

Quiz

$$3n^4 = \Theta(n^5)$$

☐ True

☒ False

$$\lim_{n \rightarrow \infty} \frac{3n^4}{n^5} = \lim_{n \rightarrow \infty} \frac{3}{n} = 0 \xrightarrow{\text{Theorem 1}} 3n^4 \neq \Theta(n^5)$$

$$2^{1.5n} \leq O(2^n)$$

☐ True

☒ False

$$\lim_{n \rightarrow \infty} \frac{2^{1.5n}}{2^n} = \lim_{n \rightarrow \infty} 2^{1.5n - n} = \lim_{n \rightarrow \infty} 2^{0.5n} = \infty$$

$$\xrightarrow{\text{Theorem 1}} 2^{1.5n} \neq O(2^n)$$

Consider the function $T : \mathbb{N} \rightarrow \mathbb{R}^+$ defined by the recurrence

$$T(1) = 10, \quad \text{and} \quad T(n) = T(n-1) + 10 \quad \text{for } n > 1.$$

Enter the smallest value k such that $T(n) \leq O(n^k)$.

Answer:

1

Teleskopieren: zwei Lösungswege

$$T(1) = 10$$

$$T(n) = T(n-1) + 10$$

$$T(2) = T(1) + 10 = 10 + 10 = 20$$

$$= (T(n-2) + 10) + 10$$

$$T(3) = T(2) + 10 = 20 + 10 = 30$$

$$= ((T(n-3) + 10) + 10) + 10$$

⋮

$$T(4) = T(3) + 10 = 30 + 10 = 40$$

$$= T(1) + \underbrace{10 + \dots + 10}_{n-1 \text{ mal}}$$

⋮

$$T(n) = 10n \leq O(n)$$

$$= 10n \leq O(n)$$

The goal of a divide-and-conquer algorithm is to...

- ☐ a. solve the problem by trying every possible candidate solution. (called naive / brute force)
- ☒ b. split the problem into smaller subproblems, solve each, and combine their solutions.
- ☐ c. make the problem larger until it becomes easier to solve.

Match the following algorithms for the *Maximum Subarray Sum* problem from the lecture with their running times (where n is the length of the array):

Naive algorithm

$$\Theta(n^3)$$

iterates over all substrings

Divide-and-Conquer algorithm

$$\Theta(n \log n)$$

splits in the middle

Inductive algorithm

$$\Theta(n)$$

uses Randmaximum

The *Maximum Subarray Sum* problem can be solved in time $O(\sqrt{n})$ (where n is the length of the array).

☐ True

$\Theta(n)$ is optimal

☒ False

(we have to read all n numbers, which takes $\Omega(n)$ time)

Let $A[1..n]$ be an input array for the *Maximum Subarray Sum* problem, where n is even. Let $k := n/2$. The maximum sum of a subarray of A that contains at least one element of $A[1..k]$ and at least one element of $A[k+1..n]$ can be computed by...

$\Rightarrow$ must contain $A[k]$ and $A[k+1]$

- ☐ a. the maximum non-empty prefix sum of $A[1..k]$ plus the maximum non-empty suffix sum of $A[k+1..n]$.
- ☒ b. the maximum non-empty suffix sum of $A[1..k]$ plus the maximum non-empty prefix sum of $A[k+1..n]$.
- ☐ c. the maximum non-empty subarray sum of $A[1..k]$ plus the maximum non-empty subarray sum of $A[k+1..n]$.
- ☐ d. the sum of all elements of A .

index: 1 2 3 $k-2$ $k-1$ k $k+1$ $k+2$ $k+3$ $n-1$ n
 $A[4, 2, -3, \dots, 2, 3, 8, -2, 1, 3, \dots, 2, -4]$

$\leftarrow$ suffix sum | prefix sum $\rightarrow$

Consider the *search* problem with a given input array $A[1..n]$ and target element b .

If A is not sorted, then the linear search algorithm with running time $\Theta(n)$ is optimal.

☒ True

we have to look at all n elements in worst case

☐ False

Consider the *search* problem with a given input array $A[1..n]$ and target element b .

For binary search, it does not matter whether A is sorted or not.

☐ True

☒ False

$A[7, 0, 1]$
 $\uparrow$
 search 7: $0 < 7 \Rightarrow$ searches on the right side and doesn't find 7

We can check whether or not an array of length n is already sorted in time $\Theta(n)$.

☒ True

☐ False

SORTED(A)

```
1 for  $i \leftarrow 1, 2, \dots, n-1$  do
2   if  $A[i] > A[i+1]$  then return false
3 return true
```

The Bubble-Sort algorithm sorts the elements of a given array by...

☐ a. always comparing the first and the last element.

☐ b. always comparing the smallest and the largest element.

☒ c. always comparing neighboring elements.

Consider the following pseudocode snippet:

Algorithm 1

```
 $i \leftarrow 1$ 
while  $i \leq 3n$  do
   $f()$ 
   $i \leftarrow i + 1$ 
 $j \leftarrow 0$ 
while  $j \leq n^2$  do
   $f()$ 
   $f()$ 
   $f()$ 
   $j \leftarrow j + 1$ 
```

Which of the following expressions correctly describes the exact number of calls to f ?

☐ a. $\sum_{i=1}^{3n} 1 + \sum_{j=1}^{n^2} 3$

☒ b. $\sum_{i=1}^{3n} 1 + \sum_{j=0}^{n^2} 3$

☐ c. $\sum_{i=1}^{3n} \sum_{j=1}^{n^2} 4$

Discussion of Exercise Sheet 3

Exercise 3.1 Asymptotic growth.

(a) Prove or disprove the following statements. Justify your answer.

(1) For all $a, b > 1$, suppose $n \in \mathbb{N}$ and $n > \max\{a, b\}$, then $\log_a(n) = \Theta(\log_b(n))$.

