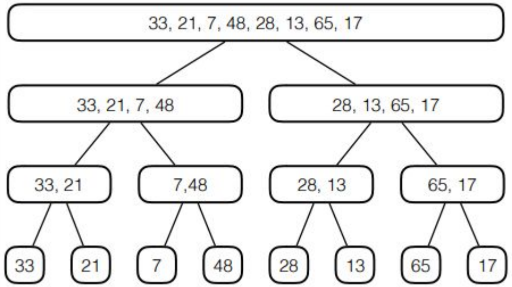


The Merge method:

gets two sorted lists and produces a sorted list with all the elements. Builds the return list by getting the minimum element of the two lists, “removing” it from the corresponding list and appending it to the list with the result (using two indexes pointing to the minimum of each list).

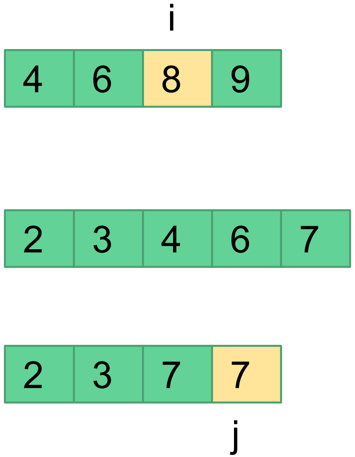


Merge sort: implementation

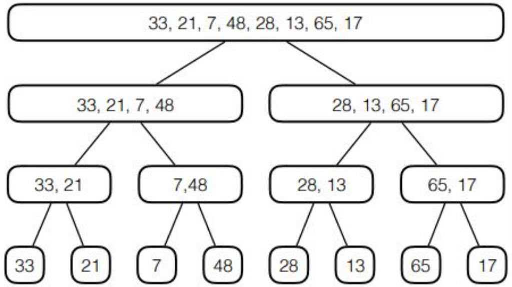


The Merge method:

gets two sorted lists and produces a sorted list with all the elements. Builds the return list by getting the minimum element of the two lists, "removing" it from the corresponding list and appending it to the list with the result (using two indexes pointing to the minimum of each list).

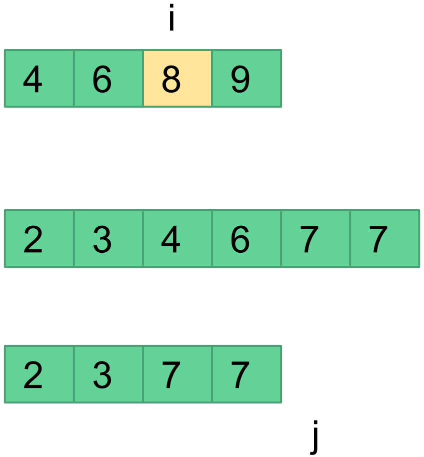


Merge sort: implementation



The Merge method:

gets two sorted lists and produces a sorted list with all the elements. Builds the return list by getting the minimum element of the two lists, “removing” it from the corresponding list and appending it to the list with the result (using two indexes pointing to the minimum of each list).



i

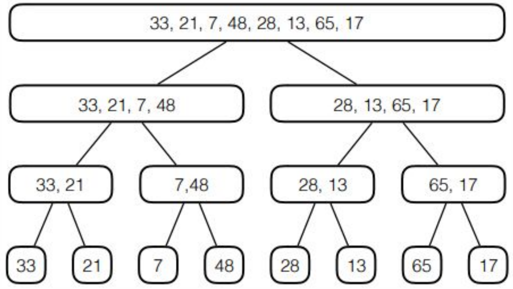
4	6	8	9

2	3	4	6	7	7

2	3	7	7

j

Merge sort: implementation



33, 21, 7, 48, 28, 13, 65, 17

33, 21, 7, 48

28, 13, 65, 17

33, 21

7,48

28, 13

65, 17

33

21

7

48

28

13

65

17

The Merge method:

gets two sorted lists and produces a sorted list with all the elements. Builds the return list by getting the minimum element of the two lists, “removing” it from the corresponding list and appending it to the list with the result (using two indexes pointing to the minimum of each list).

