QUIZ-NACHBESPRECHUNG

Consider the following pseudocode snippet:

```
i ← 1
while i ≤ n:
    i ← i + 1
    j ← 1
    while j ≤ n:
        j ← j + 1
        f()
```

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

$\sum_{i=1}^{n}\left(\sum_{j=1}^{n} 1\right)$ $\left(\sum_{i=1}^{n} 1\right) + \left(\sum_{j=1}^{n} 1\right)$

Which sorting algorithm from the lecture does the pseudocode snippet below implement?

```
for j = 1, 2, ..., n do:
    for i = 1, 2, ..., n − 1 do:
        if A[i] > A[i + 1] then:
            Swap A[i] and A[i + 1]
```

a) Bubblesort
b) Selectionsort
c) Mergesort
d) Insertionsort

Which of the following sorting algorithms have the invariant that: *after j steps, the j largest elements are at their correct place?*

~~a) Mergesort~~
b) Selectionsort
c) Bubblesort
~~d) Insertionsort~~

Every comparison-based algorithm for searching in a sorted array of size n needs at least $\Omega(\log n)$ comparisons for every input.

True or False ?

Suppose we apply *insertion sort* to the array $A_n = [2, 3, 4, \ldots, n-1, n, 1]$.

(E.g., $A_7 = [2, 3, 4, 5, 6, 7, 1]$).

Let $s(n)$ be the number of swap operations that are performed before the array is fully sorted (in ascending order).

Which of the following statements about $s(n)$ is true?

a) $s(n) = O(n)$
b) $s(n) = \Omega(n^2)$

SERIE 2 - COMMON MISTAKES

Aufgabe 2.2

$$\lim_{n\to\infty} \underbrace{\frac{1}{1000}n^3 - \frac{1}{100}n^2 - \frac{1}{10}n}_{\text{"}\infty\text{"} - \text{"}\infty\text{"} - \text{"}\infty\text{"}} = \lim_{n\to\infty} \frac{n^3}{1000}\underbrace{\left(1 - \underbrace{\frac{10}{n} - \frac{100}{n^2}}_{\to 0}\right)}_{\to 1} = \lim_{n\to\infty}\frac{n^3}{1000} = \infty$$

$\lim_{n\to\infty} \left(\frac{4}{e}\right)^n = \infty$ $\leftarrow$ hier muss zur ausreichenden Begründung $\frac{4}{e} > 1$ angegeben werden

$\lim_{n\to\infty} n^x$ dann $\lim_{n\to\infty} x$ seperat berechnet

besser: $\lim_{n\to\infty} e^{\ln(n)x}$ und dann $\lim_{n\to\infty} \ln(n)x$ seperat berechnen

Beispiel bei dem solch eine Vorgehensweise problematisch sein kann

$$\lim_{n\to\infty}\left(1+\frac{1}{n}\right)^n = e$$

immer die lim-Notation nutzen, sie wegzulassen ist falsch!

Beispiel: $\lim_{n\to\infty} \frac{n^2}{n^3} = \frac{1}{n} = 0$

gilt so nicht

NACHBESPRECHUNG SERIE 3

Aufgabe 3.2

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 $0 \le i < \mathsf{length}(S)$. For example, the prefixes of $S =$ "0110" are "0", "01", "011" and "0110".

Given a 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 $S =$ "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 prefix contains 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 $\rightarrow O(n)$

```
function PREFIXTABLE(S)        T[i] := #prefixes of S with exactly i ones
    T[0..n] ← a new array of size (n+1)            ▷ Initialize array
    s ← 0
    for i ← 0, ..., n−1 do                         ▷ Enumerate all prefixes S[0..i]
        s ← s + S[i]                               ▷ s saves the number of "1" in S[0..i]
        T[s] ← T[s] + 1                            ▷ S[0..i] is a prefix with s "1"
    return T
```

(c) Consider an integer $m \in \{0, 1, \ldots, n-2\}$. 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 $i \le m < j$.

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 $i \le m < j$ 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 $S[m+1..n-1]$.*

Algorithm 3

```
function SPANNING(m, k, S)
    T1 ← SUFFIXTABLE(S[0..m])
    T2 ← PREFIXTABLE(S[m+1..n−1])
    return Σ_{p=max(0,k−(n−m−1))}^{min(k,m+1)} (T1[p] · T2[k−p])
```

Runtime: $O(n)$.

Aufgabe 3.1b)

(b) Prove the following statements.