Let $a, b > 1$ be arbitrary

$$\log_a(n) = \frac{\log_b(n)}{\log_b(a)} = \underbrace{\frac{1}{\log_b(a)}}_{\text{konstanter Faktor}} \cdot \log_b(n) = \Theta(\log_b(n))$$

Basiswechsel

konstanter Faktor

Fazit: Die Basis spielt asymptotisch keine Rolle Alle Logarithmen

wachsen asymptotisch gleich schnell

(2) For all $a, b \geq 1$, suppose $n \in \mathbb{N}$, then $a^n = \Theta(b^n)$ (Master Solution ist bisschen komisch)

Gegenbeispiel: $a = 1$, $b = 2$

$$\lim_{n \rightarrow \infty} \frac{a^n}{b^n} = \lim_{n \rightarrow \infty} \frac{1^n}{2^n} = \lim_{n \rightarrow \infty} \frac{1}{2^n} = 0$$

$$\overbrace{1^n}^1 = 1$$

Theorem 1
 $\Rightarrow a^n \neq \Theta(b^n)$

(b) (1) Prove that $\lim_{n \rightarrow \infty} \frac{n}{\log(n)} = \infty$.

Hint: Use L'Hôpital's rule.

$$\lim_{n \rightarrow \infty} \frac{n}{\log(n)} = \lim_{n \rightarrow \infty} \frac{\frac{n}{\ln(n)}}{\frac{1}{\ln(2)}} \stackrel{\text{L'Hôp}}{=} \lim_{n \rightarrow \infty} \frac{1}{\frac{1}{n}} = \lim_{n \rightarrow \infty} n \cdot \ln(2) = \infty$$

Basiswechsel

Hilfe: $\frac{\frac{A}{B}}{\frac{C}{D}} = \frac{A}{B} \cdot \frac{D}{C}$

(2) Prove that $\lim_{n \rightarrow \infty} n(e^{1/n} - 1) = 1$.

You may use the following variant of L'Hôpital's rule:

Theorem 2 (L'Hôpital's rule (going to 0)). Assume that functions $f : \mathbb{R}^+ \rightarrow \mathbb{R}$ and $g : \mathbb{R}^+ \rightarrow \mathbb{R}$ are differentiable, $\lim_{x \rightarrow \infty} f(x) = \lim_{x \rightarrow \infty} g(x) = 0$ and for all $x \in \mathbb{R}^+$, $g'(x) \neq 0$. If $\lim_{x \rightarrow \infty} \frac{f'(x)}{g'(x)} = C \in \mathbb{R}$ or $\lim_{x \rightarrow \infty} \frac{f'(x)}{g'(x)} = \infty$, then

$$\lim_{x \rightarrow \infty} \frac{f(x)}{g(x)} = \lim_{x \rightarrow \infty} \frac{f'(x)}{g'(x)}.$$

$$\lim_{n \rightarrow \infty} n \cdot (e^{\frac{1}{n}} - 1) = \lim_{n \rightarrow \infty} \frac{e^{\frac{1}{n}} - 1}{\frac{1}{n}}$$

Umformung weil wir die Form " $\frac{0}{0}$ " für L'Hôpital brauchen

oben: $\frac{1}{n} \rightarrow 0$, deshalb $e^{\frac{1}{n}} \rightarrow e^0 = 1$, deshalb $e^{\frac{1}{n}} - 1 \rightarrow 0$ ✓

unten: $\frac{1}{n} \rightarrow 0$ ✓

$$\stackrel{\text{L'Hôp.}}{=} \lim_{n \rightarrow \infty} \frac{-e^{\frac{1}{n}} \cdot \frac{1}{n^2}}{-\frac{1}{n^2}}$$

Ableitung: $(\frac{1}{n})' = (n^{-1})' = -n^{-2} = -\frac{1}{n^2}$

$$(e^{\frac{1}{n}})' = e^{\frac{1}{n}} \quad (\frac{1}{n})' = -e^{\frac{1}{n}} \cdot \frac{1}{n^2}$$

Kettenregel

$$= \lim_{n \rightarrow \infty} e^{\frac{1}{n} \rightarrow 0}$$

$$= 1 \quad \text{weil } \lim_{n \rightarrow \infty} e^{f(n)} = e^{\lim_{n \rightarrow \infty} f(n)}$$

Das "Limes hineinziehen" ist nicht immer erlaubt,

$$\text{Beispiel: } 2 = \lim_{n \rightarrow \infty} 2 = \lim_{n \rightarrow \infty} 2^{n^{\frac{1}{n}}} = \lim_{n \rightarrow \infty} (2^n)^{\frac{1}{n} \rightarrow 0} = \lim_{n \rightarrow \infty} (2^n)^0 = 1$$

Regel für A&D: erlaubt wenn die Basis konstant ist

Anderer Lösungsweg mit Substitution $y = \frac{1}{n}$

$$\lim_{n \rightarrow \infty} n \cdot (e^{\frac{1}{n}} - 1) = \lim_{y \rightarrow 0} \frac{1}{y} \cdot (e^y - 1) = \lim_{y \rightarrow 0} \frac{e^y - 1}{y} \stackrel{\text{L'Hôp}}{=} \lim_{y \rightarrow 0} \frac{e^y}{1} = 1$$

Substitution $y = \frac{1}{n} \Leftrightarrow n = \frac{1}{y}$

wenn $n \rightarrow \infty$, dann $y = \frac{1}{n} \rightarrow 0$

weil $y \rightarrow 0$
folgt $e^y \rightarrow e^0 = 1$

Exercise 3.2 Substring counting (2 bonus points). → peer grading

Given an n -bit bitstring $S[0..n-1]$ (i.e. $S[i] \in \{0, 1\}$ for $i = 0, 1, \dots, n-1$), and an integer $k \geq 0$, we would like to count the number of nonempty contiguous substrings of S with exactly k ones. Assume $n \geq 2$.

For example, when $S = \text{"0110"}$ and $k = 2$, there are 4 such substrings: "011", "11", "110", and "0110".

- (a) Design a "naive" algorithm that solves this problem with a runtime of $O(n^3)$. Describe the algorithm using pseudocode. Justify the runtime (you don't need to provide a formal proof, but you should state your reasoning).

