ETH zürich

Übungslektion 6 – Asymptotik & Sortierung

Informatik II

25. / 26. März 2025

Heutiges Programm

Bubblesort

Sortieren durch Einfügen

Wiederholung Theorie

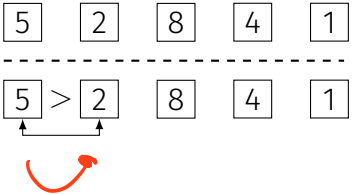
Asymptotische Laufzeit

Hausaufgaben

1. Bubblesort

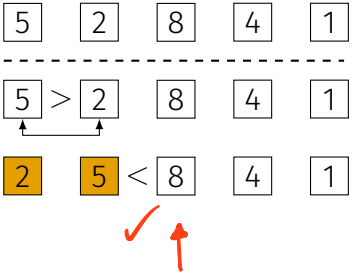
Bubblesort

Bubblesort



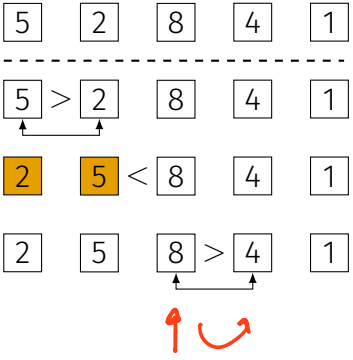
- Vergleiche jedes Paar benachbarter Elemente der Reihe nach. Wenn das erste größer ist, tausche sie aus.

Bubblesort



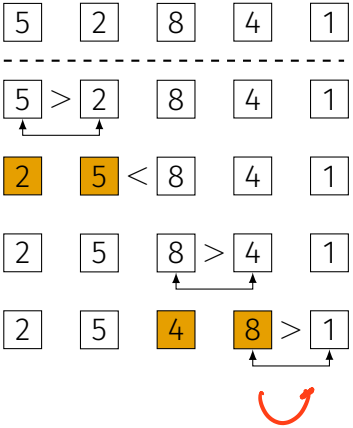
- Vergleiche jedes Paar benachbarter Elemente der Reihe nach. Wenn das erste größer ist, tausche sie aus.

Bubblesort



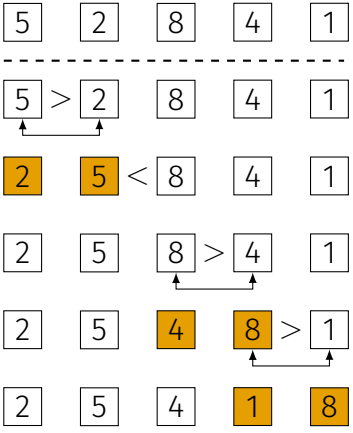
- Vergleiche jedes Paar benachbarter Elemente der Reihe nach. Wenn das erste größer ist, tausche sie aus.

Bubblesort



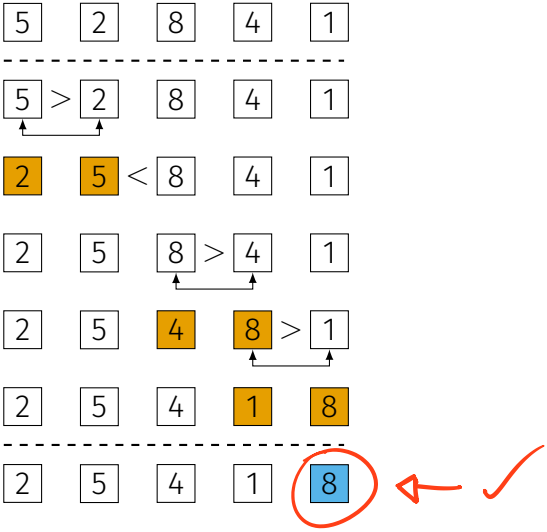
- Vergleiche jedes Paar benachbarter Elemente der Reihe nach. Wenn das erste größer ist, tausche sie aus.

Bubblesort



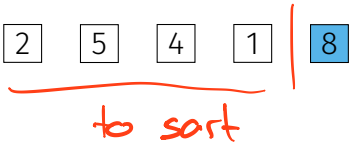
- Vergleiche jedes Paar benachbarter Elemente der Reihe nach. Wenn das erste größer ist, tausche sie aus.

Bubblesort



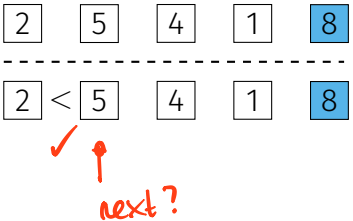
- Vergleiche jedes Paar benachbarter Elemente der Reihe nach. Wenn das erste größer ist, tausche sie aus.
- Nach einer Iteration (Iteration 0) befindet sich das größte Element am Ende der Liste.

Bubblesort



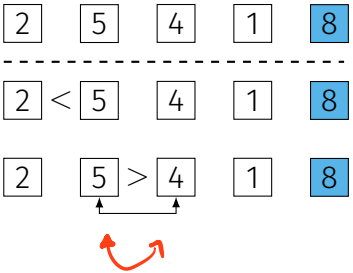
- Schleife, bis die Liste sortiert ist.

Bubblesort



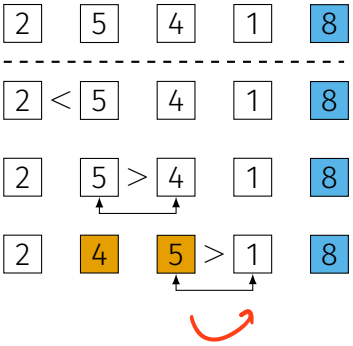
- Schleife, bis die Liste sortiert ist.

