4.2 Sortieralgorithmen

4.2.1 Zwei Versuche zum Einstieg

Versuch: Zwanzig Spielkarten

Jede Gruppe erhält zwanzig gemischte Karten und bringt sie in die richtige Reihenfolge.

Notieren Sie anschließend in zwei, drei Sätzen, *wie* Sie dabei vorgegangen sind — nicht was herausgekommen ist.

Unser Vorgehen beim Sortieren der Karten:

Vergleichen Sie die Beschreibungen in der Klasse. Fast alle lassen sich einem der vier Verfahren zuordnen, die in diesem Abschnitt folgen. Sie unterscheiden sich nicht im Ergebnis — sortiert ist sortiert — sondern im Aufwand.

Versuch: Acht Zylinder und eine Balkenwaage

Vor Ihnen stehen acht 3D-gedruckte Zylinder. Sie sind gleich groß und sehen gleich aus, wurden aber mit unterschiedlicher Dichte gedruckt und sind deshalb verschieden schwer. Ordnen Sie die Zylinder vom leichtesten zum schwersten.

Dafür gilt eine Einschränkung: Das einzige erlaubte Hilfsmittel ist die Balkenwaage, und sie beantwortet immer nur *eine* Frage — welcher von genau zwei Zylindern ist der schwerere. Sie dürfen nicht abwiegen, nicht schätzen, nicht in die Hand nehmen.

Machen Sie für jede Wägung einen Strich und versuchen Sie, mit möglichst wenigen auszukommen.

Strichliste der Wägungen — insgesamt: .....

Auswertung

Der Unterschied zwischen beiden Versuchen ist der eigentliche Inhalt dieses Kapitels. Bei den Karten sehen Sie alle Werte auf einen Blick; „such die kleinste Karte" ist für Sie eine einzige Handlung. Ein Rechner kann das nicht. Er sieht das Feld nie als Ganzes, sondern kann immer nur zwei Einträge miteinander vergleichen — genau wie die Waage. Was bei den Karten unsichtbar bleibt, wird an der Waage zur zählbaren Arbeit.

Deshalb messen wir den Aufwand eines Sortierverfahrens nicht in Sekunden, sondern in Vergleichen.

Merke

Ein Sortierverfahren verändert ein Feld nur über zwei Operationen: **vergleichen** und **vertauschen**. Wenn wir im Folgenden vom Aufwand sprechen, zählen wir genau diese beiden — nicht Sekunden. Sekunden hängen vom Rechner ab, Vergleiche nur vom Verfahren.

Halten Sie Ihre Strichliste fest. Mit acht Zylindern braucht man mindestens 16 Wägungen; kein Verfahren der Welt kommt mit weniger aus. Warum das so ist und wie man solche Grenzen ausrechnet, klärt Abschnitt 4.3 — vergleichen Sie Ihr Ergebnis dann noch einmal mit dieser Zahl.

Alle vier Verfahren arbeiten im Folgenden auf derselben Ausgangslage, damit Sie die Abläufe vergleichen können:

```
  7  2  9  4 11  1  8  5 12  3 10  6
```

Jedes Verfahren wird dreifach angegeben: als Pseudocode, in Python und in Object Pascal (Delphi/Lazarus). Die Pascal-Fassungen arbeiten auf dem Feld `feld: TFeld` aus Abschnitt 4.1 mit den Positionen 1 bis `n`.

4.2.2 Sortieren durch direktes Auswählen (Selection Sort)

Idee. An der ersten Position muss die kleinste Zahl stehen. Also durchsucht man das ganze Feld nach dem Minimum und vertauscht es mit dem ersten Element. Danach ist die erste Position endgültig besetzt, und man wiederholt das Verfahren mit dem Rest des Feldes.

```
  7  2  9  4 11  1  8  5 12  3 10  6    Ausgangslage
  1  2  9  4 11  7  8  5 12  3 10  6    Minimum 1 mit der 7 getauscht
  1  2  9  4 11  7  8  5 12  3 10  6    2 steht bereits richtig
  1  2  3  4 11  7  8  5 12  9 10  6    3 mit der 9 getauscht
  1  2  3  4 11  7  8  5 12  9 10  6    4 steht bereits richtig
  1  2  3  4  5  7  8 11 12  9 10  6    5 mit der 11 getauscht
  1  2  3  4  5  6  8 11 12  9 10  7    6 mit der 7 getauscht
  1  2  3  4  5  6  7 11 12  9 10  8    7 mit der 8 getauscht
  1  2  3  4  5  6  7  8 12  9 10 11    8 mit der 11 getauscht
  1  2  3  4  5  6  7  8  9 12 10 11    9 mit der 12 getauscht
  1  2  3  4  5  6  7  8  9 10 12 11   10 mit der 12 getauscht
  1  2  3  4  5  6  7  8  9 10 11 12   11 mit der 12 getauscht
```

Auffällig ist, wie wenig sich pro Zeile ändert: Höchstens zwei Zahlen wechseln den Platz. Selection Sort tauscht sparsam — aber es sucht teuer, denn für jede Position wird der gesamte Rest durchgesehen.

Algorithmus

```
für i von 1 bis n-1:
    minpos := i
    für j von i+1 bis n:
        wenn feld[j] < feld[minpos] dann minpos := j
    vertausche feld[i] und feld[minpos]
```

Python

```python
def selection_sort(feld):
    n = len(feld)
    for i in range(n - 1):
        minpos = i
        for j in range(i + 1, n):
            if feld[j] < feld[minpos]:
                minpos = j
        feld[i], feld[minpos] = feld[minpos], feld[i]
    return feld
```

Object Pascal

```
procedure SelectionSort(var feld: TFeld; n: integer);
var i, j, minpos, hilf: integer;
begin
  for i := 1 to n - 1 do
  begin
    minpos := i;
    for j := i + 1 to n do
      if feld[j] < feld[minpos] then
        minpos := j;
    hilf := feld[i];                { Ringtausch ueber Hilfsvariable }
    feld[i] := feld[minpos];
    feld[minpos] := hilf;
  end;
end;
```

Arbeitsauftrag 2

6. Führen Sie Selection Sort von Hand auf `9 4 7 1 5 2` durch und notieren Sie jede Zwischenzeile.
7. Zählen Sie: Wie viele Vergleiche braucht Selection Sort bei n Elementen? Stellen Sie eine Formel auf. (Hinweis: $n-1$ Vergleiche im ersten Durchlauf, $n-2$ im zweiten, …)
8. Programmieren Sie das Verfahren so, dass nach jedem Durchlauf der Zustand des Feldes ausgegeben wird. Behalten Sie diese Ausgabefunktion bei — die folgenden Verfahren nutzen sie ebenfalls.
9. Begründen Sie: Selection Sort braucht immer gleich viele Vergleiche, egal wie die Eingabe aussieht. Welchen Vorteil und welchen Nachteil hat das?

4.2.3 Sortieren durch direktes Einfügen (Insertion Sort)

Idee. So sortieren die meisten Menschen ein Blatt Spielkarten: Man nimmt eine Karte heraus und schiebt sie an der richtigen Stelle in den bereits sortierten Teil ein. Der Anfang des Feldes ist dabei immer sortiert und wächst mit jedem Schritt um ein Element.

```
 7  2  9  4 11  1  8  5 12  3 10  6    Ausgangslage, die 7 gilt als sortiert
 2  7  9  4 11  1  8  5 12  3 10  6    2 vor die 7 geschoben
 2  7  9  4 11  1  8  5 12  3 10  6    9 bleibt, wo sie ist
 2  4  7  9 11  1  8  5 12  3 10  6    4 zwischen 2 und 7 eingefügt
 2  4  7  9 11  1  8  5 12  3 10  6    11 bleibt, wo sie ist
 1  2  4  7  9 11  8  5 12  3 10  6    1 ganz nach vorn
 1  2  4  7  8  9 11  5 12  3 10  6    8 zwischen 7 und 9
 1  2  4  5  7  8  9 11 12  3 10  6    5 zwischen 4 und 7
 1  2  4  5  7  8  9 11 12  3 10  6    12 bleibt, wo sie ist
 1  2  3  4  5  7  8  9 11 12 10  6    3 zwischen 2 und 4
 1  2  3  4  5  7  8  9 10 11 12  6    10 zwischen 9 und 11
 1  2  3  4  5  6  7  8  9 10 11 12    6 zwischen 5 und 7
```

Hier ist es umgekehrt zu Selection Sort: Gesucht wird wenig, denn die Suche bricht ab, sobald der Platz gefunden ist. Dafür wird viel geschoben.

