INSTITUT FÜR INFORMATIK FREIBURG

Vorlesung Informatik 2
Algorithmen und Datenstrukturen

(04 – Entwurfsverfahren)

Prof. Dr. Susanne Albers

Entwurfsverfahren für Algorithmen

- Divide and Conquer
- Greedy Verfahren
- Dynamische Programmierung
- Vollständige Aufzählung
- Backtracking
- Scan- (oder Sweep) Verfahren

Divide-and-Conquer Verfahren

Teile und Herrsche; Divide et impera

Allgemeine Problem-unabhängige Formulierung des Prinzips:

D&C-Verfahren = Methode *V* zur Lösung des Problems *P* der Größe *n*:

1. (Direkte Lösung) Falls $n < d$, löse das Problem direkt, sonst
2. (Divide) Teile *P* in zwei oder mehr kleine Teilprobleme $P_1, \ldots, P_k$, $k \geq 2$.
3. (Conquer) Löse jedes P_i rekursiv mit der Methode *V* (auf gleiche Art)
4. (Merge) Setze die Teillösungen zusammen

Merkmale von D&C-Verfahren:

- Breite Anwendbarkeit: Suchen, Sortieren, Geometrie, DFT
- Laufzeitanalyse über Lösungen von Rekursionsgleichungssystemen
- Effizienz des Merge-Schritts (# und Größe der Teilprobleme) ist entscheidend

Produkt von Polynomen in Koeffizientendarstellung (1)

Seien $a(x) = a_0 x^0 + a_1 x^1 + \ldots + a_{n-1} x^{n-1}$

$b(x) = b_0 x^0 + b_1 x^1 + \ldots + b_{n-1} x^{n-1}$

Polynome vom Grad $\leq$ n -1, d.h. mit $\leq$ n Koeffizienten; o.B.d.A. $n = 2^k, k \in N$

Problem: Berechne $c(x) = a(x) \cdot b(x)$ in Koeffizientendarstellung mit möglichst wenig Multiplikationen:

$$
\begin{aligned}
a(x) * b(x) = {} & [a_0 b_0] && * x^0 \\
& + [a_1 b_0 + a_0 b_1] && * x^1 \\
& + [a_2 b_0 + a_1 b_1 + a_0 b_2] && * x^2 \\
& \vdots \\
& + [a_{n-1} b_{n-1}] && * x^{2n-2}
\end{aligned}
$$

Naives Verfahren benötigt n^2 viele Koeffizientenmultiplikationen.

Geht es besser?

Produkt von Polynomen in Koeffizientendarstellung (2)

$$\text{Betrachte } a(x) = a_l(x) + x^{(\frac{n}{2})} a_r(x)$$

$$b(x) = b_l(x) + x^{(\frac{n}{2})} b_r(x)$$

$a_l(x)$ und $b_l(x)$ bestehen aus den niedrigeren Koeffizienten von $a(x)$ bzw. $b(x)$ während $a_r(x)$ und $b_r(x)$ die höheren Koeffizienten enthalten. Der Polynomgrad hat sich jeweils halbiert.

$$\Rightarrow a(x) * b(x) = \boxed{a_l(x) * b_l(x)} =: A$$

$$+ x^{\left(\frac{n}{2}\right)} \boxed{a_l(x) * b_r(x) + a_r(x) * b_l(x)} =: C$$

$$+ x^n \boxed{a_r(x) * b_r(x)} =: B$$

Direkte Umsetzung in rekursives Verfahren führt zu

$$\left.\begin{aligned} T(1) &= 1 \\ T(n) &= 4T(n/2) \end{aligned}\right\} \Rightarrow T(n) = n^2 \qquad \text{(Kein Gewinn!)}$$

Produkt von Polynomen in Koeffizientendarstellung (3)

$$\Rightarrow a(x) * b(x) = \boxed{a_l(x) * b_l(x) =: A}$$
$$+ x^{\left(\frac{n}{2}\right)} \boxed{(a_l(x) * b_r(x) + a_r(x) * b_l(x)) =: C}$$
$$+ x^n \boxed{a_r(x) * b_r(x) =: B}$$

