# Selected Solutions for Chapter 17: Amortized Analysis

## Solution to Exercise 17.1-3

Let $c_i$ = cost of $i$th operation.

$$c_i = \begin{cases} i & \text{if } i \text{ is an exact power of 2}, \\ 1 & \text{otherwise}. \end{cases}$$

| Operation | Cost |
|---|---|
| 1 | 1 |
| 2 | 2 |
| 3 | 1 |
| 4 | 4 |
| 5 | 1 |
| 6 | 1 |
| 7 | 1 |
| 8 | 8 |
| 9 | 1 |
| 10 | 1 |
| ⋮ | ⋮ |

$n$ operations cost

$$\sum_{i=1}^{n} c_i \le n + \sum_{j=0}^{\lg n} 2^j = n + (2n - 1) < 3n .$$

(Note: Ignoring floor in upper bound of $\sum 2^j$.)

$$\text{Average cost of operation} = \frac{\text{Total cost}}{\text{\# operations}} < 3 .$$

By aggregate analysis, the amortized cost per operation $= O(1)$.

## Solution to Exercise 17.2-2

Let $c_i$ = cost of $i$th operation.

$$c_i = \begin{cases} i & \text{if } i \text{ is an exact power of 2 ,} \\ 1 & \text{otherwise .} \end{cases}$$

Charge each operation \$3 (amortized cost $\widehat{c}_i$).

- If $i$ is not an exact power of 2, pay \$1, and store \$2 as credit.
- If $i$ is an exact power of 2, pay \$$i$, using stored credit.

| Operation | Cost | Actual cost | Credit remaining |
|---|---|---|---|
| 1 | 3 | 1 | 2 |
| 2 | 3 | 2 | 3 |
| 3 | 3 | 1 | 5 |
| 4 | 3 | 4 | 4 |
| 5 | 3 | 1 | 6 |
| 6 | 3 | 1 | 8 |
| 7 | 3 | 1 | 10 |
| 8 | 3 | 8 | 5 |
| 9 | 3 | 1 | 7 |
| 10 | 3 | 1 | 9 |
| ⋮ | ⋮ | ⋮ | ⋮ |

Since the amortized cost is \$3 per operation, $\sum_{i=1}^{n} \widehat{c}_i = 3n$.

We know from Exercise 17.1-3 that $\sum_{i=1}^{n} c_i < 3n$.

Then we have $\sum_{i=1}^{n} \widehat{c}_i \ge \sum_{i=1}^{n} c_i \Rightarrow$ credit $=$ amortized cost $-$ actual cost $\ge 0$.

Since the amortized cost of each operation is $O(1)$, and the amount of credit never goes negative, the total cost of $n$ operations is $O(n)$.

## Solution to Exercise 17.2-3

We introduce a new field $A.max$ to hold the index of the high-order 1 in $A$. Initially, $A.max$ is set to $-1$, since the low-order bit of $A$ is at index 0, and there are initially no 1's in $A$. The value of $A.max$ is updated as appropriate when the counter is incremented or reset, and we use this value to limit how much of $A$ must be looked at to reset it. By controlling the cost of RESET in this way, we can limit it to an amount that can be covered by credit from earlier INCREMENTS.

```
INCREMENT($A$)
$i = 0$
while $i < A.length$ and $A[i] == 1$
    $A[i] = 0$
    $i = i + 1$
if $i < A.length$
    $A[i] = 1$
    // Additions to book's INCREMENT start here.
    if $i > A.max$
        $A.max = i$
else $A.max = -1$
```

```
RESET($A$)
for $i = 0$ to $A.max$
    $A[i] = 0$
$A.max = -1$
```

As for the counter in the book, we assume that it costs \$1 to flip a bit. In addition, we assume it costs \$1 to update $A.max$.

Setting and resetting of bits by INCREMENT will work exactly as for the original counter in the book: \$1 will pay to set one bit to 1; \$1 will be placed on the bit that is set to 1 as credit; the credit on each 1 bit will pay to reset the bit during incrementing.

In addition, we'll use \$1 to pay to update *max*, and if *max* increases, we'll place an additional \$1 of credit on the new high-order 1. (If *max* doesn't increase, we can just waste that \$1—it won't be needed.) Since RESET manipulates bits at positions only up to $A.max$, and since each bit up to there must have become the high-order 1 at some time before the high-order 1 got up to $A.max$, every bit seen by RESET has \$1 of credit on it. So the zeroing of bits of $A$ by RESET can be completely paid for by the credit stored on the bits. We just need \$1 to pay for resetting *max*.

Thus charging \$4 for each INCREMENT and \$1 for each RESET is sufficient, so the sequence of $n$ INCREMENT and RESET operations takes $O(n)$ time.