5. Amortised analysis of algorithms [WiP]

José Proença

Algorithms (CC4010) 2023/2024

CISTER – U.Porto, Porto, Portugal

https://cister-labs.github.io/alg2324

Overview

- Measuring precisely performance of algorithms
- Measuring asymptotically performance of algorithms
- Analysing recursive functions
- Measuring precisely the average time of algorithms
- Analysis of sequences of operations (amortised analysis)
- Next: Graph traversals and Dynamic programming

Motivation

Running once vs.
Running many times

Insert a value in a Hashtable vs.
Insert 500 values in a Hashtable

Relevant when studying data structures – when the state changes

slides by Pedro Ribeiro, slides 3

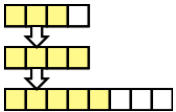
pages 1-12

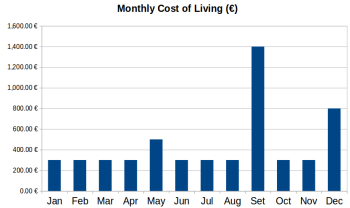
Amortized Analysis

Pedro Ribeiro

DCC/FCUP

2018/2019





Amortized Analysis

- In **amortized analysis** we are concerned about the **average** over a **sequence of operations**
  - Some operations may be costly, but others may be quicker, and in the end they even out
  - Typically applied to the analysis of a **data structure**

- This is different from the **regular worst case analysis**, where the worst cost of one operation is analyzed
  - There might be correlations between operations and worst case per operation might be too pessimistic because the only way of having an expensive operation might be to have a lot of cheap ones before

- This is different from **average case analysis**, and there is no probability or expectation involved
  - Amortized analysis still gives us a view of the **worst possible scenario**

Amortized cost

The amortized cost per operation for a sequence of n operations is the total cost of the operations divided by n

Amortized complexity

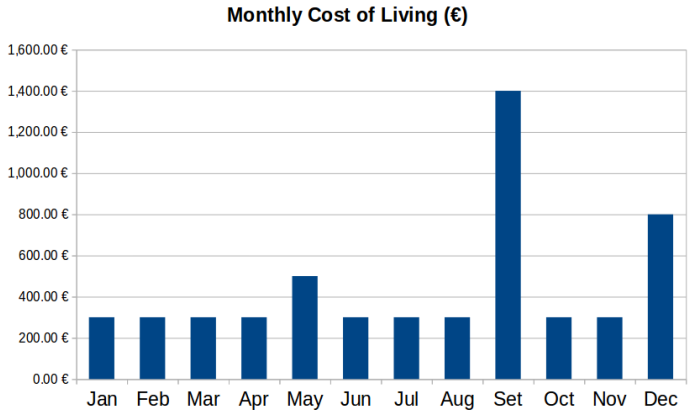
The amortized sequence complexity is the **worst case sequence complexity** (that is, the maximum possible total cost over all possible sequences of n operations) divided by n

Methods for Amortized Analysis

- **Aggregate** method (*total cost*)
  Examine total cost of operations and see the average
- **Accounting** method (*banker's view*)
  Impose extra charge on inexpensive operations, saving for future expensive operations
- **Potential** method (*physicist's view*)
  Define a (non-negative) potential function on the data structure state and use it to bound the cost

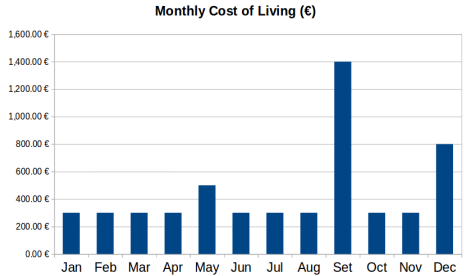
Intuition and Motivation

Real Life intuition: consider the monthly cost of living, a sequence of 12 operations:



Aggregate Method

- What is the **amortized cost** per month (operation)?
- Just **sum up** the cost of all months (operations) and **divide** by the number of months (operations)

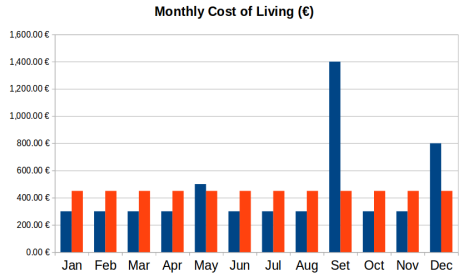


Aggregate method: sum of all months is 5,400€, which divided by 12 months gives an *amortized cost* of 450€ per month.

