Praktische Informatik für Wirtschaftsmathematiker, Ingenieure und Naturwissenschaftler I (PIWIN I, 3 V + 1 Ü) WS 2002/03

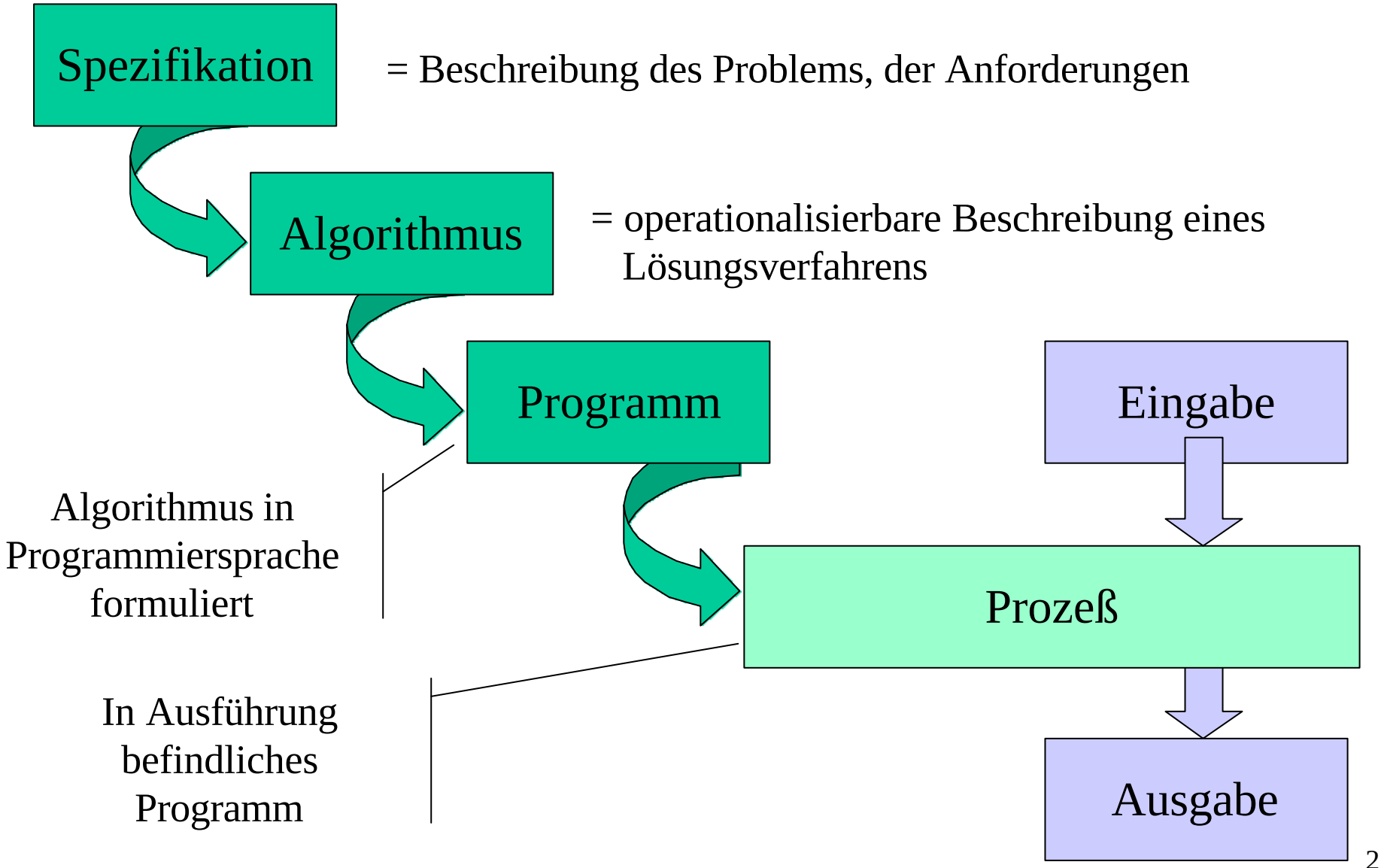
6. Vorlesungswoche

Welcher Algorithmus ist besser ?

