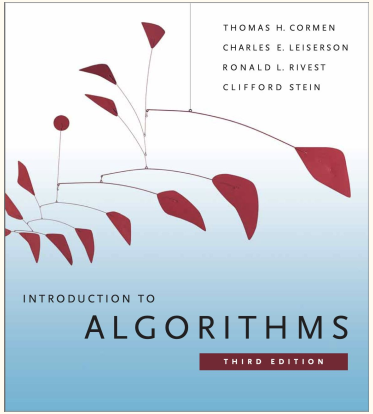
Data Structures and Algorithms

Tutorial 2. Amortized analysis

Today’s topic is covered in detail in

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. **Introduction to Algorithms.** The MIT Press 2009.



Stack data type

Consider a Stack with the following methods:

- `Stack.push(v)` — push value `v` onto the stack
- `Stack.pop()` — pop the top value from the stack

Stack data type

Consider a Stack with the following methods:

- `Stack.push(v)` — push value `v` onto the stack
- `Stack.pop()` — pop the top value from the stack
- `Stack.popMany(k)` — pop the top `k` values from the stack

Stack data type

Consider a Stack with the following methods:

- `Stack.push(v)` — push value `v` onto the stack $O(1)$
- `Stack.pop()` — pop the top value from the stack $O(1)$
- `Stack.popMany(k)` — pop the top `k` values from the stack $O(k)$

Stack data type

Consider a Stack with the following methods:

- `Stack.push(v)` — push value `v` onto the stack $O(1)$
- `Stack.pop()` — pop the top value from the stack $O(1)$
- `Stack.popMany(k)` — pop the top `k` values from the stack $O(k)$

`push(1), push(2), push(3), pop(), push(4), push(5), popMany(4)`

Question: What is the total running time for a sequence of N operations?

Amortized analysis

Amortized analysis considers a sequence of operations and guarantees the average cost of each operation in the worst case.

Amortized analysis

Amortized analysis considers a sequence of operations
and guarantees the average cost of each operation in the worst case.

Sometimes an individual operation may be expensive,
but considering all possible sequences of operations,
with amortized analysis we may show that an average cost is small.

Aggregate analysis

Aggregate analysis is the most straightforward kind of amortized analysis: we merely determine the complexity of the sequence of N operations

e.g. T(N) is upper bound for running time of a sequence of N operations

then we merely divide by N, getting

average cost of each operation in the sequence is T(N)/N

Aggregate analysis: Stack

Exercise 2.1. Analyze the total running time of a sequence of N operations (push, pop, and popMany) on an initially empty stack.

Aggregate analysis: Stack

Exercise 2.1. Analyze the total running time of a sequence of N operations (push, pop, and popMany) on an initially empty stack.

Partial solution.

The worst-case of push and pop is O(1), so a sequence of those will have a complexity of θ(N).

The worst-case of popMany is O(n) for a stack of size n.

So in the worst case the total running time for N operations will be $O(n^2)$.

This leads us to amortized cost per operation of O(n).

Aggregate analysis: Stack

Exercise 2.1. Analyze the total running time of a sequence of N operations (push, pop, and popMany) on an initially empty stack.

Partial solution.

The worst-case of push and pop is $O(1)$, so a sequence of those will have a complexity of $\theta(N)$.

The worst-case of popMany is $O(n)$ for a stack of size n.

So in the worst case the total running time for N operations will be $O(n^2)$.

This leads us to amortized cost per operation of $O(n)$.

Although this analysis is correct, the resulting upper bound is not tight.

Aggregate analysis: Stack

Exercise 2.1. Analyze the total running time of a sequence of N operations (push, pop, and popMany) on an initially empty stack.

Full solution.
The worst-case of push and pop is $O(1)$, so a sequence of those will have a complexity of $\theta(N)$.
The worst-case of popMany is $O(n)$ for a stack of size n.
However, each pop() (including those inside popMany) has to correspond to some push(). So in the worst case, there cannot be more than N pop() operations, meaning that overall running time is $O(n)$.
And amortized cost of each operation is $O(1)$.

Aggregate analysis: Binary counter

Exercise 2.2. Analyze the total running time of a sequence of increments to a binary counter, implemented as a bit array.

```
INCREMENT(A)
i = 0
while i < A.length and A[i] == 1
    A[i] = 0
    i = i + 1
if i < A.length
    A[i] = 1
```

The accounting method

1. Each specific operation in the sequence has an **actual cost**

The accounting method

1. Each specific operation in the sequence has an **actual cost**
2. We assign for each (type of) operation a fixed **amortized cost**

The accounting method

1. Each specific operation in the sequence has an **actual cost**
2. We assign for each (type of) operation a fixed **amortized cost**
3. When (amortized cost $>$ actual cost) then we get **credit:**
   1. credit $=$ amortized cost – actual cost

The accounting method

1. Each specific operation in the sequence has an **actual cost**
2. We assign for each (type of) operation a fixed **amortized cost**
3. When (amortized cost > actual cost) then we get **credit:**
    1. credit = amortized cost – actual cost
4. We have to show that we always have enough credit to cover the actual cost of any operation:

$$\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$$

The accounting method: Stack

Exercise 2.3. Perform amortized analysis using the accounting method for a sequence of N operations (push, pop, and popMany) on an initially empty stack.

The accounting method: Stack

Exercise 2.3. Perform amortized analysis using the accounting method for a sequence of N operations (push, pop, and popMany) on an initially empty stack.

