Grundlagen

(Algorithmen und Datenstrukturen I)

Prof. Dr. Oliver Braun

Letzte Änderung: 18.03.2018 18:16

Algorithmus

- es gibt keine präzise Definition
- “Handlungsvorschrift”
- wichtig: **Algorithmus** vs. **Realisierung**
- ein Algorithmus kann “gut” sein, die Realisierung aber “schlecht”
- formale Eigenschaften:
  - Korrektheit — Beweis nicht durch Testen möglich!
  - Effizienz — durch Messen oder Betrachtung der Komplexität

Problem vs. Probleminstanz

- **Problem** = das was der Algorithmus lösen will
  - z.B. das Problem eine Liste zu sortieren
- **Probleminstanz** = konkrete Eingabe
  - z.B. die Liste 4, 7, 2, 1, 5
- interessant wäre zu betrachten, wie effizient der Algorithmus im Durchschnitt für eine Probleminstanz der Größe N ist
  - aber wie soll der Durchschnitt ermittelt werden?
- deshalb: Worst-Case-Analyse

Groß-Oh — O

- für die Laufzeit $T(N)$ eines Algorithmus in Abhängigkeit von der Problemgröße N gilt für alle N:
  - $T(N) \leq c_1 \cdot N + c_2$ mit zwei Konstanten c_1 und c_2
  - sagt man $T(N)$ *ist von der Größenordnung* N
    - oder $T(N)$ *ist (in)* $O(N)$
- genauer

$$O(f) = \{g | \exists c_1 > 0 : \exists c_2 > 0 : \forall N \in Z^+ : g(N) \leq c_1 \cdot f(N) + c_2\}$$

- üblicherweise schreibt man (nicht ganz korrekt):

$O(N)$, $O(N^2)$, $O(NlogN)$,

- **Abschätzung des Wachstums nach oben**

Beispiel

- die Funktion
$$g(N) = 3N^2 + 6N + 7$$
- ist $O(N^2)$
- wähle z.B. $c_1 = 9$ und $c_2 = 7$
- Erinnerung:

$$O(f) = \{g | \exists c_1 > 0 : \exists c_2 > 0 : \forall N \in Z^+ : g(N) \leq c_1 \cdot f(N) + c_2\}$$

- allgemein ist ein Polynom vom Grade k von der Größenordnung $O(N^k)$

Groß-Omega — Ω

- **Abschätzung des Wachstums nach unten**
- erster Ansatz (präzise)

$$\Omega(g) = \{h | \exists c > 0 : \exists n_0 > 0 : \forall n > n_0 : h(n) \geq c \cdot g(n)\}$$

- danach gilt also $f \in \Omega(g)$ genau dann, wenn $g \in O(f)$
- Forderung zu scharf
  - Beispiel: Funktion $f(N)$ die für alle geraden N den Wert 1 und für alle ungeraden N den Wert N^2 hat.
  - dann können wir nur abschätzen $f \in \Omega(1)$ obwohl für unendlich viele N gilt $f(N) = N^2$
- daher: Ansatz den wir verwenden:

$$\Omega(g) = \{h | \exists c > 0 : \exists \text{ unendlich viele } n : h(n) \geq c \cdot g(n)\}$$

Groß-Theta — Θ — & das Übliche

- gilt für f sowohl $f \in O(g)$ als auch $f \in \Omega(g)$ schreiben wir

$$f \in \Theta(g)$$

- für unsere Betrachtungen sind die wichtigsten Funktionen in Abhängigkeit von der Problemgröße N:
  - logarithmisches Wachstum: $logN$
  - lineares Wachstum: N
  - $NlogN$-Wachstum: $N \cdot logN$
  - quadratisches, kubisches, ... Wachstum: N^2, N^3, ...
  - exponentielles Wachstum: 2^N, 3^N, ...
- heute allgemeine Überzeugung:
  - höchstens solche Algorithmen praktikabel, deren Laufzeit durch ein Polynom in der Problemgröße beschränkt bleibt

