Amortized analysis

Data Structures - Assignment 1

Winter 2022

1. Question

[10 marks] A sorted stack is a stack in which elements are sorted in increasing order from bottom to top. It supports the following operations:

- pop(S), which removes and returns the top element from the sorted stack S.
- push(S, x), which first pushes item x onto the top of the sorted stack S and then maintains the increasing order by repeatedly removing the item immediately below x until x becomes the largest item in the stack.
- Note that here the push operation is different from that of a standard stack. For example, suppose that the items in the sorted stack S, from bottom to top, are currently −9, −7, 0, 1, 4, 11, 12, 20. Then, after calling push(S, 5), the content of the stack, from bottom to top, becomes −9, −7, 0, 1, 4, 5.
- We implement the stack as a linked list.
- Show that the amortized costs of both operations are O(1).

Solution:

The problem is solved with both aggregate and potential methods.

In a sorted stack, the push operation maintains the increasing order with the pushed element as the largest item in the stack; the worst-case analysis is as follows:

- POP: O(1) or Constant, the pop operations only need to access the top element of the stack and require constant time.
- PUSH: O(S, x), where S is the stack and x is the element pushed. For the worst-case analysis, assume that the value of x is smaller than the lowest element in the stack; The worst-case would be to pop all the elements and push x, hence O(S+1) or O(S), where S is the number of elements in the stack.

Observations:

By analyzing the entire sequence instead of the worst-case analysis. Assuming that the starting point is an empty stack.

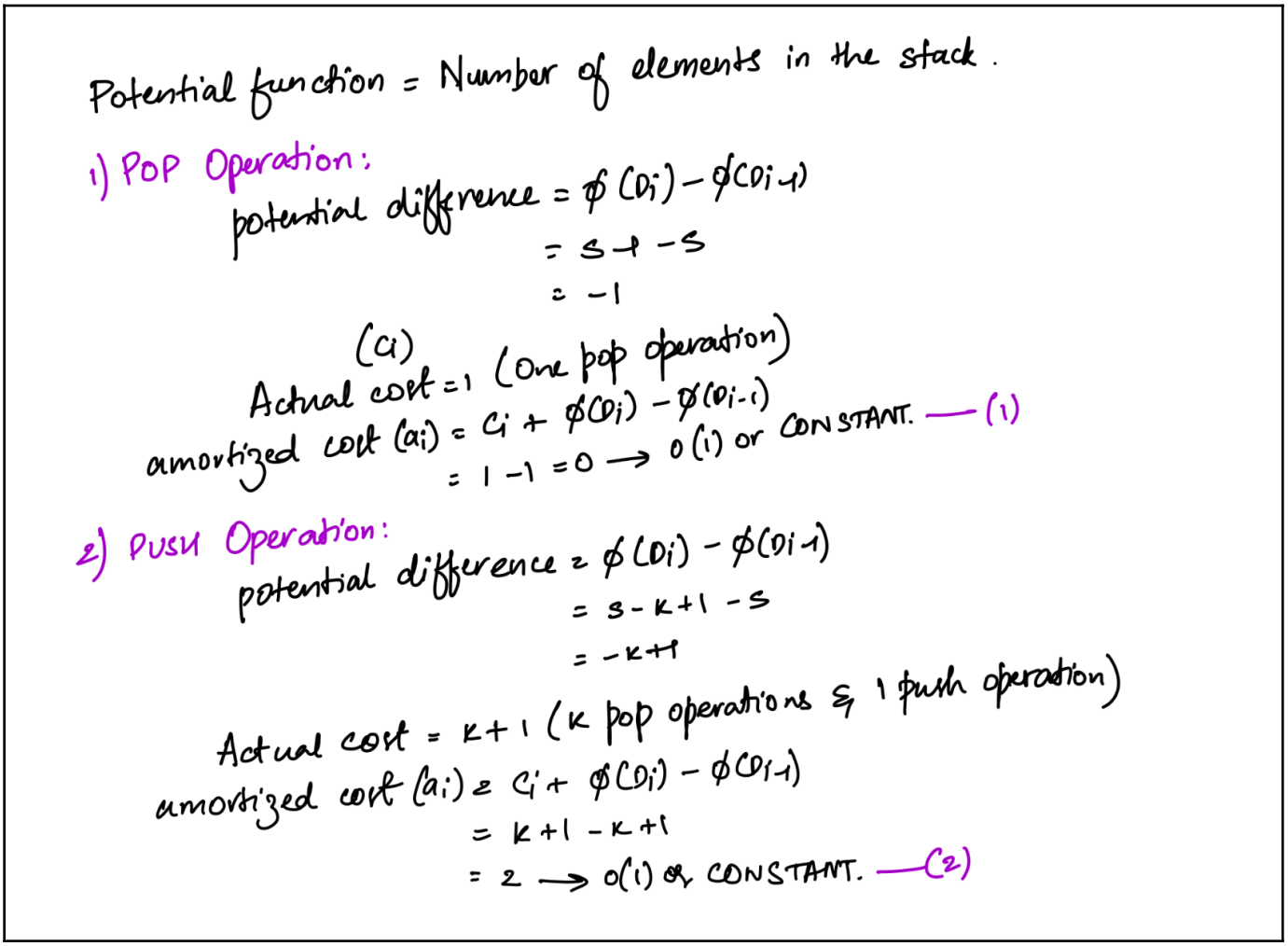
- The pop operation can be performed only once for an element after it is pushed.
- Number of pop operations cannot be more than the number of push operations.
- In the case of PUSH, since multiple POP(s) operations may be necessary, it is at most the number of prior PUSH operations, O(n) - Where n is the number of operations.
- After a PUSH operation, in the worst case, the stack has only 1 element, and in the best case, the stack has S+1 elements (Where S is the number of elements in the stack before the PUSH operations); when all elements in the stack are greater than the pushed element and all elements are smaller than the pushed element respectively.

