Amortized Analysis

Ref:

https://www.youtube.com/watch?v=d605guaOH3A

Amortized Analysis is used for algorithms where an occasional operation is very slow, but most of the other operations are faster.

Definition: Amortized analysis provides an average time per operation, considering a sequence of operations, rather than analyzing each operation in isolation.

Goal: Determine a bound on the average cost of an operation, even when individual operations may be costly.

Note#

- for different operation different computation time than go for amortized analysis
- for analysis use addition instead of multiplication.
- If worst case running time is not giving tighter bound, go for amortized analysis.

Why Amortized Analysis?

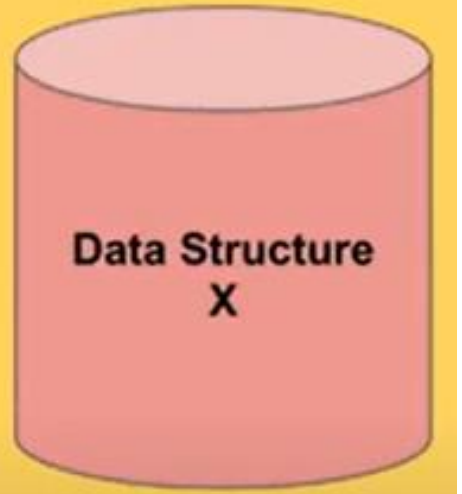
Insight into Average Behavior: Instead of focusing on the worst-case for every single operation, amortized analysis offers a realistic measure of performance over time.

Comparison with Worst-case Analysis: Worst-case analysis might overestimate the cost, leading to less efficient design. Amortized analysis mitigates this by spreading the high costs over many operations.

Imaginary Data Structure X

T(Push): sometimes O(1)
sometimes O(n/2)

Push(val)



What is T (**n** Push)?

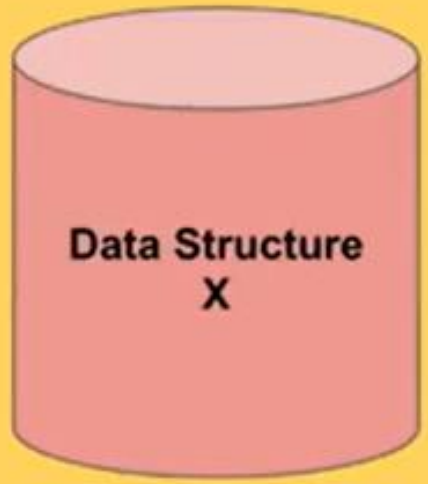
$T(n \text{ Push}) \leq n \times O(n/2) = O(n^2)$

$T(1 \text{ Push}) = O(n^2)/n = O(n)$

$O(n^2)$ is an **upper bound**, but is it **tight**?

T(Push): sometimes O(1)
sometimes O(n/2)

Push(val)



... Suppose:

T(n Push): n/2 + n/2 + n/2 + 1 + ... + 1

3 times *n-3 times*

T(n Push): 3n/2 + n - 3 = O(n) ≤ O(n²)

T(1 Push) = O(n) / n = **O(1) ≤ O(n)**

In **Amortized Analysis**:

If: $\alpha \leq$ **T(1 operation)** $\leq \beta$

we calculate run time of **T(n operations)**.

Amortized T(1 operation) = T(n operations)/n

Welcome to Amortized Analysis!

