COMP2123 Assignment 1

March 11, 2020

Student ID: 480048691

1 Problem 1

```
[ ]: function sum(A):
         answer <- 0
         for i <- 0; i < n; i++ do
             if A[i] < i then
                 answer <- answer +A[i]
         return answer
```

1.1 Counting the primitive operations

Line 1: Two operations: initialise variable `answer` and assign it to 0.

Line 2:
- Two operations: initialise `i` and assign it to 0.
- Verying that i < n, which was performed n+1 times.
- The body of the **for** loop is repeated n times (for i = 0, 1, ..., n-1).

Within the for loop:
Line 3: Two operations: indexing and comparing.
Line 4:
- Two operations: indexing and assigning, performed when the if condition is true.
- Counter i is then incremented by 1 => two operations: summing and assigning.
- Each loop there would be at most 5 operations (when the if condition is true) or at least 3 units (when the if condition is false).

Outside the for loop:
Line 5: One operation: returning.

Therefore, the number of primitive operations T(n) executed is at least: `2 + 2 + (n + 1) + 3n + 1 = 4n + 6` when all values of A[i] is greater than or equal to i and the variable `answer` will not be updated at all.

And at most `2 + 2 + (n + 1) + 5n + 1 = 6n + 6` when all values of A[i] is less than i and the variable `answer` is updated for every iteration. **O(6n + 6) = O(n) and so the upperbound of the running time of the above algorithm is O(n).**

2 Problem 2

2.1 Design operations

```
[ ]: # changes

def ENQUEUE(e):
    #the operation remains the same as that in the lecture
    #except the total variable is incremented at the end
        ...
        total = total + e
    return e

def DEQUEUE():
    #the operation remains the same as that in the lecture
    #except the total variable is decremented at the end
    if isEmpty():
        ...
    else:
        e = Q[first]
        ...
        total = total - e
    return e
```

```
[ ]: #design the GET_AVERAGE() operation

def GET_AVERAGE():
    if isEmpty():
        return `empty queue`
    return total / size()
```

2.2 The correctness of the operations

Let the variable `total` be a global variable that can be used in any method. It can now be updated at the very end of the functions Enqueue(e) and Dequeue( ) before returning the element e to ensure that element was successfully added into or removed from the queue.

For the GET_AVERAGE( ) function, two operations were used:
- **size( ):** returning the number of elements existing in queue
- **isEmpty( ):** checking if the queue is empty

If the function isEmpty( ) returns true then the operation will return an error message because we cannot calculate the average of an empty queue. Otherwise, the average will be calculated using `total / size()` and will be returned.

2.3 Running time

Each of the increment or decrement operator only adds two primitive operations (adding/subtracting and assigning) to the original Enqueue(e) and Dequeue() operations. **Thus,**

both the newly updated Enqueue(e) and Dequeue( ) functions now still run in O(1) time.

For the GET_AVERAGE( ) function, both the isEmpty( ) and size( ) functions run in O(1) time. No iterations were performed and other operations including division and returning contribute O(1) to the total running time. **Therefore, the GET_AVERAGE( ) operation also runs in O(1) time.**

3 Problem 3

3.1 Design an algorithm solving the problem

[ ]:
```
def COUNT_PAIRS(A, k):
    #Input: an array of n sorted integers A, an integer k
    #Output: Number of pairs of indices (i != j) such that A[i] * A[j] <= k
    i = 0
    j = len(A) - 1
    count = 0
    while (i < j):
        if A[i] * A[j] <= k:
            count += (j - i)
            i += 1
        else:
            j -= 1
    return count
```

3.2 The correctness of the algorithm

First of all, we can establish a fact that for any integer a, b, c, and k with $b <= c$. If $a * c <= k$, then $a * b <= k$.

Let A be an array of length n. We set i to be the first element (index 0) and j to be the last element (index n - 1). Since the given array is sorted, we can deduce that if $A[i] * A[j] <= k$ then $A[i] * A[i+1] <= A[i] * A[i+2] <= ... <= A[i] * A[j-1] <= A[i] * A[j] <= k$. This is true because all elements from `A[i+1]` to `A[j-1]` are ensured to be less than or equal to `A[j]` (property of a sorted array).

Every time the condition $A[i] * A[j] <= k$ is met, by the fact above, any product of $A[i] * A[i+1]$ to $A[i] * A[j]$ will be less than or equal to k. Thus, the number of pairs corresponding to this set of (i, j) would be `j - (i + 1) + 1 = j - i`.

We can then increment i by 1 to check for the next pair of `(new i, j)`. Again, if $A[newi] * A[j] <= k$ then the products of `A[new i]` and `A[new i + 1]` to `A[j - 1]` will also be less than k.

However, if at any point the condition $A[i] * A[j] <= k$ is not met, the above conclusion does not hold. We can then decrement j by 1 and check the condition for the new pair of `(i, new j)`.

The while loop will repeat as long as `i < j`. That is, from `i = 0` to `i = j - 1` to ensure the condition `i != j` holds. When `i = j`, the while loop will stop.

3.3 The running time of the algorithm

The while loop is effectively a cursor to move within the array A (by updating i and j).

From line 1 to line 3, each runs in O(1) as they all perform initialising a variable and assigning.

Within the while loop:
- Line 4: O(1), indexing and comparing
- Line 5: O(1), calculating and assigning
- Line 6: O(1), adding and assigning
- Line 8: O(1), subtracting and assigning

For each pair (i, j), the running time within the while loop is both maximum and minimum O(1).

Outside the while loop
- Line 9: O(1), returning count

The worst case senario is when all products of $A[i] * A[j]$ are less than or equal to k, with `i != j`.

An example of this case is when A = [1, 1, 1, 1] and k = 2. For this example, the index j will stay the same throughout the operation while i will be updated from 0 to n - 1 (the operation stops at `i = n - 1`). Each time i is updated, the conditional code will run and **in total this will take n * O(1) time, which is O(n)**.