Conclusion (Aggregate method)

- Hence, the total cost for n operations for PUSH and POP on an empty stack: O(n).
- The Amortized cost of any operation: O(n)/n = O(1) or Constant.

Potential Method

The potential function here is the number of elements in the stack (After the ith operation).



Where 'k' is the number of elements popped, i.e., the number of elements smaller or equal to the pushed element, and the '+1' indicates the pushed element.

Conclusion

From the above equations (1) and (2), the amortized costs of both PUSH and POP operations are O(1).

Implementation of PUSH

Sorted Stack PUSH operation implementation using LinkedList.

```
public static void main(String[] args) {
        LinkedList<Integer> stack = new LinkedList<Integer>();
        stack.push(-9);
        stack.push(-7);
```

```
        stack.push(0);
        stack.push(1);
        stack.push(4);
        stack.push(11);
        stack.push(12);
        stack.push(20);

        push(stack, 5);
    }

    public static void push(LinkedList<Integer> stack, Integer element) {
        if (!stack.isEmpty()) {
            while (!stack.isEmpty() && stack.peek() >= element) {
                stack.pop();
            }
        }
        stack.push(element);
    }
```

---

2. Question

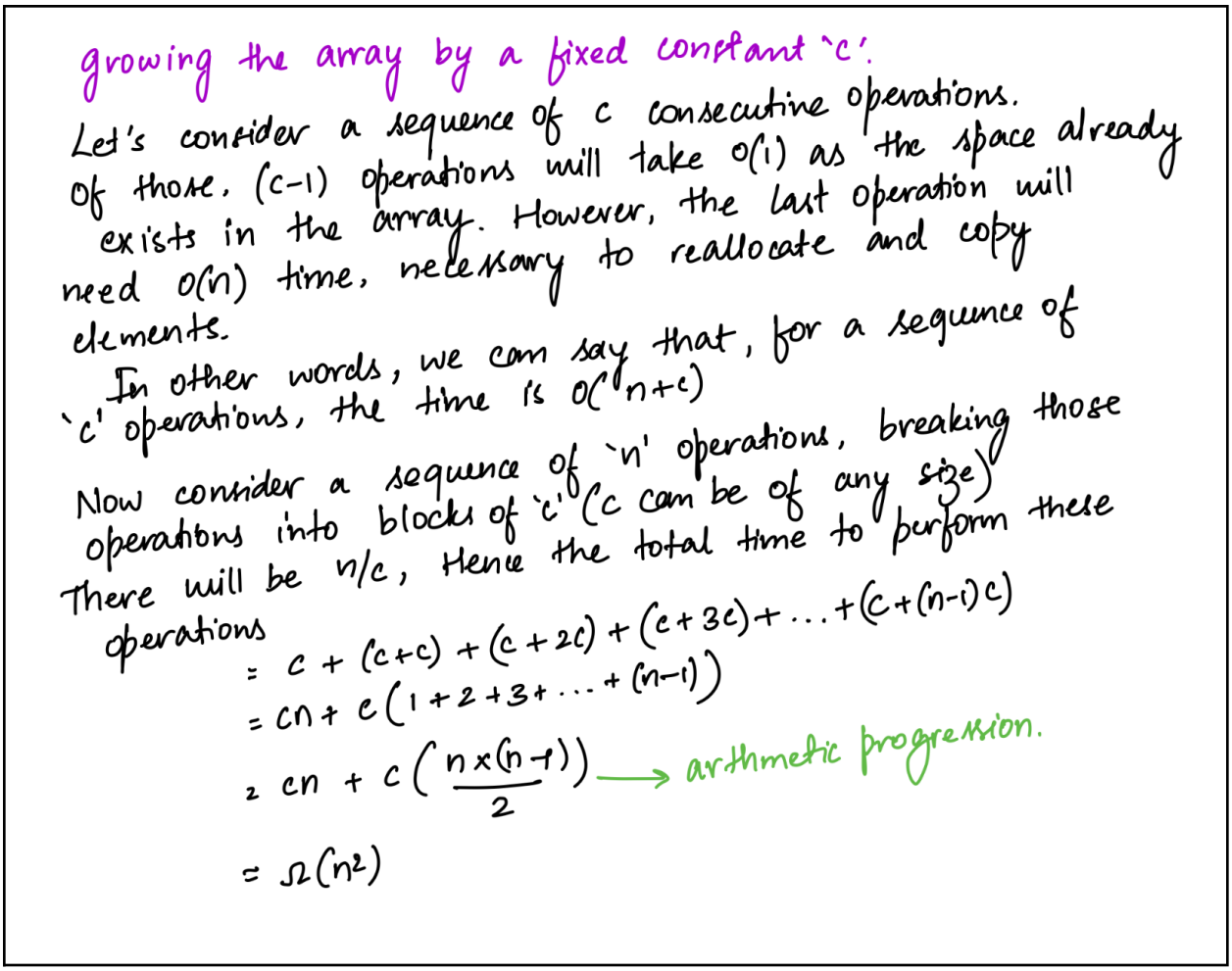
[10 marks] In the vector solution to the problem of maintaining resizable arrays shown in class, we allocate a new array with twice the size as the old one when inserting into a full array.

- In this question, we study the following alternative solution: Initially the array has a block of memory that can store at most c0 entries, where c0 is a positive constant. Each time we insert into a full array, we allocate a new memory block whose size (i.e., the maximum number of entries that it could store) is the size of the current memory block plus a fixed constant value c. We then copy all the elements from the current memory block to the newly allocated memory block, deallocate the old memory block and insert the new element into the new block. The new memory block will be used to store the content of the array afterwards, until we attempt to insert into a full array again.
