Wiederholung

Divide & Conquer Strategie

- Binäre Suche $\mathcal{O}(\log n)$
  - Rekursives Suchen im linken oder rechten Teilintervall
- Insertion-Sort $\mathcal{O}(n^2)$
  - Rekursives Sortieren von a[1..n-1], a[n]
  - Einfügen von a[n] in sortierte Folge
- Merge-Sort $\mathcal{O}(n \log n)$
  - Rekursives Sortieren von a[1..n/2], a[n/2+1..n]
  - Mergen von zwei sortierten Teilintervallen
- Quicksort $\mathcal{O}(n \log n)$ bei geeigneter Wahl des Pivotelements p
  - Partitionierung in 2 Teilarrays mit Elementen größer/kleiner gleich p
  - Rekursives Partitionieren der Teilarrays

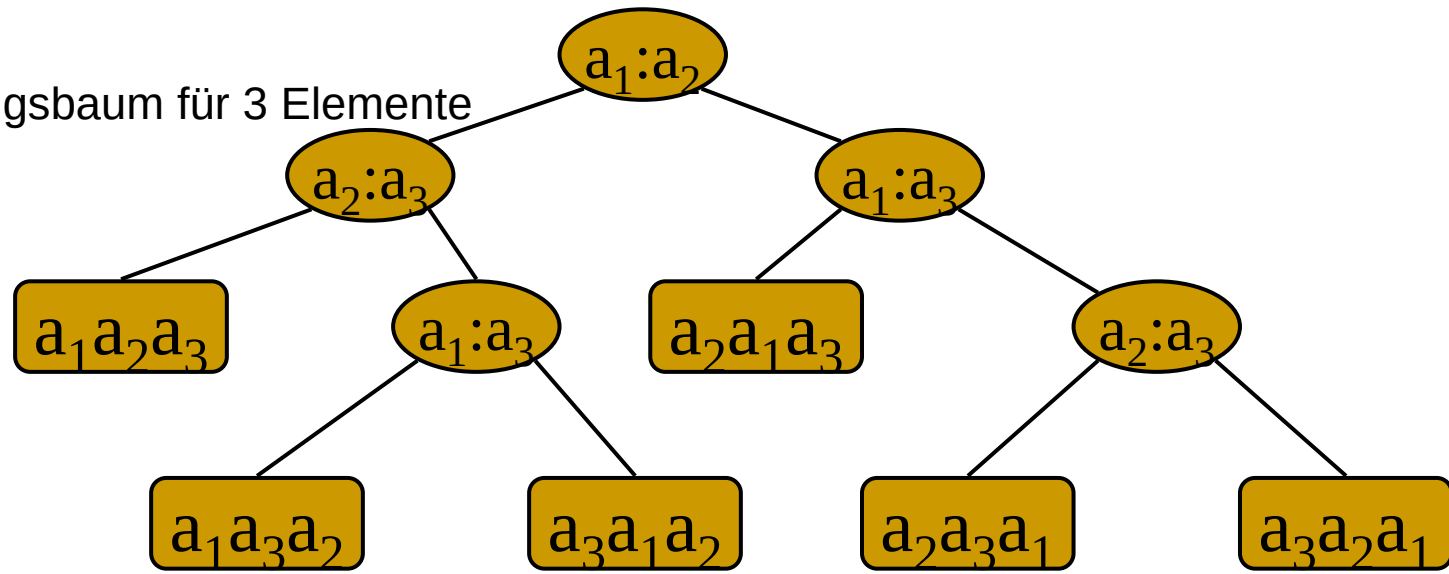
Entscheidungsbäume

Frage: Gibt es einen Algorithmus mit o(n logn) Vergleichen?

Sortieren mit Entscheidungsbäumen:

- Innere Knoten führen Vergleich a_i mit a_j durch.
  - Linkes Kind: $a_i \leq a_j$, rechtes Kind: $a_i > a_j$.
- Blätter enthalten Permutation $a_{\pi(1)},\ldots,a_{\pi(n)}$.
- Pfadlänge von der Wurzel zum Blatt entspricht Anzahl der benötigten Vergleiche.

Bsp.:Entscheidungsbaum für 3 Elemente



Untere Schranke

Satz: Jeder vergleichsbasierte Sortieralgorithmus benötigt zum Sortieren von n Elementen $\Omega(n \log n)$ Vergleiche im worst-case.

