

Kapitel 2

Kombinatorische Algorithmen

2.1 Analyse von Algorithmen

Kombinatorische Strukturen: Teilmengen einer gegebenen Menge, Permutationen, Graphen, geordnete n -tupel, Listen, Datenstrukturen, Lateinische Quadrate, ...

Beispiel Lateinisches Quadrat: Jedes Element einer gegebenen n -elementigen Menge kommt in jeder Zeile und Spalte genau einmal vor, z.B.

1	2	3	4
2	1	4	3
3	4	1	2
4	3	2	1

Kombinatorische Algorithmen: Verfahren zur Untersuchung kombinatorischer Strukturen. Dabei unterscheiden wir die Zielsetzungen:

1. *Erzeugen:* Konstruktion aller kombinatorischen Strukturen eines vorgegebenen Typs, z.B. alle 3-elementigen Teilmengen $\{a, b, c\} \subseteq \{1, 2, \dots, 100\}$, so dass $48 \leq a + b + c \leq 126$
2. *Abzählen:* Anzahlbestimmung verschiedener Strukturen eines gewissen Typs (meist einfacher als Erzeugen), z.B. Anzahl der k -elementigen Teilmengen einer n -elementigen Menge: $\binom{n}{k} = \frac{n!}{(n-k)!k!}$ (alle k -elementigen Teilmengen aufzuzählen ist viel komplizierter)
3. *Suche:* Finde mindestens ein Beispiel für eine kombinatorische Struktur eines bestimmten Typs (falls ein solches existiert), z.B. eine optimale Struktur.

Beispiel Knapsack-Problem

Gegeben:

p_0, p_1, \dots, p_{n-1}	Gewinne (Profite)
w_0, w_1, \dots, w_{n-1}	Gewichte (Aufwand)
M	obere Schranke für Aufwand

- Existiert für gegebenes P ein n -tupel $(x_0, \dots, x_{n-1}) \in \{0, 1\}^n$, so dass $\sum_{i=0}^{n-1} p_i x_i \geq P$ und $\sum_{i=0}^{n-1} w_i x_i \leq M$? (Entscheidungsproblem)
- Finde für gegebenes P alle n -tupel $(x_0, \dots, x_{n-1}) \in \{0, 1\}^n$, so dass $\sum_{i=0}^{n-1} p_i x_i \geq P$ und $\sum_{i=0}^{n-1} w_i x_i \leq M$. (Suchproblem)
- Finde ein n -tupel $(x_0, \dots, x_{n-1}) \in \{0, 1\}^n$, so dass $\sum_{i=0}^{n-1} p_i x_i$ maximal wird unter der NB $\sum_{i=0}^{n-1} w_i x_i \leq M$. (Problem der kombinatorischen Optimierung)

Beurteilung des Rechenaufwandes

$f(n)$ ist ein $O(g(n))$: $\Leftrightarrow \exists c > 0, n_0 \geq 0$, so dass $0 \leq f(n) \leq cg(n)$ für alle $n \geq n_0, n \in \mathbb{N}$ (hier und im folgenden ist $\mathbb{N} = \{0, 1, 2, \dots\}$)

Beispiel $f(n) = 3n^2 + 5n + 2$ ist ein $O(n^2)$, da $f(n) \leq 3n^2 + 5n^2 + 2n^2 = 10n^2$
 n zumeist Anzahl von Rechenschritten oder Zeiteinheiten

Man kann zeigen, dass jedes Problem auf ein oder mehrere Entscheidungsprobleme zurückgeführt werden kann. Für diese gilt dann nachfolgende Einteilung in Komplexitätsklassen:

Polynomzeitalgorithmus: Algorithmus, der die Lösung in der Zeit $O(n^k)$ ($k \geq 0$ konstant) liefert

Problem der Klasse NP: Es gibt einen nichtdeterministischen Polynomzeitalgorithmus, was JA-Antworten betrifft, d.h. jede konkret vorgelegte Lösung, welche zu einer positiven Antwort führt, kann in Polynomzeit auf Korrektheit überprüft werden (nichtdeterministisch ist die Art und Weise, wie man zu der Lösung kommt, im Extremfall durch Durchprobieren oder Raten!)

