

# INF102

## Algorithms, Data Structures and Programming

Marc Bezem<sup>1</sup>

<sup>1</sup>Department of Informatics  
University of Bergen

Fall 2015

# INF102, practical stuff

- ▶ Lecturer: Marc Bezem; Team: see homepage
- ▶ Homepage: [INF102](#) (hyperlinks in red)
- ▶ Also: [GitHub](#) (recommended); Dropbox: [slides](#), [schedule](#)
- ▶ Textbook: [Algorithms, 4th edition](#)
- ▶ Prerequisites: INF100 + 101 ( $\approx$  Ch. 1.1 + 1.2)
- ▶ Syllabus (pensum): Ch. 1.3 – 1.5, Ch. 2 – 4
- ▶ Exam: three compulsory exercises and a [written exam](#)
- ▶ Old exams: [2004–2013](#), [2014](#)
- ▶ [Table of Contents of these slides](#)

# Resources

- ▶ Good textbook, USA-style: many pages, exercises etc.
- ▶ Average speed must be ca 50 pages p/w
- ▶ Lectures (ca 24) focus on the essentials
- ▶ Slides (ca 120, dense!) summarize the lectures
- ▶ Prepare yourself by reading in advance
- ▶ Workshops: selected exercises
- ▶ Test yourself by trying some exercises in advance
- ▶ If you can do the exercises (incl. compulsory), you are fine
- ▶ Review of exercises on Friday morning

## Generic Bags, Queues and Stacks

- ▶ Generic programming in Java, example: **PolyPair.java**
- ▶ Bag, Queue and Stack are generic, iterable collections
- ▶ Queue and Stack: Ch. 9 in textbook INF100/1
- ▶ APIs include: `boolean isEmpty()` and `int size()`
- ▶ All three support adding an element
- ▶ Queue and Stack support removing an element (if any)
- ▶ FIFO Queue (en/dequeue), LIFO Stack (push/pop)
- ▶ Dijkstra's Two-Stack Expression Evaluation **Movie**
- ▶ Example:  $( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )$

## Implementations

- ▶ `ResizingArray_Stack.java`
- ▶ Arrays give direct access, but have fixed size
- ▶ Resizing takes time and space proportional to size
- ▶ `LinkedList_Stack.java`
- ▶ No fixed size, but indirect access
- ▶ Pointers take space and dereferencing takes time
- ▶ Programming with pointers: make a picture
- ▶ `LinkedList_Queue.java`

## Computation time and memory space

- ▶ Two central questions:
  - ▶ How long will my program take?
  - ▶ Will there be enough memory?
- ▶ Example: **TheeSum.java**
- ▶ Inner loop is important
- ▶ Sorting helps: **ThreeSumOptimized.java**
- ▶ Run some experiments: `1Kints.txt`, `2Kints.txt`, ...

## Methods of Analysis

- ▶ Empirical:
  - ▶ Run program with randomized inputs, measuring time & space
  - ▶ Run program repeatedly, doubling the input size
  - ▶ Measuring time: **StopWatch**
  - ▶ Plot, or log-log plot and **linear regression**
- ▶ Theoretical:
  - ▶ Define a cost model by abstraction (e.g., array accesses, comparisons, operations)
  - ▶ Try to count/estimate/average this cost as function of the input (size)
  - ▶ Use  $O(f(n))$  and  $f(n) \sim g(n)$

## ThreeSum, empirically

- ▶ Input sizes 1K, 2K, 4K, 8K take time 0.1, 0.8, 6.4 ,51.1 sec
- ▶ The log's are 3, 3.3, 3.6, 3.9 and -1, -0.1, 0.8, 1.71
- ▶ Basis of the logarithm should be the same for both
- ▶ Linear regression gives  $y \approx 3x - 10$
- ▶  $\lg(f(n)) = 3 \lg(n) - 10$  iff

$$f(n) = 10^{\lg(f(n))} = 10^{3 \lg(n) - 10} = n^3 * 10^{-10}$$

- ▶ Conclusion: cubic in the input size, with constant  $\approx 10^{-10}$
- ▶ Strong dependence on input can be a problem
- ▶ Constant  $10^{-10}$  depends on computer, exponent 3 does not