Betrachte Entscheidungsbaum T der Höhe h, der n Elemente sortiert.

- T hat mindestens n! viele Blätter (alle Permutationen).
- T hat höchstens 2^h viele Blätter (T ist Binärbaum).

  $\Rightarrow 2^h \geq n!$

  $\Rightarrow h \geq \log(n!)$ (Stirling: $n! > (n/e)^n$)

  $\Rightarrow h \geq n \log(n/e) = n \log n - n \log e = \Omega(n \log n)$.

Berechnung von Binomialkoeffizienten

Zur Erinnerung: $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$

Lösung mittels Divide and Conquer

Algorithmus Rekursiv-Binom

Eingabe: n, k

1. If (k=0) return 1;
2. If (n<k) return 0;
3. return Rekursiv-Binom(n-1,k-1) + Rekursiv-Binom(n-1,k);

Ausgabe: $\binom{n}{k}$

Korrektheit: folgt aus obiger Formel

Laufzeit von Rekursiv-Binom

Satz: Algorithmus Rekursiv-Binom benötigt zur Berechnung $\binom{2n}{n}$ von mindestens $4^n/(2n+1)$ Aufrufe.

- Rekursiv-Binom hat bei Abbruch als Rückgabewerte 0 und 1.
- $\binom{n}{k}$ wird als Summe von Rückgabewerten berechnet.
  $\Rightarrow$ Man benötigt mindestens $\binom{n}{k}$ Aufrufe.

$$\binom{2n}{n} \geq \binom{2n}{i} \quad \text{für } i = 0, 1, \ldots, 2n$$

$$\Rightarrow \quad (2n+1)\binom{2n}{n} \geq \sum_{i=0}^{2n} \binom{2n}{i} = 2^{2n}$$

$$\Rightarrow \quad \binom{2n}{n} \geq \frac{4^n}{2n+1}.$$

Dynamische Programmierung

Problem des Divide & Conquer Ansatzes:

- Subprobleme sind nicht voneinander unabhängig.
- Zwischenergebnisse werden nicht gespeichert.
- Dieselben Subprobleme werden mehrfach gelöst.

Dynamische Programmierung:

- Speicherung der Lösungen von Subproblemen
- Bottom-up Lösungsansatz:
  - Man beginnt bei trivialen Subproblemen.
  - Man kombiniert Subprobleme zu größeren Problemen.

Bsp. aus der Vorlesung:
Algorithmus von Warschall zur Berechnung der transitiven Hülle

Binomialkoeffizienten revisited

Berechnung der Binomialkoeffizienten mittels Dyn. Programmierung:

Algorithmus Dyn-Binom
Eingabe: n, k

1. for i $\leftarrow$ 0 to n
    1. a[i,0]=1; a[i,i] = 1;
2. for i $\leftarrow$ 1 to n
    1. for j $\leftarrow$ 2 to i-1
        1. a[i,j] $\leftarrow$ a[i-1,j-1] + a[i-1,j];

Ausgabe: a[n,k] = $\binom{n}{k}$

Korrektheit:

- Schritt 1: Initialisieren Grenze des Pascal‘schen Dreiecks mit Einsen.
- Schritt 2: Zeilenweise Berechnung des Pascal‘schen Dreiecks

Laufzeit: $\mathcal{O}(n^2)$, d.h. unabhängig von k.

Optimierungsprobleme

- Optimierungsprobleme besitzen mehrere Lösungen.
- Jede Lösung hat einen Wert.
- Suchen eine Lösung mit optimalem Wert.
  - Minimierungsproblem: Optimale Lösung hat minimalen Wert
  - Maximierungsproblem: Optimale Lösung hat maximalen Wert

Minimierung Matrizenmuplikation

- Seien $A_1,A_2,\ldots,A_n$ Matrizen.
- Ziel ist die effiziente Berechnung von $A_1*A_2*\ldots*A_n$.
- Matrixprodukt M ist vollständig geklammert $\Leftrightarrow$
  - M ist eine einzelne Matrix oder
  - M ist geklammertes Produkt von zwei vollständigen Matrixprodukten.

