Algorithmen und Komplexität

Prof. Dr. Raphael Herding

Laufzeitanalyse

- Bisher haben wir zur Lösung spezieller Programmieraufgaben immer den erstbesten Algorithmus genommen und implementiert.
- Dabei haben wir gesehen, dass es Algorithmen sehr unterschiedlicher Leistungsfähigkeit geben kann.
- Zu Vertiefung nutzen wie für die gleiche Aufgabe drei unterschiedliche Algorithmen.

- Wir wollen alle ganzzahligen, nicht-negativen Lösungen der Gleichung

$$x + y + z = n$$

bestimmen. Die Zahl n ist dabei beliebig, aber fest vorgegeben.

Laufzeitanalyse

- **Lösung 1**: Variablen x, y und z werden im gesamten Suchraum (0 bis n) variieren und prüfen für jede Variablenkombination eine zulässige Lösung. Hier n = 50.

	1	`void gleichung1( int n)`
		`{`
		`int x, y, z;`
	1	`for( x = 0; x <= n; x++)`
		`{`
A	51	`for( y = 0; y <= n; y++)`
		`{`
B	2601	`for( z = 0; z <= n; z++)`
		`{`
C	132651	`if( x + y + z == n)`
		`printf( "%d + %d + %d = %d\n", x, y, z, n);`
		`}`
		`}`
		`}`
		`}`

Laufzeitanalyse

	1	`void gleichung1( int n)`
		`{`
		`int x, y, z;`
	1	`for( x = 0; x <= n; x++)`
		`{`
A	51	`for( y = 0; y <= n; y++)`
		`{`
B	2601	`for( z = 0; z <= n; z++)`
		`{`
C	132651	`if( x + y + z == n)`
		`printf( "%d + %d + %d = %d\n", x, y, z, n);`

- Für n= 50 können konkrete Zahlenwerte angegeben werden.
- Für (A) gilt $(n+1)$ = 51,
- für (B) gilt $(n+1)^2$ = 2601
- für (C) gilt $(n+1)^3$ = 132651.

Laufzeitanalyse

- Uns interessieren jedoch mehr eine allgemeine Formel, anstelle von Einzelwerten.
- Da sich mit jeder Schleife die Zahl der untersuchten Fälle um den Faktor n + 1 vervielfacht, haben wir insgesamt $(n+1)^3$ Fälle zu untersuchen.
- Programm wird außerdem mit unbekannten Proportionalitätsfaktor c multipliziert. Faktor ist unterschiedlich pro Rechner.
- Die Laufzeit ist somit

$$tn(n) = c(n+1)^3.$$

	1	`void gleichung1( int n)`
		`{`
		`int x, y, z;`
	1	`for( x = 0; x <= n; x++)`
		`{`
A	51	`for( y = 0; y <= n; y++)`
		`{`
B	2601	`for( z = 0; z <= n; z++)`
		`{`
C	132651	`if( x + y + z == n)`
		`printf( "%d + %d + %d = %d\n", x, y, z, n);`

Laufzeitanalyse

- **Lösung 2**: Algorithmus als Lösung 1untersucht überflüssige Fälle.

$$x + y + z = n$$

- Wert z steht fest, sobald Werte für x und y feststehen → (z = n – x - y).
- Somit kann innere Schleife entfernt und Programm vereinfacht werden.

	1	`void gleichung2( int n)`
		`{`
		`int x, y, z;`
	1	`for( x = 0; x <= n; x++)`
		`{`
A	51	`for( y = 0; y <= n; y++)`
		`{`
B	2601	`z = n - x - y;`
	2601	`if( z >= 0)`
	1326	`printf( "%d + %d + %d = %d\n", x, y, z, n);`
		`}`
		`}`
		`}`

Laufzeitanalyse

- Anzahl der Fälle reduzieren sich deutlich auf (B)

  $(n+1)^2$.

- Laufzeit ist somit $t(n) = c(n+1)^2$.

	1	`void gleichung2( int n)`
		`{`
		`int x, y, z;`
	1	`for( x = 0; x <= n; x++)`
		`{`
A	51	`for( y = 0; y <= n; y++)`
		`{`
B	2601	`z = n - x - y;`
	2601	`if( z >= 0)`
	1326	`printf( "%d + %d + %d = %d\n", x, y, z, n);`
		`}`
		`}`
		`}`

Leistungsanalyse

- **Lösung 3:** y muss nicht immer wieder durch den kompletten Wertebereich (0-n) laufen.

$$x + y + z = n$$

- Oberhalb von y=n-x ist keine zulässige Lösung mehr für z.
- Wir können die Schleife demnach bei Überschreibung von n-x abbrechen.
- Da z=n-z-x in der Situation immer positiv, entfällt die Abfrage z => 0 ebenfalls.

	11	`void gleichung3( int n)`
		`{`
		`int x, y, z;`
	1	`for( x = 0; x <= n; x++)`
		`{`
A	51	`for( y = 0; y <= n-x; y++)`
		`{`
B	1326	`z = n - x - y;`
	1326	`printf( "%d + %d + %d = %d\n", x, y, z, n);`
		`}`
		`}`
		`}`

Laufzeitanalyse

- Jetzt sind (A) $(n+1)$ = 51 Durchläufe
- (B) $\frac{(n+1)(n+2)}{2}$ = 1326 Durchläufe

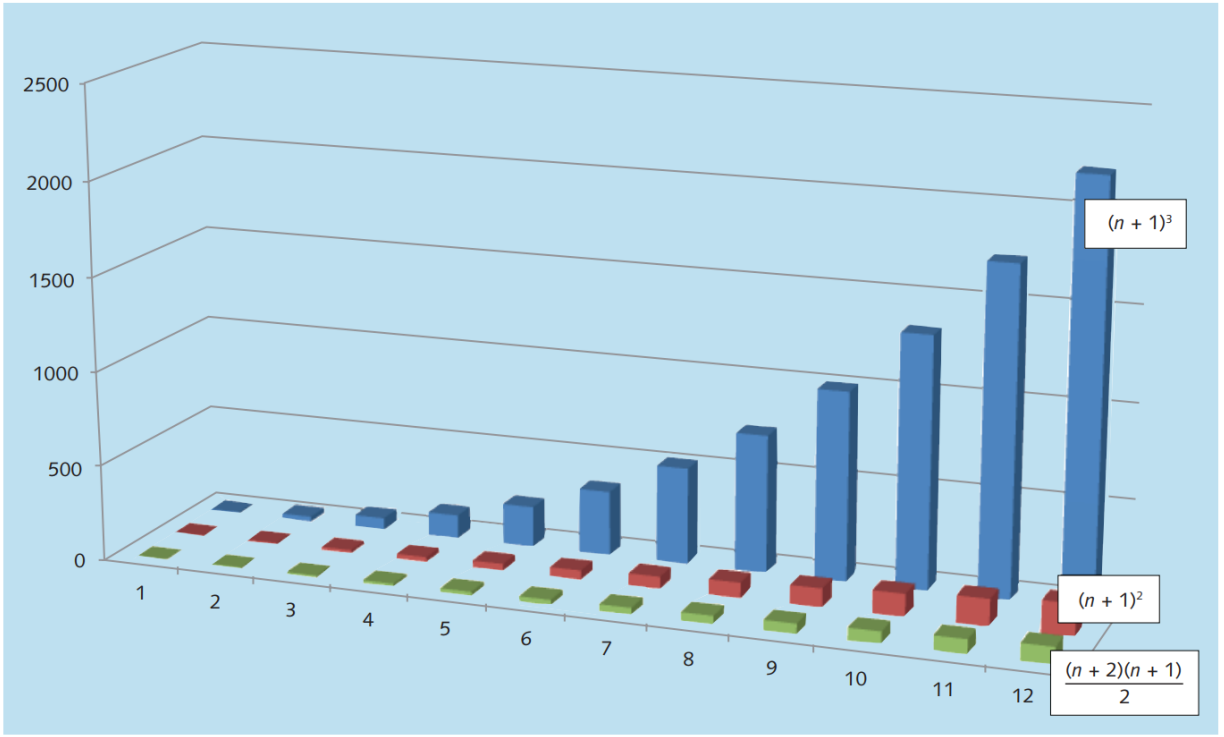
x	Möglichkeiten für y
0	n+1
1	n
…	…
n-1	2
n	1
Summe:	$\frac{(n+2)\,(n+1)}{2}$

- Laufzeit ist somit $t(n) = \frac{n(n+1)}{2}$.

```
    11      void gleichung3( int n)
            {
            int x, y, z;

    1       for( x = 0; x <= n; x++)
                {
A   51          for( y = 0; y <= n-x; y++)
                    {
B   1326            z = n - x - y;
    1326            printf( "%d + %d + %d = %d\n", x, y, z, n);
                    }
                }
            }
```

Laufzeitanalyse

- Erste Programm fällt für große n deutlich aus dem Rahmen.
- Die anderen Programme sind vergleichbar.
- Hier sind Unterschiedliche Leistungsklassen vorhanden.
- Wir benötigen eine systematische Analyse zur Bewertung der Leistungsfähigkeit.

