

Kapitel 8: Komplexitätsanalyse

- Effizienz von Algorithmen
- Messung der CPU-Zeit
- Laufzeiten von Algorithmen
- Größenordnungen von Laufzeiten
- Beispiele
- Worst-Case, Average-Case und Best-Case-Analyse
- Analyse von rekursiven Funktionen
- Analyse von Teile-und-Herrsche-Funktionen

Effizienz von Algorithmen

Ziel

- Bestimmung der **Effizienz** eines Algorithmus, d.h.
 - Laufzeitbedarf und
 - Speicherplatzbedarf
- In diesem Kapitel nur Betrachtung der Laufzeit.
Speicherplatzbedarf ist hier von nicht so großem Interesse.

Typische Aufgabenstellung

- Es stehen mehrere Algorithmen zur Verfügung, die alle dasselbe Problem lösen.
Wähle den effizientesten Algorithmus.
- Beispiel: Es gibt zahlreiche Sortierverfahren (später).
Welches Sortierverfahren ist am effizientesten?

Ansätze

- Messung der **CPU-Zeit**
- Ermittlung der **Größenordnung der Laufzeit** durch Analyse des Algorithmus

Messung der CPU-Zeit

Vorgehensweise

Implementiere den Algorithmus in einer konkreten Programmiersprache (z.B. Java) und messe CPU-Zeit (z.B. mit Funktion `System.nanoTime`) auf einem konkreten Rechner:

```
public static void main(String[] args) {  
    // ...  
    long start = System.nanoTime(); // aktuelle Zeit in nsec  
    sort(x);  
    long end = System.nanoTime();  
    double elapsedTime = (double)(end-start)/1.0e06; // Zeit in msec  
    System.out.println(" Elapsed time in " + elapsedTime + " msec");  
}
```

Vorteil:

Es werden konkrete Zeiten ermittelt

Nachteil:

Man erhält keine allgemeingültigen Aussagen:

- Zeiten beziehen sich auf bestimmte Laufzeitumgebung:
Rechner, Betriebssystem, Compiler.
- Zeiten lassen sich nur für wenige Eingabewerte ermitteln.
Keine Aussagen über andere Eingabewerte möglich.

Laufzeiten von Algorithmen (1)

Elementare Operationen

- Ein Algorithmus führt eine Folge von elementaren Operationen aus:
Addition, Subtraktion, Multiplikation, Division, Vergleiche, Zuweisungen, Feldzugriffe, etc.
- Eine elementare Operation kann in einer konstanten Zeit ausgeführt werden.
Der Einfachheit wegen wird ein **Einheitskostenmaß** angenommen,
d.h. alle elementaren Operationen benötigen die gleiche Zeit.

Laufzeit

- Die Laufzeit eines Algorithmus wird definiert als:
$$T(n) = \text{Anzahl der elementaren Operationen, die der Algorithmus durchläuft, um eine Eingabe der Größe } n \text{ zu bearbeiten.}$$
- Man beachte, dass oft die Angabe der Laufzeit in Abhängigkeit von konkreten Eingabe-Werten unzweckmäßig ist.
- Wir abstrahieren daher von den konkreten Eingabe-Werten und betrachten die Laufzeit in der Abhängigkeit von der Größe (dem Umfang) der Eingabe.
- Beispiel: bei einem Sortieralgorithmus wird die Laufzeit bestimmt, die notwendig ist um n Zahlen zu sortieren (n ist die Größe der Eingabe).

Laufzeiten von Algorithmen (2)

Beispiel

- Betrachtet werden zwei Algorithmen zur Berechnung von:

$$\text{sum}(n) = \sum_{i=1}^n i$$

- Algorithmus sum1 berechnet die Summe in einer Schleife, während Algorithmus sum2 die bekannte Formel $\text{sum}(n) = n \cdot (n+1) / 2$ ausnutzt.

```
public static int sum1(int n)
{
    int s = 0;
    for (int i = 1; i <= n; i++)
        s = s + i;
    return s;
}
```

```
public static int sum2(int n)
{
    int s;
    s = n * (n+1) / 2;
    return s;
}
```

Analyse der Laufzeiten

- sum1 benötigt $2n$ Additionen, $n+2$ Zuweisungen und $n+1$ Vergleiche.
Damit $T(n) = 4n + 3$.
- sum2 benötigt eine Addition, eine Division, eine Multiplikation und eine Zuweisung.
Damit $T(n) = 4$.

