Analysis of Algorithms I: Amortized Analysis

Xi Chen

Columbia University

Amortized analysis is a set of techniques (Aggregate analysis, the Accounting method, and the Potential method) for proving upper bounds for the running time of an algorithm, usually involving a sequence of data-structure operations. Here is what we usually do: Determine the worst-case running time of any data-structure operation in the sequence, say t; If the sequence contains n operations, then nt is an upper bound for the total running time. This analysis is technically correct (and in many cases gives us a tight upper bound for the total running time, as we have seen in many examples so far), but sometimes is overly pessimistic (as we will see in the following example).

Consider a stack S (right now no need to worry about overflow but we will come back to this issue later) that supports not only Push and Pop but the following Multipop operation as well:

Multipop(S, k):

1. while S is not empty and $k > 0$ do
2. Pop(S)
3. $k = k - 1$

It is clear that Pop is the special case when $k = 1$. We let S.num denote the number of elements in S. At the beginning, S.num $= 0$. For simplicity, we assume a Push operation costs one unit of time and a Multipop costs ℓ units of time if it makes ℓ calls to Pop.

The question is then: Starting with an empty stack, what would be a good upper bound for the worst-case total cost (or running time) of a sequence of n Push and Multipop operations? Here is a rather loose analysis: Each Push operation costs 1; Each Multipop operation costs $\leq n$ because there can never be more than n elements in the stack; Therefore, each operation in the sequence costs $\leq n$ and the worst-case total running time is $\leq n^2$. However, a better analysis reveals that the total running time is indeed $\leq 2n$.

The reason why n^2 is a very loose upper bound: While it is possible that some of the Multipop operations in the sequence may cost a lot by making a lot of calls to Pop, not all of them can. In particular, for a Multipop operation to take $\Omega(n)$ time, there must be $\Omega(n)$ Push operations before it to build up the stack with $\Omega(n)$ many elements.

Proof of the $2n$ upper bound: The total cost of all the Push operations is clearly $\leq n$. For Multipop, notice that the cost of each Multipop operations is equal to

$$\text{the number of calls made to Pop}$$

So the total cost of all Multipop operations in the sequences is equal to the total number of calls to Pop. But this can be no more than n because we only call Pop when the stack is non-empty:

$$\text{number of calls to Pop} \leq \text{number of Push in the sequence} \leq n$$

It then follows that the total cost of all Multipop operations in the sequence is $\leq n$ as well. The upper bound of $n + n = 2n$ follows.

The method we just used is called the Aggregate analysis in the textbook, where we try to analyze the total cost (or running time) directly. We can also use the Accounting method: Assume each unit of running time (e.g., a Push or a Pop operation) costs one dollar. For each operation, we associate it with two costs:

1. Actual cost: the running time of the operation. For example, the actual cost of a Push operation is 1 and the actual cost of a Multipop operation is the number of calls to Pop.
2. Amortized cost: the money we actually pay for this operation.

The tricky part: The amortized cost of an operation may not be the same as its actual cost. To apply this method, the most difficult part is to design an appropriate pricing scheme: How do we set the amortized cost of each opertion? If an operation's amortized cost exceeds its actual cost, we use the former to pay the actual cost first and store the difference in a bank account. (At the beginning, the balance is 0.) If an operation's amortized cost is less than its actual cost, we can draw from the bank account to cover the difference. We claim that if the bank account by the end (after all n operations) is nonnegative, then the total amortized cost is an upper bound for the total actual cost (or equivalently, the total running time).

More formally, given any sequence of n operations, we use c_i to denote the actual cost of the ith operation and use d_i to denote the amortized cost (charged according to a pricing scheme to be specified later). After k operations, the bank account balance is

$$\sum_{i=1}^{k} d_i - \sum_{i=1}^{k} c_i$$

If by the end the bank account balance is nonnegative, we have

$$\sum_{i=1}^{n} d_i \geq \sum_{i=1}^{n} c_i$$

so the total amortized cost is an upper bound for the total actual cost. The tricky part: How to set the amortized cost of each op?

For the stack problem, we set the amortized cost as follows:

1. If the ith operation is Push, $d_i = 2$;
2. If the ith operation is Multipop, $d_i = 0$.

Intuition behind this scheme: A Multipop may have a very high actual cost. But we do not charge it on Multipop directly. Instead, we charge (or blame) it to each Push operation because it is these Push operations that build up a large stack. So for each Push operation we charge 2 dollars: 1 for its actual cost and 1 for the cost of a future Pop (called by Multipop). It may remind you of the deposit money we pay for each bottled water for the recycling cost in the future.

