



Algorithmen und Datenstrukturen

Weitere Entwurfsmuster

Matthias Teschner
Graphische Datenverarbeitung
Institut für Informatik
Universität Freiburg

SS 09

Lernziele der Vorlesung



- Algorithmen
 - Sortieren, Suchen, Optimieren
- Datenstrukturen
 - Repräsentation von Daten
 - Listen, Stapel, Schlangen, Bäume
- Techniken zum Entwurf von Algorithmen
 - Algorithmenmuster
 - Greedy, Backtracking, Divide-and-Conquer
- Analyse von Algorithmen
 - Korrektheit, Effizienz

Überblick



- Teile und Herrsche
- Backtracking
- Greedy
- Dynamische Programmierung
- Vollständige Aufzählung
- Sweep-Verfahren

Prinzip



- Versuch-und-Irrtum-Prinzip (trial and error)
- Erreichte **Teillösung wird** schrittweise zur Gesamtlösung **ausgebaut**.
- Bewertungsfunktion prüft Gültigkeit eines Schrittes.
- Wenn klar ist, daß Teillösung nicht zur Gesamtlösung führt, **können Schritte rückgängig gemacht werden und Alternativen getestet werden**.
- Tiefensuche
- oft rekursiv implementiert

Prinzip



FindeLoesung (Stufe, Lösungsvektor)

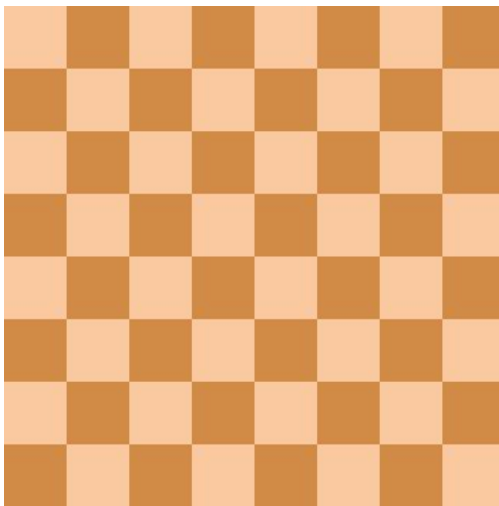
```
while (es existieren noch Teil-Lösungsschritte)
begin
  wähle einen Teil-Lösungsschritt;
  if (gültiger Teil-Lösungsschritt) then
    begin
      erweitere Lösungsvektor um
      gewählten Teil-Lösungsschritt;
      if (Lösungsvektor vollständig) then
        return true;
      else
        if (FindeLoesung(Stufe+1, Lösungsvektor)) then
          return true;
        else
          mache Teil-Lösungsschritt rückgängig;
        end
      end
    end
  return false;
```

www.wikipedia.de / Backtracking

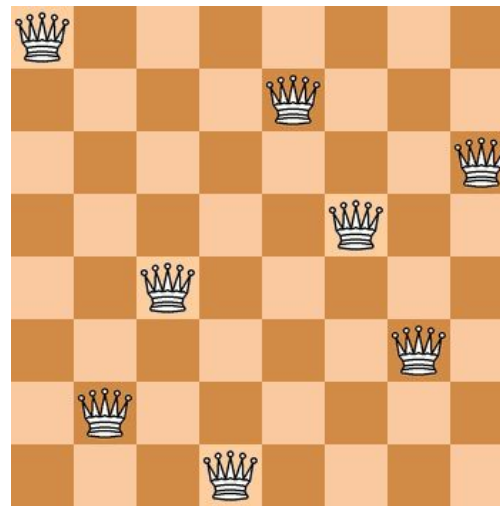
n-Damen-Problem



- Platziere n Damen, die sich nicht gegenseitig bedrohen
 - maximal eine Dame pro Zeile
 - maximal eine Dame pro Spalte
 - maximal eine Dame pro Diagonale
 - Es können maximal n Damen auf einem $n \times n$ - Feld platziert werden.



