

Iterative und rekursive Algorithmen

1. Die Multiplikation der ganzen Zahlen x und y kann auch als x -maliges Addieren der Zahl y gelöst werden. Zum Beispiel ist $3 * 4 = 4 + 4 + 4 = 12$.

Erstellen Sie eine rekursive Funktion für diese Operation.

```
function multiplikation(a, b)
begin
  if a > 1 then
    return(multiplikation(a-1, b) + b);
  else
    return(b)
  end if
end
```

2. Lösen Sie die Übung 1 mithilfe eines iterativen Algorithmus.

```
function multiplikation(a, b)
begin
  prod := b;
  for i := 2 to a
  do
    prod := prod + b;
  end do
  return(prod);
end
```

4. Berechnen Sie mit einem rekursiven Algorithmus die Anzahl der Weizenkörner, ausgehend von einem Schachspiel. Das erste Feld soll mit einem Korn belegt sein, das zweite Feld mit zwei Körnern und so fort. Die Zahl der Körner soll auf jedem Feld verdoppelt werden.

```
Aufruf der Funktion im Hauptprogramm:
koenig(1, 64);           // die Zahl 1 ist der Startwert für die Berechnung
                        // ein Schachbrett hat 8 x 8 = 64 Felder

function koenig(i, a)
begin
  if a > 1 then
    i := i * 2;           // Anzahl der Körner verdoppeln
    return(i + koenig(i, a-1));
  // Anzahl der Körner dieser Berechnung werden zu
  // dem Ergebnis der nächsten Rekursion addiert;
  // mit dem Ergebnis dieser Berechnung wird die
  // nächste Rekursion aufgerufen, es dient somit als
  // Ausgangswert für die nächste Berechnung
  else
    return(1)
  end if
end
```

6. Erstellen Sie einen iterativen Algorithmus, der den größten gemeinsamen Teiler von zwei vorgegebenen ganzen Zahlen ermittelt. Verwenden Sie dazu den Algorithmus von Euklid, der die folgenden Schritte umfasst.

✓ Ausgangspunkt sind zwei ganze Zahlen, z. B. 75 und 54.

- ✓ Bilden Sie durch die ganzzahlige Division den Quotienten aus der größeren Zahl (Dividend) und der kleineren Zahl (Divisor), z. B. 75/54.
- ✓ Ist der Rest der ganzzahligen Division verschieden von 0, wird ein neues Zahlenpaar aus der kleineren Zahl und dem Rest der ganzzahligen Division gebildet, z. B. 54 und 21.
- ✓ Bilden Sie davon analog zum ersten Zahlenpaar den Quotienten, z. B. 54/21.
- ✓ Wiederholen Sie den Vorgang so lange, bis der Rest der ganzzahligen Division 0 ist.
- ✓ Der größte gemeinsame Teiler ist der Divisor der letzten Division.

```
function ggt(a, b)
begin
  rest := 1;
  if a > b then
    g := a; // g ist die größere Zahl
    k := b; // k ist die kleinere Zahl
  else
    g := b;
    k := a;
  end if
  rest := g % k; // Rest der ganzzahligen Division ermitteln (Modulo)
  if rest > 0 then
    do
      g := k; // die kleinere Zahl wird zum Dividenden
      k := rest; // der Rest der ganzzahligen Division wird zum Divisor
      rest := g % k;
      while (rest > 0) // solange der Rest größer null ist, wird die Modulo-
      end do // operation erneut ausgeführt
    end if // ist der Rest bei der ersten Modulooperation gleich
    rest := k; // null, so ist die kleinere Zahl der größte
  return(rest) // gemeinsame Teiler
end
```

8. Die Verzeichnisse und Dateien in einem Betriebssystem sind in der Regel innerhalb einer Baumstruktur angeordnet. Um eine bestimmte Datei zu suchen, werden die Ordner und Unterordner nacheinander durchsucht, bis die Datei gefunden wird. Wie kann ein Algorithmus zum Suchen einer Datei in einer solchen Baumstruktur aussehen? Geben Sie eine iterative und eine rekursive Lösung an.

iterative Lösung:

Aufruf aus Hauptprogramm:

```
ermittle_Verzeichnisbaum(baum);
```

```
suche_datei(baum, dateiname);
```

baum ist eine Liste, die Verzeichnisse und Dateinamen enthält.
Verzeichnisse sind wieder Listen, die wiederum Verzeichnisse und
Dateinamen enthalten können usw.

Z. B.: ((a, b, c), ((d, e, f), (g, h, i, j), x, y, z), k, l, m, n, o)

```
function suche_datei(baum, dateiname)
```

```
begin
```

```
    pos := wurzel(baum);
```

```
    while (weitere_Elemente_vorhanden(pos))
```

```
        if pos zeigt auf gesuchten dateinamen then
```

```
            return (pos);
```

```
        else
```

```
            if weitere_Elemente_vorhanden(pos) then
```

```
                nächstes_element(baum);
```

```
            else
```

```
                return (null);
```

```
    end do;
```

```
end
```

Erklärung zu den aufgerufenen Funktionen:

nächstes_element(baum): liefert das nächste Element der Liste

weitere_Elemente_vorhanden(pos): liefert wahr, wenn es weitere Elemente in der
Liste gibt, sonst falsch

rekursive Lösung:

Aufruf aus Hauptprogramm:

```
ermittle_Verzeichnisbaum(baum);
```

```
suche_datei(baum, dateiname);
```

baum ist eine Liste, die Verzeichnisse und Dateinamen enthält.

Verzeichnisse sind wieder Listen, die wiederum Verzeichnisse und Dateinamen enthalten können usw.

Z. B.: ((a, b, c), ((d, e, f), (g, h, i, j), x, y, z), k, l, m, n, o)

```
function suche_datei(baum, dateiname)
begin
  pos := nächstes_element(baum);
  if pos zeigt auf Verzeichnis then
    if suche_datei(liste(pos), dateiname) = null then
      if rest_der_liste(pos) = leer then
        return(null);
      else
        suche_datei(rest_der_liste(pos), dateiname);
      end if
    else
      return(pos);
    end if
  else
    if pos zeigt auf gesuchten dateinamen then
      return(pos);
    else
      if weitere_Elemente_vorhanden(pos) then
        suche_datei(rest_der_liste(pos), dateiname);
      else
        return(null);
      end if
    end do
  end
```

Erklärung zu den aufgerufenen Funktionen:

nächstes_element(baum): liefert das nächste Element der Liste

liste(pos): gibt die Liste zurück, auf die pos verweist
(Unterverzeichnis)

rest_der_liste(pos): Rest der Liste ohne aktuelles Element

weitere_Elemente_vorhanden(pos): liefert wahr, wenn es weitere Elemente in der
Liste gibt, sonst falsch

Anmerkung: Jede Programmiersprache, die die Listenverarbeitung unterstützt, bietet Funktionen für deren Verarbeitung. Diese liefern z. B. das erste bzw. das nächste Element der Liste oder den Rest der Liste. Auch um festzustellen, ob sich weitere Elemente in der Liste befinden, gibt es eine Funktion.