Algorithmen und Datenstrukturen

Kapitel 6: Weiterführende Entwurfs- & Analysetechniken

Prof. Dr. Peter Kling

Wintersemester 2020/21

Übersicht

1) Überblick

Generische Entwurfs- & Analyseansätze

- haben viele algorithmische Probleme gesehen...
  - kleinstes Element, Sortieren, DB-Suche, Graph-Traversierung, Graph-Strukturen, ...
- ...und viele algorithmische Ansätze...
  - inkrementell, Divide & Conquer, Randomisierung, systematisch Traversierung, gierig, Tabellen, Reduktion, ...
- ...und viele Analyseansätze
  - Invarianten, Induktion, Potentialfunktionen, Master Theorem(e), amortisierte Analyse, ...

Können wir da etwas Ordnung rein bringen? 🤔

Fragen

- Welche Ansätze gibt es? Was unterscheidet sie?
- Was ist jeweils die grundsätzliche Herangehensweise?
- Wann ist welches Verfahren sinnvoll?

Typische algorithmische Ansätze

- Systematische Suche ≈ „Schau dir alles an…“
- Divide & Conquer ≈ „Zerlege Probleme in Teilprobleme!“
- **Dynamische Programmierung** ≈ „Tu nie etwas zweimal!“
- **Gierige Verfahren** ≈ „Schau niemals zurück!“
- (Lokale Suche) ≈ „Denke global, handle lokal!“

Typische algorithmische Ansätze

- Systematische Suche ≈ „Schau dir alles an…“
- Divide & Conquer ≈ „Zerlege Probleme in Teilprobleme!“
- **Dynamische Programmierung** ≈ „Tu nie etwas zweimal!“
- **Gierige Verfahren** ≈ „Schau niemals zurück!“
- (Lokale Suche) ≈ „Denke global, handle lokal!“

- zur lokalen Suche siehe z.B. Wikipedia; sie fällt hier etwas heraus, da es sich (typischerweise) um heuristische Algorithmen handelt, für die man also keine oder nur schwache Optimalitätsgarantien geben kann; sie ist aber von immenser Bedeutung in der Praxis (z. B. im maschinellen Lernen) und Gegenstand aktiver Forschung

Systematische Suche

Prinzip

Durchsuche den gesamten Lösungsraum.
Auch bekannt als „Brute-Force“ Ansatz.

Vorteile

- sehr einfach

Nachteile

- sehr Zeitaufwendig
- schlecht für große Instanzen

Mögliche Anwendungen:

- Suche in unsortierter Liste
- Suche per Broadcasting in unstrukturierten, verteilten Systemen

  Stichwort: Peer-to-Peer Systeme
- Rucksackproblem

Algorithmisches Problem

- Eingabe:
  - n Objekte und Rucksack mit Kapazität W
  - Objekt i hat Gewicht (weight) w_i und Wert (value) v_i
- Ausgabe: Menge M von Objekten die zusammen in den Rucksack passen und maximalen Gesamtwert haben.

Lösung per systematische Suche

- iteriere über alle Teilmengen M von Objekten
- merke die Menge M mit bisher maximalem Wert $\sum_{i \in M} v_i$...
- ...für die $\sum_{i \in M} w_i \leq W$ gilt

$\rightsquigarrow$ Laufzeit $O(2^n)$

Divide & Conquer

Prinzip

- Problem in Teilprobleme aufteilen
- Teilprobleme rekursiv lösen
- Lösung aus Teillösungen zusammensetzen

Vorteile

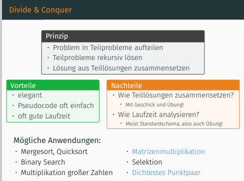
- elegant
- Pseudocode oft einfach
- oft gute Laufzeit

Nachteile

- Wie Teillösungen zusammensetzen?
  - Mit Geschick und Übung!
- Wie Laufzeit analysieren?
  - Meist Standardschema, also auch Übung!

Mögliche Anwendungen:

- Mergesort, Quicksort
- Binary Search
- Multiplikation großer Zahlen
- Matrizenmultiplikation
- Selektion
- Dichtestes Punktpaar



- Matrizenmultiplikation: Cormen (3rd) Ch. 4.2
- Selektion: Cormen (3rd) Ch. 9

Dynamische Programmierung

Prinzip

- Problem in Teilprobleme aufteilen
- Teilprobleme rekursiv lösen
- Lösung aus Teillösungen zusammensetzen

Äh, hatten wir das nicht gerade?

\+ Merken der Lösungen von häufigen Teilproblemen!

