**CS**11921 - **Fall** 2023

# Algorithm
# Design & Analysis

## Amortized Analysis

Ibrahim Albluwi

# Motivation

Problem. Given an array $B[0 \ \ldots \ k-1]$ of bits, representing a number $n < 2^k$, increment $n$.

# Motivation

Problem. Given an array $B[0 \ \ldots \ k-1]$ of bits, representing a number $n < 2^k$, increment $n$.

Examples. $\texttt{0000}\mathbf{0} \xrightarrow{\texttt{INCREMENT}} \texttt{0000}\mathbf{1}$

$n = 0, \ k = 5$ $\qquad$ $n = 1, \ k = 5$

# Motivation

Problem. Given an array $B[0 \ \ldots \ k-1]$ of bits, representing a number $n < 2^k$, increment $n$.

Examples.

$$\texttt{00000} \xrightarrow{\texttt{INCREMENT}} \underset{n = 1,\ k = 5}{\underset{\uparrow}{\texttt{0000}\mathbf{1}}} \xrightarrow{\texttt{INCREMENT}} \underset{n = 2,\ k = 5}{\underset{\uparrow}{\texttt{000}\mathbf{10}}}$$

# Motivation

Problem. Given an array $B[0 \ \ldots \ k-1]$ of bits, representing a number $n < 2^k$, increment $n$.

Examples. $\texttt{00000} \xrightarrow{\texttt{INCREMENT}} \texttt{0000}\textbf{1} \xrightarrow{\texttt{INCREMENT}} \texttt{000}\textbf{10}$

```
INCREMENT(B[], k)
    i = k-1
```

start at the rightmost bit

# Motivation

Problem. Given an array $B[0 \;\ldots\; k-1]$ of bits, representing a number $n < 2^k$, increment $n$.

Examples. $00000 \xrightarrow{\texttt{INCREMENT}} 00001 \xrightarrow{\texttt{INCREMENT}} 00010$

```
INCREMENT(B[], k)
    i = k-1
    while (B[i] == 1 and i >= 0):
        B[i] = 0
        i = i-1
```

keep flipping 1's to 0's until a 0 is reached

# Motivation

Problem. Given an array $B[0 \; \ldots \; k-1]$ of bits, representing a number $n < 2^k$, increment $n$.

Examples. $\texttt{00000} \xrightarrow{\texttt{INCREMENT}} \texttt{00001} \xrightarrow{\texttt{INCREMENT}} \texttt{00010}$

```
INCREMENT(B[], k)

  i = k-1
  while (B[i] == 1 and i >= 0):
      B[i] = 0
      i = i-1
  if (i >= 0)
      B[i] = 1
```

keep flipping `1`'s to `0`'s
until a `0` is reached

# Motivation

Problem. Given an array $B[0 \;\ldots\; k-1]$ of bits, representing a number $n < 2^k$, increment $n$.

Examples. $\texttt{00000} \xrightarrow{\texttt{INCREMENT}} \texttt{00001} \xrightarrow{\texttt{INCREMENT}} \texttt{00010}$

```
INCREMENT(B[], k)

    i = k-1

    while (B[i] == 1 and i >= 0):
        B[i] = 0
        i = i-1

    if (i >= 0)
        B[i] = 1
```

Example. `1 0 1 0 1 0 0 1 1 1 1 1 1`

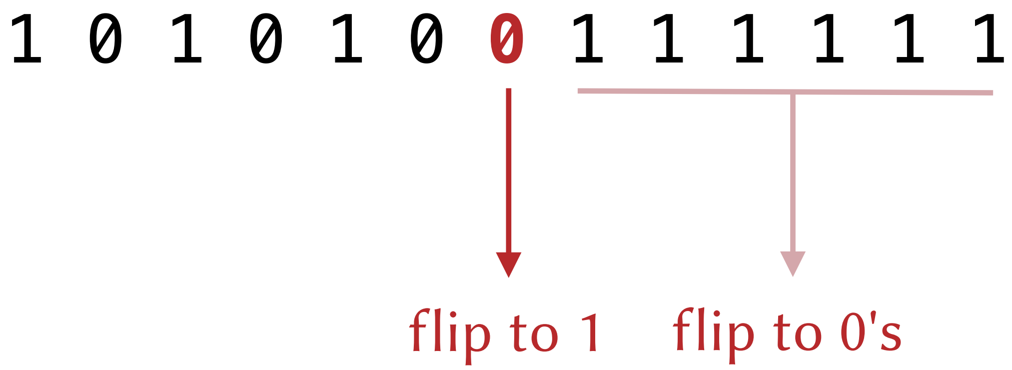
# Motivation

Problem. Given an array $B[0 \ \ldots \ k-1]$ of bits, representing a number $n < 2^k$, increment $n$.

Examples. $\texttt{00000} \xrightarrow{\texttt{INCREMENT}} \texttt{00001} \xrightarrow{\texttt{INCREMENT}} \texttt{00010}$

```
INCREMENT(B[], k)

    i = k-1

    while (B[i] == 1 and i >= 0):
        B[i] = 0
        i = i-1

    if (i >= 0)
        B[i] = 1
```

Example. 1 0 1 0 1 0 0 1 1 1 1 1 1



# Motivation

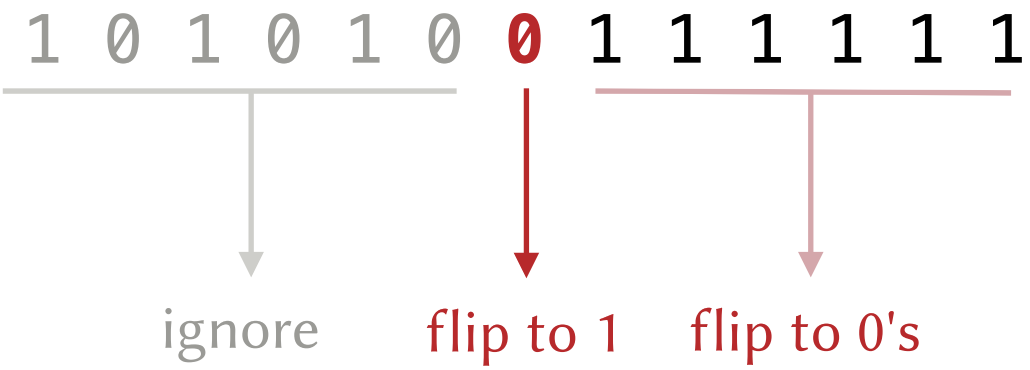
Problem. Given an array $B[0 \ \ldots \ k-1]$ of bits, representing a number $n < 2^k$, increment $n$.

Examples. $00000 \xrightarrow{\texttt{INCREMENT}} 00001 \xrightarrow{\texttt{INCREMENT}} 00010$

**INCREMENT(B[], k)**

```
i = k-1

while (B[i] == 1 and i >= 0):
    B[i] = 0
    i = i-1

if (i >= 0)
    B[i] = 1
```

Example.



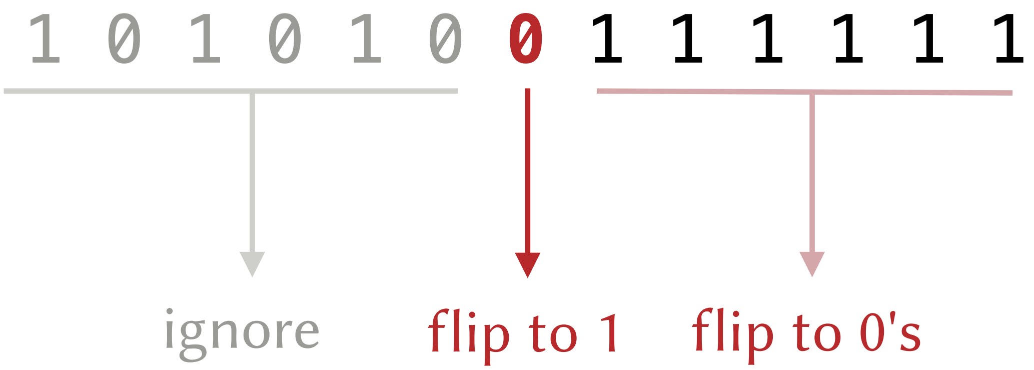
# Motivation

Problem. Given an array $B[0 \ \ldots \ k-1]$ of bits, representing a number $n < 2^k$, increment $n$.

Examples. $\texttt{00000} \xrightarrow{\texttt{INCREMENT}} \texttt{00001} \xrightarrow{\texttt{INCREMENT}} \texttt{00010}$

```
INCREMENT(B[], k)
    i = k-1
    while (B[i] == 1 and i >= 0):
        B[i] = 0
        i = i-1
    if (i >= 0)
        B[i] = 1
```

Example.



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

ignore

flip to 1

flip to 0's

**Q** What is the running time of function **`INCREMENT`**?
Choose the *best* answer.

Cost Model. Count the number of *bit flips*.

**A.** $O(1)$

**B.** $O(k)$

**C.** $O(\log n)$

**D.** $O(n)$

# Motivation

Problem. Given an array $B[0 \;\ldots\; k-1]$ of bits, representing a number $n < 2^k$, increment $n$.

Examples. 00000 $\xrightarrow{\texttt{INCREMENT}}$ 0000**1** $\xrightarrow{\texttt{INCREMENT}}$ 000**10**

```
INCREMENT(B[], k)
    i = k-1
    while (B[i] == 1 and i >= 0):
        B[i] = 0
        i = i-1
    if (i >= 0)
        B[i] = 1
```

Example. 1 0 1 0 1 0 **0** 1 1 1 1 1 1

1 0 1 0 1 0 → ignore

0 → flip to 1

1 1 1 1 1 1 → flip to 0's

**Q** What is the running time of function **INCREMENT**?
Choose the *best* answer.

Cost Model. Count the number of *bit flips.*

A. $O(1)$ ← incorrect

B. $O(k)$ ← too pessimistic!

[✓ C. $O(\log n)$]

D. $O(n)$ ← too pessimistic!

# Motivation

What is the running time for counting from `0` to `n` by calling function **`INCREMENT`** repeatedly on an array of $k$ bits initialized to `0`'s?

Cost Model. Count the number of *bit flips*.

Choose the *best* answer.

- **A.** $O(n)$
- **B.** $O(n \log n)$
- **C.** $O(nk)$
- **D.** $O(k \log n)$

# Motivation

What is the running time of counting from `0` to `n` by calling function **`INCREMENT`** repeatedly on an array of $k$ bits initialized to `0`'s?

Cost Model. Count the number of *bit flips*.

Choose the *best* answer.

😁 $O(n)$ correct and tight bound!

**B.** $O(n \log n)$

**C.** $O(nk)$

correct but too pessimistic!

**D.** $O(k \log n)$ incorrect!

# Motivation

What is the running time of counting from `0` to `n` by calling function **`INCREMENT`** repeatedly on an array of $k$ bits initialized to `0`'s?

Cost Model. Count the number of *bit flips*.

Choose the *best* answer.

😁 $O(n)$ correct and tight bound! … *why*?

**B.** $O(n \log n)$

**C.** $O(nk)$

correct but too pessimistic!

**D.** $O(k \log n)$ incorrect!

🤔 Why is it pessimistic to say: $n$ calls to **`INCREMENT`** $\times O(\log n) = O(n \log n)$?

**Answer.** Because each call to **`INCREMENT`** does not do $O(\log n)$ bit flips!

# **Motivation:** Counting Bit-Flips

Problem. What is the total number of bit-flips performed when counting from `0` to `n` by calling function **`INCREMENT`** repeatedly on an array of $k$ bits initialized to `0`'s?

Example. Counting to `15`.

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

# **Motivation:** Counting Bit-Flips

Problem. What is the total number of bit-flips performed when counting from `0` to `n` by calling function **`INCREMENT`** repeatedly on an array of $k$ bits initialized to `0`'s?

Example. Counting to `15`.

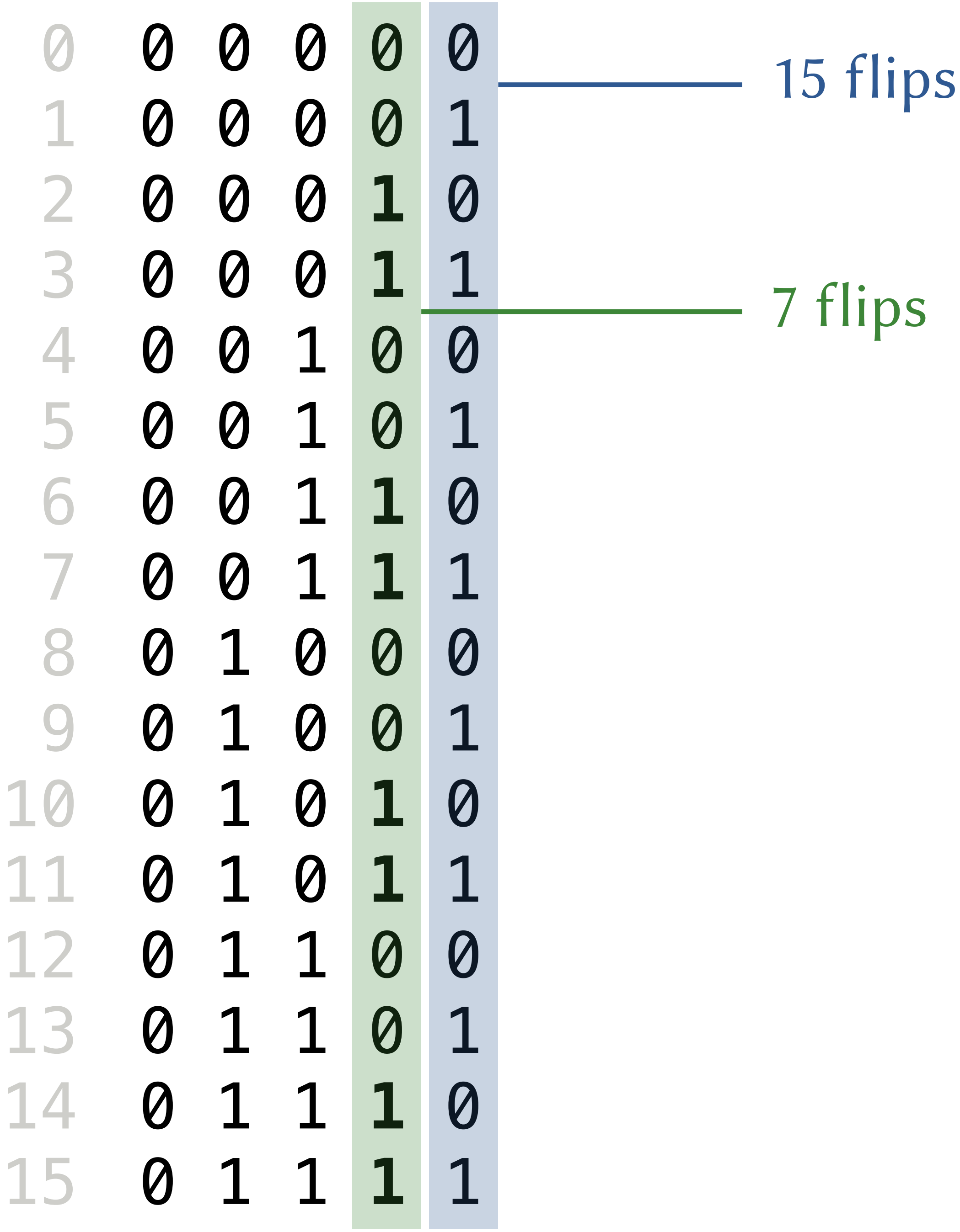
| | | | | | |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 1 |
| 2 | 0 | 0 | 0 | 1 | 0 |
| 3 | 0 | 0 | 0 | 1 | 1 |
| 4 | 0 | 0 | 1 | 0 | 0 |
| 5 | 0 | 0 | 1 | 0 | 1 |
| 6 | 0 | 0 | 1 | 1 | 0 |
| 7 | 0 | 0 | 1 | 1 | 1 |
| 8 | 0 | 1 | 0 | 0 | 0 |
| 9 | 0 | 1 | 0 | 0 | 1 |
|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

15 flips

# **Motivation:** Counting Bit-Flips

Problem. What is the total number of bit-flips performed when counting from `0` to `n` by calling function **`INCREMENT`** repeatedly on an array of $k$ bits initialized to `0`'s?

Example. Counting to `15`.

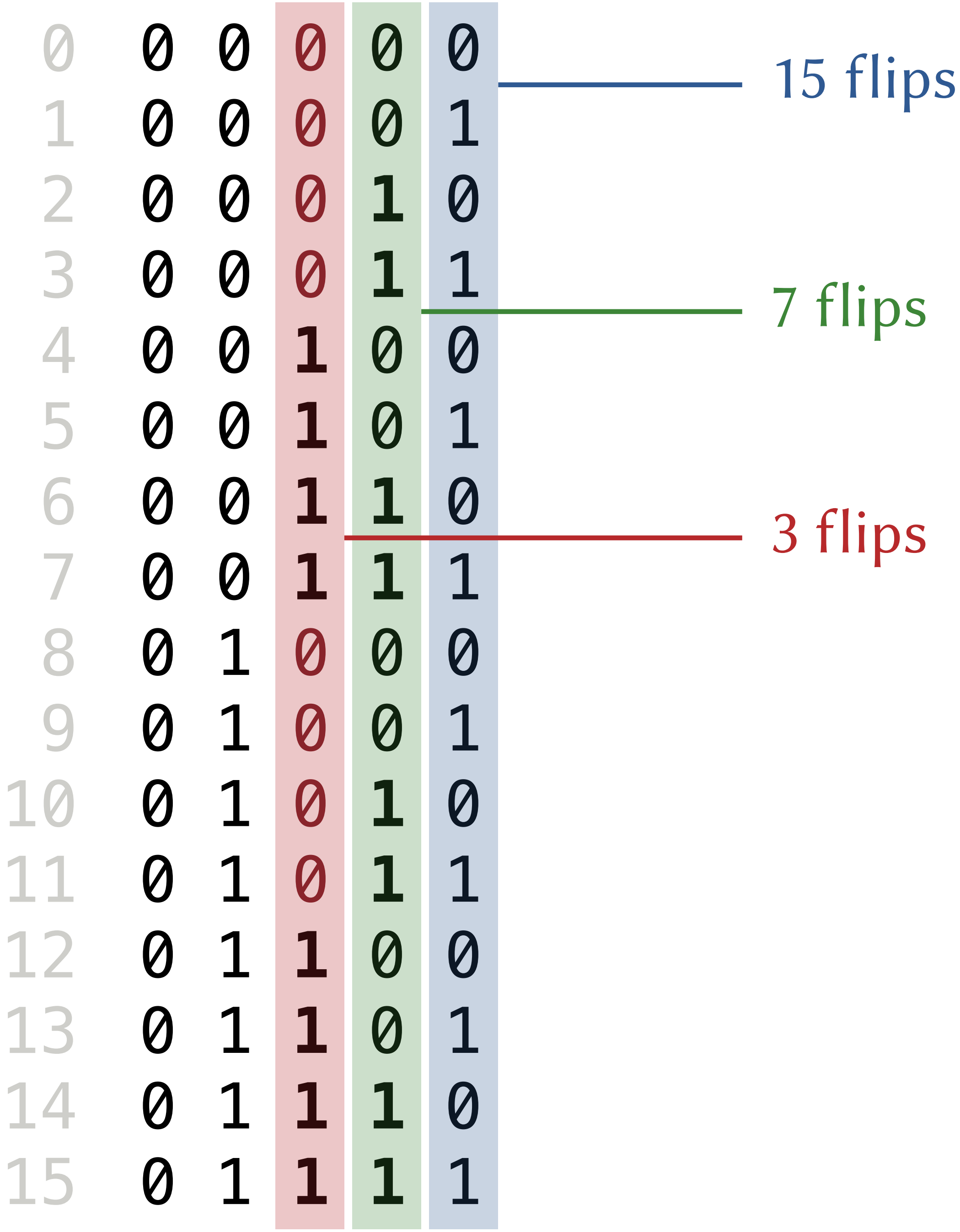


| | | | | | |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 1 |
| 2 | 0 | 0 | 0 | **1** | 0 |
| 3 | 0 | 0 | 0 | **1** | 1 |
| 4 | 0 | 0 | 1 | 0 | 0 |
| 5 | 0 | 0 | 1 | 0 | 1 |
| 6 | 0 | 0 | 1 | **1** | 0 |
| 7 | 0 | 0 | 1 | **1** | 1 |
| 8 | 0 | 1 | 0 | 0 | 0 |
| 9 | 0 | 1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

# **Motivation:** Counting Bit-Flips

Problem. What is the total number of bit-flips performed when counting from `0` to `n` by calling function **`INCREMENT`** repeatedly on an array of $k$ bits initialized to `0`'s?

Example. Counting to 15.



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

15 flips

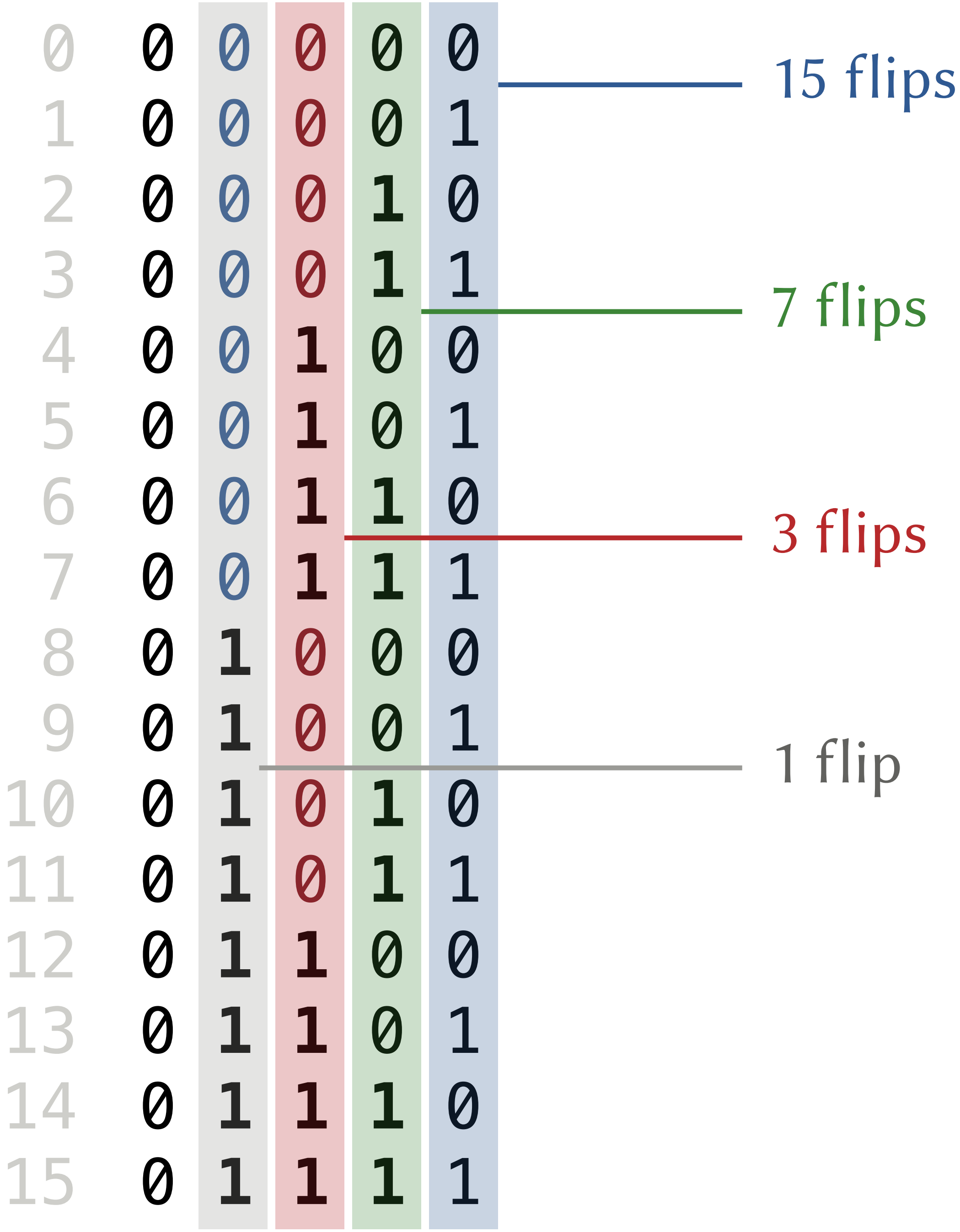
7 flips

3 flips

# **Motivation:** Counting Bit-Flips

Problem. What is the total number of bit-flips performed when counting from `0` to `n` by calling function **`INCREMENT`** repeatedly on an array of $k$ bits initialized to `0`'s?