Methods of Amortized Analysis

There are three primary techniques for performing amortized analysis:

- Aggregate Method
- Accounting Method
- Potential Method

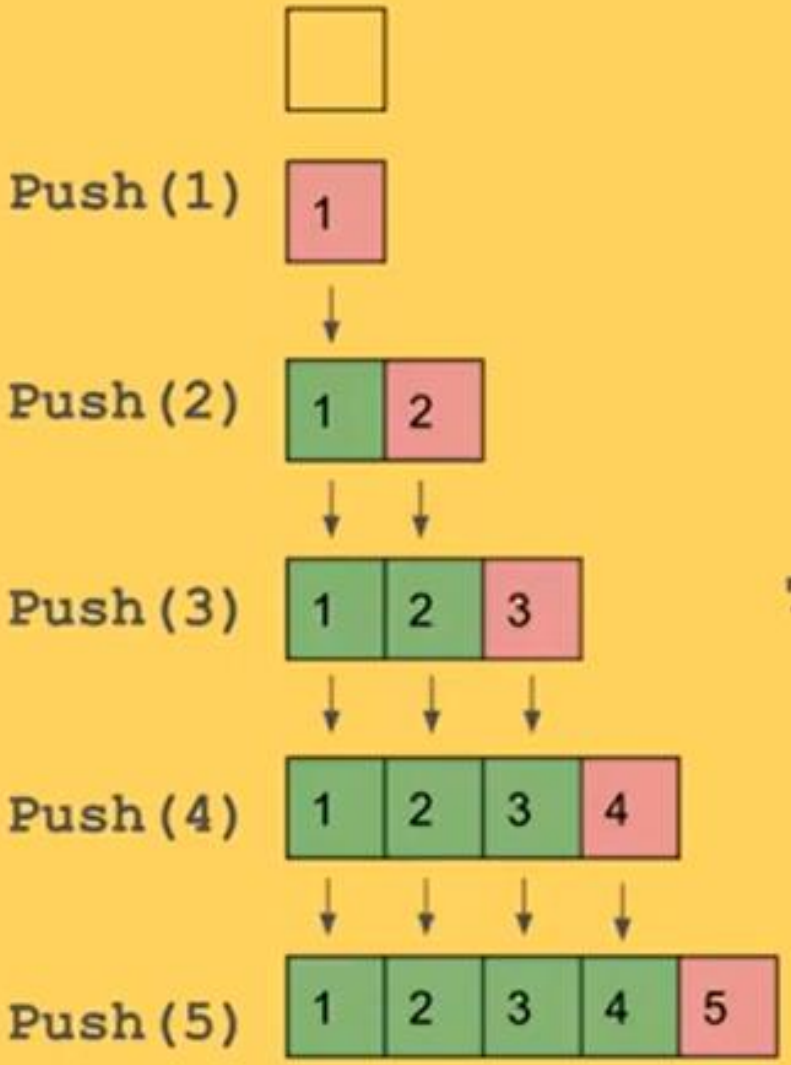
Aggregate Method:

Calculate the total cost of a sequence of “n” operations and divide by “n” to get the average cost per operation.

