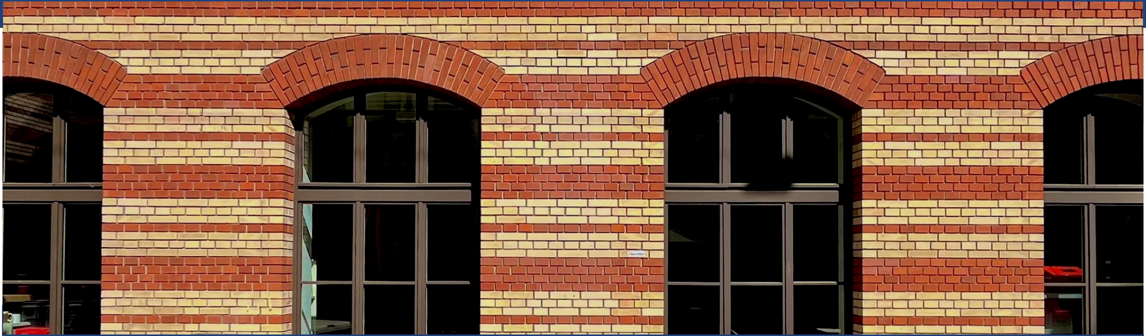


Datastructures and Algorithms

Amortized Analysis, Code Example "Dynamically Sized Array"

Adel Gavranović — ETH Zürich — 2025

Overview

50?

L

- Learning Objectives
- Entry Quiz
- Repetition theory
    - Amortized Analysis
- Code-Example: Dynamically Sized Array
- Tips for **code** expert
- Past Exam Questions

n.ethz.ch/~agavranovic

Material

Webpage

Mail

1. Follow-up

Follow-up from last session

Slide 15 "$\log(n!) \in \Theta(n \log n)$**"**

Follow-up from last session

Slide 15 "$\log(n!) \in \Theta(n \log n)$"

- I double checked and it seems to be true
- Proof via Stirling's approximation of $n!$

Follow-up from last session

Slide 15 "$\log(n!) \in \Theta(n \log n)$"

- I double checked and it seems to be true
- Proof via Stirling's approximation of $n!$

Slide 19 "Analyse the running time of (recursive) Functions"

Follow-up from last session

Slide 15 "$\log(n!) \in \Theta(n \log n)$"

- I double checked and it seems to be true
- Proof via Stirling's approximation of $n!$

Slide 19 "Analyse the running time of (recursive) Functions"

- If we have enough time left, we might revisit this

Follow-up from last session

Slide 15 "$\log(n!) \in \Theta(n \log n)$"

- I double checked and it seems to be true
- Proof via Stirling's approximation of $n!$

Slide 19 "Analyse the running time of (recursive) Functions"

- If we have enough time left, we might revisit this

Slide 39 "Master Method"

Follow-up from last session

Slide 15 "$\log(n!) \in \Theta(n \log n)$"

- I double checked and it seems to be true
- Proof via Stirling's approximation of $n!$

Slide 19 "Analyse the running time of (recursive) Functions"

- If we have enough time left, we might revisit this

Slide 39 "Master Method"

- If we have enough time left, we might revisit this too

Follow-up from last session

Slide 15 "$\log(n!) \in \Theta(n \log n)$"

- I double checked and it seems to be true
- Proof via Stirling's approximation of $n!$

Slide 19 "Analyse the running time of (recursive) Functions"

- If we have enough time left, we might revisit this

Slide 39 "Master Method"

- If we have enough time left, we might revisit this too

Slide 42 "Sorting Algorithms"

Follow-up from last session

Slide 15 "$\log(n!) \in \Theta(n \log n)$"

- I double checked and it seems to be true
- Proof via Stirling's approximation of $n!$

Slide 19 "Analyse the running time of (recursive) Functions"

- If we have enough time left, we might revisit this

Slide 39 "Master Method"

- If we have enough time left, we might revisit this too

Slide 42 "Sorting Algorithms"

- Who did have a look at it?

Slide 57 "Trade Offer regarding LaTeX"

Follow-up from last session

Slide 15 "$\log(n!) \in \Theta(n \log n)$"

- I double checked and it seems to be true
- Proof via Stirling's approximation of $n!$

Slide 19 "Analyse the running time of (recursive) Functions"

- If we have enough time left, we might revisit this

Slide 39 "Master Method"

- If we have enough time left, we might revisit this too

Slide 42 "Sorting Algorithms"

- Who did have a look at it?

