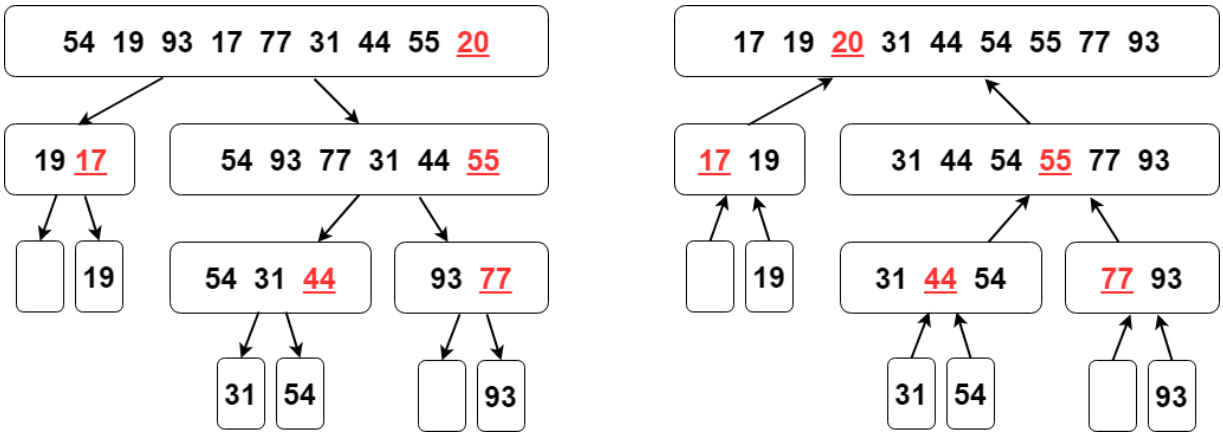
Tutorial (week 10)

All the answers can be given with pseudocode or with Python. Remember that most of the time, many solutions are possible.

1. Consider the following list $L = [54, 19, 93, 17, 77, 31, 44, 55, 20]$. when applying a recursive quick-sort algorithm on L, explain with a binary tree the successive recursive calls and with another binary tree the successive merge processes.

Solution:



2. Show that the best-case running time of quick-sort on a sequence of size n with distinct elements is $O(n \log n)$.

Solution:
Given a quick-sort binary tree T (that represents the execution of the quick-sort), let s_i denote the sum of the input sizes of the nodes at depth i in T. Clearly, $s_0 = n$, since the root r of T is associated with the entire sequence. Also, $s_1 = n - 1$, since the pivot is not propagated to the children of r. Consider next s_2. If both children of r have nonzero input size, then $s_2 = n - 3$. Otherwise (one child of the root has zero size, the other has size $n - 1$), $s_2 = n - 2$. Thus, $n - 3 \leq s_2 \leq n - 2$. Continuing this line of reasoning, we obtain that $n - (2^i - 1) \leq s_i \leq n - i$.

In the best case, each pivot choice will divide the input sequence into two sequences of equal size. Thus, the binary tree representing the execution of the quick-sort will be well balanced, and the height of the tree will be $O(\log n)$. In that case, we have that $s_i = n - (2^i - 1)$.

Thus, the best-case running time of quick-sort is:

$$\sum_{i=0}^{\log n} s_i = \sum_{i=0}^{\log n} n - (2^i - 1) = (n+1)\log n - \sum_{i=0}^{\log n} (2^i - 1) \leq (n+1)\log n \simeq O(n \log n)$$

3. Suppose we modify the deterministic version of the quick-sort algorithm so that, instead of selecting the last element in an n-element sequence as the pivot, we choose the element at index $\lfloor n/2 \rfloor$, that is, an element in the middle of the sequence. What is the running time of this version of quick-sort on a sequence that is already sorted ?

   Solution:
   $O(n \log n)$ time (in contrary to the classical quick-sort where we take the last element as pivot, which runs in $O(n^2)$ when the input sequence is already sorted, which is the worst case). This is because if we choose the pivot to be located at index $\lfloor n/2 \rfloor$, it will split a sorted sequence in half each time, since it is the median. Thus, this is the best possible case: the binary tree representing the execution of the quick-sort will be well balanced, and the height of the tree will be $O(\log n)$. The reasoning for complexity evaluation is then the same as in the previous question.

4. Consider again the modification of the deterministic version of the quick-sort algorithm so that, instead of selecting the last element in an n-element sequence as the pivot, we choose the element at index $\lfloor n/2 \rfloor$. Describe the kind of sequence that would cause this version of quick-sort to run in $\Theta(n^2)$ time.

