



Chapter 4

Amortized Analysis

Algorithm Theory
WS 2016/17

Fabian Kuhn

Amortization

- Consider sequence o_1, o_2, \dots, o_n of n operations (typically performed on some data structure D)
- t_i : execution time of operation o_i
- $T := t_1 + t_2 + \dots + t_n$: total execution time
- The execution time of a single operation might vary within a large range (e.g., $t_i \in [1, O(i)]$)
- The worst case overall execution time might still be small
→ average execution time per operation might be small in the worst case, even if single operations can be expensive

- Best case
- Worst case
- Average case
- Amortized worst case

What is the **average cost** of an operation in a **worst case sequence** of operations?

Example 1: Augmented Stack

Stack Data Type: Operations

- $S.\text{push}(x)$: inserts x on top of stack
- $S.\text{pop}()$: removes and returns top element

Complexity of Stack Operations

- In all standard implementations: $O(1)$

Additional Operation

- **$S.\text{multipop}(k)$** : remove and return top k elements
- Complexity: $O(k)$
- What is the amortized complexity of these operations?

Augmented Stack: Amortized Cost

Amortized Cost

- Sequence of operations $i = 1, 2, 3, \dots, n$
- Actual cost of op. i : t_i
- Amortized cost of op. i is a_i if for every possible seq. of op.,

$$T = \sum_{i=1}^n t_i \leq \sum_{i=1}^n a_i$$

Actual Cost of Augmented Stack Operations

- $S.\text{push}(x), S.\text{pop}()$: actual cost $t_i = O(1)$
- $S.\text{multipop}(k)$: actual cost $t_i = O(k)$
- **Amortized cost** of all three operations is **constant**
 - The total number of “popped” elements cannot be more than the total number of “pushed” elements: **cost for pop/multipop \leq cost for push**

Augmented Stack: Amortized Cost

Amortized Cost

$$T = \sum_i t_i \leq \sum_i a_i$$

Actual Cost of Augmented Stack Operations

- $S.\text{push}(x), S.\text{pop}()$: actual cost $t_i \leq c$
- $S.\text{multipop}(k)$: actual cost $t_i \leq c \cdot k$

Example 2: Binary Counter

Incrementing a binary counter: determine the bit flip cost:

Operation	Counter Value	Cost
	00000	
1	0000 1	1
2	000 10	2
3	000 11	1
4	00 100	3
5	0010 1	1
6	001 10	2
7	001 11	1
8	0 1000	4
9	0100 1	1
10	010 10	2
11	010 11	1
12	01 100	3
13	0110 1	1

Accounting Method

Observation:

- Each increment flips exactly one 0 into a 1

$$00100\mathbf{0}1111 \Rightarrow 00100\mathbf{1}0000$$

Idea:

- Have a bank account (with initial amount 0)
- Paying x to the bank account costs x
- Take “money” from account to pay for expensive operations

Applied to binary counter:

- Flip from 0 to 1: pay 1 to bank account (cost: 2)
- Flip from 1 to 0: take 1 from bank account (cost: 0)
- Amount on **bank account = number of ones**
→ We always have enough “money” to pay!

Accounting Method

Op.	Counter	Cost	To Bank	From Bank	Net Cost	Credit
	0 0 0 0 0					
1	0 0 0 0 1	1				
2	0 0 0 1 0	2				
3	0 0 0 1 1	1				
4	0 0 1 0 0	3				
5	0 0 1 0 1	1				
6	0 0 1 1 0	2				
7	0 0 1 1 1	1				
8	0 1 0 0 0	4				
9	0 1 0 0 1	1				
10	0 1 0 1 0	2				

Potential Function Method

- Most **generic** and **elegant** way to do amortized analysis!
 - But, also more abstract than the others...
- State of data structure / system: $S \in \mathcal{S}$ (state space)

Potential function $\Phi: \mathcal{S} \rightarrow \mathbb{R}_{\geq 0}$

- **Operation i :**
 - t_i : actual cost of operation i
 - S_i : state after execution of operation i (S_0 : initial state)
 - $\Phi_i := \Phi(S_i)$: potential after exec. of operation i
 - a_i : **amortized cost** of operation i :

$$a_i := t_i + \Phi_i - \Phi_{i-1}$$

Potential Function Method

Operation i :

actual cost: t_i **amortized cost:** $a_i = t_i + \Phi_i - \Phi_{i-1}$

Overall cost:

$$T := \sum_{i=1}^n t_i = \left(\sum_{i=1}^n a_i \right) + \Phi_0 - \Phi_n$$

Binary Counter: Potential Method

- **Potential function:**
 Φ : number of ones in current counter
- Clearly, $\Phi_0 = 0$ and $\Phi_i \geq 0$ for all $i \geq 0$
- Actual cost t_i :
 - 1 flip from 0 to 1
 - $t_i - 1$ flips from 1 to 0
- Potential difference: $\Phi_i - \Phi_{i-1} = 1 - (t_i - 1) = 2 - t_i$
- Amortized cost: $a_i = t_i + \Phi_i - \Phi_{i-1} = 2$

Example 3: Dynamic Array

- How to create an array where the size dynamically adapts to the number of elements stored?
 - e.g., Java “ArrayList” or Python “list”

Implementation:

- Initialize with initial size N_0
- Assumptions: Array can only grow by appending new elements at the end
- If array is full, the size of the array is increased by a factor $\beta > 1$

Operations (array of size N):

- read / write: actual cost $O(1)$
- append: actual cost is $O(1)$ if array is not full, otherwise the append cost is $O(\beta \cdot N)$ (new array size)

Example 3: Dynamic Array

Notation:

- n : number of elements stored
- N : current size of array

Cost t_i of i^{th} append operation:
$$t_i = \begin{cases} 1 & \text{if } n < N \\ \beta \cdot N & \text{if } n = N \end{cases}$$

Claim: Amortized append cost is $O(1)$

Potential function Φ ?

- should allow to pay expensive append operations by cheap ones
- when array is full, Φ has to be large
- immediately after increasing the size of the array, Φ should be small again

Dynamic Array: Potential Function



Cost t_i of i^{th} append operation: $t_i = \begin{cases} 1 & \text{if } n < N \\ \beta \cdot N & \text{if } n = N \end{cases}$

Dynamic Array: Amortized Cost

Cost t_i of i^{th} append operation: $t_i = \begin{cases} 1 & \text{if } n < N \\ \beta \cdot N & \text{if } n = N \end{cases}$