



Chapter 4

Amortized Analysis

Algorithm Theory
WS 2015/16

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., $t_i \in [1, \underline{O(i)}]$)
- 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 random input
- Amortized worst case

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

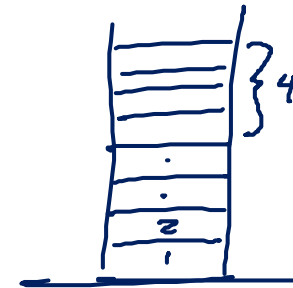
Example 1: Augmented Stack

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

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

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

n ops

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.,

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

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 $t_i \leq \underline{c}$
- $S.\text{multipop}(k)$: actual cost $t_i \leq \underline{c \cdot k}$

n operations

$\leq n$ push operations

push cost $\leq c \cdot n$

total # of elements deleted
using pop or multipop

$\leq n$ } total cost for
pop/multipop $\leq c \cdot n$

total cost $\leq \underline{2 \cdot c \cdot n}$

amortized cost $a_i = 2c$

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	000 11	→ 1
4	00 100	↗ 3
5	00 101	↖ 1
6	00 110	2
7	00 111	1
8	01000	4
9	01001	1
10	01010	2
11	01011	1
12	01100	3
13	01101	1

9 11111111 - - - 11
 10 - - - . 0

Accounting Method

Observation:

- Each increment flips exactly one 0 into a 1

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

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
	00000					0
1	0000 <u>1</u>	1	1	0	②	1
2	000 <u>1</u> 0	2	1	1	2)	1
3	0001 <u>1</u>	1	1	0	2	2
4	00 <u>1</u> 00	3	1	2	2	1
5	0010 <u>1</u>	1	1	0	2	2
6	001 <u>1</u> 0	2	1	1	2	2
7	0011 <u>1</u>	1	1	0	2	3
8	0 <u>1</u> 000	4	1	3	2	1
9	0100 <u>1</u>	1	1	0	2	2
10	010 <u>1</u> 0	2	1	1	2	②

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

$C \leq A$

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

- **Operation i :**

S_{i-1} S_i

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

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

Potential Function Method

Operation i :

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

Overall cost:

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

$\& \Phi_0 = 0$

$$\sum_{i=1}^n a_i = \begin{array}{l} (t_1) - \Phi_0 + \Phi_1 \\ + t_2 \quad - \Phi_1 + \Phi_2 \\ + t_3 \quad - \Phi_2 + \Phi_3 \\ \vdots \\ + t_n \quad - \Phi_{n-1} + \Phi_n \end{array}$$

$\sum a_i = \sum t_i + \Phi_n - \Phi_0$
 $\Phi_n \geq \Phi_0$

Binary Counter: Potential Method

- Potential function:

$$\Phi \geq 0$$

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 :

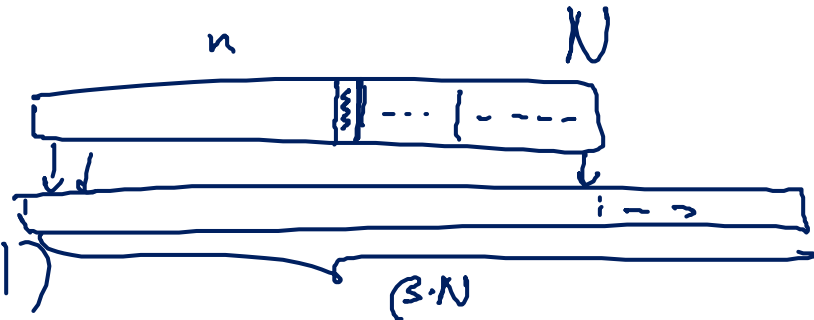
- 1 flip from 0 to 1
- $t_i - 1$ flips from 1 to 0

- 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 $N_0 = O(1)$
- 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 cost, 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: $t_i = \begin{cases} 1 & \text{if } n < N \\ \beta \cdot N & \text{if } n = N \end{cases}$

