QUICK RECAP OF FIRST LECTURE

- Introduction (no material here strictly needed)
- Practical information (definitely needed)
- Basics
  - RAM model (our computer model)
  - Pseudocode
  - Complexity analysis

> Focus on how running time scales as input size increases

TODAY: SEARCHING AND SORTING

Truly basic operations on data

Assume we have data records sorted by some key
- personal records sorted by CPR number
- car records sorted by license plates
- books sorted by title and author

Want to be able to
- search by key to find record
- sort records on key (to speed up search)

In our lectures, just focus on sorting the keys and ignore the records

SSI

SEARCH

$A[1] \leq A[2] \leq A[3] \leq \ldots$

Given
- sorted array A
- number x

Determine if exists index i s.t. $A[i] = x$

Think of number x as key, for instance
Entry $A[i]$ would then hold info about element with key x

1	2	3	4	5	6	7	8	9	10
1	4	5	7	9	10	11	14	16	20

IDEA 1

Just search through whole array

```
i := -1                                   1 op
j := 1                                    1 op
while (i<0 and j ≤ n)                     4 op
        if (A[j] == x)                    2 op
                i := j                    1 op
        j := j + 1                        2 op
return i                                  1 op
```

(the while-loop body: n times)

What is the RUNNING TIME?

Something like $2 + 4 \cdot n + 1 = 4n + 3$

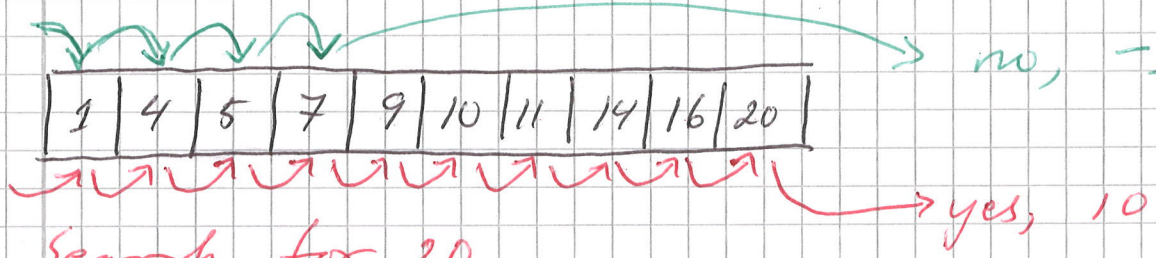
Scales like $\sim n$

FOCUS ON GROWTH OF HIGHEST-ORDER TERM

SS II

Can we be smarter?
Yes – use that array sorted
Search only until $A[j] > x$

Search for 6



Search for 20

IDEA / ALGORITHM 2

LINEÆR SØGNING

```
LINEAR-SEARCH (A, x)
    i := -1                                          1 op
    j := 1                                           1 op
    while (i<0 and j≤n and A[j]≤x)                   6 op
          if (A[j] == x)                             2 op
                i := j                               1 op
          j := j + 1                                 2 op
    return i                                         1 op
```

n times

What is the worst-case RUNNING TIME

Actually, happens only once

So we can be a little bit more careful

$2 + 9n + 1 + 1 = 10n + 4 = \boxed{\sim n}$

Is this better or worse than what we had before?

Better or worse?

Three possible answers

(1) WORSE — we have $\approx 10n$ operations in worst case instead of $\approx 8n$ operations

(2) BETTER — On average, will run through only half of array, so $\approx 10n/2 = 5n$ operations

(3) THE SAME — both scale like n

Strictly speaking, (1) is true

In practice, (2) would be true

But the most important conclusion is (3) — in the grand scheme of things, these two algorithms are not that different

Save maybe constant factor, but

WORST-CASE RUNNING TIME SCALES LIKE n

Or: is ORDER OF n

Notation $O(n)$

See textbook (and later lectures) for formal definition

Can we do FUNDAMENTALLY FASTER search?

