

# Algorithmenentwurf

Kein Patentrezept zum Entwurf von Algorithmen!

- ▶ insbesondere Ableitung von Algorithmus aus Spezifikation nicht automatisierbar

Programmieren ist **kreative** Tätigkeit

- ▶ “The Art of Computer Programming” (D. Knuth)

Unterstützung durch **Algorithmenmuster**

- ▶ auch **Design Patterns** genannt
- ▶ “best practice”

# Divide and Conquer

## Definition: Divide and Conquer

Divide and Conquer ist die **rekursive** Rückführung eines zu lösenden Problems auf mehrere **identische** Problem mit **kleinerer** Eingabemenge.

**Divide and Conquer:** zu deutsch “Teile und herrsche”

## Prinzip:

- ▶ teile große Aufgabe in mehrere kleine Teilaufgaben
- ▶ rufe denselben Algorithmus rekursiv auf den Teilaufgaben auf

# Divide and Conquer

1. **Teile** gegebene Aufgabe in mehrere getrennte Teilaufgaben
  - ▶ **löse** Teilaufgaben einzeln
  - ▶ **setze** Lösung der Gesamtaufgabe aus Teillösungen zusammen
2. Wende dieselbe Technik auf jede Teilaufgabe an, dann auf deren Teilaufgaben etc., bis die Teilaufgabe so klein ist, dass Lösung explizit berechnet werden kann
3. Jede Teilaufgabe sollte **von derselben Art** sein wie die Gesamtaufgabe, so dass der gleiche Algorithmus rekursiv aufgerufen werden kann

# Divide and Conquer

```
1 Input: Aufgabe A
2
3 DivideAndConquer(A)
4   if (A klein)
5     loese A explizit;
6   else
7     teile A in Teilaufgaben  $A_1, \dots, A_n$ ;
8     DivideAndConquer( $A_1$ )
9     ...
10    DivideAndConquer( $A_n$ )
11    berechne Loesung fuer A aus Lsgn fuer  $A_1, \dots, A_n$ 
```

# Divide and Conquer

- ▶ Berechnung der **Fibonacci**-Zahlen (untypisch, da Teilprobleme Größe  $n-1$  und  $n-2$  haben).
- ▶ Binäre Suche (nur ein Teilproblem der Größe  $\leq n/2$ ).
- ▶ Karatsuba für die Multiplikation großer Zahlen.
- ▶ **MergeSort**
- ▶ **QuickSort**
- ▶ **Fast Fourier Transformation (FFT)**
- ▶ **Medianberechnung**
- ▶ ...

# Mergesort – Sortieren durch Mischen

Mergesort ist ein schneller Sortieralgorithmus der nach dem Divide and Conquer Prinzip arbeitet



John von Neumann (1945)

# Divide and Conquer: MergeSort

Sei  $L$  verkettete Liste mit  $n$  natürlichen Zahlen  $a_i \in \mathbb{N}$ .

**Aufgabe:** sortiere  $L$  in aufsteigender Reihenfolge.

Lösung mit Divide and Conquer-Muster: **MergeSort**

Idee:

- ▶ **Divide:** teile  $L$  auf in zwei gleich große Teillisten
- ▶ **Rekursion:** rufe MergeSort rekursiv für die zwei Teillisten auf
- ▶ **Conquer:** setze die Teillisten zusammen (**merge** bzw. mischen)

Wann ist Teilliste “**klein**”, d.h. wann löst man explizit?

→ Teilliste mit nur **einem** Element → sortiert!