Example. Counting to `15`.

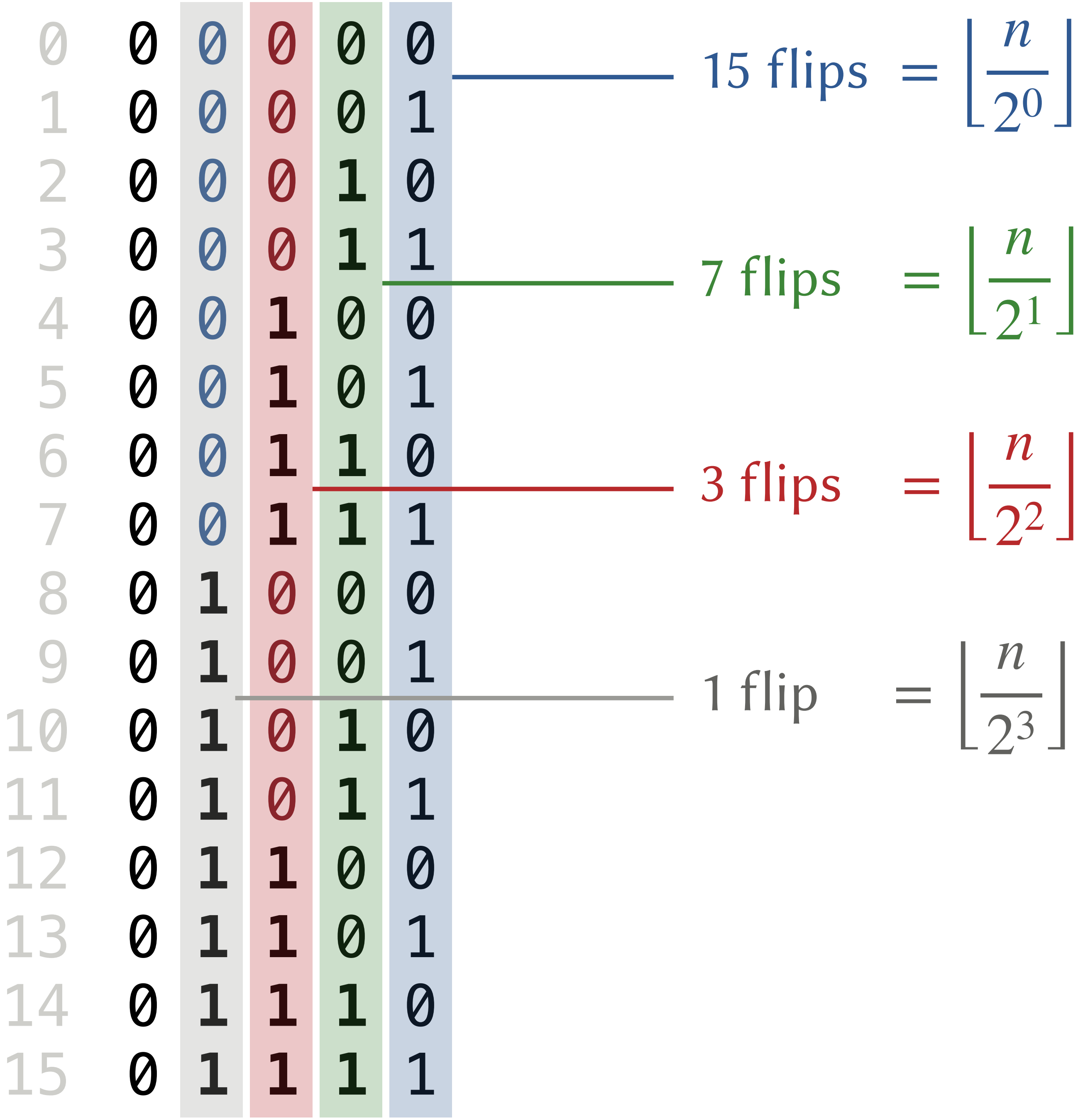


| | | | | | |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 1 |
| 2 | 0 | 0 | 0 | 1 | 0 |
| 3 | 0 | 0 | 0 | 1 | 1 |
| 4 | 0 | 0 | 1 | 0 | 0 |
| 5 | 0 | 0 | 1 | 0 | 1 |
| 6 | 0 | 0 | 1 | 1 | 0 |
| 7 | 0 | 0 | 1 | 1 | 1 |
| 8 | 0 | 1 | 0 | 0 | 0 |
| 9 | 0 | 1 | 0 | 0 | 1 |
|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

# **Motivation:** Counting Bit-Flips

Problem. What is the total number of bit-flips performed when counting from `0` to `n` by calling function **`INCREMENT`** repeatedly on an array of $k$ bits initialized to `0`'s?

Example. Counting to `15`.



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

15 flips $= \left\lfloor \frac{n}{2^0} \right\rfloor$

7 flips $= \left\lfloor \frac{n}{2^1} \right\rfloor$

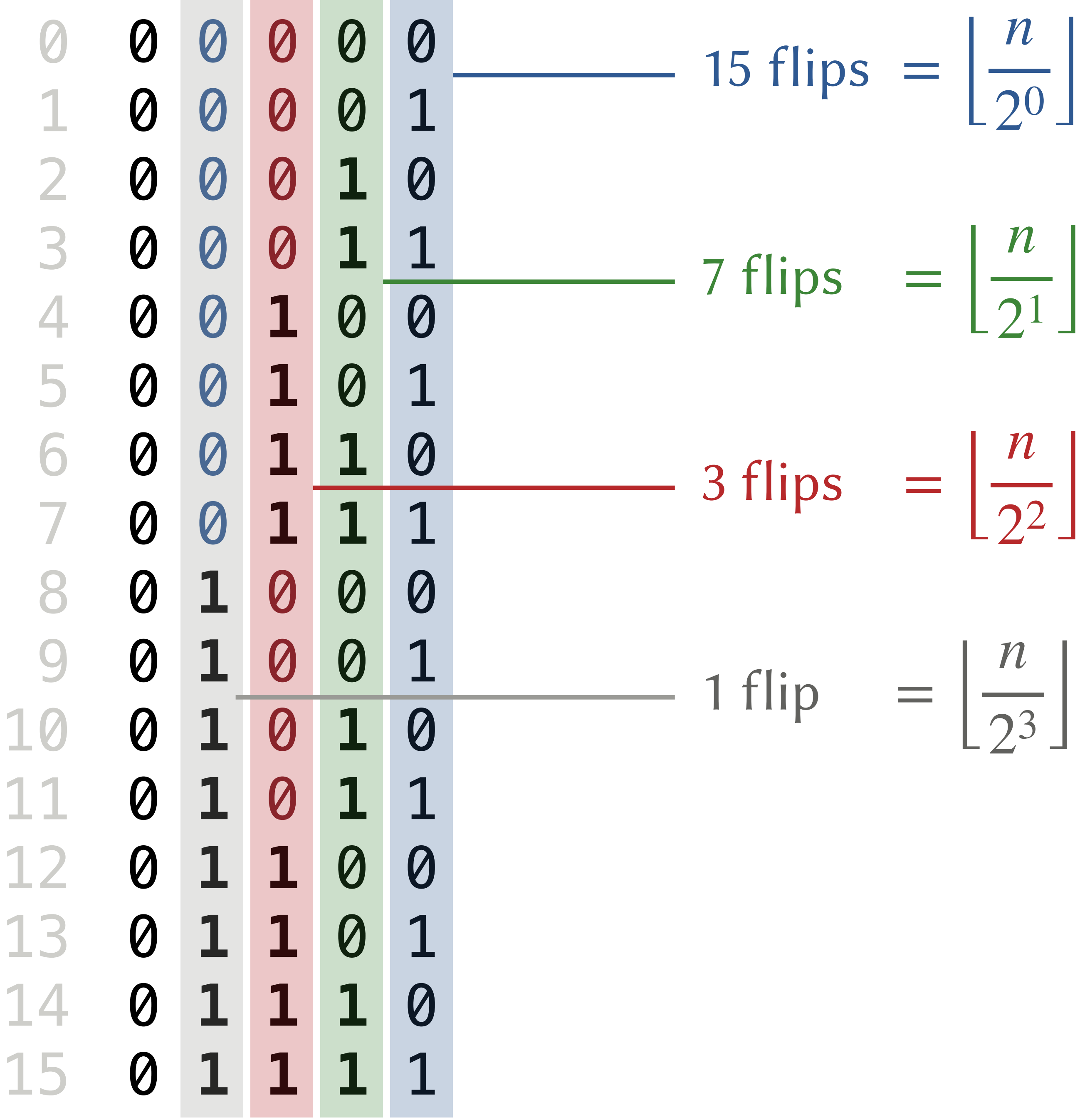
3 flips $= \left\lfloor \frac{n}{2^2} \right\rfloor$

1 flip $= \left\lfloor \frac{n}{2^3} \right\rfloor$

# **Motivation:** Counting Bit-Flips

Problem. What is the total number of bit-flips performed when counting from `0` to `n` by calling function **`INCREMENT`** repeatedly on an array of $k$ bits initialized to `0`'s?

Example. Counting to 15.



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

15 flips $= \left\lfloor \frac{n}{2^0} \right\rfloor$

7 flips $= \left\lfloor \frac{n}{2^1} \right\rfloor$

3 flips $= \left\lfloor \frac{n}{2^2} \right\rfloor$

1 flip $= \left\lfloor \frac{n}{2^3} \right\rfloor$

**In general.**

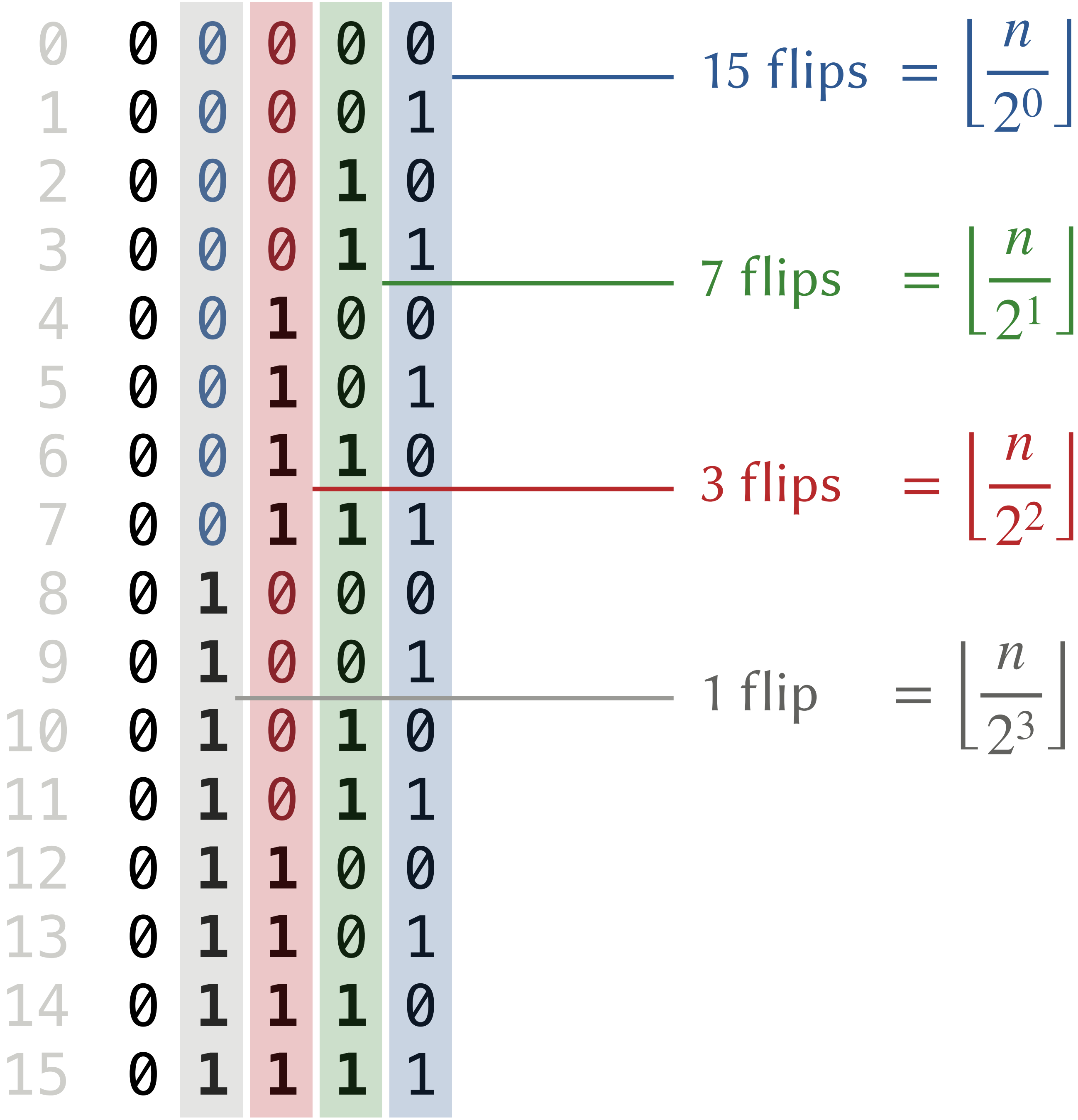
The total number of bit flips is:

$$\leq \sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor$$

# **Motivation:** Counting Bit-Flips

Problem. What is the total number of bit-flips performed when counting from `0` to `n` by calling function **`INCREMENT`** repeatedly on an array of $k$ bits initialized to `0`'s?

Example. Counting to `15`.



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

15 flips $= \left\lfloor \frac{n}{2^0} \right\rfloor$

7 flips $= \left\lfloor \frac{n}{2^1} \right\rfloor$

3 flips $= \left\lfloor \frac{n}{2^2} \right\rfloor$

1 flip $= \left\lfloor \frac{n}{2^3} \right\rfloor$

**In general.**

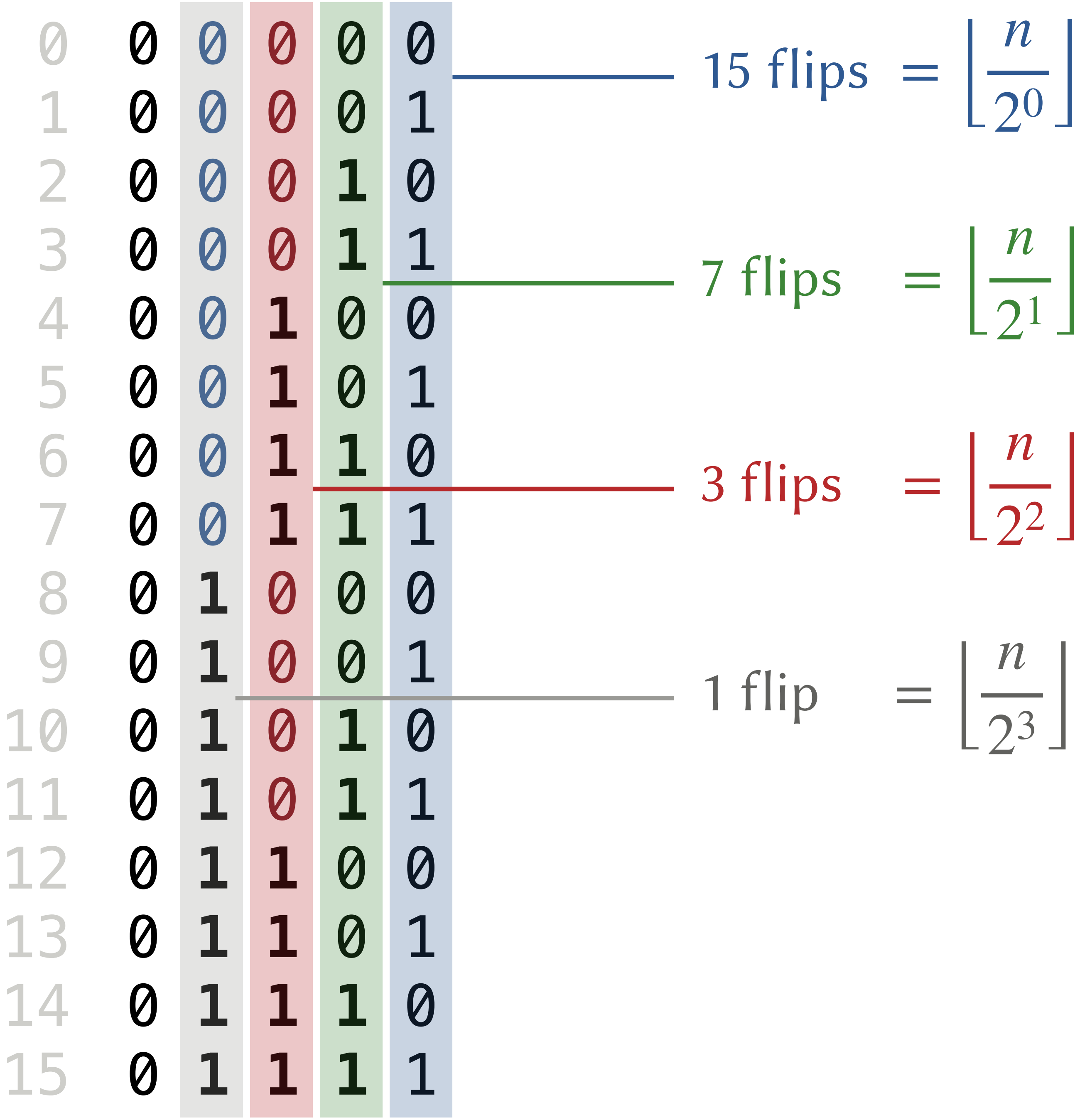
The total number of bit flips is:

$$\leq \sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor \quad \leq \quad n \sum_{i=0}^{\infty} (\tfrac{1}{2})^i$$

# **Motivation:** Counting Bit-Flips

Problem. What is the total number of bit-flips performed when counting from `0` to `n` by calling function **`INCREMENT`** repeatedly on an array of $k$ bits initialized to `0`'s?

Example. Counting to `15`.



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

15 flips $= \left\lfloor \frac{n}{2^0} \right\rfloor$

7 flips $= \left\lfloor \frac{n}{2^1} \right\rfloor$

3 flips $= \left\lfloor \frac{n}{2^2} \right\rfloor$

1 flip $= \left\lfloor \frac{n}{2^3} \right\rfloor$

**In general.**

The total number of bit flips is:

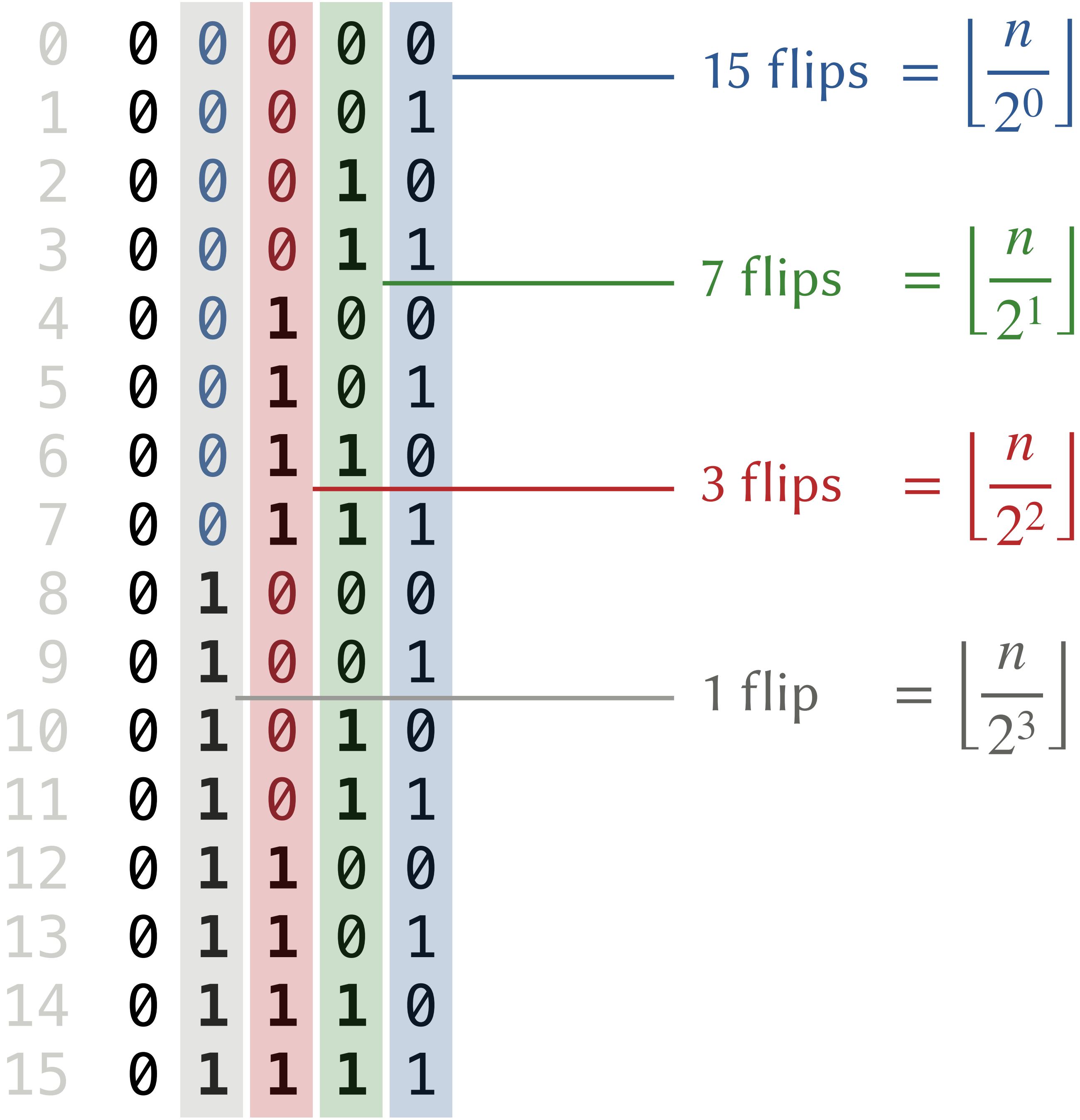
$$\leq \sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor \quad \leq \; n \sum_{i=0}^{\infty} \left(\tfrac{1}{2}\right)^i$$

$$\leq \; 2n \;\; = \;\; O(n)$$

# **Motivation:** Counting Bit-Flips

Problem. What is the total number of bit-flips performed when counting from `0` to `n` by calling function **`INCREMENT`** repeatedly on an array of $k$ bits initialized to `0`'s?

Example. Counting to `15`.