Was ist $\left[a_l(x)+a_r(x)\right]*\left[b_l(x)+b_r(x)\right]$?

$$= A+\underbrace{\left[a_l(x)*b_r(x)+a_r(x)*b_l(x)\right]}_{C}+B$$

$$\Rightarrow C=\underbrace{\left[a_l(x)+a_r(x)\right]}_{\text{Größe}\frac{n}{2}}*\underbrace{\left[b_l(x)+b_r(x)\right]}_{\text{Größe}\frac{n}{2}}-A-B$$

Es sind also jetzt nur noch 3 Teilprobleme halber Größe zu lösen: Die Bestimmung von A, B und C!

Produkt von Polynomen in Koeffizientendarstellung (4)

Die Abschätzung ergibt jetzt

$$\left.\begin{aligned} T(1) &= 1 \\ T(n) &= 3\,T(n/2) \end{aligned}\right\} \Rightarrow T(n) = n^{\log 3} \approx n^{1.584}$$

n	n^2	$n^{\log 3}$
1	1	1
16	256	81
256	65536	6561
4096	16777216	531441

Es geht noch besser mit anderem D&C-Verfahren: $o(n \log n)$

Produkt von Polynomen in Koeffizientendarstellung (5)

Das Programm in Java:

```
public static int[] prod (int[] a, int[] b) {
    int n  = a.length,                  // Problemgroesse
        nh = n/2;                       // halbe Problemgroesse
    int[] r = new int [2*n];            // Ergebnisarray
    if (n==1) {                         // Kleines Teilproblem:
        r[0] = a[0] * b[0];             // Direkte Loesung
    } else {                            // sonst:
        int[] al  = new int [nh], ar = new int [nh],        // ********
              bl  = new int [nh], br = new int [nh],        // *      *
              alr = new int [nh], blr = new int [nh];       // *      *
        for (int i=0; i<nh; i++){                   // ***********      *
            alr [i]  = al [i] = a [i];              // *                *
            blr [i]  = bl [i] = b [i];              // *                *
            alr [i] += ar [i] = a [i+nh];           // *      DIVIDE    *
            blr [i] += br [i] = b [i+nh];           // *                *
        } // ******************************        // ******************

        int[] A = prod (al, bl); // ********      // ******************
        int[] B = prod (ar, br);                   // *      CONQUER   *
        int[] C = prod (alr, blr); // *******     // ******************

        for (int i=0; i<n; i++) { // ********      // ******************
            r [i] += A [i];                        // *                *
            r [i+nh] += C [i] - A [i] - B [i];     // *      MERGE     *
            r [i+n] = B [i];                       // *                *
        } // ******************************        // ******************
    }
    return r;
}
```

Greedy-Verfahren (1)

Gierige Verfahren: Das günstigste (größte, beste) zuerst

Typische Anwendungsgebiete: Optimierungsprobleme, etwa

- beste Bearbeitungsreihenfolge für Jobs in Computer
- kürzeste Wege in einem Graphen
- minimale Zahl von Münzen in beim Geldwechseln

Bedingungen für die Anwendbarkeit:

- Die Problemlösung besteht aus einer optimalen Auswahl (Kombination, Reihenfolge,.) der Elemente einer Kandidatenmenge.
- Mit Hilfe einer speziellen Funktion kann geprüft werden, ob eine Kandidatenmenge zulässig (feasible) ist.
- Eine Bewertungsfunktion erlaubt eine Reihung von und damit Entscheidung zwischen noch nicht gewählten Kandidaten.

Greedy-Verfahren (2)

Bsp: Das Münzwechsel-Problem:

Gegeben: Wert W des Wechselgeldes und eine Reihe von Münzwerten,
etwa: 1, 2, 5, 10, 20, 50, 100, 200

Gesucht: Eine Folge von Münzwerten minimaler Länge mit Gesamtwert W

Frage: Wie findet man eine minimale Münzfolge — möglichst effizient?

```
W ≥ 0 sei gegeben
wähle B ← Wert der größten Münze
while W > 0 {
   while B≤ W {
      zahle B aus;
      W ← W – B;
   }
  B ← Wert der nächst kleineren Münze;
}
```

$$147 \xrightarrow{100} 47 \xrightarrow{20} 27 \xrightarrow{20} 7 \xrightarrow{5} 2 \xrightarrow{2} 0$$

Greedy-Verfahren (3)

Frage: Ist die gefundene Lösung beim Münzwechsel-Problem immer optimal?