To see this pricing scheme works, check the following two things. First, what is the total amortized cost? This is easy:

$$\sum_{i=1}^{n} d_i \leq 2n$$

Second, is the balance nonnegative by the end? If so we know that

$$\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} d_i \leq 2n$$

which gives us an upper bound of $2n$ for the total actual cost. Running an example would suggest the following lemma:

Lemma

At any time, if there are $k \geq 0$ elements in the stack, then our bank account balance is exactly k.

From it we know that the balance never goes to negative. Great! This implies that $2n$, the total amortized cost, is an upper bound for the total actual cost (or equivalently, the total running time). In the next slide, we prove this lemma.

We use induction. The basis is trivial because at the beginning the balance is 0 and the stack is empty. Assume by the end of the ith operation, the stack has k elements and the balance is k. We show that the statement holds after the $(i+1)$th operation. Two cases:

1. If it is a Push operation: The actual cost is 1 but the amortized cost is 2, so we store 1 dollar into the account and the new balance is $k+1$. But the number of elements in the stack also increases by 1 so they match.
2. If it is Multipop and makes $\ell \leq k$ calls to Pop: The actual cost is ℓ but the amortized cost is 0, so we need to draw ℓ dollars from the account. The new balance is $k-\ell$ but there are now only $k-\ell$ elements in the stack. Again they match.

Next example, Dynamic Stack (Dynamic Table in the textbook, same thing): Suppose we need a standard stack to handle a sequence of Push and Pop (each costs 1). For now assume we need to deal with a sequence of Push operations (no Pop). However, we do not know the length of the sequence in advance and thus, we do not know how much space we should allocate for the stack. (Given a stack S, we will use S.size to denote the space we allocate to it so S can store S.size many elements at most. We continue to use S.num to denote the current number of elements in S.)

We use the following doubling strategy: Start by allocating no space to S. When the first Push (S, x) operation arrives, we create a stack with size 1 and push the element x into the new stack. After this, we deal with each Push operation as follows:

1. If $S.\text{num} < S.\text{size}$ then
2. Push (S, x)
3. Else (S overflows because it is full: $S.\text{num} = S.\text{size}$)
4. Create a new stack of size twice of the current size
5. Copy all elements of the old stack into the new stack
6. Also push the new element into the new stack
7. Free up the space allocated to the old stack

For simplicity, the cost a Push operation is 1 if the stack is not full (so no stack expansion occurs); and the cost of a Push operation, when an expansion occurs, is:

$$\text{size of the old stack} + 1$$

This accounts for the number of operations needed to move elements from the old stack to the new one as well as pushing the new element into the new stack. For example, the cost of the first 7 Push operations are: $1, 2, 3, 1, 5, 1, 1, \ldots$

This strategy is clearly space-efficient: At any time, the space allocated is at most twice of the number of elements in the stack. But is it time/cost-efficient? What is the total cost of a sequence of n Push operations? A very loose upper bound: Every Push in the sequence costs $\leq n$ (as $S.\text{num} \leq n-1$ before any Push in the sequence) so the total cost is $\leq n^2$. Again, it is loose because not all Push operations in the sequence can cause stack expansions. Most of them cannot and only cost 1 each. A tight upper bound is indeed $3n$.

Aggregate analysis: Because we only consider a sequence of n Push here, we actually know exactly the cost of the ith Push:

$$c_i = \begin{cases} i & \text{if } i-1 \text{ is a power of 2} \\ 1 & \text{otherwise} \end{cases}$$

It comes from the observation that the size of the stack is always a power of 2 (after the first Push). Now assume at this moment the stack has size 2^k, when will the next expansion (to 2^{k+1}) happen? It happens when the stack is full (overflow) and a new Push operation comes in. As a result, this must be the $(2^k + 1)$th Push in the sequence, and the cost of this Push operation is $2^k + 1$. The formula above then follows.

Let k be the unique integer such that $n \geq 2^k + 1$ but $n < 2^{k+1} + 1$. Using the formula from the last slide, we have

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

Finally we use this problem to demonstrate the potential method. This is the most powerful method, and is somewhat similar to the accounting method but is more delicate and can deal with more difficult problems when used appropriately. The most tricky part is to design an appropriate potential function Φ, a function that maps a stack (or a data structure being considered) to a real number.

