0.3.1 Amortized analysis and potential functions

You should have seen dynamic arrays in some predecessor course: These store items exactly as in an array, but also keep track of the capacity of the array, and double this capacity whenever the array is full. One can easily argue that on average, the time to do an insertion in a dynamic array is constant. More precisely, occasionally we must copy the entire array to a new (bigger) array, which takes $\Theta(n)$ time. But every time this happens, we previously must have had $n/2$ insertions that were fast, i.e., took $\Theta(1)$ time. Summing up, therefore we are using $\Theta(n)$ time for $\Theta(n)$ operations, and so on average they take $\Theta(1)$ time.

This is a prime example of *amortized analysis*, where we analyze (in some sense) the average over multiple operations, rather than the worst that can happen for each operation. For some situations (such as dynamic arrays) a simple argument suffices to obtain an amortized bound: sum up how many cheap and expensive operations we had and divide by the number of operations. But for other examples more powerful methods are needed.

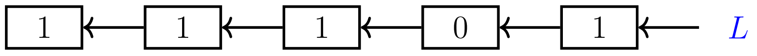
Let us first define formally what we mean by amortized run-time. We assume that some set of operations that are permitted on our data has been fixed.

Definition 1 *Let $T^{\text{actual}}(\mathcal{O})$ be the run-time of operation $\mathcal{O}$. Let $T^{\text{amort}}(\cdot)$ be a function on the operations. We say that $T^{\text{amort}}(\cdot)$* is an amortized run-time bound *if for any sequence $\mathcal{O}_1, \ldots, \mathcal{O}_k$ of operations that could occur we have*

$$\sum_{i=1}^{k} T^{\text{actual}}(\mathcal{O}_i) \leq \sum_{i=1}^{k} T^{\text{amort}}(\mathcal{O}_i). \tag{1}$$

We usually just say "$\mathcal{O}$ has amortized run-time $T^{\text{amort}}(\mathcal{O})$", even though this is a bit imprecise: In our definition, we are not computing one particular bound (depending on the operation), but only an upper bound, and it may be much too big. For example, setting $T^{\text{amort}}(\mathcal{O}) := T^{\text{actual}}(\mathcal{O})$ would give an amortized run-time bound, but not a particularly useful one. The point is to choose a function $T^{\text{amort}}(\cdot)$ that is *small*: it should be (asymptotically) no more than the actual run-time, and at least for some of the operations it should be less. (So loosely speaking, the amortized run-time of an operation is the minimum possible amortized run-time bound, except that it is not clear what 'minimum' means if there are multiple operations.)

Example: Increasing a binary counter To understand how to obtain an amortized run-time bound, we will do a toy-example where we repeatedly increase a number that has been stored as its binary encoding. Namely, assume that we have a list L that stores bits, i.e., each entry is either 0 or 1. (In our figures we show the list as drawn from right to left, since this will make the correspondence to a binary counter clearer.)

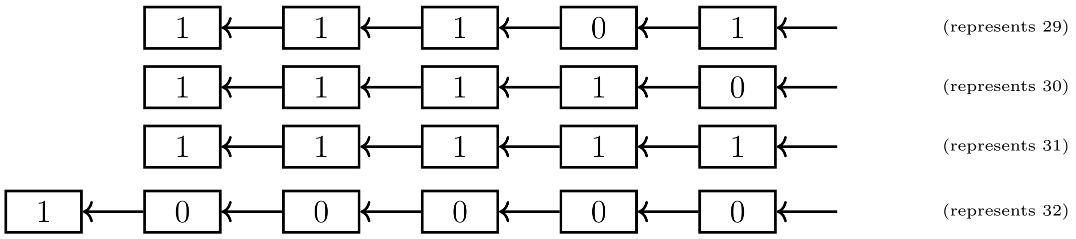


Notice how we can interpret the contents of this list as a number encoded in binary; the above example encodes $29 = (11101)_2$. We also assume that L initially represents number

0, i.e., it consists of a single list-item that stores 0. We only permit one operation, which is to increase the represented number. See Algorithm 1 for the pseudo-code and Figure 1 for a few steps on the above example.

