Week 1. Problem set (solutions by Danil Elgin)

1. In [CLRS, §16.1], a stack with an extra operation MULTIPOP is discussed. What is the total cost c of executing n of the stack operations PUSH, POP, and MULTIPOP, assuming that the stack begins with k objects? The answer must consist of exact lower and upper bounds given as formulae in terms of n, k (not asymptotic complexity!). Provide a brief justification (2–4 sentences). Formulas and equations without explanation will not be accepted.

   Answer: *$n \leq c \leq 2 * (n - 1) + k$*
   *Let us assume that the cost of push and pop equal to 1, then multipop(min(n, a)) equal to min(n, a) where a is a some number and n is the number of all elements. The best case: we use n times push operation, then the total cost is n. The worst case: we use (n - 1) times push operation and 1 time multipop operation with an argument (n - 1 + k), then the total cost is (n - 1) + (n - 1) + k. Therefore, the lower bound is n and the upper bound is 2 * (n - 1) + k .*

2. A sequence of PUSH, POP, and CLEAN operations is performed on an initially empty stack. When an element is first PUSHed to the stack, it is marked as **new**. The CLEAN operations inspects all elements of the stack:

   - Every **new** element is marked as **old** (in constant time).
   - Every **old** element is removed from the stack (in constant time).

   Perform amortized time complexity analysis using the **accounting method** [CLRS, §16.2] for a sequence of PUSH, POP, and CLEAN operations performed on an initially empty stack:

   (a) Specify actual cost, amortized cost, and accumulated credit for each operation. Assume that n_i is the size of stack before operation and k_i is number of **old** elements in the stack.

Operation	Actual cost (c_i)	Amortized cost ($\widehat{c_i}$)	Credit	Justification
PUSH	1	2	1	We pay for putting the item in the stack and that in the future it can be removed either by pop operation or clean operation.
POP	1	0	-1	For every operation pop we have operation push respectively. Each push operation has already paid for the deletion of the object using pop or clean.
CLEAN	n_i	$n_i - k_i$	$-k_i$	Amortized cost is $n_i - k_i$ because push operation has already paid for the deletion of the old objects, but we should pay for doing new objects to old. There are $n_i - k_i$ new objects in particular time.

   (b) Prove that the total amortized cost of a sequence of n operations provides an upper bound on the total actual cost of the sequence.

   Proof. For each push we save 1 credit. When we use pop we already have 1 credit for this operation because for each operation pop there must have been a corresponding operation push. When we use operation clean we already have enough credits to delete old elements because we can delete the particular element just once and operation push pay for it. However, operation clean pay for changing new elements to old. In total there are $n_i - k_i$ new elements in particular time and when we use operation clean we pay for changing. Proof in inequality: (x is a number of push operations, y is a number of pop operations, z is a

number of clean operations, k is a number of old elements. $y <= x$)
Amortized: $x * 2 + y * 0 + z(x - y - k) >= x + y + z(x - y - k)$
$2x >= x + y$
By condition $y <= x$. Hence, $2x >= x + y$. QED □

(c) Write down the asymptotic complexity for a sequence of n operations.
Answer: $O(n)$
Derivation of the answer: *Each pop and push operations have time complexity $O(1)$, so a sequence of those will have a complexity of $O(n)$.In the worst case, clean can go through all elements in the stack twice in order to get them old and then delete it. It means that time complexity of operation clean $O(n)$ because we can go through all elements just twice and no matter how many operations clean will be used. It shows that asymptotic complexity for a sequence of n operations is O(n).*

3. Solve the previous exercise using the **potential method** [CLRS, §16.3]:

(a) Define the potential function Φ on a stack; the potential function may depend on the size n of the stack and on the number k of **old** elements in the stack;
Answer: *$\Phi(D) = n$, n is the number of elements on the stack*

(b) Compute $\Phi(D_i) - \Phi(D_{i-1})$ for each possible ith operation (PUSH, POP, CLEAN)

- PUSH: $\Phi(D_i) - \Phi(D_{i-1}) = (n+1) - n = 1$
- POP: $\Phi(D_i) - \Phi(D_{i-1}) = (n-1) + n = -1$
- CLEAN: $\Phi(D_i) - \Phi(D_{i-1}) = (n-k) - n = -k$

(c) Compute amortized cost for CLEAN using your potential function;
Answer: *n - k*

(d) Write down amortized asymptotic complexity for CLEAN.
Answer: $O(n)$
Derivation of the answer: *The pop operation of deleting one element is performed in O(1) and the operation to transfer from new to old is also performed. That is, we go through k elements (k is the number of old ones) and perform the deletion operation, that is, the complexity of this is O (k), and then we go through (n - k) elements and perform the transfer operation from new to old, that is, the complexity is O (n - k), and the total O(n-k+k) = O(n)*

4. **(+0.5% extra credit)** Show how to implement an improved version of CLEAN operation from the previous exercise, such that it works in $\Theta(k)$ (worst case) where k is the number of **old** elements in the stack:

(a) Provide the pseudocode of CLEAN for an array based implementation of Stack

1 pseudocode

(b) Provide the pseudocode of CLEAN for a linked list based implementation of Stack

2 pseudocode

```
int k = 0
clean():
if stack.size() != 0
        if tail.type == "new"
                this.k = stack.size()
                tail.type = "old"
```

```
else
        int count = min (k, stack.size())
        while count != 0
                this.tail = tail.previous
                this.tail.next = null
                count -= 1
                this.size -= 1
        k = this.size
```

(c) Explain briefly for each implementation why it works in $\Theta(k)$

3 Explanation

This works with a linked list because we always have a pointer to the end. We can simply change the end and remove the pointer pointing to the last element, and repeat this k times. This means that it works in $\Theta(k)$.

References

[CLRS] Cormen, T.H., Leiserson, C.E., Rivest, R.L. and Stein, C., 2022. *Introduction to algorithms, Fourth Edition*. MIT press.