Selection Sort MmF

Der Algorithmus **Selection Sort** sortiert jede Liste mit n Zahlen in n Durchläufen aufsteigend:

1) Im ersten Durchlauf durchsucht er die Liste nach der kleinsten Zahl und tauscht sie mit der ersten Zahl.
Dann ist die kleinste Zahl also *sicher* an der richtigen Stelle.

2) Im zweiten Durchlauf durchsucht er die Liste ab der zweiten Zahl nach der kleinsten Zahl und tauscht sie mit der zweiten Zahl.
Dann sind die kleinsten 2 Zahlen also *sicher* an der richtigen Stelle.

Durchlauf 1	17	11	**2**	19	3	13	7	5
Durchlauf 2	2	11	17	19	**3**	13	7	5
Durchlauf 3	2	3	17	19	11	13	7	**5**
Durchlauf 4	2	3	5	19	11	13	**7**	17
Durchlauf 5	2	3	5	7	**11**	13	19	17
Durchlauf 6	2	3	5	7	11	**13**	19	17
Durchlauf 7	2	3	5	7	11	13	19	**17**
Durchlauf 8	2	3	5	7	11	13	17	**19**
	2	3	5	7	11	13	17	19

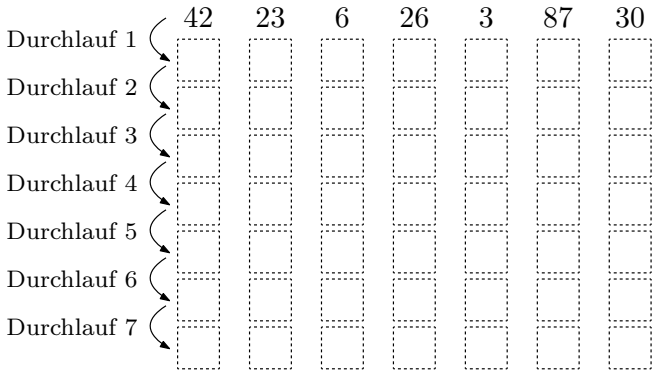
3) Im dritten Durchlauf durchsucht er die Liste ab der dritten Zahl nach der kleinsten Zahl und tauscht sie mit der dritten Zahl.
Dann sind die kleinsten 3 Zahlen also *sicher* an der richtigen Stelle.

⋮

Nach n Durchläufen ist die Liste aufsteigend sortiert.
Den letzten Durchlauf können wir auch weglassen.

Selection Sort MmF

Sortiere die Liste $(42, 23, 6, 26, 3, 87, 30)$ mit dem Sortieralgorithmus **Selection Sort** aufsteigend.



Algorithmus – Selection Sort MmF

Algorithmus: SelectionSort
Input: Liste L mit n Zahlen
Output: Liste L mit aufsteigend sortierten Zahlen

```
1:  function SelectionSort(L)
2:      for i ← 1 to n − 1 do
3:          min ← L[i]
4:          minPos ← i
5:          for j ← i + 1 to n do
6:              if L[j] < min then
7:                  min ← L[j]
8:                  minPos ← j
9:              end if
10:         end for
11:         t ← L[i]
12:         L[i] ← min
13:         L[minPos] ← t
14:     end for
15:     return L
16: end function
```

Die Zählvariable i gibt die aktuelle Durchlaufnummer an.
Den letzten Durchlauf $i = n$ lassen wir hier weg. (Zeile 2)

Im Durchlauf i durchsuchen wir $L[i]$, $L[i+1]$, $L[i+2]$, …, $L[n]$ von links nach rechts nach der kleinsten Zahl. (Zeilen 3–9)

Dafür verwenden wir zwei Hilfsvariablen:
Am Ende von Durchlauf i soll die Variable min die kleinste Zahl unter den Zahlen $L[i]$, $L[i+1]$, $L[i+2]$, …, $L[n]$ enthalten.
Die Variable $minPos$ speichert die Position („Index“) dieser kleinsten Zahl.

Zu Beginn von Durchlauf i ist $L[i]$ die kleinste gefundene Zahl. (Zeilen 3–4)
Jedes Mal, wenn wir eine kleinere Zahl finden, speichern wir diese Zahl und ihre Position ab. (Zeilen 5–9)

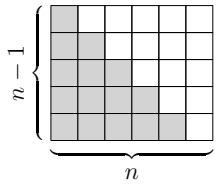
Zum Abschluss von Durchlauf i vertauschen wir die Zahlen an den Positionen i und $minPos$. Dafür ist eine Hilfsvariable zum Zwischenspeichern eines Werts notwendig. (Zeilen 11–13)

Nach den $n-1$ Durchläufen sind die Zahlen in L aufsteigend sortiert.
L wird als Output zurückgegeben. (Zeile 15)

Selection Sort – Anzahl Vergleiche

Wie viele Vergleiche benötigt der Algorithmus **Selection Sort** zum Sortieren einer Liste mit n Zahlen?

