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)

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

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

Complexity of Stack Operations

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

Additional Operation

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

- What is the amortized complexity of these operations?

push, push, pop, push, ..., multipop(3), ... - - multipop(1000), ...

n ops.

k k

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**

Augmented Stack: Amortized Cost

Amortized Cost

$\leq 2c \cdot \#op$

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

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 $t_i \leq c \cdot k$

n operations

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

total # elem. deleted $\leq p$ total pop/multipop cost: $\leq c \cdot p$

total cost $\leq 2cp$ total # ops $\geq p$

avg. cost per op $\leq 2c$ amortized cost of each op i as $a_i := 2c$

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

0000 11 11
1 00 00

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

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

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

Credit: $x \ge 0$

$C \le A$

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

often:
S_0 initial state (empty data str.)

$\Phi_0 := \Phi(S_0) = 0$

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

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

1 8 3

Potential Function Method

$\underline{\sum t_i \leq \sum a_i} + \underline{\Phi_0}$

Operation i:

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

Overall cost:

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

$\left(\substack{0 \\ \shortparallel}\right) \quad \left(\substack{0 \\ \geq}\right)$

$\underline{\sum a_i = \sum t_i + \Phi_n - \Phi_0}$

$$\sum_{i=1}^{n} a_i = t_1 - \Phi_0 + \cancel{\Phi_1} + t_2 - \cancel{\Phi_1} + \cancel{\Phi_2} + t_3 - \cancel{\Phi_2} + \cancel{\Phi_3} + \dots - \cancel{\Phi_{n-2}} + \cancel{\Phi_{n-1}} + t_{n-1} - \cancel{\Phi_{n-1}} + \Phi_n + t_n$$

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

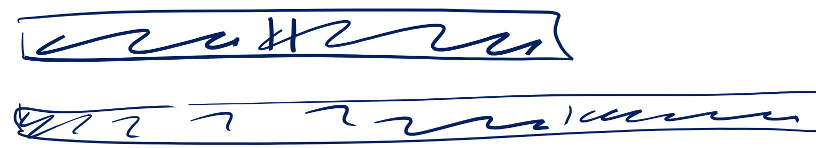
- Potential difference: $\Phi_i - \Phi_{i-1} = 1 - (t_i - 1) = 2 - t_i$

- Amortized cost: $a_i = t_i + \Phi_i - \Phi_{i-1} = 2$

Example 3: Dynamic Array

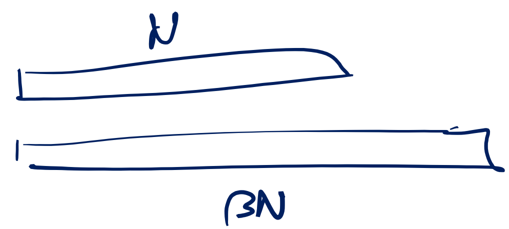
- How to create an array where the size dynamically adapts to the number of elements stored?
  - e.g., Java "ArrayList" or Python "list"

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

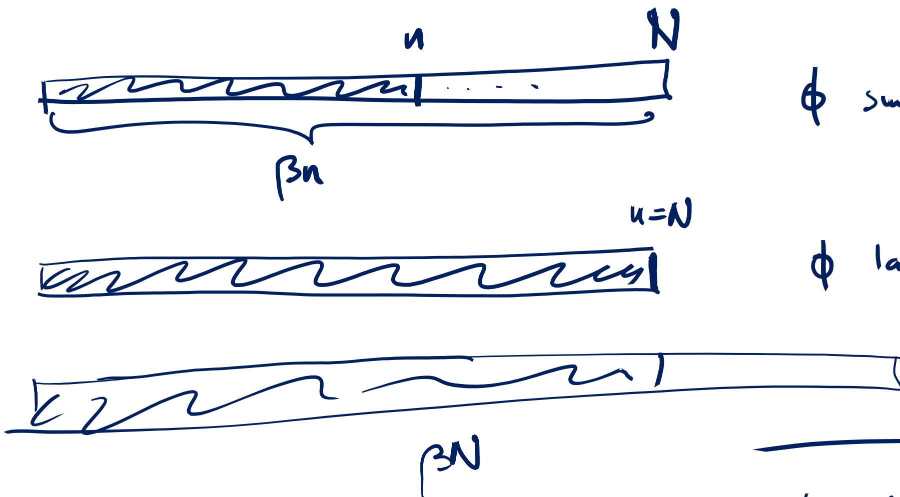
Claim: Amortized append cost is $O(1)$

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 $(\phi = 0)$

at the beginning
$N = N_0$
$n = 0$

ϕ large $(\phi \geq \beta N)$

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

$c(\beta n - N)$

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

$\Downarrow$

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

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

Dynamic Array: Amortized Cost

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

$$\phi(n, N) = \frac{\beta}{\beta - 1}(\beta n - N + N_0) \geq 0$$

$$a_i = t_i + \phi_i - \phi_{i-1}$$

$$a_i = \begin{cases} 1 + \frac{\beta^2}{\beta - 1} & \text{if } n < N \\ \beta N + \left[\frac{\beta}{\beta-1}\big(\beta(N+1) - \beta N\big) - \frac{\beta}{\beta-1}(\beta N - N) \right] & n = N \end{cases}$$

$$\frac{\beta^2}{\beta-1} + \underbrace{\frac{\beta}{\beta-1}(N - \beta N)}_{-\beta N} = \frac{\beta^2}{\beta-1}$$