Amortized Analysis

YAO ZHAO

Amortized Analysis

In computer science, amortized analysis is a method for analyzing a given algorithm's complexity, or how much of a resource, especially time or memory, it takes to execute. The motivation for amortized analysis is that looking at the worst-case run time per operation, rather than per algorithm, can be too pessimistic.

While certain operations for a given algorithm may have a significant cost in resources, other operations may not be as costly. Amortized analysis considers both the costly and less costly operations together over the whole series of operations of the algorithm. This may include accounting for different types of input, length of the input, and other factors that affect its performance.

https://en.wikipedia.org/wiki/Amortized_analysis

Methods

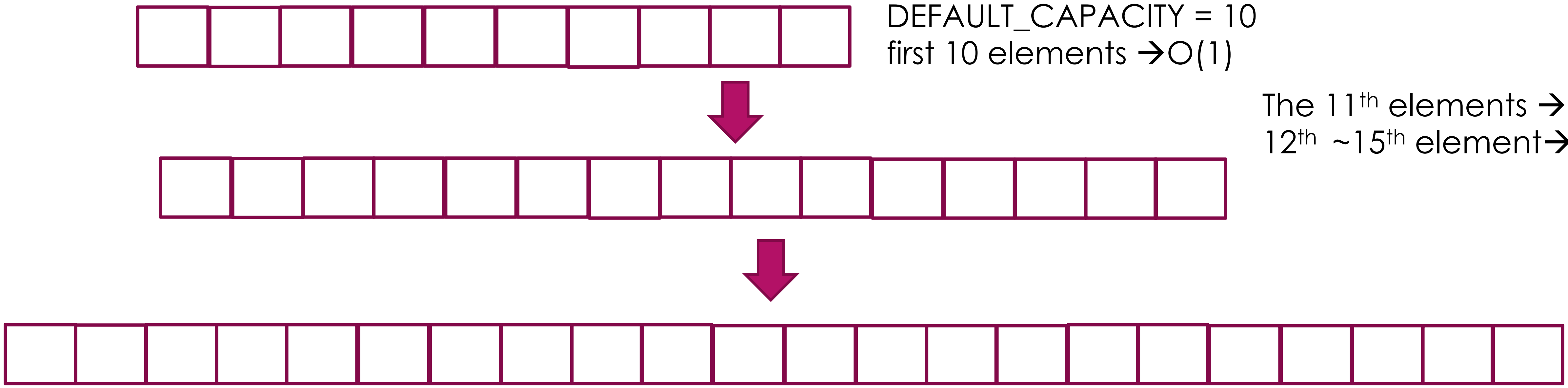
- Aggregate method
- Accounting method
- Potential method

All above are correct. The choice of which to use depends on which is most convenient for a particular situation.

Aggregate method

- Aggregate analysis determines the upper bound *T*(*n*) on the total cost of a sequence of *n* operations, then calculates the amortized cost to be *T*(*n*) / *n*.*(*Kozen, Dexter (Spring 2011). "CS 3110 Lecture 20: Amortized Analysis". Cornell University. Retrieved 14 March 2015.*)*

Example：ArrayList(JAVA)



DEFAULT_CAPACITY = 10
first 10 elements →O(1)

The 11th elements →O(n)
12th ~15th element→O(1)

The 16th elements →O(n)
17th ~22th element→O(1)

Obviously， worst case operation cost O(n), but we shouldn’t consider that insert n elements will cost $O(n^2)$

Aggregate method: ArrayList

- $T(n) = \sum_{i=1}^{n} c_i$

$$= \sum_{i=1, i-1 \neq 10\times1.5^k}^{n} c_i + \sum_{i=1, i-1 = 10\times1.5^k}^{n} c_i$$

$$= \sum_{i=1, i-1 \neq 10\times1.5^k}^{n} 1 + \sum_{i=1, i-1 = 10\times1.5^k}^{n} (10\times1.5^k + 1)$$

$$= n + \sum_{k=0}^{\log_{1.5}\frac{n}{10}} 10\times1.5^k = n + 10 * \left(\frac{1-1.5^{\log_{1.5}\frac{n}{10}+1}}{1-1.5}\right) = n + 10 * \left(\frac{1-1.5 * \frac{n}{10}}{-0.5}\right)$$

