

# 2

# Machines & Models

20 October 2025

Prof. Dr. Sebastian Wild

# Learning Outcomes

## Unit 2: *Machines & Models*

1. Understand the difference between empirical *running time* and algorithm *analysis*.
2. Understand *worst / best / average case* models for input data.
3. Know the *RAM machine* model.
4. Know the definitions of *asymptotic notation* (Big-Oh classes and relatives).
5. Understand the reasons to make *asymptotic approximations*.
6. Be able to *analyze* simple *algorithms*.

## Outline

# 2 Machines & Models

- 2.1 Algorithm analysis
- 2.2 The RAM Model
- 2.3 Asymptotics & Big-Oh
- 2.4 Teaser: Maximum subarray problem

# What is an algorithm?

An algorithm is a sequence of instructions.

think: recipe

More precisely:

e. g. Python script

1. mechanically executable  
~~ no “common sense” needed
2. finite description    ≠ finite computation!
3. solves a *problem*, i. e., a class of problem instances  
 $x + y$ , not only  $17 + 4$

- ▶ input-processing-output abstraction



Typical example: *bubblesort*

~~ not a specific program  
but the underlying idea

# What is a data structure?

A data structure is

1. a rule for **encoding data**  
(in computer memory), plus
2. **algorithms** to work with it  
(queries, updates, etc.)

**typical example:** *binary search tree*



## 2.1 Algorithm analysis

# Good algorithms

**Our goal:** Find good (best?) algorithms and data structures for a task.

Good “usually” means

- ▶ fast running *time*  
can be complicated in distributed systems
- ▶ moderate memory *space* usage

*Algorithm analysis* is a way to

- ▶ compare different algorithms,
- ▶ predict their performance in an application

# Running time experiments

Why not simply run and time it?

- ▶ results only apply to
  - ▶ single *test* machine
  - ▶ tested inputs
  - ▶ tested implementation
  - ▶ ...
- ≠ *universal truths*
- ▶ instead: consider and analyze algorithms on an abstract machine
  - ~~ provable statements for model
  - ~~ testable model hypotheses
  - ~~ Need precise model of machine (costs), input data and algorithms.



survives Pentium 4

# Data Models

Algorithm analysis typically uses one of the following simple data models:

- ▶ **worst-case performance:**

consider the *worst* of all inputs as our cost metric

- ▶ **best-case performance:**

consider the *best* of all inputs as our cost metric

- ▶ **average-case performance:**

consider the average/expectation of a *random* input as our cost metric

Usually, we apply the above for *inputs of same size  $n$* .

~~~ performance is only a **function of  $n$** .

## 2.2 The RAM Model

# Machine models

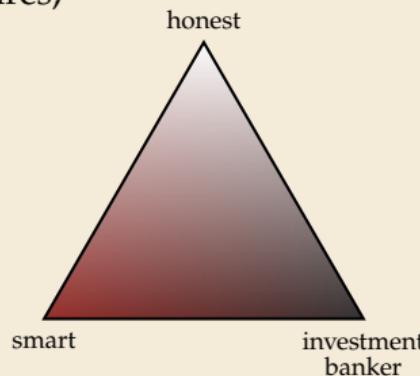
The machine model decides

- ▶ what algorithms are possible
- ▶ how they are described (= programming language)
- ▶ what an execution *costs*

**Goal:** Machine models should be

detailed and powerful enough to reflect actual machines,  
abstract enough to unify architectures,  
simple enough to analyze.

~~ usually some compromise is needed



# Random Access Machines

## Random access machine (RAM)

more detail in §2.2 of *Sequential and Parallel Algorithms and Data Structures*  
by Sanders, Mehlhorn, Dietzfelbinger, Dementiev

- ▶ unlimited *memory*  $\text{MEM}[0], \text{MEM}[1], \text{MEM}[2], \dots$
  - ▶ fixed number of *registers*  $R_1, \dots, R_r$  (say  $r = 100$ )
  - ▶ memory cells  $\text{MEM}[i]$  and registers  $R_i$  store  $w$ -bit integers, i. e., numbers in  $[0..2^w - 1]$   
 $w$  is the word width/size; typically  $w \propto \lg n \rightsquigarrow 2^w \approx n$
  - ▶ Instructions:
    - ▶ load & store:  $R_i := \text{MEM}[R_j] \quad \text{MEM}[R_j] := R_i$
    - ▶ operations on registers:  $R_k := R_i + R_j$  (arithmetic is *modulo  $2^w$ !*)  
also  $R_i - R_j, R_i \cdot R_j, R_i \text{ div } R_j, R_i \text{ mod } R_j$   
C-style operations (bitwise and/or/xor, left/right shift)
    - ▶ conditional and unconditional jumps
  - ▶ cost: number of executed instructions
- rightsquigarrow The RAM is the standard model for sequential computation.
- we will see further models later

# RAM-Program Example

## Example RAM program

---