Algorithm 1: *increment*()

```
1 curr ← $L$.head()
2 while (curr.value() equals 1) do
3     set curr.value() to be 0
4     if (curr.next() equals NIL) then append new item with value 0
5     curr← curr.next()
6 set curr.value to be 1
```



1 1 1 0 1 (represents 29)
1 1 1 1 0 (represents 30)
1 1 1 1 1 (represents 31)
1 0 0 0 0 0 (represents 32)

Figure 1: Executing *increment* three times.

Observe that the worst-case run-time of *increment* is $\Theta(|L|)$. But this bound is usually much too large. A more precise bound would be $\Theta(\#\{\text{bits that were changed}\})$. Very frequently (in fact, half of the time) the rightmost bit (i.e., the first entry of the list) will be 0; in this case only one bit gets changed and the run-time of *increment* is $O(1)$. This kind of situation ('most of the time the operation is cheap, but occasionally it will be expensive') is exactly the setup in which you should try to do an amortized analysis.

Every amortized analysis should start with the definition of *time units*. This is necessary because to check Inequality 1 we need to do arithmetic with run-times, and one should not do arithmetic when handling asymptotic notations.[11] Tyipcally we define a *time unit* to be the maximum of all constants hidden in the big-O notations. Thus for binary counters, we choose time units such that *increment* takes at most $\#\{\text{bit-changes}\}$ time units.

To find a function $T^{\text{amort}}(\cdot)$ that satisfies Inequality 1, there are three different methods:

- Aggregation method: Compute an upper bound for $\sum_{\mathcal{O}} T^{\text{actual}}(\mathcal{O})$.

  For binary counters, $\sum_{\mathcal{O}} T^{\text{actual}}(\mathcal{O})$ is at most the overall number of bit-changes. Observe that the rightmost bit is changed *every* time. But the next bit is changed only every second time (so $\lfloor n/2 \rfloor$ times in total, assume that we called *increment* n times),

[11] Plenty of course notes and even textbooks *do* intermix asymptotic notations with arithmetic. This is correct (though bad style) if the arithmetic works out when replacing all '$O(\dots)$' with '$c \cdot \dots$'. But it is dangerous to do so, and students are advised to either explicitly replace every O with a '$c\cdot$', or define time units to make O disappear as we will do here.

the next bit is changed only every fourth time, and so on. Therefore the total number of bit-changes is

$$n + \lfloor n/2 \rfloor + \lfloor n/4 \rfloor + \cdots \le n(1 + \tfrac{1}{2} + \tfrac{1}{4} + \dots) = 2n.$$

Thus if we set $T^{\text{amort}}(\textit{increment}) = 2$ then Inequality 1 holds.

Unfortunately the aggregation method works only in few situations (in particular, it tends to be nearly impossible to use if there are multiple permitted operations).

- Accounting method: Let operations also deposit/withdraw time units.

  In this method, each operation can 'leave behind' some time units for use by future operations, or vice versa, use some time units that were left behind earlier. (This is why it is also called 'money in the bank' method.)

  For the example of binary counters, we will demand that operations deposit time units such that any bit that is currently 1 has one time unit at it. (This clearly holds initially.) Consider what happens during *increment*: As we scan the list, we need one time unit for each bit b that was 1. But we know that currently there is a time unit left behind at b; use it to pay for changing b. (This maintains the invariant because b is now 0.) Finally we reach (or create) a bit b' that is 0. At this point we need one time unit to change b' to 1, and one more time unit because to keep our invariant we must deposit one time unit at b'. Overall, therefore *increment* needs only two time units, or in otherwords, $T^{\text{amort}}(\textit{increment}) = 2$ is sufficient.

- Potential function method: Measure how the structure changes.

