Sorting

In der Vorlesung geht es hauptsächlich um comparison-based sorting, welches eine theoretische Grenze hat

↳ https://stackoverflow.com/questions/4973045/generic-and-practical-sorting-algorithm-faster-than-on-log-n

↳ https://stackoverflow.com/questions/2352313/is-there-an-on-integer-sorting-algorithm

Übersicht:

(müsst ihr nicht alle für die Prüfung wissen, mehr einfach damit ihr mal viele Sorts gesehen habt.)

https://www.interviewkickstart.com/learn/time-complexities-of-all-sorting-algorithms

https://www.geeksforgeeks.org/time-complexities-of-all-sorting-algorithms/

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Selection Sort	Ω(n^2)	θ(n^2)	O(n^2)	O(1)
Bubble Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Heap Sort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(1)
Quick Sort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(n)
Merge Sort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(n)
Bucket Sort	Ω(n +k)	θ(n +k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	θ(nk)	O(nk)	O(n + k)
Count Sort	Ω(n +k)	θ(n +k)	O(n +k)	O(k)
Shell Sort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(1)
Tim Sort	Ω(n)	θ(n log(n))	O(n log (n))	O(n)
Tree Sort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(n)
Cube Sort	Ω(n)	θ(n log(n))	O(n log(n))	O(n)

← Note: Das hier ist die zusätzlich benötigte Menge an Speicherplatz.

↳ additional, auxiliary space req.

z.B. hier benötigen wir 1 Stelle mehr für den Swap.

⇒ iterative und rekursive (Merge-Sort) Sorts

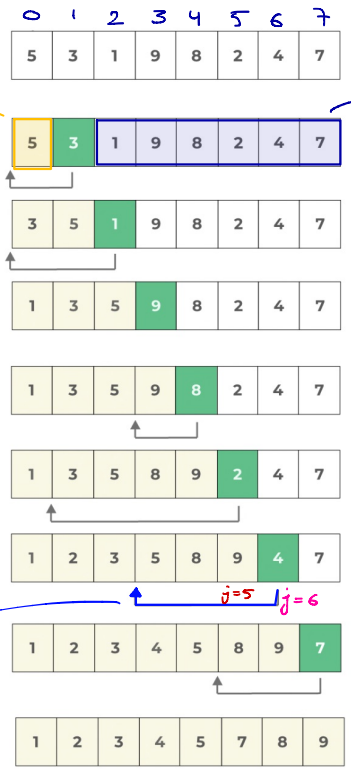
↳ for loop-based ≙ iterative Algos.

Ein Hilfsmittel, um zu beweisen, dass ein Algorithmus, der auf eine Schleife basiert ist, korrekt ist.
Sie muss drei Dinge erfüllen:

- **Initialisierung:** Sie ist vor der Iteration der Schleif wahr.
- **Fortsetzung:** Wenn sie vor der Iteration eine Schleife wahr ist, dann bleibt sie auch bis zum Beginn der nächsten Iteration wahr.
- **Terminierung:** Wenn die Schleife abbricht, dann liefert uns die Invariante eine Eigenschaft, die uns hilft zu zeigen, dass der Algorithmus korrekt ist.

→ Insertion sort auf array A

- **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.)



dieser Teil, A[0:i] ist immer sortiert (→ Invariante)

der noch zu sortierende Teil

iteration "i = 0"

i = 1 → Element A[i]

i = 2

i = 3

i = 4

i = 5

i = 6

i = 7

(alles sortiert)

das Einfügen passiert via swap / Austausch

selection sort:
https://www.geeksforgeeks.org/selection-sort/

e.g. → 9 und 4 vergleichen, swap if 9 > 4 (j = 6)

→ 8 und 4 vergleichen, swap if 8 > 4 (j = 5)

⋮

bis A[j-1] nicht mehr grösser als A[j]
d.h. solange A[j-1] > A[j]

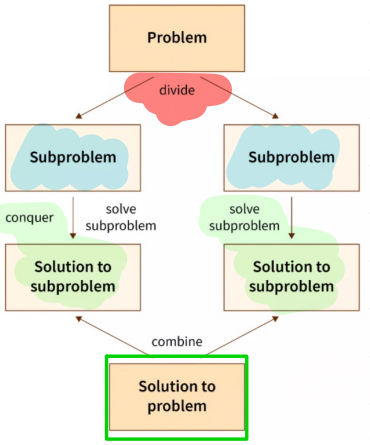
→ Merge Sort.

https://www.geeksforgeeks.org/merge-sort/

(die Variante hier ist die sog. top-down / rekursive Merge Sort Version)

https://www.baeldung.com/cs/merge-sort-top-down-vs-bottom-up

→ generelles Schema:



↳ nützliche links

https://en.wikipedia.org/wiki/Merge_sort → allg.