8x8 - Feld



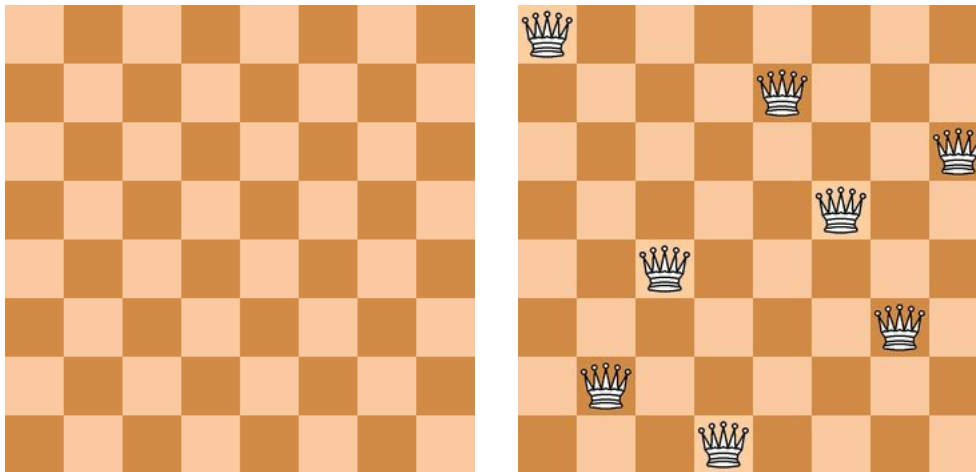
eine Lösung

www.wikipedia.de

Lösungsprinzip



- Backtracking
 - platziere schrittweise Damen, die sich nicht bedrohen
 - wenn keine weitere Dame platziert werden kann, mache den letzten Schritt rückgängig
- Effizienzsteigerung
 - nutze aus, dass pro Zeile nur eine Dame platziert werden kann



www.wikipedia.de

Implementierung



```
platziereDameInZeile (brett, zeile)
  if (zeile > n) then
  begin
    druckeErgebnis (brett);
    exit;
  end;
  else
  begin
    for spalte=1 to n do
    begin
      if (!dameIstBedrohtAuf (brett, zeile, spalte))
      begin
        setzeDame (brett, zeile, spalte);
        platziereDameInZeile (brett, zeile+1);
        entferneDame (brett, zeile, spalte); ← Hier kommen wir an,
        wenn eine Platzierung in
        zeile+1 nicht möglich war.
      end
    end
  return false; Platzierung in zeile nicht möglich
end
```

```
// Aufruf
initialisiere (brett);
platziereDameInZeile (brett, 1);
```

www.stefan-baur.de/cs.algo.8queens.html
(mit Java-Implementierung)

Illustration



- `platziereDameInZeile(brett, 1);`

x			

Illustration



- `platziereDameInZeile(brett, 1);`
- `platziereDameInZeile(brett, 2);`

x			
		x	

Illustration



- `platziereDameInZeile(brett, 1);`
- `platziereDameInZeile(brett, 2);`
- `platziereDameInZeile(brett, 3);`
 - `return false`

x			
		x	

Illustration



- `platziereDameInZeile(brett, 1);`
- `platziereDameInZeile(brett, 2);`
- `platziereDameInZeile(brett, 3);`
 - `return false`
- `entferneDame(brett, 2, 3);`
- `setzeDame(brett, 2, 4);`

x			
		x	x

Illustration



- `platziereDameInZeile(brett, 1);`
- `platziereDameInZeile(brett, 2);`
- `platziereDameInZeile(brett, 3);`
 - `return false`
- `entferneDame(brett, 2, 3);`
- `setzeDame(brett, 2, 4);`
- `platziereDameInZeile(brett, 3);`

x			
			x
	x		

Illustration



- `platziereDameInZeile(brett, 1);`
- `platziereDameInZeile(brett, 2);`
- `platziereDameInZeile(brett, 3);`
 - `return false`
- `entferneDame(brett, 2, 3);`
- `setzeDame(brett, 2, 4);`
- `platziereDameInZeile(brett, 3);`
- `platziereDameInZeile(brett, 4);`
 - `return false`

x			
			x
	x		

Illustration



- `platziereDameInZeile(brett, 1);`
- `platziereDameInZeile(brett, 2);`
- `platziereDameInZeile(brett, 3);`
 - `return false;`
- `entferneDame(brett, 2, 3);`
- `setzeDame(brett, 2, 4);`
- `platziereDameInZeile(brett, 3);`
- `platziereDameInZeile(brett, 4);`
 - `return false;`
- `entferneDame(brett, 3, 2);`
- `return false;`