Bsp: $A_1*A_2*A_3*A_4$ besitzt fünf vollständige Klammerungen
$(((A_1A_2)A_3)A_4)$
$((A_1(A_2A_3))A_4)$
$(A_1((A_2A_3)A_4))$
$(A_1(A_2(A_3A_4)))$
$((A_1A_2)(A_3A_4))$
Man kann zeigen: Die Anzahl der vollständigen Klammerungen ist exponentiell in n.

Optimierungsproblem Matrizenmultiplikation Π_M:

- Gegeben: $(p_{i-1}\times p_i)$-Matrizen A_i für i=1,…,n
- Gesucht: Vollständige Klammerung mit minimalen Multiplikationskosten

Kosten der Matrizenmultiplikation

- Seien A eine (a,b)-Matrix und B eine (b,c)-Matrix.
- Für die (a,c)-Matrix C=A*B gilt:
  - $C_{i,j} = \sum^{b}_{k=1} A_{i,k} * b_{k,j}$ für i=1,…,a und j=1,…c.
  - Berechnung von C erfordert a*b*c Multiplikationen.

Bsp: A_1 ist (10$\times$ 5), A_2 ist (5 $\times$ 10) und A_3 ist (10 $\times$ 10)

- $(A_1A_2)A_3$ benötigt 500+1000=1500 Multiplikationen
- $A_1(A_2A_3)$ benötigt 500+500=1000 Multiplikationen

Kosten einer optimalen Lösung

- Sei L ein optimale Lösung des Problems Π_M.
- Schreibe $A_{1..n}$ für A_1*...*A_n.
- Für ein k$\in$ [n-1] berechnet L zunächst $A_{\mathbf{1...k}}$ und $A_{k+1...n}$.
  - Deren Produkt liefert $A_{1...n}$

Lemma: L liefert eine optimale Lösung für $A_{1...k}$ und $A_{k+1...n}$.

Annahme: L‘ sei eine bessere Lösung für $A_{1...k}$. (analog für $A_{k+1...n}$)

- Berechne in L das Produkt $A_{1..k}$ gemäß L‘.
- Die Berechnung erfordert weniger Multiplikationen als L. (Widerspruch zur optimalen von L)

Rekursive Lösung

- Seien m[i,j] die minimalen Kosten zur Berechnung von $A_{i..j}$.
- Triviale Subprobleme $A_{i..i}$:
  - m[i,i] = 0 für i=1,…,n
- Berechnung von $A_{i..j}$ für i<j:
  - Bestimme k mit $i \leq k < j$, das folgende Kosten minimiert:
  - Berechne $A_{i..k}$ und $A_{k+1..j}$: Kosten: m[i,k]+m[k+1,j]
  - Berechne $A_{i..k} * A_{k+1..j}$: Kosten: $p_{i-1} * p_k * p_j$

$\Rightarrow m[i,j] = \min_{1 \leq k < j} \{m[i,k] + m[k+1,j] + p_{i-1} * p_k * p_j\}$ für i < j

- Gesamtkosten zur Berechnung von $A_{1..n}$: m[1,n]

Algorithmus Matrixmultiplikation

Algorithmus Matrixmult (Dynamische Programmierung)

Eingabe: $p_0, p_1, \ldots, p_n$

1. for i $\leftarrow$ 1 to n
   1. m[i,i] $\leftarrow$ 0;
2. for l $\leftarrow$ 2 to n /* l ist Länge des Intervalls [i,j] */
   1. for i $\leftarrow$ 1 to n-(l-1)
      1. j $\leftarrow$ i+l-1
      2. m[i,j] $\leftarrow \min_{i \leq k < j} \{m[i,k] + m[k+1,j] + p_{i-1} * p_k * p_j\}$
      3. s[i,j] $\leftarrow$ k, an dem Minimum angenommen wird.

Ausgabe: Arrays m und s

Korrektheit:

- Durch wachsende Intervalllängen werden bei der Minimumsberechnung nur bekannte Werte verwendet.
- m[1,n] liefert minimale Kosten, s die optimale Klammerung (Übungsaufgabe)

Laufzeit:

- Schleifen für l, i und k: $\mathcal{O}(n^3)$.

