2IL50 Data Structures

2023-24 Q3

Lecture 1: Introduction

EINDHOVEN UNIVERSITY OF TECHNOLOGY

2IL50
Introduction to Algorithms and Data Structures

2023-24 Q3

Lecture 1: Introduction

TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

Algorithms

Algorithm
a well-defined computational procedure that takes some value, or a set of values, as input and produces some value, or a set of values, as output

Algorithm
sequence of computational steps that transform the input into the output

Algorithms
exact problem statements, correct output, provable efficiency, guarantees in case of approximation

Algorithms research
design and analysis of algorithms and data structures for computational problems

Data structures

Data structure
a way to store and organize data to facilitate access and modifications

Abstract data type
describes functionality (which operations are supported)

Implementation
a way to realize the desired functionality

- how is the data stored (array, linked list, ...)
- which algorithms implement the operations

The course

Design and analysis of efficient algorithms for some basic computational problems

- Basic algorithm design techniques and paradigms
- Algorithms analysis: O-notation, recursions, ...
- Basic data structures
- Basic graph algorithms

Some administration first

before we really get started …

Organization

Lecturers:	**Prof. Dr. Bettina Speckmann**	b.speckmann@tue.nl	AUD 3
	Dr. Marcel Roeloffzen	m.j.m.roeloffzen@tue.nl	AUD 6
	Dr. Leonie Ryvkin	l.ryvkin@tue.nl	AUD 5

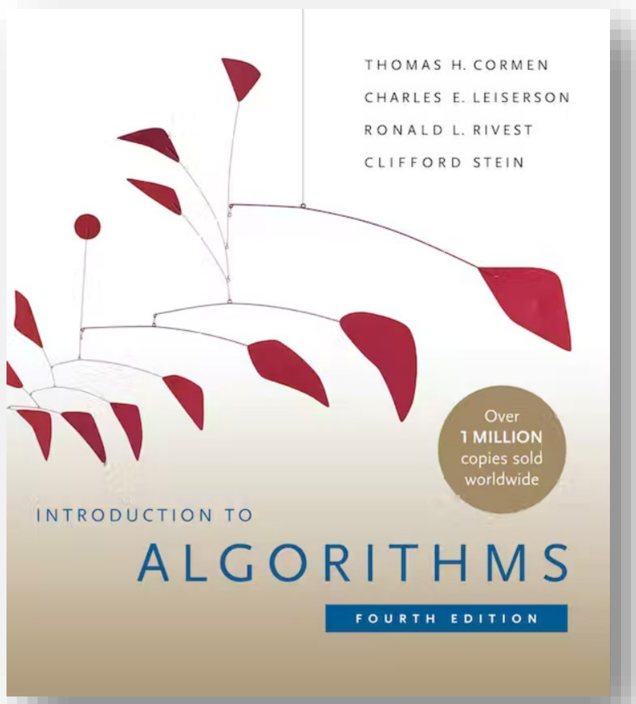
Use tag [2IL50] in the subject of your email
Better: use Slack DM

Web page: https://canvas.tue.nl/courses/25271

Enable notifications!

Book: T.H. Cormen, C.E. Leiserson, R.L. Rivest and C. Stein. **Introduction to Algorithms** (4th edition)

mandatory



Prerequisites

Being able to work with basic programming constructs

such as linked lists, arrays, loops ...

Being able to apply standard proving techniques

such as proof by induction, proof by contradiction ...

Being familiar with sums and logarithms

and the other mathematical background in the appendix of the textbook

Having successfully followed either 2IT80 (Discrete Structures) or JBI026 (Discrete Mathematics)

Data Structures has been a "deepening" course for many years already ...

Grading scheme 2IL50

1. 4 in-class interim tests, the best 3 of which count for 40% of the final grade
2. a written exam (closed book) which counts for the remaining 60% of the final grade

You must be present at the university to take the in-class interim tests. Your interim grade is the average of your best three interim test grades. Your interim grade must be at least a 5.5 to participate in the final exam. Since only the best 3 out of 4 in-class interim tests count towards your grade, there are no retakes and no alternative dates.

You must score at least a 5.0 on the exam and the weighted average of the interim grade and the final exam grade must round to a 6. If you score less than 5.0 on the final exam, then you will fail the course, regardless of your interim grade. Your grade will be the minimum of 5.0 and the grade you achieved. However, you are allowed to participate in the second chance exam. The grade of the second chance exam replaces the grade for the first exam, that is, your interim grade always counts for 40% of your grade.

In-class interim tests

1. 4 segments of 3 lectures each
2. each segment has a practice set and an assignment with 3 larger open questions
   a. practice set on Canvas - Schedule, solutions after first week
   b. assignment ans*
3. in-class interim test at end of segment ans* digital with safe exam browser (SEB)
   a. one open question from assignment
   b. several short open questions with material from this or earlier segments

must be present at the university to take in-class interim tests
your responsibility to register with the instructor/tutor
without registration test is not valid
room assignment via Canvas
"how to" see Canvas
latest version of SEB & laptop with at least 1 hour battery

Organization

Lectures	Monday 7+8	Section 1: AUD 3	Section 2: AUD 5	Section 3: AUD 6
	Wednesday 3+4	Section 1: AUD 3	Section 2: AUD 5	Section 3: AUD 6
Lectures are live, there are no streams and no recordings.				
Discussion groups	Mon 5 + Wed 2 *or*	*sign-up according to sections, location according to group name*		
	Mon 6 + Wed 1			
Led by tutors, two times 45 min, discuss practice sets and answer questions about the lecture.				
In-class interim tests	Thursday 9+10	*room assignment will be communicated beforehand*		
On-campus, digital, Ans with SEB, your laptop needs battery for ~1h.				
Slack workspace	https://join.slack.com/t/2il50ay23-24/signup			
All tutors, instructors, and teachers are present and will answer questions.				

