Algorithmen und Datenstrukturen

2. Vorlesung

Mündliche Prüfung

Laufzeitanalyse später

Viel Mathematik

Konkrete Algorithmen

Sortieren

Selection Sort (Sortieren durch Auswählen)

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.

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

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

	42	23	6	26	**3**	87	30
Durchlauf 1	3	23	**6**	26	42	87	30
Durchlauf 2	3	6	**23**	26	42	87	30
Durchlauf 3	3	6	23	**26**	42	87	30
Durchlauf 4	3	6	23	26	42	87	**30**
Durchlauf 5	3	6	23	26	30	87	**42**
Durchlauf 6	3	6	23	26	30	42	**87**
Durchlauf 7	3	6	23	26	30	42	87

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 $L[i]$, $L[i+1]$, $L[i+2]$, $\ldots$, $L[n]$ 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 $L[i]$, $L[i+1]$, $L[i+2]$, $\ldots$, $L[n]$ 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)

```python
def selection_sort(L):
    n = len(L)

    # Zeile 2: for i = 0 bis n-2
    for i in range(n - 1):
        # Zeile 3 & 4: Initialisierung
        min_val = L[i]
        min_pos = i

        # Zeile 5: Innere Schleife sucht das Minimum im Rest der Liste
        for j in range(i + 1, n):
            # Zeile 6: Vergleich
            if L[j] < min_val:
                # Zeile 7 & 8: Neues Minimum gefunden
                min_val = L[j]
                min_pos = j

        # Zeile 11 - 13: Vertauschen der Elemente mit Hilfsvariable t
        t = L[i]
        L[i] = min_val
        L[min_pos] = t

    # Zeile 15: Rückgabe der sortierten Liste
    return L
```

```
 # Zeile 11 - 13: Vertauschen der Elemente
mit Hilfsvariable t
        t = L[i]
        L[i] = min_val
        L[min_pos] = t
```

In Python kann man die Zeilen 11 bis 13 (den Tausch) auch deutlich kürzer schreiben:

```
L[i], L[min_pos] = L[min_pos], L[i]
```

Das ist die idiomatische Art in Python, Werte zu tauschen, ohne explizit eine Hilfsvariable `t` hinschreiben zu müssen.

Rekursive Variante:

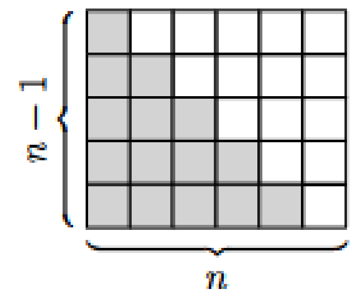
```
def recursive_selection_sort(L, i=0):
    n = len(L)

    # Basis-Fall: Wenn wir am Ende der Liste angekommen sind,
stoppen wir
    if i >= n - 1:
        return L

    # 1. Das Minimum im verbleibenden Teil suchen (entspricht
der inneren Schleife)
    min_pos = i
    for j in range(i + 1, n):
        if L[j] < L[min_pos]:
            min_pos = j

    # 2. Tauschen
    L[i], L[min_pos] = L[min_pos], L[i]

    # 3. Rekursiver Aufruf für das nächste Element (i + 1)
    return recursive_selection_sort(L, i + 1)
```

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.

In Durchlauf i finden $n - i$ Vergleiche statt.

Insgesamt werden also $1 + 2 + 3 + \cdots + (n-1) = \frac{n \cdot (n-1)}{2}$ Vergleiche benötigt.