Bubblesort



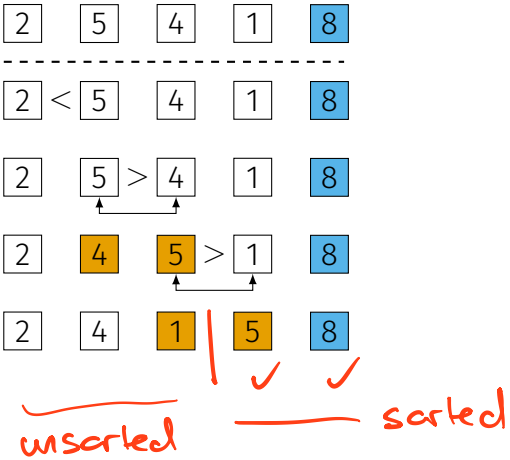
- Schleife, bis die Liste sortiert ist.

Bubblesort



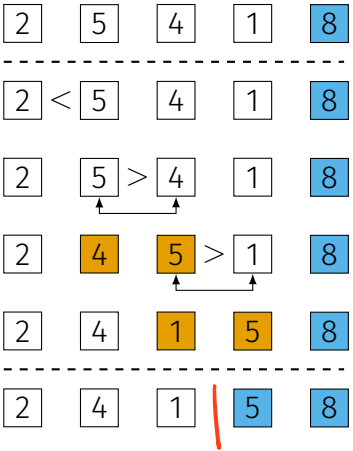
- Schleife, bis die Liste sortiert ist.

Bubblesort



- Schleife, bis die Liste sortiert ist.

Bubblesort



- Schleife, bis die Liste sortiert ist.
- **Schleifeninvariante**: Nach Iteration i sind Elemente `li[-(i+1):]` sortiert und an der richtigen Stelle.

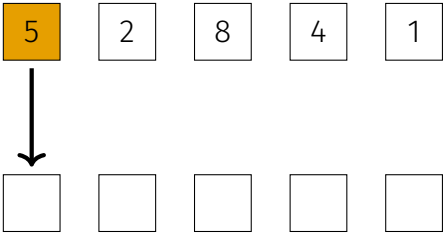
→ website Hackerearth.com

2. Sortieren durch Einfügen

Insertion Sort: Konzept

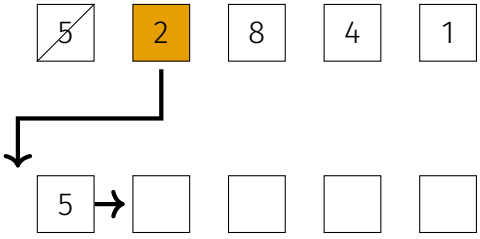
- Mit einer zweiten Liste können wir einfach jedes Element an der korrekten Stelle einsortieren.

Insertion Sort: Konzept



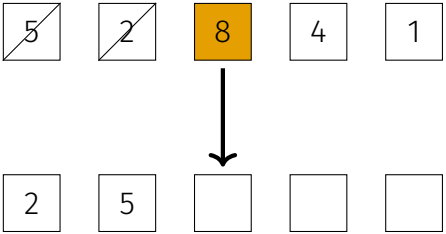
- Mit einer zweiten Liste können wir einfach jedes Element an der korrekten Stelle einsortieren.

Insertion Sort: Konzept



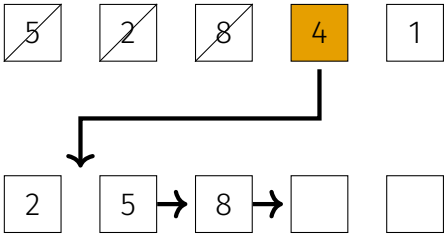
- Mit einer zweiten Liste können wir einfach jedes Element an der korrekten Stelle einsortieren.

Insertion Sort: Konzept



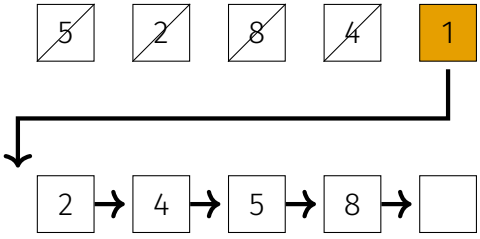
- Mit einer zweiten Liste können wir einfach jedes Element an der korrekten Stelle einsortieren.

Insertion Sort: Konzept



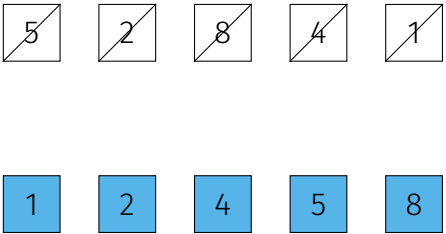
- Mit einer zweiten Liste können wir einfach jedes Element an der korrekten Stelle einsortieren.

Insertion Sort: Konzept



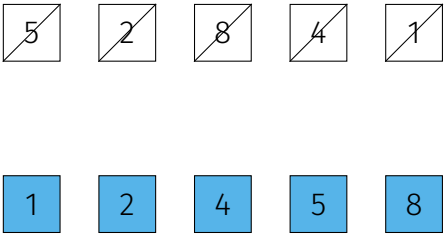
- Mit einer zweiten Liste können wir einfach jedes Element an der korrekten Stelle einsortieren.

Insertion Sort: Konzept



- Mit einer zweiten Liste können wir einfach jedes Element an der korrekten Stelle einsortieren.