Slide 57 "Trade Offer regarding LaTeX"

- Becuase we didn't get to everything last time, I deem it would be better to postpone this

2. Feedback regarding **code** expert

General things regarding **code** expert

General things regarding code expert

- Bonus open since Monday(?)
- XP needed to unlock it
- I'm correcting as quickly as I can, so you all can get started on it

Any questions regarding **code** expert on your part?

3. Learning Objectives

Objectives

- Understand the basics of the three *Amortized Analysis* methods
  - Aggregate Analysis
  - Account Method
  - Potential Method
- Be prepared for Double Ended Queue exercise on **code** expert

4. Summary

Getting on the same page

Getting on the same page

- What happened in the lectures since last time?

5. Entry Quiz

Quiz

Among a huge number (n) of students present, a price will be awarded to the student with the median Legi number. There is an argument what kind of algorithm shall be used to find this student. Mark the correct statements.

(1) In order to have a worst case runtime of $\mathcal{O}(n \log n)$, we use

- BubbleSort
- Selection Sort
- Mergesort
- Quicksort

Quiz

Among a huge number (n) of students present, a price will be awarded to the student with the median Legi number. There is an argument what kind of algorithm shall be used to find this student. Mark the correct statements.

(1) In order to have a worst case runtime of $\mathcal{O}(n \log n)$, we use

- BubbleSort
- Selection Sort
- Mergesort
- Quicksort

Quiz

Among a huge number (n) of students present, a price will be awarded to the student with the median Legi number. There is an argument what kind of algorithm shall be used to find this student. Mark the correct statements.

(2) We use Quickselect with random pivot choice. Then we have

- a worst case running time of $\mathcal{O}(n \log n)$
- a worst case running time of $\mathcal{O}(n)$
- an expected running time of $\mathcal{O}(\log n)$
- an expected running time of $\mathcal{O}(n)$

Quiz

Among a huge number (n) of students present, a price will be awarded to the student with the median Legi number. There is an argument what kind of algorithm shall be used to find this student. Mark the correct statements.

(2) We use Quickselect with random pivot choice. Then we have

- a worst case running time of $\mathcal{O}(n \log n)$
- a worst case running time of $\mathcal{O}(n)$
- an expected running time of $\mathcal{O}(\log n)$
- an expected running time of $\mathcal{O}(n)$ ☺

6.1. Amortized Analysis

Three Methods

Three Methods

- Aggregate analysis
- Account Method
- Potential Method

Example: simple multi-set

Supports operations `Insert` and `Find`.

Example: simple multi-set

Supports operations `Insert` and `Find`.

Idea:

- Collection of arrays A_i with Length 2^i

Example: simple multi-set

Supports operations `Insert` and `Find`.

Idea:

- Collection of arrays A_i with Length 2^i
- Every array is either entirely empty or entirely full and stores items in a sorted order

Example: simple multi-set

Supports operations `Insert` and `Find`.

Idea:

- Collection of arrays A_i with Length 2^i
- Every array is either entirely empty or entirely full and stores items in a sorted order
- Between the arrays there is no further relationship

Example: simple multi-set

Supports operations `Insert` and `Find`.

Idea:

- Collection of arrays A_i with Length 2^i
- Every array is either entirely empty or entirely full and stores items in a sorted order
- Between the arrays there is no further relationship

Data $\{1, 8, 10, 18, 20, 24, 36, 48, 50, 75, 99\}$, $n = 11$

$$\begin{aligned} A_0 &: \quad [50] \\ A_1 &: \quad [8, 99] \\ A_2 &: \quad \emptyset \\ A_3 &: \quad [1, 10, 18, 20, 24, 36, 48, 75] \end{aligned}$$

We use 0-indexing, such that for the lengths $|A_i| = 2^i$.

Example: simple multi-set

For any $n \in \mathbb{N}$, we can store exactly n elements in our multi set, without partially-filled arrays. Intuition: binary representation of n.

$$\begin{array}{rcccccccc}
\text{\#elements in multi-set} & = & |A_k| & + & |A_{k-1}| & + & \dots & + & |A_0| \\
 & = & b_k 2^k & + & b_{k-1} 2^{k-1} & + & \dots & + & b_0 2^0 \\
 & = & (b_k & & b_{k-1} & & \dots & & b_0)_2
\end{array}$$

Where $b_i = 0$ if $|A_i| = 0$, and 1 if $|A_i| = 2^i$.