Accounting method

- Instead of computing the average ***after*** knowing the total cost, we may think of it from a different angle:

  *How much money do I need to **earn** each month in order to **keep living**, **never being broke** and always be able to pay the bills?*



Accounting method: If I always earn 450€, then I will always have enough money to pay the bill and never become broke:

- When I earn **more** than expenditures, I save some money for the future
- When I earn **less** than expenditures, I use the money I saved

A first example: stack as an array

Let's have a look at a data structure example.

Imagine I have a **stack implemented as an array** supporting the following operations:

- **push(x)** - add an element x to the top of the stack
- x **= pop()** - remove and return the element on the top

What happens when I need to **push an element and the array is full**?

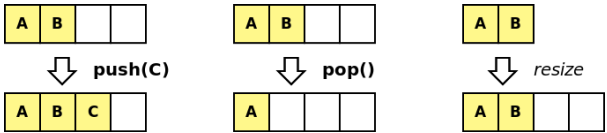
- We need to create a new, bigger array, copy the existing elements, and continue from there
- This is an **expensive operation** and a push that does this is going to cost a lot

Stack as an array

Let's first establish a cost model:

- **Inserting** an element into an existing array costs 1
- **Removing** an element from an array costs 1
- **Allocating** a new array costs 0 (zero)

This means a **push** or **pop** without resizing costs 1, and a resize would cost n, with n elements being copied to the new array



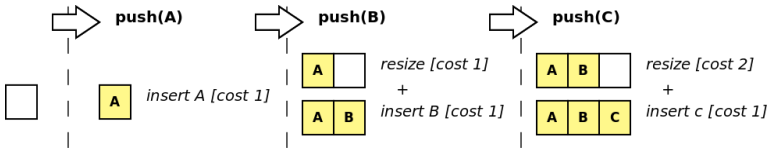
Stack as an array

What would be a good strategy for resizing the array?

A first strategy for resizing

Each time the array is full, **increment** its size by one

What is the cost of this strategy?
Let's use the **aggregate method** starting from an empty array of size 1:



Total cost of n pushes $= 1 + 2 + \ldots + n = \sum_{i=1}^{n} i = \frac{(n+1)n}{2} = \mathcal{O}(n^2)$

The **average** over n operations, our **amortized cost**, is $\mathcal{O}(n)$.

Can we do **better**? (than amortized linear cost)

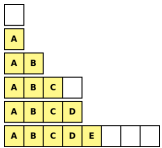
Stack as an array

Let's try something different:

A better strategy for resizing

Each time the array is full, **double its size**

Cost? Let's use the **aggregate method** as before:



The cost of **insertions** is just n
The cost of **resizes** is $1 + 2 + 4 + \ldots + 2^i$ for some $2^i < n$
This sum is smaller than $2n$

Total cost of n pushes $=$ cost(insertions) $+$ cost(resizes) $< 3n$

Average over n operations is < 3, i.e., our **amortized cost**, is $\mathcal{O}(1)$.

Stack as an array

- Using **amortized analysis** we have shown that doubling the size of the array is a good strategy with **constant amortized time** (per operation)
  - What would a (non amortized) **regular worst case analysis** say?
  - "Increase size by 100 when full" would **not** be a good strategy. Why?
  - What about a strategy for saving (unused) space when **popping**? Would halving the array be good? We'll discuss that on the exercises :)
- We are (still) missing the other methods (**accounting** and **potential**). Let's have a look at how they would work in this example.

Exercise – Queue as 2 Stacks

```
typedef struct
    queue {
   Stack A;
   Stack B;
} Queue;
```

Consider 2 operations

- enqueue *pushes* a value in stack A;
- dequeue *pops* a value from stack B if non-empty otherwise *pops* all elements from stack A and *pushes* them to stack B, except the last one that is returned;
- assume each *push* and *pop* has a cost of 1;

Ex. 5.1: What is the cost of the best and worst scenarios for each operation?

Ex. 5.2: What is the worst possible sequence of N operations?

Ex. 5.3: What is the amortised cost for each operation, using the aggregate method?

slides by Pedro Ribeiro, slides 3

pages 13-18

Stack as an array - Accounting method

How much *money* do we need to **earn** for each **push** operation, so that all future operations can be paid for?