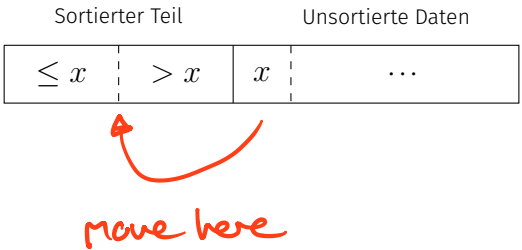
Insertion Sort: Konzept



- Mit einer zweiten Liste können wir einfach jedes Element an der korrekten Stelle einsortieren.
- **Problem**: Benötigt n zusätzlichen Speicherplatz.

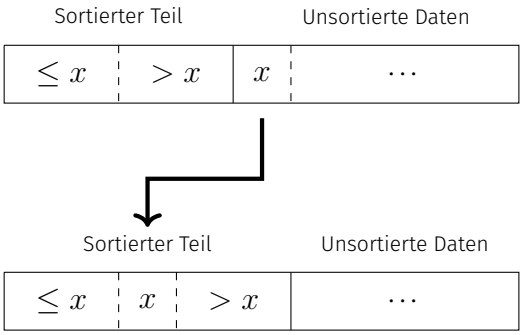
extra liste

Insertion Sort



Wir können den Speicherplatz, der in der ursprünglichen Liste frei wird verwenden.

Insertion Sort



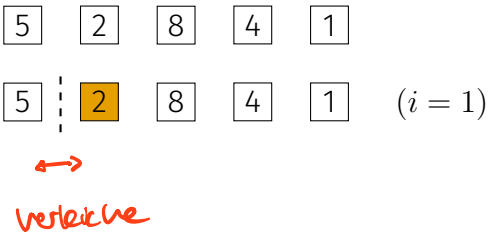
Wir können den Speicherplatz, der in der ursprünglichen Liste frei wird verwenden.

Sortieren durch Einfügen

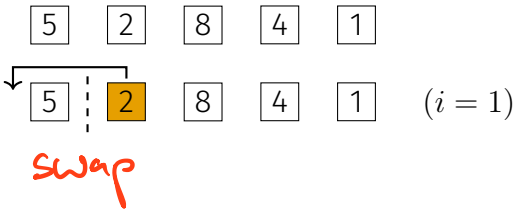
5	2	8	4	1

Sortieren durch Einfügen

- **Schleifeninvariante**: ??

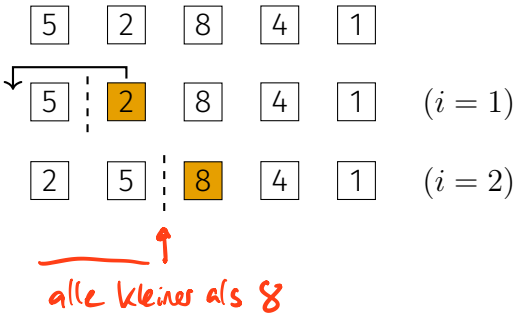


Sortieren durch Einfügen



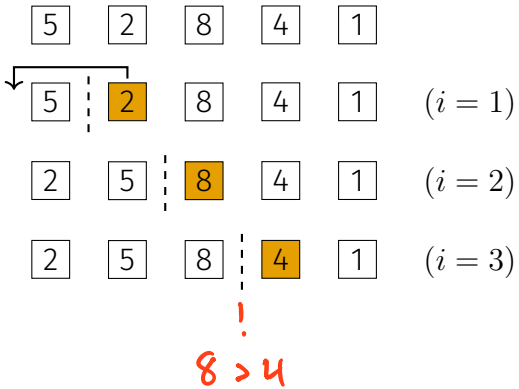
- **Schleifeninvariante**: Vor Iteration i sind Elemente in `li[:i]` sortiert. (Für Selection Sort: Vor iteration i enthält `li[:i]` die i niedrigsten Elemente von `li` in sortierter Reihenfolge.)
- Bei Iteration i, das i-te Element an der korrekten Position in die sortierte Teilliste `li[:i]` einfügen.

Sortieren durch Einfügen



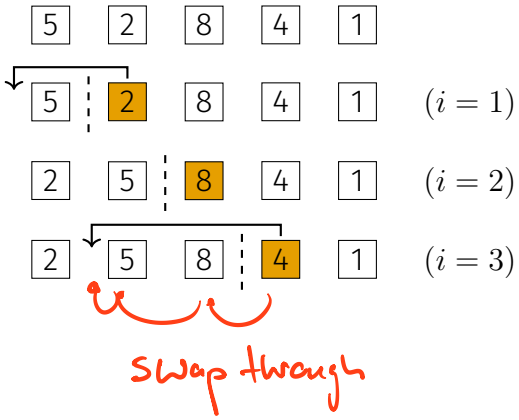
- **Schleifeninvariante**: Vor Iteration i sind Elemente in `li[:i]` sortiert. (Für Selection Sort: Vor iteration i enthält `li[:i]` die i niedrigsten Elemente von `li` in sortierter Reihenfolge.)
- Bei Iteration i, das i-te Element an der korrekten Position in die sortierte Teilliste `li[:i]` einfügen.
- Wiederholen bis alles sortiert ist ($i = n$).