Über die Komplexität von Algorithmen am Beispiel der vorgestellten Sortieralgorithmen

Unterlagen: Gumm/Sommer, Kapitel 4.2

Stationen im Entwurf von Algorithmen und Programmen



Übersicht

- Begriffe
  - Spezifikationen, Algorithmen, formale Sprachen, Grammatik
- Programmiersprachenkonzepte
  - Syntax und Semantik
  - imperative, objektorientierte, funktionale und logische Programmierung
  - formale Sprachen und Grammatik
- Grundlagen der Programmierung
  - imperative Programmierung:
    - Verfeinerung, elementare Operationen, Sequenz, Selektion, Iteration, funktionale Algorithmen und Rekursion, Variablen und Wertzuweisungen, Prozeduren, Funktionen und Modularität, Zuweisung, Sequenz
  - objektorientierte Programmierung
- Algorithmen und Datenstrukturen
- Berechenbarkeit und Entscheidbarkeit von Problemen
- Effizienz und Komplexität von Algorithmen
- Programmentwurf, Softwareentwurf

Wie bewertet man Algorithmen ?

nach Ressourcenverbrauch:

- Speicherplatz

  Wieviel Speicherplatz braucht ein Algorithmus ?

  – zur Darstellung von Eingabewerten und Zwischenergebnissen:

    Viele Algorithmen erzeugen Datenstrukturen, deren Größe von Eingabeparametern abhängen

  – für den Aufrufstack

    Insbesondere rekursive Algorithmen verursachen Speicherverbrauch durch geschachtelte Funktionsaufrufe,

    Speicherbedarf hängt von der Rekursionstiefe und der Menge lokaler Variable und Parameter je Aufruf ab

- Laufzeit

  Wieviel CPU Zeit (Berechnungsschritte) braucht ein Algorithmus ?

  – Zählen von elementaren Rechenoperationen

  – Anzahl Berechnungsschritte meist abhängig von Eingabeparametern

Wir brauchen also ein sinnvolles mathematisches Modell, ...

1. Möglichkeit: Testen von implementierten Algorithmen

- Vorgehen
  - Implementieren eines Algorithmus,
  - Messen der Laufzeit und des Speicherverbrauchs für eine Reihe von Eingabedaten
  - Statistische Aufbereitung und Vergleich der Ergebnisse
- Freiheitsgrade
  - Hardware: CPU, Speicher
  - Software: Programmiersprache, Compiler, Betriebssystem
  - Qualität der Programmierung
  - Eingabedaten
    - repräsentative, typische Daten
    - besonders ungünstige Daten
    - besonders günstige Daten
- Messung: durch Software zur Laufzeit des Programms
  - Genauigkeit der Messung: Genauigkeit von Uhren, Systemzeit vs Rechenzeit, Einfluß des Messaufwands bei der Berechnung, Profiler
  - Bestimmung der Speichergröße: bei Allokation einfach möglich
- Statistische Auswertung und Vergleich

Insgesamt: möglich, aber prinzipiellen und praktischen Schwierigkeiten

2. Möglichkeit: Bewertung anhand eines math. Modells

- Berücksichtigung der Eingabeparameter mit einem Wert, der die Größe der Eingabe bemisst
  - Größe als Charakterisierung der Eingabe im Hinblick auf den Arbeitsaufwand der entsteht
  - meist als Länge der Eingabe gewählt, z.B. sortiere n Zahlen => Parameter ist n
  - Anmerkung: Charakterisierung der Eingabe ist nicht immer einfach!
- Unterscheidung von 3 Fällen

  Bewertung von Speicherbedarf und Laufzeit bei
  - möglichst ungünstigen Eingabedaten: Worst Case
  - typischen / durchschnittlichen Eingabedaten: Average Case
  - best möglichen Eingabedaten: Best Case

Bewertung anhand eines math. Modells

- Angabe einer Funktion, die den Aufwand hinreichend genau beschreibt, um sinnvolle Vergleiche zu ermöglichen
  - Festlegen von elementaren Operationen
  - Durchzählen der Operationen für im Falle einer Berechnung
  - Abschätzen der Größenordnung
- formaler Rahmen: O-Notation (O = Ordnung, Größenord.)