# Divide and Conquer: MergeSort

```
1 List* mergeSort(List* a) {
2     // returns a list with elements from a sorted
3     // may delete the list pointed to by a
4     if (a->length() <= 1) return a;
5     List* b = a->half();
6     a = mergeSort(a);
7     b = mergeSort(b);
8     return merge(a,b);
9 }
```

## Divide and Conquer: MergeSort

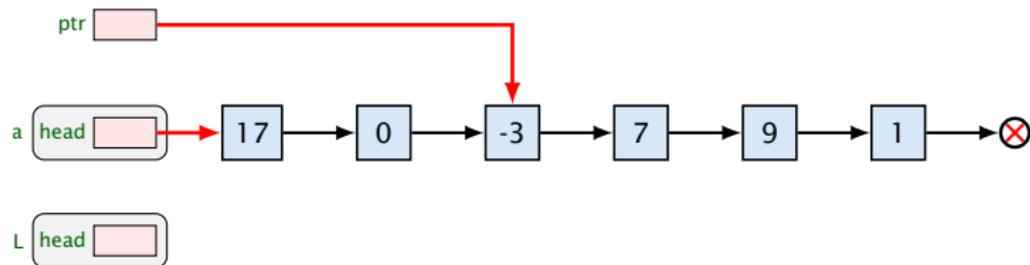
```
1 List* half() {
2     // removes elements from second half of list
3     // and returns these as a new list
4     Node* ptr = head;
5     for (int i=0; i<length()/2-1; i++)
6         ptr = ptr->next;
7     List* L = new List(ptr->next);
8     ptr->next = NULL;
9     return L;
10 }
```

# Beispiel – Halbieren

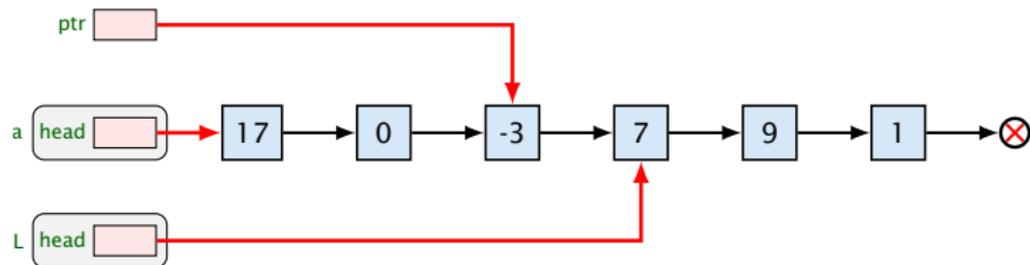


`a.half()`

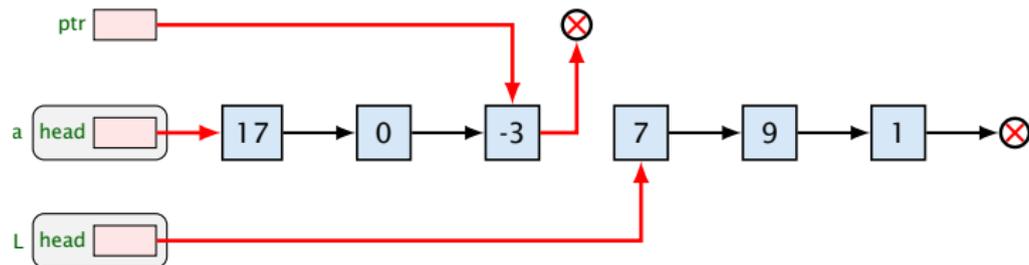
# Beispiel – Halbieren



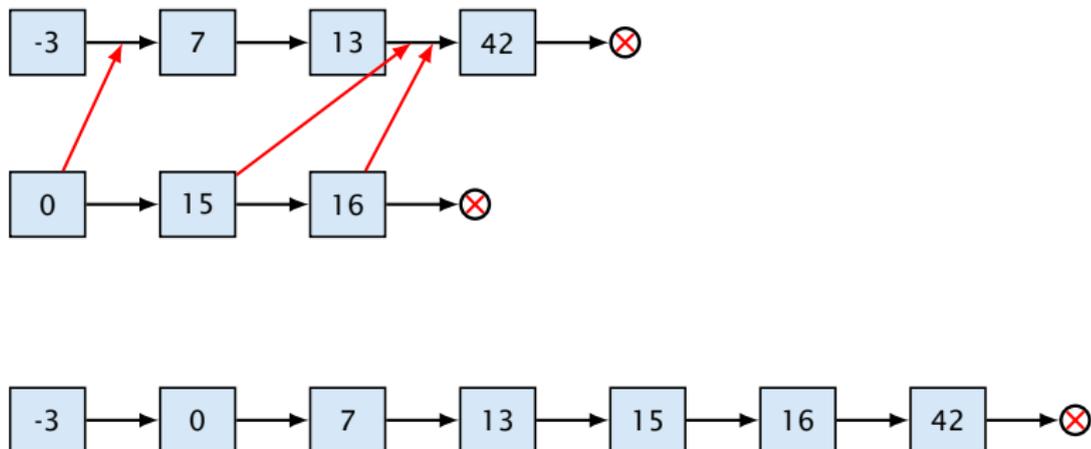
# Beispiel – Halbieren



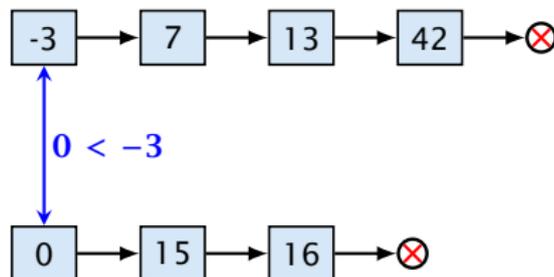
# Beispiel – Halbieren



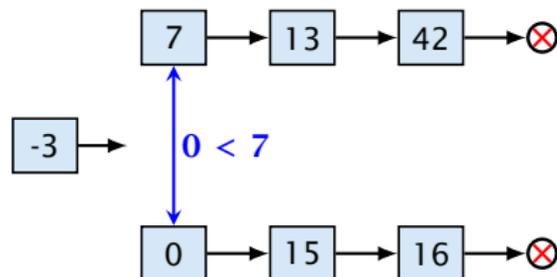
## Beispiel – Mischen



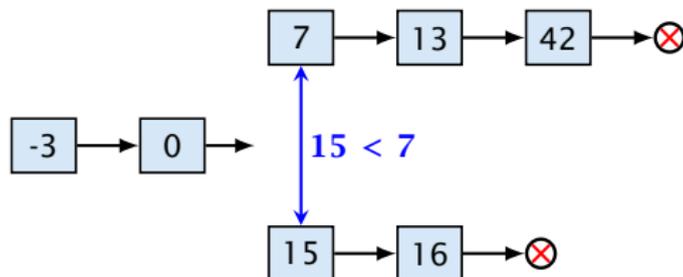
## Beispiel – Mischen



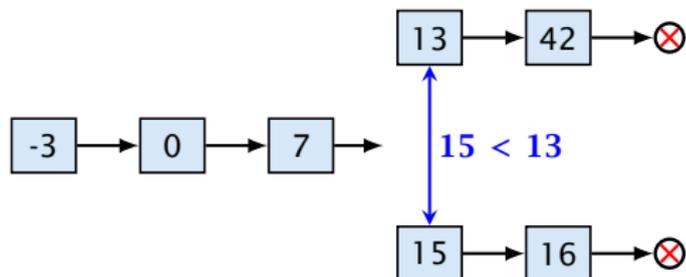
# Beispiel – Mischen



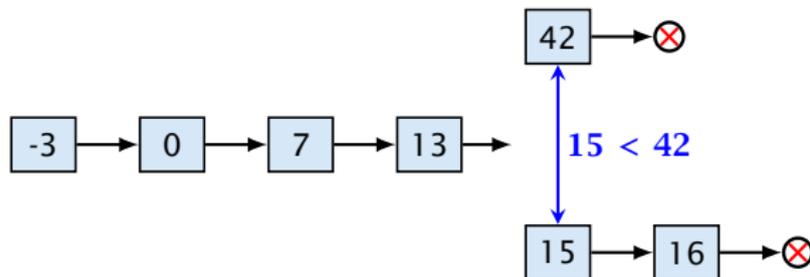
## Beispiel – Mischen



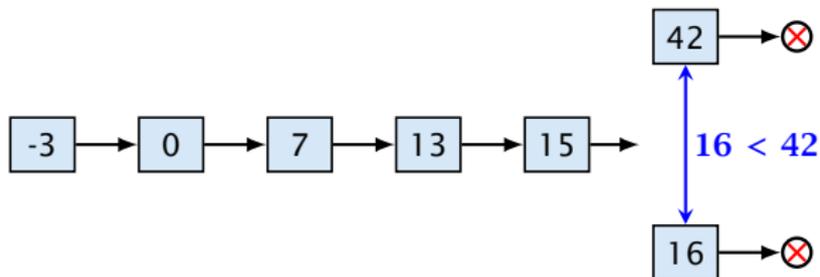
# Beispiel – Mischen



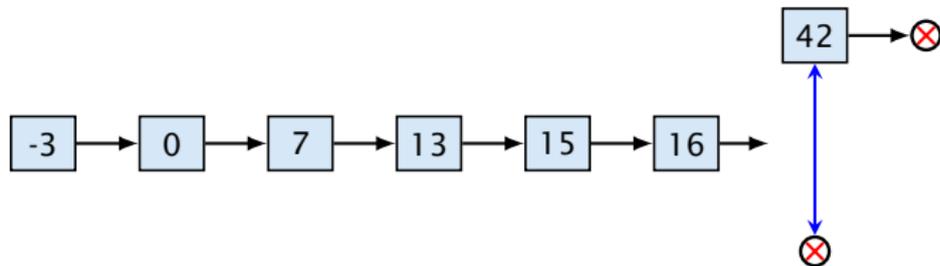
# Beispiel – Mischen



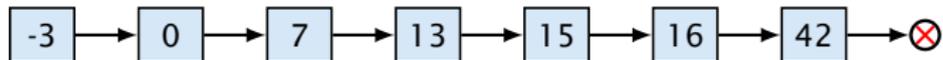
# Beispiel – Mischen



# Beispiel – Mischen



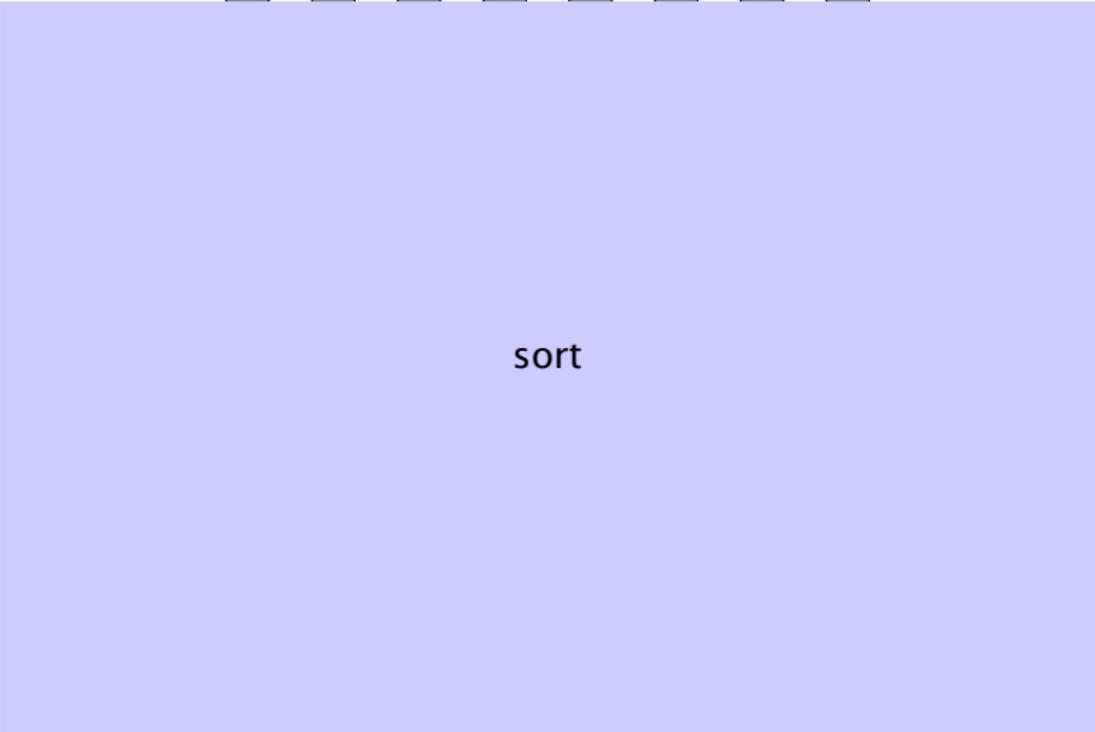
# Beispiel – Mischen



## Divide and Conquer: MergeSort

```
1 List* merge(List* a, List* b) {
2     // returns a list with elements from a and b
3     // the lists a and b may be deleted
4     if (a->empty()) { delete a; return b; }
5     if (b->empty()) { delete b; return a; }
6     List* h;
7     if (a->elementAt(0) < b->elementAt(0))
8         h = a;
9     else
10        h = b;
11    // remove element from h and recurse
12    int d = h->removeFront();
13    List* L = merge(a,b);
14    L->insertFront(d);
15    return L;
16 }
```

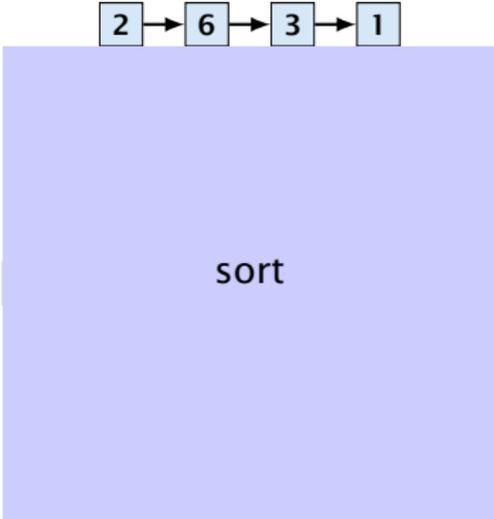
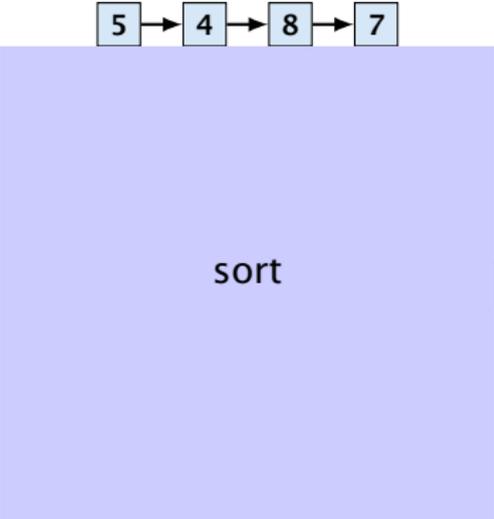
# Mergesort



# Mergesort



split



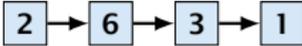
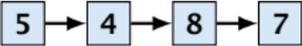
merge



# Mergesort



split



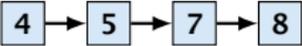
split

split



merge

merge



merge



# Mergesort



split



split

split



split

split

split

split



merge

merge

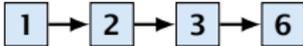
merge

merge



merge

merge



merge



## Eigenschaften

- ▶ Merge Sort benötigt zusätzlichen Speicher in Funktion `merge`; insgesamt  $n$  zusätzliche Elemente falls  $A$  Länge  $n$
- ▶ best und worst case sind identisch
- ▶ die meiste Arbeit steckt in `merge`; ein Aufruf von `merge` auf zwei Listen der Länge  $n/2$  Kosten Zeit  $\mathcal{O}(n)$ .

**Rekurrenzgleichung:**

$$T(n) = 2T\left(\frac{n}{2}\right) + \mathcal{O}(n) .$$

**Laufzeit:**  $\mathcal{O}(n \log n)$

# Sortieralgorithmen Zusammenfassung

## Insertion Sort

- ▶ in-place
- ▶ Komplexität  $\mathcal{O}(n^2)$ , best case:  $\mathcal{O}(n)$
- ▶ Komplexität  $\mathcal{O}(hn)$  falls jedes Element nur  $h$  Positionen von Zielposition entfernt

## MergeSort

- ▶ benötigt zusätzlichen Speicher
- ▶ Komplexität  $\mathcal{O}(n \log n)$

## QuickSort

- ▶ in-place
- ▶ Komplexität im Mittel  $\mathcal{O}(n \log n)$ , worst case:  $\mathcal{O}(n^2)$

# Algorithmenmuster: Greedy

**greedy** = “gierig”, “gefräßig”

## Greedy Prinzip:

- ▶ Lösung eines Problems durch **schrittweise Erweiterung** der Lösung ausgehend von Startlösung
- ▶ in jedem Schritt wähle den **bestmöglichen Schritt** (ohne Berücksichtigung zukünftiger Schritte) ⇒ **greedy**

gefundene Lösung muss nicht immer optimal sein!

# Algorithmenmuster: Greedy

```
1 Input: Aufgabe A
2
3 Greedy(A)
4     S = {}; // Loesung
5     while (S keine Loesung)
6         waehle bestmoeglichen Erweiterungsschritt s
7         erweitere S mit s
```

# Greedy: Beispiel Wechselgeld I



**Problem:** Herausgabe von Wechselgeld

- ▶ **Voraussetzung:** übliche Euro-Münzen 2€, 1€, 50ct, 20ct, 10ct, 5ct, 2ct und 1ct
- ▶ **Aufgabe:** Wechselgeld-Herausgabe mit möglichst wenig Münzen

**Beispiel:** Preis €1.11, bezahlt mit 2€-Münze. Wechselgeld: 89ct

Minimum Anzahl Münzen: 6

$$89\text{ct} = 50\text{ct} + 20\text{ct} + 10\text{ct} + 5\text{ct} + 2\text{ct} + 2\text{ct}$$

## Greedy: Beispiel Wechselgeld II

```
1 Input: Betrag b
2
3 Wechselgeld(b)
4     printf("%d = ",b);
5     count = 0;
6     while (count < b)
7         // make greedy choice
8         wähle groesste Muenze s mit count + s <= b
9         printf("%d ",s);
10        count += s;
```

**Achtung:** Abhängig vom Geldsystem liefert dieser Algorithmus nicht immer optimale Lösung!

- ▶ **Beispiel:** Münzen: 5ct, 4ct, 1ct.    **Betrag:** 8ct
- ▶ **Greedy-Lösung:**  $8\text{ct} = 5\text{ct} + 1\text{ct} + 1\text{ct} + 1\text{ct}$
- ▶ **Optimale Lösung:**  $8\text{ct} = 4\text{ct} + 4\text{ct}$

# Von Greedy lösbare Probleme

**Voraussetzungen** für Anwendbarkeit von Greedy:

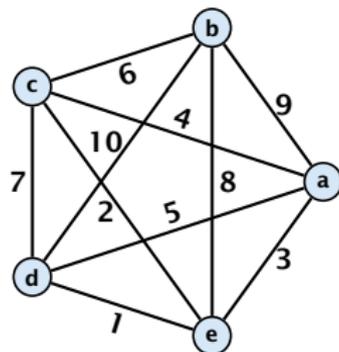
- ▶ Lösungen lassen sich schrittweise durch Hinzufügen von Elementen aufbauen, beginnend bei leerer Lösung
- ▶ Bewertungsfunktion für partielle und vollständige Lösung
- ▶ Gesucht wird eine/die optimale Lösung

# Anwendung Greedy: Glasfasernetz

**Problemstellung:** Aufbau von **möglichst billigem** Glasfasernetz zwischen  $n$  Knoten  $K_1, \dots, K_n$ , so dass alle Knoten miteinander verbunden sind (u.U. mit Umweg)

## Input:

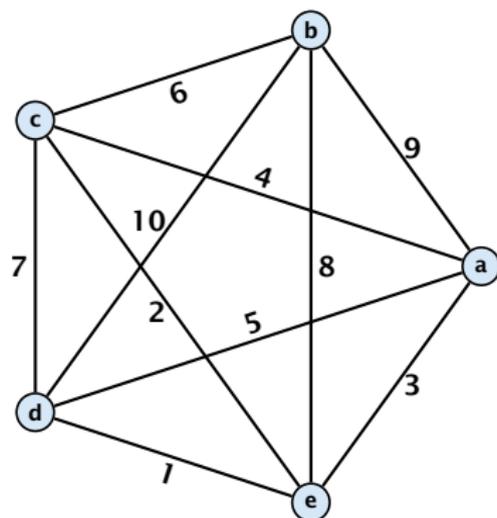
- ▶ Knoten  $a, b, c, \dots$
- ▶ Kosten  $d_{ij} > 0$  für direkte Verbindung zwischen  $i$  und  $j$  für  $i \neq j$ .



**Output:** Teilmenge aller Verbindungen, so dass

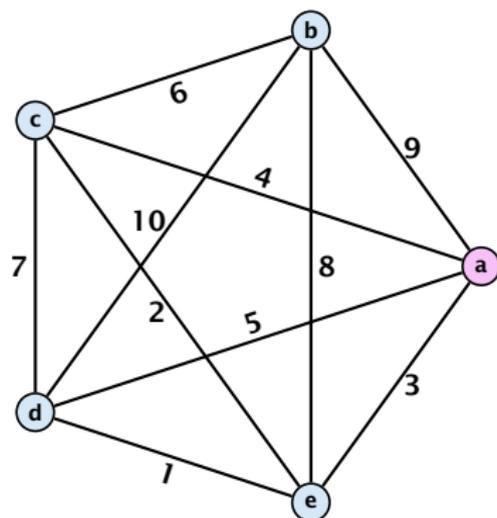
- ▶ alle Knoten verbunden sowie
- ▶ minimale Kosten

## Beispiel: Glasfasernetz



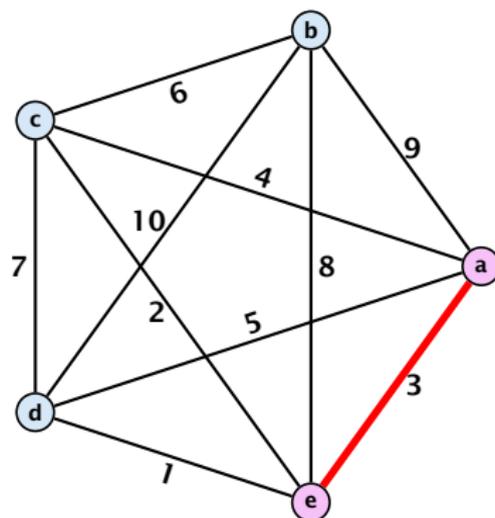
- ▶ Knoten  $a, b, c, d, e$
- ▶ Kosten  $d_{ij}$
- ▶ repräsentiert als gewichteter, ungerichteter Graph

