DIVIDE AND CONQUER

Rekursionsungleichungen

Vollständige Induktion

Beispiel: MergeSort

Konstruktive Induktion

Unbekannte Koeffizienten werden innerhalb der Induktion bestimmt

Beispiel: FIBONACCI

F(n) = **n** für n = 0, 1
f (n-1) + f(n-2) für n>1

Behauptung: für $n \geq n_0$ gilt:

$F(n) \geq a \cdot c^m$
Mit Konstanten a>0 und c>0, die noch zu bestimmen sind.

Induktionsschritt:
$F(n) = f(n-1) + f(n-2) \geq a \cdot c^{n-1} + a \cdot c^{n-2}$

Konstruktiver Teil:
Wir bestimmen ein **c>0** mit der Forderung $\mathbf{a \cdot c^{n-1} + a \cdot c^{n-2} \stackrel{!}{\geq} a \cdot c^n}$ | $\cdot 1/a \quad \cdot 1/c^{n-2}$
Umformen: $\mathbf{c + 1 > c^2}$ oder $\mathbf{c^2 - c - 1 \leq 0}$
Lösung: $\mathbf{c} \leq \frac{1+\sqrt{5}}{2}$ (negative Lösungen scheiden aus)

Induktionsanfang:
Wähle a>0 und n_0 geeignet.

WARNUNG: Der exakte Wert der Konstanten ist wichtig in der Ungleichung !

BEISPIEL: $\mathbf{T(n) = 2 \cdot T(n / |2|) + n}$

FALSCH: Beh.: $\mathbf{T(n) \leq c \cdot n \in O(n)}$
„Beweis“: $T(n) \leq 2 \cdot (c \cdot (n/2) + n)$
$\leq c \cdot n + n$
$= (c+1) \cdot n$
$\notin O(n)$

was ist falsch? → Behauptet wurde: $\mathbf{T(n) \leq c \cdot n}$
Geeignet wäre: $\mathbf{T(n) \leq (c+1) \cdot n}$

1.5 Random-Access-Maschine

wir betrachten wieder Kosten für Opeationen

bisher: Einheitsmaß oder uniformes Komplexitätsmaß
Kosten unabhängig von Wortlänge des Operanden

Lösung des Operanden wird GROSS:

- ADD: O(n) n=Zahl der BITS
- MULT: $O(n^2)$ „

logarithmide Maß der Bit-Komplexität
Kosten hängen von der Wortlänge des Operanden, d.h. Zahl der Bits, ab.

1.6 Abstrakte Datentypen

Beispiel: STACK

Opeationen: definieren die Syntax des Datentypen

Stackinit → Stack
Stackempty: Stack → Boolean

push: Element D Stack → Stack
pop: Stack → Element D Stack (nimmt erstes Element vom Stack)

Axiome: definieren die Semantk des Datentyps

..... (wieder siehe Skript S.. 50)

1.

2. Sortierverfahren

2.1. Einführung

Ordnungsprinzip erst seit dem Mittelalter. Grund: Elemente können schneller gefunden werden.

Kriterien für die Klassifizierung von Sortierverfahren: *siehe S. 52*
BucketSort = Fachverteilen, wie bei der Post. (man schaut nur, wie OFT a vorkommt.)

2.2. Elementare Sortierverfahren

elementar = Komplexität ist elementar.

2.2.1. SelectionSort: Sortieren durch auswählen

für das erste Element das ganze Feld durchgehen und das kleinste raussuchen. Dieses an die 1. Position stellen (ducrh Tausch).
Das ganze dann mit dem Rest (alles außer dem 1. Element) wiederholen.

3 7 8 6 4 **2**
2 7 8 6 4 3 ect. , siehe Skript Seite 53

Bewegungen: 3E(N-1)
Vergleiche: $N^2 / 2$

2.2.2. InsertionSort: Sortieren durch einfügen

wie beim Karten-sortieren auf der Hand beim Kartenspielen:

wir betrachten die Folge von links an. Sobald das erste Element kommt, das nicht mehr gößer als das davor ist, wird es so lange nach links geschoben, bis es genau an der richtigen Position liegt, usw.

2.2.3. BubbleSort

nun, ich denke das kennen wir alle:
Wir „bubblen“ die unterste Zahl so lange nach oben, bis sie auf eine noch größere trifft. Dann wird die „weiter-gebubbled“, usw. Am Ende steht also die größte Zahl ganz oben (bzw. rechts), und wir können das ganze mit dem Rest wiederholen.

2.2.4. Indirektes Sortieren

K: alle Werte durchnummerieren
A[k]: der Wert selbst.
P[k]: die Speicherplatznummer, die als nächstes in der sortierten Reihenfolge dran ist.

Problem: am Ende alles physikalisch zu sortieren.
Lösung: Per „Ringtausch“: 1 Objekt auslagern.