Beispiel: Maximum-Subarray-Problem

- gegeben sei eine Folge X von N ganzen Zahlen in einem Array
- gesucht ist die maximale Summe aller Elemente in einer zusammenhängenden Teilfolge
- Probleminstanz:

$$31, -41, 59, 26, -53, 58, 97, -93, -23, 84$$

- maxmimale Teilsumme: $59 + 26 + (-53) + 58 + 97 = 187$

Naives Verfahren

```
maxTSum = 0
for (u von 1 bis N) {
  for (o von u bis N) {
    bestimme die Summe der Teilfolge X[u..o]
    Summe = 0
    for (i von u bis o) {
      Summe += X[i]
    }
    maxTSum = max(Summe, maxTSum)
  }
}
```

- einfach aber ineffizient:

$$\sum_{u=1}^{N} \sum_{o=u}^{N} \sum_{i=u}^{o} 1 = \Theta(N^3)$$

Divide-and-conquer-Prinzip zur Lösung

- Idee:
  - wird eine gegebene Folge in der Mitte geteilt, liegt die maximale Teilfolge
    - entweder ganz in einem der Teile
    - oder sie umfasst die Trennstelle
  - im letzteren Fall gilt
    - die Summe der Teilfolge ist maximal unter allen Teilfolgen die das Randelement an der Trennstelle enthalten
- wir nennen die maximale Summe von Elementen, die das linke bzw. rechte Randelement enthält
  - das linke bzw. rechte Randmaximum

Berechnung des Randmaximums

- das linke Randmaximum $lmax$ für eine Folge $X[l], ..., X[r]$ kann man in $\Theta(r - l)$ Schritten bestimmen:

```
lmax = 0
summe = 0
for (i von l bis r) {
  summe += X[i]
  lmax = max(lmax, summe)
}
```

- entsprechend kann man auch das rechte Randmaximum $rmax$ mit linearem Aufwand berechnet werden

Berechnung der maximalen Teilsumme

Algorithmus maxtsum(X):

```
wenn X nur ein Element a enthält ist die maximale Teilsumme
  (a > 0) ? a : 0
sonst
 teile die Folge in zwei annähernd gleiche Teile A und B
  maxtinA = maxtsum(A)
  maxtinB = maxtsum(B)
  rmax(A) = rechtes Randmaximum von A
  lmax(B) = linkes Randmaximum von B
  maxtsum = max(maxtinA, maxtinB, rmax(A) + lmax(B))
```

Laufzeit

- sei $T(N)$ die Anzahl der Schritte um $maxtsum$ für eine Folge der Länge N zu bestimmen
- dann gilt:

$$T(N) = 2 \cdot T\left(\frac{N}{2}\right) + \text{Const} \cdot N$$

- da $T(1)$ konstant ist, bekommt man

$$T(N) = \Theta(NlogN)$$

- besser als das naive Verfahren, aber es geht noch besser...

Das Scan-line-Prinzip

- Idee: wir betrachten eine aufsteigend sortierte, lineare Folge von Inspektionsstellen
  - hier: die Positionen $1, \ldots, N$ der Eingabefolge
- wir inspizieren die Stellen der Reihe nach und führen eine vom Problem abhängige, dynamisch veränderliche Information mit
  - hier: die maximale Summe $bisMax$ des bisherigen Teilstücks und das an der Inspektionsstelle endende rechte Randmaximum $ScanMax$
- nehmen wir an wir haben bereits bis Länge l inspiziert
  - wenn wir das $l + 1$-te Element a hinzunehmen
    - dann liegt die neue maximale Teilfolge entweder bereits in den ersten l Positionen
    - oder sie enthält das neu hinzugenommene Element a

Pseudocode

```
ScanMax = 0
bisMax = 0
solange Q = {1,...,N}
  q = nächstes Element von Q
  a = Element an Position q
  // update von ScanMax und bisMax
  if ScanMax + a > 0
    then ScanMax = ScanMax + a
    else ScanMax = 0 // leere Teilfolge
  bisMax = max(bisMax, ScanMax)
```

- dieser Algorithmus ist in linearer Zeit ausführbar, d.h. er braucht nur $\Theta(N)$ Schritte
- dieser Algorithmus ist asymptotisch optimal, da es keinen Algorithmus geben kann, der nicht mindestens jedes Element einmal betrachtet

