

ADS: Algorithmen und Datenstrukturen

Zweiter Akt

Uwe Quasthoff

Institut für Informatik
Abteilung Automatische Sprachverarbeitung
Universität Leipzig

17. Oktober 2017

[Letzte Aktualisierung: 16/10/2017, 18:55]

- Die Zeitspanne für die Übungsgruppenanmeldung ist vorbei.
 - Sind Sie einer Übungsgruppe zugewiesen? Wenn nicht warum nicht??? Sollte es technische Probleme geben haben, die noch nicht geklärt sind, melden Sie sich **unmittelbar nach der VO** bei mir.
 - Sind Sie im AlmaWeb angemeldet? Wenn nicht, tun Sie das. Alle Probleme mit dem AlmaWeb klären Sie bitte mit dem Studienbüro, **nicht** mit den Vortragenden oder Übungsleitern.
- Die erste Serie Übungsaufgaben geht heute online. Abgabe nächste Woche **vor** der Vorlesung.
Absolut definitiv KEINE Ausnahmen
Beachten Sie die Regeln zur Abgabe!

Unendliche Zahlenfolge $0, 1, 1, 2, 3, 5, 8, 13, 21, \dots$

Definition $F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}$ für $n > 1$.

Ursprung: Modell einer Kaninchenpopulation

Leonardo Fibonacci (ca. 1170-1240) stieß auf diese Folge bei der einfachen mathematischen Modellierung des Wachstums einer Kaninchenpopulation nach folgender Vorschrift:

- 1 Zu Beginn gibt es ein Paar neugeborener Kaninchen.
- 2 Jedes neugeborene Paar wirft nach 2 Monaten ein weiteres Paar.
- 3 Anschließend wirft jedes Kaninchenpaar jeden Monat ein weiteres.
- 4 Kaninchen leben ewig und haben einen unbegrenzten Lebensraum.

ALSO: Jeden Monat kommt zu der Anzahl der Paare, die im letzten Monat gelebt haben, eine Anzahl von neugeborenen Paaren hinzu, die gleich der Anzahl der Paare ist, die bereits im vorletzten Monat gelebt haben, da genau diese geschlechtsreif sind und sich nun vermehren. Das entspricht der oben angegebenen Rekursionsformel. *Quelle: Wikipedia*

“Versteckte” Zeitkomplexität

Beispiel: Fibonacci-Zahlen 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

Definition $F_0 = 0$, $F_1 = 1$, $F_n = F_{n-1} + F_{n-2}$ für $n > 1$.

```
❶ int fibE (int n) {  
    if (n <= 0) return 0;  
    else if (n==1) return 1;  
    else return fibE(n-2) + fibE(n-1)  
}
```

fibE(n) berechnet F_n , aber **exponentieller Aufwand!**

```
❷ int fibL (int n, int f1, int f2) {  
    if (n <=0) return 0;  
    else if (n==1) return f1+f2;  
    else return fibL(n-1,f2+f1,f1)  
}
```

fibL(n,1,0) berechnet F_n mit **linearem Aufwand**

z.B. fibL(4,1,0)=fibL(3,1,1)=fibL(2,2,1)=fibL(1,3,2)=5

“Versteckte” Platzkomplexität am Beispiel $n!$

```
int fakL(int n) {  
    if(n<=1) return 1;  
    else return n*fakL(n-1);  
}
```

fakL(n) $\rightarrow n!$ mit **linearem Speicheraufwand** $\Theta(n)$

```
int fakK(int n,int a) {  
    if(n<=1) return a;  
    else return fakK(n-1,n*a);  
}
```

fakK($n, 1$) $\rightarrow n!$ mit **konstantem Speicheraufwand** $\Theta(1)$



Bessere Abschätzung: Wie wächst der Aufwand bei der Multiplikation großer Zahlen?



Was ist der Speicheraufwand für n und für $n!$?

Multiplikation für Fortgeschrittene

Wie in der Schule: $5432 \cdot 1995 \implies O(\ell^2)$ für ℓ -stellige Zahlen.

$$\begin{array}{r} 5432 \\ 48888 \\ 48888 \\ 27160 \\ \hline 10836840 \end{array}$$

Besser:

$$(100A + B) \times (100C + D) = 10000AC + 100(AD + BC) + BD$$

Nach $AD + BC = (A + B) \times (C + D) - AC - BD$, nur noch 3 Multiplikationen von Zahlen der halben Länge:

$$T(n) = 3T(n/2),$$

wobei der Aufwand für die Addition vernachlässigt wird.