i

2	3	4	6	7	7	8	9

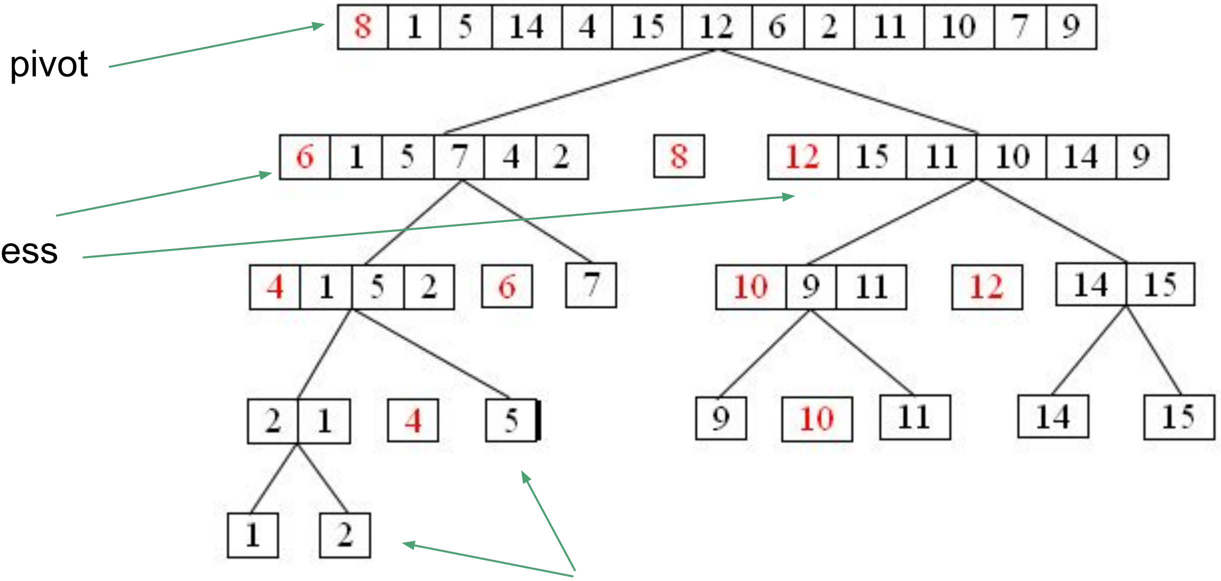
j

Quick sort

The *divide and conquer* approach is the following:

1. (divide) **Choose a pivot element u_j and partition the initial list $U=u_1,..,u_n$ in two non-empty sublists** (reordering the elements) such that all the elements in the first sublist are lower than the elements in the second. The pivot element **uj is such that all the elements u_i for $1 \leq i < j$ are lower than u_j** and **all u_k $j < k \leq n$ are higher than u_j**;
2. (conquer) **recursively partition** each sublist again **until single elements are reached**;
3. (recombine) nothing is left to do to recombine the results.

Quick sort

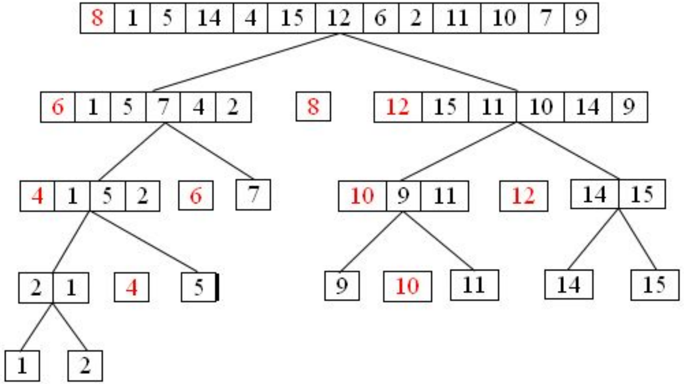


pivot

recursively process this sublists

single elements

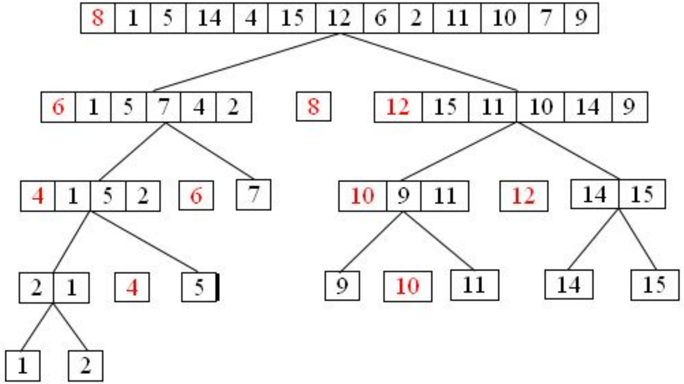
Quick sort



The algorithm makes use of the following methods:

1. (`pivot`) : gets the list, a `start` and `end` index, sets the **first element** as **pivot** and reorders all the elements in the list from `start` to `end` in such a way that a**ll the elements to the left of the pivot (i.e. having index lower) are smaller than the pivot and all the elements to the right (i.e. with index higher) are bigger than the pivot**. The function **returns the index of the pivot**;
2. (`swap`): gets two indexes and swaps their values;
3. (`recursiveQuickSort`): **gets an unordered (sub)list**, with `start` and `end` positions, **finds the pivot** and **recursively applies the same procedure to the sublists to the left and right of the pivot** (if sublist has size > 1);
4. (`quickSort`): **gets an unordered list and applies the recursive quick sort procedure to it**.