x			
			x
	x		

Illustration



- `platziereDameInZeile(brett, 1);`
- `platziereDameInZeile(brett, 2);`
- `platziereDameInZeile(brett, 3);`
 - `return false;`
- `entferneDame(brett, 2, 3);`
- `setzeDame(brett, 2, 4);`
- `platziereDameInZeile(brett, 3);`
- `platziereDameInZeile(brett, 4);`
 - `return false;`
- `entferneDame(brett, 3, 2);`
- `return false;`
- `entferneDame(brett, 2, 4);`
- `return false;`

x			
			x

n-Damen-Problem

Lösungen



n	1	2	3	4	5	6	7	8	9	10
Lösungen	1	0	0	2	10	4	40	92	352	724

- $n = 25$, $\approx 2.200.000.000.000.000$ Lösungen

Backtracking - Laufzeit



- m Möglichkeiten, eine Teillösung zu erweitern
- n Elemente der Lösung
- $m + m^2 + \dots + m^n$ mögliche Zustände

m Möglichkeiten für den ersten Schritt der Teillösung

m^2 Möglichkeiten für die ersten zwei Schritte der Teillösung

m^n Möglichkeiten für alle n Schritte der Teillösung

- exponentielle Laufzeit

Überblick



- Teile und Herrsche
- Backtracking
- Greedy
- Dynamische Programmierung
- Vollständige Aufzählung
- Sweep-Verfahren

Prinzip



- Wähle **schrittweise** den Folgezustand, der aktuell (lokal) den **größten Zugewinn** verspricht.
- Eine **Bewertungsfunktion** muss existieren, um aus möglichen Folgezuständen den optimalen zu ermitteln.
- **Schritte werden nicht rückgängig gemacht.**
- einfach zu realisieren
- löst viele Probleme nicht optimal (lokales Optimum statt globales Optimum)
- löst eventuell Probleme nicht in optimaler Zahl von Schritten, z. B. Gradientenabstiegsverfahren im Vergleich zu konjugierten Gradienten

Münzwechsel-Problem



- Eingabe: Menge von Münzwerten, Betrag W
- Ausgabe: Folge von Münzwerten mit Summe W mit minimaler Länge

- **Algorithmus:**

```
Betrag  $W \geq 0$ ;
```

```
Ergebnismenge =  $\emptyset$ ;
```

```
Wähle  $B$  = Wert der größten Münze;
```

```
while  $W > 0$  do
```

```
  begin
```

```
    while  $B \leq W$  do
```

```
      begin
```

```
         $W = W - B$ ;
```

```
        füge  $B$  zur Ergebnismenge hinzu;
```

```
      end;
```

```
       $B$  = Münze kleiner als  $B$  (nächstkleinere Münze);
```

```
    end;
```

```
  return Ergebnismenge;
```

Beispiel



- Münzen: 1, 5, 10, 20, 50, 100, 200
- Wert 147
- $147 > 0$, $200 > 147$ → nächstkleinere Münze 100
- $147 > 0$, $100 \leq 147$ → Wert=47, Ergebnismenge = {100}
- $47 > 0$, $100 > 47$ → nächstkleinere Münze 50
- ...
- Ergebnismenge = {100, 20, 20, 5, 1, 1}

- $147 \rightarrow 47 \rightarrow 27 \rightarrow 7 \rightarrow 2 \rightarrow 1 \rightarrow 0$
 100 20 20 5 1 1

Optimalität



- ist bei Greedy aufgrund der Lokalität der Entscheidung nicht immer gegeben
- Beispiel: Münzwerte: 41, 20, 1, Wert $W=60$
Greedy-Lösung: $60 = 41 + 19 \cdot 1$ (20 Münzen)
optimale Lösung: $60 = 3 \cdot 20$ (3 Münzen)
- Verhältnis von Greedy- zu optimaler Lösung kann beliebig schlecht werden.
- Beispiel: Münzwerte: 1, B , $2 \cdot B + 1$, $W = 3 \cdot B$
Greedy Lösung: $3 \cdot B = 1 \cdot (2 \cdot B + 1) + (B-1) \cdot 1$ (B Münzen)
optimale Lösung: $3 \cdot B = 3 \cdot B$ (3 Münzen)
- Verhältnis wird beliebig schlecht für wachsende B