Sortieren durch Einfügen



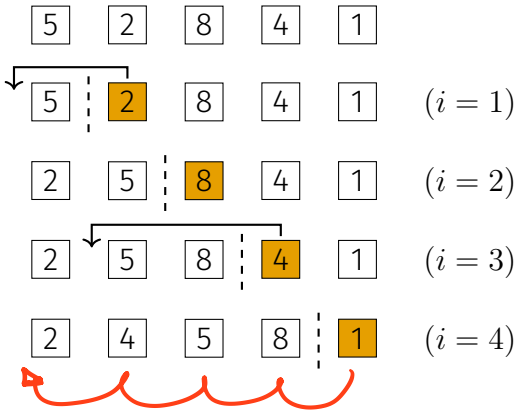
- **Schleifeninvariante**: Vor Iteration i sind Elemente in `li[:i]` sortiert. (Für Selection Sort: Vor iteration i enthält `li[:i]` die i niedrigsten Elemente von `li` in sorterter Reihenfolge.)
- Bei Iteration i, das i-te Element an der korrekten Position in die sortierte Teilliste `li[:i]` einfügen.
- Wiederholen bis alles sortiert ist ($i = n$).

Sortieren durch Einfügen



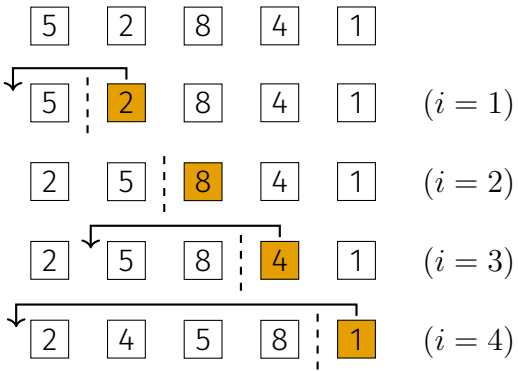
- **Schleifeninvariante**: Vor Iteration i sind Elemente in `li[:i]` sortiert. (Für Selection Sort: Vor iteration i enthält `li[:i]` die i niedrigsten Elemente von `li` in sortierter Reihenfolge.)
- Bei Iteration i, das i-te Element an der korrekten Position in die sortierte Teilliste `li[:i]` einfügen.
- Wiederholen bis alles sortiert ist ($i = n$).

Sortieren durch Einfügen



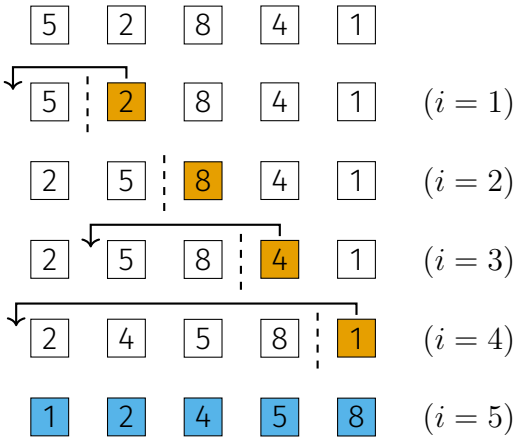
- **Schleifeninvariante**: Vor Iteration i sind Elemente in `li[:i]` sortiert. (Für Selection Sort: Vor iteration i enthält `li[:i]` die i niedrigsten Elemente von `li` in sortierter Reihenfolge.)
- Bei Iteration i, das i-te Element an der korrekten Position in die sortierte Teilliste `li[:i]` einfügen.
- Wiederholen bis alles sortiert ist $(i = n)$.

Sortieren durch Einfügen



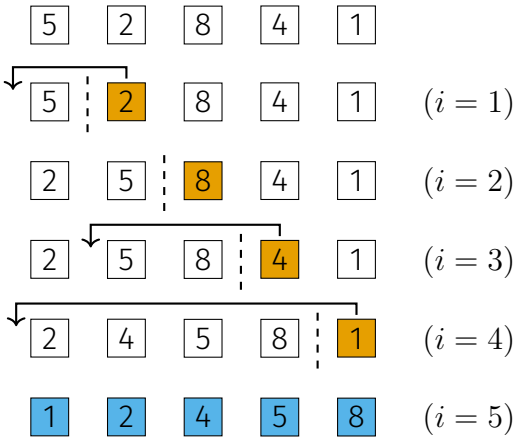
- **Schleifeninvariante**: Vor Iteration i sind Elemente in `li[:i]` sortiert. (Für Selection Sort: Vor iteration i enthält `li[:i]` die i niedrigsten Elemente von `li` in sortierter Reihenfolge.)
- Bei Iteration i, das i-te Element an der korrekten Position in die sortierte Teilliste `li[:i]` einfügen.
- Wiederholen bis alles sortiert ist $(i = n)$.

Sortieren durch Einfügen



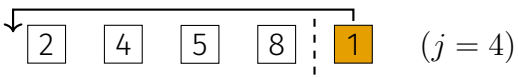
- **Schleifeninvariante**: Vor Iteration i sind Elemente in `li[:i]` sortiert. (Für Selection Sort: Vor iteration i enthält `li[:i]` die i niedrigsten Elemente von `li` in sortierter Reihenfolge.)
- Bei Iteration i, das i-te Element an der korrekten Position in die sortierte Teilliste `li[:i]` einfügen.
- Wiederholen bis alles sortiert ist ($i = n$).

Sortieren durch Einfügen



- **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.
- Wiederholen bis alles sortiert ist ($i = n$).
- **Frage**: Wie kann man die Insertion durchführen?

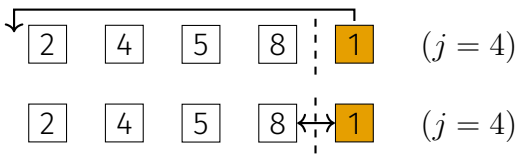
Einfügen durch Austauschen



$(j = 4)$

- Betrachten wir Iteration $i = 4$.
- Setze Variable $j = i$.