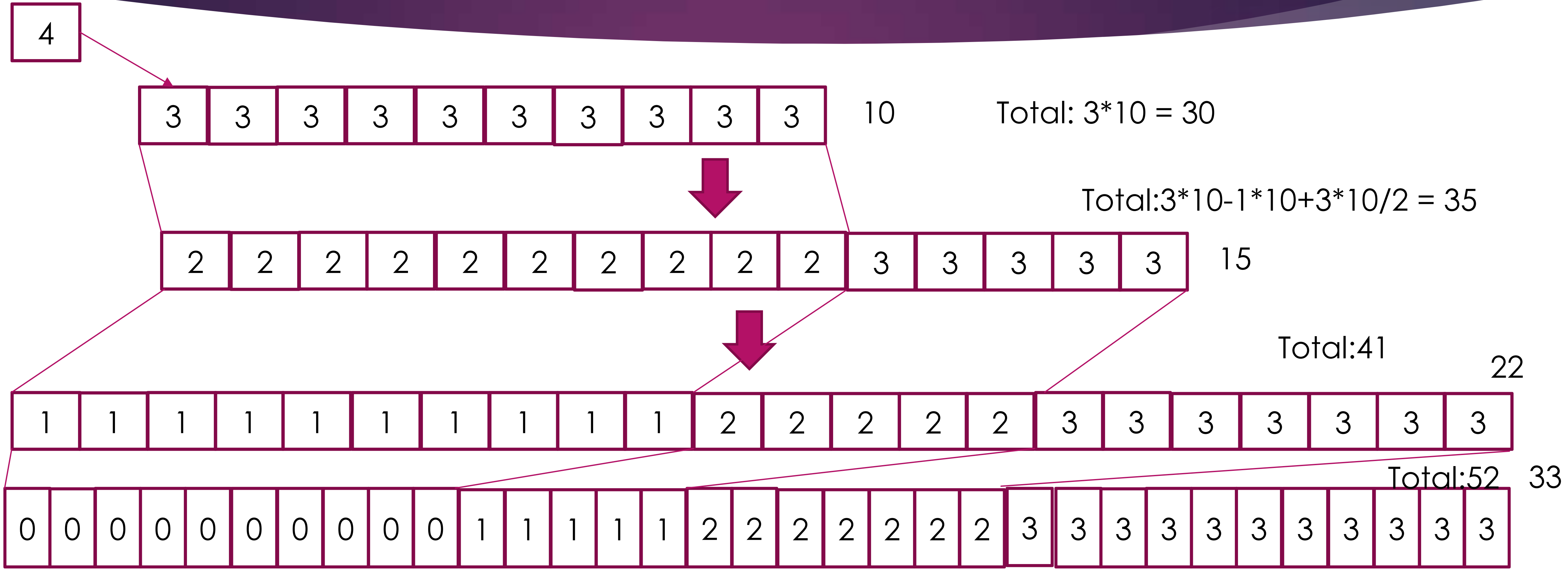
$$= n + 3n - 20 = 4n - 20$$

- $\frac{T(n)}{n} = O(1)$
- Pushing an element onto the dynamic array takes constant time when the worst case operation's cost amortize to other operations.

Accounting method

- The accounting method is a form of aggregate analysis which assigns to each operation an *amortized cost* which may differ from its actual cost. Early operations have an amortized cost higher than their actual cost, which accumulates a saved "credit" that pays for later operations having an amortized cost lower than their actual cost. Because the credit begins at zero, the actual cost of a sequence of operations equals the amortized cost minus the accumulated credit. Because the credit is required to be non-negative, the amortized cost is an upper bound on the actual cost. Usually, many short-running operations accumulate such a credit in small increments, while rare long-running operations decrease it drastically.(Kozen, Dexter (Spring 2011). "CS 3110 Lecture 20: Amortized Analysis". Cornell University. Retrieved 14 March 2015.)