Die Existenz von Polynomzeitalgorithmen zur Lösung von NP Problemen ist vielfach unklar, oft kennt man nur Verfahren, die in der Zeit $O(c^n)$ ($c > 0$ konstant) rechnen (*exponentielle Komplexität*)

Beispiel Problem aus der Klasse NPGegeben: $w_0, w_1, \dots, w_{n-1}, M \in \mathbb{N}^* = \{1, 2, \dots\}$ Gefragt: Gibt es ein n -tupel $(x_0, \dots, x_{n-1}) \in \{0, 1\}^n$, so dass $\sum_{i=0}^{n-1} w_i x_i = M$?
(Verwendung in der Kryptographie: Merkle-Hellman-Knapsack-Kryptosystem)

Worst case complexity: Rechenaufwand im ungünstigsten Fall

Average case complexity: im Durchschnitt zu erwartender Rechenaufwand

Beispiel Algorithmus *INSERTION-SORT*(A, n)

Gegeben: Liste $A = [A[0], \dots, A[n-1]]$ von Zahlen aus \mathbb{R}

Gesucht: nach aufsteigender Ordnung sortierte Liste

Algorithmus:

$$\text{for } i \leftarrow 1 \text{ to } n-1$$

$$\text{do } \begin{cases} x \leftarrow A[i] \\ j \leftarrow i-1 \\ \text{while } j \geq 0 \text{ and } A[j] > x \\ \quad \text{do } \begin{cases} A[j+1] \leftarrow A[j] \\ j \leftarrow j-1 \end{cases} \\ A[j+1] \leftarrow x \end{cases}$$

Angenommen die ersten i Elemente auf den Plätzen $0, \dots, i-1$ sind bereits richtig geordnet.

while-Schleife: findet korrekte Position von $A[i]$, verschiebt $A[j+1]$, $A[j+2], \dots, A[i-1]$ passend

a. **Worst-case Analyse:** seien c_1 bzw. c_2 die pro Durchgang der while- bzw. der do-Schleife außerhalb der while-Schleife benötigten Zeiten; while-Schleife wird im ungünstigsten Fall $(i-1)$ -mal durchlaufen;
 \Rightarrow Rechenzeit $T(n) = \sum_{i=2}^n (c_2 + c_1(i-1)) = c_2(n-1) + \frac{c_1 n(n-1)}{2} = O(n^2)$

b. **Average-case Analyse:** $n!$ Möglichkeiten für $A = [A[0], \dots, A[n-1]]$; o.B.d.A. seien $0, 1, \dots, n-1$ zu ordnen (dann A Permutation);
 $N(A, i) := |\{j/0 \leq j \leq i-1, A[j] > A[i]\}|$ für festes A und $i = 0, 1, \dots, n-1$
 \Rightarrow Rechenzeit für festes A : $\sum_{i=1}^{n-1} (c_2 + c_1 N(A, i))$ (c_1, c_2 wie oben)
Für jedes Paar $j < i$ gibt es bei festgehaltenem i genau $\frac{n!}{2}$ Permutationen mit $A[j] > A[i]$, also $\sum_A N(A, i) = i \frac{n!}{2}$
 \Rightarrow Rechenaufwand im Durchschnitt

$$\begin{aligned} \bar{T}(n) &= \frac{1}{n!} \sum_A \left(\sum_{i=1}^{n-1} (c_2 + c_1 N(A, i)) \right) \\ &= \frac{1}{n!} \sum_A (n-1)c_2 + \frac{1}{n!} \sum_A \sum_{i=1}^{n-1} c_1 N(A, i) \\ &= \frac{1}{n!} n!(n-1)c_2 + \frac{c_1}{n!} \sum_{i=1}^{n-1} \sum_A N(A, i) \\ &= (n-1)c_2 + \frac{c_1}{n!} \sum_{i=1}^{n-1} \frac{n!}{2} i \\ &= (n-1)c_2 + \frac{c_1}{2} (1+2+\dots+n-1) \\ &= (n-1)c_2 + \frac{n(n-1)}{4} c_1 \\ &\approx \frac{T(n)}{2} \end{aligned}$$

2.2 Erzeugen, Abzählen und Suche

Lexikographische Ordnung $<_{lex}$: Seien $\vec{x} = (x_0, \dots, x_{n-1}), \vec{y} = (y_0, \dots, y_{n-1})$.