```
1 // Assume: R1 stores number N
2 // Assume: MEM[0..N) contains list of N numbers
3 R2 := R1;
4 R3 := R1 - 2;
5 R4 := MEM[R3];
6 R5 := R3 + 1;
7 R6 := MEM[R5];
8 if (R4 ≤ R6) goto line 11;
9 MEM[R3] := R6;
10 MEM[R5] := R4;
11 R3 := R3 - 1;
12 if (R3 ≥ 0) goto line 5;
13 R2 := R2 - 1;
14 if (R2 > 0) goto line 4;
15 // Done:
```

---

# RAM-Program Example

## Example RAM program

```
1 // Assume: R1 stores number N
2 // Assume: MEM[0..N) contains list of N numbers
3 R2 := R1;
4 R3 := R1 - 2;
5 R4 := MEM[R3];
6 R5 := R3 + 1;
7 R6 := MEM[R5];
8 if (R4 ≤ R6) goto line 11;
9 MEM[R3] := R6;
10 MEM[R5] := R4;
11 R3 := R3 - 1;
12 if (R3 ≥ 0) goto line 5;
13 R2 := R2 - 1;
14 if (R2 > 0) goto line 4;
15 // Done: MEM[0..N) sorted
```

5.2.2

SORTING BY EXCHANGING 107

they need not be examined on subsequent passes. Horizontal lines in Fig. 14 show the progress of the sorting from this standpoint; notice, for example, that five more elements are known to be in final position as a result of Pass 4. On the final pass, no exchanges are performed at all. With these observations we are ready to formulate the algorithm.

**Algorithm B (Bubble sort).** Records  $R_1, \dots, R_N$  are rearranged in place; after sorting is complete their keys will be in order,  $K_1 \leq \dots \leq K_N$ .

- B1. [Initialize BOUND.] Set  $\text{BOUND} \leftarrow N$ . ( $\text{BOUND}$  is the highest index for which the record is not known to be in its final position; thus we are indicating that nothing is known at this point.)
- B2. [Loop on  $j$ .] Set  $t \leftarrow 0$ . Perform step B3 for  $j = 1, 2, \dots, \text{BOUND} - 1$ , and then go to step B4. (If  $\text{BOUND} = 1$ , this means go directly to B4.)
- B3. [Compare/exchange  $R_j : R_{j+1}$ .] If  $K_j > K_{j+1}$ , interchange  $R_j \leftrightarrow R_{j+1}$  and set  $t \leftarrow j$ .
- B4. [Any exchanges?] If  $t = 0$ , terminate the algorithm. Otherwise set  $\text{BOUND} \leftarrow t$  and return to step B2.

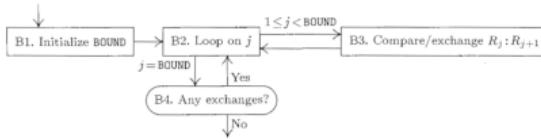
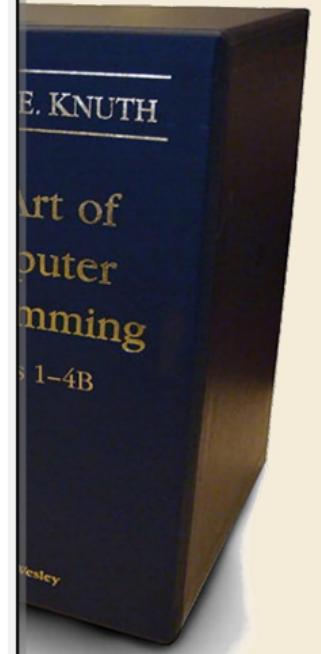


Fig. 15. Flow chart for bubble sorting.

**Program B (Bubble sort).** As in previous MIX programs of this chapter, we assume that the items to be sorted are in locations  $\text{INPUT}+1$  through  $\text{INPUT}+N$ .  $r11 \equiv t$ ;  $r12 \equiv j$ .

|    |       |      |            |       |                                                                 |