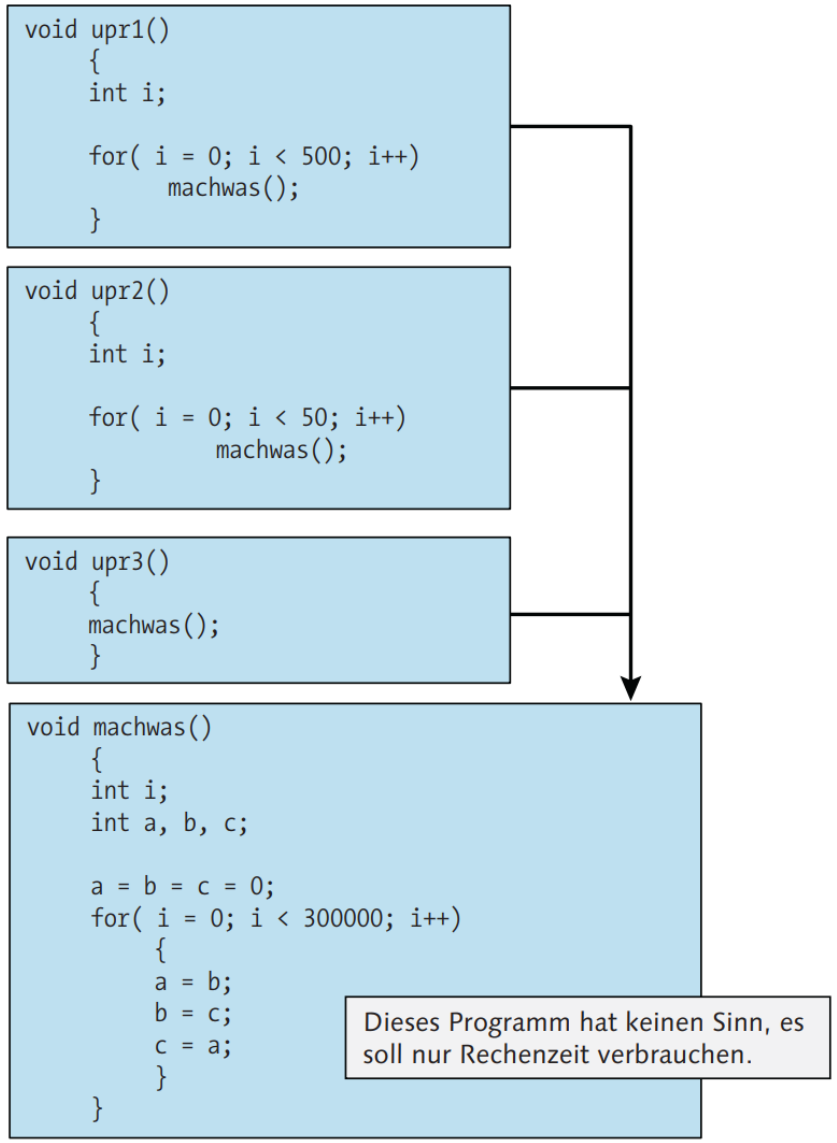


Leistungsanalyse
Einführung

- Die theoretische Analyse von Algorithmen ist ein anspruchsvolles Feld.
- Nur in einfachen Fällen kann ein Algorithmus vollständig rechnerisch bestimmt werden.
- Im Regelfall werden deshalb unwesentliche Teile ignoriert.
- Fokus auf „essenzielle“ Programmteile, die signifikante Laufzeit benötigen.
- Diese Programmteile zu ermitteln erfordert Übung.

- Dies sind zumeist:
  - Blöcke
  - Fallunterscheidungen
  - Schleifen
  - Unterprogramme

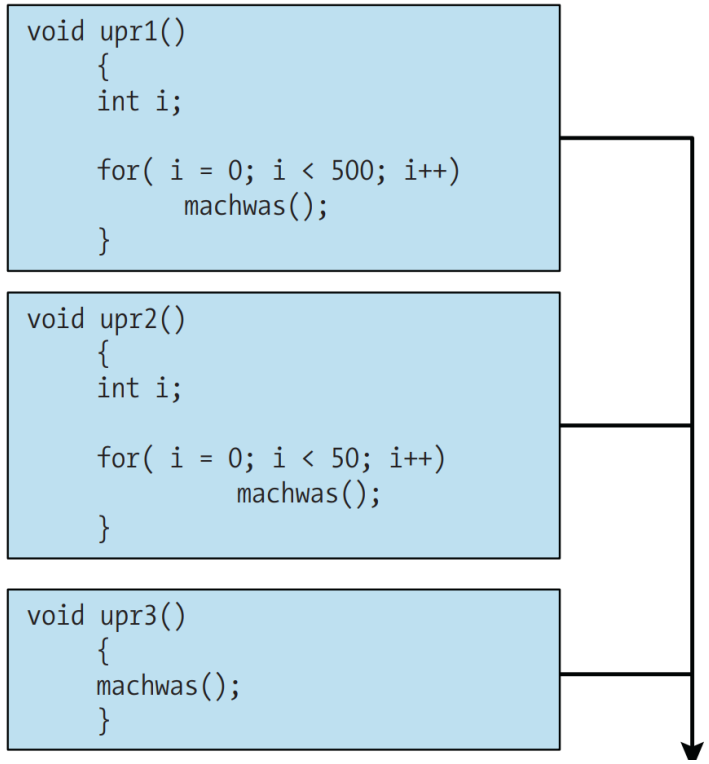
Leistungsanalyse
Beispiel mit konstanter Laufzeit



Leistungsanalyse
Beispiel mit konstanter Laufzeit

- Die drei Unterprogramme haben konstante Laufzeit.
- Vermutung:
  - upr2() → 50-fache Laufzeit zu upr3()
  - upr1() → 500-fache Laufzeit zu upr3()
  - Upr1()→ 10-fache Laufzeit von upr2()
- Die effektiven Laufzeiten (in min/h/Tage) sind aktuell nicht bekannt

  und nicht relevant.



Leistungsanalyse
Beispiel mit konstanter Laufzeit

- Der eigentliche Algorithmus, für dessen Laufzeitverhalten wir uns interessieren, ist durch das Unterprogramm test() gegeben.

```
void test(int n, int m)
    {
    int i1, i2, i3;

    for( i1 = 0; i1 < n; i1++)
        {
        upr1();
        for( i2 = 0; i2 < 2*m; i2++)
            {
            if( i2% 2)
                upr2();
            else
                for( i3 = 0; i3 < i2; i3++)
                    upr3();
            }
        }
    }
```

Leistungsanalyse
Beispiel mit konstanter Laufzeit

- Wir interessieren uns für die Laufzeit des Programms **test()**
- Die Laufzeit hängt von den Parametern **n** und **m** ab.
- Wir betrachten nur signifikante Teile (Blöcke, Bedingungen, Schleifen).
- Nicht signifikante Teile (Variablendeklaration, …) werden nicht betrachtet, da Sie kaum zur Laufzeiterhöhung beitragen.

```
void test(int n, int m)
    {
    int i1, i2, i3;

    for( i1 = 0; i1 < n; i1++)
        {
        upr1();
        for( i2 = 0; i2 < 2*m; i2++)
            {
            if( i2% 2)
                upr2();
            else
                for( i3 = 0; i3 < i2; i3++)
                    upr3();
            }
        }
    }
```

