Humboldt University of Berlin

Einführung in das wissenschaftliche Rechnen

Analysis of selected sorting Algorithms

Christian Parpart & Kei Thoma

June 25, 2019

Contents

1 Introduction

Hmm, difficult. VERY difficult.

The Sorting Hat

Whoever played card games intensively knows that shuffling is not as trivial as some might think at first glance. By nature, many card games require the players to sort the cards e.g. in old maid one sorts for pairs with the same rank, in skat the cards in trick are sorted. Therefore, good shuffling techniques are necessary to avoid clumped up cards from the previous game. However, even more challenging than shuffling is sorting cards back to their initial order. It is already tedious for humans to order playing cards where the correct order is already known. So one can imagine the difficulties computers must have to sort a large set efficiently. In this paper, we describe and analyze theoretically and experimentally two sorting algorithms, quicksort and heapsort.[1]

2 Quicksort

2.1 Intuition and Description

Quicksort is a divide-and-conquer, recursive sorting algorithm.[1, p. 145] Informally speaking, quicksort first chooses a pivot element (in our case, the last element of the list[2] is chosen as the pivot[3]), 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.

```
def partition(_partition, _low, _high):
    i = _low - 1

    # here, we choose the pivot as the far right element of the
    # partition
```

[1] The quote in the above epigraph was taken from *Harry Potter and the Philosopher's Stone* by J.K. Rowling

[2] 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, $<$.

[3] There are more sophisticated ways to choose the pivot. See section 7 for more information.

```
    pivot = _partition[_high]

    # from line 12 to line 17 we move every element smaller than
    # the pivot to the left and every other element to the right
    # then we place the pivot in the middle of the two partitions
    for j in range(_low, _high):
        if _partition[j] < pivot:
            ++i
            _partition[i], _partition[j] = _partition[j],
                                           _partition[i]
    ++i
    _partition[i], _partition[j] = _partition[j], _partition[i]

    return i

def sort_range(_partition, _low, _high):
    if _low < _high:
        pivot_index = partition(_partition)

        if pivot_index > 0:
            sort_range(_partition, _low, pivot_index - 1)
        sort_range(_partition, pivot_index + 1, _high)


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

2.2 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 green

---

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.

2.3 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.[2, 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 $O(n \ln(n))$.[1, 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.

3 Heapsort

3.1 Intuition and Description

Heapsort is a sorting algorithm which introduces a data structure, a binary tree (the *heap*). A heap is a nearly complete binary tree. For example, consider the list from previous examples

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

This list can be arranged into a heap simply by placing the first element of the list at the root, then placing the next two elements as its children and so on (see figure 1).

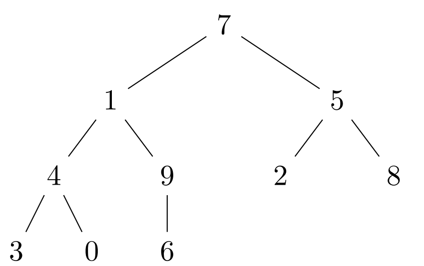


Figure 1: the given list arranged into a heap

In essence, the goal of heapsort is to first sort the heap into a *max-heap* where each parent is larger than each of its children. Then, the root element which by necessity must be the largest element is removed from the heap and the element on the lowest branch (in figure 1 where 6 currently is) is moved to the root (this procedure as a function is called *build-max-heap*).

It turns out however, that not every recursion needs to check if the heap is a max-heap. After sorting the initial heap, one can assume that the heap is already sorted except for the root. The function which partially sorts the heap after the largest element was removed and the element of the lowest branch is moved to the top is called *heapify*.[1, p. 135]

As we did with quicksort, we present the pseudocode for heapsort in the following.