BINARY SEARCH

BINÆR SØGNING

SS IV

Start right in the middle $mid := \lfloor \frac{1+n}{2} \rfloor$

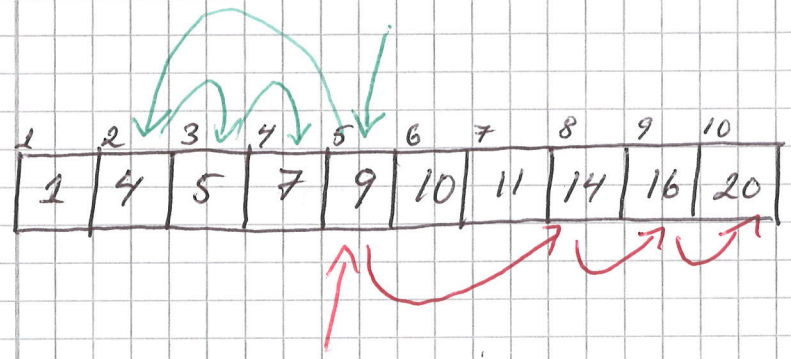
Rounding down

Often called "FLOOR"

If $A[mid] == x$ — done

If $A[mid] < x$ search in ~~left~~ right half

If $A[mid] > x$ search in ~~right~~ left half



SEARCH FOR 6

middle $\lfloor \frac{1+10}{2} \rfloor = 5$ $A[5] = 9 > 6 \Rightarrow$ LEFT

middle $\lfloor \frac{1+4}{2} \rfloor = 2$ $A[2] = 4 < 6 \Rightarrow$ RIGHT

middle $\lfloor \frac{3+4}{2} \rfloor = 3$ $A[3] = 5 < 6 \Rightarrow$ RIGHT

middle $\lfloor \frac{4+4}{2} \rfloor = 4$ $A[4] = 7 > 6$

Nothing ~~left~~ — done

SEARCH FOR 20

Middle $\lfloor (1+10)/2 \rfloor = 5$ $A[5] = 9 < 20 \Rightarrow$ RIGHT

Middle $\lfloor (6+10)/2 \rfloor = 8$ $A[8] = 14 < 20 \Rightarrow$ RIGHT

Middle $\lfloor (9+10)/2 \rfloor = 9$ $A[9] = 16 < 20 \Rightarrow$ RIGHT

Middle $\lfloor (10+10)/2 \rfloor = 10$ $A[10] = 20$ SUCCESS!

```
BINARY-SEARCH ( A, lo, hi, x )
    if ( lo > hi )
        return -1
    else
        mid := ⌊(lo + hi) / 2⌋
        if ( A[mid] == x )
            return mid
        else if ( A[mid] < x )
            return BINARY-SEARCH ( A, mid+1, hi, x )
        else // A[mid] > x
            return BINARY-SEARCH ( A, lo, mid-1, x )
```

Recursive algorithm

Something like max 15-20 operations per recursive call — constant

Time complexity = # recursive calls R

But first... CORRECTNESS?

INVARIANT If ~~exists~~ i s.t. $A[i] = x$ then $lo \leq i \leq hi$

True at beginning for binary search call with $lo = 1$; $hi = n$

Since array A sorted

if $A[mid] < x$, then x must be in upper half

if $A[mid] > x$, then x must be in lower half

And hi - lo decreases strictly for each call, so algorithm will terminate

SIDE NOTE: Apparently, many search often coded up incorrectly in practice
Termination criterion tends to fail

TIME COMPLEXITY

Constant # operations per recursive call
How many calls R ?

When array size = 1, next call will terminate (WHY?)

If array size = n, size for recursive call is $\leq \lceil \frac{n-1}{2} \rceil \leq n/2$

ROUNDING UP; "CEILING"

That is, after R calls, size $n / 2^R$

Want

$$n / 2^R \leq 1$$

$$n \leq 2^R$$

$$\log_2 n \leq R$$

In this course, logarithms will be base-2 unless otherwise stated

This is what we need in computer science — but only constant-factor difference

So time complexity of binary search scales like $\sim \boxed{\log n}$

Difference linear search — binary search?

n	$\log n$
1000	10
1,000,000	20
1,000,000,000	30

The POWER OF ASYMPTOTICS — no amount of optimization of linear search can beat binary search for large data sets