Größenordnungen von Laufzeiten (1)

Ziel

- Die Konstanten in den Laufzeiten ergeben sich aus der etwas unrealistischen Annahme, dass alle elementaren Operationen die gleiche Zeit benötigen.
- Würde eine Multiplikation zweimal so stark wie eine Addition gewichtet werden, würden sich andere Konstanten ergeben.
- Die Konstanten sind also nicht sonderlich aussagekräftig. Wir werden uns daher mit Hilfe der O-Notation von den Konstanten befreien.

Definition O-Notation

$f(n)$ ist in der Größenordnung von $g(n)$

$$f(n) = O(g(n)),$$

falls Konstante c und n_0 existieren, so daß

$$f(n) \leq c \cdot g(n) \quad \text{für } n \geq n_0.$$

Beispiel Algorithmus sum1:

- $T(n) = 4 \cdot n + 3 \leq 5 \cdot n$ für $n \geq 3$.
- Damit ist $T(n) = O(n)$. „ $T(n)$ ist linear“.

Größenordnungen von Laufzeiten (2)

Möglichst einfache und kleine Größenordnungen

- Die Laufzeit sollte mit einer möglichst einfachen und kleinen Größenordnung (siehe auch Tabelle auf der nächsten Seite) abgeschätzt werden.
- Beispiel: für $T(n) = 4n+3$ ist

$$T(n) = O(n)$$

besser als

$$T(n) = O(n^2) \text{ oder}$$

$$T(n) = O(5n)$$

Polynomielle Größenordnung

- Bei der Analyse von Laufzeiten ergibt sich oft eine polynomielle Funktion:

$$T(n) = a_0 + a_1n + \dots + a_kn^k$$

- Dann läßt sich $T(n)$ wie folgt abschätzen:

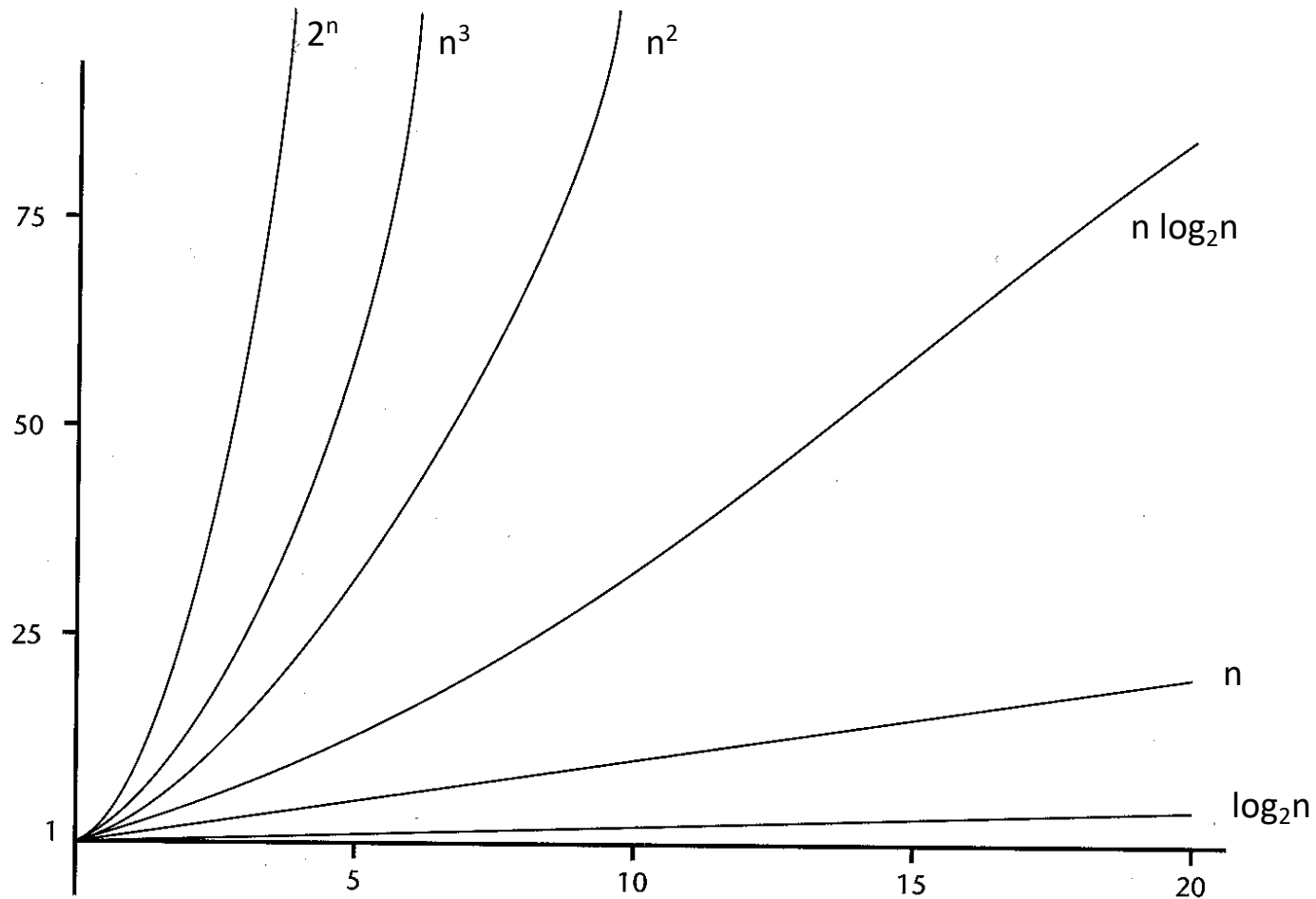
$$T(n) = O(n^k)$$

Typische Größenordnungen (1)

Größen- ordnung der Laufzeit $T(n)$	Verhalten der Laufzeit $T(n)$, falls sich n verdoppelt	Bezeichnung
$O(1)$	ändert sich nicht	konstant
$O(\log_2 n)$	wächst um eine kleine Konstante	logarithmisch
$O(\sqrt{n})$	wächst um Faktor $\sqrt{2}$	
$O(n)$	verdoppelt sich	linear
$O(n \log_2 n)$	wird etwas mehr als doppelt so groß	
$O(n^2)$	wächst um Faktor 4	quadratisch
$O(n^3)$	wächst um Faktor 8	kubisch
$O(n^k)$	wächst um Faktor 2^k	polynomiell
$O(2^n)$	wächst sehr rasch	exponentiell

Typische Größenordnungen (2)

Graphische Darstellung einiger Größenordnungen



Aufgaben (1)

Aufgabe 8.1

Analysieren Sie die Laufzeiten für folgende Programmstücke:

```
for (i = 0; i < n; i++)  
    for (j = i; j < n; j++)  
        a[i][j] = 1;
```

```
// Einheitsmatrix:  
for (i = 0; i < n; i++)  
    for (j = 0; j < n; j++)  
        a[i][j] = 0;  
for (i = 0; i < n; i++)  
    a[i][i] = 1;
```

```
// Matrixmultiplikation:  
for (i = 0; i < n; i++) {  
    for (j = 0; j < n; j++) {  
        c[i][j] = 0;  
        for (k = 0; k < n; k++)  
            c[i][j] += a[i][k] * b[k][j];  
    }  
}
```

Aufgaben (2)

Aufgabe 8.2

- Analysieren Sie die folgenden beiden Algorithmen zur Auswertung eines Polynoms n-ten Grades:

$$p(x) = a_0 + a_1x + \dots + a_nx^n$$

- Der Algorithmus **polyNaiv** berechnet jede einzelne Potenz iterativ und addiert die Potenzen mittels einer Schleife zusammen.

```
static double polyNaiv(double[] a, int n, double x) {  
  
    double s = a[0];  
    for (int i = 1; i <= n; i++) {  
        // p = x hoch i berechnen:  
        double p = 1.0;  
        for (int j = 1; j <= i; j++)  
            p = p * x;  
        s = s + a[i] * p;  
    }  
    return s;  
}
```

Aufgaben (3)

- Der Algorithmus `polyHorner` verwendet das Horner-Schema:

$$\begin{aligned} p(x) &= a_0 + a_1x + a_2x^2 + \dots + a_nx^n \\ &= a_0 + x * (a_1 + x * (a_2 + \dots + x * (a_{n-1} + x * a_n) \dots)) \end{aligned}$$

- `polyHorner` berechnet den oberen Ausdruck von innen nach außen mittels einer Schleife.

```
static double polyHorner(double[] a, int n, double x) {  
  
    double s = 0;  
  
    for (int i = n; i >= 0; i--)  
        s = s*x + a[i];  
  
    return s;  
}
```

Aufgaben (4)

Aufgabe 8.3

- Durch eine Analyse werden für zwei Algorithmen, die das gleiche Problem lösen, folgende Laufzeiten ermittelt:

Algorithmus A: $T_A(n) = C_1 \cdot n \cdot \log_2 n = O(n \log_2 n)$ und

Algorithmus B: $T_B(n) = C_2 \cdot n^2 = O(n^2)$.

