Amortized $\mathcal{O}(1)$ Complexity

Andreas Weis

CppCon 2024

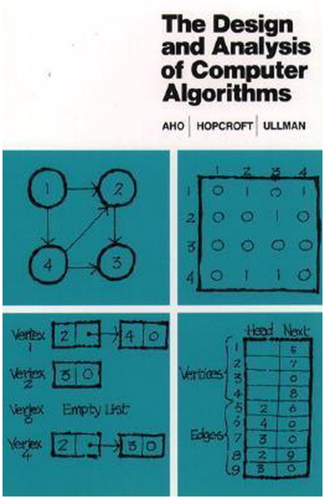
Runtime Complexity

- $f \in \mathcal{O}(g) \iff \exists\ C > 0.\ \exists\ x_0 > 0.\ \forall\ x > x_0 : |f(x)| \leq C \cdot |g(x)|$

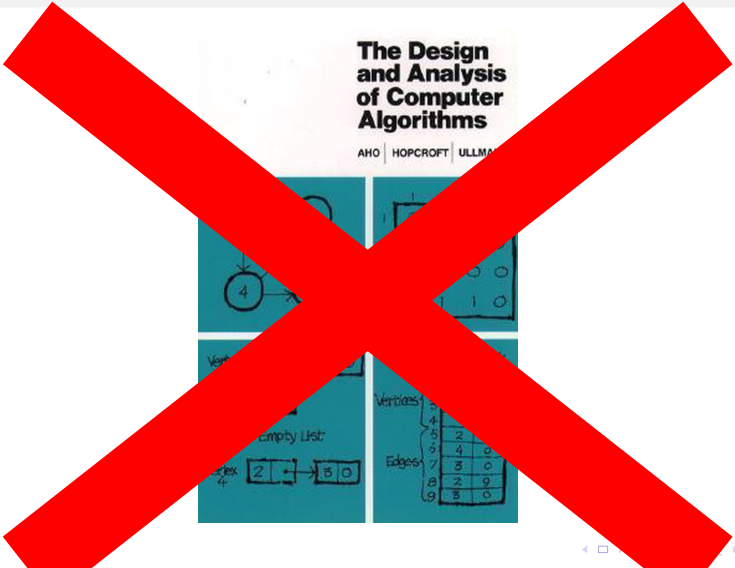
Child's play!

What is amortized complexity?

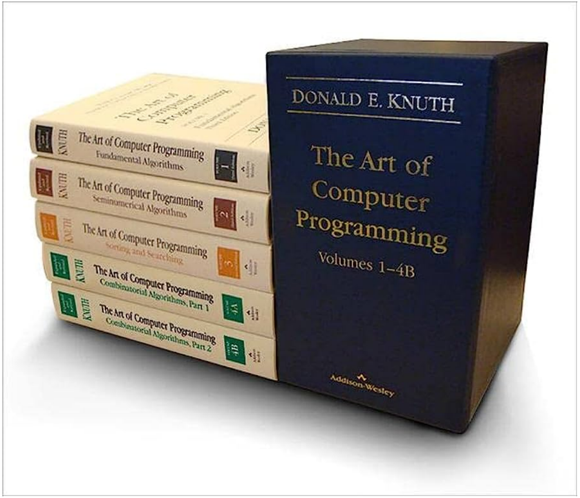
What is amortized complexity?



What is amortized complexity?



What is amortized complexity?



What is amortized complexity?

What is amortized complexity?

SIAM J. ALG. DISC. METH.
Vol. 6, No. 2, April 1985

AMORTIZED COMPUTATIONAL COMPLEXITY*

ROBERT ENDRE TARJAN†

Abstract. A powerful technique in the complexity analysis of data structures is *amortization*, or averaging over time. Amortized running time is a realistic but robust complexity measure for which we can obtain surprisingly tight upper and lower bounds on a variety of algorithms. By following the principle of designing algorithms whose amortized complexity is low, we obtain "self-adjusting" data structures that are simple, flexible and efficient. This paper surveys recent work by several researchers on amortized complexity.