$f(n)$ ist höchstens von der Ordnung $g(n)$,
falls eine Konstante C existiert, so dass für alle großen Werte N gilt :
$f(N) \leq C \cdot g(N)$
Notation : $f(n) = O(g(n))$

O-Notation ignoriert den Einfluss konstanter Faktoren (Einfluß von C), z.B. Einfluß des Compilers, CPU Takt etc.
O-Notation fokussiert auf Abschätzung des Wachstumsverhalten in Abhängigkeit von n.

Beispiel: einfacher Sortieralgorithmus

```
for (i = 0; i < n - 1; i++) {
    // Prüfe, ob a [i] Nachfolger hat, die kleiner als a [i] sind:
     for (j = i + 1; j < n; j++) {
     if (a [i] > a [j])  {                    // Ist Nachfolger a [j] kleiner als a [i] ?
                                              // Vertausche a [i] mit a [j]: Ringtausch mit Hilfsvariable z
              z = a [i]; a [i] = a [j]; a [j] = z;
}}}
```

- Elementaroperationen: Zuweisungen, Vergleiche

  je innerem Schleifendurchlauf: 1 Vergleich, 3 Zuweisungen

- Anzahl innerer Schleifendurchläufe

$i = 0 : n-1$

$i = 1 : n-2$

$i = 2 : n-3$

...

$i = n-2 : 1$

$\text{insgesamt} : \sum_{x=1}^{n-1} x \cdot 4 = 4 \cdot (n-1)n/2 = 2n^2 - 2n$

$\text{damit} : O(n^2)$

Beispiel: einfacher Sortieralgorithmus

$$\text{insgesamt} : \sum_{x=1}^{n-1} x \cdot 4 = 4 \cdot (n-1)n/2 = 2n^2 - 2n$$

$$\text{damit} : O(n^2)$$

- Beobachtungen:
- konstante Anzahl Elementaroperationen spielt keine Rolle
  - z.B. geschickte Implementierung realisiert Ringtausch mit weniger Operationen
  - z.B. Compileroptimierung reduziert die Anzahl Operationen auf Ebene der Maschinensprache
- genaue Erfassung der Anzahl Schleifendurchläufe eigentlich irrelevant,
- grobe Abschätzung: n äußere Schleifendurchläufe, maximal n innere

  => n^2 Durchläufe mit konstant vielen Operationen innerhalb der Schleife

  => $O(n^2)$ wäre in diesem Fall auch ok

Gibt es Unterschiede bzgl Best Case, Average Case, Worst Case ?

Nein!

Ist die Bemassung dennoch sinnvoll ?

Ja! Denn Unterschiede zu Quicksort und Heapsort sind beachtlich !!!