Überblick



- Teile und Herrsche
- Backtracking
- Greedy
- **Dynamische Programmierung**
- Vollständige Aufzählung
- Sweep-Verfahren

Prinzip



- berechne optimale Lösungen kleiner Teilprobleme direkt
- setze Lösungen für nächstgrößere Teilprobleme aus bekannten Lösungen für kleinere Probleme zusammen
- **berechnete Teilergebnisse werden gespeichert (tabelliert) und bei Bedarf wiederverwendet**
- wird bei Problemen angewendet, die aus **vielen gleichartigen Teilproblemen** bestehen
- **Rekursion wird** durch Wiederverwendung von bereits berechneten Teilproblemen **vermieden** (Rekursion berechnet eventuell gleiche Teilprobleme mehrmals.)

Fibonacci-Zahlen



- $f(0) = 0$
 $f(1) = 1$
 $f(n) = f(n-1) + f(n-2) \quad n \geq 2$
- oder auch
 $f(1) = 1$
 $f(2) = 1$
 $f(n) = f(n-1) + f(n-2) \quad n \geq 3$
- nach Leonardo Fibonacci benannt (1180 - 1241)

Rekursive Implementierung

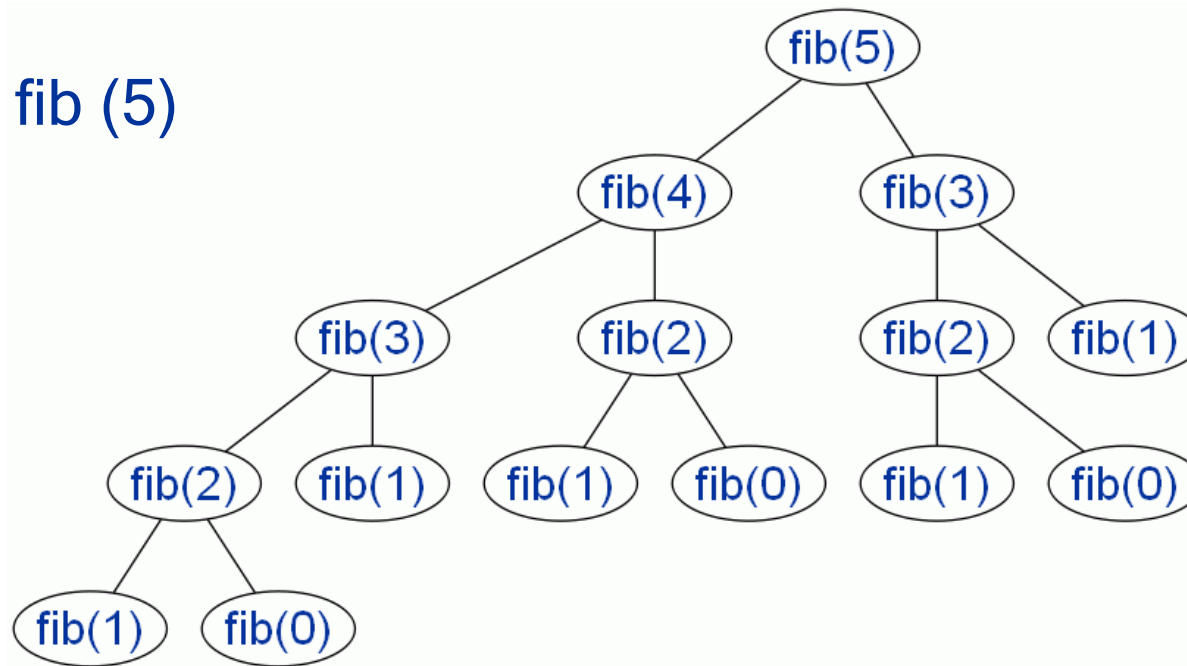


```
procedure fib( n:integer ) : integer
  if (n== 0) or (n==1) then
    return n;
  else
    return fib(n-1) + fib(n-2);
```

Illustration - Aufrufbaum



- fib (5)



- Laufzeit: $T(n) \approx 1.618^n$

$$T(n) \leq T(n-1) + T(n-2) + c, \quad T(0) = T(1) \leq c$$

$$T(n) = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right) \quad \text{Beweis durch vollständige Induktion}$$

