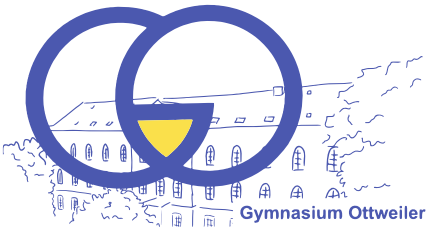
Kapitel 4

Suchen, Sortieren und Algorithmenanalyse



Skript Informatik · Grundkurs

Zwanzig Zahlen der Größe nach zu ordnen ist keine Kunst. Sie können es, ein Grundschulkind kann es, und Ihr Rechner sowieso. Interessant wird die Aufgabe erst durch drei Fragen, die dahinter liegen: *Wie genau* gehen Sie dabei vor? Gibt es ein Verfahren, das besser ist als Ihres? Und wie weist man so etwas nach, ohne einfach zwei Programme laufen zu lassen und zu stoppen?

Dieses Kapitel beantwortet die Fragen in dieser Reihenfolge. Zuerst das Suchen — die Aufgabe, derentwegen überhaupt sortiert wird. Dann vier Sortierverfahren, die dieselbe Aufgabe auf sehr verschiedene Weise lösen. Dann das Werkzeug, mit dem man sie vergleicht, ohne Rechner und Stoppuhr zu bemühen. Und am Ende die Frage, die die Informatik von der Programmierung trennt: Gibt es Probleme, für die es *überhaupt kein* Verfahren gibt?

Lernziele

Am Ende dieses Kapitels können Sie

- die sequentielle und die binäre Suche auf einem Feld erklären, von Hand durchführen und programmieren,
- begründen, warum die binäre Suche ein sortiertes Feld voraussetzt,
- die Verfahren Selection Sort, Insertion Sort, Bubble Sort und Quicksort erklären, von Hand durchführen und programmieren,
- das Prinzip „Teile und herrsche” und den Zusammenhang zur Rekursion beschreiben,
- die Zeitkomplexität eines Algorithmus abschätzen und in der Landau-Notation angeben,
- bester, schlechtester und mittlerer Fall an einem Beispiel unterscheiden,
- begründen, warum kein allgemeines Sortierverfahren schneller als $n \log n$ sein kann,
- erklären, was ein Test über die Korrektheit eines Programms aussagt — und was nicht,
- entscheidbare, partiell entscheidbare und unentscheidbare Probleme unterscheiden,
- den Beweis zur Unentscheidbarkeit des Halteproblems nachvollziehen und seine Bedeutung für die Praxis erklären.

4.1 Suchen

Sortiert wird selten um des Sortierens willen. Sortiert wird, damit man nachher schneller *findet*. Deshalb steht das Suchen hier vorn.

Die Aufgabe: In einem Feld mit n Einträgen soll festgestellt werden, ob ein gesuchter Wert vorkommt — und wenn ja, an welcher Position.

Alle Pascal-Fassungen in diesem Kapitel arbeiten auf demselben Feldtyp. Er wird einmal vereinbart und danach stillschweigend vorausgesetzt; die Positionen laufen von 1 bis `n`:

```
type
  TFeld = array [1..100] of integer;
```

In Python übernimmt eine gewöhnliche Liste diese Rolle; dort laufen die Positionen von 0 bis `len(feld) - 1`.

4.1.1 Sequentielle Suche

Das naheliegende Verfahren: von vorn durchgehen, bis der Wert auftaucht oder das Feld zu Ende ist.

Python

```
def sequentielle_suche(feld, ziel):
    for pos in range(len(feld)):
        if feld[pos] == ziel:
            return pos
    return -1                                   # nicht gefunden
```

Object Pascal

```
function SequentielleSuche(var feld: TFeld; n, ziel: Integer): Integer;
var
  pos: Integer;
begin
  Result := -1;
  for pos := 1 to n do
    if feld[pos] = ziel then
    begin
      Result := pos;
      Exit;                                     { gefunden, Schluss }
    end;
end;
```

Der Aufwand ist leicht abzuzählen:

Fall	Vergleiche
bester Fall (Wert steht vorn)	1
schlechtester Fall (Wert steht hinten oder fehlt)	n
mittlerer Fall (Wert ist vorhanden)	etwa $n/2$

Merke

Die sequentielle Suche hat einen großen Vorzug: Sie stellt **keine Bedingung** an das Feld. Es darf unsortiert sein, und sie funktioniert auf einer verketteten Liste ebenso wie auf einem Feld.