- Earn $1 for each **push**
  - We spend that $1 for inserting (the *"insert-dollar"*)
  - When we need to copy this element to a new array (when resizing), we don't have enough money...
    BROKE!
- Earn $2 for each **push**
  - $1 for inserting (the *"insert-dollar"*)
  - $1 for copying to a new array (the *"copy-dollar"*)
  - When if we need to copy it again for a second time (during a new resize)? ...
    BROKE!

Stack as an array - Accounting method

- Earn \$3 for each **push**
  - \$1 for inserting (the *"insert-dollar"*)
  - \$1 for copying to a new array (the *"copy-dollar"*)
  - \$1 for recharging old elements that have spent their copy-dollars to a new array (the *"recharge-dollar"*)
    NEVER GO BROKE!

Old elements who have used their copy-dollars

new elements that spare \$1 each for recharging old elements

We will always have enough new elements sparing enough money for all the old elements because of the way we resize: **TWICE** the old size

Amortized cost of 3 (as before) which is $\mathcal{O}(1)$

Stack as an array - Potential method

The **potential method** uses an idea similar to the banker's view but using a different methodology

Suppose we can define a potential function Φ ("Phi") on the state of a data structure with the following properties:

- $\Phi(s_0) = 0$ here s_0 is the initial state of the data structure
- $\Phi(s_t) \geq 0$ for all states s_t occurring during the sequence of operations

Intuitively, the potential function should keep track of the *"precharged cost"* at any given time, measuring how much we have *"saved"* for future operations, such as in the banker's method.

The difference is that it **depends only on the current state of the data structure**, regardless of the operations that got us in that state

Stack as an array - Potential method

Armed with this, we can define the **amortized cost** of an operation as

$\mathbf{c} + \mathbf{\Phi}(\mathbf{s}') - \mathbf{\Phi}(\mathbf{s})$

Where c is the actual cost of the operation and s and s' are the states of the data structure before and after the operation, respectively:

Amortized cost is the actual cost plus the change in potential

Now, consider a sequence of n operations with actual costs $c_0, c_1, \ldots, c_{n-1}$ leading from state s_0 to states $s_1, s_2, \ldots, s_n$.

The **total amortized cost** is equal to the sums of amortized cost:

$[c_0 + \Phi(s_1) - \Phi(s_0)] + [c_1 + \Phi(s_2) - \Phi(s_1)] + \ldots + [c_{s-1} + \Phi(s_n) - \Phi(s_{n-1})]$
$= c_0 + c_1 + \ldots + c_{n-1} + \Phi(s_n) - \Phi(s_0)$
$= c_0 + c_1 + \ldots + c_{n-1} + \Phi(s_n)$

Because Φ is always positive, we are overestimating the actual cost by $\Phi(s_n)$ and the amortized cost is an **upper bound on the actual cost**!

Stack as an array - Potential method

Let's choose a **suitable potential function** for our case:

Let $\Phi(s)$ be **2x the nr of elements in the array after the midpoint**
$\Phi(s) = max(0, 2(n - m/2)) = max(0, 2n - m)$
where n is the number of elements in the array and m the size of the array.

(Note how this corresponds to the surplus money in the banker's method)

- $\Phi(s_0) = 0$ and $\Phi(s_t) \geq 0$ for all states s_t as required

Now we need to show that the amortized cost is $\mathcal{O}(1)$
Consider a single push operation:

- If $n < m$, then the actual cost is 1 and m does not change. The potential increases by at most 2 (when $2n > m - 1$). So, the amortized cost is at most $1 + 2 = 3$.
- If $n == m$, then the actual cost is $n + 1$, but the potential drops from n to 2, which means the amortized cost is $n + 1 + (2 - n) = 3$

And we have our **constant amortized cost** of 3!

Stack as an array - Potential method

A few more considerations:

- There are a multitude of possible potential functions
- The choice of potential function is essential and influences our conclusions (we are trying to establish an upper bound)
- It is really important that the **potential function is never negative**! (or we risk going "bankrupt")
- Sometimes we might need to have an initial positive amount of money ($\Phi(s_0) > 0$). In these cases we show that the actual cost for n operations is at most the amortized cost plus the initial seed amount.

