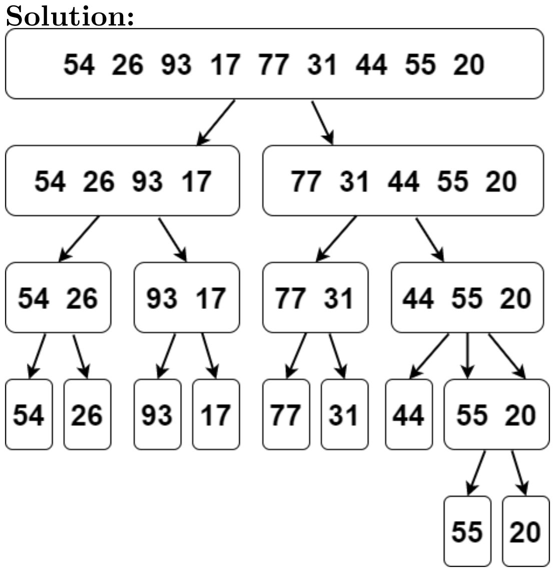
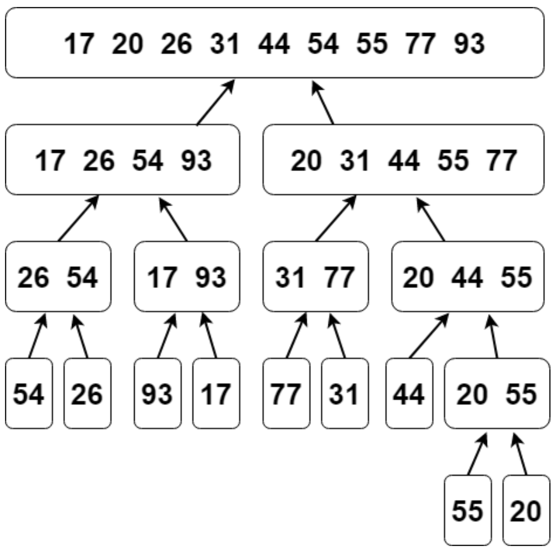
Tutorial (week 8)

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 $L = [54, 26, 93, 17, 77, 31, 44, 55, 20]$. when applying a recursive merge-sort algorithm on L, explain with a binary tree the successive recursive calls and with another binary tree the successive merge processes.

Solution:





2. Give a Python or pseudocode description of the merge-sort algorithm assuming the input is given as a linked list.

Solution:

```python
def merge_sort(L):
    '''
    INPUT: a linked list L
    OUTPUT: the corresponding sorted linked list
    '''

    # if the list is empty or contains only one element
    # the list is already sorted and we return it
    if L.head.next == L.tail or L.head.next == L.tail.prev:
        return L


    # if the list has more than 1 element
    # we first find the middle point of the list by having one pointer
    # that jumps once and another that jumps twice every iteration
    tmp = L.head.next
    middle = L.head.next
    while tmp != L.tail and tmp != None:
        tmp = (tmp.next).next
        middle = middle.next

    # we then create a left list and a right sublist according to the middle
    left_list = LinkedList()
    tmp = L.head.next
    while tmp != middle:
        left_list.append(Node(tmp.val))
        tmp = tmp.next

    right_list = LinkedList()
    tmp = middle
    while tmp != L.tail:
        right_list.append(Node(tmp.val))
        tmp = tmp.next

    # we lauch the sorting recursively on the left and right sublists
    merge_sort(left_list)
    merge_sort(right_list)

    # we empty the list L
    L.head.next = L.tail
    L.tail.prev = L.head

    # we merge the two sorted sublists together in L
    cursor_left = left_list.head.next
    cursor_right = right_list.head.next

    while cursor_left != left_list.tail or cursor_right != right_list.tail:
        if cursor_left == left_list.tail:
            L.append(Node(cursor_right.val))
            cursor_right = cursor_right.next
        elif cursor_right == right_list.tail:
            L.append(Node(cursor_left.val))
            cursor_left = cursor_left.next
        elif cursor_right.val < cursor_left.val:
            L.append(Node(cursor_right.val))
            cursor_right = cursor_right.next
        else:
            L.append(Node(cursor_left.val))
            cursor_left = cursor_left.next

    return L
```

3. Describe a variation of the merge-sort algorithm that is given a single array, S, as input, and uses only an additional array, T, as a workspace. No other memory should be used other than a constant number of variables.