ASM(MOS) subject classifications. 68C25, 68E05

1. Introduction. Webster's [34] defines "amortize" as "to put money aside at intervals, as in a sinking fund, for gradual payment of (a debt, etc.)." We shall adapt this term to computational complexity, meaning by it "to average over time" or, more precisely, "to average the running times of operations in a sequence over the sequence." The following observation motivates our study of amortization: In many uses of data structures, a sequence of operations, rather than just a single operation, is performed, and we are interested in the total time of the sequence, rather than in the times of the individual operations. A worst-case analysis, in which we sum the worst-case times of the individual operations, may be unduly pessimistic, because it ignores correlated effects of the operations on the data structure. On the other hand, an average-case analysis may be inaccurate, since the probabilistic assumptions needed to carry out the analysis may be false. In such a situation, an amortized analysis, in which we average the running time per operation over a (worst-case) sequence of operations, can yield an answer that is both realistic and robust.

To make the idea of amortization and the motivation behind it more concrete, let us consider a very simple example. Consider the manipulation of a stack by a sequence of operations composed of two kinds of unit-time primitives: *push*, which adds a new item to the top of the stack, and *pop*, which removes and returns the top item on the stack. We wish to analyze the running time of a sequence of operations, each composed of zero or more pops followed by a push. Assume we start with an empty stack and carry out m such operations. A single operation in the sequence can take up to m time units, as happens if each of the first $m-1$ operations performs no pops and the last operation performs $m-1$ pops. However, altogether the m operations can perform at most $2m$ pushes and pops, since there are only m pushes altogether and each pop must correspond to an earlier push.

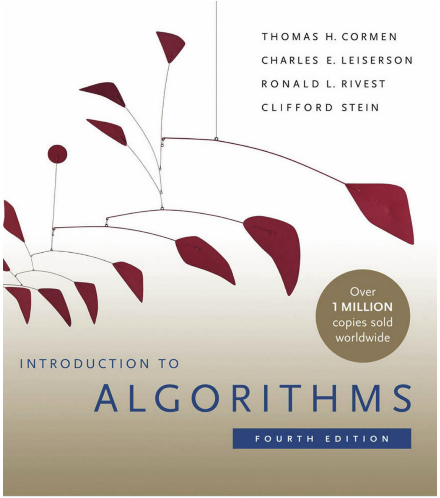
This example may seem too simple to be useful, but such stack manipulation indeed occurs in applications as diverse as planarity-testing [14] and related problems [24] and linear-time string matching [18]. In this paper we shall survey a number of settings in which amortization is useful. Not only does amortized running time provide a more exact way to measure the running time of known algorithms, but it suggests that there may be new algorithms efficient in an amortized rather than a worst-case sense. As we shall see, such algorithms do exist, and they are simpler, more efficient, and more flexible than their worst-case cousins.

---

* Received by the editors December 29, 1983. This work was presented at the SIAM Second Conference on the Applications of Discrete Mathematics held at Massachusetts Institute of Technology, Cambridge, Massachusetts, June 27–29, 1983.

† Bell Laboratories, Murray Hill, New Jersey 07974.

What is amortized complexity?



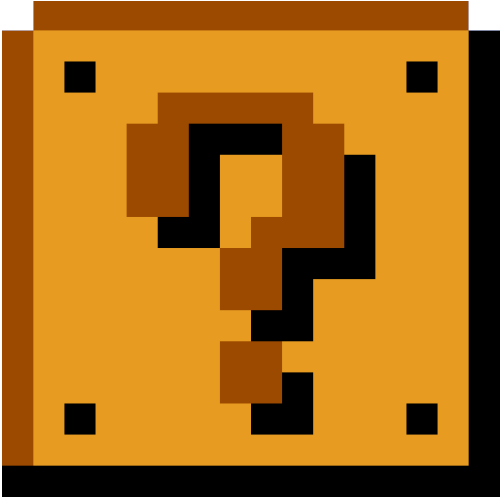
Amortized Analysis

- Aggregate analysis
- Accounting method
- Potential method

Amortized Analysis

- Aggregate analysis
- Accounting method
- Potential method

Accounting



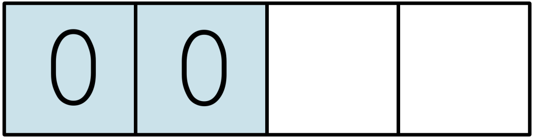
Accounting

Setting up a vector

```
std::vector<int> v(2, 0);
v.reserve(4);
```

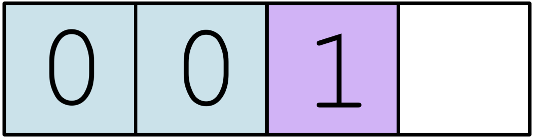
Setting up a vector

```
std::vector<int> v(2, 0);
v.reserve(4);
```



Cost of vector::push_back

```
v.push_back(1);
```



Cost of `vector::push_back`


```
v.push_back(2);
```

0	0	1	2

Cost of `vector::push_back`

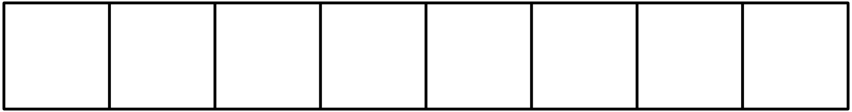
```
v.push_back(3);
```

0	0	1	2

No room!