4.1.2 Binäre Suche

Ist das Feld **sortiert**, geht es sehr viel schneller. Jeder kennt das Verfahren vom Nachschlagen im Telefonbuch: Man schlägt in der Mitte auf und weiß danach, in welcher Hälfte weitergesucht werden muss.

Definition: Binäre Suche

Man betrachtet den Eintrag in der **Mitte** des Suchbereichs.

- Ist er der gesuchte Wert, ist man fertig.
- Ist er **größer**, wird in der **linken** Hälfte weitergesucht.
- Ist er **kleiner**, wird in der **rechten** Hälfte weitergesucht.

Das wiederholt man, bis der Wert gefunden oder der Suchbereich leer ist.

Beispiel: Die 47 suchen

```
Position:   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16
Feld:       3   7  12  18  23  29  34  41  47  52  58  63  71  76  84  92
```

Schritt	links	rechts	Mitte	Wert dort	Vergleich	weiter
1	1	16	8	41	47 > 41	rechts
2	9	16	12	63	47 < 63	links
3	9	11	10	52	47 < 52	links
4	9	9	9	47	gefunden	—

Vier Vergleiche. Die sequentielle Suche hätte neun gebraucht.

Python

```python
def binaere_suche(feld, ziel):
    links = 0
    rechts = len(feld) - 1
    while links <= rechts:
        mitte = (links + rechts) // 2
        if feld[mitte] == ziel:
            return mitte
        if feld[mitte] < ziel:
            links = mitte + 1                # linke Hälfte entfällt
        else:
            rechts = mitte - 1               # rechte Hälfte entfällt
    return -1
```

Object Pascal

```
function BinaereSuche(var feld: TFeld; n, ziel: Integer): Integer;
var
  links, rechts, mitte: Integer;
begin
  links := 1;
  rechts := n;
  Result := -1;
  while links <= rechts do
  begin
    mitte := (links + rechts) div 2;
    if feld[mitte] = ziel then
    begin
      Result := mitte;
      Exit;
    end;
    if feld[mitte] < ziel then
      links := mitte + 1
    else
      rechts := mitte - 1;
  end;
end;
```

Stolperstein

Die binäre Suche setzt ein **sortiertes** Feld voraus. Auf einem unsortierten Feld liefert sie keine Fehlermeldung — sie liefert eine **falsche Antwort**, und zwar lautlos. Das ist die unangenehmste Sorte Fehler.

4.1.3 Was die Halbierung bringt

Jeder Vergleich halbiert den Suchbereich. Nach k Schritten bleiben noch etwa $n/2^k$ Einträge übrig; die Suche endet, wenn dieser Rest 1 erreicht. Daraus ergibt sich die Zahl der Schritte: etwa $\log_2 n$.

n	sequentiell (schlechtester Fall)	binär (schlechtester Fall)
16	16	5
1 000	1 000	10
1 000 000	1 000 000	20
1 000 000 000	1 000 000 000	30

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

Suchen Sie im Bedienfeld auch nach einer Zahl, die nicht vorkommt — etwa 40. Die Suche endet trotzdem nach wenigen Schritten, und der Zähler zeigt, wie viele die sequentielle Suche gebraucht hätte.

Merke

Eine Milliarde Einträge, dreißig Vergleiche. Eine Verdoppelung der Datenmenge kostet die binäre Suche **einen einzigen** Vergleich zusätzlich — die sequentielle Suche dagegen doppelt so viele.

Das ist der Grund, warum Sortieren sich lohnt: Einmal sortieren, danach beliebig oft billig suchen.

Suchen Sie im Feld oben die 71 und die 50. Notieren Sie jeden Schritt wie in der Tabelle:

Arbeitsauftrag 1

1. Führen Sie die binäre Suche nach der 92 von Hand durch. Wie viele Vergleiche brauchen Sie — und warum ist es einer mehr als bei der 47?
2. Suchen Sie die 50, die nicht im Feld steht. Woran merkt das Verfahren, dass es abbrechen muss?
3. Programmieren Sie beide Verfahren und lassen Sie sie die Zahl der Vergleiche mitzählen. Prüfen Sie die Tabelle aus Abschnitt 4.1.3 für $n = 1000$ nach.
4. Begründen Sie: Ab welcher Zahl von Suchvorgängen lohnt es sich, ein unsortiertes Feld erst zu sortieren, statt sequentiell zu suchen?
5. Schreiben Sie die binäre Suche rekursiv (siehe Abschnitt 3.1). Was ist der Basisfall?

