Infovorkurs, Teil I: Algorithmik

07.10.2015

Orga

Ich bin Gunnery Sergeant Hartman...
...Quatsch! Ich bin Lutz ☺

Meldet euch im MÜSLI bei Mathe für Informatiker an!
`www.mathi.uni-heidelberg.de/muesli/lecture/view/504`

Hier gibts später die Folien, außerdem nützliche Links zum Start hier in Heidelberg:
`www.geile-hirnbude.de/vorkurs`

Warum Computer?

Computer sind...

- schnell,
- genau,
- anspruchslos,
- und haben ein großes Gedächtnis.

Anders als Menschen!

Warum Informatiker?

Computer sind (noch)...

- unkreativ,
- stur,
- kurz: dumm.

Anders als Menschen!

Fragerunde!

www.govote.at, code 37 44 11

Was ist ein Algorithmus?

Häufigste Metapher: "Kochrezept"

1. verwandelt eine Eingabe in eine Ausgabe
2. löst ein Problem
3. endliche Beschreibung
4. Sequenz von einzelnen Schritten
5. braucht endlich viele Schritte
6. nach jedem Schritt ist klar, welcher nächste Schritt folgt

Ist das alles?

Algorithmus = Kochrezept?

Ein Kochrezept ist zu beschränkt!

- Nur ein Input möglich (200g Mehl, 125g Butter, 125g Zucker, 2 Eier...)
- Nur ein Output möglich (ein bestimmter Kuchen)

Wir wollen mehr:

6. Abstraktion auf eine (große) Klasse von **Problemen**

Was ist ein Problem?

Gegeben Matrix und Vektor, löse lineares Gleichungssystem
Gegeben aussagenlogische Formel, entscheide ob erfüllbar
Gegeben zwei natürliche Zahlen, berechne größten gemeinsamen Teiler
Gegeben einen Graphen und zwei Knoten darin, finde kürzesten Weg

Unterscheide ein Problem von einer Probleminstanz (z.B. konkrete aussagenlog. Formel)

Suchproblem

Gegeben eine Liste bestimmter **Zutaten** (Kartoffeln, Spinat) und bestimmte **Nebenbedingungen** (vegan, glutenfrei, ohne Rosinen)

Finde - falls möglich - Rezepte, die alle Zutaten verbrauchen und alle Nebenbedingungen einhalten

`http://www.chefkoch.de/rezept-reste.php`

Optimierungsproblem

Gegeben, ein Menü mit Nährstofftabelle und Preisen
Gesucht wird die **günstigste** Kombination, die den Tagesbedarf deckt:

- Brennwert $\geq$ 2000kcal
- Eiweiss $\geq$ 75g
- Kohlenhydrate $\geq$ 275g
- Zucker $\geq$ 95g
- Fett $\geq$ 67g
- gesättigtes Fett $\geq$ 22g
- Ballaststoffe $\geq$ 25g
- Salz $\geq$ 5g

Probleminstanz

Input: Menü & Nährwerttabelle von **McDonald's**
Output:

- 2x Cheeseburger
- 2x Pommes Frites gross mit Ketchup
- 1x McChicken
- 1x Chili Dip
- 1x Apfeltasche
- 1x Fruchttüte
- 1x McSundae Schokosauce
- **Kosten: 10,83€**
- Brennwert: 11.010 kcal

Vergleichbare Werte bei Burger King!

Lineare Optimierung

Allgemeine lineare Optimierung:

- Erfülle (lineare) Ungleichungen als **Nebenbedingung** (im Beispiel: Tagesbedarf)
- Maximiere/minimiere (lineare) **Zielfunktion** (Kosten oder Brennwert)
- Lasse nur Integer-Werte als Lösung zu (keine halben Burger)

Zusammenfassung Algorithmus

- verwandelt eine Eingabe in eine Ausgabe: $Alg(x, y)$
- löst damit eine (große) Klasse von Problemen $x \in X$
- nach jedem Schritt ist klar, welcher nächste Schritt folgt, insbesondere
  $\forall y, \tilde{y} : Alg(x, y) \wedge Alg(x, \tilde{y}) \Rightarrow y = \tilde{y}$

Das heißt, ein Algorithmus berechnet eine Funktion!
$\forall x \in X \exists! y \in Y : Alg(x, y)$

Achtung! Dies ist auch der Fall, falls das Problem an sich keine eindeutige Lösung hat:

Es gibt z.B. im Allgemeinen mehrere kürzeste Wege in einem Graphen
Aber ein Algorithmus gibt für jede Instanz eine konkrete Lösung an (oder einfach alle)

Fragerunde!

www.govote.at, code 56 84 06

Problem: Max

Gegeben eine (nichtleere) Menge von natürlichen Zahlen
$L = \{x_1, ...x_n\}$
Gesucht ist die Funktion, die das **Maximum** dieser Zahlen ausgibt
Schauen wir uns verschiedene Algorithmen an, die diese Funktion berechnen

Algorithmus 1

```
n ← |L|
for i = 1, ..., n do
    a ← x_i
    isMax ← True
    for j = 1, ..., n do
        b ← x_j
        if a < b then
            isMax ← False
        end
    end
    if isMax then
        max ← a
    end
end
return max
```