https://www.geeksforgeeks.org/merge-sort/ → allg.

https://www.geeksforgeeks.org/python-program-for-merge-sort/ → guter Python code

→

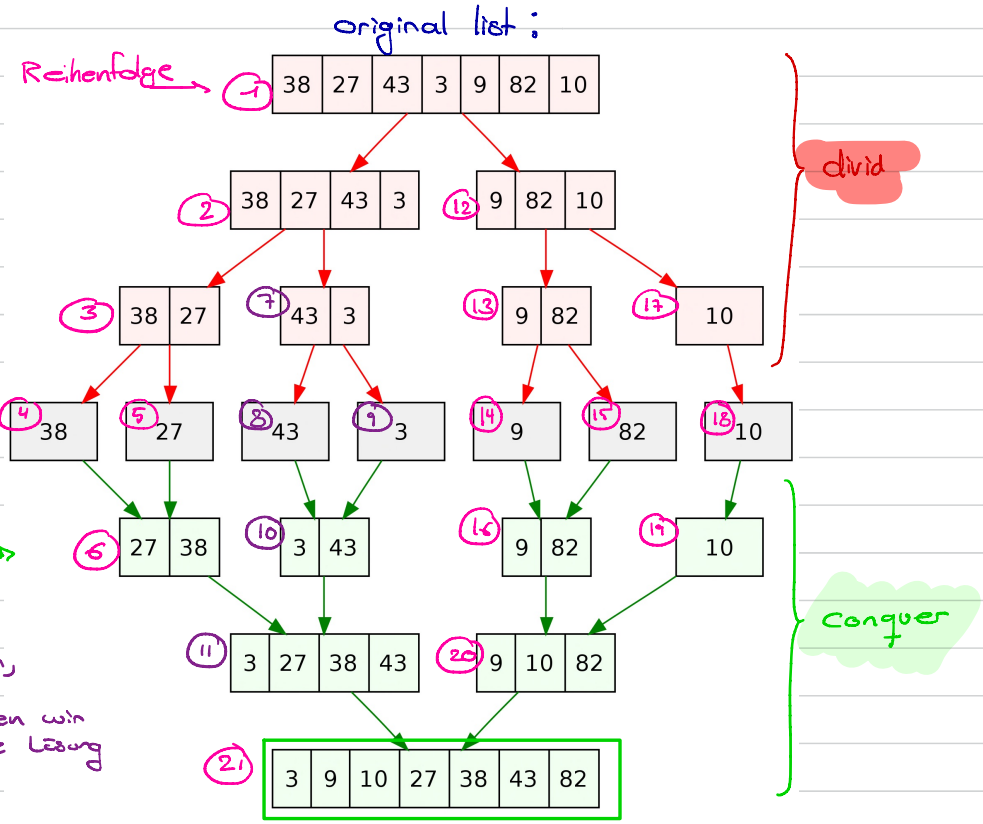
```
def mergeSort(arr, l, r):
    if l < r:
        m = l+(r-l)//2

        mergeSort(arr, l, m)
        mergeSort(arr, m+1, r)
        merge(arr, l, m, r)
```

→ vereinfacht

mergeSort (beide Hälften)
#Problemgrösse verkleinern
mergeSort (erste Hälfte)
mergeSort (zweite Hälfte)
merge (beide Hälften)

original list:



hier haben wir die Problemgrösse maximal verkleinert

d.h. wir können nun das erste Mal ein Subproblem lösen

⇒ weiter lösen können wir im Moment nicht, da uns für das weitere zusammensetzen, die zweite Hälfte fehlt. wenn wir die haben, können wir die erste nächst grössere Lösung finden.

run time analysis / time and space complexity

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

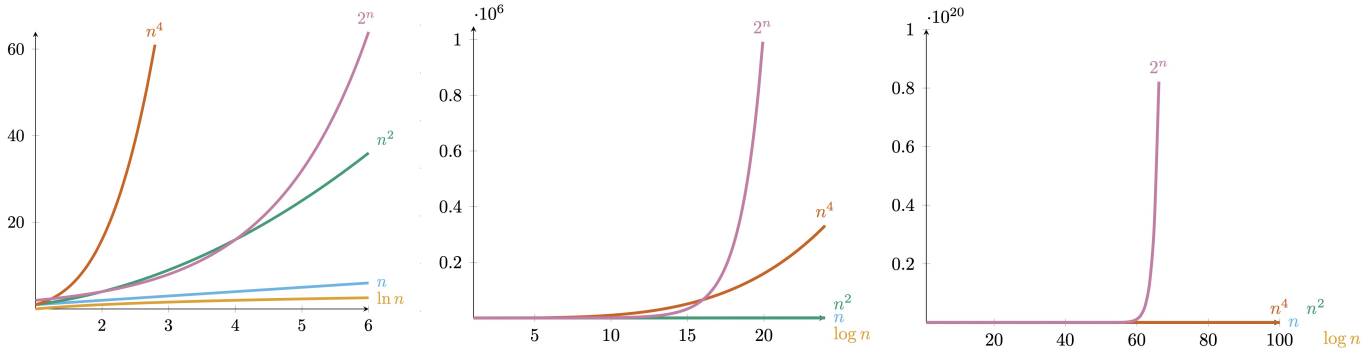
$$\mathcal{O}(g) = \{f : \mathbb{N} \to \mathbb{R} | \exists c > 0, n_0 \in \mathbb{N} | \forall n \geq n_0 : 0 \leq f(n) \leq c \cdot g(n)\}$$
$$\Omega(g) = \{f : \mathbb{N} \to \mathbb{R} | \exists c > 0, n_0 \in \mathbb{N} | \forall n \geq n_0 : 0 \leq c \cdot g(n) \leq f(n)\}$$
$$\Theta(g) = \mathcal{O}(g)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