Betrachte etwa Münzwerte 1, 20, 41 und $W = 60$:

Greedy-Lösung: $60 = 1 \cdot 41 + 19 \cdot 1$ (20 Münzen)
optimale Lösung: $60 = 3 \cdot 20$ (3 Münzen)

Antwort: Das hängt entscheidend vom Münzsystem ab!

Das Verhältnis V aus #Münzen nach Greedy zu optimaler #Münzen kann beliebig schlecht werden:

Münzwerte: $1, B, 2B + 1$
$W = 3B$
$\Big\} V = \frac{B}{3}$

Greedy-Verfahren (4)

Allgemeine Formulierung des Greedy-Verfahrens:

```
solution = Ø;                    // wird zur Lösungsmenge erweitert
while (solution unvollständig AND Kandidatenmenge ≠ Ø) do
        x ← bestes Element aus Kandidatenmenge; // x wird aus Kandidatenmenge
                                                   entfernt
        füge x zu solution hinzu;
        if (solution unzulässig) then entferne x aus solution;

if (solution ist vollständig) then return solution else return Ø;
```

Beachte: Einmal getroffene Entscheidung über Auswahl eines bestmöglichen Elementes wird bei Greedy-Verfahren nicht revidiert.

Dynamische Programmierung (1)

Name deutet nicht auf Art der Programmerstellung sondern auf Tabellierungstechnik hin.

Rekursiver Ansatz: Lösen eines Problems durch Lösen mehrerer kleinerer Teilprobleme, aus denen sich die Lösung für das Ausgangsproblem zusammensetzt.

Phänomen: Mehrfachberechnungen von Lösungen.

Methode: Speichern einmal berechneter Lösungen in einer Tabelle für spätere Zugriffe.

Beispiel: Fibonacci Zahlen

$$F(0) = 0$$

$$F(1) = 1$$

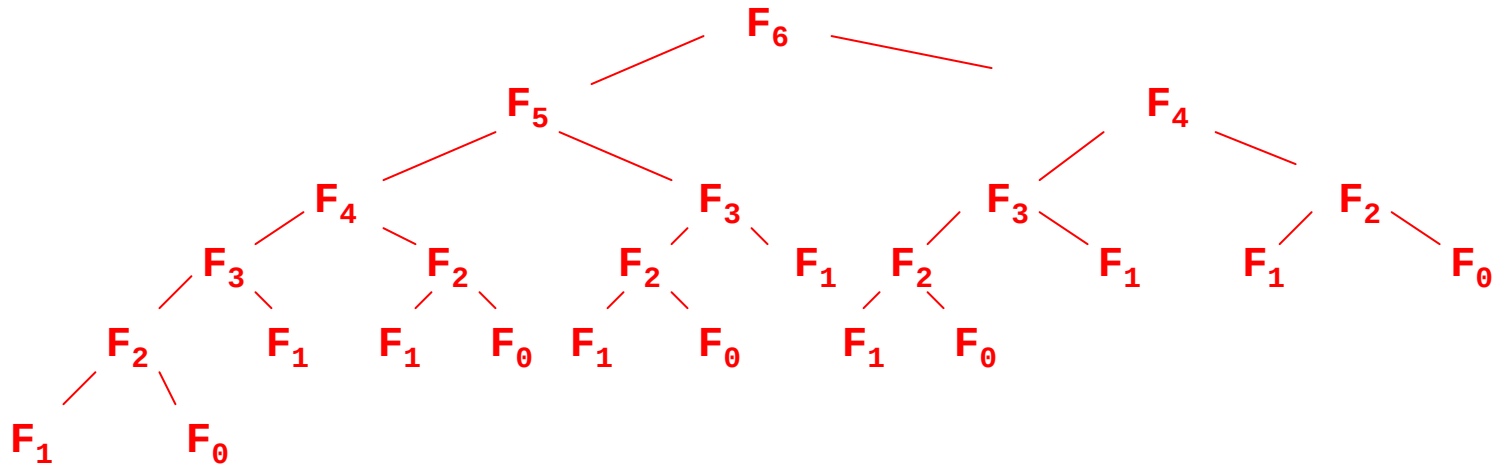
$$F(n) = F(n-1) + F(n-2), \quad \text{falls} \quad n \geq 2$$

