

Kapitel 12

Leistungsanalyse und Leistungsmessung

Die Qualität eines Programms liegt nicht darin, dass es fehlerfrei läuft.

Die Qualität eines Autos liegt ja auch nicht darin, dass es fährt.

Es gibt viele Kriterien, die zur Beurteilung der Programmqualität herangezogen werden können. Dazu gehören auch Kriterien, die zur Programmlaufzeit völlig bedeutungslos sind. Zum Beispiel:

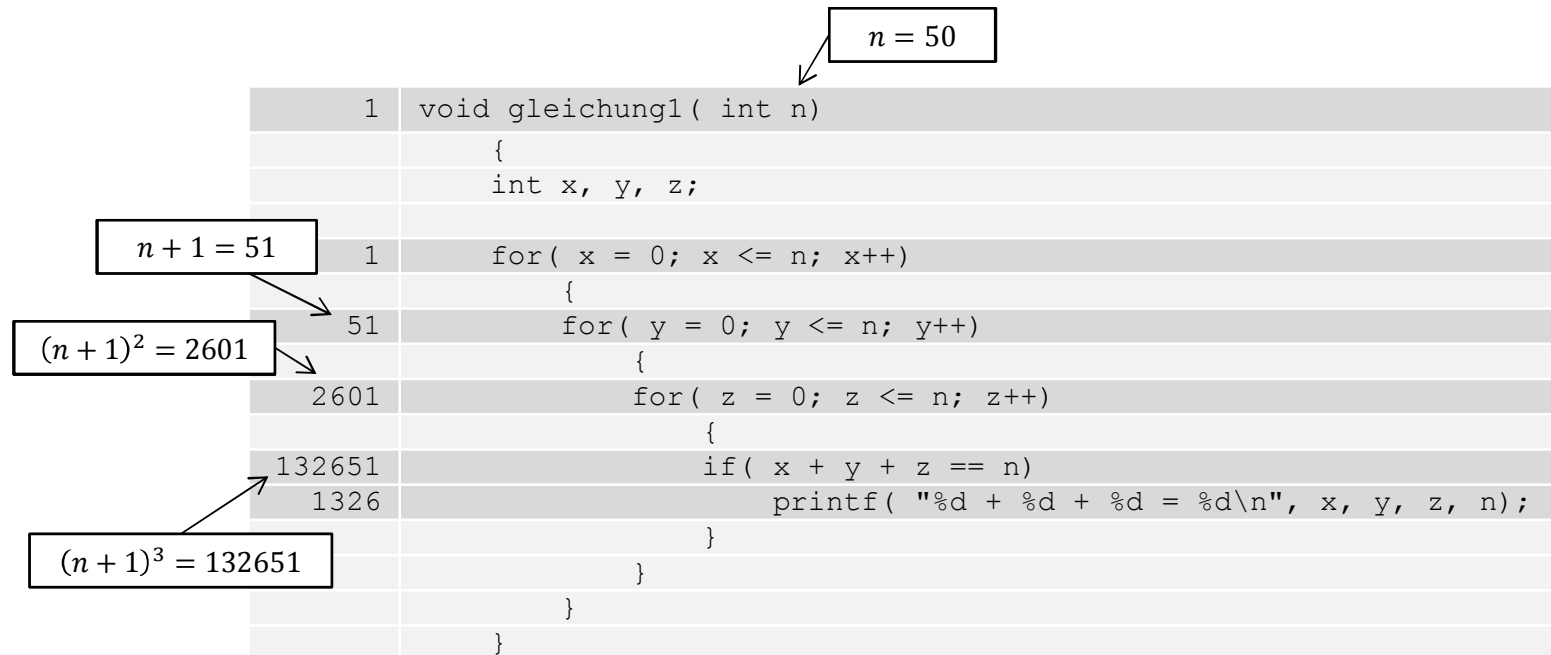
- Ist ein Programm ausreichend und verständlich kommentiert?
- Ist ein Programm flexibel anpassbar an zukünftige Anforderungen?

Wir interessieren uns hier allerdings nur für Kriterien, die zur Laufzeit große Bedeutung haben. Im Vordergrund steht dabei die Frage, ob ein Programm optimal mit den vom Betriebssystem bereitgestellten Ressourcen (Speicher und Rechenzeit) umgeht. Um Fragen nach dem Ressourcenverbrauch beantworten zu können, müssen wir lernen, Programme zu analysieren, um die zu erwartende Rechenzeit und den zu erwartenden Speicherverbrauch zu berechnen. Zunächst beschäftigen wir uns nur mit der Rechenzeit.

In der Regel hängt die Rechenzeit eines Programms vom Umfang der zu verarbeitenden Daten ab. Es ist offensichtlich, dass ein Sortierprogramm mehr Rechenzeit benötigt, wenn mehr Daten zu sortieren sind. Die Rechenzeit ist also nicht eine Zahl t sondern eine Funktion $t(n)$, wobei n die Anzahl der zu bearbeitenden Datensätze ist.

In diesem Abschnitt wollen wir Methoden erarbeiten, um aus dem Programmcode eine Funktion für die Laufzeit zu ermitteln.

Aufgabe: Suche alle ganzzahligen, nicht-negativen Lösungen der Gleichung $x + y + z = n$. Die Zahl n ist dabei beliebig, aber fest vorgegeben.



Analyse: Da sich mit jeder Schleife die Zahl der untersuchten Fälle um den Faktor $n+1$ vervielfacht, sind insgesamt $(n + 1)^3$ Fälle zu untersuchen und das Programm hat mit einem unbekannten Proportionalitätsfaktor c die Laufzeit

$$t(n) = c(n + 1)^3$$

Der konkrete Wert des Proportionalitätsfaktors interessiert nicht, zumal dieser Faktor auf unterschiedlich schnellen Rechnern unterschiedlich ausfallen wird und somit keine Kenngröße des Algorithmus ist.

Erste Optimierung

Sind x und y gewählt, so gibt es für z nur einen möglichen Wert $z = n - x - y$. Damit entfällt die innere Schleife:

		<div>$n = 50$</div>
1	void gleichung2(int n)	
	{	
	int x, y, z;	
	1	for(x = 0; x <= n; x++)
		{
<div>$n + 1 = 51$</div>	51	for(y = 0; y <= n; y++)
		{
<div>$(n + 1)^2 = 2601$</div>	2601	z = n - x - y;
	2601	if(z >= 0)
	1326	printf("%d + %d + %d = %d\n", x, y, z, n);
		}
		}
		}

Analyse: Die Anzahl der betrachteten Fälle reduziert sich von $(n + 1)^3$ auf $(n + 1)^2$ und es ist davon auszugehen, dass dieses Programm mit einer Laufzeit von $t(n) = c(n + 1)^2$ bei gleicher Funktionalität, insbesondere für großes n deutlich schneller ist.

Zweite Optimierung

Da oberhalb von $y = n - x$ keine Lösungen für z mehr gefunden werden können, kann man die Schleife über y bei Überschreitung des Werts $n - x$ abbrechen. Da $z = n - x - y$ in dieser Situation stets größer oder gleich 0 ist, ist die Abfrage $z \geq 0$ dann nicht mehr erforderlich:

	1	void gleichung3(int n	$n = 50$
		{	
		int x, y, z;	
	1	for(x = 0; x <= n; x++)	
		{	
$n + 1 = 51$	51	for(y = 0; y <= n-x; y++)	
		{	
$\frac{(n+1)(n+2)}{2} = 1326$	1326	z = n - x - y;	
	1326	printf("%d + %d + %d = %d\n", x, y, z, n);	
		}	
		}	
		}	

Analyse: Bei gegebenem x gibt es für y nur noch $n - x + 1$ Möglichkeiten.

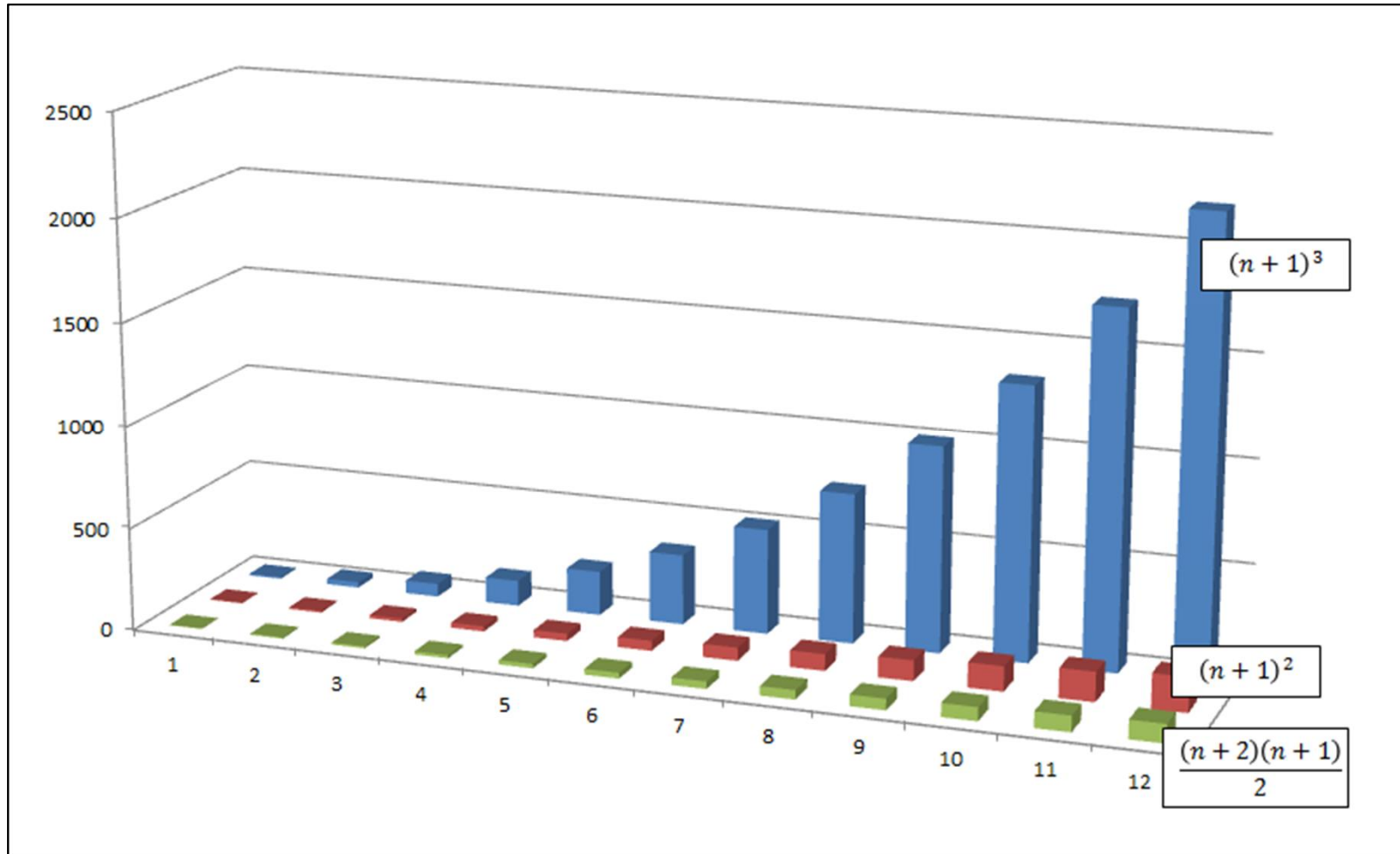
Die Laufzeit ist $t(n) = c \frac{(n+2)(n+1)}{2}$

Weitere Verbesserungen sind nicht zu erwarten, da dies die effektive Zahl der Lösungen (Suchraum = Lösungsraum) ist.

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

Vergleich der drei Lösungen

Die drei Lösungen können über ihre Laufzeitfunktionen miteinander verglichen werden.



Die Schere zwischen der ersten und den beiden anderen Lösungen geht für große n immer weiter auf. Die dritte Lösung ist etwa doppelt so schnell wie die zweite.

Eine vollständige Laufzeitanalyse

```
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();
        }
    }
}
```

```
void upr1()
{
    int i;

    for( i = 0; i < 500; i++)
        machwas();
}
```

```
void upr2()
{
    int i;

    for( i = 0; i < 50; i++)
        machwas();
}
```

```
void upr3()
{
    machwas();
}
```

```
void machwas()
{
    int i;
    int a, b, c;

    a = b = c = 0;
    for( i = 0; i < 300000; i++)
    {
        a = b;
        b = c;
        c = a;
    }
}
```

Dieses Programm hat keinen Sinn, es soll nur Rechenzeit verbrauchen.

