Amortized Analysis

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

- In Amortized Analysis, we analyze a sequence of operations and guarantee a worst-case average time that is lower than the worst-case time of a particularly expensive operation.

- The example data structures whose operations are analyzed using Amortized Analysis are Hash Tables, Disjoint Sets, and Splay Trees.

- Amortized analysis is a technique used in computer science to analyze the average-case time complexity of algorithms that perform a sequence of operations, where some operations may be more expensive than others.

- The idea is to spread the cost of these expensive operations over multiple operations, so that the average cost of each operation is constant or less.

- Amortized analysis provides a useful way to analyze algorithms that perform a sequence of operations where some operations are more expensive than others, as it provides a guaranteed upper bound on the average time complexity of each operation, rather than the worst-case time complexity.

- For example, consider the dynamic array data structure that can grow or shrink dynamically as elements are added or removed. The cost of growing the array is proportional to the size of the array, which can be expensive. However, if we amortize the cost of growing the array over several insertions, the average cost of each insertion becomes constant or less.

Aggregate Analysis

In **Amortized Analysis**:

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

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

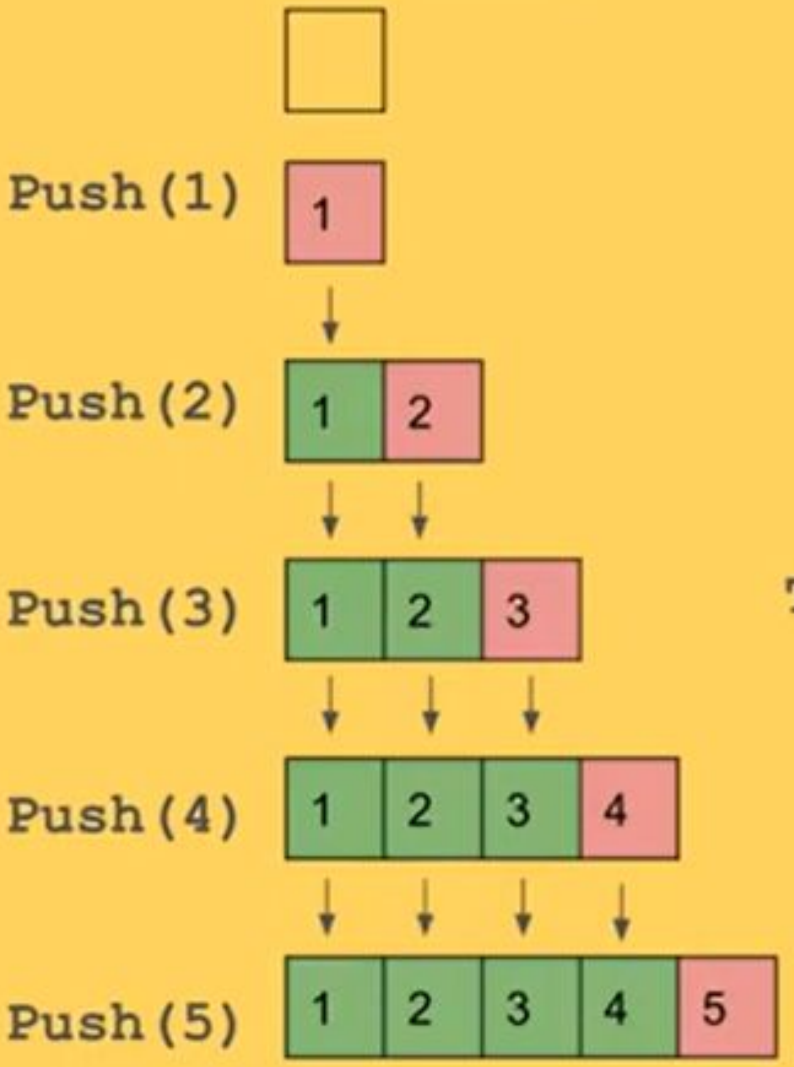
Amortized T(1 operation) = T(n operations)/n

Welcome to Amortized Analysis!