Just iterate over all possible substrings from i to j and count the ones

Algorithm 1 Naive substring counting

$c \leftarrow 0$	▷ Initialize counter of substrings with k ones
for $i \leftarrow 0, \dots, n-1$ do	▷ Enumerate all nonempty substrings $S[i..j]$
for $j \leftarrow i, \dots, n-1$ do	
$x \leftarrow 0$	▷ Initialize counter of ones
for $\ell \leftarrow i, \dots, j$ do	▷ Count ones in substring
if $S[\ell] = 1$ then	
$x \leftarrow x + 1$	
if $x = k$ then	▷ If there are k ones in substring, increment c
$c \leftarrow c + 1$	
return c	▷ Return number of substrings with k ones

Runtime: The nested for-loops have three levels and each level has at most n iterations, leading to $O(n^3)$ iterations in total. Each iteration runs in $O(1)$ time. Thus the total running time is $O(n^3)$.

- (b) We say that a bitstring S' is a (non-empty) prefix of a bitstring S if S' is of the form $S[0..i]$ where $0 \leq i < \text{length}(S)$. For example, the prefixes of $S = \text{"0110"}$ are "0", "01", "011" and "0110".

Given an n -bit bitstring S , we would like to compute a table T indexed by $0..n$ such that for all i , $T[i]$ contains the number of prefixes of S with exactly i ones.

For example, for $S = \text{"0110"}$, the desired table is $T = [1, 1, 2, 0, 0]$, since, of the 4 prefixes of S , 1 prefix contains zero "1", 1 prefix contains one "1", 2 prefixes contain two "1", and 0 prefixes contain three "1" or four "1".

Design an algorithm `PREFIXTABLE` that computes T from S in time $O(n)$, assuming S has size n . Describe the algorithm using pseudocode. Justify the runtime (you don't need to provide a formal proof, but you should state your reasoning).

Algorithm 2

function <code>PREFIXTABLE</code> (S)	
$T[0..n] \leftarrow$ an array of $(n+1)$ zeros	▷ Initialize array
$s \leftarrow 0$	
for $i \leftarrow 0, \dots, n-1$ do	▷ Enumerate all prefixes $S[0..i]$
$s \leftarrow s + S[i]$	▷ s saves the number of "1" in $S[0..i]$
$T[s] \leftarrow T[s] + 1$	▷ $S[0..i]$ is a prefix with s "1"
return T	

= number of
1s seen so far

- we have exactly one new prefix for every i
- s stores the number of 1s of this prefix
- Therefore we add 1 to $T[s]$, because we have seen one more prefix with s 1s

Remark: This algorithm can also be applied on a reversed bitstring to compute the same table for all suffixes of S . In the following, you can assume an algorithm SUFFIXTABLE that does exactly this.
(difficult)

- (c) Consider an integer $m \in \{0, 1, \dots, n-2\}$. Using PREFIXTABLE and SUFFIXTABLE, design an algorithm SPANNING(m, k, S) that returns the number of substrings $S[i..j]$ of S that have exactly k ones and such that $i \leq m < j$.

$\Rightarrow$ substring must include index m and $m+1$

For example, if $S = "0110"$, $k = 2$, and $m = 0$, there exist exactly two such strings: "011" and "0110". Hence, SPANNING(m, k, S) = 2.

Describe the algorithm using pseudocode. Mention and justify the runtime of your algorithm (you don't need to provide a formal proof, but you should state your reasoning).

Hint: Each substring $S[i..j]$ with $i \leq m < j$ can be obtained by concatenating a string $S[i..m]$ that is a suffix of $S[0..m]$ and a string $S[m+1..j]$ that is a prefix of $S[m+1..n-1]$.

$S: [1, 0, \dots, 0, 1, 1, \boxed{1}, \boxed{0}, 0, 1, \dots, 1, 0]$

$\leftarrow$ Suffixtable | Prefixtable $\rightarrow$

p ones | $k-p$ ones

total: $p + k-p = k$ ones

Algorithm 3

function SPANNING(m, k, S)

$T_1 \leftarrow$ SUFFIXTABLE($S[0..m]$)

$T_2 \leftarrow$ PREFIXTABLE($S[m+1..n-1]$)

return $\sum_{p=\max(0, k-(n-m-1))}^{\min(k, m+1)} (T_1[p] \cdot T_2[k-p])$ iterate over all possible p 's

$p \leq k$ because we don't want more than k ones

$p \leq m+1$ because the left side has length $m+1$

$p \geq 0$ because we can't have a negative number of ones

$p \geq k - (n-m-1)$ because the right side has length $(n-m-1)$

Runtime: Computing both tables takes $O(n)$ time, and the sum has at most n terms, each taking $O(1)$ time. Thus the total runtime is $O(n)$.

Exercise 3.3 Counting function calls in loops (2 bonus points).

For each of the following code snippets, compute the number of calls to f as a function of $n \in \mathbb{N}$. Provide **both** the exact number of calls and a maximally simplified asymptotic bound in Θ notation.

Algorithm 1

(a) $i \leftarrow 0$
 while $i \leq n$ do
 $f()$
 $f()$
 $i \leftarrow i + 1$
 $j \leftarrow 0$
 while $j \leq 2n$ do
 $f()$
 $j \leftarrow j + 1$

$$\begin{aligned}
 \text{Number of calls} &= \sum_{i=0}^n 2 + \sum_{j=0}^{2n} 1 \\
 &= (n+1) \cdot 2 + (2n+1) \cdot 1 \\
 &= 2n+2 + 2n+1 \\
 &= 4n+3 \\
 &= \Theta(4n)
 \end{aligned}$$

Algorithm 2

(b) $i \leftarrow 1$
 while $i \leq n$ do
 $j \leftarrow 1$
 while $j \leq i^3$ do
 $f()$
 $j \leftarrow j + 1$
 $i \leftarrow i + 1$

$$\begin{aligned}
 \text{Number of calls} &= \sum_{i=1}^n \sum_{j=1}^{i^3} 1 \\
 &= \sum_{i=1}^n i^3 \\
 &= \frac{n^2(n+1)^2}{4} \\
 &= \Theta(n^4)
 \end{aligned}$$

Hint: You may use $\sum_{i=1}^n i^3 = \frac{n^2(n+1)^2}{4}$.