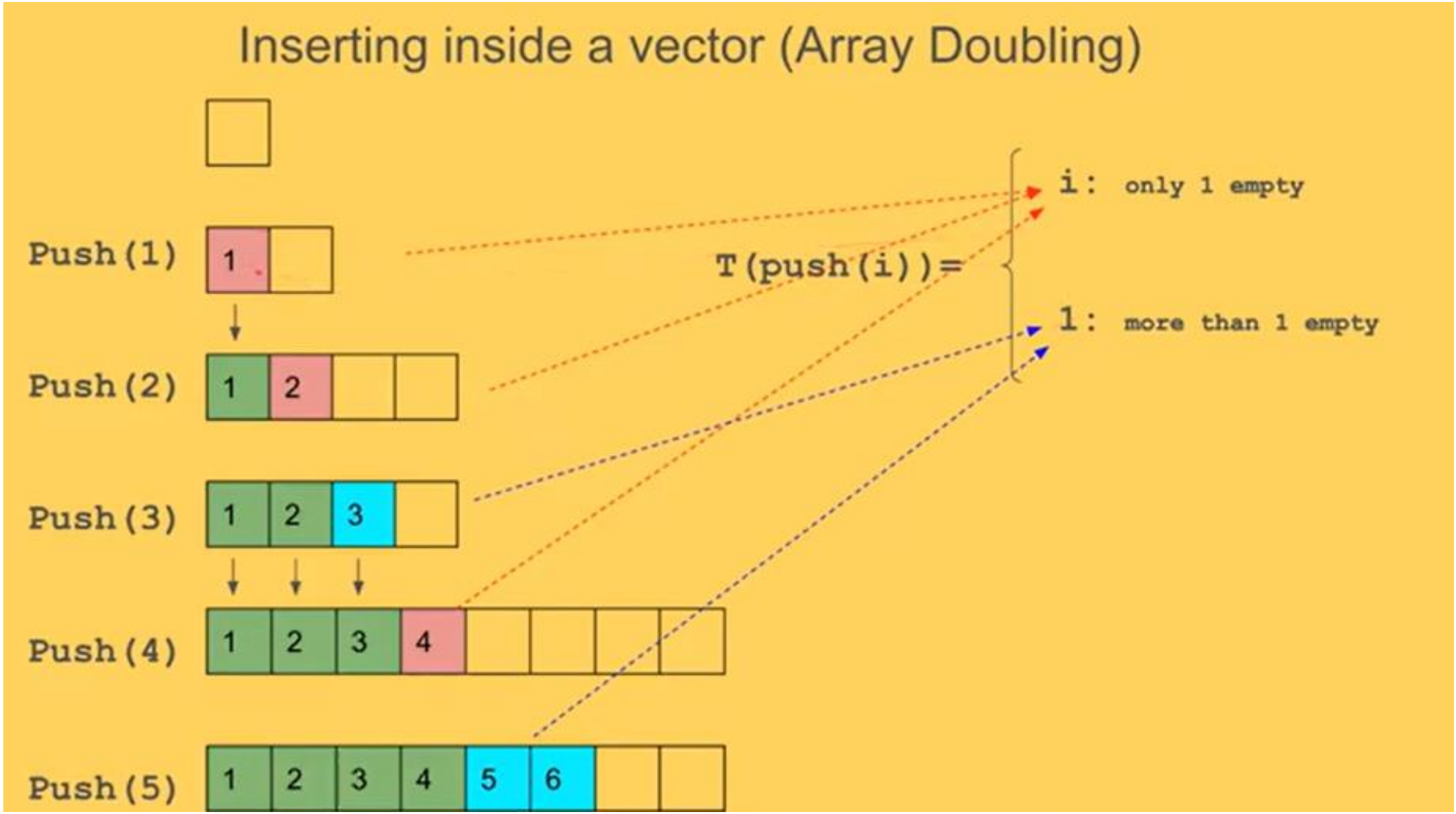
Example: Dynamic Array Insertion

When an array reaches its capacity, it doubles in size. This resizing operation is costly, but amortized analysis reveals that it’s efficient over many operations.

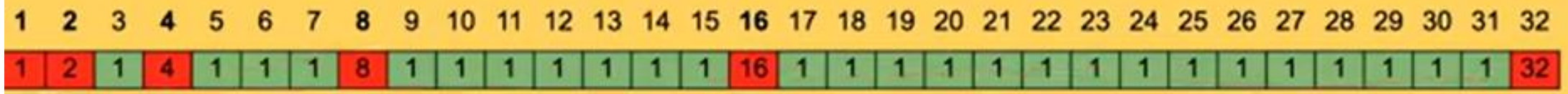
Inserting inside a vector (Simple Method)



$$T(Push(i)) = T(i-1\ Copy) + T(1\ Push) = O(i)$$


Inserting inside a vector (Array Doubling)
Push(1)
1
Push(2)
1
2
Push(3)
1
2
3
Push(4)
1
2
3
4
Push(5)
1
2
3
4
5
6
T(push(i))=
i: only 1 empty
1: more than 1 empty

$$T(\text{Push}(i)) = \begin{cases} \text{Sometimes: } i \\ \text{Sometimes: } 1 \end{cases}$$



