Informatik II Woche 7

Improved Insertion Sort, Laufzeit bei Rekursion, Mergesort

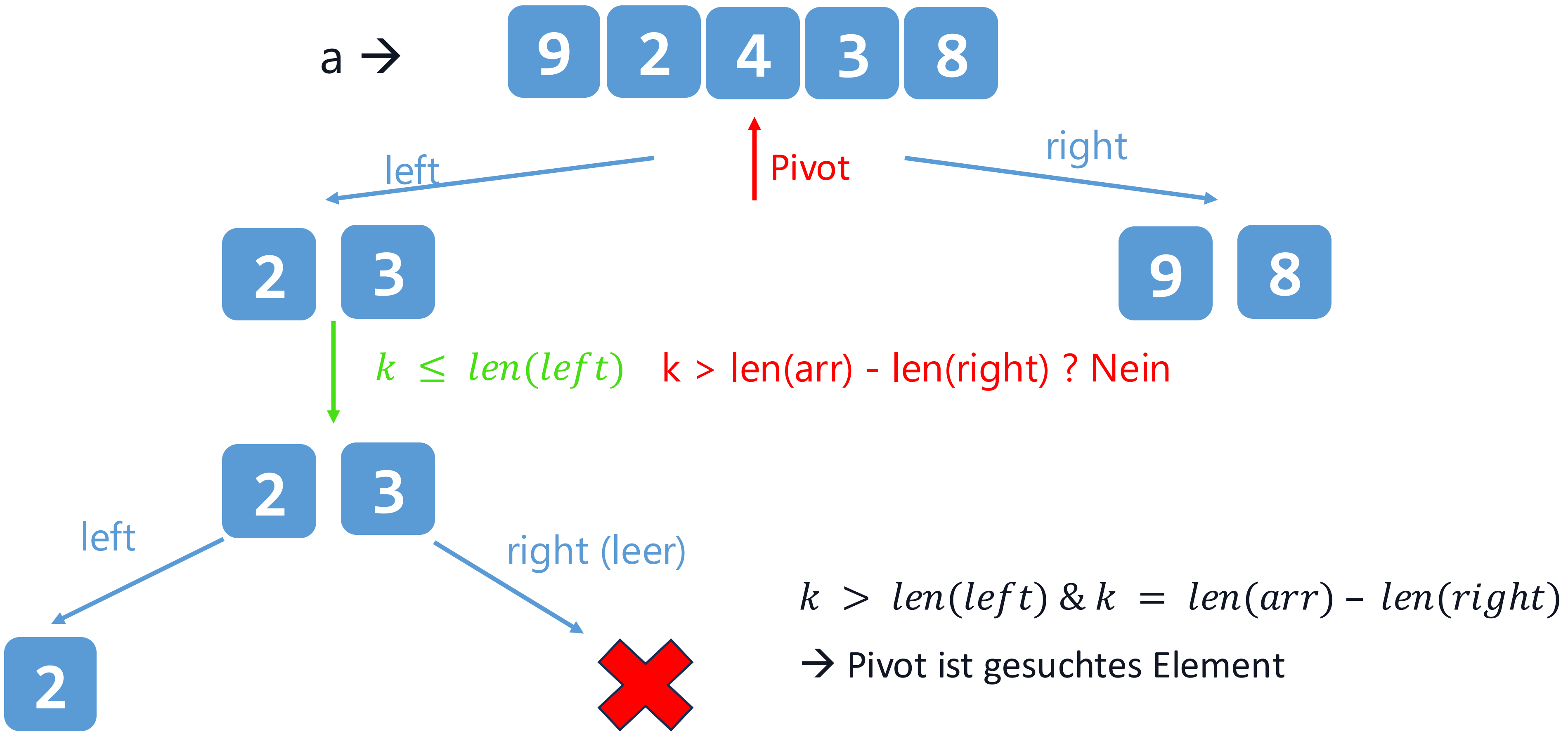
Website: n.ethz.ch/~kvaratharaja/

Die Slides basieren auf den offiziellen Übungsslides der Kurswebsite: https://lec.inf.ethz.ch/mavt/informatik2/2025/

k-kleinstes Element

Die Funktion find_kth_smallest soll das k-kleinste Element in einer Liste findet. Vervollständingen Sie die Funktion mit Hilfe eines Teile-und-Beherrsche (divide and conquer)-Verfahrens.

Beispiel: finde 2. kleinstes Element:



Heute

1. **Insertion Sort improved**
2. **Asymptotische Laufzeit Rekursiver Funktionen**
3. **Divide & Conquer Sortieralgorithmen**
4. **Inclass-Exercise**
5. **Hausaufgaben**

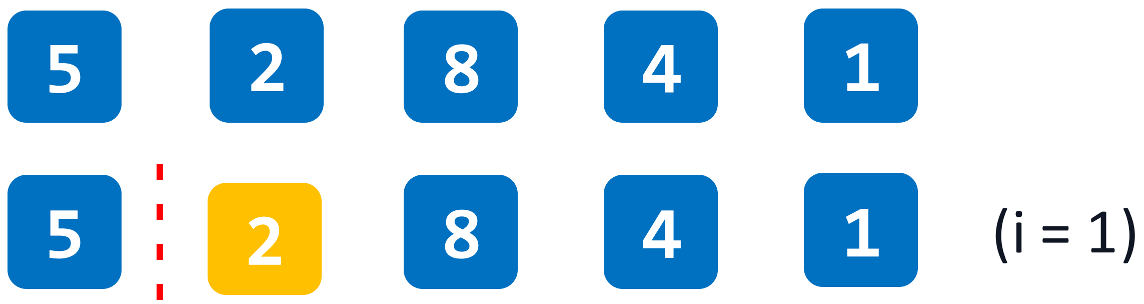
1. Insertion Sort improved

Letztes Mal: Insertion Sort

5 2 8 4 1

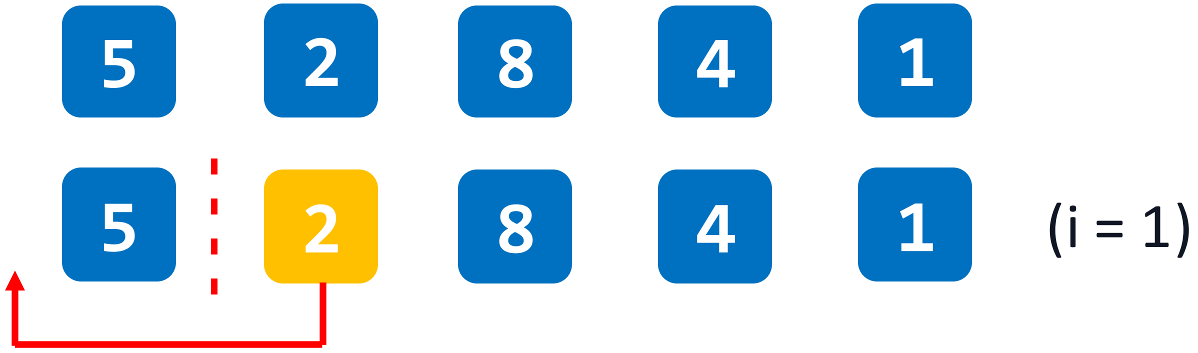
- **Schleifeninvariante:** Vor Iteration i sind Elemente in li[:i] sortiert.
- Bei Iteration i, das i-te Element an der korrekten Position in die sortierte Teilliste li[:i] einfügen.
- Wiederholen bis das gesamte Array sortiert ist (i = n).

Letztes Mal: Insertion Sort



- **Schleifeninvariante:** Vor Iteration i sind Elemente in li[:i] sortiert.
- Bei Iteration i, das i-te Element an der korrekten Position in die sortierte Teilliste li[:i] einfügen.
- Wiederholen bis das gesamte Array sortiert ist (i = n).