Die richtige Wahl einer Datenstruktur

- Algorithmen hängen von den genutzten Datenstrukturen ab
- Beispiel: Telefonbuch
  - effizienter Algorithmus zum Suchen einer Telefonnummer zu einem Namen möglich
  - Algorithmus zur "Rückwärtssuche" nur *brute-force* möglich
- gegeben sei Menge von Daten und eine Folge von Operationen mit diesen Daten
- finde Speicherform für die Daten und Algorithmen für die auszuführenden Operationen so, dass die Operationen in der gegebenen Folge möglichst effizient ausführbar sind

Abstrakter Datentyp (ADT)

- es ist heute üblich Daten und Operationen mit den Daten als Einheit aufzufassen
- diese Einheit nennt man **Abstrakter Datentyp**, kurz **ADT**
- ein ADT besteht aus
  - einer oder mehreren Mengen von Objekten[1]
  - darauf definierte Operationen

[1] Gemeint sind Mathematische Objekte. Ein ADT hat nichts mit Objekten in der OOP zu tun!

Beispiel: Der ADT Polynom

- enthält als Menge der Objekte:
  - die Menge der Polynome mit ganzzahligen Koeffizienten
- als Menge der Operationen, z.B.
  - genau die Addition und Multiplikation von zwei Polynomen
- nimmt man z.B. die erste Ableitung dazu, hat man einen **anderen** ADT
- denn: *ADT ist Einheit von Daten und Operationen*

Datentypen, ADTs und Datenstrukturen?

Datentypen In der Programmiersprache üblicherweise vorhandene Datentypen, wie z.B. `int`, `double`, ...

Abstrakte Datentypen Mathematisches Konzept. Eine oder mehrere, mit den üblichen mathematischen Methoden festgelegten Mengen von Objekten und darauf definierten Operationen.

Datenstrukturen Realisierung der Objektmengen eines ADT mit den Mitteln einer Programmiersprache, auch mit Speicherstruktur oder Implementierung eines ADT bezeichnet.

Der ADT Lineare Liste

- Menge der Objekte
  - Menge aller endlichen Folgen von Elementen eines Grundtyps[2]
- Operationen
  - *Einfügen(x,p,L)*: Einfügen des neuen Elementes x in die Liste L an die Position p
  - *Entfernen(p,L)*: Entfernen des Elements an der Position p aus der Liste L
  - *Suchen(x,L)*: Gibt die Position (von links erste) Position von x in der Liste L an, *0* sonst
  - *Zugriff(p,L)*: Liefert das Element an Position p aus der Liste L, undefiniert sonst
- evtl. für weitere Anwendungsfälle mehr Operationen, z.B. Verketten von Listen, Extrahieren von Teillisten

[2] Streng genommen müsste man für jeden Grundtyp einen eigenen ADT angeben.

Mögliche Implementierungen von Listen

1. **Sequentiell gespeicherte lineare Listen**

   Listenelemente sind in eine zusammenhängenden Speicherbereich abgelegt, so dass über eine Adressberechnung auf das i-te Element zugegriffen werden kann.

2. **Verkettet gespeicherte lineare Listen**

   Listenelemente in Speicherzellen abgelegt, deren Zusammenhang wird durch Zeiger hergestellt wird.

Sequentielle Speicherung linearer Listen

- Implementierung: ArrayList
- um in einer sequentiell gespeicherten Liste der Länge N ein Element einzufügen oder zu entfernen, müssen offenbar im ungünstigsten Fall $\Omega(N)$ Elemente verschoben werden
- da alle Positionen gleich wahrscheinlich sind, kann man davon ausgehen, dass im Mittel die Hälfte aller Elemente verschoben werden muss
- das Suchen benötigt auch im Mittel und im schlechtesten Fall $\Theta(N)$ Schritte

Verkettete Speicherung linearer Listen

- Implementierung und Effizienzbetrachtungen
  - Aufgabenblatt 4

Stapel und Schlange

- wie Liste, aber
  - bei Stapel (*stack*)
    - Einfügen nur vorne
    - Entfernen nur vorne
  - bei Schlange (*queue*)
    - Einfügen nur hinten
    - Entfernen nur vorne