COMPLEXITY ANALYSIS
Focus on how algorithm running time scales when input size increases

SORTING

Input Array A of length n with elements that can be compared

Output A still contains same elements, but sorted so that $A[1] \leq A[2] \leq A[3] \leq \ldots$

16	11	9	1	4	14	5	7	20	10

Fundamental task in data processing

Makes it possible to

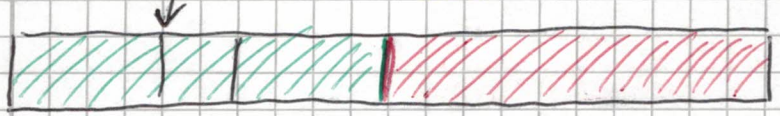
- search efficiently
- identify duplicates
- analyze statistical properties
- Et cetera...

INSERTION SORT (INDSÆTTELSESORTERING)

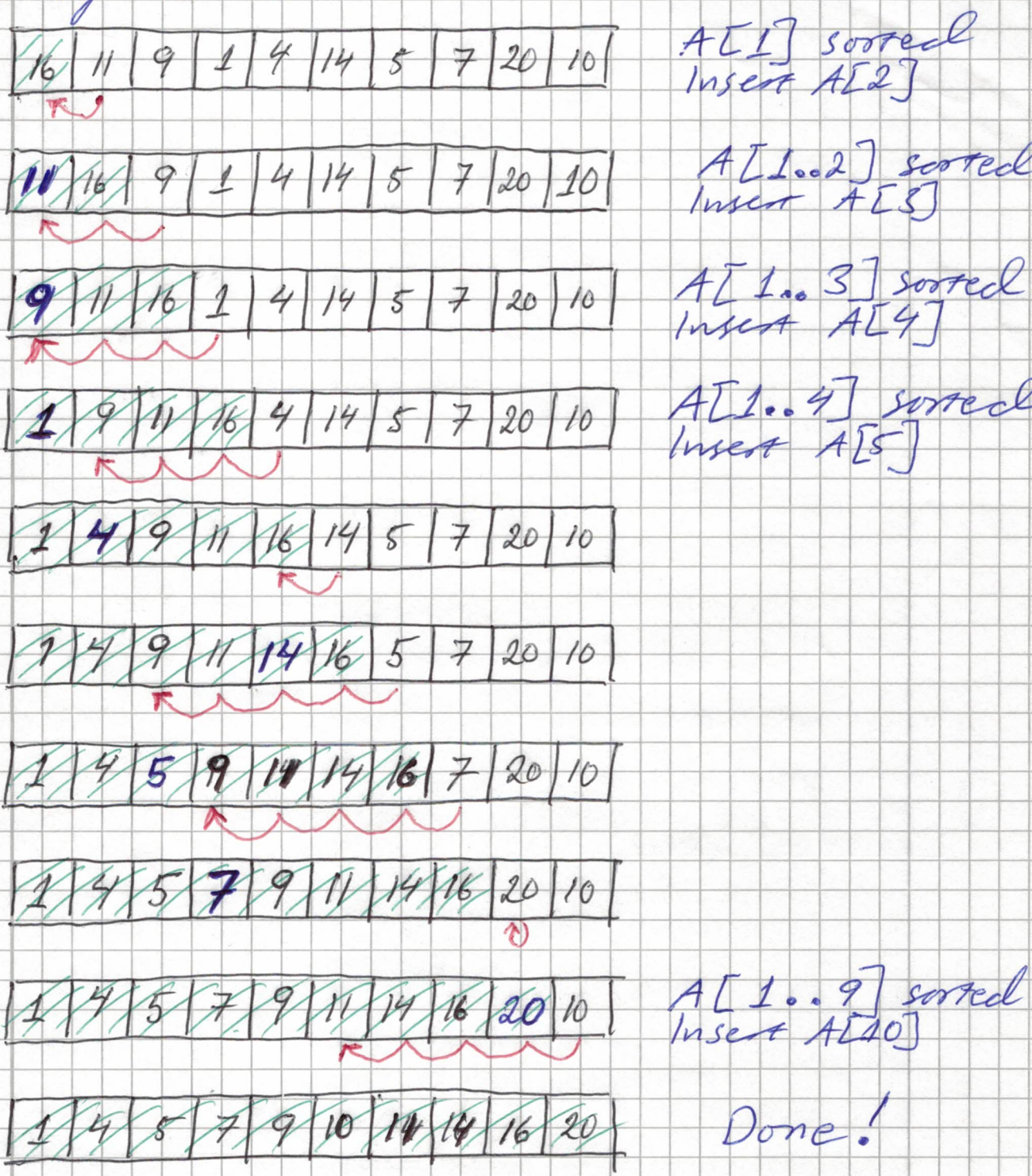
Maintain sorted part and unsorted part of array