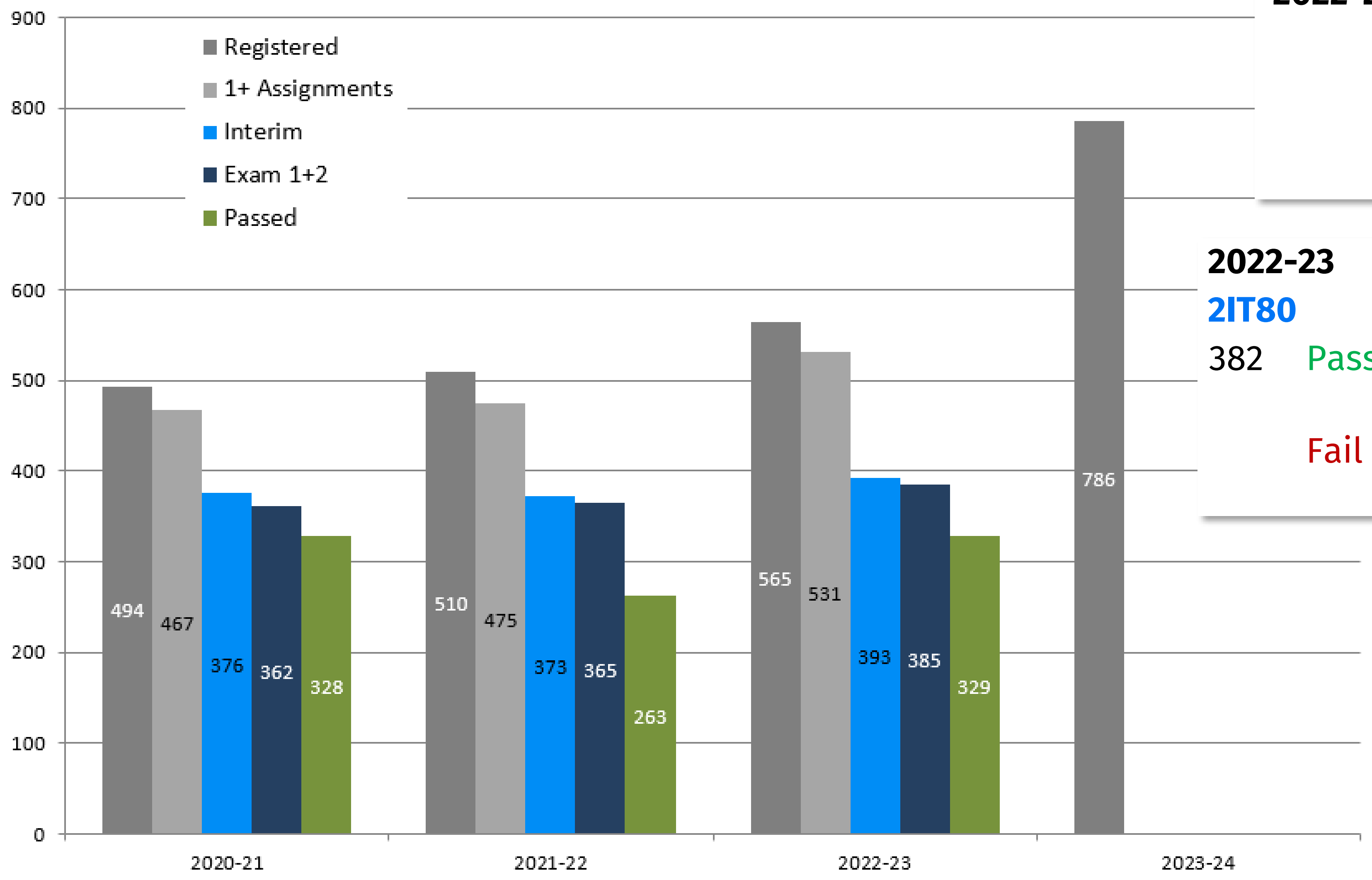
Discussion groups: by section

No-show without prior notification: removed from group

Schedule

Date		H	Topic	Slides	Material	Practice Sets
Feb	5	5+6	-----			
		7+8	Introduction		Chapters 1 and 2	Practice 1
	7	1+2	DG Introduction			
		3+4	Analysis of algorithms		Chapters 2, 3, and 4	
			Carnival			
	19	5+6	DG Analysis of algorithms			
		7+8	Heaps		Chapter 6	
	21	1+2	DG Heaps			
		3+4	Sorting in linear time		Chapter 8	
	22	9+10	Interim Test Segment 1			
	26	5+6	DG Sorting in linear time			
		7+8	QuickSort and selection		Chapters 7 and 9	
	28	1+2	DG QuickSort and selection			
		3+4	Hash tables		Chapter 11	
March	4	5+6	DG Hash tables			
		7+8	Binary search trees		Chapter 12 and 13	
	6	1+2	DG Binary search trees			
		3+4	Augmenting data structures			
	7	9+10	Interim Test Segment 2		Chapter 13 and 17	
	11	5+6	DG Augmenting data structures			
		7+8	Range searching		CG Book Chapter 5	
	13	1+2	DG Range searching			
		3+4	-----			
	14	9+10	Interim Test Segment 3			
	18	5+6	-----			
		7+8	Union-Find		Chapter 19	
	20	1+2	DG Union-Find			
		3+4	Elementary graph algorithms		Chapter 20	
	25	5+6	DG Elementary graph algorithms			
		7+8	Minimum spanning trees		Chapter 21	
	27	1+2	DG Minimum spanning trees			
		3+4	-----			
	28	9+10	Interim Test Segment 4			
April	1		Easter Monday			
	3	1+2	DG Practice exam			
		3+4	Recap "things you should know for the exam"			
			First exam week			
	17		Exam 13:30 - 16:30			
June	26		Resit 18:00 - 21:00			