Gymnasium Ottweiler

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?

4.3 Algorithmenanalyse

4.3.1 Zuverlässigkeit

Die wichtigste Forderung an ein Programm ist nicht Schnelligkeit, sondern dass es tut, was es soll. Dass auch jahrelang scheinbar zuverlässige Software plötzlich falsche Ergebnisse liefert, zeigen diese Fälle:

- **Ariane 5, 1996.** Vierzig Sekunden nach dem Start zerstörte sich die Rakete selbst. Ursache war eine aus der Ariane 4 übernommene Routine, die eine 64-Bit-Gleitkommazahl in eine 16-Bit-Ganzzahl umwandelte. Für die schnellere Ariane 5 war der Wert zu groß. Der Code funktionierte fehlerfrei — nur für die falsche Rakete.
- **Jahr-2000-Problem.** Um teuren Speicherplatz zu sparen, wurde die Jahreszahl jahrzehntelang zweistellig gespeichert. Datumsvergleiche konnten 1900 und 2000 nicht unterscheiden. Die weltweite Umstellung kostete geschätzt mehrere hundert Milliarden Dollar.
- **Boeing 737, Startabbruch bei 60 Knoten.** Das Startprogramm behandelte die Fälle $v < 60$ und $v > 60$ — nicht aber $v = 60$. Meldeten die Sensoren genau diesen Wert, erkannte das Programm einen undefinierten Zustand und brach ab.

Der letzte Fall ist der lehrreichste, weil er keine exotische Technik voraussetzt: Ein einziger vergessener Randfall genügt. Als Faustzahl rechnet man in der Softwaretechnik mit mindestens einem Fehler je 10 KByte Quelltext.

Definitionen. Der Nachweis, dass ein Informatiksystem zuverlässig arbeitet, heißt **Validierung**. Man unterscheidet dabei zwei Wege:

- Bei der **Verifikation** wird formal, also durch logische Herleitung bewiesen, dass der Algorithmus seine Bedingungen einhält.
- Beim **Testen** wird das Programm mit ausgewählten Daten erprobt, deren Ergebnis bekannt ist.

Ein Algorithmus heißt **korrekt**, wenn er seiner Spezifikation genügt: Zu allen Eingabedaten, die eine bestimmte Vorbedingung erfüllen, liefert er Ausgabedaten, die eine bestimmte Nachbedingung erfüllen.

Merke

Testen kann nur die Anwesenheit von Fehlern zeigen, nie ihre Abwesenheit (Edsger Dijkstra). Wer alle Eingaben durchprobieren wollte, müsste bei einem Programm mit zwei 32-Bit-Zahlen als Eingabe rund $1{,}8 \cdot 10^{19}$ Fälle prüfen.

Arbeitsauftrag 7

27. Erklären Sie den Unterschied zwischen Verifikation und Testen an Ihrem Sortierprogramm: Was ließe sich beweisen, was nur testen?
28. Formulieren Sie Vor- und Nachbedingung für ein Sortierprogramm möglichst genau. Achtung: „Die Ausgabe ist aufsteigend sortiert" allein genügt nicht — warum?
29. Welche Testfälle würden Sie für ein Sortierprogramm wählen? Nennen Sie mindestens fünf und begründen Sie jeden.

4.3.2 Effizienz und Größenordnung

Zwei Algorithmen heißen **äquivalent**, wenn sie gleichen Eingabedaten gleiche Ausgabedaten zuordnen. Von zwei äquivalenten Algorithmen ist derjenige **effizienter**, der mit weniger Laufzeit und/oder weniger Speicherplatz auskommt. Beides zugleich lässt sich meist nicht beliebig verbessern: Ab einer gewissen Schwelle wird Geschwindigkeit mit Speicher erkauft und umgekehrt.