$\vec{x} <_{lex} \vec{y}$: $x_0 < y_0$ oder
 $(x_0 = y_0$ und $x_1 < y_1)$ oder
 $(x_0 = y_0$ und $x_1 = y_1$ und $x_2 < y_2)$ oder
 \vdots
 $(x_0 = y_0, x_1 = y_1, \dots, x_k = y_k$ und $x_{k+1} < y_{k+1})$ oder
 \vdots

Beispiel

- Lexikographische Ordnung aller Tripel $\vec{x} \in \{0, 1\}^3$ (vom kleinsten zum größten):
 $(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1), (1, 0, 0), (1, 0, 1), (1, 1, 0), (1, 1, 1)$
- Finde alle Teilmengen von $\{a, b, c\}$: Setze $x_0 = 1$, falls a vorkommt, $x_0 = 0$ sonst. Analog x_2, x_3 für b, c ;
lexikographische Ordnung aus a. $\Rightarrow \emptyset, \{c\}, \{b\}, \{b, c\}, \{a\}, \{a, c\}, \{a, b\}, \{a, b, c\}$

\Rightarrow Problem: finde den unmittelbaren Nachfolger

Teilmengen

Problem: Generiere alle k -elementigen Teilmengen $T = \{t_1, t_2, \dots, t_k\}$ einer n -elementigen Menge $S = \{1, 2, \dots, n\}$.

Wähle Bezeichnung der Elemente in T so, dass $t_1 < t_2 < \dots < t_k$; setze $\vec{T} := (t_1, \dots, t_k)$.

Die lexikographische Ordnung auf allen T induziert lexikographische Ordnung auf allen \vec{T} , also reduziert sich Problem darauf, ausgehend von $(1, 2, \dots, k)$ zu jedem (t_1, \dots, t_k) seinen Nachfolger zu finden.

Beispiel $S = \{1, 2, 3, 4, 5\}$, alle 3-elementigen Teilmengen:
 $(1, 2, 3), (1, 2, 4), (1, 2, 5), (1, 3, 4), (1, 3, 5), (1, 4, 5), (2, 3, 4), (2, 3, 5), (2, 4, 5), (3, 4, 5)$

Algorithmus *K-SUBSET-LEX-SUCCESSOR* (\vec{T}, k, n)

```

 $\vec{U} \leftarrow \vec{T}$ 
 $i \leftarrow k$ 
while  $(i \geq 1)$  and  $(t_i = n - k + i)$ 
  do  $i \leftarrow i - 1$ 
if  $i = 0$ 
  then return ("undefined")
else {
  for  $j \leftarrow i$  to  $k$ 
    do  $u_j \leftarrow t_i + 1 + j - i$ 
  return  $(\vec{U})$ 

```

Beispiel Nachfolger von $\vec{T} = (2, 3, 5)$ in obigem Beispiel

Bestimme K-SUBSET-LEX-SUCCESSOR $((2, 3, 5), 3, 5)$:

$\vec{U} \leftarrow (t_1, t_2, t_3) = (2, 3, 5)$
 $i = 3$
 $i = 3, t_3 = 5 - 3 + 3 = 5; \quad i \leftarrow i - 1$
 $i = 2, t_2 \neq 5 - 3 + 2$
 $i \neq 0$, daher else: $\begin{cases} \text{for } j = 2 \text{ bis } 3 \\ \quad u_j \leftarrow t_2 + 1 + j - 2 =_{(t_2=3)} 2 + j \quad \Rightarrow u_2 = 4, u_3 = 5 \\ \text{return } (u_1, u_2, u_3) = (2, 4, 5) \end{cases}$

Permutationen

Permutation π von $M = \{1, 2, \dots, n\}$: bijektive Abbildung $\pi : M \rightarrow M$; dargestellt als Liste $[\pi[1], \dots, \pi[n]]$

Problem: Ordne die $n!$ Listen lexikographisch.

Beispiel $3! = 6$ Permutationen von $M = \{1, 2, 3\}$:

$[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]$

Man erhält alle $n!$ Permutationen, indem man ausgehend von $[1, 2, \dots, n]$ zu jeder Liste $[\pi[1], \dots, \pi[n]]$ die unmittelbar nachfolgende Liste generiert.