Some statistics



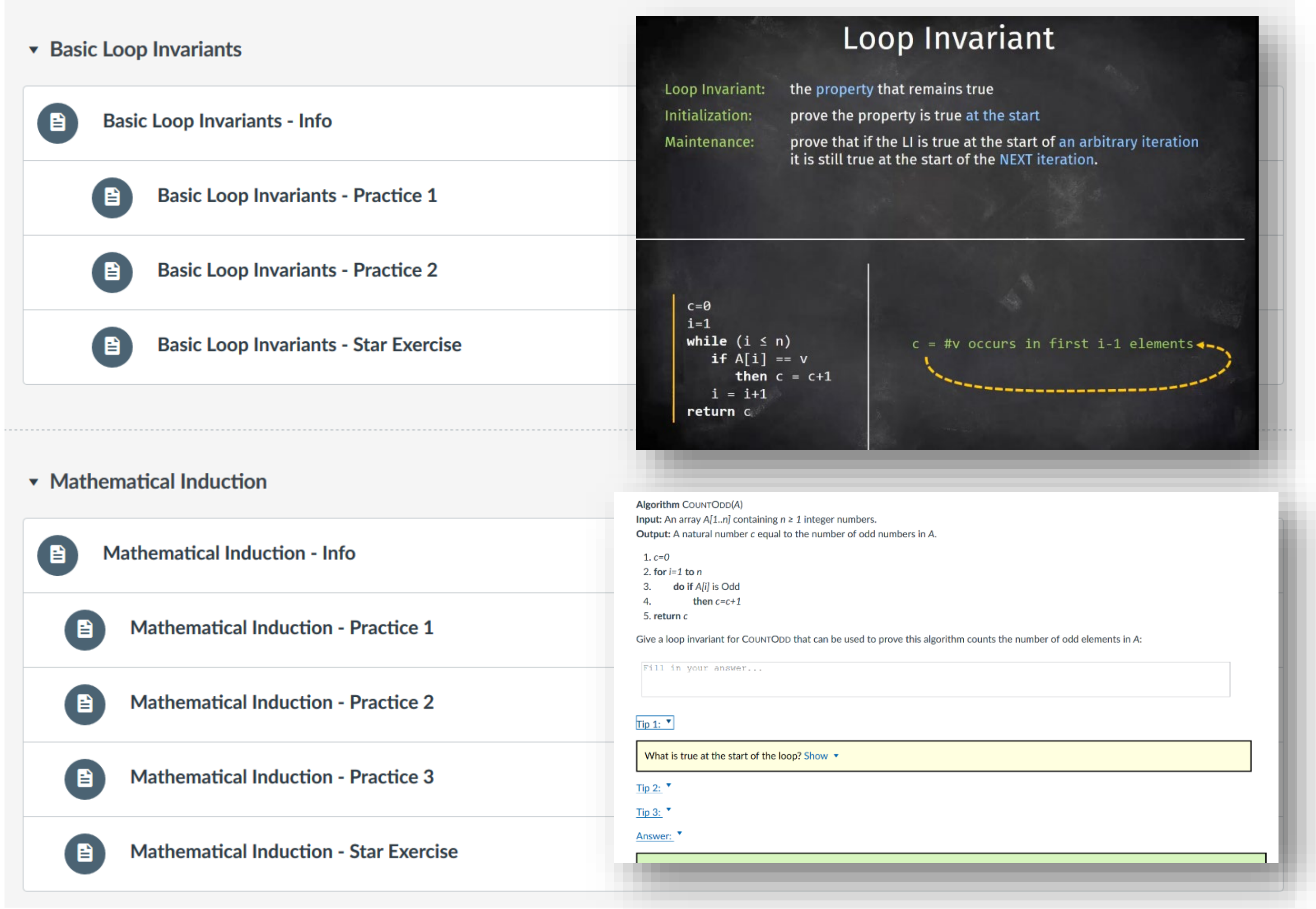
2022-23	Assignments		Passed	
	45 ... 49	72	36	50.0%
	50 ... 54	83	65	78.3%
	55 ... 59	71	63	88.7%
	60 +	166	164	98.8%

