Zählschleife `for`

- Mathematische Symbole $\sum_{j=1}^{n}$ und $\prod_{j=1}^{n}$
- Zählschleife
- `for`

Schleifen

- Schleifen führen einen oder mehrere Befehle wiederholt aus
- In Aufgabenstellung häufig Hinweise, wie
  - Vektoren & Matrizen
  - Laufvariablen $j = 1, \ldots, n$
  - Summen $\sum_{j=1}^{n} a_j := a_1 + a_2 + \cdots + a_n$
  - Produkte $\prod_{j=1}^{n} a_j := a_1 \cdot a_2 \cdots a_n$
  - Text wie z.B. *solange bis* oder *solange wie*
- Man unterscheidet
  - Zählschleifen (`for`): Wiederhole etwas eine gewisse Anzahl oft
  - Bedingungsschleifen: Wiederhole etwas bis eine Bedingung eintritt

Die `for`-Schleife

- `for (init. ; cond. ; step-expr.) statement`
- Ablauf einer `for`-Schleife
  - (1) Ausführen der Initialisierung `init.`
  - (2) Abbruch, falls Bedingung `cond.` nicht erfüllt
  - (3) Ausführen von `statement`
  - (4) Ausführen von `step-expr.`
  - (5) Sprung nach (2)
- `statement` ist
  - entweder eine Zeile
  - oder mehrere Zeilen in geschwungenen Klammern { ... }, sog. Block

```
 1  #include <stdio.h>
 2
 3  main() {
 4    int j = 0;
 5
 6    for (j=5; j>0 ; j=j-1)
 7      printf("%d ",j);
 8
 9    printf("\n");
10  }
```

- `j=j-1` in 6 ist Zuweisung, keine math. Gleichheit!
- Output:

```
5 4 3 2 1
```


Vektor einlesen & ausgeben

```
 1  #include <stdio.h>
 2
 3  void scanvector(double input[], int dim) {
 4    int j = 0;
 5    for (j=0; j<dim; j=j+1) {
 6      input[j] = 0;
 7      printf("%d: ",j);
 8      scanf("%lf",&input[j]);
 9    }
10  }
11
12  void printvector(double output[], int dim) {
13    int j = 0;
14    for (j=0; j<dim; j=j+1) {
15      printf("%f ",output[j]);
16    }
17    printf("\n");
18  }
19
20  main() {
21    double x[5];
22    scanvector(x,5);
23    printvector(x,5);
24  }
```

- Funktionen müssen Länge von Arrays kennen!
  - d.h. zusätzlicher Input-Parameter nötig
- Arrays werden mit Call by Reference übergeben!

Minimum eines Vektors

```
 1  #include <stdio.h>
 2  #define DIM 5
 3
 4  void scanvector(double input[], int dim) {
 5    int j = 0;
 6    for (j=0; j<dim; j=j+1) {
 7      input[j] = 0;
 8      printf("%d: ",j);
 9      scanf("%lf",&input[j]);
10    }
11  }
12
13  double min(double input[], int dim) {
14    int j = 0;
15    double minval = input[0];
16
17    for (j=1; j<dim; j=j+1) {
18      if (input[j]<minval) {
19        minval = input[j];
20      }
21    }
22    return minval;
23  }
24
25  main() {
26    double x[DIM];
27    scanvector(x,DIM);
28    printf("Minimum des Vektors ist %f\n",
29           min(x,DIM));
30  }
```


Beispiel: Summensymbol $\sum$

- Berechnung der Summe $S = \sum_{j=1}^{N} a_j$:
  - Abkürzung $\sum_{j=1}^{N} a_j := a_1 + a_2 + \ldots + a_N$
- Definiere theoretische Hilfsgröße $S_k = \sum_{j=1}^{k} a_k$
- Dann gilt
  - $S_1 = a_1$
  - $S_2 = S_1 + a_2$
  - $S_3 = S_2 + a_3$ etc.
- Realisierung also durch N-maliges Aufsummieren
  - ACHTUNG: Zuweisung, keine Gleichheit
    - S = a_1
    - S = S + a_2
    - S = S + a_3 etc.