Algorithmus *PERM-LEX-SUCCESSOR* (n, π)

<pre> $\pi[0] \leftarrow 0$ $i \leftarrow n - 1$ while $\pi[i + 1] < \pi[i]$ do $i \leftarrow i - 1$ if $i = 0$ then return ("undefined") $j \leftarrow n$ while $\pi[j] < \pi[i]$ do $j \leftarrow j - 1$ $t \leftarrow \pi[j]$ $\pi[j] \leftarrow \pi[i]$ $\pi[i] \leftarrow t$ for $h \leftarrow i + 1$ to n do $\rho[h] \leftarrow \pi[h]$ for $h \leftarrow i + 1$ to n do $\pi[h] \leftarrow \rho[n + i + 1 - h]$ return (π) </pre>	<p>Erklärung:</p> <p>Erste while-Schleife: bestimmt Index i, für den $\pi[i + 1] > \pi[i + 2] > \dots > \pi[n]$, jedoch $\pi[i] < \pi[i + 1]$; $\pi[0] = 0$ stellt sicher, dass Schleife mit $0 \leq i \leq n - 1$ endet, wobei $i = 0$ bedeutet, dass $\pi = [n, n - 1, \dots, 1]$ (größtes Element der Liste) vorliegt</p> <p>Zweite while-Schleife: sucht unter Elementen $\pi[i + 1], \dots, \pi[n]$ das letzte, das größer als $\pi[i]$ ist; seine Position sei j</p> <p>Dann: vertausche $\pi[i]$ und $\pi[j]$</p> <p>Zuletzt: Umkehrung der Ordnung der neuen Liste $\pi[i + 1], \dots, \pi[n]$</p>
---	---

Beispiel PERM-LEX-SUCCESSOR $(7, [3, 6, 2, 7, 5, 4, 1])$

erste while-Schleife: $i = 3, \pi[3] = 2$; zweite while-Schleife: $j = 6, \pi[6] = 4$; \Rightarrow neue Liste := $[3, 6, 4, 7, 5, 2, 1]$; zuletzt wird $[7, 5, 2, 1]$ ersetzt durch $[1, 2, 5, 7]$
 \Rightarrow Nachfolger von π : $[3, 6, 4, 1, 2, 5, 7]$

Anzahlen von Klasseneinteilungen

Stirlingsche Zahl 2. Art $S(m, n)$: Anzahl der Partitionen (Klasseneinteilungen) von $M = \{1, 2, \dots, m\}$, $m \in \mathbb{N}^*$, in genau $n \leq m$ nichtleere Teilmengen

Beispiel $M = \{1, 2, 3, 4\}$; verwende Abkürzung: $\{1|2\ 3\ 4\}$ statt $\{\{1\}, \{2, 3, 4\}\}, \dots$

$$\begin{aligned} n = 1 & \quad \{M\} & \Rightarrow S(4, 1) = 1 \\ n = 2 & \quad \{1|2\ 3\ 4\}, \{2|1\ 3\ 4\}, \{3|1\ 2\ 4\}, \{4|1\ 2\ 3\}, \\ & \quad \{1\ 2|3\ 4\}, \{1\ 3|2\ 4\}, \{1\ 4|2\ 3\} & \Rightarrow S(4, 2) = 7 \\ n = 3 & \quad \{1|2|3\ 4\}, \{1|3|2\ 4\}, \{1|4|2\ 3\}, \\ & \quad \{2|3|1\ 4\}, \{2|4|1\ 3\}, \{3|4|1\ 2\} & \Rightarrow S(4, 3) = 6 \\ n = 4 & \quad \{1|2|3|4\} & \Rightarrow S(4, 4) = 1 \end{aligned}$$

Satz 1 $S(m, n) = \frac{1}{n!} \sum_{j=1}^n (-1)^{n-j} \binom{n}{j} j^m$ (ohne Beweis)

Beweis des Satzes verwendet das *Inklusions-Exklusions-Prinzip*: Seien E_1, \dots, E_t Eigenschaften, die die Elemente einer endlichen Menge X haben oder nicht haben. $X_i := \{a \in X \mid a \text{ hat die Eigenschaft } E_i\}$, $i = 1, 2, \dots, t$
 $\Rightarrow |X - \bigcup_{i=1}^t X_i| = |X| - \sum_{i=1}^t |X_i| + \sum_{i < j, i, j=1}^t |X_i \cap X_j| \mp \dots (-1)^t |X_1 \cap \dots \cap X_t|$

Beispiel $X = \{1, 2, \dots, 12\}$; Eigenschaften $E_1 : 2$ teilt $a \in X$, $E_2 : 3$ teilt $a \in X$, $E_3 : 4$ teilt $a \in X$
 $\Rightarrow X_1 = \{2, 4, 6, 8, 10, 12\}$, $X_2 = \{3, 6, 9, 12\}$, $X_3 = \{4, 8, 12\}$
 $\Rightarrow |X - (X_1 \cup X_2 \cup X_3)| = 12 - (6 + 4 + 3) + (2 + 3 + 1) - 1 = 4$, d.h. 4 Zahlen werden weder durch 2,3 oder 4 geteilt (nämlich 1,5,7,10).

Satz 2 Für $n, m \in \mathbb{N}^*$, $n \geq m$ gilt:

$$S(m, n) = nS(m-1, n) + S(m-1, n-1).$$

Beweis Sei $S^*(m, n)$ die Menge aller Partitionen von $M = \{1, 2, \dots, m\}$ in genau n Klassen. Für $m-1$ sei $S^*(m-1, n) = \{\{A_1, A_2, \dots, A_n\}\}$. Für jede Partition $\{A_1, \dots, A_n\}$ von $\{1, 2, \dots, m-1\}$ bilden wir die n neuen Partitionen $\{A_1, A_2, \dots, A_i \cup \{m\}, A_{i+1}, \dots, A_n\}$, $i = 1, 2, \dots, n$, welche zu $S^*(m, n)$ gehören; das sind $nS^*(m-1, n)$ Stück.

Es fehlen dann in $S^*(m, n)$ noch alle Partitionen, bei denen $\{m\}$ alleine eine Klasse bildet, d.h. alle Partitionen der Form $\{A_1, \dots, A_{n-1}, \{m\}\}$; das sind $S^*(m-1, n-1)$ Stück.

$$\Rightarrow S(m, n) = nS(m-1, n) + S(m-1, n-1).$$

\Rightarrow Algorithmus: *STIRLING-NUMBERS* (m, n)

```

S(0, 0) ← 1
for i ← 1 to m
  do S(i, 0) ← 0
for i ← 0 to m
  do S(i, i+1) ← 0
for i ← 1 to m
  do { for j ← 1 to min{i, n}
      do S(i, j) ← jS(i-1, j) + S(i-1, j-1)
    }
return (S)

```

Die Zahlen $B(m) := \sum_{n=1}^m S(m, n)$, welche die Anzahlen aller Partitionen von $\{1, 2, \dots, m\}$ wiedergeben, heißen *Bell-Zahlen*. Sie genügen der Rekursion

$$B(m) = \sum_{i=0}^{m-1} \binom{m-1}{i} B(i) \text{ mit } B(0) = 1$$

(ohne Beweis). Weiters kann man zeigen: $\sum_{m=0}^{\infty} \frac{B(m)}{m!} x^m = e^{e^x - 1}$.

Gibt man die Anzahlen a_m von Objekten mit gewissen Eigenschaften, welche von der Zahl m abhängen, in der Form $\sum_{m=0}^{\infty} a_m x^m$ bzw. $\sum_{m=0}^{\infty} \frac{a_m}{m!} x^m$ an, so heißen die durch diese Reihen definierten Funktionen $f(x)$ *erzeugende* bzw. *exponentiell erzeugende Funktionen*.

Vorteil: Entwickelt man $f(x)$ in eine Taylorreihe, so erhält man als Koeffizienten von x^m die Anzahlen a_m beziehungsweise $\frac{a_m}{m!}$.