Example: simple multi-set

Data $\{1, 8, 10, 18, 20, 24, 36, 48, 50, 75, 99\}$, $n = 11$

$$\begin{aligned} A_0&: \quad [50] \\ A_1&: \quad [8, 99] \\ A_2&: \quad \emptyset \\ A_3&: \quad [1, 10, 18, 20, 24, 36, 48, 75] \end{aligned}$$

Algorithm `Find`:

Example: simple multi-set

Data $\{1, 8, 10, 18, 20, 24, 36, 48, 50, 75, 99\}$, $n = 11$

$$\begin{aligned} A_0&: \quad [50] \\ A_1&: \quad [8, 99] \\ A_2&: \quad \emptyset \\ A_3&: \quad [1, 10, 18, 20, 24, 36, 48, 75] \end{aligned}$$

Algorithm `Find`: Perform a binary search on each array
Worst-case Runtime:

Example: simple multi-set

Data $\{1, 8, 10, 18, 20, 24, 36, 48, 50, 75, 99\}$, $n = 11$

$$\begin{aligned} A_0&: \quad [50] \\ A_1&: \quad [8, 99] \\ A_2&: \quad \emptyset \\ A_3&: \quad [1, 10, 18, 20, 24, 36, 48, 75] \end{aligned}$$

Algorithm `Find`: Perform a binary search on each array
Worst-case Runtime: $\Theta(\log^2 n)$,

Example: simple multi-set

Data $\{1, 8, 10, 18, 20, 24, 36, 48, 50, 75, 99\}$, $n = 11$

$$\begin{array}{ll} A_0: & [50] \\ A_1: & [8, 99] \\ A_2: & \emptyset \\ A_3: & [1, 10, 18, 20, 24, 36, 48, 75] \end{array}$$

Algorithm `Find`: Perform a binary search on each array
Worst-case Runtime: $\Theta(\log^2 n)$,

$$\log 1 + \log 2 + \log 4 + \cdots + \log 2^k = \sum_{i=0}^{k} \log_2 2^i = \frac{k \cdot (k+1)}{2} \in \Theta(\log^2 n).$$

$(k = \lfloor \log_2 n \rfloor)$

Example: simple multi-set

Algorithm `Insert(x)`:

Example: simple multi-set

Algorithm `Insert(x)`:

- New array $A'_0 \leftarrow [x]$, $i \leftarrow 0$

Example: simple multi-set

Algorithm `Insert(x)`:

- New array $A'_0 \leftarrow [x]$, $i \leftarrow 0$
- while $A_i \neq \emptyset$, set A'_{i+1} =Merge(A_i, A'_i), $A_i \leftarrow \emptyset$, $i \leftarrow i+1$

Example: simple multi-set

Algorithm `Insert(x)`:

- New array $A'_0 \leftarrow [x]$, $i \leftarrow 0$
- while $A_i \neq \emptyset$, set A'_{i+1} =Merge(A_i, A'_i), $A_i \leftarrow \emptyset$, $i \leftarrow i+1$
- Set $A_i \leftarrow A'_i$

Example: simple multi-set

Algorithm `Insert(x)`:

- New array $A'_0 \leftarrow [x]$, $i \leftarrow 0$
- while $A_i \neq \emptyset$, set A'_{i+1} =Merge(A_i, A'_i), $A_i \leftarrow \emptyset$, $i \leftarrow i+1$
- Set $A_i \leftarrow A'_i$

Insert(11)

Pre-insert

A_0: $[50]$
A_1: $[8, 99]$
A_2: $\emptyset$
A_3: $[1, 10, 18, \ldots, 75]$

Example: simple multi-set

Algorithm `Insert(x)`:

- New array $A'_0 \leftarrow [x]$, $i \leftarrow 0$
- while $A_i \neq \emptyset$, set A'_{i+1} =Merge(A_i, A'_i), $A_i \leftarrow \emptyset$, $i \leftarrow i+1$
- Set $A_i \leftarrow A'_i$

Insert(11)

	Pre-insert	Temporary
A_0:	$[50]$	
A_1:	$[8, 99]$	
A_2:	$\emptyset$	
A_3:	$[1, 10, 18, \ldots, 75]$	

$\leftarrow$

Example: simple multi-set

Algorithm `Insert(x)`:

- New array $A'_0 \leftarrow [x]$, $i \leftarrow 0$
- while $A_i \neq \emptyset$, set A'_{i+1} =Merge(A_i, A'_i), $A_i \leftarrow \emptyset$, $i \leftarrow i+1$
- Set $A_i \leftarrow A'_i$

Insert(11)

	Pre-insert		Temporary
A_0:	$[50]$	A'_0:	$[11]$
A_1:	$[8, 99]$		
A_2:	$\emptyset$		
A_3:	$[1, 10, 18, \ldots, 75]$		

$\leftarrow$

Example: simple multi-set

Algorithm `Insert(x)`:

- New array $A'_0 \leftarrow [x]$, $i \leftarrow 0$
- while $A_i \neq \emptyset$, set A'_{i+1} =Merge(A_i, A'_i), $A_i \leftarrow \emptyset$, $i \leftarrow i+1$
- Set $A_i \leftarrow A'_i$

Insert(11)

	Pre-insert		Temporary
A_0:	$[50]$	A'_0:	$[11]$
A_1:	$[8, 99]$	A'_1:	$[11, 50]$
A_2:	$\emptyset$		
A_3:	$[1, 10, 18, \ldots, 75]$		

Example: simple multi-set

Algorithm `Insert(x)`:

- New array $A'_0 \leftarrow [x]$, $i \leftarrow 0$
- while $A_i \neq \emptyset$, set A'_{i+1} =Merge(A_i, A'_i), $A_i \leftarrow \emptyset$, $i \leftarrow i + 1$
- Set $A_i \leftarrow A'_i$

Insert(11)

	Pre-insert		Temporary
A_0:	$[50]$	A'_0:	$[11]$
A_1:	$[8, 99]$	A'_1:	$[11, 50]$
A_2:	$\emptyset$	A'_2:	$[8, 11, 50, 99]$
A_3:	$[1, 10, 18, \ldots, 75]$		

$\leftarrow$

Example: simple multi-set

Algorithm `Insert(x)`:

- New array $A'_0 \leftarrow [x]$, $i \leftarrow 0$
- while $A_i \neq \emptyset$, set A'_{i+1} =Merge(A_i, A'_i), $A_i \leftarrow \emptyset$, $i \leftarrow i+1$
- Set $A_i \leftarrow A'_i$

Insert(11)

Pre-insert		Temporary			Post-insert	
A_0:	$[50]$	A'_0:	$[11]$	$\Longrightarrow$	A_0:	$\emptyset$
A_1:	$[8, 99]$	A'_1:	$[11, 50]$		A_1:	$\emptyset$
A_2:	$\emptyset$	A'_2:	$[8, 11, 50, 99]$		A_2:	$[8, 11, 50, 99]$
A_3:	$[1, 10, 18, \ldots, 75]$				A_3:	$[1, 10, 18, \ldots, 75]$

Costs insert

In the following example: $n = 2^k$, $k = \log_2 n$

Costs insert

In the following example: $n = 2^k$, $k = \log_2 n$

Assumption: creating new array A'_i with length 2^i (and, for $i > 0$ subsequent merge of A'_{i-1} and A_{i-1}) has costs $\Theta(2^i)$

Costs insert

In the following example: $n = 2^k$, $k = \log_2 n$

Assumption: creating new array A'_i with length 2^i (and, for $i > 0$ subsequent merge of A'_{i-1} and A_{i-1}) has costs $\Theta(2^i)$

In the worst case, inserting an element into the data structure provides $\log_2 n$ such operations.

Costs insert

In the following example: $n = 2^k$, $k = \log_2 n$

Assumption: creating new array A'_i with length 2^i (and, for $i > 0$ subsequent merge of A'_{i-1} and A_{i-1}) has costs $\Theta(2^i)$

In the worst case, inserting an element into the data structure provides $\log_2 n$ such operations.
$\Rightarrow$ **Worst-case Costs Insert**:

$$\sum_{i=0}^{k} 2^i = 2^{k+1} - 1 \in \Theta(n).$$

$\leftarrow$

Aggregate analysis

Level	Costs	Example Array
0	1	$[*]$
1	2	$[*, *]$
2	4	$[*, *, *, *]$
3	8	$\emptyset$
4	16	$[*, *, *, *, *, *, *, *, *, *, *, *, *, *, *, *]$

Aggregate analysis

Level	Costs	Example Array
0	1	$[*]$
1	2	$[*, *]$
2	4	$[*, *, *, *]$
3	8	$\emptyset$
4	16	$[*, *, *, *, *, *, *, *, *, *, *, *, *, *, *, *]$

Observation: Starting with an empty container, an insertion sequence reaches level 0 each time, level 1 (with costs 2) every second time, level 2 (with costs 4) every fourth time, etc.

←

Aggregate analysis

Level	Costs	Example Array
0	1	$[*]$
1	2	$[*, *]$
2	4	$[*, *, *, *]$
3	8	$\emptyset$
4	16	$[*, *, *, *, *, *, *, *, *, *, *, *, *, *, *, *]$

Observation: Starting with an empty container, an insertion sequence reaches level 0 each time, level 1 (with costs 2) every second time, level 2 (with costs 4) every fourth time, etc.

