Algorithms and Data Structures Conditional Course

Lecture 2

Runtime Analysis, Sorting II

Fabian Kuhn
Algorithms and Complexity

UNI FREIBURG

Runtime Analysis I

UNI FREIBURG

- How can we analyze the runtime of an algorithm?
  - runtime is different on different computers…
  - depends on compiler, programming language, etc.
- We need an abstract measure to express the runtime
- **Idea**: Count the number of (basic) operations
  - instead of directly measuring the time
  - the number of basic operations is independent of computer, compiler
  - It is a good measure for the runtime if all basic operations require about the same time.

Basic Operations

What is a basic operation?

- Simple arithmetic operations / comparisons
  - +, -, *, /, % (mod), <, >, ==, …
- One memory access
  - reading or writing a variable
  - not clear if this is really a basic operation?
- One function call
  - Of course only jumping to the function code
- **Intuitively:** one line of program code
- **Better:** one line of assembly language code
- **Even better (?):** one processor cycle

- **We will see:** It is only important that the number of basic opertions is roughly proportional to the actual running time.

RAM Model

RAM = Random Access Machine

- Standard model to analyze algorithms!
- Basic operations (as "defined") all require one time unit
- In particular, all memory accesses are equally expensive:

  Each memory cell (1 machine word) can be read or written in 1 time unit

  - In particular ignores memory hierarchies
  - In most cases, it is however a reasonable assumption

- There are alternative abstract models:
  - to explicitly capture memory hierarchies
  - for huge data volumes (cf. big data)
    - e.g.: streaming-models: memory has to be read sequentially
  - for distributed / parallel architectures
    - memory access can be local or over the network…

Runtime analysis II

UNI FREIBURG

So far: Number of basic operations is proportional to the runtime

- We can also achieve this without counting the basic operations exactly!

Simplification 1: We only calculate an upper bound (or a lower bound) on the number of basic operations

  – such that the upper / lower bound is still proportional to the runtime…

- No. of basic op. can depend on several properties of the input
  – Size/length of input, but, e.g., for sorting also the ordering in the input

Simplification 2: Most important parameter is input size n
We always consider the runtime $T(n)$ as a function of n.

  – And we ignore other properties of the input

Selection Sort: Analysis

```
SelectionSort(A):
  for i=0 to n-2 do
     minIdx = i
     for j=i to n-1 do
        if A[j] < A[minIdx] then
           minIdx = j
     swap(A[i], A[minIdx])
```

minIdx = i: $\le c_1$

if A[j] < A[minIdx] then / minIdx = j: $\le c_2$

swap(A[i], A[minIdx]): $\le c_3$

#basic op. $\le c \cdot \underbrace{\text{\#inner for loop iterations}}_{x(n)}$

$$x(n) = \sum_{i=0}^{n-2} (n-i) = \sum_{h=2}^{n} h \le \sum_{h=1}^{n} h = \frac{n(n+1)}{2} \le n^2$$

Selection Sort: Analysis

```
SelectionSort(A):
  for i=0 to n-2 do
    minIdx = i
    for j=i to n-1 do
      if A[j] < A[minIdx] then
        minIdx = j
    swap(A[i], A[minIdx])
```

- `minIdx = i`: $\leq c_1$
- `if A[j] < A[minIdx] then minIdx = j`: $\leq c_2$, $\geq c_2'$
- `swap(A[i], A[minIdx])`: $\leq c_3$

$$\underbrace{\#\text{basic op.}}_{T(n)} \leq c \cdot \underbrace{\#\text{inner for loop iterations}}_{x(n) \leq n^2}$$

Runtime $T(n) \leq c \cdot n^2$

Selection Sort: Analysis

$\boldsymbol{T(n)}$: Number of basic operations of Selection Sort algorithms for arrays of length n

Lemma: *There is a* ***constant*** $\boldsymbol{c_U > 0}$, *such that* $\boldsymbol{T(n) \leq c_U \cdot n^2}$

Lemma: *There is a* ***constant*** $\boldsymbol{c_L > 0}$, *such that* $\boldsymbol{T(n) \geq c_L \cdot n^2}$

Runtime analysis

UNI FREIBURG

Summary

- We can only obtain a value that is proportional to the runtime.
- However, we also do not want anything else:
  - Analysis should be independent of computer / compiler / etc.
  - We want to have statements that are valid in 10 / 100 /… years

- We will always get statements of the following form:

  There is a constant C, such that

$$T(n) \leq C \cdot f(n) \quad \text{or} \quad T(n) \geq C \cdot f(n)$$

- The Big-O notation allows to simplify / generalize this kind of statements…

Big-O Notation

- Formalism to describe the asymptotic growth of functions.
  - For formal definitions: see next slide…

- There is a const. $C > 0$, s. t. $T(n) \leq C \cdot f(n)$ becomes:

$$T(n) \in O(f(n))$$

- There is a const. $C > 0$, s. t. $T(n) \geq C \cdot g(n)$ becomes:

$$T(n) \in \Omega(g(n))$$

- For Selection Sort:

$$\left.\begin{array}{l} \boldsymbol{T(n) \in O(n^2)} \\ \boldsymbol{T(n) \in \Omega(n^2)} \end{array}\right\} \boldsymbol{T(n) \in \Theta(n^2)}$$

Big-O Notation : Definitions

$$\boldsymbol{O}\big(\boldsymbol{g}(\boldsymbol{n})\big) \coloneqq \{\boldsymbol{f}(\boldsymbol{n}) \mid \exists \boldsymbol{c}, \boldsymbol{n_0} > \mathbf{0}\ \forall \boldsymbol{n} \geq \boldsymbol{n_0} : \boldsymbol{f}(\boldsymbol{n}) \leq \boldsymbol{c} \cdot \boldsymbol{g}(\boldsymbol{n})\}$$

- Function $f(n) \in O(g(n))$, if there are constants c and n_0 s. t. $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$

$$\boldsymbol{\Omega}\big(\boldsymbol{g}(\boldsymbol{n})\big) \coloneqq \{\boldsymbol{f}(\boldsymbol{n}) \mid \exists \boldsymbol{c}, \boldsymbol{n_0} > \mathbf{0}\ \forall \boldsymbol{n} \geq \boldsymbol{n_0} : \boldsymbol{f}(\boldsymbol{n}) \geq \boldsymbol{c} \cdot \boldsymbol{g}(\boldsymbol{n})\}$$