Beispiel: Summensymbol $\sum$

```
 1  #include <stdio.h>
 2
 3  main() {
 4    int j = 0;
 5    int n = 100;
 6
 7    int sum = 0;
 8
 9    for (j=1; j<=n; j=j+1) {
10      sum = sum+j;
11    }
12
13    printf("sum_{j=1}^{%d} j = %d\n",n,sum);
14  }
```

- Programm berechnet $\sum_{j=1}^{n} j$ für $n = 100$.
- Output:

```
sum_{j=1}^{100} j = 5050
```

- ACHTUNG: Bei iterierter Summation nicht vergessen, Ergebnisvariable auf Null zu setzen vgl. Zeile 7
  - Anderenfalls: Falsches/Zufälliges Ergebnis!
- statt `sum = sum + j;`
  - Kurzschreibweise `sum += j;`

Beispiel: Produktsymbol $\prod$

```
 1  #include <stdio.h>
 2
 3  main() {
 4    int j = 0;
 5    int n = 5;
 6
 7    int factorial = 1;
 8
 9    for (j=1; j<=n; j=j+1) {
10      factorial = factorial*j;
11    }
12
13    printf("%d! = %d\n",n,factorial);
14  }
```

- Prg berechnet Faktorielle $n! = \prod_{j=1}^{n} j$ für $n = 5$.
- Output:

```
5! = 120
```

- ACHTUNG: Bei iteriertem Produkt nicht vergessen, Ergebnisvariable auf Eins zu setzen vgl. Zeile 7
  - Anderenfalls: Falsches/Zufälliges Ergebnis!

Matrix-Vektor-Multiplikation

- Man darf `for`-Schleifen schachteln
  - Typisches Beispiel: Matrix-Vektor-Multiplikation
- Seien $A \in \mathbb{R}^{M \times N}$ Matrix, $x \in \mathbb{R}^N$ Vektor
- Def $b := Ax \in \mathbb{R}^M$ durch $b_j = \sum_{k=0}^{N-1} A_{jk} x_k$
  - Indizierung in C startet bei 0
- $Ax = b$ ist also Schreibweise für lineares GLS

$$\begin{array}{ccccccc} A_{00}x_0 & + & A_{01}x_1 & + \ldots + & A_{0,N-1}x_{N-1} & = & b_0 \\ A_{10}x_0 & + & A_{11}x_1 & + \ldots + & A_{1,N-1}x_{N-1} & = & b_1 \\ A_{20}x_0 & + & A_{21}x_1 & + \ldots + & A_{2,N-1}x_{N-1} & = & b_2 \\ \vdots & & \vdots & & \vdots & & \vdots \\ A_{M-1,0}x_0 & + & A_{M-1,1}x_1 & + \ldots + & A_{M-1,N-1}x_{N-1} & = & b_{M-1} \end{array}$$

- Implementierung
  - äußere Schleife über j, innere für Summe

```
for (j=0; j<M; j=j+1) {
  b[j] = 0;
  for (k=0; k<N; k=k+1) {
    b[j] = b[j] + A[j][k]*x[k];
  }
}
```

- ACHTUNG: Init. `b[j] = 0` nicht vergessen!

Matrix spaltenweise speichern

- math. Bibliotheken speichern Matrizen idR. spaltenweise als Vektor
  - $A \in \mathbb{R}^{M \times N}$, gespeichert als $a \in \mathbb{R}^{MN}$
  - $a = (A_{00}, A_{10}, ..., A_{M-1,0},\ A_{01}, A_{11}, \ldots,\ A_{M-1,N-1})$
  - A_{jk} entspricht also a_ℓ mit $\ell = j + k \cdot M$
- Matrix-Vektor-Produkt
  - $b := Ax \in \mathbb{R}^M$, $b_j = \sum_{k=0}^{N-1} A_{jk} x_k$
  - mit `double A[M][N];`

```
for (j=0; j<M; j=j+1) {
  b[j] = 0;
  for (k=0; k<N; k=k+1) {
    b[j] = b[j] + A[j][k]*x[k];
  }
}
```

- Matrix-Vektor-Produkt (spaltenweise gespeichert)
  - mit `double A[M*N];`

```
for (j=0; j<M; j=j+1) {
  b[j] = 0;
  for (k=0; k<N; k=k+1) {
    b[j] = b[j] + A[j+k*M]*x[k];
  }
}
```


MinSort (= Selection Sort)