Pushing in a Vector

(Vector Doubling)

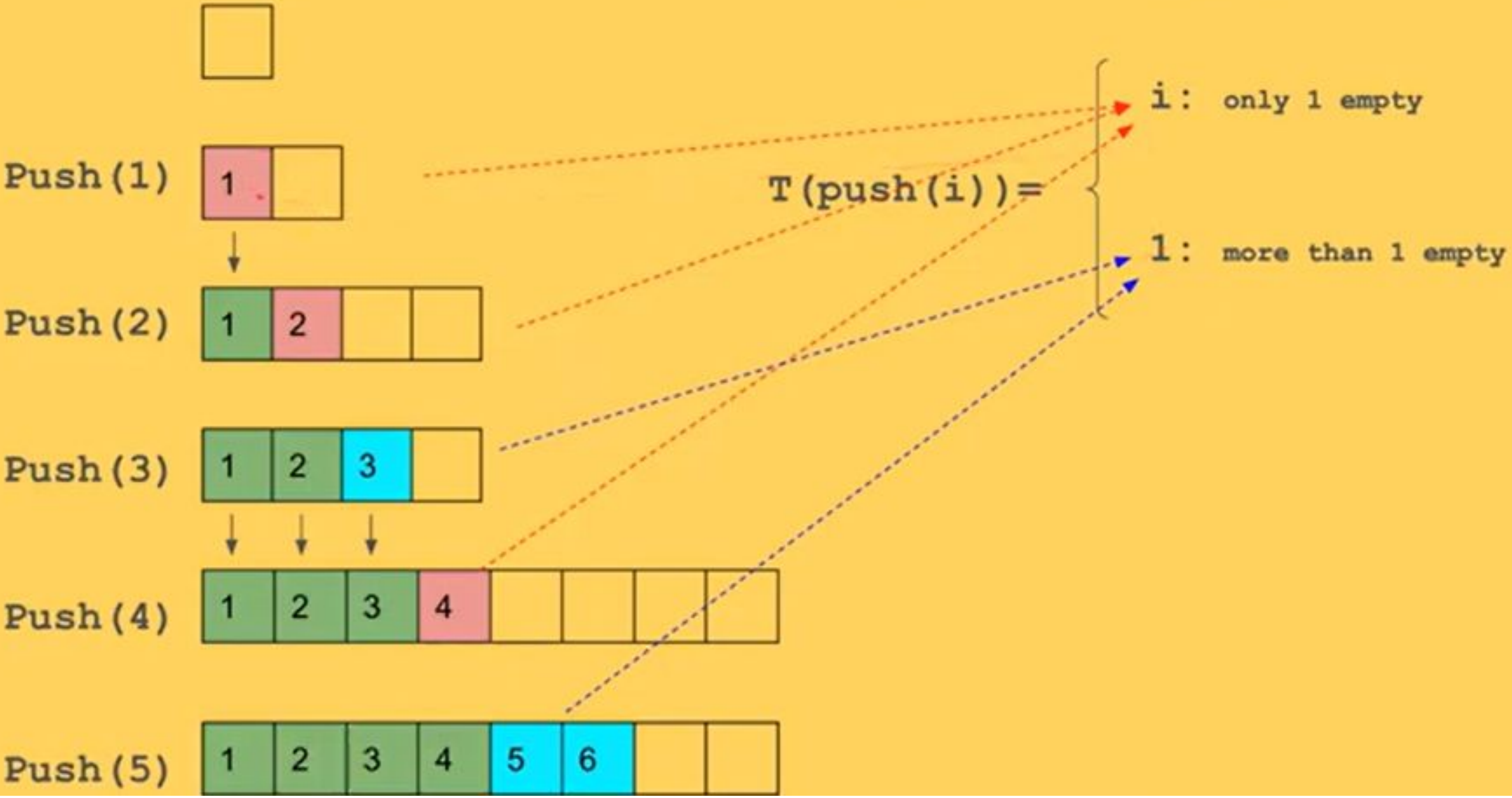
Inserting inside a vector (Simple Method)



T(Push(i))=

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

Inserting inside a vector (Array Doubling)



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

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
1	2	1	4	1	1	1	8	1	1	1	1	1	1	1	16	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	32

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

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

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

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

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

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

$$T(n\ Push) \le C_1 * n \le C_2 * n^2$$

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

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

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

Accounting (Banker) Method

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

Example 1 : Stack operations :

We know, Actual costs of stack operations

PUSH	1
POP	1
MULTIPOP	min(S, K)

Lets assign the following amortized cost.

PUSH	2
POP	0
MULTIPOP	0

Lets analyze a sequence of 4 push, pop on an initially empty stack.

Operations	Amortized cost	Actual cost	Credit
Push (A,S)	2	1	1
Push (B,S)	2	1	1+1 = 2
Pop (S)	0	1	2-1 = 1
Push (C,S)	2	1	1+1 = 2
Pop (S)	0	1	2-1 = 1
Pop (S)	0	1	1-1 = 0
Push (D,S)	2	1	0+1 = 1
Pop (S)	0	1	1-1 = 0

