Amortized Analysis

October 16, 2013

Amortization is an analysis technique that can influence the design of algorithms in a profound way. Later in this course, we will encounter data structures that owe their very existence to the insight gained in performance due to amortized analysis.

Binary counting. We illustrate the idea of amortization by analyzing the cost of counting in binary. Think of an integer as a linear array of bits, \( n = \sum_{i \geq 0} A[i] \cdot 2^i \).

The following loop keeps incrementing the integer stored in \( A \).

\[
\text{loop } i = 0; \\
\quad \text{while } A[i] = 1 \text{ do } A[i] = 0; i++ \text{ endwhile; } \\
\quad A[i] = 1. \\
\text{forever.}
\]

We define the cost of counting as the total number of bit changes that are needed to increment the number one by one. What is the cost to count from 0 to \( n \)? Figure 1 shows that counting from 0 to 15 requires 26 bit changes. Since \( n \) takes only \( 1 + \log_2 n \) steps, this implies that the cost of counting from 0 to \( n \) is at most \( n \log_2 n + n \). Even though the upper bound of \( 1 + \log_2 n \) is tight for the worst single step, we can show that the total cost is much less than \( n \) times that. We do this with two slightly different amortization methods referred to as aggregation and accounting.

\[
\lfloor \log_2 n \rfloor \text{ bits or positions in } A, \text{ a single increment does at most } 1 + \log_2 n \text{ steps. This implies that the cost of counting from 0 to } n \text{ is at most } n \log_2 n + n. \text{ Even though the upper bound of } 1 + \log_2 n \text{ is tight for the worst single step, we can show that the total cost is much less than } n \text{ times that. We do this with two slightly different amortization methods referred to as aggregation and accounting.}
\]

\*

These notes are extracted from the lecture notes on “Design and Analysis of Algorithms”, Fall 2005 by Herbert Edelsbrunner.
**Aggregation.** The aggregation method takes a global view of the problem. The pattern in Figure 1 suggests we define $b_i$ equal to the number of 1s and $t_i$ equal to the number of trailing 1s in the binary notation of $i$. Every other number has no trailing 1, every other number of the remaining ones has one trailing 1, etc. Assuming $n = 2^k - 1$ we therefore have exactly $j - 1$ trailing 1s for $2^{k-j} = (n + 1)/2^j$ integers between 0 and $n - 1$. The total number of bit changes is therefore

$$\sum_{i=0}^{n-1} (t_i + 1) = (n + 1) \cdot \sum_{j=1}^{k} \frac{j}{2^j}.$$  

We use index transformation to show that the sum on the right is less than 2:

$$\sum_{j \geq 1} \frac{j}{2^j} = \sum_{j \geq 1} \frac{j - 1}{2^{j-1}}$$

$$= 2 \cdot \sum_{j \geq 1} \frac{j}{2^j} - \sum_{j \geq 1} \frac{1}{2^{j-1}}$$

$$= 2.$$  

Hence the cost is less than $2(n + 1)$. The *amortized cost* per operation is $\frac{T(n)}{n}$, which is about 2.

**Accounting.** The idea of the accounting method is to charge each operation what we think its amortized cost is. If the amortized cost exceeds the actual cost, then the surplus remains as a credit associated with the data structure. If the amortized cost is less than the actual cost, the accumulated credit is used to pay for the cost overflow. Define the amortized cost of a bit change $0 \rightarrow 1$ as $2$ and that of $1 \rightarrow 0$ as $0$. When we change 0 to 1 we pay $1$ for the actual expense and $1$ stays with the bit, which is now 1. This $1$ pays for the (later) cost of changing the 1 to 0. Each increment has amortized cost $2$, and together with the money in the system, this is enough to pay for all the bit changes. The cost is therefore at most $2n$.

We see how a little trick, like making the $0 \rightarrow 1$ changes pay for the $1 \rightarrow 0$ changes, leads to a very simple analysis that is even more accurate than the one obtained by aggregation.

**Potential functions.** We can further formalize the amortized analysis by using a potential function. The idea is similar to accounting, except there is no explicit credit saved anywhere. The accumulated credit is an expression of the well-being or potential of the data structure. Let $c_i$ be the actual cost of the $i$-th operation and $D_i$ the data structure after the $i$-th operation. Let $\Phi_i = \Phi(D_i)$ be the potential of $D_i$, which is some numerical value depending on the concrete application. Then we define $a_i = c_i + \Phi_i - \Phi_{i-1}$ as the *amortized cost* of the $i$-th operation. The sum of amortized
costs of $n$ operations is

$$\sum_{i=1}^{n} a_i = \sum_{i=1}^{n} (c_i + \Phi_i - \Phi_{i-1}) = \sum_{i=1}^{n} c_i + \Phi_n - \Phi_0.$$ 

We aim at choosing the potential such that $\Phi_0 = 0$ and $\Phi_n \geq 0$ because then we get $\sum a_i \geq \sum c_i$. In words, the sum of amortized costs covers the sum of actual costs. To apply the method to binary counting we define the potential equal to the number of 1s in the binary notation, $\Phi_i = b_i$. It follows that

$$\Phi_i - \Phi_{i-1} = b_i - b_{i-1} = (b_{i-1} - t_{i-1} + 1) - b_{i-1} = 1 - t_{i-1}.$$ 

The actual cost of the $i$-th operation is $c_i = 1 + t_{i-1}$, and the amortized cost is $a_i = c_i + \Phi_i - \Phi_{i-1} = 2$. We have $\Phi_0 = 0$ and $\Phi_n \geq 0$ as desired, and therefore $\sum c_i \leq \sum a_i = 2n$, which is consistent with the analysis of binary counting with the aggregation and the accounting methods.