Dynamische Programmierung (2)

Erster Ansatz:

procedure *F* (*n* : *integer*) : *integer*
if (*n* = 0) or (*n* = 1)
then return *n*
else return *fib*(*n* – 1) + *fib*(*n* – 2)

Nachteil: Laufzeit $T(n)$ exponentiell in n:



Dynamische Programmierung (3)

Die # der rekursiven Aufrufe nimmt mit wachendem Argument exponentiell zu:

$$T(n) \approx [1.447 \times 1.618^{n} - 1]$$

Problem: Es werden mehrfach die gleichen Teilprobleme gelöst

Vermeidung : Durch Tabellierung; Lege Tabelle für auftretende Funktionswerte an; jeder Wert wird nur einmal berechnet und später dann über die Tabelle zugegriffen

```
procedure F (n : integer) : integer
1       f_0 := 0; f_1 := 1
2       for k := 2 to n do
3                   f_k := f_k-1 + f_k-2
4       return f_n
```

Dynamische Programmierung (4)

Bsp: Das Münzwechsel2-Problem:

Gegeben: Wert *G* des Wechselgeldes und eine Reihe von Münzwerten,

Werte:	1	2	5	10	20	50	100	200
Indices:	0	1	2	3	4	5	6	7

Gesucht: Die # der verschiedenen Möglichkeiten *G* in Münzen auszuzahlen

Frage: Wie findet man diese # — möglichst effizient?

Sei $W(G, i)$ = # der verschiedenen Möglichkeiten, *G* auszuzahlen unter Nutzung von Münzen höchstens mit Index *i*

$$W(G,i) = \begin{cases} W(G - Betrag[i], i) & \text{,Münze } i \text{ wird mind. } 1\times \text{ benutzt} \\ +W(G,i\text{-}1) & \text{,Münze } i \text{ wird nicht benutzt} \end{cases}$$

Dynamische Programmierung (5)

Algorithmus Geldwechsel:

Input: Positiver Geldbetrag *G*; Array *betrag* [0,*n*-1] der Münzwerte

Output: Anzahl der Zahlungsmöglichkeiten

1. **for** $g \leftarrow 1$ **to** *G* **do** $W(g,0) \leftarrow 1$;
2. **for** $i \leftarrow 1$ **to** n-1 **do** $W(0,i) \leftarrow 1$;
3. **for** $g \leftarrow 1$ **to** *G* **do**
4. **for** $i \leftarrow 1$ **to** n-1 **do**
5. **if** *g-betrag*[*i*] ≥ 0 **then**
6. $W(g,i) \leftarrow W(g\text{-}betrag[i],i) + W(g,i\text{-}1)$;
7. **else** $W(g,i) \leftarrow W(g,i\text{-}1)$;
8. **return** $W(G,n\text{-}1)$;

Dynamische Programmierung (6)

Es wurde eine 2-dim. Tabelle mit Indices 0 . . . 7 und 0 . . .G benutzt: Größe in O(G)

Geht es besser?

Ja, in O(1)

Vollständige Aufzählung (1)

Prinzip: Systematische Erzeugung aller potentiellen Kombinationen (Reihenfolgen), welche zu einer Lösung des Problems führen können. Dabei Auswahl einer (optimalen) Lösung.

Anwendbarkeit: Probleme, bei denen es viele Kandidaten für eine Lösung gibt, von denen eine ausgewählt werden soll, etwa bei Optimierungsproblemen.