One by one, take elements from unsorted part...

... and insert in correct place



Where to start? A list of size 1 is always sorted



QUESTIONS about the overall idea?

Now need to

- provide precise description of algorithm
- argue correctness
- analyze time complexity

```
INSERTION SORT (A)
    n := length(A)                          1 op
    for (i := 2 upto n)                     ≈ n times
        j := i                              1 op
        while (j > 1 and A[j-1] > A[j])     ≤ i-1 times
            tmp := A[j-1]                   constant
            A[j-1] := A[j]                  # operations
            A[j] := tmp
            j := j - 1
```

CORRECTNESS

Invariant: At top of ~~for~~ ~~while~~ loop, $A[1..i-1]$ is sorted

- True when we start
- Given that $A[1..i-1]$ sorted, while loop shifts $A[i]$ into correct position
- So at end of while loop $A[1..i]$ sorted

Hence, after final iteration of for loop $A[1..n]$ is sorted, as desired. ✓

TIME COMPLEXITY

For some constant c we get # operations

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

How large is this?

$$\sum_{i=1}^{n-1} i \leq \sum_{i=1}^{n} n = n^2$$

$$1 + 2 + \cdots + \left(\frac{n}{2} - 1\right) + \frac{n}{2} + \left(\frac{n}{2} + 1\right) + \cdots + (n-2) + (n-1)$$

$$\geq \frac{n}{2} \cdot \frac{n}{2} = \frac{n^2}{4}$$

So $\sum_{i=1}^{n-1} i$ scales like n^2

Time complexity of insertion sort scales like $\boxed{\sim n^2}$

Is this good or bad?