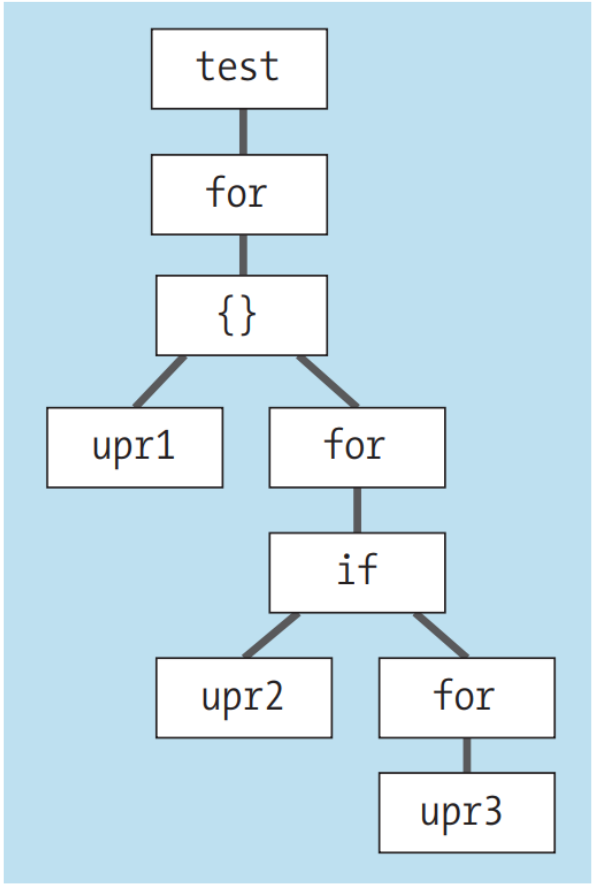
Leistungsanalyse
Beispiel mit konstanter Laufzeit

- Wir zerlegen den Algorithmus grafisch in die signifikanten

  Bestandteile.

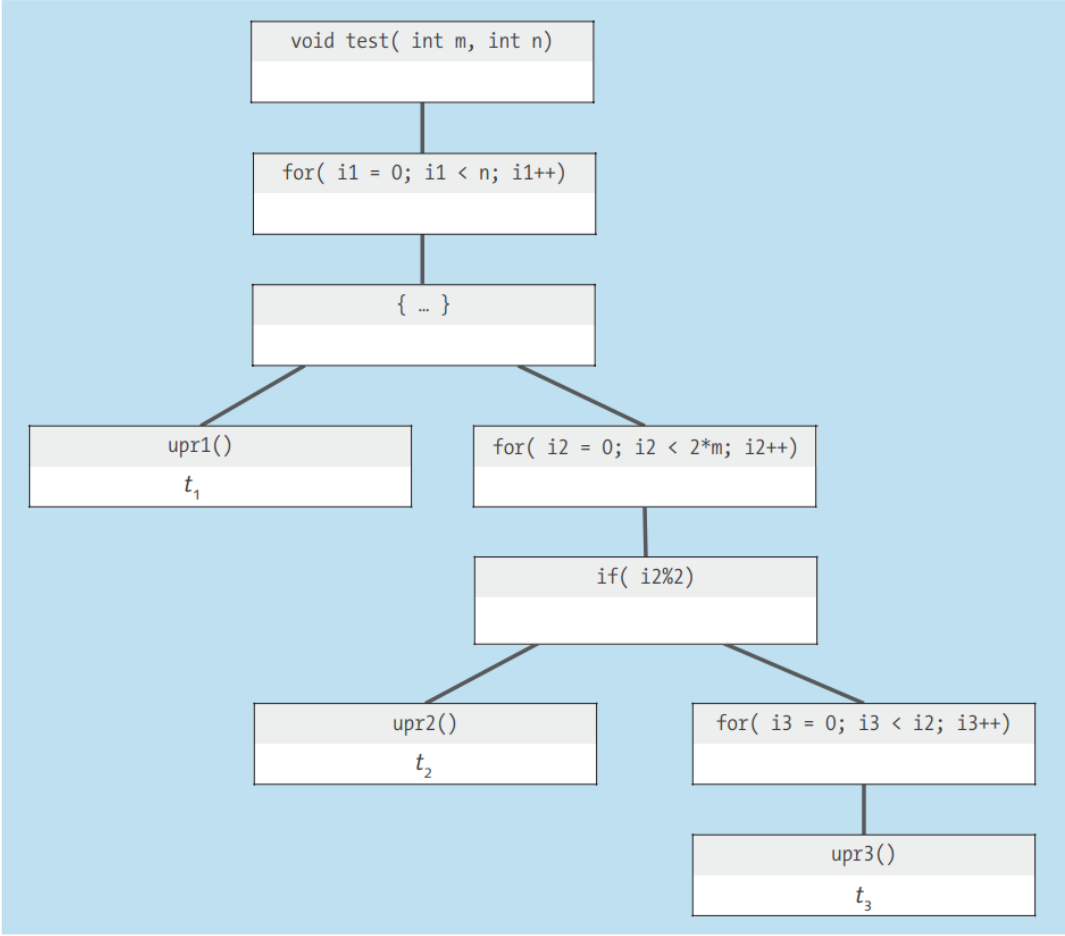
```
void test(int n, int m)
    {
    int i1, i2, i3;

    for( i1 = 0; i1 < n; i1++)
        {
        upr1();
        for( i2 = 0; i2 < 2*m; i2++)
            {
            if( i2 % 2)
                upr2();
            else
                for( i3 = 0; i3 < i2; i3++)
                    upr3();
            }
        }
    }
```

Unterprogramme werden nicht weiter zerlegt, da die Laufzeit konstant ist.