Effizienz: Meist schlecht, nicht polynomiell.

Beispiel:

Travelling Salesman-Problem (TSP)

Problem-Instanz:

Städte: $1 \quad 2 \; . \; . \; . \; n$

Entfernungen: $d_{ij} \in R^{+} \; (1 \leq i, j \leq n)$ mit $d_{ij} = 0$

Zulässige Lösung: Permutation π von $(1, \; 2, \; . \; . \; . \; , \; n)$

Zielfunktion: $P(\pi) = \sum_{i=1}^{n} d_{\pi(i),\pi(i \bmod n+1)}$

Optimale Lösung: Zulässige Lösung π mit minimalem $P(\pi)$

Vollständige Aufzählung (2)

```
public static void main (String[] arg){
    System.out.println ("TSP");
    int N = 5;
    int[][] mat = { {0, 2, 2, 5, 4}, {3, 0, 5, 2, 2},
                    {6, 7, 0, 1, 6}, {3, 2, 7, 0, 3},
                    {5, 4, 3, 8, 0} }; // Matrix der Kosten
    Perm p = new Perm (N); // Kann Permutationen von 0 .. N-1 liefern
    int cost = Integer.MAX_VALUE;
    int [] c, best = { };
    while ((c = p.getNext()) != null) { // Naechste Permutation
        int actcost = 0;
        for (int j=0; j<N; j++){ actcost += mat [c[j]][c[(j+1)%N]]; }
        System.out.println (arrStr(c)+" --> "+actcost);
        if (actcost < cost){
            cost = actcost;
            best = c;
        }
    }
    System.out.println("Die geringsten Kosten "+cost+
                       " verursacht "+arrStr(best));
} // main
```

Vollständige Aufzählung (3)

Ein Programmlauf:

```
[0,1,2,3,4] --> 16
[0,1,2,4,3] --> 24
[0,1,3,2,4] --> 22
[0,1,3,4,2] --> 16
[0,1,4,2,3] --> 11
[0,1,4,3,2] --> 25
[0,2,1,3,4] --> 19
[0,2,1,4,3] --> 22
[0,2,3,1,4] --> 12
[0,2,3,4,1] --> 13
[0,2,4,1,3] --> 17
[0,2,4,3,1] --> 21
...
[0,4,2,3,1] --> 13
[0,4,3,1,2] --> 25
[0,4,3,2,1] --> 29
Die geringsten Kosten 11 verursacht [0,1,4,2,3]
```

Vollständige Aufzählung (4)

Wie kann man die Permutationen von *n* Elementen systematisch erzeugen?

Jedes Element steht einmal vorne, während die restlichen *n* - 1 Elemente zu permutieren sind → Rekursiver Ansatz.

```
void permutations (set S, object[] perm, int i)
// perm enthält Elemente auf Stellen 0..i-1; S enthält die Elemente, die noch nicht in perm
    vorkommen; i ist die nächste Stelle, die zu besetzen ist.
1.  if (S = ∅) then gib perm aus;
2.  else
3.          for all  elem ∈ S do
4.                  perm[i] ←elem;
5.                  permutations(S\{elem}, perm, i+1);
```

Backtracking (1)

Backtracking ist eine spezielle Methode zur systematischen und erschöpfenden Suche im Raum sämtlicher Möglichkeiten.