Gesucht ist eine Formel

$$t(n, m) = \dots ? \dots,$$

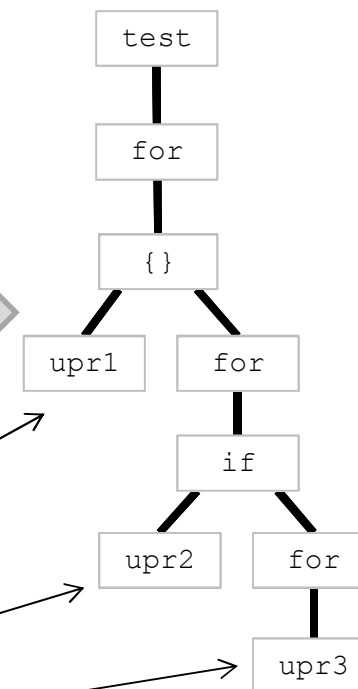
die das Laufzeitverhalten des Programms
test vollständig beschreibt.

Zerlegung des Programms in seine Bausteine

- Blöcke
- Fallunterscheidungen
- Schleifen
- Unterprogramme

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

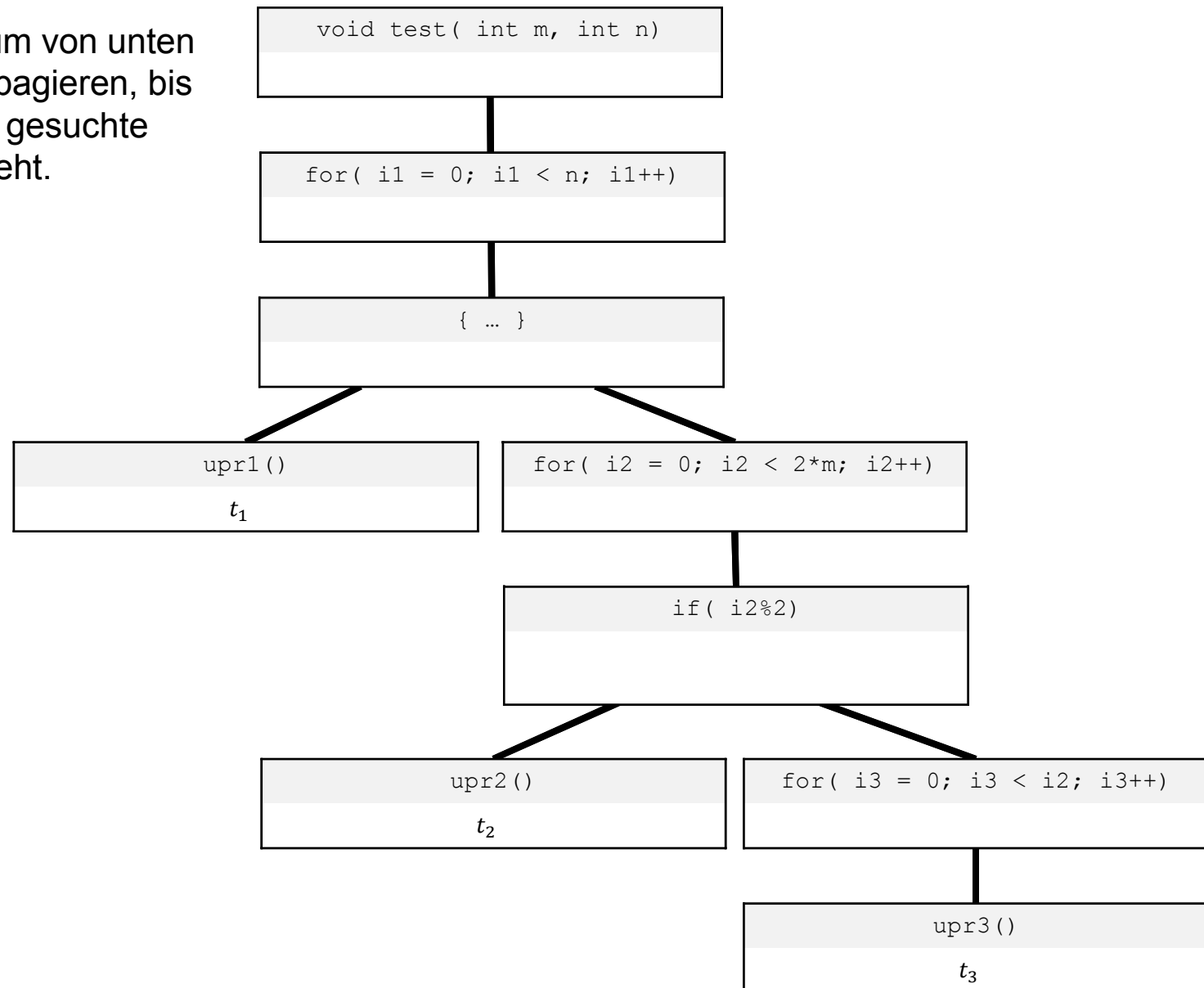
    for( i1 = 0; i1 < n; i1++)
    {
        upr1();
        for( i2 = 0; i2 < 2*m; i2++)
        {
            if( i2 % 2)
                upr2();
            else
                for( i3 = 0; i3 < i2; i3++)
                    upr3();
        }
    }
}
```



Die Laufzeiten an den Endknoten sind konstant. Wir vermuten, dass `upr2` die 50-fache und `upr1` die 500-fache Laufzeit von `upr3` hat. Die effektiven Laufzeiten sind uns nicht bekannt. Wir werden sie später durch Messung ermitteln.

Idee:

Laufzeiten im Baum von unten nach oben zu propagieren, bis an der Wurzel die gesuchte Gesamtlaufzeit steht.



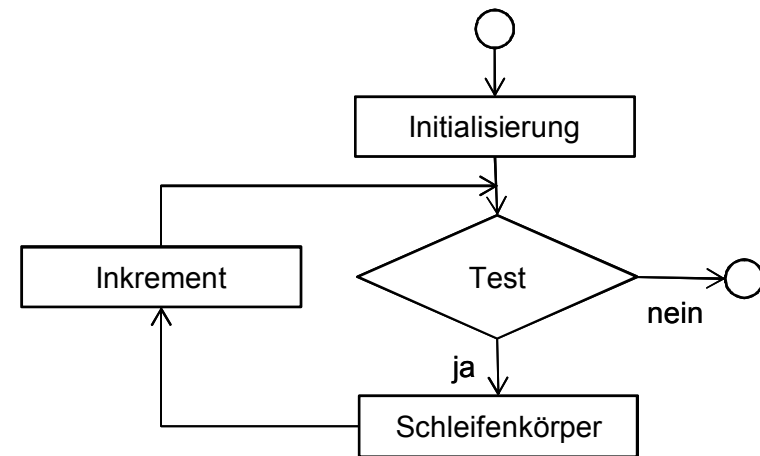
Laufzeit von Schleifen

Gegeben sei eine **Schleife** der Form:

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

Des Weiteren sei:

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:

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

Diese Formel ist in der Regel zu komplex, um angewandt zu werden.

Vereinfachte Berechnung von Schleifenlaufzeiten

Wenn die Laufzeiten zur Schleifensteuerung eine im Vergleich zum Schleifenkörper vernachlässigbare Größenordnung haben, kann die Formel wie folgt vereinfacht werden:

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

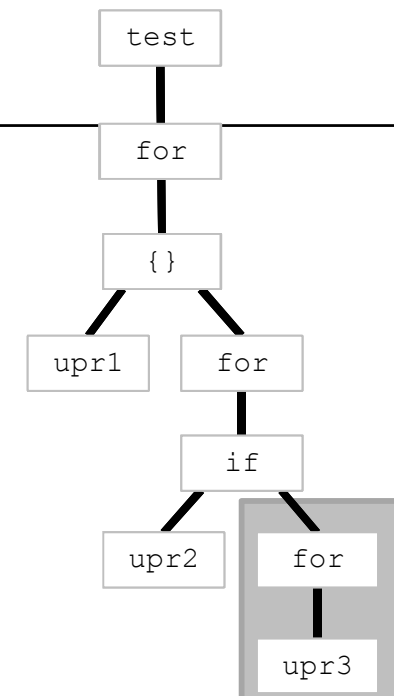
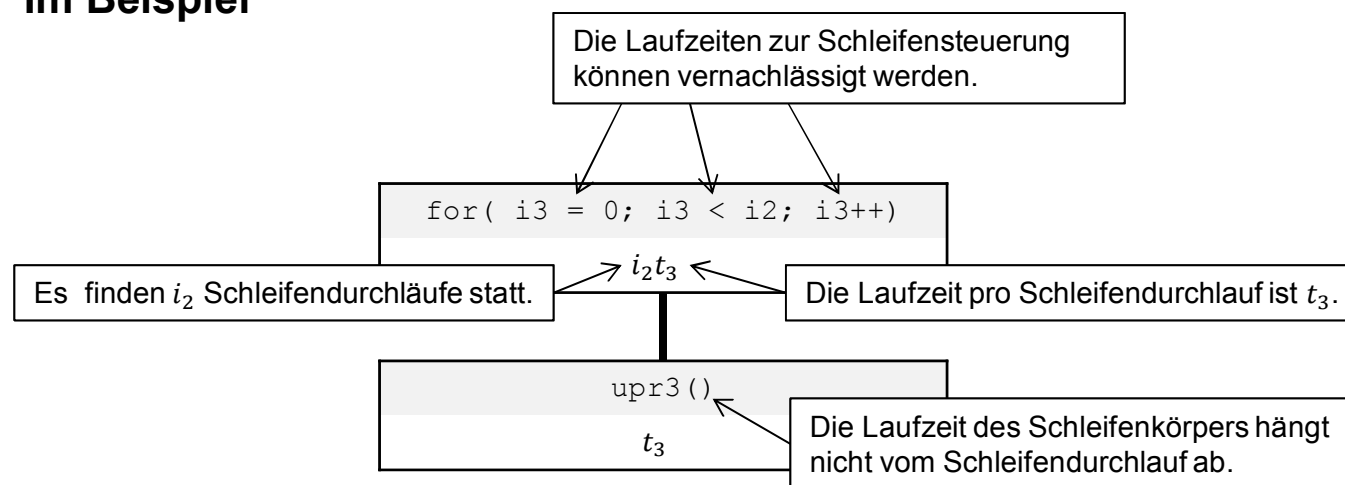
Gibt es zusätzlich eine gemeinsame obere Schranke t_{max} für die Laufzeit des Schleifenkörpers, so ist:

$$t(n) \leq nt_{max}$$

Ist die Laufzeit des Schleifenkörpers sogar unabhängig vom einzelnen Schleifendurchlauf, so ergibt sich

$$t(n) = nt_{body}$$

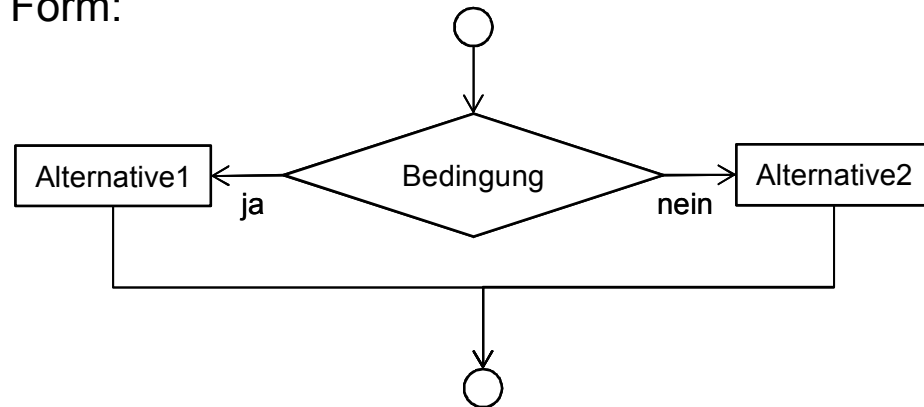
Im Beispiel



Laufzeit von Fallunterscheidungen

Gegeben sei eine **Fallunterscheidung** der Form:

```
if( bedingung)
  alternative1
else
  alternative2
```



Des Weiteren sei:

t_{bed} die Laufzeit des zur Überprüfung der Bedingung
 t_{alt1} die Laufzeit der Alternative1
 t_{alt2} die Laufzeit der Alternative2

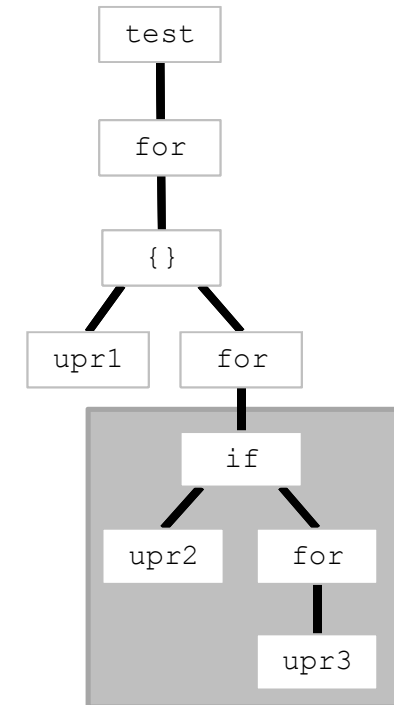
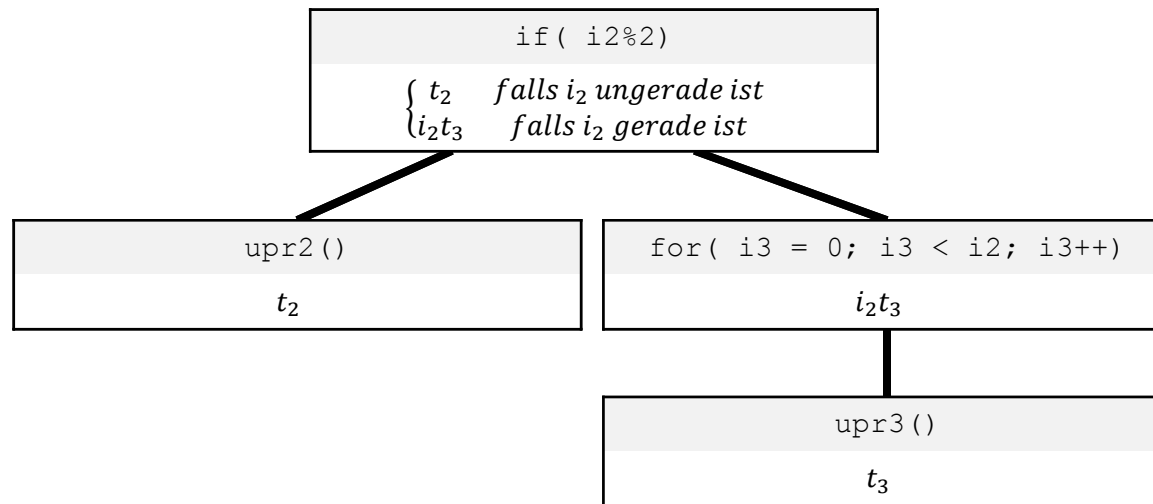
Dann berechnet sich die Laufzeit der Fallunterscheidung wie folgt:

$$t = t_{bed} + \begin{cases} t_{alt1} & \text{falls die Bedingung erfüllt ist} \\ t_{alt2} & \text{falls die Bedingung nicht erfüllt ist} \end{cases}$$

Ist die Laufzeit zur Überprüfung der Bedingung vernachlässigbar im Vergleich zur Laufzeit der Alternativen, so vereinfacht sich die Formel zu:

$$t = \begin{cases} t_{alt1} & \text{falls die Bedingung erfüllt ist} \\ t_{alt2} & \text{falls die Bedingung nicht erfüllt ist} \end{cases}$$

Anwendung im Beispiel



Fallunterscheidungen sind im Folgenden schwer zu handhaben, da beide Fälle getrennt weiter behandelt werden müssten. Man sollte daher versuchen, die beiden Fälle rechnerisch zusammenzufassen.

Beseitigen der Fallunterscheidung durch Abschätzung

Wenn es eine gemeinsame obere Schranke t_{max} für die Laufzeiten der beiden Alternativen gibt, so kann man die Laufzeit der Fallunterscheidung abschätzen:

$$t \leq t_{bed} + t_{max}$$

Als obere Schranke ist die Laufzeitsumme der beiden Alternativen geeignet:

$$t \leq t_{bed} + t_{alt1} + t_{alt2}$$

Auch hier kann t_{bed} weggelassen werden, wenn die Laufzeit zur Prüfung der Bedingung klein im Vergleich zu den anderen Laufzeiten ist.

In unserem Beispiel können wir wie folgt abschätzen: $t \leq t_2 + i_2 t_3$

Achtung:

Bei den einzelnen Laufzeiten handelt es sich im Allgemeinen um Funktionen, die noch von außen liegenden Parametern abhängen. Es geht hier also nicht darum, einfach nur den größeren zweier Werte zu bestimmen, sondern eine Funktion zu finden, die (möglichst knapp) oberhalb der Funktionen für die Alternativen verläuft und möglichst einfach ist. Eine solche Funktion ist nicht immer leicht zu finden.

Unter Umständen kommt man bei einer Abschätzung durch die Einbeziehung seltener, aber rechenintensiver Sonderfälle zu sehr ungünstigen Werten, die die wirkliche Leistungsfähigkeit des Algorithmus nicht mehr wiedergeben.

Beseitigen der Fallunterscheidung durch statistische Betrachtung

Gibt es Informationen darüber, mit welcher Wahrscheinlichkeit p ($0 \leq p \leq 1$) die Bedingung in der Fallunterscheidung wahr wird, so lässt sich die mittlere Laufzeit wie folgt ermitteln:

$$t = t_{bed} + pt_{alt1} + (1 - p)t_{alt2}$$

Wie üblich kann die Laufzeit zur Prüfung der Bedingung weggelassen werden, wenn sie durch die anderen Terme dominiert wird.

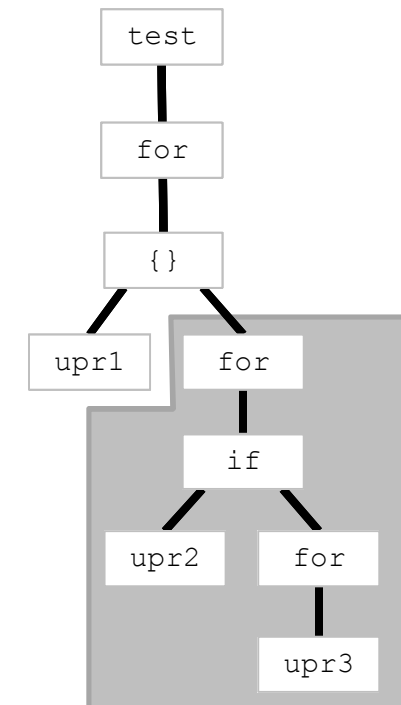
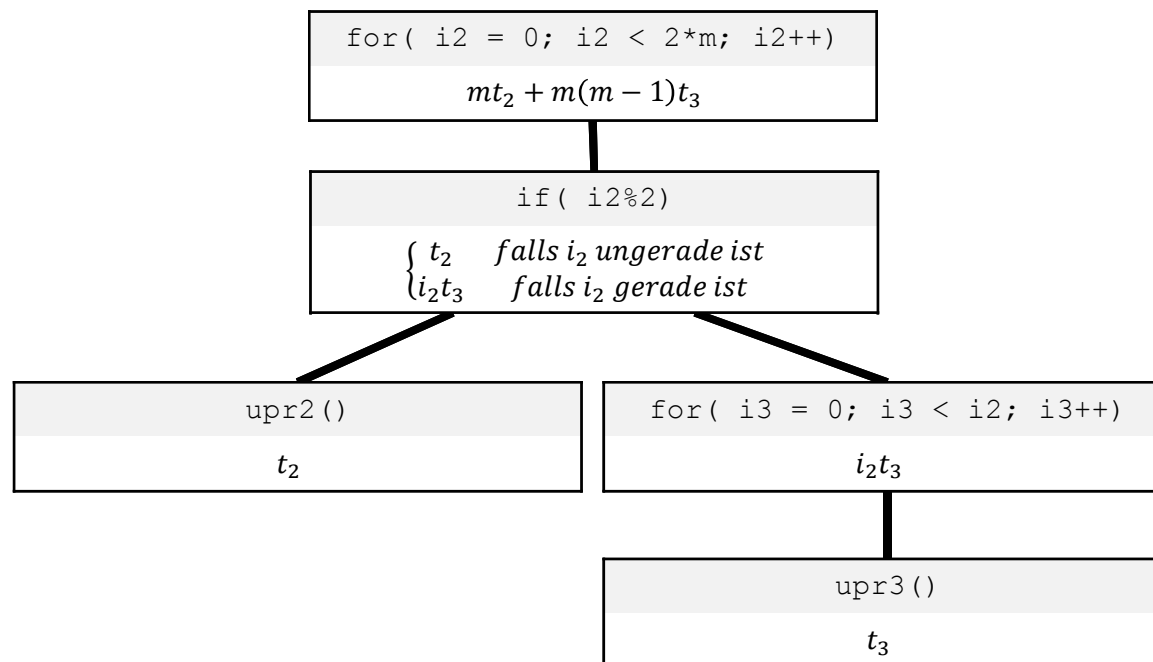
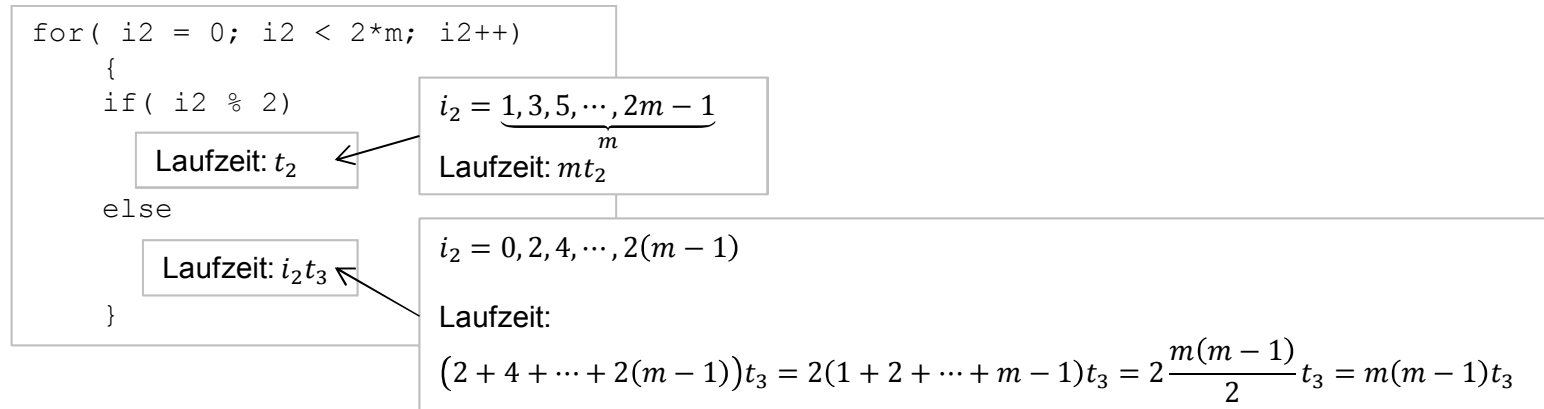
Da in unserem Beispiel die Fallunterscheidung gleich häufig mit geraden und ungeraden Werten für i_2 gerufen wird, können wir t wie folgt berechnen:

$$t = \frac{1}{2}t_2 + \frac{1}{2}i_2t_3$$

Eine solche Formel liefert keinen exakten Wert sondern eine statistische Aussage.

Elimination der Fallunterscheidung

In unserem Beispiel können wir die Fallunterscheidung vollständig "herausrechnen"



Laufzeit von Blöcken

Gegeben sei ein **Block** der Form:

```
{  
  anweisung_1  
  anweisung_2  
  ...  
  anweisung_n  
}
```

Des Weiteren sei

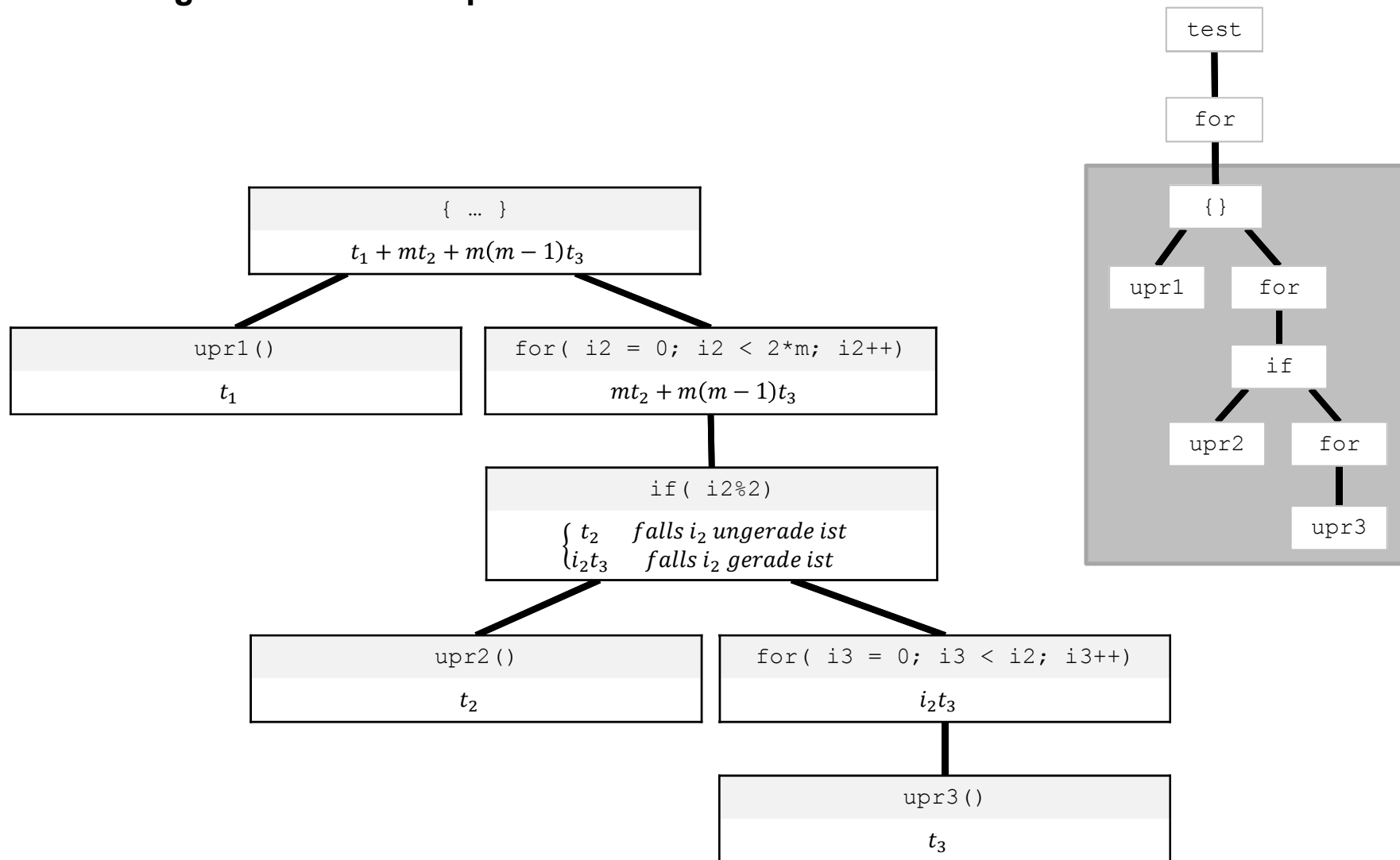
t_k die Laufzeit der k-ten Anweisung im Block.

Dann berechnet sich die Gesamtlaufzeit des Blocks nach der Formel:

$$t = t_1 + t_2 + \dots + t_n$$

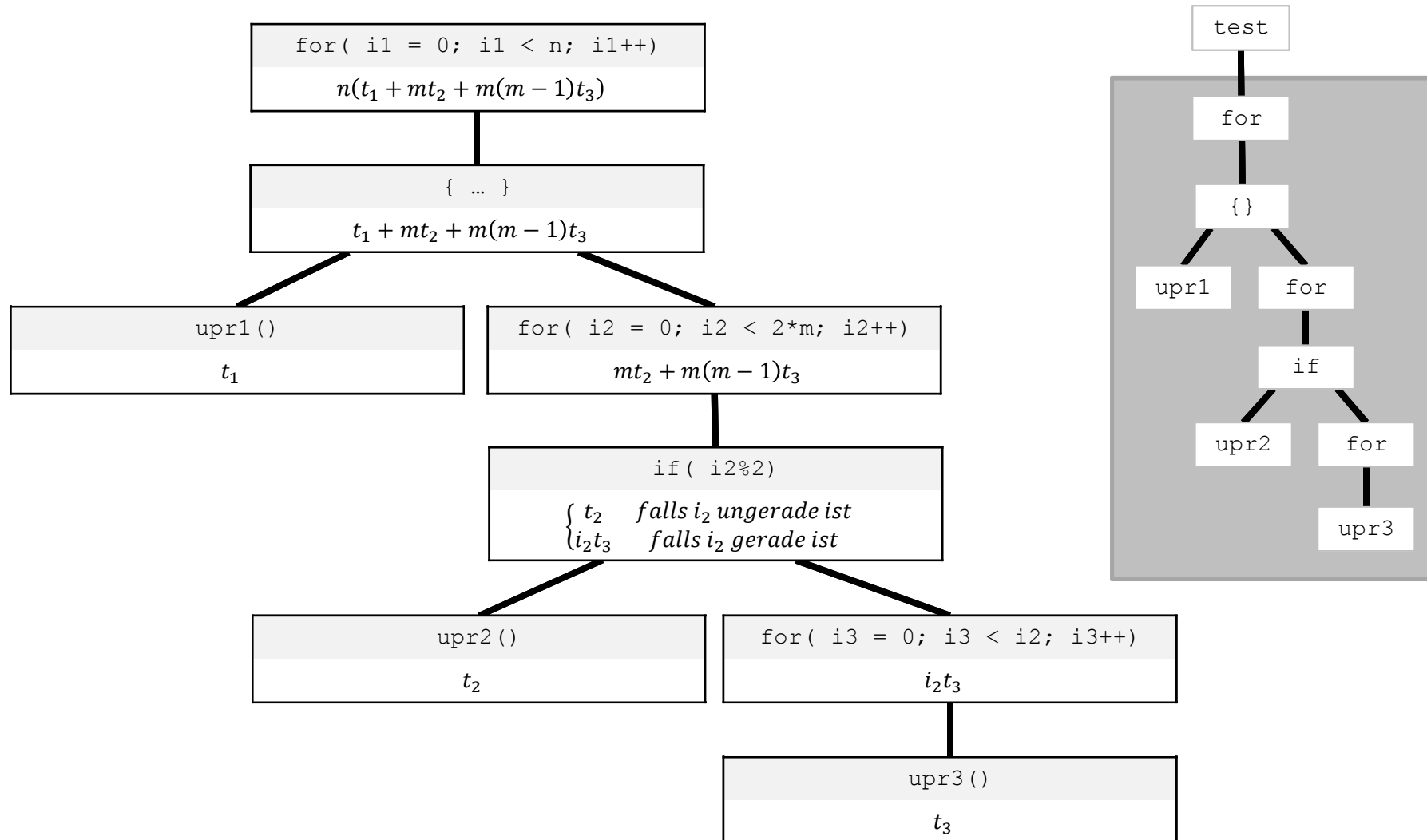
Dominierte Terme können weggelassen werden.

Anwendung in unserem Beispiel:

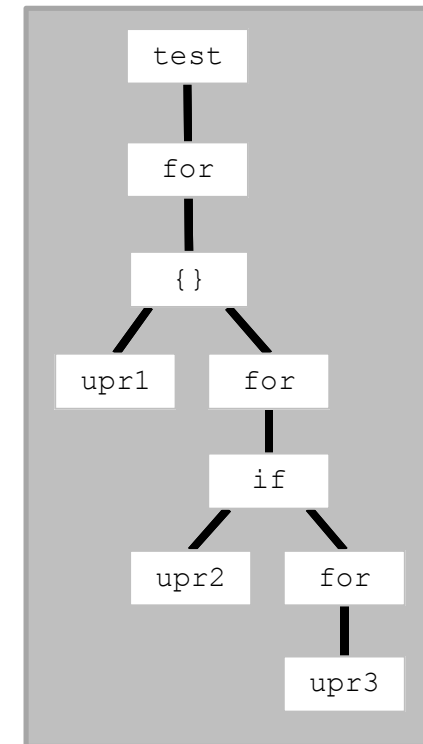
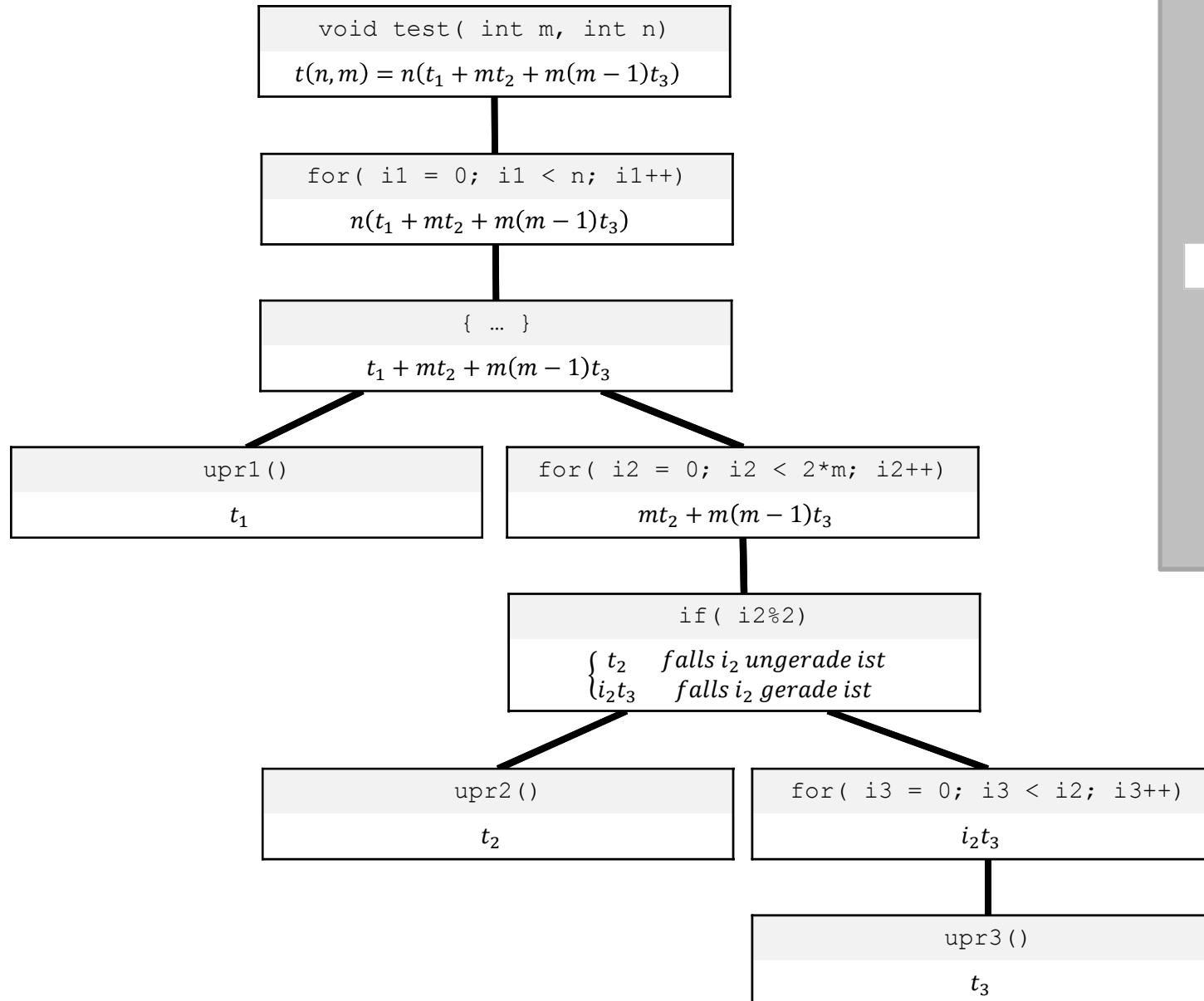


Auflösung der äußeren Schleife

Die Laufzeit des Schleifenkörpers ist unabhängig vom Schleifendurchlauf und die Laufzeiten zur Schleifensteuerung können vernachlässigt werden.



Zusammenfassung und Ergebnis



Die gesuchte Laufzeitformel

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

Bezüglich n haben alle drei Unterprogramme das gleiche Gewicht.

Bezüglich m spielt das dritte Unterprogramm, obwohl es nur ein Bruchteil der Laufzeit des ersten hat, eine bedeutend gewichtigere Rolle. Wenn m zum Beispiel den Wert 1000 hat, geht das erste Unterprogramm einfach, das zweite tausendfach und das dritte nahezu millionenfach in die Laufzeitbilanz ein. Dies zeigt, dass man sich bei einer Optimierung des Algorithmus in erster Linie auf das dritte Unterprogramm konzentrieren sollte.

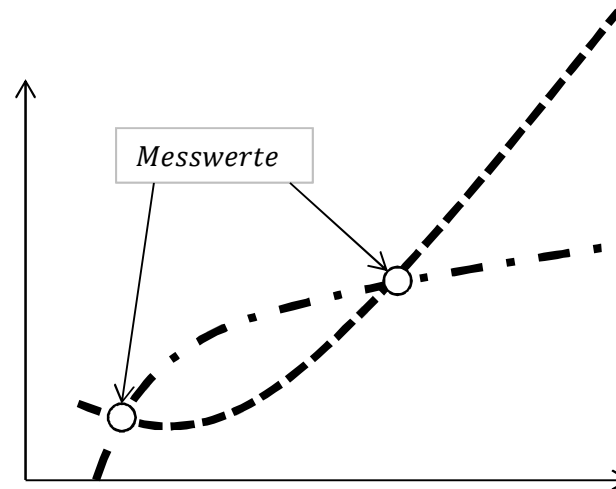
Da $t_1 = 500t_3$ und $t_2 = 50t_3$ ist, können wir mit einem Proportionalitätsfaktor c schreiben:

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

Den Proportionalitätsfaktor können wir nur durch konkrete Messungen ermitteln. Er gilt dann aber nur für den Rechner, auf dem die Messung durchgeführt wurde.

Laufzeitmessungen

Eine Messung oder eine Messreihe ist in der Regel viel einfacher durchzuführen als eine mathematische Analyse eines Programms, hat aber weniger Aussagekraft. Ohne zusätzliche Überlegungen sagen einzelne Messwerte wenig über den Gesamtverlauf einer Kurve:



Auch die Hinzunahme weiterer Messpunkte führt nicht zu der Sicherheit, die eine Analyse liefert. Da aber eine vollständige Analyse von Algorithmen oft unmöglich ist, muss man letztlich doch auf Messungen zurückgreifen. Parallel zu den Messungen sollte man sich aber stets anhand mathematischer Überlegungen darüber Rechenschaft ablegen, inwieweit die Messergebnisse plausibel und verallgemeinerungsfähig sind.

Fragestellungen für die Laufzeitmessung:

- Wie oft werden Teile des Programms durchlaufen?
- Wie viel Rechenzeit wird für Teile des Programms benötigt?

```
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();
        }
    }
}
```

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

Überdeckungsanalyse

Messungen müssen mit konkreten Werten für die Parameter n und m durchgeführt werden.
Wir setzen mehr oder weniger willkürlich $n=17$ und $m=13$.

Wir erwarten dann:

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

$n = 17$ 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`

Die Überdeckungsanalyse bestätigt das zuvor mathematisch hergeleitete Ergebnis:

1	<code>void test(int n, int m)</code>
	<code>{</code>
	<code>int i1, i2, i3;</code>
	<code>for(i1 = 0; i1 < n; i1++)</code>
	<code>{</code>
17	<code>upr1();</code>
	<code>for(i2 = 0; i2 < 2*m; i2++)</code>
	<code>{</code>
442	<code>if(i2 % 2)</code>
221	<code>upr2();</code>
	<code>else</code>
221	<code>for(i3 = 0; i3 < i2; i3++)</code>
2652	<code>upr3();</code>
	<code>}</code>
	<code>}</code>
	<code>}</code>

Laufzeitmessungen

Messungen mit einem Profiler liefern Laufzeiten für die Unterprogramme upr1, upr2 und upr3.

Mit der Formel $t(n, m) = n(t_1 + mt_2 + m(m - 1)t_3)$ kann man dann die Laufzeiten für beliebige Parameter vorhersagen. Die Vorhersagen werden mit den gemessenen Laufzeiten für verschiedene, zufällig gewählte Parameterwerte verglichen:

n	m	upr1		upr2		upr3		Laufzeit gerechnet	Laufzeit gemessen	Abweichung
		Aufrufe	Zeit	Aufrufe	Zeit	Aufrufe	Zeit			
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%

Messungen gelten nur für den Rechner, auf dem sie gemacht wurden. Je nach gewählter Messmethode fließen zeitabhängige, wenig kontrollierbare Störgrößen (z.B. Rechnerauslastung) in die Messergebnisse ein. Die analytisch hergeleitete Formel ist dagegen unabhängig solchen Störgrößen.

Mathematische Grundlagen

Für die weiteren Überlegungen dieses Abschnitts setze ich voraus, dass Sie einige wichtige mathematische Grundfunktionen und Formeln beherrschen. Im Einzelnen handelt es sich um:

Potenzfunktionen $(1, n, n^2, n^3, \dots)$

Wurzelfunktionen $(\sqrt[n]{n}, \sqrt[n]{n}, \sqrt[n]{n}, \dots)$

Exponentialfunktionen $(2^n, 3^n, 4^n, \dots)$

Logarithmen $(\log_2(n), \log_3(n), \log_4(n), \dots)$

Ferner benötigen wir einige Formeln, die Sie in jeder Formelsammlung finden:

$$1 + 2 + 3 + \dots + n = \frac{n(n+1)}{2} \text{ (Summe der ersten } n \text{ Zahlen)}$$

$$1 + 3 + 5 + 7 + \dots + 2n - 1 = n^2 \text{ (Summe der ersten } n \text{ ungeraden Zahlen)}$$

$$1 + 2^2 + 3^2 + \dots + n^2 = \frac{n(n+1)(2n+1)}{6} \text{ (Summe der ersten } n \text{ Quadratzahlen)}$$

$$q^0 + q^1 + q^2 + \dots + q^n = \frac{q^{n+1} - 1}{q - 1} \text{ für } q \neq 1 \text{ (Summenformel der geometrischen Reihe)}$$

Gibt es nur Polynome als Laufzeitfunktion?

Versuchen Sie, eine Formel $t(n)$ für den Rückgabewert (= Laufzeit) folgender Funktion zu finden:

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

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

    return k-1;
}
```

Für $n = 1 \dots 20$ erhalten wir folgende Werte:

```
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

Wie ist das allgemeine Bildungsgesetz?

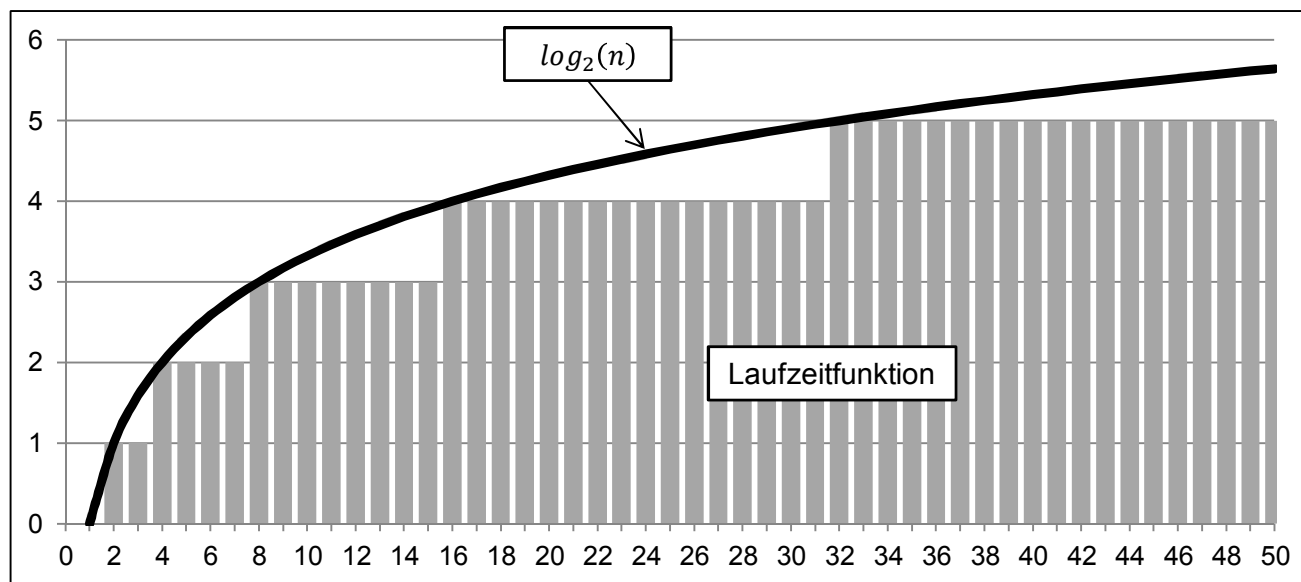
Analyse der Schleife

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

Der Wert x wird bei jedem Schleifendurchlauf verdoppelt. Wie oft muss man 1 verdoppeln, bis die Schranke n erreicht ist? Im k -ten Schleifendurchlauf ist $x = 2^k$. Damit folgt:

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

Somit läuft k immer bis zum nächstliegenden ganzzahligen Wert des Zweierlogarithmus von n . Die Laufzeitfunktion ist also eine "diskrete Abtastung" des Logarithmus: $t(n) \leq \log_2(n)$



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

Ein weiteres Beispiel

Versuchen Sie eine Formel $t(n)$ für den Rückgabewert (= Laufzeit) dieser Funktion zu finden:

```
int funktion2( int n)
{
    int x, y;
    int k = 0;

    for( x = 0, y = 1; x <= n; x += y, y += 2)
        k++;

    return k-1;
}
```

Für $n = 1 \dots 20$ erhalten wir folgende Werte:

```
void main()
{
    int n;