Prinzip (Abgrenzung gegenüber D&D)

- formuliere Problem rekursiv
- vermeide mehrfache Berechnung von Teilergebnissen
- Verwende „bottom-up“ Implementierung

Dynamische Programmierung

Prinzip (Abgrenzung gegenüber D&D)

- formuliere Problem rekursiv
- vermeide mehrfache Berechnung von Teilergebnissen
- verwende „bottom-up“ Implementierung

Vorteile

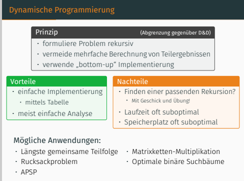
- einfache Implementierung
  - mittels Tabelle
- meist einfache Analyse

Nachteile

- Finden einer passenden Rekursion?
  - Mit Geschick und Übung!
- Laufzeit oft suboptimal
- Speicherplatz oft suboptimal

Mögliche Anwendungen:

- Längste gemeinsame Teilfolge
- Rucksackproblem
- APSP
- Matrixketten-Multiplikation
- Optimale binäre Suchbäume



- Längste gemeinsame Teilfolge: Cormen (3rd) Ch. 15.4
- Rucksackproblem: siehe Unterkapitel zu Dynamischer Programmierung
- Matrixketten-Multiplikation: Cormen (3rd) Ch. 15.2
- Optimale binäre Suchbäume: Cormen (3rd) Ch. 15.5

Gierige Verfahren

Prinzip

- typischerweise für Optimierungsprobleme
  - finde zulässige Lösung...
  - ...die bzgl. einer bestimmten Zielfunktion optimal ist
- baue Lösungen iterativ in „kleinen“ Schritten:
  - treffe in jedem Schritt gierige (aktuell beste)...
  - ...und irreversible Entscheidung

min & max

Vorteile

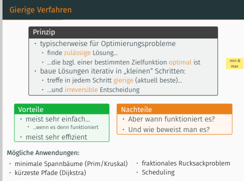
- meist sehr einfach...
  - ...wenn es denn funktioniert
- meist sehr effizient

Nachteile

- Aber wann funktioniert es?
- Und wie beweist man es?

Mögliche Anwendungen:

- minimale Spannbäume (Prim/Kruskal)
- kürzeste Pfade (Dijkstra)
- fraktionales Rucksackproblem
- Scheduling



- man kann im übrigen durchaus argumentieren, dass Dijkstra ein Algorithmus nach dem Prinzip der dynamischen Programmierung ist
- gierige Algorithmen sind meist ein guter Startpunkt, wenn man versucht ein Optimierungsproblem zu lösen
- es gibt eine starke und sehr allgemeine Theorie zu gierigen Algorithmen; Stichwort Matroide (siehe auch Cormen (3rd) Ch. 16.4); Matroide sind nicht Teil dieses Kurses!

2) Dynamische Programmierung

Prinzip (Abgrenzung gegenüber D&D)

- formuliere Problem rekursiv
- vermeide mehrfache Berechnung von Teilergebnissen
- verwende „bottom-up“ Implementierung

Typisches Anwendungsgebiet

- Optimierungsprobleme
- optimale Lösung zusammensetzbar aus optimalen Teillösungen
- jedes Teilproblem kann ggfs. mehrfach vorkommen

Definition 6.1

Seien $X = (x_1, x_2, \ldots, x_m)$ und $Y = (y_1, y_2, \ldots, y_n)$ zwei Folgen über einem endlichen Alphabet Σ. Dann heißt Y Teilfolge von X, wenn es Indizes $i_1 < i_2 < \cdots < i_n$ gibt mit $x_{i_j} = y_j$ für alle $j \in \{1, 2, \ldots n\}$

Y	B	C	A	C			
X	A	B	A	C	A	B	C

- Y ist Teilfolge von X...
- ...mit $(i_1, i_2, i_3, i_4) = (2, 4, 5, 7)$

Algorithmisches Problem: Längste gemeinsame Teilfolge

- Eingabe: Folgen $X = (x_1, x_2, \ldots, x_m)$ und $Y = (y_1, y_2, \ldots, y_n)$
- Ausgabe: längste Folge Z die Teilfolge von X und Y ist

Beispiel

X: A B A C A B C

Y: B A C C A B B C

Z_1: B C A C

Z_2: B A C A C

Triviale Lösung: Brute-Force

- überprüfe alle $2^{\min\{m,n\}}$ Teilfolgen der kleineren Folge

$\rightsquigarrow$ exponentielle Laufzeit

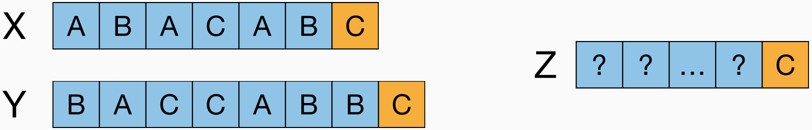
Theorem 6.1

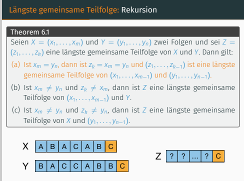
Seien $X = (x_1, \ldots, x_m)$ und $Y = (y_1, \ldots, y_n)$ zwei Folgen und sei $Z = (z_1, \ldots, z_k)$ eine längste gemeinsame Teilfolge von X und Y. Dann gilt:

(a) Ist $x_m = y_n$, dann ist $z_k = x_m = y_n$ und $(z_1, \ldots, z_{k-1})$ ist eine längste gemeinsame Teilfolge von $(x_1, \ldots, x_{m-1})$ und $(y_1, \ldots, y_{n-1})$.

(b) Ist $x_m \neq y_n$ und $z_k \neq x_m$, dann ist Z eine längste gemeinsame Teilfolge von $(x_1, \ldots, x_{m-1})$ und Y.

(c) Ist $x_m \neq y_n$ und $z_k \neq y_n$, dann ist Z eine längste gemeinsame Teilfolge von X und $(y_1, \ldots, y_{n-1})$.