Letztes Mal: Insertion Sort



- **Schleifeninvariante:** Vor Iteration i sind Elemente in li[:i] sortiert.
- Bei Iteration i, das i-te Element an der korrekten Position in die sortierte Teilliste li[:i] einfügen.
- Wiederholen bis das gesamte Array sortiert ist (i = n).

Letztes Mal: Insertion Sort

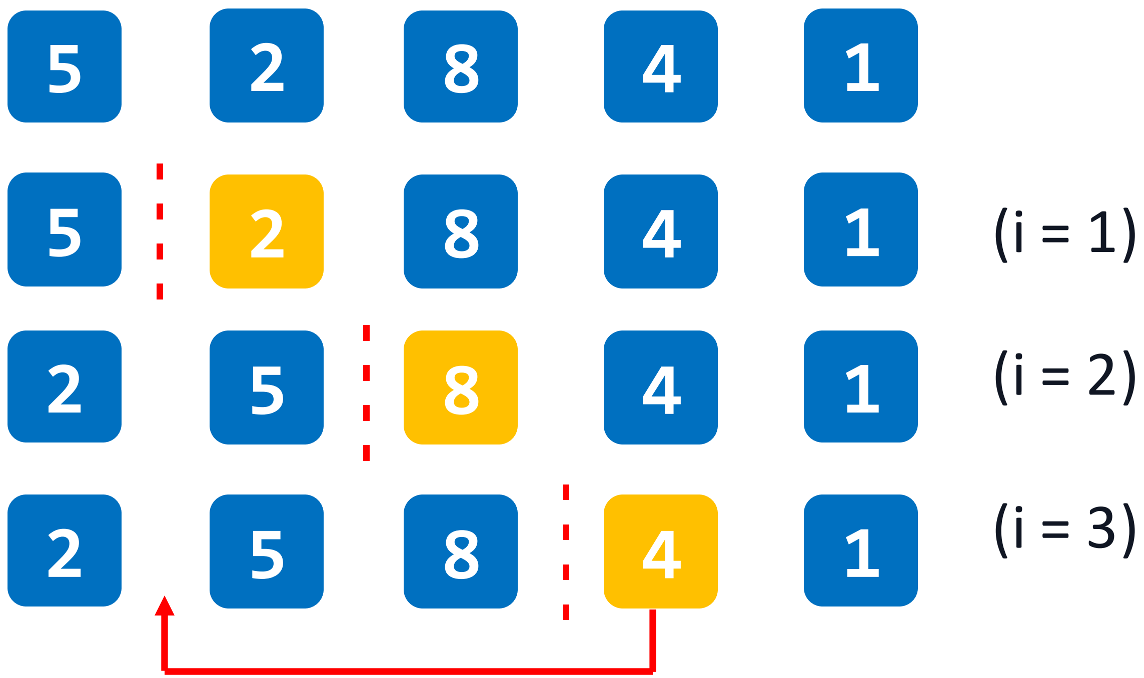
- **Schleifeninvariante:** Vor Iteration i sind Elemente in li[:i] sortiert. (Für Selection Sort: Vor iteration i enthält li[:i] die i niedrigsten Elemente von li in sortierter Reihenfolge.)

- Bei Iteration i, das i-te Element an der korrekten Position in die sortierte Teilliste li[:i] einfügen.

- Wiederhole bis alles sortiert ist

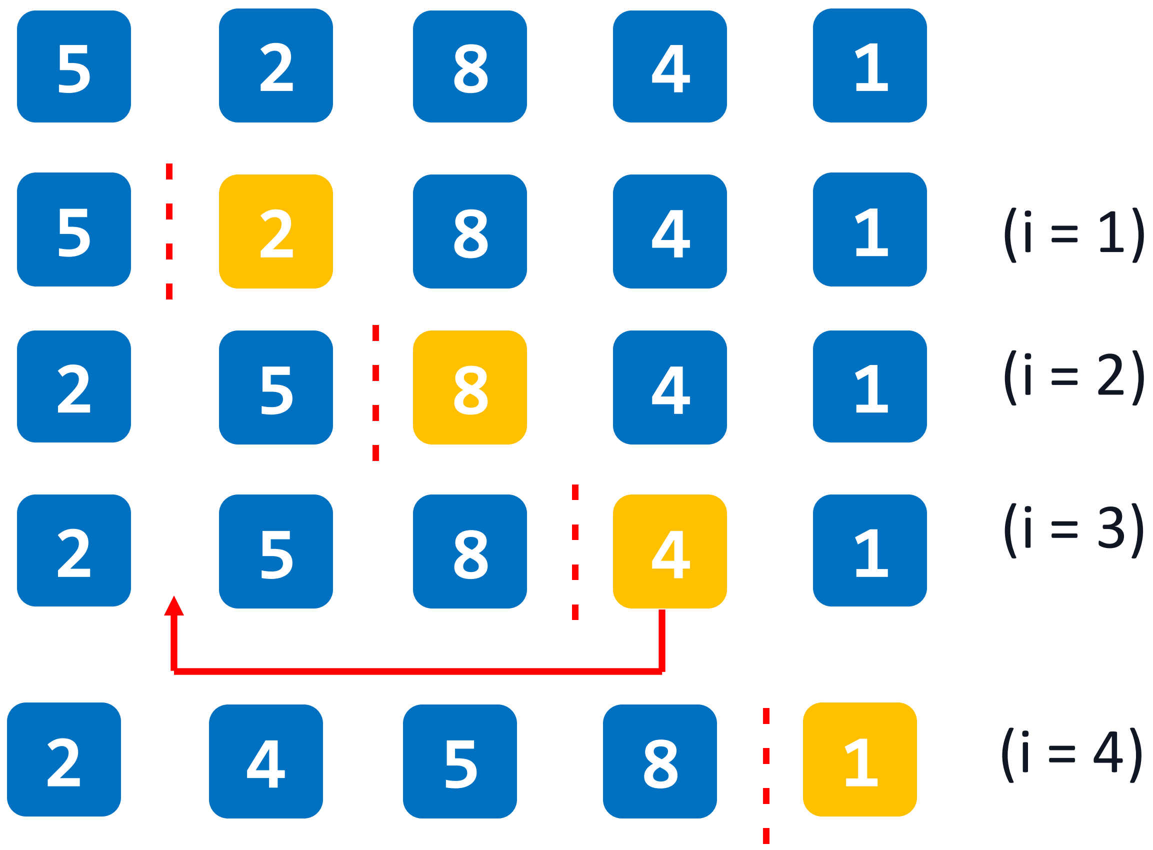
  (i = n)