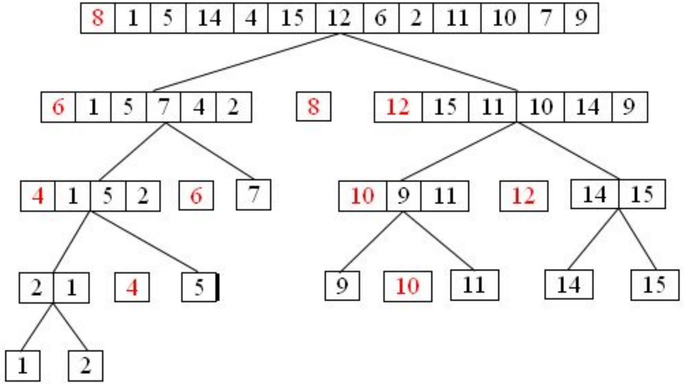
Pivot method



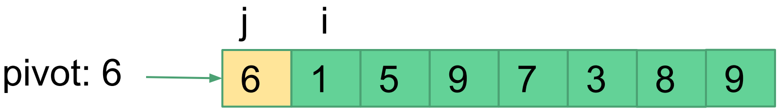
Pivot partitions the list in two: lower than 6 and higher than 6
Pivot called on this list:

6	1	5	9	7	3	8	9

Pivot method



Pivot called on this list:

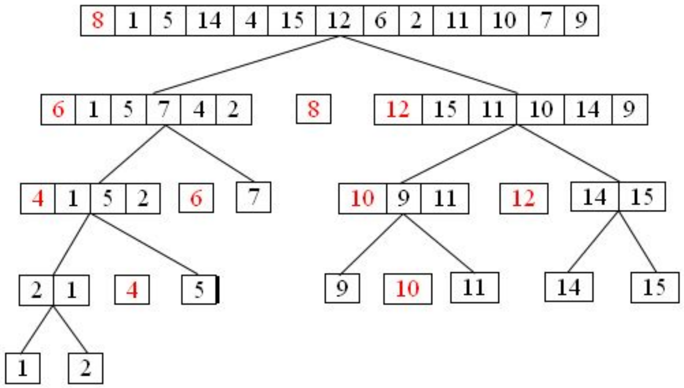


j i

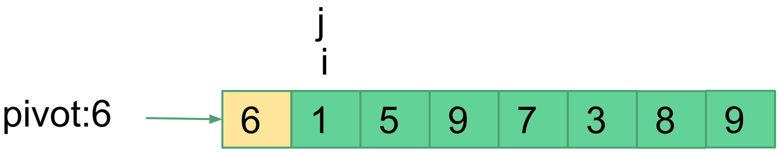
pivot: 6

6	1	5	9	7	3	8	9

Pivot method

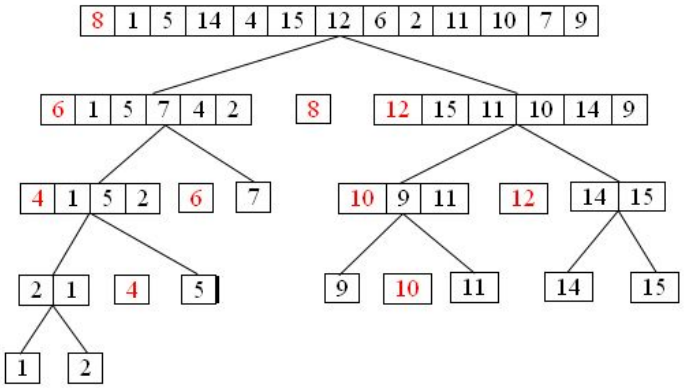


Pivot called on this list:

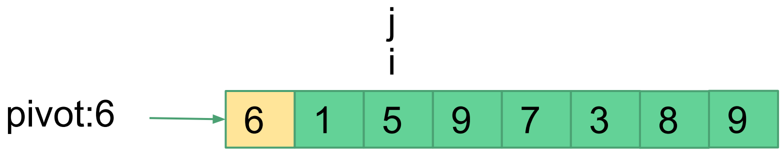


6 > 1
increment j
swap (i,j)

Pivot method

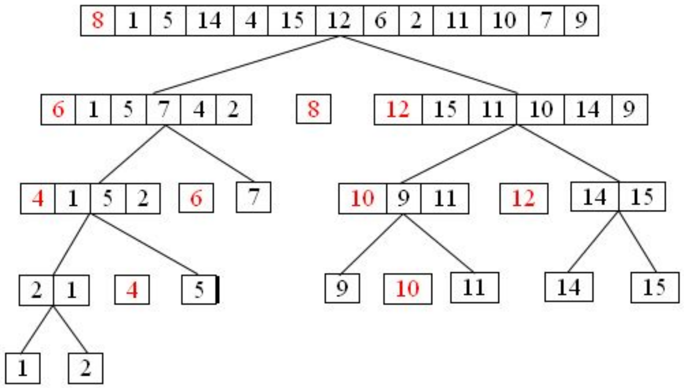


Pivot called on this list:



6 > 5
increment j
swap (i,j)

Pivot method



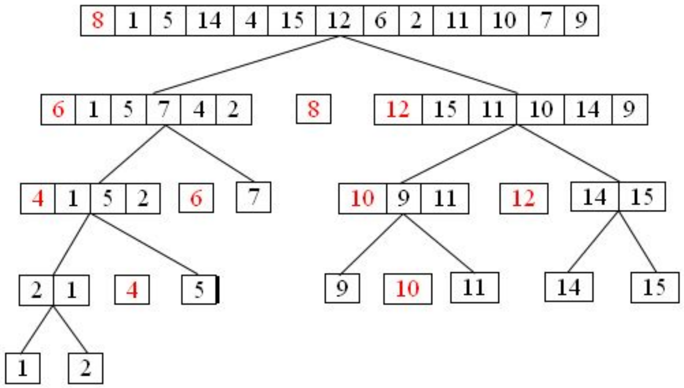
Pivot called on this list:

pivot:6 →

		j	i				
6	1	5	9	7	3	8	9

6 < 9
do nothing

Pivot method



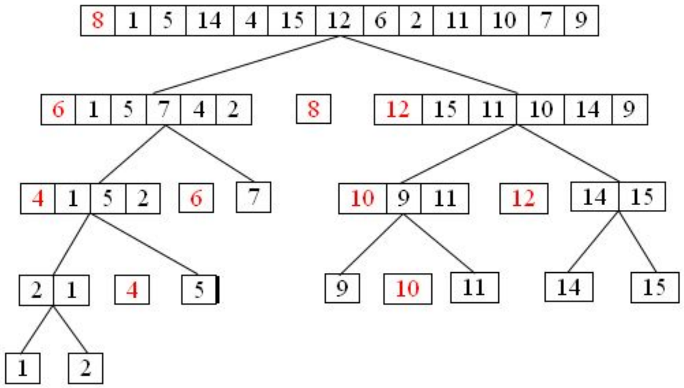
Pivot called on this list:

pivot:6 →

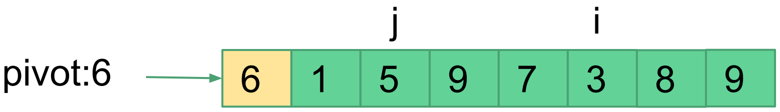
		j		i			
6	1	5	9	7	3	8	9

6 < 7
do nothing

Pivot method



Pivot called on this list:



j i

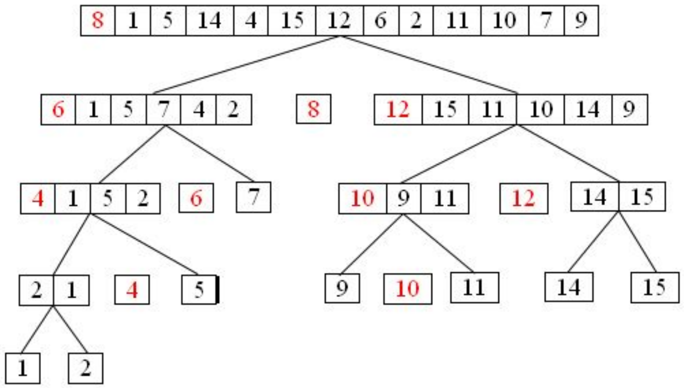
pivot:6

6	1	5	9	7	3	8	9

6 > 3
increment j

j always points to the last
element smaller than pivot

Pivot method



Pivot called on this list:

pivot:6 →

			j		i		
6	1	5	9	7	3	8	9

6 > 3
increment j
swap i,j

			j		i		
6	1	5	3	7	9	8	9

j always points to the last element smaller than pivot

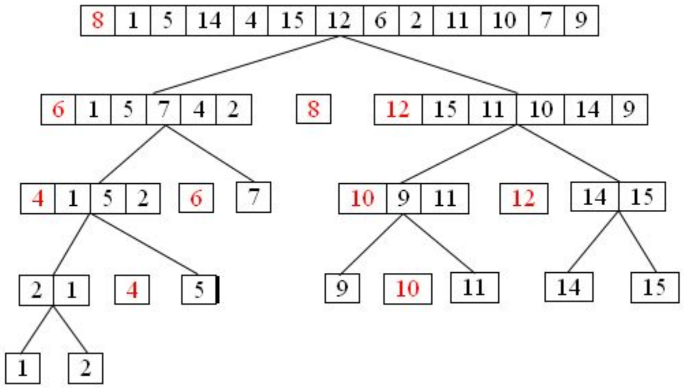
Pivot method

Pivot called on this list:

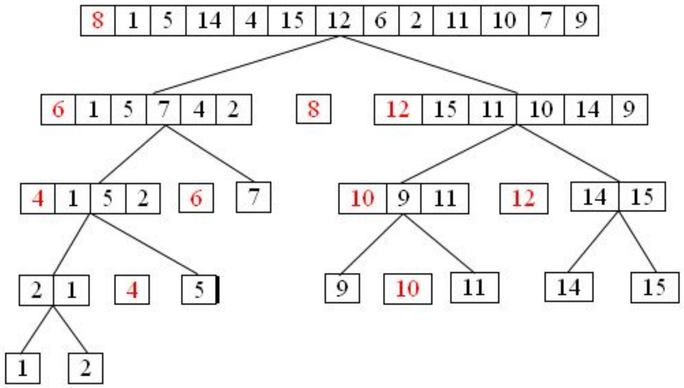
pivot:6 →

			j		i		
6	1	5	3	7	9	8	9

6 < 9
do nothing



Pivot method



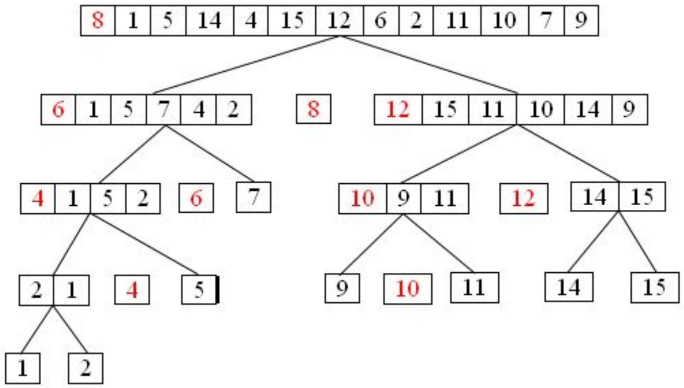
Pivot called on this list:

pivot:6 →

			j			i	
6	1	5	3	7	9	8	9

6 < 8
do nothing

Pivot method



Pivot called on this list:

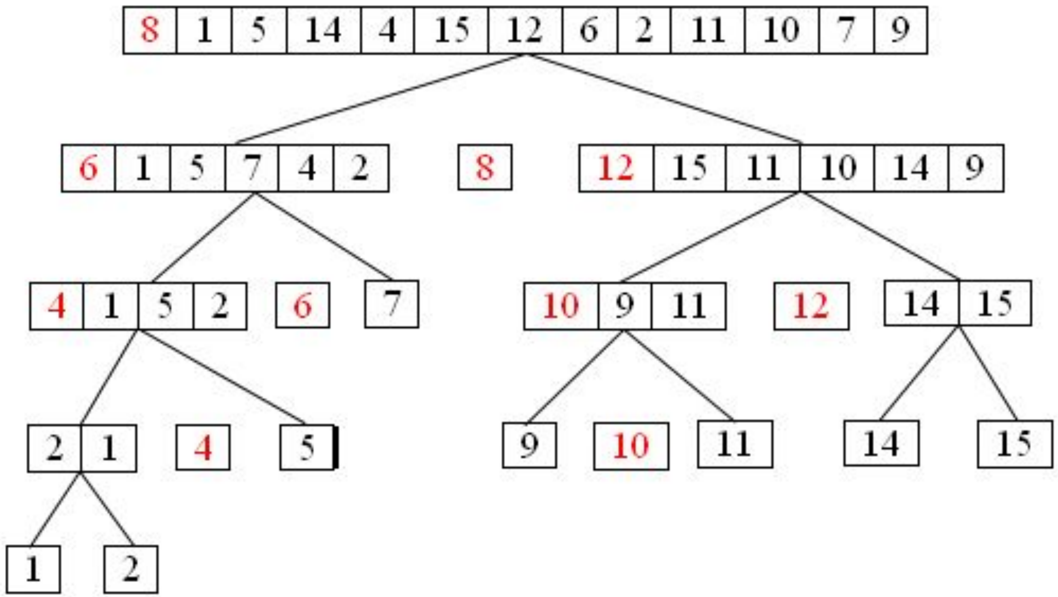
pivot:6 →

			j				i
6	1	5	3	7	9	8	9

6 < 9
do nothing
END!
swap 0,j
return j

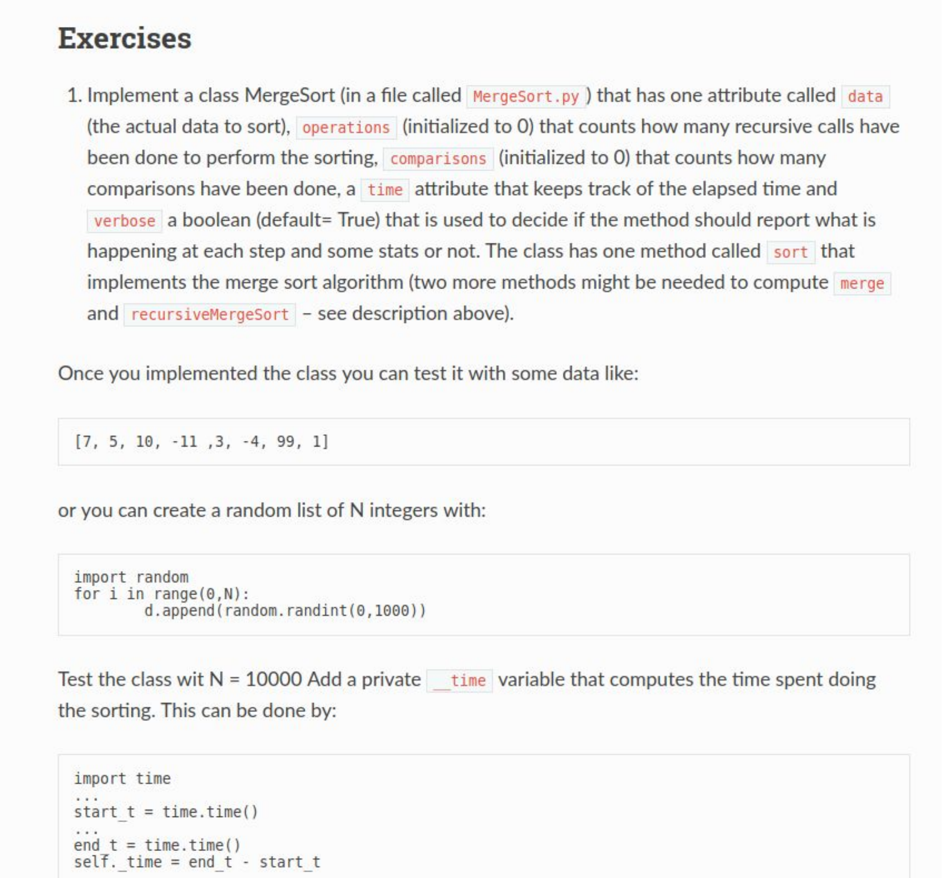
3	1	5	6	7	9	8	9

Quick sort



The **average case complexity** of the quick sort algorithm is ***O*(*nlogn*)** with ***n*** number of elements in the list. The **worst case complexit**y is *O*(*n*^2) **which is worse than merge sort's *O*(*nlogn*)**, but in general it performs better than mergesort.

http://qcbprolab.readthedocs.io/en/latest/practical15.html



Exercises

1. Implement a class MergeSort (in a file called `MergeSort.py`) that has one attribute called `data` (the actual data to sort), `operations` (initialized to 0) that counts how many recursive calls have been done to perform the sorting, `comparisons` (initialized to 0) that counts how many comparisons have been done, a `time` attribute that keeps track of the elapsed time and `verbose` a boolean (default= True) that is used to decide if the method should report what is happening at each step and some stats or not. The class has one method called `sort` that implements the merge sort algorithm (two more methods might be needed to compute `merge` and `recursiveMergeSort` – see description above).

Once you implemented the class you can test it with some data like:

```
[7, 5, 10, -11 ,3, -4, 99, 1]
```

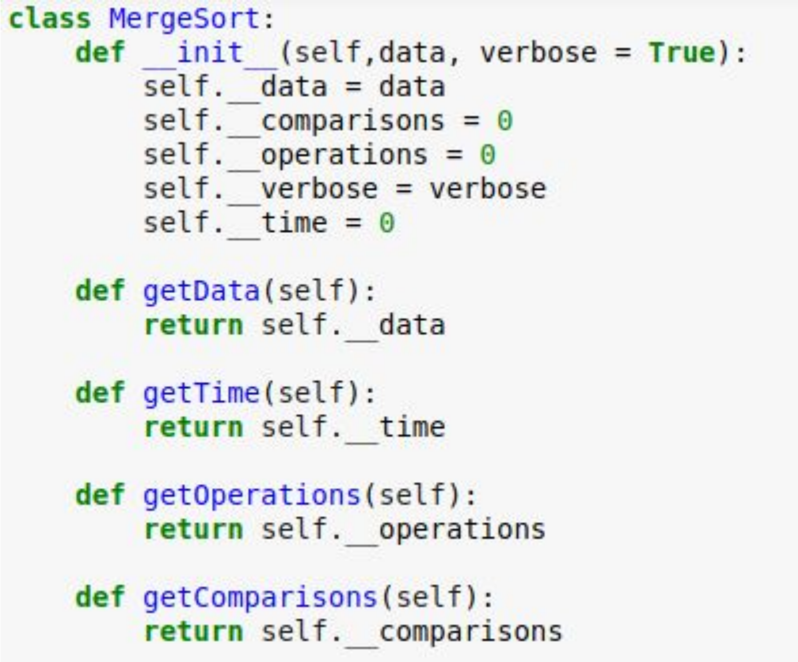
or you can create a random list of N integers with:

```
import random
for i in range(0,N):
        d.append(random.randint(0,1000))
```