- siehe Cormen (3rd) Theorem 15.1 für Ausformulierung dieser Beweisskizze

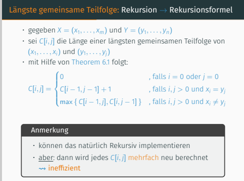
Längste gemeinsame Teilfolge: Rekursion → Rekursionsformel

- gegeben $X = (x_1, \ldots, x_m)$ und $Y = (y_1, \ldots, y_n)$
- sei $C[i, j]$ die Länge einer längsten gemeinsamen Teilfolge von $(x_1, \ldots, x_i)$ und $(y_1, \ldots, y_j)$
- mit Hilfe von Theorem 6.1 folgt:

$$C[i,j] = \begin{cases} 0 & \text{, falls } i = 0 \text{ oder } j = 0 \\ C[i-1, j-1] + 1 & \text{, falls } i, j > 0 \text{ und } x_i = y_j \\ \max\{\, C[i-1, j], C[i, j-1] \,\} & \text{, falls } i, j > 0 \text{ und } x_i \neq y_j \end{cases}$$

Anmerkung

- können das natürlich Rekursiv implementieren
- aber: dann wird jedes $C[i, j]$ mehrfach neu berechnet
  $\rightsquigarrow$ ineffizient



- Insbesondere wird $C[i,j]$ in verschiedenen Zweigen des Rekursionsbaums ständig neu berechnet!

Längste gemeinsame Teilfolge: Pseudocode

Erinnerung

$$C[i,j] = \begin{cases} 0 & \text{, falls } i = 0 \text{ oder } j = 0 \\ C[i-1, j-1] + 1 & \text{, falls } i, j > 0 \text{ und } x_i = y_j \\ \max\{\, C[i-1, j], C[i, j-1] \,\} & \text{, falls } i, j > 0 \text{ und } x_i \neq y_j \end{cases}$$

Algorithmus 6.1: LCSubSeq(X, Y)

```
m ← length(X)
n ← length(Y)
for i ← 0 to m: C[i, 0] ← 0
for j ← 0 to n: C[0, j] ← 0
for i ← 1 to m
    for j ← 1 to n
        „berechne C[i, j] nach obiger Formel“
return C
```

Längste gemeinsame Teilfolge: Pseudocode

Erinnerung

$$C[i,j] = \begin{cases} 0 & \text{, falls } i = 0 \text{ oder } j = 0 \\ C[i-1,j-1]+1 & \text{, falls } i,j > 0 \text{ und } x_i = y_j \\ \max\{\, C[i-1,j], C[i,j-1] \,\} & \text{, falls } i,j > 0 \text{ und } x_i \neq y_j \end{cases}$$

Algorithmus 6.1: LCSubSeq(X, Y)

```
m ← length(X)
n ← length(Y)
for i ← 0 to m: C[i, 0] ← 0
for j ← 0 to n: C[0, j] ← 0
for i ← 1 to m
    for j ← 1 to n
        „berechne C[i, j] nach obiger Formel“
return C
```

- LCSubSeq steht für „longest common subsequence“

Längste gemeinsame Teilfolge: Beispiel & Extraktion

- ausfüllen der Tabelle mittels Rekursion
- Extraktion längster Teilfolge: speichere zusätzlich „Richtung“

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Wie gut ist dieser DP-Ansatz?

Theorem 6.2

Für zwei Folgen X und Y der Längen n und m berechnet Algorithmus LCSubSeq die Länge einer längsten gemeinsamen Teilfolge in Laufzeit $\Theta(n \cdot m)$.

Theorem 6.3

Gegeben die Ausgabe-Tabelle von Algorithmus LCSubSeq, kann eine längste gemeinsame Teilfolge in Laufzeit $\Theta(n + m)$ bestimmt werden.

Algorithmisches Problem

Erinne-rung

- Eingabe:
  - n Objekte und Rucksack mit Kapazität W
  - Objekt i hat Gewicht (weight) w_i und Wert (value) v_i
- Ausgabe: Menge M von Objekten die zusammen in den Rucksack passen und maximalen Gesamtwert haben.

Beispiel für Rucksackgröße $W = 6$

Größe	5	2	1	3	7	4
Wert	11	5	2	8	14	9

- Objekte 1 und 3 passen und haben Gesamtwert 13 $\rightsquigarrow$ Optimal?
- Objekte 2, 3 und 4 passen und haben Gesamtwert 15!

Rucksackproblem: Herleitung einer Rekursion

- sei O eine optimale zulässige Teilmenge der n Objekte
  - also $\sum_{i \in O} w_i \leq W$...
  - ...und $\sum_{i \in O} v_i$ maximal
- sei $\mathrm{OPT}(i, w)$ der Wert einer optimalen Lösung...
  - ...wenn wir nur Objekte 1 bis i packen dürfen und...
  - ...maximal Gewicht w erlaubt ist
- beachte: $\mathrm{OPT}(n, W) = \sum_{i \in O} v_i$