- Außerdem ergibt eine Laufzeitmessung auf einem konkreten Rechner für eine Eingabe der Größe $n = 1000$ folgende CPU-Zeiten:

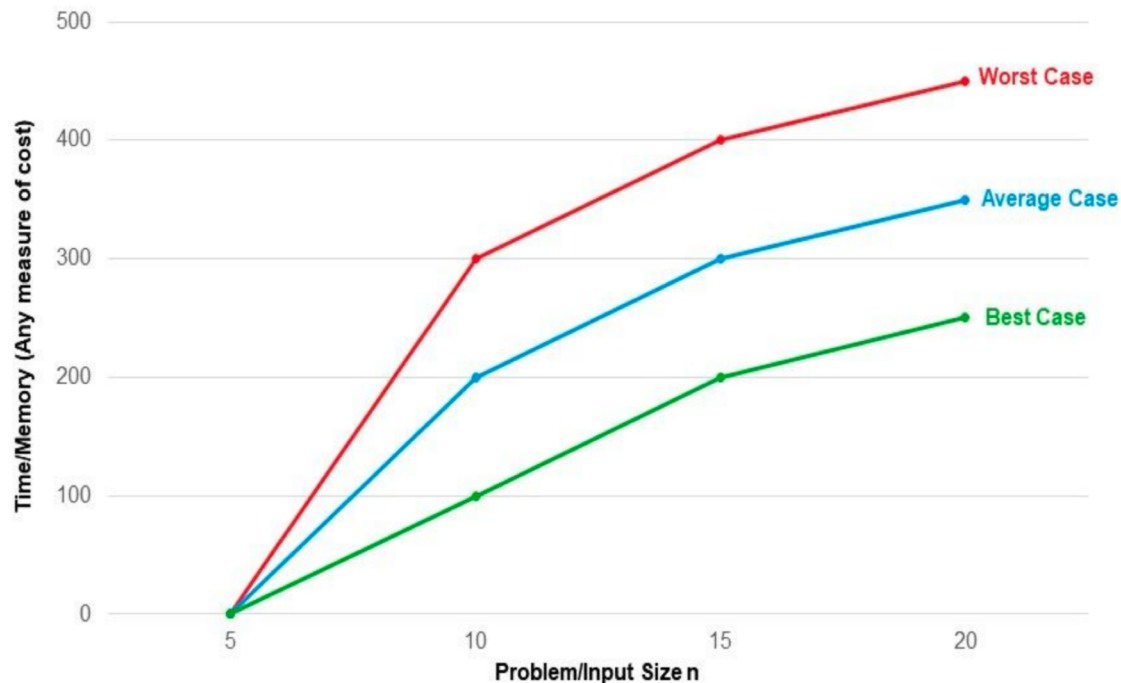
Algorithmus A: 2 msec

Algorithmus B: 1 msec

- Skizzieren Sie $T_A(n)$ und $T_B(n)$ in einem Schaubild.
- Welche CPU-Zeiten erwarten Sie für $n = 10^6$?

Worst-Case, Average-Case und Best-Case

- In vielen Fällen hängt die Laufzeit nicht nur von der Größe der Eingabe sondern auch von den konkreten Eingabewerten ab.
- Beispiel: Die Laufzeit einiger Sortierv Verfahren ist für ein bereits sortiertes Feld um eine Größenordnung schneller als für ein ungeordnetes Feld.



<https://www.slideteam.net/graphical-representation-of-best-average-and-worst-case.html>

Worst-Case-Analyse

- Eingaben betrachten, bei der Laufzeiten **maximal** sind.
- Wird am häufigsten durchgeführt.

Average-Case-Analyse

- **Mittelwert** der Laufzeiten für alle Eingaben der Größe n bilden.
- In der Praxis sehr nützlich, technisch aber schwieriger durchzuführen.

Best-Case-Analyse

- Eingaben betrachten, bei der Laufzeiten **minimal** sind.
- Wird eher selten durchgeführt.

Beispiel: Sequentielle Suche (1)

```
static int suche(int[] a, int x) {  
  
    int n = a.length;  
    int i = 0;  
    while(i < n) {  
        if (x == a[i])  
            break;  
        i++;  
    }  
    return i;  
}
```

Funktion sucht x in einem Feld
a[0], a[1], ... a[n-1] und
liefert i, falls x == a[i] und sonst n.

Best-Case-Analyse

- x befindet sich am Anfang des Feldes, d.h. $x == a[0]$.
- Es werden 5 Operationen benötigt: $n = a.length$, $i=0$, $i < n$, $a[i]$, $x == a[i]$.
- Damit: $T(n) = 5 = O(1)$

Worst-Case-Analyse

- x kommt nicht im Feld a vor.
- Damit: $T(n) = 4n + 3 = O(n)$

Beispiel: Sequentielle Suche (2)

```
static int suche(int[] a, int x) {  
  
    int n = a.length;  
    int i = 0;  
    while(i < n) {  
        if (x == a[i])  
            break;  
        i++;  
    }  
    return i;  
}
```

Average-Case-Analyse

- Es gibt $n+1$ unterschiedliche Fälle:

$x == a[0], x == a[1], \dots x == a[n-1]$ oder x kommt in a nicht vor.

- Damit:

$$\begin{aligned} T(n) &= \{ 0 \cdot 4 + 5 + 1 \cdot 4 + 5 + 2 \cdot 4 + 5 + \dots + (n-1) \cdot 4 + 5 + n \cdot 4 + 3 \} / (n+1) \\ &= \{ 4 \cdot (1+2+\dots+n) + 5n+3 \} / (n+1) \\ &= \{ 2n(n+1) + 5n+3 \} / (n+1) \\ &\approx 2n + 5 \\ &= O(n). \end{aligned}$$

Aufgabe

Aufgabe 8.4

Die Funktion `istPrim` prüft, ob `n` eine Primzahl ist.
Führen Sie eine Worst-Case-Analyse durch.

```
static boolean istPrim(int n) {  
  
    for (int i = 2; i*i <= n; i++) {  
        if (n%i == 0) // i ist Teiler von n  
            return false;  
    }  
    return true;  
}
```

Analyse von rekursiven Funktionen

Aufrufstruktur-Methode

- Stelle Aufrufstruktur dar.
- Ermittle Laufzeit für jeden Aufruf und addiere alle Laufzeiten zusammen

Beispiel fak-Funktion

```
static int fak(int n) {  
    if (n == 0)  
        return 1;  
    else  
        return n * fak(n-1);  
}
```

Aufrufstruktur:

fak(n)
|
fak(n-1)
|
...
|
fak(1)
|
fak(0)

Laufzeiten:

c_1
 c_1
...
 c_1
 c_0

Summe aller Laufzeiten:

$$T(n) = c_1 n + c_0 = O(n).$$

Analyse von Teile-und-Herrsche-Funktionen (1)

Schema für Teile-und-Herrsche-Funktion mit zwei rekursiven Aufrufen

Funktion $f(x,n)$ {

Die Funktion f bearbeitet die Eingabe x
der Größe n nach dem Teile-und-Herrsche-Prinzip.

```
if (n == 1) {  
    // Basisfall:
```

```
löse Problem direkt; Ergebnis sei lös; // (1)
return lös;
```

}

```
else {
```

// Teileschritt:

teile x in zwei Teilprobleme x_1 und x_2 jeweils der Größe $n/2$; // (2)

```
lös1 = f(x1,n/2);
```

```
lös2 = f(x2,n/2);
```

// Herrscheschrift:

```
Setze Lösung lös für x aus lös1 und lös2 zusammen;           // (3)
```

```
return lös;
```

}

}

Analyse von Teile-und-Herrsche-Funktionen (2)

Schema für Teile-und-Herrsche-Funktion mit linearer Aufrufkette

Funktion $f(x,n)$ {

Die Funktion f bearbeitet die Eingabe x
der Größe n nach dem Teile-und-Herrsche-Prinzip.

if ($n == 1$) {

// Basisfall:

löse Problem direkt; Ergebnis sei lös; // (1)

return lös;

}

else {

// Teileschritt:

reduziere x auf ein Teilproblem x_1 der Größe $n/2$; // (2)

lös1 = $f(x_1, n/2)$;

// Herrscheschritt:

Berechne Lösung lös für x aus lös1; // (3)

return lös;

}

}

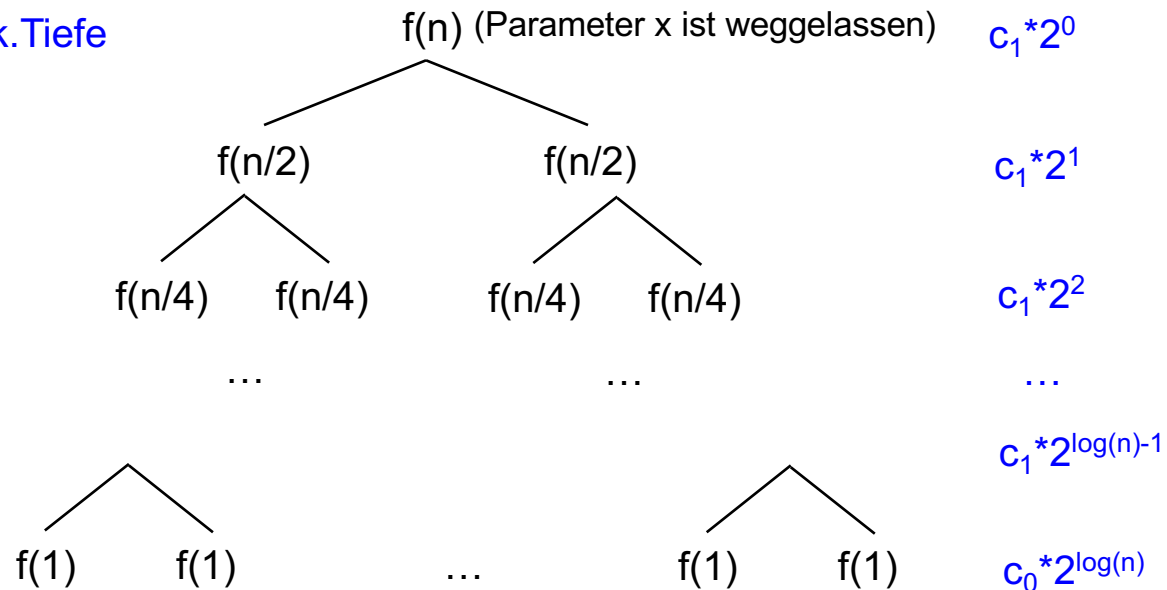
Analyse von Teile-und-Herrsche-Funktionen (3)

Laufzeiten für verschiedene Arten von Teile-und-Herrsche-Funktionen

	Art der Teile-und-Herrsche-Funktion	Laufzeit
A	Ein Problem der Größe n wird in zwei Teilprobleme der Größe $n/2$ zerlegt. Basisfall (1): konstante Laufzeit. Teileschritt (2) und Herrsche-Schritt (3): konstanter Aufwand .	$O(n)$
B	Ein Problem der Größe n wird in zwei Teilprobleme der Größe $n/2$ zerlegt. Basisfall (1): konstante Laufzeit. Teileschritt (2) und Herrsche-Schritt (3): linearer Aufwand .	$O(n \log n)$
C	Ein Problem der Größe n wird in nur einem Teilproblem der Größe $n/2$ zerlegt. (Aufrufstruktur bildet eine lineare Kette). Basisfall (1): konstante Laufzeit. Teileschritt (2) und Herrsche-Schritt (3): konstanter Aufwand .	$O(\log n)$
D	Ein Problem der Größe n wird in nur einem Teilproblem der Größe $n/2$ zerlegt. (Aufrufstruktur bildet eine lineare Kette). Basisfall (1): konstante Laufzeit. Teileschritt (2) und Herrsche-Schritt (3): linearer Aufwand .	$O(n)$

Analyse für Fall A

Maximale Rek.Tiefe
 $= \log_2(n)$

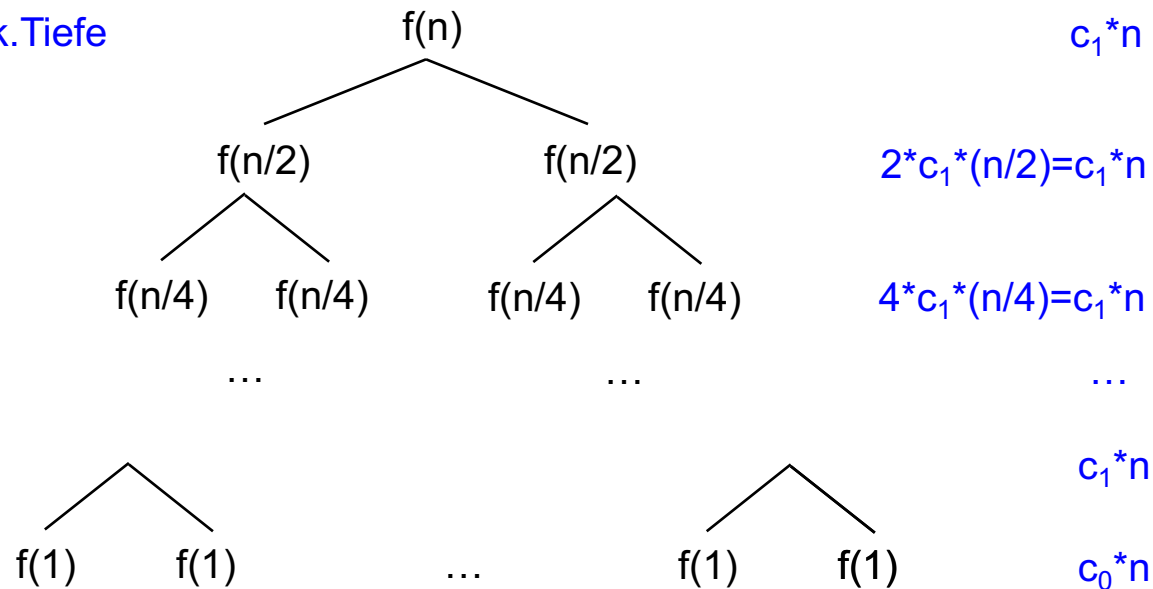


Laufzeiten für jede
Rekursionstiefe

Summe aller Laufzeiten:

$$\begin{aligned} T(n) &= c_1 * (2^0 + 2^1 + 2^2 + \dots + 2^{\log(n)-1}) + c_0 * 2^{\log(n)} \\ &= c_1 * (2^{\log(n)} - 1) + c_0 * n \\ &= c_1 * (n - 1) + c_0 * n \\ &= O(n). \end{aligned}$$

Maximale Rek.Tiefe
= $\log_2(n)$



Laufzeiten für jede Rekursionstiefe

Summe aller Laufzeiten:

$$T(n) = c_1 * n * \log(n) + c_0 * n$$
$$= O(n * \log(n)).$$

Analyse für Fall C und D

Maximale Rek.Tiefe
 $= \log_2(n)$

$f(n)$	c_1	$c_1 * n$
$f(n/2)$	c_1	$c_1 * (n/2)$
$f(n/4)$	c_1	$c_1 * (n/4)$
...
	c_1	
$f(1)$	c_0	c_0

Laufzeiten für Fall C

Laufzeiten für Fall D

Summe aller Laufzeiten im Fall C:

$$\begin{aligned} T(n) &= c_1 * \log(n) + c_0 \\ &= O(\log(n)). \end{aligned}$$

Summe aller Laufzeiten im Fall D:

$$\begin{aligned} T(n) &= c_1 * (1 + \frac{1}{2} + \frac{1}{4} + \dots) * n + c_0 \\ &= O(n). \end{aligned}$$

Aufgaben (1)

Aufgabe 8.5

Die Funktion max berechnet das Maximum in einem Feld nach dem Teile-und-Herrsche-Prinzip.

Beurteilen Sie, ob diese Funktion eine bessere Laufzeit erwarten lässt als eine herkömmliche Funktion, die iterativ (d.h. mit einer Schleife) das Maximum bestimmt.

```
static int max(int[] a, int li, int re) {  
  
    if (li == re)  
        return a[li];  
    else {  
        int m = (li+re)/2;  
        int maxL = max(a,li,m);  
        int maxR = max(a,m+1,re);  
        if (maxL >= maxR)  
            return maxL;  
        else  
            return maxR;  
    }  
}
```

Aufgaben (2)

Aufgabe 8.6

Welche Laufzeit hat die binäre Suche aus Kapitel 7?

Aufgabe 8.7

- Was leistet folgende Funktion f ?
- Welche Laufzeit hat die Funktion f, wenn beide Felder a und b die Größe n haben?

```
static boolean f(int[] a, int[] b) {  
  
    assert isSorted(b);  
  
    for (int x : a) {  
        if (!binSuche(b,x))  
            return false;  
    }  
    return true;  
}
```

Aufgaben (3)

Aufgabe 8.8

Die Klasse `TreeSet` aus der Java-API gestattet die Speicherung einer Menge von Elementen als binären Suchbaum (später).