Jede Abfrage der Form „Ist $a < b$?“, „Ist $a = b$?“, „Ist $a \leq b$?“ usw. zählt als Vergleich.



Zwei sortierte Listen verschmelzen

Aktivität:	Zwei *sortierte* Listen verschmelzen
Benötigtes Material:	≈ 30 Karten, die jeweils mit einer Zahl beschriftet sind.
Anzahl Personen:	2
Vorbereitung:	Mit den Karten 2 etwa gleich große Stapel bilden, die jeweils aufsteigend sortiert sind. Die beiden Personen erhalten jeweils einen Stapel und zeigen die kleinste Zahl in ihrem Stapel her.
Aufgabenstellung:	Die beiden Personen sollen aus den sortierten Stapeln einen gemeinsamen sortierten Stapel erzeugen.

Sortierte Listen verschmelzen

Gegeben sind zwei aufsteigend sortierte Zahlenlisten L_1 und L_2.
Die beiden Listen sollen zu einer aufsteigend sortierten Gesamtliste L verschmolzen werden.
Dazu vergleichen wir immer wieder die kleinsten Zahlen der beiden Listen.
Die kleinere Zahl verschieben wir an das Ende der sortierten Gesamtliste L:

Sortierte Liste L_1				Sortierte Liste L_2					Sortierte Gesamtliste L							
2	11	17	19	**3**	5	7	13	⟶	2							
	11	17	19	**3**	5	7	13	⟶	2	3						
	11	17	19		**5**	7	13	⟶	2	3	5					
	11	17	19			**7**	13	⟶	2	3	5	7				
	11	17	19				**13**	⟶	2	3	5	7	11			
		17	19				**13**	⟶	2	3	5	7	11	13		
		17	19					⟶	2	3	5	7	11	13	17	19

Sobald L_1 oder L_2 leer ist, können wir den Rest der anderen Liste an die Gesamtliste anhängen.
Angenommen, die beiden sortierten Listen enthalten zusammen n Zahlen.
Erkläre, warum damit zum Verschmelzen der Listen weniger als n Vergleiche notwendig sind.

Merge Sort – Iterative Lösung

Aktivität:	Merge Sort „merge“ ist englisch für „verschmelzen“.
Benötigtes Material:	≈ 30 Karten, die jeweils mit einer Zahl bedruckt sind.
Anzahl Personen:	beliebig
Vorbereitung:	Jede Person erhält 1 Karte. Jede Person hat also einen sortierten „Stapel“.
Aufgabenstellung:	Das Ziel ist am Ende einen einzigen sortierten Kartenstapel zu haben. Dazu treffen sich immer wieder 2 Personen und verschmelzen wie zuvor ihre Stapel. Diese Treffen können *gleichzeitig* stattfinden, bis nur noch ein sortierter Stapel übrig ist.

Merge Sort – Anzahl Vergleiche

Der Algorithmus **Merge Sort** sortiert jede Liste mit $n = 2^k$ Zahlen in k Durchläufen:
Rechts siehst du ein Beispiel mit $k = 3$ und $n = 2^3 = 8$.

Wir starten mit 8 sortierten Listen mit jeweils einer Zahl.
Nach Durchlauf 1 enthalten die 4 sortierten Listen jeweils 2 Zahlen.
Nach Durchlauf 2 enthalten die 2 sortierten Listen jeweils 4 Zahlen.
Nach Durchlauf 3 enthält die eine sortierte Liste alle 8 Zahlen.

	17	11	2	19	3	13	7	5
Durchlauf 1	11	17	2	19	3	13	5	7
Durchlauf 2	2	11	17	19	3	5	7	13
Durchlauf 3	2	3	5	7	11	13	17	19

In jedem Durchlauf sind weniger als $n = 8$ Vergleiche notwendig. Zum Sortieren von $n = 2^k$ Zahlen benötigt **Merge Sort** also weniger als $n \cdot k = n \cdot \log_2(n)$ Vergleiche. $n = 2^k \iff k = \log_2(n)$

Bei $n = 9, 10, \ldots, 16$ Zahlen sind dann 4 Durchläufe notwendig. Allgemein benötigt Merge Sort weniger als $n \cdot \lceil \log_2(n) \rceil$ Vergleiche.

Analyse von Algorithmen

Wenn zwei Algorithmen das gleiche Problem lösen, können wir sie nach mehreren Kriterien vergleichen:

- Wie viele Operationen benötigt der Algorithmus? Wertzuweisungen ($x \leftarrow 42$) Vergleiche („Ist $a < b$?“)
- Wie viel Speicherplatz ist insgesamt notwendig?