Problem



- Rekursiver Ansatz berechnet mehrfach identische Teilprobleme
- Speicherung (Tabellierung) von Teillösungen wäre sinnvoll

Dynamisches Programmieren



- Problem P: $f(n) = f(n-1) + f(n-2)$
- Bestimmung der Menge T, die alle Teilprobleme von P enthält, welche für die Gesamtlösung bekannt sein müssen $\{ f(n-1), f(n-2), \dots, f(1), f(0) \}$
- Bestimmung einer Reihenfolge T_1, \dots, T_k der Teilprobleme in T, sodass für die Lösung von T_i nur auf bekannte T_j mit $j < i$ zurückgegriffen werden muss $f(0), f(1), \dots, f(n-2), f(n-1)$
- Sukzessive Berechnung und Speicherung (Tabellierung) der Lösungen zu T_1, T_2, \dots, T_k, P $f(0), f(1), \dots, f(n-2), f(n-1), f(n)$

Fibonacci-Zahlen

Iterativer Ansatz



- **procedure** fib (n:integer) : integer
 f(0) := 0;
 f(1) := 1;
 for k:=2 **to** n **do**
 f(k) := f(k-1) + f(k-2);
 return f(n);
- lineare Laufzeit
- benötigt ein Feld f der Größe n zur Speicherung der Teillösungen f(k)
- linearer Speicherbedarf $\in O(n)$

Fibonacci-Zahlen

Alternativer Ansatz



- **procedure** fib (n:integer) : integer
 f_vorletzte := 0; f_letzte := 1;
 for k:=2 **to** n **do**
 begin
 f_aktuell := f_letzte + f_vorletzte;
 f_vorletzte := f_letzte;
 f_letzte := f_aktuell;
 end
 if (n ≤ 1) **then**
 return n;
 else
 return f_aktuell;
- lineare Laufzeit
- konstanter Platzbedarf (unabhängig von n)

Fibonacci-Zahlen

Effiziente rekursive Implementierung



```
■ procedure fib ( n:integer) : integer
    f(0) := 0;
    f(1) := 1;
    for k:=2 to n do
        f(k) := -1;
    return lookupFib (n);
```

Feld für Teillösungen
{ 0, 1, -1, -1, ..., -1 }

```
■ procedure lookupFib ( k:integer) : integer
    if ( f(k) != -1 ) then
        return f(k);
    else
    begin
        f(k) := lookupFib(k-1) + lookupFib(k-2);
        return f(k);
    end;
```

Teillösung existiert bereits.

Teillösung muss
berechnet werden.

Jede Teillösung wird
nur einmal berechnet !

Überblick



- Teile und Herrsche
- Backtracking
- Greedy
- Dynamische Programmierung
- **Vollständige Aufzählung**
- Sweep-Verfahren

Prinzip



- Systematische Erzeugung aller Lösungskandidaten
- Auswahl des optimalen Lösungskandidaten anhand einer Zielfunktion (Kostenfunktion)
- meist schlechte Effizienz

Problem des Handlungsreisenden



- Eingabe: Städte 1, ..., n mit Entfernungen $d_{ij} > 0$ für $i \neq j$
- Ausgabe: Reihenfolge der Städte mit minimaler Gesamtstrecke



Optimaler Reiseweg durch 15 Städte
(43.589.145.600 mögliche Reisewege)

www.wikipedia.de – Problem
des Handlungsreisenden

Implementierung



- **procedure** shortestPath (n:integer) : integer
 - initialize(d(n,n)); Matrix mit allen Entfernungen
 - initializePermutations (P, n); Menge aller möglichen Reihenfolgen.
(0, 2, 1, 3, 4) ist ein Element dieser Menge für n=5.
 - minDist = ∞;
 - forall** curOrder in P **do** curOrder ist eine mögliche Reihenfolge.
 - begin**
 - curDist :=0;
 - for** i:=0 **to** n-2 **do**
 - curDist := curDist +
d (curOrder(i), curOrder(i+1));
 - curDist := curDist +
d (curOrder(n-1), curOrder(0));
 - if** (curDist<minDist) minDist=curDist;
 - end;**
 - return** minDist;