Polynomial time, so good

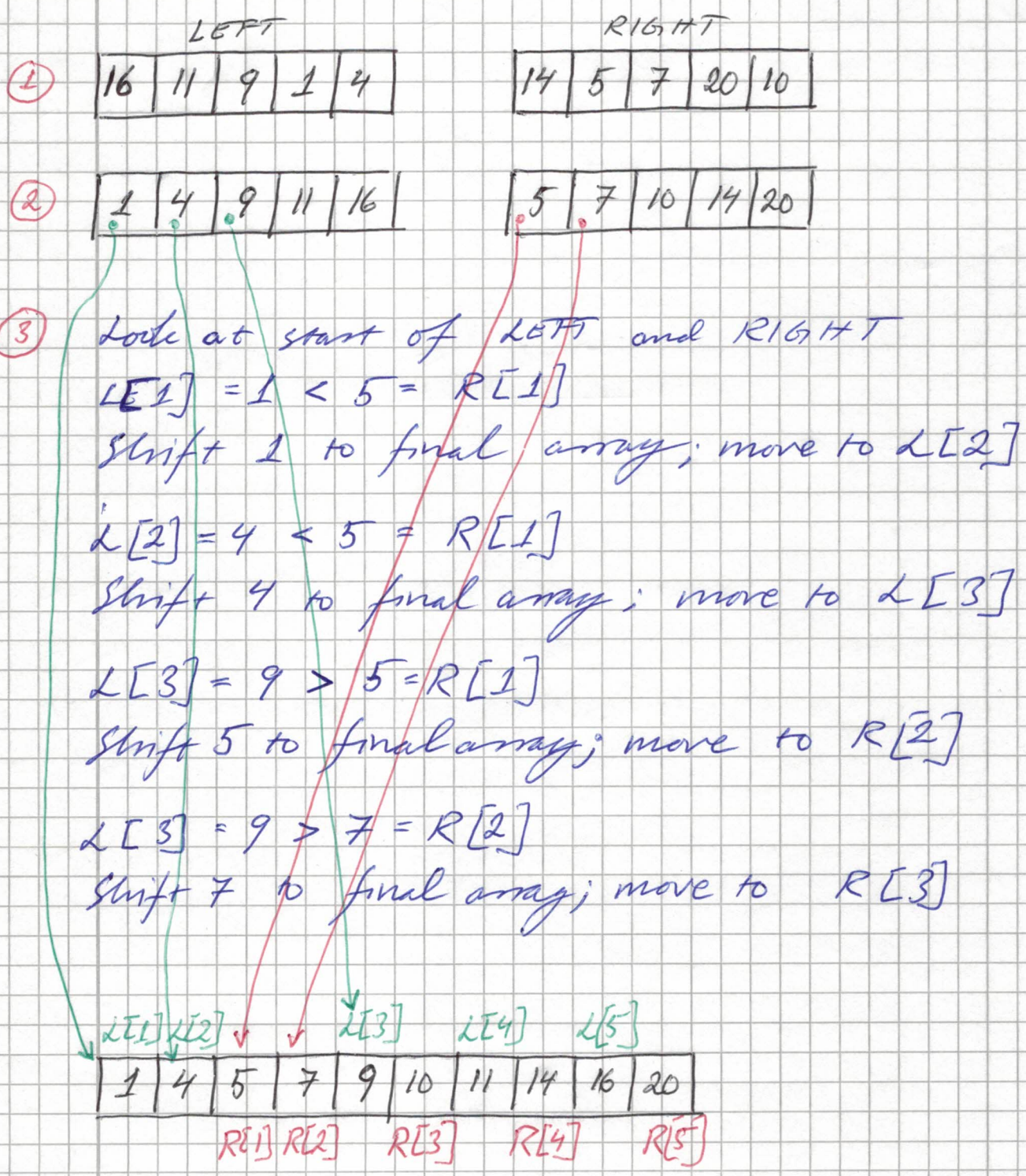
But for large datasets (10,000 or 100,000 elements) even quadratic time is fairly slow...

Can we do better?

MERGE SORT (FLETTESORTERING)

Idea:

1. Split the array in halves
2. Sort halves separately
3. Merge into one sorted list!



Let us ignore STEP 2 for now and focus on merge

```
MERGE ( L, R, M )
  n := length(L)
  m := length(R)
  M := make sure array of length n+m
  Add L[n+1] := ∞        larger than values allowed in array
  Add R[m+1] := ∞
  i := 1
  j := 1
  for (k := 1 upto n+m)                    n+m iterations
      if (L[i] ≤ R[j])
          M[k] := L[i]
          i := i+1
      else
          M[k] := R[j]
          j := j+1
```

constant # ops

CORRECTNESS OF MERGE

Essentially formalize the idea we had in the example

Find invariant; argue that each iteration in for loop maintains it

See pages 31 - 34 in CLRS

Exercise Our pseudo-code above is slightly different from CLRS — rewrite CLRS proof to work for pseudo-code above

TIME COMPLEXITY

$\sim (n+m)$ linear in size of inputs

Fine, but what about step 2 ?!
Use same idea recursively ?
But where does recursion end ?
Answer: Array of size 1 is sorted

```
MERGESORT (A)
    if length(A) > 1
        mid := ⌊(1 + length(A)) / 2⌋
        L := new array of size mid
        R := new array of size length(A) − mid
        for (i := 1 upto mid)
            L[i] := A[i]
        for (i := mid + 1 upto length(A))
            R[i − mid] := A[i]
        MERGESORT (L)
        MERGESORT (R)
        MERGE (L, R, A)
```

