Block 4 (Montag 19.2.2024)

5 Infos Übungen

Die Übungen finden in Präsenz statt, im W1 0-008. Dort gibt es Computer mit Linux. Sie können nach Möglichkeit auch eigene Laptops mit Linux nutzen, vorbereitet so dass sie benutzen können: Shell, Editor, Compiler, gnuplot, gdb, (möglichst) valgrind. Von der Nutzung umfangreicher und leistungsfähiger Entwicklungsumgebungen wird abgeraten, da das für kleine Lern-Projekte mehr Aufwand als Nutzen bringt.
Teilnahme an Übungen, einzeln ODER 2er Gruppen möglich, 2er Gruppen sind zu bevorzugen.
Die Übungen finden "in-time" statt: Die Übungsblätter + Material wird direkt vor der Übung im StudIP bereit gestellt, während der Übungszeit bearbeitet und (zum Ende hin) bewertet.
Grunsätzlicher Ablauf: Die Aufgaben basieren (meistens) auf teilfertigen Programmen. Die Programme müssen vervollständigt (oder manchmal ganz geschrieben) werden, dazu Simulationen durchgeführt und (manchmal) Daten ausgewertet oder geplottet.
Innerhalb, also meistens zum Ende, der Übungen müssen die selber geschriebenen Teile sowie die erzielten Ergebnisse vorgeführt werden (=Abnahme). Alle Gruppenmitglieder müssen beitragen. Es werden darauf Punkte (bis zu 10) vergeben, für die ganze Gruppe die gleiche Punktzahl.
Da üblicherweise die Simulationen und Datenauswertung darauf basieren, dass das Programm funktioniert, ist es im *Notfall* auch möglich, dass der Dozent bzw. Tutoren in das Programm schaut und hilft Fehler zu suchen (also quasi auch die Aufgabe mit bearbeitet), natürlich ohne Garantie. In diesem Fall werden je nach Aufwand Punkte abgezogen, auch wenn zum Schluss das ganze Blatt bearbeitet ist! Allgemeine Fragen können natürlich jederzeit an den Dozenten gestellt werden, ohne Punktabzug.
Es gibt ingesamt 8 Termine zu je 4h. Jeweils 14:15 bis (ca.) 17:45, je nach Dauer der Abnahmen.

6 Algorithmen

VIDEO: `video04a_complexity_r`

6.1 Leichte, schwere und unlösbare Probleme

$t(x)$= 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:
Polynomiale Probleme (Klasse P) werden als "leicht" angesehen, exponentielle Probleme werden "schwer" genannt. Für einige (gerade in der Praxis

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	1.3×10^{30}	1.1×10^{301}
$n!$	3.6×10^6	10^{158}	4×10^{2567}

wichtige) Probleme: kein polynomialer Algorithmus *bekannt*. Es gibt bisher aber noch 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 gegebens 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)

Es gibt (akademische) Funktionen, die sogar *nicht berechnbar* sind: Beweis über Diagonalisierung (analog zum Cantorschen Beweis, dass es nichtrationale Zahlen gibt)
VIDEO: `video04b_rekursion_r`

6.2 Rekursion und Iteration

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

Beispiel: Fakultät $n!$:

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

einfache Übersetzung in C-Funktion

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

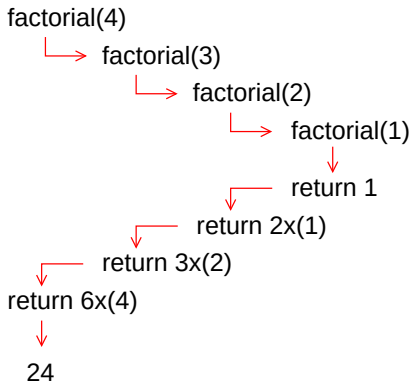

Figure 1: Hierarchie von rekursive Aufrufen bei der Berechnung von factorial(4).

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

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

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 $\tilde{T}(1) = C \rightarrow$
Lösung: $\tilde{T}(n) = Cn$, d.h. Laufzeit linear in n (aber exponentiell in der Länge der Eingabe (=Anzahl der Bits), da $n = 2^{\log_2 n}$).

[Selbsttest]

Geben Sie eine iterative Berechung der Fakultät über Schleife an. Wie ist die Zeitkomplexittät?