Lösung: $T(n) = n^{\log_2 3} \approx n^{1.585} \ll n^2$

Für *große* Zahlen geht's also intelligenter als in der Schule!

- Allgemeines Theorem zur Lösung von Funktionalgleichungen (Rekursionsgleichungen) der Form

$$T(n) = aT\left(\frac{n}{b}\right) + g(n), \quad a \geq 1, b > 1$$



Warum können wir i.d.R. $T(1) = 1$ annehmen?

- Funktionalgleichung beschreibt *algorithmische Strategie* "Divide-and-Conquer":
 - Zerlege Gesamtproblem in b gleich große Teilprobleme.
 - Für Gesamtproblem löse a solche Teilprobleme.
 - Für Zerlegung und Kombination der Teillösungen entstehen jeweils Overhead-Kosten $g(n)$.

- Es gibt mehrere Lösungen je nach Verhalten von $g(n)$.
- Sei jetzt $g(n)$ polynomial, d.h. $g(n) = \Theta(n^k)$:

$$T(n) = aT\left(\frac{n}{b}\right) + \Theta(n^k), \quad a \geq 1, b > 1$$

- Dann unterscheide 3 Fälle:

$$T(n) = \begin{cases} \Theta(n^k) & \text{falls } a < b^k \\ \Theta(n^k \log n) & \text{falls } a = b^k \\ \Theta(n^{\log_b(a)}) & \text{falls } a > b^k \end{cases}$$

Das Mastertheorem — Beispiele

Sei wieder $g(n)$ polynomial, $g(n) = \Theta(n^k)$.

Setze $k = 2$ und $b = 3$ und betrachte Beispiele für $a \left\{ \begin{array}{l} < \\ = \\ > \end{array} \right\} n^k$.

$$a = 8 \quad T(n) = 8T\left(\frac{n}{3}\right) + \Theta(n^2) \quad \Rightarrow \quad T(n) = \Theta(n^2)$$

$$a = 9 \quad T(n) = 9T\left(\frac{n}{3}\right) + \Theta(n^2) \quad \Rightarrow \quad T(n) = \Theta(n^2 \log_2 n)$$

$$a = 10 \quad T(n) = 10T\left(\frac{n}{3}\right) + \Theta(n^2) \quad \Rightarrow \quad T(n) = \Theta(n^{\log_3 10})$$

$$T(n) = \begin{cases} \Theta(n^k) & \text{falls } a < b^k \\ \Theta(n^k \log n) & \text{falls } a = b^k \\ \Theta(n^{\log_b(a)}) & \text{falls } a > b^k \end{cases}$$

- Komplexität / Effizienz
= wesentliche Eigenschaft von Algorithmen
- meist asymptotische Worst-Case-Abschätzung in Bezug auf Problemgröße n
 - Unabhängigkeit von konkreten Umgebungsparametern (Hardware, Betriebssystem, ...)
 - asymptotisch "schlechte" Verfahren können bei kleiner Problemgröße ausreichen
- wichtige Klassen: $O(1)$, $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$, \dots , $O(2^n)$
- zu gegebener Problemstellung gibt es oft Algorithmen mit stark unterschiedlicher Komplexität
 - unterschiedliche Lösungsstrategien
 - Raum/Zeit- "Tradeoff": Zwischenspeichern von Ergebnissen statt mehrfacher Berechnung
- Abschätzung der Komplexität aus Programmfragmenten nach Rechenregeln

- Lineare Listen
- Sequentielle Suche
- Binäre Suche
- Weitere Suchverfahren auf sortierten Feldern
 - Sprungsuche
 - Exponentielle Suche
 - Interpolationssuche
- Auswahlproblem

- Endliche Folge von Elementen des selben Typs
 $L = \langle a_0, a_1, \dots, a_{n-1} \rangle$ falls $n (= L.length) > 0$
 $L = \langle \rangle$ leere Liste, $n = 0$.
- Reihenfolge der Elemente ist wesentlich (\leftrightarrow Mengen)

Zwei grundsätzlich unterschiedliche Implementierungen

- Sequenzielle Speicherung (Reihung, Array)
 - statische Datenstruktur (Größenänderung teuer)
 - wahlfreier Zugriff über (berechenbaren) Index
 - in Vorlesung/Übung: Index 0-basiert, d.h. Elemente $L[0], L[1], \dots, L[n-1]$
 - Varianten: 1-dimensionale vs. mehrdimensionale Arrays
- Verkettete Speicherung
 - dynamische Datenstruktur (Einfügen und Löschen billig)
 - kein wahlfreier Zugriff über Index
 - Verkettung (jeweils zum Nachfolger) erlaubt sequenzielles Durchlaufen
 - Varianten: einfache vs. doppelte Verkettung etc.