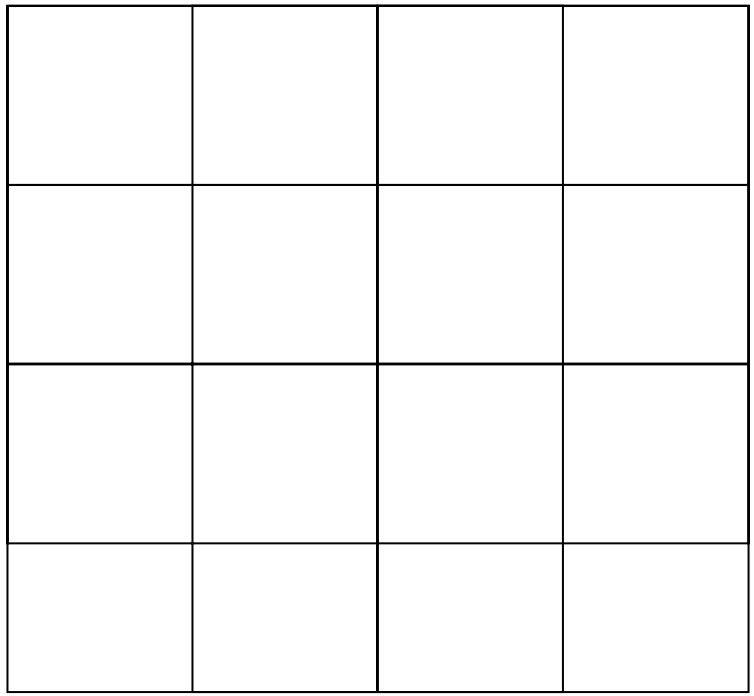
- Korrekte Teillösungen werden schrittweise bis zu Gesamtlösungen erweitert.
- Ist die Erweiterung der aktuellen Teillösung nicht mehr möglich, wird der vorhergehende Schritt zurückgenommen und durch den nächsten Erweiterungsschritt ersetzt.

Beispiel: *N*-Dame Problem

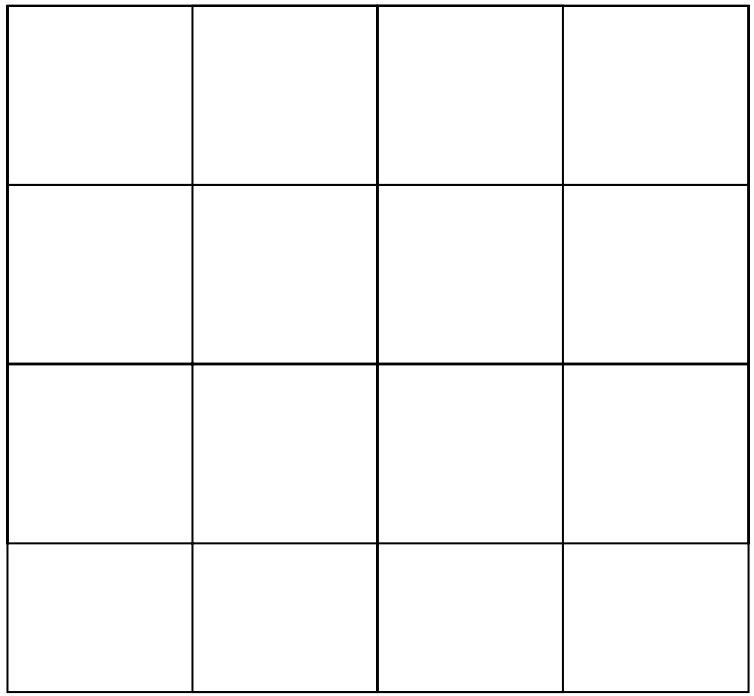
Gegeben: $N \times N$ Schachbrett S

Aufgabe: Positioniere N Damen auf S, so dass sie sich gegenseitig (nach Schachregeln) nicht schlagen können.
(Keine zwei Damen dürfen die gleiche Zeile, Spalte, oder Diagonale besetzen.)

4-Damen Problem



4-Damen Problem



Backtracking (2)

Beispiel für $N = 4$:

	i=0	1	2	3
j=0			**D**	
1	**D**			
2				**D**
3		**D**		

Die Lösungen lassen sich als Vektoren schreiben: (1, 3, 0, 2) bedeutet, dass die Dame in Spalte 0 in Zeile 1 steht, die folgenden in den Zeilen 3, 0, 2.

Aufsteigende Diagonalen (von links nach rechts) können mit (i + j) durchnummeriert werden, absteigende mit (i - j + N - 1);
die Nummern sind dann jeweils 0, . . . ($2N$ - 2).