Hint: *For these examples, computing the limits as in Theorem 1 is hard or the limits do not even exist. Try to prove the statements directly with inequalities as in the definition of the O-notation.*

(1) $\sqrt{n^2+n+1} = \Theta(n)$

(2) $\sum_{i=1}^{n} \log(i^i) \ge \underline{\Omega}(n^2 \log n)$

Hint: *Recall exercise 1.2 and try to do an analogous computation here.*

(3) $\log(n^2+n) = \Theta(\log(n+1)) = \Theta(\log(n))$

(1) $\sqrt{n^2} = n \le \sqrt{n^2+n+1} \le \sqrt{n^2+2n+1} = \sqrt{(n+1)^2} = n+1$

(2) $\sum_{i=1}^{n} \log(i^i) = \sum_{i=1}^{n} i \cdot \log(i) \ge \sum_{\lceil \frac{n}{2} \rceil}^{n} i \cdot \log(i) \ge \frac{n}{2} \cdot \frac{n}{2} \cdot \log\left(\frac{n}{2}\right) \ge \Omega(n^2 \log n)$

(3) $\log(n+1) \le \log(n^2+n) = \log(n \cdot (n+1)) = \log(n) + \log(n+1) \le 2\log(n+1)$

Aufgabe 3.3

Algorithm 2

$i \leftarrow 1$
while $i \leq 2n$ **do**
  $j \leftarrow 1$
  while $j \leq i^3$ **do**
    $k \leftarrow n$
    while $k \geq 1$ **do**
      $f()$
      $k \leftarrow k - 1$
    $j \leftarrow j + 1$
  $i \leftarrow i + 1$

$$\#\text{calls} = \sum_{i=1}^{2n} \sum_{j=1}^{i^3} n = \sum_{i=1}^{2n} i^3 \cdot n = n \sum_{i=1}^{2n} i^3 = n \cdot \left(\frac{2n \cdot (2n+1)}{2}\right)^2 = \dots = \Theta(n^5)$$

Suchalgorithmen

Fall: Array ist unsortiert → lineare Suche

```
for i ← 1..n do
    if A[i] = b then return i
return "nicht gefunden"
```

Fall: Array ist sortiert → binäre Suche

Idee: nimm mittleres Element A[m], key gesuchter Wert
- wenn A[m] = key, haben wir das Element gefunden, yay!
- A[m] > key ⇒ suche links weiter (rekursiv)
- A[m] < key ⇒ suche rechts weiter (rekursiv)

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

```
ℓ ← 1; r ← n                         ▷ Initialer Suchbereich
while ℓ ≤ r do
    m ← ⌊(ℓ + r)/2⌋
    if A[m] = b then return m        ▷ Element gefunden
    else if A[m] > b then r ← m − 1  ▷ Suche links weiter
    else ℓ ← m + 1                   ▷ Suche rechts weiter
return "Nicht vorhanden"
```

Laufzeit: $O(\log n)$

geht es besser → Nein

Beweis mit Hilfe eines Entscheidungsbaums
- innere Knoten ≙ Vergleich
- Blätter ≙ Rückgabewerte
- Höhe des Baums ≙ #Vergleiche im worst case

- jedes Element muss gefunden werden können + Option "nicht gefunden"
  → mind. n+1 Blätter

⇒ $\underline{n+1} \leq \#\text{Knoten im Entscheidungsbaum} \leq \underline{2^{\overset{\text{Höhe}}{h}+1}}$, da Binärbäume der Höhe h max. $2^{h+1}-1$ Knoten haben können

⟺ $h \geq \log_2(n+1) - 1 = \Omega(\log n)$

Sortieralgorithmen

BUBBLESORT

Invariante: Nach j Iterationen der äußeren Schleife sind die j größten Elemente an der richtigen Stelle

Laufzeit: $\Theta(n^2)$ Vergleiche, $O(n^2)$ Vertauschungen $\Rightarrow O(n^2)$

Pseudocode:
```
for j ← 1...n do
    for i ← 1...n-1 do
        if A[i] > A[i+1] then swap
```

SELECTIONSORT

Invariante: Nach j Iterationen der äußeren Schleife sind die j größten Elemente an der richtigen Stelle

Laufzeit: $\Theta(n^2)$ Vergleiche, $O(n)$ Vertauschungen $\Rightarrow O(n^2)$

Pseudocode:
```
for j = n...1 do
    k ← Index des Maximums in A[1...j]
    vertausche A[j] und A[k]
```

INSERTIONSORT

Invariante: Nach k Iterationen sind die ersten k Elemente sortiert.

binäre Suche

Laufzeit: $O(n \log n)$ Vergleiche, $O(n^2)$ Vertauschungen $\Rightarrow O(n^2)$

Pseudocode:

Insertion-Sort($A[1..n]$)

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

MERGESORT

Idee:
- Array sortieren bis man ein atomares Element erhält
- Teilarrays mergen und dabei sortieren

Laufzeit: $O(n \log n)$

Pseudocode:

```
MERGESORT(A[1..n], l, r)                         ▷ sortiert A[l, ..., r]

if l < r then
    m ← ⌊(l + r)/2⌋
    MERGESORT(A, l, m)                           ▷ sortiere linke Hälfte
    MERGESORT(A, m + 1, r)                       ▷ sortiere rechte Hälfte
    MERGE(A, l, m, r)                            ▷ verschmelze beide Hälften
```