|----|-------|------|------------|-------|-----------------------------------------------------------------|
| 01 | START | ENT1 | N          | 1     | <u>B1. Initialize BOUND, <math>t \leftarrow N</math>.</u>       |
| 02 | 1H    | ST1  | BOUND(1:2) | A     | <u>B2. Loop on <math>j</math>. <math>j \leftarrow 1</math>.</u> |
| 03 |       | ENT2 | 1          | A     | <u>B2. Loop on <math>j</math>. <math>j \leftarrow 1</math>.</u> |
| 04 |       | ENT1 | 0          | A     | <u>B2. Loop on <math>j</math>. <math>j \leftarrow 1</math>.</u> |
| 05 |       | JMP  | BOUND      | A     | <u>B2. Loop on <math>j</math>. <math>j \leftarrow 1</math>.</u> |
| 06 | 3H    | LDA  | INPUT,2    | C     | <u>B3. Compare/exchange <math>R_j : R_{j+1}</math>.</u>         |
| 07 |       | CMPA | INPUT+1,2  | C     | <u>B3. Compare/exchange <math>R_j : R_{j+1}</math>.</u>         |
| 08 |       | JLE  | 2F         | C     | No exchange if $K_j \leq K_{j+1}$ .                             |
| 09 |       | LDX  | INPUT+1,2  | B     | $R_{j+1}$                                                       |
| 10 |       | STX  | INPUT,2    | B     | $\rightarrow R_j$ .                                             |
| 11 |       | STA  | INPUT+1,2  | B     | (old $R_j \rightarrow R_{j+1}$ ).                               |
| 12 |       | ENT1 | 0,2        | B     | $t \leftarrow j$ .                                              |
| 13 | 2H    | INC2 | 1          | C     | $j \leftarrow j + 1$ .                                          |
| 14 | BOUND | ENTX | =*,2       | A + C | $tX \leftarrow j - \text{BOUND}$ . [Instruction modified]       |
| 15 |       | JXN  | 3B         | A + C | Do step B3 for $1 \leq j < \text{BOUND}$ .                      |
| 16 | 4H    | J1P  | 1B         | A     | <u>B4. Any exchanges? To B2 if <math>t &gt; 0</math>.</u>       |



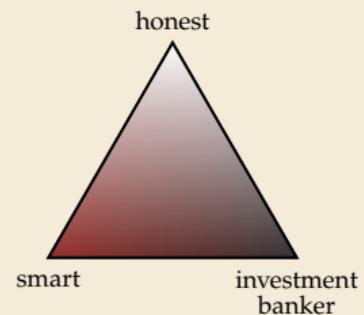
# Pseudocode

- ▶ Programs for the random-access machine are very low level and detailed
  - ≈ assembly / machine language

Typical simplifications when describing and analyzing algorithms:

- ▶ more abstract *pseudocode* code that humans understand (easily)
  - ▶ control flow using **if**, **for**, **while**, etc.
  - ▶ variable names instead of fixed registers and memory cells
  - ▶ memory management (more below)
- ▶ count dominant *elementary operations* (e.g. memory accesses) instead of all RAM instructions

In both cases: We *can* go to full detail where needed / desired.



# Pseudocode – Example

## RAM-Program

