# COL106 Assignment 2: Proving Amortized Time Complexity

Aastha A K Verma (2022CS11607)

August 2023

## 1 Introduction

Amortized time analysis is a technique in which we find the average measure over time for a worst-case sequence of operations in an algorithm. This is a useful method of finding tight bounds on running time (Theta).

The general observation is that some costly (few in number) operations can be compensated because of the larger number of low-cost operations.

## 2 Algorithm used

Typical dynamic arrays which don't follow the principle of hysteresis (i.e., the same resizing factor is used for growth and shrinkage) can have $O(n^2)$ worst-case time complexity (as discussed in class). To use time efficiently, we need to create a delay between the resizing during pushes and pops.

As discussed in class and according to the assignment requirements, the following algorithm has been followed in my implementation:

1. **Push**: Push normally when there are empty spaces in the current array. When the current array fills out completely,
   (a) Double the capacity.
   (b) Reallocate space for an array with the new capacity.
   (c) Copy over the elements from the old array.
   (d) Continue pushing as before.
2. **Pop**: Push normally when the current array is more than one-fourth filled. When the current array becomes three-quarters empty,
   (a) Quarter the capacity.
   (b) Reallocate space for an array with the new capacity.
   (c) Copy over the elements from the old array.
   (d) Continue popping as before.
   (e) When capacity $< 4096$, wait till only 256 elements remain after popping, then decrease the capacity to 1024. (We have waited till 256 to prevent sudden continuous resizing operations when the number of elements goes from 1024 to 1025 and back to 1024.)

**Claim**: The amortized time complexity of the push or pop operation is $\Theta(1)$ (constant time).

# 3 Proof

## 3.1 Notation

Let the cost of primitive operations be as follows:
a: push
b: pop
c: allocation of one empty space for an array
d: copy an element to a new location

Let there be N operations in total. Let X and Y represent the total number of pushes and pops respectively. Clearly, Y $\leq$ X. Let T(x) represent the time required for x operations.

## 3.2 Analysis

Notice that we can take the worst-case sequence of operations to be one where the growth and shrink operations are least spaced out. Let the array initially have $N_0 = 2^{k-1} + n$ number of elements, where $2^{k-1}$ is the greatest power of 2 less than or equal to $N_0$, and $2^{k-1} \geq n \geq 0$. Also, $X + Y = N$

Current capacity = $2^k$

Sequence of operations is described in the given image:

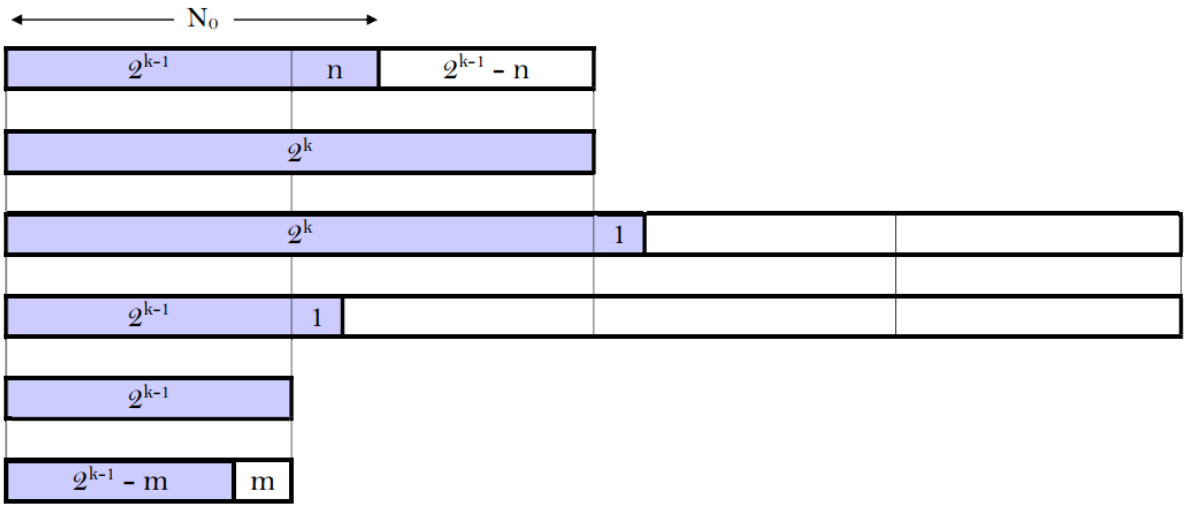


Figure 1: Schematic diagram for stack operations (Coloured part represents filled part of the array)

Here, T(X+Y) =
T(push $2^{k-1} - n$ elements) + T(create new array of $2^{k+1}$ capacity) +
T(copy $2^k$ elements from old array) + T(push new element) + T(pop $2^{k-1} + 1$ elements) +
T(create new array of $2^{k-1}$ capacity) + T(copy $2^{k-1}$ elements from old array) + T (pop m elements)

$$= a(2^{k-1} - n) + c(2^{k+1}) + d(2^k) + a + b(2^{k-1} + 1) + c(2^{k-1}) + d(2^{k-1}) + mb$$

$$= (a + b + 5c + 3d)2^{k-1} + a + b + mb - an$$

We now need to calculate the average of T(X+Y). Consider $3 \times 2^{k-3} > m \geq 0$ and $2^{k-1} \geq n \geq 0$. Averaging m and n over these (linear) ranges, we get

$$m_{avg} = (3 \times 2^{k-3} - 1)/2$$

and

$$n_{avg} = (2^{k-1})/2 = 2^{k-2}$$

Plugging these values, we get $T_{avg}(X, Y) =$

$$= (a + b + 5c + 3d)2^{k-1} + a + b + m_{avg}b - a_{avg}n$$

$$= (\frac{a}{2} + \frac{11b}{8} + 5c + 3d)2^{k-1} + a - \frac{b}{2} =$$

Thus,

$$T_{avg}(X + Y) = \Theta(2^k)$$

Now, from the described sequence of operations, we get

$$X = 2^{k-1} - n + 1$$

$$\Rightarrow X_{avg} = 2^{k-1} - 2^{k-2} + 1 = 2^{k-2} + 1$$

$$Y = 2^{k-1} + 1 + m$$

$$\Rightarrow Y_{avg} = 2^{k-1} + 1 + = 2^{k-1} + 1 + 3(2^{k-3} - 1)/2$$

Thus,

$$X + Y = \Theta(2^k)$$

Therefore, the amortized time complexity is calculated as follows:

$$\frac{T_{avg}(X + Y)}{X + Y} = \frac{\Theta(2^k)}{\Theta(2^k)} = \frac{\Theta(N)}{\Theta(N)} = \Theta(1) \ \square$$

# 4 Conclusion

We have shown that the push/pop operations run in $\Theta(1)$ time. this is because the costly resizing operations are compensated by the low-cost 'usual' pushes/pops. The hysteresis method saves us from consecutive resizing, thus giving us an efficient, low-memory-consuming mechanism.