Laufzeitabschätzungen für Sortieralgorithmen

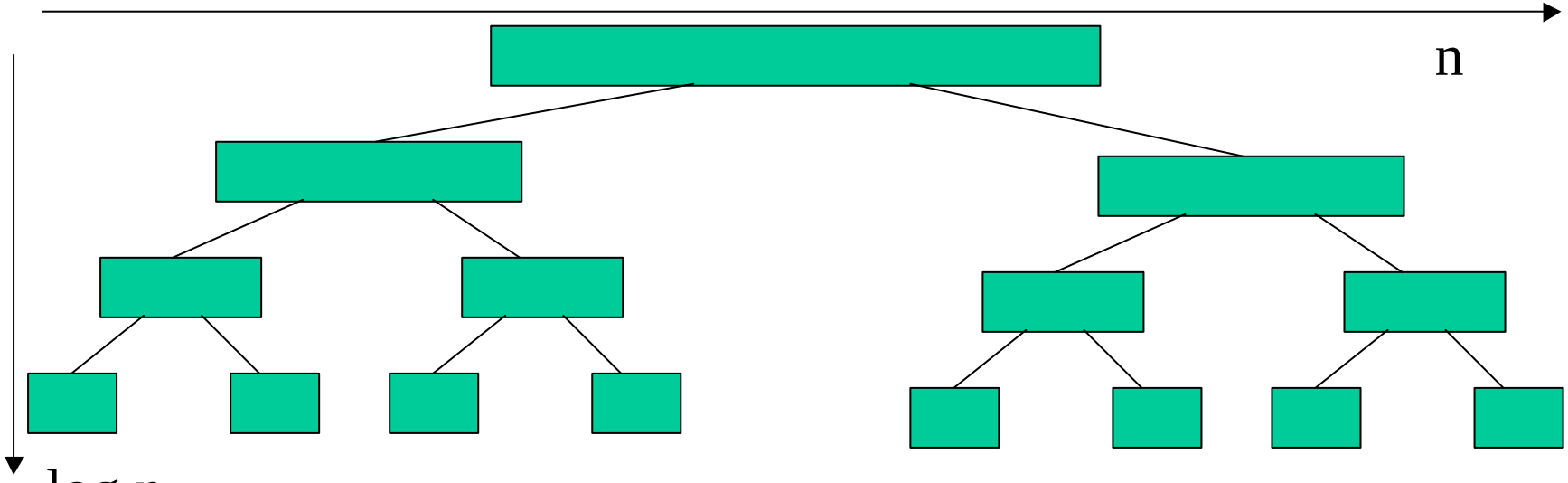
- 1. Algorithmus:
  - Anmerkung: entspricht in etwa SelectionSort (Gumm/Sommer)
  - Best Case, Worst Case, Average Case: $O(n^2)$
- Heapsort:
  - Anmerkung: es existiert eine bessere, aber etwas kompliziertere Variante, die nach dem Tausch des Wurzelknotens die Rekonstruktion des Heaps über eine binäre Suche auf einem Weg kleinster Sohnknoten realisiert
  - Worst Case $O(n \log n)$ $(2\ n \log n)$
- Quicksort:
  - Worst Case $O(n^2)$
  - Average Case $O(n \log n)$ $(1{,}386\ n \log n)$

Laufzeitabschätzung für Quicksort

- Worst Case: je Aufteilung nur 1 Element abgetrennt
  - Rekursionstiefe: n
  - Ebenen 1,2,..., n mit n, n-1, n-2, ..., 1 zu betrachtenden Elementen

  => $O(n^2)$
- Best Case: gleichmäßige Aufteilung je Partitionierung
  - Rekursionstiefe: log n
  - je Ebene über alle Aufrufe hinweg n Elemente zu betrachten

  => O(n log n)



Wiederholung: Heapsort

```
class Heap {
   private int[] a;
   Heap(int k) {
         a = new int[k+1];
   }
   void Setze(int i, int x) { a[i] = x; }

   int Gib(int i) { return a[i]; }

   private void Tausche(int eins, int zwei) {
         int t = a[eins];
         a[eins] = a[zwei];
         a[zwei] = t;
   }
```

Wiederholung: Heapsort

```
void BaueHeap() {
        for (int i = (a.length-1)/2; i >= 1; i--)
              Heapify(i, a.length-1);
}

void Sortiere() {
        BaueHeap();
        for (int i = a.length-1; i > 1; i--) {
              Tausche(1, i);
              Heapify(1, i-1);
        }
}
```

als n/2 Aufrufe von heapify

als n Aufrufe von heapify

Wiederholung: Heapsort, Herstellung des Heaps

```
void Heapify(int dieserKnoten, int heapGröße) {
   int links = 2 * dieserKnoten,
       rechts = links + 1,
       derSohn;
   if (links <= heapGröße && rechts > heapGröße) {
      if (a[dieserKnoten] > a[links])
         Tausche(dieserKnoten, links);
   }
   else {
      if (rechts <= heapGröße) {
         derSohn = (a[links]< a[rechts])? links: rechts;
         if (a[dieserKnoten] > a[derSohn] ) {
            Tausche(dieserKnoten, derSohn);
            Heapify(derSohn, heapGröße);
         }
      }
   }
}
```