Suchen

LINEAR-SEARCH($A[1..n], b$)

```

1 for  $i \leftarrow 1, 2, \dots, n$  do
2   if  $A[i] = b$  then return  $i$ 
3 return "nicht gefunden"

```

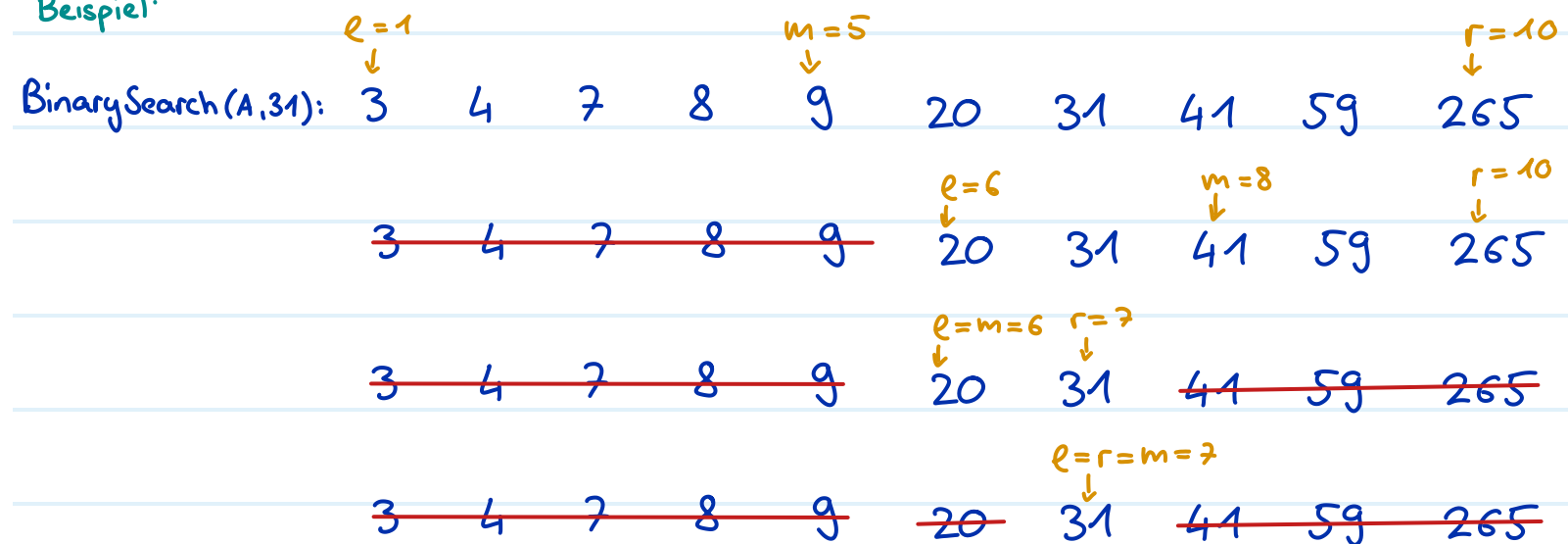
- Anwendbar für beliebige Arrays (auch unsortierte)
- Worst case Laufzeit: $\Theta(n)$

BINARY-SEARCH($A[1..n], b$)

1 $\ell \leftarrow 1; r \leftarrow n$	▷ Initialer Suchbereich
2 while $\ell \leq r$ do	
3 $m \leftarrow \lfloor (\ell + r)/2 \rfloor$	
4 if $A[m] = b$ then return m	▷ Element gefunden
5 else if $A[m] > b$ then $r \leftarrow m - 1$	▷ Suche links weiter
6 else $\ell \leftarrow m + 1$	▷ Suche rechts weiter
7 return "Nicht vorhanden"	

- Anwendbar nur für sortierte Arrays
- In jeder Iteration halbieren wir den Suchbereich
- Worst case Laufzeit: $\Theta(\log n)$

Beispiel:



Intuition für die Laufzeit:

- $\log(n) =$ "wie oft muss ich n halbieren bis ich bei 1 ankomme?" (für $n=2^k$)
- Allgemein: $\log_a(n) =$ "wie oft muss ich n durch a teilen bis ich bei 1 ankomme?" (für $n=a^k$)

Bsp $\log(32) = 5$ weil $32 \xrightarrow{1} 16 \xrightarrow{2} 8 \xrightarrow{3} 4 \xrightarrow{4} 2 \xrightarrow{5} 1$

$\log(8) = 3$ weil $8 \xrightarrow{1} 4 \xrightarrow{2} 2 \xrightarrow{3} 1$

$\log(4) = 2$ weil $4 \xrightarrow{1} 2 \xrightarrow{2} 1$

$\log_3(27) = 3$ weil $27 \xrightarrow{1} 9 \xrightarrow{2} 3 \xrightarrow{3} 1$

Mehr Intuition für log Gesetze.

$\log(a \cdot b) = \log(a) + \log(b):$

$$\underbrace{\log(4 \cdot 8)}_{\substack{\text{wie oft } 4 \cdot 8 = 32 \\ \text{halbieren bis 1}}} = \underbrace{\log(4)}_{\substack{\text{wie oft } 4 \\ \text{halbieren bis 1}}} + \underbrace{\log(8)}_{\substack{\text{wie oft } 8 \\ \text{halbieren bis 1}}}$$

$$\log(\underbrace{(2 \cdot 2)}_5 \underbrace{(2 \cdot 2 \cdot 2)}_3) = \log(\underbrace{2 \cdot 2}_2) + \log(\underbrace{2 \cdot 2 \cdot 2}_3)$$

$\log_a(x) = \frac{\log_b(x)}{\log_b(a)}$

$$\log_4(64) = \frac{\log_2(64)}{\log_2(4)}, \quad 3 = \frac{6}{2}$$

$$\log_2(64) = 6: \quad 64 \xrightarrow{1} 32 \xrightarrow{2} 16 \xrightarrow{3} 8 \xrightarrow{4} 4 \xrightarrow{5} 2 \xrightarrow{6} 1$$

$$\log_4(64) = 3: \quad 64 \xrightarrow{1} 16 \xrightarrow{2} 4 \xrightarrow{3} 1$$

$$\log_2(4) = 2: \quad 4 \xrightarrow{1} 2 \xrightarrow{2} 1$$

Korrektur-Faktor
weil der violette Pfeil
genau zwei blauen entspricht.