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

15 flips $= \left\lfloor \frac{n}{2^0} \right\rfloor$

7 flips $= \left\lfloor \frac{n}{2^1} \right\rfloor$

3 flips $= \left\lfloor \frac{n}{2^2} \right\rfloor$

1 flip $= \left\lfloor \frac{n}{2^3} \right\rfloor$

**In general.**

The total number of bit flips is:

$$\le \sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor \le n \sum_{i=0}^{\infty} \left(\tfrac{1}{2}\right)^i$$

$$\le 2n = O(n)$$

**Implication.**

Since **`INCREMENT`** is called $n$ times and the running time is $O(n)$ in total, the running time of each call to **`INCREMENT`** in the sequence of calls is $O(1)$ *on average!*

**Takeaway**

When analyzing the worst case running time of a *sequence* of operations, we can:

**Takeaway** When analyzing the worst case running time of a *sequence* of operations, we can:

① Analyze the worst case running time of a single operation and then multiply it by the number of times the operation is performed.

Example. Running time of $n$ increments $= n \times O(\log n)$

Problem. Might *overestimate* the worst case running time.

**Takeaway** When analyzing the worst case running time of a *sequence* of operations, we can:

① Analyze the worst case running time of a single operation and then multiply it by the number of times the operation is performed.

Example. Running time of $n$ increments $= n \times O(\log n)$

Problem. Might *overestimate* the worst case running time.

OR

② Reason about the total running time of the whole sequence of operations together.

Example. Incrementing $n$ times can't flip bits more than $2n$ times.

# Motivation

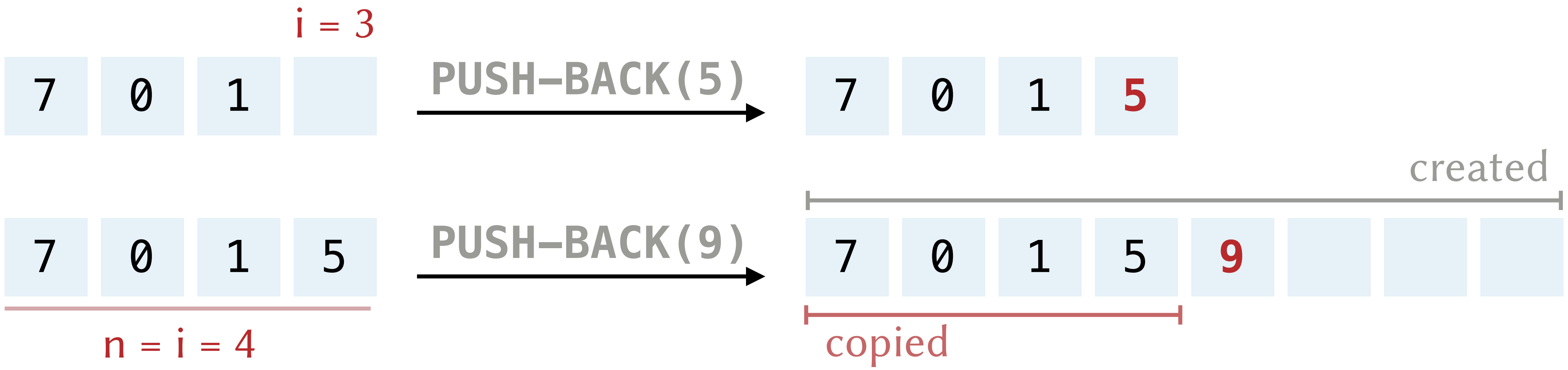
Problem. Given an array $A[]$ of size $n$, implement a **`PUSH-BACK`**`(x)` operation that inserts $x$ at the last vacant cell. If the number of occupied cells $i$ equals $n$, double the size of the array before inserting.

Example.

# Motivation

Problem. Given an array $A[]$ of size $n$, implement a **`PUSH-BACK`**`(x)` operation that inserts $x$ at the last vacant cell. If the number of occupied cells $i$ equals $n$, double the size of the array before inserting.

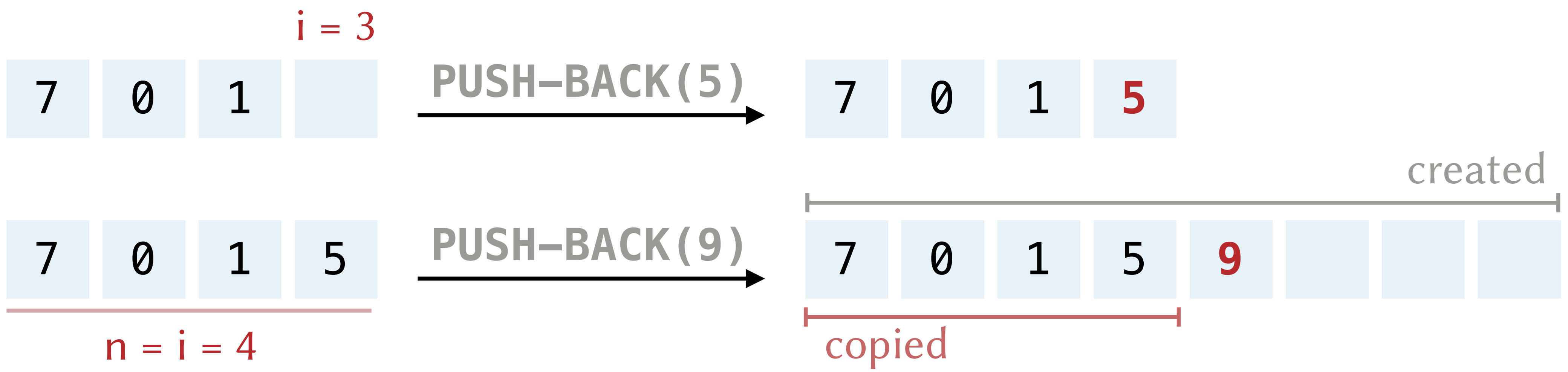
Example.



# Motivation

Problem. Given an array $A[]$ of size $n$, implement a **PUSH-BACK**`(x)` operation that inserts $x$ at the last vacant cell. If the number of occupied cells $i$ equals $n$, double the size of the array before inserting.

Example.



Assuming `n` and `i` are globally accessible.

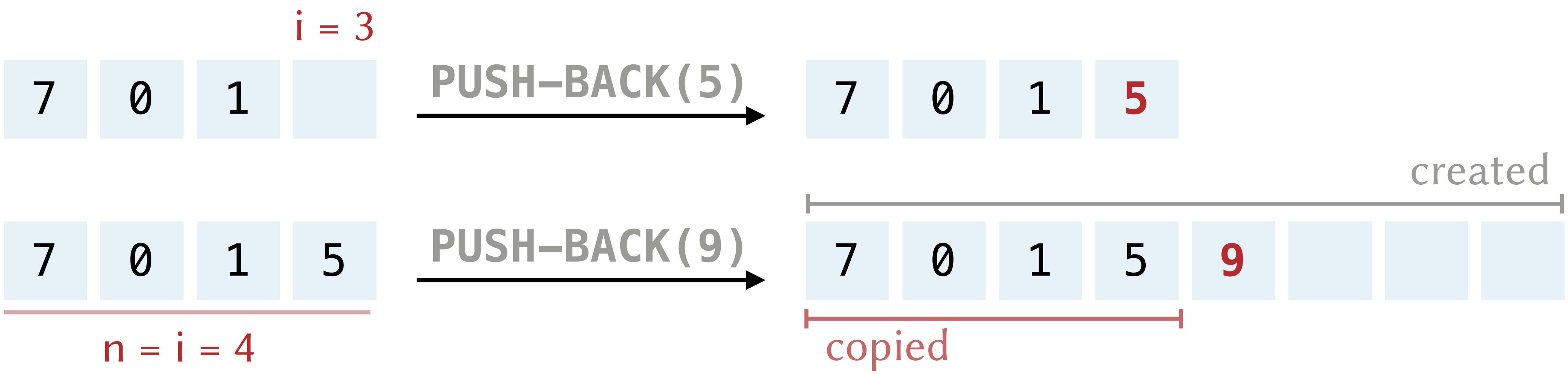
```
PUSH-BACK(A[], x)

if (i < n): A[i] ← x
```

simply insert at index `i` if there is a vacant cell

# Motivation

Problem. Given an array $A[]$ of size $n$, implement a **PUSH–BACK**(x) operation that inserts $x$ at the last vacant cell. If the number of occupied cells $i$ equals $n$, double the size of the array before inserting.

Example.



Assuming `n` and `i` are globally accessible.

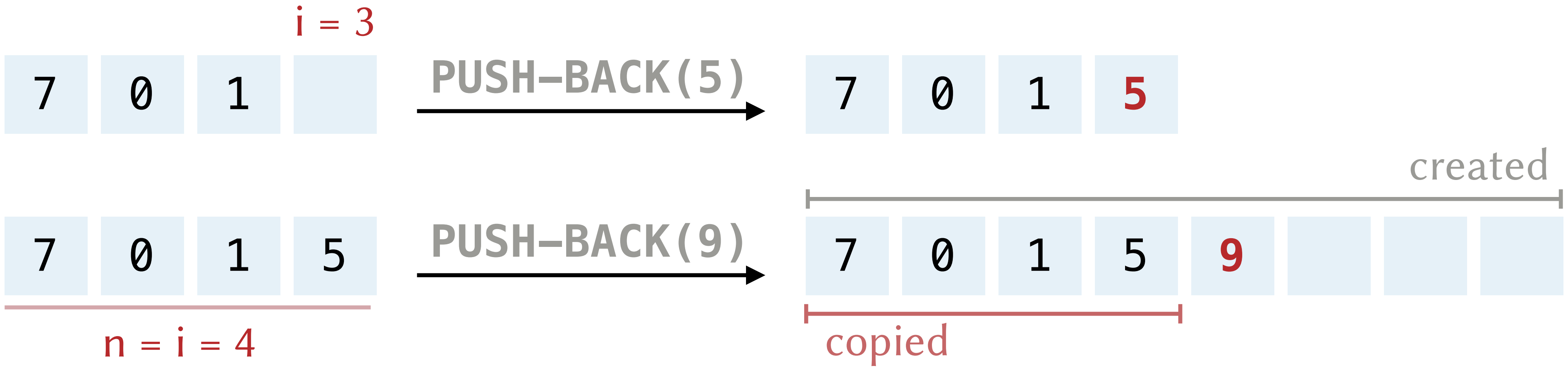
```
PUSH–BACK(A[], x)

if (i < n): A[i] ← x
else: Create array B of size n x 2
      B[0 … n−1] ← A[0 … n−1]
      A ← B
      n ← n x 2
```

if there are no vacant cells, double the size of the array

# Motivation

Problem. Given an array $A[]$ of size $n$, implement a **PUSH-BACK**(x) operation that inserts $x$ at the last vacant cell. If the number of occupied cells $i$ equals $n$, double the size of the array before inserting.

Example.



Assuming n and i are globally accessible.

**PUSH-BACK**(A[], x)

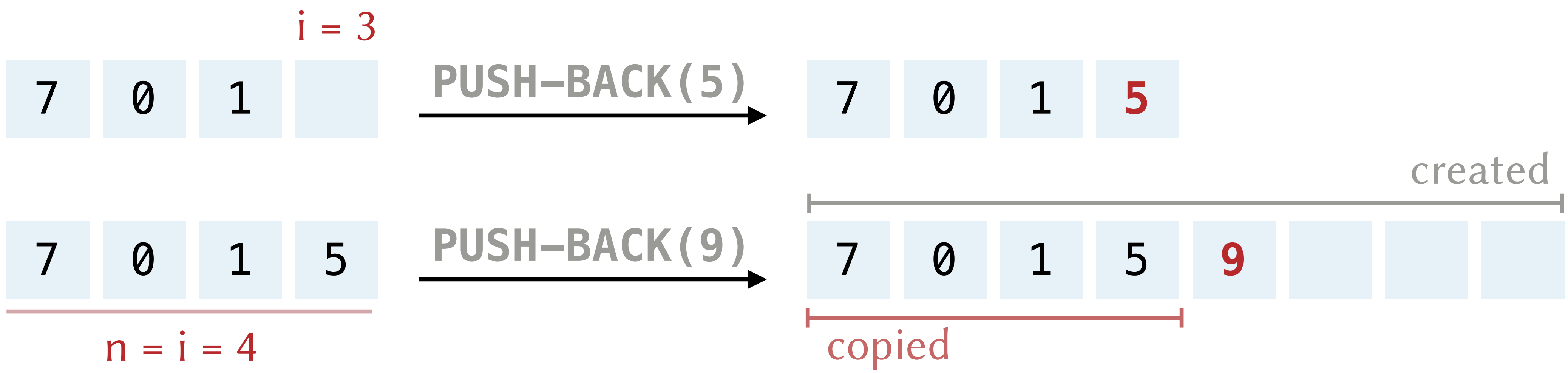
```
if (i < n): A[i] ← x
else:  Create array B of size n x 2
       B[0 … n-1] ← A[0 … n-1]
       A ← B
       n ← n x 2
       A[i] ← x
i ← i+1
```

insert in the newly created array

# Motivation

Problem. Given an array $A[]$ of size $n$, implement a **PUSH-BACK**(x) operation that inserts $x$ at the last vacant cell. If the number of occupied cells $i$ equals $n$, double the size of the array before inserting.

Example.



Assuming `n` and `i` are globally accessible.

```
PUSH-BACK(A[], x)

if (i < n): A[i] ← x
else:  Create array B of size n x 2
       B[0 … n-1] ← A[0 … n-1]
       A ← B
       n ← n x 2
       A[i] ← x
i ← i+1
```

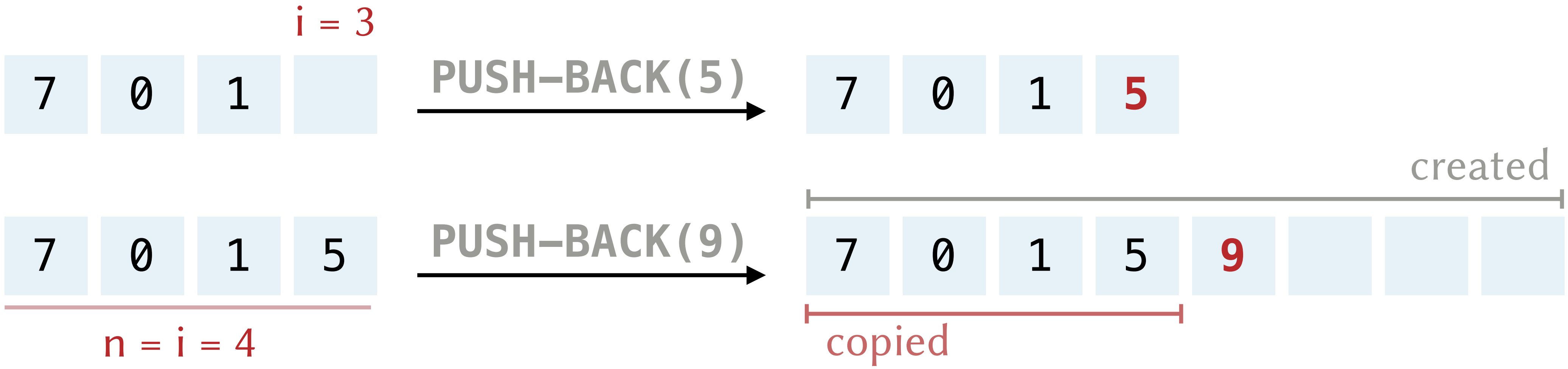
**Q** What is the worst case running time of function **PUSH-BACK**?

Cost Model. Number of copied elements

# Motivation

Problem. Given an array $A[]$ of size $n$, implement a **PUSH–BACK**(x) operation that inserts $x$ at the last vacant cell. If the number of occupied cells $i$ equals $n$, double the size of the array before inserting.

Example.



i = 3

7 0 1

PUSH–BACK(5)

7 0 1 **5**

created

7 0 1 5

n = i = 4

PUSH–BACK(9)

7 0 1 5 **9**

copied

Assuming n and i are globally accessible.

```
PUSH–BACK(A[], x)

if (i < n): A[i] ← x
else:  Create array B of size n x 2
       B[0 … n−1] ← A[0 … n−1]
       A ← B
       n ← n x 2
       A[i] ← x
i ← i+1
```

**Q** What is the worst case running time of function **PUSH–BACK**?

Cost Model. Number of copied elements

**A** $\Theta(n)$ because $n + 1$ elements are copied if the array is doubled

# Exercise

What is the worst case running time of calling **`PUSH-BACK`** $n$ times on a resizing array that is initially of size 1?

Cost Model. Count the number of *element copies*.

Choose the *best* answer.

**A.** $O(n^2)$

**B.** $O(n \log n)$

**C.** $O(n)$

**D.** $O(\log n)$

# Exercise

What is the worst case running time of calling **`PUSH-BACK`** $n$ times on a resizing array that is initially of size 1?

**Cost Model.** Count the number of *element copies*.

Choose the *best* answer.

**A.** $O(n^2)$

**B.** $O(n \log n)$

correct but too pessimistic!

😁 $O(n)$

correct and tight bound!

**D.** $O(\log n)$

incorrect!

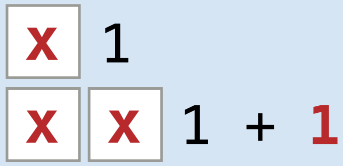
# Exercise

What is the worst case running time of calling **PUSH–BACK** $n$ times on a resizing array that is initially of size 1?

☒ 1

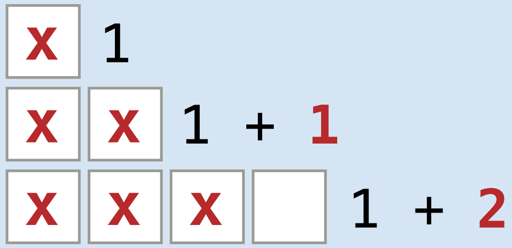
# Exercise

What is the worst case running time of calling **`PUSH–BACK`** $n$ times on a resizing array that is initially of size 1?



# Exercise

What is the worst case running time of calling **`PUSH–BACK`** $n$ times on a resizing array that is initially of size 1?



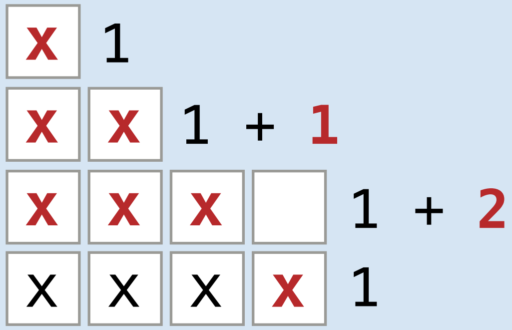
x 1

x x 1 + 1

x x x 1 + 2

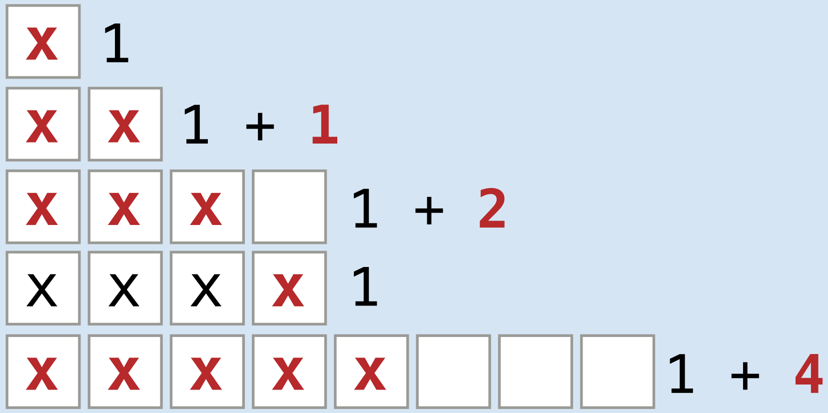
# Exercise

What is the worst case running time of calling **`PUSH–BACK`** *n* times on a resizing array that is initially of size 1?



# Exercise

What is the worst case running time of calling **PUSH–BACK** $n$ times on a resizing array that is initially of size 1?



x 1

x x 1 + 1

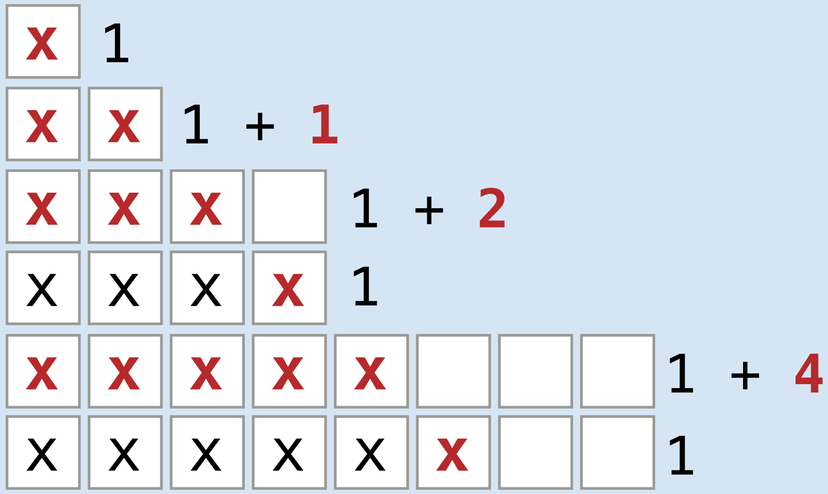
x x x 1 + 2

x x x x 1

x x x x x 1 + 4

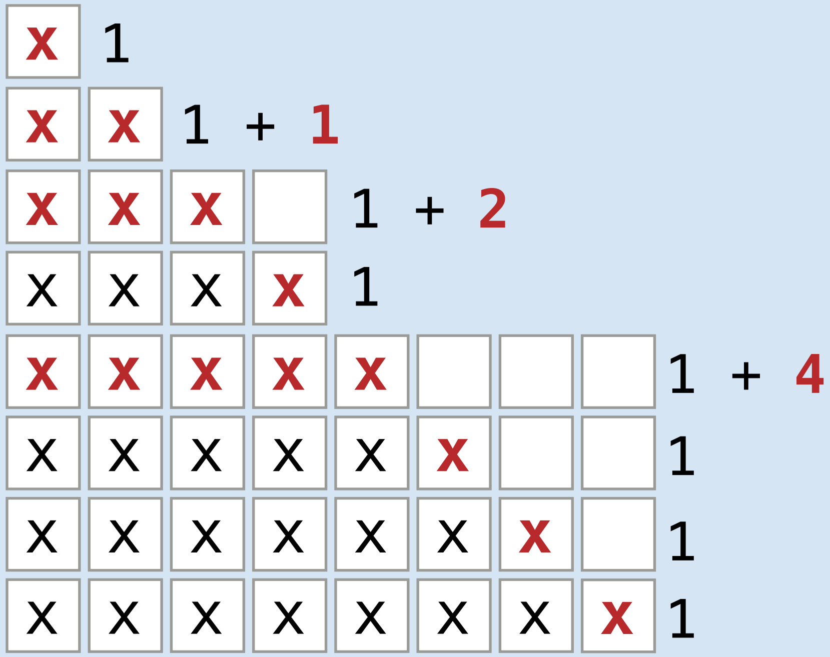
# Exercise

What is the worst case running time of calling **PUSH–BACK** $n$ times on a resizing array that is initially of size 1?