$$T(32\ \text{Push}) = T(\text{Push}(1)) + T(\text{Push}(2)) + \ldots + T(\text{Push}(32))$$

$$= (1+2+4+8+16+32) + (32-6)*1$$

$$= (64-1) + (32-(\log_2(32)+1))$$

$$T(n\ \text{Push}) = 2*n - 1 + n - \log_2(32) - 1 = 3n - \log_2(n) - 2$$

$$T(n\ \text{Push}) = O(3n) = O(n)$$

$$T(\text{push}(i)) = \begin{cases} \text{Sometimes: } i \\ \text{Sometimes: } 1 \end{cases}$$

$$T(n \text{ Push }) \leq C_1 * n \leq C_2 * n^2$$

$$\boxed{T(n \text{ Push }) = O(n)}$$

$$\text{On Average, } T(1 \text{ Push }) = T(n \text{ Push })/n = O(1) \leq O(n)$$

Amortized runtime of **each push** is: **O(1)**

2. Accounting Method/Banker Method

Assign a “charge” to each operation, sometimes overcharging simpler operations to “save” for more expensive ones. The saved charges cover the cost of expensive operations.

Idea:

- Spend more, keep extra amount in Bank.
- Use Bank credit in the future.
- Make sure your bank account is not negative.

Action	Normal Cost	Amortized Cost
Buying Stamp	$1	$2
Buying Envelope	$1	$0

$1 for stamp and $1 deposited in the bank

Use the previous deposited money.

$$T(\text{push}(i)) = \begin{cases} \text{Sometimes: } i \\ \text{Sometimes: } 1 \end{cases}$$

Each push costs 3 units.(1 push takes 1 unit but ramaining 2 units stored in Bank for copy in future)