- Gegeben: Ein Vektor $x \in \mathbb{R}^n$
- Ziel: Sortiere x, sodass $x_1 \leq x_2 \leq \cdots \leq x_n$
- Algorithmus (1. Schritt)
  - suche Minimum x_k von $x_1, \ldots, x_n$
  - vertausche x_1 und x_k, d.h. x_1 ist kleinstes Elt.
- Algorithmus (2. Schritt)
  - suche Minimum x_k von $x_2, \ldots, x_n$
  - vertausche x_2 und x_k, d.h. x_2 zweit kleinstes Elt.
- nach $n-1$ Schritten ist x sortiert
- Hinweise zur Realisierung (vgl. UE)
  - Länge n ist Konstante im Hauptprogramm
    - d.h. n ist im Hauptprg nicht veränderbar
  - aber n ist Inputparameter der Funktion `minsort`
    - d.h. Funktion arbeitet für beliebige Länge


```
 1  #include <stdio.h>
 2  #define DIM 5
 3
 4  void scanvector(double input[], int dim) {
 5    int j = 0;
 6    for (j=0; j<dim; j=j+1) {
 7      input[j] = 0;
 8      printf("%d: ",j);
 9      scanf("%lf",&input[j]);
10    }
11  }
12
13  void printvector(double output[], int dim) {
14    int j = 0;
15    for (j=0; j<dim; j=j+1) {
16      printf("%f ",output[j]);
17    }
18    printf("\n");
19  }
20
21  void minsort(double vector[], int dim) {
22    int j, k, argmin;
23    double tmp;
24    for (j=0; j<dim-1; j=j+1) {
25      argmin = j;
26      for (k=j+1; k<dim; k=k+1) {
27        if (vector[argmin] > vector[k]) {
28          argmin = k;
29        }
30      }
31      if (argmin > j) {
32        tmp = vector[argmin];
33        vector[argmin] = vector[j];
34        vector[j] = tmp;
35      }
36    }
37  }
38
39  main() {
40    double x[DIM];
41    scanvector(x,DIM);
42    minsort(x,DIM);
43    printvector(x,DIM);
44  }
```

Aufwand

- Aufwand von Algorithmen
- Landau-Symbol $\mathcal{O}$
- `time.h`, `clock_t`, `clock()`

Aufwand eines Algorithmus

- wichtige Kenngröße für Algorithmen
  - um Algorithmen zu bewerten / vergleichen
- Aufwand = Anzahl benötigter Operationen
  - Zuweisungen
  - Vergleiche
  - arithmetische Operationen
- programmspezifische Operationen nicht gezählt
  - Deklarationen & Initialisierungen
  - Schleifen, Verzweigungen etc.
  - Zählvariablen
- Aufwand wird durch „einfaches“ Zählen ermittelt
- Konventionen zum Zählen nicht einheitlich
- in der Regel ist Aufwand für worst case interessant
  - d.h. maximaler Aufwand im schlechtesten Fall

Beispiel: Maximum suchen

```
 1  double maximum(double vector[], int n) {
 2    int i = 0;
 3    double max = 0;
 4
 5    max = vector[0];
 6    for (i=1; i<n; i=i+1) {
 7      if (vector[i] > max) {
 8        max = vector[i];
 9      }
10    }
11    return max;
12  }
```

- Aufwand:
  - 1 Zuweisung ⇝ Zeile 5
  - $n-1$ Schritte mit jeweils ⇝ Zeile 6–10
    - 1 Vergleich ⇝ Zeile 7
    - 1 Zuweisung (worst case!) ⇝ Zeile 8
- insgesamt $1 + 2(n-1) = 2n-1$ Operationen

Landau-Symbol $\mathcal{O}$ (= groß-O)

- oft nur Größenordnung des Aufwands interessant
- Schreibweise $f = \mathcal{O}(g)$ für $x \to x_0$
  - heißt $\limsup_{x \to x_0} \left| \frac{f(x)}{g(x)} \right| < \infty$
  - d.h. $|f(x)| \leq C\,|g(x)|$ für $x \to x_0$.
  - d.h. f wächst höchstens so schnell wie g
- Beispiel: Maximum suchen
  - Aufwand $2n-1 = \mathcal{O}(n)$ für $n \to \infty$
- häufig entfällt „für $x \to x_0$“
  - dann Grenzwert x_0 kanonisch z.B. $2n-1 = \mathcal{O}(n)$
- Sprechweise:
  - Algorithmus hat linearen Aufwand, falls Aufwand $\mathcal{O}(n)$ bei Problemgröße n
    - Maximumssuche hat linearen Aufwand
  - Algorithmus hat fastlinearen Aufwand, falls Aufwand $\mathcal{O}(n \log n)$ bei Problemgröße n
  - Algorithmus hat quadratischen Aufwand, falls Aufwand $\mathcal{O}(n^2)$ bei Problemgröße n
  - Algorithmus hat kubischen Aufwand, falls Aufwand $\mathcal{O}(n^3)$ bei Problemgröße n