2022-23

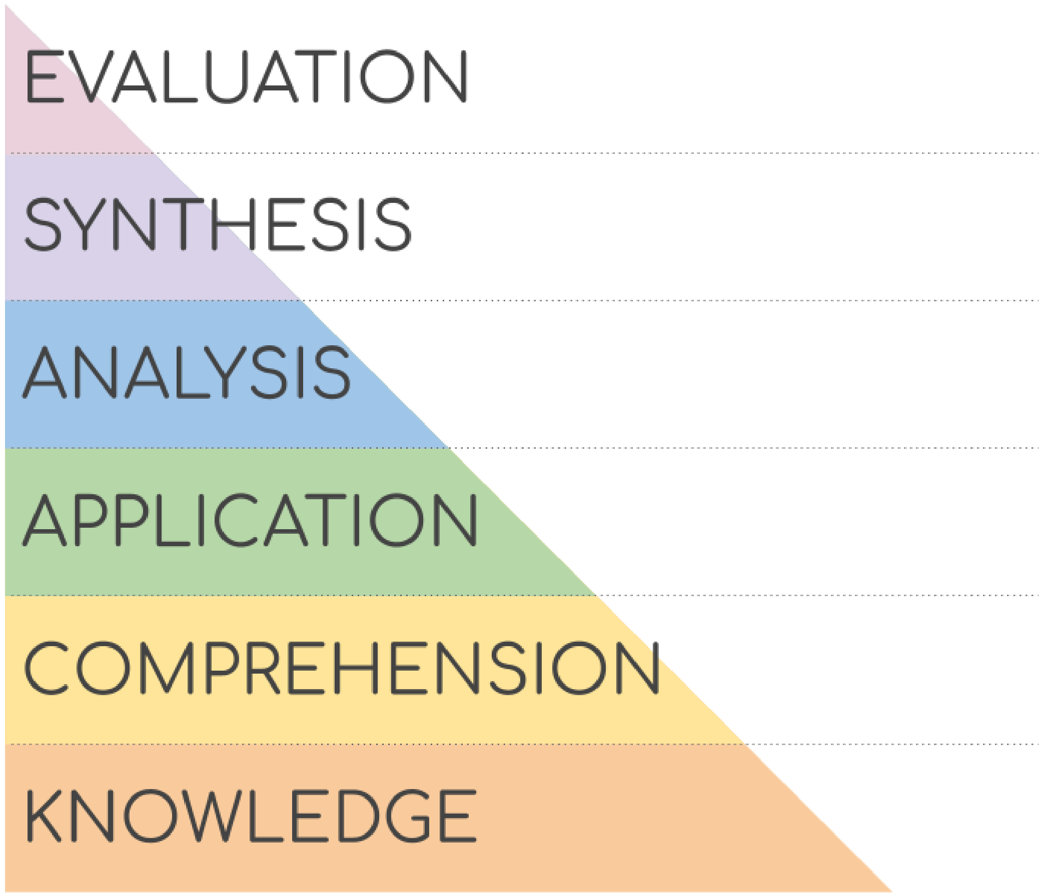
2IT80				2IL50			
382	Pass	252	➞	229	Pass	190	83%
					Fail	39	17%
	Fail	130	➞	94	Pass	25	27%
					Fail	69	73%

Additional study material

Check out the Modules on Canvas … movies and additional exercises



Levels of knowledge



https://en.wikipedia.org/wiki/Bloom%27s_taxonomy

Level 6: design

Level 5: evaluate

Level 4: create

... a non-trivial proof or algorithm

Level 3: reason

... about your understanding of concepts

Level 2: apply

... step-by-step instructions

Level 1: reproduce

... basic facts and understanding

Course objectives

Practice sets, assignments, and interim tests labelled with both levels and objectives

Running time analysis:

- Asymptotic notation
- Best / average / worst case
- Analyzing loops
- Deriving recurrences
- Solving recurrences (Master Theorem, substitution)
- Lower bounds
- Running time analysis

Proving correctness:

- Proving correctness
- Iterative algorithms (loop invariant)
- Recursive algorithms (induction)

Sorting:

- Comparison-based sorting (insertion sort, mergesort, heapsort, quicksort)
- Properties of sorting algorithms (in-place, stable)
- Linear-time sorting (counting sort, radixsort, bucketsort)
- Order statistics

Data structures:

- Heaps
- Hash tables
- (Balanced) binary search trees
- RB-trees
- RB-tree augmentation
- Range trees / KD-trees
- Union-Find

Graphs:

- Graph representations
- BFS / DFS, topological sort
- Minimum spanning trees

How to study effectively

reproduce

apply

reason

create

(evaluate)

(design)

Lectures
reproduce

+ →

Book
reproduce + apply

+ →

Practice sets
apply + reason

+ →

Discussion group
reason + create

= →

Learning

There is more than lectures for a reason ...

How to study effectively

Engage your senses.

Hear...
See...
Feel...
Smell...
Taste....

Data Structures

Lectures — hear → + → **Book** — read → + → **Practice sets** — write

→ + → **Discussion group** — hear + read + write → = → **Learning**

Sorting

let’s get started …

The sorting problem

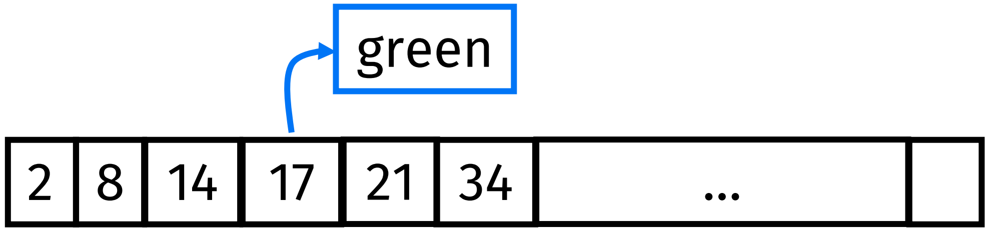
Input: a sequence of n numbers $A = \langle a_1, a_2, \ldots, a_n \rangle$
Output: a permutation of the input such that $\langle a_{i1} \leq a_{i2} \leq \cdots \leq a_{in} \rangle$

The input is typically stored in an array

- We count positions starting at 1 ➡ $A[1]$ is the first element.
- $A[1{:}k]$ is the subarray of A from element 1 till k.

