Grundlagen der Theoretischen Informatik

Sebastian Iwanowski
FH Wedel

Kap. 4: Einführung in die Komplexitätstheorie

Entwurf von Algorithmen

Gegeben eine Spezifikation:
Wie konstruiert man einen Algorithmus dafür ?

Da die allgemeine Aufgabe nicht automatisierbar ist, kann es nur Empfehlungen für Lösungsstrategien geben

Suchproblem:

```
procedure search (data, k): integer
```

Eingabe: Datenfeld aus n Zahlen, gesuchte Zahl k

Ausgabe: Position von k im Datenfeld (wenn vorhanden, sonst 0)

Sortierproblem:

```
procedure sort (data): array
```

Eingabe: Datenfeld aus n Zahlen

Ausgabe: Ein neues Datenfeld mit denselben Zahlen, aber aufsteigend sortiert

Entwurf von Algorithmen: Suchproblem

1. Strategie: Tue das Nächstliegende (greedy-Strategie)

Lösungsskizze: *Lineare Suche*

- **Durchlaufe die Elemente vom Datenfeld der Reihe nach und vergleiche sie mit k. Erhöhe in jedem Durchlauf den Positionszähler.**
- **Wenn im Durchlauf das Element gefunden wird, dann gib den Positionszähler aus.**
- **Wenn im Durchlauf das Element nicht gefunden wird, dann gib 0 aus.**

```
procedure searchLin (data, k): integer
begin
  pos := 1;
  while pos ≤ length(data) do
  begin
    if data[pos] = k then return pos;
    pos := pos + 1;
  end {while}
  return 0;
end {searchLin}
```

```
procedure search (data, k): integer
begin
  return searchLin (data, k)
end {search}
```

Entwurf von Algorithmen: Suchproblem

1. Strategie: Tue das Nächstliegende (greedy-Strategie)

Lösungsskizze: *Lineare Suche*

- Durchlaufe die Elemente vom Datenfeld der Reihe nach und vergleiche sie mit k. Erhöhe in jedem Durchlauf den Positionszähler.
- Wenn im Durchlauf das Element gefunden wird, dann gib den Positionszähler aus.
- Wenn im Durchlauf das Element nicht gefunden wird, dann gib 0 aus.

```
procedure searchLinRec (data, pos, k): integer
begin
  if pos > length(data) return 0;
  if data[pos] = k then return pos;
  return searchLinrec (data, pos+1, k)
end {searchLinRec}
```

```
procedure search (data, k): integer
begin
  return searchLinRec (data, 1, k)
end {search}
```

Entwurf von Algorithmen: Suchproblem

2. Strategie: Teile und herrsche (divide and conquer)

Lösungsskizze (Variante für sortierte Daten): *Binärsuche*

- **Vergleiche k mit dem Element in der Mitte:**
- **Wenn k gleich ist, dann gib die Mittelposition aus.**
- **Wenn k kleiner ist, dann suche in der linken Hälfte und gib das Ergebnis aus.**
- **Wenn k größer ist, dann suche in der rechten Hälfte und gib das Ergebnis aus.**

```
procedure binarySearch (data, left, right, k): integer
begin
  if left > right then return 0;
  mid := (left + right) div 2;
  if k = data[mid] then return mid;
  if k < data[mid] then return binarySearch (data, left, mid-1, k);
  return binarySearch (data, mid+1, right, k)
end {binarySearch}
```

```
procedure search (data, k): integer
begin
  return binarySearch (data, 1, length (data), k)
end {search}
```

Klassifikation von Algorithmen

Wie klassifiziert man Algorithmen ?

- **offensichtlich nicht durch die Unterscheidung rekursiv / iterativ !**
- **Unterscheidung nach Lösungsstrategie (greedy, divide and conquer, ...) schon besser !**

Welche Implementierungsvariante (rekursiv / iterativ) ist zu einem gegebenen Algorithmus zu bevorzugen ?

Grundprinzip:

- **Wähle die leichter verständliche und damit besser zu verifizierende Variante, wenn das nicht zu viel kostet**

!! Gegenbeispiel: Berechnung von n! !!

Bewertung von Algorithmen

Was wird bewertet ?

- benötigte Rechenzeit
- benötigter Speicherplatz

Welche Eigenschaften sollte eine Bewertung haben ?

- unabhängig von der Implementierung
- unabhängig vom eingesetzten Computer

Wie ist das möglich ?

Grundprinzip:

- Vernachlässige Konstante, die nicht von der Problemgröße abhängen !
- Implementierungen gelten als „gleich gut“,
  wenn sie sich nur um eine Konstante unterscheiden.