(©V. Gruhn, U Dortmund)

Aufwand für Heapify(i,m)

- falls Teilbaum Tiefe d hat, erfordert Heapify maximal 5d Vergleiche und d Vertauschungen
- in Heapify(i,m) ist der längste Pfad i, $2i, 2^2 i, \ldots, 2^l i$, wobei l die größte ganze Zahl mit $2^l i < m$ ist, also $l = \left\lfloor \log \frac{m}{i} \right\rfloor \leq \log n$
- in der 1. Phase zur Herstellung des Heaps:
  n/2 Aufrufe von Heapify für Werte i=1 bis n/2, somit $\frac{n}{2} \log n$
- in der 2. Phase:
  n Aufrufe von Heapify, somit n log n

Insgesamt: Worst Case: O(n log n)

Genauere Abschätzung zeigt:

1. Phase sogar in O(n) weil heapify nur für n/2 Bäume der Tiefe 1, n/4 Bäume der Tiefe 2, ... 1 Baum der Tiefe log(n) aufgerufen wird.

Daher:
$$S = n/2 \cdot 1 + n/4 \cdot 2 + \ldots 2(\log(n) - 1) + 1 \cdot \log(n)$$
$$2S = n \cdot 1 + n/2 \cdot 2 + \ldots + 2 \cdot \log(n)$$
$$2S - S = S = n + n/2 + n/4 + \ldots + 4 + 2 - \log(n) \leq 2n \in O(n)$$

Laufzeitvergleiche mit Messungen nach Gumm/Sommer

- 266 Mhz Pentium II PC
- SelectionSort:

  N=5000 0,35 s N=10000 1,55 s N=15000 3,5s

  bei sortieren und unsortierten Daten gleich

  Anmerkung: Bubblesort bei sortierten Daten besser!
- Heapsort:

  zufällig

  N=0,5 Mio 2,25 s N=1 Mio 5,1 s N=2 Mio 11,6 s

  sortiert

  N=0,5 Mio 1,7 s N=1 Mio 3,5 s N=2 Mio 7,3s
- Quicksort:

  zufällig

  N=0,5 Mio 0,9 s N=1 Mio 1,8 s N=2 Mio 3,9 s

  sortiert

  N=0,5 Mio 0,3 s N=1 Mio 0,7 s N=2 Mio 1,5s

Fazit nach Gumm/Sommer

- bei wenigen Datensätzen
  - Laufzeit unkritisch -> einfaches Verfahren ok
- bei „fast sortierten“ Datensätzen
  - Insertion oder Bubblesort (hier nicht vorgestellt, Idee bei Bubblesort: Vergleichen von Nachbarn, ggfs Vertauschen, Strategie: durch fortgesetzte lokale Korrekturen globale Lösung erreichen)
- bei vielen Daten, die zufällig sortiert sind oder häufig sortiert werden müssen,
  - ggfs DistributionSort (hier nicht behandelt) and das Problem anpassen
- flexible Lösung, Risiko eines schlechten Worst Case Verhaltens akzeptabel
  - Quicksort
- in allen anderen Fällen ShellSort (hier nicht behandelt) oder Heapsort

Zusammenfassung

- Bewertung von Algorithmen
  - Speicherplatz und Laufzeit
- Verfahren
  - Messungen
  - Laufzeitabschätzungen mit O-Notation
- Beispiele
  - SelectionSort
  - HeapSort
  - QuickSort
- Anmerkung:
  - viele Algorithmen sind in Bibliotheken realisiert und verfügbar
  - zugehörige Dokumentation sollte Angaben zum Laufzeitverhalten machen