Basis = Pfeil-Länge

$$\log(a^b) = b \cdot \log(a):$$

$$\begin{aligned} \log(4^3) &= \log(\underbrace{(2 \cdot 2) \cdot (2 \cdot 2)}_{\substack{\text{wie oft } 4^3=64 \\ \text{halbieren bis 1}}}) = \log(2 \cdot 2) + \log(2 \cdot 2) + \log(2 \cdot 2) \\ &= 3 \cdot \log(2 \cdot 2) \\ &\quad \substack{\text{wie oft 4} \\ \text{halbieren bis 1}} \end{aligned}$$

$$6 = 3 \cdot 2$$

$$\log(4^3) = 6 \cdot \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \\ 64 & \xrightarrow{1} 32 & \xrightarrow{2} 16 & \xrightarrow{3} 8 & \xrightarrow{4} 4 & \xrightarrow{5} 2 & \xrightarrow{6} 1 \end{array}$$

$$\log(4) = 2 : \begin{array}{c} 1 \\ 4 \xrightarrow{1} 2 \xrightarrow{2} 1 \end{array}$$

Deshalb ist $\log(1) = 0$, weil wir 0 Pfeile bis 1 brauchen, da wir ja schon bei 1 sind

Deshalb ist $\log_a(a) = 1$, weil wir von a nach 1 genau einen Pfeil $\xrightarrow{a}$ brauchen.

Für $0 < x < 1$ ist $\log(x)$ negativ, weil wir die Pfeile rückwärts bis 1 gehen

$$\log\left(\frac{1}{4}\right) = -2 : \begin{array}{c} -1 \\ \frac{1}{4} \xleftarrow{:2} \frac{1}{2} \xleftarrow{:2} 1 \end{array}$$

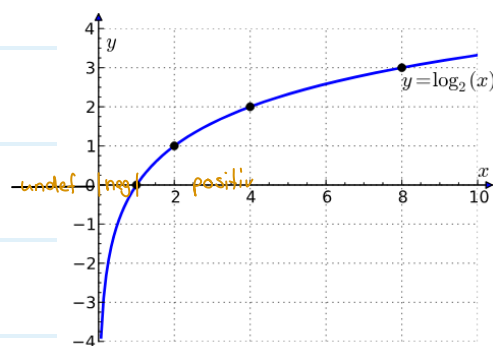
$$\log\left(\frac{1}{32}\right) = -5 : \begin{array}{c} -1 \\ \frac{1}{32} \xleftarrow{:2} \frac{1}{16} \xleftarrow{:2} \frac{1}{8} \xleftarrow{:2} \frac{1}{4} \xleftarrow{:2} \frac{1}{2} \xleftarrow{:2} 1 \end{array}$$

$$\log_3\left(\frac{1}{27}\right) = -3 : \begin{array}{c} -1 \\ \frac{1}{27} \xleftarrow{:3} \frac{1}{9} \xleftarrow{:3} \frac{1}{3} \xleftarrow{:3} 1 \end{array}$$

Deshalb ist $\log(x)$ undefiniert für $x < 0$, weil wenn

wir bei x (negativ) starten, werden wir durch teilen

nie zu 1 (positiv) kommen können



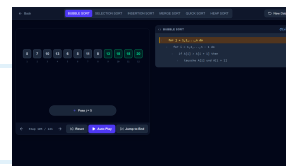
Sortieren

BUBBLE-SORT($A[1..n]$)

```

1 for  $j \leftarrow 1, 2, \dots, n$  do
2   for  $i \leftarrow 1, 2, \dots, n-1$  do
3     if  $A[i] > A[i+1]$  then
4       tausche  $A[i]$  und  $A[i+1]$ 

```



Vergleiche: $n \cdot (n-1) = \Theta(n^2)$

Vertauschungen: $O(n^2)$

Laufzeit: $\Theta(n^2)$

$\text{Inv}(j)$ Nach j Iterationen der äusseren Schleife sind die j grössten Elemente am richtigen Ort. (d.h. an ihrem finalen Index)



Beweis-Schema für Sortier-Algorithmen:

→ Ex 4.4

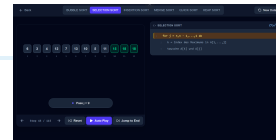
1.) $\text{Inv}(j) = \dots$ formulieren

2.) Base case: $\text{Inv}(0)$ zeigen (oder $\text{Inv}(1)$ je nach Algorithmus)

3.) IH Wir nehmen $\text{Inv}(j)$ an für ein beliebiges $0 \leq j \leq n-1$ (Grenzen je nach Alg anpassen)

4.) I.S. Unter dieser Annahme zeigen wir, dass nach einer weiteren Iteration $\text{Inv}(j+1)$ gilt. ($\text{Inv}(j) \Rightarrow \text{Inv}(j+1)$)
 oder was auch immer zwischen $\text{Inv}(j)$ und $\text{Inv}(j+1)$ passiert

5.) Termination: Wir zeigen, dass falls $\text{Inv}(n)$ gilt, dass dann das Array sortiert ist (d.h. $A[i] \leq A[i+1]$ für $1 \leq i \leq n-1$)



SELECTION-SORT($A[1..n]$)

- 1 **for** $j \leftarrow n, n-1, \dots, 1$ **do**
- 2 $k \leftarrow$ Index des Maximums in $A[1, \dots, j]$
- 3 tausche $A[k]$ und $A[j]$

Vergleiche: $\sum_{j=1}^n j-1 = \sum_{j=0}^{n-1} j = \frac{n(n-1)}{2} = \Theta(n^2)$

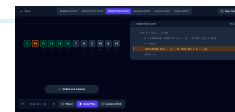
$\xrightarrow{\text{Index Shift}}$ $\xrightarrow{\text{Gauss'sche Summe}}$

Vertauschungen: $O(n)$

Laufzeit: $\Theta(n^2)$

Inv(j) gleich wie Bubble-Sort: Nach j Iterationen der äusseren Schleife sind die j grössten Elemente am richtigen Ort.