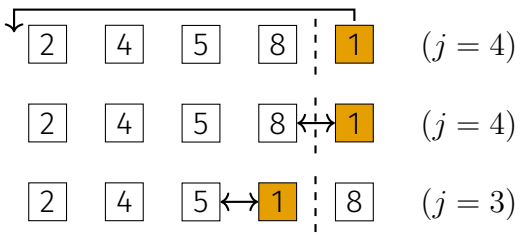
Einfügen durch Austauschen



(Webseite)

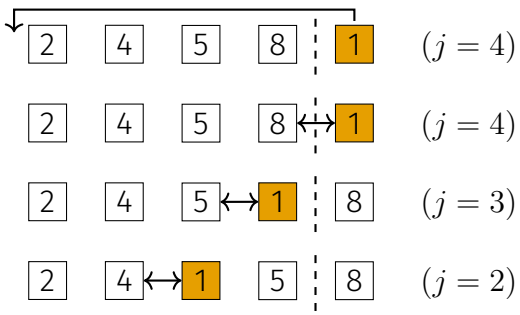
- Betrachten wir Iteration $i = 4$. letzter schritt
- Setze Variable $j = i$.
- Vergleiche `li[j]` und `li[j-1]` und tausche, falls `li[j-1]` > `li[j]`.

Einfügen durch Austauschen



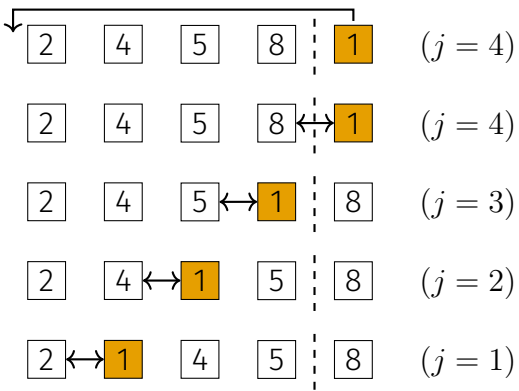
- Betrachten wir Iteration $i = 4$.
- Setze Variable $j = i$.
- Vergleiche `li[j]` und `li[j-1]` und tausche, falls `li[j-1]` > `li[j]`.
- Wiederhole bis $j = 0$ or `li[j-1]` $\leq$ `li[j]`.

Einfügen durch Austauschen



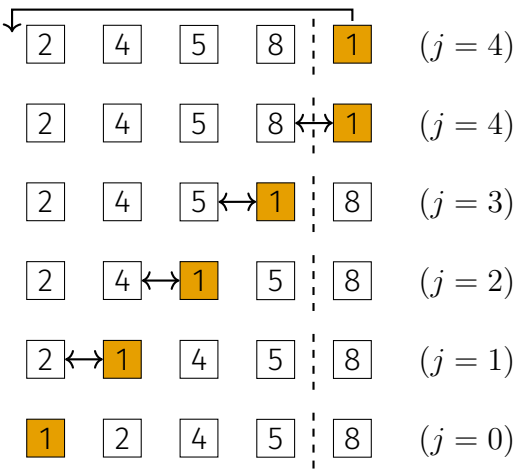
- Betrachten wir Iteration $i = 4$.
- Setze Variable $j = i$.
- Vergleiche `li[j]` und `li[j-1]` und tausche, falls `li[j-1]` > `li[j]`.
- Wiederhole bis $j = 0$ or `li[j-1]` $\leq$ `li[j]`.

Einfügen durch Austauschen



- Betrachten wir Iteration $i = 4$.
- Setze Variable $j = i$.
- Vergleiche `li[j]` und `li[j-1]` und tausche, falls `li[j-1]` > `li[j]`.
- Wiederhole bis $j = 0$ or `li[j-1]` $\leq$ `li[j]`.

Einfügen durch Austauschen



- Betrachten wir Iteration $i = 4$.
- Setze Variable $j = i$.
- Vergleiche `li[j]` und `li[j-1]` und tausche, falls `li[j-1]` > `li[j]`.
- Wiederhole bis $j = 0$ or `li[j-1]` $\leq$ `li[j]`.

code

→ show website

Implementierung von Insertion Sort

CodeExpert In-Class Aufgabe: **Insertion Sort Laufzeit**
Auf `https://expert.ethz.ch/enrolled/SS25/mavt2/codeExamples`

Kahoot

Danach werden wir folgendes diskutieren:

- Was ist das worst-case Szenario für das Sortieren einer Liste mit Insertion Sort?
- Was ist die Θ Laufzeitkomplexität von Insertion Sort in diesem Fall?
- Was ist das best-case Szenario für das Sortieren einer Liste mit Insertion Sort?
- Was ist die Θ Laufzeitkomplexität von Insertion Sort in diesem Fall?

Implementierung von Insertion Sort

CodeExpert In-Class Aufgabe: **Insertion Sort Laufzeit**
Auf `https://expert.ethz.ch/enrolled/SS25/mavt2/codeExamples`

Danach werden wir folgendes diskutieren:

- Was ist das worst-case Szenario für das Sortieren einer Liste mit Insertion Sort?
  $\Rightarrow$ Falsch herum sortiert
- Was ist die Θ Laufzeitkomplexität von Insertion Sort in diesem Fall?
  $\Rightarrow \Theta(n^2)$
- Was ist das best-case Szenario für das Sortieren einer Liste mit Insertion Sort?
  $\Rightarrow$ Bereits richtig sortiert
- Was ist die Θ Laufzeitkomplexität von Insertion Sort in diesem Fall?
  $\Rightarrow \Theta(n)$

3. Wiederholung Theorie

Asymptotisches Verhalten

- Was sind $\Omega(g(n))$, $\Theta(g(n))$, $\mathcal{O}(g(n))$?