```
1 // Bubblesort
2 // Assume: R1 stores number N
3 // Assume: MEM[0..N) contains list of N numbers
4 R2 := R1;
5 R3 := R1 - 2;
6 R4 := MEM[R3];
7 R5 := R3 + 1;
8 R6 := MEM[R5];
9 if (R4 ≤ R6) goto line 12;
10 MEM[R3] := R6;
11 MEM[R5] := R4;
12 R3 := R3 - 1;
13 if (R3 ≥ 0) goto line 6;
14 R2 := R2 - 1;
15 if (R2 > 0) goto line 5;
16 // Done: MEM[0..N) sorted
```

## Pseudocode Algorithm

```
1 procedure bubblesort(A[0..N]):
2     for i := N, N - 1, ..., 1
3         for j := N - 2, N - 3, ..., 0
4             if A[j] > A[j + 1]:
5                 Swap A[j] and A[j + 1]
6             end if
7         end for
8     end for
```

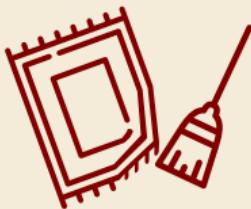
↔ much more **readable**

- ▶ closer to modern high-level programming languages
- ▶ **but:** only allow primitive operations that correspond to  $O(1)$  RAM instructions

↔ analysis

# Memory management & Pointers

- ▶ A random-access machine is a bit like a bare CPU . . . without any operating system
  - ~~> cumbersome to use
- ▶ All high-level programming languages / operating systems add *memory management*:
  - ▶ Instruction to *allocate* a contiguous piece of memory of a given size (like `malloc`).
    - ▶ used to allocate a new array (of a fixed size) or
    - ▶ a new object/record (with a known list of instance variables)
    - ▶ There's a similar instruction to `free` allocated memory again or an automated garbage collector.
  - ~~> A *pointer* is a memory address (i. e., the  $i$  of `MEM[i]`).
  - ▶ Support for procedures (a. k. a. functions, methods) calls including recursive calls
    - ▶ (this internally requires maintaining call stack)



We will mostly ignore *how* all this works here.

## 2.3 Asymptotics & Big-Oh

# Why asymptotics?

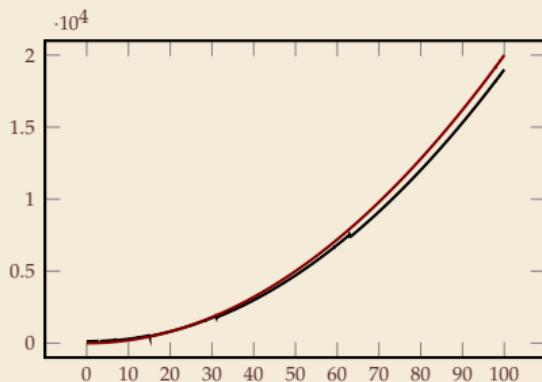
Algorithm analysis focuses on (the limiting behavior for infinitely) **large inputs**.

- ▶ abstracts from unnecessary detail
- ▶ simplifies analysis
- ▶ often necessary for sensible comparison

Asymptotics = approximation around  $\infty$

**Example:** Consider a function  $f(n)$  given by

$$2n^2 - 3n\lfloor \log_2(n+1) \rfloor + 7n - 3\lfloor \log_2(n+1) \rfloor + 120 \sim 2n^2$$



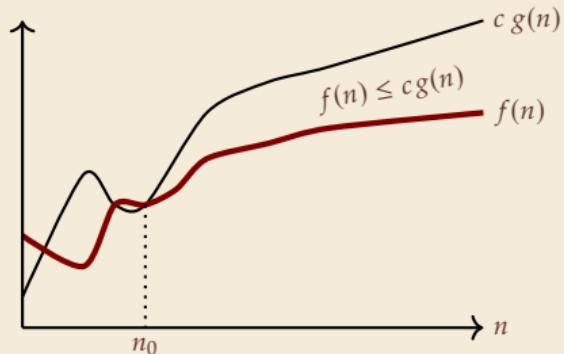
# Asymptotic tools – Formal & definitive definition

- if, and only if
- ▶ “Tilde Notation”:  $f(n) \sim g(n)$  iff  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 1$   
„ $f$  and  $g$  are *asymptotically equivalent*“
  - ▶ “Big-Oh Notation”:  $f(n) \in O(g(n))$  iff  $\left| \frac{f(n)}{g(n)} \right|$  is bounded for  $n \geq n_0$   