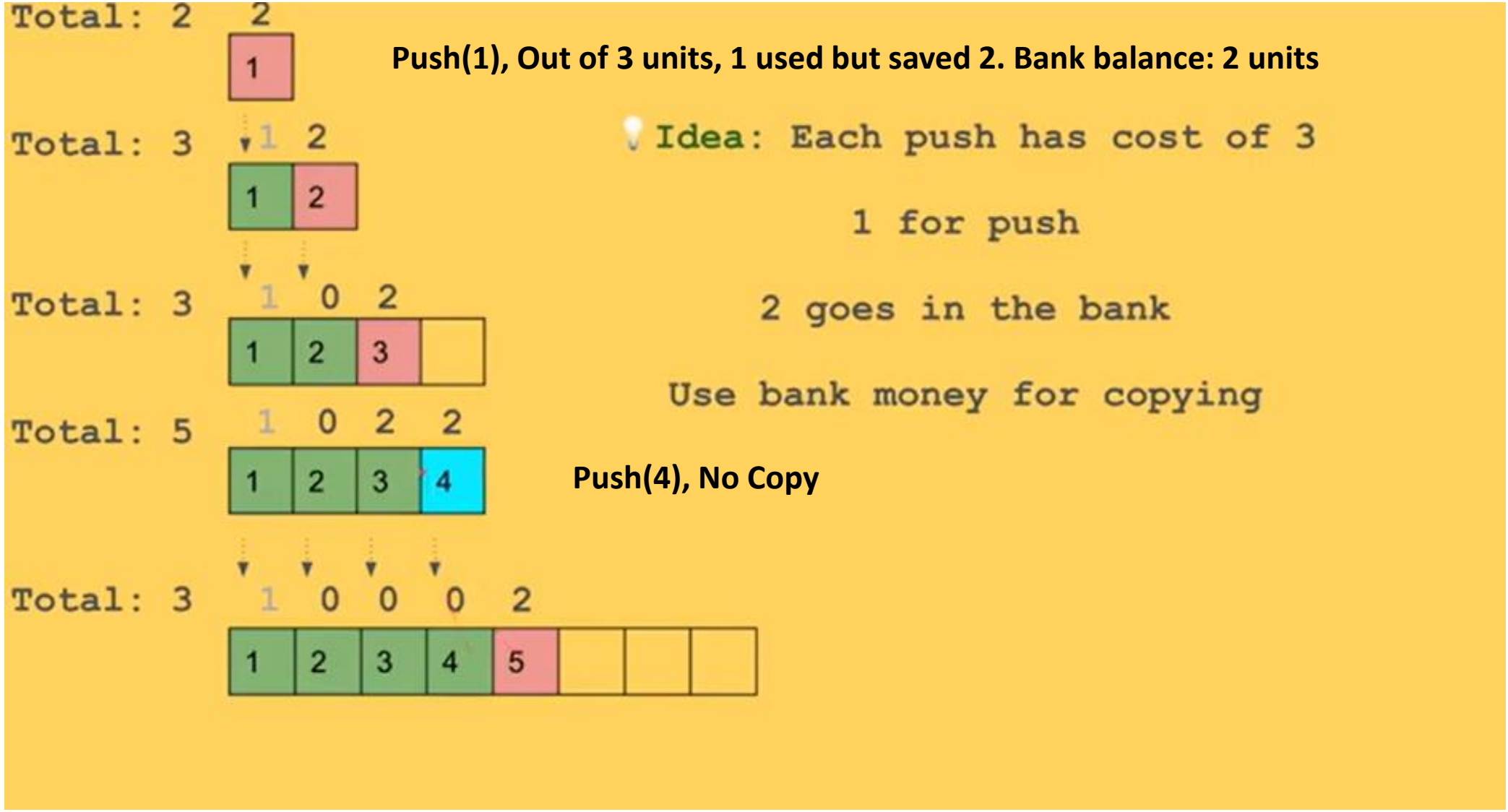
$$T(n \text{ Push }) = O(3n) = O(n)$$

Idea: What if from the beginning, we consider each push costs 3 units?

