7.1 Recursive Sorting Algorithms

SORTING:
- Most fundamental data processing operation in CS
- Mergesort & quicksort

MERGESORT & QUICK SORT:
- Use these steps:

1. Split up the input into two or more parts
2. Recurse on each part separately
3. Combine the results of the previous step into a single result

MERGESORT:
- Divide and conquer 😎💪🔥
- Divides input into two halves & recursively sorts each half, then merges sorted halves

```
def mergesort(lst: List) -> List:
    """Return a sorted list with the same elements as <lst>.
    This is a *non-mutating* version of mergesort; it doesn't mutate the input list.
    """
    if len(lst) < 2:
        return lst[:]
    else:
        # divide the list into two parts, and sort them recursively.
        mid = len(lst) // 2
        left_sorted = mergesort(lst[:mid])
        right_sorted = mergesort(lst[mid:])

        # Merge the two halves using the helper function _merge
        return _merge(left_sorted, right_sorted)

def _merge(lst1: List, lst2: List) -> List:
    """Return a sorted list with the elements in <lst1> and <lst2>.
    Precondition: <lst1> and <lst2> are sorted.
    """
    index1 = 0
    index2 = 0
    merged = []

    while index1 < len(lst1) and index2 < len(lst2):
        if lst1[index1] <= lst2[index2]:
            merged.append(lst1[index1])
            index1 += 1
        else:
            merged.append(lst2[index2])
            index2 += 1
    # After the while.. either index1 == len(lst1) or index2 == len(lst2)
    assert index1 == len(lst1) or index2 == len(lst2)

    # The remaining elements of the other list can all be added to the end of <merged>
    # Note that at the most ONE of lst1[index1:] and lst2[index2:] is not empty (if the lists aren't parallel)
    # To keep the code simple we include both as we are adding empty and smth :)
    return merged + lst1[index1:] + lst2[index2:]
```

QUICKSORT:
- Divides the input based on approximations
- Ex: dividing a class by surname (A-L, M-Z)
- ALGORITHM:
    1. Picks up some element in its input list and calls it the pivot
    2. Splits up the two parts (Partitioning step) :
        - Elements <= pivot
        - Elements > pivot
    3. Sort each part recursively
    4. Concatenates the two sorted parts, putting pivot in between them

```
def quicksort(lst: List) -> List:
    """Return a sorted list with the same elements as <lst>.
    This is a *non-mutating* version of quicksort;
    it does not mutate the input list.
    """
    if len(lst) < 2:
        return lst[:]
    else:
        # Pick pivot to be the first element
        # Could make lost of choices here (e.g. last, random)
        pivot = lst[0]

        # Partition rest of list into two halves using helper function
        smaller, bigger = _partition(lst[1:], pivot)  # Yields a tuple of two items

        # Recurse on each partition
        smaller_sorted = quicksort(smaller)
        bigger_sorted = quicksort(bigger)

        # Return! Notice the simple combining step
        return smaller_sorted + [pivot] + bigger_sorted


def _partition(lst: List, pivot: Any) -> Tuple[List, List]:
    """Return a partition of <lst> with the chosen pivot.
    Return two lists:
    the first contains the items in <lst> <= pivot
    the second contains the items in <lst> > pivot
    """
    smaller = []
    bigger = []

    for item in lst:
        if item <= pivot:
            smaller.append(item)
        else:
            bigger.append(item)

    return smaller, bigger
```