also write ‘=’ instead  
need supremum since limit might not exist!  
iff  $\limsup_{n \rightarrow \infty} \left| \frac{f(n)}{g(n)} \right| < \infty$
  - ▶ Variants:
    - ▶  $f(n) \in \Omega(g(n))$  iff  $g(n) \in O(f(n))$   
“Big-Omega”
    - ▶  $f(n) \in \Theta(g(n))$  iff  $f(n) \in O(g(n))$  and  $f(n) \in \Omega(g(n))$   
“Big-Theta”
  - ▶ “Little-Oh Notation”:  $f(n) \in o(g(n))$  iff  $\lim_{n \rightarrow \infty} \left| \frac{f(n)}{g(n)} \right| = 0$   
similarly:  $f(n) \in \omega(g(n))$  if  $\lim = \infty$

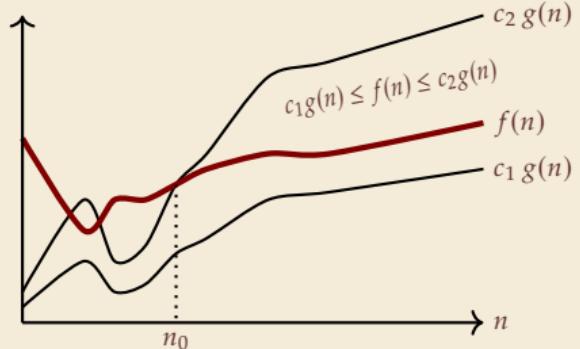
(Benefit of this definition: Works for any  $f, g : \mathbb{R} \rightarrow \mathbb{R}$  and is easy to generalize to limits other than  $n \rightarrow \infty$ )

# Asymptotic tools – Intuition

- $f(n) = O(g(n))$ :  $f(n)$  is **at most**  $g(n)$  up to constant factors and for sufficiently large  $n$



- $f(n) = \Theta(g(n))$ :  $f(n)$  is **equal to**  $g(n)$  up to constant factors and for sufficiently large  $n$



Plots can be misleading!

Example ↗

# Asymptotics – Example 1

Basic examples:

- ▶  $20n^3 + 10n \ln(n) + 5 \sim 20n^3 = \Theta(n^3)$
- ▶  $3 \lg(n^2) + \lg(\lg(n)) = \Theta(\log n)$
- ▶  $10^{100} = O(1)$

Use *wolframalpha* to compute/check limits, but also practice it with pen and paper!

# Asymptotics – Basic facts

Rules to work with Big-Oh classes:

- ▶  $f = \Theta(f)$  (reflexivity)
- ▶  $f = \Theta(g) \wedge g = \Theta(h) \implies f = \Theta(h)$
- ▶  $c \cdot f(n) = \Theta(f(n))$  for constant  $c \neq 0$
- ▶  $f \sim g \iff f = g \cdot (1 \pm o(1))$
- ▶  $\Theta(f) \cdot \Theta(g) = \Theta(f \cdot g)$
- ▶  $\Theta(f) + \Theta(g) = \Theta(f + g) = \Theta(\max\{f, g\})$     largest summand determines  $\Theta$ -class

# Asymptotics – Frequently encountered classes

Frequently used orders of growth:

- ▶ constant  $\Theta(1)$
- ▶ logarithmic  $\Theta(\log n)$       Note:  $a, b > 0$  constants  $\rightsquigarrow \Theta(\log_a(n)) = \Theta(\log_b(n))$
- ▶ linear  $\Theta(n)$
- ▶ linearithmic  $\Theta(n \log n)$
- ▶ quadratic  $\Theta(n^2)$
- ▶ cubic  $\Theta(n^3)$
- ▶ polynomial  $O(n^c)$  for some constant  $c$
- ▶ exponential  $O(c^n)$  for some constant  $c > 1$       Note:  $a > b > 0$  constants  $\rightsquigarrow b^n = o(a^n)$

# Asymptotics – Example 2

*Square-and-multiply algorithm*

for computing  $x^m$  with  $m \in \mathbb{N}$

Inputs:

- ▶  $m$  as binary number (array of bits)
- ▶  $n = \#$ bits in  $m$
- ▶  $x$  a floating-point number

---