Die Laufzeit in Sekunden zu messen, hilft beim Vergleich von Algorithmen wenig. Sie hängt vom Prozessor ab, vom Compiler, vom Cache, vom Betriebssystem und davon, was sonst noch läuft. Deshalb zählt man stattdessen **Elementarschritte** — bei Sortierverfahren die Vergleiche.

Bezeichnung. Ein Problem der Größe n ist ein Problem, dessen Eingabe aus n Elementen besteht. Die Zahl $T(n)$ der Rechenschritte, die ein Algorithmus dafür benötigt, heißt **Zeitkomplexität**.

Zur Ermittlung nimmt man drei Vereinfachungen vor:

1. Man betrachtet nur die **dominante Operation**, von der die Laufzeit wesentlich abhängt.
2. Man unterscheidet **besten**, **schlechtesten** und **mittleren Fall**.
3. Man lässt additive und multiplikative Konstanten weg und geht zur **Größenordnung** über.

Der dritte Punkt ist der wichtigste und beim ersten Lesen der befremdlichste. Er besagt: Ob ein Verfahren n^2 oder $\frac{n^2}{2} + 3n + 17$ Schritte braucht, ist für große n gleichgültig — entscheidend ist das Wachstum. Notiert wird das in der **Landau-Notation**:

- $\mathcal{O}(n)$ — **lineares** Wachstum. Doppelt so viele Daten, doppelte Laufzeit.
- $\mathcal{O}(n^2)$ — **quadratisches** Wachstum. Doppelt so viele Daten, vierfache Laufzeit.
- $\mathcal{O}(n \log n)$ — dazwischen, und für Sortierverfahren die entscheidende Klasse.
- $\mathcal{O}(2^n)$ — **exponentielles** Wachstum. Ein Element mehr verdoppelt die Laufzeit. Schon bei kleinen Eingaben praktisch unbrauchbar.

4.3.3 Unsere vier Verfahren im Vergleich

Die folgende Tabelle enthält gemessene Vergleichszahlen: Mittelwerte über 300 zufällige Felder. Die letzte Spalte ist mit den Formeln hochgerechnet.

Verfahren	**Zeitkomplexität**	$n = 20$	$n = 1000$	$n = 100\,000$ **(gerechnet)**
Selection Sort	$\mathcal{O}(n^2)$, genau $\frac{n(n-1)}{2}$	190	499 500	$5 \cdot 10^9$
Bubble Sort	$\mathcal{O}(n^2)$, etwa $\frac{n^2}{2}$	181	498 530	$5 \cdot 10^9$
Insertion Sort	$\mathcal{O}(n^2)$, im Mittel etwa $\frac{n^2}{4}$	113	250 315	$2{,}5 \cdot 10^9$
Quicksort	$\mathcal{O}(n \log n)$, etwa $1{,}4\, n \log_2 n$	71	10 947	$2{,}3 \cdot 10^6$

Der Sprung in der letzten Spalte ist der eigentliche Punkt dieses Abschnitts: Bei hunderttausend Elementen trennt Quicksort von den anderen drei Verfahren nicht ein Faktor 2 oder 10, sondern ein Faktor von über tausend — gegenüber Selection und Bubble Sort sogar mehr als zweitausend. Ein Rechner, der die quadratischen Verfahren in einer Stunde schafft, wäre mit Quicksort nach wenigen Sekunden fertig. Kein Hardwarekauf der Welt gleicht das aus — der Unterschied liegt im Algorithmus.

Wichtig ist dabei die Unterscheidung der Fälle:

Verfahren	bester Fall	mittlerer Fall	schlechtester Fall
Selection Sort	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
Bubble Sort	$\mathcal{O}(n)$ (sortiert)	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
Insertion Sort	$\mathcal{O}(n)$ (sortiert)	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
Quicksort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$ (ungünstiges Pivot)

Quicksort ist also nicht in jedem Fall schnell: Wählt man stets das letzte Element als Pivot und ist das Feld bereits sortiert, zerlegt jede Runde das Feld in einen leeren und einen fast vollständigen Teil — das Verfahren wird quadratisch. In Python macht sich dieser Fall sogar doppelt bemerkbar: Die Rekursion wird dann n Ebenen tief, und ab etwa tausend Elementen bricht das Programm mit `RecursionError` ab, lange bevor die Laufzeit auffällt. Genau das haben Sie in den Aufgaben 21 und 25 beobachtet. Abhilfe schafft eine bessere Pivotwahl, etwa das mittlere von drei Elementen.

