

---

Vorlesung 3 (Montag 19.2.2018)

---

### 3 Einführung

Kreditpunkte: Teilnahme an Übungen, einzeln ODER 2er Gruppen möglich, Lösungen der (Teil-) Aufgaben müssen ab 12:30 Uhr (!) vorgestellt werden und werden vom Betreuer bewertet mit:

- X (0 Punkte): nicht bzw nur sehr rudimentär bearbeitet
- – (1 Punkt): Aufgabe wesentlich bearbeitet aber Pflichtteil nicht abgeschlossen
- ◦ (2 Punkte): Aufgabe abgeschlossen, mit viel Mühe aber es war detaillierte (!) Hilfestellung nötig (Betreuer schaut in das Programm und sucht selber Fehler). Fragen können aber immer gestellt werden.
- + (3 Punkte): Aufgabe abgeschlossen, im wesentlichen keine Probleme

An einem Tag kann es eine oder mehrere (selten bis nicht) Aufgaben geben. Zum Bestehen sind 50% der Gesamt-Punkte nötig.

### 4 Algorithmen

#### 4.1 Leichte, schwere und unlösbare Probleme

$t_{\mathcal{P}}(x)$  = Laufzeit eines Programms  $\mathcal{P}$  bei Eingabe  $x$ . In der theoretischen Informatik: Laufzeit wird auf Modell-Computern untersucht (z.B. *Turing Maschinen*).

$n = |x|$  = “Größe” der Eingabe (z.B. Anzahl der Bits, um das Problem zu kodieren). *Zeitkomplexität* eines Programms  $\mathcal{P}$  = langsamste (*worst case*) Laufzeit als Funktion von  $n$ :

$$T(n) = \max_{x: |x|=n} t(x) \tag{1}$$

---

Example: Laufzeit

$T(n)$	$T(10)$	$T(100)$	$T(1000)$
$f(n)$	23000	46000	69000
$g(n)$	1000	10000	100000
$h(n)$	100	10000	1000000

□

[Selbsttest]

---

Welches Program ist (asymptotisch) das langsamste?

---

**O notation**

$T(n) \in O(g(n))$ :  $\exists c > 0, n_0 \geq 0$  mit  $T(n) \leq cg(n) \forall n > n_0$ . “ $T(n)$  ist höchstens von der Ordnung  $g(n)$ .”

Typische Zeitkomplexitäten:

Table 1: Wachstum von Funktionen in Abhängigkeit der Inputgröße  $n$ .

$T(n)$	$T(10)$	$T(100)$	$T(1000)$
$n$	10	100	1000
$n \log n$	10	200	3000
$n^2$	$10^2$	$10^4$	$10^6$
$n^3$	$10^3$	$10^6$	$10^9$
$2^n$	1024	$1.3 \times 10^{30}$	$1.1 \times 10^{301}$
$n!$	$3.6 \times 10^6$	$10^{158}$	$4 \times 10^{2567}$

Polynomiale Probleme (Klasse P) werden als “leicht” angesehen, exponentielle Probleme werden “schwer” genannt. Für einige (gerade in der Praxis wichtige) Probleme: kein polynomialer Algorithmus *bekannt*. Es gibt bisher keinen Beweis, dass es nicht doch polynomiale Algorithmen gibt. *NP-schwer* Probleme, laufen allerdings polynomial auf einen *nichtdeterministischen* (Modell) Computer.

Entscheidungsprobleme: Nur Ausgaben “Ja”/“Nein” möglich. Bsp: Ist die Grundzustandsenergie eines Systems  $\leq E_0$  (gegebener Parameter)?

Man kann für einige Probleme beweisen, dass sie *unentscheidbar* sind, d.h. es gibt *keinen allgemeinen* Algorithmus, der “Ja“ und “Nein“ für alle möglichen Eingaben=Probleminstanzen ausgibt. Probleme sind aber *beweisbar*, d.h. man kann eine der möglichen Antworten beweisen, aber nicht beide.