Sequenzielle Suche

Sequenzielle Suche nach Element mit Schlüsselwert x :

Durchlaufe Liste der Reihe nach und vergleiche jeden Schlüssel mit x

Kosten:

- erfolglose Suche erfordert n Schlüsselvergleiche
- erfolgreiche Suche verlangt im ungünstigsten Fall n Schlüsselvergleiche
- mittlere Anzahl von Schlüsselvergleichen bei erfolgreicher Suche

$$C_{avg}(n) = \frac{1}{n} \cdot \sum_{i=1}^n i = \frac{n+1}{2}$$

Sequenzielle Suche funktioniert *“immer”*.



Lässt sich schneller sequenziell suchen, wenn die Liste sortiert ist?

- In sortierten Listen geht Suche deutlich schneller als sequenziell:
- **Binäre Suche** (*Suche durch rekursive binäre Zerlegung*)

Suche x in Liste L :

```
if (L empty)
  report "nicht gefunden";
else
  mid = floor(L.length/2);
  if (x==L[mid]) report "gefunden";
  if (x<L[mid]) Suche x in (Sub-)Liste L[0..mid-1];
  if (x>L[mid]) Suche x in (Sub-)Liste L[mid+1..n-1];
```

- Kosten für erfolgreiche Binärsuche

$$C_{\min}(n) = 1, \quad C_{\max}(n) = \lfloor \log_2(n) \rfloor + 1, \quad C_{\text{avg}}(n) \approx \log_2(n) \quad n \rightarrow \infty$$

- **Verbesserung von $O(n)$ auf $O(\log n)$**

Funktionen `floor(.)`, `[.]` \equiv abrunden, `ceiling(.)`, `[.]` \equiv aufrunden

HILFE: Die Funktionen “floor” und “ceiling”

$\lfloor x \rfloor$, auch $\text{floor}(x)$ genannt, beschreibt die größte ganze Zahl nicht größer als x , also den “ganzzahligen Anteil” von x .

Beispiele: $\lfloor \pi \rfloor = 3$. $\lfloor -\pi \rfloor = -4$.

Eigenschaften:

- 1 $\lfloor x \rfloor \leq x < \lfloor x \rfloor + 1$
- 2 $k \in \mathbb{N}$ impliziert $\lfloor k + x \rfloor = k + \lfloor x \rfloor$

$\lceil x \rceil$, $\text{ceiling}(x)$, ist die kleinste ganze Zahl nicht kleiner als x .

Beispiele: $\lceil \pi \rceil = 4$. $\lceil -\pi \rceil = -3$

Eigenschaften:

- 1 $x \leq \lceil x \rceil < x + 1$
- 2 $k \in \mathbb{N}$ impliziert $\lceil k \rceil = \lfloor k \rfloor = k$
- 3 $k \in \mathbb{N}$ impliziert $\lceil k/2 \rceil + \lfloor k/2 \rfloor = k$

Vorbedingung (immer noch): Liste L ist (aufsteigend) sortiert

Prinzip: L wird in Abschnitte fester Länge m unterteilt. Springe über die Abschnitte, um den Abschnitt des Schlüssels zu bestimmen.

Abschnitte: $0 \dots (m-1)$, $m \dots (2m-1)$, $2m \dots (3m-1)$, usw.

Einfache Sprungsuche:

- Springe zu Positionen $i \cdot m$ (für $i = 1, 2, \dots$; der Reihe nach)
- Sobald $x < L[im]$, kann x nur in i -tem Abschnitt $(i-1)m \dots im-1$ liegen; dieser wird sequenziell nach x durchsucht

Mittlere Suchkosten: ein Sprung koste a , ein Vergleich b :

$$C_{avg}(n) = \frac{1}{2} \frac{n}{m} a + \frac{1}{2} mb$$

Minimiere $C_{avg}(n)$ über m (\gg Ableitung nach $m \ll \stackrel{!}{=} 0$; Übung)

\Rightarrow optimale Sprungweite $m = \lfloor \sqrt{(a/b)n} \rfloor$; dann ist Komplexität in $O(\sqrt{n})$



Worst case Komplexität?