Algorithmus

```
für i von 2 bis n:
    wert := feld[i]
    j := i
    solange j > 1 und feld[j-1] > wert:
        feld[j] := feld[j-1]        (nach rechts schieben)
        j := j - 1
    feld[j] := wert
```

Python

```python
def insertion_sort(feld):
    for i in range(1, len(feld)):
        wert = feld[i]
        j = i
        while j > 0 and feld[j - 1] > wert:
            feld[j] = feld[j - 1]
            j -= 1
        feld[j] = wert
    return feld
```

Object Pascal

```
procedure InsertionSort(var feld: TFeld; n: integer);
var i, j, wert: integer;
begin
  for i := 2 to n do
  begin
    wert := feld[i];
    j := i;
    while (j > 1) and (feld[j - 1] > wert) do
    begin
      feld[j] := feld[j - 1];        { nach rechts schieben }
      j := j - 1;
    end;
    feld[j] := wert;
  end;
end;
```

Arbeitsauftrag 3

10. Führen Sie Insertion Sort von Hand auf `9 4 7 1 5 2` durch.
11. Wie verhält sich das Verfahren, wenn das Feld bereits sortiert ist? Wie viele Vergleiche sind dann nötig? Vergleichen Sie mit Selection Sort.
12. Programmieren Sie Insertion Sort. Zählen Sie Vergleiche und Verschiebungen mit.
13. Warum steht in der Bedingung `j > 1 und feld[j-1] > wert` die Prüfung auf `j > 1` **vor** dem Vergleich? Was passiert bei umgekehrter Reihenfolge? Prüfen Sie außerdem nach, was die Delphi-Einstellung *Complete Boolean Evaluation* (`{$B+}` bzw. `{$B-}`) damit zu tun hat.

4.2.4 Sortieren durch Vertauschen (Bubble Sort)

Idee. Das einfachste Verfahren überhaupt: Man vergleicht benachbarte Zahlen und vertauscht sie, wenn die größere vorn steht. Das wiederholt man so lange, bis ein kompletter Durchlauf ohne Vertauschung bleibt. Große Zahlen steigen dabei wie Blasen nach hinten — daher der Name.

```
 7  2  9  4 11  1  8  5 12  3 10  6    Ausgangslage
 2  7  4  9  1  8  5 11  3 10  6 12    nach Durchlauf 1: die 12 steht hinten
 2  4  7  1  8  5  9  3 10  6 11 12    nach Durchlauf 2: die 11 sitzt
 2  4  1  7  5  8  3  9  6 10 11 12    nach Durchlauf 3
 2  1  4  5  7  3  8  6  9 10 11 12    nach Durchlauf 4
 1  2  4  5  3  7  6  8  9 10 11 12    nach Durchlauf 5
 1  2  4  3  5  6  7  8  9 10 11 12    nach Durchlauf 6
 1  2  3  4  5  6  7  8  9 10 11 12    nach Durchlauf 7
 1  2  3  4  5  6  7  8  9 10 11 12    Durchlauf 8 ohne Vertauschung: fertig
```

Nach jedem Durchlauf steht mindestens eine weitere Zahl endgültig am Ende. Der nächste Durchlauf darf deshalb einen Schritt früher aufhören — das spart die Hälfte der Vergleiche und ist in Aufgabe 16 gefragt.

Algorithmus

```
wiederhole
    getauscht := falsch
    für j von 1 bis n-1:
        wenn feld[j] > feld[j+1] dann
            vertausche feld[j] und feld[j+1]
            getauscht := wahr
bis getauscht = falsch
```

Python

```python
def bubble_sort(feld):
    n = len(feld)
    for i in range(n - 1):
        getauscht = False
        for j in range(n - 1 - i):
            if feld[j] > feld[j + 1]:
                feld[j], feld[j + 1] = feld[j + 1], feld[j]
                getauscht = True
        if not getauscht:
            break
    return feld
```