- Function $f(n) \in \Omega(g(n))$, if there are constants c and n_0 s. t. $f(n) \geq c \cdot g(n)$ for all $n \geq n_0$

$$\boldsymbol{\Theta}\big(\boldsymbol{g}(\boldsymbol{n})\big) \coloneqq \boldsymbol{O}\big(\boldsymbol{g}(\boldsymbol{n})\big) \cap \boldsymbol{\Omega}\big(\boldsymbol{g}(\boldsymbol{n})\big)$$

- Function $f(n) \in \Theta(g(n))$, if there are constants c_1, c_2 and n_0 s. t. $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ for all $n \geq n_0$, resp. if $f(n) \in O(n)$ and $f(n) \in \Omega(n)$

Big-O Notation : Definitions

$$o(g(n)) := \{f(n) \mid \forall c > 0\ \exists n_0 > 0\ \forall n \geq n_0 : f(n) \leq c \cdot g(n)\}$$

- Function $f(n) \in o(g(n))$, if for all constants $c > 0$, we have $f(n) \leq c \cdot g(n)$ (for sufficiently large n, indep. of c)

$$\omega(g(n)) := \{f(n) \mid \forall c > 0\ \exists n_0 > 0\ \forall n \geq n_0 : f(n) \geq c \cdot g(n)\}$$

- Function $f(n) \in \omega(g(n))$, if for all constants $c > 0$, we have $f(n) \geq c \cdot g(n)$ (for sufficiently large n, indep. of c)

In particular:

$$f(n) \in o(g(n)) \quad \Longrightarrow \quad f(n) \in O(g(n))$$

$$f(n) \in \omega(g(n)) \quad \Longrightarrow \quad f(n) \in \Omega(g(n))$$

Big-O Notation : Intuitively

$f(n) \in O(g(n))$:

- $f(n)$ " $\leq$ " $g(n)$, asymptotically…
- $f(n)$ asymptotically grows at most as fast as $g(n)$

$f(n) \in \Omega(g(n))$:

- $f(n)$ " $\geq$ " $g(n)$, asymptotically…
- $f(n)$ asymptotically grows at least as fast as $g(n)$

$f(n) \in \Theta(g(n))$:

- $f(n)$ " $=$ " $g(n)$, asymptotically…
- $f(n)$ asymptotically grows equally fast as $g(n)$

Big-O Notation : Intuitively

$f(n) \in o(g(n))$:

- $f(n)$ " $<$ " $g(n)$, asymptotically…
- $f(n)$ asymptotically grows slower than $g(n)$

$f(n) \in \omega(g(n))$:

- $f(n)$ " $>$ " $g(n)$, asymptotically…
- $f(n)$ asymptotically grows faster than $g(n)$

If $f(n)$ and $g(n)$ grow monotonically, we have:

$$f(n) \in o(g(n)) \quad \Leftrightarrow \quad f(n) \notin \Omega(g(n))$$

$$f(n) \in \omega(g(n)) \quad \Leftrightarrow \quad f(n) \notin O(g(n))$$

Definition by Limits (simplified)

The following definitions hold for monotonically growing functions

$$f(n) \in O\big(g(n)\big), \qquad \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$$

$$f(n) \in \Omega\big(g(n)\big), \qquad \lim_{n\to\infty} \frac{f(n)}{g(n)} > 0$$

$$f(n) \in \Theta\big(g(n)\big), \qquad 0 < \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$$

$$f(n) \in o\big(g(n)\big), \qquad \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

$$f(n) \in \omega\big(g(n)\big), \qquad \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

Big-O Notation : Remarks

Writing Convention:

- $O(g(n)), \Omega(g(n)), \ldots$ are sets (of functions)
- Correct way of writing (in principle): $f(n) \in O(g(n))$
- Very common way of writing: $f(n) = O(g(n))$

Examples:

- $T(n) = O(n^2)$ instead $T(n) \in O(n^2)$
- $T(n) = \Omega(n^2)$ instead $T(n) \in \Omega(n^2)$
- $f(n) = n^2 + O(n)$:

$$f(n) \in \{g(n) : \exists h(n) \in O(n) \text{ s.t. } g(n) = n^2 + h(n)\}$$

- $a(n) = (1 + o(1)) \cdot b(n)$

Big-O Notation : Remarks

Writing Convention:

- $O(g(n)), \Omega(g(n)), \ldots$ are sets (of functions)
- Correct way of writing (in principle): $f(n) \in O(g(n))$
- Very common way of writing: $f(n) = O(g(n))$

Asymptotic Behavior of General Limits:

- Same notation is used more generally, e.g., $f(x)$ for $x \to 0$
- E.g., Taylor approx.: $e^x = 1 + x + O(x^2)$, or $e^x = 1 + x + o(x)$

Alternative Definition for $\Omega(g(n))$:

$g(n) = n^2$, $f(n) = \begin{cases} n^2, & n \text{ even} \\ 1, & n \text{ odd} \end{cases}$

$$\Omega(g(n)) := \{f(n) \mid \exists c, n_0 > 0\ \forall n \geq n_0 : f(n) \geq c \cdot g(n)\}$$

$$\Omega(g(n)) := \{f(n) \mid \exists c > 0\ \forall n_0 > 0\ \exists n \geq n_0 : f(n) \geq c \cdot g(n)\}$$

- We will use the 1st definition
- The two definitions are only different for non-monotonic functions

Big-O Notation : Examples

Selection Sort:

- Runtime $T(n)$, there are constants $c_1, c_2 : c_1 n^2 \leq T(n) \leq c_2 n^2$

$$T(n) \in O(n^2), \qquad T(n) \in \Omega(n^2), \qquad T(n) \in \Theta(n^2)$$

- $T(n)$ grows more than linear in n: $T(n) \in \omega(n)$

Further examples:

- $f(n) = 10n^3,\ g(n) = n^3/1000$: $f(n) \in \Theta(g(n))$
- $f(n) = e^n,\ g(n) = n^{100}$: $f(n) \in \omega(g(n))$
- $f(n) = n/\log_2 n,\ g(n) = \sqrt{n}$: $f(n) \in \omega(g(n))$
- $f(n) = n^{1/256},\ g(n) = 10 \ln n$: $f(n) \in \omega(g(n))$
- $f(n) = \log_{10} n, g(n) = \log_2 n$: $f(n) \in \Theta(g(n))$
- $f(n) = n^{\sqrt{n}}, g(n) = 2^n$: $f(n) \in o(g(n))$

$$\lim_{n\to\infty} \frac{e^n}{n^{100}} \to \infty$$

$$\frac{f(n)}{g(n)} = \frac{\sqrt{n}}{\log_2 n} = \frac{2^{t/2}}{t}$$

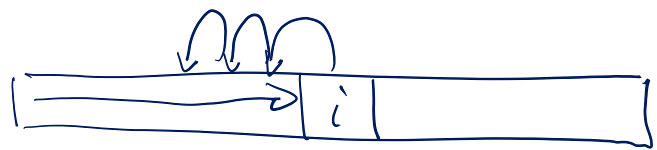
$$\log_{10} n = \frac{\log_2 n}{\log_2 10}$$

$$\log\left(n^{\sqrt{n}}\right) = \sqrt{n} \cdot \log n, \log(2^n) = n$$

Analysis Insertion Sort

```
InsertionSort(A):
  for i = 1 to n-1 do
     // prefix A[1..i] is already sorted
     pos = i
     while (pos > 0) and (A[pos] < A[pos-1]) do
        swap(A[pos], A[pos-1])
        pos = pos - 1
```

$X(n)$: #while loop iter.



$$x(n) \leq \sum_{i=1}^{n-1} i = O(n^2)$$

$$T(n) = O(n^2)$$

$$x(n) \geq \sum_{i=1}^{n-1} 1 = \Omega(n)$$

Worst case, best case, average case

Worst Case Analysis

- Analyze runtime $T(n)$ for a worst possible input of size n
- Important / standard way of analyzing algorithms

Best Case Analyse

- Analyze runtime $T(n)$ for a best possible input of size n
- Usually not very interesting…

Average Case Analyse

- Analyze runtime $T(n)$ for a typical input of size n
- Problem: what is a typical input?
  - Standard approach: use a random input
  - Not clear, how close real inputs and random inputs are…
  - Possible alternative: smoothed analysis (we will not look at this)

How good is quadratic runtime?

Quadratic = 2x as large input → 4x as long runtime

– For large n, this already seems to grow quite fast…

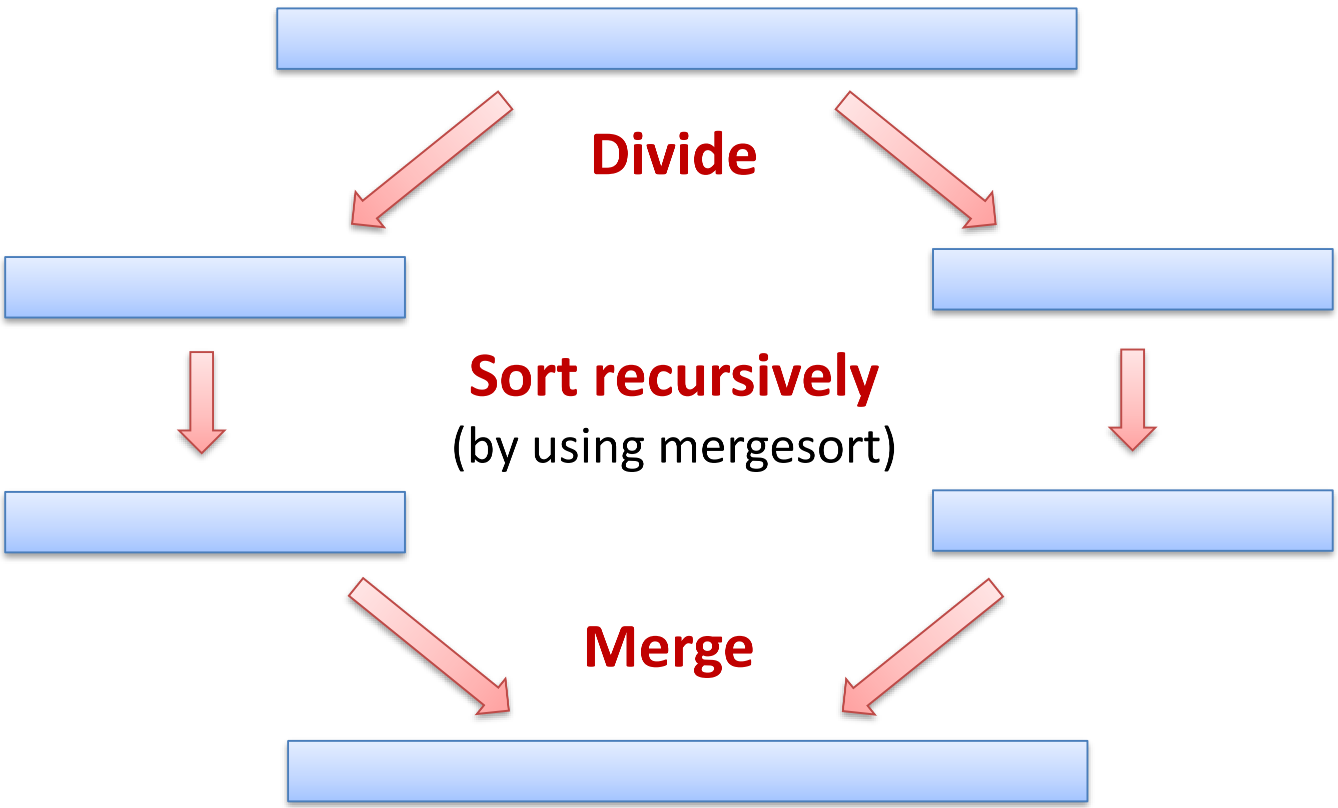
Example calculation:

- Assume that the number of basic operations $T(n) = n^2$
- Additionally, assume there is 1 basic operation per processor cycle
- For a 1Ghz processor, we get 1 ns per basic operation

Input size n	4 bytes per number	Runtime $T(n)$
10^3 numbers	$\approx 4\text{KB}$	$10^{3\cdot 2} \cdot 10^{-9}\ \text{s} = 1\ \text{ms}$
10^6 numbers	$\approx 4\text{MB}$	$10^{6\cdot 2} \cdot 10^{-9}\ \text{s} = 16.7\ \text{min}$
10^9 numbers	$\approx 4\text{GB}$	$10^{9\cdot 2} \cdot 10^{-9}\ \text{s} = 31.7\ \text{years}$

too slow for large problems!