    for( n = 1; n <= 20; n++)
        printf( "%2d %2d\n", n, funktion2( n));
}
```

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

Wie ist das allgemeine Bildungsgesetz?

Analyse der Schleife

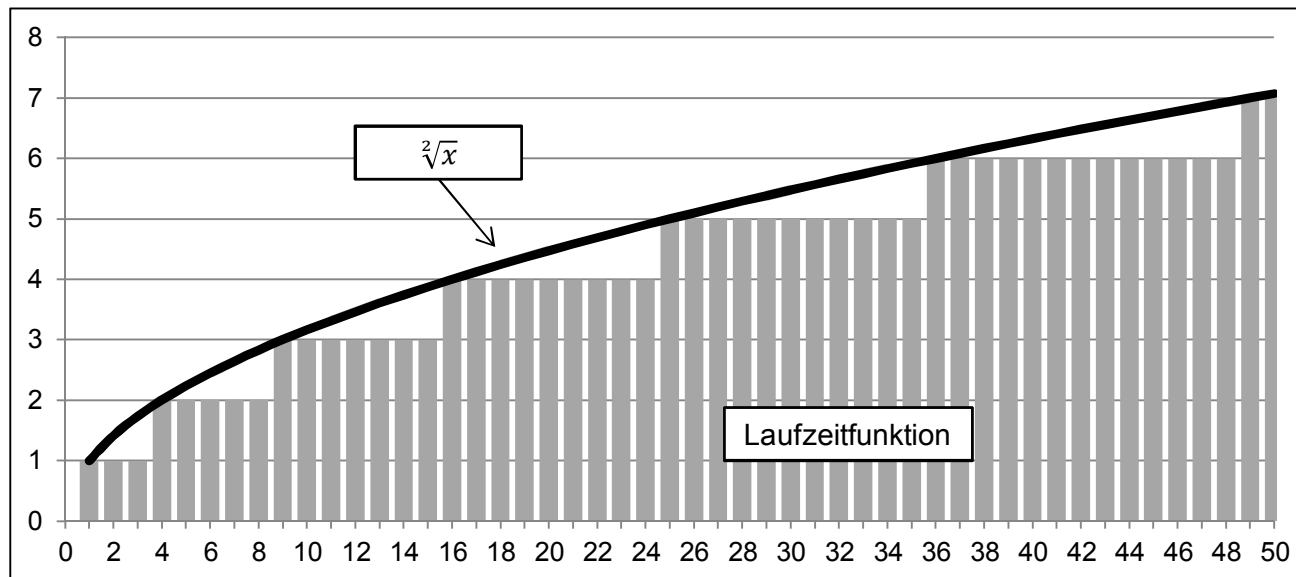
```
for( x = 0, y = 1; x <= n; x += y, y += 2)
    k++;
```

Die Variable y durchläuft die ungeraden Zahlen (1, 3, 5, ...). Die Variable x berechnet also Summen ungerader Zahlen. Die Summe der ersten k ungeraden Zahlen ist k^2 . Damit folgt:

$$x \leq n \Leftrightarrow k^2 \leq n \Leftrightarrow k \leq \sqrt{n}.$$

Das heißt: $t(n) \leq \sqrt{n}$

Die Laufzeitfunktion ist also eine Diskretisierung der Wurzelfunktion:

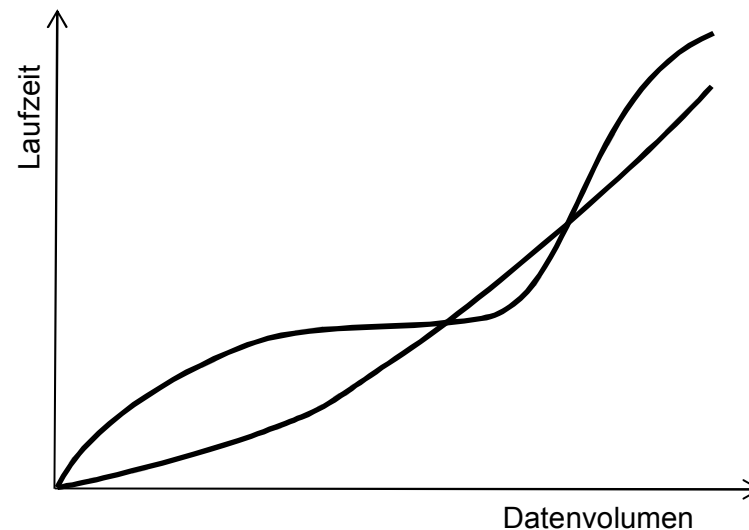


1	1
2	1
3	1
4	2
5	2
6	2
7	2
8	2
9	3
10	3
11	3
12	3
13	3
14	3
15	3
16	4
17	4
18	4
19	4
20	4

Vergleich von Laufzeitfunktionen

Will man verschiedene Algorithmen bezüglich ihrer Laufzeitqualität vergleichen, muss man ihre Laufzeitfunktionen vergleichen.

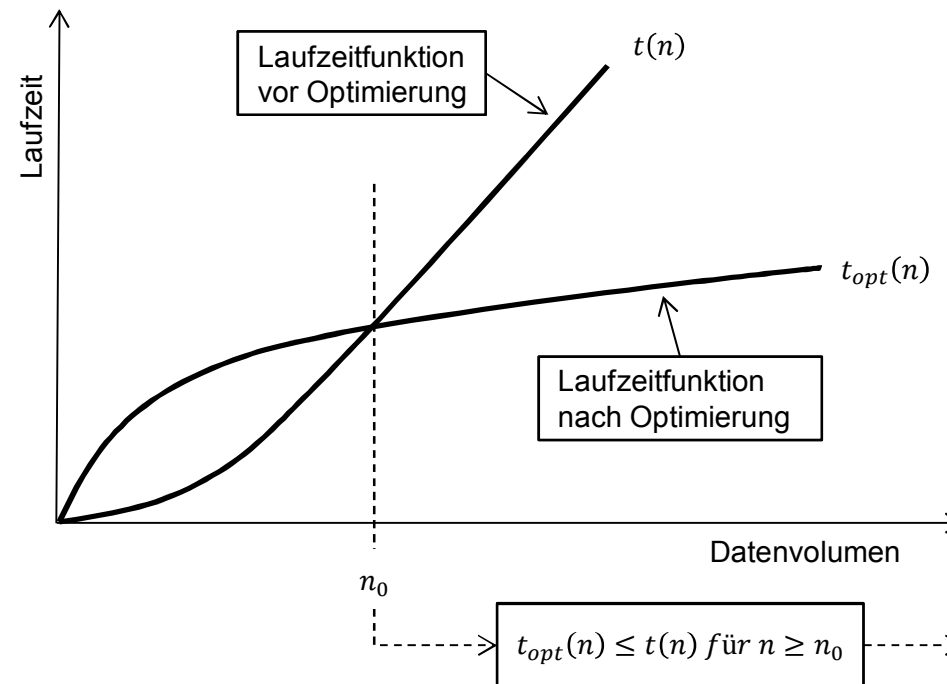
Der Vergleich zweier Funktionen ist nicht so einfach, wie der Vergleich zweier Zahlenwerte, da beim Vergleich von Funktionen unendlich viele Funktionswerte betrachtet werden müssen. Die Idealvorstellung, dass eine Laufzeitfunktion immer (d. h. für alle Funktionswerte) besser ist als eine andere, wird sich im Allgemeinen nicht ergeben.



Wir benötigen einen Vergleichsbegriff für Funktionen, der von unwesentlichen Details abstrahiert und sich auf das Wesentliche konzentriert. Was aber ist das Wesentliche?

Worauf es beim Vergleich von Laufzeitfunktionen nicht ankommt

Bei der Optimierung einer Programmierschleife gelingt es, die Laufzeit des Schleifenkörpers deutlich zu verkürzen. Leider muss dabei ein größerer Aufwand zur Initialisierung der Schleife in Kauf genommen werden. Das neue Programm ist für kleine Datenmengen schlechter und erst für große Datenmengen besser als das alte:

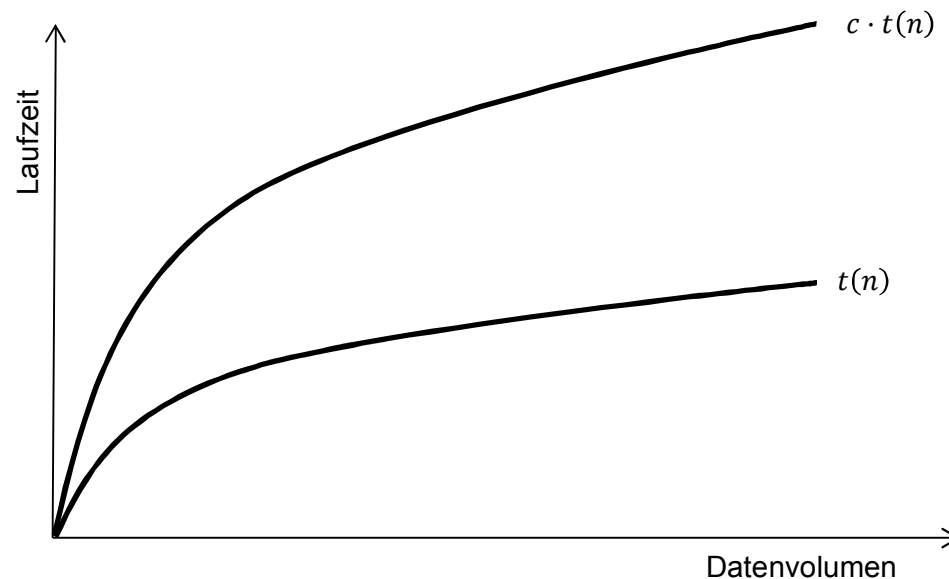


Welches Programm würden Sie in dieser Situation bevorzugen?

Wir erwarten nicht, dass eine Laufzeitfunktion »immer« besser sein muss als eine andere, sondern nur »fast immer« das heißt, ab einem bestimmten Wert n_0 , der beliebig, aber fest ist.

Worauf es beim Vergleich von Laufzeitfunktionen nicht ankommt

Sie haben ein Programm geschrieben haben, das Integer-Zahlen sortiert. Dieses Programm stellen Sie auf die Sortierung von Gleitkommazahlen um. Da ein Rechner Gleitkommazahlen nicht so effizient verarbeiten kann wie Integer-Zahlen, wird sich die Laufzeit des Programms durch diese Änderung um einen konstanten Faktor c verschlechtern:



Der Algorithmus ist durch diese Änderung aber nicht schlechter geworden. Es handelt sich nach wie vor um den gleichen Algorithmus mit dem gleichen "abstrakten" Laufzeitverhalten.

Ein konstanter Faktor soll beim Vergleich von Laufzeitfunktionen keine Rolle spielen.

Laufzeitklassen

Zur Beurteilung der Leistungsfähigkeit von Algorithmen benötigen wir ein Klassifizierungsschema für Laufzeitfunktionen, das von der konkreten Formel der Laufzeitfunktion abstrahiert, trotzdem aber die wesentlichen Informationen über das qualitative Verhalten der Funktion »im Unendlichen« enthält. Sehr hilfreich sind dafür die folgenden Begriffsbildungen:

Unter einer **Laufzeitfunktion** wollen wir im Folgenden stets eine nicht negative Funktion von den natürlichen Zahlen in die natürlichen Zahlen verstehen.

Für zwei Laufzeitfunktionen f und g schreiben wir $f \leq g$, wenn es eine Konstante $c > 0$ und eine natürliche Zahl n_0 so gibt, dass $f(n) \leq c \cdot g(n)$ für alle natürlichen Zahlen $n > n_0$ gilt

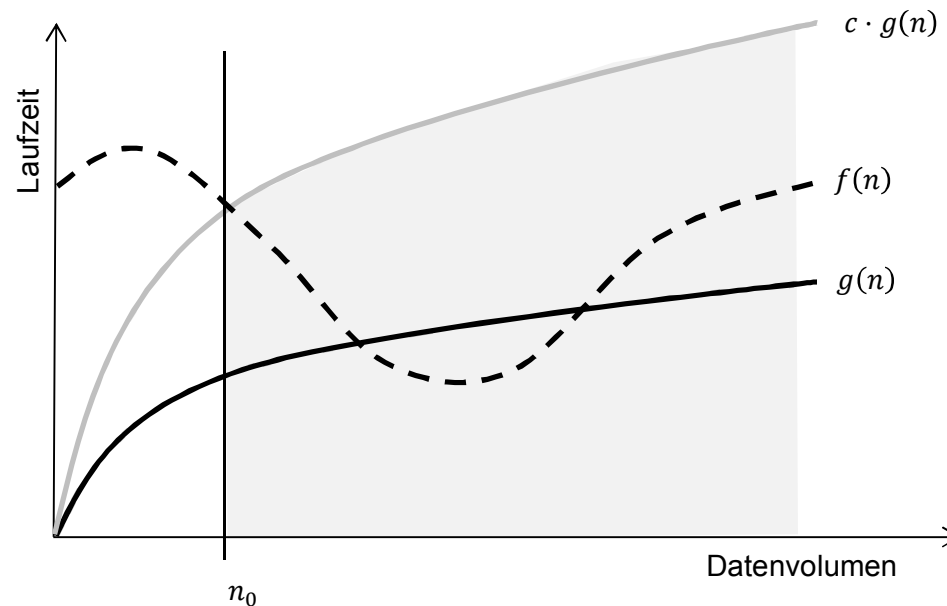
Gilt sowohl $f \leq g$ als auch $g \leq f$ so schreiben wir $f \approx g$.

Gilt $f \leq g$, aber nicht $f \approx g$, so schreiben wir auch $f < g$.

Mit $O(g)$ (sprich "Groß O von g") bezeichnen wir die Menge aller Funktionen f für die $f \leq g$ gilt. In diesem Sinne kann man statt $f \leq g$ auch $f \in O(g)$ schreiben. Weit verbreitet ist auch die Notation $f = O(g)$.

Mit $\Theta(g)$ (sprich "Groß Theta von g") bezeichnen wir die Menge aller Funktionen f für die $f \approx g$ gilt. In diesem Sinne kann man statt $f \approx g$ auch $f \in \Theta(g)$ schreiben. Weit verbreitet ist auch die Notation $f = \Theta(g)$.

Interpretation: $f \preccurlyeq g$ bedeutet, dass f nicht "wesentlich" schneller wächst als g .



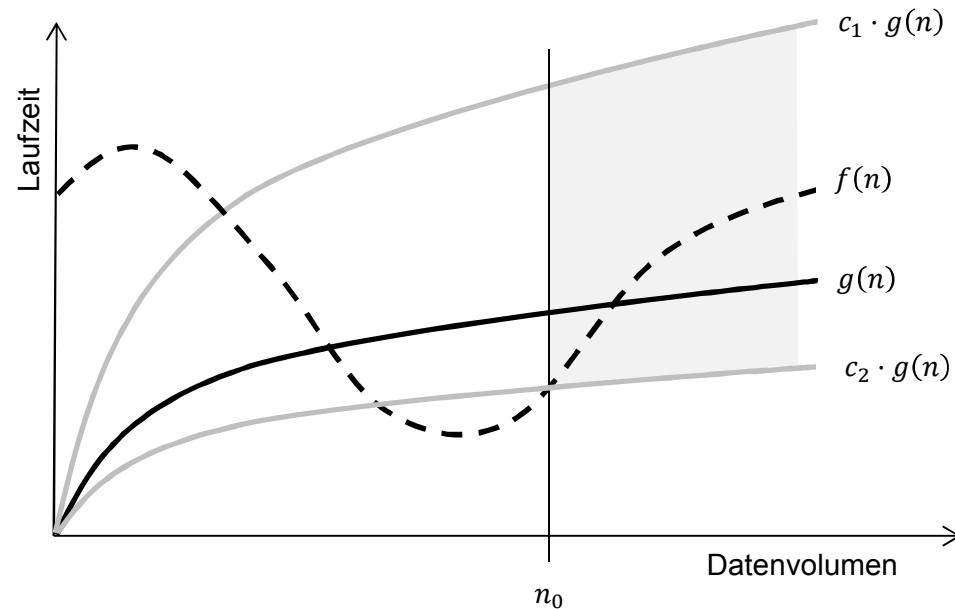
Es handelt sich um eine infinitesimale Eigenschaft. Darunter verstehen wir eine Eigenschaft, die sich erst "im Unendlichen zeigt", da es keinen Einfluss hat, wenn man f an endlich vielen Stellen abändert.

Trotz $f \preccurlyeq g$ kann f (teilweise oder sogar immer) größer als g sein. Es muss nur eine durch $c \cdot g(n)$ gegebene Schranke geben unterhalb derer sich f fast immer bewegt.

Da f aus dem durch g vorgegeben "Kanal" ab einer bestimmten Stelle nicht mehr nach oben ausbrechen kann hat f maximal das Wachstum von g .

f könnte auch langsamer wachsen als g , da es nach unten keine "Auffanglinie" gibt.

Interpretation: $f \approx g$ bedeutet, dass f und g "im Wesentlichen" gleich schnell wachsen.



Es handelt sich um eine infinitesimale Eigenschaft, die erhalten bleibt, wenn man f an endlich vielen Stellen abändert.

$f \approx g$ bedeutet, dass sich f fast immer in einem durch g vorgegebenen Kanal bewegt. Da f aus diesem "Kanal" ab einer bestimmten Stelle nicht mehr - weder nach oben noch nach unten - ausbrechen kann hat f das gleiche Wachstum wie g .

Die Konstanten c_1 , c_2 und n_0 müssen nicht "scharf" gewählt sein. Eine gewisse Willkür in der Wahl von c_1 , c_2 und n_0 führt oft zur Verwirrung.

Polynomiale Laufzeit (Abschätzung nach oben)