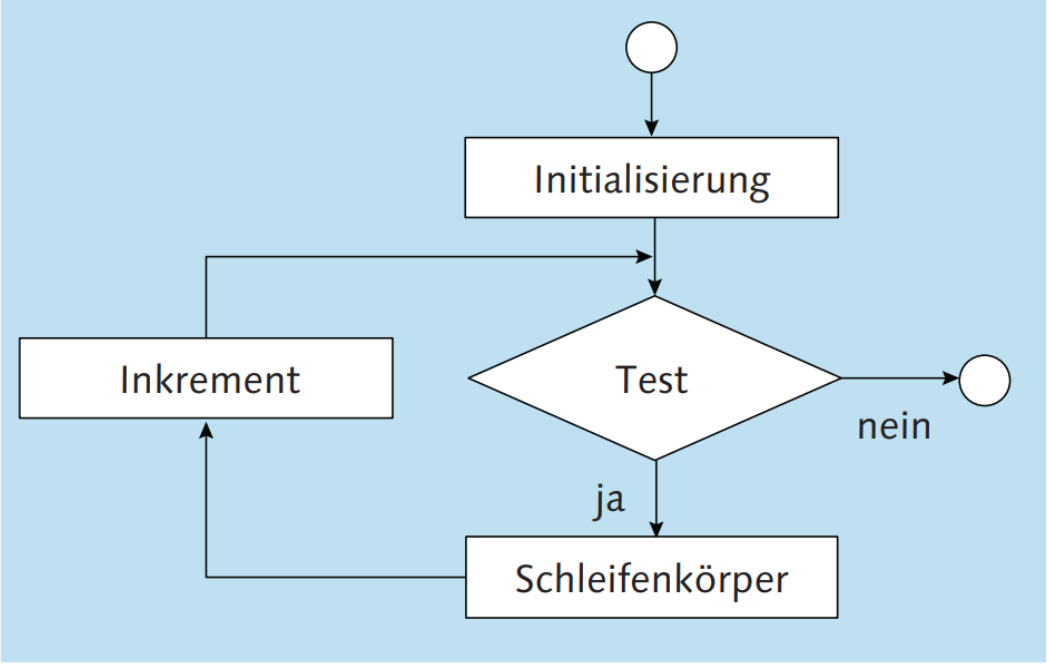
Leistungsanalyse
Beispiel mit konstanter Laufzeit

- Diese Laufzeiten werden im Baum über die Knoten nach oben propagiert.
- Die Unterprogramme erhalten die Variable (tx) für die konstante Laufzeit.



Leistungsanalyse
Beispiel mit konstanter Laufzeit

- Im Rahmen der Anwendungsprogrammierung wissen wir, dass eine Schleife aus verschiedenen Bestandteilen besteht.
- Diese gehen alle in die Laufzeitberechnung ein.
- Ein Initialisierung kann auch sehr lange dauern, selbiges gilt für die Testbedingung.




```
for( init; test; incr)
     body
```

Leistungsanalyse
Beispiel mit konstanter Laufzeit

- Die detaillierte Bestimmung ist wie folgt:

t_{init} die Laufzeit der Initialisierung

$t_{test}(k)$ die Laufzeit des Tests vor dem k-ten Schleifendurchlauf

$t_{body}(k)$ die Laufzeit des Schleifenkörpers im k-ten Durchlauf

$t_{incr}(k)$ die Laufzeit des Inkrements am Ende des k-ten Durchlaufs

Dann berechnet sich die Laufzeit der Schleife nach n Durchläufen wie folgt:

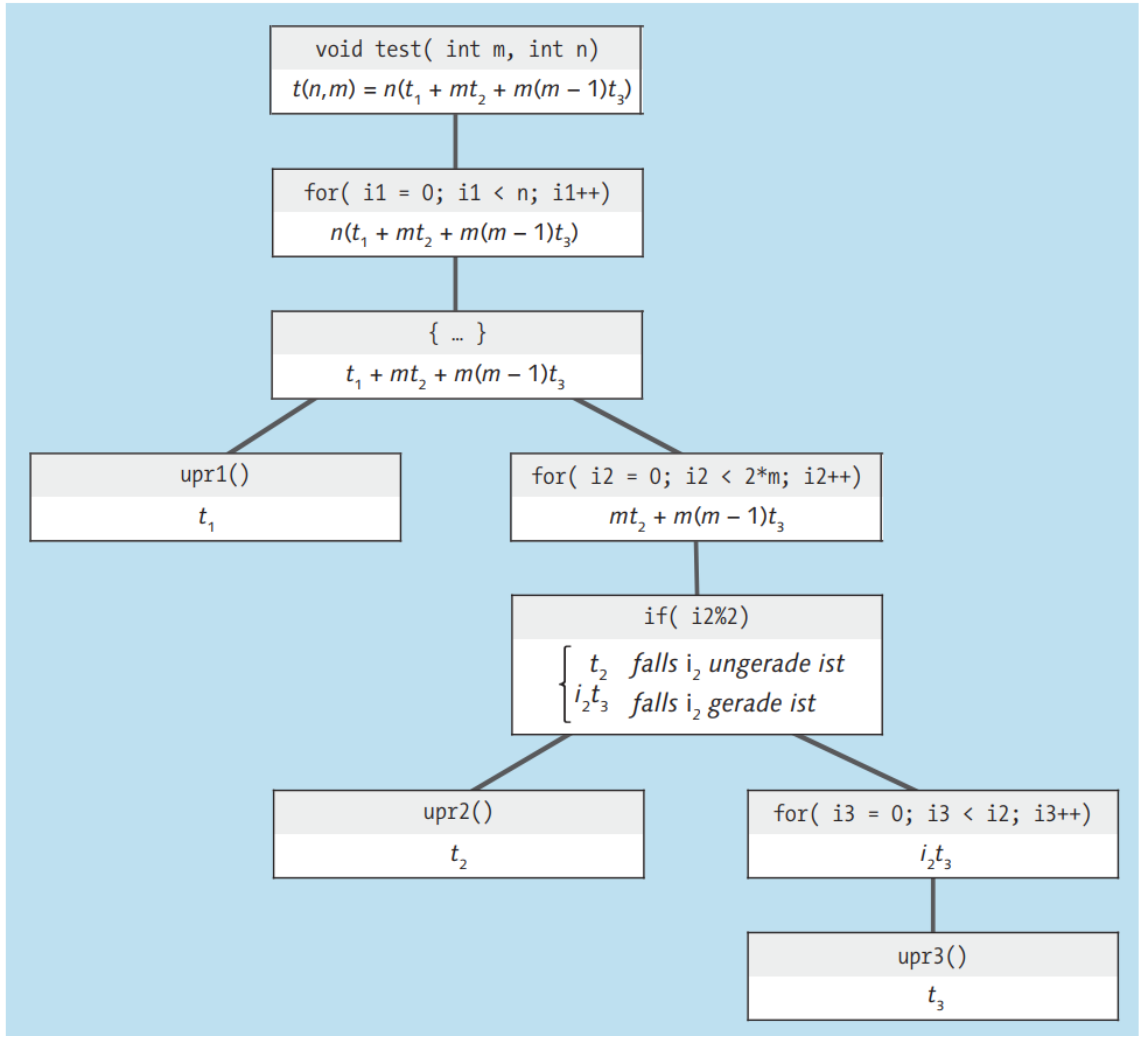
$$
\begin{aligned}
t(n) = {} & t_{init} + t_{test}(1) \\
& + t_{body}(1) + t_{incr}(1) + t_{test}(2) \\
& + t_{body}(2) + t_{incr}(2) + t_{test}(3) \\
& \ldots \\
& + t_{body}(n) + t_{incr}(n) + t_{test}(n+1)
\end{aligned}
$$