## ThreeSum, theoretically

- ▶ Number of different picks of triples:  $g(n) = n(n-1)(n-2)/6$
- ▶ Inner loop executed  $g(n)$  times
- ▶  $g(n) = n^3/6 - n^2/2 + n/3$
- ▶ Cubic term  $n^3/6$  wins for large  $n$
- ▶ Computational model # array accesses:  $n^3/2$
- ▶ Cost array access  $t$  sec: time  $t * n^3/2$  sec
- ▶ Cost models are abstractions! (NB cache)

## Big Oh, and $\sim$

- ▶ Q: 'wins for large  $n$ ' uhh???
- ▶ A: Big Oh, and  $\sim$  will clear this up
- ▶ Costs are positive quantities, so  $f, g, \dots : \mathbb{N} \rightarrow \mathbb{R}^+$
- ▶ MNF130:  $f(n)$  is  $O(g(n))$  if there exist  $c, N$  such that  $f(n) \leq cg(n)$  for all  $n \geq N$
- ▶ Example:  $n^2$  and even  $99n^3$  are  $O(n^3)$ , but  $n^3$  is not  $O(n^{2.9})$
- ▶ INF102:  $f(n) \sim g(n)$  if  $1 = \lim f(n)/g(n)$
- ▶ If  $f(n) \sim g(n)$ , then  $f(n)$  is  $O(g(n))$  and  $g(n)$  is  $O(f(n))$
- ▶ Big Oh and  $\sim$  aim to capture 'order of growth'
- ▶ Big Oh abstracts from constant factors,  $\sim$  does not
- ▶ Large constant factors are important!

## Important orders of growth

- ▶ constant:  $c$  ( $f(n) = c$  for all  $n$ )
- ▶ linear:  $n$  (compare all for  $n = 20$  sec)
- ▶ linearithmetic:  $n \lg n$
- ▶ quadratic:  $n^2$
- ▶ cubic:  $n^3$
- ▶ exponential:  $2^n$
- ▶ general form:  $an^b(\lg n)^c$

## Logarithms and Exponents

- ▶ Definition:  $\log_x z = y$  iff  $x^y = z$  for  $x > 0$
- ▶ Inverses:  $x^{\log_x y} = y$  and  $\log_x x^y = y$
- ▶ Exponent:  $x^{(y+z)} = x^y x^z$ ,  $x^{(yz)} = (x^y)^z$
- ▶ Logarithm:  $\log_x(yz) = \log_x y + \log_x z$ ,  $\log_x z = \log_x y \log_y z$
- ▶ Double exponent: e.g.  $2^{(2^n)}$  (not used in INF102)
- ▶ Double logarithm:  $\log(\log n)$  (not used in INF102)

## Examples

- ▶ Worst case: guaranteed, independent of input
- ▶ Average case: not guaranteed, dependent of input *distribution*
- ▶ Linked list implementations of Stack, Queue and Bag: all operations take constant time in the worst case
- ▶ Resizing array implementations of Stack, Queue and Bag: adding and deleting take linear time in the worst case (easy)
- ▶ Amortized: worst-case cost *per operation*. E.g., each 10-th operation may have cost 21, all others cost 1, so  $\leq 3$  p/o.
- ▶ Resizing arrays: adding and deleting take constant time *per operation* in the worst case (proof is difficult)
- ▶ Special case of resizing array that is only growing:  
 $1(2)2(4)3(4)4(8)5(6)6(8)7(16)8(9) \dots 16(32) \dots$ , with  $(n)$  the new size.  
 Resizing to  $(n)$  costs  $2n$  array accesses, so in total  
 $(1+4)+(1+8)+(2+16)+(4+32)+(8+64) \dots$ , so 9 p/push.

## Staying Connected

- ▶ MNF130: relation  $R \subseteq V \times V$  is an *equivalence* if
  - ▶  $R$  is *reflexive*:  $\forall x \in V. R(x, x)$
  - ▶  $R$  is *symmetric*:  $\forall x, y \in V. R(x, y) \rightarrow R(y, x)$
  - ▶  $R$  is *transitive*:  $\forall x, y, z \in V. R(x, y) \wedge R(y, z) \rightarrow R(x, z)$
- ▶ We assume connectedness to be an equivalence
- ▶ Dynamic connectivity means that  $R$  can grow and shrink
- ▶ Example: if the 'Bergensbanen' is broken, Oslo and Bergen are no longer connected by rail
- ▶ We want efficient algorithms and datastructures for testing whether two objects are connected
- ▶ Clear relationship with paths in graphs, more in Ch. 4
- ▶ Here we take  $V = \{0, \dots, N - 1\}$ .