Merke

Kein allgemeines Sortierverfahren, das nur vergleicht, kommt mit weniger als $\mathcal{O}(n \log n)$ Vergleichen aus. Begründung: Bei n Elementen gibt es $n!$ mögliche Reihenfolgen. Jeder Vergleich hat zwei Ausgänge und kann die Zahl der in Frage kommenden Reihenfolgen also höchstens halbieren. Man braucht deshalb mindestens $\log_2(n!) \approx n \log_2 n$ Vergleiche, um eine einzige übrig zu lassen. Quicksort liegt im Mittel in dieser Größenordnung — besser geht es nicht.

Arbeitsauftrag 8

30. Ein Algorithmus braucht für $n = 1000$ genau 2 Sekunden. Wie lange braucht er für $n = 4000$, wenn er linear, quadratisch bzw. exponentiell wächst?
31. Begründen Sie mit der Formel $\frac{n(n-1)}{2}$, warum Selection Sort bei Verdoppelung der Elementzahl etwa viermal so viele Vergleiche braucht. Prüfen Sie es mit der Visualisierung nach.
32. Warum hat Bubble Sort einen besseren besten Fall als Selection Sort, obwohl beide im Mittel gleich viele Vergleiche brauchen?
33. Erweitern Sie Ihre Sortierprogramme um Zähler für Vergleiche und Vertauschungen. Sortieren Sie damit Felder der Größe 100, 200, 400 und 800 und tragen Sie die Werte in eine Tabelle ein. Bestätigen Sie das quadratische Wachstum.
34. Ein Verfahren mit $\mathcal{O}(2^n)$ schafft $n = 40$ in einer Stunde. Schätzen Sie die Laufzeit für $n = 60$. Kommentieren Sie das Ergebnis.
35. Diskutieren Sie: Für $n = 10$ ist Insertion Sort in der Praxis schneller als Quicksort, obwohl die Größenordnung schlechter ist. Wie passt das zusammen? (Stichwort: weggelassene Konstanten.)

4.4 Grenzen der Berechenbarkeit

LK

> LK
>
> Der ganze Abschnitt gehört zum LK-Themenfeld *Berechenbarkeit und ihre Grenzen*. Für den Grundkurs ist er nicht verbindlich — er beantwortet aber die Frage, die sich nach jedem Laufzeitvergleich aufdrängt: Gibt es Probleme, für die gar kein Verfahren existiert?

Lange hielt man jedes mathematische Problem für lösbar, wenn man sich nur genug anstrengt. David Hilbert (1862–1943) formulierte diese Überzeugung so: Jedes bestimmte mathematische Problem müsse einer strengen Erledigung fähig sein — entweder durch eine Antwort oder durch den Nachweis, dass eine Lösung unmöglich ist.

Im 20. Jahrhundert stellte sich heraus: Das stimmt nicht. Es gibt Probleme, die sich computergerecht aufschreiben lassen und trotzdem von keinem Algorithmus gelöst werden können. Und zwar nicht, weil die Rechner zu langsam wären — sondern grundsätzlich.

Definition. Eine Menge M natürlicher Zahlen heißt **entscheidbar**, wenn es einen Algorithmus gibt, der für jede natürliche Zahl nach endlich vielen Schritten feststellt, ob sie zu M gehört oder nicht. Man nennt ihn ein **zweiseitiges Entscheidungsverfahren**.

Sie heißt **partiell entscheidbar**, wenn es einen Algorithmus gibt, der nur die Zugehörigkeit feststellt: Gehört die Zahl dazu, hält er mit „ja" an; gehört sie nicht dazu, läuft er möglicherweise ewig. Das ist ein **einseitiges Entscheidungsverfahren**.

4.4.1 Das Goldbach-Problem: entscheidbar

Die Goldbachsche Vermutung (1742) lautet: Jede gerade Zahl ≥ 4 lässt sich als Summe zweier Primzahlen schreiben.

$$4 = 2 + 2 \qquad 6 = 3 + 3 \qquad 8 = 5 + 3 \qquad 10 = 7 + 3 \qquad 12 = 7 + 5$$

Bis heute ist sie weder bewiesen noch widerlegt. Für die Informatik stellen wir eine schwächere Frage: Gibt es einen Algorithmus, der für eine gegebene gerade Zahl $n \geq 4$ entscheidet, ob sie die Goldbach-Eigenschaft hat?