Beispiel Bell-Zahlen

$$f(x) = e^{e^x - 1}; \quad f(x) = \sum_{k=0}^{\infty} \frac{f^{(k)}(0)}{k!} x^k$$

$$f(0) = 1; \quad f'(x) = e^{e^x - 1} e^x = e^{e^x - 1 + x} \Rightarrow f'(0) = 1$$

$$f''(x) = e^{e^x - 1 + x} (e^x + 1) \Rightarrow f''(0) = 2$$

$$f'''(x) = e^{e^x - 1 + 3x} + 3e^{e^x - 1 + 2x} + e^{e^x - 1 + x} \Rightarrow f'''(0) = 5$$

analog $f^{(iv)}(0) = 15$

Damit $f(x) = 1 + x + x^2 + \frac{5}{3!} x^3 + \frac{15}{4!} x^4 + \dots$, d.h. $\frac{B(0)}{0!} = 1, \frac{B(1)}{1!} = 1, \frac{B(2)}{2!} = 1, \frac{B(3)}{3!} = \frac{5}{3!}, \frac{B(4)}{4!} = \frac{15}{4!}$.

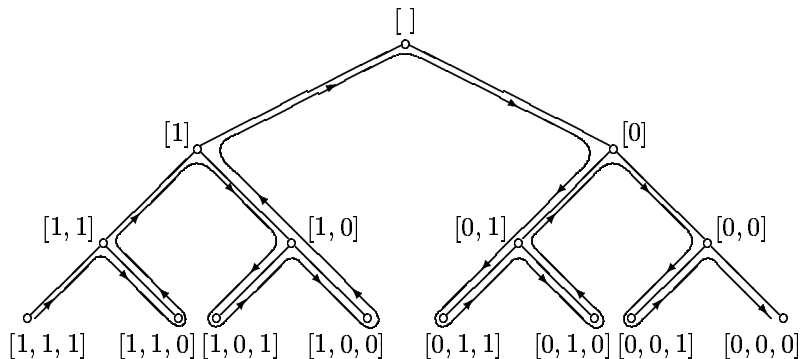
Suche eines Optimums

Beispiel Knapsack (vergleiche Kapitel 2.1)

Gegeben $P = [p_0, \dots, p_{n-1}]$, $W = [w_0, \dots, w_{n-1}]$ und M . Gesucht ist der größte Wert $OptP$, den $P = \sum_{i=0}^{n-1} x_i p_i$ unter den Bedingungen $\sum_{i=0}^{n-1} w_i x_i \leq M$ mit $x_i \in \{0, 1\}$ für $i = 0, 1, \dots, n - 1$ annehmen kann.

Für ein aktuell betrachtetes $X = [x_0, \dots, x_{n-1}]$ bezeichne $CurP$ den zugehörigen Wert von $\sum_{i=0}^{n-1} x_i p_i$. Die Stelle(n) von X , wo $OptP$ angenommen wird, werden mit $OptX$ bezeichnet. Berechnung von $OptP$ und $OptX$ erfolgt mit Hilfe eines sogenannten *Backtracking Algorithmus*:

Wir generieren alle 2^n n -tupel $[x_0, \dots, x_{n-1}]$ in umgekehrt lexikographischer Ordnung und gehen von einem X zum nächsten, so wie aus dem nachstehenden Graphen ersichtlich ist:



Dieser Graph ist ein *Baum* (kreisloser Graph); $[\]$ ist seine *Wurzel* (*Zustands-Baum*).

Generierung der einzelnen Niveaus $l = 0, 1, 2, 3$ von Knoten erfolgt mittels Algorithmus *KNAPSACK* (l):

```

global  $X, OptP, OptX$ 
if  $l = n$ 
  then  $\left\{ \begin{array}{l} \text{if } \sum_{i=0}^{n-1} w_i x_i \leq M \\ \quad \text{then } \left\{ \begin{array}{l} CurP \leftarrow \sum_{i=0}^{n-1} p_i x_i \\ \text{if } OptP > CurP \\ \quad \text{then } \left\{ \begin{array}{l} OptP \leftarrow CurP \\ OptX \leftarrow [x_0, \dots, x_{n-1}] \end{array} \right. \end{array} \right. \end{array} \right.$ 
  else  $\left\{ \begin{array}{l} x_l \leftarrow 1 \\ KNAPSACK(l+1) \\ x_l \leftarrow 0 \\ KNAPSACK(l+1) \end{array} \right.$ 

```