## 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