Warm-up

Problem 1. Sort the following functions in increasing order of asymptotic growth

$$n, n^3, n\log n, n^n, \frac{3^n}{n^2}, n!, \sqrt{n}, 2^n$$

Solution 1.

$$\sqrt{n}, n, n\log n, n^3, 2^n, \frac{3^n}{n^2}, n!, n^n$$

Problem 2. Sort the following functions in increasing order of asymptotic growth

$$\log\log n, \log n!, 2^{\log\log n}, n^{\frac{1}{\log n}}$$

Solution 2.

$$n^{\frac{1}{\log n}}, \log\log n, 2^{\log\log n}, \log n!$$

Problem 3. Suppose we implement a stack using a singly linked list. What would be the complexity of the push and pop operations? Try to be as efficient as possible.

Solution 3. To push an element we add it at be beginning of the list. To pop an element we delete and return the first element of the list. Both operations take $O(1)$ time.

Problem 4. Suppose we implement a queue using a singly linked list. What would be the complexity of the enqueue and dequeue operations? Try to be as efficient as possible.

Solution 4. To enqueue an element we add it at be end of the list. To dequeue an element we remove it from the front of the list. To keep the enqueue operations efficient, we can keep a pointer to the last element of the list, so that both operations take $O(1)$.

Problem 5. Consider the following pseudocode fragment.

```
1: function printFive($n$)
2:     for $i \leftarrow 1$; $i \leq 5$; i++ do
3:         Print a single character $'n'$
```

Using the O-notation, upperbound the running time of printFive.

Solution 5. Printing a single character takes constant time. The for-loop is always executed exactly five times, so the running time is therefore

$$\sum_{i=1}^{5} O(1) = 5 \cdot O(1) = O(1).$$

Problem 6. Consider the following pseudocode fragment.

```
1: function STARS($n$)
2:     for $i \leftarrow 1$; $i \leq n$; i++ do
3:         Print '*' $i$ times
```

a) Using the O-notation, upperbound the running time of STARS.

b) Using the Ω-notation, lowerbound the running time of STARS to show that your upperbound is in fact asymptotically tight.

Solution 6.

a) The first iteration prints 1 star, second prints two, third prints three and so on. The total number of stars is $1 + 2 + \cdots + n$, namely,

$$\sum_{j=1}^{n} j \leq \sum_{j=1}^{n} n = n^2 = O(n^2).$$

b) Assume for simplicity that n is even. To lowerbound the running time, we consider only the number of stars printed during the last $\frac{n}{2}$ iterations. Since this is part of the full execution, analyzing only this part gives a lower bound on the total running time. The main observation we need is that for each of the considered iterations, we print at least $n/2$ stars, allowing us to lower bound the total number of stars printed:

$$\sum_{j=1}^{n} j \geq \sum_{j=n/2+1}^{n} \frac{n}{2} = \frac{n^2}{4} = \Omega(n^2).$$

Problem solving

Problem 7. We want to extend the queue that we saw during the lectures with an operation GETAVERAGE() that returns the average value of all elements stored in the queue. This operation should run in $O(1)$ time and the running time of the other queue operations should remain the same as those of a regular queue.

a) Design the GETAVERAGE() operation. Also describe any changes you make to the other operations, if any.

b) Briefly argue the correctness of your operation(s).

c) Analyse the running time of your operation(s).

Solution 7.

a) Recall that the average is the total sum of the elements divided by the number of elements. Since we already store the size of the queue, it suffices to add a single new variable that stores the sum of the elements, say *sum*. When a new element get enqueued or dequeued, the sum needs to be updated.

```
function getAverage()
    if isEmpty() then
        return "queue empty"
    else
        return $sum/size$
```

```
function newEnqueue($e$)
    $sum \leftarrow sum + e$
    enqueue($e$)
```

```
function newDequeue()
    $e \leftarrow$ dequeue()
    $sum \leftarrow sum - e$
    return $e$
```

b) The correctness follows directly from the definition of average, assuming we maintain the sum correctly. Since we add the enqueued element's value to the sum and subtract it when it's dequeued, the sum is maintained correctly.

c) The getAverage operation checks if the queue is empty, which takes $O(1)$ time and either throws an error ($O(1)$ time) or returns the required division of two integers ($O(1)$ time). Hence, the total running time is $O(1)$ as required.
We modified the enqueue and dequeue operations. Adding the new element to the *sum* takes $O(1)$ time, so enqueue still runs in $O(1)$ time. Similarly, subtracting the removed element from the *sum* takes $O(1)$ time, so dequeue still runs in $O(1)$ time.

Problem 8. Using only two stacks, provide an implementation of a queue. Analyze the time complexity of enqueue and dequeue operations.

Solution 8. The simplest solution is to push elements as they arrive into the first stack. When we are required to carry out a dequeue operation, we transfer all the elements to the second stack, pop once to later return the element on the top of the second stack, and then transfer back all the remaining elements back to the first stack.

This strategy works because when we transfer the elements from one stack to the next, we reverse the order of the elements. Before we transfer things, the most recent element to be queued is at the top of the first stack. After we transfer we have the oldest element queued at the top of the second stack. Finally, when we transfer the elements back to the first, we go back to the original stack order.

If the queue holds n elements each enqueue operation takes $O(1)$ time and each dequeue takes $O(n)$ time since we need to transfer all n elements twice.

Problem 9. Consider the problem of given an integer n, generating all possible permutations of the numbers $\{1, 2, \ldots, n\}$. Provide a non-recursive algorithm for this problem using a stack.

Solution 9. For the non-recursive version we simulate the calls to the helper function with a stack. We use the tuple (c, i, j) to denote stages of a call. The tuple $(\text{"start"}, i, j)$ corresponds to the start of the for loop for some choice of (i, j) and the tuple $(\text{"finish"}, i, j)$ to the part of the body of the for loop after the recursive call to `helper`.

```
function PERMUTATIONS($n$)
    # input: integer n
    # do: print all permutations of order $n$
    $A \leftarrow [1, 2, ..., n]$
    $S \leftarrow$ a stack with the tuple ("start", 0, 0)
    while $S$ is not empty do
        $c, i, j \leftarrow S$.pop()
        if $c =$ "start" then
            if $i = n$ then
                Print $A$
            else
                $A[i], A[j] \leftarrow A[j], A[i]$
                $S$.push("finish", $i$, $j$)
                $S$.push("start", $i + 1$, $i + 1$)
        if $c =$ "finish" then
            $A[i], A[j] \leftarrow A[j], A[i]$
            if $j < n - 1$ then
                $S$.push("start", $i$, $j + 1$)
```