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

B ← new Array with r − l + 1 cells               ▷ so gross wie A[l, ..., r]
i ← l                                            ▷ erstes unbenutztes Element in linker Hälfte
j ← m + 1                                        ▷ erstes unbenutztes Element in rechter Hälfte
k ← 1                                            ▷ nächste Position in B
while i ≤ m and j ≤ r do                         ▷ beide Hälften noch nicht ausgeschöpft
    if A[i] < A[j] then
        B[k] ← A[i]
        i ← i + 1
        k ← k + 1
    else
        B[k] ← A[j]
        j ← j + 1
        k ← k + 1
übernimm Rest links bzw. rechts                  ▷ wenn die andere Hälfte ausgeschöpft ist
kopiere B nach A[l, ..., r]
```

FS23

Claim	true	false
There exist arrays of length n which can be sorted with `BubbleSort` after $\Theta(n)$ swaps.	☒	☐
There exist arrays of length n for which the runtime of `InsertionSort` is $\Theta(n)$.	☐	☒
Consider a sequence of n numbers $\{x_1, ..., x_n\}$, where $0 \leq x_i \leq 1000$, $\forall i = 1, ..., n$, is given as input. There exists an algorithm with runtime $O(n)$ which sorts any such sequence.	☐	☐
There exist a comparison-based sorting algorithm that can sort any array of length n in runtime $O(n)$.	☐	☒

Beispiel: 5|1|1|2|3|4

FS22

Claim	true	false
In the worst case, selection sort ($n-1$) needs less swaps than insertion sort ($O(n^2)$).	☒	☐
The worst case for bubble sort is when the array is already sorted.	☐	☒
Quicksort is asymptotically faster than bubble sort in the worst case.	☐	☐

FS20

/ 2 P

e) *Sorting algorithms:*

Below you see four sequences of snapshots, each obtained during the execution of one of the following five algorithms: `InsertionSort`, `SelectionSort`, `QuickSort`, `MergeSort`, and `BubbleSort`. For each sequence, write down the corresponding algorithm.

8	6	4	2	5	1	3	7
6	4	2	5	1	3	7	8
4	2	5	1	3	6	7	8

Algorithm: BubbleSort

8	6	4	2	5	1	3	7
1	6	4	2	5	8	3	7
1	2	4	6	5	8	3	7

Algorithm: SelectionSort

8 |	6 |	4 |	2 |	5 |	1 |	3 |	7
6	8 |	2	4 |	1	5 |	3	7
2	4	6	8 |	1	3	5	7

Algorithm: MergeSort

8	6	4	2	5	1	3	7
6	8	4	2	5	1	3	7
4	6	8	2	5	1	3	7

Algorithm: InsertionSort

Serie HS23: Proof correctness of Bubble Sort via principle of mathematical induction

Hint: *Use the invariant $I(j)$ that was introduced in the lecture: "After j iterations the j largest elements are at the correct place."*

We prove the invariant in the hint by mathematical induction on j.

- **Base Case.**
  We prove the statement for $j = 1$. Assume that the largest element of A is at position l in the beginning. After the first $l - 1$ iterations of the second for-loop, it is still at position l. For all further steps with $i \geq l$, $A[i]$ contains the largest element and thus the largest element is swapped to position $i + 1$. Hence, in the end the largest element is at position n, which shows $I(1)$.

- **Induction Hypothesis.**
  We assume that the invariant is true for $j = k$ for some $k \in \mathbb{N}$, $k < n$, i.e. after k iterations the k largest elements are at the correct position.

- **Inductive Step.**
  We must show that the invariant also holds for $j = k + 1$. By the induction hypothesis the k largest elements are at the correct position after k steps, i.e. at the positions $A[n - k + 1 \ldots n]$. We now consider step $k + 1$. Note that in this iteration the positions of the k largest elements are not changed since for $i \geq n - k$, we will never have $A[i] > A[i + 1]$. Thus, in order to show $I(k + 1)$ it is enough to show that after step $k + 1$ also the $(k + 1)$st largest element is at the correct position. The $(k + 1)$st largest element is the largest element of $A[1 \ldots n - k]$ (all elements that are larger than it come later by $I(k)$). Thus, by the argumentation in the base case, after $i = n - k - 1$ iterations in the second for-loop, it is at position $A[n - k]$. But for the other k iterations of the second for-loop, nothing changes as was already argued before (the largest elements do not change their position). Thus, after step $k + 1$, the $k + 1$ largest elements are at the correct position, which shows $I(k + 1)$.

By the principle of mathematical induction, $I(j)$ is true for all $j \in \mathbb{N}$, $j \leq n$. In particular, $I(n)$ holds, which means that after the first n iterations the n largest elements are at the correct position. This shows that after n steps the array is sorted, which shows correctness of the Bubble Sort algorithm.

HS20

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.*

12

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

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

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

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

$\frac{n}{2}$