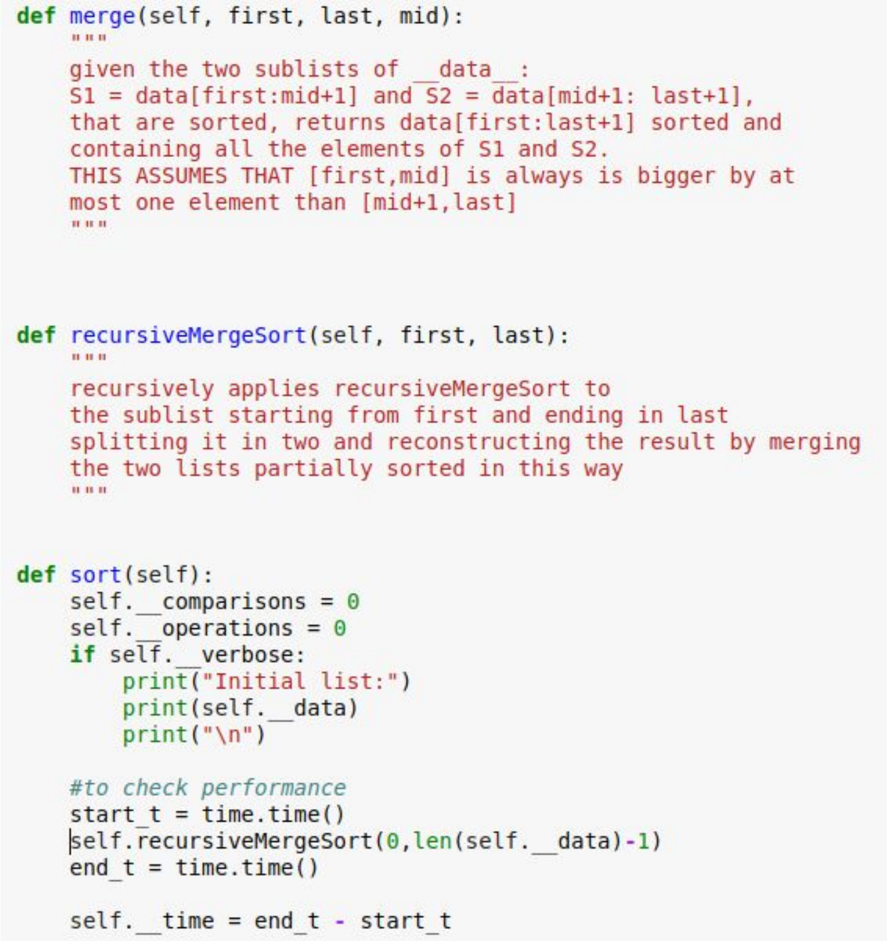
Test the class wit N = 10000 Add a private `__time` variable that computes the time spent doing the sorting. This can be done by:

```
import time
...
start_t = time.time()
...
end_t = time.time()
self._time = end_t - start_t
```