- Prove that, under this scheme, performing a series of n insertions into an initially empty resizable array takes Ω(n^2) time, no matter what constant value we assign to c.

Solution:

The example in the lecture increased the size of the array by a constant factor, K = 2 (double the size of the array). However, the question now assumes that the array size increase by a fixed constant c.

- So, Consider an array A with c entries to start with (Or an empty array with the need to resize on the first insert); when we add the c + 1th item to the array, the array will be resized by adding an additional c number of spots; Hence the size of the new arrays is 2c, and c number of elements are copied to the new array and similarly, when we add 2c + 1th element, the array is resized, and 2c items are copied to the new array.



Pre-work (Accounting Method)

We know that:

- Creating an empty array: O(1)
- Inserting an element:
    - O(1): Array capacity > Array Size
    - O(n): Array Capacity == Array Size

The basic idea behind the accounting method for amortized analysis is to charge more for O(1) insert operations, save up and pay for the expensive O(n) insertion operations.

Consider the typical dynamic array increasing by a factor of c rather than the constant c, let's say by a factor of 2. Then, saving up two extra operations for every O(1) insertion (A total of 3) to pay for the expensive O(n) insertion would suffice with a time of O(n).

However, for Constant increase. The amount of operations saved is always constant between the capacity increases; i,e. When a new array of size: Current size + constant c has to be allocated, irrespective of the amount of work done to increase the capacity grows linearly with the size of the array. Leading to Ω(n^2) time for a series of n operations.

