PAGE
DATE

Exercise 1:

a-
```
Concat(List A, List B)

        List of Element  new_List

        New_List.h := A.h

        New_List.last := B.last

        A.last -> next := B.h

        return New_List
```

Proof: This algorithm is correct because in order to concatenate two lists A, B the head would have to be the head of A and the last element would be the last element of B and the two lists need to be linked together which is what this algorithm does

running time: this has a running time of $O(1)$. This is because we are assigning 3 containers and that's it and since assignment is of constant time we get

$$O(1) + O(1) + O(1) = O(1)$$

CamScanner

PAGE
DATE

```
6>  Class List of Element:

      ---           rest of the class
      ---
       size:=0

        size ( ): return size

     InsertAfter (---)
        ----
        ----
           size++

     Remove (~~Han~~---)

           size--

     Concact (---)
      ---
     New_List.size = A.size() + B.size()
```

CamScanner

PAGE
DATE

Exercise 2:

Insertion sort: is stable because if Two elements are equal it doesn't swap Them Thus leaving Them in Their original relative order, and also since we are just swapping every Two neighbouring Elements we dont get a situation ~~were~~ where two equal elements swap orders

Selection sort:

is unstable For example The array [(2,A), (2,B), (1,A)] when we apply selection sort we Find The max element of The remaining array and put it at The end in This case we get [(2,B), (1,A), (2,A)] For The First iteration and since now The last element is considered sorted IT won't be swapped with anything again and Thus The relative ordering has been ruined

To make selection sort stable ~~we~~ instead of finding
The maximum element and put it at the end of the array
we look for the min element and put it at the beginning

merge sort is stable because when we merge the
left and right arrays we check if $A[i] \leq B[j]$
The equality in the check means that elements
in A (The left array) go before elements in B (the right)
if they are equal thus perserving the ordering

Homework Sheet 3

Author	**Matriculation Number**	**Tutor**
Abdullah Oğuz Topçuoğlu	7063561	Maryna Dernovaia
Ahmed Waleed Ahmed Badawy Shora	7069708	Jan-Hendrik Gindorf
Yousef Mostafa Farouk Farag	7073030	Thorben Johr

Exercise 3

We can do something similar to LsdRadix sort we saw in the lecture.
Pseduocode:

```
function SortGridPoints(A[1..n]) {
    redefine key(point) := point.y
    CountingSort(A)

    redefine key(point) := point.x
    CountingSort(A)
}
```

Correctness:

- The first CountingSort sorts the points by their y coordinates.
- The second CountingSort sorts the points by their x coordinates, but since CountingSort is stable, the order of points with the same x coordinate is preserved.
- Therefore after both sorts the points are sorted lexicographically by (x, y).
- Thats also what we did in the lecture for LSDRadixSort we started from the least significant digit to the most significant digit. The same idea applies here.

Running Time Analysis:

- Each CountingSort runs in time O(n + k) where k is the range of the keys.
- Here the keys are the x and y coordinates of the points.
- Since the points are connected, the range of x coordinates is at most n and the range of y coordinates is also at most n.
- Therefore each CountingSort runs in time O(n + n) = O(n).
- Since we perform two CountingSorts, the total running time is O(n) + O(n) = O(n).

Exercise 4

(a)

We can use the LSDRadixSort again.
Pseudocode:

```
function SortStrings(A[1..n], length) {
    for i in length..1 {
        redefine key(string) := string[i]
        CountingSort(A)
    }
}
```

Correctness:

- We sort the strings starting from the last character to the first character (least significant to most significant).
- Each CountingSort is stable, so the order of strings with the same character at position i is preserved.
- Therefore after sorting by all character positions, the strings are sorted lexicographically.
- So just like LSDRadixSort we saw in the lecture where d is alphabet size, U is the alphabet

Running Time Analysis:

- Each CountingSort runs in time O(n + k) where k is the range of the keys.
- Here the keys are the english letters 'a' to 'z', so k = 26.
- Therefore each CountingSort runs in time O(n + 26) = O(n).
- Since we perform CountingSort 26 times, the total running time is O(26 * n) = O(n).

(b)

The idea is i will try to convert this problem to the one we solved in part (a). I will treat every word as if they have the same length which is the maximum length of the strings in the input And then i will introduce a special character for non existent characters in the shorter strings that will come before 'a' in the alphabet. So that when we sort the strings lexicographically the shorter strings will come before the longer strings with the same prefix. So i change the alphabet to be {#, a, b, ..., z} where # is the special character.

Pseudocode:

```
function SortStringsVariableLength(A[1..n], lengths[1..n]) {
    maxLength = 0
    for i in 1..n {
        if lengths(i) > maxLength {
            maxLength = lengths(i)
        }
    }

    for i in maxLength..1 {
        if i > length(string) {
            redefine key(string) := '#'
        } else {
            redefine key(string) := string[i]
        }

        CountingSort(A)
    }
}
```

Correctness:
We proved this in part (a) already.
Running Time Analysis:

- Finding the maximum length takes O(n) time.
- Each CountingSort runs in time O(n + k) where k is the range
- Here the keys are the english letters 'a' to 'z' plus the special character '#', so k = 27.
- Therefore each CountingSort runs in time O(n + 27) = O(n).
- Since we perform CountingSort maxLength times, the total running time is O(maxLength * n + n) = O(m + n) = O(m) where m is the total length of all strings.