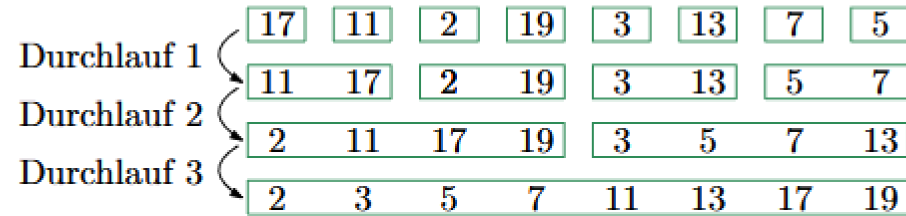
Mergesort

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	$\longrightarrow$	2							
	11	17	19	**3**	5	7	13	$\longrightarrow$	2	3						
	11	17	19		**5**	7	13	$\longrightarrow$	2	3	5					
	11	17	19			**7**	13	$\longrightarrow$	2	3	5	7				
	11	17	19				**13**	$\longrightarrow$	2	3	5	7	11			
		17	19				**13**	$\longrightarrow$	2	3	5	7	11	13		
		17	19					$\longrightarrow$	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.
In jeder Zeile findet genau ein Vergleich statt.
Spätestens nach $n-1$ Vergleichen ist eine der beiden Listen leer.

Der Algorithmus **Merge Sort** sortiert jede Liste mit $n = 2^k$ Zahlen in k Durchläufen:



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.

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

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)

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)

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.

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

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

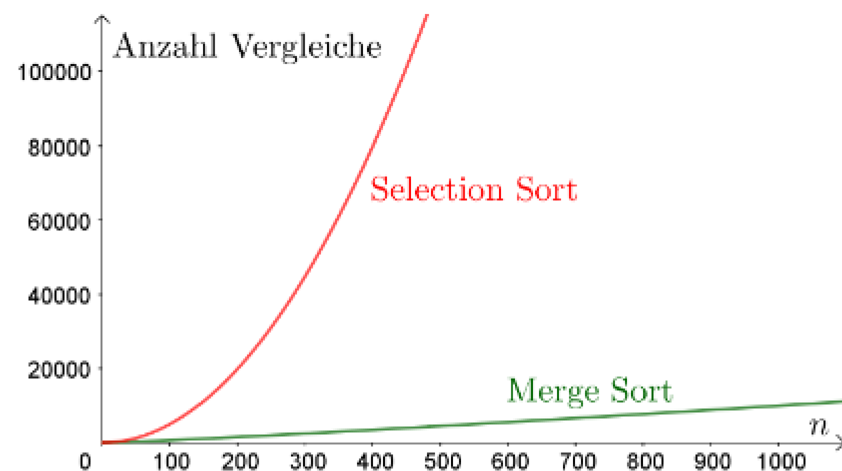
... **Merge Sort** insgesamt rund $n \cdot \log_2(n)$ Vergleiche.

Die Graphen der zugehörigen Funktionen siehst du rechts.

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

... **Selection Sort** insgesamt 523776 Vergleiche.

... **Merge Sort** insgesamt rund 10240 Vergleiche.



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

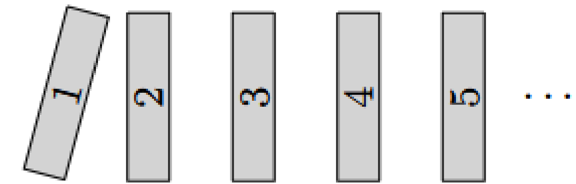
Dieser Algorithmus MERGESORT löst ein Problem, indem er sich selbst aufruft.
In der Informatik spricht man dann von **rekursiven Algorithmen**. „recur“ ist englisch für „wiederkehren“.

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:

Der erste Dominostein fällt um.

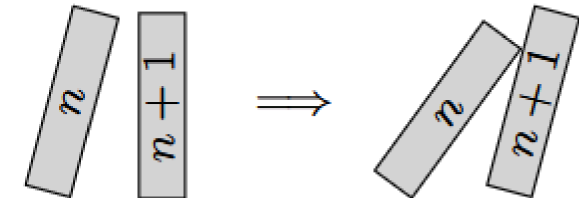
Eine Liste mit nur einer Zahl ist automatisch sortiert. In Zeile 4 bricht MERGESORT ab und gibt die Liste unverändert zurück.



2) Induktionsschritt:

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

Wenn die gesamte Liste aus $n + 1$ Zahlen besteht, dann enthalten FIRSTHALF(L) und SECONDHALF(L) jeweils höchstens n Zahlen. In Zeile 6 und 7 liefert MERGESORT also sortierte Listen L_1 und L_2. Die beiden sortierten Listen werden in Zeile 8 zu einer sortierten Liste verschmolzen und zurückgegeben.



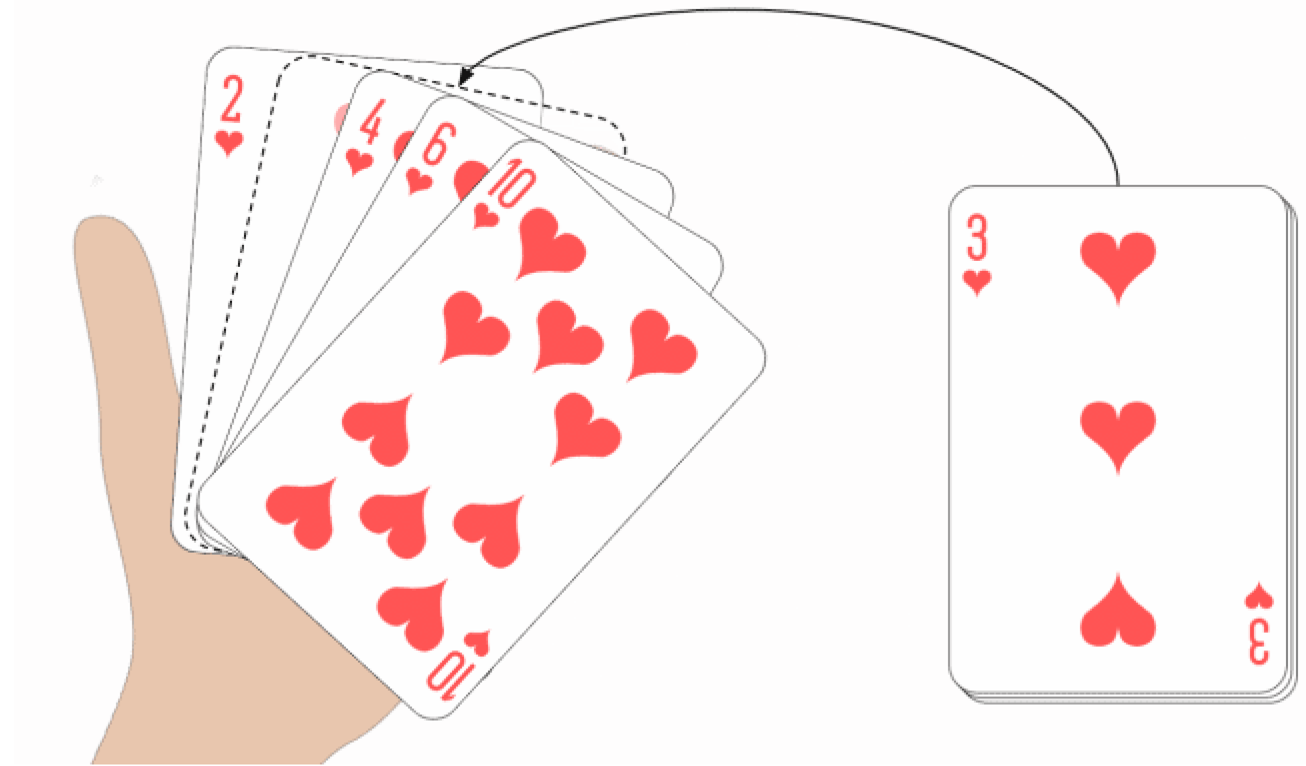
MERGESORT sortiert also jede noch so lange Liste richtig.

Jeder Dominostein fällt schließlich um.