Solution.

Operation	Actual cost	Amortized cost
`push(v)`	1	2
`pop()`	1	0
`popMany(k)`	min(n, k)	0

The accounting method: Stack

Exercise 2.3. Perform amortized analysis using the accounting method for a sequence of N operations (push, pop, and popMany) on an initially empty stack.

Operation	Actual cost	Amortized cost
`push(v)`	1	2
`pop()`	1	0
`popMany(k)`	min(n, k)	0

Solution.

For each push() we save 1 credit.

For each pop() there must have been a corresponding push() that would pay for it. Similarly, for popMany() there will be exactly the necessary number of push() operations to pay for popMany().

The accounting method: Binary counter

Exercise 2.4. Using the accounting method, perform amortized analysis of a sequence of increments to a binary counter, implemented as a bit array.

```
INCREMENT(A)
i = 0
while i < A.length and A[i] == 1
    A[i] = 0
    i = i + 1
if i < A.length
    A[i] = 1
```

The potential method

1. Instead of credit we use **potential** that is determined by the state of a data structure (not operations).

The potential method

1. Instead of credit we use **potential** that is determined by the state of a data structure (not operations).
2. We define a **potential function** $\Phi(D_i)$ that maps each state of the data structure to a real number.

The potential method

1. Instead of credit we use **potential** that is determined by the state of a data structure (not operations).
2. We define a **potential function** $\Phi(D_i)$ that maps each state of the data structure to a real number.
3. The amortized cost of an operation is then defined as its actual cost plus a difference of potentials:

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

The potential method

1. Instead of credit we use **potential** that is determined by the state of a data structure (not operations).
2. We define a **potential function** $\Phi(D_i)$ that maps each state of the data structure to a real number.
3. The amortized cost of an operation is then defined as its actual cost plus a difference of potentials:

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

4. We need to show that we have enough potential:

$$\forall i, \Phi(D_i) \geq \Phi(D_0)$$

The potential method: Stack

Exercise 2.5. Perform amortized analysis using the potential method for a sequence of N operations (push, pop, and popMany)on an initially empty stack.

The potential method: Stack

Exercise 2.5. Perform amortized analysis using the potential method for a sequence of N operations (push, pop, and popMany)on an initially empty stack.

Solution.
Let $\Phi(D) = n$, if n is the number of elements on the stack D.

The potential method: Stack

Exercise 2.5. Perform amortized analysis using the potential method for a sequence of N operations (push, pop, and popMany)on an initially empty stack.

Solution.
Let $\Phi(D) = n$, if n is the number of elements on the stack D.
It is clear that for all i we have $\Phi(D_i) \geq 0 = \Phi(D_0)$.
Now we compute the amortized cost of operations:

The potential method: Stack

Exercise 2.5. Perform amortized analysis using the potential method for a sequence of N operations (push, pop, and popMany)on an initially empty stack.

Solution.
Let $\Phi(D) = n$, if n is the number of elements on the stack D.
It is clear that for all i we have $\Phi(D_i) \geq 0 = \Phi(D_0)$.
Now we compute the amortized cost of operations:

- `push()` —
$$\begin{aligned} \hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= 1 + (n+1) - n \\ &= 2 \end{aligned}$$

The potential method: Stack

Exercise 2.5. Perform amortized analysis using the potential method for a sequence of N operations (push, pop, and popMany)on an initially empty stack.

Solution.
Let $\Phi(\mathrm{D}) = \mathrm{n}$, if n is the number of elements on the stack D.
It is clear that for all i we have $\Phi(\mathrm{D_i}) \geq 0 = \Phi(\mathrm{D_0})$.
Now we compute the amortized cost of operations:

- pop() —
$$\begin{aligned} \hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= 1 + n - (n+1) \\ &= 0 \end{aligned}$$

The potential method: Stack

Exercise 2.5. Perform amortized analysis using the potential method for a sequence of N operations (push, pop, and popMany)on an initially empty stack.

Solution.

Let $\Phi(\mathrm{D}) = \mathrm{n}$, if n is the number of elements on the stack D.

It is clear that for all i we have $\Phi(\mathrm{D_i}) \geq 0 = \Phi(\mathrm{D_0})$.

Now we compute the amortized cost of operations:

- `popMany()` — $$\begin{aligned} \hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= \min(n,k) - \min(n,k) \\ &= 0 \end{aligned}$$

The potential method: Binary counter

Exercise 2.6. Using the potential method, perform amortized analysis of a sequence of increments to a binary counter, implemented as a bit array.

```
INCREMENT(A)
i = 0
while i < A.length and A[i] == 1
    A[i] = 0
    i = i + 1
if i < A.length
    A[i] = 1
```

Amortized analysis: extra exercises

Exercise 2.7. See exercises in Chapter 17 of Cormen et al.

Exercise 2.8. Consider a queue implemented as a pair of stacks:

- a queue is empty if both stacks are empty
- we push(v) into the rear stack
- and pop() from the front stack; if the front stack is empty, we repeatedly pop elements from a rear stack and push them into the front stack, and then perform the regular pop()

Perform amortized analysis of a sequence of push() and pop() operations performed on an initially empty queue.

Summary

- Amortized complexity
- Aggregate analysis
- The accounting method (banker’s method)
- The potential method (physicist’s method)

Summary

- Amortized complexity
- Aggregate analysis
- The accounting method (banker’s view)
- The potential method (physicist’s view)

See you next week!