Conclusion

Hence, while performing a series of insert operations into an initially empty resizable array takes Ω(n^2), no matter what constant value we assign to c.
The amortized cost for each insert operation will then be O(n^2)/(n) ≈ O(n)

To quickly conclude both the methods, for dynamic arrays, the average amortized cost for every insertion is O(n) if the size of the array increases by a constant c (add c) and O(1) when size is increased by a factor c (multiply c).

---

3. Question

[15 marks] In class we saw the example of incrementing an initially 0 binary counter. Now, suppose that the counter is expressed as a base-3 number. That is, it has digits from {0, 1, 2}.

- [5 marks] Write down the pseudocode for INCREMENT for this counter. The running time should be proportional to the number of digits that this algorithm changes.
- [5 marks] Define the actual cost of INCREMENT to be the exact number of digits changed during the execution of this algorithm. Let C(n) denote the total cost of calling INCREMENT n times over an initially 0 base-3 counter. Use amortized analysis to show that C(n) ≤ (3/2)n for any positive integer n. To simplify your work, assume that the counter will not overflow during this process.
- Note: I will give the potential function as a hint after the problem statement, though you will learn more if you try to come up with your own potential function.
- [5 marks] Prove that for any c such that C(n) ≤ cn for any positive integer n, the inequality c ≥ 3/2 holds. Again, assume that the counter will not overflow during this process.

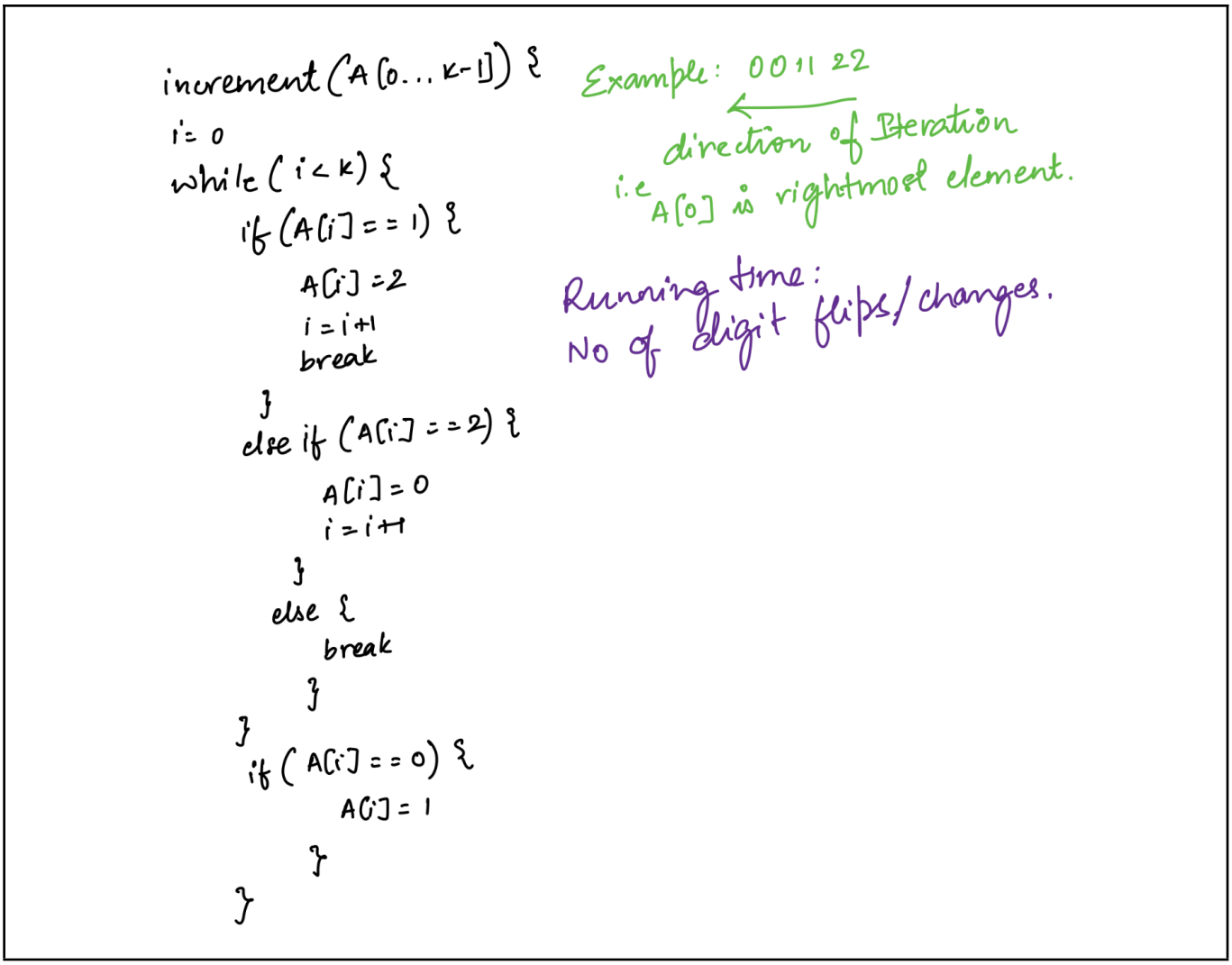
Hint: Let Di be the counter after the i-th INCREMENT, and Let |Di|d be the number of digit d in Di for d ∈ {0, 1, 2}. Define Φ(Di) = (|Di|1)/2 + |Di|2.

