Amortized Complexity – Main Idea

Worst case analysis of run time complexity is often too pessimistic.

Average case analysis may be difficult because (i) it is not clear what is "average data", (ii) uniformly random data is usually not average, (iii) probabilistic arguments can be difficult.

Amortized complexity analysis is a different way to estimate run times. The main ideas is to spread out the cost of operations, charging more than necessary when they are cheap — thereby "saving up for later use" when they occasionally become expensive.

The Accounting Method

We choose a set (typically a singleton, but always a small set) of *elementary operations* whose cost can be set to a constant (typically 1, or other small constants).

In running the algorithm some steps make use of just the elementary operations, others use them in combination — *aggregate operations*.

We invent an *accounting* or *potential function* $\phi : \mathcal{S} \rightarrow N$ where $\mathcal{S}$ is the state space of the (data) structure and N is the set of natural numbers. $\phi(T)$ denotes the amount of *(cyber) dollars* "deposited" or "saved up" in state T.

Costs of Operations

Performing operations cost dollars.

Suppose an operation F maps a state T_1 to a state T_2. Let its cost be $Cost_F(T_1, T_2)$ dollars. If this is "cheap" we may decide to actually *charge it more than the cost* in order to "save up" for more expensive operations later. If so, we "deposit" the excess in state T_2.

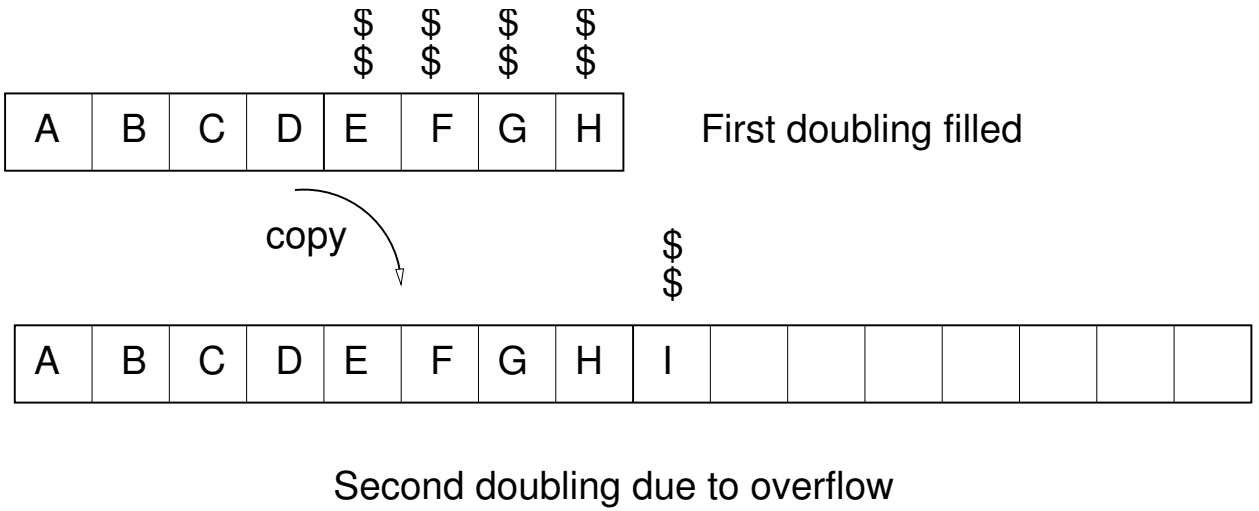
The dollar amounts in T_1 and T_2, i.e, respectively $\phi(T_1)$ and $\phi(T_2)$ may be different. If $\phi(T_1) > \phi(T_2)$ then $\phi(T_1) - \phi(T_2)$ dollars is released to help meet the cost $Cost_F(T_1, T_2)$. If $\phi(T_1) < \phi(T_2)$ there is a shortfall, then we have to *pay* for the operation from "external" funds. Summing payments over many operations is the measure of complexity.

Example: Extendable Arrays — 6.1.5

Think of a queue implemented as an array. We have an array A of size N. If we fill it up and need more space we create an array B of size $2 * N$. Then we copy the elements of A into the first half of B. Copying N elements costs N dollars.

If we start with an array of size 1, how do we analyse the performance of pushes?

An accounting function ϕ is updated as follows. Let T_1 be the state of a newly doubled array, only the first half of which is filled. T_2 is the next state in which the overflow element is inserted into the first position of the second half. We charge 3 dollars for this insertion, paying 1 dollar for the cost of doing it, and save 2 dollars. Formally, $\phi(T_2) = \phi(T_1) + 2$. This is done for every push into the new second half.



$ –– amount "deposited" into account

Operations:
Push –– charge $3 per push. Cost of push = $1.
Excess $2 is deposited into account.
Double array –– charge $N for copying N elements
into first half of doubled array. Use account to
to pay for this –– we prove there is always enough.

Figure 1: Account updating for push ops

Amortized complexity of N pushes is $\mathcal{O}(N)$

Overflow occurs when an array A has reached size 2^i for $i \geq 0$.

To copy its elements into the first half of a new array B of size 2^{i+1} costs 2^i dollars. But from the previous round of insertions into the second half of array A, we have deposited 2 dollars into the account function for each such insertion. So the total amount deposited is $2 \times 2^{i-1} = 2^i$ dollars. Hence we have enough to pay for the copying.

Therefore, it suffices to charge exactly 3 dollars for each insertion, i.e. this is an $\mathcal{O}(1)$ operation. Hence for N insertions the amortized complexity is $\mathcal{O}(N)$.

Comments on the Textbook Explanations I

The text slides on amortized complexity for splaying are very high level outlines. Here are a few comments on the details in the textbook explanation.

The account for a state (tree) is distributed across subaccounts for nodes, the amount of dollars deposited in each node is equal to its rank.

In splaying a node x to a new position denoted by x' you will notice that some nodes have their account increased and others have theirs decreased, but most are not changed.

Comments on the Textbook Explanations II

So, if there is a net positive change, we the affected nodes can "donate" dollars to pay for the zig-zig or zig-zag (2 dollars each), or a zig (1 dollar). If negative, then *we* pay the difference.

The *invariant* mentioned means this: each node before and after splaying can *maintain* an account (may be less or more than its old account) according to its new rank. But this is only possible because *we* pay to make up any shortfall.

The text goes on to show that this payment is bounded by $O(log\ n)$ dollars for an arbitrary splay.