Therefore, Total amortized cost $= (2+2+0+2+0+0+2+0)$

$= 8$ (for 4 push operation

$= 2 \times 4$

for n operation, $= 2n$

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

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

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

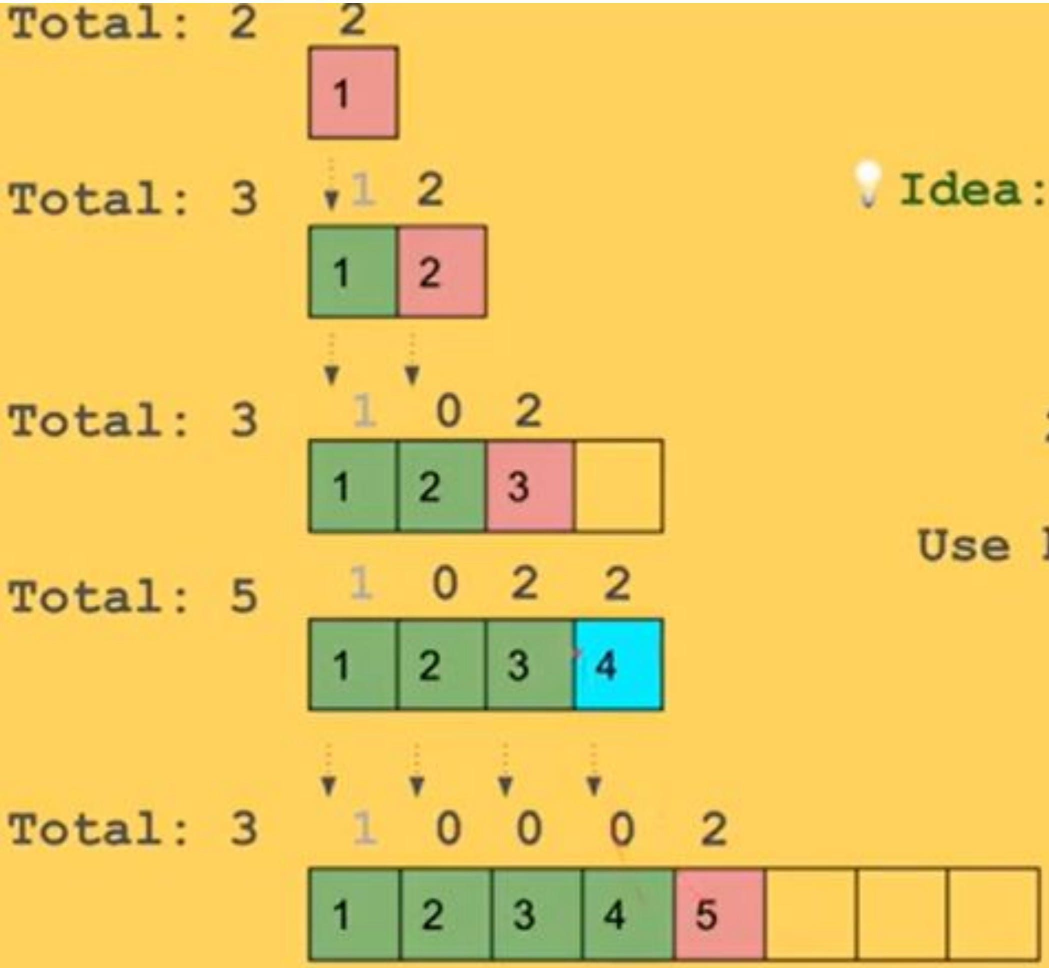
If we count **extra cost**, we save it in a **bank** to use it later!

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

Your balance is:

- $100.00

⚠ Add money to resolve the issue!