Solution:

Note: Edge cases such as empty arrays are not explicitly considered in the pseudo-code (ideally, assuming those to be added in the posterior testing rather than prior analysis).

Pseudo code

Continued on the next page



While this might be obvious, having an explicit note to mention that the break statement inside the conditions does not come out of the while loop but rather breaks that iteration.

Acknowledgement: Since it's easier to represent equations in LaTex or hand-written, I have hand-written [4] the rest of the proof. However, the geometric progression and observations are in text format.

Potential Method:

Amortized analysis for increment in counter for base-3 number. {0, 1, 2}, [2][3]

Continued on the next page

Using potential method, for $c = 3/2$, show that $C(n) \leq cn$

Terminologies

- D_i be the counter after the i^{th} increment (in other words: state of the counter at point i)
- $|D_i|_d$ be the number of digit d in D_i for $d \in \{0, 1, 2\}$.
- b_i is the number of 2s, from D_{i-1}

From the pseudo-code, it is clear that there are 2 cases.

① From the right-most sequence of digits (b_i), there is a Zero (0)

$D_{i-1} = (0/1/2), (0/1/2) \ldots (0/1/2)\ 0\ \underbrace{22\ldots2}_{b_i}$ → remains the same

$D_i = (0/1/2), (0/1/2) \ldots (0/1/2)\ 1\ \underbrace{00\ldots0}$ → Potential Function

(1) — $\phi(D_i) = \phi(D_{i-1}) - b_i + 1/2$ $\quad (\phi(D_i) = |D_i|_1/2 + |D_i|_2)$

② From the right-most sequence of digits, there is a 1 (one)

$D_{i-1} = (0/1/2), (0/1/2) \ldots (0/1/2)\ 1\ \underbrace{22\ldots2}_{b_i}$

$D_i = (0/1/2), (0/1/2) \ldots (0/1/2)\ 2\ \underbrace{00\ldots0}$

$$\begin{aligned} \phi(D_i) &= \phi(D_{i-1}) - b_i + 1 - 1/2 \\ &= \phi(D_{i-1}) - b_i + 1/2 \quad \text{— (2)} \end{aligned}$$

Note: Overflow case is not considered

$\therefore$ The potential $\phi(D_i) = \phi(D_{i-1}) - b_i + 1/2$ — (3)

And the actual cost = $b_i + 1$ → flip the number prior to b_i — (4)

(Total no of FLIPS) (c_i) ↑ no of 2s flip/change

The amortized cost $a_i = c_i + \phi(D_i) - \phi(D_{i-1})$

$= b_i + 1 + \phi(D_{i-1}) - b_i + 1/2 - \phi(D_{i-1})$

$a_i = 1 + 1/2 = 3/2$ — (5)

Continued on the next page: In equation (6), c = 3/2 as proved (as mentioned at the start).