- Halteproblem: Hält ein gegebenes Programm bei gegebener Eingabe? (Falls es hält, kann man es beweisen: man lässt es laufen)
- Korrektheitsproblem: Erzeugt ein gegebenes Programm die erwünschte Ausgabe für *alle* Eingaben? (Falls nicht, kann man das leicht beweisen: man lässt es für eine Eingabe laufen, wo es nicht stimmt, ggf. systematisch nach solchen Eingabe suchen)

Es gibt (akademische) Funktionen, die sogar *nicht berechnbar* sind: Beweis über Diagonalisierung (analog zum Cantorsche Beweis, dass es nicht-rationale Zahlen gibt)

## 4.2 Rekursion und Iteration

Prinzip der *Rekursion*: Unterprogramm ruft sich selbst auf. Natürlich für rekursive Definitionen.

Beispiel: Fakultät  $n!$ :

$$n! = \begin{cases} 1 & \text{falls } n = 1 \\ n \times (n-1)! & \text{sonst} \end{cases} \quad (2)$$

einfache Übersetzung in C-Funktion

```
double factorial(int n)
{
    if(n==1)
        return(1.0);
    else
        return(n*factorial(n-1));
}
```

Analyse der Laufzeit von `factorial()` mittels *Rekurrenzgleichungen*:

$$\tilde{T}(n) = \begin{cases} C & \text{for } n = 1 \\ C + \tilde{T}(n-1) & \text{for } n > 1 \end{cases} \quad (3)$$

Lösungsmöglichkeit: als DGL:  $\frac{d\tilde{T}}{dn} = \frac{\tilde{T}(n) - \tilde{T}(n-1)}{n - (n-1)} = C \rightarrow \tilde{T}(n) = CN + K$ ,  
mit  $\tilde{T}(1) = C \rightarrow$

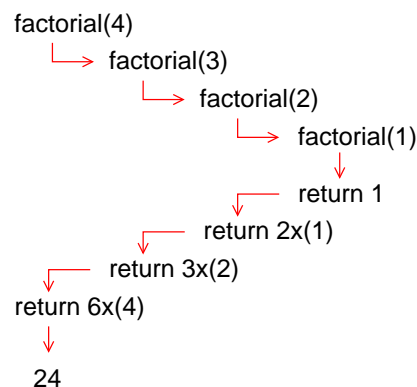


Figure 1: Hierarchie von rekursive Aufrufen bei der Berechnung von  $\text{factorial}(4)$ .

Lösung:  $\tilde{T}(n) = Cn$ , d.h. Laufzeit linear in  $n$  aber exponentiell in der Länge der Eingabe (=Anzahl der Bits).

Auch iterative Berechnung der Fakultät über Schleife möglich, auch  $O(n)$ .

### 4.3 Divide-and-conquer (Teile und herrsche)

[Selbsttest]

Überlegen Sie sich erst für 5 Minuten einen Algorithmus (Grundidee), der Elementen  $\{a_0, \dots, a_{n-1}\}$  "der Größe nach" sortiert.

Dann diskutieren Sie 5 Minuten mit ihrem Nachbarn ihre Lösungen.

Welche asymptotische Laufzeit  $T(n)$  vermuten Sie für Ihr Verfahren?

ACHTUNG: Lesen Sie den Rest des Abschnitts NICHT, bevor Sie sich etwas überlegt haben

Hier: Prinzip (verwendet Rekursion):

1. reduziere Problem auf einige kleinere Probleme
2. löse die kleineren Probleme
3. setze Lösung des großen Problems aus den Lösungen der kleineren Probleme zusammen.

Beispiel: Sortierung von Elementen  $\{a_0, \dots, a_{n-1}\}$  mittels *Mergesort*.