Den gibt es — und er ist einfach:

Python

```python
def prim(zahl):
    if zahl < 2:
        return False
    i = 2
    while i * i <= zahl:
        if zahl % i == 0:
            return False
        i += 1
    return True


def goldbach(n):
    for k in range(2, n // 2 + 1):
        if prim(k) and prim(n - k):
            return True
    return False
```

Object Pascal

```
function Prim(zahl: integer): boolean;
var i: integer;
begin
  Result := zahl >= 2;
  i := 2;
  while i * i <= zahl do
  begin
    if zahl mod i = 0 then
    begin
      Result := false;
      exit;
    end;
    i := i + 1;
  end;
end;

function Goldbach(n: integer): boolean;
var k: integer;
begin
  Result := false;
  for k := 2 to n div 2 do
    if Prim(k) and Prim(n - k) then
    begin
      Result := true;
      exit;
    end;
end;
```

Die Schleife läuft höchstens bis $n/2$, endet also garantiert. Damit ist die Goldbach-Eigenschaft entscheidbar. Beachten Sie den Unterschied: *Für jede einzelne Zahl* ist die Frage beantwortbar. Ob die Vermutung *für alle* Zahlen gilt, beantwortet der Algorithmus nicht — dafür müsste er unendlich viele Zahlen prüfen.

4.4.2 Wundersame Zahlen: partiell entscheidbar

Die **Collatz-Funktion** (L. Collatz, um 1930) ist definiert durch

$$f(n) = \begin{cases} n/2 & \text{falls } n \text{ gerade} \\ 3n+1 & \text{falls } n \text{ ungerade} \end{cases}$$

Betrachtet werden dabei nur natürliche Zahlen $n \geq 1$. Eine solche Zahl heißt **wundersam**, wenn wiederholtes Anwenden von f irgendwann bei 1 ankommt. Für $n = 5$ ergibt sich die Folge

$$5 \to 16 \to 8 \to 4 \to 2 \to 1$$

Die 5 ist also wundersam. Der naheliegende Algorithmus:

Python

```python
def wundersam(n):
    while n > 1:
        n = n // 2 if n % 2 == 0 else 3 * n + 1
    return True
```

Object Pascal

```
function Wundersam(n: integer): boolean;
begin
  while n > 1 do
    if n mod 2 = 0 then
      n := n div 2
    else
      n := 3 * n + 1;
  Result := true;
end;
```

Ist n wundersam, hält er nach endlich vielen Schritten und antwortet „ja". Ist n es nicht, läuft er ewig — und wir erfahren nie, ob er noch anhält oder nicht. Er ist also nur ein einseitiges Entscheidungsverfahren.

::: stolperstein Der Rumpf prüft `n > 1` und meldet alles andere ungeprüft als wundersam — für $n = 0$ oder $n = -7$ antwortet er also sofort „ja", obwohl diese Zahlen die 1 nie erreichen. Das Programm setzt den Definitionsbereich $n \geq 1$ stillschweigend voraus. Wer es ernst meint, fängt den Fall ab: In Python `if n < 1: raise ValueError(...)`, in Pascal eine Prüfung vor der Schleife. Dieselbe Lücke steckt in sehr vielen Programmen, die „für die sinnvollen Eingaben" geschrieben wurden. ::: Ein zweiseitiges ist bis heute nicht bekannt; Collatz vermutete, dass jede Zahl wundersam ist, bewiesen ist auch das nicht.

4.4.3 Game of Life: nachweislich unentscheidbar

Im Spiel des Lebens (J. Conway) lebt jede Zelle eines Gitters oder ist tot. Eine tote Zelle wird lebendig, wenn genau drei ihrer acht Nachbarn leben. Eine lebende Zelle stirbt, wenn mindestens vier Nachbarn leben (Übervölkerung) oder höchstens einer (Vereinsamung).

