Chapter 4
Amortized Analysis

Algorithm Theory
WS 2017/18

Fabian Kuhn

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
- $\boldsymbol{T} \coloneqq \boldsymbol{t_1} + \boldsymbol{t_2} + \cdots + \boldsymbol{t_n}$: 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

running time for a typical input

random

- 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

- $S.\mathrm{push}(x)$: 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?

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.,

$$T = \sum_{i=1}^{n} t_i \leq \sum_{i=1}^{n} a_i \quad (+\, C)$$

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**

Amortized Cost

$$T = \sum_i t_i \le \sum_i a_i$$

Actual Cost of Augmented Stack Operations

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

<u>n operations</u>

$p \le n$ push ops. total push cost $\le c \cdot p$

total # del. elements $\le p$ total pop/multipop cost $\le c \cdot p$

total cost $\le 2 \cdot c \cdot p$

avg. cost per op. $\le \frac{2cp}{n} \le \frac{2cp}{p} = \underline{2c}$

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	0001**1**	1
4	00**100**	3
5	0010**1**	1
6	001**10**	2
7	0011**1**	1
8	0**1000**	4
9	0100**1**	1
10	010**10**	2
11	0101**1**	1
12	01**100**	3
13	0110**1**	1

Accounting Method

Observation:

- Each increment flips exactly one 0 into a 1

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

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					0
1	0 0 0 0 **1**	1	1	0	2	1
2	0 0 0 **1** **0**	2	1	1	2	1
3	0 0 0 1 **1**	1	1	0	2	2
4	0 0 **1** **0** **0**	3	1	2	2	1
5	0 0 1 0 **1**	1	1	0	2	2
6	0 0 1 **1** **0**	2	1	1	2	2
7	0 0 1 1 **1**	1	1	0	2	3
8	0 **1** **0** **0** **0**	4	1	3	2	1
9	0 1 0 0 **1**	1	1	0	2	2
10	0 1 0 **1** **0**	2	1	1	2	2

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

Credit: $x \geq 0$

$C \leq A$

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

  initial potential $\Phi_0 \geq 0$
  typically $\Phi_0 = 0$

- **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:

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

 $\phi_i \geq 0$

Operation i:

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

Overall cost:

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

$$\boxed{\sum a_i = \sum t_i + \phi_n - \phi_0}$$

$$\sum_{i=1}^{n} a_i = t_1 - \phi_0 + \phi_1 - \phi_1 + \phi_2 - \phi_2 + \phi_3 \ldots - \phi_{n-2} + \phi_{n-1} - \phi_{n-1} + \phi_n$$
$$+ t_2$$
$$+ t_3$$
$$\vdots$$
$$+ t_{n-1}$$
$$+ t_n$$

$$\phi_n \geq \phi_0$$

$$\sum t_i \leq \sum a_i$$

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"

\# elem : n

size : N

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 $O(1)$

$n < N$

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

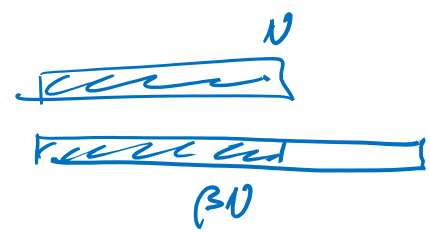
Example 3: Dynamic Array

Notation:

- n: number of elements stored
- N: current size of array (before operation)

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

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

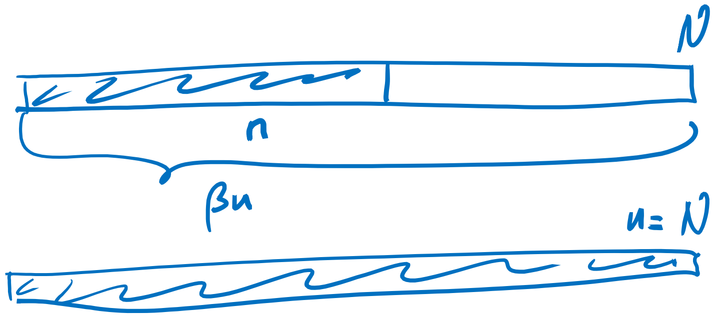


Potential function Φ?

- 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 $\quad (\phi = 0)$

ϕ large $\quad (\phi \geq \beta N)$

at the beginning:

$N = N_0$

$n = 0$

$$\phi(n, N) = c \cdot (\beta n - N) + c N_0$$

$$c(\beta N - N) \geq \beta N$$

$$c(\beta - 1) \geq \beta$$

$$c \geq \frac{\beta}{\beta - 1}$$

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

Dynamic Array: Amortized Cost $a_i = t_i + \phi_i - \phi_{i-1}$

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

$$\phi(n,N) = \frac{\beta}{\beta-1}\left(\beta n - N + N_0\right)$$

amortized cost a_i

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

case 2 ($n=N$):

$$a_i = \beta N + \left[\underbrace{\frac{\beta}{\beta-1}\left(\beta(N+1) - \beta N\right) - \frac{\beta}{\beta-1}\left(\beta N - N\right)}_{\frac{\beta^2}{\beta-1} - \underbrace{\frac{\beta}{\beta-1}(\beta-1)N}_{\beta N}}\right] = \frac{\beta^2}{\beta-1}$$

$$\boxed{\text{amortized cost: } 1 + \frac{\beta^2}{\beta-1}}$$