Analysis Merge Sort

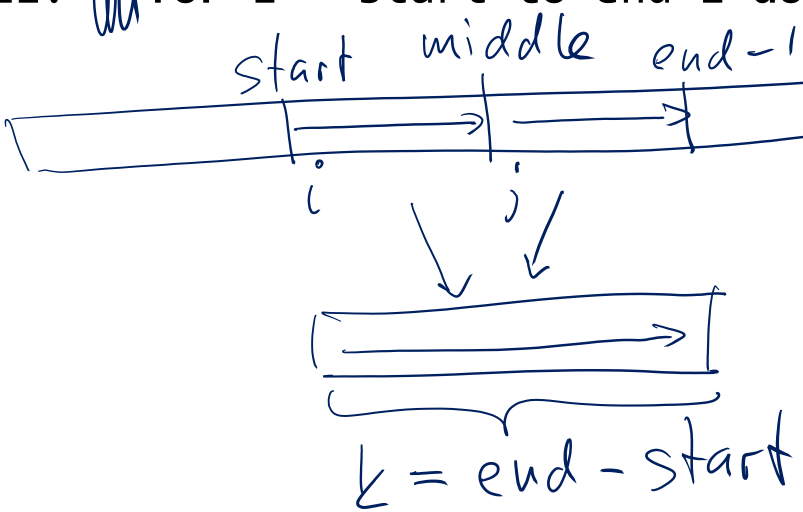


- Divide is trivial → cost $O(1)$
- Recursive sorting: We will look at this...
- Merge: We will look at this first...

Analysis Merge Step

```
MergeSortRecursive(A, start, end, tmp)                // sort A[start..end-1]
    ⋮
   pos = start; i = start; j = middle
   while (pos < end) do
      if (i < middle) and (A[i] < A[j]) then
         tmp[pos] = A[i]; pos++; i++
      else
         tmp[pos] = A[j]; pos++; j++
   for i = start to end-1 do A[i] = tmp[i]
```

$O(k)$ (while loop), $O(1)$ (loop body), $O(k)$ (for loop)



runtime: $O(k)$

Analysis Merge Sort

Runtime $T(n)$ consists of:

- Divide and Merge: $O(n)$
- 2 recursive calls to sort $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$ elements

Recursive formulation of $T(n)$:

- There is a constant $b > 0$, s. t.

$$T(n) \leq T\left(\left\lceil\frac{n}{2}\right\rceil\right) + T\left(\left\lfloor\frac{n}{2}\right\rfloor\right) + b \cdot n, \qquad \mathrm{T}(1) \leq b$$

- We simplify a bit and ignore all the rounding:

$$\boldsymbol{T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + b \cdot n, \qquad T(1) \leq b}$$

assume : n power of 2

Analysis Merge Sort

$$T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + b \cdot n, \;\; T(1) \leq b$$

Let's just try and see what we get…

$$T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + b \cdot n \qquad\qquad T\left(\frac{n}{2}\right) \leq 2 \cdot T\left(\frac{n}{4}\right) + b \cdot \frac{n}{2}$$

$$\leq 4 \cdot T\left(\frac{n}{4}\right) + 2 \cdot b \cdot \frac{n}{2} + b \cdot n$$

$$= 4 \cdot T\left(\frac{n}{4}\right) + 2 \cdot b \cdot n$$

$$\leq 4\left(2 \cdot T\left(\frac{n}{8}\right) + b \cdot \frac{n}{4}\right) + 2 \cdot b n$$

$$= 8 \cdot T\left(\frac{n}{8}\right) + 3 \cdot b \cdot n$$

$$\vdots$$

$$\leq 2^k \cdot T\left(\frac{n}{2^k}\right) + k \cdot b \cdot n$$

$$\vdots$$

$$\leq n \cdot T(1) + \log_2(n) \cdot b \cdot n \leq \underbrace{b \cdot n(1 + \log_2(n))}_{\text{guess}}$$

Analysis Merge Sort

$T(n) = O(n \cdot \log n)$

Recursive equation: $T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + b \cdot n,\ T(1) \leq b$

Guess: $T(n) \leq b \cdot n \cdot (1 + \log_2 n)$

Proof by induction:

Base: $n = 1$ $\quad T(1) \leq b \cdot 1 \cdot (1 + \log_2 1) = b$ ✓ (where $\log_2 1 = 0$)

Step:

$T(n) \leq 2\,T\left(\frac{n}{2}\right) + b \cdot n$

$\overset{(IH)}{\leq} 2\left(b\frac{n}{2}\left(1 + \log_2 \frac{n}{2}\right)\right) + bn$ (where $1 + \log_2 \frac{n}{2} = \log_2 n$)

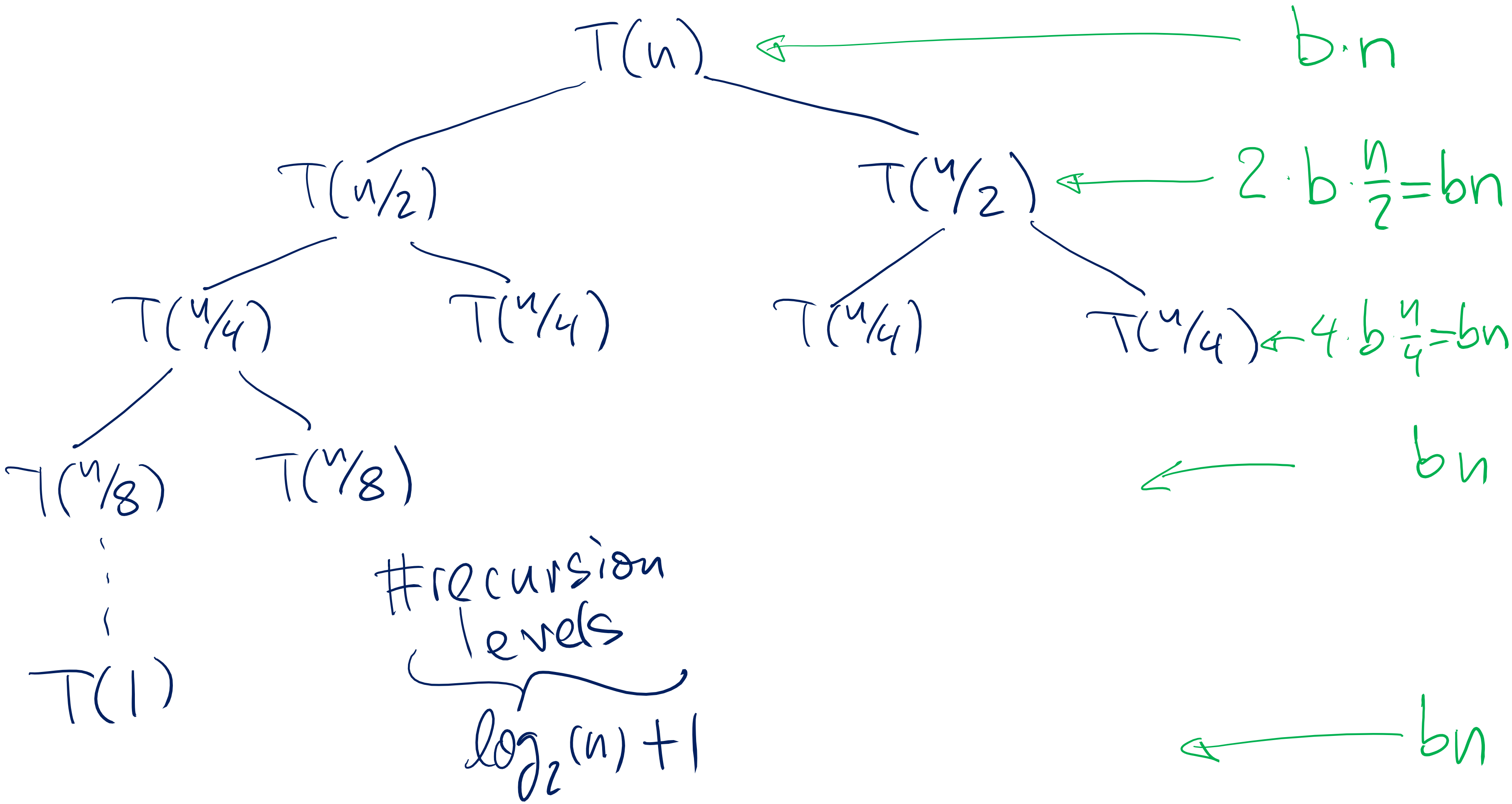
$\log_2 \frac{n}{2} = \log_2 n - \log_2 2$, with $\log_2 2 = 1$