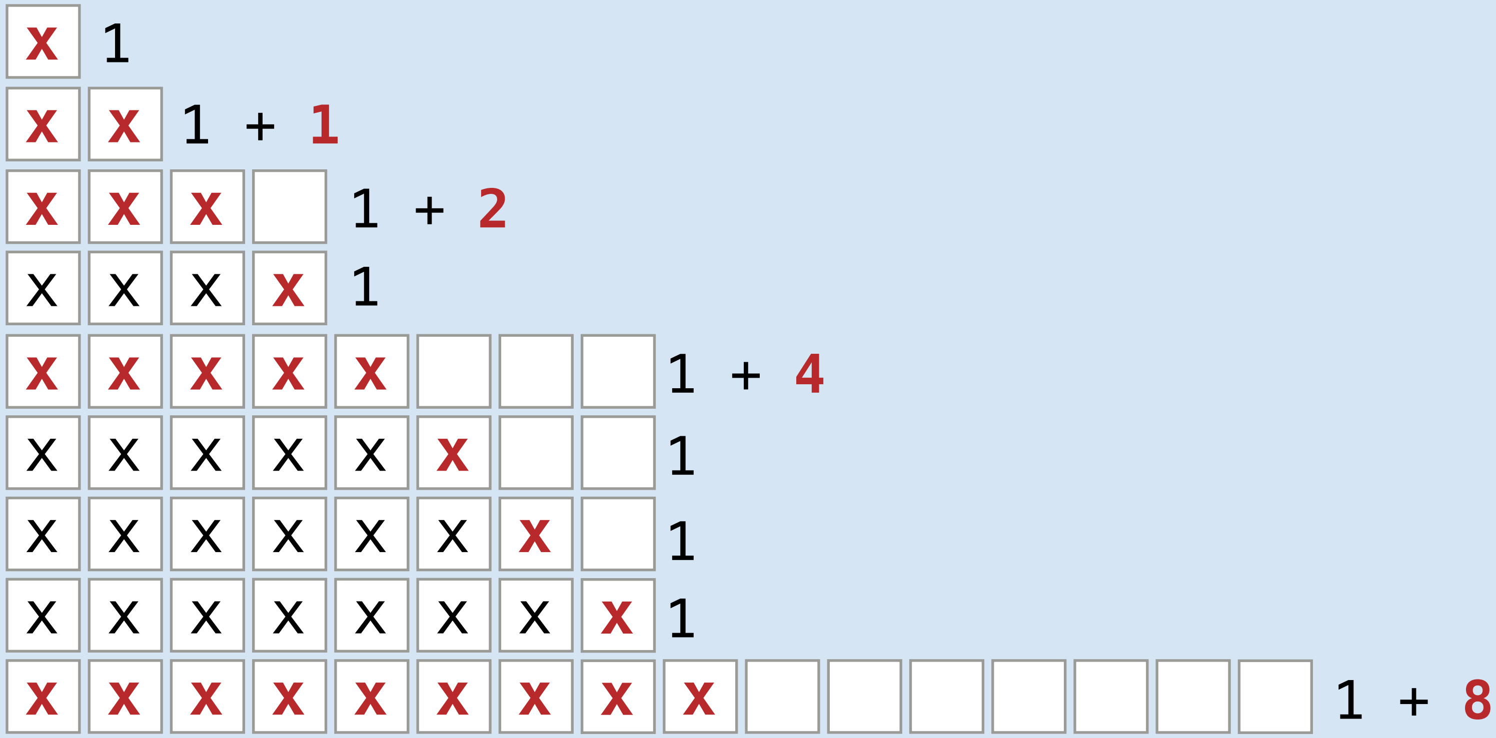
# Exercise

What is the worst case running time of calling **PUSH–BACK** $n$ times on a resizing array that is initially of size 1?



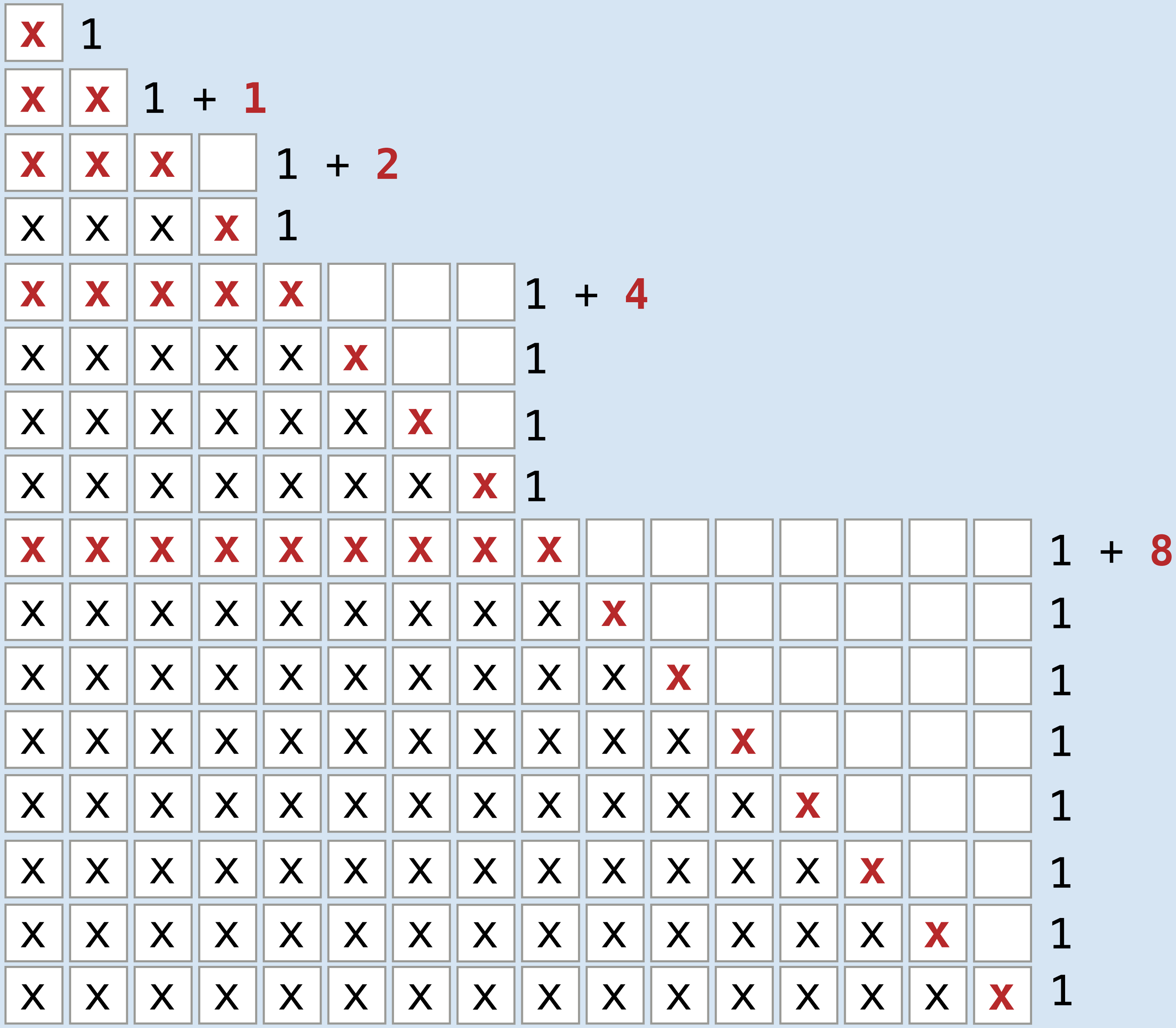
# Exercise

What is the worst case running time of calling **PUSH–BACK** $n$ times on a resizing array that is initially of size 1?



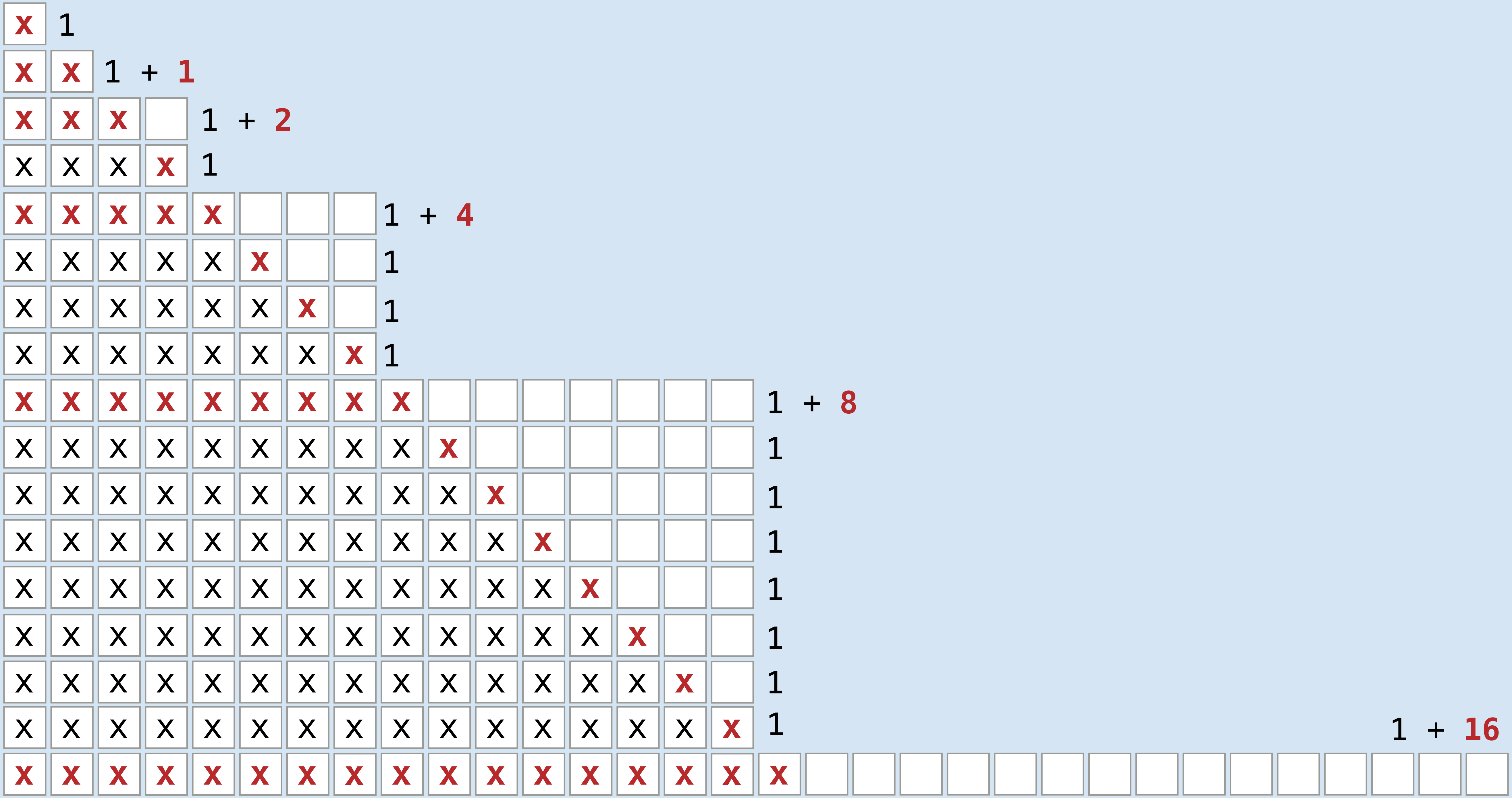
# Exercise

What is the worst case running time of calling **`PUSH–BACK`** $n$ times on a resizing array that is initially of size 1?



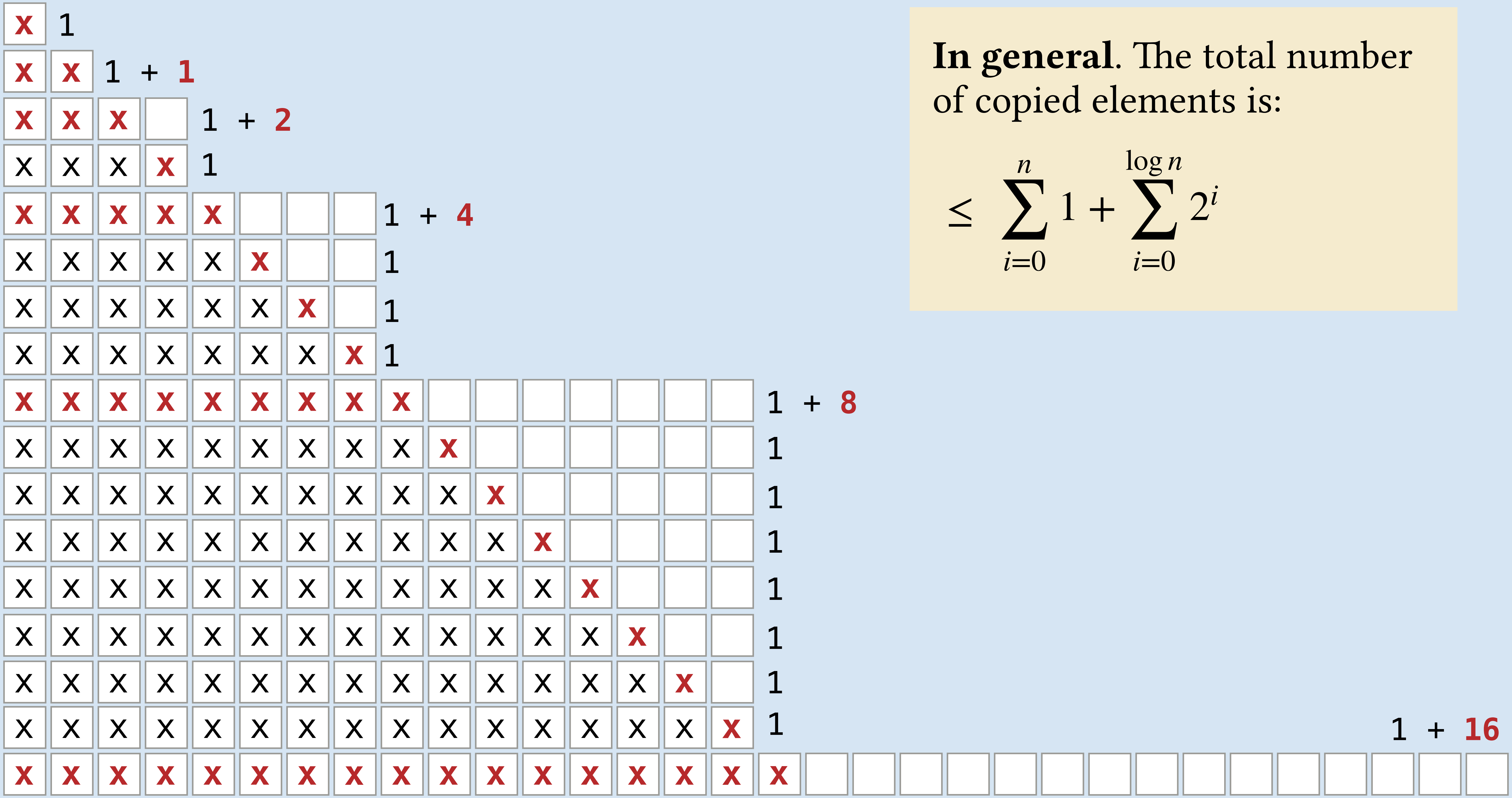
# Exercise

What is the worst case running time of calling **`PUSH-BACK`** $n$ times on a resizing array that is initially of size 1?



# Exercise

What is the worst case running time of calling **`PUSH–BACK`** $n$ times on a resizing array that is initially of size 1?

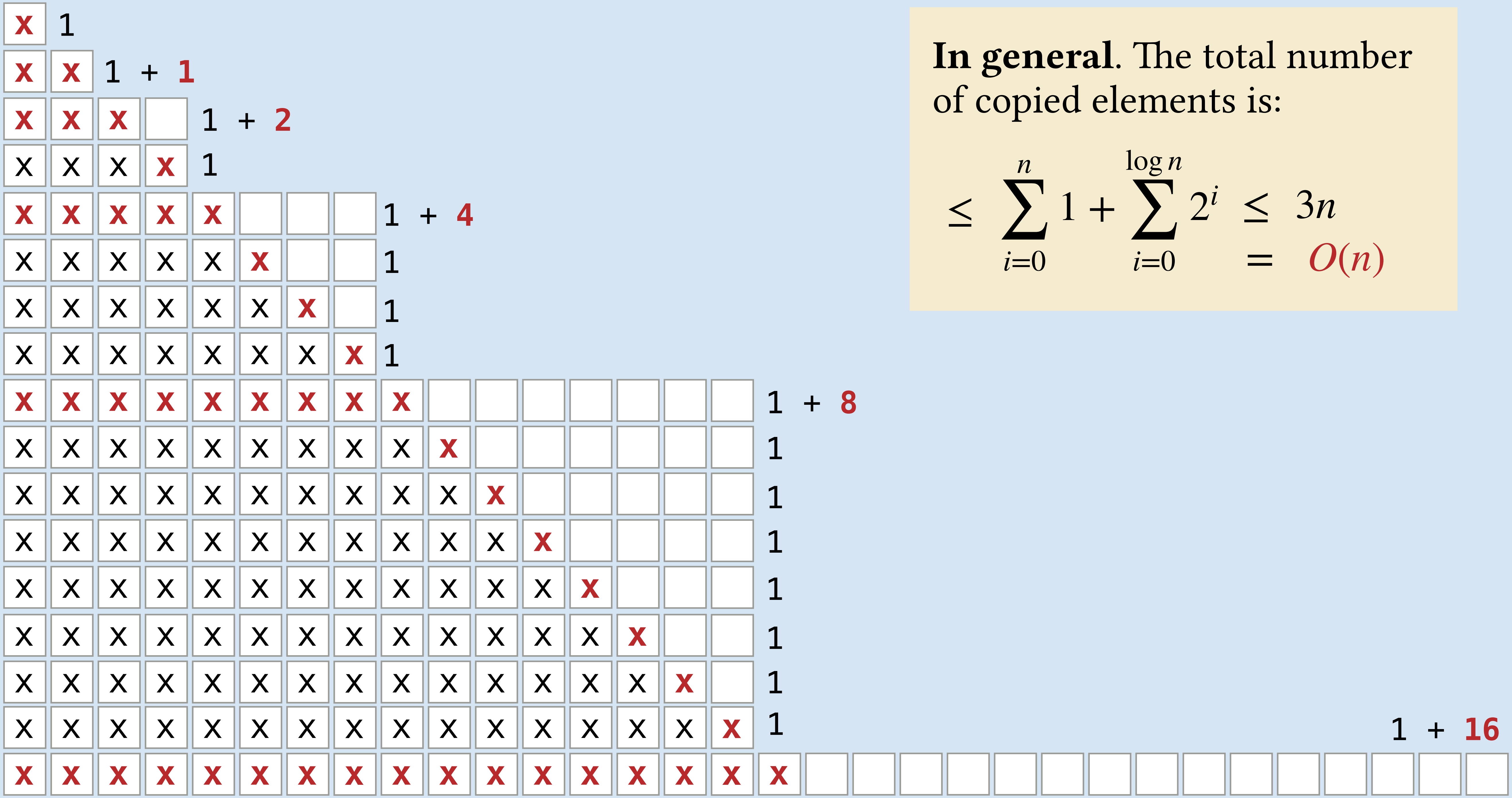


**In general**. The total number of copied elements is:

$$\leq \sum_{i=0}^{n} 1 + \sum_{i=0}^{\log n} 2^i$$

# Exercise

What is the worst case running time of calling **`PUSH−BACK`** $n$ times on a resizing array that is initially of size 1?

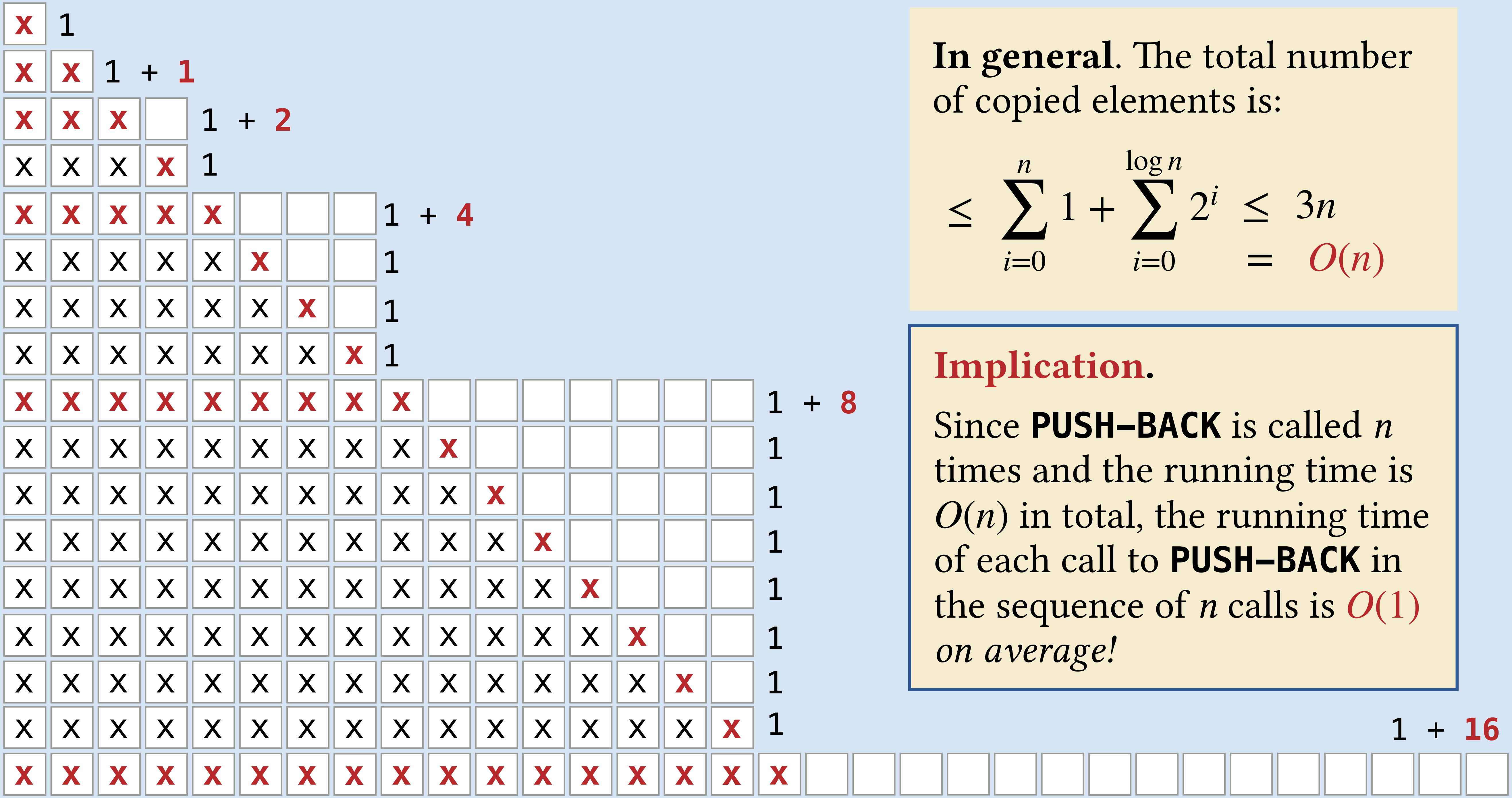


**In general**. The total number of copied elements is:

$$\le \sum_{i=0}^{n} 1 + \sum_{i=0}^{\log n} 2^i \le 3n = O(n)$$

# Exercise

What is the worst case running time of calling **PUSH–BACK** $n$ times on a resizing array that is initially of size 1?



**In general.** The total number of copied elements is:

$$\leq \sum_{i=0}^{n} 1 + \sum_{i=0}^{\log n} 2^i \leq 3n = O(n)$$

**Implication.**

Since **PUSH–BACK** is called $n$ times and the running time is $O(n)$ in total, the running time of each call to **PUSH–BACK** in the sequence of $n$ calls is $O(1)$ *on average!*

*welcome to*

# Amortized Analysis

# Amortized Analysis

Goal. Analyze the worst case running time of a sequence of operations.

Worst Case Analysis

**`PUSH-BACK`** runs in $\Theta(n)$ in the worst case.

Interpretation. At least one of the cases can make the function run in $\Theta(n)$.

**`INCREMENT`** runs in $\Theta(\log n)$ in the worst case.

Interpretation. At least one of the cases can make the function run in $\Theta(\log n)$.

**V.S.**

Amortized Analysis

The running time of **`PUSH-BACK`** is $O(1)$ amortized.

The running time of **`INCREMENT`** is $O(1)$ amortized.