- Total costs: $1 \cdot \frac{n}{1} + 2 \cdot \frac{n}{2} + 4 \cdot \frac{n}{4} + \cdots + 2^k \cdot \frac{n}{2^k} =$

Aggregate analysis

Level	Costs	Example Array
0	1	$[*]$
1	2	$[*, *]$
2	4	$[*, *, *, *]$
3	8	$\emptyset$
4	16	$[*, *, *, *, *, *, *, *, *, *, *, *, *, *, *, *]$

Observation: Starting with an empty container, an insertion sequence reaches level 0 each time, level 1 (with costs 2) every second time, level 2 (with costs 4) every fourth time, etc.

- Total costs: $1 \cdot \frac{n}{1} + 2 \cdot \frac{n}{2} + 4 \cdot \frac{n}{4} + \cdots + 2^k \cdot \frac{n}{2^k} = (k+1)n$
  This is in $\Theta(n \log n)$ because $k = \log_2 n$.
- **Amortized cost per operation**: $\Theta((n \log n)/n) = \Theta(\log n)$.

←

Accounting method

Ideas?

Accounting method

- Every element i $(1 \leq i \leq n)$ pays $a_i = \log_2 n$ coins when it is inserted into the data structure.

Accounting method

- Every element i $(1 \leq i \leq n)$ pays $a_i = \log_2 n$ coins when it is inserted into the data structure.
- The element pays the allocation of the first array and every subsequent merge-step that can occur until the element has reached array A_{k+1} $(k = \lfloor \log_2 n \rfloor)$.

Accounting method

- Every element i $(1 \leq i \leq n)$ pays $a_i = \log_2 n$ coins when it is inserted into the data structure.
- The element pays the allocation of the first array and every subsequent merge-step that can occur until the element has reached array A_{k+1} $(k = \lfloor \log_2 n \rfloor)$.
- The account provides enough credit to pay for all Merge operations of the n elements.

Accounting method

- Every element i $(1 \leq i \leq n)$ pays $a_i = \log_2 n$ coins when it is inserted into the data structure.
- The element pays the allocation of the first array and every subsequent merge-step that can occur until the element has reached array A_{k+1} $(k = \lfloor \log_2 n \rfloor)$.
- The account provides enough credit to pay for all Merge operations of the n elements.

$\Rightarrow$ **Amortized costs** for insertion $\mathcal{O}(\log n)$

Potential method

Ideas?

Potential method

We know from the accounting method that **each element on the way to higher levels requires** $\log n$ **coins**, i.e. that an element on level i still needs to posess $k - i$ coins.

Potential method

We know from the accounting method that **each element on the way to higher levels requires** $\log n$ **coins**, i.e. that an element on level i still needs to posess $k - i$ coins. We use the **potential**

Potential method

We know from the accounting method that **each element on the way to higher levels requires** $\log n$ **coins**, i.e. that an element on level i still needs to posess $k - i$ coins. We use the **potential**

$$\Phi_j = \sum_{0 \leq i \leq k : A_i \neq \emptyset} (k - i) \cdot 2^i$$

Potential method

For the **change of the potential** $\Phi_j - \Phi_{j-1}$ we only have to consider the lower l levels that are occupied at time point $j-1$ (in analogy to the binary counter). Let l be the smallest index such that array A_l is empty.

Potential method

For the **change of the potential** $\Phi_j - \Phi_{j-1}$ we only have to consider the lower l levels that are occupied at time point $j-1$ (in analogy to the binary counter). Let l be the smallest index such that array A_l is empty.

After merging arrays $A_0 \ldots A_{l-1}$, array A_l is full and arrays $A_i (0 \le i < l)$ are now empty. Therefore:

Potential method

For the **change of the potential** $\Phi_j - \Phi_{j-1}$ we only have to consider the lower l levels that are occupied at time point $j-1$ (in analogy to the binary counter). Let l be the smallest index such that array A_l is empty.

After merging arrays $A_0 \ldots A_{l-1}$, array A_l is full and arrays $A_i (0 \leq i < l)$ are now empty. Therefore:

$$\Phi_j - \Phi_{j-1} = (k-l) \cdot 2^l - \sum_{i=0}^{l-1} (k-i) \cdot 2^i$$

←

Potential method

For the **change of the potential** $\Phi_j - \Phi_{j-1}$ we only have to consider the lower l levels that are occupied at time point $j-1$ (in analogy to the binary counter). Let l be the smallest index such that array A_l is empty.