Unterscheiden zwei Fälle

Fall 1: $n \notin O$

- dann gilt $\mathrm{OPT}(n, W) = \mathrm{OPT}(n-1, W)$

Fall 2: $n \in O$

- dann gilt $\mathrm{OPT}(n, W) = v_n + \mathrm{OPT}(n-1, W - w_n)$

Rucksackproblem: Die Rekursion als Formel

$$\mathrm{OPT}(i, w) = \begin{cases} 0 & \text{, falls } i = 0 \text{ oder } w = 0 \\ \mathrm{OPT}(i-1, w) & \text{, falls } i, w > 0 \text{ und } w_i > w \\ \max\{\,\mathrm{OPT}(i-1, w), v_i + \mathrm{OPT}(i-1, w-w_i)\,\} & \text{, falls } i, w > 0 \text{ und } w_i \le w \end{cases}$$

Algorithmus 6.2: KnapsackDP(n, W)

---

for $i \leftarrow 0$ **to** n: $A[i, 0] \leftarrow 0$
for $w \leftarrow 0$ **to** W: $A[0, w] \leftarrow 0$
for $i \leftarrow 1$ **to** n
    for $w \leftarrow 1$ **to** W
        „berechne $A[i, w]$ nach obiger Formel (A statt OPT)“
return $A[n, W]$

Rucksackproblem: Die Rekursion als Formel

$$\mathrm{OPT}(i,w) = \begin{cases} 0 & \text{, falls } i = 0 \text{ oder } w = 0 \\ \mathrm{OPT}(i-1,w) & \text{, falls } i,w > 0 \text{ und } w_i > w \\ \max\{\,\mathrm{OPT}(i-1,w), v_i + \mathrm{OPT}(i-1,w-w_i)\,\} & \text{, falls } i,w > 0 \text{ und } w_i \le w \end{cases}$$

Algorithmus 6.2: KnapsackDP(n, W)

```
for i ← 0 to n: A[i,0] ← 0
for w ← 0 to W: A[0,w] ← 0
for i ← 1 to n
    for w ← 1 to W
        „berechne A[i,w] nach obiger Formel (A statt OPT)“
return A[n,W]
```

- man könnte hier „schummeln“ und $\mathrm{OPT}(i,w)$ für $w < 0$ als $-\infty$ definieren
- dann kann man sich den mittleren Sonderfall sparen
- im Englischen heißt das Rucksackproblem „Knapsack Problem“

Rucksackproblem: Beispiel

i	w_i	v_i
n	3	3
$n-1$	4	7
$\vdots$	7	3
	1	2
	2	3
$\vdots$	1	1
2	3	4
1	5	2

0	1	...					...	W
0	2	3	5	7	9	10	12	13
0	2	3	5	7	9	10	12	13
0	2	3	5	6	7	9	10	10
0	2	3	5	6	7	9	10	10
0	1	3	4	5	7	8	8	8
0	1	1	4	5	5	5	5	6
0	0	0	4	4	4	4	4	6
0	0	0	0	0	2	2	2	2
0	0	0	0	0	0	0	0	0

Rucksackproblem: Beispiel

i	w_i	v_i
n	3	3
$n-1$	4	7
⋮	7	3
	1	2
	2	3
⋮	1	1
2	3	4
1	5	2

0	1	...					...	W
0	2	3	5	7	9	10	12	13
0	2	3	5	7	9	10	12	13
0	2	3	5	6	7	9	10	10
0	2	3	5	6	7	9	10	10
0	1	3	4	5	7	8	8	8
0	1	1	4	5	5	5	5	6
0	0	0	4	4	4	4	4	6
0	0	0	0	0	2	2	2	2
0	0	0	0	0	0	0	0	0

- um die einzupackenden Objekte zu bekommen, würde man sich auch hier entsprechende „Zeiger“ speichern, ähnlich wie wir es bei den längsten gemeinsamen Teilfolgen gesehen haben

Wie gut ist dieser DP-Ansatz?

Theorem 6.4

Für n Objekte und einen Rucksack der Größe W berechnet Algorithmus KnapsackDP den Wert einer optimalen Lösung des Rucksackproblems in Laufzeit $\Theta(n \cdot W)$.

Theorem 6.5

Gegeben die Ausgabe-Tabelle von Algorithmus KnapsackDP, kann eine Objektemenge mit optimalem Wert die in den Rucksack passt in Laufzeit $\Theta(n + m)$ bestimmt werden.

3) Gierige Algorithmen

Prinzip

- typischerweise für Optimierungsprobleme
  - finde zulässige Lösung...
  - ...die bzgl. einer bestimmten Zielfunktion optimal ist
- baue Lösungen iterativ in „kleinen" Schritten:
  - treffe in jedem Schritt gierige (aktuell beste)...
  - ...und irreversible Entscheidung

Am Beispiel minimaler Spannbäume:

- Probleminstanz: gewichteter, ungerichteter zusammenhängender Graph
- Zulässige Lösungen: Spannbäume
- Zielfunktion: Gewicht eines Spannbaums
- Gesucht: Spannbaum mit minimalem Gewicht

Gierige Algorithmen: Idee & Prim

1) bestimme Lösung durch sukzessives Erweitern bereits gefundener Teillösungen
2) Erweiterung durch lokal optimale Wahl
3) Analyse muss zeigen, dass lokal optimale Wahlen zu global optimaler Lösung führt

1) Algorithmus von Prim bestimmt minimalen Spannbaum durch sukzessives Hinzufügen von Kanten
2) Prim wählt leichteste Kante, die isolierten Knoten mit Teilbaum verbindet
3) Analyse mit Hilfe von Schnitten zeigt, dass durch lokal optimale Wahl erzeugter Teilbaum immer in minimalem Spannbaum enthalten ist