Asymptotisches Verhalten

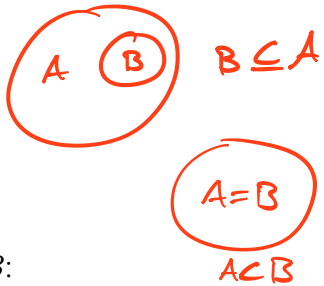
- Was sind $\Omega(g(n))$, $\Theta(g(n))$, $\mathcal{O}(g(n))$?

➜ Mengen von Funktionen!

Asymptotisches Verhalten

- Was sind $\Omega(g(n))$, $\Theta(g(n))$, $\mathcal{O}(g(n))$?

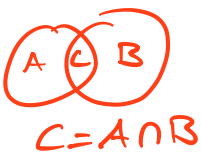
➜ Mengen von Funktionen!



Wiederholung, Mengen A, B:

Teilmenge	$A \subseteq B$
echte Teilmenge	$A \subsetneq B$
Schnittmenge	$A \cap B$

$A \neq B$



Asymptotisches Verhalten

Gegeben Funktion $f : \mathbb{N} \to \mathbb{R}$.
Definition:

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

→ show curves @ ZF

Intuition:
$f \in \mathcal{O}(g)$: (Laufzeit) f wächst asymptotisch **nicht mehr** als g. Algorithmus mit Laufzeit f ist **nicht schlechter** als einer mit g. allgemein / worst case
$f \in \Omega(g)$: (Laufzeit) f wächst asymptotisch **nicht weniger** als g. Algorithmus mit Laufzeit f ist **nicht besser** als einer mit g. best case
$f \in \Theta(g)$: f wächst asymptotisch **gleich schnell** wie g. Algorithmus mit Laufzeit f ist **gleich gut** wie einer mit g. → exact

Asymptotisches Verhalten

Einige nützliche Formeln:

ZF

- $$\sum_{i=0}^{n-1} 1 = n$$

Asymptotisches Verhalten

Einige nützliche Formeln:

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

Asymptotisches Verhalten

Einige nützliche Formeln:

- $$\sum_{i=0}^{n-1} 1 = n$$
- $$\sum_{i=0}^{n} i = \frac{n \cdot (n+1)}{2}$$
- $$\sum_{i=0}^{n} i^2 = \frac{n \cdot (n+1)(2n+1)}{6}$$

(das muss man nicht auswendig wissen)

∇ auf ZF

4. Asymptotische Laufzeit

Asymptotische Laufzeiten mit Θ

CodeExpert In-Class Aufgabe: **Nicht-rekursive Snippets Laufzeiten**

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

Asymptotische Laufzeiten mit Θ

Snippet 0. (not on CodeExpert)

```
# pre: n is an integer
def run(n):
  for m in range(0, n):
    for k in range(0, n):
      op()
```

- Wie oft wird `op()` aufgerufen?

$$\sum_{k=0}^{n} \cdot \sum_{k=0}^{n} (1) = n \cdot n = n^2$$

Asymptotische Laufzeiten mit Θ

Snippet 0. (not on CodeExpert)

```
# pre: n is an integer
def run(n):
  for m in range(0, n):
    for k in range(0, n):
      op()
```

- Wie oft wird `op()` aufgerufen?
- Antwort:

$$n \cdot n \in \Theta(n^2)$$

Asymptotische Laufzeiten mit Θ

Snippet 1.

```
# pre: n is an integer
def run(n):
  for m in range(0, n):
    for k in range(0, m):
      op()
```

n
m=n

- Wie oft wird `op()` aufgerufen?

$$\sum_{m=0}^{n}\left(\sum_{k=0}^{m} 1\right) = n^2$$

$$\sum_{m=0}^{n-1}\sum_{k=0}^{m-1}(1) = \sum_{m=0}^{n-1} \overset{2f}{=} \frac{n(n+1)}{2} \in \mathcal{O}(n^2)$$

Asymptotische Laufzeiten mit Θ

Snippet 1.

```
# pre: n is an integer
def run(n):
  for m in range(0, n):
    for k in range(0, m):
      op()
```

- Wie oft wird `op()` aufgerufen?
- Antwort:

$$\sum_{m=0}^{n-1} \sum_{k=0}^{m-1} 1$$

Asymptotische Laufzeiten mit Θ

Snippet 1.

```
# pre: n is an integer
def run(n):
  for m in range(0, n):
    for k in range(0, m):
      op()
```

- Wie oft wird `op()` aufgerufen?
- Antwort:

$$\sum_{m=0}^{n-1} \sum_{k=0}^{m-1} 1 = \sum_{m=0}^{n-1} m$$

Asymptotische Laufzeiten mit Θ

Snippet 1.

```
# pre: n is an integer
def run(n):
  for m in range(0, n):
    for k in range(0, m):
      op()
```

- Wie oft wird `op()` aufgerufen?
- Antwort:

$$\sum_{m=0}^{n-1} \sum_{k=0}^{m-1} 1 = \sum_{m=0}^{n-1} m$$
$$= \frac{(n-1) \cdot n}{2}$$

Asymptotische Laufzeiten mit Θ

Snippet 1.

```
# pre: n is an integer
def run(n):
  for m in range(0, n):
    for k in range(0, m):
      op()
```

- Wie oft wird `op()` aufgerufen?
- Antwort:

$$\sum_{m=0}^{n-1} \sum_{k=0}^{m-1} 1 = \sum_{m=0}^{n-1} m$$
$$= \frac{(n-1) \cdot n}{2} \in \Theta(n^2)$$

Asymptotische Laufzeiten mit Θ

Snippet 2.