Bewertung von Algorithmen

Definition Komplexitätsklasse:

Ein Algorithmus gehört bzgl. der Rechenzeit (des Speicherplatzes) zur Komplexitätsklasse **O(f(n))**, wenn es eine Konstante c gibt, sodass gilt:

Die Rechenzeit (Der Speicherplatz) für ein Problem der Größe n benötigt maximal c·f(n) Rechenschritte (Speicherplätze).

- c darf nicht von der Problemgröße n abhängen.
- c darf vom Algorithmus abhängen.
- c darf vom Computer und von der Implementierung abhängen.
- O wird das Landau-Symbol genannt (nach Edmund Landau, 1877-1938)

Typische Komplexitätsklassen von Algorithmen:

O(1), O(log n), O(n), O(n log n), O (n^2), O(P(n)), wobei P ein Polynom ist

O(2^n), O(exp(n))

Bewertung von Algorithmen

Beispiel für einen 500 Mhz-Rechner mit Rechenzeit $2ns$ pro Taktzyklus:

$O(\ldots)$	Wert von n: 2	4	8	16	32	64	128
$O(\log_2 n)$	$2ns$	$4ns$	$6ns$	$8ns$	$10ns$	$12ns$	$14ns$
$O(n)$	$4ns$	$8ns$	$16ns$	$32ns$	$64ns$	$128ns$	$256ns$
$O(n \log_2 n)$	$4ns$	$16ns$	$48ns$	$128ns$	$320ns$	$768ns$	$1792ns$
$O(n^2)$	$8ns$	$32ns$	$128ns$	$512ns$	$2\mu s$	$8\mu s$	$32\mu s$
$O(n^3)$	$16ns$	$128ns$	$1\mu s$	$8\mu s$	$65\mu s$	$524\mu s$	$4ms$
$O(2^n)$	$8ns$	$32ns$	$512ns$	$131\mu s$	$8.59s$	$1169a$	$2 \cdot 10^{22}a$
$O(3^n)$	$18ns$	$162ns$	$13\mu s$	$86ms$	$42.89d$	$2 \cdot 10^{14}a$	$7.5 \cdot 10^{44}a$
$O(n!)$	$4ns$	$48ns$	$81\mu s$	$11.6h$	$1.67 \cdot 10^{28}a$	$1.9 \cdot 10^{74}a$	$2 \cdot 10^{214}a$

Tabelle 24: Berechnungsdauer von n Rechenschritten nach Komplexitätsklasse

Sekundärquelle: Vorlesung GdP an der FH Wedel, Prof. Dr. Ralf Möller, SS 2003, Primärquelle unbekannt

Entwurf von Algorithmen: Suchproblem

2. Strategie: Teile und herrsche (divide and conquer)

Einsatz der Binärsuche für allgemeine Daten:

```
procedure search (data, k): integer
begin
  (sortedData, indexConversion) := sort (data);
  resultIndex := binarySearch (sortedData, 1, length (data), k);
  if resultIndex = 0 then return 0;
  return indexConversion[resultIndex]
end {search}
```

Die Prozedur **`sort`** hat 2 Rückgabewerte:

- Ein Feld **`sortedData`**, das dieselben Elemente wie **`data`** enthält, aber in sortierter Reihenfolge.

- Ein Feld **`indexConversion`**, in dem zu Index j der Index vermerkt ist, an dem das Element, das jetzt in **`sortedData[j]`** steht, vorher in **`data`** gestanden hat (**`sortedData[j] = data [indexConversion[j]]`**).

Offene Frage: Lohnt sich das Vorsortieren ?

Entwurf von Algorithmen: Sortierproblem

1. Strategie: Tue das Nächstliegende (greedy-Strategie)

Lösungsskizze: *Selectionsort*

- Durchlaufe die Positionen des neuen Datenfelds der Reihe nach.
- Suche das kleinste Element ab der laufenden Position im neuen Datenfeld.
- Vertausche dieses Element mit dem Element der laufenden Position.
- Gib am Ende des Durchlaufs das neue Datenfeld aus.

```
procedure selectionsort (data): array
begin
  pos := 1;
  while pos ≤ length(newData) do
  begin
    newPos := minPos (data, pos, length(data));
    aux := data[pos];
    data[pos] := data[newPos];
    data[newPos] := aux;
    pos := pos + 1;
  end {while}
  return data;
end {selectionsort}
```

```
procedure sort (data): array
begin
  newData := copy (data);
  return selectionsort (newData)
end {sort}
```