Matrix-Vektor Multiplikation

```
 1  void MVM(double A[], double x[], double b[],
 2           int m, int n) {
 3    int i = 0;
 4    int j = 0;
 5
 6    for (j=0; j<m; j=j+1) {
 7      b[j] = 0;
 8      for (k=0; k<n; k=k+1) {
 9        b[j] = b[j] + A[j+k*m]*x[k];
10      }
11    }
12  }
```

- `m` Schritte ⇝ Zeile 6–11
  - 1 Zuweisung ⇝ Zeile 7
  - jeweils `n` mal ⇝ Zeile 8–10
    - 1 Multiplikation ⇝ Zeile 9
    - 1 Addition ⇝ Zeile 9
    - 1 Zuweisung ⇝ Zeile 9
- Insgesamt $3mn + m$ Operationen
- Aufwand $\mathcal{O}(mn)$
  - bzw. Aufwand $\mathcal{O}(n^2)$ für $m = n$
  - d.h. quadratischer Aufwand für $m = n$

Suchen im Vektor

```
 1  int search(int vector[], int value, int n) {
 2
 3    int j = 0;
 4    for (j=0; j<n; j=j+1) {
 5      if (vector[j] == value) {
 6        return j;
 7      }
 8    }
 9
10    return -1;
11  }
```

- Aufgabe:
  - Suche Index `j` mit `vector[j]` = `value`
  - Rückgabe `-1`, falls nicht ex.
- Achtung bei Gleichheit mit `double` (später!)
- `n` Schritte
  - 1 Vergleich
- Insgesamt n Operationen
- Aufwand $\mathcal{O}(n)$

Binäre Suche im sortierten Vektor

```
 1  int binsearch(int vector[], int value, int n) {
 2
 3    int j = 0;
 4    int start = 0;
 5    int end = n-1;
 6
 7    for ( ; start <= end ; ) {
 8      j = 0.5*(end+start);
 9      if (vector[j] == value) {
10        return j;
11      }
12      else if (vector[j] > value) {
13        end = j-1;
14      }
15      else {
16        start = j+1;
17      }
18      printf("%d %d j=%d\n",start,end,j);
19    }
20
21    return -1;
22  }
```

- Voraussetzung: Vektor ist aufsteigend sortiert
- Modifiziere Idee des Bisektionsverfahrens
  - Betrachte halben Vektor, falls `vector[j]` ≠ `value`
- Frage: Wieviele Iterationen hat der Algorithmus?
  - jeder Schritt halbiert Vektor
  - max. Anzahl Schritte k erfüllt $n/2^k = 1$
  - also maximal $k = \log_2 n$ Schritte
    - je 2 Vergl. + 2 Zuw. + 1 Mult. + 1 Add.
- Aufwand $\mathcal{O}(\log_2 n)$, d.h. logarithmischer Aufwand

Minsort

```
 1  void minsort(int vector[], int n) {
 2    int j,k,argmin;
 3    double tmp;
 4
 5    for (j=0; j<n-1; j=j+1) {
 6      argmin = j;
 7      for (k=j+1; k<n; k=k+1) {
 8        if (vector[argmin] > vector[k]) {
 9          argmin = k;
10        }
11      }
12      if (argmin > j) {
13        tmp = vector[argmin];
14        vector[argmin] = vector[j];
15        vector[j] = tmp;
16      }
17    }
18  }
```

- `n-1` Schritte
  - 1 Zuweisung
  - jeweils $n - (j + 1) = n - j - 1$ mal
    - 1 Vergleich
    - 1 Zuweisung (worst case!)
  - jeweils 1 Vergleich
  - jeweils 3 Zuweisungen (worst case!)
- Insgesamt $5(n-1) + \sum_{j=0}^{n-2}(n-j-1)2$
  $= 5(n-1) + 2\sum_{k=1}^{n-1} k$
  $= 5(n-1) + 2\frac{n(n-1)}{2}$
  $= n^2+4n-5$, d.h. quadratischer Aufwand $\mathcal{O}(n^2)$