Grundidee: Teile Menge in zwei gleichgroße Mengen, sortiere beide und füge sie dann sortiert zusammen.

```

/** sorts 'n' integer numbers in the array 'a' in ascending **/
/** order using the mergesort algorithm                               **/
void mergesort(int n, int *a)
{
    int *b,*c;                                /* two arrays */
    int n1, n2;                                /* sizes of the two arrays */
    int t, t1, t2;                            /* (loop) counters */

    if(n<=1)                                  /* at most one element ? */
        return;                               /* nothing to do */
    n1 = n/2; n2 = n-n1;                      /* calculate half sizes */

    /* array a is distributed to b,c. Note: one could do it */
    /* using one array alone, but yields less clear algorithm */
    b = (int *) malloc(n1*sizeof(int));
    c = (int *) malloc(n2*sizeof(int));
    for(t=0; t<n1; t++)                        /* copy data */
        b[t] = a[t];
    for(t=0; t<n2; t++)
        c[t] = a[t+n1];

    mergesort(n1, b);                          /* sort two smaller arrays */
    mergesort(n2, c);

    t1 = 0; t2 = 0;                            /* assemble solution from subsolutions */
    for(t=0; t<n; t++)
        if( ((t1<n1)&&(t2<n2)&&(b[t1]<c[t2])) ||
            (t2==n2))
            a[t] = b[t1++];
        else
            a[t] = c[t2++];

    free(b); free(c);
}

```

---

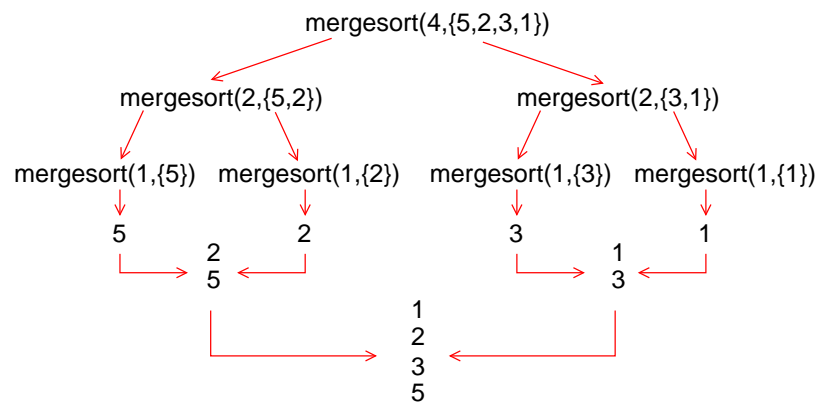
[Selbsttest]

---

Laden Sie sich `mergesort.c` vom StudIP/kopieren Sie vom Stick.

Erweitern Sie die `mergesort()` Unteroutine, so dass am Anfang und Ende jeweils das Array `a[]` vor/nach dem Sortieren angezeigt wird.

Compilieren Sie und lassen Sie `mergesort` mit Argument 8 laufen. Versuchen Sie die Ausgabe nachzuvollziehen.

Figure 2: Aufruf von `mergesort(4, {5, 2, 3, 1})`.

---

Laufzeit: Aufteilung der Mengen sowie Zusammensetzen:  $O(n)$ ; rekursive Aufrufe:  $T(n/2)$ .

Rekurrenz:

$$T(n) = \begin{cases} C & (n = 1) \\ Cn + 2T(n/2) & (n > 1) \end{cases} \quad (4)$$

Lösung für große  $n$ :  $T(n) = \frac{C}{\log 2} n \log n$ . Beweis durch Einsetzen

$$\begin{aligned}
 T(2n) &= C2n + 2T(2n/2) \\
 &= C2n + 2\left(\frac{C}{\log 2} n \log n\right) \\
 &= \frac{C}{\log 2} 2n \log 2 + \frac{C}{\log 2} 2n \log n \\
 &= \frac{C}{\log 2} 2n \log(2n)
 \end{aligned}$$