```
def heapsort(_list):
    def heapify(_list, _n, _i):
        largest_element_index = _i

        LEFT_CHILD_INDEX = 2 * _i + 1
        if LEFT_CHILD_INDEX < _n:
            if (_list[LEFT_CHILD_INDEX] >
                _list[largest_element_index]):
                largest_element_index = LEFT_CHILD_INDEX

        RIGHT_CHILD_INDEX = 2 * _i + 2
        IF RIGHT_CHILD_INDEX < _n:
            if (_list[RIGHT_CHILD_INDEX] >
                _list[largest_element_index]):
                largest_element_index = RIGHT_CHILD_INDEX

        if largest_element_index != _i:
            _list[_i], _list[largest_element_index] =
              _list[largest_element_index], _list[_i]

            heapify(_list, _n, largest_element_index)

    i = floor(len(_list) / 2) - 1

    while i >= 0:
        heapify(_list, len(_list), i)
        --i

    i = len(_list) - 1
    while i >= 0:
        _list[0], _list[i] = _list[0], _list[i]
        heapify(_list, i, 0)
        --i
```

3.2 Worked Example

As before, consider the list

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

We will sort this list using heapsort in the following (see figure 2). Due to space restrictions, steps between max heaps were condensed into one.

Color Key

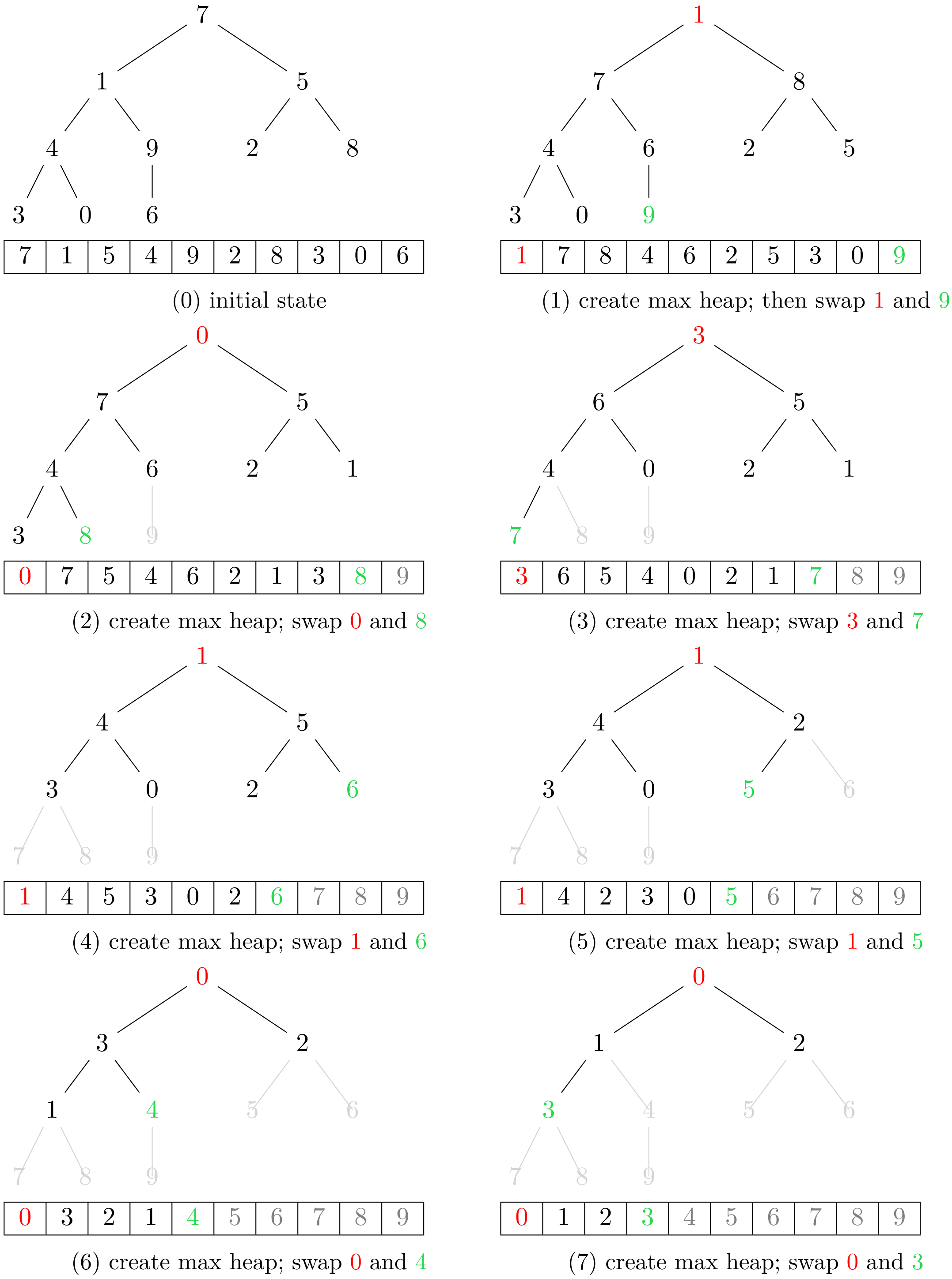
- the numbers swapped in the last step are in red and green, where green indicates that the number was previously the root of the heap
- numbers grayed out are the numbers which are correctly sorted (removed from the heap)

3.3 Complexity

The initial build-max-heap takes time $O(n)$ and heapify which takes time $O(\ln(n))$ and is called $n-1$ times. Together, this means that the complexity of heapsort is $O(n\ln(n))$.[1, p. 136]

Heapsort is not stable. Consider the following list $2^*, 1, 2)$. If heapsort is applied, this list is sorted to $2, 1, 2^*$ thus the algorithm is unstable.

Figure 2: Worked example of heapsort.



(0) initial state

(1) create max heap; then swap 1 and 9

(2) create max heap; swap 0 and 8

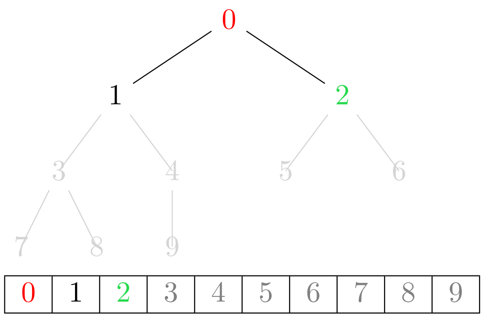
(3) create max heap; swap 3 and 7

(4) create max heap; swap 1 and 6

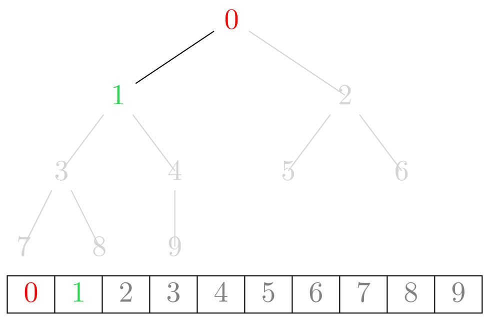
(5) create max heap; swap 1 and 5

(6) create max heap; swap 0 and 4

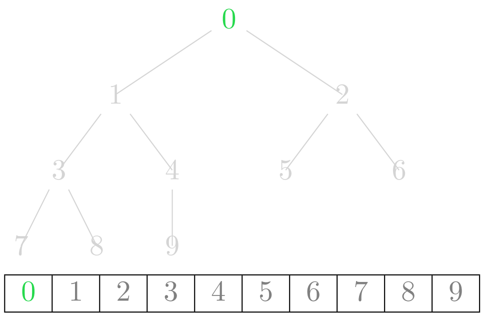
(7) create max heap; swap 0 and 3



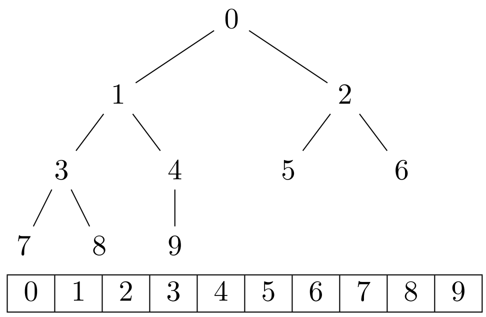
(8) create max heap; swap 0 and 2



(9) create max heap; swap 0 and 1



(10) create max heap; swap 0 and 0



