universität freiburg

Algorithm Theory – WS 2024/25

Chapter 4 : Amortized Analysis

Fabian Kuhn
Dept. of Computer Science
Algorithms and Complexity

Amortization

- Consider sequence $o_1, o_2, \dots, o_n$ of n operations (typically performed on some data structure D)

- $\boldsymbol{t_i}$: execution time of operation o_i
- $\boldsymbol{T := t_1 + t_2 + \cdots + t_n}$: 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

Analysis of Algorithms

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

The best case usually does not occur and is not really interesting.

The **standard** way of algorithm analysis.

Assume that the input is **random** according to some given distribution.

Average cost per operation in a **worst case sequence of operations**

- a form of worst-case analysis for sequences of operations

Example 1: Augmented Stack

Stack Data Type: Operations

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

Complexity of Stack Operations

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

Additional Operation

- $\boldsymbol{S}$**.multipop(**$\boldsymbol{k}$**)** : remove and return top k elements
- Complexity: $O(k)$

What is the amortized complexity of these operations?

Intuitively: amortized cost per operation is constant

- We can only delete items from S that were previously pushed to S.
- The total time for deleting is not more than for pushing.

Augmented Stack: Amortized Cost

Amortized Cost

- Sequence of operations $i = 1, 2, 3, \dots, n$
- Actual cost of op. i: $\boldsymbol{t_i}$
- Amortized cost of op. i is $\boldsymbol{a_i}$ 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

- $S.\mathrm{push}(x), S.\mathrm{pop}()$: actual cost $t_i = O(1)$
- $S.\mathrm{multipop}(k)$: 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

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

$\boldsymbol{n}$ **operations:** p push operations, the rest are pop and multipop op.

- $p \leq n$ push op. $\implies$ total push cost $\leq c \cdot p$
- total #deleted elem. $\leq p$ $\implies$ total pop/multipop cost $\leq c \cdot p$

$\implies$ total cost $\leq 2 \cdot c \cdot p$

- **Average cost per operation** $\boldsymbol{\leq \frac{2cp}{n} \leq \frac{2cp}{p} = 2c}$

Example 2: Binary Counter

Incrementing a binary counter:

determine the bit flip cost:

Operation	Counter Value	Cost
	00000	
1	00001	1
2	00010	2
3	00011	1
4	00100	3
5	00101	1
6	00110	2
7	00111	1
8	01000	4
9	01001	1
10	01010	2
11	01011	1
12	01100	3
13	01101	1

Accounting Method

Observation:

- Each increment flips exactly one 0 into a 1

$$0010001111 \Longrightarrow 0010010000$$

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

amortized cost

Op.	Counter	Cost	To Bank	From Bank	Net Cost	Balance
	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				

$$C + \underbrace{T - F}_{B \geq 0} = A \qquad B \geq 0$$

$$\Longrightarrow A \geq C$$

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 $\boldsymbol{\Phi}: \boldsymbol{\mathcal{S}} \to \mathbb{R}_{\geq 0}$

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

$$\boldsymbol{a_i \coloneqq t_i + \Phi_i - \Phi_{i-1}}$$

Potential Function Method

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

Operation i:

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

Overall cost:

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

$$\begin{aligned}
\sum_{i=1}^{n} t_i = a_1 & + \Phi_0 - \Phi_1 \\
+ a_2 & \qquad + \Phi_1 - \Phi_2 \\
+ a_3 & \qquad\qquad + \Phi_2 - \Phi_3 \\
+ a_4 & \qquad\qquad\qquad + \Phi_3 \cdots \\
\vdots & \\
+ a_{n-1} & \qquad\qquad\qquad\qquad \cdots - \Phi_{n-1} \\
+ a_n & \qquad\qquad\qquad\qquad + \Phi_{n-1} - \Phi_n
\end{aligned}$$

Binary Counter: Potential Method

- **Potential function:**

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

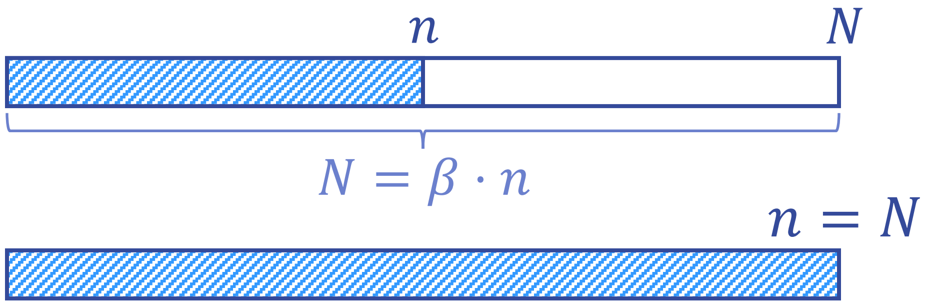
Claim: Amortized append cost is $O(1)$

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: $t_i = \begin{cases} 1 & \text{if } n < N \\ \beta \cdot N & \text{if } n = N \end{cases}$



Φ small $(\Phi = 0)$

Φ large $(\Phi \geq \beta N)$

At the start:

$N = N_0$
$n = 0$

We need: $\Phi \geq 0$

Let's try: $\Phi(n, N) = c \cdot (\beta n - N) \;+\; c \cdot N_0$

$$c(\beta N - N) \geq \beta N$$
$$c(\beta - 1) \geq \beta$$
$$c \geq \frac{\beta}{\beta - 1}$$

$$\Phi(n, N) = \frac{\beta}{\beta - 1} \cdot (\beta n - N + N_0)$$

Dynamic Array: Amortized Cost

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

Potential function:

$$\Phi(n, N) = \frac{\beta}{\beta - 1} \cdot (\beta n - N + N_0)$$

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

Case 1 ($n < N$): $a_i = 1 + \frac{\beta}{\beta-1} \cdot (\beta(n+1) - \beta n) = 1 + \frac{\beta^2}{\beta-1}$

Case 2 ($n = N$): $t_i = \beta n = \beta N$

$$a_i = \beta N + \frac{\beta}{\beta-1} \cdot [\beta(N+1) - \beta N - (\beta N - N)]$$
$$= \beta N + \frac{\beta^2}{\beta-1} - \frac{\beta}{\beta-1} \cdot (\beta - 1)N = \frac{\beta^2}{\beta-1}$$

Amortized cost $\leq 1 + \frac{\beta^2}{\beta-1}$