Die Antworten auf diese Fragen können dabei von der Problemgröße abhängen.
Bei Sortieralgorithmen ist die Problemgröße die Anzahl n der Zahlen, die sortiert werden sollen.
Bei der Analyse von Algorithmen wollen wir Antworten auf diese Fragen für *großes* n finden.

Selection Sort vs. Merge Sort

Eine Liste mit n Zahlen soll sortiert werden.
Dafür benötigt der Algorithmus …

… **Selection Sort** insgesamt $\dfrac{n \cdot (n-1)}{2}$ Vergleiche.

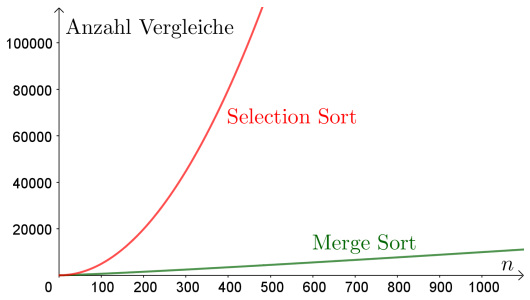
… **Merge Sort** insgesamt rund $n \cdot \log_2(n)$ Vergleiche.

Die Graphen der zugehörigen Funktionen siehst du rechts.

Bei $n = 2^{10} = 1024$ Zahlen benötigt der Algorithmus …

… **Selection Sort** insgesamt ⟨ ⟩ Vergleiche.

… **Merge Sort** insgesamt rund ⟨ ⟩ Vergleiche.



Die Analyse von Algorithmen und die Suche nach effizienten Algorithmen sind Disziplinen, die eine Schnittstelle zwischen Informatik und Mathematik bilden.

Merge Sort – Rekursive Lösung

Algorithmus: MergeSort
Input: Liste L mit n Zahlen
Output: Liste L mit aufsteigend sortierten Zahlen

```
1:  function MergeSort(L)
2:      n ← Length(L)
3:      if n < 2 then
4:          return L
5:      else
6:          L1 ← MergeSort(FirstHalf(L))
7:          L2 ← MergeSort(SecondHalf(L))
8:          return MergeSortedLists(L1, L2)
9:      end if
10: end function
```

Falls die Input-Liste L weniger als 2 Zahlen enthält, ist sie automatisch sortiert und kann sofort als Output zurückgegeben werden. (Zeilen 2–4)

Ansonsten teilen wir L in zwei Hälften FirstHalf(L) und SecondHalf(L).
Diese beiden kürzeren Listen lassen wir vom gleichen Algorithmus MergeSort sortieren. (Zeilen 6–7)
Warum das funktioniert, erfährst du gleich.

Die beiden sortierten Listen L_1 und L_2 verschmelzen wir wie zuvor zu einer sortierten Gesamtliste und geben sie als Output zurück. (Zeile 8)

Rekursive Algorithmen

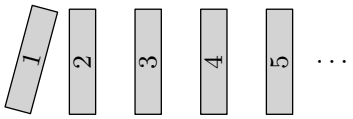
Dieser Algorithmus MergeSort löst ein Problem, indem er sich selbst aufruft.
In der Informatik spricht man dann von **rekursiven Algorithmen**. „recur“ ist englisch für „wiederkehren“.

Warum funktioniert der rekursive Merge Sort – Algorithmus?

Hinter rekursiven Algorithmen steckt das mathematische Prinzip der vollständigen Induktion.
Wir zeigen, dass MergeSort jede Liste mit $n = 1, 2, 3, 4, 5, \ldots$ Zahlen richtig sortiert.

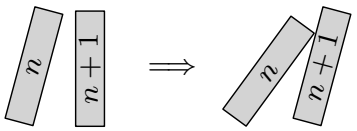
1) Induktionsanfang:
Erkläre, warum MergeSort jede Liste mit einer einzigen Zahl richtig sortiert.

Der erste Dominostein fällt um.



2) Induktionsschritt:
Du darfst verwenden, dass MergeSort jede Liste mit *höchstens* $n \geq 1$ Zahlen richtig sortiert.
Erkläre, warum MergeSort dann auch jede Liste mit $n + 1$ Zahlen richtig sortiert.

Wenn die ersten n Dominosteine umfallen, dann fällt auch der $(n+1)$-te Dominostein um.



MergeSort sortiert also jede noch so lange Liste richtig. Jeder Dominostein fällt schließlich um.

Programmablauf

Jeder Pfeil nach unten ist ein MergeSort-Aufruf.
Jeder Pfeil nach oben ist mit dem zugehörigen Output beschriftet.
In welcher Reihenfolge werden diese Schritte beim rekursiven MergeSort-Algorithmus durchgeführt?
Fülle in die Lücken die richtige Reihenfolge $1, 2, 3, 4, 5, 6, \ldots$ ein: