LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR MATHEMATIK, INFORMATIK UND STATISTIK
INSTITUT FÜR INFORMATIK
LEHRSTUHL FÜR DATENBANKSYSTEME UND DATA MINING

# Kapitel 6 - Algorithmische Paradigmen

Backtracking
Divide and Conquer
Sweeplinetechnik
Heuristiken
Greedy Algorithmen
Dynamische Programmierung
Branch and Bound
Online-Algorithmen

DBS

# Paradigma: Backtracking

**Idee:**

- Lösung eines Problems durch Versuch und Irrtum (trial and error) .
- Schrittweises „Herantasten“ an die Gesamtlösung.
- Kann eine Teillösung nicht zu einer Gesamtlösung erweitert werden:
  - Letzten Schritt rückgängig machen.
  - Weitere, alternative Schritte probieren.

**Voraussetzung:**

- Die Lösung setzt sich aus „Komponenten zusammen“
- Für jede Komponente gibt es mehrere Wahlmöglichkeiten
- Teillösungen können getestet werden.

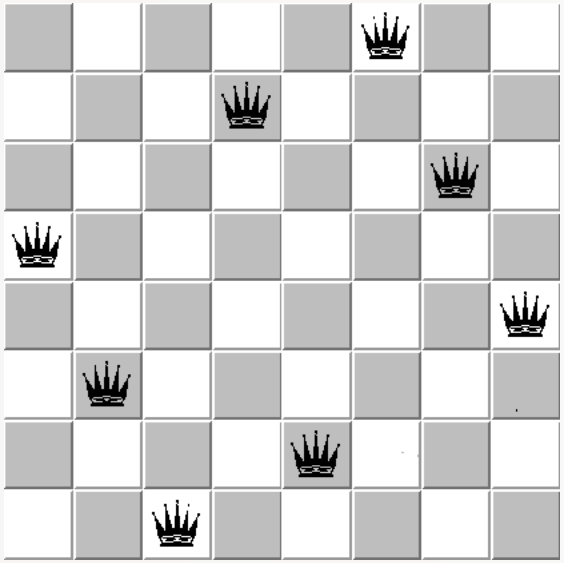
**Geeignete Probleme:**

- *Damenproblem*
- Sudoku
- Solitär-Brettspiel
- Wegsuche von A nach B

# Damenproblem

## Problem:

Auf einem Schachfeld der Größe n * n sollen n Damen so positioniert werden, dass sie sich gegenseitig nicht schlagen können.



Für *n = 8 gibt es:*

- $\binom{64}{8} = 4\,426\,165\,368$ Positionierungen.
- 92 Lösungen.
- 12 nicht unter Symmetrie äquivalente Lösungen

**Optimierte Suche:**

In jeder Spalte und in jeder Zeile kann jeweils nur eine Dame stehen.
Somit lassen sich die Positionierungen auf *8! = 40 320* reduzieren.

# n-Damenproblem

Für n = 26 wurden im Jahre 2009 alle Lösungen berechnet:

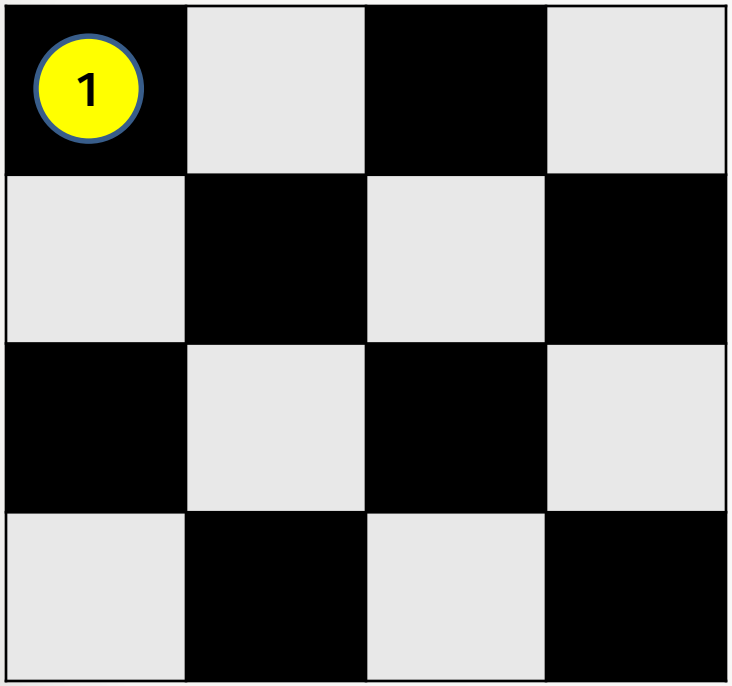
- Lösungen insgesamt 22.317.699.616.364.044
- Eindeutige Lösungen 2.789.712.466.510.289

Für n = 27 wurden im Jahre 2016 alle Lösungen berechnet:

- Lösungen insgesamt 234.907.967.154.122.528
- Eindeutige Lösungen 29.363.495.934.315.694

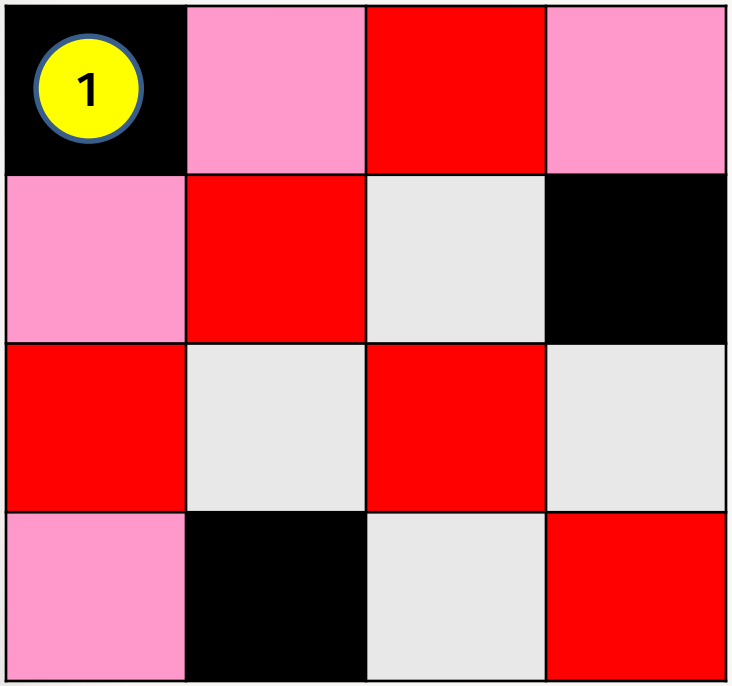
- Die Berechnung aller Lösungen für n = 28 steht noch aus.

# 4 Damenproblem



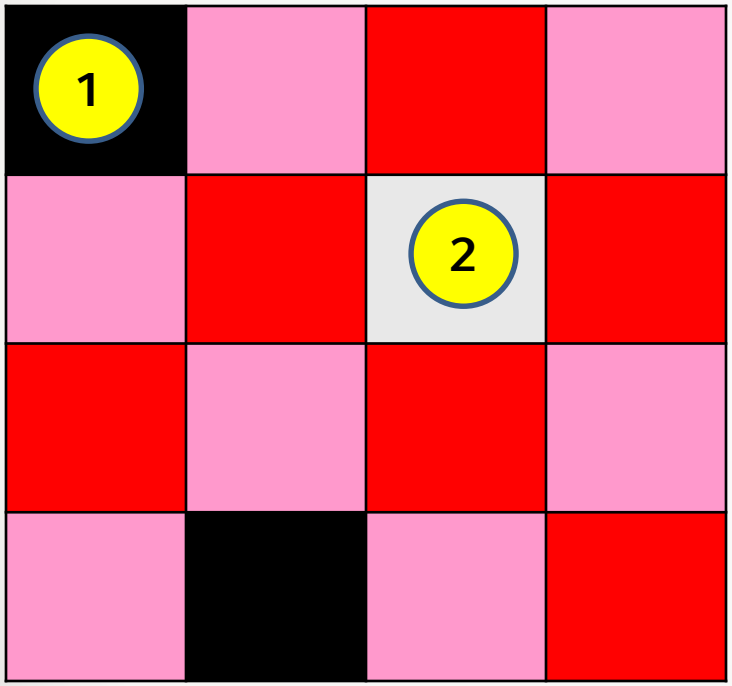

- Erste Dame auf das erste Feld setzen

# 4 Damenproblem



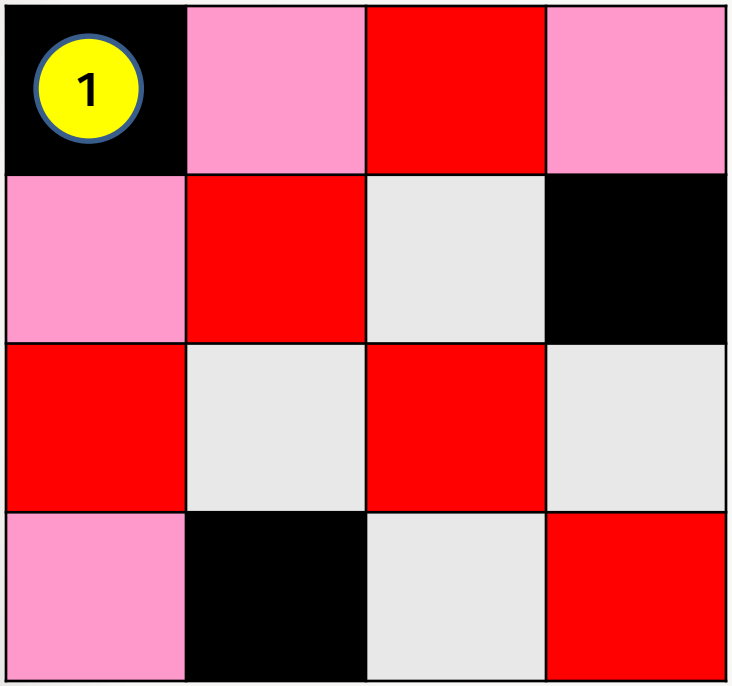

- Erste Dame auf das erste Feld setzen
- Markierte Felder sind bedroht

# 4 Damenproblem



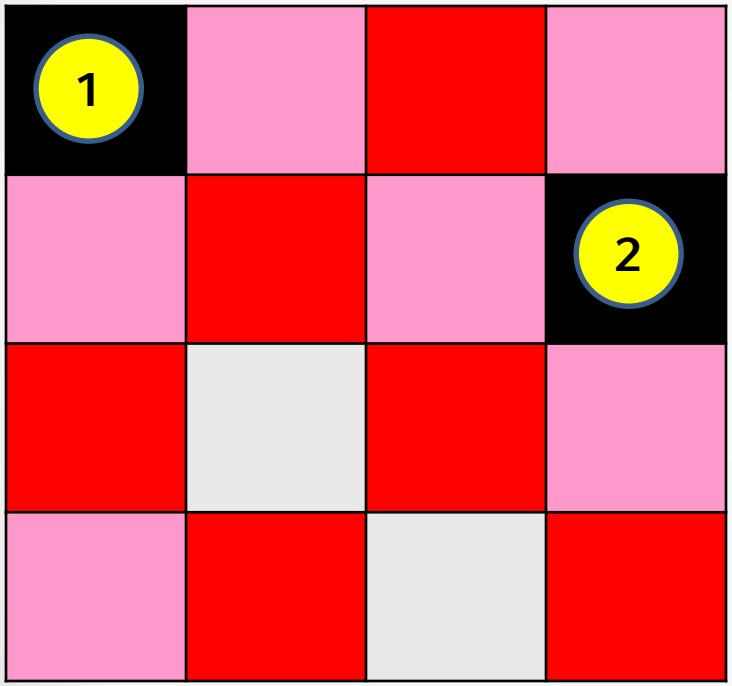

- Erste Dame auf das erste Feld setzen
- Markierte Felder sind bedroht
- Zweite Dame auf erstes freies Feld setzen
- Dritte Reihe komplett bedroht!
- Keine Lösung möglich.
- Backtracking ist nötig

# 4 Damenproblem



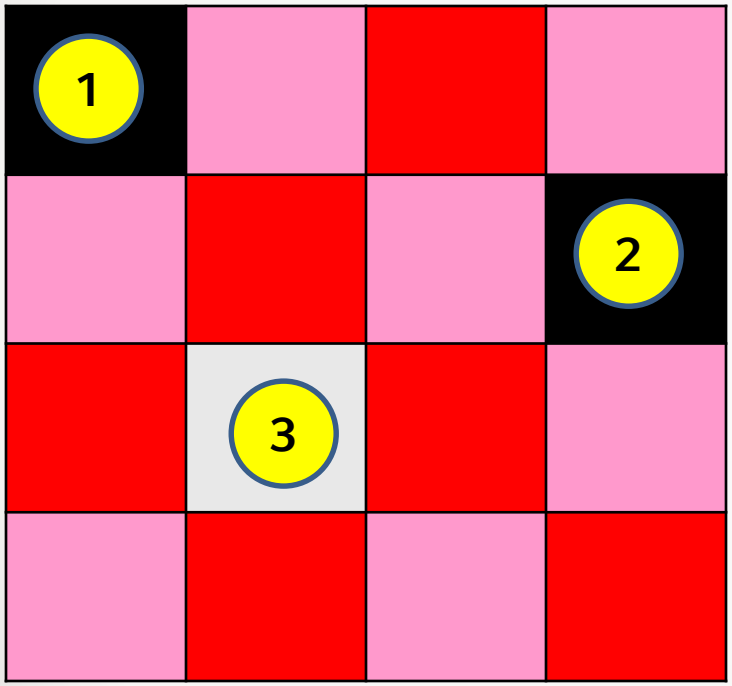

- …
- Backtracking ist nötig
- Dame zwei wieder entfernen

# 4 Damenproblem



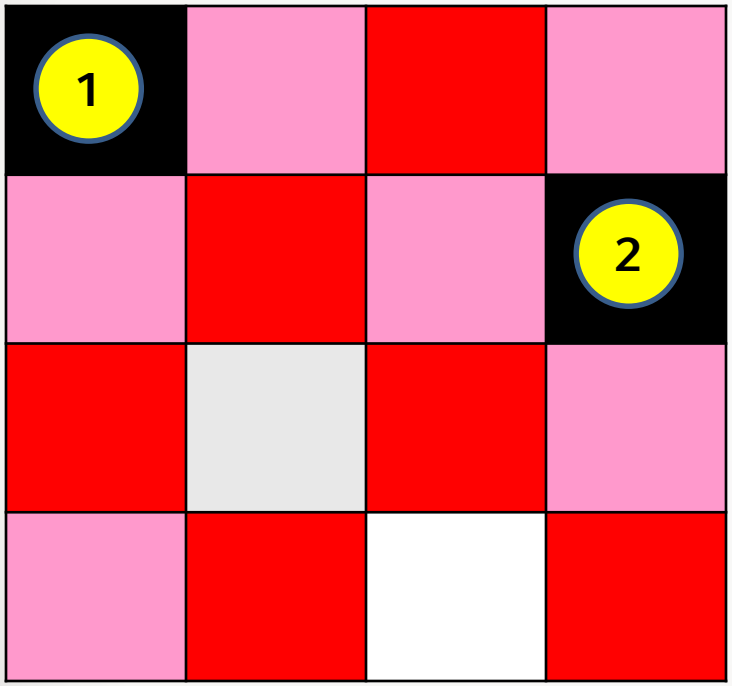

- …
- Backtracking ist nötig
- Dame zwei wieder entfernen
- Zweite Dame auf das nächste freie Feld

# 4 Damenproblem



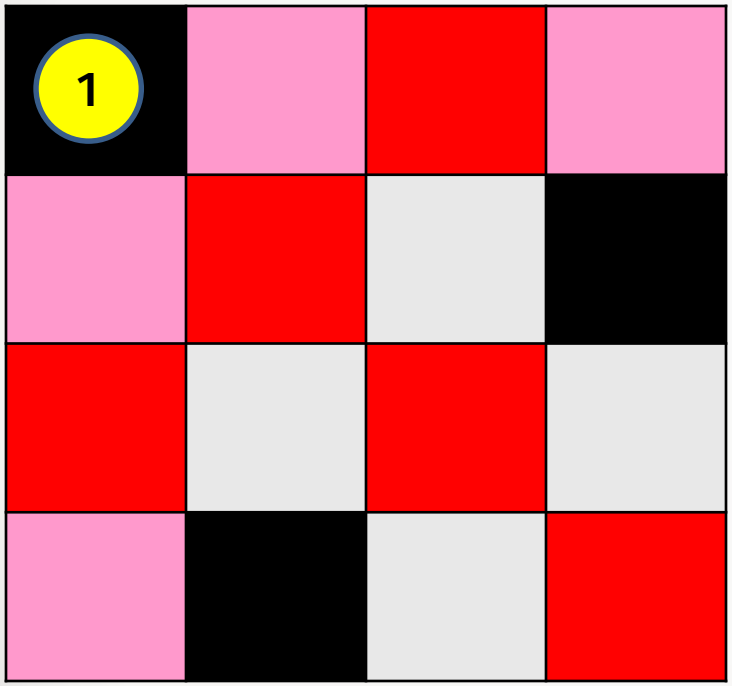

- …
- Backtracking nötig
- Dame zwei wieder entfernen
- Zweite Dame auf das nächste freie Feld
- Dritte Dame auf das erste freie Feld
- Kein freies Feld für vierte Dame!
- Backtracking …

# 4 Damenproblem



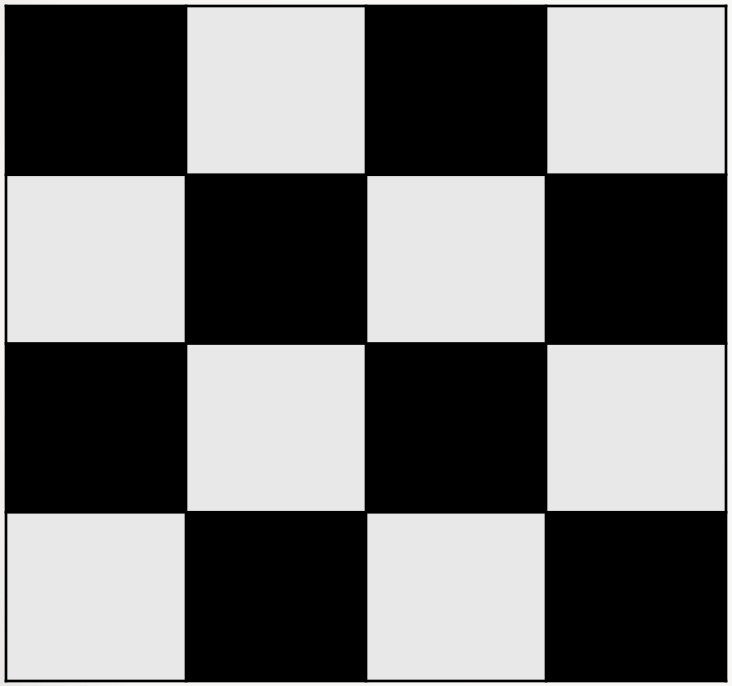

- …
- Kein Weiteres Feld in der dritten Reihe frei
- Weiteres Backtracking

# 4 Damenproblem



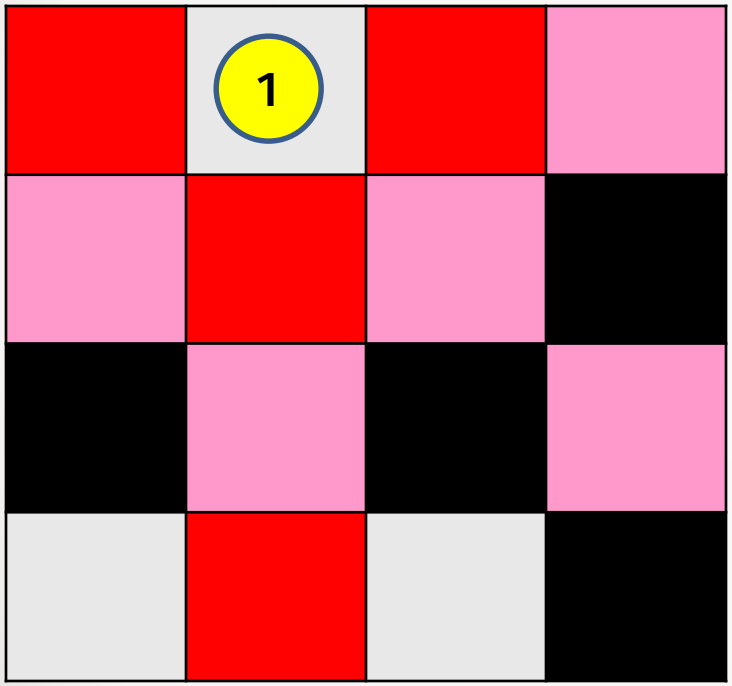

- …
- Kein Weiteres Feld in der dritten Reihe frei
- Weiteres Backtracking
- Kein weiteres Feld in der zweiten Reihe frei
- Weiteres Backtracking …

# 4 Damenproblem



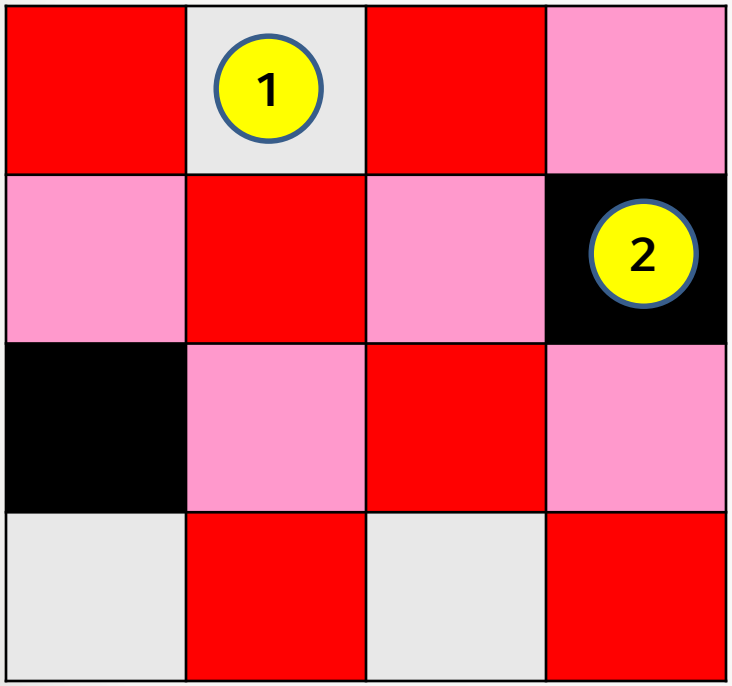
- …
- Kein Weiteres Feld in der dritten Reihe frei
- Weiteres Backtracking
- Kein weiteres Feld in der zweiten Reihe frei
- Weiteres Backtracking …

# 4 Damenproblem



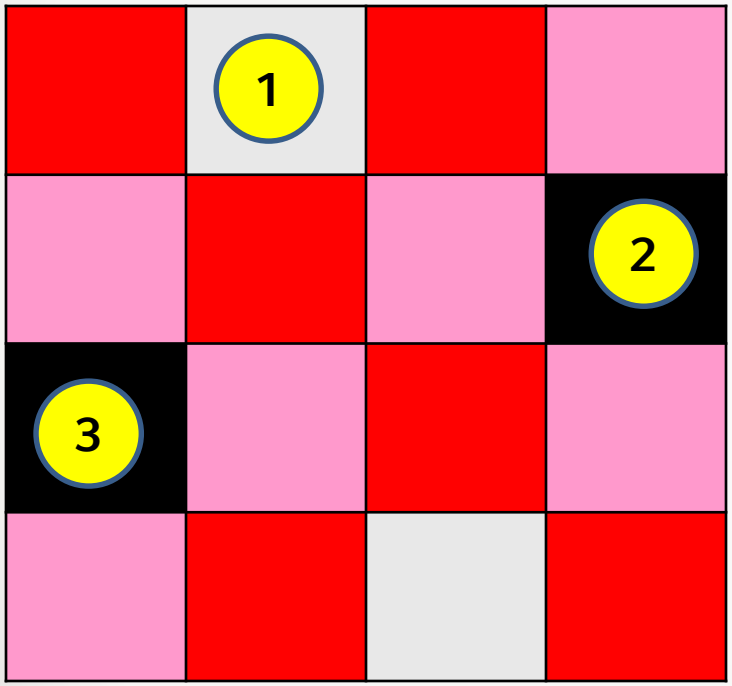

- …
- Erste Dame auf zweites Feld setzen

# 4 Damenproblem



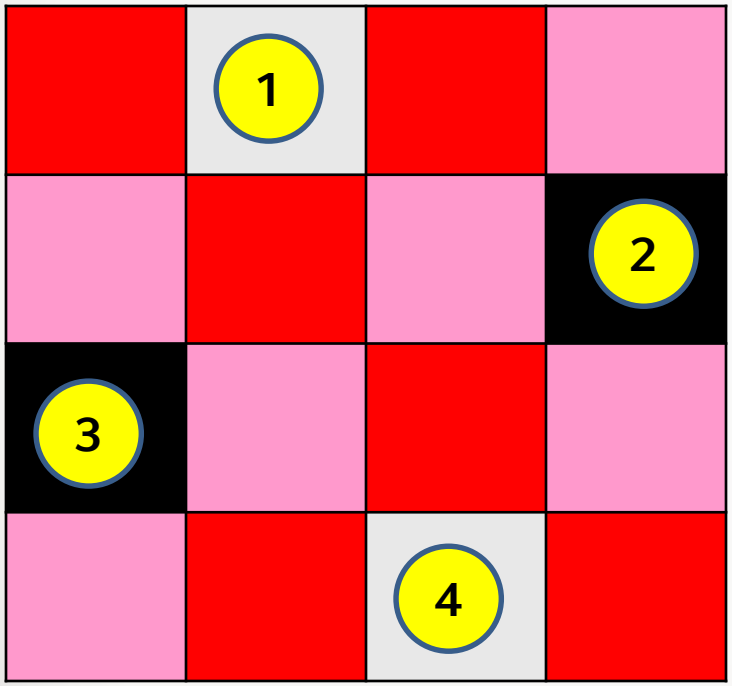

- …
- Erste Dame auf zweites Feld setzen
- Zweite Dame auf erstes freies Feld setzen

# 4 Damenproblem




- …
- Erste Dame auf zweites Feld setzen
- Zweite Dame auf erstes freies Feld setzen
- Dritte Dame auf erstes freies Feld setzen

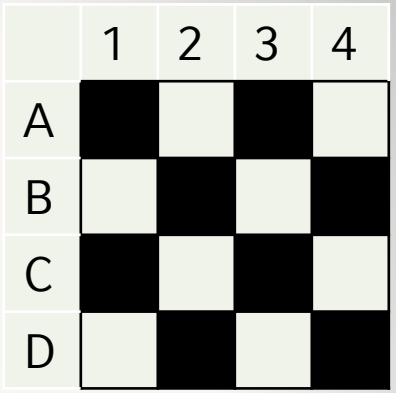
# 4 Damenproblem

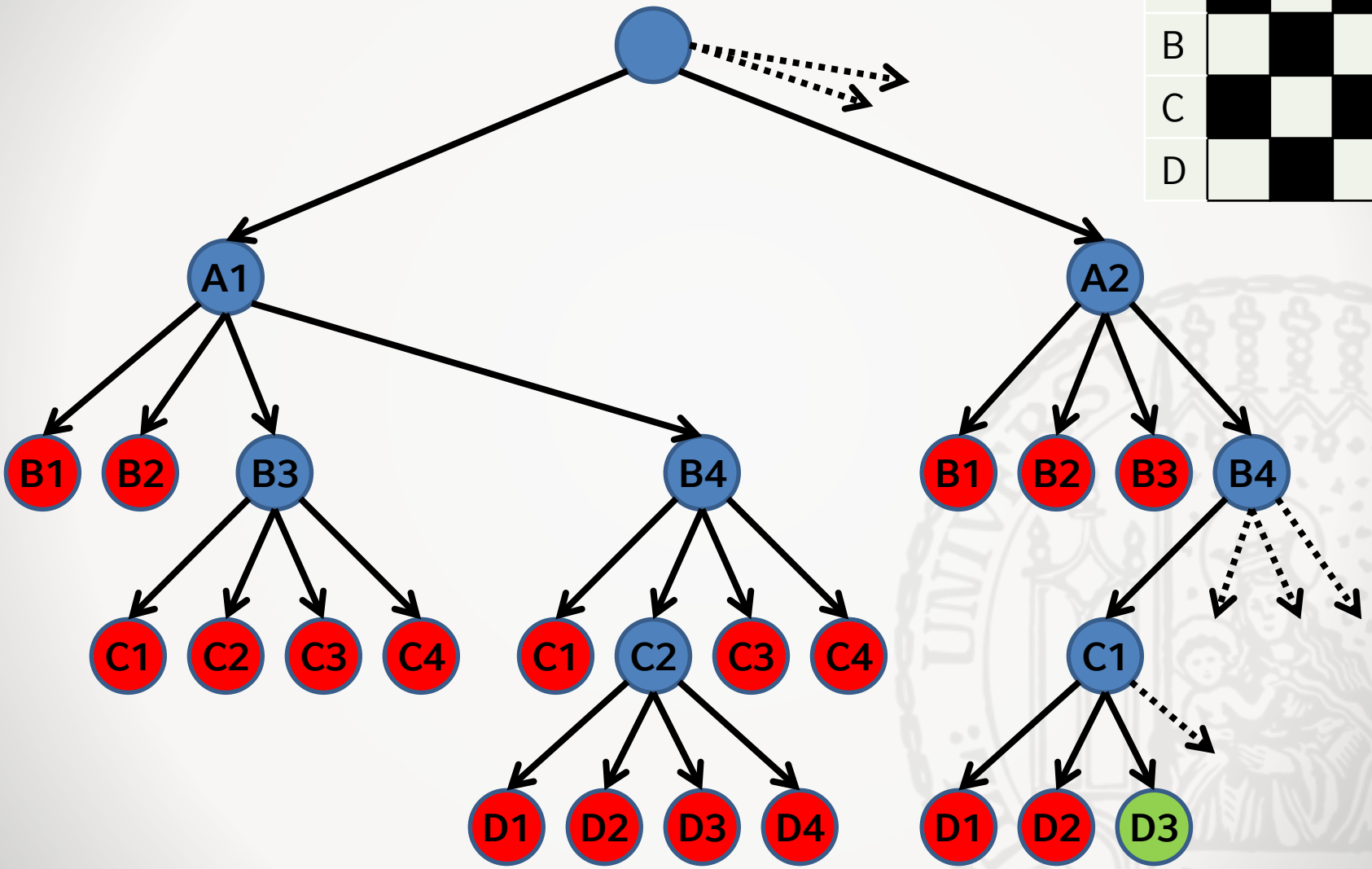



- …
- Erste Dame auf zweites Feld setzen
- Zweite Dame auf erstes freies Feld setzen
- Dritte Dame auf erstes freies Feld setzen
- Vierte Dame auf erstes freies Feld setzen

- Es wurde eine Lösung gefunden!

# 4 Damenproblem: Suchbaum



# n- Damenproblem in Java

```
public static boolean damen( boolean[][] feld, int reihe){

  if(bedroht (feld))                // testen ob Teillösung ungültig
    return false;
  if(reihe == feld.length)          // testen ob Gesamtlösung erreicht
    return true;

  for(int i = 0; i < feld[reihe].length; i++){

      feld[reihe][i] = true;         // versuchen …
      if(damen(feld, reihe+1))
        return true;
      else
        feld[reihe][i] = false;     // backtracking

  }
  return false;
}
```

Aufruf mit:
damen(feld, 0)

# Backtracking- Allgemeiner Algorithmus

BackTracking ( Stufe, Lösungsvektor )

**falls** Teillösung ungültig gib **falsch** zurück

**falls** alle Komponenten gesetzt sind gib **richtig** zurück

**solange** es auf der aktuellen Stufe noch Wahlmöglichkeiten gibt:

wähle einen neuen Teil-Lösungsschritt

setze die entsprechende Komponente

**falls BackTracking (** Stufe+1, Lösungsvektor **)** gib **richtig** zurück

**sonst** mache Wahl rückgängig

*Wenn es keinen neuen Teil-Lösungsschritt mehr gibt:* ***Keine Lösung!***

# Paradigma: Divide-and-Conquer

**Idee:**

Große Probleme in kleinere zerteilen, die leichter zu lösen sind und die Gesamtlösung ermöglichen

Drei Schritte:

| | |
|---|---|
| **Divide:** | Das Problem wird in mehrere Unterprobleme aufgeteilt |
| **Conquer:** | Die (kleineren) Unterprobleme werden rekursiv gelöst.<br>Sind die Unterprobleme klein genug werden sie direkt gelöst. |
| **Merge:** | Die Lösungen der Teilprobleme werden zu einer Gesamtlösung kombiniert. |

# Divide-and-Conquer Algorithmen

## Merge-Sort:

**Divide:** Die zu sortierende Liste der Länge $n$ wird in zwei zu sortierende Listen der Länge $n/2$ aufgeteilt.

**Conquer:** Die Listen werden weiterhin in kleinere Listen aufgeteilt. Einelementige Listen sind automatisch sortiert.

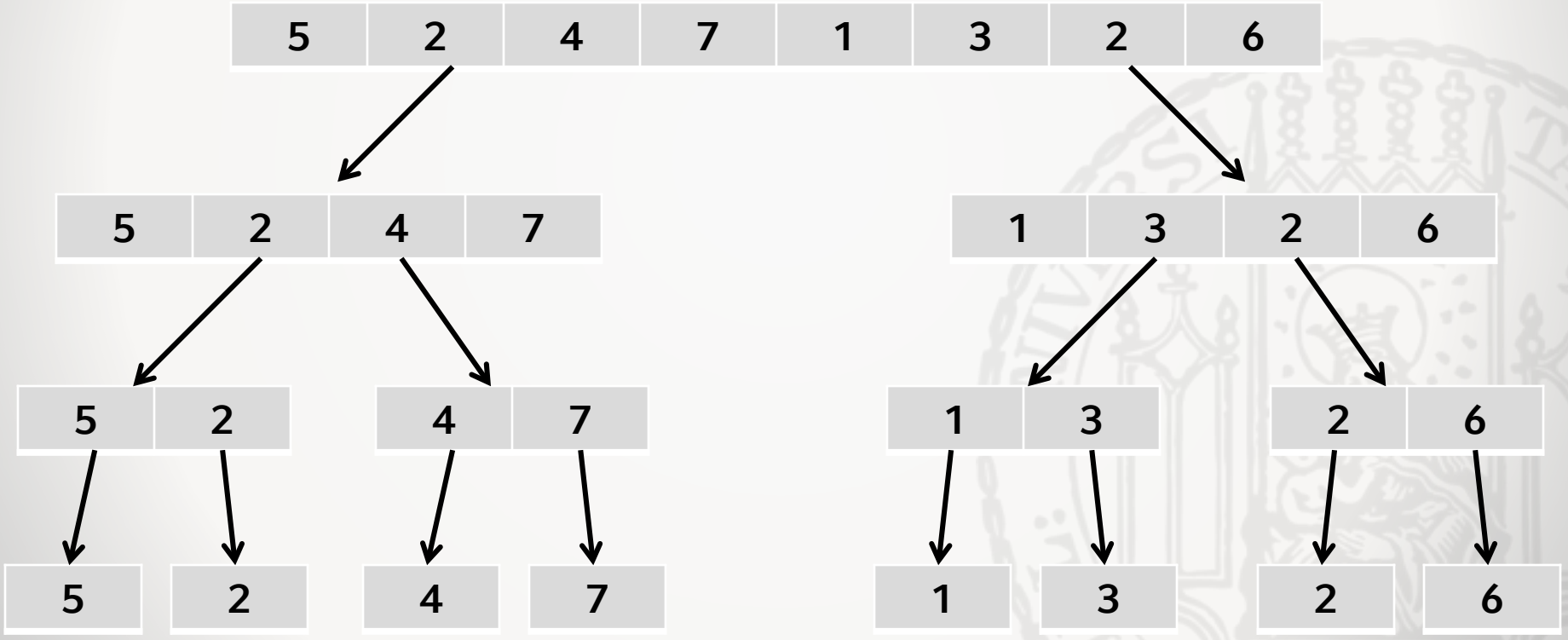
**Combine:** Die zwei sortierten Teillisten werden zusammengefügt (merge) um eine sortierte Liste zu erhalten.

## Weitere Algorithmen:

- Binarysearch
- Quicksort

# Merge-Sort: Divide

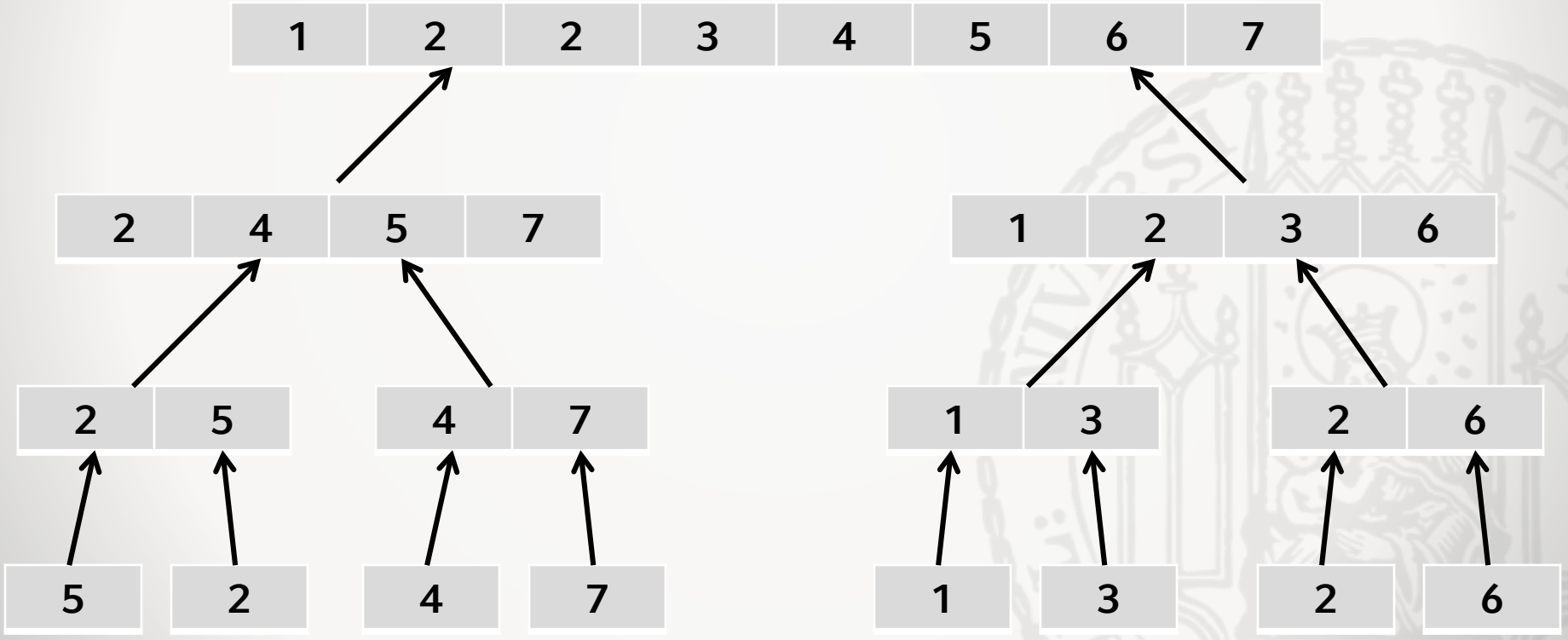
Die zu sortierende Liste der Länge $n$ wird in zwei zu sortierende Listen der Länge $n/2$ aufgeteilt



# Merge-Sort: Conquer und Merge

Einelementige Listen sind immer sortiert.

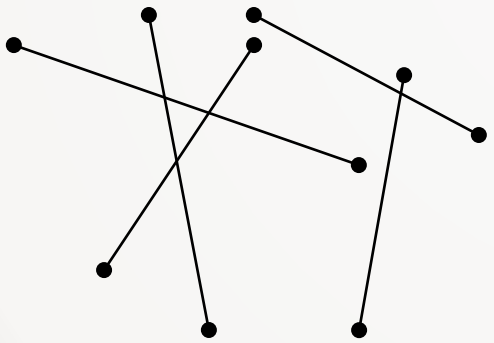
Längere Listen werden durch „mergen“ der sortierten Teillisten sortiert.



# Problemstellung: Schnitt von Liniensegmenten

- Gegeben:
  Eine Menge von abgeschlossenen Liniensegmenten in der Ebene. (abgeschlossene Liniensegmente enthalten ihre Endpunkte)

- Gesucht:
  Schnittpunkte der Segmente



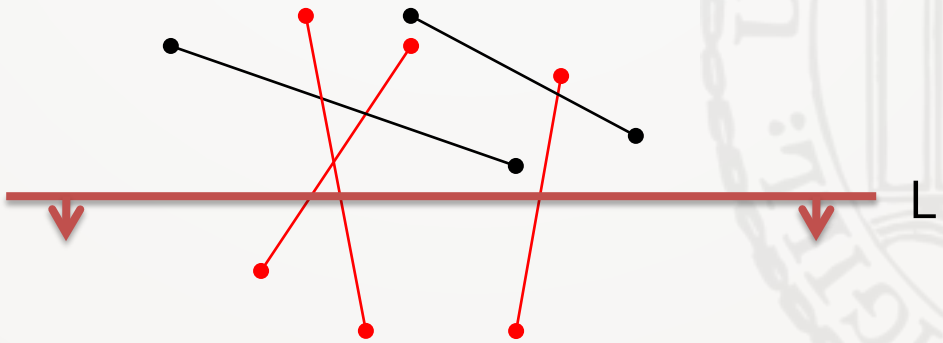
- Naive Brute-Force-Lösung:

Testen aller Segmentpaare in $O(n^2)$
-> Nur im Worst Case optimal.

- Gewünscht:
  Algorithmus, dessen Laufzeit geringer ist, insbesondere abhängig von der Größe des Outputs, d.h. „output-sensitiv“.

# Einfacher Sweep-Line-Algorithmus

- Idee: Vermeide den Test von Segmenten, die weit voneinander entfernt liegen.
- Durchführung: Teste nur die Segmentpaare, die gleichzeitig von einer horizontalen Sweep Line L geschnitten werden.
- Die Sweep Line wandert von oben nach unten und stoppt an bestimmten Events. Dieser Algorithmus wird über zwei Datenstrukturen durchgeführt:
  - Events = Endpunkte aller Segmente; verwaltet über eine Event-Liste, welche dem „Fahrplan“ des Algorithmus entspricht.
  - Status von L = Menge der von L aktuell geschnittenen Segmente. Der Status wird nach jedem Event aktualisiert und ist zu Beginn und am Ende leer.

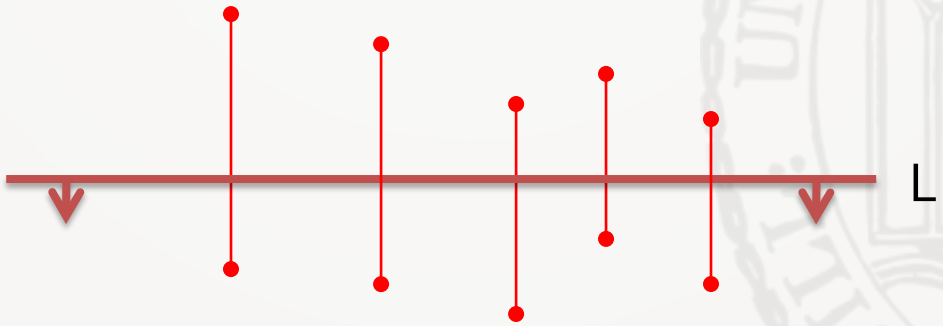


# Einfacher Sweep-Line-Algorithmus

- Ist das Event der obere Endpunkt eines Segments, so wird das Segment gegen alle Segmente im Status auf Schnitt getestet und dann dem Status hinzugefügt.
- Ist das Event der untere Endpunkt eines Segments, so wird das Segment aus dem Status entfernt.

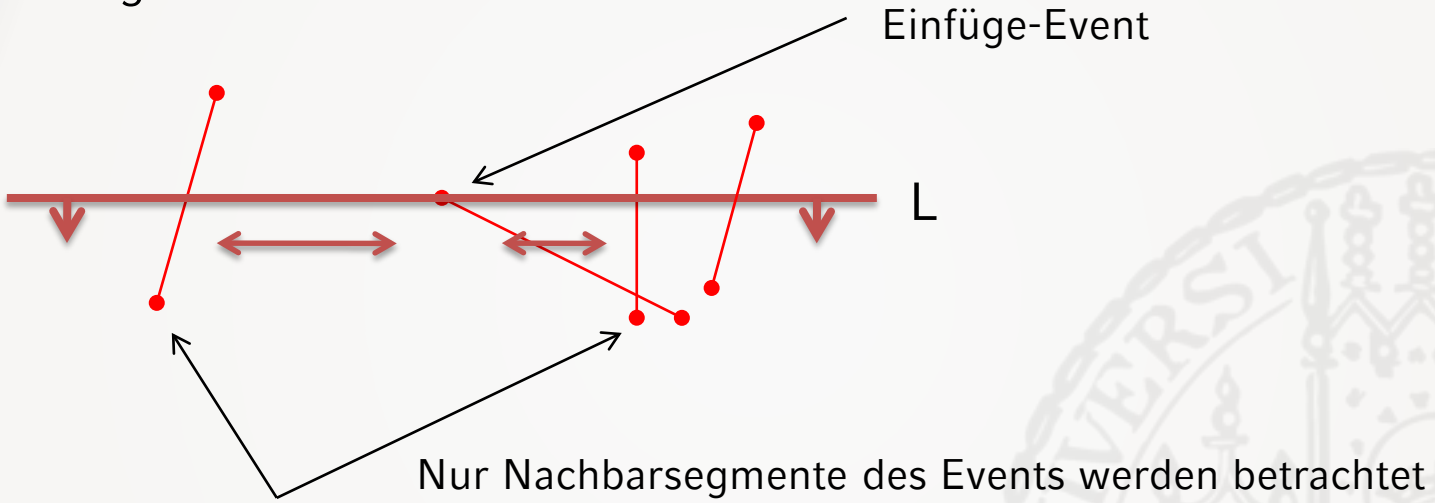
$\Rightarrow$ Es werden nur Paare getestet, die gleichzeitig von einer horizontalen Linie geschnitten werden.

Problem: Laufzeit immer noch unabhängig von der Größe des Outputs.



# Verbesserter Sweep-Line-Algorithmus

- Ordnen der Segmente im Status von links nach rechts.
- Vorteil: Bei Events werden nur noch Tests gegen Nachbarsegmente durchgeführt.




- Zu Beachten: Bei Schnittpunkten vertauscht sich die Ordnung der beteiligten Segmente im Status.

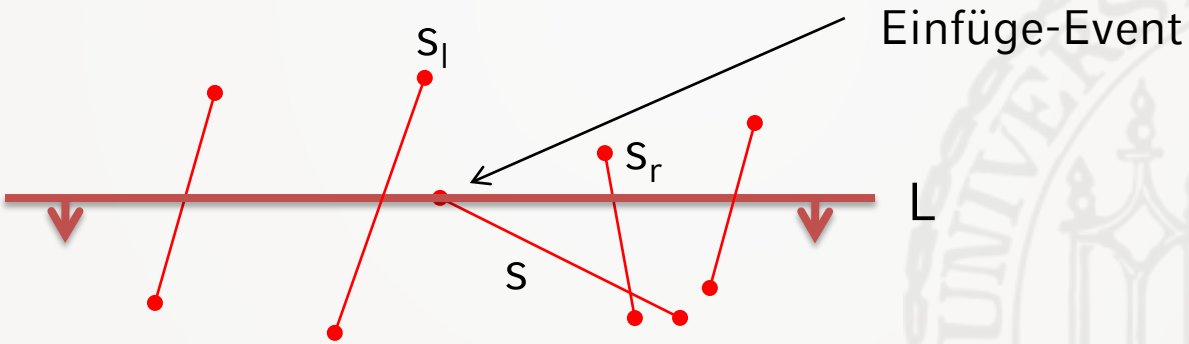
$\Rightarrow$ Schnittpunkte sind ebenfalls Events (Vertauschung), d.h. nach Erkennen eines Schnittpunktes wird dieser in die Eventliste eingefügt.

# Event-Verarbeitung

- Event = Oberer Endpunkt eines Segments s

⇒ Das Segment s wird in den Status eingefügt und muss gegen seine zwei Nachbarn $s_l$ und $s_r$ getestet werden

⇒ Nur Schnittpunkte unterhalb von L sind von Interesse; diese werden als Events in die Queue eingefügt.

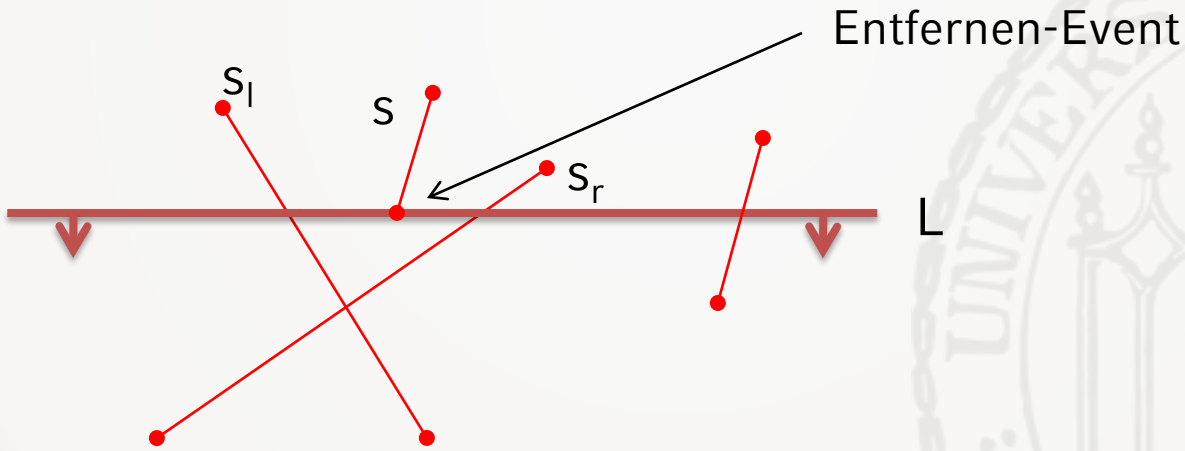


# Event-Verarbeitung

- Event = Unterer Endpunkt eines Segments s

$\Rightarrow$ Das Segment wird aus dem Status entfernt und die Nachbarn von s werden Nachbarn und müssen auf Schnitt getestet werden

$\Rightarrow$ Schnittpunkte werden ggf. als Events in die Queue eingefügt.

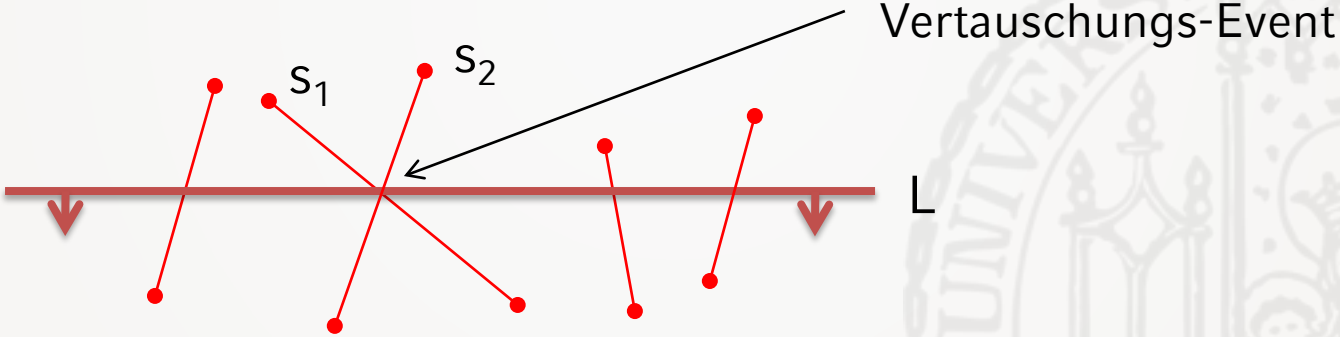


# Event-Verarbeitung

- Event = Schnittpunkt zwischen Segmenten $s_1$ und $s_2$

$\Rightarrow$ Die Reihenfolge von $s_1$ und $s_2$ im Status wird getauscht und die Segmente jeweils gegen (höchstens) einen neuen Nachbarn getestet.

$\Rightarrow$ Schnittpunkte werden ggf. als Events in die Queue eingefügt.



# Algorithmus: Datenstrukturen

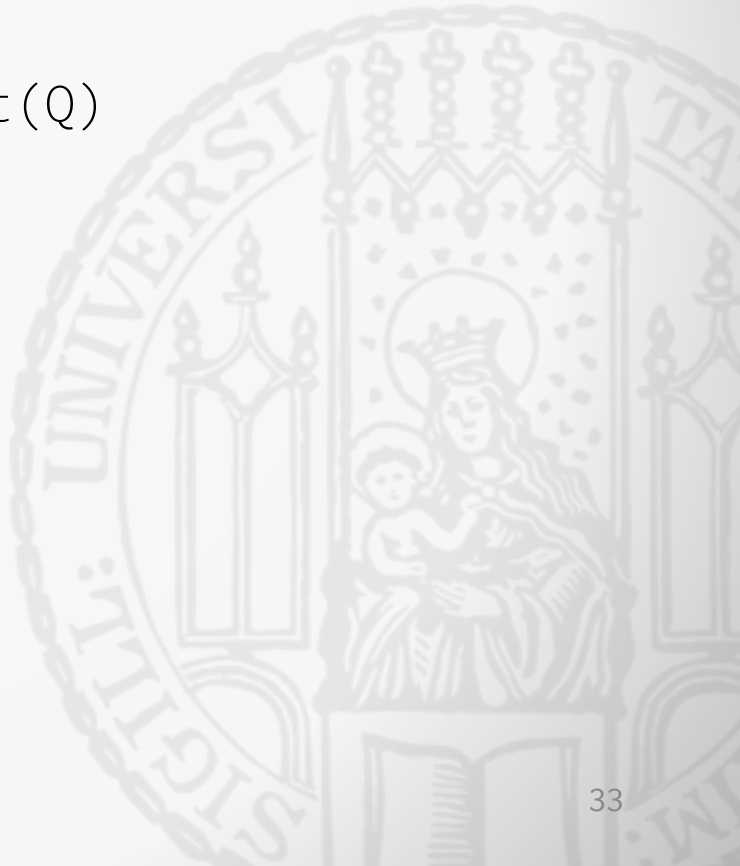
- Event-Queue Q als balancierter binärer Suchbaum oder Heap
  - Lexikographische Ordnung:
    $p < q \Leftrightarrow ( p_y < q_y$ oder $p_y = q_y$ und $p_x < q_x)$
    d.h. der linke Endpunkt eines horizontalen Segments wird zuerst abgearbeitet.
  - Operationen: Liefere nächstes Event, Prüfe ob Event bereits vorhanden

- Status T als balancierter binärer Suchbaum
  - Segmente sind entlang der Sweep Line L geordnet.
  - Operationen: Finde Segment links/auf/rechts von einem Punkt q

- Bezeichungen für disjunkte Segmentmengen:
  - U(p) = Segmente, die p als oberen Endpunkt haben.
  - L(p) = Segmente in T, die p als unteren Endpunkt haben.
  - C(p) = Segmente in T, die p im Inneren enthalten.

# Algorithmus

```
FindIntersections (Q) {
    Q ← empty
    insert segment endpoints into Q
    T ← empty
    while (Q is not empty) {
          p ← GetnextEventPoint(Q)
          removeEventPoint(Q,p)
          HandleEventPoint(p)
    }
}
```



# Algorithmus

```
HandleEventPoint(p) {
    determine U(p), L(p), C(p)
    if ( |U(p)| + |L(p)| +|C(p)| <1) report p as intersection
    remove L(p), C(p) from T          // Löschen & Neueinfügen von C(p)
    insert U(p), C(p) into T          //ändert die Ordnung der Segmente
    if (U(p) ∪ C(p) is empty)
          s_l,s_r ← left, right neighbors of p in T
          FindNewEvents(s_l, s_r, p)
    else
          s‘ ← leftmost segment of U(p) ∪ C(p)
          s_l ← left neighbor of s‘ in T
          FindNewEvents(s_l, s‘, p)

          s‘‘ ← rightmost segment of U(p) ∪ C(p)
          s_r ← left neighbor of s‘‘ in T
           FindNewEvent(s‘‘, s_r, p)
}
```

# Algorithmus

```
FindNewEvents(s_l, s_r, p) {
  if (s_l, s_r intersect below L || s_l, s_r
  intersect on L and on the right of p) {
        r ← Intersection( s_l, s_r)
        if (r not in Q) insert(r, Q)
  }
}
```

# Komplexität: Verbesserter Sweep-Line-Alg.

- Speicherkomplexität:

  $O(n + r)$, wobei n die Anzahl der Segmente ist, und $r$ die Anzahl ihrer Schnittpunkte. Die Anzahl der Events ist $2n + r$ ($2n$ Endpunkte und $r$ Schnittpunkte).

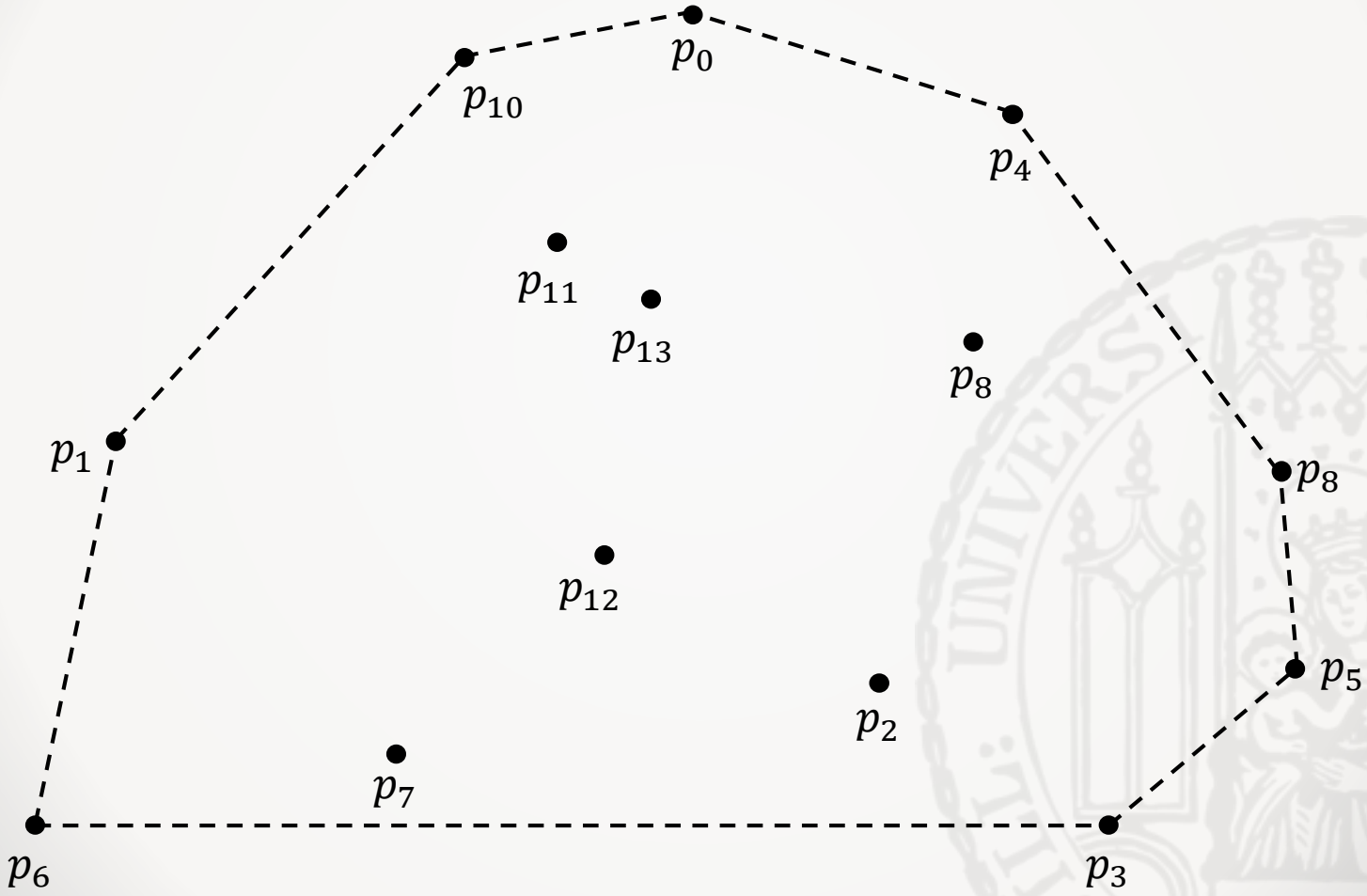
- Laufzeitkomplexität:

  $O\big((n + r) \cdot \log n\big)$, wobei sich $\log n$ auf die Suchkomplexität im binären Statusbaum bezieht. Dieser enthält maximal $n$ Segmente gleichzeitig und erfährt $n + r$ Einfügungen und $n + r$ Entfernungen.

## Konvexe Hülle

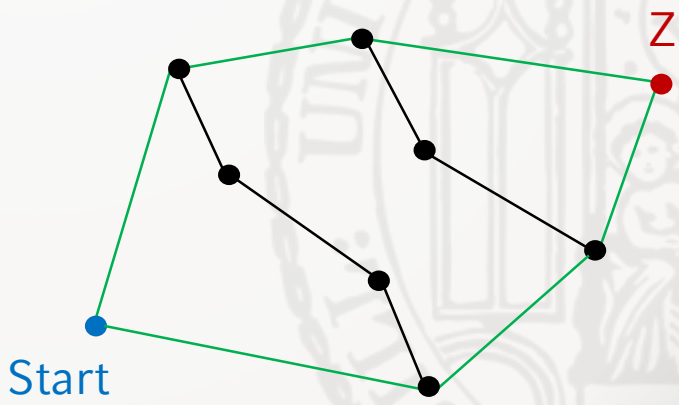
- Definition:
  - Die konvexe Hülle einer Punktmenge $M$ ist die kleinste konvexe Punktmenge, die $M$ umfasst.
  - Ist $M$ eine endliche Menge, so ist die konvexe Hülle ein (konvexes) Polygon, dessen Ecken eine Teilmenge von $M$ bilden.
- Erinnerung:
  - $X$ ist konvex, wenn für $x, y \in X$ auch alle Punkte der Strecke von $x$ nach $y$ in $X$ liegen.
- Bestimmung der konvexen Hülle:
  - Berechnung einer Liste von Punkten $p_1, \ldots, p_h$ zu einer gegebenen (endlichen) Punktmenge $M$ der Größe $n$, sodass die $p_i$ die Ecken der konvexen Hülle von $M$ in der richtigen Reihenfolge sind.

# Konvexe Hülle: Beispiel



# Konvexe Hülle: Motivation

- Data Mining
  - Grafische Erkennung von Ausreißern
  - Größter Durchmesser: Beschränkung der Suche auf die konvexe Hülle
- Computergrafik:
  - Näherung für Objektgeometrie
  - Oberflächlicher Test: Überlappen die konvexen Hüllen nicht, so überlappen auch nicht die Punktmengen
- Pathfinding
  - Berechnung einer Route um Hindernisse



# Konvexe Hülle: Bestimmung

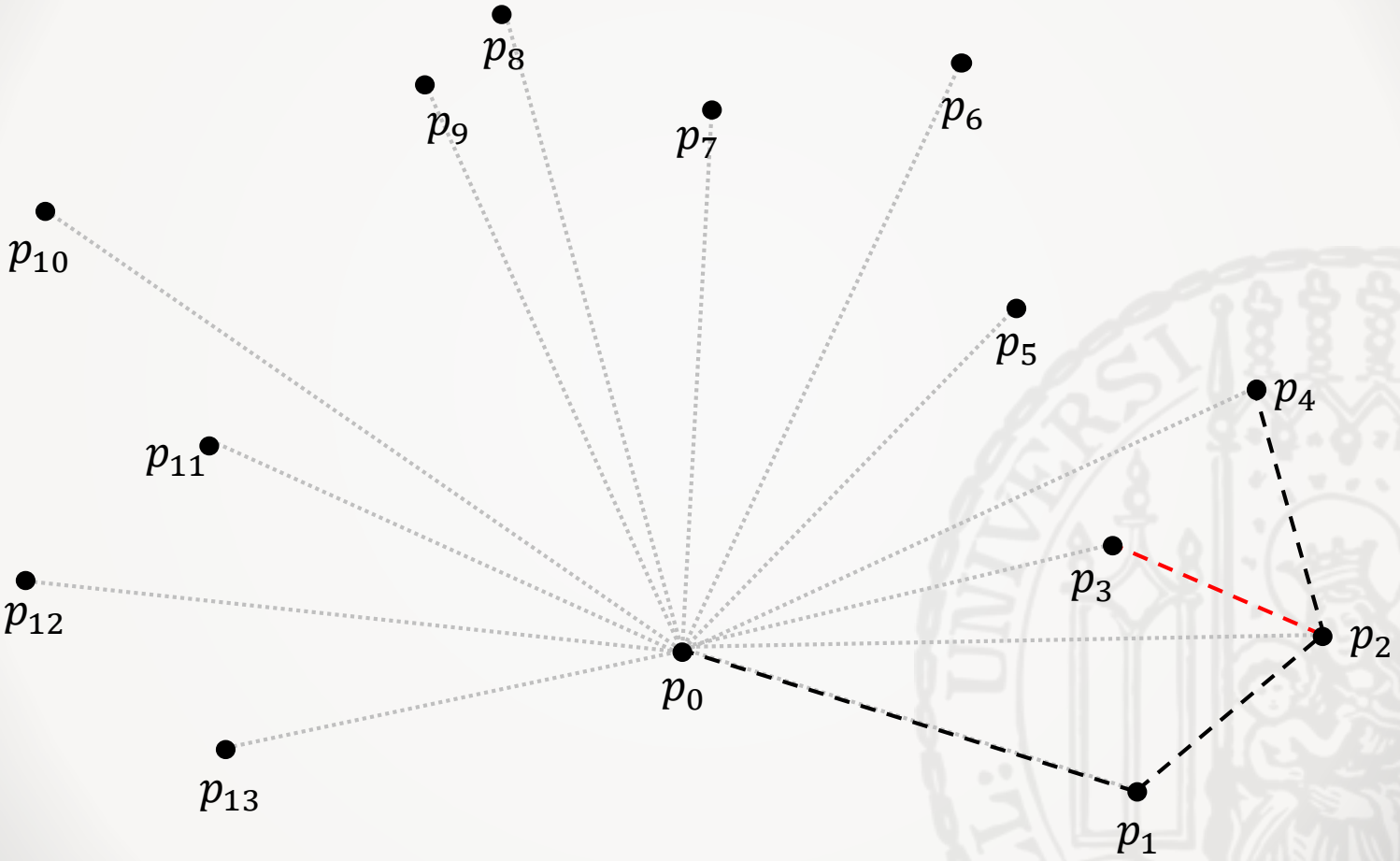
- Inkrementelles Verfahren:
  - Sukzessives Hinzufügen von Punkten zur konvexen Hülle der bisher gesehenen Punkte. Bei naiver Implementierung: $\Theta(n^2)$.
- **Graham's scan**:
  - Anordnung der Punkte nach Polarwinkel und überstreicht dann mit einer rotierenden Halbgeraden. Bei Umsetzung mit Stack: $O(n \log n)$.
- **Jarvis' march** (auch *gift wrapping*):
  - Man „umwickelt" die Punktmenge von unten her links und rechts. $O(nr)$, wobei $r$ die Größe der konvexen Hülle ist.
- Divide-and-conquer:
  - Jeweils zwei konvexe Polygone zusammenfügen, gesamt: $\Theta(n \log n)$.
- Sweep Line:
  - Neue, äußere Punkte im Status behandeln, gesamt: $O(n \log n)$
- Bestes Verfahren:
  - Ähnlich wie Medianfindung in Linearzeit: $O(n \log r)$.

## Konvexe Hülle: Graham's Scan

Algorithmus:

- Auswahl eines beliebigen Punktes $p_0$ aus $M$.
  Strategie: Wähle Punkt mit kleinster $y$-Koordinate.
- Sortierung der übrigen Punkte $p_1, \ldots, p_n$ nach Polarwinkel relativ zu $y$. Verarbeite der Reihe nach $p_1, p_2, \ldots, p_n$
- Verwendung eines Stacks, dessen Inhalt immer gerade die konvexe Hülle der bisher gesehenen Punkte ist
- Enthält der Stack weniger als zwei Elemente, oder bildet ein neuer Punkt $p$ mit den obersten beiden Punkten auf dem Stack eine Linkskurve, so wird $p$ auf den Stack gelegt.
- Bildet ein neuer Punkt $p$ mit den obersten beiden Punkten auf dem Stack eine Rechtskurve oder liegt geradeaus, so entfernen wir Punkte vom Stack bis eine Linkskurve vorliegt.
- Sind alle Punkte verarbeitet, so enthält der Stack die Ecken der konvexen Hülle

# Konvexe Hülle: Graham's Scan – Beispiel



## Konvexe Hülle: Graham's Scan – Laufzeit

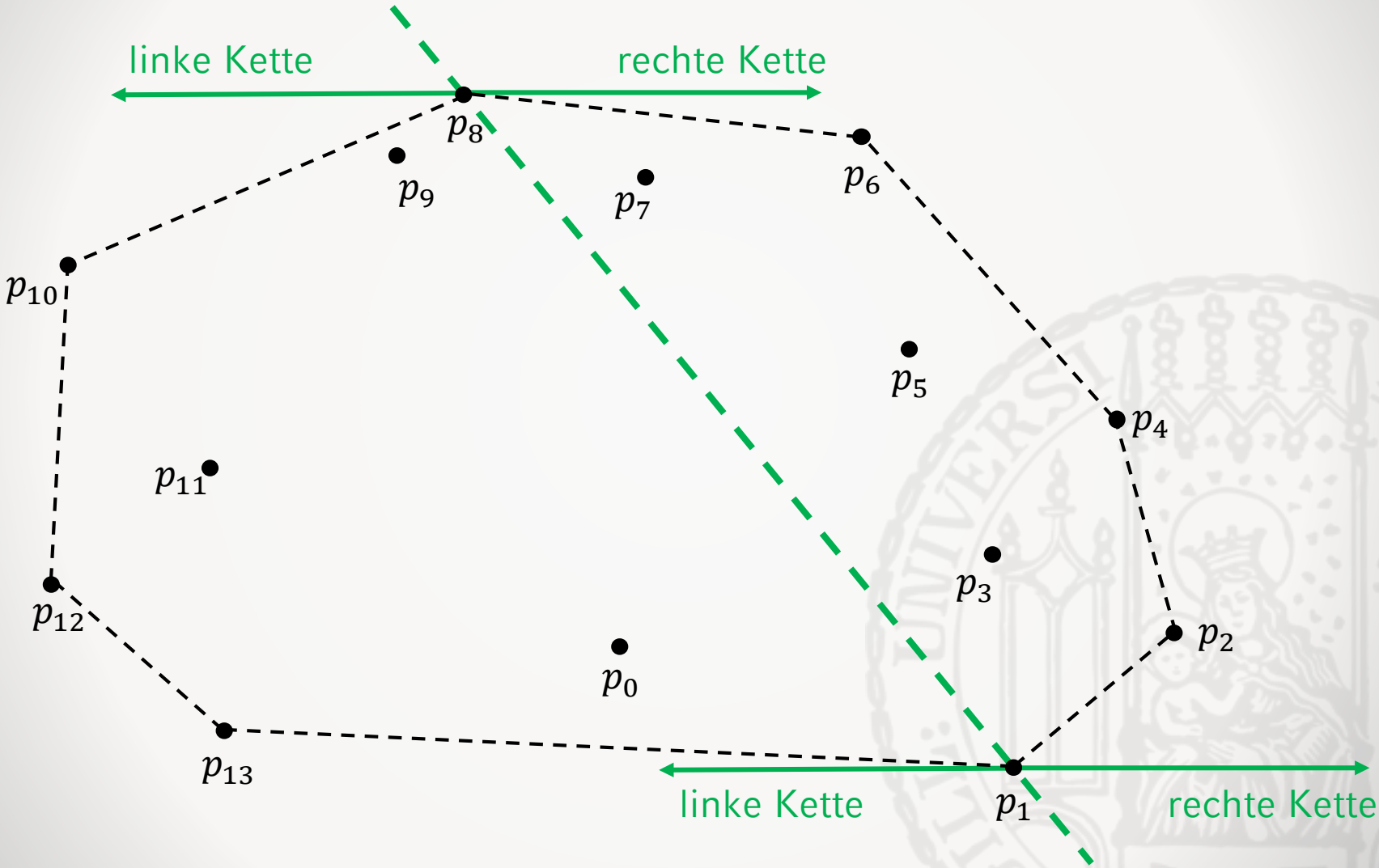
- Das Verarbeiten des $i$-ten Punktes kann bis zu $i - 2$ Operationen erfordern.
  - Laufzeit $\Theta(n^2)$? Nein!!!
  - Jeder Punkt wird höchstens einmal auf den Keller gelegt.
  - Dadurch wird jeder Punkt höchstens einmal wieder entfernt
  - Es finden also nur $O(n)$ Stackoperationen statt!
- Zudem: $O(n)$ Kreuzprodukte und Vergleiche notwendig!
- Die Laufzeit wird vom Sortierprozess dominiert: $O(n \log n)$.

- *Erinnerung Kreuzprodukt: Für zwei Vektoren $u, v$ ist das Kreuzprodukt $u \times v$ der zu $u$ und $v$ orthogonale Vektor.*

# Konvexe Hülle: Jarvis‘ March

- Algorithmus:
  - Wir befestigen einen Faden am Punkt mit der niedrigsten y-Koordinate
  - Wir wickeln das eine Ende des Fadens rechts herum, das andere links herum, solange bis sich die beiden Enden oben treffen
  - Der nächste Eckpunkt ist jeweils der mit dem kleinsten Polarwinkel in Bezug auf den als letztes Hinzugefügten.
  - Der Polarwinkelvergleich kann allein mit Kreuzprodukten ermittelt werden
- Laufzeit:
  - Insgesamt $O(nr)$ für $r =$ Ergebnisgröße
  - Besser als Graham‘s Scan wenn $r = o(\log n)$
  - Verallgemeinerbar auf 3D

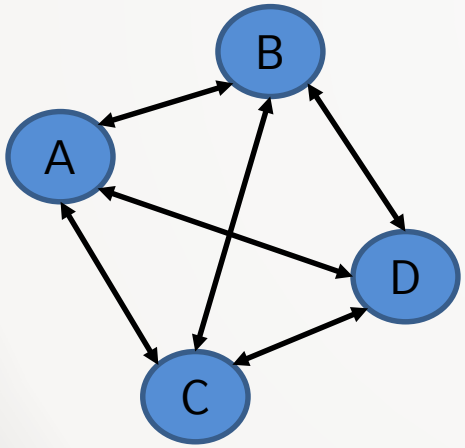
# Konvexe Hülle: Jarvis‘ March – Beispiel



# Traveling Salesperson Problem (TSP)

## Kombinatorisches Problem:

- Ein Handlungsreisender plant eine Rundreise durch mehrere Städte.
- Start und Ziel der Rundreise ist eine vorgegebene Stadt.
- Jede Stadt soll nur einmal besucht werden (Ausnahme Start und Ziel)
- Die Kosten der Reise sollen minimal sein.
- In welcher Reihenfolge müssen die Städte besucht werden?




***Entfernungstabelle***:

| von \ nach | A | B | C | D |
|---|---|---|---|---|
| A | --- | 10 | 15 | 20 |
| B | 5 | --- | 9 | 10 |
| C | 6 | 13 | --- | 12 |
| D | 8 | 8 | 9 | --- |

Eine optimale Route mit Kosten von ***35*** verläuft über: ***A -> B -> D -> C -> A***

# Traveling Salesperson Problem (TSP)

**Anwendung:**

- Tourenplanung in der Logistik
- Design von Mikrochips
- Genom-Sequenzierung
- …

**Problem:**

- Anzahl der Rundreisen: $(n-1)\cdot(n-2)\cdot\ldots\cdot2\cdot1 = (n-1)!$

  1.307.674.368.000 für 15 Knoten
  15.511.210.043.330.985.984.000.000 für 25 Knoten

→ Durchprobieren (Brute Force) für großes n praktisch nicht möglich

- Tatsächlich existiert bis heute kein exakter Algorithmus, der das TSP effizient (in nicht-exponentieller Zeit) löst
- Man geht in der Informatik davon aus, dass ein solcher auch nicht existieren kann („NP vollständig“)

**Wie gehen wir vor um ein solches NP-vollständiges Problem zu lösen?**

# Paradigma: Branch and Bound

## Idee:

- Methode zur Lösung von (NP-vollständigen) Optimierungsproblemen, bei denen keine anderen effizienten Verfahren bekannt sind.
- Wie bei Backtracking (trial and error).
- Schrittweises Annähern an die Gesamtlösung.
- Für die Teillösungen werden Schranken (Bounds) berechnet.
- Teillösungen werden dann verworfen, wenn das Optimum nicht mehr erreicht werden kann.

## Vorgehensweise:

- Berechne Bound $b$ der leeren Lösung $v$ und füge $\mathrm{x} = (v, b)$ in die Liste $Q$ ein.
- *Wiederhole:*
  - Entferne $(v, b)$ mit minimalem Bound $b$ aus $Q$
  - Falls $x$ vollständige Lösung *gib diese aus*
  - Sonst: Erweitere $x$, bestimme Bound $b_w$ für jede Erweiterung $w$ und füge $(w, b_w)$ in $Q$ ein.

# Branch für TSP

**Aufteilung des Lösungsraums in:**

- Rundreisen, die nicht direkt von $i$ nach $j$ gehen und
- Rundreisen, die direkt von $i$ nach $j$ gehen

Wird in der Entfernungstabelle $M[i,j]$ auf „–“ gesetzt, sind nur noch Rundreisen möglich, die Kante $(i,j)$ nicht enthalten.
Somit kann die Aufteilung des Lösungsraums wie folgt realisiert werden:

**Branch 1: Rundreisen die nicht direkt von i nach j gehen:**

Setze: $M[i,j] = -$ , d.h. Entfernen der Kante $(i,j)$

**Branch 2: Rundreisen die direkt von i nach j gehen:**

Setze: $M[i,k] = -$ , d.h. Entfernen der Kante $(i,k)$ *für* $k \neq j$
Setze: $M[l,j] = -$ , d.h. Entfernen der Kante $(l,j)$ *für* $l \neq i$
Setze: $M[j,i] = -$ , d.h. Entfernen der Kante $(j,i)$

→ Ein Baum entsteht, dessen Blätter die möglichen Pfade repräsentieren.

# Entscheidungsbaum: Branch and Bound

- Ziel: Verwerfen von Teillösungen

  → Zweige des Baumes werden nicht weiter verfolgt

- Zwei Möglichkeiten:
  - Tiefensuche um erste vollständige Lösung zu erhalten (in Blatt)
    - Die Lösung dient als obere Schranke des Ergebnisses
    - Alle Teillösungen deren Bound (untere Schranke) bereits über der Schranke liegen werden verworfen
  - Heuristische obere Schranke bereits in inneren Knoten des Baums

# Einfacher Bound für TSP

- Gesucht ist eine möglichst große **untere Schranke** für die Kosten.

  →Je genauer, desto näher am Optimum.
- **Idee:** Jede Stadt wird genau ein Mal **verlassen.**
- Aus jeder Zeile wird *genau ein* Wert entnommen.
- Die Summe der minimalen Werte jeder Zeile ist eine untere Schranke
- Gleiches gilt für die minimalen Werte der Spalten.

| von \ nach | A | B | C | D | ***min*** |
|---|---|---|---|---|---|
| A | --- | 10 | 15 | 20 | 10 |
| B | 5 | --- | 9 | 10 | 5 |
| C | 6 | 13 | --- | 12 | 6 |
| D | 8 | 8 | 9 | --- | 8 |
| | | | | Summe | ***29*** |

Kann man diese Schranke noch verbessern?

# Verbesserter Bound für TSP

**Idee:** Gibt es Städte, bei welchen noch nicht die minimalen Kosten für das **Betreten** angenommen wurden?

Reduzierung der Matrix:

- Bestimmung des Minimums $m_i$ der Einträge der Zeile `i`
- $m_i$ wird von allen Einträgen in der Zeile `i` abgezogen
- Wir reduzieren jede Zeile der Entfernungsmatrix `M` und erhalten die Matrix `M'` sowie die Werte $m_i, \dots, m_n$ als Minima der Zeilen von `M` und $m_{n+1}, \dots, m_{2n}$ als Minima der Spalten von `M'`.
- Der Bound entspricht $\sum_{i=1}^{2n} m_i$.

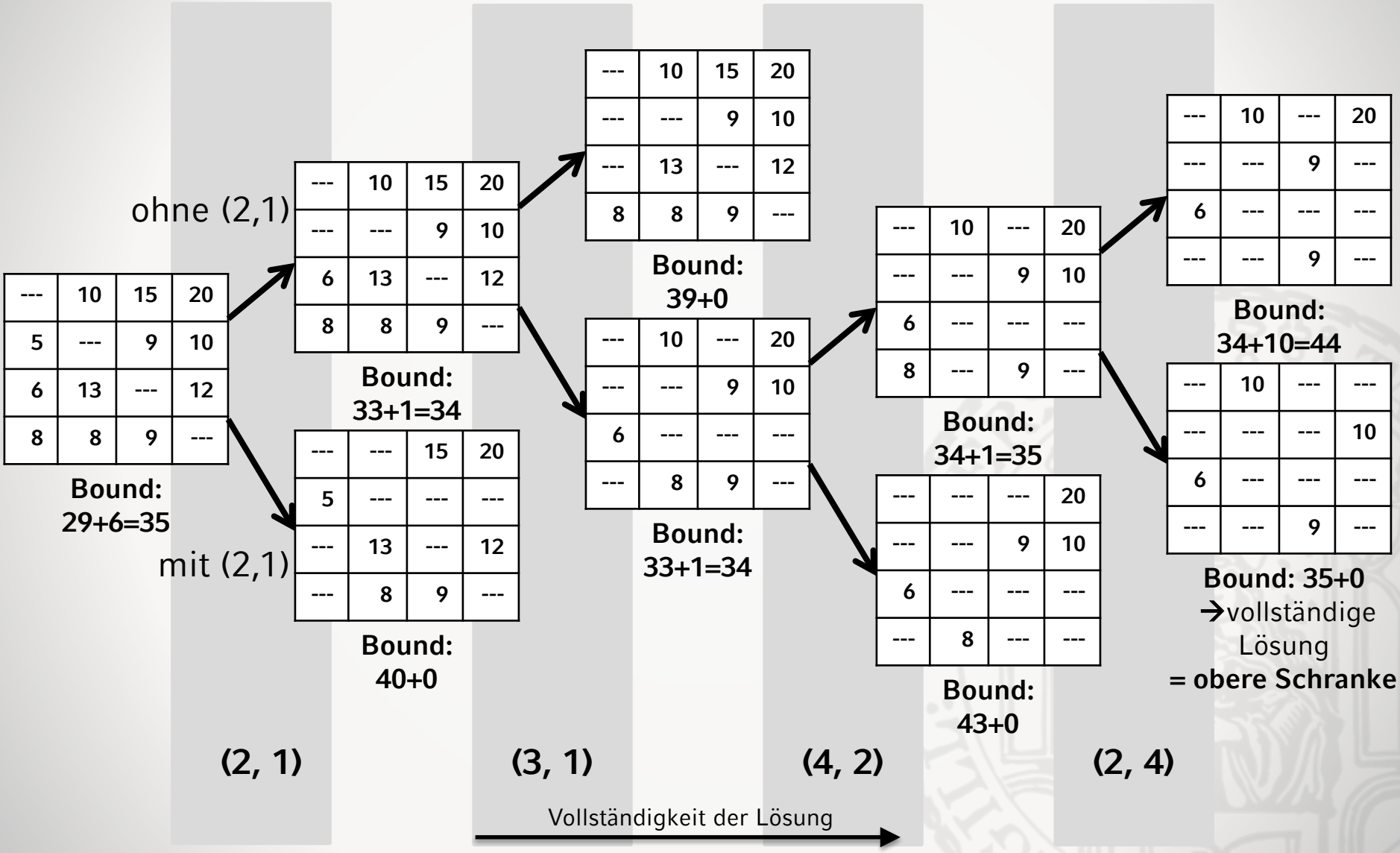
**Beispiel:**

| | | | | |
|---|---|---|---|---|
| --- | 10 | 15 | 20 | ***10*** |
| 5 | --- | 9 | 10 | ***5*** |
| 6 | 13 | --- | 12 | ***6*** |
| 8 | 8 | 9 | --- | ***8*** |
| | | | | ***29*** |

| | | | | |
|---|---|---|---|---|
| --- | 0 | 5 | 10 | |
| 0 | --- | 4 | 5 | |
| 0 | 7 | --- | 6 | |
| 0 | 0 | 1 | --- | |
| ***0*** | ***0*** | ***1*** | ***5*** | ***6*** |

Bound: ***29 + 6 = 35***

# Branch and Bound für TSP



# Optimierungsprobleme

Unter Optimierungsproblemen versteht man Probleme mit den folgenden Eigenschaften:

- Es gibt eine Menge von Lösungen
- Es gibt eine Bewertungsfunktion für Lösungen
- Gesucht ist eine beste Lösung (beste Bewertung)

**Bekannte Optimierungsprobleme:**

- Kürzester Weg im Graph
- *Rucksackproblem*
- Maximaler Fluss in einem Netzwerk
- *Traveling Salesperson Problem (TSP)*
- Minimaler Spannbaum

# Paradigma: Greedy (gierige) Algorithmen

**Idee:**

Schrittweise die jeweils für den Moment beste Entscheidung treffen.

- Meist schnelle Lösung
- Liefert oft keine optimalen Lösungen

**Beispiel:**

Prim-Algorithmus zur Berechnung des MST

**Vorgehen:**

Die besten Kanten werden sukzessive hinzugenommen.

# Rucksackproblem

Gegeben:

- Ein Behälter mit definierter Größe bzw. maximalem Gewicht *G*.
- Verschiedene Objekte definierter Größe/Gewicht und definierten Wertes.
  - Objekte: 1, …, i, …, n
  - Größe/Gewicht: $g_1$, …, $g_i$, …, $g_n$
  - Wert: $v_1$, …, $v_i$, …, $v_n$

Gesucht:

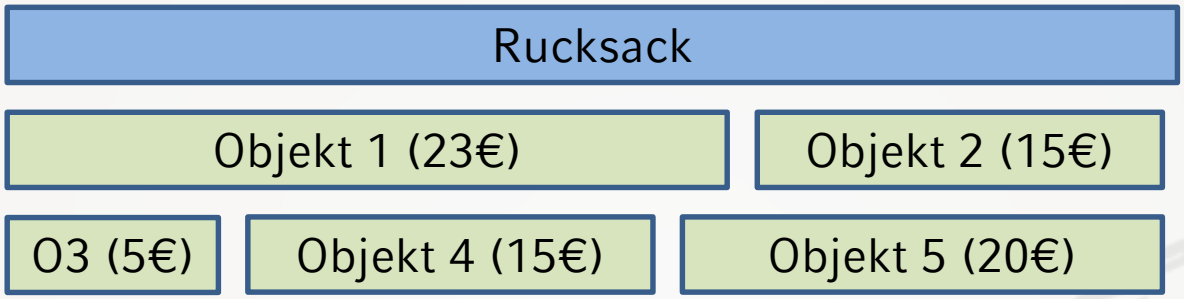
Maximaler Wert des Behälterinhaltes

Anwendung:

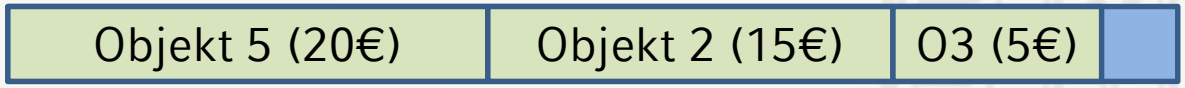
- LKW-Kapazität optimal nutzen
- Lagerhaltung
- …

# Rucksackproblem - Beispiel

Rucksack und Objekte:




Optimale Lösung:



# Greedy – Rucksackproblem

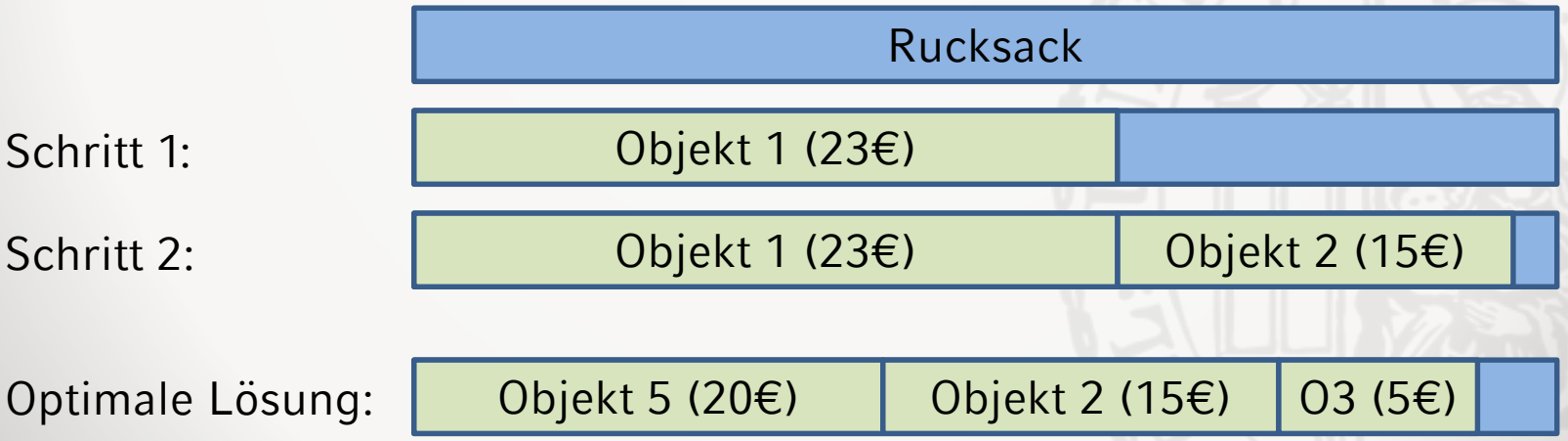
**Greedy:**

Wähle jeweils aus den übrigen Objekt jenes mit der höchsten Bewertung, das noch in den Rucksack passt und packe dieses hinzu.