Überblick



- Teile und Herrsche
- Backtracking
- Greedy
- Dynamische Programmierung
- Vollständige Aufzählung
- Sweep-Verfahren

Prinzip



- Raum mit allen Elementen des Problems wird ausgefegt bzw. abgetastet (sweep).
- Eine Status-Struktur wird während des Ab tastens aller Elemente mitgeführt und aktualisiert (Sweep-Status-Struktur).
- In der Regel wird das Problem mit einer (n-1)-dimensionalen Struktur abgetastet.
 - 1D-Feld wird durch Punkt abgetastet (ausgefegt).
 - 2D-Ebene wird durch 1D-Gerade abgetastet.
 - 3D-Raum wird durch 2D-Ebene abgetastet.
- Wenn alle Elemente des Problems besucht (abgetastet) wurden, kann aus der Sweep-Status-Struktur die Lösung ermittelt werden.

1D-Problem - Maximum



- ```
procedure max(q:array of integer):integer
 maxSoFar := q[0];
 for i:=0 to q.length()-1 do
 if (maxSoFar < q[i]) then
 maxSoFar := q[i];
 return maxSoFar;
```
- 1D-Feld q wird abgetastet (jedes Element q [ i ] wird einmal besucht).
- 0D-Struktur (maxSoFar) wird mitgeführt und aktualisiert, um daraus nach dem Abtasten das Maximum zu ermitteln.

# 1D-Problem

## Maximale Teilsumme



- Eingabe: Feld  $q$  von  $n$  ganzen Zahlen
- Ausgabe: Maximale positive Teilsumme aufeinanderfolgender Elemente von  $q$ . Existiert keine positive Teilfolge, ist das Ergebnis 0

J. L. Bentley. *Programming Pearls*. Addison-Wesley, Reading, MA, 1985.

- ```
procedure maxSubArray(q:array of integer):integer
  maxSoFar := 0; maxEndingHere := 0;
  for i:=0 to q.length()-1 do
  begin
    maxEndingHere := max (0,maxEndingHere+q[i]);
    maxSoFar := max (maxSoFar,maxEndingHere);
  end
  return maxSoFar;
```

0D-Status-Struktur
Abtastung aller Elemente
des 1D-Problems

O(n)
schneller als Teile-
und-Herrsche-Ansatz

Illustration



q	2	-3	5	-2	-1	8	-4	1	-9	3	-1	9	-1	2
maxEndingHere	2	0	5	3	2	10	6	7	0	3	2	11	10	12
maxSoFar	2	2	5	5	5	10	10	10	10	10	10	11	11	12

- Funktioniert nur, wenn man negative Teilsummen als Ergebnis ausschließt. Dadurch kann man negative Werte von maxEndingHere ausschließen und auf 0 setzen.
- Negative Teilsummen am Anfang und am Ende eines potentiellen Ergebnisses können dann nicht zum Ergebnis beitragen.

Dichtestes Paar einer Menge von Zahlen (1D)



- Eingabe: Feld q von n ganzen Zahlen
- Ausgabe: $\min |q[i] - q[j]|$ mit $0 \leq i, j \leq n-1$

- naive Lösung ist in $O(n^2)$

- Idee:
 - Betrachte nur Paare, die der Größe nach benachbart sind.
 - Wenn die Elemente in q sortiert sind, kann eine einfache Sweep-Lösung realisiert werden.

Dichtestes Paar einer Menge von Zahlen (1D)



■ `procedure minDiff (q:array of integer) : integer`

`q := sort (q);`

sortiere Elemente in q
 $O(n \log n)$
(lernen wir noch kennen)

`minDiffSoFar := q[1] - q[0];`

Sweep-Status

`for i:=2 to q.length()-1 do`

Taste das Problem ab

`if minDiffSoFar > q[i] - q[i-1] then`

`minDiffSoFar = q[i] - q[i-1];`

`return minDiffSoFar;`

$O(n \log n)$

Sweep ist in $O(n)$. Laufzeit ist durch das Sortieren dominiert.