The amortized cost for n operations:

$$\sum_{i=0}^{n} a_i = \sum_{i=0}^{n} c_i + \sum_{i=0}^{n} (\phi(D_i) - \phi(D_{i-1}))$$

$$= \sum_{i=0}^{n} c_i + \phi(D_1) - \phi(D_0)$$

we know that $\phi(D_0) = 0$, $\phi(D_i) \geq 0$

Therefore, $\sum_{i=0}^{n} c_i = \sum_{i=0}^{n} a_i - \phi(D_1)$ — (6)

total cost of calling INCREMENT n times over an initially zero base-3 counter

from (5) Hence, $\frac{3}{2}n \geq C(n)$

prove that for any c, such that $C(n) \leq cn$ for any positive integer n, $c \geq 3/2$ holds good. $(c-\epsilon)$ will not satisfy the above equation.

reiterating the question/equation

prove that for any $c = \frac{3}{2} - \epsilon$, $\epsilon > 0$, such that $C(n) > (\frac{3}{2} - \epsilon)n$ holds

For a Base-3 counter, we know that the rightmost element changes every time, second rightmost every 3 time and so on (i.e 3^k, for $k = 0$ to δ)

$n = 3^\delta$ — The Last element by 3^δ times.

working backwards: consider $\delta = \text{ciel}(-\log_3 \epsilon)$

$$\delta \geq -\log_3 \epsilon$$

$$-\delta \leq \log_3 \epsilon$$

$$3^{-\delta} \leq \epsilon \quad \text{— (7)}$$

$(c-\epsilon)n$, where $c = 3/2$

$$C(n) = \sum_{k=0}^{\delta} 3^k$$

$$C(n) = 3^{\delta+1} - 1 \quad \text{— (8)}$$

$$(c-\epsilon)n \leq (c - 3^{-\delta})n \quad \text{— using (7)}$$

So, $(\frac{3}{2} - \epsilon)n \leq (\frac{3}{2} - 3^{-\delta})n$

for $n = 3^\delta$ — choosing $n = 3^\delta$

$$\leq \left(\frac{3}{2} - 3^{-\delta}\right) 3^\delta = \left(\frac{3^{\delta+1}}{2} - 1\right)$$

$$\leq 3^{\delta+1} - 1 = C(n) \quad \text{— using (8)}$$

$(c-\epsilon)n \leq C(n)$ or $C(n) > (3/2 - \epsilon)n$

Pre-work for the above solution (Aggregate Method):

Note: I initially started solving the problem using aggregate analysis. Hence, some of the work of finding the progression for the base-3 counter is as follows.

Amortized analysis for increment in counter for base-3 [3] number. {0, 1, 2}
Consider the numbers from 0 to 15:

Base-10	**Base-3**
1	0 0 1
2	0 0 2
3	0 1 0
4	0 1 1
5	0 1 2
6	0 2 0
7	0 2 1
8	0 2 2
9	1 0 0
10	1 0 1
11	1 0 2
12	1 1 0
13	1 1 1
14	1 1 2
15	1 2 0

Observations:

- From the above bits, it is clear that the rightmost bit, i.e., the bit in the 0th place changes (flipped) in every operation (Increment).
- Similarly, the second rightmost bit is flipped 1-time in 3 increments, and 3rd rightmost is flipped 1-time in 9 increments, and so on.
- Hence, the total number of flips is equal to the summation of the number of times every bit changes.
    - For some c, Assume n < 3^(c + 1)
- For n number of increments, the summation (Total number of changes) would look like:

- n + n/3 + n/9 + n/27....
- n/3^0 + n/3^1 + n/3^2 + n/3^3 +....+ n/3^c < (3n)/2
- From geometric progression, we know that S = a/1 - r = n/1 - 1/3 = (3n)/2

Furthermore, assuming that it takes constant time to flip a bit, the total cost is O(n). The Amortized cost per operation: O(n)/n = O(1) or Constant.

---

4. Question

[15 marks] In this problem, we consider two stacks, X and Y. n and m denote the sizes of X and Y (here the size of a stack is the number of elements currently in the stack), respectively. We maintain these two stacks to support the following operations:

- PushX(a): Push element a onto stack X;
- PushY(a): Push element a onto stack Y;
- MultiPopX(k): pop min(k, n) elements out of stack X;
- MultiPopY(k): pop min(k, m) elements out of stack Y;
- Move(k): pop min(k, n) elements out of stack X, and each time an element is popped, it is pushed onto stack Y.

We represent each stack using a doubly-linked list, so that PushX, PushY, and a single pop operation on either stack can be performed in O(1) worst-case time.

- [6 marks] What is the worst-case running time of MultiPopX(k), MultiPopY(k) and Move(k)?
- [9 marks] Perform amortized analysis to show that each of these five operations uses O(1) amortized time.