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.
$f \in \Omega(g)$: (Laufzeit) f wächst asymptotisch **nicht weniger** als g. Algorithmus mit Laufzeit f ist **nicht besser** als einer mit g.
$f \in \Theta(g)$: f wächst asymptotisch **gleich schnell** wie g. Algorithmus mit Laufzeit f ist **gleich gut** wie einer mit g.

- **Obere Schranke:** $\mathcal{O}(n^2)$
  Die Laufzeit verhält sich **höchstens** wie $n \mapsto n^2$.
  verdoppelte Problemgrösse ⇒ Laufzeit **höchstens** vervierfacht
- **Untere Schranke:** $\Omega(n^2)$
  Die Laufzeit verhält sich **mindestens** wie $n \mapsto n^2$.
  verdoppelte Problemgrösse ⇒ Laufzeit **mindestens** vervierfacht
- **Genau:** $\Theta(n^2)$
  Die Laufzeit verhält sich wie $n \mapsto n^2$.
  verdoppelte Problemgrösse ⇒ Laufzeit vervierfacht

→ Vorsicht: es geht wirklich um das Verhalten für "grosse" n (d.h. ab irgendeinem beliebig grossen n_0)

https://www.geeksforgeeks.org/difference-between-big-oh-big-omega-and-big-theta/



2^n sieht erstmal "nicht viel grösser" aus wächst dann aber viel stärker.

(aus der Vorlesung)

Zusammenfassung: Laufzeit (1/3)

Wie berechnet man Laufzeiten?

- Definieren Sie Parameter, die die Eingabegrösse quantifizieren (dies ist in der Regel die Länge der Eingabe n).
- Grundoperationen mit grundlegenden Daten (Vergleiche, Zuweisungen, Arithmetik usw.) kosten 1.
- Eine Schleife kostet die Summe der Kosten jeder Iteration.

Zusammenfassung: Laufzeit (2/3)

- Versuchen Sie, eine vereinfachte Formel $T(n)$ abzuleiten, die die Kosten Ihres Algorithmus zusammenfasst.
- Normalerweise ist $T(n)$ eine Summe von Termen. Nehmen Sie den dominanten Term $f(n)$ ohne seinen Koeffizienten. Der dominante Term ist derjenige, für den $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$, für jeden anderen Term $g(n)$ in $T(n)$.
- Schlussfolgern Sie, dass $T(n) = \Theta(f(n))$.
- Wenn es zu schwierig ist, $T(n)$ abzuleiten, berechnen Sie zumindest eine obere Schranke $U(n)$.
- Schlussfolgern Sie, dass $T(n) = \mathcal{O}(f(n))$, wobei $f(n)$ der dominante Term von $U(n)$ ist.

Zusammenfassung: Laufzeit (3/3)

- Beachten Sie, dass $T(n) = \mathcal{O}(f(n))$ nur bedeutet, dass asymptotisch die Laufzeit Ihres Algorithmus **schliesslich um einen Faktor von $f(n)$ nach oben begrenzt ist**. $T(n) = \Theta(f(n))$ ist präziser. Es bedeutet, dass Ihr Algorithmus in Zeit proportional zu $f(n)$ ausgeführt wird.
- Wenn es Ihnen gelingt, eine untere Schranke für $T(n)$ abzuleiten, sagen wir $L(n)$, dann können Sie schliessen, dass $T(n) = \Omega(g(n))$, wobei $g(n)$ der dominierende Term von $L(n)$ ist.
- Wenn $T(n)$ sowohl $\mathcal{O}(f(n))$ als auch $\Omega(f(n))$ ist, dann ist $T(n)$ auch $\Theta(f(n))$! Dieser Trick kann manchmal nützlich sein für Algorithmen, die schwer zu analysieren sind.