Illustration



q	32	-2	31	0	30	3	27	7	23	8	17	10	11	17
q sortiert	-2	0	3	7	8	10	11	17	17	23	27	30	31	32
minDiffSoFar			2	2	1	1	1	1	0	0	0	0	0	0

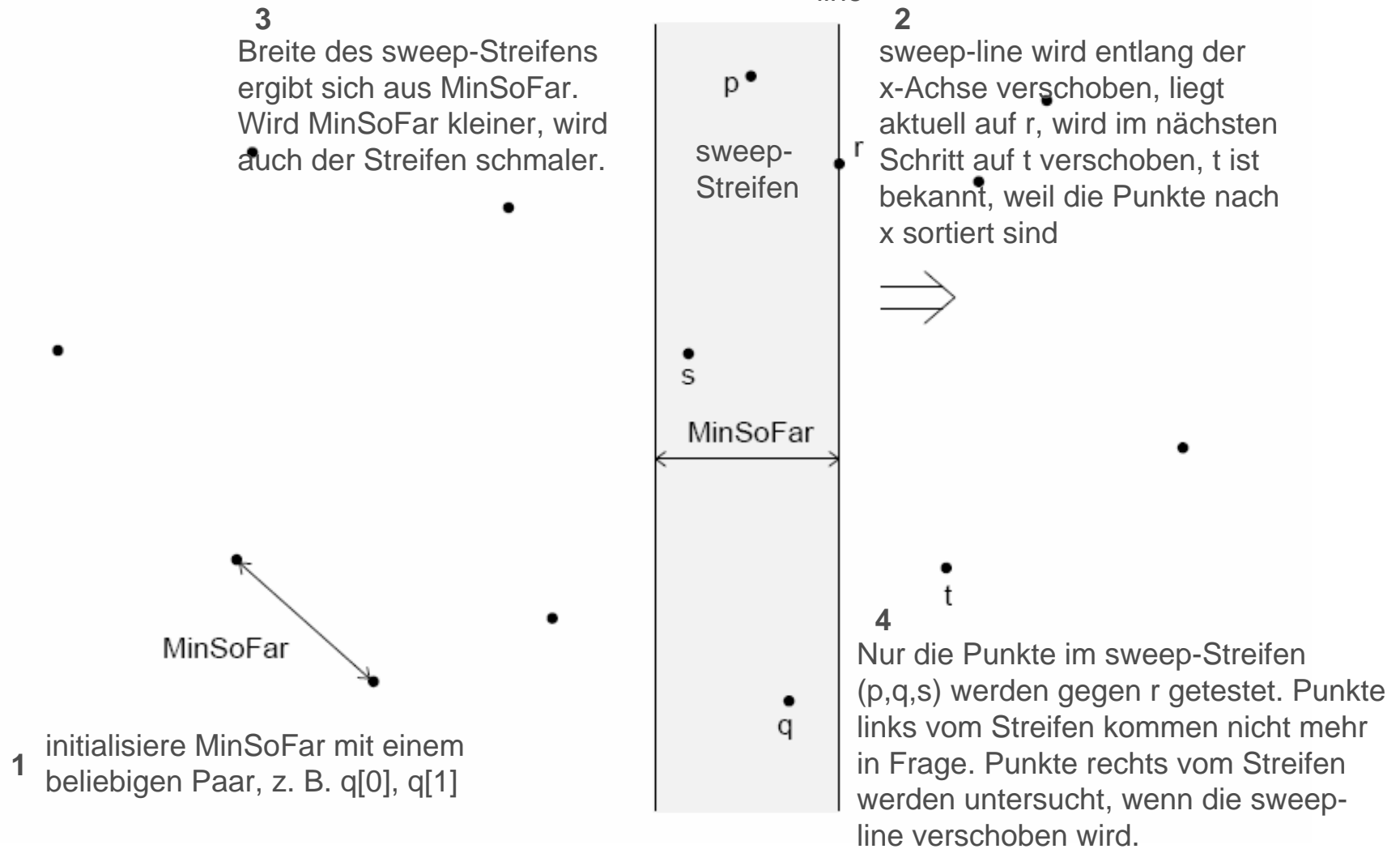
Dichtestes Paar einer Menge von 2D-Punkten



- Eingabe: Menge q von n 2D-Punkten
- Ausgabe: $\min (\text{dist}(q [i], q [j]))$ mit $0 \leq i, j \leq n-1$
und $\text{dist}(a,b) = ((a.x - b.x)^2 + (a.y - b.y)^2)^{1/2}$