MergeSort class

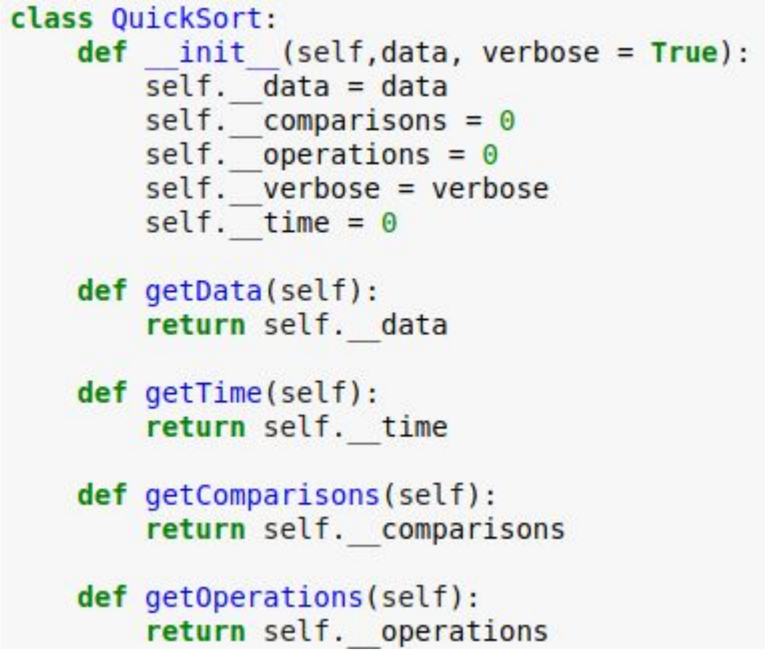


```python
class MergeSort:
    def __init__(self,data, verbose = True):
        self.__data = data
        self.__comparisons = 0
        self.__operations = 0
        self.__verbose = verbose
        self.__time = 0

    def getData(self):
        return self.__data

    def getTime(self):
        return self.__time

    def getOperations(self):
        return self.__operations

    def getComparisons(self):
        return self.__comparisons
```

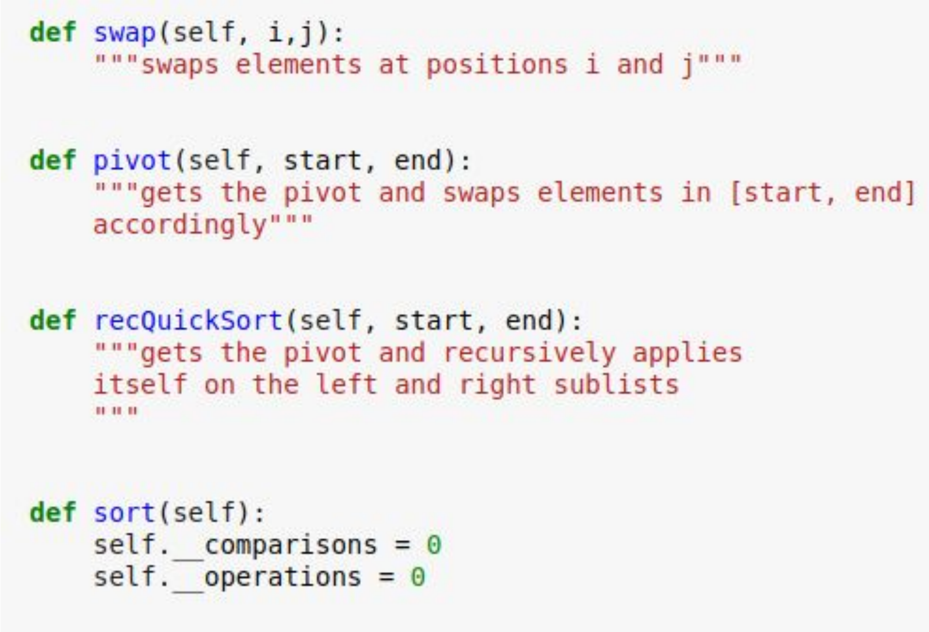


```python
def merge(self, first, last, mid):
    """
    given the two sublists of __data__:
    S1 = data[first:mid+1] and S2 = data[mid+1: last+1],
    that are sorted, returns data[first:last+1] sorted and
    containing all the elements of S1 and S2.
    THIS ASSUMES THAT [first,mid] is always is bigger by at
    most one element than [mid+1,last]
    """



def recursiveMergeSort(self, first, last):
    """
    recursively applies recursiveMergeSort to
    the sublist starting from first and ending in last
    splitting it in two and reconstructing the result by merging
    the two lists partially sorted in this way
    """


def sort(self):
    self.__comparisons = 0
    self.__operations = 0
    if self.__verbose:
        print("Initial list:")
        print(self.__data)
        print("\n")

    #to check performance
    start_t = time.time()
    self.recursiveMergeSort(0,len(self.__data)-1)
    end_t = time.time()

    self.__time = end_t - start_t
```

QuickSort class



```python
class QuickSort:
    def __init__(self,data, verbose = True):
        self.__data = data
        self.__comparisons = 0
        self.__operations = 0
        self.__verbose = verbose
        self.__time = 0

    def getData(self):
        return self.__data

    def getTime(self):
        return self.__time

    def getComparisons(self):
        return self.__comparisons

    def getOperations(self):
        return self.__operations
```



```python
def swap(self, i,j):
    """swaps elements at positions i and j"""


def pivot(self, start, end):
    """gets the pivot and swaps elements in [start, end]
    accordingly"""


def recQuickSort(self, start, end):
    """gets the pivot and recursively applies
    itself on the left and right sublists
    """


def sort(self):
    self.__comparisons = 0
    self.__operations = 0
```