Gierige Algorithmen: Idee & Kruskal

1) bestimme Lösung durch sukzessives Erweitern bereits gefundener Teillösungen
2) Erweiterung durch lokal optimale Wahl
3) Analyse muss zeigen, dass lokal optimale Wahlen zu global optimaler Lösung führt

1) Algorithmus von Kruskal bestimmt minimalen Spannbaum durch sukzessives Hinzufügen von Kanten
2) Kruskal wählt leichteste Kante, die Zusammenhangskomponenten verbindet
3) Analyse mit Hilfe von Schnitten zeigt, dass durch lokal optimale Wahl erzeugter Teilbaum immer in minimalem Spannbaum enthalten ist

Beispielproblem 1: Gieriges 1-Prozessor Scheduling

Gegeben

- ein Prozessor und n Jobs
- keine Parallelität: Prozessor bearbeitet zu jedem Zeitpunkt ≤ 1 Jobs
- angefangene Jobs können nicht abgebrochen werden
- j-ter Job hat Größe p_j

Gesucht

Abarbeitungsreihenfolge, so dass durchschnittliche Bearbeitungszeit minimal ist.

Beispielproblem 1: Gieriges 1-Prozessor Scheduling

Gegeben

- ein Prozessor und n Jobs
- keine Parallelität: Prozessor bearbeitet zu jedem Zeitpunkt ≤ 1 Jobs
- angefangene Jobs können nicht abgebrochen werden
- j-ter Job hat Größe p_j

Gesucht

Abarbeitungsreihenfolge, so dass durchschnittliche Bearbeitungszeit minimal ist.

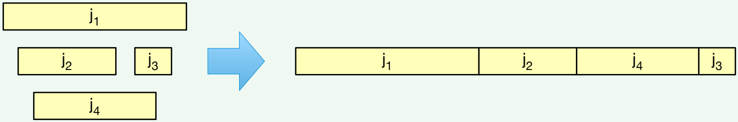
- wenn man Jobs nicht abbrechen darf, spricht man von non-preemptive Scheduling
- p_j: das „p“ steht für processing volume

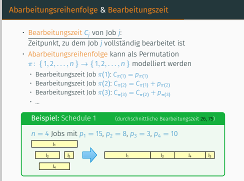
Abarbeitungsreihenfolge & Bearbeitungszeit

- Bearbeitungszeit C_j von Job j:
  Zeitpunkt, zu dem Job j vollständig bearbeitet ist
- Abarbeitungsreihenfolge kann als Permutation $\pi: \{1, 2, \ldots, n\} \to \{1, 2, \ldots, n\}$ modelliert werden
  - Bearbeitungszeit Job $\pi(1)$: $C_{\pi(1)} = p_{\pi(1)}$
  - Bearbeitungszeit Job $\pi(2)$: $C_{\pi(2)} = C_{\pi(1)} + p_{\pi(2)}$
  - Bearbeitungszeit Job $\pi(3)$: $C_{\pi(3)} = C_{\pi(2)} + p_{\pi(3)}$
  - ...

Beispiel: Schedule 1 (durchschnittliche Bearbeitungszeit $26,75$)

$n = 4$ Jobs mit $p_1 = 15$, $p_2 = 8$, $p_3 = 3$, $p_4 = 10$





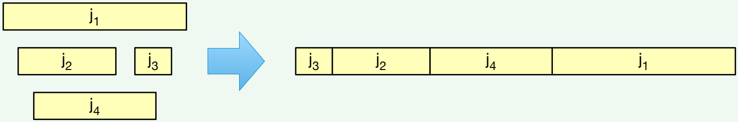
- C_j: das „C“ steht für completion time

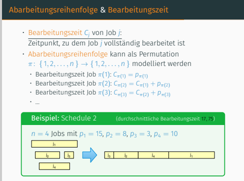
Abarbeitungsreihenfolge & Bearbeitungszeit

- Bearbeitungszeit C_j von Job j:
  Zeitpunkt, zu dem Job j vollständig bearbeitet ist
- Abarbeitungsreihenfolge kann als Permutation $\pi\colon \{1, 2, \ldots, n\} \to \{1, 2, \ldots, n\}$ modelliert werden
  - Bearbeitungszeit Job $\pi(1)$: $C_{\pi(1)} = p_{\pi(1)}$
  - Bearbeitungszeit Job $\pi(2)$: $C_{\pi(2)} = C_{\pi(1)} + p_{\pi(2)}$
  - Bearbeitungszeit Job $\pi(3)$: $C_{\pi(3)} = C_{\pi(2)} + p_{\pi(3)}$
  - ...

Beispiel: Schedule 2 (durchschnittliche Bearbeitungszeit $17,75$)

$n = 4$ Jobs mit $p_1 = 15$, $p_2 = 8$, $p_3 = 3$, $p_4 = 10$





- C_j: das „C“ steht für completion time

Wie gut ist gieriges Scheduling?