Jeder Punkt hat eine reelwertige x- und ein y-Komponente
- naive Lösung ist in $O(n^2)$
- Idee:
 - Verfolge eine 1D-Gerade, die über alle 2D-Punkte bewegt wird.
 - Um den jeweils nächsten zu besuchenden Punkt zu ermitteln, wird die Punktmenge sortiert, beispielsweise in Bezug auf die x-Komponente

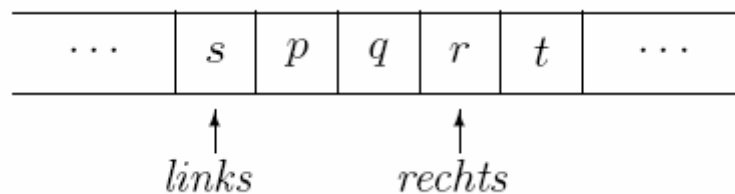
Illustration



Sweep-Status



- enthält den bisher kleinsten Abstand zweier Punkte MinSoFar sowie alle Punkte im sweep-Streifen
- Aktualisierung des Sweep-Status:
 - 1) Wenn der linke Streifenrand über einen Punkt p wandert, dann entferne den Punkt p aus dem Sweep-Status
 - 2) Wenn der rechte Streifenrand auf den nächsten Punkt p wandert, dann füge den Punkt p in den Sweep-Status ein, dann setze den linken Streifenrand auf $p.x - \text{MinSoFar}$ und entferne alle Punkte gemäß Regel 1), dann teste alle Punkte im Streifen gegen p auf minimale Distanz und aktualisiere ggf. MinSoFar und den linken Streifenrand.



"links" und "rechts" werden nach rechts über das Feld geschoben, um alle Punkte abzuarbeiten. Nur Punkte zwischen "links" und "rechts" werden gegen den Punkt "rechts" getestet.

Pseudocode



```
procedure minDist (q:array of point2D) : real
```

```
  q := sortWithRespectToX (q);           sortieren O (n log n)
  status := ∅; status.add(q[0]); status.add(q[1]);
  MinSoFar := dist(q[0],q[1]);           initialisiere sweep-Status
  links := 0;                            MinSoFar und Menge {q[0], q[1]}
  rechts := 2;

  while (rechts ≤ q.length()-1) do       sweep – rechter Rand
  begin                                  läuft über alle Punkte
    if (q[links].x + MinSoFar ≤ q[rechts].x) then
    begin
      status.remove(q[links]);           Entferne Punkte aus dem Status, die links vom
      links := links + 1;                Streifen liegen. Der "else"-Teil wird erst durchlaufen,
    end                                  wenn hier alle entsprechenden Punkte entfernt wurden.
    else
    begin
      MinSoFar := minDistBand(status, q[rechts], MinSoFar);
      status.add(q[rechts]);             Verschiebe den rechten Rand. minDistBand testet den
      rechts := rechts + 1;              neuen rechten Randpunkt gegen alle Punkte im Streifen
    end                                  und aktualisiert ggf. MinSoFar.
  end
  return MinSoFar;                       insgesamt O (n log n)
                                          ohne Beweis
```

Zusammenfassung



- Teile und Herrsche
 - zerlege ein Problem in kleinere Teilprobleme, löse diese
 - je nach Größe - rekursiv oder direkt, setze die Teillösungen zusammen
- Backtracking
 - erweitere Teillösungen schrittweise, Bewertungsfunktion prüft Gültigkeit einer Teillösung, Teilschritte können rückgängig gemacht werden
- Greedy
 - wähle Schritt mit dem momentan größten Zugewinn, Bewertungsfunktion ermittelt beste Alternative in einem Schritt, gewählte Teilschritte werden nicht rückgängig gemacht

Zusammenfassung



- **Dynamische Programmierung**
 - berechne, speichere und wiederverwende Teillösungen bei Problemen, die aus vielen gleichartigen Teilproblemen bestehen
- **Vollständige Aufzählung**
 - generiere und bewerte alle Lösungskandidaten mit einer Zielfunktion, um die beste Lösung zu ermitteln
- **Sweep-Verfahren**
 - taste alle Elemente des Problems mit einer Status-Datenstruktur ab, die Status-Datenstruktur wird aktualisiert und enthält das Ergebnis

Nächstes Thema



- Datenstrukturen
 - Listen