Gesucht ist ein Algorithmus, der für eine beliebige Anfangsgeneration entscheidet, ob sie irgendwann ausstirbt. Man kann die Generationen einfach nachrechnen: Stirbt die Population, ist die Antwort da. Stirbt sie nicht, rechnet der Algorithmus ewig. Wieder nur ein einseitiges Verfahren — mit einem Unterschied zur Collatz-Frage: Für das Game of Life ist *bewiesen*, dass es kein zweiseitiges Verfahren geben kann.

4.4.4 Das Halteproblem

Der wichtigste Satz dieses Abschnitts geht auf Alan Turing (1936) zurück — auf dieselbe Arbeit, die die Turingmaschine aus Abschnitt 1.2 einführte. Turing brauchte die Maschine genau dafür: um überhaupt sagen zu können, was „es gibt kein Verfahren" bedeutet.

Der Satz lässt sich für jede Programmiersprache formulieren:

Merke: Satz von Turing (Halteproblem)

Es gibt kein Programm, das für ein beliebiges Programm P und eine beliebige Eingabe x entscheidet, ob P angewendet auf x nach endlich vielen Schritten anhält. Die Halte-Eigenschaft ist unentscheidbar und nur partiell entscheidbar.

Beweis durch Widerspruch. Nehmen wir an, es gäbe eine Funktion `haelt(P, x)`, die für den Quelltext `P` und die Eingabe `x` den Wert `True` liefert, wenn `P` bei Eingabe `x` anhält, und sonst `False`.

```python
def haelt(P, x):
    # Annahme: liefert True, falls P bei Eingabe x anhaelt, sonst False
    ...
```

```pascal
function Haelt(P, x: string): boolean;
begin
  { Annahme: liefert true, falls P bei Eingabe x anhaelt, sonst false }
end;
```

Mit dieser Funktion bauen wir ein zweites Programm:

```python
def seltsam(P):
    if haelt(P, P):
        while True:      # Endlosschleife
            pass
    return "fertig"
```

```pascal
procedure Seltsam(P: string);
begin
  if Haelt(P, P) then
    while true do ;      { Endlosschleife }
  writeln('fertig');
end;
```

`seltsam` tut also genau das Gegenteil dessen, was `haelt` vorhersagt: Hält `P` bei Eingabe des eigenen Quelltextes an, so läuft `seltsam` ewig weiter; hält `P` nicht an, so hält `seltsam` an.

Da `P` beliebig ist, dürfen wir `P = seltsam` einsetzen. Dann gilt:

- Ist `haelt(seltsam, seltsam) = True`, so gerät `seltsam` in die Endlosschleife — hält also **nicht** an. Widerspruch.
- Ist `haelt(seltsam, seltsam) = False`, so überspringt `seltsam` die Schleife und gibt „fertig" zurück — hält also **doch** an. Widerspruch.

`seltsam` hält bei Eingabe des eigenen Quelltextes genau dann an, wenn es nicht anhält. Beides ist unmöglich. Also war die Annahme falsch: Die Funktion `haelt` kann es nicht geben. ■

Merke

Der Beweis sagt nichts über fehlende Rechenleistung. Er zeigt, dass die Annahme selbst widersprüchlich ist. Kein schnellerer Rechner, keine bessere Programmiersprache und keine künstliche Intelligenz ändern daran etwas.

4.4.5 Was folgt daraus für die Praxis?

Das Halteproblem ist kein Kuriosum für Theoretiker, sondern die Ursache für eine Reihe sehr praktischer Grenzen:

- Kein Werkzeug kann für beliebige Programme zuverlässig entscheiden, ob sie sich aufhängen. Entwicklungsumgebungen warnen vor *erkennbaren* Endlosschleifen — mehr ist nicht möglich.
- Kein Virenscanner kann durch reine Programmanalyse für jedes Programm entscheiden, ob es sich schädlich verhält. Deshalb arbeiten Scanner mit Signaturen, Heuristiken und Beobachtung zur Laufzeit.
- Allgemein gilt (Satz von Rice): Jede nichttriviale Frage über das *Verhalten* eines beliebigen Programms ist unentscheidbar. Fragen über seinen *Text* — Zeilenzahl, benutzte Schlüsselwörter — sind dagegen problemlos beantwortbar.