Insertion Sort

(Sortieren durch Einfügen)

Insertion Sort



Das Vorgehen ist mit der Sortierung eines Spielkartenblatts vergleichbar. Am Anfang liegen die Karten des Blatts verdeckt auf dem Tisch. Die Karten werden nacheinander aufgedeckt und an der korrekten Position in das Blatt, das in der Hand gehalten wird, eingefügt. Um die Einfügestelle für eine neue Karte zu finden, wird entweder die Karte sukzessive (von links nach rechts) mit den bereits einsortierten Karten des Blattes verglichen, oder eine binäre Suche durchgeführt. Zu jedem Zeitpunkt sind die Karten in der Hand sortiert und bestehen aus den bereits vom Tisch entnommenen Karten. Zum Einfügen der neuen Karte müssen alle auf der Hand nachfolgenden eine Position weiter nach rechts wandern.

Insertion Sort

6 5 3 1 8 7 2 4

Zähle i von 1 bis n-1		
	einzusortierender_wert = A[i]	
	j = i	
	Solange j > 0 und A[j-1] > einzusortierender_wert	
		A[j] = A[j-1]
		j = j - 1
	A[j] = einzusortierender_wert	

```
INSERTIONSORT(A)

for i = 1 to (Länge(A)-1) do
     einzusortierender_wert = A[i]
     j = i
     while (j > 0) and (A[j-1] > einzusortierender_wert) do
          A[j] = A[j - 1]
          j = j - 1
     end while
     A[j] = einzusortierender_wert
end for
```

Bubblesort

Bubblesort

6 5 3 1 8 7 2 4

In der Bubble-Phase wird die Eingabe-Liste von links nach rechts durchlaufen. Dabei wird in jedem Schritt das aktuelle Element mit dem rechten Nachbarn verglichen. Falls die beiden Elemente das Sortierkriterium verletzen, werden sie getauscht. Am Ende der Phase steht bei auf- bzw. absteigender Sortierung das größte bzw. kleinste Element der Eingabe am Ende der Liste.
Die Bubble-Phase wird solange wiederholt, bis die Eingabeliste vollständig sortiert ist. Dabei muss das letzte Element des vorherigen Durchlaufs nicht mehr betrachtet werden, da die restliche zu sortierende Eingabe keine größeren bzw. kleineren Elemente mehr enthält.

Bubblesort

```
55 07 78 12 42    1. Durchlauf
07 55 78 12 42
07 55 78 12 42
07 55 12 78 42    Letzter Vergleich
07 55 12 42 78    2. Durchlauf
07 55 12 42 78
07 12 55 42 78    Letzter Vergleich
07 12 42 55 78    3. Durchlauf
07 12 42 55 78    Letzter Vergleich
07 12 42 55 78    4. Durchlauf + Letzter Vergleich
07 12 42 55 78    Fertig sortiert.
```

Bubblesort

Der Algorithmus in seiner einfachsten Form als Pseudocode:

```
bubbleSort(Array A)
  for (n = A.size; n > 1; n = n - 1) { // äußere Schleife
    for (i = 0; i < n - 1; i = i + 1) { // innere Schleife
      if (A[i] > A[i + 1]) {
        A.swap(i, i + 1)
      }
    }
  }
```

Allerdings nutzt diese einfachste Variante nicht die Eigenschaft aus, dass nach einer Iteration, in der keine Vertauschungen stattfanden, auch in den restlichen Iterationen keine Vertauschungen mehr stattfinden.

Bubblesort

Der folgende Pseudocode berücksichtigt dies:

```
1   bubbleSort2(Array A)
2     n = A.size
3     do { // äußere Schleife
4       swapped = false
5       for (i = 0; i < n - 1; i = i + 1) { // innere Schleife
6         if (A[i] > A[i + 1]) {
7           A.swap(i, i + 1)
8           swapped = true
9         }
10      }
11      n = n - 1
12    } while (swapped)
```