Komplexität von Algorithmen und O-Notation

Zeitaufwand

Finden der max. Abschnittssumme in Array von n Zahlen

→ Teilfolge von aufeinanderfolg. Zahlen, die unter allen möglichen Teilfolgen die größte Summe liefert

→ Hat man z.B. die folgende Zahlenreihe:

-59, 52, 46, 14, -50, 58, -87, -77, 34, 15

so liefert folg. Teil-Zahlenfolge max. Abschnittssumme:

52 + 46 + 14 + -50 + 58 = 120

→ z. B. benötigt, um bestimmte grafische Muster zu erkennen oder aber zur Analyse von Aktienkursen, wo man Börsenkurse nachträglich untersucht, um einen besten Einkaufs- und Verkaufstag zu finden, so dass man einen maximalen Gewinn erzielt hätte.

Komplexität von Algorithmen und O-Notation

Zeitaufwand (Kubischer Algorithmus)

```
int maxfolge1(int z[], int n) {
    int i, j, k, sum, max = -10000000;
     for (i = 0; i < n; i++)
          for (j = i; j < n; j++) {
               sum = 0;
               for (k = i; k <= j; k++)
                    sum += z[k];
               if (sum > max)
                    max = sum;
          }
     return max;
}
```

Aufgrund der 3 ineinander geschachtelten for-Schleifen:

Zeitbedarf der Funktion maxfolge1() in etwa proportional zu n^3

Komplexität von Algorithmen und O-Notation

Zeitaufwand (Quadratischer Algorithmus)

```
int maxfolge2(int z[], int n) {
    int i, j, sum, max = -10000000;
     for (i=0; i<n; i++) {
        sum = 0;
        for (j=i; j<n; j++) {
            sum += z[j];
             if (sum > max)
                max = sum;
        }
     }
     return max;
}
```

Zugriff auf bereits berechnete Summe S(i,j-1)

→ **S(i,j) = S(i,j-1) + z[j]**

→ **spart äußere Schleife**

Aufgrund der 2 ineinander geschachtelten for-Schleifen:

Zeitbedarf der Funktion maxfolge2() in etwa proportional zu n^2

Komplexität von Algorithmen und O-Notation

Zeitaufwand (Linearer Algorithmus = prop. zu n (Optimum))

```
int maxfolge3(int z[], int n) {
    int i, s, gesamtmax = -10000000, endesumme = 0;
    for (i=0; i<n; i++) {
        endesumme = ((s=endesumme+z[i]) > 0) ? s : 0;
        if (endesumme > gesamtmax)
            gesamtmax = endesumme;
    }
    return gesamtmax;
}
```

gesamtmax = bisher max. Abschnittsumme
endesumme = Absch.summe akt. Teilstücks

Wird endesumme durch Addieren nächst. Zahl > gesamtmax
→ gesamtmax = endesumme
Wird endesumme durch Addieren nächst. Zahl negativ, wird dieser Wert 0 zugewiesen

Komplexität von Algorithmen und O-Notation

Bei Problemgröße n = 10 000 gilt:

`maxfolge1()`: $t(n) = 10000^3 = 10^{4^3} = 10^{12} = 1$ Billion

`maxfolge2()`: $t(n) = 10000^2 = 10^{4^2} = 10^8 = 100$ Millionen
also 10000 mal schneller als `maxfolge1()`

`maxfolge3()`: $t(n) = 10000^1 = 10^4$
also 10000 mal schneller als `maxfolge2()` und
100 Millionen mal schneller als `maxfolge1()`.

Zahl t(n) ist die Zeitkomplexität des jeweiligen Algorithmus

...... 594.880 Sek.; maxfolge1 −−> 359995
...... 0.210 Sek.; maxfolge2 −−> 359995
...... 0.000 Sek.; maxfolge3 −−> 359995

Komplexität von Algorithmen und O-Notation

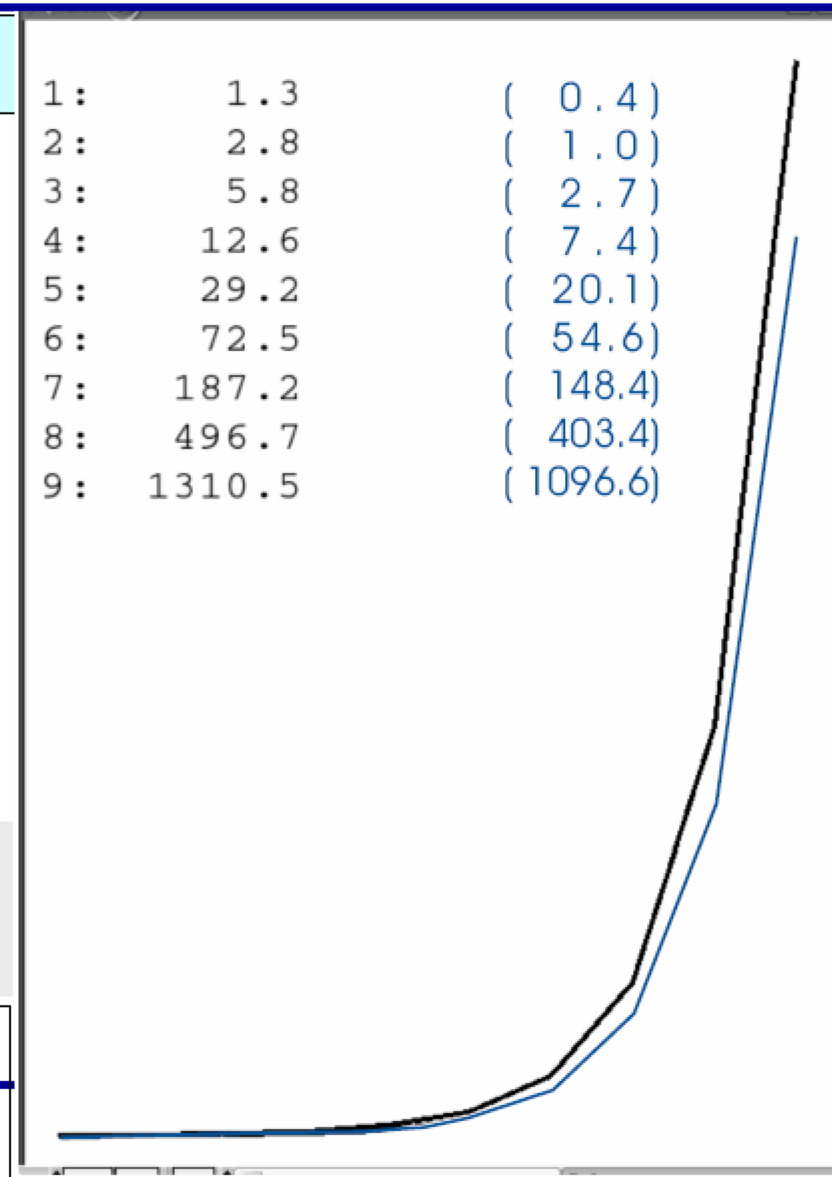
Exponentieller Algorithmus

```
int prim(int zahl, int teiler) {
    if (zahl < 2 || zahl%2 == 0 || zahl%teiler == 0)
        return 0; /* keine Primzahl */
    else if (teiler*teiler > zahl)
        return 1; /* Primzahl */
    return prim(zahl, teiler+1);
}
```

prim() testet rekursiv, ob Zahl eine Primzahl → Aufruf prim(zahl, 2)

Komplexität von Algorithmen und O-Notation

Exponentieller Algorithmus



prim()-Aufrufe abhängig von der Stellenzahl

Komplexität von Algorithmen und O-Notation

Speicherplatzbedarf

Neben dem Zeitaufwand ist der Speicherplatzbedarf für die Bestimmung der Komplexität von Algorithmen wichtig.

→ Anhand zweier Algorithmen zur Lösung des gleichen Problems wird hier gezeigt, wie sich die Wahl eines Algorithmus auf Speicherplatzbedarf auswirken kann:

> Man habe in Array der Größe n ganze Zahlen aus dem Intervall [0, n−1] gespeichert. Die Aufgabe ist es nun, festzustellen, ob keine Zahl doppelt im Array vorkommt.

Komplexität von Algorithmen und O-Notation

Speicherplatzbedarf (Quadratischer Algorithmus)

```
int doppcheck1(int z[]) {
    for (int i=0; i<MAX-1; i++)
        for (int j=i+1; j<MAX; j++)
            if (z[i] == z[j])
                return 1;
    return 0;
}
```

Benötigt wenig Speicherplatz: nur ein Array mit n Zahlen. Da er jedoch zwei ineinander geschachtelte Schleifen hat, ist seine Laufzeit proportional zu n^2 (quad. Algorithmus), während Speicherplatzbedarf in etwa proportional zu n ist.

Komplexität von Algorithmen und O-Notation

Speicherplatzbedarf (Linearer Algorithmus)

```
int hilf[MAXZAHL] = {0};
 ....
int doppcheck2(int z[]) {
    for (int i=0; i<MAX; i++)
        if (hilf[z[i]] != 0)
            return 1;
        else
            hilf[z[i]] = 1;
    return 0;
}
```

Zeitkomplexität ist nun proportional zu n.

Speicherplatzbedarf (durch zusätzl. Hilfsarray) nimmt abh. von größtmögl. Zahl im Array exponentiell zu.

Problemgröße m = Bitzahl zum Speichern der größten Zahl
→ Platzkomplexität s(n) in etwa proportional zu 2^m

Lineare Zeitkompl. durch exponentielle Platzkompl. erkauft.

Komplexität von Algorithmen und O-Notation

Klassifikation von Algorithmen

1	*konstant*	Jede Anweisung eines Programms wird höchstens einmal ausgeführt. Dies ist der Idealzustand für einen Algorithmus.
log *n*	*logarithmisch*	Speicher- oder Zeitverbrauch wachsen nur mit der Problemgröße *n*. Die Basis des Logarithmus wird häufig 2 sein, d. h. vierfache Datenmenge verursacht doppelten Ressourcenverbrauch, 8-fache Datenmenge verursacht 3-fachen Verbrauch und 1024-fache Datenmenge 10-fachen Verbrauch.
n	*linear*	Speicher- oder Zeitverbrauch wachsen direkt proportional mit der Problemgröße *n*.

Komplexität von Algorithmen und O-Notation

Klassifikation von Algorithmen

$n \log n$	$n \log n$	Der Ressourcenverbrauch liegt zwischen n (*linear*) und n^2 (*quadratisch*).
n^2	*quadratisch*	Speicher- oder Zeitverbrauch wachsen quadratisch mit der Problemgröße. Solche Algorithmen lassen sich praktisch nur für kleine Probleme anwenden.
n^3	*kubisch*	Speicher- oder Zeitverbrauch wachsen kubisch mit der Problemgröße. Solche Algorithmen lassen sich in der Praxis nur für sehr kleine Problemgrößen anwenden.
2^n	*exponentiell*	Bei doppelter, dreifacher und 10-facher Datenmenge steigt der Ressourcenverbrauch auf das 4-, 8- bzw. 1024-fache. Solche Algorithmen sind praktisch kaum verwendbar.

Komplexität von Algorithmen und O-Notation

Klassifikation von Algorithmen

- Eine Komplexitätsfunktion n^k, die sich asymptotisch wie ein Polynom vom Grad k verhält, nennt man **polynomial.**
- Noch stärker als exponentiell nimmt die Komplexität der Fakultätsfunktion n! zu (superexponentiell).

Neben diesen grundlegenden Komplexitätsfunktionen existieren weitere Zwischenformen, wie z.B.:

$n^{\frac{3}{2}}$	Algorithmus mit n^2 Speicher- und n^3 Zeitverbrauch (bei großem n näher bei $n \log n$ als bei n^2).
$n \cdot \log^2 n$	Algorithmus, der ein Problem zweistufig in Teilprobleme zerlegt (bei großem n näher bei $n \log n$ als bei n^2).

Komplexität von Algorithmen und O-Notation

Klassifikation von Algorithmen

n	$\lg n$	$\lg^2 n$	$\sqrt{n}$	$n \lg n$	$n \lg^2 n$	$n^{\frac{3}{2}}$	n^2
10	3	9	3	30	90	32	100
100	6	36	10	600	3600	1000	10 000
1000	9	81	32	9000	81 000	31 623	1 000 000
10 000	13	169	100	130 000	1 690 000	1 000 000	100 000 000
100 000	16	256	316	1 600 000	25 600 000	31 622 777	10 Milliarden
1 000 000	19	361	1000	19 000 000	361 000 000	1 Milliarde	1 Billion

Komplexität von Algorithmen und O-Notation

Klassifikation von Algorithmen

Unterschiedl. Komplexitätsfunktionen → wie lange entspr. Progr. dauert, wenn pro Sek. 1 Mio. Operationen ausgeführt:

Gib ein n ein: **1000000**

	Jahre	Tage	Std	Min	Sek	
lg n	0	0	0	0	0	19
lg^2 n	0	0	0	0	0	361
sqrt n	0	0	0	0	0	1000
n lg n	0	0	0	0	19	19000000
n lg^2 n	0	0	0	6	1	361000000
n^(3/2)	0	0	0	16	40	1000000000
n^2	0	11	13	46	40	1000000000000
n^3	31709	289	1	46	40	1000000000000000000

Die O-Notation

Sequenzielle Suche (O(n)-Algorithmus)

```
int seqsuche(int z[], int n, int zahl) {
    int i;
    for (i=0; i<n; i++)
        if (z[i] == zahl)
            return i; /* Zahl gefunden */
    return -1; /* Zahl nicht gefunden */
}
```

1. Wir konzentrieren uns auf die entscheidende Operation, was hier der Vergleich **z[i] == zahl.**

2. Wir betrachten nur den ungünstigsten, den günstigsten und den mittleren (durchschnittlichen) Fall.

3. Wir streichen schließl. alle additiven und multiplikativen Konstanten, um O-Notation für Algorithmus zu erhalten.

Die O-Notation

Sequenzielle Suche (O(n)-Algorithmus)

1. **Günstigster Fall:** Gesuchte Zahl befindet sich an erster Position im Array → nur ein Vergleich
2. **Ungünstigster Fall:** Gesuchte Zahl befindet sich an letzter Position im Array → n Vergleiche
3. **Durchschnittlicher Fall:** Nimmt man Gleichverteilung an, ist gesuchte Zahl mit jeweils gleicher Wahrscheinlichkeit p an jeder möglichen Stelle im Array
   → Zusätzl. W'keit q, dass Zahl überhaupt nicht im Array
   → $np + q = 1$ → $p = (1-q) / n$

$$t(n) = (1p + 2p + 3p + \ldots + np) + nq$$
$$= (1 + 2 + 3 + \ldots + n) \cdot p + nq$$

$$\boxed{t(n) = \frac{n \cdot (n+1)}{2} p + nq}$$

Die O-Notation

Sequenzielle Suche (O(n)-Algorithmus)

1. Gesuchte Zahl im Array: **$t(n) \sim (n+1)/2 \rightarrow t(n) = O(n)$**

   Hier gilt für die Wahrscheinlichkeiten: $q = 0$ und $p = 1/n$

$$t(n) = \frac{n+1}{2} = \frac{n}{2} + \frac{1}{2} = n \cdot \frac{1}{2} + \frac{1}{2}$$

   Durch Streichen der additiven Konstante (1/2) und der multiplikativen Konstante (1/2) erhalten wir: $t(n) = O(n)$.

2. Zahl nicht im Array: **$t(n) \sim (n+1)/4 + n/2 \rightarrow t(n) = O(n)$**

   Hier z.B. W'keit $q = ½$, $p = \frac{1-q}{n} = \frac{1-\frac{1}{2}}{n} = \frac{1}{2n}$

$$\boxed{t(n) = \frac{n \cdot (n+1)}{2} p + nq} \rightarrow \boxed{t(n) = \frac{n+1}{4} + \frac{n}{2} = n \cdot \frac{1}{4} + \frac{1}{4} + n \cdot \frac{1}{2}}$$

Streichen additiver und multipl. Konst. $t(n) = 2n \rightarrow t(n) = O(n)$

Die O-Notation

Schnelles Potenzieren nach Legendre: O(log n)

```
x = a; y = b; z = 1;
while (y>0) {
    if (y ungerade)
        z = z*x
    y = y/2
    x = x*x
}
/* z = Potenz von a hoch b */
```

Potenz 4^7 wird in Produkt folg. Exponenten zerlegt: $4^1 \cdot 4^2 \cdot 4^4$

→ Exponent 7 wird in Summe von 2er-Potenzen umgewandelt

→ Pro Schleifendurchlauf wird n (hier Exponent) halbiert, bis Grenze 1 erreicht:

$$\boxed{\frac{n}{2^x} = 1 \rightarrow x = \log_2 n}$$

→ Legendre-Algorithmus hat Größenordnung O($\log_2$ n)

$$\boxed{1 \cdot 4^7 = 4 \cdot 4^6 = 4 \cdot (4^2)^3 = 4 \cdot 16^3 = 4 \cdot 16 \cdot (16^2) = 4 \cdot 16 \cdot 256^1 = 16384 \cdot 256^0}$$