Solution:

Worst-case running time of MultiPopX(k), MultiPopY(k) and Move(k):

MultiPopX: For Stack X with Sx being the number of elements in the stack, in the MultiPopX(k) operation, to pop k elements from the stack X.
The number of operations performed on the stack using a doubly-linked list is a minimum of k and n (i.e., if k is greater than n, the stack is already empty after all n elements are popped.) = min(k, n).

With that said, the worst-case time complexity would be to pop all the elements of the Stack X. Hence worst-case time complexity is O(n)

MultiPopY: The MultiPopY(k) is very similar to MultiPopX(k), where m is the number of elements in the stack. The number of operations would be min(k, m), and the worst-case time complexity would be O(m).

Move: The Move(k) pops k number of elements from Stack X and pushes k number of elements to stack Y, although this operation is done in a sequence of alternative pop and push operations.

In the worst-case, all the elements of Stack X are popped and pushed to Stack Y; hence the time complexity is O(2n) = O(n).

Potential Method:

Amortized analysis for each of the five operations:

Perform amortized analysis to show that each of the 5 operations uses O(1) amortized cost

> 2 stacks X and Y
> No of elements in X = n and No of elements in Y = m
Potential Function (After finding a suitable function)
$= 3n + m.$

The pre-conditions are met as $\phi(D_0) = 0$ (Assuming both stacks X and Y are empty initially.
and $\phi(D_i) \geq 0$, as the size of a stack cannot be negative.

The amortized cost for all 5 operations:

1) PushX(a): push element a onto stack X;

The potential difference $= \phi(D_i) - \phi(D_{i-1})$
$= 3(n+1) + m - (3n+m)$
$= 3$

The actual cost = 1 (One push operation)
The amortized cost $(a_i) = c_i + \phi(D_i) - \phi(D_{i-1})$
$= 1 + 3 = 4$ — (1)

2) PushY(a): Push element a onto stack Y;

Potential difference $= \phi(D_i) - \phi(D_{i-1})$
$= 3n + (m+1) - (3n+m)$
$= 1$

The actual cost $(c_i) = 1$ (one push operation)
The amortized cost $(a_i) = c_i + \phi(D_i) - \phi(D_{i-1})$
$= 1 + 1 = 2$ — (2)

Continued on the next page (Hand-written).

3) MultiPopX(k): pop min(k, n) elements out of stack X;

Potential difference $= \phi(D_i) - \phi(D_{i-1})$
$= 3(n-k) + m - (3n+m)$
$= -3k$

The actual cost $(c_i) = k$ (to pop k elements)
i.e. k pop operations.

amortized cost $(a_i) = c_i + \phi(D_i) - \phi(D_{i-1})$
$= k - 3k$
$= -2k$ — (3)

4) MultiPopY(k): pop min(k, m) elements out of stack Y;

potential difference $= \phi(D_i) - \phi(D_{i-1})$
$= 3n + (m-k) - (3n+m)$
$= -k$

Actual cost $(c_i) = k$ (k push operations)

amortized cost $(a_i) = c_i + \phi(D_i) - \phi(D_{i-1})$
$= k - k = 0$ — (4)

5) Move(k):

potential difference $= \phi(D_i) - \phi(D_{i-1})$
$= 3(n-k) + (m+k) - 3n + m$
$= -2k$

Actual cost $(c_i) = k + k = 2k$ (k pop & k push operations)

amortized cost $(a_i) = c_i + \phi(D_i) - \phi(D_{i-1})$
$= 2k - 2k$
$= 0$ — (5)

Conclusion

From equations (1), (2), (3), (4), and (5), it is clear that each of these five operations uses O(1) amortized time.

Furthermore, consider Equation (3) for MultiPopX(k), which has an amortized cost of -2k; this is justified because PushX(a) adds the credit for the push operation(s) any further addition of credits by MultiPopX(k) would be excessive.

References

[1] "Amortized analysis," Wikipedia, Nov. 04, 2021. https://en.wikipedia.org/wiki/Amortized_analysis (accessed Jan. 15, 2022).

[2] "Epsilon-Delta Definition of a Limit | Brilliant Math & Science Wiki," brilliant.org. https://brilliant.org/wiki/epsilon-delta-definition-of-a-limit (accessed Jan. 15, 2022).

[3] "Ternary numeral system," *Wikipedia*, Jan. 17, 2022. https://en.wikipedia.org/wiki/Ternary_numeral_system (accessed Jan. 16, 2022).

[4] "Notability," *notability.com*. https://notability.com/ (accessed Jan. 16, 2022).