Interpretation. If we perform a sequence of operations, the running time overall will be in the order of $n$ in the worst case and each single operation will have performed a constant amount of work on average.

# Amortized Analysis

Goal. Analyze the worst case running time of a sequence of operations.

Worst Case Analysis

**`PUSH-BACK`** runs in $\Theta(n)$ in the worst case.

Interpretation. At least one of the cases can make the function run in $\Theta(n)$.

**`INCREMENT`** runs in $\Theta(\log n)$ in the worst case.

Interpretation. At least one of the cases can make the function run in $\Theta(\log n)$.

`V.S.`

Amortized Analysis

The running time of **`PUSH-BACK`** is $O(1)$ amortized.

The running time of **`INCREMENT`** is $O(1)$ amortized.

Interpretation. If we perform a sequence of operations, the running time overall will be in the order of $n$ in the worst case and each single operation will have performed a constant amount of work on average.

! Amortized analysis can be done in multiple ways. The method we used so far is called the **aggregate method**.

# Amortized Analysis: Accountant's Method

Idea. Use cheap frequent operations to pay for rare but expensive operations.

# Amortized Analysis: Accountant's Method

Idea. Use cheap frequent operations to pay for rare but expensive operations.

- Assume that each unit of work costs `$1` and operation $i$ costs $c_i$.

# **Amortized Analysis:** Accountant's Method

Idea. Use cheap frequent operations to pay for rare but expensive operations.

- Assume that each unit of work costs `$1` and operation $i$ costs $c_i$ .
- Assign a new cost $\hat{c}_i$ for operation $i$ (can be <, > or = to $c_i$ ).

# Amortized Analysis: Accountant's Method

Idea. Use cheap frequent operations to pay for rare but expensive operations.

- Assume that each unit of work costs `$1` and operation $i$ costs $c_i$ .
- Assign a new cost $\hat{c}_i$ for operation $i$ (can be <, > or = to $c_i$ ).

- If $\hat{c}_i > c_i$ save the extra credit in the bank for use by other operations.
- If $\hat{c}_i < c_i$ consume from the credit stored in the bank.

# Amortized Analysis: Accountant's Method

Idea. Use cheap frequent operations to pay for rare but expensive operations.

- Assume that each unit of work costs `$1` and operation $i$ costs $c_i$ .
- Assign a new cost $\hat{c}_i$ for operation $i$ (can be <, > or = to $c_i$ ).

- If $\hat{c}_i > c_i$ save the extra credit in the bank for use by other operations.
- If $\hat{c}_i < c_i$ consume from the credit stored in the bank.

Goal. Show that credit always remains nonnegative, implying that $\sum \hat{c}_i \geq \sum c_i$

# Amortized Analysis: Accountant's Method

Idea. Use cheap frequent operations to pay for rare but expensive operations.

- Assume that each unit of work costs `$1` and operation $i$ costs $c_i$ .
- Assign a new cost $\hat{c}_i$ for operation $i$ (can be <, > or = to $c_i$ ).
- If $\hat{c}_i > c_i$ save the extra credit in the bank for use by other operations.
- If $\hat{c}_i < c_i$ consume from the credit stored in the bank.

Goal. Show that credit always remains nonnegative, implying that $\sum \hat{c}_i \geq \sum c_i$

---

Example. Array resizing.

**Actual Costs.**

- Copying a single element:
- Resizing the array:

**New Costs.**

- Copying a single element:
- Resizing the array:

# Amortized Analysis: Accountant's Method

Idea. Use cheap frequent operations to pay for rare but expensive operations.

- Assume that each unit of work costs `$1` and operation $i$ costs $c_i$ .
- Assign a new cost $\hat{c}_i$ for operation $i$ (can be <, > or = to $c_i$ ).
- If $\hat{c}_i > c_i$ save the extra credit in the bank for use by other operations.
- If $\hat{c}_i < c_i$ consume from the credit stored in the bank.

Goal. Show that credit always remains nonnegative, implying that $\sum \hat{c}_i \geq \sum c_i$

---

Example. Array resizing.

**Actual Costs.**

- Copying a single element: `$1`
- Resizing the array: `$n`
  ($n$ = number of elements added so far)

**New Costs.**

- Copying a single element:
- Resizing the array:

# Amortized Analysis: Accountant's Method

Idea. Use cheap frequent operations to pay for rare but expensive operations.

- Assume that each unit of work costs `$1` and operation $i$ costs $c_i$ .
- Assign a new cost $\hat{c}_i$ for operation $i$ (can be <, > or = to $c_i$ ).

- If $\hat{c}_i > c_i$ save the extra credit in the bank for use by other operations.
- If $\hat{c}_i < c_i$ consume from the credit stored in the bank.

Goal. Show that credit always remains nonnegative, implying that $\sum \hat{c}_i \geq \sum c_i$

---

Example. Array resizing.

**Actual Costs.**

- Copying a single element: **`$1`**
- Resizing the array: **`$n`**
  ($n$ = number of elements added so far)

**New Costs.**

- Copying a single element: **`$2`**
- Resizing the array: **`$0`**

# **Amortized Analysis:** Accountant's Method

Idea. Use cheap frequent operations to pay for rare but expensive operations.

- Assume that each unit of work costs `$1` and operation $i$ costs $c_i$ .
- Assign a new cost $\hat{c}_i$ for operation $i$ (can be <, > or = to $c_i$ ).
- If $\hat{c}_i > c_i$ save the extra credit in the bank for use by other operations.
- If $\hat{c}_i < c_i$ consume from the credit stored in the bank.

Goal. Show that credit always remains nonnegative, implying that $\sum \hat{c}_i \geq \sum c_i$

---

Example. Array resizing.

**Actual Costs.**

- Copying a single element: **`$1`**
- Resizing the array: **`$n`**
  ($n$ = number of elements added so far)

**New Costs.**

- Copying a single element: **`$2`**
- Resizing the array: **`$0`**

! We need to show that the bank credit will always remain nonnegative. I.e. The total new cost is not less than (equal or worse than) the total actual cost.

# **Accountants Method:** Resizing Array Demo

**Actual Costs.**

- Copying a single element: `$1`
- Resizing the array: `$n`

**New Costs.**

- Copying a single element: `$2`
- Resizing the array: `$0`

`PUSH-BACK` use `1$` and save `1$`

`$1`

# **Accountants Method:** Resizing Array Demo

**Actual Costs.**

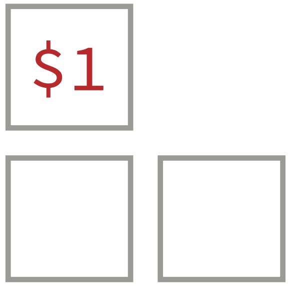
- Copying a single element: `$1`
- Resizing the array: `$n`

**New Costs.**

- Copying a single element: `$2`
- Resizing the array: `$0`

**`PUSH-BACK`** use `1$` and save `1$`

**`RESIZE`** use `1$` from the saved credit
to copy 1 element



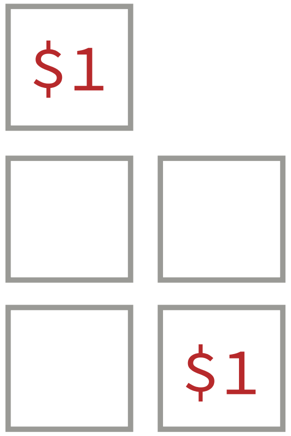
# **Accountants Method:** Resizing Array Demo

**Actual Costs.**

- Copying a single element: `$1`
- Resizing the array: `$n`

**New Costs.**

- Copying a single element: `$2`
- Resizing the array: `$0`



**PUSH-BACK** use `1$` and save `1$`

**RESIZE** use `1$` from the saved credit

**PUSH-BACK** use `1$` and save `1$`

# **Accountants Method:** Resizing Array Demo

**Actual Costs.**

- Copying a single element: `$1`
- Resizing the array: `$n`

**New Costs.**

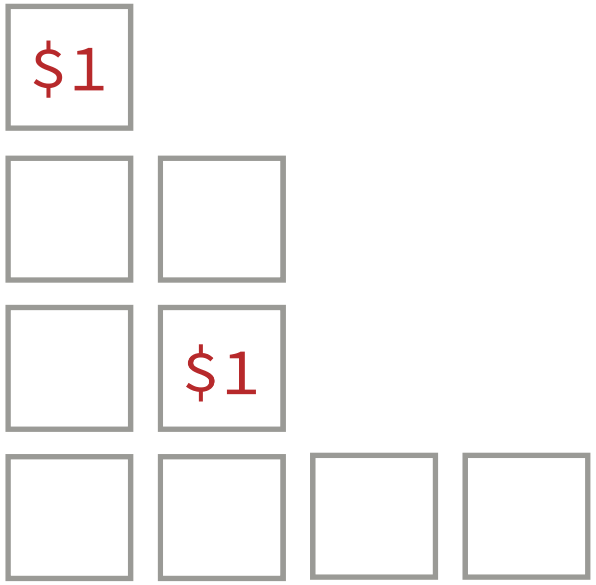
- Copying a single element: `$2`
- Resizing the array: `$0`

**PUSH–BACK** use 1$ and save 1$

**RESIZE** use 1$ from the saved credit

**PUSH–BACK** use 1$ and save 1$

**RESIZE** use 2$ from the saved credit to copy 2 elements



**OOPS!**

There is only `$1` in the credit

The chosen new costs are bad!

# **Accountants Method:** Resizing Array Demo

**Actual Costs.**

- Copying a single element: `$1`
- Resizing the array: `$n`

**New Costs.**

- Copying a single element: `$3`
- Resizing the array: `$0`

**SECOND ATTEMPT.** Use `$3` instead of `$2`

# Accountants Method: Resizing Array Demo

**Actual Costs.**

- Copying a single element: `$1`
- Resizing the array: `$n`

**New Costs.**

- Copying a single element: `$3`
- Resizing the array: `$0`

**SECOND ATTEMPT.** Use `$3` instead of `$2`

**PUSH–BACK** use `1$` and save `2$`

| `$2` |
|---|

# **Accountants Method:** Resizing Array Demo

**Actual Costs.**

- Copying a single element: $1
- Resizing the array: $n

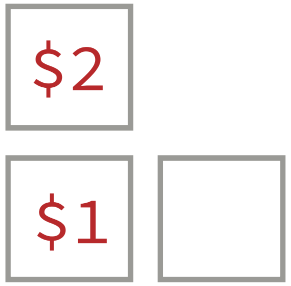
**New Costs.**

- Copying a single element: $3
- Resizing the array: $0

**SECOND ATTEMPT.** Use $3 instead of $2

**PUSH–BACK** use 1$ and save 2$

**RESIZE** use 1$ from the saved credit
to copy one element



# Accountants Method: Resizing Array Demo

**Actual Costs.**

- Copying a single element: $1
- Resizing the array: $n

**New Costs.**

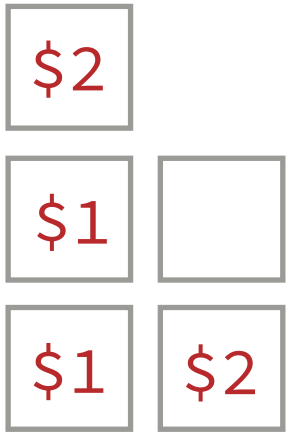
- Copying a single element: $3
- Resizing the array: $0

**SECOND ATTEMPT.** Use $3 instead of $2

**PUSH–BACK** use 1$ and save 2$

**RESIZE** use 1$ from the saved credit

**PUSH–BACK** use 1$ and save 2$



# Accountants Method: Resizing Array Demo

**Actual Costs.**

- Copying a single element: `$1`
- Resizing the array: `$n`

**New Costs.**

- Copying a single element: `$3`
- Resizing the array: `$0`

**`SECOND ATTEMPT.`** Use `$3` instead of `$2`

**`PUSH-BACK`** use `1$` and save `2$`

**`RESIZE`** use `1$` from the saved credit

**`PUSH-BACK`** use `1$` and save `2$`

**`RESIZE`** use `2$` from the saved credit to copy 2 elements

# **Accountants Method:** Resizing Array Demo

**Actual Costs.**

- Copying a single element: $1
- Resizing the array: $n

**New Costs.**

- Copying a single element: $3
- Resizing the array: $0

**SECOND ATTEMPT.** Use $3 instead of $2

| | | | | |
|---|---|---|---|---|
| **PUSH-BACK** use 1$ and save 2$ | $2 | | | |
| **RESIZE** use 1$ from the saved credit | $1 | | | |
| **PUSH-BACK** use 1$ and save 2$ | $1 | $2 | | |
| **RESIZE** use 2$ from the saved credit | $1 | | | |
| **PUSH-BACK** use 1$ and save 2$ | $1 | | $2 | |

# Accountants Method: Resizing Array Demo

**Actual Costs.**

- Copying a single element: `$1`
- Resizing the array: `$n`

**New Costs.**

- Copying a single element: `$3`
- Resizing the array: `$0`

**SECOND ATTEMPT.** Use `$3` instead of `$2`

| | | | | |
|---|---|---|---|---|
| **PUSH–BACK** use 1$ and save 2$ | $2 | | | |
| **RESIZE** use 1$ from the saved credit | $1 | | | |
| **PUSH–BACK** use 1$ and save 2$ | $1 | $2 | | |
| **RESIZE** use 2$ from the saved credit | $1 | | | |
| **PUSH–BACK** 2 times: use 1$ and save 2$ | $1 | | $2 | |

# Accountants Method: Resizing Array Demo

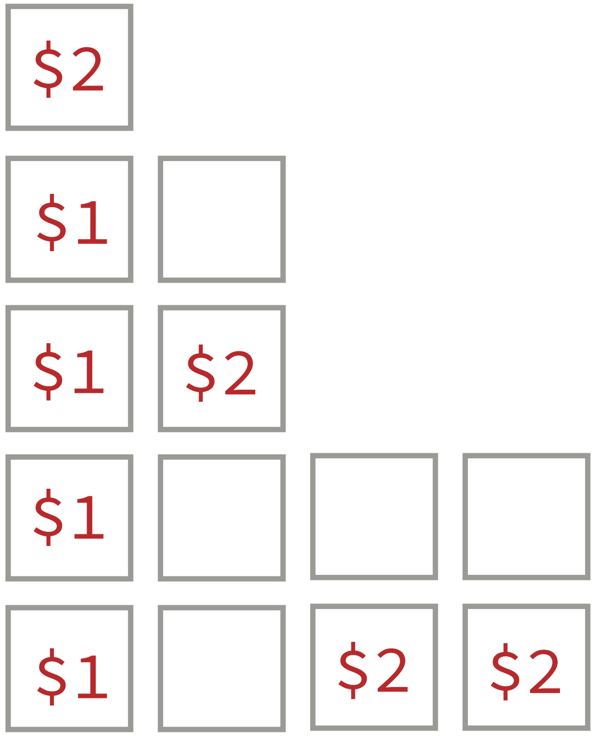
**Actual Costs.**

- Copying a single element: `$1`
- Resizing the array: `$n`

**New Costs.**

- Copying a single element: `$3`
- Resizing the array: `$0`

**SECOND ATTEMPT.** Use `$3` instead of `$2`



**PUSH–BACK** use 1$ and save 2$

**RESIZE** use 1$ from the saved credit

**PUSH–BACK** use 1$ and save 2$

**RESIZE** use 2$ from the saved credit

**PUSH–BACK** 2 times: use 1$ and save 2$

# **Accountants Method:** Resizing Array Demo

**Actual Costs.**

- Copying a single element: $1
- Resizing the array: $n

**New Costs.**

- Copying a single element: $3
- Resizing the array: $0

**SECOND ATTEMPT.** Use $3 instead of $2

**PUSH–BACK** use 1$ and save 2$

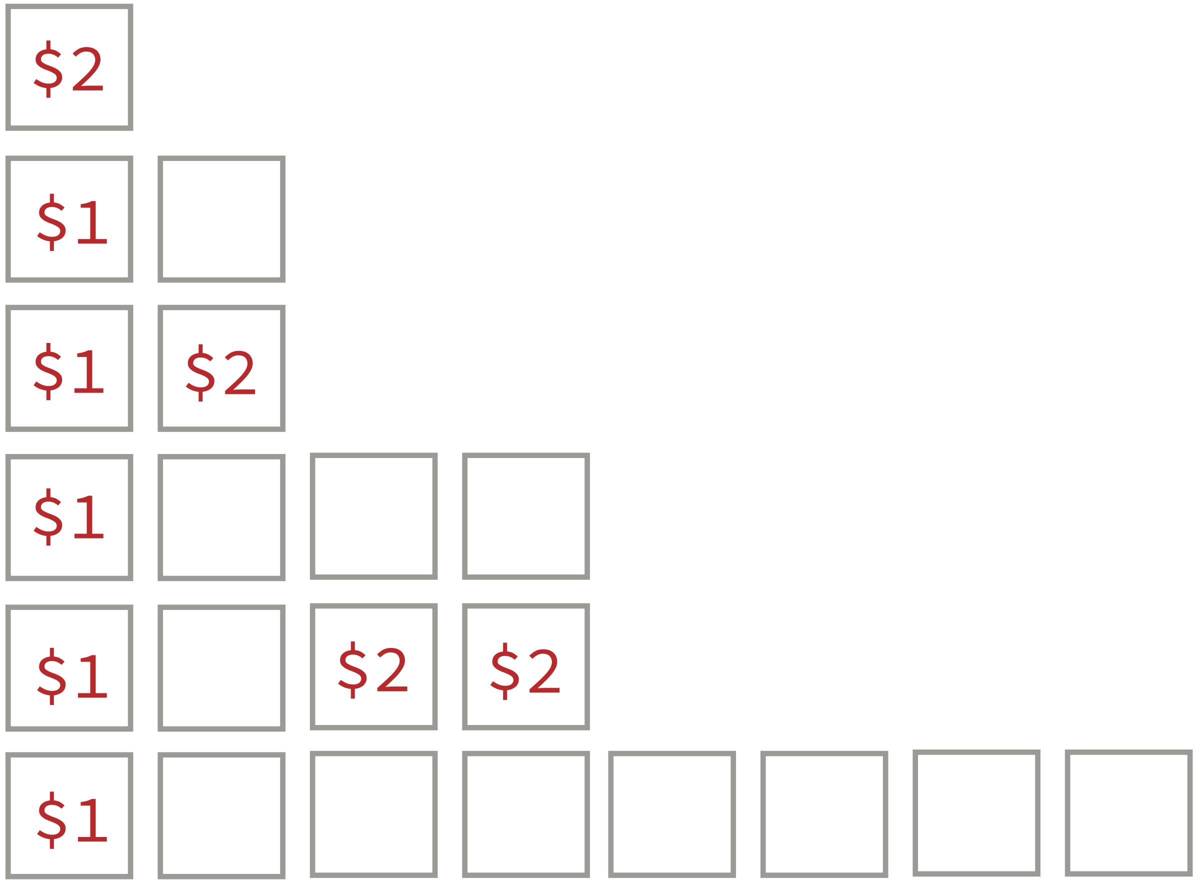
**RESIZE** use 1$ from the saved credit

**PUSH–BACK** use 1$ and save 2$

**RESIZE** use 2$ from the saved credit

**PUSH–BACK** 2 times: use 1$ and save 2$

**RESIZE** use 4$ from the saved credit to copy 4 elements



# **Accountants Method:** Resizing Array Demo

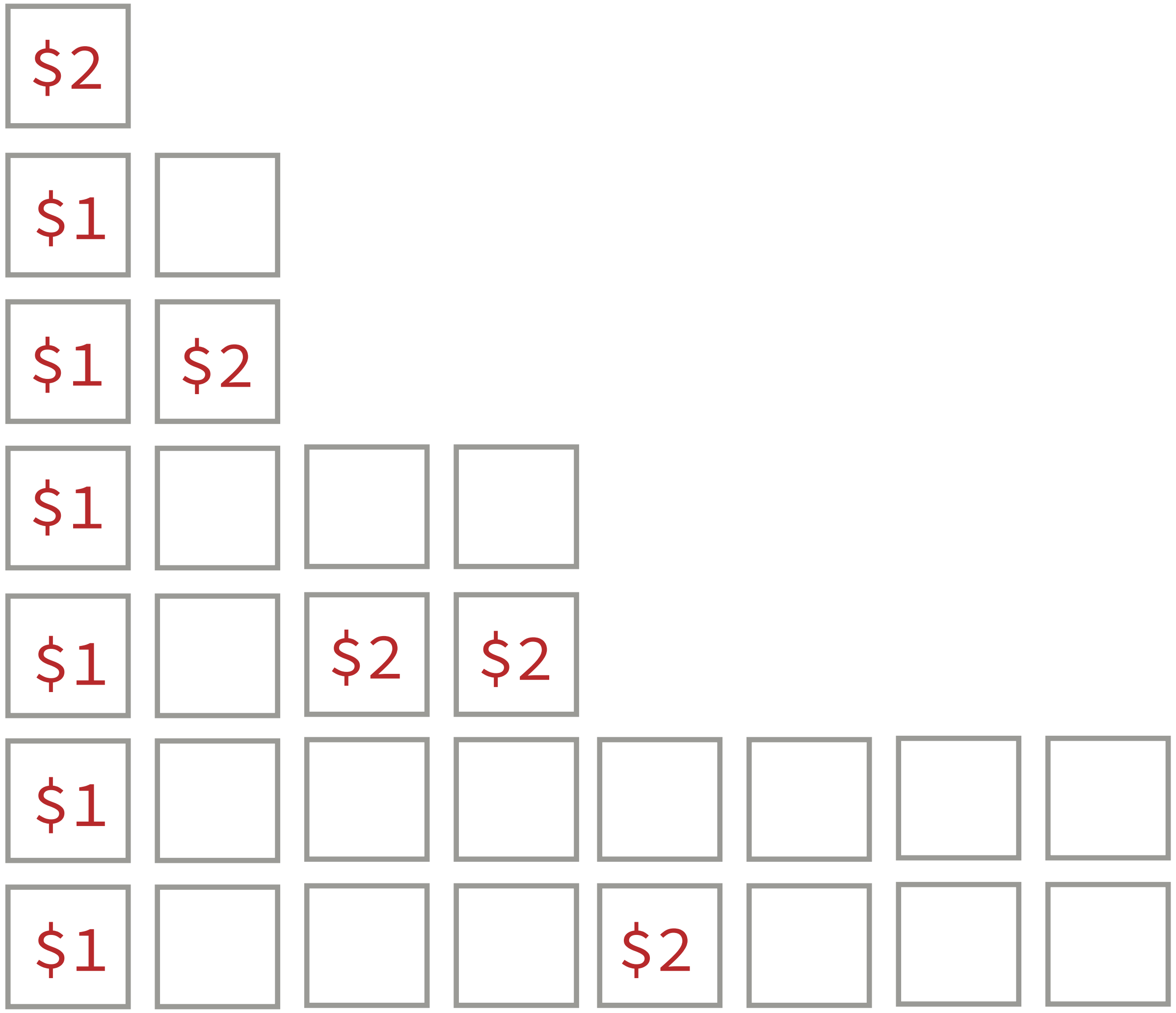
**Actual Costs.**

- Copying a single element: `$1`
- Resizing the array: `$n`

**New Costs.**

- Copying a single element: `$3`
- Resizing the array: `$0`

**`SECOND ATTEMPT.`** Use `$3` instead of `$2`



| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **`PUSH-BACK`** use `1$` and save `2$` | `$2` | | | | | | | |
| **`RESIZE`** use `1$` from the saved credit | `$1` | | | | | | | |
| **`PUSH-BACK`** use `1$` and save `2$` | `$1` | `$2` | | | | | | |
| **`RESIZE`** use `2$` from the saved credit | `$1` | | | | | | | |
| **`PUSH-BACK`** 2 times: use `1$` and save `2$` | `$1` | | `$2` | `$2` | | | | |
| **`RESIZE`** use `4$` from the saved credit | `$1` | | | | | | | |
| **`PUSH-BACK`** use `1$` and save `2$` | `$1` | | | | `$2` | | | |