Numbers ≈ Keys

Additional information (satellite data) may be stored with keys



We will study several solutions ≈ algorithms for this problem

Abstract Data Type

Abstract Data Type (ADT) for the sorting problem that

- maintains a set S of n elements
- tracks the order of the elements
- can exchange the order of elements

Array collection of objects stored in linear order
accessible by an integer index

a_1	a_2	a_3	a_4	a_5	…	a_n

$a_i.\mathrm{key}$
$a_i.\mathrm{data}$

set(A, i, x)	:	$A[i] = x$
$x =$ get(A, i)	:	$x = A[i]$
create array A of size n	:	?

Array

set(A, i, x) : $A[i] = x$ 1 operation

x = get(A, i) : $x = A[i]$ 1 operation

create array A

Create array A of size n:

Array(n, $\mathrm{element\_size}$)
1. A = allocate $n \cdot \mathrm{element\_size}$ space

º&á	æÑĭ	ɍꞪ	ẫ♥	%Ø	...	^¨©

Array

$\text{set}(A, i, x)$	: $A[i] = x$	1 operation
$x = \text{get}(A, i)$	: $x = A[i]$	1 operation
create array A	: $A = \text{new Array}[n]$	$n + 1$ operations

Create array A of size n:

```
Array(n, element_size)
    A = allocate n · element_size space      1 operation
    for i = 1 to n                         } n operations
        A[i] = NIL                         }
                                           ----------------
                                           n + 1 operations
```

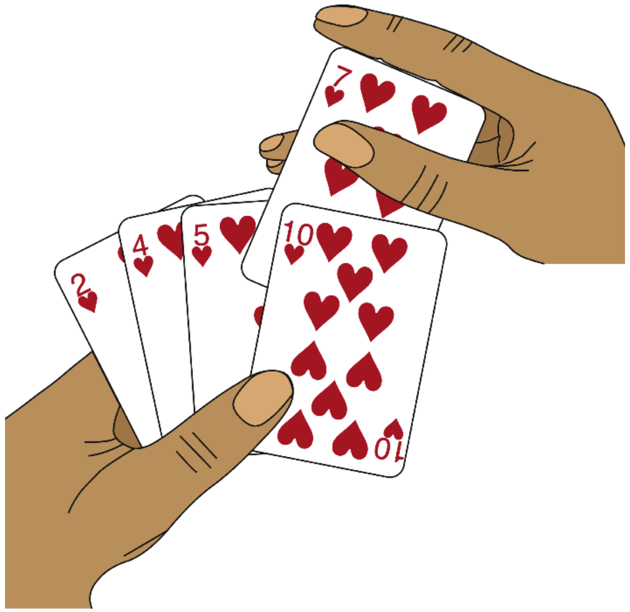
NIL	NIL	NIL	NIL	NIL	...	NIL

Describing algorithms

A complete description of an algorithm consists of three parts:

1. the algorithm
   - *expressed in whatever way is clearest and most concise, can be English and / or pseudocode*
   - *including clear specification of used data structures*
2. a proof of the algorithm's correctness
3. a derivation of the algorithm's running time

InsertionSort



Like sorting a hand of playing cards:

- start with empty left hand, cards on table
- remove cards one by one, insert into correct position
- to find position, compare to cards in hand from right to left
- cards in hand are always sorted

InsertionSort is

- a good algorithm to sort a small number of elements
- an incremental algorithm

Incremental algorithms

process the input elements one-by-one
and maintain the solution for the elements processed so far.

Incremental algorithms

Incremental algorithms
process the input elements one-by-one
and maintain the solution for the elements processed so far.

In pseudocode:

IncAlg(A)

// incremental algorithm which computes the solution of a problem
// with input $A = \{x_1, \ldots, x_n\}$

initialize: compute the solution for $\{x_1\}$
for $j = 2$ **to** n
    compute the solution for $\{x_1, \ldots, x_j\}$ using the (already computed) solution for $\{x_1, \ldots, x_{j-1}\}$

InsertionSort

InsertionSort(A)

// incremental algorithm that sorts array $A[1:n]$ in non-decreasing order

initialize: sort $A[1]$

for $j = 2$ **to** $A.\mathrm{length}$

    sort $A[1:j]$ using the fact that $A[1:j-1]$ is already sorted

InsertionSort

```
InsertionSort(A)
    // incremental algorithm that sorts array A[1:n] in non-decreasing order
initialize: sort A[1]
for j = 2 to A.length
        key = A[j]
        i = j − 1
        while i > 0 and A[i] > key
                A[i + 1] = A[i]
                i = i − 1
        A[i + 1] = key
```

1				i	j		n
1	3	14	17	28	6	…	

$\text{key} = 6$

InsertionSort

```
InsertionSort(A)
    // incremental algorithm that sorts array A[1:n] in non-decreasing order
initialize: sort A[1]
for j = 2 to A.length
        key = A[j]
        i = j − 1
        while i > 0 and A[i] > key
                A[i + 1] = A[i]
                i = i − 1
        A[i + 1] = key
```

1			i	i	j		n
1	3	14	17	28	28	...	

$\text{key} = 6$

InsertionSort

InsertionSort(A)

// incremental algorithm that sorts array $A[1:n]$ in non-decreasing order

initialize: sort $A[1]$

for $j = 2$ **to** $A.\text{length}$

    $\text{key} = A[j]$

    $i = j - 1$

    while $i > 0$ and $A[i] > \text{key}$

        $A[i+1] = A[i]$

        $i = i - 1$

    $A[i+1] = \text{key}$

1		i	i		j		n
1	3	14	17	17	28	...	