## Staying Connected

- ▶ MNF130: relation  $E \subseteq V \times V$  is an *equivalence* if
  - ▶  $E$  is *reflexive*:  $\forall x \in V. E(x, x)$
  - ▶  $E$  is *symmetric*:  $\forall x, y \in V. E(x, y) \rightarrow E(y, x)$
  - ▶  $E$  is *transitive*:  $\forall x, y, z \in V. E(x, y) \wedge E(y, z) \rightarrow E(x, z)$
- ▶ We assume connectedness to be an equivalence
- ▶ Dynamic connectivity means that  $R$  can grow and shrink
- ▶ Example: if the 'Bergensbanen' is broken, Oslo and Bergen are no longer connected by rail
- ▶ We want efficient algorithms and datastructures for testing whether two objects are connected
- ▶ Clear relationship with paths in graphs, (connected) components (MNF130)
- ▶ We take  $V = \{0, \dots, N - 1\}$ .

## Union Find

- ▶ UF, idea: every component has an identifier ('hub'), which has edges ('spokes') to the elements of its component
- ▶ API: **UF**
- ▶ Implementations with `int[] id` containing the identifiers
  - ▶ **SlowUF.java**
  - ▶ **FastUF.java**
  - ▶ **WeightedUF.java**
- ▶ WeightedUF: log depth of tree (Proposition X)



# Sorting

- ▶ Sorting: putting objects in a certain order
- ▶ MNF130: relation  $R \subseteq V \times V$  is a *total order(ing)* if
  1.  $R$  is *reflexive*:  $\forall x \in V. R(x, x)$
  2.  $R$  is *transitive*:  $\forall x, y, z \in V. R(x, y) \wedge R(y, z) \rightarrow R(x, z)$
  3.  $R$  is *antisymmetric*:  $\forall x, y \in V. R(x, y) \wedge R(y, x) \rightarrow x = y$
  4.  $R$  is *total*:  $\forall x, y \in V. R(x, y) \vee R(y, x)$
- ▶ Natural orderings:
  - ▶ Numbers of any type: ordinary  $\leq$  and  $\geq$
  - ▶ Strings: lexicographic
  - ▶ Objects of a Comparable type:  $v.\text{compareTo}(w) < 0$

## Sorting (ctnd)

- ▶ Bubble sort: `ExampleSort.java`
- ▶ Certification: `assert isSorted(a)` in `main()`
- ▶ No guarantee against modifying the array (but `exch()` is safe)
- ▶ Costmodel 1: number of `exch()`'s and `less()`'s
- ▶ Costmodel 2: number of array accesses
- ▶ Pitfalls: cache misses, expensive `v.compareTo(w) < 0`
- ▶ Why studying sorting? (`java.util.Arrays.sort()`)
- ▶ Comparing sorting algorithms: `CompareSort.java`

## Selection Sort

- ▶ Bubble sort:  $\sim n^2/2$  compares, 0 . .  $\sim n^2/2$  exchanges
- ▶ Selection sort:
  - ▶ Find index of a minimal value  $a[1..n]$ , exchange with  $a[1]$
  - ▶ Find index of a minimal value  $a[2..n]$ , exchange with  $a[2]$
  - ▶ ... until  $n-1$
- ▶ Selection sort:  $\sim n^2/2$  compares,  $n-1$  exchanges

```
public static void sort(Comparable[] a) {  
    int N = a.length;  
    for (int i=0; i<N-1; i++){  
        int min=i;  
        for (int j=i+1; j<N; j++) if (less(a[j],a[min])) min=j;  
        exch(a,i,min);  
    }  
}
```