The reason why Φ is called a potential function is due to its similarity with the conversion between potential and kinetic energy in physics. The potential method has the following four steps:

1. Design a function Φ that maps a stack to a real number.
2. Given any sequence of n operations (for now any sequence of n Push operations, will allow Pop later), let S_0 denote the initial stack with $S_0.\text{num} = S_0.\text{size} = 0$ and let S_i denote the stack after the first i operations. Show that $\Phi(S_n) \geq \Phi(S_0)$.
3. Use Φ to derive the amortized cost of the ith operation:

$$d_i = c_i + \Phi(S_i) - \Phi(S_{i-1})$$

4. Conclude that $\sum_{i=1}^{n} d_i$ is an upper bound for $\sum_{i=1}^{n} c_i$.

To prove the conclusion of Step 4, we have

$$d_1 = c_1 + \Phi(S_1) - \Phi(S_0)$$
$$d_2 = c_2 + \Phi(S_2) - \Phi(S_1)$$
$$\ldots$$
$$d_n = c_n + \Phi(S_n) - \Phi(S_{n-1})$$

Summing up all these n equations gives us

$$\sum_{i=1}^{n} d_i = \sum_{i=1}^{n} c_i + \Phi(S_n) - \Phi(S_0) \geq \sum_{i=1}^{n} c_i$$

where the last inequality uses $\Phi(S_n) \geq \Phi(S_0)$ from Step 2.

Now we use the following function Φ to give an upper bound for a Dynamic Stack, given a sequence of n Push operations:

$$\Phi(S) = 2 \cdot S.\text{num} - S.\text{size}$$

It is easy to check that $\Phi(S_0) = 0$ because $S_0.\text{num} = S_0.\text{size} = 0$. We also have $\Phi(S_i) \geq 0$ for any $i = 1, 2, \ldots, n$. This is simply because the doubling strategy guarantees that at any time, the number of elements in S is at least half of its size (space-efficient). This finishes Step 2 because $\Phi(S_n) \geq 0 = \Phi(S_0)$.

In Step 3, we show that $d_i \leq 3$ for any $i = 1, 2, \ldots, n$. For $i = 1$, we have $d_1 = 1 + \Phi(S_1) - \Phi(S_0) = 1 + 1 - 0 = 2$. For any $i > 1$, we consider the following two cases: Case 1: The ith Push operation does not trigger an expansion (so $c_i = 1$). Use S and S' to denote the stack before and after the operation, then we have

$$S'.\text{num} = S.\text{num} + 1 \quad \text{and} \quad S'.\text{size} = S.\text{size}$$

As a result, we have

$$d_i = 1 + (2 \cdot S'.\text{num} - S'.\text{size}) - (2 \cdot S.\text{num} - S.\text{size}) = 3$$

Case 2: The ith Push operation does trigger an expansion. Let S and S' denote the stack before and after the operation, then $S.\text{num} = S.\text{size}$, $c_i = S.\text{size} + 1$ and

$$S'.\text{num} = S.\text{num} + 1 \quad \text{and} \quad S'.\text{size} = 2 \cdot S.\text{size}$$

As a result, we have

$$d_i = c_i + (2 \cdot S'.\text{num} - S'.\text{size}) - (2 \cdot S.\text{num} - S.\text{size}) = 3$$

Therefore, we conclude in Step 4 that the total amortized cost $\sum_{i=1} d_i \leq 3n$ is an upper bound for the total actual cost $\sum_{i=1}^{n} c_i$.

Read Section 17.4, in which we use the following contraction strategy to support dynamic Pop: Whenever a Pop operation causes the table to become less than 1/4 full, we create a new stack of half the size of the current stack; copy all elements from the current stack into the new one; and free up space allocated for the current stack. Space-efficient: For any sequence of Push and Pop operations, the number of elements in the stack is always at least 1/4 of the stack size.

Using a carefully designed potential function (See equation 17.6), one can show that the amortized cost of any operation (no matter it is Push or Pop) in the sequence is bounded above by a constant. Thus, the total running time remains $O(n)$. This is an example that demonstrates the power of the potential method (mission impossible for the Aggregate analysis).