$\text{key} = 6$

InsertionSort

InsertionSort(A)

// incremental algorithm that sorts array $A[1:n]$ in non-decreasing order

initialize: sort $A[1]$

for $j = 2$ **to** $A.\mathrm{length}$

  $\mathrm{key} = A[j]$

  $i = j - 1$

  while $i > 0$ and $A[i] > \mathrm{key}$

    $A[i+1] = A[i]$

    $i = i - 1$

  $A[i+1] = \mathrm{key}$

1	i	i			j		n
1	3	14	14	17	28	...	

$\mathrm{key} = 6$

InsertionSort

InsertionSort(A)

// incremental algorithm that sorts array $A[1{:}n]$ in non-decreasing order

initialize: sort $A[1]$

for $j = 2$ **to** $A.\text{length}$

    $\text{key} = A[j]$

    $i = j - 1$

    while $i > 0$ and $A[i] > \text{key}$

        $A[i+1] = A[i]$

        $i = i - 1$

    $A[i+1] = \text{key}$

InsertionSort is an in-place algorithm: the numbers are rearranged within the array with only constant extra space.

1	i				j		n
1	3	6	14	17	28	...	

$\text{key} = 6$

Correctness proof

Use a loop invariant to understand why an algorithm gives the correct answer.

Loop invariant (for InsertionSort)

At the start of iteration j of the "outer" **for** loop the subarray $A[1{:}\, j-1]$ consists of the elements originally in $A[1{:}\, j-1]$ but in sorted order.

Correctness proof

To prove correctness with a loop invariant we need to show three things:

Initialization
Loop invariant is true prior to the first iteration of the loop.

Maintenance
If the loop invariant is true before an iteration of the loop, it remains true before the next iteration.

Termination
When the loop terminates, the loop invariant (usually along with the reason that the loop terminated) gives us a useful property that helps show that the algorithm is correct.

Correctness proof

InsertionSort(A)

```
initialize: sort A[1]
for j = 2 to A.length
    key = A[j]
    i = j − 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i − 1
    A[i + 1] = key
```

Loop invariant

At the start of iteration j of the "outer" **for** loop the subarray $A[1{:}j-1]$ consists of the elements originally in $A[1{:}j-1]$ but in sorted order.

Initialization

Just before the first iteration, $j = 2$ ➡ $A[1{:}j-1] = A[1]$, which is the element originally in $A[1]$ and trivially sorted. Thus the loop invariant holds.

Correctness proof

InsertionSort(A)

initialize: sort $A[1]$
for $j = 2$ **to** $A.\text{length}$
  $\text{key} = A[j]$
  $i = j - 1$
  while $i > 0$ and $A[i] > \text{key}$
    $A[i+1] = A[i]$
    $i = i - 1$
  $A[i+1] = \text{key}$

Loop invariant

At the start of iteration j of the "outer" **for** loop the subarray $A[1{:}\,j-1]$ consists of the elements originally in $A[1{:}\,j-1]$ but in sorted order.

Maintenance

Assume that at the start of the loop the loop invariant holds.

Then $A[1{:}\,j-1]$ is sorted. We need **to prove** that at the end $A[1{:}\,j]$ consists of the original elements in sorted order.
<...*insert proof*...>

Thus, the loop invariant holds before the start of the next iteration.

Correctness proof

InsertionSort(A)

initialize: sort $A[1]$
for $j = 2$ **to** $A.\text{length}$
    $\text{key} = A[j]$
    $i = j - 1$
    while $i > 0$ and $A[i] > \text{key}$
        $A[i+1] = A[i]$
        $i = i - 1$
    $A[i+1] = \text{key}$

Loop invariant

At the start of iteration j of the “outer” **for** loop the subarray $A[1{:}\,j-1]$ consists of the elements originally in $A[1{:}\,j-1]$ but in sorted order.

Termination

The outer **for** loop ends when $j > n$; this is when $j = n + 1$ ➡ $j - 1 = n$.

By the loop invariant $A[1{:}\,n]$ consists of the elements originally in $A[1{:}\,n]$ in sorted order.

Another sorting algorithm

using a different paradigm …

MergeSort

A divide-and-conquer sorting algorithm

Divide-and-conquer

- break the problem into two or more subproblems
- solve the subproblems recursively
- and then combine these solutions to create a solution to the original problem

Divide-and-conquer

D&CAlg(A)

// divide-and-conquer algorithm that computes the solution of a problem
// with input $A = \{x_1, \dots, x_n\}$

if # elements of A is small enough (for example 1)
    compute sol (the solution for A) brute-force
else
    split A in, for example, 2 non-empty subsets A_1 and A_2
    sol_1 = D&CAlg(A_1)
    sol_2 = D&CAlg(A_2)
    compute sol (the solution for A) from sol_1 and sol_2
return sol

MergeSort

```
MergeSort(A)
    // divide-and-conquer algorithm that sorts array A[1:n]
    if A.length == 1
        compute sol (the solution for A) brute-force
    else
        split A in 2 non-empty subsets A_1 and A_2
        sol_1 = MergeSort(A_1)
        sol_2 = MergeSort(A_2)
        compute sol (the solution for A) from sol_1 and sol_2
```

MergeSort

MergeSort(A)

// divide-and-conquer algorithm that sorts array $A[1:n]$

if $A.\mathrm{length} == 1$

skip

else

$n = A.\mathrm{length};\ n_1 = \left\lfloor \frac{n}{2} \right\rfloor;\ n_2 = \left\lceil \frac{n}{2} \right\rceil$