Backtracking (3)

boolean damen (int *i*, boolean[][] S)
// Setzt auf Schachbrett S Dame in Spalte *i*, sofern möglich.

```
 1. int j ← -1;
 2. if i = N then return true;
 3. else
 4.         j ← findePosition(i, j, S);
 5.         while j ≠ -1 do
 6.                 S[ j ][ i ] ← true;
 7.                 Z[ j ] ← true; U[ i+j ] ← true; D[ i – j + N - 1] ← true;
 8.                 if damen(i+1, S) then return true;
 9.                 else
10.                         S[ j ][ i ] ← false;
11.                         Z[ j ] ← false; U[ i + j ] ← false; D[ i – j + N - 1] ← false;
12.                         j  ← findePosition( i, j, S );
13.         return false;
```

Backtracking (4)

int findePosition(int *i*, int *j*, boolean[][] *S*)
// Liefert nächste mögliche Zeile für Dame in Spalte *i*, beginnend nach Zeile *j*;
-1 wenn keine gefunden.

1. *j* ← *j* + 1;
2. **while** *j* < *N* and (Z[*j*] or U[*i* + *j*] or D[*i* – *j* + *N* - 1]) **do**
3. *j* ← *j* +1;
4. **if** *j* < N **then return** *j* **else** return -1;

Backtracking (5)

Kennzeichen von Problemen, bei denen Backtracking Anwendung findet:

- Lösung kann als Vektor $a[0], a[1], \ldots$ (möglicherweise unbestimmter, aber) endlicher Länge codiert werden.
- Jedes Element $a[i]$ ist ein Element einer endlichen Menge $A[i]$.
- Für jedes $a \in A[i]$ kann entschieden werden, ob es als Erweiterung einer Teillösung $a[0], \ldots, a[i-1]$ in Frage kommt.

Beispiele für die Anwendbarkeit von Backtracking:

- Labyrinth-Suche: Finde einen Weg vom Start zum Ziel.
- Hamiltonscher Zyklus in einem Graphen: Finde einen Weg durch einen Graphen, der jeden Knoten genau einmal berührt und dann zum Ausgangspunkt zurückkehrt.

Scan- oder Sweep-Verfahren

Merkmal: Eine räumliche Dimension wird in eine zeitliche Dimension verwandelt.

→ Ein statisches d-dimensionales Problem wird als ein dynamisches $(d - 1)$–dimensionales Problem behandelt. Dazu wird eine Linie (Ebene, . . .) entlang einer Dimension geführt und eine damit verbundene Datenstruktur konsistent bei gewissen Ereignissen geändert (Invariante).

Bsp: Maximum-Subarray Problem; statisches 1-dim. Problem wird als dynamisches 0-dim. Problem behandelt.

Weitere Problem-Beispiele:

- Dichtestes Paar einer Menge von n reellen Zahlen $\in O(n \log n)$
- Dichtestes Paar einer Menge von n Punkten in R^2 $\in O(n \log n)$
- k Schnittpunkte einer Menge von n Liniensegmenten in der Ebene $\in O((n + k) \log n)$

Scan-Line-Verfahren für das Maximum-Subarray Problem (1)

Das Maximum-Subarray Problem:

Gegeben: Folge *X* von *n* ganzen Zahlen im Array

Gesucht: Maximale Summe einer zusammenhängenden Teilfolge von *X*

Beispiel:

31 -41 59 26 -53 58 97 -93 -23 84

Das ist hier maximal $\overbrace{\qquad\qquad}^{\sum = 187}$

Frage: Wie findet man die maximale Teilfolge — möglichst effizient?

Scan-Line-Verfahren für das Maximum-Subarray Problem (2)

Scan-Line Ansatz

Scan-Line-Verfahren für das Maximum-Subarray Problem (3)

Der Scan-Line Ansatz:

```
scanMax = 0;
bisMax = 0;
für i von 1 bis n:
  scanMax += X[i]
  falls (scanMax < 0) scanMax = 0;
  bisMax = max(scanMax, bisMax);
max = bisMax;
```

$$T(n) \leq a \cdot n + b \in O(n)$$

→ Die Lösung des Maximum-Subarray Problems erfordert Aufwand $\Theta(n)$
(Jedes Element muss mindestens einmal betrachtet werden)