- Mit der Methode `add(x)` lässt sich ein Element `x` zur Menge hinzufügen. Finden Sie in der Java-API die Laufzeit dieser Methode heraus.
- Welche Laufzeit hat die Funktion `f`? Setzen Sie dazu für `Math.random()` eine konstante Laufzeit voraus.

```
public static void f(int n) {  
    TreeSet<Double> s = new TreeSet<Double>();  
  
    for (int i = 0; i < n; i++) {  
        double x = Math.random();  
        s.add(x)  
    }  
}
```

Aufgaben (4)

Aufgabe 8.9

- Gegeben sei ein Kursverlauf x einer Aktie, wobei $x[i]$ der Kurswert am Tag i ist ($0 \leq i \leq n-1$).
- Gesucht ist ein Einkaufstag e und ein Verkaufstag $v \geq e$, so dass $(x[v] - x[e]) / x[e]$ (relativer Gewinn) maximal ist.
- Überzeugen Sie sich, dass die beiden folgenden Algorithmen das Problem lösen:
 - Naives Verfahren: Maximumbildung über alle (e,v) -Paare.
 - Teile-und-Herrsche-Verfahren.
- Geben Sie die Laufzeiten für beide Algorithmen an.

Aufgaben (5)

```
class Aktienkurs {  
    public static class OptEinVerkauf {  
        public int einkauf;  
        public int verkauf;  
        public double gewinn;  
        OptEinVerkauf(int e, int v, double g) {  
            einkauf = e; verkauf = v; gewinn = g;  
        }  
    }  
  
    public static OptEinVerkauf optEinVerkaufNaiv(double[] x) {  
        int e = 0, v = 0;  
        double g = 0;  
        for (int i = 0; i < x.length-1; i++) {  
            for (int j = i+1; j < x.length; j++) {  
                double gNeu = (x[j]-x[i])/x[i];  
                if (gNeu > g) {  
                    e = i;  
                    v = j;  
                    g = gNeu;  
                }  
            }  
        }  
        return new OptEinVerkauf(e, v, g);  
    }  
}
```

Liefert zu Kursverlauf x
den optimalen Ein- und
Verkaufstag zurück.

Naives Verfahren.

Aufgaben (6)

```
private static class TuHReturn {  
    OptEinVerkauf optev;  
    int min;  
    int max;  
    TuHReturn(OptEinVerkauf o, int mi, int ma) {  
        optev = o; min = mi; max = ma;  
    }  
}  
  
public static OptEinVerkauf optEinVerkaufTuH(double[ ] x) {  
    TuHReturn ret = optEinVerkaufTuH(x, 0, x.length-1);  
    return ret.optev;  
}
```

Rückgabewerttyp für
Teile-und-Herrsche-
Funktion

Liefert zu Kursverlauf x
den optimalen Ein- und
Verkaufstag zurück.

Teile-und-Herrsche-
Verfahren.

Aufgaben (7)

```
private static TuHReturn optEinVerkaufTuH(double[] x, int li, int re) {  
    if (li == re) {  
        return new TuHReturn(new OptEinVerkauf(li, li, 0), li, li);  
    }  
    else {  
        int m = (li+re)/2;  
  
        // Optimaler Ein/Verkauf in linker Hälfte:  
        TuHReturn retL = optEinVerkaufTuH(x, li, m);  
        int eL = retL.optev.einkauf;  
        int vL = retL.optev.verkauf;  
        double gL = retL.optev.gewinn;  
  
        // Optimaler Ein/Verkauf in rechter Hälfte:  
        TuHReturn retR = optEinVerkaufTuH(x, m+1, re);  
        int eR = retR.optev.einkauf;  
        int vR = retR.optev.verkauf;  
        double gR = retR.optev.gewinn;  
  
        // Einkauf in linker Hälfte und Verkauf in rechter Hälfte:  
        double gLR = (x[retR.max] - x[retL.min]) / x[retL.min];  
    }  
}
```

Liefert zu Kursverlauf x im Bereich von li bis re einschl. den optimalen Ein- und Verkaufstag zurück.

Teile-und-Herrsche-Verfahren.

Aufgaben (8)

```
// Opt. Ein- u. Verkaufstag ermitteln:  
OptEinVerkauf oev;  
if (gL >= gR && gL >= gLR)  
    oev = retL.optev;  
else if (gR >= gLR)  
    oev = retR.optev;  
else  
    oev = new OptEinVerkauf(retL.min, retR.max, gLR);
```

```
// Min u Max berechnen:  
int min, max;  
if (x[retL.min] <= x[retR.min])  
    min = retL.min;  
else  
    min = retR.min;  
if (x[retL.max] >= x[retR.max])  
    max = retL.max;  
else  
    max = retR.max;
```

```
// Opt. Ein/Verkauf zurückliefern:  
return new TuHReturn(oev, min, max);
```

```
}  
}
```