$= bn\log n + bn = bn(1 + \log_2 n)$ ✓ ∎

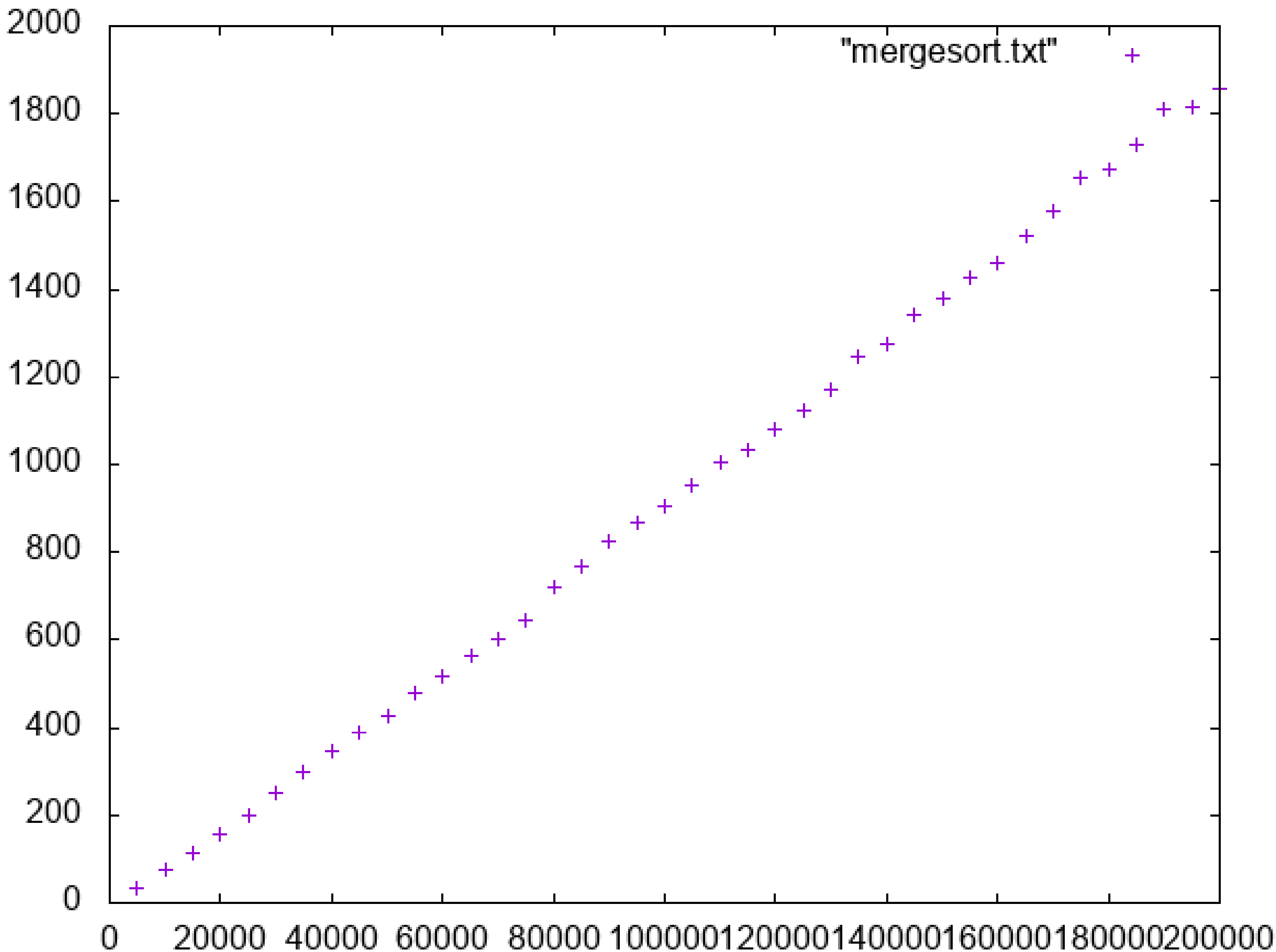
Alternative Analysis of Merge Sort

Recursive equation: $T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + b \cdot n,\ T(1) \leq b$