Zeitmessung

- Wozu Zeitmesssung?
  - Vergleich von Algorithmen
  - Vergleich von Implementierungen
  - Überprüfen theoretischer Voraussagen

- theoretische Voraussagen
  - linearer Aufwand
    - Problemgröße $n \Rightarrow Cn$ Operationen
    - Problemgröße $kn \Rightarrow Ckn$ Operationen
    - d.h. 3× Problemgröße ⇒ 3× Rechenzeit
  - quadratischer Aufwand
    - Problemgröße $n \Rightarrow Cn^2$ Operationen
    - Problemgröße $kn \Rightarrow Ck^2n^2$ Operationen
    - d.h. 3× Problemgröße ⇒ 9× Rechenzeit
  - etc.

- Bibliothek `time.h`
  - Datentyp `clock_t` für Zeitvariablen
    für Ausgabe Typecast nicht vergessen!
  - Funktion `clock()` liefert Rechenzeit
    seit Programmbeginn
  - Konstante `CLOCKS_PER_SEC` zum Umrechnen:
    `Zeitvariable/CLOCKS_PER_SEC` liefert
    Angabe in Sekunden

Beispiel: Zeitmessung

```
 1  #include <stdio.h>
 2  #include <time.h>
 3
 4  #define DIM 1000
 5  #define VAL 500
 6
 7  int search(int vector[], int value, int n);
 8  int binsearch(int vector[], int value, int n);
 9  void minsort(int vector[], int n);
10
11  main() {
12    clock_t t1,t2;
13    int i = 0;
14    int v[DIM];
15
16    for(i=0; i<DIM; i=i+1) {
17      printf("v[%d]=",i);
18      scanf("%d",&v[i]);
19    }
20    t1 = clock();
21    i = search(v,VAL,DIM);
22    t2 = clock();
23    printf("search: %f\n",
24           (double)(t2-t1)/CLOCKS_PER_SEC);
25    t1 = clock();
26    minsort(v,DIM);
27    t2 = clock();
28    printf("minsort: %f\n",
29           (double)(t2-t1)/CLOCKS_PER_SEC);
30    t1 = clock();
31    i = binsearch(v,VAL,DIM);
32    t2 = clock();
33    printf("binary search: %f\n",
34           (double)(t2-t1)/CLOCKS_PER_SEC);
35  }
```


Vergleich von Laufzeit

	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(\log_2 n)$
n	`search`	`minsort`	`binsearch`
1.000	0.00	0.00	0.00
2.000	0.00	0.00	0.00
4.000	0.00	0.02	0.00
8.000	0.00	0.07	0.00
16.000	0.00	0.29	0.00
32.000	0.00	1.11	0.00
64.000	0.00	4.45	0.00
128.000	0.00	17.81	0.00
256.000	0.00	71.20	0.00
512.000	0.00	284.38	0.00
1.024.000	0.00	$\geq$ 18min	0.00
2.048.000	0.00	$\geq$ 72min	0.00
4.096.000	0.01	$\geq 4,5$h	0.00
8.192.000	0.02	$\geq$ 18h	0.00
16.384.000	0.04	$\geq$ 3d	0.00
32.768.000	0.07	$\geq$ 12d	0.00
65.536.000	0.15	$\geq 1,5$m	0.00
131.072.000	0.31	$\geq$ 6m	0.00
262.144.000	0.60	$\geq$ 2y	0.00
524.288.000	1.21	$\geq$ 8y	0.00
1.048.576.000	2.42	$\geq$ 32y	0.00

- log. Aufwand perfekt, denn $2^{30} > 1.048.576.000$
- auch linearer Aufwand liefert sehr gute Rechenzeit
- Quadratischer Aufwand für große n spürbar
- Fazit: Algorithmen sollen kleinen Aufwand haben
  - Ziel der numerischen Mathematik
  - nicht immer möglich

Bedingungsschleifen

- Bedingungsschleife
- kopfgesteuert vs. fußgesteuert
- Operatoren `++` und `--`

- `while`
- `do - while`

Die while-Schleife

- Formal: `while(condition) statement`
  - vgl. `binsearch`: `for( ; condition ; )`
- *Vor* jedem Durchlauf wird `condition` geprüft & Abbruch, falls nicht erfüllt
  - sog. kopfgesteuerte Schleife
- Eventuell also kein einziger Durchlauf!
- `statement` kann Block sein

```
 1  #include <stdio.h>
 2
 3  main() {
 4    int counter = 5;
 5
 6    while (counter > 0) {
 7      printf("%d ",counter);
 8      counter = counter-1;
 9    }
10    printf("\n");
11  }
```