Object Pascal

```
procedure BubbleSort(var feld: TFeld; n: integer);
var i, j, hilf: integer;
    getauscht: boolean;
begin
  for i := 1 to n - 1 do
  begin
    getauscht := false;
    for j := 1 to n - i do
      if feld[j] > feld[j + 1] then
      begin
        hilf := feld[j];
        feld[j] := feld[j + 1];
        feld[j + 1] := hilf;
        getauscht := true;
      end;
    if not getauscht then
      break;
  end;
end;
```

Arbeitsauftrag 4

14. Führen Sie Bubble Sort von Hand auf `5 3 4 1 2` durch. Wie viele Durchläufe brauchen Sie?
15. Warum ist die Abfrage `getauscht` sinnvoll? Konstruieren Sie eine Eingabe, bei der sie viel spart, und eine, bei der sie nichts bringt.
16. Programmieren Sie Bubble Sort in zwei Fassungen: einmal mit fester Schleifengrenze, einmal mit dem verkürzten Bereich `n-1-i`. Zählen Sie jeweils die Vergleiche und vergleichen Sie.
17. Bubble Sort und Selection Sort brauchen etwa gleich viele Vergleiche, Bubble Sort aber deutlich mehr Vertauschungen. Begründen Sie das anhand der Abläufe oben.

4.2.5 Quicksort

LK

LK

Quicksort gehört zu den *höheren* Sortierverfahren und steht nur im LK-Lehrplan (dort zusammen mit Mergesort). Verbindlich für den Grundkurs sind Bubble-, Selection- und Insertion Sort. Quicksort taucht auch im Vergleich in Abschnitt 4.3.3 wieder auf.

Die ersten drei Verfahren arbeiten das Feld Position für Position ab. Quicksort verfolgt eine andere Strategie: **Teile und herrsche** (divide and conquer). Statt das Problem direkt zu lösen, zerlegt man es in zwei kleinere Probleme derselben Art — und wendet darauf dasselbe Verfahren an. Quicksort ist damit **rekursiv**.

Idee. Man wählt ein Element als **Pivot** (hier: das letzte des Bereichs) und sortiert das Feld so um, dass links vom Pivot nur kleinere und rechts nur größere Zahlen stehen. Das Pivot steht danach an seiner endgültigen Position — ohne dass der Rest sortiert wäre. Anschließend behandelt man den linken und den rechten Teilbereich auf dieselbe Weise.

```
 7  2  9  4 11  1  8  5 12  3 10  6    Ausgangslage
 2  4  1  5  3  6  8  7 12 11 10  9    Pivot 6 aus [1..12] landet auf Position 6
 2  1  3  5  4  6  8  7 12 11 10  9    Pivot 3 aus [1..5]  landet auf Position 3
 1  2  3  5  4  6  8  7 12 11 10  9    Pivot 1 aus [1..2]  landet auf Position 1
 1  2  3  4  5  6  8  7 12 11 10  9    Pivot 4 aus [4..5]  landet auf Position 4
 1  2  3  4  5  6  8  7  9 11 10 12    Pivot 9 aus [7..12] landet auf Position 9
 1  2  3  4  5  6  7  8  9 11 10 12    Pivot 7 aus [7..8]  landet auf Position 7
 1  2  3  4  5  6  7  8  9 11 10 12    Pivot 12 aus [10..12] landet auf Position 12
 1  2  3  4  5  6  7  8  9 10 11 12    Pivot 10 aus [10..11] landet auf Position 10
```

Acht Zerlegungen statt elf Auswahlrunden bei Selection Sort — und in jeder Zeile ist deutlich mehr in Bewegung. Der Grund: Jede Zerlegung halbiert die Arbeit ungefähr, statt sie nur um ein Element zu verkürzen.

Algorithmus

```
quicksort(links, rechts):
    wenn links < rechts:
        p := zerlege(links, rechts)
        quicksort(links, p-1)        (linker Rekursionsaufruf)
        quicksort(p+1, rechts)       (rechter Rekursionsaufruf)

zerlege(links, rechts):
    pivot := feld[rechts]
    i := links - 1
    für j von links bis rechts-1:
        wenn feld[j] <= pivot:
            i := i + 1
            vertausche feld[i] und feld[j]
    vertausche feld[i+1] und feld[rechts]
    gib i+1 zurück
```