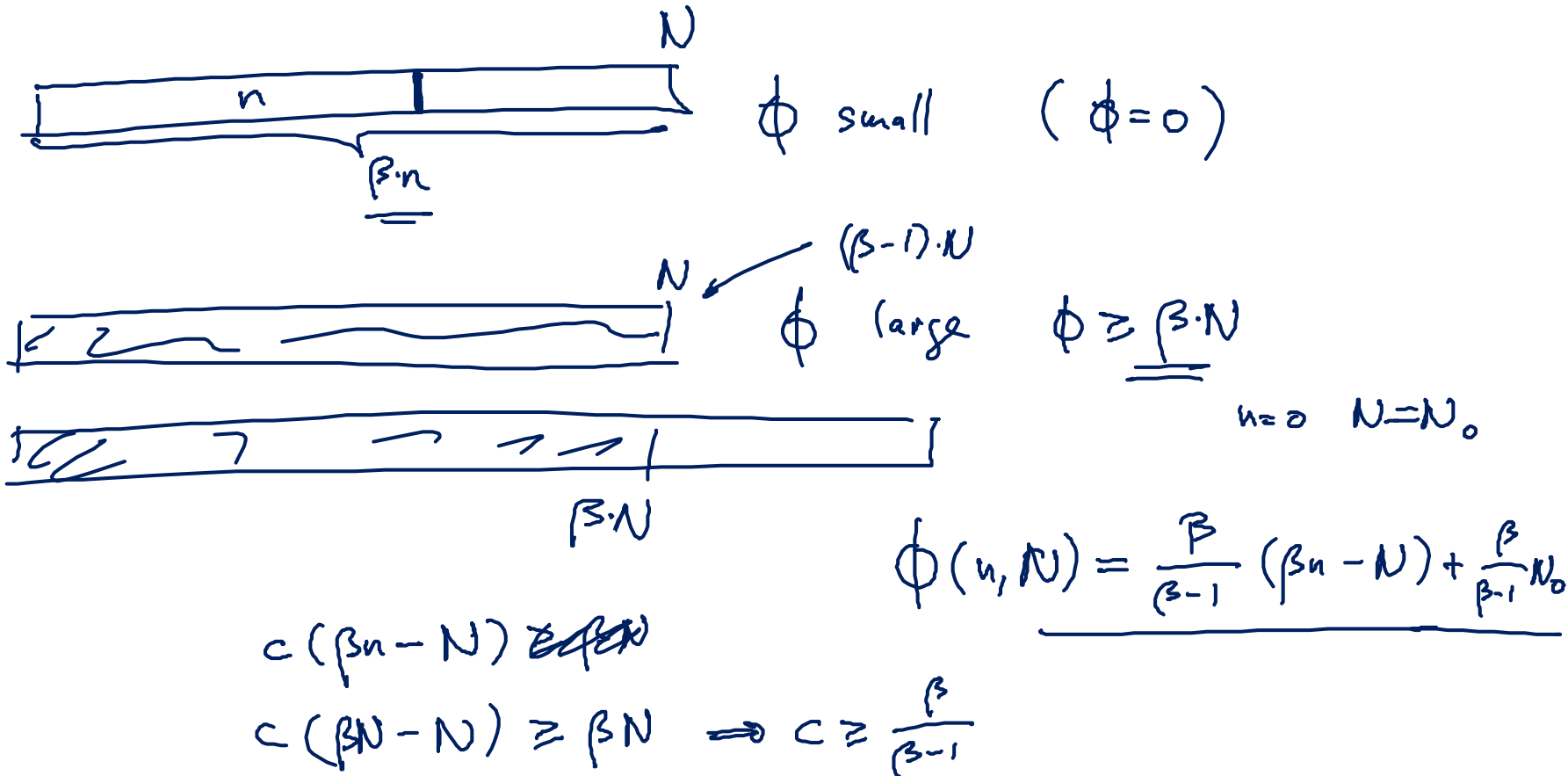
Claim: Amortized append cost is $O(1)$

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



ϕ small ($\phi = 0$)

ϕ large $\phi \geq \underline{\beta \cdot N}$

$n = 0 \quad N = N_0$

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

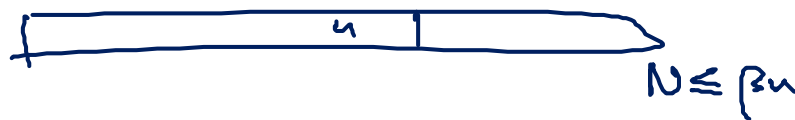
$c(\beta n - N) \leq c \beta n$

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

Dynamic Array: Amortized Cost

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

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



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

$$a_i = \left\{ \begin{array}{l} 1 + \frac{\beta^2}{\beta-1} \\ \beta \cdot N + \left[\frac{\beta}{\beta-1} (\beta N - \beta N) - \frac{\beta}{\beta-1} (\beta N - N) \right] \end{array} \right. \quad (n = N)$$

$$\frac{\beta}{\beta-1} (-(\beta-1)N) = 0$$