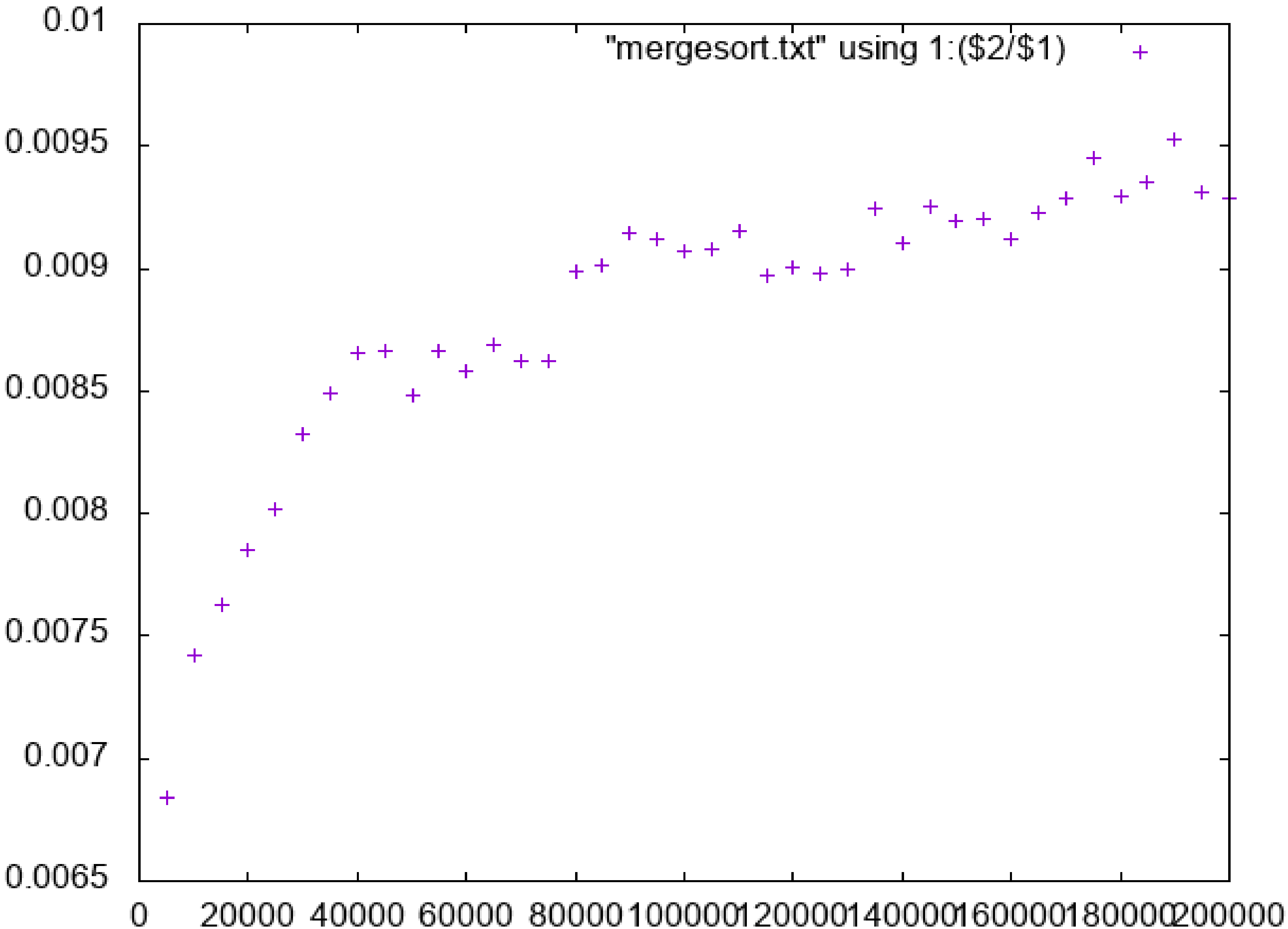
Consider the recursion tree:



Merge Sort Measurements



Merge Sort Measurements



Summary Analysis Merge Sort

The runtime of Merge Sort is $T(n) \in O(n \cdot \log n)$.

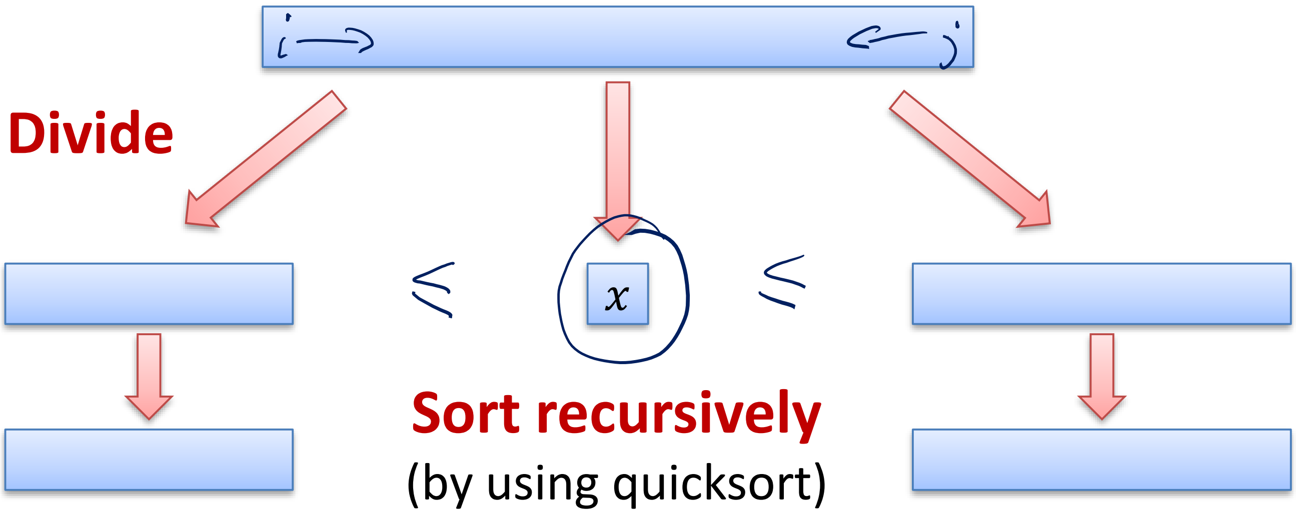
- grows almost linearly with the input size n...

How good is this?