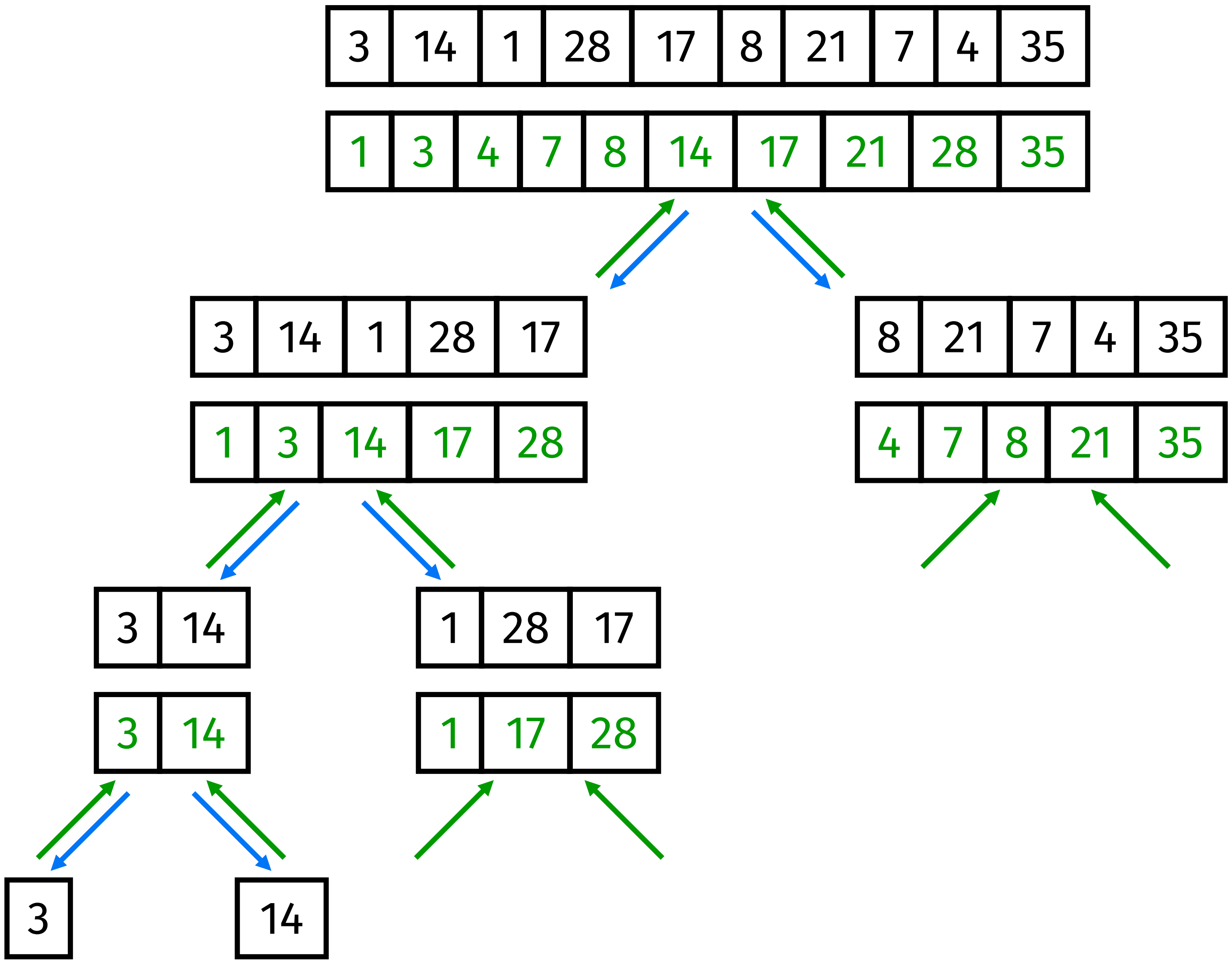
copy $A[1:n_1]$ to auxiliary array $A_1[1:n_1]$

copy $A[n_1 + 1:n]$ to auxiliary array $A_2[1:n_2]$

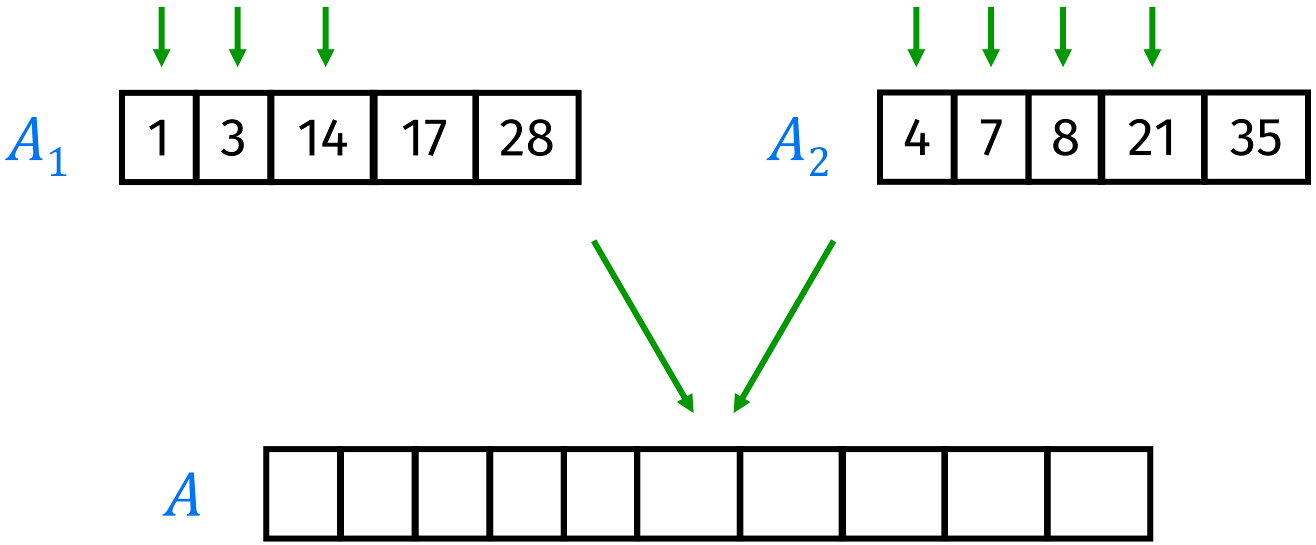
MergeSort(A_1); MergeSort(A_2)