Wenn man sich auf das Wachstum von Laufzeitfunktionen konzentriert, kann man die Funktionen oft erheblich vereinfachen, indem man Funktionsterme eliminiert, die keinen wesentlichen Beitrag zum Wachstum der Funktion liefern. Wir stellen uns vor, dass wir die folgende Laufzeitfunktion zu einem Algorithmus ermittelt haben:

$$t(n) = n^4 + 3n^3 - 2n^2 + n - 3$$

Wir wollen diese Funktion nach oben abschätzen. Dazu lassen wir negative Terme einfach weg, da sie das Wachstum bremsen. Wir erhalten:

$$t(n) \leq n^4 + 3n^3 + n$$

Das können wir, wegen $n \geq 1$, weiter abschätzen:

$$t(n) \leq n^4 + 3n^4 + n^4 = 5n^4$$

Insgesamt haben wir damit erhalten:

$$t(n) \leq n^4$$

Da wir eine analoge Abschätzung für ein beliebiges Polynom durchführen können, erkennen wir, dass das Wachstum eines polynomialen Ausdrucks durch die höchste Potenz dominiert wird.

Polynomiale Laufzeit (Abschätzung nach unten)

Umgekehrt erhält man durch weglassen positiver Terme:

$$t(n) = n^4 + 3n^3 - 2n^2 + n - 3 \geq n^4 - 2n^2 - 3$$

Die negativen Terme vergrößern wir noch, indem wir zur dritten Potenz übergehen:

$$t(n) \geq n^4 - 2n^3 - 3n^3 = n^4 - 5n^3$$

Jetzt opfern wir die Hälfte der höchsten Potenz, um damit die niederen Potenzen zu eliminieren:

$$t(n) \geq \frac{1}{2}n^4 + \frac{1}{2}n^4 - 5n^3 = \frac{1}{2}n^4 + \frac{1}{2}n^3(n - 10)$$

Für $n \geq 10$ ist der letzte Term nicht negativ und kann weggelassen werden:

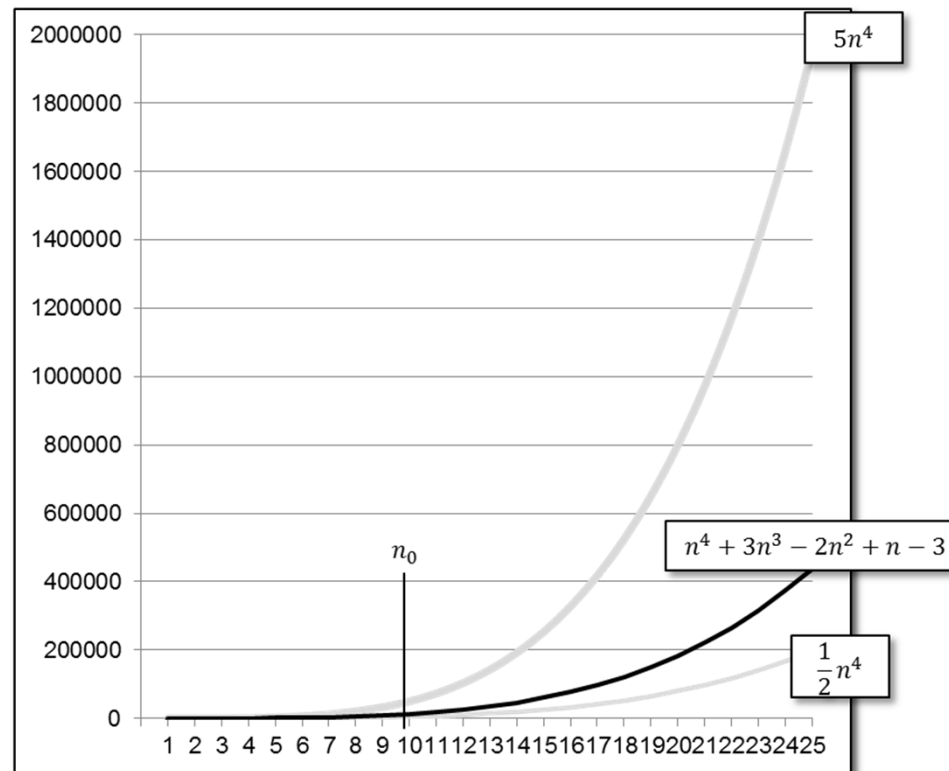
$$t(n) \geq \frac{1}{2}n^4 \text{ für } n \geq 10$$

Das bedeutet: $t(n) \geq n^4$

Abschätzungen enthalten immer eine gewisse Willkür. Man könnte hier durchaus exakter abschätzen. Aber das ist nicht nötig. Stellen Sie sich vor, dass Sie sich beim Bäcker ein Brötchen kaufen wollen. Durch einen flüchtigen Blick ins Portemonnaie sehen Sie, dass Sie noch Geldscheine haben. Dann würden Sie doch auch nicht anfangen, Ihr Kleingeld zu zählen, um festzustellen, ob es für ein Brötchen reicht.

Polynomiale Laufzeit (Zusammenfassung)

Insgesamt haben wir erhalten, dass ab $n \geq 10$ gilt: $\frac{1}{2}n^4 \leq t(n) \leq 5n^4$. Also: $t(n) \approx n^4$



Beachten Sie, dass wir analoge Abschätzungen für jedes Polynom durchführen können, sodass wir eine allgemein gültige Erkenntnis gewonnen haben:

Das Wachstum polynomialer Funktionen wird durch die höchste vorkommende Potenz bestimmt.

Polynomiale Laufzeitklassen

Für eine polynomiale Laufzeitfunktion

$$t(n) = a_k n^k + a_{k-1} n^{k-1} + \dots + a_2 n^2 + a_1 n + a_0 \text{ mit } a_k > 0$$

$$\text{gilt: } t(n) \approx n^k$$

Wir müssen bei polynomialen Laufzeitfunktionen also immer nur auf die höchste Potenz achten. Wegen des zusätzlichen Faktors n , den man durch keine Konstante einfangen kann haben höhere Potenzen ein echt größeres Wachstum. Die Laufzeitfunktionen im polynomialen Bereich sind also nach Potenzen geordnet:

$$1 < n < n^2 < n^3 < \dots < n^k < \dots$$

Das gleiche gilt auch für nicht ganzzahlige Potenzen – also auch für die Wurzeln ($\sqrt[k]{n} = n^{\frac{1}{k}}$). Auch hier "zählt" immer nur die höchste Potenz und es ist insgesamt:

$$1 < \dots < \sqrt[k]{n} < \dots < \sqrt[3]{n} < \sqrt{n} < n < n^2 < n^3 < \dots < n^k < \dots$$

Logarithmische und exponentielle Laufzeitklassen

Bei den Logarithmen muss man nur wissen, dass sich Logarithmen unterschiedlicher Basis nur durch einen konstanten Faktor unterscheiden. Somit haben alle Logarithmen ungeachtet ihrer Basis das gleiche Wachstumsverhalten, welches schwächer als jede Potenz ist. Wir können also unsere Kette wie folgt erweitern:

$$1 < \log(n) < \dots < \sqrt[k]{n} < \dots < \sqrt[3]{n} < \sqrt{n} < n < n^2 < n^3 < \dots < n^k < \dots$$

Am oberen Ende der Kette stehen die Exponentialfunktionen, die je nach Größe ihrer Basis unterschiedlich schnell wachsen und jede Potenz in Ihrem Wachstum übertreffen. Damit erhalten wir:

$$1 < \log(n) \dots < \sqrt[k]{n} < \dots < \sqrt[3]{n} < \sqrt{n} < n < n^2 < n^3 < \dots < n^k < \dots < 2^n < 3^n < \dots < k^n < \dots$$

Diese Kette zeigt nur einige wichtige Vertreter von Laufzeitfunktionen. Beliebige Funktionen mit gebrochener Basis (z.B. $\left(\frac{3}{2}\right)^n$) oder gebrochenem Exponenten (z.B.: $n^{\frac{3}{2}}$) oder Kombinationen dieser Grundtypen (z.B. $n \cdot \log(n)$) können vorkommen. Diese Funktionen bilden nur ein Gerüst, anhand dessen man weitere Laufzeitfunktionen einordnen kann:

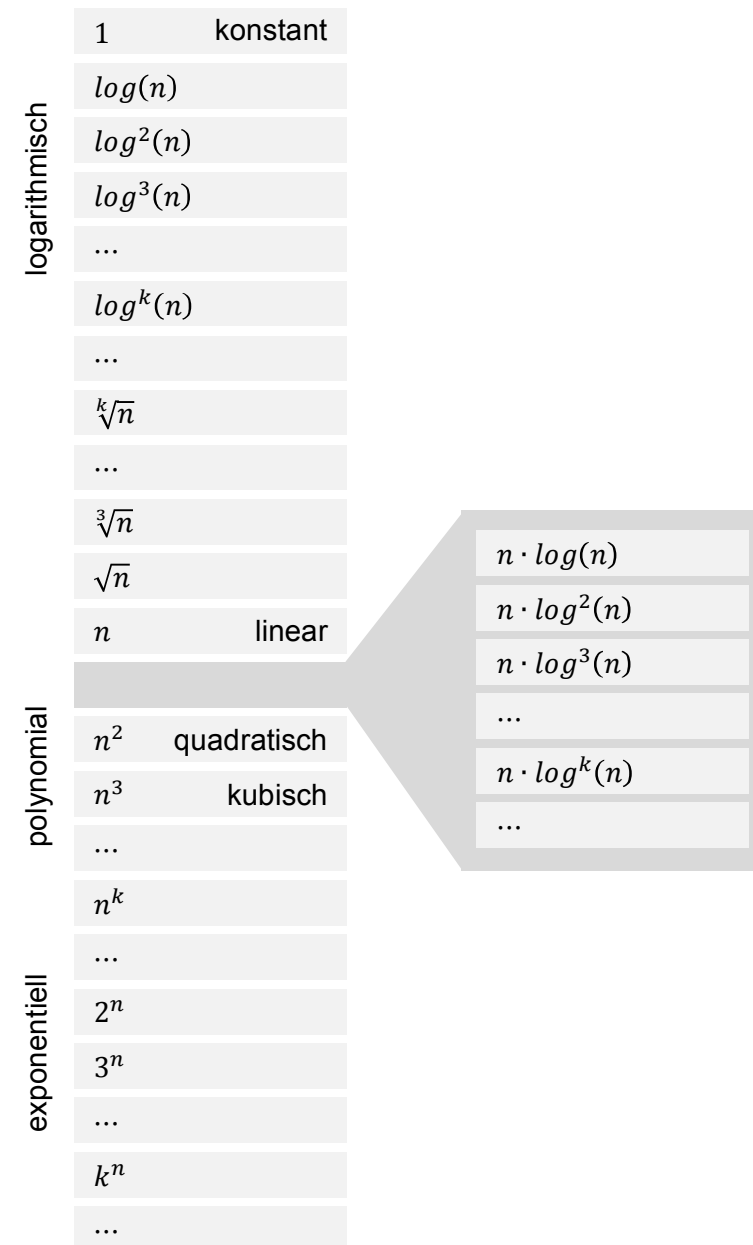
Klassifizierungsschema für Laufzeitfunktionen

Es ergibt sich ein Klassifizierungsschema für Laufzeitfunktionen. Dieses Schema ermöglicht eine Zuordnung von Laufzeitfunktionen zu sog. **Laufzeitklassen**.

Die Grafik zeigt wichtige Laufzeitklassen mit von oben nach unten zunehmender Laufzeitkomplexität.

Algorithmen sollten, damit sie untereinander vergleichbar sind, immer einer Laufzeitklasse zugeordnet werden.

Ziel ist es, zur Lösung eines Problems Algorithmen möglichst niedriger Laufzeitkomplexität zu finden.



Zusammenfassung der wichtigsten Laufzeitklassen

Algorithmen exponentieller Laufzeitkomplexität (untereinander abgestuft nach der Größe der Basis). Dies sind die Algorithmen mit inakzeptabel wachsendem Zeitbedarf. Der Programmierer sollte diese Algorithmen meiden, wo immer es möglich ist.

Algorithmen polynomialer Laufzeitkomplexität (untereinander abgestuft nach der höchsten vorkommenden Potenz). Dies sind Algorithmen mit einem akzeptabel wachsenden Zeitbedarf. Natürlich ist man hier immer bemüht, die höchste vorkommende Potenz so niedrig wie möglich zu halten.

Algorithmen logarithmischer Laufzeitkomplexität (untereinander gleichwertig, unabhängig von der Basis). Dies sind Algorithmen mit einem sehr moderaten Wachstum, die sich jeder Programmierer wünscht.

Algorithmen konstanter Laufzeit. Dies sind natürlich die besten Algorithmen. Nur kommen sie bei ernsthaften Problemen in der Regel nicht vor.

Beispiel 1