linear time

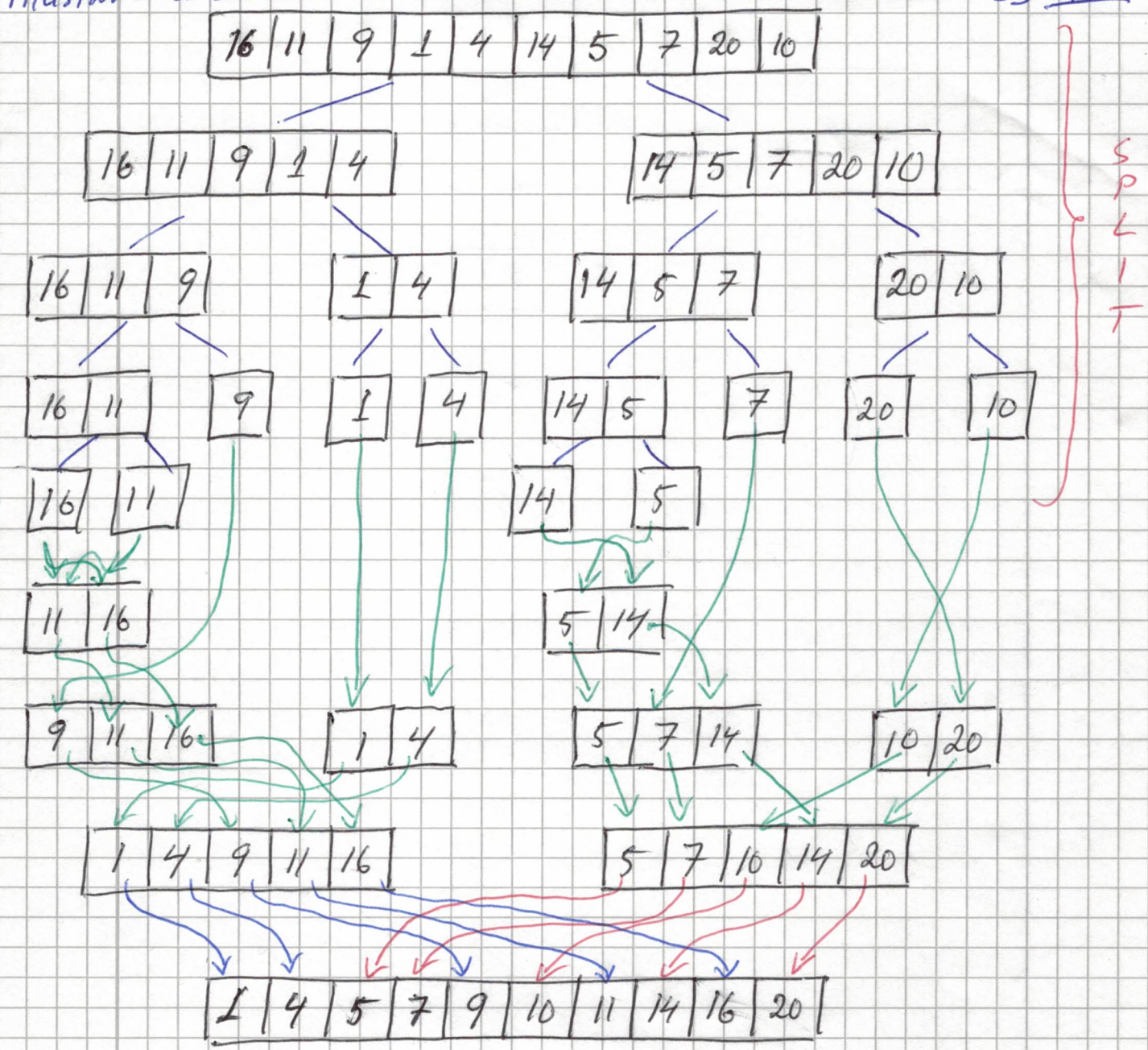
linear time

CORRECTNESS

Always terminates — arrays get smaller and smaller

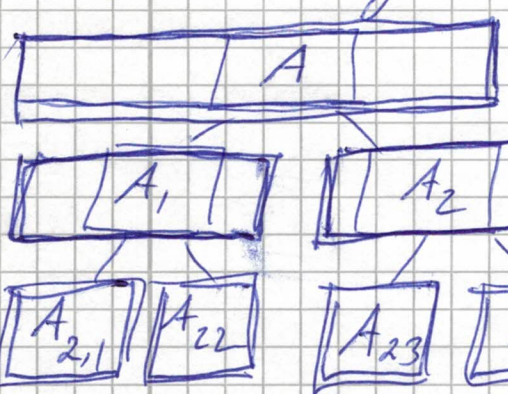
If mergesort correct on arrays of length $< n$, then recursive calls will sort L and R
Then merge method will sort A correctly
Formally, argue by INDUCTION
(which we will see soon in the course)