After merging arrays $A_0 \ldots A_{l-1}$, array A_l is full and arrays $A_i (0 \leq i < l)$ are now empty. Therefore:

$$\Phi_j - \Phi_{j-1} = (k-l) \cdot 2^l - \sum_{i=0}^{l-1} (k-i) \cdot 2^i$$

Real costs:

$$t_j = \sum_{i=0}^{l} 2^i = 2^{l+1} - 1$$

Potential method

$$\Phi_j - \Phi_{j-1} = (k-l) \cdot 2^l - \sum_{i=0}^{l-1} (k-i) \cdot 2^i$$

Potential method

$$\begin{aligned}\Phi_j - \Phi_{j-1} &= (k-l) \cdot 2^l - \sum_{i=0}^{l-1} (k-i) \cdot 2^i \\ &= (k-l) \cdot 2^l - k \cdot (2^l - 1) + \sum_{i=0}^{l-1} i \cdot 2^i\end{aligned}$$

Potential method

$$
\begin{aligned}
\Phi_j - \Phi_{j-1} &= (k-l)\cdot 2^l - \sum_{i=0}^{l-1}(k-i)\cdot 2^i \\
&= (k-l)\cdot 2^l - k\cdot(2^l-1) + \sum_{i=0}^{l-1} i\cdot 2^i \\
&= (k-l)\cdot 2^l - k\cdot(2^l-1) + l\cdot 2^l - 2^{l+1} + 2
\end{aligned}
$$

←

Potential method

$$
\begin{aligned}
\Phi_j - \Phi_{j-1} &= (k-l)\cdot 2^l - \sum_{i=0}^{l-1}(k-i)\cdot 2^i \\
&= (k-l)\cdot 2^l - k\cdot(2^l-1) + \sum_{i=0}^{l-1} i\cdot 2^i \\
&= (k-l)\cdot 2^l - k\cdot(2^l-1) + l\cdot 2^l - 2^{l+1} + 2 \\
&= k - 2^{l+1} + 2
\end{aligned}
$$

$\leftarrow$

Potential method

$$
\begin{aligned}
\Phi_j - \Phi_{j-1} &= (k-l)\cdot 2^l - \sum_{i=0}^{l-1}(k-i)\cdot 2^i \\
&= (k-l)\cdot 2^l - k\cdot(2^l-1) + \sum_{i=0}^{l-1} i\cdot 2^i \\
&= (k-l)\cdot 2^l - k\cdot(2^l-1) + l\cdot 2^l - 2^{l+1} + 2 \\
&= k - 2^{l+1} + 2
\end{aligned}
$$

$$
\implies \Phi_j - \Phi_{j-1} + t_j = k - 2^{l+1} + 2 + 2^{l+1} - 1 = k + 1 \in \Theta(\log n)
$$

See CLRS Chapter 16.

$\sum i \cdot \lambda^i$

Always the same trick:

$\sum i \cdot \lambda^i$

Always the same trick:

$$\lambda \cdot \sum_{i=0}^{n} i \cdot \lambda^i - \sum_{i=0}^{n} i \cdot \lambda^i = \sum_{i=0}^{n} i \cdot \lambda^{i+1} - \sum_{i=0}^{n} i \cdot \lambda^i = \sum_{i=1}^{n+1} (i-1) \cdot \lambda^i - \sum_{i=0}^{n} i \cdot \lambda^i$$

$\sum i \cdot \lambda^i$

Always the same trick:

$$\lambda \cdot \sum_{i=0}^{n} i \cdot \lambda^i - \sum_{i=0}^{n} i \cdot \lambda^i = \sum_{i=0}^{n} i \cdot \lambda^{i+1} - \sum_{i=0}^{n} i \cdot \lambda^i = \sum_{i=1}^{n+1} (i-1) \cdot \lambda^i - \sum_{i=0}^{n} i \cdot \lambda^i$$

$$= n \cdot \lambda^{n+1} + \sum_{i=1}^{n} (i-1) \cdot \lambda^i - i \cdot \lambda = n \cdot \lambda^{n+1} - \sum_{i=1}^{n} \lambda^i$$

$\leftarrow$

$\sum i \cdot \lambda^i$

Always the same trick:

$$\lambda \cdot \sum_{i=0}^{n} i \cdot \lambda^i - \sum_{i=0}^{n} i \cdot \lambda^i = \sum_{i=0}^{n} i \cdot \lambda^{i+1} - \sum_{i=0}^{n} i \cdot \lambda^i = \sum_{i=1}^{n+1} (i-1) \cdot \lambda^i - \sum_{i=0}^{n} i \cdot \lambda^i$$

$$= n \cdot \lambda^{n+1} + \sum_{i=1}^{n} (i-1) \cdot \lambda^i - i \cdot \lambda = n \cdot \lambda^{n+1} - \sum_{i=1}^{n} \lambda^i$$

$$= n \cdot \lambda^{n+1} - \frac{\lambda^{n+1} - 1}{\lambda - 1} + 1$$

$\sum i \cdot \lambda^i$

Always the same trick:

$$\begin{aligned}
\lambda \cdot \sum_{i=0}^{n} i \cdot \lambda^i - \sum_{i=0}^{n} i \cdot \lambda^i &= \sum_{i=0}^{n} i \cdot \lambda^{i+1} - \sum_{i=0}^{n} i \cdot \lambda^i = \sum_{i=1}^{n+1} (i-1) \cdot \lambda^i - \sum_{i=0}^{n} i \cdot \lambda^i \\
&= n \cdot \lambda^{n+1} + \sum_{i=1}^{n} (i-1) \cdot \lambda^i - i \cdot \lambda = n \cdot \lambda^{n+1} - \sum_{i=1}^{n} \lambda^i \\
&= n \cdot \lambda^{n+1} - \frac{\lambda^{n+1} - 1}{\lambda - 1} + 1 \\
\implies (\lambda - 1) \cdot \sum_{i=0}^{n} i \cdot \lambda^i &= n \cdot \lambda^{n+1} - \frac{\lambda^{n+1} - 1}{\lambda - 1} + 1
\end{aligned}$$

$\sum i \cdot \lambda^i$

Always the same trick:

$$
\begin{aligned}
\lambda \cdot \sum_{i=0}^{n} i \cdot \lambda^i - \sum_{i=0}^{n} i \cdot \lambda^i &= \sum_{i=0}^{n} i \cdot \lambda^{i+1} - \sum_{i=0}^{n} i \cdot \lambda^i = \sum_{i=1}^{n+1} (i-1) \cdot \lambda^i - \sum_{i=0}^{n} i \cdot \lambda^i \\
&= n \cdot \lambda^{n+1} + \sum_{i=1}^{n} (i-1) \cdot \lambda^i - i \cdot \lambda = n \cdot \lambda^{n+1} - \sum_{i=1}^{n} \lambda^i \\
&= n \cdot \lambda^{n+1} - \frac{\lambda^{n+1} - 1}{\lambda - 1} + 1 \\
\implies (\lambda - 1) \cdot \sum_{i=0}^{n} i \cdot \lambda^i &= n \cdot \lambda^{n+1} - \frac{\lambda^{n+1} - 1}{\lambda - 1} + 1
\end{aligned}
$$

For $\lambda = 2$:

$\sum i \cdot \lambda^i$

Always the same trick:

$$\begin{aligned}
\lambda \cdot \sum_{i=0}^{n} i \cdot \lambda^i - \sum_{i=0}^{n} i \cdot \lambda^i &= \sum_{i=0}^{n} i \cdot \lambda^{i+1} - \sum_{i=0}^{n} i \cdot \lambda^i = \sum_{i=1}^{n+1} (i-1) \cdot \lambda^i - \sum_{i=0}^{n} i \cdot \lambda^i \\
&= n \cdot \lambda^{n+1} + \sum_{i=1}^{n} (i-1) \cdot \lambda^i - i \cdot \lambda = n \cdot \lambda^{n+1} - \sum_{i=1}^{n} \lambda^i \\
&= n \cdot \lambda^{n+1} - \frac{\lambda^{n+1} - 1}{\lambda - 1} + 1 \\
\implies (\lambda - 1) \cdot \sum_{i=0}^{n} i \cdot \lambda^i &= n \cdot \lambda^{n+1} - \frac{\lambda^{n+1} - 1}{\lambda - 1} + 1
\end{aligned}$$

For $\lambda = 2$:

$$\sum_{i=0}^{n} i \cdot 2^i = n \cdot 2^{n+1} - 2^{n+1} + 1 + 1 = (n-1) \cdot 2^{n+1} + 2$$

←

Quiz

```
void g(unsigned n){
  for (unsigned k = 1; k != n ; ++k){
     // what does the following code do?
    unsigned prev = k-1;
    for (unsigned num = k; num != 0; num /= 2){
      if (num % 2 != prev % 2)
        f();
      prev /= 2;
    }
  }
}
```

Q: Asymptotic number of calls of f?

Quiz

```
void g(unsigned n){
  for (unsigned k = 1; k != n ; ++k){
     // call f for all bits that toggle from k-1 to k
    unsigned prev = k-1;
    for (unsigned num = k; num != 0; num /= 2){
      if (num % 2 != prev % 2)
        f();
      prev /= 2;
    }
  }
}
```

Q: Asymptotic number of calls of f?

A: $\Theta(n)$ (Counting example from class).

Recap dynamically allocated memory

Important: Every `new` needs its `delete` and only one!

Recap dynamically allocated memory

Important: Every `new` needs its `delete` and only one!