Example：ArrayList(JAVA)



Potential method

- The potential method is a form of the accounting method where the saved credit is computed as a function (the "potential") of the state of the data structure. The amortized cost is the immediate cost plus the change in potential.(Kozen, Dexter (Spring 2011). "CS 3110 Lecture 20: Amortized Analysis". Cornell University. Retrieved 14 March 2015.)

function Φ

Φ : maps states of the data structure to non-negative numbers.

$$T_{\text{amortized}}(o) = T_{\text{actual}}(o) + C \cdot (\Phi(S_{\text{after}}) - \Phi(S_{\text{before}})),$$

where C is a non-negative constant of proportionality (in units of time) that must remain fixed throughout the analysis. constant C is usually omitted when using big O notation.

Define Φ

For any sequence of operations $O = o_1, o_2, \ldots, o_n$, define:

- The total amortized time: $T_{\text{amortized}}(O) = \sum_{i=0}^{n} T_{\text{amortized}}(o_i)$,
- The total actual time: $T_{\text{actual}}(O) = \sum_{i=0}^{n} T_{\text{actual}}(o_i)$.

Then:

$$T_{\text{amortized}}(O) = \sum_{i=1}^{n} (T_{\text{actual}}(o_i) + C \cdot (\Phi(S_i) - \Phi(S_{i-1}))) = T_{\text{actual}}(O) + C \cdot (\Phi(S_n) - \Phi(S_0)),$$

$$T_{\text{actual}}(O) = T_{\text{amortized}}(O) - C \cdot (\Phi(S_n) - \Phi(S_0)).$$

Since $\Phi(S_0) = 0$ and $\Phi(S_n) \geq 0$, $T_{\text{actual}}(O) \leq T_{\text{amortized}}(O)$

so the amortized time can be used to provide an accurate upper bound on the actual time of a sequence of operations, even though the amortized time for an individual operation may vary widely from its actual time.

Example：ArrayList(JAVA)

- Define Φ = 3(*n-1*) - 2N, *n means the number of the elements, N means the capacity.*
- S_0 *n =11 N=15(ArrayList is special, it has capacity 10 when there is an element. So, we do our analysis with N = 15 the Arraylist initialized with having11 elements and capacity 15)*
- *when n* = N*2/3+1 , Φ >= 0; when n = 11, N = 15, Φ = 0; when n=N, Φ should be >=n to pay for the enlarging the table; When insert a element, but not enlarge the table

$$\begin{aligned} T_{amortized}(o_i) &= T_{actural}(o_i) + \phi(S_i) - \phi(S_{i-1}) \\ &= 1 + (3 * (n_i - 1) - 2N_i) - (3 * (n_{i-1} - 1) - 2N_{i-1}) = 1 + 3 * n_i - 3 * n_{i-1} = 4 \end{aligned}$$

- When insert a element, and enlarge the table

$$\begin{aligned} T_{amortized}(o_i) &= T_{actural}(o_i) + \phi(S_i) - \phi(S_{i-1}) \\ &= i + (3 * (n_{i-1} + 1 - 1) - 2 * 1.5 * N_{i-1}) - (3 * (n_{i-1} - 1) - 2N_{i-1}) = i + (3 - N_{i-1}) = 4 \end{aligned}$$

$$\sum_{i=1}^{n} T_{amortized}(o_i) = 4n$$

- this shows that any sequence of *n* dynamic array operations takes O(*n*) actual time in the worst case.

Reference

- Kozen, Dexter (Spring 2011). "CS 3110 Lecture 20: Amortized Analysis". Cornell University. Retrieved 14 March 2015.