Illustrate execution with recursion tree



TIME COMPLEXITY

How many recursive calls? ~ log n



Whole array time ~ n

two halves ~ n/2 + n/2

Quarters ~ n/4 + n/4 + n/4 + n/4

Work at every level of recursion ~ n

levels of recursion ~ log n

Total time ~ n log n

Merge sort is an example of the paradigm DIVIDE AND CONQUER (DEL OG HERSK)

		MERGE SORT
DIVIDE	Split problem into smaller parts	Split array in two
CONQUER	Solve smaller problems recursively	Sort recursively
COMBINE	Construct solution to original problem from solutions to subproblems	Merge sorted subarrays into sorted full array

ASYMPTOTIC NOTATION

Given input of size n, how does running time of algorithm scale with n in the worst case?

Intuitively: Find highest-order term; ignore rest
Shave off constant factors

But what is the FORMAL DEFINITION?

Want to say "big-oh of $f(n)$"

"Running time is $O(f(n))$"

if it scales no worse than $f(n)$

What does this mean? Say running time $R(n)$

(1) $R(n)$ is $O(f(n))$ if $R(n) \leq f(n)$

NO! $5n^2 + 10n - 8$ should be $O(n^2)$ according to intuition above

(2) $R(n)$ is $O(f(n))$ if there is some constant K such that $R(n) \leq K \cdot f(n)$

STILL NOT QUITE HAPPY! What about running time like $5n^2 + 1{,}000{,}000n$? Seems we should have $K \approx 5$, but need $n \geq 1{,}000{,}000$ before n^2 starts dominating $1{,}000{,}000n$

$R(n)$ is $O(f(n))$ if
- exists large enough constant K
- exists possible constant n_0

such that for all $n \geq n_0$

$$0 \leq R(n) \leq K \cdot f(n)$$

Is $5n^2 + 1{,}000{,}000\,n$ now $O(n^2)$?

YES! Choose, e.g.,

$K = 6$ $\quad n_0 = 1{,}000{,}000$

For $n \geq 1{,}000{,}000$ we have

$$5n^2 + 1{,}000{,}000\,n \leq 6n^2$$

Estimate $O(f(n))$ does NOT have to be tight!

$5n^2 + 1{,}000{,}000\,n$ is also $O(n^{42})$ (WHY?)

But try to choose $f(n)$ as small as possible to be as informative as possible

INTUITION

$R(n)$ is $O(f(n))$ if

"$R(n) \leq f(n)$ for large enough n except for constant factors"

Two additional pieces of notation

"$R(n)$ is at least as bad as $f(n)$"

$R(n)$ is $\Omega(f(n))$ "big omega of $f(n)$"

$R(n)$ is $\Omega(f(n))$ if
- exists ~~large~~ small enough constant $K > 0$
- exists large enough constant n_0

such that for all $n \geq n_0$

$$0 \leq K \cdot f(n) \leq R(n)$$

Maybe merge sort analysis was hard to understand, but we are sure it required at least linear time; so, merge sort is $\Omega(n)$

Here, try to choose $f(n)$ as large as possible — more informative

$R(n)$ is $\Theta(f(n))$ if
- $R(n)$ is $O(f(n))$
- $R(n)$ is also $\Omega(f(n))$

That is, an estimate $\Theta(f(n))$ gets the asymptotic growth exactly right.

"The bound is tight except for constant factors"

Most of our bounds will be tight $\Theta(f(n))$ bounds.

But computer scientists are a bit lazy — quite often write $O(f(n))$ even when we know $\Theta(f(n))$

Maybe because for algorithms we care most about UPPER BOUND on running time — want to know that algorithm runs fast