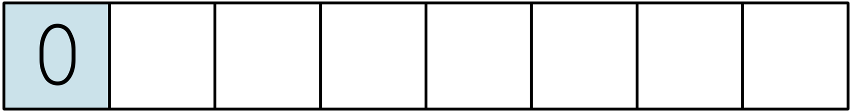
Cost of `vector::push_back`

```
v.push_back(3);
```

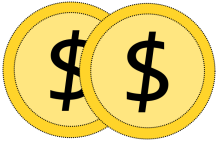


Cost of `vector::push_back`

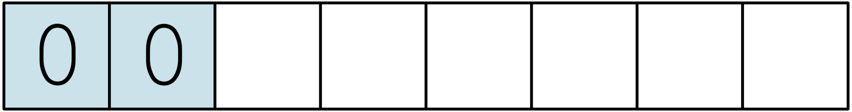
```
v.push_back(3);
```



Cost of `vector::push_back`



```
v.push_back(3);
```

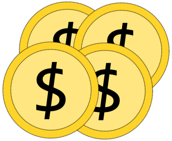


Cost of `vector::push_back`

```
v.push_back(3);
```

0	0	1					

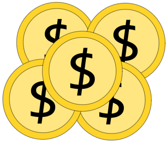
Cost of `vector::push_back`



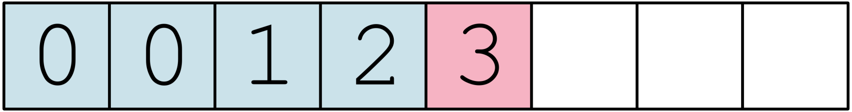
```
v.push_back(3);
```

0	0	1	2				

Cost of `vector::push_back`



```
v.push_back(3);
```



Worst case complexity?

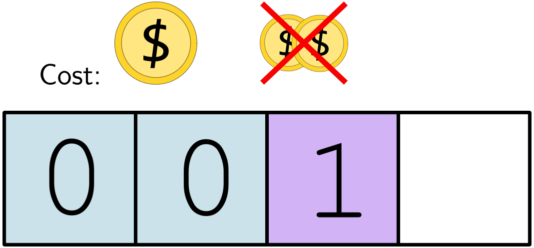
Worst case complexity?

$$\mathcal{O}(n)$$

n: Number of elements in the `vector`

push_back with an $\mathcal{O}(n)$ budget

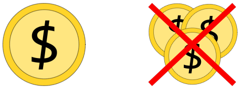

```
v.push_back(1);
```



push_back with an $\mathcal{O}(n)$ budget

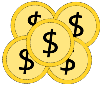
Budget: 

```
v.push_back(2);
```

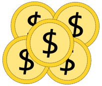
Cost: 

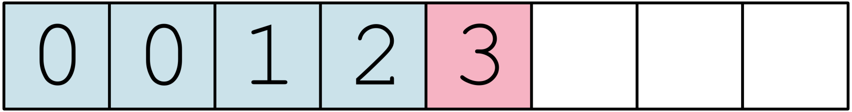
0	0	1	2

push_back with an $\mathcal{O}(n)$ budget

Budget: 

```
v.push_back(3);
```

Cost: 



I know what you're thinking...

That's a lot of wasted coins!

Piggybank Image Attribution: Videoplasty.com, CC-BY-SA 4.0 International

The rules of accounting

- Each operation has a fixed budget, given by the bounding function g
- Coins that are not spent on the operation itself go into the account
- If an operation runs out of coins, it may take coins from the account
- The account can not go negative. No debts!
- If this works for *every possible sequence of operations*, then the amortized complexity is $\mathcal{O}(g)$

The rules of accounting

- Each operation has a fixed budget, given by the bounding function g
- Coins that are not spent on the operation itself go into the account
- If an operation runs out of coins, it may take coins from the account
- The account can not go negative. No debts!
- If this works for *every possible sequence of operations*, then the amortized complexity is $\mathcal{O}(g)$