Entwurf von Algorithmen: Sortierproblem

2. Strategie: Teile und herrsche (divide and conquer)

Lösungsskizze: *Mergesort*

- Teile das Datenfeld in 2 Hälften auf.
- Sortiere beide Hälften getrennt.
- Mische die beiden sortierten Hälften in ein zweites Datenfeld.

```
procedure mergesort
       (fromData, toData, left, right)
begin
  if left < right
  then
    begin
      mid := (left + right) div 2;
      mergesort (toData, fromData,
                              left, mid);
      mergesort (toData, fromData,
                           mid+1, right);
      merge (fromData, toData,
             left, mid, mid+1, right);
    end {if}
  end {mergesort}
```

Rekursive Darstellungsvariante

```
procedure sort (data): array
begin
  data1 := copy (data);
  data2 := copy (data);
  mergesort (data1,
     data2, 1, length(data));
  return data2
end {sort}
```

Entwurf von Algorithmen: Sortierproblem

2. Strategie: Teile und herrsche (divide and conquer)

Lösungsskizze: *Mergesort*

- Teile das Datenfeld in 2 Hälften auf.
- Sortiere beide Hälften getrennt.
- Mische die beiden sortierten Hälften in ein zweites Datenfeld.

```
procedure mergesortIter (data): array
begin
  data2 := copy (data); n := length(data);
  sortedLength := 1;
  while sortedLength < n do
  begin
    left1 := 1;
    while (left1+sortedLength) < n do
    begin
      right1 := left1+sortedLength; left2 := right1+1; right2 := left2+sortedLength;
      merge (data, data2, left1, right1, left2, right2);
      left1 := right2 + 1
    end;
    sortedLength := sortedLength + sortedLength;
    aux := data; data := data2; data2 := aux
  end;
  return data
end {sort2}
```

Iterative Darstellungsvariante

```
procedure sort (data): array
begin
  newData := copy (data);
  return mergesortIter(newData)
end {sort}
```

Entwurf von Algorithmen: Sortierproblem

2. Strategie: Teile und herrsche (divide and conquer)

Ausformulierung der Hilfsprozedur *merge*:

```
procedure merge (fromData, toData, left1,
                   right1, left2, right2)
begin
  pos1 := left1; pos2 := left2; pos := left1;
  while (pos ≤ right2) do
  begin
    if pos1 > right1
      then
        assign (fromData, toData, pos2, pos)
      else if pos2 > right2
        then
          assign (fromData, toData, pos1, pos)
      else if fromData[pos1] ≤ fromData[pos2]
        then
          assign (fromData, toData, pos1, pos)
        else
          assign (fromData, toData, pos2, pos);
    pos := pos + 1
  end {while}
end {merge}
```

```
procedure assign (fromData, toData,
                     fromPos, toPos)
begin
  toData[toPos] := fromData[fromPos];
  fromPos := fromPos + 1;
end {assign}
```

Bei assign muss der Parameter fromPos als call by reference deklariert werden !

Komplexitätsklassen

Bewertung von Algorithmen

Gegeben ein Algorithmus:
Finde die günstigste Komplexitätsklasse bzgl. **Laufzeit** und Speicherplatz:

- **im ungünstigsten Fall (worst case)**
- im Durchschnittsfall (average case)

Beispiele:

	Laufzeit:	Speicherplatz:
Lineare Suche:	O(n)	O(n)
Binärsuche:	O(log n)	O(n)
Selectionsort:	$O(n^2)$	O(n)
Mergesort:	O(n log n)	O(n)
Quicksort:	$O(n^2)$ **w.c.** O(n log n) **a.c.**	O(n)

Komplexitätsklassen

Bewertung von Problemen

Gegeben ein Problem:
Finde die günstigste Komplexitätsklasse, zu der es einen Algorithmus gibt, der das Problem ***im allgemeinen Fall*** löst.

Ein Problem gehört bzgl. der Rechenzeit (des Speicherplatzes) zur Komplexitätsklasse **Ω(f(n))**, wenn es eine Konstante c gibt, so dass gilt:

Die Rechenzeit (Der Speicherplatz) ***jedes*** Algorithmus für das Problem der Größe n benötigt ***mindestens*** $c \cdot f(n)$ Rechenschritte (Speicherplätze).

Beispiele:

	Laufzeit:	Speicherplatz:
Allgemeine Suche:	$\Omega(n)$	$\Omega(n)$
Allgemeines Sortieren:	$\Omega(n \log n)$	$\Omega(n)$

Komplexitätsklassen

Wie macht man die Komplexitätsklasse unabhängig vom Computer ?