Zwei-Ebenen-Sprungsuche

- Prinzip: Statt sequenzieller Suche im lokalisierten Abschnitt wird wiederum eine Quadratwurzel-Sprungsuche angewendet, bevor dann sequenziell gesucht wird
- Mittlere Suchkosten: $C_{avg}(n) \leq \frac{1}{2}a\sqrt{n} + \frac{1}{2}b\sqrt{\sqrt{n}} + \frac{1}{2}c\sqrt{\sqrt{n}}$
 a Kosten eines Sprungs auf der ersten Ebene
 b Kosten eines Sprungs auf der zweiten Ebene
 c Kosten eines sequenziellen Vergleichs
- Verbesserung durch optimale Abstimmung der Sprungweiten m_1 und m_2 der beiden Ebenen
- Mit $a = b = c$ ergeben sich als optimale Sprungweiten $m_1 = n^{2/3}$ und $m_2 = n^{1/3}$ und mittlere Suchkosten $C_{avg}(n) = \frac{3}{2}an^{1/3}$
- Verallgemeinerung zu n -Ebenen-Verfahren ergibt ähnlich günstige Kosten wie Binärsuche



Wann ist Sprungsuche vorteilhaft? Allgemein: wenn Binärsuche nicht sinnvoll ist; z.B. bei blockweisem Einlesen der sortierten Sätze vom Externspeicher oder ...



Was tun, falls die Länge eines sortierten Suchbereichs zunächst unbekannt ist?

• Vorgehensweise

- 1 für Suchschlüssel x wird zunächst obere Grenze für den zu durchsuchenden Abschnitt bestimmt:

```
int i = 1; while (x > L[i]) i=2*i;
```

- 2 danach gilt $L[i/2] < x \leq L[i]$; d.h. Suchabschnitt: $i/2 + 1 \dots i$.
- 3 suche innerhalb des Abschnitts mit irgendeinem Verfahren

• Analyse

Annahme: Liste enthält nur Schlüssel aus \mathbb{N} ohne Duplikate

→ Schlüsselwerte wachsen mindestens so stark wie die Elementindizes i

→ höchstens $\log_2 x$ Verdopplungen, denn die Zahl der Sprünge ist $\log_2 i$

Bestimmung des gesuchten Intervalls: $\leq \log_2 x$ Schlüsselvergleiche

Suche innerhalb des Abschnitts (z.B. Binärsuche): $\leq \log_2 x$ Vergleiche

• Gesamtaufwand $O(\log_2 x)$

Idee: Im Prinzip wie Binärsuche, aber noch schnellere Einschränkung des Suchbereichs, indem wir versuchen den Index des Suchschlüssels x aus den Schlüsselwerten zu erraten.

Vorgehensweise: Berechne nächste Suchposition aus den Grenzen u und v des aktuellen Suchbereichs $[u, v]$:

$$p = u + \frac{x - L[u]}{L[v] - L[u]}(v - u)$$

Dann zerlege den Suchbereichs bei $\lfloor p \rfloor$ und fahre rekursiv fort. (Also: Interpolationssuche ist wie in Binärsuche, aber anstelle der mittigen Teilung wird versucht, an der *richtigen Stelle* zu teilen.)

Analyse: Sinnvoll, falls Schlüsselwerte im betreffenden Bereich einigermaßen gleichverteilt. Dann im Mittel nur

$$C_{avg}(n) = 1 + \log_2 \log_2 n \quad \text{Vergleiche.}$$

Im Extremfall “besonders irreguläre Schlüsselverteilung” aber $O(n)$.