Lemma 6.1

Werden n Jobs gemäß der Permutation π bearbeitet, so beträgt die durchschnittliche Bearbeitungszeit

$$P(\pi) := \frac{1}{n} \cdot \sum_{j=1}^{n} (n - j + 1) \cdot p_{\pi(j)}.$$

Lemma 6.2

Eine Permutation π führt genau dann zu einer minimalen durchschnittlichen Bearbeitungszeit, wenn $p_{\pi(1)} \leq p_{\pi(2)} \leq \cdots \leq p_{\pi(n)}$.

2021-02-25

Lemma 6.1
Werden n Jobs gemäß der Permutation π bearbeitet, so beträgt die durchschnittliche Bearbeitungszeit

$$P(\pi) := \frac{1}{n} \cdot \sum_{j=1}^{n} (n-j+1) \cdot p_{\pi(j)}.$$

Lemma 6.2
Eine Permutation π führt genau dann zu einer minimalen durchschnittlichen Bearbeitungszeit, wenn $p_{\pi(1)} \leq p_{\pi(2)} \leq \cdots \leq p_{\pi(n)}$.

- Beweis von Lemma 6.1 erfolgt per vollständiger Induktion über n

Beweis Lemma 6.2

- Beweis per Widerspruch: Annahme π optimal aber...
- ...existiert i mit $p_{\pi(i)} > p_{\pi(i+1)}$
- sei $\tilde{\pi}$ wie π, außer dass $\tilde{\pi}(i) = \pi(i+1)$ und $\tilde{\pi}(i+1) = \pi(i)$
- nach Lemma 6.1 gilt

$$
\begin{aligned}
& n \cdot P(\pi) - n \cdot P(\tilde{\pi}) \\
= & \sum_{j=1}^{n} (n-j+1) \cdot p_{\pi(j)} - \sum_{j=1}^{n} (n-j+1) \cdot p_{\tilde{\pi}(j)} \\
= & (n-i+1) \cdot p_{\pi(i)} + (n-i) \cdot p_{\pi(i+1)} - (n-i+1) \cdot p_{\tilde{\pi}(i)} - (n-i) \cdot p_{\tilde{\pi}(i+1)} \\
= & p_{\pi(i)} - p_{\pi(i+1)} > 0
\end{aligned}
$$

- Widerspruch zur Optimalität von π! ⚡ $\square$

Beweis Lemma 6.2

- Beweis per Widerspruch: Annahme π optimal aber...
- ...existiert i mit $p_{\pi(i)} > p_{\pi(i+1)}$
- sei $\tilde{\pi}$ wie π, außer dass $\tilde{\pi}(i) = \pi(i+1)$ und $\tilde{\pi}(i+1) = \pi(i)$
- nach Lemma 6.1 gilt

$$\begin{aligned} & n \cdot P(\pi) - n \cdot P(\tilde{\pi}) \\ = & \sum_{j=1}^{n} (n-j+1) \cdot p_{\pi(j)} - \sum_{j=1}^{n} (n-j+1) \cdot p_{\tilde{\pi}(j)} \\ = & (n-i+1) \cdot p_{\pi(i)} + (n-i) \cdot p_{\pi(i+1)} - (n-i+1) \cdot p_{\tilde{\pi}(i)} - (n-i) \cdot p_{\tilde{\pi}(i+1)} \\ = & p_{\pi(i)} - p_{\pi(i+1)} > 0 \end{aligned}$$

- Widerspruch zur Optimalität von π! ⚡ □

- strenggenommen zeigen wir hier nur „π optimal“ $\Longrightarrow$ „π ist sortiert“
- die Rückrichtung folgt natürlich direkt, da alle sortierten π zur gleichen durchschnittlichen Bearbeitungszeit führen

Beispielproblem 2: Gieriges Mehr-Prozessor Scheduling

Gegeben

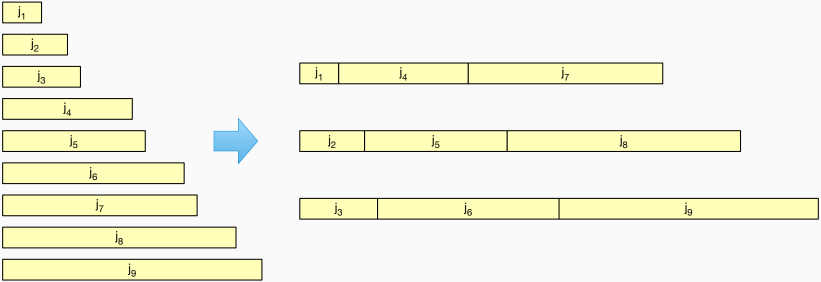
- m identische Prozessor und n Jobs
- keine Parallelität (auf einem Prozessor)
- angefangene Jobs können nicht abgebrochen werden
- j-ter Job hat Größe p_j

Gesucht

Aufteilung der Jobs auf Prozessoren sowie für jeden Prozessor eine Abarbeitungsreihenfolge der ihm zugewiesenen Jobs, so dass die durchschnittliche Bearbeitungszeit minimal ist.