INSERTION-SORT($A[1..n]$)



- 1 **for** $j \leftarrow 2, 3, \dots, n$ **do**
- 2 $k \leftarrow$ kleinster Index in $\{1, \dots, j-1\}$ mit $A[j] \leq A[k]$ $\rightarrow$ mit BinarySearch
 $\triangleright A[j]$ gehört an diese Stelle k
- 3 $x \leftarrow A[j]$ $\triangleright$ merke $A[j]$
- 4 verschiebe $A[k, \dots, j-1]$ nach $A[k+1, \dots, j]$
- 5 $A[k] \leftarrow x$

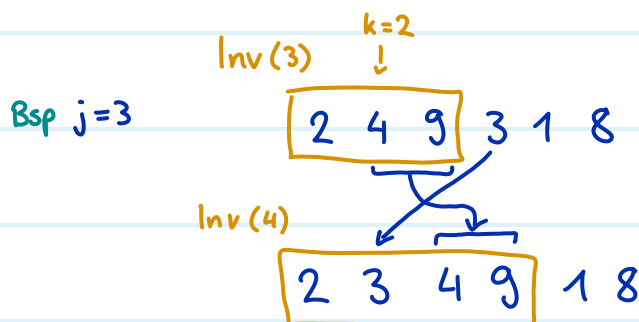
Vergleiche: $O(n \log n)$
 für BinarySearch

Vertauschungen: $O(n^2)$

Laufzeit: $O(n^2)$

Inv(j) Nach j Iterationen der Schleife ist das Teilarray $A[1..j]$ sortiert

(es muss aber nicht zwingend die j kleinsten Elemente enthalten)



MERGESORT($A[1..n], l, r$)

▷ sortiert $A[l, \dots, r]$

1 **if** $l < r$ **then**

2 $m \leftarrow \lfloor (l + r) / 2 \rfloor$

3 MERGESORT(A, l, m)

4 MERGESORT($A, m + 1, r$)

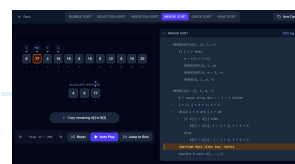
5 MERGE(A, l, m, r)

▷ sortiere linke Hälfte

▷ sortiere rechte Hälfte

▷ verschmelze beide Hälften

} divide and conquer



MERGE($A[1..n], l, m, r$)

1 $B \leftarrow$ neues Array mit $r - l + 1$ Zellen

2 $i \leftarrow l$

3 $j \leftarrow m + 1$

4 $k \leftarrow 1$

5 **while** $i \leq m$ **and** $j \leq r$ **do**

6 **if** $A[i] < A[j]$ **then**

7 $B[k] \leftarrow A[i]$

8 $i \leftarrow i + 1$

9 $k \leftarrow k + 1$

10 **else**

11 $B[k] \leftarrow A[j]$

12 $j \leftarrow j + 1$

13 $k \leftarrow k + 1$

14 übernimm Rest links bzw. rechts

15 kopiere B nach $A[l, \dots, r]$

▷ so gross wie $A[l, \dots, r]$

▷ erstes unbenutztes Element in linker Hälfte

▷ erstes unbenutztes Element in rechter Hälfte

▷ nächste Position in B

▷ beide Hälften noch nicht ausgeschöpft

▷ wenn die andere Hälfte ausgeschöpft ist

Laufzeit von MERGE $O(n)$

Totale Laufzeit:

$$T(n) \leq \underbrace{2 \cdot T\left(\frac{n}{2}\right)}_{\substack{\text{zwei Hälften} \\ \text{rekursiv sortieren}}} + \underbrace{d \cdot n}_{\text{für MERGE}}$$

und $T(1) = c$

Teleskopieren
 $\Rightarrow$

$$T(n) \leq O(n \log n)$$

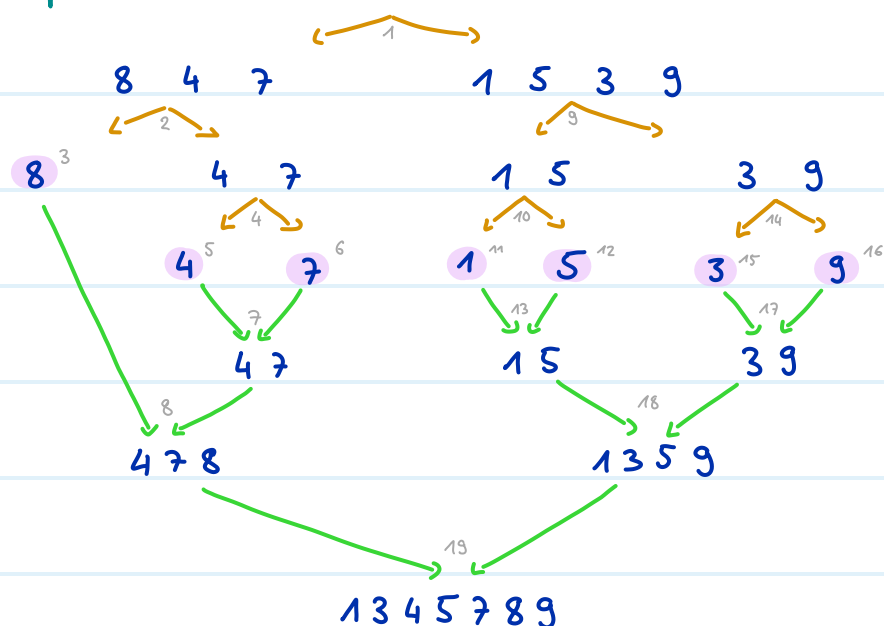
w
für Tiefe des Rekursionsbaumes
MERGE

Zusätzlicher Speicher $O(n)$ für das Array B

Bsp

8 4 7 1 5 3 9

Reihenfolge im Algorithmus



split up

Base Case

MERGE

Aufgaben:

g) *Sorting algorithms:*

i) Consider the sequence 6, 5, 4, 1, 2, 3. How many **swaps** does Bubble Sort perform to sort this sequence? *Give the exact number of swaps required.*