# **Accountants Method:** Resizing Array Demo

**Actual Costs.**

- Copying a single element: `$1`
- Resizing the array: `$n`

**New Costs.**

- Copying a single element: `$3`
- Resizing the array: `$0`

**`SECOND ATTEMPT.`** Use `$3` instead of `$2`

**`PUSH-BACK`** use `1$` and save `2$`

**`RESIZE`** use `1$` from the saved credit

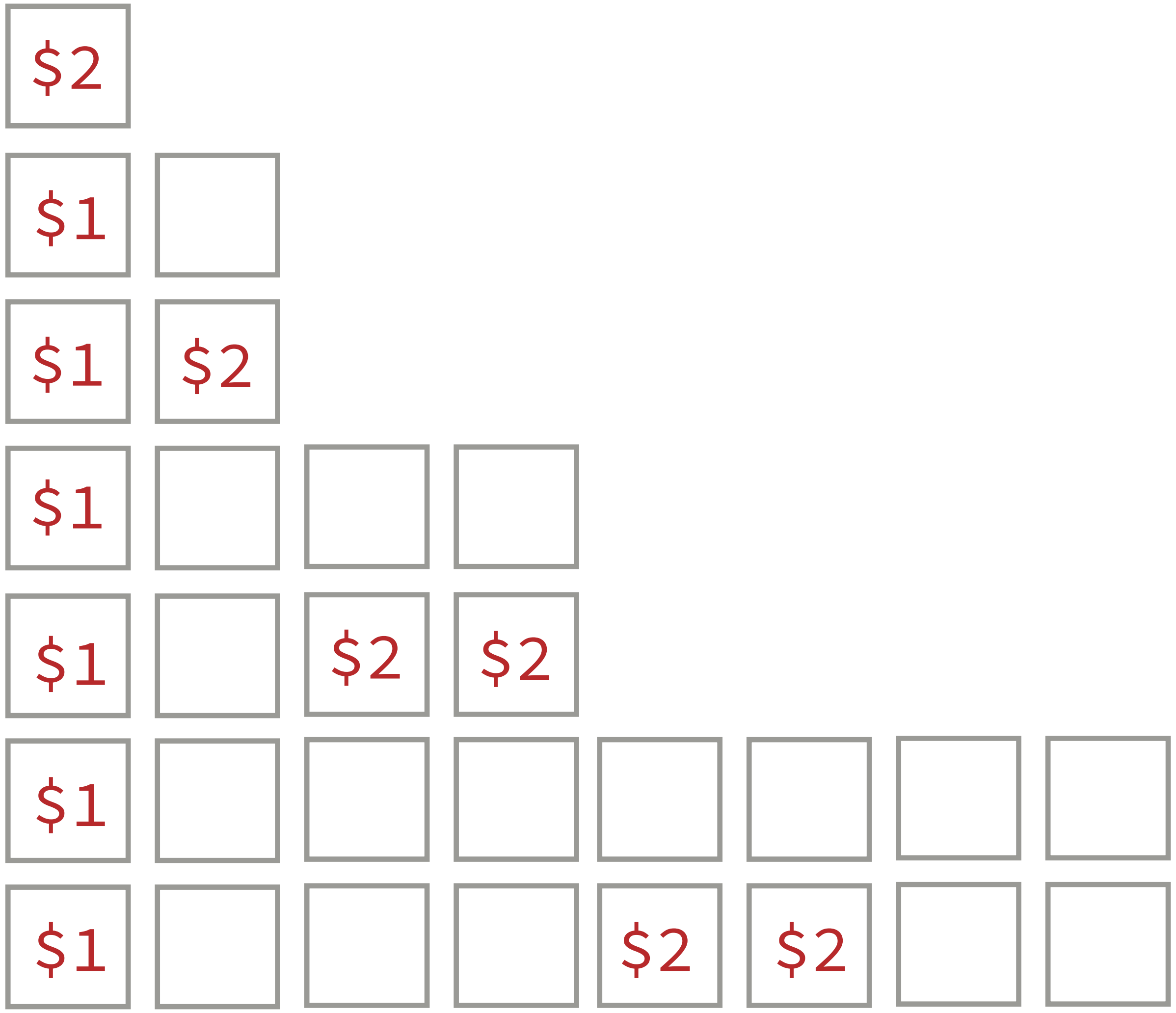
**`PUSH-BACK`** use `1$` and save `2$`

**`RESIZE`** use `2$` from the saved credit

**`PUSH-BACK`** 2 times: use `1$` and save `2$`

**`RESIZE`** use `4$` from the saved credit

**`PUSH-BACK`** 2 times: use `1$` and save `2$`



# **Accountants Method:** Resizing Array Demo

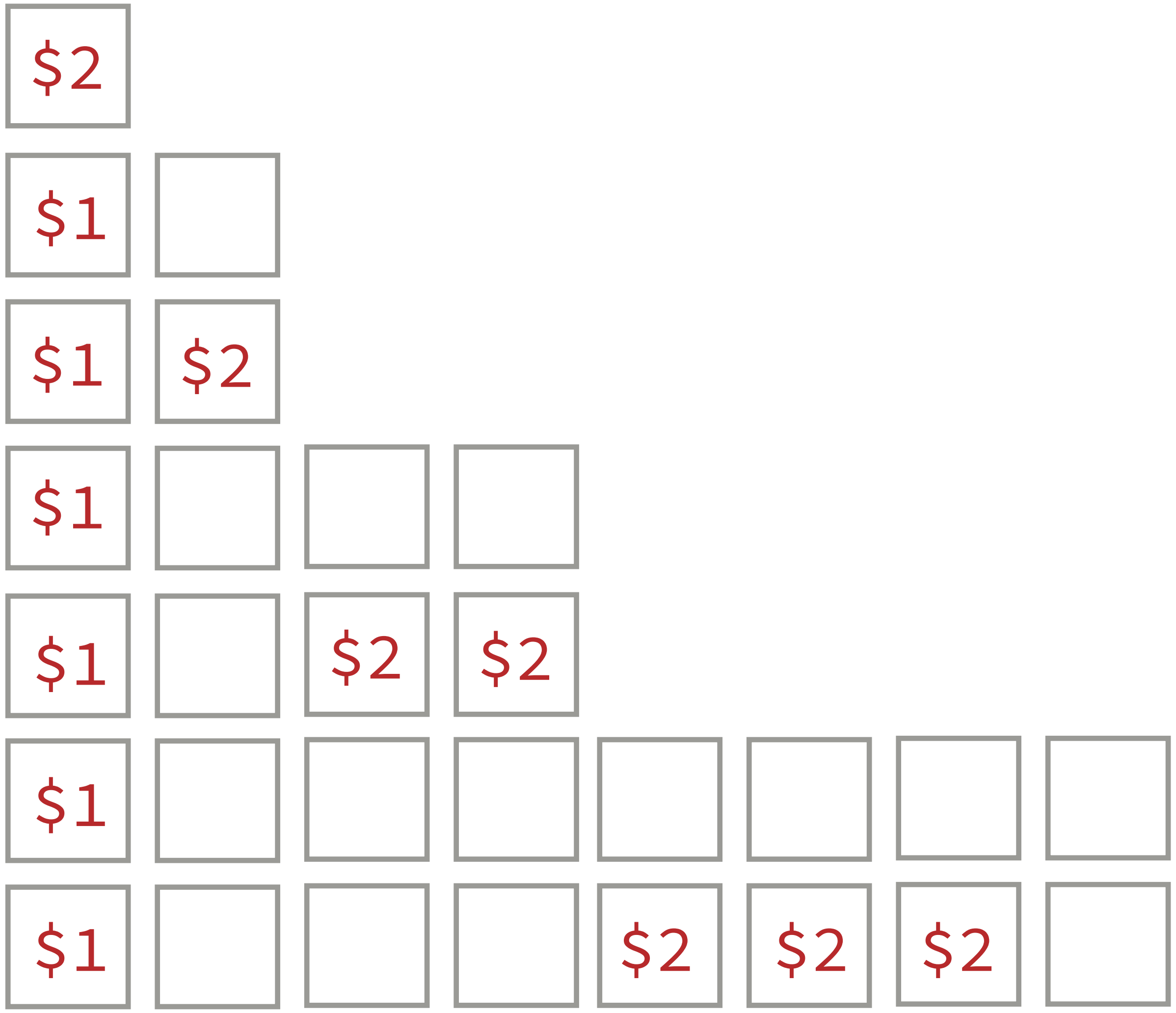
**Actual Costs.**

- Copying a single element: $1
- Resizing the array: $n

**New Costs.**

- Copying a single element: $3
- Resizing the array: $0

**SECOND ATTEMPT.** Use $3 instead of $2



| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **PUSH–BACK** use 1$ and save 2$ | $2 | | | | | | | |
| **RESIZE** use 1$ from the saved credit | $1 | | | | | | | |
| **PUSH–BACK** use 1$ and save 2$ | $1 | $2 | | | | | | |
| **RESIZE** use 2$ from the saved credit | $1 | | | | | | | |
| **PUSH–BACK** 2 times: use 1$ and save 2$ | $1 | | $2 | $2 | | | | |
| **RESIZE** use 4$ from the saved credit | $1 | | | | | | | |
| **PUSH–BACK** 3 times: use 1$ and save 2$ | $1 | | | | $2 | $2 | $2 | |

# **Accountants Method:** Resizing Array Demo

**Actual Costs.**

- Copying a single element: $1
- Resizing the array: $n

**New Costs.**

- Copying a single element: $3
- Resizing the array: $0

**SECOND ATTEMPT.** Use $3 instead of $2

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **PUSH-BACK** use 1$ and save 2$ | $2 | | | | | | | |
| **RESIZE** use 1$ from the saved credit | $1 | | | | | | | |
| **PUSH-BACK** use 1$ and save 2$ | $1 | $2 | | | | | | |
| **RESIZE** use 2$ from the saved credit | $1 | | | | | | | |
| **PUSH-BACK** 2 times: use 1$ and save 2$ | $1 | | $2 | $2 | | | | |
| **RESIZE** use 4$ from the saved credit | $1 | | | | | | | |
| **PUSH-BACK** 4 times: use 1$ and save 2$ | $1 | | | | $2 | $2 | $2 | $2 |

# Accountants Method: Resizing Array Demo

**Actual Costs.**

- Copying a single element: $1
- Resizing the array: $n

**New Costs.**

- Copying a single element: $3
- Resizing the array: $0

**SECOND ATTEMPT.** Use $3 instead of $2

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **PUSH-BACK** use 1$ and save 2$ | $2 | | | | | | | |
| **RESIZE** use 1$ from the saved credit | $1 | | | | | | | |
| **PUSH-BACK** use 1$ and save 2$ | $1 | $2 | | | | | | |
| **RESIZE** use 2$ from the saved credit | $1 | | | | | | | |
| **PUSH-BACK** 2 times: use 1$ and save 2$ | $1 | | $2 | $2 | | | | |
| **RESIZE** use 4$ from the saved credit | $1 | | | | | | | |
| **PUSH-BACK** 4 times: use 1$ and save 2$ | $1 | | | | $2 | $2 | $2 | $2 |

😁 There is enough to pay for a new resize!

# **Accountants Method:** Resizing Array Demo

**Actual Costs.**

- Copying a single element: `$1`
- Resizing the array: `$n`

**New Costs.**

- Copying a single element: `$3`
- Resizing the array: `$0`

Claim. Credit will always remain nonnegative.

Proof. From the examples in the previous slide, this is true for 9 **`PUSH-BACK`** operations. Assume that the claim is true for $k$ **`PUSH-BACK`** operations.

# Accountants Method: Resizing Array Demo

**Actual Costs.**

- Copying a single element: `$1`
- Resizing the array: `$n`

**New Costs.**

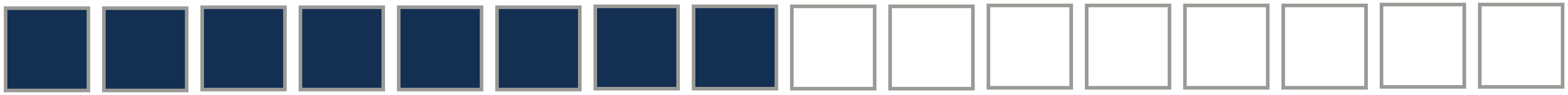
- Copying a single element: `$3`
- Resizing the array: `$0`

Claim. Credit will always remain nonnegative.

Proof. From the examples in the previous slide, this is true for 9 **`PUSH−BACK`** operations. Assume that the claim is true for $k$ **`PUSH−BACK`** operations.

**Note that:**

- Copying a single element without resize adds **$2** to the credit.
- Resizing produces an array of size $n$, where $\frac{n}{2}$ cells are empty.



# Accountants Method: Resizing Array Demo

**Actual Costs.**

- Copying a single element: `$1`
- Resizing the array: `$n`

**New Costs.**

- Copying a single element: `$3`
- Resizing the array: `$0`

Claim. Credit will always remain nonnegative.

Proof. From the examples in the previous slide, this is true for 9 **PUSH−BACK** operations. Assume that the claim is true for $k$ **PUSH−BACK** operations.

**Note that:**

- Copying a single element without resize adds **$2** to the credit.
- Resizing produces an array of size $n$, where $\frac{n}{2}$ cells are empty.
- Therefore, the next resize happens after $\frac{n}{2}$ copy operations, which adds $\$2 \times \frac{n}{2} = \$n$ to the credit.

# **Accountants Method:** Resizing Array Demo

**Actual Costs.**

- Copying a single element: `$1`
- Resizing the array: `$n`

**New Costs.**

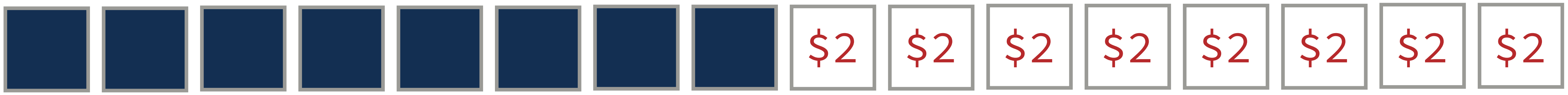
- Copying a single element: `$3`
- Resizing the array: `$0`

Claim. Credit will always remain nonnegative.

Proof. From the examples in the previous slide, this is true for 9 **`PUSH−BACK`** operations. Assume that the claim is true for $k$ **`PUSH−BACK`** operations.

**Note that:**

- Copying a single element without resize adds **`$2`** to the credit.
- Resizing produces an array of size $n$, where $\frac{n}{2}$ cells are empty.
- Therefore, the next resize happens after $\frac{n}{2}$ copy operations, which adds `$2` $\times \frac{n}{2} =$ `$n` to the credit.
- This is enough to copy the $n$ elements in the next resize operation.



# **Accountants Method:** Resizing Array Demo

**Actual Costs.**

- Copying a single element: `$1`
- Resizing the array: `$n`

**New Costs.**

- Copying a single element: `$3`
- Resizing the array: `$0`

Claim. Credit will always remain nonnegative.

Proof. From the examples in the previous slide, this is true for 9 **PUSH−BACK** operations. Assume that the claim is true for $k$ **PUSH−BACK** operations.

**Note that:**

- Copying a single element without resize adds **$2** to the credit.
- Resizing produces an array of size $n$, where $\frac{n}{2}$ cells are empty.
- Therefore, the next resize happens after $\frac{n}{2}$ copy operations, which adds \$2 $\times \frac{n}{2} =$ \$n to the credit.
- This is enough to copy the $n$ elements in the next resize operation.

Hence, the claim is true for the $k + 1$ **PUSH−BACK** operation.

# **Accountants Method:** Resizing Array Demo

**Actual Costs.**

- Copying a single element: `$1`
- Resizing the array: `$n`

**New Costs.**

- Copying a single element: `$3`
- Resizing the array: `$0`

**Amortized Cost.**

- Copying a single element: `O(1)`
- Resizing the array: `O(1)`

Claim. Credit will always remain nonnega

Proof. From the examples in the previous
operations. Assume that the claim is true

**Note that:**

- Copying a single element without resize adds **$2** to the credit.
- Resizing produces an array of size $n$, where $\frac{n}{2}$ cells are empty.
- Therefore, the next resize happens after $\frac{n}{2}$ copy operations, which adds $\$2 \times \frac{n}{2} = \$n$ to the credit.
- This is enough to copy the $n$ elements in the next resize operation.

Hence, the claim is true for the $k + 1$ **PUSH−BACK** operation.

# **Accountants Method:** Resizing Array Demo

**Actual Costs.**

- Copying a single element: `$1`
- Resizing the array: `$n`

**New Costs.**

- Copying a single element: `$3`
- Resizing the array: `$0`

**Amortized Cost.**

- Copying a single element: `O(1)`
- Resizing the array: `O(1)`

*Assuming arrays are not resized unless they are full and that their size doubles when they resize.*

Claim. Credit will always remain nonnega

Proof. From the examples in the previous operations. Assume that the claim is true

**Note that:**

- Copying a single element without resize adds **$2** to the credit.
- Resizing produces an array of size $n$, where $\frac{n}{2}$ cells are empty.
- Therefore, the next resize happens after $\frac{n}{2}$ copy operations, which adds $\$2 \times \frac{n}{2} = \$n$ to the credit.
- This is enough to copy the $n$ elements in the next resize operation.

Hence, the claim is true for the $k + 1$ **PUSH−BACK** operation.

# Accountants Method: Resizing Array Demo

**Actual Costs.**

- Copying a single element: `$1`
- Resizing the array: `$n`

**New Costs.**

- Copying a single element: `$3`
- Resizing the array: `$0`

**Amortized Cost.**

- Copying a single element: `O(1)`
- Resizing the array: `O(1)`

*Assuming arrays are not resized unless they are full and that their size doubles when they resize.*

**Interpretation.**

Any sequence of **PUSH−BACK** operations performs *at most* $\Theta(n)$ operations in total.

Claim. Credit will always remain nonnega

Proof. From the examples in the previous
operations. Assume that the claim is true

**Note that:**

- Copying a single element without
- Resizing produces an array of size
- Therefore, the next resize happens
  $\$2 \times \frac{n}{2} = \$n$ to the credit.
- This is enough to copy the $n$ elements in the next resize operation.

Hence, the claim is true for the $k+1$ **PUSH−BACK** operation.

# **Accountants Method:** Incrementing a Binary Counter

**Actual Costs.**

- Flipping `0`'s to `1`'s:
- Flipping `1`'s to `0`'s: up to

**New Costs.**

- Flipping `0`'s to `1`'s:
- Flipping `1`'s to `0`'s:

| | | | | | | |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | |
| 1 | 0 | 0 | 0 | 0 | 1 | |
| 2 | 0 | 0 | 0 | 1 | 0 | |
| 3 | 0 | 0 | 0 | 1 | 1 | |
| 4 | 0 | 0 | 1 | 0 | 0 | Flipping 0's is cheap! |
| 5 | 0 | 0 | 1 | 0 | 1 | |
| 6 | 0 | 0 | 1 | 1 | 0 | |
| 7 | 0 | 0 | 1 | 1 | 1 | Flipping 1's can be expensive! |
| 8 | 0 | 1 | 0 | 0 | 0 | |
| 9 | 0 | 1 | 0 | 0 | 1 | |
| 10 | 0 | 1 | 0 | 1 | 0 | |
| 11 | 0 | 1 | 0 | 1 | 1 | |
| 12 | 0 | 1 | 1 | 0 | 0 | |
| 13 | 0 | 1 | 1 | 0 | 1 | |
| 14 | 0 | 1 | 1 | 1 | 0 | |
| 15 | 0 | 1 | 1 | 1 | 1 | |

# **Accountants Method:** Incrementing a Binary Counter

**Actual Costs.**

- Flipping `0`'s to `1`'s: `$1`
- Flipping `1`'s to `0`'s: up to `$log(n)`

**New Costs.**

- Flipping `0`'s to `1`'s:
- Flipping `1`'s to `0`'s:

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

Flipping 0's is cheap!

Flipping 1's can be expensive!

# **Accountants Method:** Incrementing a Binary Counter

**Actual Costs.**

- Flipping `0`'s to `1`'s: `$1`
- Flipping `1`'s to `0`'s: up to `$log(n)`

**New Costs.**

- Flipping `0`'s to `1`'s: `$2`
- Flipping `1`'s to `0`'s: `$0`

**Let's Try!**

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

Flipping 0's is cheap!

Flipping 1's can be expensive!