the heap is correctly sorted

4 sorting_analysis.py User Manual

The companion Python module to this paper `sort_analysis.py` was written to verify experimentally the aforementioned theoretical results.

The user can select the desired algorithm by using `-quicksort` or `-heapsort` and specifiying the path to a text file containing words separated by spaces (see figure 6 and 7).

`-latex_trace` outputs a latex file documenting the steps of the algorithms as readable as possible.

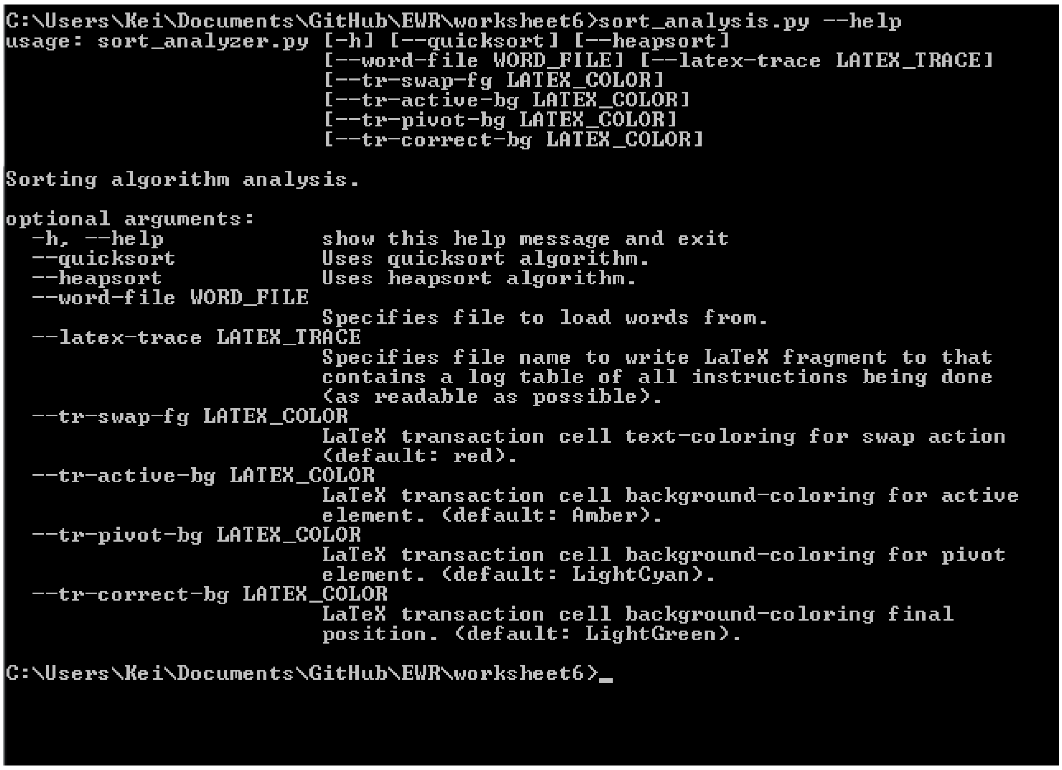
5 sorting_analysis.py API

5.1 sort_A(_list)

These following functions can be imported for custom use.

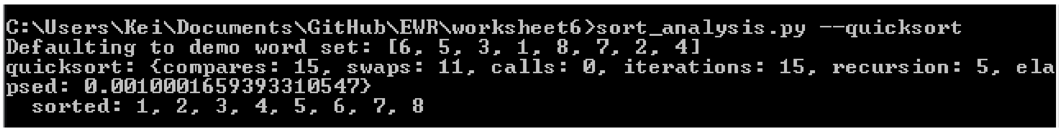
Arguments

1. `_list` (list): the list to be sorted, each element must implement the comparison operator for < (__lt__)



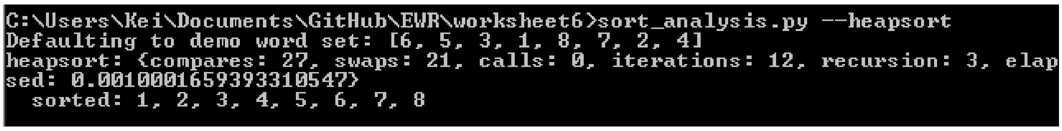
```
C:\Users\Kei\Documents\GitHub\EWR\worksheet6>sort_analysis.py --help
usage: sort_analyzer.py [-h] [--quicksort] [--heapsort]
                        [--word-file WORD_FILE] [--latex-trace LATEX_TRACE]
                        [--tr-swap-fg LATEX_COLOR]
                        [--tr-active-bg LATEX_COLOR]
                        [--tr-pivot-bg LATEX_COLOR]
                        [--tr-correct-bg LATEX_COLOR]