VIDEO: `video04c_dvidide_and_conquer_r`

6.3 Divide-and-conquer (Teile und herrsche)

[Selbsttest]

Überlegen Sie sich erst für 5 Minuten einen Algorithmus (Grundidee), der Elementen $\{a_0, \ldots, a_{n-1}\}$ "der Größe nach" sortiert.
Dann diskutieren Sie 5 Minuten mit ihrem Nachbarn ihre Lösungen.
Welche asymtotische 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 $\{a_0, \ldots, a_{n-1}\}$ 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 mergsort algorithm                      **/
void my_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];

  my_mergesort(n1, b);                /* sort two smaller arrays */
  my_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);
}
```

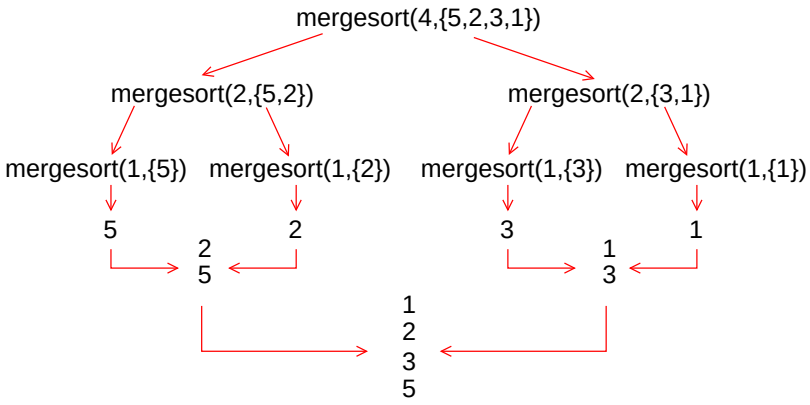

Figure 2: Aufruf von `mergesort`$(4, \{5, 2, 3, 1\})$.

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

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

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

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

VIDEO: `video04d_dynamic_programming_r`

6.4 Dynamisches Programmieren

Fibonacci Zahlen:

$$\text{fib}(n) = \begin{cases} 1 & (n = 1) \\ 1 & (n = 2) \\ \text{fib}(n-1) + \text{fib}(n-2) & (n > 2) \end{cases} \tag{5}$$

Z.B. $\text{fib}(4) = \text{fib}(3) + \text{fib}(2) = (\text{fib}(2) + \text{fib}(1)) + \text{fib}(2) = 3$,
$\text{fib}(5) = \text{fib}(4) + \text{fib}(3) = 3 + 2 = 5$.
Wächst sehr schnell: $\text{fib}(10) = 55$, $\text{fib}(20) = 6765$, $\text{fib}(30) = 83204$, $\text{fib}(40) > 10^8$

Anzahl der Aufrufe bei rekursiver Implentierung wächst auch exponentiell mit n, schneller als die Funktion!

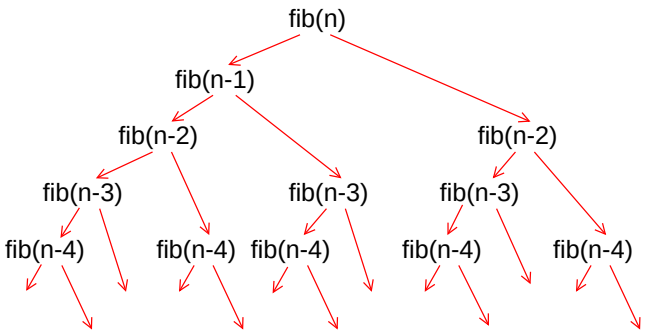

Figure 3: Hierarchie der Aufrufe für fib(n).

[Selbsttest]

Wie man man einen schnelleren Algorithmus finden?

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

Besser: dynamisches Programmieren. Prinzip: Berechne Lösung für kleinere Probleme und benutze sie (nicht rekursiv) um größere Probleme iterativ zu lösen.

```
/** calculates Fibonacci number of 'n' dynamically **/
double fib_dynamic(int n)
{
  double *fib, result;
  int t;
  if(n<=2)                                    /* simple case ? */
    return(1);                          /* return result directly */
  fib = (double *) malloc(n*sizeof(double));
  fib[1] = 1.0;                       /* initialise known results */
  fib[2] = 1.0;

  for(t=3; t<n; t++)          /* calculate intermediate results */
    fib[t]=fib[t-1]+fib[t-2];

  result = fib[n-1]+fib[n-2];
  free(fib);
  return(result);
}
```

[Selbsttest]

Wie ist die Laufzeit des Verfahrens?
Wettbewerb: was ist das größte n, das fib(n) noch eine endliche Ausgabe erzeugt: jede(r) gibt eine Schätzung ab.

Noch schneller: Formel

$$\mathrm{fib}(n) = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right) \tag{6}$$