1 Quicksort

1.1 Intuition and Description

Quicksort is a divide-and-conquere, recursive sorting algorithm.[**?**, p. 145] 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.

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

1.2 Worked Example

To demonstrate quicksort concretely, we will apply the algorithm on the list

$$(7, 1, 5, 4, 9, 2, 8, 3, 0, 6).$$

Color Key

- partitions to be sorted in the following steps are marked with orange
- partitions currently ignored are colored in gray
- pivots are marked with teal
- elements which were swapped are in red
- finally, previous pivot element placed correctly after the partitioning are indicated with

7	1	5	4	9	2	8	3	0	6	(0) the initial state
7	1	5	4	9	2	8	3	0	6	(1) choose pivot
1	7	5	4	9	2	8	3	0	6	(2) swap 7 and 1
1	5	7	4	9	2	8	3	0	6	(3) swap 7 and 5
1	5	4	7	9	2	8	3	0	6	(4) swap 7 and 4
1	5	4	2	9	7	8	3	0	6	(5) swap 7 and 2
1	5	4	2	3	7	8	9	0	6	(6) swap 9 and 3
1	5	4	2	3	0	8	9	7	6	(7) swap 7 and 0
1	5	4	2	3	0	6	9	7	8	(8) swap 8 and the pivot
1	5	4	2	3	0	6	9	7	8	(9) 6 is in the correct place
partition the sequence into $(1, 5, 4, 2, 3, 0)$ and $(9, 7, 8)$										
1	5	4	2	3	0	6	9	7	8	(10) sort left side
1	5	4	2	3	0	6	9	7	8	(11) choose pivot
0	5	4	2	3	1	6	9	7	8	(12) swap 1 and the pivot
0	5	4	2	3	1	6	9	7	8	(13) 0 is in the correct place
partition the sequence into $()$ and $(5, 4, 2, 3, 1)$										
0	5	4	2	3	1	6	9	7	8	(14) nothing to sort on the left side
0	5	4	2	3	1	6	9	7	8	(15) sort right side
0	5	4	2	3	1	6	9	7	8	(16) choose pivot
0	1	4	2	3	5	6	9	7	8	(17) swap 5 and the pivot

0	1	4	2	3	5	6	9	7	8	(18) 1 is in the correct place
partition the sequence into () and $(4, 2, 3, 5)$										
0	1	4	2	3	5	6	9	7	8	(19) nothing to sort on the left side
0	1	4	2	3	5	6	9	7	8	(20) sort right side
0	1	4	2	3	5	6	9	7	8	(21) choose pivot
0	1	4	2	3	5	6	9	7	8	(22) 5 is in the correct place
partition the sequence into $(4, 2, 3)$ and ()										
0	1	4	2	3	5	6	9	7	8	(23) sort left side
0	1	4	2	3	5	6	9	7	8	(24) choose pivot
0	1	2	4	3	5	6	9	7	8	(25) swap 4 and 2
0	1	2	3	4	5	6	9	7	8	(26) swap 4 and the pivot
0	1	2	3	4	5	6	9	7	8	(27) 3 is in the correct place
0	1	2	3	4	5	6	9	7	8	(28) nothing to sort on the right side
partition the sequence into (2) and (4)										
0	1	2	3	4	5	6	9	7	8	(27) sort left side
0	1	2	3	4	5	6	9	7	8	(28) 2 is in the correct place
0	1	2	3	4	5	6	9	7	8	(29) sort right side
0	1	2	3	4	5	6	9	7	8	(30) 4 is in the correct place
0	1	2	3	4	5	6	9	7	8	(31) sort right side
0	1	2	3	4	5	6	9	7	8	(32) choose pivot
0	1	2	3	4	5	6	7	9	8	(33) swap 9 and 7
0	1	2	3	4	5	6	7	8	9	(34) swap 9 and pivot
0	1	2	3	4	5	6	7	8	9	(35) 8 is in the correct place
partition the sequence into (7) and (9)										
0	1	2	3	4	5	6	7	8	9	(36) sort left side
0	1	2	3	4	5	6	7	8	9	(37) 7 is in the correct place
0	1	2	3	4	5	6	7	8	9	(38) sort left side
0	1	2	3	4	5	6	7	8	9	(39) 9 is in the correct place
0	1	2	3	4	5	6	7	8	9	(40) every thing is correctly sorted

Table 1: An example of quicksort applied to the sequence $(7, 1, 5, 4, 9, 2, 8, 3, 0, 6)$. Note that the table above is presented purely to illustrate the procedure of the algorithm and may not reflect one-to-one its implementation on a computer. For example, before swapping two numbers, the algorithm needs to compare each number leading up to that number to the pivot which was skipped in the table to improve readability. The numbers in the parentheses in the most right columns are also merely for referencing a specific row and do not correlate with the number of steps the algorithm needs to sort the given sequence.

We start with a sequence $(7, 1, 5, 4, 9, 2, 8, 3, 0, 6)$ which has ten distinct elements from 0 to 9. The far right element, 6, is chosen as the pivot (row 1). At the same time, define a counter i and set it to -1. Then, start comparing each element from the left to right to the pivot. If the element is larger (or equal) than the pivot, nothing happens, but if it is smaller than the pivot, increment i by one and swap the number that was compared to the pivot with the number on i-th place of the sequence. For example, $7 > 6$ hence nothing is changed, but $1 < 6$ therefore, i is set to 0 and 5 is swapped with the number on the 0th place which is 7 (row 2). The next number 5 is also smaller than the pivot 6, therefore, i is increment to 1 and 5 is swapped with the number on the first place which is again 7. This procedure is done for each number (compare rows 3 to 7). Finally, the pivot is swapped with the number on the i-th place. In our case, i is 6 at the end and the initial pivot 6 is correctly placed after the swap (see rows 8 and 9).

After placing the initial pivot correctly, the sequence is partitioned into the left and the right side of the pivot which only contain numbers smaller or larger (or equal) than the pivot respectively i.e. the two partitions are $(1, 5, 4, 2, 3, 0)$ and $(9, 7, 8)$ with the first partition containing only numbers smaller than the pivot. Now, quicksort which is a recursive algorithm is applied to both partitions. For example, in the left partition, 0 is chosen as the pivot (row 11).

There are few interesting points. On row 14, the first partition is empty, therefore nothing is sorted. Few rows after in row 22, we bluntly wrote that 5 is in the correct place, but we've skipped multiple steps before. In actuallity, because every number in the partition $(4, 2, 3, 5)$ are smaller (or equal) to the pivot 5 they are all swapped with themselves i.e. 4 is swapped with 4 and 2 is swapped with 2 and so on.

1.3 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 most possible even partitions.

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.[**?**, p. 137]

If every split produces partition with equal length (or length differing exactly by one), the algorithm achives the best case performance of $O(n \ln(n))$. The average case of quicksort is closer to the best case than to the worst case. Indeed, the average case running time of quicksort is again $O(n \ln(n))$.[**?**, p. 150]

Quicksort is not stable. We will show this with an example. Consider a list $(2, 2^*, 1)$. After choosing 1 as the initial pivot, the list is sorted almost immediately into $(1, 2^*, 2)$. 2 and 2^* do not retain their order, hence quicksort is not stable.