Letztes Mal: Insertion Sort



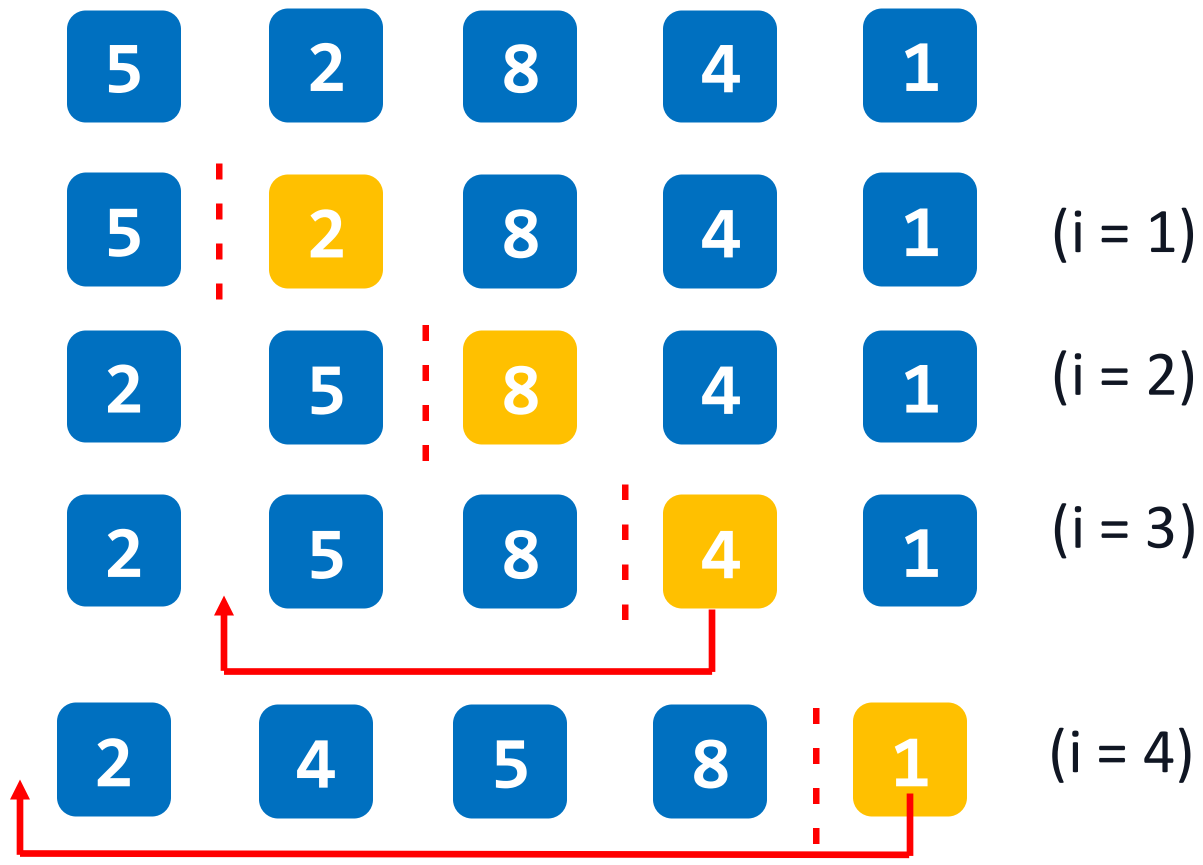
- **Schleifeninvariante:** Vor Iteration i sind Elemente in li[:i] sortiert.
- Bei Iteration i, das i-te Element an der korrekten Position in die sortierte Teilliste li[:i] einfügen.
- Wiederholen bis das gesamte Array sortiert ist (i = n).

Letztes Mal: Insertion Sort



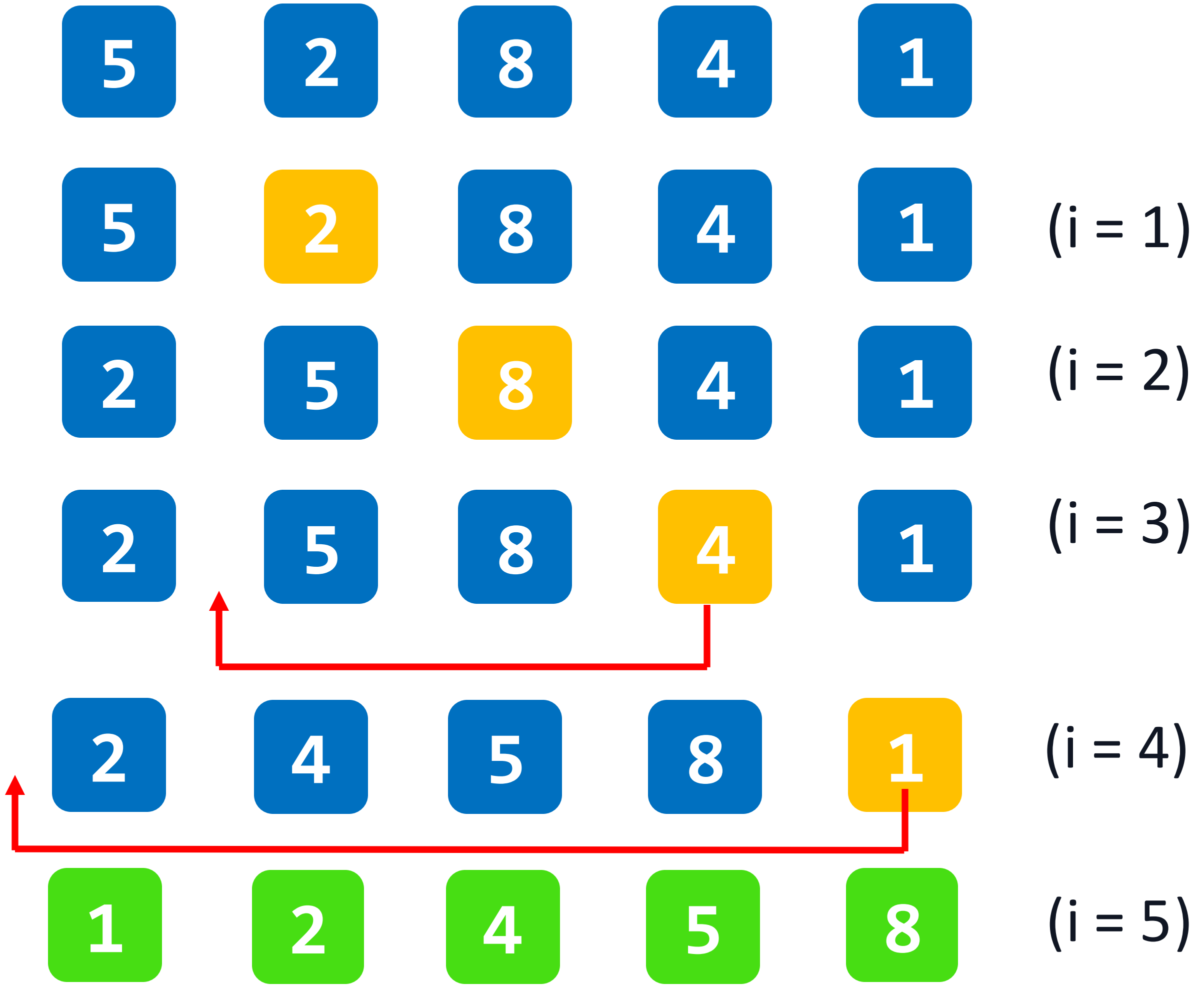
- **Schleifeninvariante:** Vor Iteration i sind Elemente in li[:i] sortiert.
- Bei Iteration i, das i-te Element an der korrekten Position in die sortierte Teilliste li[:i] einfügen.
- Wiederholen bis das gesamte Array sortiert ist (i = n).

Letztes Mal: Insertion Sort



- **Schleifeninvariante:** Vor Iteration i sind Elemente in li[:i] sortiert.
- Bei Iteration i, das i-te Element an der korrekten Position in die sortierte Teilliste li[:i] einfügen.
- Wiederholen bis das gesamte Array sortiert ist (i = n).

Letztes Mal: Insertion Sort



- **Schleifeninvariante:** Vor Iteration i sind Elemente in li[:i] sortiert.
- Bei Iteration i, das i-te Element an der korrekten Position in die sortierte Teilliste li[:i] einfügen.
- Wiederholen bis das gesamte Array sortiert ist (i = n).

Letztes Mal: Insertion Sort

Input: Array A = (A[1],...,A[n]), n ≥ 0.