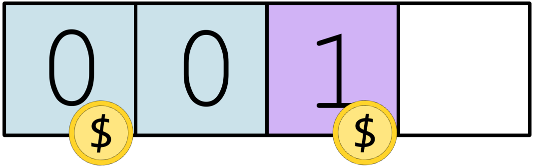
push_back with $\mathcal{O}(1)$ budget and account

push_back with $\mathcal{O}(1)$ budget and account

```
v.push_back(1);
```

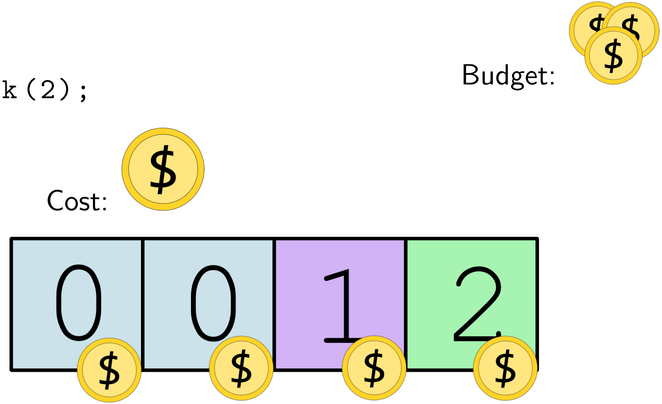
Budget:

Cost:



push_back with $\mathcal{O}(1)$ budget and account

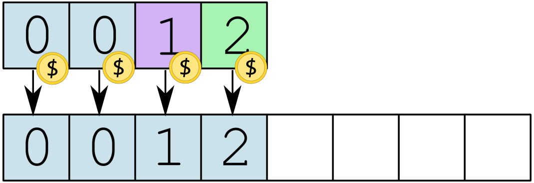
```
v.push_back(2);
```



push_back with $\mathcal{O}(1)$ budget and account

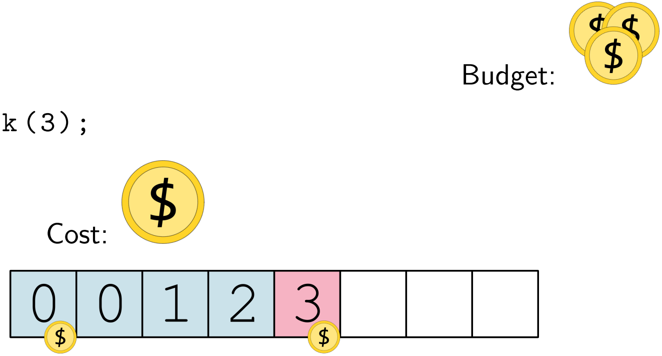
Budget: 

```
v.push_back(3);
```



push_back with $\mathcal{O}(1)$ budget and account

```
v.push_back(3);
```



Et voilà...

Amortized $\mathcal{O}(1)$ complexity!

Let's do another!

Let's do another!

26.4.2 Ranges [range.range]

1 The `range` concept defines the requirements of a type that allows iteration over its elements by providing an iterator and sentinel that denote the elements of the range.

```
template<class T>
  concept range =
    requires(T& t) {
      ranges::begin(t);          // sometimes equality-preserving (see below)
      ranges::end(t);
    };
```

2 The required expressions `ranges::begin(t)` and `ranges::end(t)` of the `range` concept do not require implicit expression variations (18.2).

3 Given an expression `t` such that `decltype((t))` is `T&`, `T` models `range` only if

(3.1) — [`ranges::begin(t)`, `ranges::end(t)`) denotes a range (25.3.1),

(3.2) — both `ranges::begin(t)` and `ranges::end(t)` are amortized constant time and non-modifying, and

(3.3) — if the type of `ranges::begin(t)` models `forward_iterator`, `ranges::begin(t)` is equality-preserving.

4 [*Note 1*: Equality preservation of both `ranges::begin` and `ranges::end` enables passing a range whose iterator type models `forward_iterator` to multiple algorithms and making multiple passes over the range by repeated calls to `ranges::begin` and `ranges::end`. Since `ranges::begin` is not required to be equality-preserving when the return type does not model `forward_iterator`, it is possible for repeated calls to not return equal values or to not be well-defined. — *end note*]

Ranges filter view

```
std::vector<int> v(1 << 20, 0);
v.back() = 42;
```

Ranges filter view

```cpp
std::vector<int> v(1 << 20, 0);
v.back() = 42;

auto is_non_zero = [](int i) { return i != 0; };
auto rng_non_zero =
  v | std::ranges::views::filter(is_non_zero);
```

Ranges filter view

```
std::vector<int> v(1 << 20, 0);
v.back() = 42;

auto is_non_zero = [](int i) { return i != 0; };
auto rng_non_zero =
  v | std::ranges::views::filter(is_non_zero);

auto it1 = rng_non_zero.begin();  // O(n)
```

Memoization!