Arbeitsauftrag 9

36. Bestimmen Sie die Collatz-Folge für $n = 7$ und $n = 15$. Wie viele Schritte brauchen Sie jeweils?
37. Programmieren Sie den Goldbach-Test und prüfen Sie alle geraden Zahlen von 4 bis 10 000.
38. Programmieren Sie einen Collatz-Rechner, der die Folge und ihre Länge ausgibt. Suchen Sie die Zahl unter 1000 mit der längsten Folge.
39. Erklären Sie mit eigenen Worten den Unterschied zwischen entscheidbar und partiell entscheidbar. Ordnen Sie Goldbach-Eigenschaft, Wundersamkeit und Halte-Eigenschaft zu.
40. Warum genügt es für den Beweis des Halteproblems nicht zu sagen, `haelt` sei „zu schwer zu programmieren"? Was leistet der Widerspruchsbeweis stattdessen?
41. Ein Mitschüler schlägt vor: „Man lässt das Programm einfach eine Stunde laufen. Läuft es dann noch, hält es nicht an." Nehmen Sie Stellung.
42. Erklären Sie in fünf Sätzen, warum Turing für diesen Beweis die Maschine aus Abschnitt 1.2 brauchte — warum also die Aussage „es gibt kein Verfahren" ohne einen festgelegten Verfahrensbegriff gar nicht prüfbar wäre.

4.5 Zusammenfassung

- Die sequentielle Suche braucht im schlechtesten Fall n Vergleiche und stellt keine Bedingung an das Feld. Die binäre Suche kommt mit etwa $\log_2 n$ Vergleichen aus, setzt dafür aber ein **sortiertes** Feld voraus — und genau darum wird überhaupt sortiert.
- Sortierverfahren unterscheiden sich nicht im Ergebnis, sondern im Aufwand. Gezählt werden Vergleiche und Vertauschungen, nicht Sekunden.
- Selection Sort sucht teuer und tauscht sparsam, Insertion Sort umgekehrt. Bubble Sort ist am einfachsten und am langsamsten. Alle drei sind $\mathcal{O}(n^2)$.
- Quicksort zerlegt das Problem rekursiv (Teile und herrsche) und erreicht im Mittel $\mathcal{O}(n \log n)$ — bei ungünstiger Pivotwahl aber $\mathcal{O}(n^2)$.
- Die Landau-Notation beschreibt das Wachstum der Laufzeit und lässt Konstanten weg. Für große n entscheidet allein die Größenordnung.
- Kein vergleichendes Sortierverfahren kann besser als $\mathcal{O}(n \log n)$ sein.
- Testen zeigt die Anwesenheit von Fehlern, nie ihre Abwesenheit.
- Entscheidbar heißt: Ein Algorithmus antwortet immer mit ja oder nein. Partiell entscheidbar heißt: Er antwortet nur im Ja-Fall zuverlässig.
- Das Halteproblem ist unentscheidbar. Daraus folgen harte Grenzen für Werkzeuge, die das Verhalten fremder Programme beurteilen sollen.

Wiederholungsaufgaben

43. Ein sortiertes Feld enthält eine Million Einträge. Wie viele Vergleiche braucht die binäre Suche im schlechtesten Fall, wie viele die sequentielle? Begründen Sie, warum sich das Sortieren trotz seines Aufwands lohnt, sobald mehrfach gesucht wird.
44. Beschreiben Sie die vier Sortierverfahren in je zwei Sätzen, ohne Quelltext zu benutzen.
45. Sortieren Sie `6 1 9 3 7 2` mit allen vier Verfahren von Hand und zählen Sie jeweils die Vergleiche. Welches Verfahren gewinnt bei so kleinen Feldern?
46. Erklären Sie, warum die Aussage „Quicksort ist schneller als Bubble Sort" ohne Zusatz unvollständig ist.
47. Ein Programm braucht bei 1000 Datensätzen 1 Sekunde, bei 2000 Datensätzen 4 Sekunden. Welche Zeitkomplexität vermuten Sie? Wie lange dauert es bei 10 000 Datensätzen?
48. Begründen Sie, warum die Grenze $\mathcal{O}(n \log n)$ nur für *vergleichende* Verfahren gilt. Recherchieren Sie ein Verfahren, das schneller ist, und erklären Sie, was es anders macht.
49. Nehmen Sie Stellung: „Mit genug Rechenleistung lässt sich jedes Problem lösen."

Gymnasium Ottweiler