- Example calculation:
  - Again assume that 1 basic operation = 1 ns
  - We will be a bit more conservative than before and assume that $$T(n) = 10 \cdot n \log n$$

Input size n	4 byte numbers	Runtime $T(n) = 10 \cdot n \log n$	n^2
$2^{10} \approx 10^3$ numbers	$\approx 4\text{KB}$	$10 \cdot 10 \cdot 2^{10} \cdot 10^{-9}\ \text{s} \approx 0.1\ \text{ms}$	$1\ \text{ms}$
$2^{20} \approx 10^6$ numbers	$\approx 4\text{MB}$	$10 \cdot 20 \cdot 2^{20} \cdot 10^{-9}\ \text{s} \approx 0.2\ \text{s}$	$16.7\ \text{min}$
$2^{30} \approx 10^9$ numbers	$\approx 4\text{GB}$	$10 \cdot 30 \cdot 2^{30} \cdot 10^{-9}\ \text{s} \approx 5.4\ \text{min}$	$31.7\ \text{years}$
$2^{40} \approx 10^{12}$ numbers	$\approx 4\text{TB}$	$10 \cdot 40 \cdot 2^{40} \cdot 10^{-9}\ \text{s} \approx 122\ \text{h}$	$> 10^7\ \text{years}$

Quick Sort : Analysis



- Runtime depends on how we choose the pivots
- Runtime to sort array of size n if pivot partitions array into parts of sizes λn and $(1-\lambda)n$:

$$\boldsymbol{T(n) = T(\lambda n) + T\big((1-\lambda)n\big) + \text{"Find pivot + Divide"}}$$

- **Divide:**
  - We iterate over the array from both sides, $O(1)$ cost per step
    → Time to partition array of length n: $O(n)$

Quick Sort : Analysis

If we can also find a pivot in time $O(n)$ such that such that the array is partitioned into parts of sizes λn and $(1-\lambda)n$:

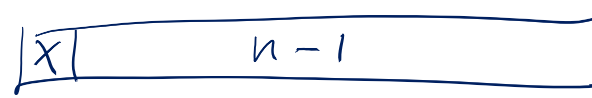
- There is a constant $b > 0$, s. t.

$$T(n) \leq T(\lambda n) + T\big((1-\lambda)n\big) + b \cdot n, \qquad T(1) \leq b$$

Extreme case I) $\lambda = 1/2$ (best case):

$$T(n) \leq 2T\left(\frac{n}{2}\right) + bn, \qquad T(1) \leq b$$

- As for Merge Sort: $T(n) \in O(n \log n)$



Extreme case II) $\lambda n = 1, (1-\lambda)n = n-1$ (worst case):

$$T(n) = T(n-1) + bn, \qquad T(1) \leq b$$

Quick Sort : Worst Case Analysis

Extreme case II) $\lambda n = 1, (1-\lambda)n = n-1$ (worst case):

$$T(n) = T(n-1) + bn, \qquad T(1) \le b$$

In this case, we obtain $T(n) \in \Theta(n^2)$:

$$T(n) = T(n-1) + bn$$
$$= T(n-2) + b(n-1) + bn$$
$$= T(n-3) + b(n-2+n-1+n)$$
$$\vdots$$
$$= T(n-k) + b(n-k+1+\dots+n)$$
$$\vdots$$
$$= T(1) + b(2+3+\dots+n)$$
$$\le b(1+2+\dots+n)$$
$$= b\cdot\frac{n(n+1)}{2} = \Theta(n^2)$$

Guess: $T(n) \le b\cdot\frac{n(n+1)}{2}$

Base: $(n=1)$ $T(1) \le b\cdot\frac{1\cdot 2}{2} = b$ ✓

Step:

$$T(n) \le T(n-1) + b\cdot n \overset{(IH)}{\le} b\frac{(n-1)n}{2} + b\cdot n = b\frac{n(n+1)}{2}$$ ✓ □

Quick Sort With a Random Pivot

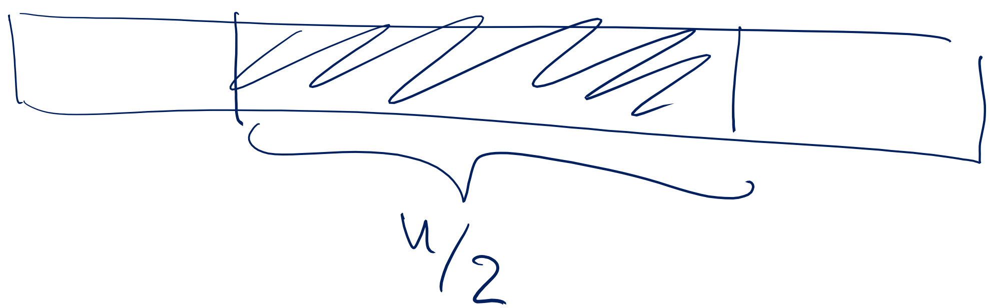
Partition For Random Pivot:

- Runtime $\boldsymbol{T(n) = O(n \log n)}$ for all inputs
  - but only in Erwartungswert and with very high probability

Intuition:

- With probability $1/2$, we get parts of size $\geq n/4$, s. t.

$$T(n) \quad \leq \quad T\left(\frac{n}{4}\right) + T\left(\frac{3n}{4}\right) + bn$$



Quick Sort With a Random Pivot

Partition For Random Pivot:

- Runtime $\boldsymbol{T(n) = O(n \log n)}$ for all inputs
  - but only in Erwartungswert and with very high probability

Analysis:

- We will not do this here
  - see, e.g., Cormen et al. or the algorithm theory lecture
- Possible approach: write recursion in terms of expected values

$$\mathbb{E}[T(n)] \leq \mathbb{E}[T(N_L) + T(n - N_L)] + bn$$

Sorting Lower Bound

Task: Sort sequence $a_1, a_2, \dots, a_n$

- Goal: lower bound (worst-case) runtime

Comparison-based sorting algorithms

- Comparisons are the only allowed way to determine the relative order between elements
- Hence, the only thing that can influence the sequence of elements in the final sorted sequence are comparisons of the kind

$$a_i = a_j, a_i \leq a_j, a_i < a_j, a_i \geq a_j, a_i > a_j$$

- If we assume that the elements are pair-wise distinct, we only need comparisons of the form $a_i \leq a_j$