$5 + 4 + 3 = \underline{12}$

6, 5, 4, 1, 2, 3 → 5, 4, 1, 2, 3, 6 → 4, 1, 2, 3, 5, 6 → 1, 2, 3, 4, 5, 6

ii) Consider the sequence 6, 5, 4, 1, 2, 3. How many swaps does Selection Sort perform to sort this sequence? *Give the exact number of swaps required.*

4 $6, 5, 4, 1, 2, 3 \rightarrow 3, 5, 4, 1, 2, 6 \rightarrow 3, 2, 4, 1, 5, 6 \rightarrow 3, 2, 1, 4, 5, 6 \rightarrow 1, 2, 3, 4, 5, 6$

iii) Let $n \in \mathbb{N}$ be an even number and consider the sequence with the following structure:

$$2, 1, 4, 3, 6, 5, \dots, n, n-1.$$

How many swaps does Insertion Sort perform to sort this sequence? *Give the exact number, not just the asymptotics.*

$$\frac{n}{2} \quad 2, 1, 4, 3, 6, 5, \dots, n, n-1$$

2, 1, 4, 3, 6, 5, ..., n, n-1

True/False Questions

- ☒ All four algorithms can sort any list of comparable elements correctly.
- ☒ Only Merge Sort can guarantee a worst-case time complexity better than $O(n^2)$. (among these four)
- ☒ Bubble Sort compares distant elements in the array during each pass.
- ☒ Insertion Sort inserts each new element into the already sorted part at the final correct position.
- ☒ Selection Sort may perform fewer comparisons if the input list is already sorted.
- ☒ Merge Sort always splits the list into two equal halves.
- ☒ Bubble Sort and Insertion Sort both have an average runtime of $O(n \log n)$.
- ☒ Merge Sort's runtime does not depend on the initial order of the elements.
- ☒ Selection Sort performs fewer swaps than Bubble Sort in most cases.
- ☒ Insertion Sort requires extra arrays to perform its sorting.
- ☒ Merge Sort typically requires additional memory proportional to the size of the input.
- ☒ Selection Sort always performs exactly $n - 1$ comparisons.
- ☒ Bubble Sort can terminate early when no swaps occur during a pass.
- ☒ Merge Sort is an example of a "divide and conquer" algorithm.
- ☒ Bubble Sort is usually faster than Merge Sort on large, random datasets.
- ☒ Selection Sort adapts its runtime when the input is already sorted.
- ☒ Merge Sort can be implemented recursively or iteratively.
- ☒ Bubble Sort, Insertion Sort, and Selection Sort all sort in-place.
- ☒ Merge Sort sorts in-place without using additional arrays.
- ☒ All four algorithms determine order by comparing pairs of elements.
- ☒ Merge Sort divides the list until each sublist contains exactly one element.
- ☒ The worst-case runtime of Bubble Sort and Selection Sort is the same.
- ☒ Insertion Sort moves elements to make space for the current value being inserted.
- ☒ Merge Sort avoids element comparisons when merging sublists.
- ☒ Selection Sort and Bubble Sort perform the same number of swaps in total.
- ☒ Insertion Sort can outperform Merge Sort on very small arrays.
- ☒ Bubble Sort is primarily used in practical applications for large datasets.
- ☒ Any sorting algorithm is in $\Omega(n \log n)$.

- ✓ Selection Sort can be better than Bubble Sort.
- ✗ Merge Sort is always faster than Insertion Sort.
- ✓ Merge Sort has $O(n^2)$ comparisons. $O(n \log n) \neq O(n^2)$
- ✗ Insertion Sort runs in $O(n)$ time on sorted input.
- ✗ Bubble Sort's best case is faster than Selection Sort's best case.
- ✗ Merge Sort's runtime depends on how sorted the array already is.
- ✗ Selection Sort's number of comparisons depends on input order.
- ✗ Every comparison-based sort algorithm must perform $\Omega(n \log n)$ comparisons in every case.

Varia

Anmerkung zu Maximum Subarray Sum (Code Expert)

Variante 1: Teilproblem: $R[i] :=$ grösste Summe eines Subarrays, das in Index i endet

Rekursion: $R[i+1] = \max\{a_{i+1}, R[i] + a_{i+1}\}$

Lösung: $\max_{1 \leq i \leq n} R_i$ oder 0 falls alle Arrayelemente negativ sind

Variante 2: Teilproblem: $R[i] :=$ grösste Summe eines Subarrays, das in Index i endet,
oder 0 falls sie negativ ist

Rekursion: $R[i+1] = \max\{0, R[i] + a_{i+1}\}$

Lösung: $\max_{1 \leq i \leq n} R_i$

Beispiel: Array: $[2, -5, 4, 1, -2, 7, -3, -14, 1, 2]$

R mit Variante 1. $[2, -3, 4, 5, 3, 10, 7, -7, 1, 3]$

R mit Variante 2. $[2, 0, 4, 5, 3, 10, 7, 0, 1, 3]$

Formeln für Summen

Index Shift:
$$\sum_{i=a}^b f(i) = \sum_{i=a+c}^{b+c} f(i-c)$$

Reverse:
$$\sum_{i=a}^b f(i) = \sum_{i=a}^b f(a+b-i) \quad \text{und insbesondere} \quad \sum_{i=0}^n f(i) = \sum_{i=0}^n f(n-i)$$