```
int programm1( int n)
{
    int i, k;
    int z = 0;

    for( i = 1; i <= n; i++)
    {
        for( k = 1; k <= i; k += (i/10 + 1))
            z++;
    }
    return z;
}
```

```
1: 1
2: 3
3: 6
4: 10
5: 15
6: 21
7: 28
8: 36
9: 45
10: 50
11: 56
12: 62
13: 69
14: 76
15: 84
```

Welche Laufzeitklasse vermuten Sie?

Beispiel 1

```
int programm1( int n)
{
    int i, k;
    int z = 0;

    for( i = 1; i <= n; i++)
    {
        for( k = 1; k <= i; k += (i/10 + 1))
            z++;
    }
    return z;
}
```

```
1: 1
2: 3
3: 6
4: 10
5: 15
6: 21
7: 28
8: 36
9: 45
10: 50
11: 56
12: 62
13: 69
14: 76
15: 84
```

Analyse

Die äußere Schleife dieses Programms wird linear (1-n) durchlaufen, die innere dagegen höchstens 11 mal.

Daraus folgt: $n \leq t(n) \leq 11n$.

Das Programm ist also linear: $t(n) \approx n$.

Beispiel 2

```
int programm2( int n)
{
    int i, k;
    int z = 0;

    for( i = 1; i <= n; i *= 2)
    {
        for( k = 1; k <= i; k++)
            z++;
    }
    return z;
}
```

1:	1
2:	3
3:	3
4:	7
5:	7
6:	7
7:	7
8:	15
9:	15
10:	15
11:	15
12:	15
13:	15
14:	15
15:	15

Welche Laufzeitklasse vermuten Sie?

Beispiel 2

```
int programm2( int n)
{
    int i, k;
    int z = 0;

    for( i = 1; i <= n; i *= 2)
    {
        for( k = 1; k <= i; k++)
            z++;
    }
    return z;
}
```

1:	1
2:	3
3:	3
4:	7
5:	7
6:	7
7:	7
8:	15
9:	15
10:	15
11:	15
12:	15
13:	15
14:	15
15:	15

Analyse

Wenn man sich die äußere Schleife wegdenkt und stattdessen einfach den Maximalwert $i = n$ annimmt, sieht man, dass das Programm mindestens linear ist.

Andererseits verdoppelt i mit jedem Schritt in der äußeren Schleife seinen Wert, erreicht also das Schleifenende nach $\log_2(n)$ Schritten. Wir stellen uns jetzt vor, dass wir im s -ten dieser Schritte sind. Dann hat i den Wert 2^s .

Dann wurden in der Inneren Schleife bisher $1 + 2 + 2^2 + \dots + 2^s$ Schritte durchgeführt. Nach der Summenformel der geometrischen Reihe ist:

$$1 + 2 + 2^2 + \dots + 2^s = \frac{2^{s+1} - 1}{2 - 1} \leq 2^{s+1} = 2 \cdot 2^s$$

Da es maximal $\log_2(n)$ Schritte gibt ist $t(n) \leq 2 \cdot 2^{\log_2(n)} = 2n$

Insgesamt ist also $n \leq t(n) \leq 2n$. Daher ist dieses Programm linear: $t(n) \approx n$.

Beispiel 3

```
int programm3( int n)
{
    int i, k;
    int z = 0;

    for( i = 1; i <= n; i++)
    {
        for( k = 1; k <= i; k *= 2)
            z++;
    }
    return z;
}
```

```
1: 1
2: 3
3: 5
4: 8
5: 11
6: 14
7: 17
8: 21
9: 25
10: 29
11: 33
12: 37
13: 41
14: 45
15: 49
```

Welche Laufzeitklasse vermuten Sie?

Beispiel 3

```
int programm3( int n)
{
    int i, k;
    int z = 0;

    for( i = 1; i <= n; i++)
    {
        for( k = 1; k <= i; k *= 2)
            z++;
    }
    return z;
}
```

1:	1
2:	3
3:	5
4:	8
5:	11
6:	14
7:	17
8:	21
9:	25
10:	29
11:	33
12:	37
13:	41
14:	45
15:	49

Analyse

In der inneren Schleife gibt es $\log_2(i)$ Durchläufe. Das bedeutet, wenn man die Formel $\log(a) + \log(b) = \log(a \cdot b)$ iteriert anwendet:

$$t(n) \approx \log_2(1) + \log_2(2) + \dots \log_2(n) = \log_2(n!)$$

Mit den Abschätzungen $n^{\frac{n}{2}} \leq n! \leq n^n$ folgt dann einerseits:

$$t(n) \approx \log_2(n!) \leq \log_2(n^n) = n \cdot \log_2(n)$$

und andererseits

$$t(n) \approx \log_2(n!) \geq \log_2\left(n^{\frac{n}{2}}\right) = \frac{n}{2} \cdot \log_2(n)$$

Insgesamt ist daher: $t(n) \approx n \cdot \log(n)$


Vergleich von Programm1 und Programm3

Obwohl Programm3 wegen $n < n \cdot \log(n)$ in einer schlechteren Laufzeitklasse ist, als Programm1, hat man bei Betrachtung der Bildschirmausgaben den gegenteiligen Eindruck:


Das liegt daran, dass die Entscheidung erst "im Unendlichen" fällt.

Erst bei $n = 1919$ entscheidet sich, welche Funktion die größere ist.

Programm 1
Laufzeitklasse: n



1:	1	1
2:	3	3
3:	6	5
4:	10	8
5:	15	11
6:	21	14
7:	28	17
8:	36	21
9:	45	25
10:	50	29
11:	56	33
12:	62	37
13:	69	41
14:	76	45
15:	84	49



Programm 3
Laufzeitklasse: $n \cdot \log(n)$

1915:	19033	19029
1916:	19043	19040
1917:	19053	19051
1918:	19063	19062
1919:	19073	19073
1920:	19083	19084
1921:	19093	19095
1922:	19103	19106
1923:	19113	19117
1924:	19123	19128
1925:	19133	19139

Beispiel 4

```
int programm4( int n)
{
    int i, k;
    int z = 0;

    for( i = 1; i <= n; i++)
    {
        for( k = 1; k <= i*i; k++)
            z++;
    }
    return z;
}
```

```
1: 1
2: 5
3: 14
4: 30
5: 55
6: 91
7: 140
8: 204
9: 285
10: 385
11: 506
12: 650
13: 819
14: 1015
15: 1240
```

Welche Laufzeitklasse vermuten Sie?

Beispiel 4

```
int programm4( int n)
{
    int i, k;
    int z = 0;

    for( i = 1; i <= n; i++)
    {
        for( k = 1; k <= i*i; k++)
            z++;
    }
    return z;
}
```

```
1: 1
2: 5
3: 14
4: 30
5: 55
6: 91
7: 140
8: 204
9: 285
10: 385
11: 506
12: 650
13: 819
14: 1015
15: 1240
```

Analyse

In der inneren Schleife gibt es immer i^2 Durchläufe, sodass es insgesamt:

$$t(n) = 1 + 2^2 + 3^2 + \dots + n^2 = \frac{n(n+1)(2n+1)}{6}$$

Durchläufe gibt.

Es ist also: $t(n) \approx n^3$.

Beispiel 5

```
int programm5( int n)
{
    int i, k;
    int z = 0;

    for( i = 1; i <= n; i *= 2)
    {
        for( k = 1; k <= i; k *= 2)
            z++;
    }
    return z;
}
```

```
1: 1
2: 3
3: 3
4: 6
5: 6
6: 6
7: 6
8: 10
9: 10
10: 10
11: 10
12: 10
13: 10
14: 10
15: 10
```

Welche Laufzeitklasse vermuten Sie?

Beispiel 5

```
int programm5( int n)
{
    int i, k;
    int z = 0;

    for( i = 1; i <= n; i *= 2)
    {
        for( k = 1; k <= i; k *= 2)
            z++;
    }
    return z;
}
```

1:	1
2:	3
3:	3
4:	6
5:	6
6:	6
7:	6
8:	10
9:	10
10:	10
11:	10
12:	10
13:	10
14:	10
15:	10

In beiden Schleifen wachsen die Schleifenindices exponentiell, erreichen also sehr schnell ihr Ziel. Die äußere Schleife benötigt, wegen der Verdopplung $\log_2(n)$ Schritte, wobei beim s-ten Schritt i den Wert $i = 2^s$ hat. In der inneren Schleife benötigt die Variable k dann, ebenfalls wegen der Verdopplung, s Schritte, um diesen Wert von i zu erreichen. Das heißt, im s-ten Schleifendurchlauf der äußeren Schleife macht die innere Schleife genau s Durchläufe. Da die äußere Schleife $\log_2(n)$ Durchläufe macht ergibt das:

$$t(n) = 1 + 2 + 3 + \dots + \log_2(n) = \frac{\log_2(n)(\log_2(n) + 1)}{2} = \frac{1}{2}(\log_2^2(n) + \log_2(n))$$

Also: $t(n) \approx \log^2(n)$

↖
Bis zum nächst kleineren ganzzahligen Wert

Dieses Programm ist das mit der niedrigsten Laufzeitkomplexität unter den sechs Beispielen.

Beispiel 6

```
int programm6( int n)
{
    int i, k, m;
    int z = 0;

    for( i = 1, m = 1; i <= n; i++, m *= 2)
    {
        for( k = 1; k <= m; k++)
            z++;
    }
    return z;
}
```

```
1: 1
2: 3
3: 7
4: 15
5: 31
6: 63
7: 127
8: 255
9: 511
10: 1023
11: 2047
12: 4095
13: 8191
14: 16383
15: 32767
```

Welche Laufzeitklasse vermuten Sie?

Beispiel 6

```
int programm6( int n)
{
    int i, k, m;
    int z = 0;

    for( i = 1, m = 1; i <= n; i++, m *= 2)
    {
        for( k = 1; k <= m; k++)
            z++;
    }
    return z;
}
```

```
1: 1
2: 3
3: 7
4: 15
5: 31
6: 63
7: 127
8: 255
9: 511
10: 1023
11: 2047
12: 4095
13: 8191
14: 16383
15: 32767
```

Analyse

Hier muss der Zähler k einem exponentiell wachsendem m hinterherlaufen. Die Variable m hat immer den Wert $m = 2^{i-1}$. In der inneren Schleife werden also immer $m = 2^{i-1}$ Durchläufe ausgeführt. Damit ist:

$$t(n) = 1 + 2 + 2^2 + 2^3 + \dots + 2^{n-1} = \frac{2^n - 1}{2 - 1} = 2^n - 1 \approx 2^n$$

Dieses Programm ist das mit der höchsten Laufzeitkomplexität unter den sechs Beispielen.

Man könnte argumentieren, dass es eigentlich egal ist, welcher Leistungsklasse ein Algorithmus angehört, da unsere Rechner immer schneller werden und irgendwann so schnell sein werden, dass die Frage nach der Effizienz von Algorithmen zu den Akten gelegt werden kann. Dem kann man zweierlei entgegenhalten. Zum einen ist Effizienz ein grundsätzlicher Wert, den man immer anstreben sollte, denn auch auf einem schnelleren Rechner bleibt ein »guter« Algorithmus besser als ein »schlechter«. Schnelle Rechner machen aus schlechten Programmen keine guten Programme. Ein zweites Argument ist aber noch gewichtiger. Wir stellen uns vor, dass ein heutiger Rechner bei einer bestimmten Zeitvorgabe x Datenelemente bearbeiten kann und fragen uns, wie viele Elemente er schafft, wenn der Rechner 10, 100 oder 1000 mal schneller wird. Das Ergebnis hängt natürlich von der Laufzeitklasse des Algorithmus ab:

Laufzeitklasse	Heutiger Rechner	10 mal schnellerer Rechner	100 mal schnellerer Rechner	1000 mal schnellerer Rechner
$\log(n)$	x	x^{10}	x^{100}	x^{1000}
n	x	$10 \cdot x$	$100 \cdot x$	$1000 \cdot x$
n^2	x	$3.16 \cdot x$	$10 \cdot x$	$31.6 \cdot x$
2^n	x	$x + 3.32$	$x + 6.64$	$x + 9.96$
10^n	x	$x + 1$	$x + 2$	$x + 3$

Die Tabelle zeigt, dass selbst eine Vertausendfachung der Rechnerleistung nur geringe Gewinne im Bereich der exponentiell wachsenden Algorithmen bringt. Selbst ein 1000-mal schnellerer Rechner schafft es nur, ein Problem der Leistungsklasse 2^n für knapp 10 Elemente mehr in der gleichen Zeitvorgabe zu lösen. Für uns bedeutet dies, dass die Suche nach Algorithmen niedriger Zeitkomplexität immer ein wichtiges Anliegen der Programmierung sein wird und es keinen Sinn macht, auf zukünftige Rechner zu warten. Unser Ziel muss immer sein, einen Algorithmus in eine möglichst optimale Leistungsklasse zu bringen.