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

→ simplify $\log_2, \log_4 \to \log$

Division gibt (meistens) Log!

- Wie oft wird `op()` aufgerufen?

Bedingung

$\frac{n}{2^{count}} \geq 1$

$n \geq 2^{count}$

$2^{count} \leq n \quad // \log$

$count \leq \log_2(n)$

Asymptotische Laufzeiten mit Θ

Snippet 2.

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

- Wie oft wird `op()` aufgerufen?
- Antwort: $\Theta(\log n)$.

Asymptotische Laufzeiten mit Θ

Snippet 2.

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

- Wie oft wird `op()` aufgerufen?
- Antwort: $\Theta(\log n)$.

- `op()` wird so lange aufgerufen, wie $n/2^{count} \geq 1$, d.h. solange $count \leq \log n$

Asymptotische Laufzeiten mit Θ

Snippet 2.

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

- Wie oft wird `op()` aufgerufen?
- Antwort: $\Theta(\log n)$.

- `op()` wird so lange aufgerufen, wie $n/2^{count} \geq 1$, d.h. solange $count \leq \log n$

- Beachten Sie, dass *count* die Anzahl der Iterationen in der Schleife verfolgt.

Asymptotische Laufzeiten mit Θ

Snippet 2.

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

- Wie oft wird `op()` aufgerufen?
- Antwort: $\Theta(\log n)$.

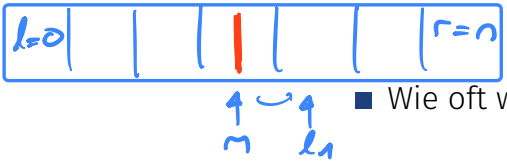
- `op()` wird so lange aufgerufen, wie $n/2^{count} \geq 1$, d.h. solange $count \leq \log n$

- Beachten Sie, dass *count* die Anzahl der Iterationen in der Schleife verfolgt.

- Daher wird `op()` zwischen $\lfloor \log n \rfloor$ und $\lceil \log n \rceil$ mal aufgerufen.

Asymptotische Laufzeiten mit Θ

Snippet 3.



- Wie oft wird `op()` aufgerufen?

```
def run(n):
  l = 0
  r = n
  while l < r:
    op()
    m = (l + r) // 2
    if h(m):
      l = m + 1
    else:
      r = m - 1

# n is an integer
# h returns a bool
```

jede n=1

r - l = länge = n

jede iteration halbiert intervall → l = m+1, r = m-1

$n \to \frac{n}{2} \to \frac{n}{4} \dots \frac{n}{2^{count}} = 1$ (länge intervall am Ende)

$n = 2^{count} \to count = \log_2(n)$

$\Rightarrow O(\log n)$

Asymptotische Laufzeiten mit Θ

Snippet 3.

```
def run(n):
  l = 0
  r = n
  while l < r:
    op()
    m = (l + r) // 2
    if h(m):
      l = m + 1
    else:
      r = m - 1

# n is an integer
# h returns a bool
```

- Wie oft wird `op()` aufgerufen?
- Antwort: $\Theta(\log n)$.

Asymptotische Laufzeiten mit Θ

Snippet 3.

```
def run(n):
  l = 0
  r = n
  while l < r:
    op()
    m = (l + r) // 2
    if h(m):
      l = m + 1
    else:
      r = m - 1

# n is an integer
# h returns a bool
```

- Wie oft wird `op()` aufgerufen?
- Antwort: $\Theta(\log n)$.

- Ähnlich wie bei Snippet 2, wenn man den Wert von $r - l$ betrachtet.

Asymptotische Laufzeiten mit Θ

Snippet 3.

```
def run(n):
  l = 0
  r = n
  while l < r:
    op()
    m = (l + r) // 2
    if h(m):
      l = m + 1
    else:
      r = m - 1

# n is an integer
# h returns a bool
```

- Wie oft wird `op()` aufgerufen?
- Antwort: $\Theta(\log n)$.
- Ähnlich wie bei Snippet 2, wenn man den Wert von $r - l$ betrachtet.
- Am Anfang haben wir $r - l = n$.

Asymptotische Laufzeiten mit Θ

Snippet 3.

```
def run(n):
  l = 0
  r = n
  while l < r:
    op()
    m = (l + r) // 2
    if h(m):
      l = m + 1
    else:
      r = m - 1

# n is an integer
# h returns a bool
```

- Wie oft wird `op()` aufgerufen?
- Antwort: $\Theta(\log n)$.

- Ähnlich wie bei Snippet 2, wenn man den Wert von $r - l$ betrachtet.
- Am Anfang haben wir $r - l = n$.
- Bei jeder Iteration wird l oder r auf den Mittelwert $m = (l + r)/2$ gesetzt.
- Bei der Iteration c gilt also immer, dass $n/2^c - 1 \leq r - l < n/2^c$.

Asymptotische Laufzeiten mit Θ

- Wie oft wird `op()` aufgerufen?