Beispiel für Mehr-Prozessor Scheduling

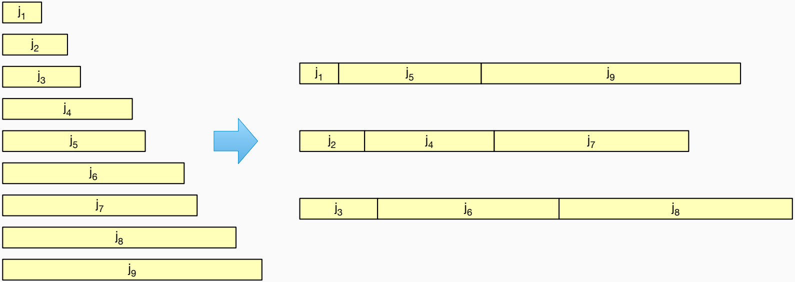
- $m = 3$ Prozessoren
- $n = 9$ Jobs
  - $p_1 = 3$, $p_2 = 5$, $p_3 = 6$, $p_4 = 10$, $p_5 = 11$, $p_6 = 14$, $p_7 = 15$, $p_8 = 18$, $p_9 = 20$



Schedule 1: durchschnittliche Bearbeitungszeit $18,\overline{33}$

Beispiel für Mehr-Prozessor Scheduling

- $m = 3$ Prozessoren
- $n = 9$ Jobs
  - $p_1 = 3$, $p_2 = 5$, $p_3 = 6$, $p_4 = 10$, $p_5 = 11$, $p_6 = 14$, $p_7 = 15$, $p_8 = 18$, $p_9 = 20$



Schedule 2: durchschnittliche Bearbeitungszeit $18,\overline{33}$

Wie gut ist gieriges Mehr-Prozessor Scheduling?

Lemma 6.3

Betrachte eine Permutation π mit $p_{\pi(1)} \leq p_{\pi(2)} \leq \cdots \leq p_{\pi(n)}$. Betrachte den Schedule, der Jobs gemäß π scheduled und dabei Job $\pi(i)$ auf Prozessor $i \mod m$ zuweist. Der so konstruierte Scheduling besitzt die minimale durchschnittliche Bearbeitungszeit.

Beweisskizze.

- Analogon von Lemma 6.1 gilt für jeden Prozessor
- $\rightsquigarrow$ wenn Prozessor $i \geq 2$ Jobs mehr als Prozessor i' hat...
  ...besserer Schedule durch verschieben des ersten Jobs von i auf i'
- vertauschen des j-ten Jobs zwischen Prozessoren i und i' ändert die durchschnittliche Bearbeitungszeit nicht
- durchschnittliche Bearbeitungszeit auf einem Prozessor minimal bei aufsteigender Sortierung (Lemma 6.2) $\square$

Wie gut ist gieriges Mehr-Prozessor Scheduling?

Lemma 6.3

Betrachte eine Permutation π mit $p_{\pi(1)} \leq p_{\pi(2)} \leq \cdots \leq p_{\pi(n)}$. Betrachte den Schedule, der Jobs gemäß π scheduled und dabei Job $\pi(i)$ auf Prozessor $i \mod m$ zuweist. Der so konstruierte Scheduling besitzt die minimale durchschnittliche Bearbeitungszeit.

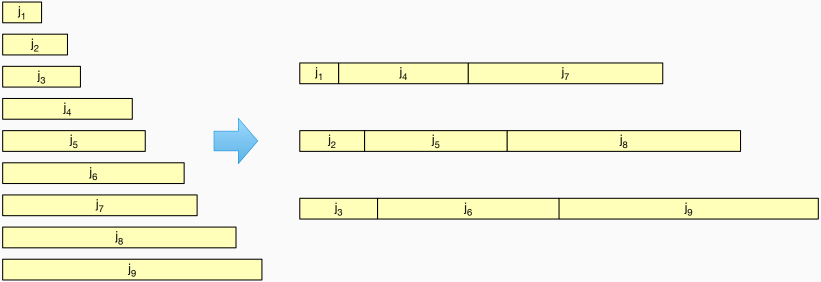
Beweisskizze.

- Analogon von Lemma 6.1 gilt für jeden Prozessor
- $\rightsquigarrow$ wenn Prozessor $i \geq 2$ Jobs mehr als Prozessor i' hat…
  …besserer Schedule durch verschieben des ersten Jobs von i auf i'
- vertauschen des j-ten Jobs zwischen Prozessoren i und i' ändert die durchschnittliche Bearbeitungszeit nicht
- durchschnittliche Bearbeitungszeit auf einem Prozessor minimal bei aufsteigender Sortierung (Lemma 6.2) □

- wir gehen hier davon aus, dass die Prozessoren von 0 bis $m-1$ nummeriert sind

Ist gierig immer gut?

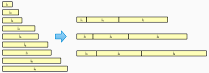
- betrachte wieder m Prozessoren und n Jobs
- neues Ziel: minimiere spätesten Endzeitpunkt



Schedule 1: Endzeitpunkt 40

Ist gierig immer gut?

- betrachte wieder m Prozessoren und n Jobs
- neues Ziel: minimiere spätesten Endzeitpunkt

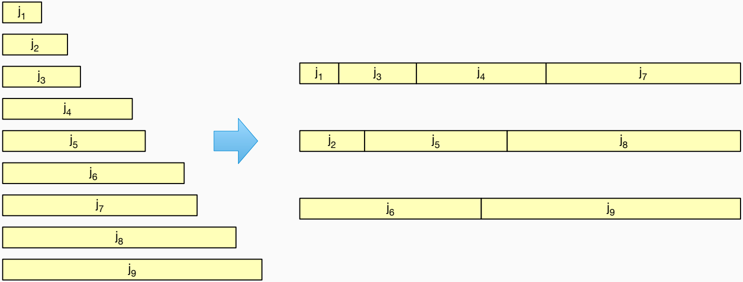


Schedule 1: Endzeitpunkt 40

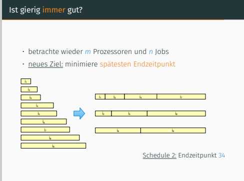
- man nennt dieses Kriterium in der Literatur „Makespan“

Ist gierig immer gut?