Der Schlüssel zum Erfolg: Die Turingmaschine

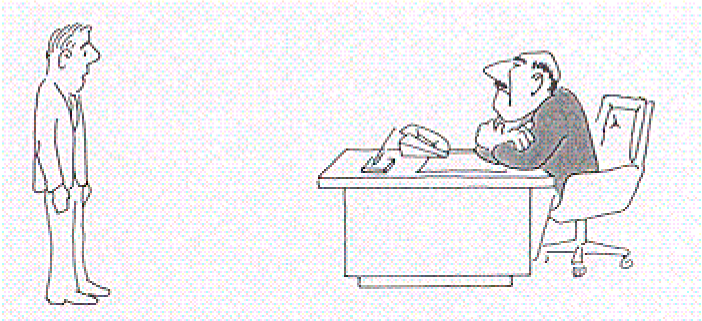
- von Alan Turing 1937 konstruierter theoretischer „Urcomputer“

Für jeden bis heute konstruierten Computer gilt:

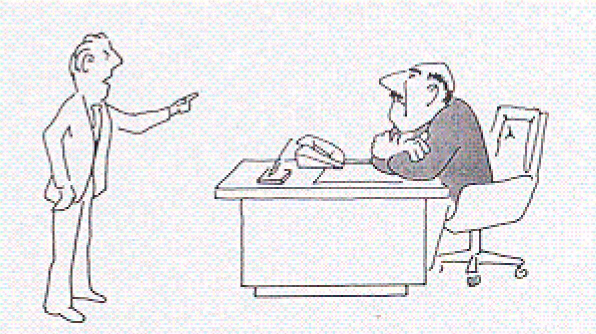
- Wenn ein Problem der Größe n
  auf einem beliebigen Computer lösbar ist mit f(n) Rechenschritten,
  dann ist es auch auf einer Turingmaschine lösbar mit O(P(f(n)))
  Rechenschritten für ein vom Computer abhängiges Polynom P.

NP-Vollständigkeit

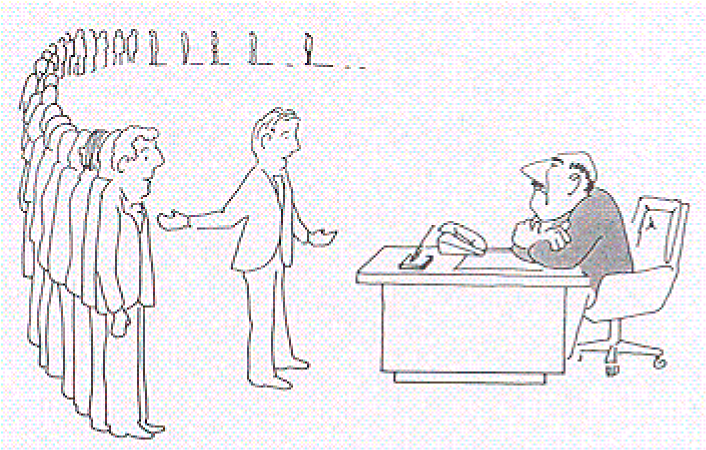
Was sagen Sie Ihrem Chef, wenn er Ihnen eine Aufgabe stellt,
die Sie nicht lösen können ?



Ich habe in GTI leider nicht aufgepasst,
als das behandelt wurde !



Das kann man nicht lösen !



Ich konnte das zwar nicht lösen,
aber diese berühmten Leute hier können das auch nicht !

nach: Michael Garey / David Johnson:
Computers and Intractability, WH Freeman 1979

NP-Vollständigkeit

NP-vollständige Probleme sind folgendermaßen charakterisiert:

- Das Problem ist auf einer (hypothetischen) **nichtdeterministischen** Turingmaschine in polynomialer Zeit (O(P(n)) für ein Polynom P) lösbar.

  Das sind die Probleme, deren Lösung man in polynomialer Zeit auf einer deterministischen Turingmaschine verifizieren kann.

- Jedes NP-vollständige Problem ist **genau dann** auf einer deterministischen Turingmaschine in polynomialer Zeit lösbar,
  wenn *alle* NP-vollständigen Probleme auf einer deterministischen Turingmaschine in polynomialer Zeit lösbar sind.

  Ein NP-vollständiges Problem ist also ein schwerstmögliches Problem, dessen Lösung man in polynomialer Zeit auf einer deterministischen Turingmaschine verifizieren kann.

Offene Frage der Informatik:

1) Sind NP-vollständige Probleme auf einer deterministischen Turingmaschine in polynomialer Zeit lösbar ?

2) Oder gehören sie zu einer Komplexitätsklasse Ω(f(n)), wobei f(n) stärker wächst als jedes Polynom P(n) ?