# **Accountants Method:** Incrementing a Binary Counter

**Actual Costs.**

- Flipping `0`'s to `1`'s: `$1`
- Flipping `1`'s to `0`'s: up to `$log(n)`

**New Costs.**

- Flipping `0`'s to `1`'s: `$2`
- Flipping `1`'s to `0`'s: `$0`

`00000` → `00001`

credit=

add `$2` and use `$1`

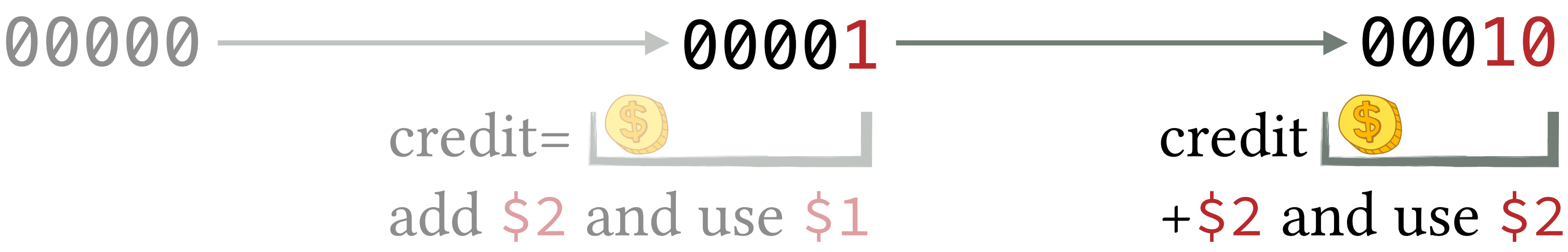
# **Accountants Method:** Incrementing a Binary Counter

**Actual Costs.**

- Flipping `0`'s to `1`'s: `$1`
- Flipping `1`'s to `0`'s: up to `$log(n)`

**New Costs.**

- Flipping `0`'s to `1`'s: `$2`
- Flipping `1`'s to `0`'s: `$0`



`00000` → `00001` → `00010`

credit=

add `$2` and use `$1`

credit

+`$2` and use `$2`

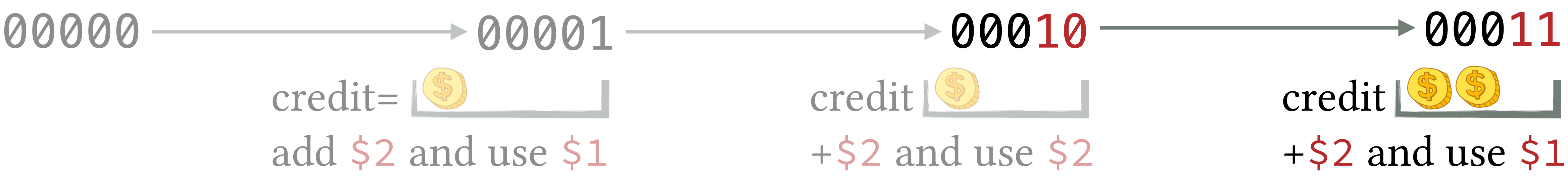
# **Accountants Method:** Incrementing a Binary Counter

**Actual Costs.**

- Flipping `0`'s to `1`'s: `$1`
- Flipping `1`'s to `0`'s: up to `$log(n)`

**New Costs.**

- Flipping `0`'s to `1`'s: `$2`
- Flipping `1`'s to `0`'s: `$0`



`00000` → `00001` → `00010` → `00011`

credit=
add `$2` and use `$1`

credit
+`$2` and use `$2`

credit
+`$2` and use `$1`

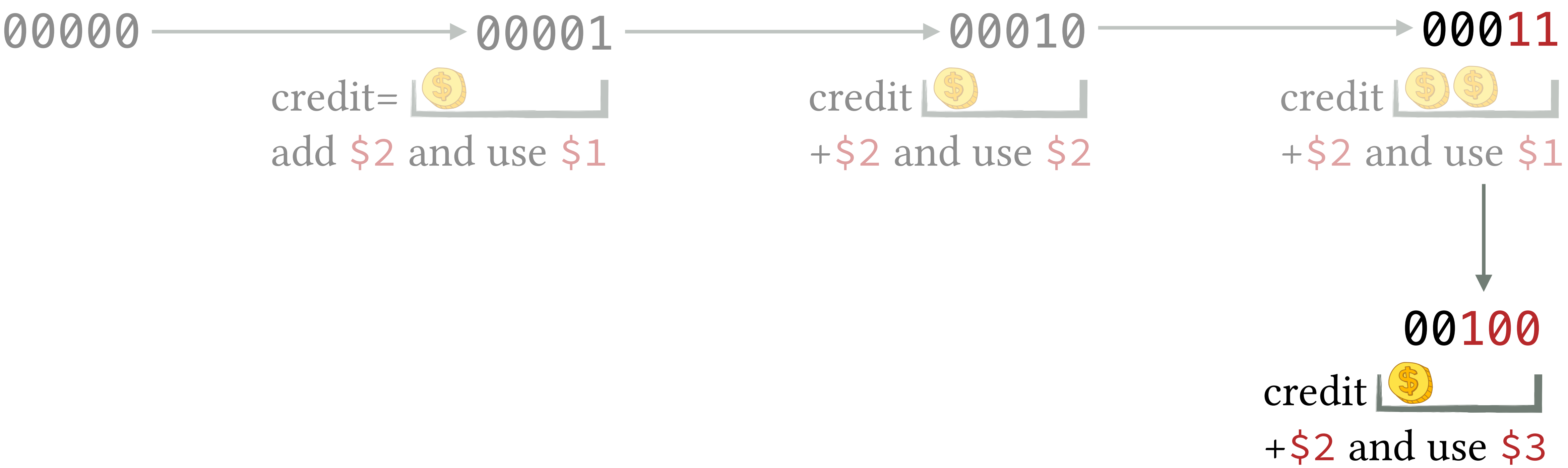
# **Accountants Method:** Incrementing a Binary Counter

**Actual Costs.**

- Flipping `0`'s to `1`'s: `$1`
- Flipping `1`'s to `0`'s: up to `$log(n)`

**New Costs.**

- Flipping `0`'s to `1`'s: `$2`
- Flipping `1`'s to `0`'s: `$0`



`00000` → `00001` → `00010` → `00011`

credit=

add `$2` and use `$1`

credit

+`$2` and use `$2`

credit

+`$2` and use `$1`

`00100`

credit

+`$2` and use `$3`

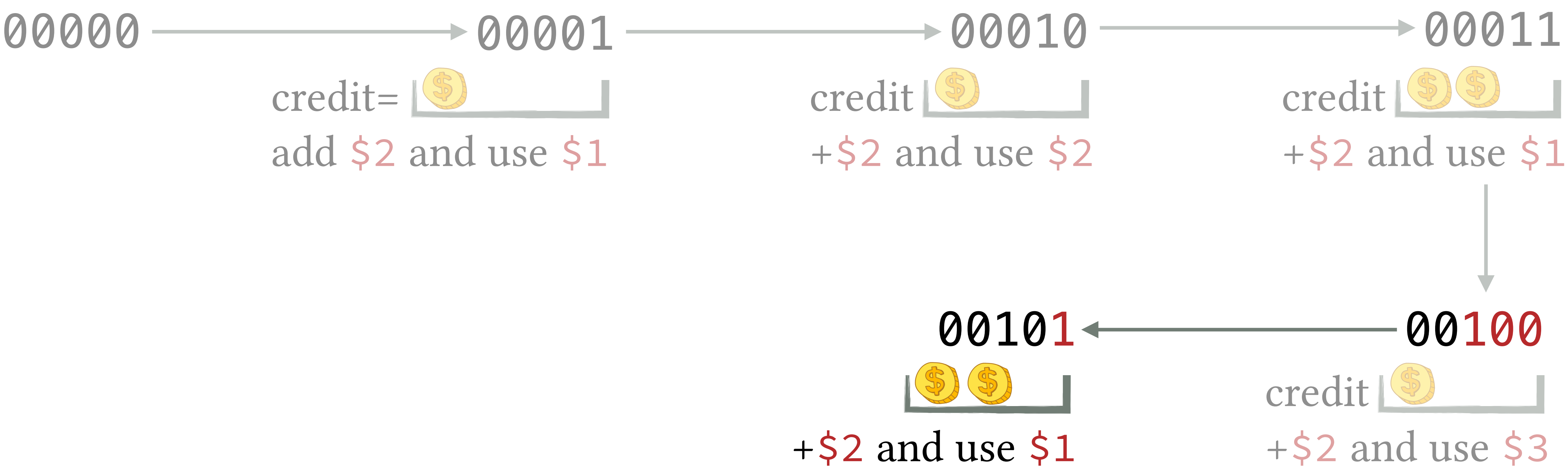
# **Accountants Method:** Incrementing a Binary Counter

**Actual Costs.**

- Flipping `0`'s to `1`'s: `$1`
- Flipping `1`'s to `0`'s: up to `$log(n)`

**New Costs.**

- Flipping `0`'s to `1`'s: `$2`
- Flipping `1`'s to `0`'s: `$0`



`00000` → `00001` → `00010` → `00011` → `00100` → `00101`

`00001`: credit= · add `$2` and use `$1`

`00010`: credit · +`$2` and use `$2`

`00011`: credit · +`$2` and use `$1`

`00100`: credit · +`$2` and use `$3`

`00101`: +`$2` and use `$1`

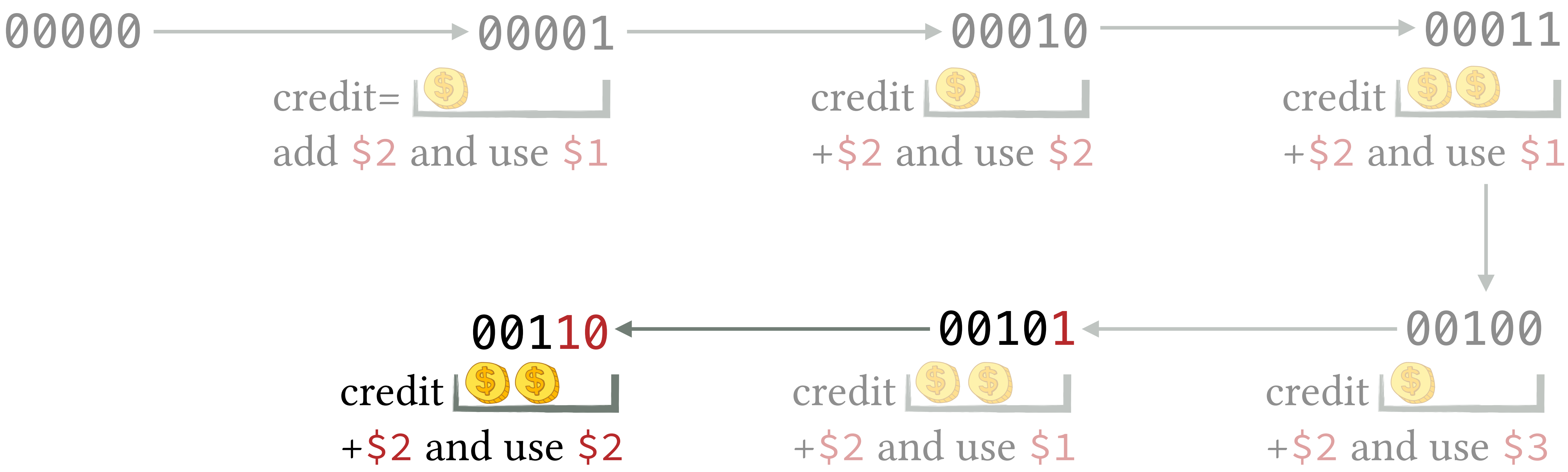
# **Accountants Method:** Incrementing a Binary Counter

**Actual Costs.**

- Flipping `0`'s to `1`'s: `$1`
- Flipping `1`'s to `0`'s: up to `$log(n)`

**New Costs.**

- Flipping `0`'s to `1`'s: `$2`
- Flipping `1`'s to `0`'s: `$0`



00000 → 00001

credit=

add $2 and use $1

00001 → 00010

credit

+$2 and use $2

00010 → 00011

credit

+$2 and use $1

00011 → 00100

credit

+$2 and use $3

00100 → 00101

credit

+$2 and use $1

00101 → 00110

credit

+$2 and use $2

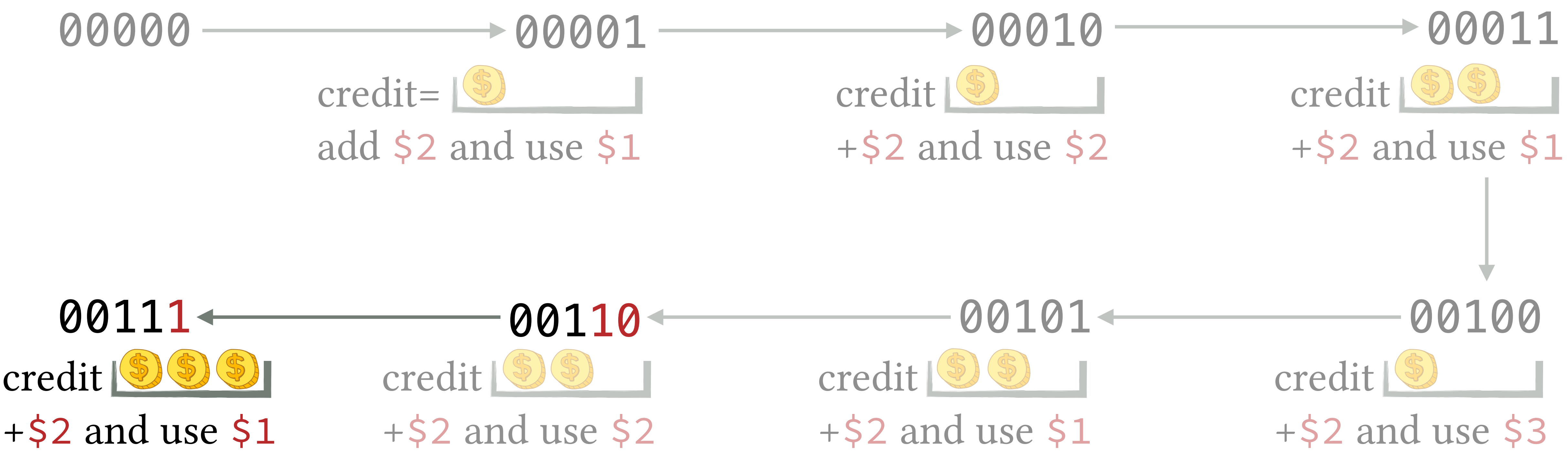
# **Accountants Method:** Incrementing a Binary Counter

**Actual Costs.**

- Flipping `0`'s to `1`'s: `$1`
- Flipping `1`'s to `0`'s: up to `$log(n)`

**New Costs.**

- Flipping `0`'s to `1`'s: `$2`
- Flipping `1`'s to `0`'s: `$0`



`00000` → `00001`

credit=

add `$2` and use `$1`

→ `00010`

credit

+`$2` and use `$2`

→ `00011`

credit

+`$2` and use `$1`

↓ `00100`

credit

+`$2` and use `$3`

← `00101`

credit

+`$2` and use `$1`

← `00110`

credit

+`$2` and use `$2`

← `00111`

credit

+`$2` and use `$1`

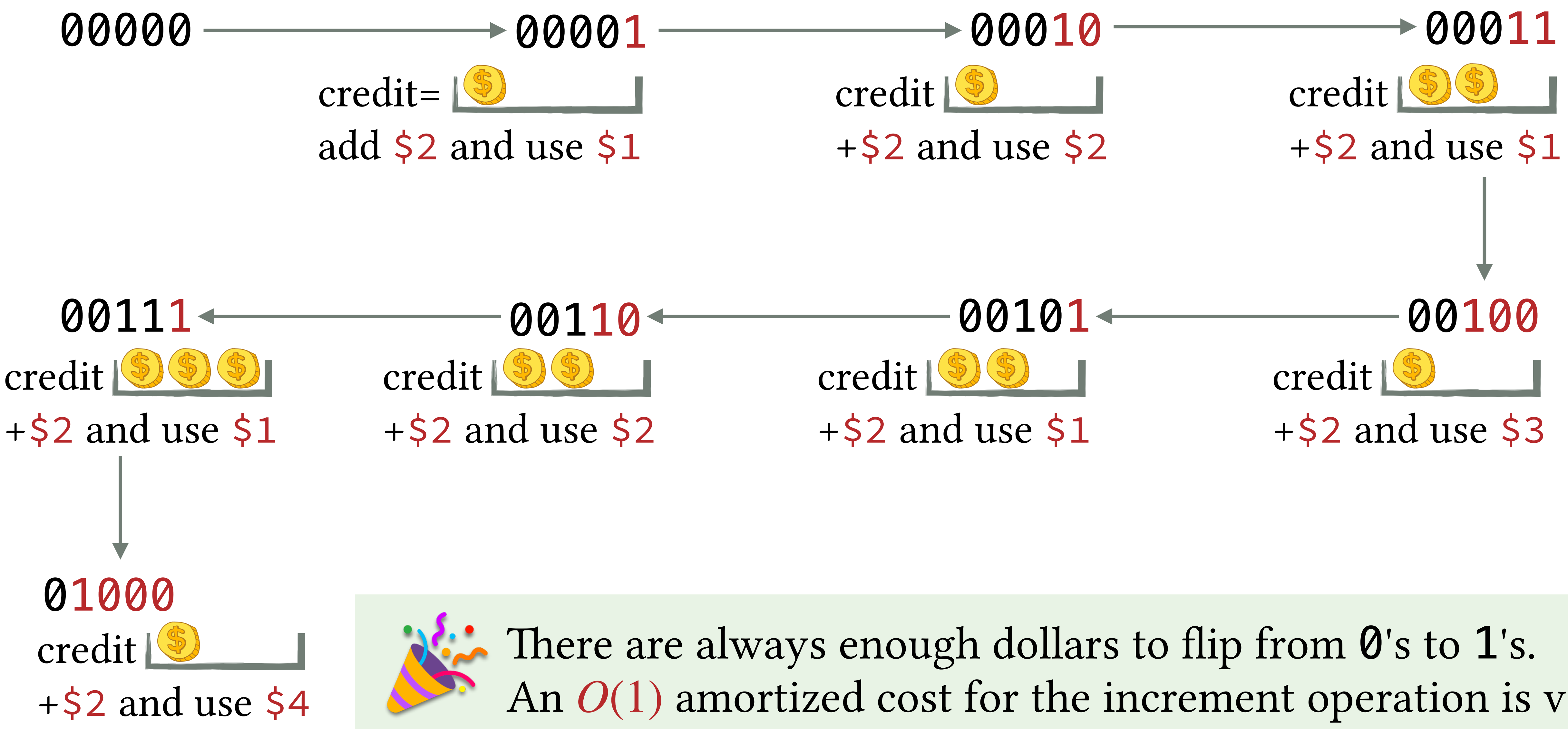
# **Accountants Method:** Incrementing a Binary Counter

**Actual Costs.**

- Flipping `0`'s to `1`'s: `$1`
- Flipping `1`'s to `0`'s: up to `$log(n)`

**New Costs.**

- Flipping `0`'s to `1`'s: `$2`
- Flipping `1`'s to `0`'s: `$0`



`00000`

`00001`
credit=
add `$2` and use `$1`

`00010`
credit
+`$2` and use `$2`

`00011`
credit
+`$2` and use `$1`

`00100`
credit
+`$2` and use `$3`

`00101`
credit
+`$2` and use `$1`

`00110`
credit
+`$2` and use `$2`

`00111`
credit
+`$2` and use `$1`

`01000`
credit
+`$2` and use `$4`

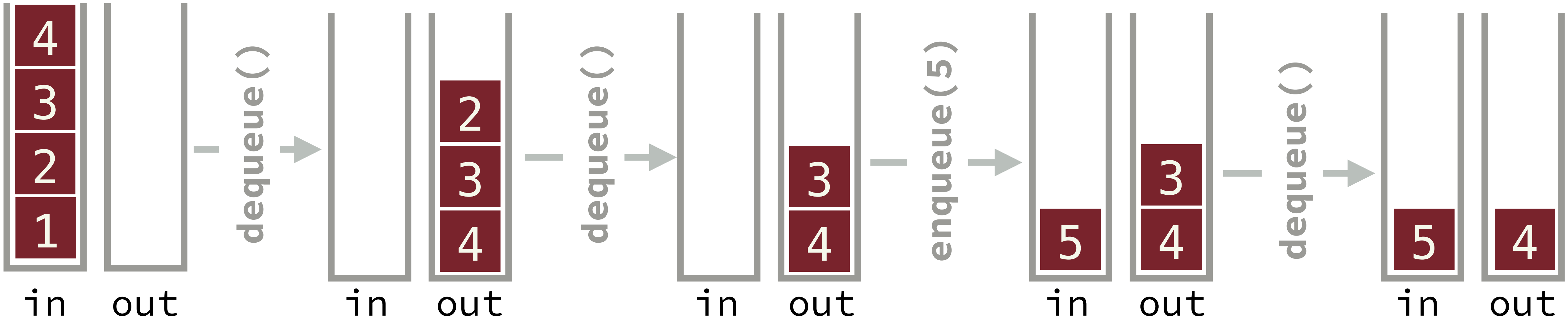
There are always enough dollars to flip from `0`'s to `1`'s.
An $O(1)$ amortized cost for the increment operation is valid!

Whenever we get a `1`, we store an extra `$1` to flip the `1` back to a `0` later on

# **Example:** A Queue using Two Stacks

# **Example:** A Queue using Two Stacks

```
CLASS Queue

Define in  as a Stack
Define out as a Stack

ENQUEUE(x):
  in.PUSH(x)

DEQUEUE(){
  if (out.IS-EMPTY()):
      while (not in.IS-EMPTY()):
          out.PUSH(in.POP());

  return out.POP();
```

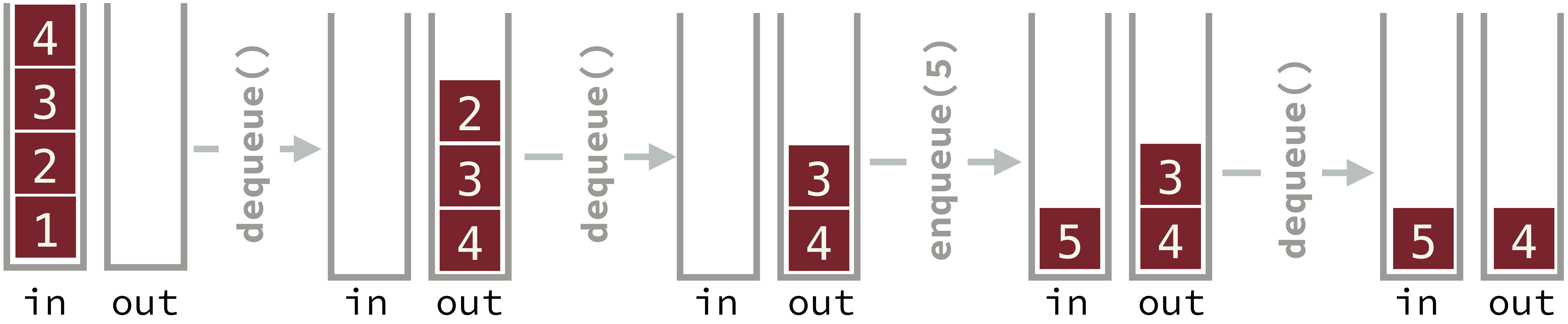


# **Example:** A Queue using Two Stacks

```
CLASS Queue

Define in  as a Stack
Define out as a Stack

ENQUEUE(x):
  in.PUSH(x)

DEQUEUE(){
  if (out.IS-EMPTY()):
      while (not in.IS-EMPTY()):
          out.PUSH(in.POP());

  return out.POP();
```

**Claim.** Any sequence of $N$ **ENQUEUE** and **DEQUEUE** operations runs in $\Theta(N)$ in the worst case.

**Exercise.** Use the Accounting Method to show that the amortized cost for each of the **ENQUEUE** and **DEQUEUE** operations is $O(1)$.



# **Example:** A Queue using Two Stacks

```
CLASS Queue

Define in  as a Stack
Define out as a Stack

ENQUEUE(x):
  in.PUSH(x)

DEQUEUE(){
  if (out.IS-EMPTY()):
      while (not in.IS-EMPTY()):
          out.PUSH(in.POP());

  return out.POP();
```

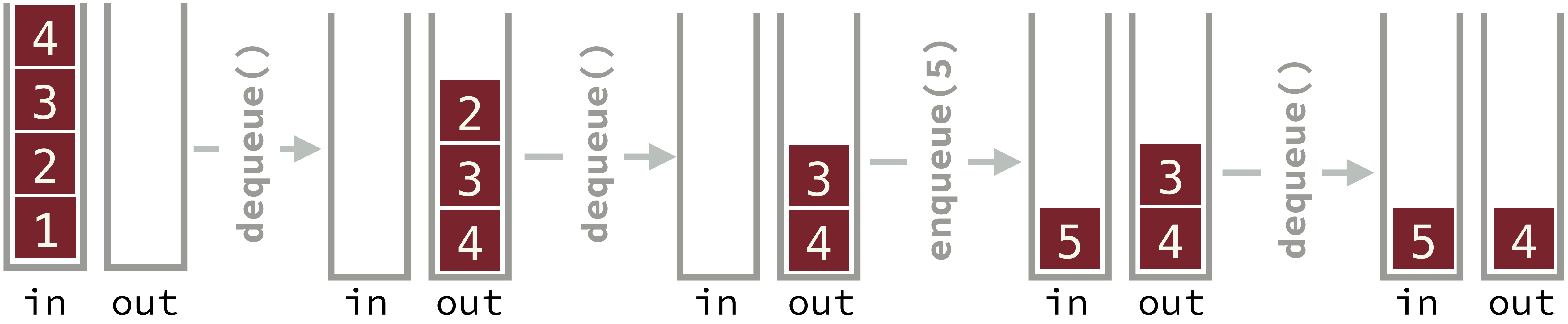
**Claim.** Any sequence of $N$ **ENQUEUE** and **DEQUEUE** operations runs in $\Theta(N)$ in the worst case.

**Exercise.** Use the Accounting Method to show that the amortized cost for each of the **ENQUEUE** and **DEQUEUE** operations is $O(1)$.

**Solution.**

Pay $3 for each **ENQUEUE** operation and $0 for each **DEQUEUE** operation.

Each enqueued item will have $2 saved that can be used later for moving it to the out stack and then for popping it.



# PITFALL

Confusing average case analysis with amortized analysis.

**Average Case Analysis.** Uses probabilistic assumptions to describe how an algorithm is *expected* to behave.

**Example.** The statement "the average case for quicksort is $\Theta(n \log n)$" provides an *expectation* for the performance assuming the probability of all element permutations is the same. The algorithm is **not** guaranteed to have this performance.

**Amortized Analysis.** Does not make any probabilistic assumptions. Provides a *guarantee* for the performance of a sequence of operations.

**Example.** The statement "The **PUSH** and **POP** operations run in $O(1)$ amortized time" mean that every possible sequence of **PUSH** and **POP** operations is guaranteed to have an average running time of $O(1)$ per operation.