   Solution:
   The previous question's implementation of the merge-sort uses $O(n)$ variables, since each recursive call will create its own variables and there are $O(n)$ recursive calls that are done. Instead, one could use an iterative implementation variant of the merge sort.

```
def merge_sort_single_additional_array(S):
    '''
    INPUT: an array S
    OUTPUT: the correspondin sorted array
    '''

    # we will test successive powers of two for sublist sizes
    # aka we start from 1, then 2, then 4, then 8, etc.
    # and we continue until the sublist size is bigger than the list itself
    sub_list_size = 1
    while sub_list_size < len(S):

        # we make a copy of S in T, so that we can properly doing the merge
        T = S[:]

        # we will test all adjacent sublists of size sub_list_size
        # and merge them together while keeping the property that they are sorted
        for begin in range(0,len(S),2*sub_list_size):

            # The two adjacent sublists are seen as two parts (left and right) each
                of size sub_list_size
            begin = begin
            mid = min(len(S),begin + sub_list_size)
            end = min(len(S),begin + 2*sub_list_size)

            pointer_S = begin            # number of items handled for the two
                adjacent sublists
            pointer_left = begin         # pointer to the current left part item
            pointer_right = mid          # pointer to the current right part item

            # continue the merge as long as we are not finished
            while pointer_left < mid or pointer_right < end:

                # if the left part is over, we take from the right part
                if pointer_left == mid:
                    S[pointer_S] = T[pointer_right]
                    pointer_right += 1
                # if the right part is over, we take from the left part
                elif pointer_right == end:
                    S[pointer_S] = T[pointer_left]
                    pointer_left += 1
                # if right item is smaller than the left item, we take from the right
                     part
                elif T[pointer_left] >= T[pointer_right]:
                    S[pointer_S] = T[pointer_right]
                    pointer_right += 1
                # if left item is smaller than the right item, we take from the left
                    part
                elif T[pointer_left] < T[pointer_right]:
                    S[pointer_S] = T[pointer_left]
                    pointer_left += 1

                pointer_S += 1

        sub_list_size *= 2

    return S
```

4. Let A be a collection of objects. Describe an efficient method for converting A into a set. That is, remove all duplicates from A. What is the running time of this method ?

Solution:
First we sort the objects of A. Then we can walk through the sorted sequence and remove all dupli-

cates. This takes $O(n \log n)$ time to sort and n time to remove the duplicates. Overall, therefore, this is an $O(n \log n)$-time method.

```
def convert_into_set(A):
    '''
    INPUT: a collection A
    OUTPUT: a set A (the duplicates of A are removed)
    '''

    # sort the collection
    A.sort()

    # remove the duplicate elements by going
    # from the end to the beginning (to avoid index issues)
    for i in reversed(range(1,len(A))):
        if A[i] == A[i-1]:
            del A[i]
```

5. Suppose we are given two n-element sorted sequences A and B that should not be viewed as sets (that is, A and B may contain duplicate entries). Describe an $O(n)$-time method for computing a sequence representing the set $A \cup B$ (with no duplicates).

   Solution:
   Merge sequences A and B into a new sequence C (i.e., call $merge(A, B, C)$). Do a linear scan through the sequence C removing all duplicate elements (i.e., if the next element is equal to the current element, remove it).

```
def merge_to_one(A,B):
    '''
    INPUT: two collections A and B
    OUTPUT: the set C = A U B
    '''

    # initialise the output list and the pointers
    C = []
    pointer_A = 0
    pointer_B = 0
    last_added_element = None

    # sort the collections A and B (can be ignored if considered already sorted)
    A.sort()
    B.sort()

    # add all the elements of A and B, like a merge process in a merge-sort,
    # but checking that you don't add twice the same element
    # (we maintain in variable last_added_element the last added element)
    while pointer_A != len(A) or pointer_B != len(B):
        if pointer_A == len(A):  # if you finished list A already
            if last_added_element != B[pointer_B]:
                C.append(B[pointer_B])
                last_added_element = B[pointer_B]
            pointer_B += 1
        elif pointer_B == len(B): # if you finished list B already
            if last_added_element != A[pointer_A]:
                C.append(A[pointer_A])
                last_added_element = A[pointer_A]
            pointer_A += 1
        elif B[pointer_B] < A[pointer_A]:   # if the next element to add is from B
            if last_added_element != B[pointer_B]:
                C.append(B[pointer_B])
                last_added_element = B[pointer_B]
            pointer_B += 1
        else:                               # if the next element to add is from A
            if last_added_element != A[pointer_A]:
                C.append(A[pointer_A])
                last_added_element = A[pointer_A]
            pointer_A += 1

    return C
```

6. Suppose we are given a sequence S of n elements, each of which is colored red or blue. Assuming S is represented as an array, give an in-place method for ordering S so that all the blue elements are listed before all the red elements. Can you extend your approach to three colors?

   Solution:
   For the red and blue elements, we can order them by doing the following. Start with a marker at the beginning of the array and one at the end of the array. While the first marker is at a blue element, continue incrementing its index. Likewise, when the second marker is at a red element, continue decrementing its index. When the first marker has reached a red element and the second a blue element, swap the elements. Continue moving the markers and swapping until they meet. At this point, the sequence is ordered. With three colors in the sequence, we can order it by doing the above algorithm twice. In the first run, we will move one color to the front, swapping back elements of the other two colors. Then we can start at the end of the first run and swap the elements of the other two colors in exactly the same way as before. Only this time the first marker will begin where it stopped at the end of the first run.

7. Suppose we are given an n-element sequence S such that each element in S represents a different vote in an election, where each vote is given as an integer representing the ID of the chosen candidate. Without making any assumptions about who is running or even how many candidates there are, design an $O(n \log n)$-time algorithm to see who wins the election S represents, assuming the candidate with the most votes wins.

   Solution:
   First sort the sequence S by the candidate's ID. Then walk through the sorted sequence, storing the current max count and the count of the current candidate ID as you go. When you move on to a new ID, check it against the current max and replace the max if necessary.

Hints

- **Question 3: Avoid the use of recursion, by doing things bottom-up by levels, and use the auxiliary array as a "buffer" so as to swap S and T with each level.**
- **Question 4: Sort first**
- **Question 7: Sort first, choosing the appropriate key for comparisons.**