**Bewertungen:**

- Je kleiner desto besser
- **Je wertvoller desto besser**
- Je größer das Verhältnis Wert zu Größe desto besser

**Greedy-Ansatz führt nicht zur optimalen Lösung!**



# Paradigma: Dynamische Programmierung

**Idee:**

Erst kleinste Probleme lösen und dann deren Lösungen sukzessive nutzen um größere Problemlösungen zu konstruieren.

Paradigma bereits bekannt:

- Floyd-Algorithmus

Jetzt:

- Edit Distanz
- Dynamischer-Algorithmus für das Rucksackproblem

# Vorgehensweise
# Dynamische Programmierung

1. Definition des Optimierungskriteriums
2. Definition von Teilproblemen
3. Zerlegung in Teilprobleme => führt zu Rekursionsgleichungen
4. Auswertung der Rekursionsgleichung mittels Tabelle

# Beispiel: Edit-Distanz

- Ähnlichkeitssuche in Datenbanken:
  - Z.B. Suchen von ähnlichen DNA-Sequenzen
  - Benötigt die Definition eines Ähnlichkeitsmaßes

- Ein Ansatz ist die Edit-Distanz zweier Sequenzen *s* und *t*:
  - Minimale Anzahl von Operationen, die benötigt werden um s in t zu überführen
  - Operationen sind Umbenennen, Löschen oder Hinzufügen einzelner Zeichen der Sequenz

- Alignments:
  - Hilfsmittel zur Veranschaulichung der Edit-Distanz
  - Ein Alignment zweier Sequenzen ist die zeichenweise Anordnung, so dass jeder Buchstabe einem anderen der anderen Sequenz oder einer Lücke „-“ zugeordnet ist
  - Die relative Ordnung der Zeichen bleibt dabei gewahrt

# Edit-Distanz: Rekursionsgleichung

- Die Berechnung der Edit-Distanz zweier Sequenzen entspricht also der Suche nach einem Alignment, so dass der Ähnlichkeitswert maximal bzw. die Distanz minimal ist
- Zu einer Sequenz *s* seien $s_i$ diejenige Sequenz, die aus den ersten *i* Zeichen von *s* besteht
- Sei $|s| = l$ , $|q| = m$. Die Edit-Distanz lässt sich dann rekursiv wie folgt bestimmen, wobei $d(s,q) \coloneqq d(s_i, q_m)$ :

$$d(s_i, q_j) = \begin{cases} j & falls\ i = 0 \\ i & falls\ j = 0 \\ d(s_{i-1}, q_{j-1}) & falls\ s[i] = q[j] \\ 1 + \ min \begin{cases} d(s_{i-1}, q_{j-1}) \\ d(s_i, q_{j-1}) \\ d(s_{i-1}, q_j) \end{cases} & sonst \end{cases}$$

# Edit-Distanz: Beobachtungen

- Naiver Ansatz: Löse das Problem mittels der Rekursionsgleichung
  - Exponentieller Zeitaufwand
  - entsteht durch Mehrfachberechnungen der einzelnen Teillösungen
- Idee:
  - Bottom-Up statt Top-Down Berechnung
- Ansatz Dynamische Programmierung:
  - Teilberechnungen in eine Matrix eintragen
  - Aus den einzelnen Teillösungen für kleinere Teilprobleme Lösungen für größere Teilprobleme (bis zum Gesamtproblem) zusammensetzen
- Platz- bzw. Zeitkomplexität:
  - Jeder Matrixeintrag kann in $O(1)$ bestimmt werden
  - Komplexität beträgt also $O(|s| \cdot |t|)$

# Edit-Distanz: Algorithmus

Gegeben: Zwei Sequenzen *s* und *t*

- Erstelle eine $|s + 1| \times |t + 1|$ Matrix D, die die einzelnen Distanzen enthält

1. Basisfälle:
   - $i=j=0$ ➔ $D[i][j]=D[0][0]=0$
   - $i=0,\ 0<j\leq|t|$ ➔ $D[i][j]=D[0][j]=j$ (Hinzufügen von j Zeichen, 1. Zeile)
   - $0<i\leq|s|,\ j=0$ ➔ $D[i][j]=D[i][0]=i$ (Hinzufügen von i Zeichen, 1. Spalte)

2. Ausfüllen der restlichen Tabelle:
   - Zeilen- oder spaltenweises Ausfüllen
   - $d(s_i, t_j)$ ergibt sich aus $d(s_{i-1}, t_{j-1})$, $d(s_{i-1}, t_j)$ sowie $d(s_i, t_{j-1})$, also: D[i][j] ergibt sich aus D[i-1][j-1], D[i-1][j] sowie D[i][j-1]

- Die gesuchte Edit-Distanz ist dann $d(s_{|s|}, t_{|t|})$ bzw. der letzte Matrixeintrag (rechts unten), also $D[|s|][|t|]$

# Beispiel zur Edit-Distanz

Berechne die Edit-Distanz der beiden Sequenzen HALLO und AUTO:

| | | A | U | T | O |
|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 |
| H | 1 | | | | |
| A | 2 | | | | |
| L | 3 | | | | |
| L | 4 | | | | |
| O | 5 | | | | |

# Beispiel zur Edit-Distanz

Berechne die Edit-Distanz der beiden Sequenzen HALLO und AUTO:

| | | A | U | T | O |
|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 |
| H | 1 | 1 | 2 | 3 | 4 |
| A | 2 | 1 | 2 | 3 | 4 |
| L | 3 | 2 | 2 | 3 | 4 |
| L | 4 | 3 | 3 | 3 | 4 |
| O | 5 | 4 | 4 | 4 | **3** |

# Rucksackproblem

## 1. Definition des Optimierungskriteriums:

Für jedes Objekt *i* ist zu entscheiden, ob es eingepackt wird.

$g_j$: Gewicht von Objekt *j*

$v_j$: Wert von Objekt *j*

$$a_i \in \{0,1\}$$

$$a^n := a_1,\ldots,a_n$$

$$\max_{a^n}\left\{\sum_{j=1}^{n} a_j v_j : \sum_{j=1}^{n} a_j g_j \leq G\right\}$$

Gesucht sind $a_1, \ldots, a_n$, sodass das zulässige Gewicht nicht überschritten wird und der Gesamtwert maximal ist.

## 2. Definition von Teilproblemen und einer Hilfsgröße:

Betrachtet wird der Wert *w(i,h)* eines nur bis *h* gefüllten Rucksackes, wobei nur die Objekte *1,…, i* verwendet werden.

**Hilfsgröße:**

$$w(i,h) = max_{a^i}\{\sum_{j=1}^{i} a_j v_j \mid \sum_{j=1}^{i} a_j g_j \leq h\}$$

## 3. Zerlegung in Teilprobleme

**Ansatz:** Konstruiere eine Lösung für i Objekte auf Basis der Lösung von i-1 Objekten

$$w(i,h) = max \begin{cases} 0 + \ max_{a^{i-1}} \left\{ \sum_{j=1}^{i-1} a_j v_j \,\middle|\, \sum_{j=1}^{i-1} a_j g_j \leq h \right\}, \\ v_i + max_{a^{i-1}} \left\{ \sum_{j=1}^{i-1} a_j v_j \,\middle|\, \sum_{j=1}^{i-1} a_j g_j \leq h - g_i \right\} \end{cases}$$

$$= max\{\, w(i-1,h),\, v_i + w(i-1, h-g_i)\}$$

$w(i-1, h-g_i)$: „höchster Wert wenn Objekt i noch nicht verwendet wurde und noch Platz für das Objekt ist.“

## 4. Auswertung der Rekursionsgleichung

**Randbedingungen:**

$$i = 0 : w(i,h) = 0$$
$$h \leq 0 : w(i,h) = 0$$

**Implementierung: 2 Schleifen**

```
for i = 1, …, n do
    for h = 0, …, G do
        w[i,h] = max{ … }
```

- Lösung für *n* Objekte und Rucksack der Größe *G* steht in `w[n,G]`

# Rucksackproblem in Java

```
public static int pack(int capacity, int[] value, int[] weight){

  int[][] matrix = new int[value.length+1][capacity+1];
  int count = value.length;

  for(int i = 1; i <= count; i++){
    for(int h = 1; h <= capacity; h++){

      if(weight[i] <= h)
        matrix[i][h] =
          max(matrix[i-1][h], value[i] + matrix[i-1][h-weight[i]]);
      else
        matrix[i][h] = matrix[i-1][h];

    }
  }

  return matrix[count][capacity];

}
```

Rekursionsgleichung:

$$w(i,h) = \max\{w(i-1,h), v_i + w(i-1,h-g_i)\}$$

# Rucksackproblem

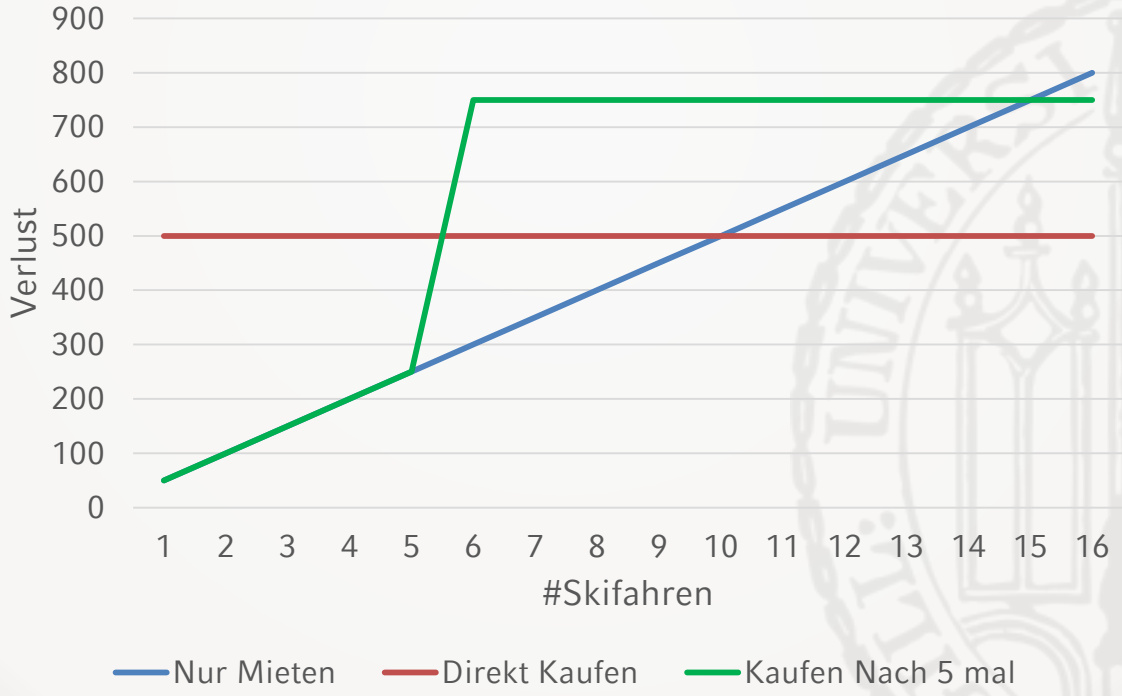
**Rucksackgröße:** 10

| ***Objekte*** | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| Größe | 4 | 4 | 3 | 1 | 6 | 2 | 1 |
| Wert | 5 | 7 | 3 | 2 | 10 | 1 | 2 |

| *Objekte* \ *Größe* | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 5 | 5 | 5 | 5 | 5 | 5 | 5 |
| 2 | 0 | 0 | 0 | 0 | 7 | 7 | 7 | 7 | 12 | 12 | 12 |
| 3 | 0 | 0 | 0 | 3 | 7 | 7 | 7 | 10 | 12 | 12 | 12 |
| 4 | 0 | 2 | 2 | 3 | 7 | 7 | 9 | 10 | 12 | 14 | 14 |
| 5 | 0 | 2 | 2 | 3 | 7 | 7 | 10 | 12 | 12 | 14 | 17 |
| 6 | 0 | 2 | 2 | 3 | 7 | 7 | 10 | 12 | 12 | 14 | 17 |
| 7 | 0 | 2 | 4 | 4 | 7 | 9 | 10 | 12 | 14 | 14 | 17 |

# Rent-Or-Buy-Problem

- Angenommen, Sie wollen Ski fahren.
  Lohnt es sich, das Equipment zu kaufen oder ist Mieten günstiger?
- Miete: 50 €
- Kaufpreis: 500 €
- Problem: Vielleicht merken Sie nach n Malen, dass es keinen Spaß macht.