Memoization!

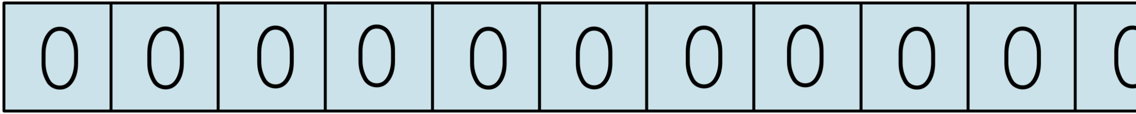
```
auto rng_non_zero = ...;

auto it1 = rng_non_zero.begin();  // O(n)
```

Memoization!

```
auto rng_non_zero = ...;
auto it1 = rng_non_zero.begin();  // O(n)
auto it2 = rng_non_zero.begin();  // O(1) (cached)
```

Ranges find with $\mathcal{O}(1)$ budget and account

```cpp
auto it1 = rng_non_zero.begin();
```



Ranges find with $\mathcal{O}(1)$ budget and account

```
auto it1 = rng_non_zero.begin();
```

Let's use another method then...

- Aggregate analysis
- ~~Accounting method~~
- Potential method

Aggregate Analysis

Let $T(n) = \sum_{i=1}^{n} c_i$ be the worst case execution time for executing an arbitrary sequence of calls to `filter_view::begin()` for a range of length s.
$\frac{T(n)}{n}$ is the amortized cost per call.
Let c_i be the cost of the i-th call to `begin()`.
Then $c_i = \begin{cases} i = 1 : \mathcal{O}(s) \\ i > 1 : \mathcal{O}(1) \end{cases}$.
Then for $n = 1$: $T(1) = c_1 = \mathcal{O}(s)$. Thus: $\frac{T(1)}{1} = \mathcal{O}(s)$.

$\Longrightarrow$ Amortized complexity is linear.

Potential Method

Let c_i be the actual cost, and $\hat{c}_i$ be the amortized cost of the ith operation. Let Φ_i be the *potential* of the filter view after applying the ith operation, and $\Phi_0 = 0$ be the initial potential.
The amortized cost for the sequence of operations is
$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n}(c_i) + \Phi_n - \Phi_0$.

Assume $\forall i.\ \hat{c}_i \in \mathcal{O}(1)$. Then for $n = 1$:
$\sum_{i=1}^{1} \hat{c}_i = \hat{c}_1 \in \mathcal{O}(1)$, but $c_1 \in \mathcal{O}(s)$.
Thus $\Phi_1 < 0$, which violates $\Phi_i \geq \Phi_0 \implies \hat{c}_i$ is not a valid upper bound.

$\implies$ Amortized complexity is not $\mathcal{O}(1)$.

Runtime Complexity

- $f \in \mathcal{O}(g) \iff \exists\ C > 0.\ \exists\ x_0 > 0.\ \forall\ x > x_0 : |f(x)| \leq C \cdot |g(x)|$

Runtime Complexity

- $f \in \mathcal{O}(g) \iff \exists\ C > 0.\ \exists\ x_0 > 0.\ \forall\ x > x_0 : |f(x)| \leq C \cdot |g(x)|$

Runtime Complexity

- $f \in \mathcal{O}(g) \iff \exists\ C > 0.\ \exists\ x_0 > 0.\ \forall\ x > x_0 : |f(x)| \leq C \cdot |g(x)|$

x here is the length of the input, the n from the proofs before was the length of the sequence of operations!

Runtime Complexity

- $f \in \mathcal{O}(g) \iff \exists\ C > 0.\ \exists\ x_0 > 0.\ \forall\ x > x_0 : |f(x)| \leq C \cdot |g(x)|$

x here is the length of the input, the n from the proofs before was the length of the sequence of operations!

For x we can ignore small numbers, for n we can *not*!

$\mathcal{O}(n)$ with memoization $\neq$ amortized $\mathcal{O}(1)$

Thanks for your attention.

ComicSansMS