# The Potential Method

Definition. Given a sequence of $n$ operations, we define $\Phi(i)$ as a non-negative function that describes the potential after operation $i$, where the amortized cost of operation $i$ is:

$$\hat{c}_i = c_i + \Phi(i) - \Phi(i-1) \longrightarrow \text{change in potential}$$

amortized cost of operation $i$

actual cost of operation $i$

# The Potential Method

Definition. Given a sequence of $n$ operations, we define $\Phi(i)$ as a non-negative function that describes the potential after operation $i$, where the amortized cost of operation $i$ is:

$$\hat{c}_i = c_i + \Phi(i) - \Phi(i-1) \rightarrow$$

change in potential

amortized cost of operation *i*

actual cost of operation *i*

**Claim**. If $\Phi(0) = 0$ and $\Phi(i) \geq 0$ then $\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$

I.e. the amortized cost for each operation is an upper bound for the actual cost!

# The Potential Method

Definition. Given a sequence of $n$ operations, we define $\Phi(i)$ as a non-negative function that describes the potential after operation $i$, where the amortized cost of operation $i$ is:

$$\hat{c}_i = c_i + \Phi(i) - \Phi(i-1)$$

change in potential

amortized cost of operation $i$

actual cost of operation $i$

**Claim**. If $\Phi(0) = 0$ and $\Phi(i) \geq 0$ then $\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$

I.e. the amortized cost for each operation is an upper bound for the actual cost!

**Proof**. $\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} c_i + \Phi(i) - \Phi(i-1)$

# The Potential Method

Definition. Given a sequence of $n$ operations, we define $\Phi(i)$ as a non-negative function that describes the potential after operation $i$, where the amortized cost of operation $i$ is:

$$\hat{c}_i = c_i + \Phi(i) - \Phi(i-1)$$

change in potential

amortized cost of operation $i$

actual cost of operation $i$

**Claim**. If $\Phi(0) = 0$ and $\Phi(i) \geq 0$ then $\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$

I.e. the amortized cost for each operation is an upper bound for the actual cost!

**Proof**. $\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} c_i + \Phi(i) - \Phi(i-1) \quad = \sum_{i=1}^{n} c_i + \sum_{i=1}^{n} (\Phi(i) - \Phi(i-1))$

# The Potential Method

Definition. Given a sequence of $n$ operations, we define $\Phi(i)$ as a non-negative function that describes the potential after operation $i$, where the amortized cost of operation $i$ is:

$$\hat{c}_i = c_i + \Phi(i) - \Phi(i-1)$$

change in potential

amortized cost of operation *i*

actual cost of operation *i*

**Claim**. If $\Phi(0) = 0$ and $\Phi(i) \geq 0$ then $\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$

I.e. the amortized cost for each operation is an upper bound for the actual cost!

**Proof.**

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} c_i + \Phi(i) - \Phi(i-1) = \sum_{i=1}^{n} c_i + \sum_{i=1}^{n} (\Phi(i) - \Phi(i-1))$$

$$= \sum_{i=1}^{n} c_i + \Phi(1) - \Phi(0) + \Phi(2) - \Phi(1) + \Phi(3) - \Phi(2)$$

$$+\Phi(4) - \Phi(3) + \ldots + \Phi(n) - \Phi(n-1)$$

# The Potential Method

**Definition.** Given a sequence of $n$ operations, we define $\Phi(i)$ as a non-negative function that describes the potential after operation $i$, where the amortized cost of operation $i$ is:

$$\hat{c}_i = c_i + \Phi(i) - \Phi(i-1) \longrightarrow$$

change in potential

amortized cost of operation *i*

actual cost of operation *i*

**Claim.** If $\Phi(0) = 0$ and $\Phi(i) \geq 0$ then $\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$

I.e. the amortized cost for each operation is an upper bound for the actual cost!

**Proof.**

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} c_i + \Phi(i) - \Phi(i-1) = \sum_{i=1}^{n} c_i + \sum_{i=1}^{n} (\Phi(i) - \Phi(i-1))$$

$$= \sum_{i=1}^{n} c_i + \cancel{\Phi(1)} - \Phi(0) + \cancel{\Phi(2)} - \cancel{\Phi(1)} + \cancel{\Phi(3)} - \cancel{\Phi(2)} + \cancel{\Phi(4)} - \cancel{\Phi(3)} + \dots + \Phi(n) - \cancel{\Phi(n-1)}$$

# The Potential Method

Definition. Given a sequence of $n$ operations, we define $\Phi(i)$ as a non-negative function that describes the potential after operation $i$, where the amortized cost of operation $i$ is:

$$\hat{c}_i = c_i + \Phi(i) - \Phi(i-1)$$

change in potential

amortized cost of operation *i*

actual cost of operation *i*

**Claim**. If $\Phi(0) = 0$ and $\Phi(i) \geq 0$ then $\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$

I.e. the amortized cost for each operation is an upper bound for the actual cost!

**Proof**. 
$$\begin{aligned}
\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} c_i + \Phi(i) - \Phi(i-1) &= \sum_{i=1}^{n} c_i + \sum_{i=1}^{n} (\Phi(i) - \Phi(i-1)) \\
&= \sum_{i=1}^{n} c_i + \cancel{\Phi(1)} - \Phi(0) + \cancel{\Phi(2)} - \cancel{\Phi(1)} + \cancel{\Phi(3)} - \cancel{\Phi(2)} \\
&\quad + \cancel{\Phi(4)} - \cancel{\Phi(3)} + \ldots + \Phi(n) - \cancel{\Phi(n-1)} \\
&= \sum_{i=1}^{n} c_i - \Phi(0) + \Phi(n)
\end{aligned}$$

# The Potential Method

Definition. Given a sequence of $n$ operations, we define $\Phi(i)$ as a non-negative function that describes the potential after operation $i$, where the amortized cost of operation $i$ is:

$$\hat{c}_i = c_i + \Phi(i) - \Phi(i-1) \longrightarrow$$

change in potential

amortized cost of operation *i*

actual cost of operation *i*

**Claim**. If $\Phi(0) = 0$ and $\Phi(i) \geq 0$ then $\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$

I.e. the amortized cost for each operation is an upper bound for the actual cost!

**Proof.**

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} c_i + \Phi(i) - \Phi(i-1) \quad = \sum_{i=1}^{n} c_i + \sum_{i=1}^{n} (\Phi(i) - \Phi(i-1))$$

$$= \sum_{i=1}^{n} c_i + \cancel{\Phi(1)} - \Phi(0) + \cancel{\Phi(2)} - \cancel{\Phi(1)} + \cancel{\Phi(3)} - \cancel{\Phi(2)} + \cancel{\Phi(4)} - \cancel{\Phi(3)} + \ldots + \Phi(n) - \cancel{\Phi(n-1)}$$

$$= \sum_{i=1}^{n} c_i - \cancel{\Phi(0)} + \Phi(n) \quad \geq \sum_{i=1}^{n} c_i$$

# The Potential **Method**

Definition. Given a sequence of $n$ operations, we define $\Phi(i)$ as a non-negative function that describes the potential after operation $i$, where the amortized cost of operation $i$ is:

$$\hat{c}_i = c_i + \Phi(i) - \Phi(i-1)$$

change in potential

amortized cost of operation *i*

actual cost of operation *i*

**Claim**. If $\Phi(0) = 0$ and $\Phi(i) \geq 0$ then $\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$

I.e. the amortized cost for each operation is an upper bound for the actual cost!

**Proof**.

$$\begin{aligned}\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} c_i + \Phi(i) - \Phi(i-1) &= \sum_{i=1}^{n} c_i + \sum_{i=1}^{n} (\Phi(i) - \Phi(i-1)) \\ &= \sum_{i=1}^{n} c_i + \cancel{\Phi(1)} - \Phi(0) + \cancel{\Phi(2)} - \cancel{\Phi(1)} + \cancel{\Phi(3)} - \cancel{\Phi(2)} \\ &\quad + \cancel{\Phi(4)} - \cancel{\Phi(3)} + \ldots + \Phi(n) - \cancel{\Phi(n-1)} \\ &= \sum_{i=1}^{n} c_i - \cancel{\Phi(0)} + \Phi(n) \geq \sum_{i=1}^{n} c_i\end{aligned}$$

## **How do we choose the** Potential **function?**

# Example # 1: Resizing Arrays

**Bad Choice of $\Phi(i)$.** Let $\Phi(i)$ be equal to the number of elements in the array after applying operation operation $i$.

- $\Phi(0) = 0$ because initially the array has 0 elements
- $\Phi(i) = i$ is always non-negative

**Case # 1 - no resize:**

$$\hat{c}_i = \underbrace{1}_{\text{actual cost}} + \Phi(i) - \Phi(i-1) = 1 + \underbrace{i}_{\text{number of elements now}} - (\underbrace{i}_{\text{number of elements before}} - 1) \overbrace{= 2}^{\text{amortized cost} = O(1)}$$

# Example # 1: Resizing Arrays

**Bad Choice of $\Phi(i)$.** Let $\Phi(i)$ be equal to the number of elements in the array after applying operation operation $i$.

- $\Phi(0) = 0$ because initially the array has 0 elements
- $\Phi(i) = i$ is always non-negative

amortized cost = $O(1)$

Case # 1 - no resize: $\hat{c}_i = 1 + \Phi(i) - \Phi(i-1) = 1 + i - (i-1) = 2$

actual cost

number of elements now

number of elements before

Case # 2 - with resize: $\hat{c}_i = i + \Phi(i) - \Phi(i-1) = i + i - (i-1) = i + 1$

amortized cost = $O(n)$

This upper bound is not interesting!
It is the same as the actual worst-case running time we already know!

😒 **WHO CARES?**

# Example # 1: Resizing Arrays

Good Choice of $\Phi(i)$. Let $N_i =$ the capacity of the array after operation $i$, and let

$$\text{let } \Phi(i) = 2\left(i - \frac{N_i}{2}\right) = 2i - N_i$$

Double the number of elements in the second half of the array (after operation *i*)

Hence:

- $\Phi(0) = 0$ because i = 0 and $N_i = 0$
- $\Phi(i) = 2i - N \geq 0$ because the array is *never* less than half full

Case # 1 - no resize:

$$\hat{c}_i = 1 + \Phi(i) - \Phi(i-1) = 1 + (2i - N_i) - (2(i-1) - N_{i-1})$$

$$= 1 + \cancel{2i} - \cancel{N_i} - \cancel{2i} + 2 + \cancel{N_{i-1}} = 3$$

actual cost

capacity before and after is the same

# Example # 1: Resizing Arrays

Good Choice of $\Phi(i)$. Let $N_i =$ the capacity of the array after operation $i$, and let

$$\text{let } \Phi(i) = 2\left(i - \frac{N_i}{2}\right) = 2i - N_i$$

Double the number of elements in the second half of the array (after operation *i*)

Hence:

- $\Phi(0) = 0$ because i = 0 and $N_i = 0$
- $\Phi(i) = 2i - N \geq 0$ because the array is *never* less than half full

Case # 1 - no resize: $\hat{c}_i = 1 + \Phi(i) - \Phi(i-1) = 1 + (2i - N_i) - (2(i-1) - N_{i-1})$

$$= 1 + \cancel{2i} - \cancel{N_i} - \cancel{2i} + 2 + \cancel{N_{i-1}} = 3$$

Case # 2 - with resize: $\hat{c}_i = i + \Phi(i) - \Phi(i-1) = i + (2i - N_i) - \left(2(i-1) - \frac{N_i}{2}\right)$

actual cost

capacity before is half the current capacity

# Example # 1: Resizing Arrays

Good Choice of $\Phi(i)$. Let $N_i =$ the capacity of the array after operation $i$, and let

$$\text{let } \Phi(i) = 2(i - \frac{N_i}{2}) = 2i - N_i$$

Double the number of elements in the second half of the array (after operation *i*)

Hence:

- $\Phi(0) = 0$ because i = 0 and $N_i = 0$
- $\Phi(i) = 2i - N \geq 0$ because the array is *never* less than half full

Case # 1 - no resize:

$$\hat{c}_i = 1 + \Phi(i) - \Phi(i-1) = 1 + (2i - N_i) - (2(i-1) - N_{i-1})$$

$$= 1 + \cancel{2i} - \cancel{N_i} - \cancel{2i} + 2 + \cancel{N_{i-1}} = 3$$

Case # 2 - with resize:

$$\hat{c}_i = i + \Phi(i) - \Phi(i-1) = i + (2i - N_i) - (2(i-1) - \frac{N_i}{2})$$

$$= i + \cancel{2i} - N_i - \cancel{2i} + 2 + \frac{N_i}{2}$$

$$= \frac{N_i}{2} + 1 - N_i + 2 + \frac{N_i}{2}$$

current number of elements = half the current capacity + 1

# Example # 1: Resizing Arrays

Good Choice of $\Phi(i)$. Let $N_i =$ the capacity of the array after operation $i$, and let

$$\text{let } \Phi(i) = 2(i - \frac{N_i}{2}) = 2i - N_i$$

← Double the number of elements in the second half of the array (after operation $i$)

Hence:

- $\Phi(0) = 0$ because i = 0 and $N_i = 0$
- $\Phi(i) = 2i - N \geq 0$ because the array is *never* less than half full

Case # 1 - no resize: $\hat{c}_i = 1 + \Phi(i) - \Phi(i-1) = 1 + (2i - N_i) - (2(i-1) - N_{i-1})$

$$= 1 + \cancel{2i} - \cancel{N_i} - \cancel{2i} + 2 + \cancel{N_{i-1}} = 3$$

Case # 2 - with resize: $\hat{c}_i = i + \Phi(i) - \Phi(i-1) = i + (2i - N_i) - (2(i-1) - \frac{N_i}{2})$

$$= i + \cancel{2i} - N_i - \cancel{2i} + 2 + \frac{N_i}{2}$$

$$= \cancel{\frac{N_i}{2}} + 1 - \cancel{N_i} + 2 + \cancel{\frac{N_i}{2}}$$

$$= 3$$

← amortized cost = $O(1)$

This upper bound is interesting*!*

This is much lower than worst case running time we know, which is O($n$)

🙂 YES!!

# Example # 2: Counting Bit-Flips

**Potential Function.** Let $\Phi(i) = N_i$ , where $N_i$ *is* the number of 1s after operation *i*.

Hence:

- $\Phi(0) = 0$ because the number of 1s in 000000… is 0
- $\Phi(i) = N_i \geq 0$ because the number of bits is never negative

amortized cost = $O(1)$

Case # 1 - Flip $0 \rightarrow 1$: $\hat{c}_i = \underbrace{1}_{\text{actual cost}} + \Phi(i) - \Phi(i-1) = 1 + \underbrace{N_i}_{\text{number of 1s now}} - (\underbrace{N_i - 1}_{\text{number of 1s before}}) = 2$

# Example # 2: Counting Bit-Flips

Potential Function. Let $\Phi(i) = N_i$, where $N_i$ *is* the number of 1s after operation $i$.

Hence:

- $\Phi(0) = 0$ because the number of 1s in 000000… is 0
- $\Phi(i) = N_i \geq 0$ because the number of bits is never negative

amortized cost = $O(1)$

Case # 1 - Flip 0 → 1: $\hat{c}_i = 1 + \Phi(i) - \Phi(i-1) = 1 + N_i - (N_i - 1) = 2$

Case # 2 - Flip 1 → 0: $\hat{c}_i = N_{i-1} + 1 + \Phi(i) - \Phi(i-1) = N_{i-1} + 1 + 1 - N_{i-1}$

in the worst case, all the 1s are flipped to 0s and one 0 is flipped to 1

in the worst case, only one 1 remains

# Example # 2: Counting Bit-Flips

Potential Function. Let $\Phi(i) = N_i$, where $N_i$ *is* the number of 1s after operation *i*.

Hence:

- $\Phi(0) = 0$ because the number of 1s in 000000… is 0
- $\Phi(i) = N_i \geq 0$ because the number of bits is never negative

amortized cost = $O(1)$

Case # 1 - Flip 0 → 1: $\hat{c}_i = 1 + \Phi(i) - \Phi(i-1) = 1 + N_i - (N_i - 1) = 2$

Case # 2 - Flip 1 → 0: $\hat{c}_i = N_{i-1} + 1 + \Phi(i) - \Phi(i-1) = N_{i-1} + 1 + 1 - N_{i-1}$

$= 2$ ← amortized cost = $O(1)$

# Example # 3: A Queue using Two Stacks

Potential Function. Let $\Phi(i) = 2a_i + b_i$ , where:

- $a_i$ = number of elements in the **`in`** stack after operation $i$
- $b_i$ = the number of elements in the **`out`** stack after operation $i$

Hence:

- $\Phi(0) = 0$ Initially, no elements in any of the stacks
- $\Phi(i) = 2a_i + b_i \geq 0$ The number of elements is never negative

Case # 1 - Enqueue: $\hat{c}_i = 1 + \Phi(i) - \Phi(i-1) = 1 + (2a_i + b_i) - (2(a_i - 1) + b_i)$

actual cost

the **`in`** stack increases by 1

the **`out`** stack does not change

# Example # 3: A Queue using Two Stacks

Potential Function. Let $\Phi(i) = 2a_i + b_i$ , where:

- $a_i$ = number of elements in the **`in`** stack after operation $i$
- $b_i$ = the number of elements in the **`out`** stack after operation $i$

Hence:

- $\Phi(0) = 0$ Initially, no elements in any of the stacks
- $\Phi(i) = 2a_i + b_i \geq 0$ The number of elements is never negative

Case # 1 - Enqueue:

$$\hat{c}_i = 1 + \Phi(i) - \Phi(i-1) = 1 + (2a_i + b_i) - (2(a_i - 1) + b_i)$$

$$= 1 + \cancel{2a_i} + \cancel{b_i} - \cancel{2a_i} + 2 - \cancel{b_i}$$

$= 3$ ← amortized cost = $O(1)$

# Example # 3: A Queue using Two Stacks

Potential Function. Let $\Phi(i) = 2a_i + b_i$, where:

- $a_i$ = number of elements in the **`in`** stack after operation $i$
- $b_i$ = the number of elements in the **`out`** stack after operation $i$

Hence:

- $\Phi(0) = 0$ Initially, no elements in any of the stacks
- $\Phi(i) = 2a_i + b_i \geq 0$ The number of elements is never negative

Case # 1 - Enqueue:

$$\hat{c}_i = 1 + \Phi(i) - \Phi(i-1) = 1 + (2a_i + b_i) - (2(a_i - 1) + b_i)$$

$$= 1 + \cancel{2a_i} + \cancel{b_i} - \cancel{2a_i} + 2 - \cancel{b_i}$$

$= 3$ ← amortized cost = $O(1)$

Case # 2 - Dequeue:
assuming **`out`** is empty

$$\hat{c}_i = a_{i-1} + 1 + \Phi(i) - \Phi(i-1)$$

↑

move $a_{i-1}$ elements from **`in`**
to **`out`** + pop 1 element

# Example # 3: A Queue using Two Stacks

Potential Function. Let $\Phi(i) = 2a_i + b_i$ , where:

- $a_i$ = number of elements in the **`in`** stack after operation $i$
- $b_i$ = the number of elements in the **`out`** stack after operation $i$

Hence:

- $\Phi(0) = 0$ Initially, no elements in any of the stacks
- $\Phi(i) = 2a_i + b_i \geq 0$ The number of elements is never negative

Case # 1 - Enqueue:

$$\hat{c}_i = 1 + \Phi(i) - \Phi(i-1) = 1 + (2a_i + b_i) - (2(a_i - 1) + b_i)$$

$$= 1 + \cancel{2a_i} + \cancel{b_i} - \cancel{2a_i} + 2 - \cancel{b_i}$$

$$= 3 \leftarrow$$

amortized cost = $O(1)$

Case # 2 - Dequeue:

assuming **`out`** is empty

$$\hat{c}_i = a_{i-1} + 1 + \Phi(i) - \Phi(i-1)$$

$$= a_{i-1} + 1 + (2 \times 0 + b_i) - (2a_{i-1} + 0)$$

**`in`** becomes empty

**`out`** was empty

# Example # 3: A Queue using Two Stacks

Potential Function. Let $\Phi(i) = 2a_i + b_i$, where:

- $a_i$ = number of elements in the **`in`** stack after operation $i$
- $b_i$ = the number of elements in the **`out`** stack after operation $i$

Hence:

- $\Phi(0) = 0$ Initially, no elements in any of the stacks
- $\Phi(i) = 2a_i + b_i \geq 0$ The number of elements is never negative

Case # 1 - Enqueue:

$$\hat{c}_i = 1 + \Phi(i) - \Phi(i-1) = 1 + (2a_i + b_i) - (2(a_i - 1) + b_i)$$

$$= 1 + \cancel{2a_i} + \cancel{b_i} - \cancel{2a_i} + 2 - \cancel{b_i}$$

$$= 3 \longleftarrow$$ amortized cost = $O(1)$

Case # 2 - Dequeue:
assuming **`out`** is empty

$$\hat{c}_i = a_{i-1} + 1 + \Phi(i) - \Phi(i-1)$$

$$= a_{i-1} + 1 + (2 \times 0 + b_i) - (2a_{i-1} + 0)$$

$$= a_{i-1} + 1 + b_i - 2a_{i-1} =$$

# Example # 3: A Queue using Two Stacks

Potential Function. Let $\Phi(i) = 2a_i + b_i$ , where:

- $a_i$ = number of elements in the **`in`** stack after operation $i$
- $b_i$ = the number of elements in the **`out`** stack after operation $i$

Hence:

- $\Phi(0) = 0$ Initially, no elements in any of the stacks
- $\Phi(i) = 2a_i + b_i \geq 0$ The number of elements is never negative

Case # 1 - Enqueue:

$$\hat{c}_i = 1 + \Phi(i) - \Phi(i-1) = 1 + (2a_i + b_i) - (2(a_i - 1) + b_i)$$

$$= 1 + \cancel{2a_i} + \cancel{b_i} - \cancel{2a_i} + 2 - \cancel{b_i}$$

$$= 3$$

amortized cost = $O(1)$

Case # 2 - Dequeue:
assuming **`out`** is empty

$$\hat{c}_i = a_{i-1} + 1 + \Phi(i) - \Phi(i-1)$$

$$= a_{i-1} + 1 + (2 \times 0 + b_i) - (2a_{i-1} + 0)$$

$$= \cancel{a_{i-1}} + 1 + b_i - \cancel{2}a_{i-1} = 1 + b_i - a_{i-1}$$

# Example # 3: A Queue using Two Stacks

**Potential Function.** Let $\Phi(i) = 2a_i + b_i$ , where:

- $a_i$ = number of elements in the `in` stack after operation $i$
- $b_i$ = the number of elements in the `out` stack after operation $i$

Hence:

- $\Phi(0) = 0$ Initially, no elements in any of the stacks
- $\Phi(i) = 2a_i + b_i \geq 0$ The number of elements is never negative

**Case # 1 - Enqueue:**

$$\hat{c}_i = 1 + \Phi(i) - \Phi(i-1) = 1 + (2a_i + b_i) - (2(a_i - 1) + b_i)$$

$$= 1 + \cancel{2a_i} + \cancel{b_i} - \cancel{2a_i} + 2 - \cancel{b_i}$$

$$= 3 \longleftarrow \text{amortized cost} = O(1)$$

**Case # 2 - Dequeue:**
assuming `out` is empty

$$\hat{c}_i = a_{i-1} + 1 + \Phi(i) - \Phi(i-1)$$

$$= a_{i-1} + 1 + (2 \times 0 + b_i) - (2a_{i-1} + 0)$$

$$= \cancel{a_{i-1}} + 1 + b_i - \cancel{2a_{i-1}} = 1 + \underline{b_i - a_{i-1}} = 0 \longleftarrow \text{amortized cost} = O(1)$$

# of elements in `out` now is 1 less than the # of elements in `in` before