Python

```python
def quicksort(feld, links=0, rechts=None):
    if rechts is None:
        rechts = len(feld) - 1
    if links < rechts:
        p = zerlege(feld, links, rechts)
        quicksort(feld, links, p - 1)
        quicksort(feld, p + 1, rechts)
    return feld


def zerlege(feld, links, rechts):
    pivot = feld[rechts]
    i = links - 1
    for j in range(links, rechts):
        if feld[j] <= pivot:
            i += 1
            feld[i], feld[j] = feld[j], feld[i]
    feld[i + 1], feld[rechts] = feld[rechts], feld[i + 1]
    return i + 1
```

Object Pascal

```
function Zerlege(var feld: TFeld; links, rechts: integer): integer;
var pivot, i, j, hilf: integer;
begin
  pivot := feld[rechts];
  i := links - 1;
  for j := links to rechts - 1 do
    if feld[j] <= pivot then
    begin
      i := i + 1;
      hilf := feld[i];
      feld[i] := feld[j];
      feld[j] := hilf;
    end;
  hilf := feld[i + 1];               { Pivot an seinen Platz }
  feld[i + 1] := feld[rechts];
  feld[rechts] := hilf;
  Result := i + 1;
end;

procedure QuickSort(var feld: TFeld; links, rechts: integer);
var p: integer;
begin
  if links < rechts then
  begin
    p := Zerlege(feld, links, rechts);
    QuickSort(feld, links, p - 1);   { linker Rekursionsaufruf }
    QuickSort(feld, p + 1, rechts);  { rechter Rekursionsaufruf }
  end;
end;
```

Merke

Rekursion braucht zwei Dinge: einen **Rekursionsschritt**, der das Problem verkleinert, und einen **Abbruchfall**, der ohne weiteren Aufruf auskommt. Bei Quicksort ist der Abbruchfall die Bedingung `links < rechts` — ein Bereich mit einem oder null Elementen ist bereits sortiert.

Arbeitsauftrag 5

18. Zerlegen Sie `8 3 11 5 1 9 4` von Hand mit dem letzten Element als Pivot. Wo landet das Pivot?
19. Zeichnen Sie für das Beispiel oben einen Baum der Rekursionsaufrufe. Wie tief ist er?
20. Programmieren Sie Quicksort. Lassen Sie nach jeder Zerlegung das Feld ausgeben.
21. Was passiert, wenn das Feld bereits sortiert ist und man immer das letzte Element als Pivot wählt? Rechnen Sie den Fall für `1 2 3 4 5` durch. (Diese Beobachtung wird in Abschnitt 4.3 wichtig.)
22. Ändern Sie Ihr Programm so, dass als Pivot das mittlere Element gewählt wird. Prüfen Sie, ob das Problem aus Aufgabe 21 verschwindet.

4.2.6 Werkzeug: die Sortier-Visualisierung

Die Abläufe oben stehen still auf Papier. In der Webfassung dieses Skripts steht an dieser Stelle der **Sortier-Visualizer**: Sie können die Anzahl der Elemente einstellen, die Ausgangslage wählen, Schritt für Schritt vorgehen oder im Modus *Rennen* die Verfahren gegeneinander antreten lassen. Vergleiche und Vertauschungen werden mitgezählt, und daneben steht der Quelltext des gerade laufenden Verfahrens.

> In der Webfassung steht hier ein Bedienfeld zum Ausprobieren: **Werkzeugseite**.

Arbeitsauftrag 6

23. Lassen Sie im Modus *Rennen* alle vier Verfahren auf 40 zufälligen Zahlen antreten. Notieren Sie die Vergleichszahlen. In welcher Reihenfolge werden die Verfahren fertig?
24. Wiederholen Sie das Rennen mit der Ausgangslage *fast sortiert*. Ein Verfahren gewinnt jetzt deutlich, ein anderes verliert stark. Welche? Erklären Sie beides mit den Abläufen aus Abschnitt 4.2.
25. Stellen Sie *umgekehrt sortiert* ein und beobachten Sie Quicksort. Verbinden Sie Ihre Beobachtung mit Ihrem Ergebnis aus Aufgabe 21.
26. Verdoppeln Sie die Anzahl der Elemente von 20 auf 40 und notieren Sie die Vergleiche für Selection Sort. Um welchen Faktor wächst die Zahl? Was sagt das über den Zusammenhang zwischen n und dem Aufwand?