Sorting algorithm analysis.

optional arguments:
  -h, --help            show this help message and exit
  --quicksort           Uses quicksort algorithm.
  --heapsort            Uses heapsort algorithm.
  --word-file WORD_FILE
                        Specifies file to load words from.
  --latex-trace LATEX_TRACE
                        Specifies file name to write LaTeX fragment to that
                        contains a log table of all instructions being done
                        (as readable as possible).
  --tr-swap-fg LATEX_COLOR
                        LaTeX transaction cell text-coloring for swap action
                        (default: red).
  --tr-active-bg LATEX_COLOR
                        LaTeX transaction cell background-coloring for active
                        element. (default: Amber).
  --tr-pivot-bg LATEX_COLOR
                        LaTeX transaction cell background-coloring for pivot
                        element. (default: LightCyan).
  --tr-correct-bg LATEX_COLOR
                        LaTeX transaction cell background-coloring final
                        position. (default: LightGreen).

C:\Users\Kei\Documents\GitHub\EWR\worksheet6>_
```

Figure 5: Screenshot of the help command.



```
C:\Users\Kei\Documents\GitHub\EWR\worksheet6>sort_analysis.py --quicksort
Defaulting to demo word set: [6, 5, 3, 1, 8, 7, 2, 4]
quicksort: {compares: 15, swaps: 11, calls: 0, iterations: 15, recursion: 5, ela
psed: 0.0010001659393310547}
  sorted: 1, 2, 3, 4, 5, 6, 7, 8
```

Figure 6: Screenshot of quicksort demonstration.