Output: Output: Sortiertes Array A

```
for i in range(1,len(A)):
    j=i
    while j>0 and A[j-1] > A[j]:
        A[j], A[j-1] = A[j-1], A[j]
        j = j-1
```

Anzahl Vergleiche im schlechtesten Fall? $\Theta(n^2)$
Anzahl Vertauschungen im schlechtesten Fall? $\Theta(n^2)$

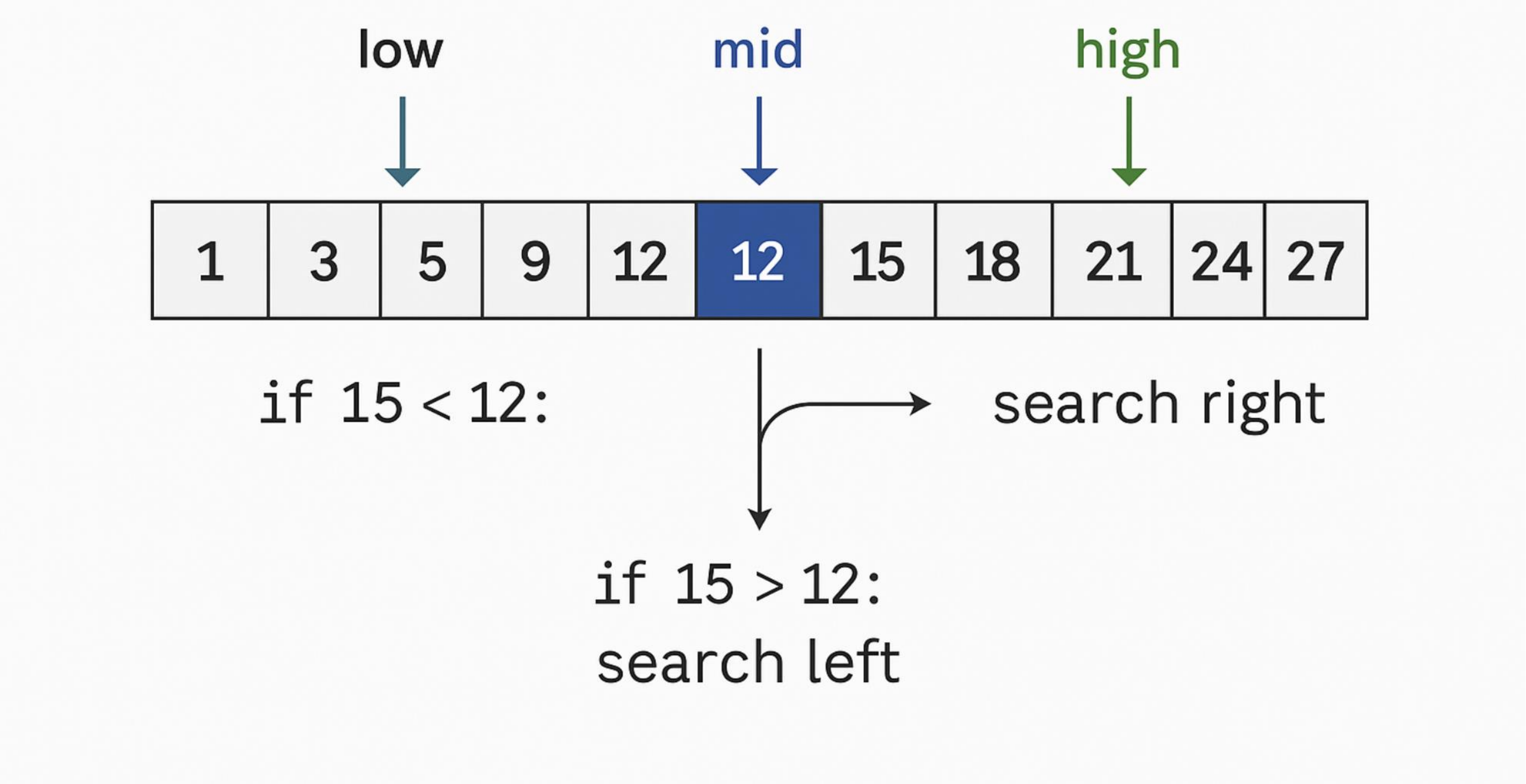
- **Frage:** Wie können wir die Anzahl Vergleiche für den schlechtesten Fall verbessern?
- Der Suchbereich (Einfügebereich) ist bereits sortiert. Konsequenz: binäre Suche möglich

Recap Binary Search

```
def bin_search(a, l, r, b):
    if r < l:
        return None
    else:
        m = (l + r) // 2
        if a[m] == b:
            return m
        elif b < a[m]:
            return bin_search(a, l, m-1, b)
        else: # a[m] > b
            return bin_search(a, m+1, r, b)
```

Recap Binary Search

Ist 15 im Array vorhanden?



Binärer Insertion Sort

Input: Array A = (A[1],...,A[n]), n ≥ 0.

Output: Output: Sortiertes Array A

```
for i in range(1,len(A)):
    x = A[i]
    p = BinarySearchIndex(A,0,i-1,x)
    for j in range(i-1,p-1,-1):
        A[j+1] = A[j]
    A[p] = x
```

Anzahl Vergleiche im schlechtesten Fall? $\sum_{k=1}^{n-1} a \cdot \log k = a \log((n-1)!) \in \Theta(n*\log(n))$

Anzahl Vertauschungen im schlechtesten Fall? $\sum_{k=2}^{n} (k-1) \in \Theta(n^2)$

2. Asymptotische Laufzeit Rekursiver Funktionen

Asymptotisches Verhalten: Notation

Was sind $O(g)$, $\Omega(g)$, $\Theta(g)$?

→ Funktionen!

- $O(g) = \{f: \mathbb{N} \rightarrow \mathbb{R} \mid \exists c > 0, n_0 \in \mathbb{N} \mid \forall n \geq n_0 : 0 \leq f(n) \leq c \cdot g(n)\}$
- $\Omega(g) = \{f: \mathbb{N} \rightarrow \mathbb{R} \mid \exists c > 0, n_0 \in \mathbb{N} \mid \forall n \geq n_0 : 0 \leq c \cdot g(n) \leq f(n)\}$
- $\Theta(g) = O(g) \cap \Omega(g)$

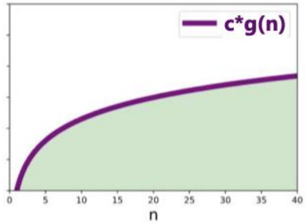
Aber was bedeutet das intuitiv gesehen? 🤔

Asymptotisches Verhalten: Intuition

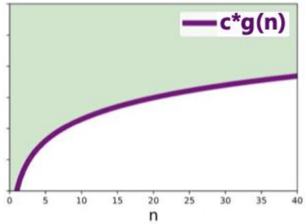
Also was sind $O(g), \Omega(g)$ oder $\Theta(g)$? → Asymptotische Ober- und Untergrenzen!

"der Graph von f bleibt immer im grünen Bereich ab einem Wert n"

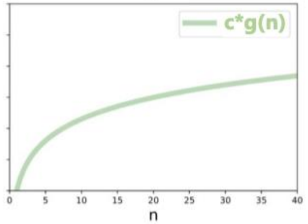
$f \in O(g)$: f wächst **nicht schneller** als c $\cdot g(n)$.



$f \in \Omega(g)$: f wächst **nicht langsamer** als c $\cdot g(n)$.



$f \in \Theta(g)$: f wächst **etwa gleich** wie c $\cdot\, g(n)$.



Übung 1

Gebe die Anzahl Aufrufe der Funktion f abhängig von n in der Θ-Notation an.

```python
#pre: n is a positive integer
def g(n):
    count = 0
    while n // (2 ** count) >= 1:
        f()
        count += 1
```

- Das Programm ruft die Funktion f auf, solange $n/2^{\text{count}} \geq 1$ ist.
- Umgeschrieben: $\log n \geq count$
- Die Variable $count$ misst die Anzahl der Iteration der while Schleife, und dabei auch die Anzahl Aufrufe der Funktion f.