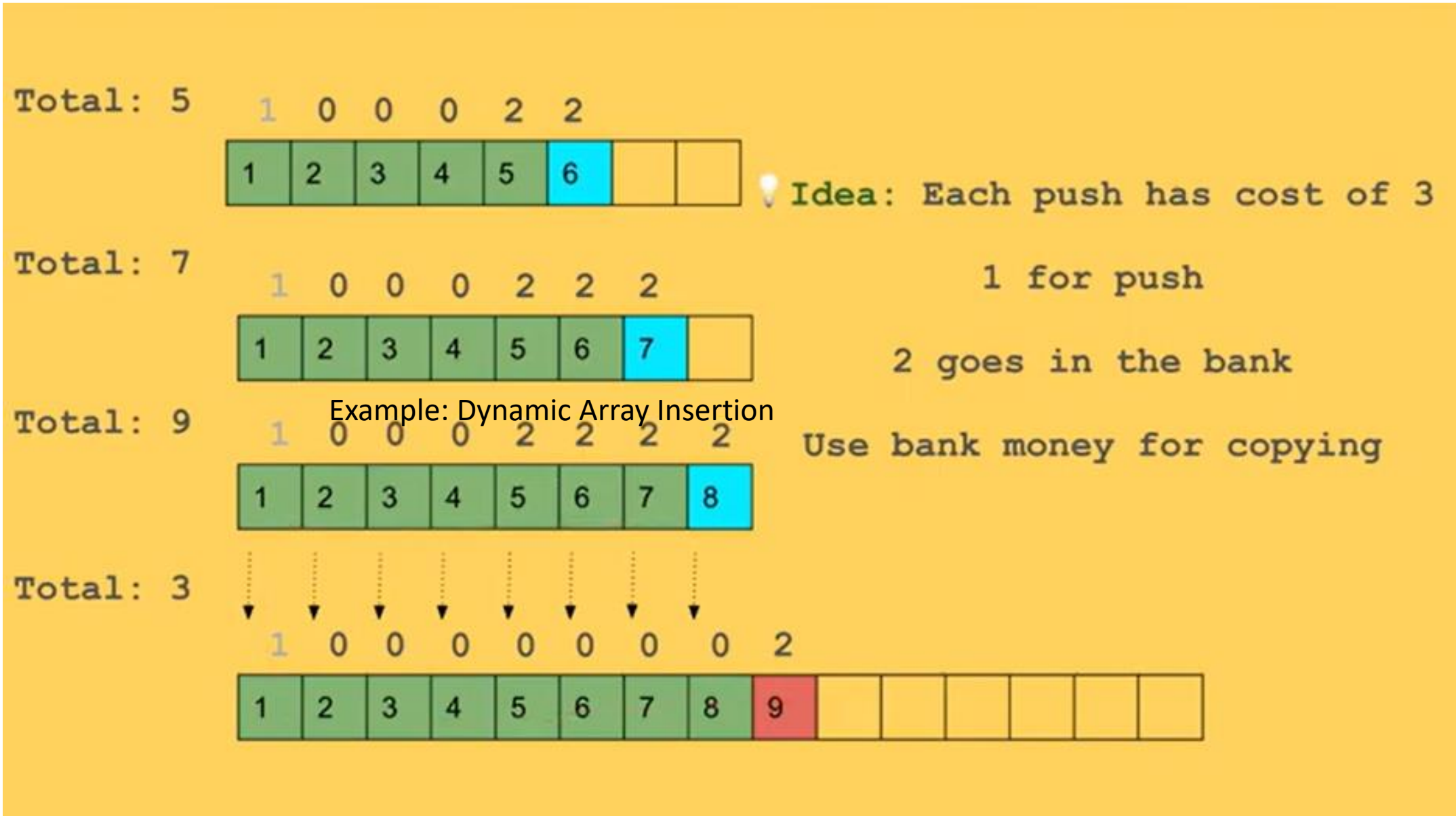
If we count extra cost, we save it in a bank to use it later!

- Careful so that your balance doesn't become negative.

Example: Dynamic Array Insertion (Banker Method)



Example: Dynamic Array Insertion



Because we were able to keep our balance positive, each push is O(3)

Amortized runtime of **each push** is: **O(1)**

Binary Counter (Aggregate Method)

- **Problem:**
  - Given a string of size **n** of all **0's**, implement a binary counter.

"00000000"
"00000001"
"00000010"
"00000011"
"00000100"
"00000101"
"00000110"
"00000111"

Each bit in a binary counter flips when incremented. Incrementing might cause multiple bits to flip, making some operations costly.

Algorithm:

- Start from right to left
- Flip the first consecutive 1's until a 0
- Flip the 0

Question: What is run time of each pass, T(pass i)?

In pass i, we flip at most $\log_2(i)$ bits

"000000**0..00**"

$$1 \leq T(\text{pass } i) \leq \log_2(i)$$

$$T(\text{count to } n) \leq \log_2(1) + \log_2(2) + \ldots + \log_2(n)$$

$$T(\text{count to } n) = O(n.\log(n))$$

15 = 01111
16 = 10000

For i=16, will flip all four bits from 1 to 0, the number of flips is exactly lg(16)=4.

$$1 \leq T(\text{pass } i) \leq \log_2(i)$$

What if we count the total number of flips for counting to n?

Amortized Analysis!