2.2.5. BucketSort:

Wir erstellen ein Array mit jedem möglichen vorkommenden Wert. Dann wird eingetragen, wie oft der Wert jeweils vorkommt.

2.3. QuickSort

1. setze rechtes Element einer zu sortierenden Unterliste auf das Pivot-Element „v“.
2. laufe mit dem Zeiger i von links solange durch die Liste, bis ein Element kommt, das größer ist als v, bleibe dort stehen.
3. laufe mit dem Zeiger j von rechts so lange durch die Liste, bis ein Element kommt, dass kleiner ist als v, und bliebe dort stehen.
4. Vertausche die Elemente unter i und j. Laufe dann weiter, und zwar so lange, bis sich i und j sich kreuzen.
5. Fall 1:
    a. i steht jetzt 1 über j, zwischen i und j ist praktisch der „Trennstrich“: alles links von ihm ist kleiner v, alles rechts ist größer v.
    b. die (blöde?) Prozedur vertauscht nun noch mal: links vom Strich steht dann ein größeres, rechts ein kleineres Element.
    c. Es wird nun also noch mal zurückgetauscht, und das größere Element mit dem PIVOT vertauscht.
    d. FERTIG.
6. Fall 2:
    a. I und j stehen übereinander:
    b. Hin und zurücktauschen wie oben, dann dieses mit dem PIVOT vertauschen → fertig.

Diese Prozedur wird nun rekursiv aufgerufen, und zwar für den Bereich LINKS vom Pivot, dann für den Bereich RECHTS vom Pivot-Element.

Varianten und Verbesserungen:

- PIVOT-Element: v:=(a[l]+a8[r] / 2 ;
- PIVOT-Element: medion-of-three
  - kein Sentinel-Element nötig
  - worst-case nicht weniger Wahrscheinlichkeit
  - insgesamt: 5% kürzere Laufzeit
- QuickSort ist ineffizient bei kleinen Arrays, z.B. M=12 oder 22
  Abhilfe:
- Speicherplatzbdarf:
  - indirekt über den Stack wegen Rekursion
  - Beschränkung auf 2 – ld n ist möglich: bearbeite zuerst den kürzeren der beiden Teilarrays
- Iterative Variante von QuickSort

2.4. HeapSort

Wir stellen uns unsere zu sortierende Reihe als einen Baum vor.
„Heap-Eeigenschaft“: im Baum sind dir Zahlen geordnet, d.h. das „Mutterblatt“ ist immer größer.

2.5. Untere und obere Schranke

2.6. Schranken für n!

3. Suchen in Mengen

3.1. Einrührung

Man denke sich eine Datenbank.

U = Universum: alle möglichen Schlüssel.
S = vorkommende Schlüssel

Seach(x,S), Insert(x,S), Delete(x,S)

→ Operationen lernen! (S.82)

3.2. Einfache Implementierungen

3.2.1. Ungeordnete Arrays und Listen

Menge als Array oder Liste implementieren.

3.2.2. Vergleichsbasierte Methoden

Ordnung auf der Menge nötig.

Suchen im geordneten Array!

1. Sequentielle / lineare Suche: alle einzeln durchgehen.
2. Binärsuche: immer halbieren.
3. Interpolationssuche: (wenn ca. linear): vorher etwa Position bestimmen.

3.2.3. Bitvektordarstellung (kleines Universum)

Wir stellen ein Bit-Array mit allen möglichen Werten dar. (z.B. 1..100). Wenn nun ein Wert daraus in die Liste soll, wird das Bit auf 1 gesetzt.

3.2.4. Kleines Universum

„lazy initialisation“: spart Zeit bei der Initialisierung:

Bit-Array: alle möglichen Werte.
Pointer-Array: parallel dazu Pointers auf Werte im Stack
Stack: hier liegen die Werte. Egal wo ein neues Element dazukommt: es wird an der nächstmöglichen Position eingefügt.

3.3. Hashing

3.3.1. Grundbegriffe

Bei 1000 möglichen Werten nicht mehr 1000 Speicherplätze (für jeden einen), sonden weniger (z.B. 100), und dann Verteilung auf diese 100 nach Schlüsel (z.B. 329 und 429 beide auf 29 → Kollisionskontrolle nötig, s.u.:
Wenn Kollision:

- entweder Verweis auf Ausweichadresse
- oder alternative Hashfunktion

3.3.2. Hashfunktionen

Divisions-Rest-Methode:

H = x mod m (s.o.: 1000 mögl. Werte, 100 Plätze → nur 2 Stellen speichern (325 und 425 beide auf 25)

Mittel-Quadrat-Methode:

325 speichern auf VorletzteZweiStellen(325^2)

3.3.3. Kollisionsstrategien (Sondieren)

Wahrscheinlichkeit, daß keine Kollision auftritt:

m = 100: Größe der Hashtabelle
n = 10: Anzahl der einzutragenden Elemente

Die Wahrscheinlichkeit keiner Kollision ist nun beim 1. Element 100%, beim 2. 99%, beim 3. 98% usw.
Also: 100%E 99% E 98% E 97% E 96% E 95% E 94% E 93% E 92% E91% = **62,8%**

Approximation: die Wahrscheinlichkeit bleibt gleich, wenn m (100) quatratisch mit n (10) wächst.