- Sollte die Initialisierung und der Vergleich in Schleifen zu vernachlässigen sein, dann vereinfacht sich die Formel zu:

$$t(n) = t_{body}(1) + t_{body}(2) + \ldots + t_{body}(n)$$

Leistungsanalyse
Beispiel mit konstanter Laufzeit

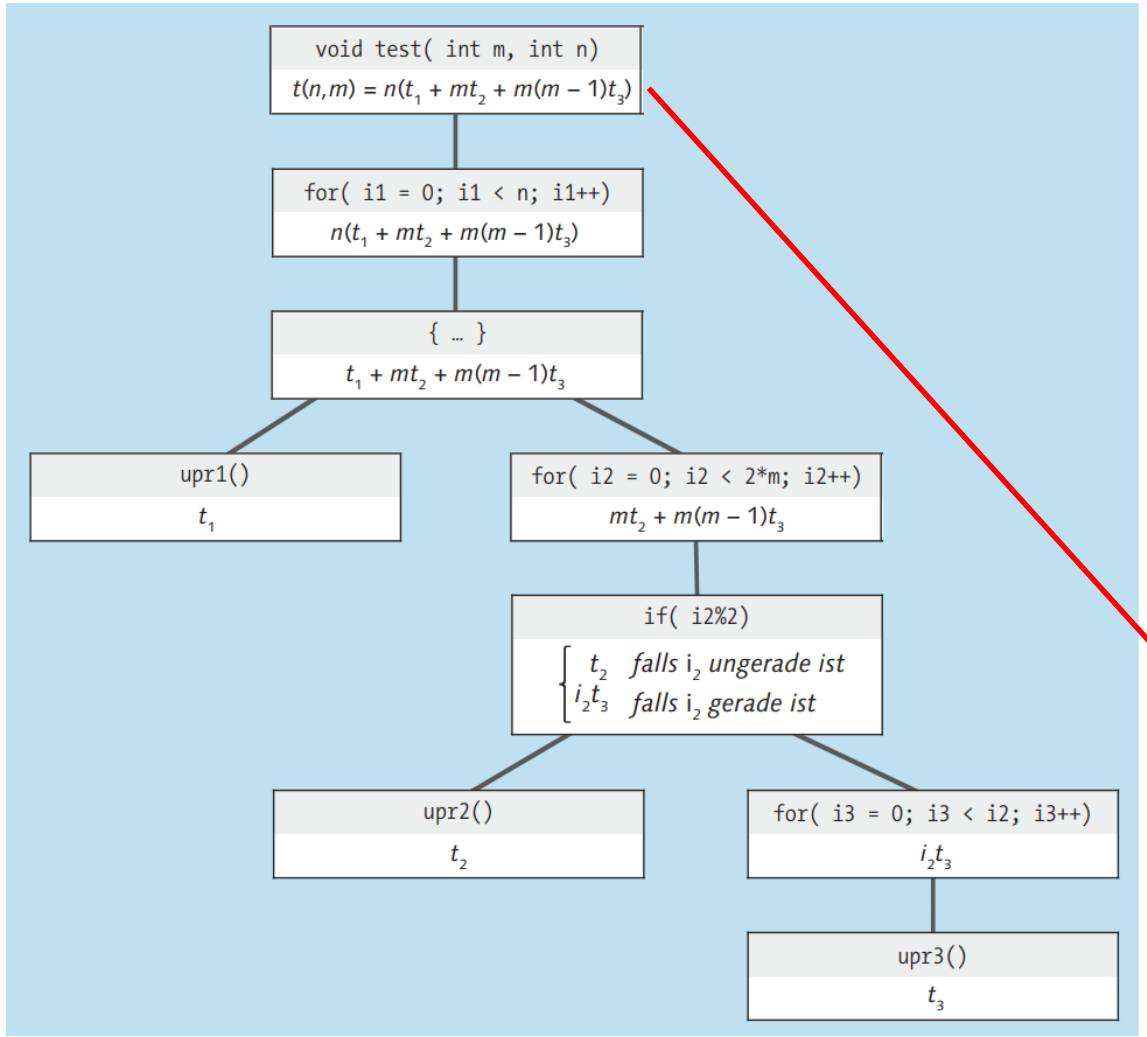
- Dem Schema nun für das ganze Programm test() folgende, ergibt sich:




```
void test(int n, int m)
    {
    int i1, i2, i3;

    for( i1 = 0; i1 < n; i1++)
        {
        upr1();
        for( i2 = 0; i2 < 2*m; i2++)
            {
            if( i2 % 2)
                upr2();
            else
                for( i3 = 0; i3 < i2; i3++)
                    upr3();
            }
        }
    }
```