"00000000"

"00000001"

"00000010"

"00000011"

"00000100"

"00000101"

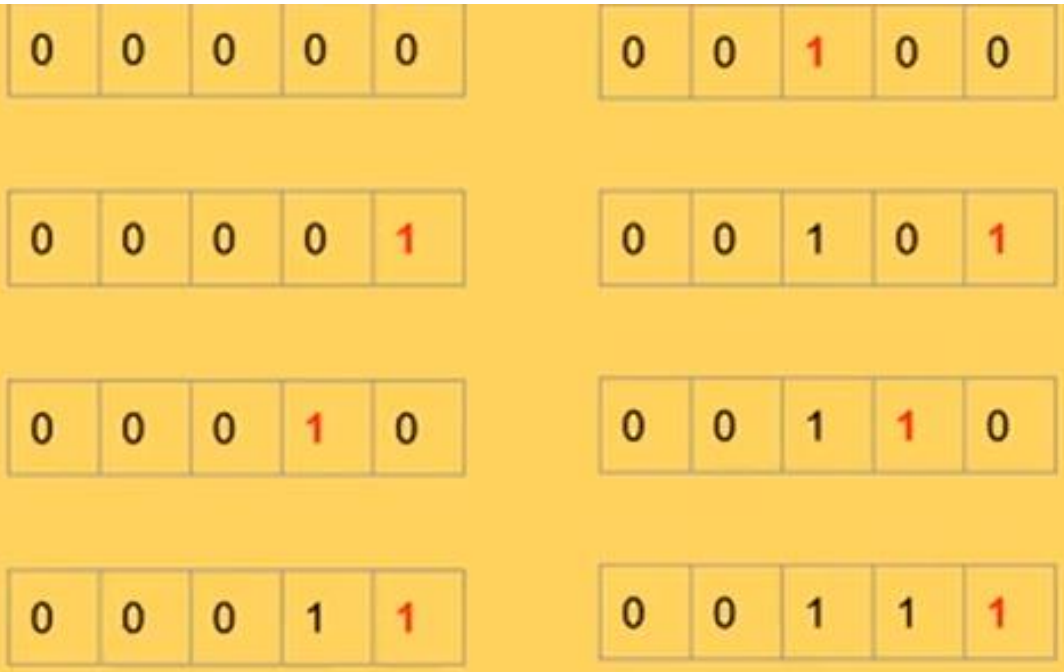
"00000110"

"00000111"

$$T(\text{count to } n) = n + n/2 + n/4 + \ldots = 2n = O(2n) \leq \log_2(i)$$

$$T(\text{pass } i) = T(\text{count to } n)/n = O(2) \leq \log_2(i)$$

Accounting (Banker) Method



Observation: Each time we have at most one Flip-to-1

Idea: What if:

Flip-to-1: cost of 2 (1 to flip, 1 in bank)

Flip-to-0: cost of 1 (Use the bank money)

				1
0	0	0	0	1

			1	
0	0	0	1	0

			1	1
0	0	0	1	1

		1		1
0	0	1	0	1

		1	1	
0	0	1	1	0

		1	1	1
0	0	1	1	1

Amortized `T(pass i) = O(2)`

Summary

- Amortization
  - When the cost of **one operation** is **variable,** calculate cost of **n operations** instead
  - T(1 operation) = T(n operation) / n
  - **Aggregate** method
- Banker Method
  - Spend more, keep extra amount in bank
  - Use bank credit in the future
  - Make sure your bank account is non negative!
- Potential Method

3. Potential Method

Maintain a potential function that reflects the “stored energy” or potential to pay for future operations. The difference in potential helps balance the high-cost operations.

Example: Dynamic Array Insertion (Potential Method)

Define the Potential Function: We'll create a potential function Φ that reflects the "stored energy" of the system. This potential helps us account for the cost of resizing over multiple insertions, rather than having a high cost only when resizing occurs.

Let **n** be the current number of elements in the array.

Let **c** be the current capacity of the array.

Define the potential function Φ as: $\Phi = 2(n - c)$

Intuitively, this potential function accumulates potential energy as the array fills up, ensuring that when

we need to resize, we have enough "saved potential" to cover the cost.

Calculate Amortized Cost per Insertion:

- **The amortized cost $\widehat{T}(i)$ for each insertion is given by:**
- $\widehat{T}(i) = Actual\ Cost + \Delta\phi$

where ΔΦ=Φnew−Φold