   Solution:
   The sequence should have the property that the selected pivot is the largest or smallest element in the subsequence. If that is the case, the pivots will never split the sequences at all. The binary tree representing the execution of the quick-sort will be completely unbalanced, and the height of the tree will be $\Theta(n)$. The reasoning for complexity evaluation is then the same as in the previous question, resulting in a complexity of $\Theta(n^2)$.

5. Suppose we are given a sequence S of n elements, on which a total order relation is defined. Describe an efficient method for determining whether there are two equal elements in S. What is the running time of your method ?

   Solution:
   Sort the elements of S, which takes $O(n \log n)$ time. Then, step through the sequence looking for two consecutive elements that are equal, which takes an additional $O(n)$ time. Overall, this method takes $O(n \log n)$ time.

6. Let A and B be two sequences of n integers each. Given an integer x, describe an $O(n \log n)$-time algorithm for determining if there is an integer a in A and an integer b in B such that $x = a + b$.

   Solution:
   Create a sequence C from sequence B, by storing $c = x - b$ for each element b of B. Building this sequence costs $O(n)$. Then, sort both sequences A and C independently, which costs $O(n \log n)$. Finally, simply check if there is an element that belongs to both A and C (this can be done in $O(n)$ by scanning the two sorted lists, similarly to what a merge process is doing in a merge-sort). Assumes that such an element exists, it means that we found a a and c value such that $a = c$. Thus, $a = c = x - b$, which implies that we found a and b values such that $a + b = x$, in which case we return True. If not such pair is found, we return False. The complexity of the overall algorithm is dominated by the sorting of A and C, and thus is $O(n \log n)$.

7. Given a sequence of numbers, $(x_1, x_2, ..., x_n)$, the mode is the value that appears the most number of times in this sequence. Give an efficient algorithm to compute the mode for a sequence of n numbers. What is the running time of your method ?

Solution:
Sort the numbers by nondecreasing values. Next we can scan the sequence to keep track, for each run of numbers that are all the same, how long that sequence is. In addition, we can store the length and x_i-value for the longest sequence we have seen so far. When we complete the scan of the sequence, this x_i value is the mode. The running time for this method is dominated by the time to sort the sequence, which can be done in $O(n \log n)$ time in the worst-case by using the merge-sort algorithm.

8. Develop an algorithm that computes the k-th smallest element in a set of n distinct integers with an average complexity $O(n)$ time.

Solution:
This is known as the quick-select algorithm, as it is quite a similar strategy to the quick-sort. The idea is to select a pivot value (just like for quick-sort we can take the last element of the set, or we can randomize the choice of pivot) and separate the set in two subsets according to the pivot value. Then, by looking at the size of the subsets, one can easily guess in which of them the k-th smallest element is located. Thus, instead of continuing the process recursively for both subsets, one just continues it on the proper subset (and not on the other one).

The expected complexity can be evaluated with a similar reasoning as for quick-sort: at each recurrence, there is probability $1/2$ that the pivot will separate the list into two sublists such that none of them is smaller than $1/4$ (good pivot). One therefore expects 2 recurrences to get at least a good pivot. One can write the following recurrence inequation: $T(n) \leq 2 \cdot c \cdot n + T(3n/4)$, where $c \cdot n$ is the time taken for the partition (c is a constant). Using the master theorem (with $a = 1$ and $b = 4/3$), we have that $T(n)$ is $O(n)$.

```
def partition(my_array):
    '''
    This function will partition the input array into two lists
    according to the pivot (the last element of the input array)
    INPUT: an array of the elements
    OUTPUT: two subarrays created according to the pivot
    '''
    pivot = my_array[-1]  # select the last element as pivot for the partition
    left = right = []

    # we scan all elements of the array and we place them in left or right
    # sublists depending on their relative value to the pivot
    for i in range(len(my_array)-1):
        if my_array[i] < pivot:   # if smaller than the pivot ==> left
            left = left + [my_array[i]]
        else:                     #if greater or equal to the pivot ==> right
            right = right + [my_array[i]]

    return left,right



def quickselect(my_array,k):
    '''
    INPUT: an array of elements, a positive integer k
    OUTPUT: the k-th smallest element of the input array
    '''
    # base case: the array is a single element
    if len(my_array) == 1: return my_array[0]

    # general case
```

```
    else:
        (left,right) = partition(my_array)  # we first divide the array

        # the pivot is the k-th smallest element
        if len(left) == k-1:  return my_array[-1]

        # the k-th smallest element is in the left subarray
        elif len(left) >= k:  return quickselect(left,k)

        # the k-th smallest element is in the right subarray
        else:                 return quickselect(right,k-len(left)-1)
```

Hints

- **Question 3: Consider how the pivots work on a sorted sequence.**
- **Question 4: Review the worst-case performance example for standard quick-sort.**
- **Question 5: Sort first.**
- **Question 8: Consider a strategy similar to the quicksort (using array partition according to a pivot value).**