Leistungsanalyse
Beispiel mit konstanter Laufzeit

- Dem Schema nun für das ganze Programm test() folgende, ergibt sich:




```
void test(int n, int m)
    {
    int i1, i2, i3;

    for( i1 = 0; i1 < n; i1++)
        {
        upr1();
        for( i2 = 0; i2 < 2*m; i2++)
            {
            if( i2% 2)
                upr2();
            else
                for( i3 = 0; i3 < i2; i3++)
                    upr3();
            }
        }
    }
```

Wir wissen, dass t1=500*t3 und t2=50*t3

$$t(n,m) = cn(m^2 + 49m + 500)$$

Leistungsanalyse
Beispiel mit konstanter Laufzeit

- Den Proportionalitätsfaktor können wir nur durch konkrete Messungen ermitteln, da er von der Ausführungsumgebung abhängt.
- Er gilt somit pro Rechner individuell.

```
void test(int n, int m)
    {
    int i1, i2, i3;

    for( i1 = 0; i1 < n; i1++)
        {
        upr1();
        for( i2 = 0; i2 < 2*m; i2++)
            {
            if( i2 % 2)
                upr2();
            else
                for( i3 = 0; i3 < i2; i3++)
                    upr3();
            }
        }
    }
```

Wir wissen, dass t1=500*t3 und t2=50*t3

$$t(n,m) = cn(m^2 + 49m + 500)$$

Leistungsanalyse
Beispiel mit konstanter Laufzeit

- Welchen Einfluss die drei Unterprogramme (t1,t2,t3) auf

  die Gesamtlaufzeit des Programms?

  ursprüngliche Formel:

  $$t(n,m) = n(t_1 + mt_2 + m(m-1)t_3)$$

```
void test(int n, int m)
    {
    int i1, i2, i3;

    for( i1 = 0; i1 < n; i1++)
        {
        upr1();
        for( i2 = 0; i2 < 2*m; i2++)
            {
            if( i2 % 2)
                upr2();
            else
                for( i3 = 0; i3 < i2; i3++)
                    upr3();
            }
        }
    }
```

- Wir sehen:
  - n geht in gleichermaßen in alle Unterprogramme ein.
  - m hingehend geht einfach in t2 und um ein vielfaches in t3 ein, obwohl t3 die geringste Laufzeit hat.
- Wir sehen demzufolge, dass man sich bei der Optimierung vorranging auf das upr3() konzentrieren sollte.

Wir wissen, dass t1=500*t3 und t2=50*t3

$$t(n,m) = cn(m^2 + 49m + 500)$$

Leistungsanalyse
Überdeckungsanalyse

- Für konkrete Messungen müssen wir konkreten Werten für die Parameter n und m wählen.
- Wir setzen exemplarisch n = 17 und m = 13 (willkürlich).
- Wir erwarten demnach:
  - $n = 17$ Aufrufe von `upr1`
  - $n \cdot m = 17 \cdot 13 = 221$ Aufrufe von `upr2`
  - $n \cdot m \cdot (m-1) = 17 \cdot 13 \cdot 12 = 2652$ Aufrufe von `upr3`

ursprüngliche Formel:

$$t(n,m) = n(t_1 + mt_2 + m(m-1)t_3)$$

Leistungsanalyse
Überdeckungsanalyse

- $n = 17$ Aufrufe von `upr1`
- $n \cdot m = 17 \cdot 13 = 221$ Aufrufe von `upr2`
- $n \cdot m \cdot (m-1) = 17 \cdot 13 \cdot 12 = 2652$ Aufrufe von `upr3`

ursprüngliche Formel:

$$t(n,m) = n(t_1 + mt_2 + m(m-1)t_3)$$

- Die Überdeckungsanalyse bestätigt das zuvor theoretisch hergeleitete Ergebnis

```
    1  void test(int n, int m)
       {
       int i1, i2, i3;

       for( i1 = 0; i1 < n; i1++)
           {
   17      upr1();
           for( i2 = 0; i2 < 2*m; i2++)
               {
  442          if( i2 % 2)
  221              upr2();
               else
  221              for( i3 = 0; i3 < i2; i3++)
 2652                  upr3();
               }
           }
       }
```

Leistungsanalyse
Performanceanalyse

- Nachdem die Überdeckungsanalyse exemplarisch die Korrektheit der Laufzeitformel bestätigt hat, erfolgt eine Performancemessung.
- Dazu wird zunächst ein Laufzeitprofil (Parameterbereich, zeitlicher Veränderung, …) erstellt.
- Anschließend wird das Profil ausgeführt.

n	m	upr1		upr2		upr3		Laufzeit	Laufzeit	Abweichung
		Aufrufe	Zeit	Aufrufe	Zeit	Aufrufe	Zeit	gerechnet	gemessen	
5	12	5	511,70	60	51,15	660	1,01	6294,10	6293,26	0,01%
17	13	17	509,89	221	50,98	2652	1,00	22586,71	22598,55	0,05%
7	33	7	512,04	231	50,93	7392	1,01	22815,03	22780,88	0,15%
9	9	9	511,23	81	50,90	648	1,00	9371,97	9373,75	0,02%
12	21	12	511,01	252	51,04	5040	1,01	24084,60	24066,44	0,08%
7	2	7	511,74	14	51,40	14	1,02	4316,06	4316,07	0,00%
14	10	14	510,31	140	51,13	1260	1,01	15575,14	15571,09	0,03%
19	3	19	511,04	57	51,17	114	1,01	12741,59	12741,02	0,00%
6	13	6	509,72	78	50,91	936	1,00	7965,30	7969,45	0,05%
5	18	5	509,85	90	51,00	1530	1,01	8684,55	8679,61	0,06%