Offenes Hashing (Hashing mit Verkettung):
Bei einer Kollision wird einfach das nächste Element an dieselbe Stelle noch drangehängt.

Geschlossene Hashverfahren
Hierbei wird ein Array verwendet. Bei einer Kollision gibt es:

1.: Lineares Sondieren:
es werden einfach die folgenden Speicherplätze durchprobiert, ob Platz ist.
NACHTEIL: Häufung von „vollen" Bereichen, Label „deleted" nötig, damit der Compiler weiß, dass er weitersuchen soll (und nicht schon an einem leeren Platz angelangt ist.)

2.: Quadratisches Sondieren:
Wir gehen 1,2,3,4... Plätze weiter, sondern 1^2, 2^2, 3^2, 4^2, ...

3.3.4. Komplexität des geschlossenen Hashings

?????

3.3.5. Zusammenfassung der Hashverfahren

- effizient: zwischen O(1) und O(n) (wenn er eine lange Liste an einer Hash-Position durchsuchen muss)
- Funktionen auf einer Ordnung (z.B. ListOrder) nicht möglich!
- ANWENDUNG: wenig Objekte bei grossem Wertebereich + schneller Zugriff nötig.
  BEISPIEL: Symboltabelle von Compilern.

3.4. Binäre Suchbäume

3.4.1. Allgemeine binäre Suchbäume

Wdh.: binäre Suche = unseren Abschnitt immer wieder halbieren.

4. 4. Graphen

4.1.1. 4.4 Kürzeste Pfade / Wege in Graphen:

- von Startknoten A zu Endknoten B: ein Pfad
- Startknoten A: Pfade zu *allen* anderen Knoten
- beste Pfade für *alle* Knotenpaare (A,B)

Single-Source Best Path: OIJKSTRA-Algorithmus, 1959

gerichteter Graph mit Bewertungsfunktion:
Kante von v nach w:

c[v,w] ³ **0** falls Kante (v → w) existiert, und ¥, falls die Kante nicht existiert.

AUFGABE:
Startknoten v0, Endknoten w.
Finde den Pfad von v0 nach w mit minimalen Gesamtkosen, d.h. eine Folge von Knoten
v0, v1, v2, ... , vn mit unbekannter Länge n und $v_n = w$.

ACHTUNG: Unterscheide:
Kurzer / langer Pfad = Zahl der Kanten
bester / optimaler Pfad: minimale Kosten, d.h. Summe der Zahlen über den Kanten möglichst klein.

Verallgemeinerung de Aufgabe:
Gegeben Startknoten v: Finde zu jedem Knoten w den besten Pfad von v0 nach w.

Anmerkungen:
- **Hinzufügen** einer Kante zu einem Pfad: Gesamtkosten **wachsen**.
- Der beste Pfad kann **keinen Zyklus** haben. (logisch).
- Sei w Knoten. Falls die besten Pfade von v0 zu allen anderen Knoten @ w höhere Kosten haben als der Pfad v0 → w, dann ist der beste Pfad v0 → w bereits gefunden.
  (logisch, denn wenn die Wege zu allen anderen Punkten mehr Kosten haben als zu w, müsste zu w ja mindestens ja noch was dazukommen. Wenn die Kosten zu w aber schon kleiner sind, *muss* das ja der beste Pfad sein.)
- Zu jedem Knoten kann der beste Pfad maximal |V|-1 Kanten haben.
  (|V| = Anzahl der Kanten. Logisch, denn würden wir eine Kante 2x besuchen, so hatte der Pfad ja einen Kreis und wäre somit nicht optimal.)

Kantenmenge V, n:= |V|, Startknoten V0:

ANSATZ: Wir bestimmen rekursiv eine Folge von paarweise verschiedenen Knoten $w_1, w_2, w_3, w_4,...,w_{n-1}$
- zu denen der beste Pfad bekannt ist und für die gilt: $C[v_0, w_{i-1}] \leq C[v_0, w_i]$ für i von 1 bis n-1.

Definition: $S_k = \{w_0, w_1, ..., w_k\}$

$D(S_k, v)$ = Kosten des besten Pfades von irgendeinem Knoten Knoten in S_k nach v (Pfad v_0→ w liegt in S_k mit Ausnahme der letzten Knoten)