amortize.

    1: to pay off (an obligation, such as a mortgage) gradually usually by periodic payments of principal and interest or by payments to a sinking fund
    // *amortize a loan*

    2: to gradually reduce or write off the cost or value of (something, such as an asset)
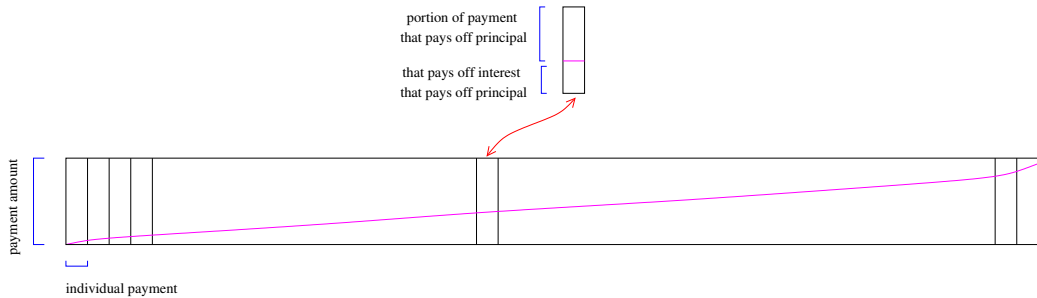   // *amortize goodwill*
    // *amortize machinery*

    History and Etymology for amortize
    Middle English *amortisen* to kill, alienate in mortmain, from Anglo-French *amorteser*, alteration of *amortir*, from Vulgar Latin **admortire* to kill, from Latin *ad-* + *mort-*, *mors* death—more at *MURDER*

portion of payment
that pays off principal

that pays off interest
that pays off principal

payment amount

individual payment

```java
public interface Stack<E> {
    void push(E item);
    E top();
    E pop();
    boolean isEmpty();
}

public class MultipopStackDecorator<E> implements Stack<E> {
    private Stack<E> internal;
    public MultipopStackDecorator(Stack<E> internal) { this.internal = internal; }
    public void push(E item) { internal.push(item); }
    public E top() { return internal.top(); }
    public E pop() { return internal.pop(); }
    public boolean isEmpty() { return internal.isEmpty(); }

    public void multipop(int k) {
        for (; k > 0; k--) internal.pop();
    }
}
```

$$\left.\begin{array}{l} \texttt{multipop}(n) \\ \texttt{multipop}(n) \\ \quad \cdots \\ \quad \cdots \\ \texttt{multipop}(n) \\ \texttt{multipop}(n) \end{array}\right\} n$$

$$\left.\begin{array}{l} \texttt{push}(n) \\ \texttt{push}(n) \\ \quad \cdots \\ \quad \cdots \\ \texttt{push}(n) \end{array}\right\} n-1$$
$$\texttt{multipop}(n)$$

```java
public class BinaryCounter {
    private boolean[] bits;

    public BinaryCounter(int max) {
        bits = new boolean[(int) Math.ceil(Math.lg(max))];
    }

    public increment() {
        int i = 0;
        while (i < bits.length && bits[i]) bits[i++] = false;
        if (i < bits.legnth) bits[i] = true;
    }

    public read() {
        int result = 0
        for (int i = 0; i < bits.length; i++)
            result += 1 << i;
        return result;
    }
}
```

operation	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
cost	1	2	1	4	1	1	1	8	1	1	1	1	1	1	1	16
running total	1	3	4	8	9	10	11	19	20	21	22	23	24	25	26	42

operation	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
cost	1	2	1	4	1	1	1	8	1	1	1	1	1	1	1	16
running total	1	3	4	8	9	10	11	19	20	21	22	23	24	25	26	42

The total is

$$
\begin{aligned}
n + \sum_{j=1}^{\lfloor \lg n \rfloor} 2^j - \lfloor \lg n \rfloor &= n + \frac{2^{\lfloor \lg n \rfloor + 1} - 2}{2 - 1} - \lfloor \lg n \rfloor \\
&= n + 2 \cdot 2^{\lfloor \lg n \rfloor} - 2 - \lfloor \lg n \rfloor \\
&\leq n + 2 \cdot 2^{\lg n} \\
&= n + 2n = 3n
\end{aligned}
$$

Suppose you own a major soft drink company, and you have a million vending machines in place around the country that are built to dispense bottles for a nickel.

What do you do when you need to increase the price of a bottle, say, to 6¢?

(a) Put up with decreasing profits for the next 50 years or so.

(b) Invest untold sums into replacing all vending machines.

(c) Lobby the government to recollect and destroy all nickels and then to issue a new 6¢ coin with the same weight and size as a nickel.

(d) Amortize the price increase by making every 6th bottle in the machines empty.

17.2-2. The idea is that for every operation, we should pay for itself and for all the powers of two that follow it. Of course, we don't know how many powers of 2 will be executed. Instead, consider each operation to cost 3 units: one for itself, one counting towards the next power of 2 ("next" is "inclusive"—it the operation is itself a power to 2, this contributes to its own real cost), and one to pay for an earlier (before the previous power of 2, inclusive) operation's contribution towards the next power of 2. $\hat{c}_i = 3$. So,

$$\sum_{i=1}^{n} \hat{c}_i = 3n \geq \sum_{i=1}^{n} c_i$$

. . . where the last inequality was shown in 17.1-3.

Φ maps (the state of) a data structure to some value. $\Phi(D-i) - \Phi(D_{i-1})$ represents how the change in state from D_{i-1} to D_i affects that value.

For the multi-pop stack, $\Phi(D)$ is the number of items in the stack.

For the binary counter, $\Phi(D)$ is the number of 1s.

17.3-2. For each operation, we want first to raise the potential by 3 (from the same principle as before), but then also subtract the amount that is actually used up.

$$\Phi(D_i) - \Phi(D_{i-1}) = \begin{cases} 3 - i & i \text{ is a power of 2} \\ 3 - 1 = 2 & \text{otherwise} \end{cases}$$

So, for non-powers-of-two,

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 2 = 3$$

For powers-of-two,

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = i + 3 - i = 3$$

. . . which concurs with our accounting-method analysis.