Number of elements:
$$\sum_{i=a}^b 1 = b - a + 1 \quad (\text{falls } b \geq a)$$

Split up by range:
$$\sum_{i=a}^b f(i) = \sum_{i=a}^m f(i) + \sum_{i=m+1}^b f(i) \quad (\text{falls } a \leq m < b)$$

Peeling off terms:
$$\sum_{i=a}^b f(i) = f(a) + \sum_{i=a+1}^b f(i) = \left(\sum_{i=a}^{b-1} f(i) \right) + f(b) \quad (\text{falls } a < b)$$

Linearity:
$$\sum_{i=a}^b (f(i) \pm g(i)) = \sum_{i=a}^b f(i) \pm \sum_{i=a}^b g(i) \quad \text{and} \quad \sum_{i=a}^b c \cdot f(i) = c \sum_{i=a}^b f(i)$$

Sum of one element:
$$\sum_{i=a}^a f(i) = f(a)$$

Index renaming:
$$\sum_{i=a}^b f(i) = \sum_{j=a}^b f(j)$$

Swap order:
$$\sum_{i=a}^b \sum_{j=c}^d f(i, j) = \sum_{j=c}^d \sum_{i=a}^b f(i, j)$$

Triangular bounds:
$$\sum_{i=1}^n \sum_{j=1}^i f(i, j) = \sum_{j=1}^n \sum_{i=j}^n f(i, j)$$

Upper bound: $\sum_{i=a}^b f(i) \leq M (b-a+1)$ falls $f(i) \leq M$ für $i = a, \dots, b$

Lower bound: $\sum_{i=1}^n f(i) \geq \sum_{i=\lceil \frac{n}{2} \rceil}^n f(i) \geq \underbrace{(n - \lceil \frac{n}{2} \rceil + 1)}_{\text{Anzahl Terme}} \cdot f(\lceil \frac{n}{2} \rceil) \geq \frac{n}{2} \cdot f(\frac{n}{2})$ falls f monoton steigend ist

Integral trick: $\int_{a-1}^b f(x) dx \leq \sum_{i=a}^b f(i) \leq \int_a^{b+1} f(x) dx$ falls f monoton steigend ist

Aufgabe

Prove that $\sum_{i=1}^n \log(i^3) i^2 = \Theta(n^3 \log n)$

obere Schranke: $\sum_{i=1}^n \log(i^3) i^2 \leq n \log(n^3) n^2 = n^3 \cdot 3 \log(n) = O(n^3 \log(n))$
 $\uparrow$
 log. Gesetz

untere Schranke: $\sum_{i=1}^n \log(i^3) i^2 \geq \sum_{i=\lceil \frac{n}{2} \rceil}^n \log(i^3) i^2$

$$\geq \underbrace{(n - \lceil \frac{n}{2} \rceil + 1)}_{\text{Anzahl Terme}} \cdot \log\left(\left(\frac{n}{2}\right)^3\right) \left(\frac{n}{2}\right)^2$$

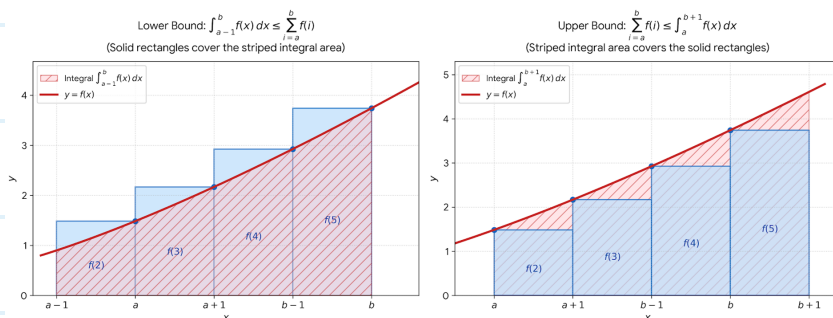
$$\geq \frac{n}{2} \cdot \log\left(\left(\frac{n}{2}\right)^3\right) \cdot \left(\frac{n}{2}\right)^2$$

$$= 3 \frac{n}{2} \cdot \log\left(\frac{n}{2}\right) \left(\frac{n}{2}\right)^2$$

$$= \frac{3}{8} n^3 (\log(n) - \log(2))$$

$$\geq \Omega(n^3 \log(n))$$

daraus folgt: $\sum_{i=1}^n \log(i^3) i^2 = \Theta(n^3 \log(n))$



Neue Gruppen

Group	Names	Emails
Group 1	Max Schaffner Charlyne Dippon	schaffnerma@student.ethz.ch cdippon@student.ethz.ch
Group 2	Gréta Frei Noa Kündig	grfrei@student.ethz.ch nkuendig@student.ethz.ch
Group 3	Lauren Thai Simon Spycher	lathai@student.ethz.ch sispycher@student.ethz.ch
Group 4	Besart Memisi Marc Leonard Overbeck	bmemisi@student.ethz.ch moverbeck@student.ethz.ch
Group 5	Paul Iurian Lucien Isler	piurian@student.ethz.ch islerlu@student.ethz.ch
Group 6	Raghav Tandon Sih-Cheng Chen	tandonr@student.ethz.ch chensih@student.ethz.ch
Group 7	Nicolas Fierz Floria Schwedt	nfierz@student.ethz.ch fschwedt@student.ethz.ch
Group 8	Durvish Battini Lucien Gees Noel Hobi	dbattini@student.ethz.ch lugees@student.ethz.ch hobino@student.ethz.ch
Group 9	Nico Realì Benjamin Stupf	nreali@student.ethz.ch bstupf@student.ethz.ch
Group 10	Elaine Banaag Aquino Silvan Wohlgensinger	ebanaag@student.ethz.ch swohlgensing@student.ethz.ch
Group 11	Nelio Gautschi Colin Jenny	ngautsch@student.ethz.ch cojenny@student.ethz.ch
Group 12	Matin Kamil Saih Zhentian Qi	kamilm@student.ethz.ch zhenqi@student.ethz.ch