→Darum ist diese Anzahl $\log n$.

→Anzahl $\Theta(\log n)$

Übung 2

Gebe die Anzahl Aufrufe der Funktion f abhängig von n in der Θ-Notation an.

```
#pre: n is a positive integer
def g(n):
    if n >= 1:
        f()
        g(n // 2)
```

Sei $T(n)$ die Anzahl der Aufrufe der Funktion f. Wenn $n \geq 1, gilt$: $T(n) = 1 + T\left(\frac{n}{2}\right)$

Teleskopieren: $T(n) = 1 + \boxed{T\left(\frac{n}{2}\right)} = 1 + \boxed{1 + T\left(\frac{n}{2^2}\right)} = k + T\left(\frac{n}{2^k}\right)$

- f wird aufgerufen, solange $\frac{n}{2^k} \geq 1$, → $\log n \geq k$, einsetzen in die Formel:
- $T(n) = \log n + \cancelto{0}{T(1)}$ → Aufrufe der Funktion: $\Theta(\log n)$

```python
#pre: n is a positive integer
def g(n):
    if n >= 1:
        for i in range(n):
            f()
        g(n // 2)
```

Sei $T(n)$ die Anzahl der Aufrufe der Funktion f. Wenn $n \geq 1, gilt$:

$$T(n) = n + \boxed{T\left(\frac{n}{2}\right)}$$

$$= n + \boxed{\frac{n}{2} + T\left(\frac{n}{2^2}\right)}$$

$$= n + \frac{n}{2} + \cdots + \frac{n}{2^k} + T\left(\frac{n}{2^{k+1}}\right)$$

$$= T(n) = n\sum_{j\ <=\ k}\frac{1}{2^j} + T\left(\frac{n}{2^{k+1}}\right)$$

$$\leq 2n + T\left(\cancel{\frac{n}{2^{k+1}}}\right)$$

Das gilt weil:

$$\sum_{j\ <=\ k} 1/2^j \leq 2$$

Es gibt k, wofür $\lfloor n/2k+1 \rfloor = 0$, dann auch $T(\lfloor n/2k+1 \rfloor) = T(0) = 0$.
Das bedeutet, dass $T(n) \leq 2n \in O(n)$
Analog dazu kann man zeigen, dass $T(n) \in \Omega(n)$, weil g(n) ruft f zumindest n-mal.
Darum, $T(n) \in \Theta(n)$.

Etwas umständlich oder? 🤔

Master Theorem

Recursive Runtime Trick, kein Teleskopieren mehr nötig! 😲

Step 1: Schreibe die rekursive Funktion T(n) in die folgende mathematische Form um:

$$T(n) = \underbrace{a \cdot T\left(^{n}/_{b}\right)}_{\text{recursive call(s)}} + \underbrace{\Theta\left(n^{k} \cdot \log^{p}(n)\right)}_{\text{Additional runtime in each call of T(n)}}$$

recursive call(s)

Additional runtime in each call of T(n) (e.g. for-loop). *Tipp: Θ() direkt bestimmen und Koeffizientenvergleich*

Step 2: Überprüfe die folgenden Cases, Runtime wird einer von denen sein

1. if a $> b^k$, then $T(n) = \theta(n^{\log_b(a)})$
2. if a $= b^k$, and
   a) if p > -1, then $T(n) = \theta(n^{\log_b(a)} \cdot \log^{p+1}(n))$
   b) if p $= -1$, then $T(n) = \theta(n^{\log_b(a)} \cdot \log(\log(n)))$
   c) if p < -1, then $T(n) = \theta(n^{\log_b(a)})$
3. if a $< b^k$, and
   a) if p >= 0, then $T(n) = \theta(n^k \cdot \log^p(n))$
   b) if p < 0, then $T(n) = \theta(n^k)$

Übung 3 mit Master Theorem

Gebe die Anzahl Aufrufe der Funktion f abhängig von n in der Θ-Notation an.

```
#pre: n is a positive integer
def g(n):
    if n >= 1:
        for i in range(n):
            f()
        g(n // 2)
```

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + \theta(n^k \cdot log^p(n))$$

1. if a > b^k, then $T(n) = \theta(n^{log_b(a)})$
2. if a = b^k, and
   a) if p > −1, then $T(n) = \theta(n^{log_b(a)} \cdot log^{p+1}(n))$
   b) if p = −1, then $T(n) = \theta(n^{log_b(a)} \cdot log(log(n)))$
   c) if p < −1, then $T(n) = \theta(n^{log_b(a)})$
3. if a < b^k, and
   a) if p >= 0, then $T(n) = \theta(n^k \cdot log^p(n))$
   b) if p < 0, then $T(n) = \theta(n^k)$

$T(n) = n + T\left(\frac{n}{2}\right)$, in diesem Beispiel $a = 1, b = 2, k = 1, p = 0$: case 3a)

→ $T(n) = \Theta(n)$

Übung 4 mit Teleskopieren

Sei $T(n)$ die Anzahl der Aufrufe der Funktion f.
Wenn $n \geq 1, gilt{:}\; T(n) = n + \boxed{2T(n/2)}$

$$= n + \boxed{2\left(\frac{n}{2}\right) + 2^2 T\left(\frac{n}{2^2}\right)}$$

$$= n + 2\left(\frac{n}{2}\right) + \ldots + 2^k \left(\frac{n}{2^k}\right) + 2^{(k+1)} T\left(\frac{n}{2^{k+1}}\right)$$

$$= n\,(k + 1) + 2^{(k+1)} T\left(\frac{n}{2^{k+1}}\right)$$

```python
#pre: n is a positive integer
def g(n):
    if n >= 1:
        for i in range(n):
            f()
        g(n // 2)
        g(n // 2)
```

- f wird aufgerufen, solange $\frac{n}{2^k} \geq 1$, → $\log n = k$, einsetzen in die Formel:

$$T(n) = n\,(\log n + 1) + 2n\,T\left(\frac{1}{2}\right) = n\,(\log n + 1) + 2n\,T(0) = n\,(\log n + 1).$$

→ Anzahl Aufrufe der Funktion: $\Theta(n \log n)$

Übung 4 mit Master Theorem

Gebe die Anzahl Aufrufe der Funktion f abhängig von n in der Θ-Notation an.

```
#pre: n is a positive integer
def g(n):
    if n >= 1:
        for i in range(n):
            f()
        g(n // 2)
        g(n // 2)
```

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + \theta(n^k \cdot log^p(n))$$