```
C:\Users\Kei\Documents\GitHub\EWR\worksheet6>sort_analysis.py --heapsort
Defaulting to demo word set: [6, 5, 3, 1, 8, 7, 2, 4]
heapsort: {compares: 27, swaps: 21, calls: 0, iterations: 12, recursion: 3, elap
sed: 0.0010001659393310547}
  sorted: 1, 2, 3, 4, 5, 6, 7, 8
```

Figure 7: Screenshot of heapsort demonstration.

Returns

- (int, int): 2-tuple with first value the number of operations (compares + swaps) and second value the time consumed in milliseconds.

Description

Performs quicksort on given parameter _list.

5.2 sort_B(_list)

Arguments

1. `_list` (list): the list to be sorted, each element must implement the comparison operator for < (__lt__)

Returns

- (int, int): 2-tuple with first value the number of operations (compares + swaps) and second value the time consumed in milliseconds.

Description

Performs heapsort on given parameter _list.

5.3 read_words_from_file(_filename)

Arguments

1. `_filename` (str): Path to local file.

Returns

- (list): List of words (as strings).

Description

Opens given filename, reads its full contens, and splits it into words, delimited by whitesapce.

6 Experiments

We will test quicksort and heapsort by sorting 5 lists all containing exactly 100 integers from 1 to 100.

	time elapsed (in s)[4]	compares	swaps	iteration	recursion
Test 1					
quicksort	171.30	1474	1288	1474	34
heapsort	18.49	1031	600	150	7
Test 2					
quicksort	24.00	1022	515	1022	20
heapsort	23.55	962	516	151	7
Test 3					
quicksort	5.75	672	381	647	12
heapsort	18.19	1033	581	150	7
Test 4					
quicksort	3.16	576	343	576	11
heapsort	18.79	1039	588	150	7
Test 5					
quicksort	4.40	749	414	749	16
heapsort	16.23	1004	559	150	7

Figure 8: The results of the python module. For the input lists of the tests, see the appendix.

Test 1 was a list of integers already sorted (1, 2, 3, ..., 99, 100) while the list used in test 2 was reversed sorted (i.e. 100, 99, 98, ..., 2, 1). The other three lists were a list of random integers from 1 to 100 each occuring once. See figure 8 for detailed results.

One of our theoretical results was that the worst case scenario for quicksort is when the split is as unbalanced as possible. This is verified in our first test. Quicksort is much slower at sorting already sorted lists than random ones. However, in general, quicksort is a much more efficient algorithm than heapsort. other observation is that the heapsort's complexity stays relativly the same as stated in our theory (heapsort has complexity $O(n \ln(n))$ for both best and worst cases).

From this, we can gather that while quicksort is often faster than heapsort, heapsort has better performance if quicksort encounters its worst case.

7 Conclusion and the Future

One of the immediate optimization idea for quicksort is to make the choice of the pivot more intelligently. We have seen that with the way our current set up for quicksort lists which are already sorted are one of the worst cases. In practice, having the worst case for already sorted lists is not desireble. Median of three, for example, could be a way of choosing the pivot more smartly which could be implemented in the future.

As we have already mentioned before, the strenth of heapsort lies in its consistent performance while the weakness of quicksort is its worst case. Therefore, it would make sense to combine these two algorithms to cover up each of their weaknesses. Indeed, such an algorithm is called introsort which would be the next goal to achive after implementing quicksort and heapsort.

8 References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* The MIT Press, Second Edition, Cambridge, Massachusetts, 2003.

[2] Steven S. Skiena. *The Algorithm Design Manual* Springer, Second Edition, 2008.

9 Appendix

The following five lits of integers were used in section 6, experiments.

Test 1 (sorted)

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100

Test 2 (reversed sorted)

100 99 98 97 96 95 94 93 92 91 90 89 88 87 86 85 84 83 82 81 80 79 78 77 76 75 74 73 72 71 70 69 68 67 66 65 64 63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Test 3 (random)

60 26 76 82 95 59 27 20 42 67 11 57 39 1 45 61 3 65 53 15 49 77 43 99 97 80 48 79 17 93 73 89 64 88 25 75 7 63 21 58 10 9 32 18 29 69 90 96 2 37 4 19 66 16 81 35 30 51 14 33 70 5 94 56 41 44 38 54 23 91 6 84 86 68 13 62 31 71 87 72 12 34 47 50 98 55 78 83 46 85 74 40 22 28 8 36 52 100 92 24

Test 4 (random)

66 12 80 61 19 47 72 10 18 82 21 59 43 76 31 34 81 87 11 84 71 78 5 29 60 24 89 41 13 68 74 79 48 38 25 73 56 62 86 26 33 50 93 69 22 70 27 98 95 64 6 23 97 90 7 96 9 20 8 17 46 15 35 85 94 4 52 100 44 92 3 30 54 49 32 53 88 83 63 91 36 37 77 57 16 55 51 1 14 75 67 2 99 39 28 40 65 45 58 42

Test 5 (random)

93 21 42 55 94 89 82 16 6 67 41 97 30 88 22 75 9 17 3 49 62 80 40 48 63 33 92 86 66 50 84 45 18 31 32 85 24 37 44 83 43 95 51 68 29 81 98 26 100 27 58 76 72 11 23 59 19 46 28 60 53 8 74 69 10 7 1 56 20 54 36 77 39 47 71 87 38 4 78 34 2 96 61 35 57 70 64 15 14 73 5 52 65 79 25 99 12 91 90 13