Das Rucksackproblem

Maximierungsproblem Rucksackproblem Π_R:

Gegeben:

- n Gegenstände mit Gewichten $w_i \in \mathbb{N}$ und Profiten $p_i \in \mathbb{N}$, i=1,…,n.
- Kapazitätsschranke B

Gesucht:

- $\max_{J \subseteq [n]} \{\sum_{j \in J} p_j \mid \sum_{j \in J} w_j \leq B\}$
  bzw. dasjenige $J \subseteq [n]$, für das das Maximum angenommen wird

Struktur einer optimalen Lösung

Sei I eine optimale Lösung von Π_R und i $\in$ [n].

- Sei Π_i := (Π_R ohne Gegenstand i).
- Fall 1: i $\notin$ I
  - I ist optimale Lösung von Π_i mit Schranke B.
- Fall 2: i $\in$ I

Lemma: Sei i $\in$ I. I\{i} ist optimale Lösung von Π_i mit Schranke B-g_i.

Annahme: I‘ sei eine bessere Lösung von Π_i

$\Rightarrow \quad \sum_{j \in I'} p_j > \sum_{j \in I \setminus \{i\}} p_j$ mit $\sum_{j \in I'} w_j \leq B\text{-}g_i$

$\Rightarrow \sum_{j \in I' \cup \{i\}} p_j > \sum_{j \in I} p_j$ mit $\sum_{j \in I' \cup \{i\}} w_j \leq B$

$\Rightarrow$ I $\cup$ {i} ist bessere Lösung für Π als I

(Widerspruch: I ist optimal)

Gewichtsfunktion f(i,t)

Sei $f(i,t) := \min_{J \subseteq [i]} \{\sum_{j \in J} w_i \mid \sum_{j \in J} p_j \geq t\}$, d.h.

- J ist Teilmenge der ersten i Gegenstände mit
  - minimalem Gewicht
  - Gegenstände aus J liefern mindestens Profit t
- Falls kein solches J existiert, definiere $f(i,t) = \infty$
- Falls $f(i,t) \leq B$, ist J eine gültige Lösung.

Sei $P = \sum_{i \in [n]} p_i$ der maximal erreichbare Profit.

- Optimale Lösung: $\max\{t \in [P] \mid f(n,t) \leq B\}$.
- Lösung der trivialen Subprobleme
  - $f(1,t) = w_1$ für $t \in [p_1]$.
  - $f(1,t) = \infty$ für $p_1 < t \leq P$

Rekursive Lösung

Lemma: Für $i \in [n]$ und $t \in [P]$ gilt

$$f(i,t) = \min\{ f(i-1, t), w_i + f(i-1, t-p_i) \} \quad \text{falls } t>p_i$$
$$f(i,t) = \min\{ f(i-1, t), w_i) \} \quad \text{sonst}$$

- f(i,t) nehme für J das minimale Gewicht an.
- Fall 1: $i \notin J$
  - Optimale Lösung für Gegenstände 1,…,i-1 ist optimal für 1,…i, d.h. f(i,t) = f(i-1,t).
- Fall 2: i in J
  - Optimale Lösung für 1,…,i-1 mit Profit $t-p_i$ liefert optimale Lösung für 1,…,i: $f(i,t) = w_i + f(i-1,t-p_i)$ für $t-p_i > 0$ und $f(i,t) = w_i$ sonst.

Algorithmus Rucksack

Algorithmus Rucksack

Eingabe: $w_1,\ldots,w_n$, $p_1,\ldots,p_n$, B

1. $P \leftarrow \sum_{i \in [n]} p_i$
2. for $t \leftarrow 1$ to P
   1. if $(t \leq p_1)$ then $f[1,t] \leftarrow w_1$;
   2. else $f[1,t] \leftarrow \infty$;
3. for $i \leftarrow 2$ to n
   1. for $t \leftarrow 1$ to P
      1. If $(t \leq p_i)$ then $f[i,t] \leftarrow \min\{f[i-1,t], w_i\}$;
      2. else $f[i,t] \leftarrow \min\{f[i-1,t], w_i + f[i-1,t-p_i]\}$;

Ausgabe: max{ t | f[n,t] $\leq$ B}

- Korrektheit: folgt aus Lemma auf voriger Folie.
- Laufzeit: $\mathcal{O}(np) = \mathcal{O}(n \sum_{i \in [n]} p_i)$.
  - Beachte: Eingabelänge der p_i ist $\mathcal{O}(\sum_{i \in [n]} \log p_i)$
  - Algorithmus ist exponentiell in der Eingabelänge.