- Output:

```
5 4 3 2 1
```


Operatoren ++

- `++a` und `a++` sind arithmetisch äquivalent zu `a=a+1`
- Zusätzlich aber Auswertung von Variable `a`
- Präinkrement `++a`
  - Erst erhöhen, dann auswerten
- Postinkrement `a++`
  - Erst auswerten, dann erhöhen

```
 1  #include <stdio.h>
 2
 3  main() {
 4    int a = 0;
 5    int b = 43;
 6
 7    printf("1) a=%d, b=%d\n",a,b);
 8
 9    b = a++;
10    printf("2) a=%d, b=%d\n",a,b);
11
12    b = ++a;
13    printf("3) a=%d, b=%d\n",a,b);
14  }
```

- Output:

```
1) a=0, b=43
2) a=1, b=0
3) a=2, b=2
```


Operatoren ++ und --

- Analog zu `a++` und `++a` gibt es
  - Prädekrement `--a`
    - Erst verringern, dann auswerten
  - Postdekrement `a--`
    - Erst auswerten, dann verringern
- Beachte Unterschied in Bedingungsschleife!

```
 1  #include <stdio.h>
 2
 3  main() {
 4    int counter = 5;
 5
 6    while (--counter>0) {
 7      printf("%d ",counter);
 8    }
 9    printf("\n");
10  }
```

- Output: `4 3 2 1` (für `--counter` in 6)
- Output: `4 3 2 1 0` (für `counter--` in 6)

Bisektionsverfahren (Wh)

- Gegeben: stetiges $f : [a, b] \to \mathbb{R}$ mit $f(a)f(b) \leq 0$
  - Toleranz $\tau > 0$
- Tatsache: Zwischenwertsatz $\Rightarrow$ mind. eine Nst
  - denn $f(a)$ und $f(b)$ haben versch. Vorzeichen
- Gesucht: $x_0 \in [a, b]$ mit folgender Eigenschaft
  - $\exists \widetilde{x}_0 \in [a, b] \quad f(\widetilde{x}_0) = 0$ und $|x_0 - \widetilde{x}_0| \leq \tau$
- Bisektionsverfahren = iterierte Intervallhalbierung
  - Solange Intervallbreite $|b - a| > 2\tau$
    - Berechne Intervallmittelpunkt m und $f(m)$
    - Falls $f(a)f(m) \leq 0$, betrachte Intervall $[a, m]$
    - sonst betrachte halbiertes Intervall $[m, b]$
  - $x_0 := m$ ist schließlich gesuchte Approximation
- Verfahren basiert nur auf Zwischenwertsatz
- terminiert nach endlich vielen Schritten, da jeweils Intervall halbiert wird
- Konvergenz gegen Nst. $\widetilde{x}_0$ für $\tau = 0$.

Bisektionsverfahren

```
 1  #include <stdio.h>
 2  #include <math.h>
 3
 4  double f(double x) {
 5    return x*x + exp(x) -2;
 6  }
 7
 8  double bisection(double a, double b, double tol){
 9    double fa = f(a);
10    double m = 0.5*(a+b);
11    double fm = 0;
12
13    while ( b - a > 2*tol ) {
14      m = 0.5*(a+b);
15      fm = f(m);
16      if ( fa*fm <= 0 ) {
17        b = m;
18      }
19      else {
20        a = m;
21        fa = fm;
22      }
23    }
24    return m;
25  }
26
27  main() {
28    double a = 0;
29    double b = 10;
30    double tol = 1e-12;
31    double x = bisection(a,b,tol);
32
33    printf("Nullstelle x=%g\n",x);
34    printf("Funktionswert f(x)=%g\n",f(x));
35  }
```


Euklids Algorithmus

- Gegeben: zwei ganze Zahlen $a, b \in \mathbb{N}$
- Gesucht: größter gemeinsamer Teiler $ggT(a,b) \in \mathbb{N}$