Merge(A, A_1, A_2)

MergeSort


3 14 1 28 17 8 21 7 4 35
1 3 4 7 8 14 17 21 28 35
3 14 1 28 17
1 3 14 17 28
8 21 7 4 35
4 7 8 21 35
3 14
3 14
1 28 17
1 17 28
3
14

MergeSort: Merging



MergeSort: correctness proof

Induction on n (# of input elements)

- proof that the base case (n small) is solved correctly
- proof that if all subproblems are solved correctly,
  then the complete problem is solved correctly

MergeSort

```
MergeSort(A)
    // divide-and-conquer algorithm that sorts array A[1:n]
    if A.length == 1
        skip
    else
        n = A.length;  n_1 = ⌊n/2⌋;  n_2 = ⌈n/2⌉
        copy A[1:n_1] to auxiliary array A_1[1:n_1]
        copy A[n_1 + 1:n] to auxiliary array A_2[1:n_2]
        MergeSort(A_1);  MergeSort(A_2)
        Merge(A, A_1, A_2)
```

MergeSort: correctness proof

MergeSort(A)
// divide-and-conquer algorithm that sorts array $A[1{:}n]$
if $A.\text{length} == 1$
skip
else
$n = A.\text{length};\ n_1 = \left\lfloor \frac{n}{2} \right\rfloor;\ n_2 = \left\lceil \frac{n}{2} \right\rceil$
copy $A[1{:}n_1]$ to auxiliary array $A_1[1{:}n_1]$
copy $A[n_1 + 1{:}n]$ to auxiliary array $A_2[1{:}n_2]$
MergeSort(A_1); MergeSort(A_2)
Merge(A, A_1, A_2)

Lemma

MergeSort sorts an array A of length n correctly.

Proof by strong induction on n. (sketch)

Base case ($n = 1$).

An array containing only one element is trivially sorted.

MergeSort correctly does not make any changes. ✓

Inductive step

Let $k \geq 1$.

Let $k_1 = (k+1)/2$ and $k_2 = (k+1)/2$. Note that $k_1 < k+1$ and $k_2 < k+1$.

Inductive hypothesis

MergeSort sorts any array A of length $k' \leq k$ correctly.

By the IH A_1 and A_2 are correctly sorted.

Remains to show: Merge(A, A_1, A_2) correctly constructs a sorted array A out of the sorted arrays A_1 and A_2 … *see practice set*

Analysis of algorithms

some informal thoughts – for now …

Analysis of algorithms

Can we say something about the running time of an algorithm without implementing and testing it?

```
InsertionSort(A)
initialize: sort A[1]
for j = 2 to A.length
    key = A[j]
    i = j - 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i - 1
    A[i + 1] = key
```

Analysis of algorithms

Analyze the running time as a function of n (# of input elements)

- best case
- average case
- worst case

An algorithm has worst case running time $T(n)$ if for any input of size n the maximal number of elementary operations executed is $T(n)$.

elementary operations

add, subtract, multiply, divide, load, store, copy, conditional and unconditional branch, return ...

Analysis of algorithms: example

		$n = 10$	$n = 100$	$n = 1000$
InsertionSort:	$15n^2 + 7n - 2$	1568	150698	1.5×10^7
MergeSort:	$300\, n \log n + 50n$	10466	204316	3.0×10^6

InsertionSort
$6 \times$ faster

InsertionSort
$1.35 \times$ faster

MergeSort
$5 \times$ faster

$n = 1{,}000{,}000$ InsertionSort 1.5×10^{13}

MergeSort 6×10^9 $2500 \times$ faster !

Analysis of algorithms

It is extremely important to have efficient algorithms for large inputs

The rate of growth (or order of growth) of the running time is far more important than constants

InsertionSort: $\Theta(n^2)$

MergeSort: $\Theta(n \log n)$

Θ-notation

Intuition: concentrate on the leading term, ignore constants

$19\,n^3 + 17\,n^2 - 3n$	becomes	$\Theta(n^3)$
$2\,n\log n + 5\,n^{1.1} - 5$	becomes	$\Theta(n^{1.1})$
$n - ¾\,n\,\sqrt{n}$	becomes	---

(*precise definition next lecture…*)

Some rules and notation

$\log n$ denotes $\log_2 n$

We have for $a, b, c > 0$:

$$\log_c(ab) =$$

$$\log_c a + \log_c b$$

$$\log_c (a^b) =$$

$$b \log_c a$$

$$\log_a b \quad =$$

$$\log_c b \ / \ \log_c a$$

Find the leading term

$\log^{35} n$ vs. $\sqrt{n}$?

- logarithmic functions grow slower than polynomial functions
- $\log^a n$ grows slower than n^b for all constants $a > 0$ and $b > 0$

n^{100} vs. 2^n ?

- polynomial functions grow slower than exponential functions
- n^a grows slower than b^n for all constants $a > 0$ and $b > 1$