- betrachte wieder m Prozessoren und n Jobs
- neues Ziel: minimiere spätesten Endzeitpunkt



Schedule 2: Endzeitpunkt 34

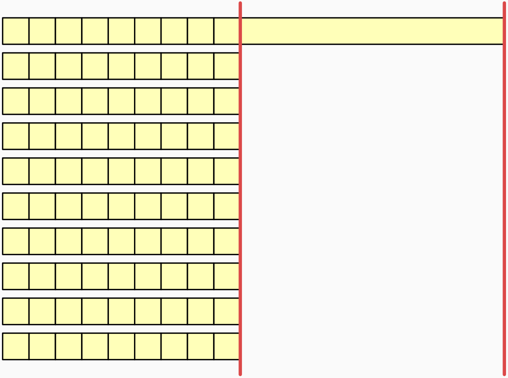


- man nennt dieses Kriterium in der Literatur „Makespan“

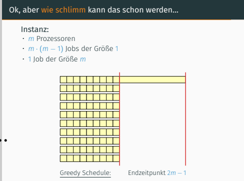
Ok, aber wie schlimm kann das schon werden...

Instanz:

- m Prozessoren
- $m \cdot (m-1)$ Jobs der Größe 1
- 1 Job der Größe m



Greedy Schedule: Endzeitpunkt $2m-1$

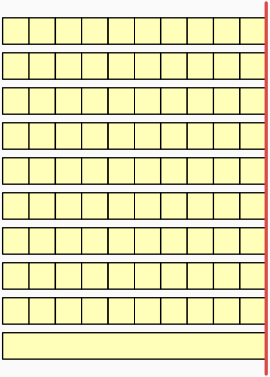


- Greedy ist also um einen Faktor $2 - 1/m$ schlechter als der optimale Algorithmus
- in der Tat ist dies auch die schlimmste Instanz; d. h. für keine andere Instanz wird der Faktor zwischen dem Greedy-Schedule und dem optimalen Schedule schlimmer als $2 - 1/m$
- man sagt deshalb, dass Greedy eine Approximationsgüte von $2 - 1/m$ besitzt

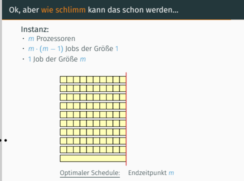
Ok, aber wie schlimm kann das schon werden...

Instanz:

- m Prozessoren
- $m \cdot (m-1)$ Jobs der Größe 1
- 1 Job der Größe m



Optimaler Schedule: Endzeitpunkt m



- Greedy ist also um einen Faktor $2 - 1/m$ schlechter als der optimale Algorithmus
- in der Tat ist dies auch die schlimmste Instanz; d. h. für keine andere Instanz wird der Faktor zwischen dem Greedy-Schedule und dem optimalen Schedule schlimmer als $2 - 1/m$
- man sagt deshalb, dass Greedy eine Approximationsgüte von $2 - 1/m$ besitzt

Wie wird man besser?

Wie können wir den Algorithmus für das schlechte Beispiel reparieren?

- führe Greedy Schedule aus, aber...
- ...sortiere die Jobs zuerst absteigend nach ihrer Größe

$\rightsquigarrow$ nie schlechter als $4/3 \cdot \mathrm{OPT}$

  - Beweis z. B. in *Methoden des Algorithmenentwurfes* (Master Modul Informatik)

Worst-case Instanz?

- m Maschinen
- je zwei Jobs der Länge $m+1$, $m+2$, ..., $2m$ und...
- ...ein Job der Länge m

$\rightsquigarrow$ um Faktor $4/3$ schlechter als OPT

Wie wird man besser?

Wie können wir den Algorithmus für das schlechte Beispiel reparieren?

- führe Greedy Schedule aus, aber...
- ...sortiere die Jobs zuerst absteigend nach ihrer Größe

⇝ nie schlechter als $4/3 \cdot \mathrm{OPT}$
  - Beweis z. B. in *Methoden des Algorithmenentwurfes* (Master Modul Informatik)

Worst-case Instanz?

- m Maschinen
- je zwei Jobs der Länge $m+1$, $m+2$, ..., $2m$ und...
- ...ein Job der Länge m

⇝ um Faktor $4/3$ schlechter als OPT

- Gute Übung: Zeige, dass die Worst-case Instanz tatsächlich um den Faktor $4/3$ schlechter ist als OPT!

Ein Beispiel aus der Praxis

Datenkompression:

- reduziert Größe von Dateien
- viele Verfahren für unterschiedliche Anwendungen

  MP3, MPEG, JPEG, ...
- Greedy-Verfahren: Huffman-Kodierung

Takeaway für Greedy-Algorithmen

- schön, einfach, elegant
- leider nicht immer optimal
- typischerweise ein erster Schritt im Algorithmendesign