1 Quicksort

1.1 Intuition and Description

Quicksort is a recursive sorting algorithm. Informally speaking, quicksort first chooses a pivot element (in our case, the last element of the list[1] is chosen as the pivot[2]), then compares every element of the given list to the pivot placing elements smaller than the pivot to the left and every other element to the right. This partitions the list into two. The initial pivot is placed between the two partitions. Note that the pivot is correctly placed since every element smaller than the pivot are in the left partition. Then, the quicksort algorithm is applied to both partitions. The recursion is broken if the current partition only contains zero or one elements.

More formally, we present the pseudocode for the procedure.

```
1  def partition(_partition, _low, _high):
2      i = _low - 1
3
4      # here, we choose the pivot as the far right element of the
5      # partition
6      pivot = _partition[_high]
7
8      # from line 12 to line 17 we move every element smaller than
9      # the pivot to the left and every other element to the right
10     # then we place the pivot in the middle of the two partitions
11     for j in range(_low, _high):
12         if _partition[j] < pivot:
13             ++i
14             _partition[i], _partition[j] = _partition[j],
15                                            _partition[i]
16     ++i
17     _partition[i], _partition[j] = _partition[j], _partition[i]
18
19     return i
20
21 def sort_range(_partition, _low, _high):
22     if _low < _high:
23         pivot_index = partition(_partition)
24
25         if pivot_index > 0:
26             sort_range(_partition, _low, pivot_index - 1)
27         sort_range(_partition, pivot_index + 1, _high)
28
29
```

[1] For the purpose of this paper, we define a list to be an ordered set for which an order relation such as $<$ is defined. In terms of computer science, this equates to an array-like data type (in Python this would be a list) with elements which can be compared. An concrete example would be $(0, 1, 2, 3)$ which is incidentally already sorted according to the smaller relation, $<$.

[2] There are more sophisticated ways to choose the pivot. See section XXX for more information.

```
30 # entry point of the algorithm
31 def quicksort(_list):
32     list_length = len(_list)
33     sort_range(_list, 0, list_length - 1)
```

1.2 Complexity

The complexity of quicksort highly depends on the choice of the pivot. We have set the pivot naïvely to be the last element in the partition which is not optimal. In general, a median pivot is the most desireble since it splits the list into two partitions with, concerning complexity, equal lengths.

The worst case for quicksort is when every recursion creates a partition of maximum length. This results in the complexity of $O(n^2)$ (same as selection sort). While a very rare case, this happens most interestingly when the list is already sorted due to the nature how we pick our pivot.

By selecting a pivot in the center half of the list, that is the pivot is larger than the 1/4 smallest elements, but smaller than the 1/4 of the largest elements, one can achieve with some luck the best case complexity of $O(n \ln(n))$.[**?**, p. 137]