Die O-Notation

Teilersuche (kein wirklicher O(√*n*)-Algorithmus)

```
void teilsuch(int zahl) {
    int t, w = (int)sqrt(zahl);
    for (t=1; t<w; t++)
        if (zahl%t == 0)
            Ausgabe: t und zahl/t;
    if (zahl%w==0)
        Ausgabe: w;
}
```

Ermittelt alle Teiler zu Zahl

→ √n Divisionen, da bei ermittelt. Teiler nicht nur Teiler selbst, sondern auch komplementärer Teiler (Zahl geteilt durch Teiler)

→scheint Zeitkompl.: t(n) = √n Vorsicht, denn bei Komplexität am Anwachsen der Stellenzahl

→ exponentieller Algorithmus

→ n = d Dezimalstellen:

$$\boxed{O(\sqrt{n}) = O(\sqrt{10^d}) = O(3.162278^d)}$$

Die O-Notation

Primzahlen mit dem Sieb des Eratosthenes $O(n^2)$

1	2	3	$\cancel{4}$	5	$\cancel{6}$	7	$\cancel{8}$	9	$\cancel{10}$	11	$\cancel{12}$	13	$\cancel{14}$	15	$\cancel{16}$	17	$\cancel{18}$	......
1	2	3	$\cancel{4}$	5	$\cancel{6}$	7	$\cancel{8}$	$\cancel{9}$	$\cancel{10}$	11	$\cancel{12}$	13	$\cancel{14}$	$\cancel{15}$	$\cancel{16}$	17	$\cancel{18}$	......

```
Eingabe: Zahl n
for (i=1 ; i<=n ; i++) /*   prim[0] wird nicht benutzt        */
   prim[i] = 1; /* zunaechst alle Zahlen erst mal Primzahlen */
for (i=2 ; i<=n/2 ; i++)
   if (prim[i])    /*  entspricht  if (prim[i]!=0)       */
      for (j=2*i ; j<=n ; j = j+i)
         prim[j] = 0;
Ausgabe: Die Primzahlen von 1 bis n sind:
for (i=2 ; i<=n ; ++i)
   if (prim[i])
      Ausgabe: i
```

aufgrund 2er geschachtelter for-Schleifen → $O(n^2)$

lässt sich optimieren
→ nur ungerade Zahlen und äußere Schleife nur bis $\sqrt{n}$
→ Größenordnung $O(n^{3/2})$

Die O-Notation

Primzahlensieb mit einem O(n log n)-Algorithmus

```
m = n/2−3; /∗ n = Primzahlen bis zu dieser oberen Grenze ∗/
print 2 /∗ einzige gerade Primzahl ∗/
for (i=0; i<=m; i++)
    prim[i] = 1;
for (i=0; i<=m; i++) {
    if (prim[i]) { /∗ prim[i] ist genau dann 1, wenn 2i+3 Primzahl ist ∗/
        p = i+i+3;
        k = i+p;
        z++;
        print p;
        while (k<=m) {
            prim[k] = 0;
            k += p;
        }
    }
}
print 'z' Primzahlen gefunden
```

Wahl eines Algorithmus

- **Kosten-/Nutzen-Analyse**
  Algorithmus, der nur wenige Male verwendet wird, muss eventuell nicht optimal sein → hier Zeitaufwand zur Verbesserung des Algorithmus abwägen.
- **Bessere Algorithmen nicht immer wesentlich komplizierter.**
- **Auffinden von „Flaschenhälsen“**
  reicht oft aus, indem man diese optimiert, damit dann das gesamte Programm in akzeptabler Zeit Ergebnisse liefert.
- **Messen mit typischen Eingabedaten**
  Einfacher Sortieralgorithmus evtl. weniger Zeit und Speicher, wenn Daten bereits weitgehend sortiert sind, während ein theoret. schneller und komplizierter Algorithmus wesentlich mehr Speicherplatz benötigt und evtl. sogar langsamer ist.