Therefore “Rule of three”:

- constructor
- copy constructor
- destructor

Recap dynamically allocated memory

Important: Every `new` needs its `delete` and only one!

Therefore “Rule of three”:

- constructor
- copy constructor
- destructor

Being lazy “Rule of two”:

- never copy (unsafe)
- make copy constructor private (safe) or deleted

7. Code-Example: Dynamically Sized Array

Dynamically Sized Array

- Preparation for exercise "Double Ended Queue"
- We're going to implement our own `std::vector`

8. Tips for **code** expert

Tips for code expert Exercise 3

Tips for **code** expert Exercise 3

These are the ones due Thu 13.03.2025, 23:59 (in 2 days)

Recursive Function Analysis

- Please make sure to upload your Solution correctly
- Ideally as a PDF

Tips for code expert Exercise 3

These are the ones due Thu 13.03.2025, 23:59 (in 2 days)

Recursive Function Analysis

- Please make sure to upload your Solution correctly
- Ideally as a PDF

The Master Method

- Do we want to go over this again?

Tips for **code** expert Exercise 3

These are the ones due Thu 13.03.2025, 23:59 (in 2 days)

Recursive Function Analysis

- Please make sure to upload your Solution correctly
- Ideally as a PDF

The Master Method

- Do we want to go over this again?

Throwing Eggs

- Don't spend too much time on this

Tips for **code** expert Exercise 3

These are the ones due Thu 13.03.2025, 23:59 (in 2 days)

Recursive Function Analysis

- Please make sure to upload your Solution correctly
- Ideally as a PDF

The Master Method

- Do we want to go over this again?

Throwing Eggs

- Don't spend too much time on this

Mergesort

- Classic coding exercise

Tips for **code** expert Exercise 3

These are the ones due Thu 13.03.2025, 23:59 (in 2 days)

Recursive Function Analysis

- Please make sure to upload your Solution correctly
- Ideally as a PDF

The Master Method

- Do we want to go over this again?

Throwing Eggs

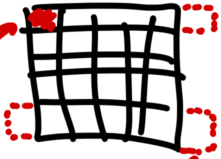
- Don't spend too much time on this

Mergesort

- Classic coding exercise

Matrices

- Go over the concepts of iterators and `const` again if needed



begin()

end()

dep. on. impl.

```
void funct(...) const {
    //...
}
```

Tips for **code** expert Exercise 4

These are the ones due Thu 20.03.2025, 23:59 (next week)

Task "Stable and In-Situ Sorting"

Tips for code expert Exercise 4

These are the ones due Thu 20.03.2025, 23:59 (next week)

Task "Stable and In-Situ Sorting"

- "...in their unmodified form..."

Tips for **code** expert Exercise 4

These are the ones due Thu 20.03.2025, 23:59 (next week)

Task "Stable and In-Situ Sorting"

- "...in their unmodified form..."

Task "Amortized Analysis: Dynamic Array"

Tips for code expert Exercise 4

These are the ones due Thu 20.03.2025, 23:59 (next week)

Task "Stable and In-Situ Sorting"

- "...in their unmodified form..."

Task "Amortized Analysis: Dynamic Array"

- Ottman/Widmayer, Chapter 3.3 (depending on version)
- Cormen et al, Chapter 17 (or 16 depending on version)

Tips for **code** expert Exercise 4

These are the ones due Thu 20.03.2025, 23:59 (next week)

Task "Stable and In-Situ Sorting"

- "…in their unmodified form…"

Task "Amortized Analysis: Dynamic Array"

- Ottman/Widmayer, Chapter 3.3 (depending on version)
- Cormen et al, Chapter 17 (or 16 depending on version)

Task "Double Ended Queue"

Tips for **code** expert Exercise 4

These are the ones due Thu 20.03.2025, 23:59 (next week)

Task "Stable and In-Situ Sorting"

- "...in their unmodified form..."

Task "Amortized Analysis: Dynamic Array"

- Ottman/Widmayer, Chapter 3.3 (depending on version)
- Cormen et al, Chapter 17 (or 16 depending on version)

Task "Double Ended Queue"

- Takes time – make sure to start early!
- Dynamic data types and memory management (fun!)
- By the way: *the name Double Ended Queue may be misleading because it suggests to be implemented with a linked list. This would make it hard, if not impossible, to fulfill the requirements stated above. Rather think of something like a vector and extend it with* `push_front()`

9. Past Exam Questions

No new ones this week. . .

Let's just go over the ones from last time. . .

10. Outro

General Questions?

See you next time!

Have a nice week!

□ error when accessing code example on homepage

□ no tools allowed

□ check if $c < 1$ or $c \in (0,1)$

follow-up for next week