We can even establish different amortized costs for different operations:

- In the case of the stack as an array what would be the amortized cost of a **pop()** operation?
- Because a *pop()* can never increase Φ (it may even decrease it), its amortized cost will at most 1 (the actual cost for removing)

Exercise – Queue as 2 Stacks (again)

```
typedef struct
    queue {
   Stack A;
   Stack B;
} Queue;
```

Consider 2 operations

- enqueue *pushes* a value in stack A;
- dequeue *pops* a value from stack B if non-empty otherwise *pops* all elements from stack A and *pushes* them to stack B, except the last one that is returned;
- assume each *push* and *pop* has a cost of 1;

Ex. 5.4: What is the amortised cost for each operation, using the potential method?

slides by Pedro Ribeiro, slides 3

pages 19-23

Binary Counter

Let's see another example and use again the three methods to solve it.

Consider the problem of storing a big **binary counter** in an array A, where each position $A[i]$ stores the i-th bit.

Consider we use a **cost model** where flipping a bit costs \$1.

All entries start at 0 and at each step we will be incrementing the counter:

...	A[3]	A[2]	A[1]	A[0]	Cost
	0	0	0	0	
	0	0	0	**1**	\$1
	0	0	**1**	**0**	\$2
	0	0	1	**1**	\$1
	0	**1**	**0**	**0**	\$3
	0	1	0	**1**	\$1
	0	1	**1**	**0**	\$2
	0	1	1	**1**	\$1
	1	**0**	**0**	**0**	\$4

Binary Counter

What is the **cost of each increment**?

- Suppose that after incrementing we get the number n
- In the worst case we need to change all the bits of n, i.e., $\mathcal{O}(\log n)$
- A traditional worst case analysis for n operations would point to something like $\mathcal{O}(n \log n)$ for the total cost

We can intuitively see that the actual cost should have **a more tight bound**, because the worst case (changing all bits) does not occur often.

But how can we show that? This is the goal of **amortized analysis**!

Binary Counter - Aggregate Method

Let's consider how often we flip each bit during n operations:

- $A[0]$ is always flipped: n times
- $A[1]$ is flipped every 2nd time: at most $n/2$ times
- $A[2]$ is flipped every 4th time: at most $n/4$ times
- ...
- $A[i]$ is flipped every 2^i-th time: at most $n/2^i$ times

So the total cost is bounded by:

$n + n/2 + n/4 + n/8 + \ldots \leq 2n$ (geometric series with ratio $1/2$)

The total cost of n increments is therefore $\mathcal{O}(n)$

The **amortized cost per increment** is only 2, which is constant!

Binary Counter - Accounting Method

How much money do we need to **earn** at each **increment** operation so that all future operations can be paid for?

$2 suffices! (as we might guess from the aggregate method)

Note that we still need to prove that it never gets us below zero

- At any increment there is only one bit flipping from 0 to 1
- Use $1 to pay for that flip
- Save $1 to pay for any future flip of that bit from 1 to 0

We will never be broke because in any increment we are paying for the $0 \rightarrow 1$ flip, and all $1 \rightarrow 0$ flips can use their previously saved money.

Binary Counter - Potential Method

Can you think of a **suitable potential function**?

Let $\Phi(s)$ be **the quantity of 1 bits in the number representation**

(Note how this corresponds to the surplus money in the banker's method)

- $\Phi(s_0) = 0$ and $\Phi(s_t) \geq 0$ for all states s_t as required

Now we need to show that the amortized cost is $\mathcal{O}(1)$
Consider a single increment operation:

- Because there is only one $0 \rightarrow 1$ flip, the actual cost is $1 + nr(1 \rightarrow 0)$
- The change in potential after every operation is $1 - nr(1 \rightarrow 0)$
- The amortized cost is therefore $1 + nr(1 \rightarrow 0) + 1 - nr(1 \rightarrow 0) = 2$

And we have our **constant amortized cost** of 2!

slides by Pedro Ribeiro, slides 3

pages 29-30

Recap - Amortized Analysis

First slide again

- In **amortized analysis** we are concerned about the **average** over a **sequence of operations**
  - Some operations may be costly, but others may be quicker, and in the end they even out
  - Typically applied to the analysis of a **data structure**
- This is different from the **regular worst case analysis**, where the worst cost of one operation is analyzed
  - There might be correlations between operations and worst case per operation might be too pessimistic because the only way of having an expensive operation might be to have a lot of cheap ones before
- This is different from **average case analysis**, and there is no probability or expectation involved
  - Amortized analysis still gives us a view of the **worst possible scenario**