```
1 def pow(x, m):
2     # compute binary representation of exponent
3     exponent_bits = bin(m)[2:]
4     result = 1
5     for bit in exponent_bits:
6         result *= result
7         if bit == '1':
8             result *= x
9     return result
```

---

- ▶ Cost:  $C = \#$  multiplications
- ▶  $C = n$  (line 6) +  $\#$ one-bits in binary representation of  $m$  (line 8)  
 $\rightsquigarrow n \leq C \leq 2n$

# Asymptotics with several variables

- ▶ **Example:** Algorithms on graphs with  $n$  vertices and  $m$  edges.
    - ▶ want to say: Algorithm  $A$  takes time  $\Theta(n + m)$ .
    - ▶ But what does that even mean formally?!
  - ▶ **Inconsistent and incompatible definitions used in the literature!**
  - ▶ **Here:**
    - ▶ (implicitly) always have a single “*main*” variable  $n$ : with  $n \rightarrow \infty$
    - ▶ all other variables are *functions* of  $n$ :  $m = m(n)$
    - ▶ must make *conditions* on functions explicit:  $m(n) \in \Omega(n)$  and  $m(n) \in O(n^2)$ .
      - ~~> Can make statements like
- $$O(n + m) \subseteq O(nm) \quad (n \rightarrow \infty, m \in \Omega(1))$$

## 2.4 Teaser: Maximum subarray problem

# Bring on the puzzles!

Time for a concrete example of algorithm design!

- ▶ we will illustrate the algorithm design process on a “toy problem”
- ▶ clean abstract problem, but nontrivial to solve!

## Maximum (sum) subarray problem

- ▶ Given:  $A[0..n]$  with  $A[i] \in \mathbb{Z}$  for  $0 \leq i < n$ .
- ▶ Abbreviate  $s(i, j) := \sum_{k=i}^{j-1} A[k]$
- ▶ Goal: Compute  $s := \max\{s(i, j) : 0 \leq i \leq j \leq n\}$  and a pair  $(i, j)$  with  $s = s(i, j)$ .  
will ignore that here; easy to modify algorithms

## Modeling decisions:

- ▶ input size: # numbers  $n$
- ▶ assume all integers (and sums) fit in  $\mathcal{O}(1)$  words
- ~~> count # additions as elementary operation

## Applications:

- ▶ largest gain of a stock  $A[i]$  price change on day  $i$
- ▶ signal detection in biological sequence analysis
- ▶ 2D generalization used in image analysis

# Template for Describing an Algorithm

## 1. Algorithmic Idea

Abstract idea that makes the algorithm work (prose)  
(an expert could fill in the rest from here)

## 2. Pseudocode

structured description of procedure including edge cases  
should be unambiguous and close to real code

## 3. Correctness proof

argument why the correct result is computed  
often uses induction and invariants

## 4. Algorithm analysis

analysis of the efficiency of the algorithm  
usually want  $\Theta$ -class of worst-case running time  
where interesting, also space usage

# Brute force approach

- ▶ Let's start with the simplest thinkable solution

## 1. ♣ Algorithmic Idea

try all contiguous subarrays  $A[i..j]$ )

## 2. </> Pseudocode

---