Snippet 4.

```
# pre: n is an integer
def run(n):
  for m in range(0, n):
    for k in range(0, m*m):
      op()
```

$m = n$ m^2

n^3

$$\sum_{n=0}^{n-1} \cdot \sum_{k=0}^{m^2-1} 1 = \sum_{m=0}^{n-1} m^2$$

- $\sum_{i=0}^{n} i^2 = \frac{n \cdot (n+1)(2n+1)}{6} = \Theta(n^3)$

$1^2 + 2^2 \dots n^2 = \frac{n(n+1)(2n+1)}{6} = O(n^3)$

Asymptotische Laufzeiten mit Θ

Snippet 4.

```
# pre: n is an integer
def run(n):
  for m in range(0, n):
    for k in range(0, m*m):
      op()
```

- Wie oft wird `op()` aufgerufen?
- Antwort:

$$\sum_{m=0}^{n-1} \sum_{k=0}^{m^2-1} 1$$

Asymptotische Laufzeiten mit Θ

Snippet 4.

```
# pre: n is an integer
def run(n):
  for m in range(0, n):
    for k in range(0, m*m):
      op()
```

- Wie oft wird `op()` aufgerufen?
- Antwort:

$$\sum_{m=0}^{n-1} \sum_{k=0}^{m^2-1} 1 = \sum_{m=0}^{n-1} m^2$$

Asymptotische Laufzeiten mit Θ

Snippet 4.

```
# pre: n is an integer
def run(n):
  for m in range(0, n):
    for k in range(0, m*m):
      op()
```

- Wie oft wird `op()` aufgerufen?
- Antwort:

$$\sum_{m=0}^{n-1} \sum_{k=0}^{m^2-1} 1 = \sum_{m=0}^{n-1} m^2 = \frac{(n-1) \cdot n \cdot (2n-1)}{6}$$

Asymptotische Laufzeiten mit Θ

Snippet 4.

```
# pre: n is an integer
def run(n):
  for m in range(0, n):
    for k in range(0, m*m):
      op()
```

- Wie oft wird `op()` aufgerufen?
- Antwort:

$$\begin{aligned}\sum_{m=0}^{n-1}\sum_{k=0}^{m^2-1} 1 &= \sum_{m=0}^{n-1} m^2 \\ &= \frac{(n-1)\cdot n\cdot(2n-1)}{6} \\ &\in \Theta(n^3)\end{aligned}$$

5. Hausaufgaben

Übung 5: Algorithmen und Effizienz

Auf `https://expert.ethz.ch/enrolled/SS25/mavt2/exercises`

Übung 5: Algorithmen und Effizienz

- Asymptotische Laufzeit
- Längstes gemeinsames Präfix
- Dutch Flag (Die niederländische Flagge)
- k-kleinstes Element

Abgabedatum: Montag 31.03.2024, 20:00 CET

KEINE HARDCODIERUNG

Fragen?

6. Herleitungen

Einige Formeln mit Herleitung

Summen

$$\sum_{i=0}^{n} i = ?$$

Summen

$$\sum_{i=0}^{n} i = \frac{n \cdot (n+1)}{2}$$

Summen

$$\sum_{i=0}^{n} i = \frac{n \cdot (n+1)}{2}$$

Warum?

Summen

$$\sum_{i=0}^{n} i = \frac{n \cdot (n+1)}{2}$$

Warum?

Intuition

$$1 + ... + 100 = (1 + 100) + (2 + 99) + (3 + 98) + ... + (50 + 51)$$

Summen

$$\sum_{i=0}^{n} i = \frac{n \cdot (n+1)}{2}$$

Warum?

Intuition

$$1 + ... + 100 = (1 + 100) + (2 + 99) + (3 + 98) + ... + (50 + 51)$$

Formaler?

Summen

$$\sum_{i=0}^{n}(n-i) = ?$$

Summen

$$\sum_{i=0}^{n}(n-i) = \sum_{i=0}^{n} i$$

Summen

$$\sum_{i=0}^{n}(n-i) = \sum_{i=0}^{n} i$$

$$\begin{aligned}\Rightarrow 2 \cdot \sum_{i=0}^{n} i &= \sum_{i=0}^{n} i + \sum_{i=0}^{n}(n-i) \\ &= \sum_{i=0}^{n}(i + (n-i)) = \sum_{i=0}^{n} n = (n+1) \cdot n\end{aligned}$$

Summen

$$\sum_{i=0}^{n}(n-i) = \sum_{i=0}^{n} i$$

$$\begin{aligned}\Rightarrow 2 \cdot \sum_{i=0}^{n} i &= \sum_{i=0}^{n} i + \sum_{i=0}^{n}(n-i) \\ &= \sum_{i=0}^{n}(i + (n-i)) = \sum_{i=0}^{n} n = (n+1) \cdot n\end{aligned}$$

Summen

$$\sum_{i=0}^{n} i^2 = ?$$

Summen

$$\sum_{i=0}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

Summen

$$\sum_{i=0}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

Das muss man nicht auswendig wissen. Aber man sollte wissen, dass es ein Polynom dritten Grades in n ist

Summen

Wie kommt man darauf?

Summen

Wie kommt man darauf? Interessanter Trick: Einerseits

$$\sum_{i=0}^{n} i^3 - \sum_{i=1}^{n} (i-1)^3 = \sum_{i=0}^{n} i^3 - \sum_{i=0}^{n-1} i^3 = n^3,$$

Summen

Wie kommt man darauf? Interessanter Trick: Einerseits

$$\sum_{i=0}^{n} i^3 - \sum_{i=1}^{n} (i-1)^3 = \sum_{i=0}^{n} i^3 - \sum_{i=0}^{n-1} i^3 = n^3,$$

andererseits

$$\begin{aligned}\sum_{i=0}^{n} i^3 - \sum_{i=1}^{n} (i-1)^3 &= \sum_{i=1}^{n} i^3 - \sum_{i=1}^{n} (i-1)^3 \\ &= \sum_{i=1}^{n} i^3 - (i-1)^3 = \sum_{i=1}^{n} 3 \cdot i^2 - 3 \cdot i + 1\end{aligned}$$