**Exercise 3.2 (option, no duty)**

The QUICKSORT algorithm of the textbook contains two recursive calls to itself. After the call to PARTITION, the left subarray is recursively sorted and then the right subarray is recursively sorted. The second recursive call in QUICKSORT is not really necessary; it can be avoided by using an iterative control structure. This technique, called ***tail recursion***, is provided automatically by good compilers.   
Consider the following version of quicksort, which simulates tail recursion.

TAIL-RECURSIVE-QUICKSORT(*A*, *p*, *r*)

1 **while** *p* < *r*

2 **do** // Partition and sort left subarray.

3 *q* ← PARTITION(*A*, *p*, *r*)

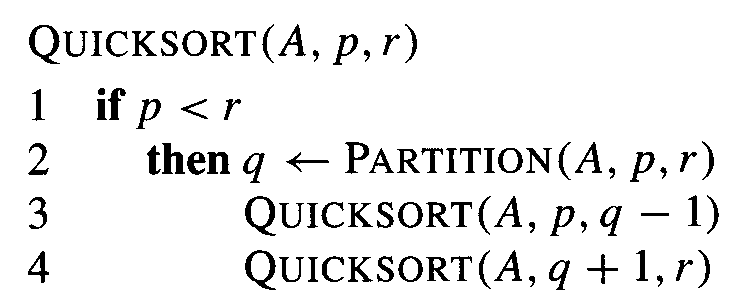
4 TAIL-RECURSIVE-QUICKSORT (*A*, *p*, *q* - 1)

5 *p* ← *q* + 1

**a)** Argue that TAIL-RECURSIVE-QUICKSORT (*A*, 1, length[*A*]) correctly sorts the array *A*.

**Proof:**

Pseudocode of the original algorithm:



The left recursion of the above iterative Quicksort (TAIL-RECURSIVE-QUICKSORT) is equal to the original algorithm. The right recursion is replaced by the iteration. Therefore, we have to check that the iteration correctly implements this right recursion. The right recursive call in the original algorithm looks as follows:

QUICKSORT (*A*, *q + 1*, *r*)

So, in the original algorithm, the right recursive call starts with a parameter *p* that is equal to *q + 1* and *r* stays unchanged. Line 5 in the iterative implementation performs this modification of *p*. *r* requires no modification because it stays unchanged.

In the original algorithm, the partitioning is done if we have *p* < *r* (line 1 of Quicksort Pseudocode). So, in other words, the partitioning continues recursively while we have *p* < *r* . The “**while** *p* < *r*” in line 1 of the iterative Quicksort exactly implements this kind of behavior.

Compilers usually execute recursive procedures by using a ***stack*** that contains pertinent information, including the parameter values, for each recursive call. The information for the most recent call is at the top of the stack, and the information for the initial call is at the bottom. When a procedure is invoked, its information is ***pushed*** onto the stack; when it terminates, its information is ***popped***. Since we assume that array parameters are represented by pointers, the information for each procedure call on the stack requires *O*(1) stack space. The ***stack depth*** is the maximum amount of stack space used at any time during a computation.

**b)** Describe a scenario in which the stack depth of TAIL-RECURSIVE-QUICKSORT is on an -element input array.

The scenario is an ascendingly sorted sequence consisting of elements.

Argumentation:

For an ascendingly sorted sequence, TAIL-RECURSIVE-QUICKSORT behaves exactly as the standard Quicksort. As Quicksort, TAIL-RECURSIVE-QUICKSORT shows its worst case behavior and requires a recursive calls. Each of these recursive calls requires some constant space on the stack. Therefore, summarily, our stack depth becomes proportional to , which is expressed by in asymptotic notation.

**c)** Modify the code for TAIL-RECURSIVE-QUICKSORT so that the worst-case stack depth is . Maintain the expected running time of the algorithm.

The trick is to check the sizes of the sequences that we get after the partitioning and to forward the shorter into the recursion while the longer sequence is handled by the iteration:

As Pseudocode:

TAIL-RECURSIVE-QUICKSORT(*A*, *p*, *r*)

1 **while** *p* < *r*

2 **do**

3 *q* ← PARTITION(*A*, *p*, *r*)  
4 **if** *r - q* > *q – p*

*// Shorter sequence is on the left side*

5 TAIL-RECURSIVE-QUICKSORT (*A*, *p*, *q* - 1)

6 *p* ← *q* + 1

7 **else**

*// Shorter sequence is on the right side*

8 TAIL-RECURSIVE-QUICKSORT (*A*, *q + 1*, *r*)

9 *r* ← *q* - 1

In the above Pseudcode, the size of the shorter sequence is always smaller than . Because the shorter sequence is handled by the reclusive calls, the recursion depth is limited by . Therefore the stack depth is . However, in the case of repeated equal splittings we actually reach this . Therefore the worst-case bound with respect to stack-depth can be considered as “tight” (-notation) (because we have a case, where is actually reached. Please note, this tight bound is only correct for the worst case with respect to stack-depth!) In the best case the stack depth is because for an ascendingly sorted sequence each recursive call terminates immediately.

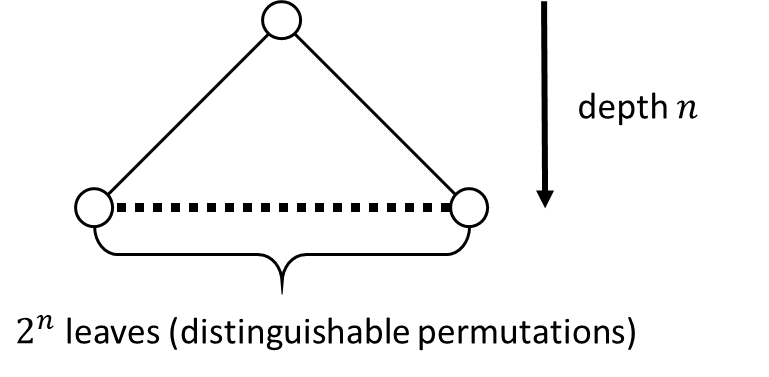
**Exercise 3.3 (option, no duty)**

Show that there is no comparison sort whose running time is linear for at least half of the many inputs of length .

***Tip:*** This is quite easy to prove by using the decision tree.

Proof:

A linear runtime implies a linear number of comparisons. Therefore, we consider the number of permutations (“decision tree leaves”) that we can distinguish if we limit the number of comparisons of a single computation (single path in our decision model) to .



In our decision tree model, we could distinguish different paths and so many permutations. Now we have to consider this many permutations in proportion to the half of the many inputs (permutations) of length . We see that we always have (where is even significantly smaller than ). So, can never cover half of the many inputs. This, in turn, proves our statement that there can be no comparison sort whose running time is linear for at least half of the many inputs of length .