```
1   $s = 0$ 
2  for  $i = 0, \dots, n - 1$ 
3      for  $j = i, \dots, n$ 
4           $t = 0$ 
5          for  $k = i, \dots, j - 1$ 
6               $t = t + A[k]$ 
7          end for
8          if  $t > s$  then  $s := t$ 
9      end for
10 end for
```

---

## 3. ◎ Correctness proof

direct by definition of  $s$

## Maximal subarray problem

- ▶ Given:  $A[0..n)$  with  $A[i] \in \mathbb{Z}$  for  $0 \leq i < n$ .
- ▶ Abbreviate  $s(i, j) := \sum_{k=i}^{j-1} A[k]$
- ▶ Goal: Compute  $s := \max\{s(i, j) : 0 \leq i \leq j \leq n\}$  and a pair  $(i, j)$  with  $s = s(i, j)$ .

## 4. ─ Algorithm analysis

# additions

$$\begin{aligned} &= \sum_{i=0}^{n-1} \sum_{j=i}^n \sum_{k=i}^{j-1} 1 = \sum_{i=0}^{n-1} \sum_{j=i}^n (j - i) \\ &= \sum_{i=0}^{n-1} \sum_{j=0}^{n-i} j = \sum_{i=0}^{n-1} \frac{(n-i)(n-i+1)}{2} \\ &= \frac{1}{2} \sum_{i=1}^n i(i+1) = \frac{1}{2} \sum_{i=1}^n i^2 + \frac{1}{2} \sum_{i=1}^n i \\ &= \frac{n(n+1)(2n+1)}{12} + \frac{n(n+1)}{4} \\ &= \frac{n(n+1)(n+2)}{6} \sim \frac{1}{6}n^3 = \Theta(n^3) \end{aligned}$$

# Reusing sums

## 1. ♀ Algorithmic Idea

- ▶ brute force algorithm is unnecessarily wasteful!
- ▶ can use  $s(i, j) = s(i, j - 1) + A[j - 1]$

## 2. </> Pseudocode

---

```
1   $s = 0$ 
2  for  $i = 0, \dots, n - 1$ 
3       $t = 0$ 
4      for  $j = i + 1, \dots, n$ 
5           $t = t + A[j - 1]$ 
6          if  $t > s$  then  $s := t$ 
7      end for
8  end for
```

---



*Can we possibly do better?*

- ▶ There are  $\binom{n}{2} \sim \frac{1}{2}n^2$  different  $s(i, j) \dots$
- ~~~ Can't look at all of them

## 3. ◎ Correctness proof: as above

## 4. ── Algorithm analysis: $\sum_{i=0}^{n-1} \sum_{j=i+1}^n 1 = \frac{n(n+1)}{2} \sim \frac{1}{2}n^2 = \Theta(n^2)$ additions

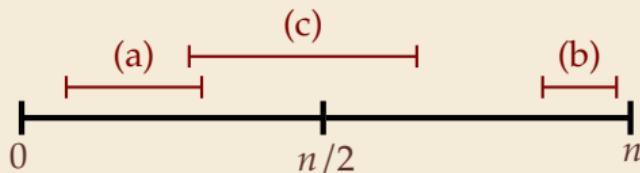
# A subquadratic solution

## 💡 Algorithmic idea:

Consider  $n/2$ -mark.

Only 3 options for optimal solution  $s(i, j)$ :

- (a)  $0 \leq i \leq j < \lceil \frac{n}{2} \rceil$  (left)
- (b)  $\lceil \frac{n}{2} \rceil \leq i \leq j \leq n$  (right)
- (c)  $i < \lceil \frac{n}{2} \rceil \leq j$  (straddle)



💡 optimal straddle easy to compute!

- ▶ independently find best left endpoint  $i$  for  $s(i, \lceil \frac{n}{2} \rceil)$  and best right endpoint  $j$  for  $s(\lceil \frac{n}{2} \rceil, j)$
- ▶ for (a) and (b), recurse on instance of half the size!