Algorithm 1:

Algorithmus 2

```
n ← |L|
for i = 1, ..., n do
    a ← x_i
    isMax ← True
    for j = 1, ..., n do
        b ← x_j
        if a < b then
            isMax ← False
        end
    end
    if isMax then
        max ← a
        return max
    end
end
```

Algorithm 2:

Algorithmus 3

```
n ← |L|
for i = 1, ..., n do
    a ← x_i
    isMax ← True
    for j = i + 1, ..., n do
        b ← x_j
        if a < b then
            isMax ← False
        end
    end
    if isMax then
        max ← a
        return max
    end
end
```

Algorithm 3:

Algorithmus 4

$n \leftarrow |L|$
$max \leftarrow x_1$
for $i = 2, \ldots, n$ **do**
  if $max < x_i$ **then**
    $max \leftarrow x_i$
  end
end
return max

Algorithm 4:

Laufzeiten

Eingabe	**Algo1**	**Algo2**	**Algo3**	**Algo4**
$[1, \ldots, 10]$	$3 * 10^{-6} s$	$3 * 10^{-6} s$	$7 * 10^{-7} s$	$2 * 10^{-7} s$
$[1, \ldots, 10^2]$	$7 * 10^{-5} s$	$7 * 10^{-5} s$	$1 * 10^{-5} s$	$4 * 10^{-7} s$
$[1, \ldots, 10^4]$	**0.5s**	**0.5s**	$1 * 10^{-3} s$	$3 * 10^{-5} s$
$[1, \ldots, 10^6]$	**8min**	**8min**	**4min**	$2 * 10^{-3} s$
$[1, \ldots, 10^8]$	**57d**	**57d**	**28d**	$0,2 s$
$[10^8, 1, \ldots, 10^8 - 1]$	**57d**	$0,2 s$	$0,2 s$	$0,2 s$

Beobachtungen

- Das Ergebnis hängt nicht vom Algorithmus ab, die Laufzeit schon
- Laufzeit wächst mit der Eingabegröße
- Aber auch nicht immer!

Wie können wir ein Maß definieren für die allgemeine Laufzeit eines Algorithmus?

worst-case-Komplexität

- Messe Dauer in Abhängigkeit von der Eingabegröße
- Untersuche dabei den schlimmsten (d.h. langsamsten) Fall als Abschätzung
- Zeiteinheit sei eine "elementare Operation"
- Vergleich von diesen Laufzeitfunktionen "asymptotisch"

Algorithmus 1

```
n ← |L|
for i = 1, ..., n do
    a ← x_i
    isMax ← True
    for j = 1, ..., n do
        b ← x_j
        if a < b then
            isMax ← False
        end
    end
    if isMax then
        max ← a
    end
end
return max
```

Algorithm 1:

Algorithmus 1

```
1
for i = 1, ..., n do
    1
    1
    for j = 1, ..., n do
        1
        if a < b then
            1
        end
    end
    if isMax then
        1
    end
end
1
```

Algorithm 1:

Algorithmus 1

```
1
for i = 1, ..., n do
    1
    1
    for j = 1, ..., n do
        1
        2
    end
    2
end
1
```

Algorithm 1:

Algorithmus 1

```
1
for i = 1, ..., n do
    1
    1
    for j = 1, ..., n do
        1
        2
    end
    2
end
1
```

Algorithm 1:

Algorithmus 1

```
1
for i = 1, ..., n do
    2
    for j = 1, ..., n do
        3
    end
    2
end
1
```

Algorithm 1:

Algorithmus 1

1
for $i = 1, \ldots, n$ **do**
  2
  $3n$
  2
end
1

Algorithm 1:

Algorithmus 1

1
for $i = 1, \ldots, n$ **do**
 | $3n + 4$
end
1

Algorithm 1:

Algorithmus 1

1
for $i = 1, \ldots, n$ **do**
 | $3n + 4$
end
1

Algorithm 1:

Algorithmus 1

$$n \cdot (3n + 4) + 2$$
$$= 3n^2 + 4n + 2$$

Algorithm 1:

Vergleich

Algo1:
$n \cdot (3n + 4) + 2$
$= 3n^2 + 4n + 2$

Algo2:
$3n^2 + 4n + 2$
Gleiches worst-case-Verhalten!

Algo3:
$3 + \sum_{i=1}^{n} (3 + (n - i) \cdot 3) =$
$= \frac{3}{2}n^2 + \frac{3}{2}n + 3$
(umordnen,ausklammern,kleiner Satz von Gauß)

Algo4:
$2n + 1$

Analyse

- Algo 1 und Algo 2 sind im worst case gleich langsam
- Algo 3 ist grob doppelt so schnell wie Algo 2

Eine Optimierung um den Faktor 2 ist gut.