- Euklidischer Algorithmus:
  - Falls $a = b$, gilt $ggT(a,b) = a$
  - Vertausche a und b, falls $a < b$
  - Dann gilt $ggT(a,b) = ggT(a-b,b)$, denn:
    - Sei g Teiler von a, b
    - d.h. $ga_0 = a$ und $gb_0 = b$ mit $a_0, b_0 \in \mathbb{N}$, $g \in \mathbb{N}$
    - also $g(a_0 - b_0) = a - b$ und $a_0 - b_0 \in \mathbb{N}$
    - d.h. g teilt b und $a - b$
    - d.h. $ggT(a,b) \leq ggT(a-b,b)$
    - analog $ggT(a-b,b) \leq ggT(a,b)$
  - Ersetze a durch $a - b$, wiederhole diese Schritte

- Erhalte $ggT(a,b)$ nach endlich vielen Schritten:
  - Falls $a \neq b$, wird also $n := \max\{a,b\} \in \mathbb{N}$ pro Schritt um mindestens 1 kleiner
  - Nach endl. Schritten gilt also nicht mehr $a \neq b$

Euklids Algorithmus

```
 1  #include <stdio.h>
 2
 3  main() {
 4    int a = 200;
 5    int b = 110;
 6    int tmp = 0;
 7
 8    printf("ggT(%d,%d)=",a,b);
 9
10    while (a != 0) {
11      if ( a < b) {
12        tmp = a;
13        a = b;
14        b = tmp;
15      }
16      a = a-b;
17    }
18
19    printf("%d\n",b);
20  }
```

- berechnet ggT von $a, b \in \mathbb{N}$
- basiert auf $ggT(a,b) = ggT(a-b,b)$ für $a > b$
- Für $a = b$ gilt $ggT(a,b) = b$ und $a - b = 0$

- Output:

```
ggT(200,110)=10
```


Euklids Algorithmus (verbessert)

```
 1  #include <stdio.h>
 2
 3  main() {
 4    int a = 200;
 5    int b = 2110;
 6    int tmp = 0;
 7
 8    printf("ggT(%d,%d)=",a,b);
 9
10    while (b != 0) {
11      tmp = b;
12      b = a%b;
13      a = tmp;
14    }
15
16    printf("%d\n",a);
17  }
```

- Euklids Algorithmus berechnet ggT von $a, b \in \mathbb{N}$
- basiert auf $ggT(a,b) = ggT(a-b,b)$ für $a > b$

- Für $a < b$ vertausche a und b
  - Divsionsrest von a/b ist $a\%b = a$
- Für $a > b$ iteriere $a := a - b$ bis $a < b$
  - d.h. bis $a = a\%b$
  - dann vertausche a und b
- Für $a = b$ gilt $ggT(a,b) = a$ und $a\%b = 0$

Die do-while-Schleife

- Formal: `do statement while(condition)`
- *Nach* jedem Durchlauf wird `condition` geprüft & Abbruch, falls nicht erfüllt
  - sog. fußgesteuerte Schleife
- Also *mindestens ein* Durchlauf!
- `statement` kann Block sein

```
 1  #include <stdio.h>
 2
 3  main() {
 4    int counter = 5;
 5
 6    do {
 7      printf("%d ",counter);
 8    }
 9    while (--counter>0);
10    printf("\n");
11  }
```

- Output:

  ```
  5 4 3 2 1
  ```

- `counter--` in 9 liefert Output: `5 4 3 2 1 0`

Ein weiteres Beispiel

```
 1  #include <stdio.h>
 2
 3  main() {
 4    int x[2] = {0,1};
 5    int tmp = 0;
 6    int c = 0;
 7
 8    printf("c=");
 9    scanf("%d",&c);
10
11    printf("%d %d ",x[0],x[1]);
12
13    do {
14      tmp = x[0]+x[1];
15      x[0] = x[1];
16      x[1] = tmp;
17      printf("%d ",tmp);
18    }
19    while(tmp<c);
20
21    printf("\n");
22  }
```

- Fibonacci-Folge strebt gegen unendlich
  - $x_0 := 0$, $x_1 := 1$ und $x_{n+1} := x_{n-1} + x_n$ für $n \in \mathbb{N}$
- Ziel: Berechne erstes Folgenglied mit $x_n > c$ für gegebene Schranke $c \in \mathbb{N}$
- für Eingabe $c = 1000$ erhalte Output:

  ```
  c=1000
  0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
  ```