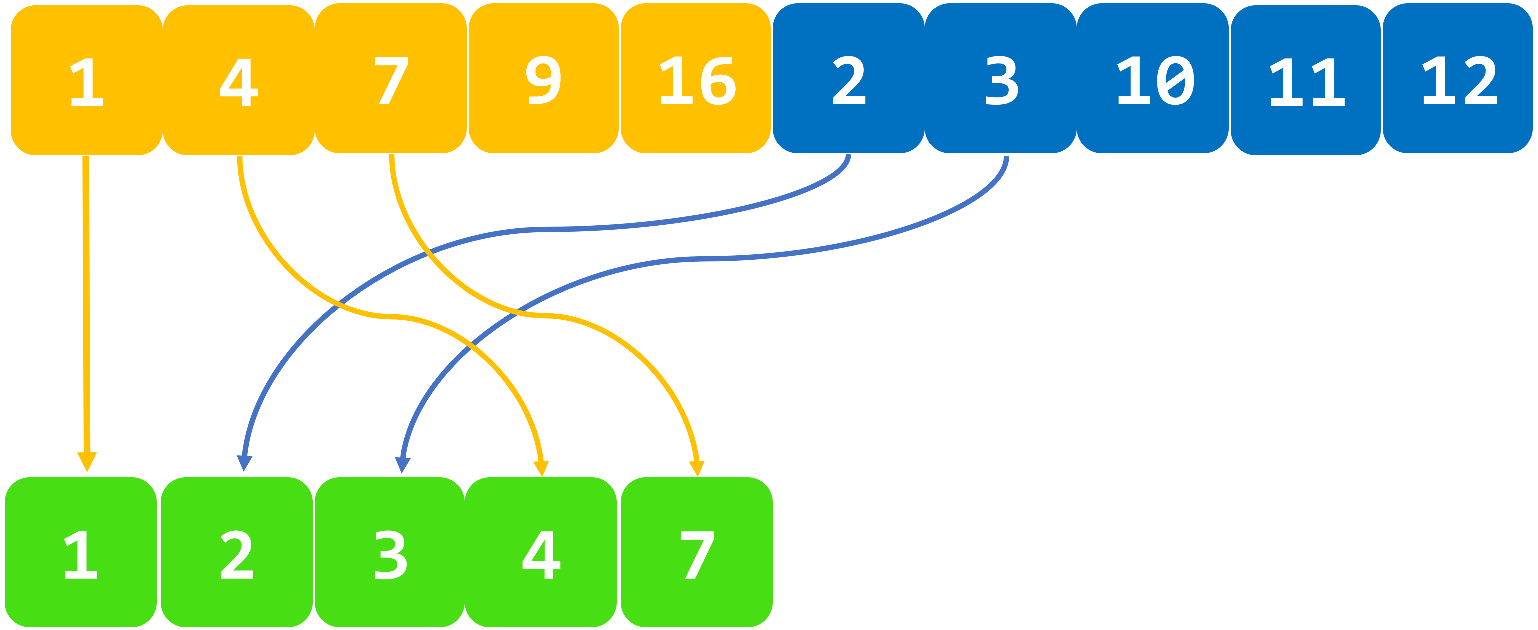
1. if a $> b^k$, then $T(n) = \theta(n^{log_b(a)})$
2. if a $= b^k$, and
   a) if p > -1, then $T(n) = \theta(n^{log_b(a)} \cdot log^{p+1}(n))$
   b) if p $= -1$, then $T(n) = \theta(n^{log_b(a)} \cdot log(log(n)))$
   c) if p < -1, then $T(n) = \theta(n^{log_b(a)})$
3. if a $< b^k$, and
   a) if p >= 0, then $T(n) = \theta(n^k \cdot log^p(n))$
   b) if p < 0, then $T(n) = \theta(n^k)$

$T(n) = n + 2T\left(\frac{n}{2}\right)$, hier $a = 2, b = 2, k = 1, p = 0$, case 2a)

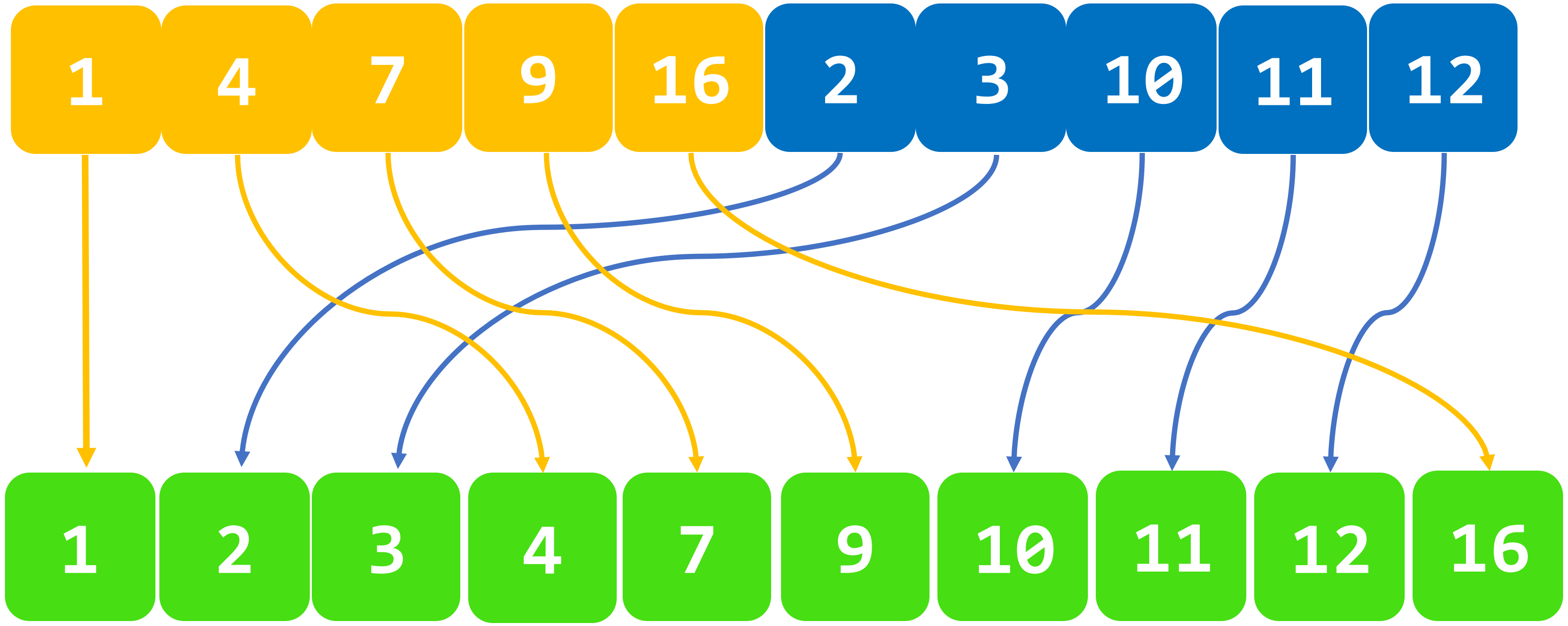
→ $T(n) = \Theta(n \log n)$

3. Divide & Conquer Sortieralgorithmen

Merge



Merge



Algorithmus merge

```python
def merge(a1, a2):
    b,i,j= [], 0, 0
    while i < len(a1) and j < len(a2):
        if a1[i] < a2[j]:
            b.append(a1[i]
            i += 1
        else:
            b.append(a2[j])
            j += 1
    b += a1[i:]
    b += a2[j:]
    return b
```

Algorithmus merge_sort

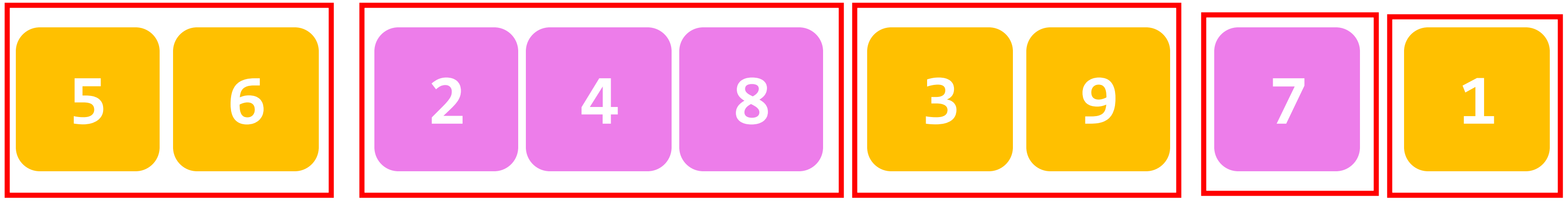
```
def merge_sort(a):
    if len(a) <= 1:
        return a
    else:
        sorted_a1 = merge_sort(a[:len(a) // 2])
        sorted_a2 = merge_sort(a[:len(a) // 2])
        return merge(sorted_a1, sorted_a2)
```

Natürlicher 2-Wege Mergesort

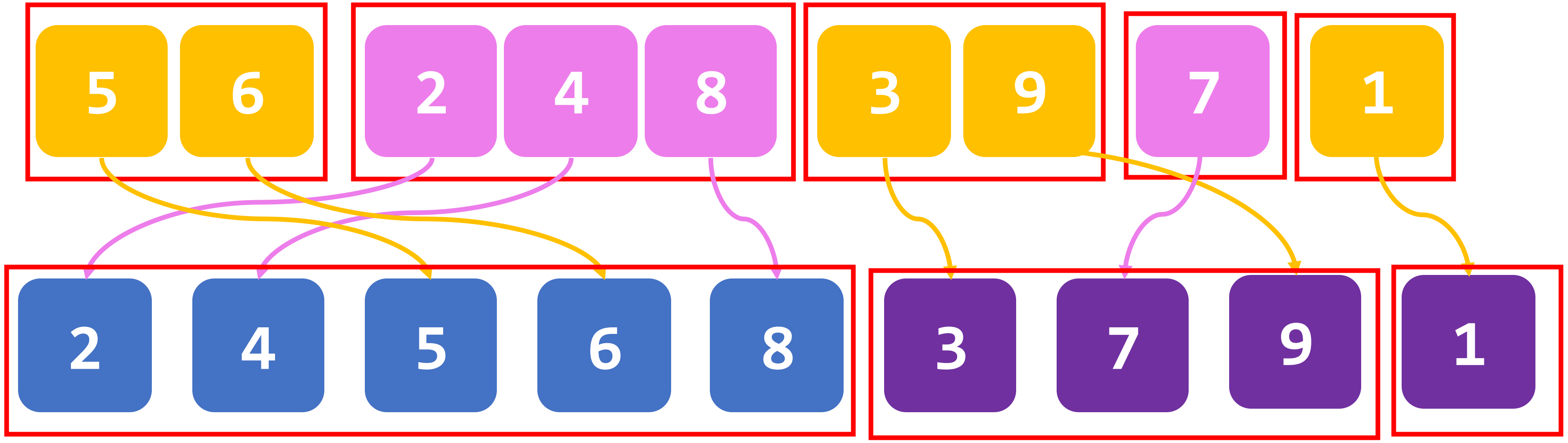
Der **natürliche 2-Wege-Merge Sort** ist eine Variation des klassischen Merge Sort, die vorhandene **aufsteigende Teilfolgen (Runs)** im Array erkennt und nutzt.

Anstatt das Array starr zu halbieren, werden jeweils **zwei benachbarte Runs gemerged**, bis das ganze Array sortiert ist.
Besonders effizient bei **fast sortierten Daten**, da weniger Merges und Durchläufe nötig sind.