- 1 comparison $=$ 1 basic operation

Comparison-Based Sorting Algorithms

UNI FREIBURG

Alternative View

- Every program (for a deterministic, comp.-based sorting alg.) can be brought into a form where every if/while/…-condition is of the following form:

$$\textbf{if}\ (a_i \leq a_j)\ \textbf{then}\ \ldots$$

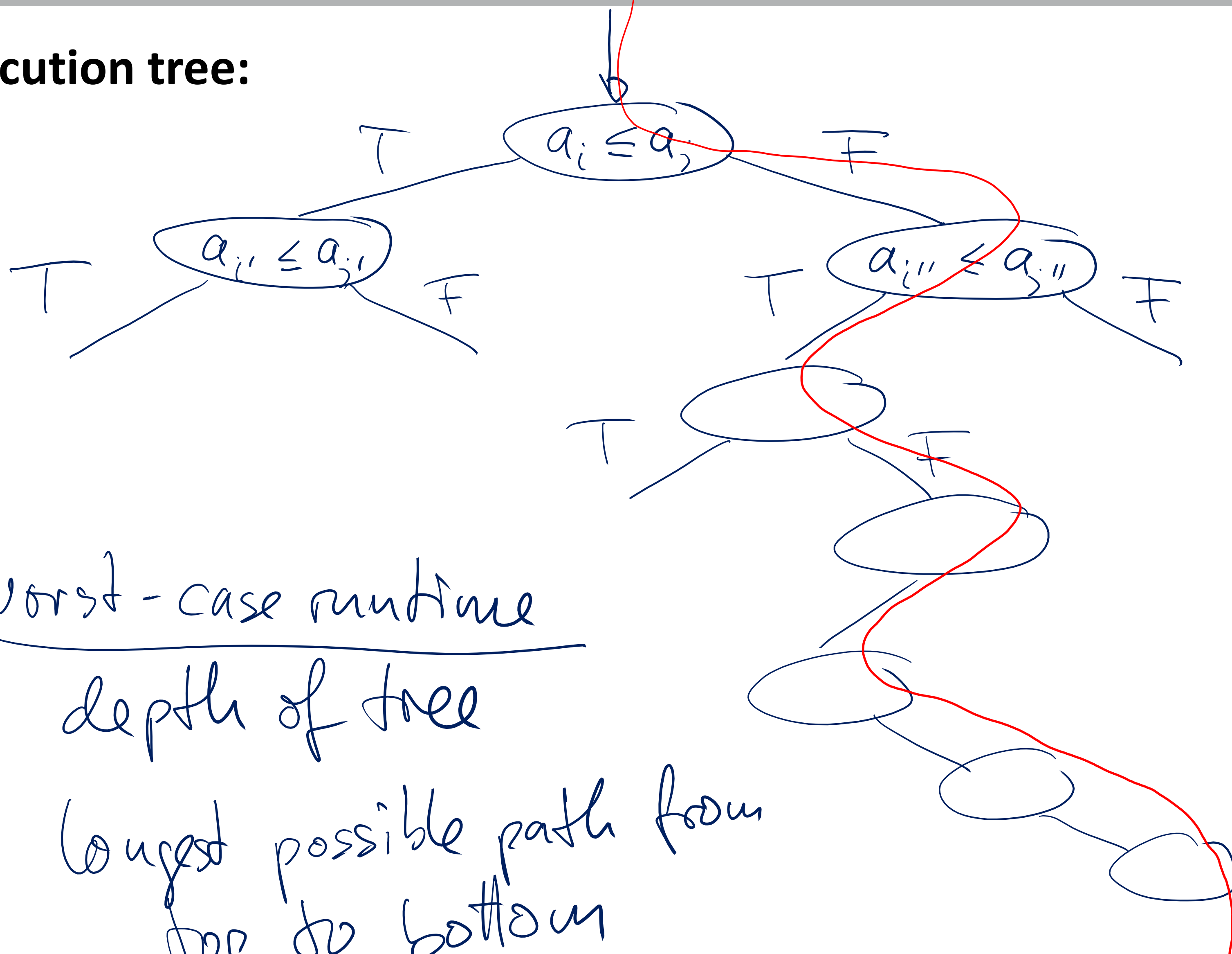
- In each execution of an algorithm, the results of these comparisons induce a sequence of T/F (true/false) values:

$$\textbf{TFFTTTFTFFTTFFFFFTFTTT}\ \ldots$$

- This sequence uniquely determines how the values of the array are rearranged (permuted) by the algorithm.
- Different inputs with the same values therefore must lead to different T/F sequences.

Comparison-Based Sorting Algorithms

Execution tree:



worst-case runtime

depth of tree

longest possible path from top to bottom

Comp.-Based Sorting: Lower Bound

- In comparison-based sorting algorithms, the execution depends on the initial ordering of the values in the inputs, but it does not depend on the actual values.
  - We restrict to cases where the values are all distinct.
- W.l.o.g. we can assume that we have to sort the numbers $1, \dots, n$.
- Different inputs have to be handled differently.
- Different inputs result in different T/F sequences
- Runtime of an execution $\geq$ length of the resulting T/F sequence
- Worst-Case runtime $\geq$ Length of longest T/F sequence:
  - We want a lower bound
  - Count no. of possible inputs → we need at least as many T/F sequences...

Comp.-Based Sorting: Lower Bound

Number of possible inputs (input orderings):

$$n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1$$

Number of $\mathrm{T/F}$ sequences of length $\leq k$: Länge $= k$: 2^k

T/F T/F ... T/F
1 2 ... $\leq k$

$$2^k + 2^{k-1} + 2^{k-2} + \ldots + 1 \leq 2^{k+1}$$

Theorem: Every comparison-based sorting algorithm requires $\Omega(n \cdot \log n)$ comparisons in the worst case.

Runtime $\leq T$

$$2^{T+1} \geq n!$$

$$T+1 \geq \log_2(n!)$$

$$= \Omega(n \log n).$$

$$\left(\frac{n}{2}\right)^{n/2} \leq n! \leq n^n$$

$$\frac{n}{2} \cdot \log\left(\frac{n}{2}\right) \leq \log(n!) \leq n \cdot \log(n)$$

$$\log(n!) = \Theta(n \log n)$$

Sorting in Linear Time

- Not possible with comparison-based algorithms
  - Lower bound also holds for randomized algorithms...
- Sometimes, we can be faster
  - If we can exploit special properties of the input
- Example: Sort n numbers $a_i \in \{0,1\}$:
  1. Count number of zeroes and ones in time $O(n)$
  2. Write solution to array in time $O(n)$

Counting Sort

Task:

- Sort integer array A of length n
- We know that for all $i \in \{0, \dots, n\}$, $A[i] \in \{0, \dots, k\}$

Algorithm:

```
counts = new int[k+1]        // new int array of length k
for i = 0 to k do counts[i] = 0
for i = 0 to n-1 do counts[A[i]]++
i = 0;
for j = 0 to k do
    for l = 1 to counts[j] do
        A[i] = j; i++
```

$O(n+k)$