## Insertion sort

- ▶ Insertion sort:
  - ▶ Insert  $a[2]$  on its correct place in (sorted)  $a[1..1]$
  - ▶ Insert  $a[3]$  on its correct place in (sorted)  $a[1..2]$
  - ▶ ... until  $a[n]$
- ▶ Very good for partially sorted arrays, costs:
  - ▶ Best case:  $n-1$  compares and 0 exchanges
  - ▶ Worst case:  $\sim n^2/2$  compares and exchanges
  - ▶ Average case:  $\sim n^2/4$  compares and exchanges (distinct keys)

```
public static void sort(Comparable[] a) {  
    int N = a.length;  
    for (int i=1; i<N; i++){  
        for (int j=i; j>0 && less(a[j],a[j-1]); j--)  
            exch(a,j,j-1);  
    }  
}
```

## Shell sort

- ▶ Insertion sort:
  - ▶ Very good for partially sorted arrays
  - ▶ Slow in transport: step by step `exch(a,j,j-1)`
- ▶ Idea: h-sort, `a[i], a[i+h], a[i+2h], ...` sorted (any `i`)

```
public static void hsort(int h, Comparable[] a) {  
    int N = a.length;  
    for (int i=h; i<N; i++)  
        for (int j=i; j-h>=0 && less(a[j],a[j-h]); j-=h)  
            exch(a,j,j-h);  
}
```

- ▶ Insertion sort: `hsort(1,a)`
- ▶ Shell sort: e.g., `hsort(10,a); hsort(1,a)`

## Shell sort (ctnd)

- ▶ `hsort(10,a); hsort(1,a)` faster than just `hsort(1,a)` !
- ▶ Q: How is this possible?
- ▶ A: `hsort(10,a)` transports items in steps of 10, which would be done by `hsort(1,a)` in 10 steps of 1
- ▶ What about `hsort(100,a); hsort(10,a); hsort(1,a)`?
- ▶ To be expected: depends on the length N of the array
- ▶ Book:

# Mergesort

- ▶ Top-down (recursive) algorithm:
  - ▶ Mergesort left half, mergesort right half
  - ▶ Merge the results
- ▶ Using an auxiliary array: [TopDownMergeSort.java](#), [Movie](#)
- ▶ Bottom-up algorithm:
  - ▶ Merge  $a[0], a[1], a[2], a[3], a[4], a[5], \dots$
  - ▶ Merge  $a[0..1], a[2..3], a[4..5], a[6..7], \dots$
  - ▶ Merge  $a[0..3], a[4..7], a[8..11], a[12..15], \dots$
- ▶ Also using an auxiliary array: [BottomUpMergeSort.java](#)

## The complexity of sorting

- ▶ Mergesort uses between  $\sim (n/2) \lg n$  and  $\sim n \lg n$  compares
- ▶ Mergesort uses between  $\sim 6n \lg n$  array accesses
- ▶ Mergesort uses  $\sim 2n$  space (plus some var's)
- ▶ Q: How fast can compare-based sorting be?
- ▶ Book:



# Quicksort

- ▶ Top-down (recursive) algorithm:
  - ▶ Choose a (pivot value)  $v$  in the array
  - ▶ Partition the array in non-empty parts  $\leq v$  and  $\geq v$
  - ▶ Quicksort the two parts
- ▶ Pros: in-place, average computation time  $O(n \lg n)$
- ▶ Cons: stack space for the recursion, worst-case  $O(n^2)$
- ▶ Implementation: **QuickSort.java**

## ToC and topics of general interest

- ▶ Table of Contents on next slide (all items clickable)
- ▶ Practical stuff: slide 2

Introduction

Ch.1.3 Bags, Queues and Stacks

Ch.1.4 Analysis of Algorithms

Ch.1.5 Case Study: Union-Find

Ch.2.1 Elementary Sorts

Ch.2.2 Mergesort

Ch.2.3 Quicksort

Ch.2.4 Priority Queues

Ch.3.1 Symbol Tables

Ch.3.2 Binary Search Trees

Ch.3.3 Balanced Search Trees

Ch.3.4 Hash Tables

Ch.3.5 Applicatios

Ch.4.1 Undirected Graphs

Ch.4.2 Directed Graphs

Ch.4.3 Minimum Spanning  
Trees

Ch.4.4 Shortest Paths

Table of Contents