# A subquadratic solution – Pseudocode & Correctness

```
1 procedure findMaxSubarraySum( $A[\ell..r]$ ):  
2     if  $r - \ell \leq 0$   
3         return 0  
4     if  $r - \ell == 1$   
5         return max{0,  $A[\ell]$ }  
6      $m := \lceil (\ell + r)/2 \rceil$   
7      $s_{(a)} := \text{findMaxSubarraySum}(A[\ell, m])$   
8      $s_{(b)} := \text{findMaxSubarraySum}(A[m, r])$   
9     // Find left endpoint of straddle:  
10     $s_\ell := 0; t := 0$   
11    for  $i = m - 1, m - 2, \dots, \ell$   
12         $t := A[i] + t$   
13         $s_\ell := \max\{s_\ell, t\}$   
14    end for  
15    // Find right endpoint of straddle:  
16     $s_r := 0; t := 0$   
17    for  $j = m + 1, \dots, r$   
18         $t := t + A[j - 1]$   
19         $s_r := \max\{s_r, t\}$   
20    end for  
21     $s_{(c)} := s_\ell + s_r$   
22    return max{ $s_{(a)}, s_{(b)}, s_{(c)}$ }
```

## ◎ Correctness proof:

- ▶ Induction over  $n = r - \ell$ 
  - ▶ **basis:** for  $n \leq 1 \checkmark$
  - ▶ **hypothesis:** Assume findMaxSubarraySum returns correct result for all arrays of up to  $n - 1$  elements
  - ▶ **step:** For array of  $n \geq 2$  elements, distinguish cases (a), (b), (c)
    - (a) and (b)  $\rightsquigarrow$  IH  $\checkmark$
    - (c) “from inspection of the code”

# A subquadratic solution – Analysis

```
1 procedure findMaxSubarraySum( $A[\ell..r]$ ):  
2     if  $r - \ell \leq 0$   
3         return 0  
4     if  $r - \ell == 1$   
5         return max{0,  $A[\ell]$ }  
6      $m := \lceil (\ell + r)/2 \rceil$   
7      $s_{(a)} := \text{findMaxSubarraySum}(A[\ell, m])$   
8      $s_{(b)} := \text{findMaxSubarraySum}(A[m, r])$   
9     // Find left endpoint of straddle:  
10     $s_\ell := 0; t := 0$   
11    for  $i = m - 1, m - 2, \dots, \ell$   
12         $t := A[i] + t$   
13         $s_\ell := \max\{s_\ell, t\}$   
14    end for  
15    // Find right endpoint of straddle:  
16     $s_r := 0; t := 0$   
17    for  $j = m + 1, \dots, r$   
18         $t := t + A[j - 1]$   
19         $s_r := \max\{s_r, t\}$   
20    end for  
21     $s_{(c)} := s_\ell + s_r$   
22    return max{ $s_{(a)}, s_{(b)}, s_{(c)}$ }
```

## Algorithm analysis:

- ▶ Write  $n = r - \ell$
- ▶ # additions in non-recursive part:  
$$(m - \ell) + (r - m) + 1 = n + 1$$
- ▶ Write  $C(n)$  for total # additions for  $n$  elements
  - ~~ 
$$C(n) = C(\lceil \frac{n}{2} \rceil) + C(\lfloor \frac{n}{2} \rfloor) + n + 1$$
  - ▶ for  $n = 2^k$  for  $k \in \mathbb{N}_0$ , this simplifies to  
$$C(2^k) = 2C(2^{k-1}) + 2^k + 1$$
- ~~ 
$$C(n) \sim n \log_2(n)$$

## A lower bound

- **Theorem:** Every correct algorithm has a running time of  $\Omega(n)$ .

# An optimal algorithm

## 💡 Algorithmic idea:

In a clever sweep, we can compute best  $s(i, r)$  and best  $s(i, j)$  with  $i \leq j \leq r$  for all  $r$ .

## </> Pseudocode

---

```
1 procedure findMaxSubarraySum( $A[0..n]$ )
2     suffixMax := 0; globalMax := 0
3     for  $r = 1, \dots, n$ 
4         suffixMax := max{suffixMax +  $A[r - 1]$ , 0}
5         globalMax := max{globalMax, suffixMax}
6     return globalMax
```

---