Häufigkeitsgeordnete lineare Listen

- **Prinzip:** Ordne die Liste nach Zugriffshäufigkeiten und suche sequenziell. Sinnvoll, falls die Zugriffshäufigkeiten für die einzelnen Elemente einer Liste sehr unterschiedlich sind
- **Analyse:** (Sei p_i die Zugriffswahrscheinlichkeit für Element i)
Mittlere Suchkosten bei (erfolgreicher) sequenzieller Suche

$$C_{avg}(n) = 1p_1 + 2p_2 + 3p_3 + \dots + np_n = \sum_{i=1}^n ip_i$$

um C_{avg} zu minimieren: organisiere Liste, so daß $p_1 \geq p_2 \geq \dots \geq p_n$.

- **Beispiel** *Zugriffsverteilung nach 80-20-Regel*
80% der Suchanfragen betreffen 20% des Datenbestandes und von diesen 80% wiederum 80% (also insgesamt 64%) der Suchanfragen richten sich an 20% von 20% (d.h. insgesamt 4%) der Daten.

 **erwarteter Suchaufwand** $C_{avg}(n) = ?$

- **Problem in der Praxis:** Zugriffshäufigkeiten meist unbekannt
→ selbstorganisierende (adaptive) Listen

- **Anwendung:** Nützlich, falls häufig gleiche Objekte gesucht werden (Beispiel: Suchbegriffe bei Suchmaschinen)
- **Prinzip:**
 - ① gesucht wird immer sequenziell von vorn
 - ② die Reihenfolge der Objekte in der Liste passt sich dem Zugriffsmuster an (dazu gibt es verschiedene Regeln)
- **Ansatz 1: FC-Regel (Frequency count)**
 - führe einen Häufigkeitszähler pro Element ein
 - jeder Zugriff auf ein Element erhöht dessen Häufigkeitszähler
 - falls erforderlich, wird danach die Liste lokal neu geordnet, so dass die Häufigkeitszähler der Elemente eine absteigende Reihenfolge bilden
 - hoher Wartungsaufwand und zusätzlicher Speicherplatzbedarf

- **Ansatz 2: T-Regel (Transpose)**
 - das Zielelement eines Suchvorgangs wird jeweils mit dem unmittelbar vorangehenden Element vertauscht
 - häufig referenzierte Elemente wandern (langsam) an den Listenanfang
- **Ansatz 3: MF-Regel (Move-to-Front)**
 - das Zielelement eines Suchvorgangs wird nach jedem Zugriff an die erste Position der Liste gesetzt
 - relative Reihenfolge der übrigen Elemente bleibt gleich
 - in jüngster Vergangenheit referenzierte Elemente sind am Anfang der Liste (Zeitliche Lokalität im Zugriffsmuster kann gut genutzt werden)



Welche grundsätzliche Implementierung der Liste wählen sie je nach Ansatz?

Aufgabe: Finde das i -kleinste Element einer Liste L

— Spezialfälle: kleinstes (Minimum), zweitkleinstes, ... Element, mittlerer Wert (Median), größtes (Maximum)

Bei sortierter Liste: trivial

Bei unsortierter Liste: Minimum/Maximum-Bestimmung hat lineare Kosten $O(n)$

Algorithmus: Einfache Lösung für i -kleinstes Element

Wiederhole $i-1$ mal:

Bestimme kleinstes Element in L und entferne es aus L
gib Minimum der Restliste als Ergebnis zurück

Komplexität $O(in)$, bzw. für $i \in O(n)$ (z.B. "Median"): $O(n^2)$

JEDOCH: es geht besser

Allgemeines Verfahren, um große Probleme schneller zu lösen:

- 1 Ist das Problem zu groß, um sich “trivial” effizient lösen zu lassen, zerlege das Problem in zwei oder mehrere Teilprobleme. (Divide)
- 2 Löse die Teilprobleme einzeln nach der gleichen Methode (d.h. rekursiv). (Conquer)
- 3 Konstruiere die Gesamtlösung aus den Teillösungen. (Merge)

Das Verfahren wird in Schritt 2 rekursiv angewendet, bis hinreichend triviale (kleine) Probleme vorliegen, die sich mit anderen Mitteln lösen lassen.

Frage: Wieso lässt sich damit die Komplexität verringern?

Sehen wir uns zunächst Beispiele an

Suche in sortierter Liste als Beispiel für D&C

Sequenzielle Suche → Binärsuche

Teilungsschritt: Die Liste wird in der Mitte geteilt

Triviale Liste: Liste der Länge 1; einfacher Vergleich

Gesamtlösung: Erfolgreich, falls in einem Teil gefunden

Unser Algorithmus zur Binärsuche lässt sich umformen zu:

Falls die Liste die Länge 1 hat: Prüfe, ob es sich um das gesuchte Element handelt und gib JA oder NEIN zurück.