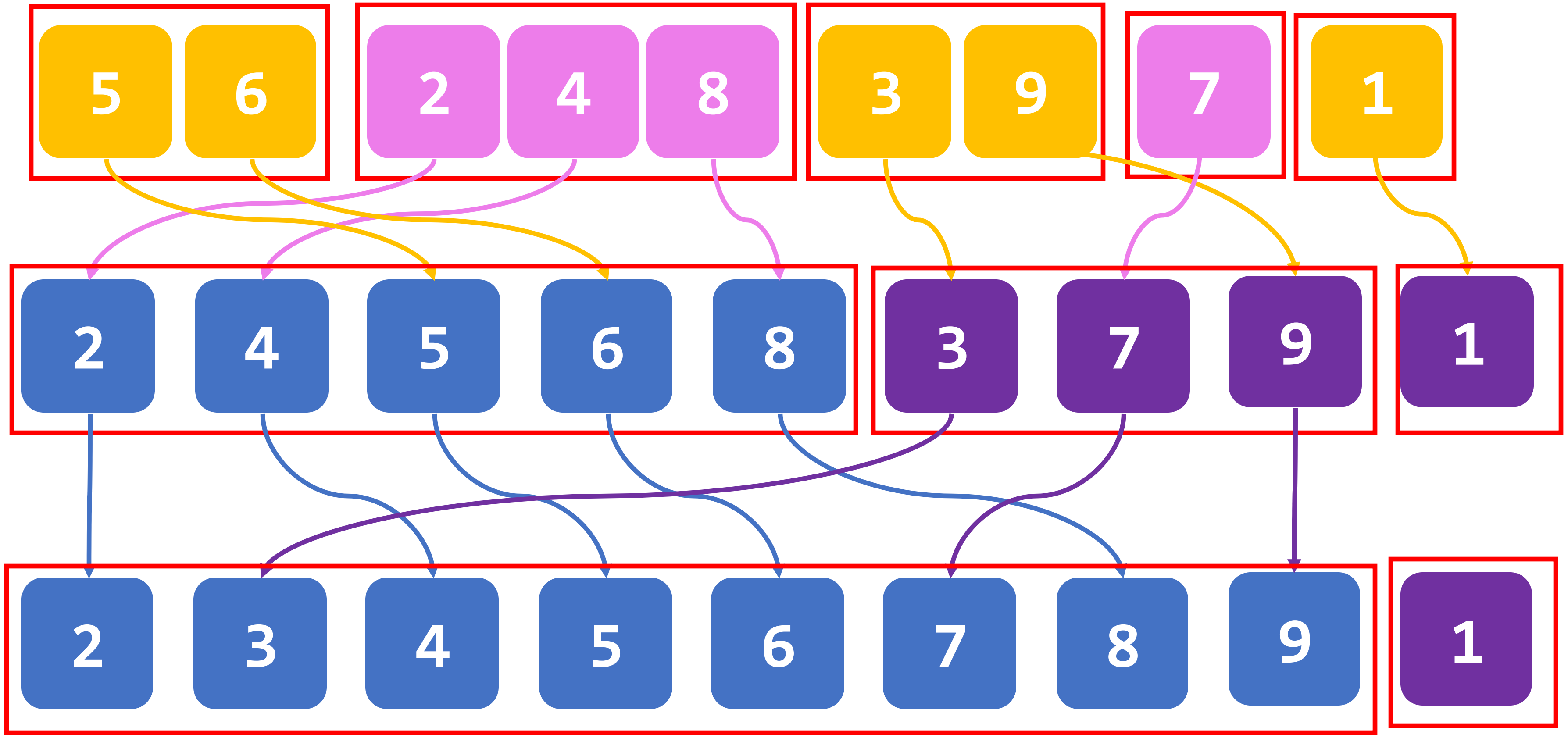
Natürlicher 2-Wege Mergesort



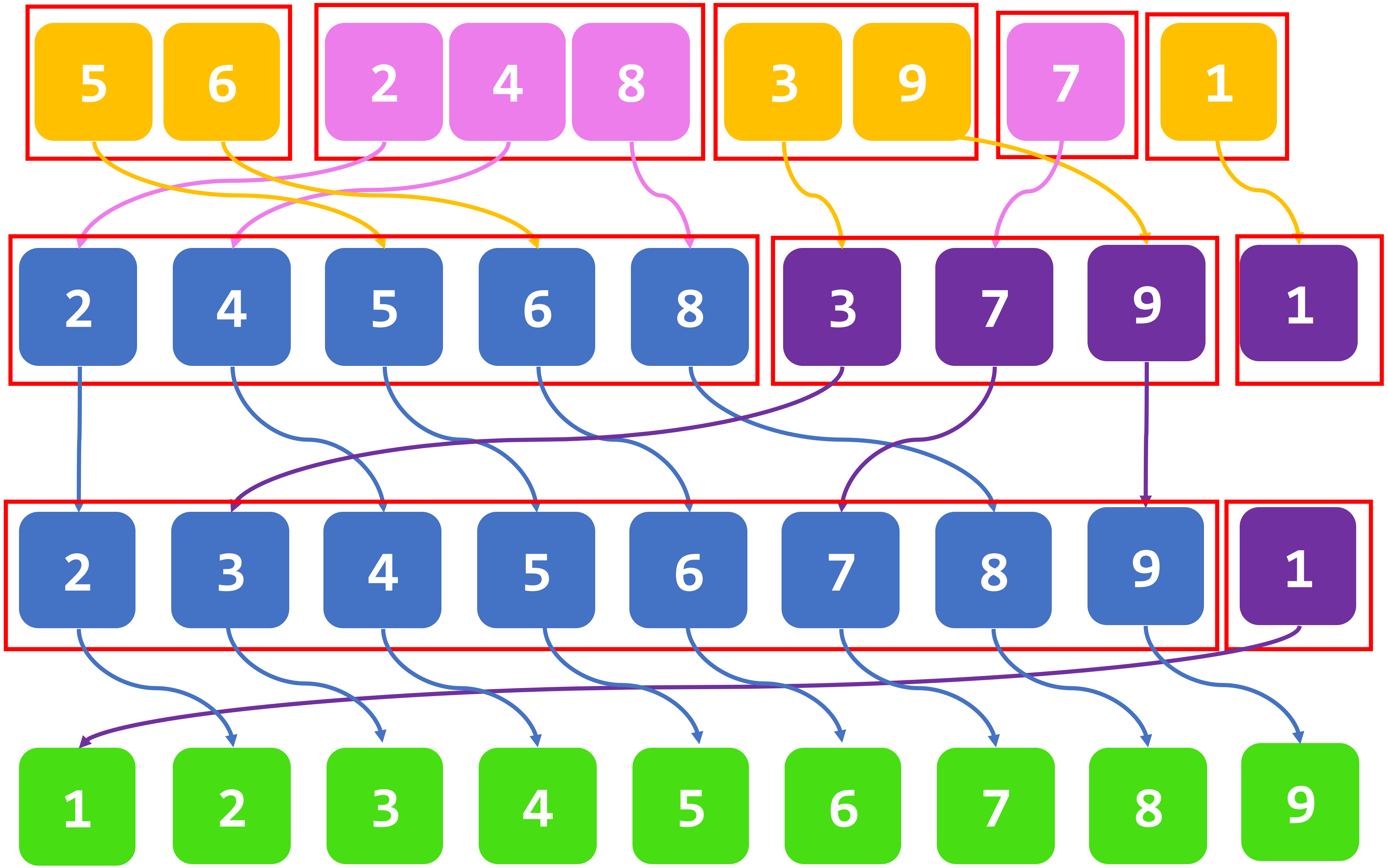
Natürlicher 2-Wege Mergesort



Natürlicher 2-Wege Mergesort



Natürlicher 2-Wege Mergesort



Unterschiede

Mergesort

- Führt Mergesort auf unabhängig von der Eingabe
- Zerteilt das Array einfach in der Mitte (respektiv so nah wie möglich)

Natural Mergesort

- Passt sich der Eingabefolge an
- Zerteilt das Array an den Stellen, wo es nicht mehr aufsteigend ist (Runs)

Algorithmus NaturalMergesort(A)

Input: Array A der Länge n > 0, **Output**: Array A sortiert

```
def NaturalMergesort(A):
    l = -1
    while l != 0:
        r=0
        while r < len(A)-1:
            l=r
            m=l
            while m < len(A)-1 and A[m+1] >= A[m]:
                m = m+1
            if m < len(A)-1:
                r = m+1
                while r < len(A)-1 and A[r+1] >= A[r]:
                    r = r+1
                Merge(A,l,m,r)
            else:
                r = len(A)-1
            r+=1
```

4. In-Class Exercise

In-class Exercise: Timsort

CodeExpert in-class task:

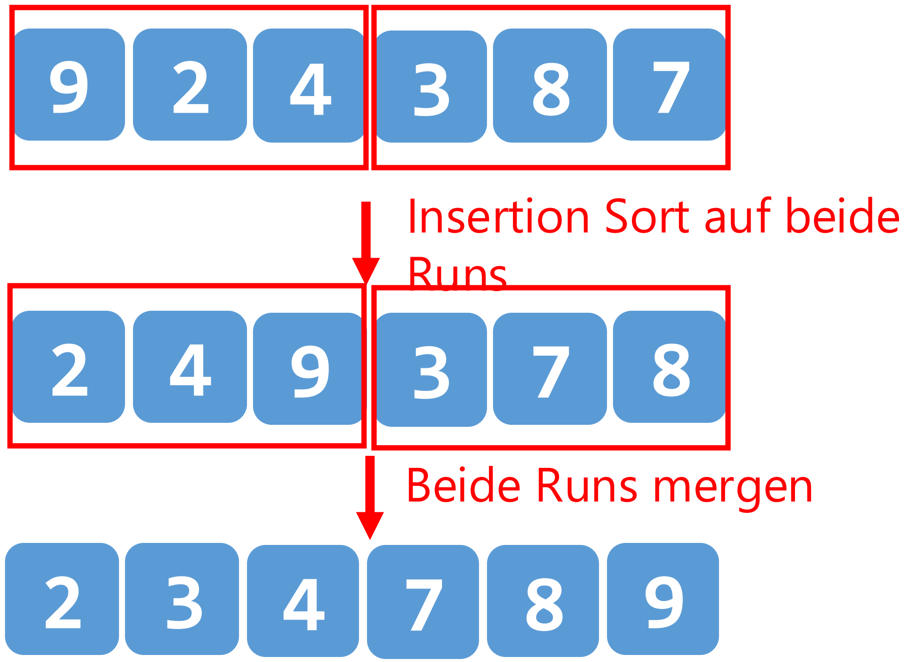
https://expert.ethz.ch/enrolled/SS25/mavt2/codeExamples

- Timsort, eine Kombination aus Insertion Sort und Merge Sort, ist der Standard-Sortieralgorithmus in Python. Mehr dazu in der Aufgabenbeschreibung auf Code Expert.

Timsort

Idee:

- 1. Teile das Array zuerst in Runs
- 2. Sortiere die einzelnen Runs mit Insertion Sort
- 3. Merge die einzelnen Runs zu einem einzigen Array



Timsort Pseudo Code

```
def merge(arr, l, m, r):
    Teile arr in left[l..m] und
    right[m+1..r]
    i, j, k ← 0, 0, l
    Solange Elemente in beiden Listen:
        arr[k] ← kleinstes von left[i]
oder                right[j]
        k++, i++ oder j++

    Füge Rest von left ein (falls
vorhanden)
    Füge Rest von right ein (falls
vorhanden)
```

```
def timsort(arr):
    n ← len(arr)
    minRun ← calcMinRun(n)

    Für jedes Teilstück der Länge minRun:
        Sortiere mit insertionSort()

    size ← minRun
    while size < n:
        Für alle Paare benachbarter Runs:
            Merge sie mit merge()
        Verdopple size
```

5. Hausaufgaben

Exercise 6: Search and Sort

Auf https://expert.ethz.ch/mycourses/SS25/mavt2/exercises

- Eierwerfen
- Vergleich von Sortieralgorithmen
- Verbesserter Insertion Sort
- Invarianten der Suchalgorithmen

Fällig bis Montag 7.04.2025, 20:00 CET

NO HARDCODING

Credits

Die Slide(-templates) stammen ursprünglich von Julian Lotzer und Daniel Steinhauser, vielen Dank!

→Checkt ihre Websites ab für zusätzliches Material in Informatik I, Informatik II und Stochastik & Machine Learning.

- https://n.ethz.ch/~jlotzer/
- https://n.ethz.ch/~dsteinhauser/