Recap - Methods for Amortized Analysis

- **Aggregate** method (*total cost*)
  Examine total cost of operations and see the average
- **Accounting** method (*banker's view*)
  Impose extra charge on inexpensive operations, saving for future expensive operations
- **Potential** method (*physicist's view*)
  Define a (non-negative) potential function on the data structure state and use it to bound the cost

Queue as 2 stacks – another explanation

slides by Pedro Ribeiro, slides 3

pages 24-28

Queue as two stacks

Imagine you have access to an implementation of a **stack** (LIFO) supporting the following operations (with each operation costing 1):

- **push(x)** - add an element x to the top of the stack
- **pop()** - remove and return the element on the top
- **empty()** - return a boolean value indicating if the stack is empty

Now imagine you want to implement a **queue** (FIFO) supporting the following two operations

- **enqueue(x)** - add an element x to the end of the queue
- **dequeue()** - remove and return the element at the head of the queue

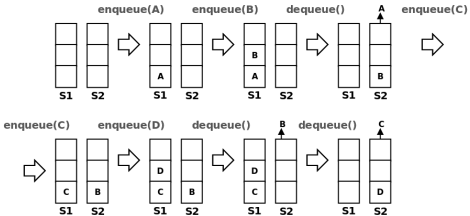
Could you implement the queue using only the stacks API?

Queue as two stacks

One way would be using two stacks S1 and S2 and the following algorithm:

- **enqueue(x)** - push x into S1
- **dequeue()** - if $S2$ is empty, then pop all elements from S1, pushing each element in turn into $S2$. Now, pop from S2 and return the result

Can you see why this algorithm is correct?



What is the **cost per operation**? A typical worst case analysis would establish $\mathcal{O}(n)$ for the dequeue() operation, but this is clearly a weak bound (not all dequeues are expensive). Let's use... **amortized analysis**!

Queue as two stacks - Aggregate Method

What is the **total cost**?

- Each element is clearly pushed at most twice and popped at most twice (once for each stack).
- If an element is enqueued and never dequeued, it is pushed at most twice and popped at most once. The amortized cost for enqueue is 3.
- The amortized cost of the dequeue covers the final pop, plus the verification of the stack being empty. Its amortized cost is 2.

And we have our **constant amortized cost**!

Queue as two stacks - Accounting Method

How much money do we need to **earn** at each **enqueue** and **dequeue** operation so that all future operations can be paid for?

$3 for enqueue suffices! (and we will never be broke)

- Use $1 to pay for that push into S1
- Save $1 to pop it out of S1
- Save $1 to pay for the push to S2

$2 for dequeue suffices! (and we will never be broke)

- Use $1 for verifying if the stack is empty
- The saved money allows for inverting S1 into S2 if needed
- Use $1 to pop it out of S2

And we have our **constant amortized cost**!

Queue as two stacks - Potential Method

Can you think of a **suitable potential function**?

Let $\Phi(s)$ be **twice the number of elements in S1**

(Note how this corresponds to the surplus money in the banker's method)

- $\Phi(s_0) = 0$ and $\Phi(s_t) \geq 0$ for all states s_t as required
- An **enqueue** will have an actual cost of 1 (push into S1) plus an increase of 2 in the potential, leading to an amortized cost of 3.
- A **dequeue** will have at least an actual cost of 2 (empty + pop out of S2). In case S2 is empty and x is the number of elements in S1, we will be popping x times out of S1 plus pushing x into S2 (an added actual cost of $2x$), but this will be canceled out by a decrease of potential precisely in $2x$. The amortized cost is 2.

And we have our **constant amortized cost**!

Exercises

```
node *insert_rem(node *p, int x) {
  node *new = malloc(sizeof(node));
  new->value = x;
  while (p && x > p->value)
    { aux=p; p=p->next; free(aux);}
  new->next = p;
  return new;
}
```

- inserting to the head costs 1;
- removing from the head costs 1;
- E.g., `insert_rem(4,[1,3,5,7])` returns `[4,5,7]` and costs 3.

Ex. 5.5: What is the asymptotic execution time for the best and worst cases?

Ex. 5.6: What is the amortised cost for `insert_rem`, using the aggregate method?

Ex. 5.7: Present a potential function for `insert_rem` and use it to compute the constant amortised cost.