Wir sind allerdings interessiert an ganzen Problemklassen. Da spielt das Wachstum in Abhängigkeit von n die größte Rolle:

- Wenn das Problem sich in seiner Größe verdoppelt, braucht Algo 4 rund doppelt so lang
- Algo 1, 2 und 3 brauchen (mehr als) 4 mal so lang als zuvor

$\mathcal{O}$-Notation

Wir wollen konstante Summanden und Faktoren vernachlässigen, sie sind nicht wesentlich.
Für $f : \mathbb{N} \to \mathbb{N}$ definiere
$\mathcal{O}(f) = \{g : \mathbb{N} \to \mathbb{N} | \exists m \exists c \forall n : g(n) \leq m \cdot f(n) + c\}$

Man identifiziert Algorithmen der Einfachheit halber mit ihren Zeitkomplexitäten. Daher:

- Algo1, Algo2 und Algo3 $\in \mathcal{O}(n^2)$
- Algo4 $\in \mathcal{O}(n)$
- Übrigens auch Algo4 $\in \mathcal{O}(n^2)$
- Aber Algo1-3 $\notin \mathcal{O}(n)$

Sortierproblem

Gegeben eine endliche Liste von Zahlen $L = (a_1, \ldots, a_n)$
Gesucht ist die sortierte Liste dieser Zahlen:
$b_1 \leq b_2 \leq \cdots \leq b_n$
mit $\{a_1, \ldots, a_n\} = \{b_1, \ldots, b_n\}$

SelectionSort

Input: L
$M \leftarrow \emptyset$
while $L \neq \emptyset$ **do**
    $m \leftarrow max(L)$
    füge m hinten in M ein
    entferne m aus L
end
return M

Algorithm 1: SelectionSort

Komplexität SelectionSort

1
while $L \neq \emptyset$ **do**
  $\mathcal{O}(n)$
  1
  1
end
1

Algorithm 1: SelectionSort

Komplexität SelectionSort

$$n \cdot \mathcal{O}(n) + 4 = \mathcal{O}(n^2)$$

Algorithm 1: SelectionSort

MergeSort, Subroutine Merge

Input: $K = [K_0, \dots], L = [L_0, \dots]$ **sortierte endliche Listen**
$i \leftarrow 0$
$j \leftarrow 0$
$M \leftarrow \emptyset$
while $i < |K| \vee j < |L|$ **do**
  if $(i < |K| \wedge j < |L| \wedge K_i < L_j) \vee (i = |K| \wedge j < |L|)$ **then**
    füge L_j hinten in M ein
    $j \leftarrow j + 1$
  else
    /* hier gilt umgekehrt
      $(i < |K| \wedge j < |L| \wedge K_i \geq L_j) \vee (j = |L| \wedge i < |K|)$ */
    füge K_i hinten in M ein
    $i \leftarrow i + 1$
  end
end
return M

Algorithm 1: Merge

Merge hat Komplexität $\mathcal{O}(|K| + |L|)$

MergeSort

Input: L
$n \leftarrow |L|$
if $n = 1$ **then**
  return L
end
$L_v \leftarrow$ erste $\lceil \frac{n}{2} \rceil$ Elemente
$L_h \leftarrow$ letzte $\lfloor \frac{n}{2} \rfloor$ Elemente
$K_v \leftarrow \text{MergeSort}(L_v)$
$K_h \leftarrow \text{MergeSort}(L_h)$
return $\mathit{Merge}(K_v, K_h)$

Algorithm 1: MergeSort

MergeSort ist rekursiv, d.h. er ruft sich selbst auf!

Komplexität MergeSort

Betrachte den Ablauf bei einer Liste aus 8 Elementen:
$(a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8)$
$(a_1, a_2, a_3, a_4)(a_5, a_6, a_7, a_8)$
$(a_1, a_2)(a_3, a_4)(a_5, a_6)(a_7, a_8)$
$(a_1)(a_2)(a_3)(a_4)(a_5)(a_6)(a_7)(a_8)$

Der kollektive Aufwand für Aufteilen sowie Merge() ist in jeder Zeile $\in \mathcal{O}(|L|)$
Wie lang wird diese Liste (relativ zu n)?

Komplexität MergeSort

$(a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8)$
$(a_1, a_2, a_3, a_4)(a_5, a_6, a_7, a_8)$
$(a_1, a_2)(a_3, a_4)(a_5, a_6)(a_7, a_8)$
$(a_1)(a_2)(a_3)(a_4)(a_5)(a_6)(a_7)(a_8)$

Jedes Mal halbiert sich die Länge der individuellen Listen, bis die maximale Länge 1 ist. Falls n keine 2er-Potenz ist, schätzt man nach oben mit der nächsthöheren ab. Damit sind es höchstens $\mathcal{O}(\lceil log_2(n) \rceil + 1) = \mathcal{O}(log_2(n)) = \mathcal{O}(log(n))$ viele Zeilen.
Der Gesamtaufwand ist damit in $\mathcal{O}(n \cdot log(n))$. Es gilt $\mathcal{O}(n \cdot log(n)) \subsetneq \mathcal{O}(n^2)$.

Damit ist MergeSort schneller als SelectionSort. Man kann sogar zeigen, dass $\mathcal{O}(n \cdot log(n))$ optimale worst-case-Schranke für das Sortierproblem ist.