  Rather than explicitly saying where time units are stored (as done in the accounting method), the potential function method expresses this via a function. This makes the analysis much cleaner and simpler (presuming that you found the right function), and is by far the safest choice for amortized analysis if you have many different sequences of operations. But it is also the hardest method to understand initially.

Definition 2 *A* potential function *is a function $\Phi(\cdot)$ that maps the current status of the data onto a number such that for any feasible sequence $\mathcal{O}_1, \dots, \mathcal{O}_k$ of operations*

- $\Phi(i) \ge 0$ *for* $1 \le i \le k$, *where* $\Phi(i)$ *is a shortcut for '*$\Phi(\cdot)$ *applied after operations* $\mathcal{O}_1, \dots, \mathcal{O}_i$ *have been executed',*
- $\Phi(0) = 0$.

The above definition is very unspecific; there are many ways of defining a potential function. For our binary counter example, we could use

$$\Phi(i) = \underbrace{\#\{\text{bits that are currently } 1\}}_{p};$$

this is clearly non-negative and $\Phi(0) = 0$, so this satisfies the conditions of a potential function. You may notice that this potential function is exactly the same as the number of time units that were 'deposited' in the accounting method; this is not a coincidence.

The potential function changes with each applied operations; sometimes the notation Φ_{before} and Φ_{after} for the status before/after applying an operation will be helpful. (We also use it for all other variables that change, e.g. p_{before} and p_{after} for the number of bits that are 1 before/after the operation.)

Lemma 1 *For any potential function $\Phi(\cdot)$, the function*

$$T^{\text{amort}}(\mathcal{O}) := \max_{\textit{applications of } \mathcal{O}} \left\{ \underbrace{T^{\text{actual}}(\mathcal{O})}_{\textit{actual run-time}} + \underbrace{\Phi_{\text{after}} - \Phi_{\text{before}}}_{\textit{potential-difference}} \right\}$$

upper-bounds the amortized run-time.

Proof: Fix an arbitrary sequence of operations $\mathcal{O}_1, \dots, \mathcal{O}_k$. Summing up the amortized times and using a telescoping sum we see that Inequality 1 holds:

$$\begin{aligned}
\sum_{i=1}^{k} T^{\text{amort}}(\mathcal{O}_i) &\geq \sum_{i=1}^{k} \Big(T^{\text{actual}}(\mathcal{O}_i) + \Phi(i) - \Phi(i-1) \Big) \\
&= \sum_{i=1}^{k} T^{\text{actual}}(\mathcal{O}_i) + \sum_{i=1}^{k} \Big(\Phi(i) - \Phi(i-1) \Big) \\
&= \sum_{i=1}^{k} T^{\text{actual}}(\mathcal{O}_i) + \underbrace{\Phi(k)}_{\geq 0} - \underbrace{\Phi(0)}_{=0} \geq \sum_{i=1}^{k} T^{\text{actual}}(\mathcal{O}_i)
\end{aligned}$$

□

So to get an amortized time-bound, given a potential function, we compute $T^{\text{actual}}(\mathcal{O}) + \Phi_{\text{after}} - \Phi_{\text{before}}$, and find an upper bound for it that holds no matter what the situation was when we applied $\mathcal{O}$. For our binary counter example, we know that the actual run-time (measured in time units) was at most the number ℓ of bit-changes, i.e., we changed the rightmost ℓ bits from $01^{\ell-1}$ to $10^{\ell-1}$. With this we know *exactly* how the number of ones in the bitstring changes: $p_{\text{after}} = p_{\text{before}} - (\ell - 1) + 1$. Therefore

$$\begin{aligned}
T^{\text{actual}}(\textit{increment}) + \Phi_{\text{after}} - \Phi_{\text{before}} &\leq \ell + p_{\text{after}} - p_{\text{before}} \\
&= \ell + \big(p_{\text{before}} - (\ell - 1) + 1 \big) - p_{\text{before}} = 2.
\end{aligned}$$

So the amortized time for *increment* is at most 2 time units, hence in $O(1)$.