Einfache Optimierungen bei der Implementierung

Konstanten statt Variablen (constant propagation)

```
/* ohne Optimierung */
for (i=1; i  <= 1000000000; i++) {
    x = 2;
    y = x + 5;
    a = x;
    b = 0;
    c = a / x;
    d = x*c*c*c*c;
    e = x+b+b+b;
}
```

```
/* mit Optimierung */
for (i=1; i<= 1000000000; i++) {
    x = 2;
    y = 7;

    b = 0;
    c = 1;

    a = d = e = x;
}
```

... ohne Optimierung: 52.00
.... mit Optimierung: 9.17

Einfache Optimierungen bei der Implementierung

Einmalige Berechnung gleicher Ausdrücke

```
/* ohne Optimierung */
for (i=1; i<=100000000; i++) {
    x = (a+b+sqrt(a-b))/3;
    y = (a+b-sqrt(a-b))/3;


    f = r*r*4*atan(1);
    u = 2*r*4*atan(1);
}
```

```
/* mit Optimierung */
for (i=1; i<=100000000; i++) {
    c = a+b;
    d = sqrt(a-b);
    x = (c+d)/3;
    y = (c-d)/3;
    f = r*z;
    u = 2*z;
}
```

... ohne Optimierung: 26.16
.... mit Optimierung: 6.52

Einfache Optimierungen bei der Implementierung

Keine überflüssigen Funktionsaufrufe

```
/* ohne Optimierung */
for (i=1; i<=100000000; i++) {
    x = ceil(pow(2.17, 2));
    x = a/8;
    c = b*16;
    if (d%2 != 0)
        e = x+3;
}
```

```
/* mit Optimierung */
for (i=1; i<=100000000; i++) {
    y = 2.17*2.17;    x = y + 0.5;
    x = a >> 3;
    c = b << 4;
    if (d&1)
        e = x+3;
}
```

... ohne Optimierung: 16.11
.... mit Optimierung: 2.88

Einfache Optimierungen bei der Implementierung

Keine überflüssigen oder doppelten Berechnungen

```
/* ohne Optimierung */
for (i=1; i<=1000000000; i++) {
    a = x*y;
    b = x;    c = b*y;    d = x*y;
}
```

```
/* mit Optimierung */
for (i=1; i<=1000000000; i++) {
    b = x;
    a = c = d = x*y;
}
```

... ohne Optimierung: 9.44
.... mit Optimierung: 8.90

Hier hat Optimierer des Compilers schon einiges optimiert.

Einfache Optimierungen bei der Implementierung

Vermeiden überflüssiger Schleifendurchläufe

```
/* ohne Optimierung */
for (i=1; i<=100000; i++)
    for (j=0; j<=1000; j++)
        if (j%10==0) array[j] = j;
```

```
/* mit Optimierung */
for (i=1; i<=100000; i++)
    for (j=0; j<=1000; j+=10)
        array[j] = j;
```

... ohne Optimierung: 16.65
.... mit Optimierung: 0.56

Einfache Optimierungen bei der Implementierung

Entfernen invarianter Ausdrücke aus Schleifen

```
/* ohne Optimierung */
for (i=1; i<=10000; i++) {
    for (j=1; j<=1000; j++)
        array[j] = a+b*sin(2.33);

    for (j=0; j<strlen(string)-sqrt(h); j++)
        string[j] = '-';
}
```

```
/* mit Optimierung */
x = a+b*sin(2.33);
for (i=1; i<=10000; i++) {
    for (j=1; j<=1000; j++)
        array[j] = x;
    l = strlen(string)-sqrt(h);
    for (j=0; j<l; j++)
        string[j] = '-';
}
```

... ohne Optimierung: 11.03
.... mit Optimierung: 0.28

Einfache Optimierungen bei der Implementierung

Zusammenfassen mehrerer Schleifen zu einer

```
/* ohne Optimierung */
for (i=1; i<=10000000; i++) {
    for (j=0; j<=1000; j++)
        a[j] = j;
    for (j=0; j<=1000; j++)
        b[j] = a[j] + x;
    for (j=0; j<=1000; j++)
        c[j] = a[j];
}
```

```
/* mit Optimierung */
for (i=1; i<=10000000; i++) {
    for (j=0; j<=1000; j++) {
        a[j] = j;
        b[j] = j + x;
        c[j] = j;
    }
}
```

... ohne Optimierung: 89.37
.... mit Optimierung: 62.86