# Online-Algorithmen

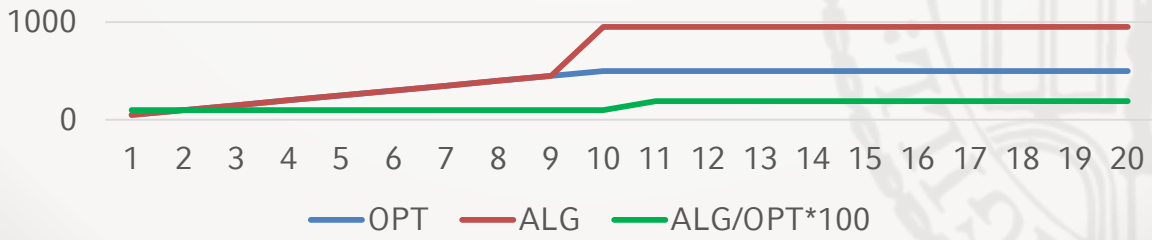
- Ein Online-Algorithmus berechnet ein Ergebnis schon dann, wenn noch nicht alle Eingaben zur Verfügung stehen.
- Das Ski-Beispiel steht stellvertretend für eine Klasse von Problemen.
- Wichtiger Vertreter für die Informatik: *Paging*
  Soll eine Speicherseite im Hauptspeicher oder auf der Festplatte festgehalten werden?
  - Hauptspeicher ist teurer, aber schneller
  - Strategie: Häufig benutzte Seiten im RAM, weniger benutzte auf der Festplatte
  - Was wird in naher Zukunft wieder gebraucht?
  - Welche Strategie ist eine gute Strategie?
- Eine gute Strategie sollte derart handeln, sodass die zusätzlichen Kosten im Worst-Case im Verhältnis zur optimalen Lösung minimiert werden.
- Um die Qualität einer Strategie zu bestimmen, bedient man sich der

$$Competitive\ ratio = \max_{x \in Input} \left( \frac{ALG(x)}{OPT(x)} \right)$$

# Competitive Ratio in Rent-Or-Buy

$$Competitive\ ratio = \max_{x \in Input} \left( \frac{ALG(x)}{OPT(x)} \right)$$

- Strategie „Kaufe sofort“
  Worst-Case: Einmal Skifahren → Kosten ALG = 500, OPT = 50; CR = 10
- Strategie „Miete immer“
  Worst-Case: Nie aufhören mit dem Skifahren → Kosten ALG unbegrenzt
  CR = ∞
- Strategie „Besser spät als nie“
  (miete, bis der Kaufpreis erreicht wurde, dann kaufe)
  - Falls wir 1-10 mal Ski fahren, sind wir optimal, denn auch OPT würde dann nur mieten
  - Darüber hinaus hätte OPT direkt gekauft, da $10 \cdot 50€ = 500€$
  - Unsere Strategie verursacht Kosten von $450€ + 500€ = 950€$
  - Damit ist diese Strategie die beste der drei Strategien mit einer Competitive Ratio = 1.9
  - Diese Strategie hat beweisbar die beste Competitive Ratio



# Competitive Ratio in Rent-Or-Buy (2)

- Seien $r$ die Mietkosten pro Tag und $b$ der Kaufpreis.
- Jeder Onlinealgorithmus kauft nach $k-1$ Tagen

$$c(ALG) = (k-1)r + b$$

- Ein optimaler Algorithmus OPT kauft, falls die Miete teurer wäre als ein Kauf:

$$c(OPT) = \min\{kr, b\}$$

- Für das Verhältnis gilt stets:

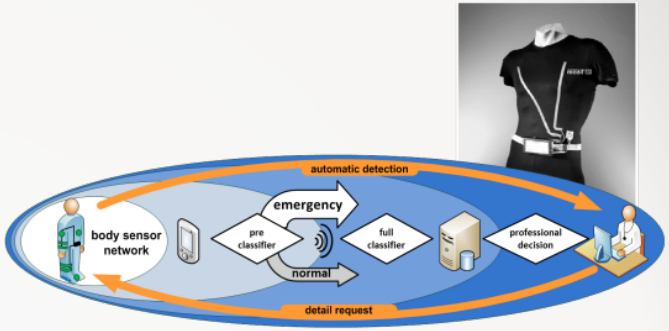
Fall 1: $kr \leq b$
Fall 2: $b \leq kr$

$$\frac{c(ALG)}{c(OPT)} = \frac{kr - r + b}{\min\{kr, b\}} = \max\left\{\frac{kr - r + b}{kr}, \frac{kr - r + b}{b}\right\} \geq \frac{b - r + b}{b} = 2 - \frac{r}{b}$$

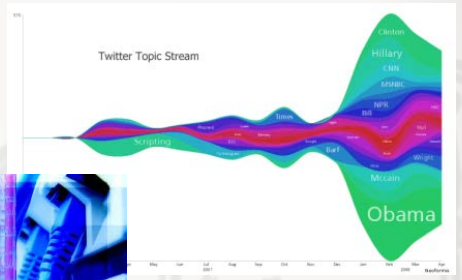
- Daher gibt es keinen Onlinealgorithmus, der eine bessere Competitive Ratio besitzt.
- Die „Besser spät als nie“-Strategie erreicht genau diese CR mit $k = \frac{b}{r}$ (falls $r$ $b$ teilt).

$$\frac{c(ALG)}{c(OPT)} = \frac{(k-1)r + b}{\min\{kr, b\}} = \frac{b - r + b}{\min\{b, b\}} = 2 - \frac{r}{b}$$

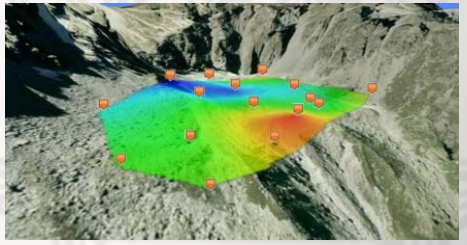
# Stream-Algorithmen




Daten sind nicht immer statisch verfügbar
=> Nach Berechnung eines Ergebnisses kann dieses schon veraltet sein







# Beispiel: Postversand



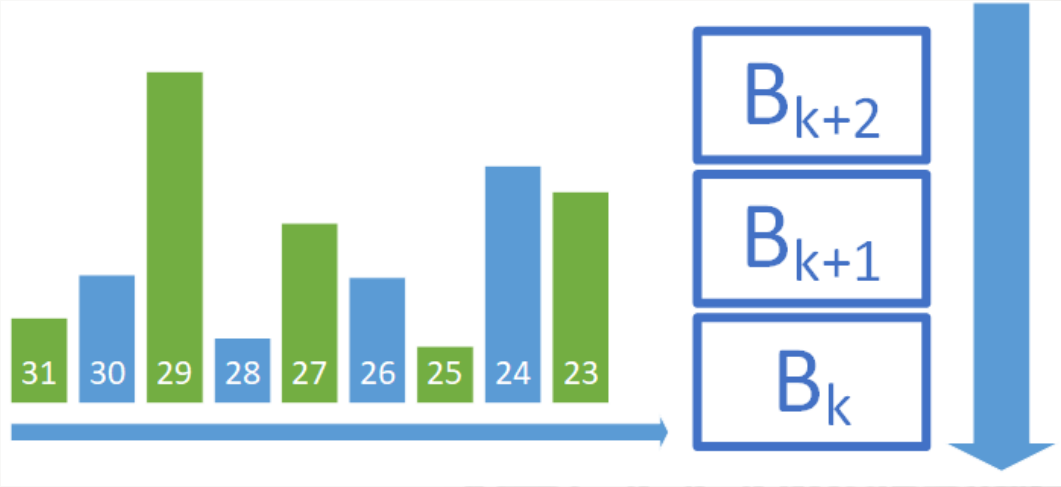
- Frachtzentrum:
  Befülle m Container (BxHxT) mit möglichst vielen Paketen unterschiedlicher Packmaße.
- Zu wenig Pakete bedeutet finanzieller Verlust, zu viele Pakete ist technisch unmöglich.
- Problem ist vergleichbar mit dem Rucksackproblem. Die räumlichen Eigenschaften machen es komplizierter.
- Greedy Ansatz zu naiv?
- Eventuell: Bei vielen unterschiedlichen Packmaßen entstehen viele Lücken. Sind dagegen Pakete normiert, ist es (etwas) einfacher.

- Weiteres Problem: Die Menge der Pakete ist nie abgearbeitet!

# Online-Rucksack-Problem

Gegeben:

- Eine unendliche Folge von Behältern $B_1, B_2, \ldots$ mit maximalem Gewicht $G$ und einer Terminierungszeit $t_1^B, t_2^B, \ldots$ .
- Unendlich viele verschiedene Objekte $o \in (W, T)$ mit definiertem Gewicht $w$ und einer Ankunftszeit $t$.
  $(w_1, t_1), (w_2, t_2), \ldots$




Gesucht:

Verteilung aller Objekte $f(o) = i$, sodass Kosten minimal sind.
Die Kosten für einen Container sind Wartezeiten:

$$\sum_{o \in (W,T)} |T_{f(o)} - t(o)|$$

# Online-Rucksack-Problem (2)

- Probleme:
  - Variety: Objekte haben unterschiedliche Formen
    - Paketgrößen, Gewicht, Inhalt, Sperrgepäck
  - Volume: viele Objekte
    - Lange Lagerung => teuer und verringert Serviceleistung
  - Velocity: hohe Ankunftsfrequenz
    - Ebenfalls variierende Frequenz, nachts und vor Festtagen kommen mehr Pakete
  - Veracity: Objekte sind unsicher
    - Zoll entfernt Pakete, Beschädigungen, Falsche Adressierung, Flüge fallen aus
- Im Voraus kaum berechenbar: Kein Orakel verfügbar, dass die nächsten 1000 Pakete vorhersagt.
- Kein langes Ablegen der Pakete möglich, Menge zu groß.
- Dennoch sind Gewinnspannen kritisch klein.
- Finde also eine gute Lösung möglichst schnell, anstatt eine fast beste Lösung später.

# Stream-Algorithmen Anforderungen

- Unendlicher Stream
  - Jedes Objekt nur einmal berühren
- Begrenzte Zeit
  - Schneller Zugriff, günstige Methoden
- Begrenzter Speicher
  - Kompression
- Veränderungen der Verteilungen
  - Aging & Aktualisierung des Modells
- Verrauschte Daten
  - Rauschunterdrückung
- Variierende Datenraten
  - Arbeite genau, wenn viel Zeit vorhanden ist
  - Beschleunige bei vielen Datenmengen

CheckListe für Stream Algorithmen:

- Single scan, missing random access
- Fast access, cheap methods
- Compression
- Aging & updating models
- Noise handling
- Handle varying data rates

# Sampling

- Wähle eine repräsentierende Submenge der Daten


- Dazu ist die Auswahlstrategie entscheidend
  - online random sampling, extreme case sampling, min-wise sampling, …
- Sample-Menge muss endlich sein
- So lassen sich gut statistische Auswertungen anfertigen

- Für das Beispiel keine Lösung, aber evtl. doch hilfreich:
  - Wie oft gibt es Sperrgepäck?
  - Wie groß ist das Durchschnittsgewicht jedes Objekts?
  - Wann bilden sich Wartezeiten/Stoßzeiten?

# Buffering

- Benutze endlich großen Zwischenspeicher (Buffer) (FiFo Warteschlange)


- Verarbeite Objekte nicht aus dem Stream, sondern der Reihe nach aus dem Speicher
- Kann Wartezeiten verringern
- Die Wahrscheinlichkeit eines Fehlers (buffer overflow) hängt ab von
  - Buffergröße, Budget und Ankunftsverteilung
- Für das Beispiel auch keine absolute Lösung:
  - Begrenzte Vorschau auf die nächsten Objekte, die in den Buffer (Lagerhalle) passen
  - Eventuell teuer im Verhältnis zum Mehrwert

# Zählen im Stream

- Problem: Zähle die Anzahl $M$ unterschiedlicher Objekte im Stream (z.B. Wieviele Pakete der Standardgrößen M, L, XL)

A A F A B B A F F A A F A B A

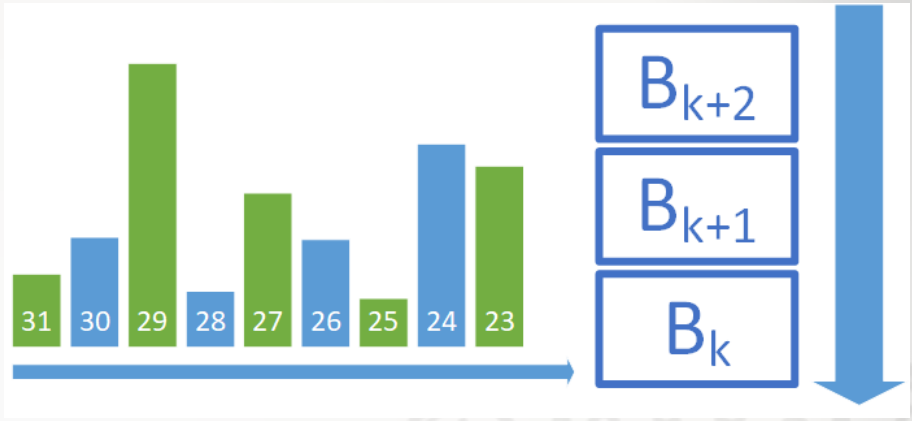
- $|Q|$ sei die Anzahl der unterschiedlichen Eingabewerte
- Eine exakte Lösung benötigt $O(M)$, möglicherweise $O(|Q|)$
- Approximative Lösung möglich in $O(\log|Q|)$ möglich:
  - Benutze Hash-Funktion $h(o): O \rightarrow [1 \dots |Q|]$
  - Außerdem Hash-Sketch: Bit-Vektor $B$ der Länge $\log|Q|$
  - Für jedes neue Objekt $o$ setze $B[lsb(h(o))] = 1$
  - Sei $R$ die Position der rechtesten Null in $B$
    - Für große Streams[1] $E(R) = \log(\phi M)$ mit $\phi = 0.7735$
  - Schätze die Anzahl der unterschiedlichen Objekte auf $\widehat{M} = 2^R/\phi$
  - Beispiel: $|Q| = 7, h(x) = 3 \cdot x \ mod \ 7$

$$4,5,6,4,6,4,5,6, \dots \rightarrow M \sim 2.6$$

[1] Flajolet et al.: *Probabilistic counting algorithms for data base applications.* Journal of Computer and System Sciences 1985

# Online-Rucksack-Problem

Können wir nun das Problem lösen?




- Nicht optimal, da die Zukunft unbekannt
- Aber besser als raten:
  - Buffering erlaubt überspringen von Paketen
  - Statistiken über den Paketstream geben Charakteristik des aktuellen Buffers
  - Falls eher große/schwere Pakete (überdurchschnittlich viele) im Buffer, dann lieber die größten Pakete bevorzugen. Diese sind schwer zu verarbeiten.
  - Falls eher kleine Pakete ankommen (die man leichter unterbringen kann), dann neue schwerste Pakete bevorzugen.
  - Je nach unterschiedlichen Eigenschaften (Trend-Detection) kann auf die Zukunft reagiert und Parameter verändert werden.
  - Dies ist nur eine Heuristik. In der Praxis wird die Verteilung schon vor der Ankunft auf dem Weg berechnet.

# Recap: Was haben wir gemacht?

- Was ist ein Algorithmus? Eigenschaften, Komplexitätsklassen
- Sortieren von Listen
- Suchen in Listen
- Hashing
- Baumstrukturen (Binärbäume, balancierte Bäume, Mehrwegbäume)
- Graphdarstellungen und –traversierungen
- Kürzeste Wege und minimale Spannbäume bestimmen
- Backtracking
- Divide and Conquer
- Geometrische Verfahren
- Greedy-Algorithmen
- Dynamische Programmierung
- Branch and Bound
- Online-Algorithmen

## Klausur

- Wann: 23.07.2018, 16 Uhr
- Wo: Hauptgebäude (Zuteilung folgt auf der Webseite)
- Anmeldung per UniWorX bis Do, 12.07.2018
- Abmeldung bis 13.07.2018

# Viel Erfolg!