D1

CS225 Homework 1

Group Member: Li Rong, Jiang Wenhan, Zhong Tiantian, Zhou Ruidi

Last Modified: February 24, 2022

1. **Our Solution:**

   We suppose the list L has members: $length$ denoting the length of the list, or number of elements in the list; $data[i]$ denoting the i-th element of the list. Assuming the index starts from 1.

   Therefore, just cut down the length of l to disable elements after the i-th in the list and w obtain $L.length := L.length - k$, so the elements $data[i+1, i+2, \cdots, length]$ will be "ignored", or deleted.

   Since this only requires one instruction thus the complexity has nothing to do with k, the answer should be $\Theta(1)$.

2. (a) **Our Solution:**

   When function $g : T' \times T' \to T'$ satisfies a rule similar to "Law of Association". Denoting a list as t, of which the i-th element is t_i, the sublist containing a-th to b-th element is $t_{a..b}$. The condition is that for every k_1, k_2 satisfying $a < k_1, k_2 < b$, we have $g(t_{a..k_1}, t_{k_1+1..b}) = g(t_{a..k_2}, t_{k_2+1..b})$.

   The reason for this requirement is, any list l with more than three elements can be divided into two lists in $n-1$ ways, where n is the length of the list l. Assuming $g(t_{a..k_1}, t_{k_1+1..b}) \neq g(t_{a..k_2}, t_{k_2+1..b})$, we find if the division happens at the k-th element, i.e. dividing the list into $t_{a..k}$ and $t_{k+1..b}$, the result depends on the value of k, which is against definity.

   (b) **Our Solution:**

Algorithm 1: Get list length

```
input : A list
output: The length of list
begin getLength
    if list[id] reaches EndOfList then
        if id = 0 then
            return zero length
        else
            return 1
        end
        return getLength (list, id +1)
    end
end
```

Length of the list For the first operation, we iterate every element until reaching the end of the list.
Since the algorithm just iterate from the first element to the last, the complexity is $\Theta(n)$.

Perform a function to all element For the second operation, assuming the function f accepts an element x in l. For our structural recursion, refer to Algorithm **??**.
Same as above, since the algorithm iterate from the first to the last, the complexity should be $\Theta(n)$.

Find sublists For the structural recursion, time consuming (time complexity) is $\Theta(1)$ for one loop. And it will recursive for n times which means the time complexity will be $\Theta(n)$. Refer to Algorithm **??**

   (c) **Our Solution:**

   Complexity for all operations are $\Theta(n)$, because they all iterate from the first element to the last, each element costing a constant time.

3. **Our Solution:**

   Please refer to Figure **??**. The left half is denoted as Figure I, and right Figure II.

Algorithm 2: Do a Function for All Elements

input : list, the id of currently processing element, the array of result [] containing return values for all element before the id-th
output: result [] containing return values for all elements

```
begin Function
    if length = 0 then
        Return the value for an empty list, and exit
    end
    if id = length − 1 then
        Return in result [id ] the function value
    end
    if id < length then
        Compute the function value
        result[id] ← the function value
        Call Function (list, id + 1, result[])
    end
end
```

Algorithm 3: Find Sublists

input : A list list, an index *id*
output: The sublist of elements in list satisfying condition φ

```
begin findSublist
    if id == 0 then
        Exit the call and return to caller with an empty sublist
    end
    if id > length(list) then
        Exit this call
    end
    if list[id] satisfies condition φ then
        Append (list [id ], sublist)
    end
    findSublist (list, id + 1)
end
```

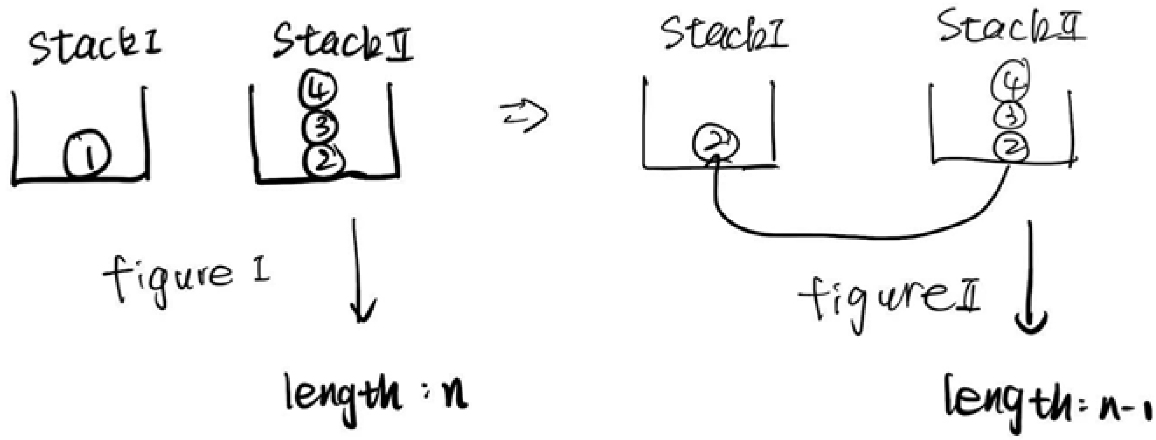

Figure 1: Figure for exercise 3.

- As mentioned in figure I, when we have two stacks, we can store the first data in the stack I and then we should store other data in stack II (FI: FIRST INPUT)
- When we need data, we will fetch the first data in the stack I (FO: FIRST OUTPUT) and then move it.
- Then we will use pointer to read the second data (we can read and copy any data in the stack no matter where it is) and copy it to the stack I, which means we can read the second data (FIFO). Then delete the data in the stack I and copy the third one into the stack I and repeat the steps. By this means, we can read and use the data in order. (As mentioned in the Figure I to Figure II)
- Each time we move one data from stack II to stack1, we minus the length of the stack II by one (If the original length of the stack II is n, when we move the second data from stack II to stack1, its length becomes $n - 1$, when we move the third data from stack II to stack I, the length of the stack becomes $n - 2$). By this means, Each FIFO operation takes amortised constant time. (As mentioned in the length indication of the figure I and figure II)

4. **Our Solution:**

   Please refer to the code bundle.