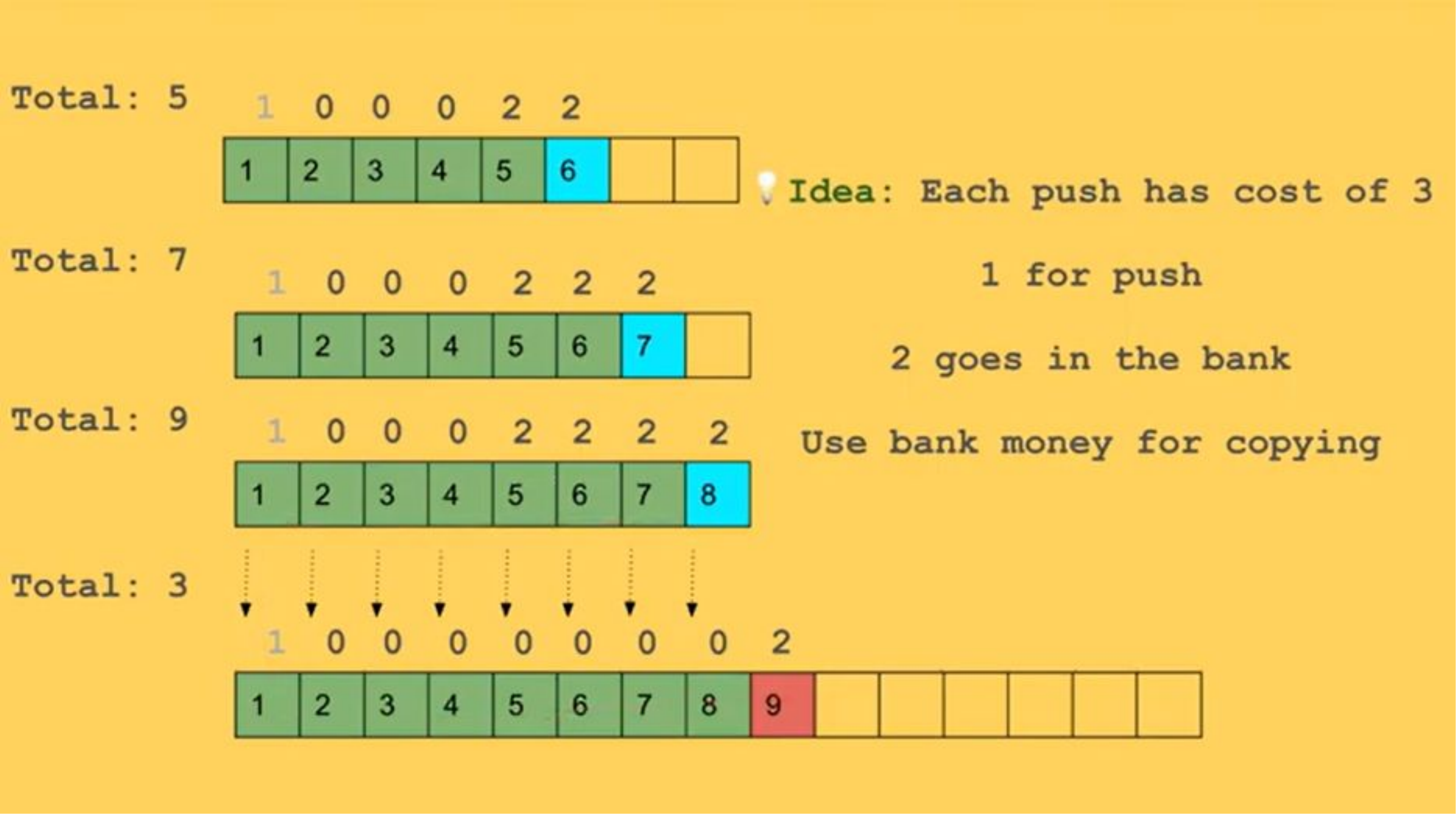


Idea: Each push has cost of 3

1 for push

2 goes in the bank

Use bank money for copying


Total: 5
1 0 0 0 2 2
1 2 3 4 5 6
Idea: Each push has cost of 3
1 for push
2 goes in the bank
Use bank money for copying
Total: 7
1 0 0 0 2 2 2
1 2 3 4 5 6 7
Total: 9
1 0 0 0 2 2 2 2
1 2 3 4 5 6 7 8
Total: 3
1 0 0 0 0 0 0 0 2
1 2 3 4 5 6 7 8 9

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

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

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

"00000000"

"00000001"

"00000010"

"00000011"

"00000100"

"00000101"

"00000110"

"00000111"

💡**Algorithm:**

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

Pseudo code :

```
INCREMENT (A)
    i = 0
    while i < A.length and A[i] == 1
        A[i] = 0
        i = i + 1
    If i < A.length
        A[i] = 1
```

Counter value	A[3]	A[2]	A[1]	A[0]	Cost	Total cost
0	0	0	0	0	0	0
1	0	0	0	1	1	0+1=1
2	0	0	1	0	2	1+2=3
3	0	0	1	1	1	3+1=4
4	0	1	0	0	3	4+3=7
5	0	1	0	1	1	7+1=8
6	0	1	1	0	2	8+2=10
7	0	1	1	1	1	10+1=11

Asymptotic Analysis :

We observe some operations only flip one bit. and some operations flip more than one bit.

A single execution of INCREMENT takes time $O(k)$ in the worst case.

Thus a sequence of n INCREMENT operations on an initially zero counter takes time $O(nk)$ in the worst-case.

Aggregate Analysis or Amortized Analysis :

We can observe that not all the bits flip each time INCREMENT.

A[0] flip each time INCREMENT.

A[1] flips only every other time

i.e., A[1] to flip $\frac{n}{2}$ time.

Similarily,

A[2] flips every fourth time or $\frac{n}{4}$ time in a sequence of n INCREMENT operations.

"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

0	0	0	0	0	0	0	1	0	0
0	0	0	0	1	0	0	1	0	1
0	0	0	1	0	0	0	1	1	0
0	0	0	1	1	0	0	1	1	1

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)