Leistungsanalyse
Performanceanalyse

n	m	upr1		upr2		upr3		Laufzeit	Laufzeit	Abweichung
		Aufrufe	Zeit	Aufrufe	Zeit	Aufrufe	Zeit	gerechnet	gemessen	
5	12	5	511,70	60	51,15	660	1,01	6294,10	6293,26	0,01%
17	13	17	509,89	221	50,98	2652	1,00	22586,71	22598,55	0,05%
7	33	7	512,04	231	50,93	7392	1,01	22815,03	22780,88	0,15%
9	9	9	511,23	81	50,90	648	1,00	9371,97	9373,75	0,02%
12	21	12	511,01	252	51,04	5040	1,01	24084,60	24066,44	0,08%
7	2	7	511,74	14	51,40	14	1,02	4316,06	4316,07	0,00%
14	10	14	510,31	140	51,13	1260	1,01	15575,14	15571,09	0,03%
19	3	19	511,04	57	51,17	114	1,01	12741,59	12741,02	0,00%
6	13	6	509,72	78	50,91	936	1,00	7965,30	7969,45	0,05%
5	18	5	509,85	90	51,00	1530	1,01	8684,55	8679,61	0,06%

- Nach erneutem betrachten der Formel

$$t(n, m) = n(t_1 + mt_2 + m(m-1)t_3)$$

- sehen wir, dass m quadratisch und n linear in die Laufzeit eingeht.
- Hohe m-Werte führen demnach zu hoher Laufzeit.

Laufzeitklassen

- Da die vollständige Analyse (siehe vorher) in der Praxis unmöglich ist, genügen und zumeist Abschätzungen.
- Diese werde in Laufzeitklassen angegeben.
- Einige Beispiele dafür sind, wobei n den variablen Parameter darstellt:
  - Potenzfunktionen $(1, n, n^2, n^3, \ldots)$
  - Wurzelfunktionen $(\sqrt[2]{n}, \sqrt[3]{n}, \sqrt[4]{n}, \ldots)$
  - Exponentialfunktionen $(2^n, 3^n, 4^n, \ldots)$
  - Logarithmen $(log_2(n), log_3(n), log_4(n), \ldots)$
- Wir sind demnach nur an der oberen Schranke (Worst-Case) interessiert.

Laufzeitklassen Beispiel

- n=1-20, siehe schwarze Ausgabe

```
int funktion1( int n)
     {
     int x;
     int k = 0;

     for( x = 1; x <= n; x *= 2)
          k++;

      return k-1;
     }
```

```
void main()
     {
     int n;

     for( n = 1; n <= 20; n++)
          printf( "%2d %2d\n", n, funktion1( n));
     }
```

```
 1  0
 2  1
 3  1
 4  2
 5  2
 6  2
 7  2
 8  3
 9  3
10  3
11  3
12  3
13  3
14  3
15  3
16  4
17  4
18  4
19  4
20  4
```

Laufzeitklassen
Beispiel

- n=1-20, siehe schwarze Ausgabe
- Funktion erhöht Wert immer bei einer Zweierpotenz (2,4,8,16, ...).
- Das liegt daran, dass die Variable x immer ihren Wert verdoppelt, bis sie die Schranke n erreicht.
- Es ergibt sich: $x = 2^k$

$$x \leq n \Leftrightarrow 2^k \leq n \Leftrightarrow log_2(2^k) \leq log_2(n) \Leftrightarrow k \cdot log_2(2) \leq log_2(n) \Leftrightarrow k \leq log_2(n)$$

```
int funktion1( int n)
     {
     int x;
     int k = 0;

     for( x = 1; x <= n; x *= 2)
          k++;

      return k-1;
     }
```

```
void main()
     {
     int n;

     for( n = 1; n <= 20; n++)
          printf( "%2d %2d\n", n, funktion1( n));
     }
```

```
 1  0
 2  1
 3  1
 4  2
 5  2
 6  2
 7  2
 8  3
 9  3
10  3
11  3
12  3
13  3
14  3
15  3
16  4
17  4
18  4
19  4
20  4
```

Laufzeitklassen
Beispiel

- n=1-20, siehe schwarze Ausgabe
- Funktion erhöht Wert immer bei einer Zweierpotenz (2,4,8,16, …).
- Das liegt daran, dass die Variable x immer ihren Wert verdoppelt, bis sie die Schranke n erreicht.
- Es ergibt sich: $x = 2^k$

$$x \le n \Leftrightarrow 2^k \le n \Leftrightarrow log_2(2^k) \le log_2(n) \Leftrightarrow k \cdot log_2(2) \le log_2(n) \Leftrightarrow k \le log_2(n)$$

```
int funktion1( int n)
     {
     int x;
     int k = 0;

     for( x = 1; x <= n; x *= 2)
          k++;

      return k-1;
     }
```

```
void main()
     {
     int n;

     for( n = 1; n <= 20; n++)
          printf( "%2d %2d\n", n, funktion1( n));
     }
```

```
 1  0
 2  1
 3  1
 4  2
 5  2
 6  2
 7  2
 8  3
 9  3
10  3
11  3
12  3
13  3
14  3
15  3
16  4
17  4
18  4
19  4
20  4
```

- Da wir nur an einer Abschätzung interessiert sind, gilt hier $t(n) \le log_2(n)$