Sonst zerlege die Liste in der Mitte und prüfe durch Vergleich des gesuchten Elements mit der Trennstelle, in welcher Hälfte das gesuchte Element liegen muss.

Wende diesen Algorithmus rekursiv auf diese Teilliste an.

Zur Erinnerung:

Verbesserung der Zeitkomplexität von $O(n)$ zu $O(\log n)$

 **Wie folgt diese Komplexität aus dem Mastertheorem?**

2. D&C-Beispiel: Maximale Teilsumme

Idee: Wie zuvor zerlege die gegebene Folge in zwei Teile; bestimme die maximalen Teilsummen für beide Teile.

i	0	1	2	3	4	5	6	7	8	9	10
$F[i]$	+2	+5	-8	+4	+2	+5	+7	-2	-7	+3	+5

ABER: Die maximale Teilfolge könnte gerade die Schnittstelle einschließen

DESHALB: linkes und rechtes Randmaximum einer Folge F

- linkes Randmaximum von F : maximale Summe aller Teilfolgen, die bis zum linken Rand (Anfang) von F reichen
- analog: rechtes Randmaximum

Randmaxima lassen sich in $O(n)$ bestimmen!

Rekursiver (D&C-)Algorithmus für maximale Teilsumme

- falls Eingabefolge F nur aus einer Zahl z besteht, nimm $\max(z, 0)$
- falls F wenigstens 2 Elemente umfasst
 - Zerlege F in etwa gleich große Hälften
 - bestimme *rekursiv* die maximalen Teilsummen m_l und m_r der linken und der rechten Hälfte
 - bestimme (in linearer Zeit)
 - das rechte Randmaximum s_l der linken Hälfte und
 - das linke Randmaximum s_r der rechten Hälfte
 - Maximale Teilsumme von F ist $\max(m_l, s_l + s_r, m_r)$



Komplexitätsverbesserung?

Zurück zum Auswahlproblem

Suche i -kleinstes Element von n unterschiedlichen Elementen

- 1 bestimme Pivot-Element p (z.B. letztes Element)

$$L = [-----]p$$

- 2 Teile die n Elemente bezüglich p in 2 Teillisten L_1 und L_2 . L_1 enthält die i Elemente $< p$; L_2 die restlichen Elemente $> p$.

$$L' = [.....L_1.....]p[.....L_2.....]$$

- 3 Falls $L_1.length = i - 1$, dann ist p das i -kleinste Element
Falls $L_1.length > i - 1$, wende das Verfahren rekursiv auf L_1 an
Falls $L_1.length < i - 1$, wende Verfahren rekursiv auf L_2 an; aber bestimme das $(i - 1 - L_1.length)$ -kleinste Element.
- 4 **Komplexität:** Nehmen wir an, dass die Liste jeweils halbiert werden kann, dann ergibt sich aus $T(n) = T(n/2) + \Theta(n)$ nach Mastertheorem: $O(n)$



Ist die Annahme für Komplexitätsabschätzung realistisch?

Weitere Beispiele folgen beim Thema Sortieren, z.B. Quicksort.



In welchem Sinne war unsere schnelle Multiplikation eine Anwendung von Teile-und-Herrsche?

Annahme: wir wollen ein Problem der Größe $n = 2^r$ lösen und zerlegen jeweils in zwei gleich große Teile.

Beobachtungen:

- 1 die maximale Rekursionstiefe ist $r = \log_2 n$
- 2 die Anzahl der Teilprobleme in Rekursionstiefe k ist 2^k
- 3 muss in jeder Tiefe nur ein Teilproblem gelöst werden (wie beim Suchen), so sind es insgesamt $r = \log_2 n$ Teilprobleme
- 4 so entsteht der Faktor $\log n$ bei der Komplexität
- 5 muss jedes Teilproblem gelöst werden, so sind n elementare Probleme zu lösen und diese Lösungen zu verknüpfen. Das macht speziell Sinn für Probleme der Komplexität $O(n^2)$ oder schlechter (z.B. maximale Teilsumme)



Erklären Sie damit, dass Teile-und-Herrsche keine Verbesserung bei sequenzieller Suche in einer ungeordneten Liste bringt.