## Beispiel: Glasfasernetz



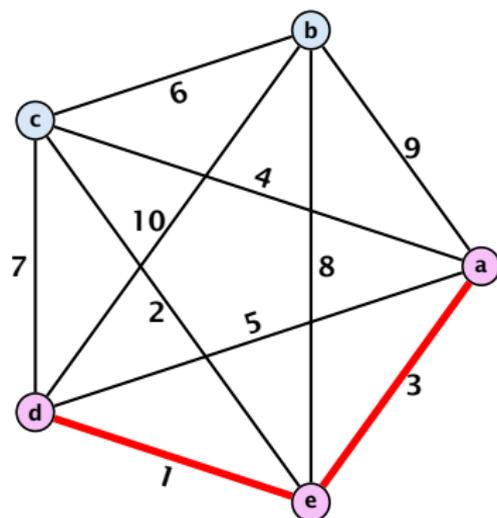
- ▶ Startknoten *a*
- ▶ beste Verbindung aus  $\{a\}$  führt zu *e* (Kosten 3)

## Beispiel: Glasfasernetz



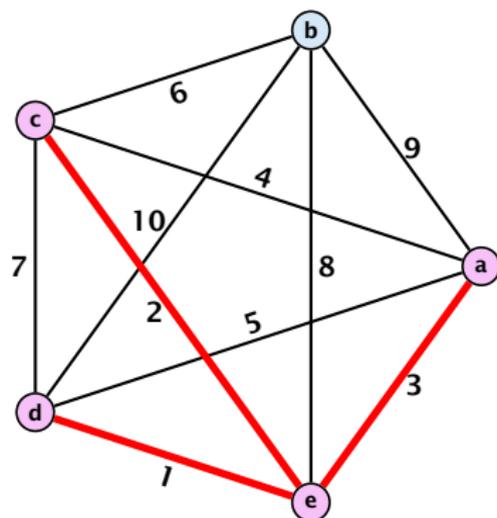
- ▶ Menge  $\{a, e\}$
- ▶ beste Verbindung aus  $\{a, e\}$  führt zu  $d$  (Kosten 1)

## Beispiel: Glasfasernetz



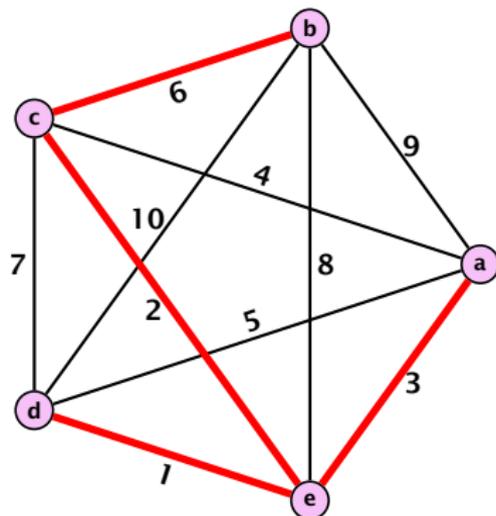
- ▶ Menge  $\{a, d, e\}$
- ▶ beste Verbindung aus  $\{a, d, e\}$  führt zu  $c$  (Kosten 2)

## Beispiel: Glasfasernetz



- ▶ Menge  $\{a, c, d, e\}$
- ▶ beste Verbindung aus  $\{a, c, d, e\}$  führt zu  $b$  (Kosten 6)

## Beispiel: Glasfasernetz



- ▶ alle Knoten verbunden
- ▶ Algorithmus fertig

**Ergebnis:** ein sog. **minimaler Spannbaum** (minimum spanning tree) des Graphen

# Glasfasernetz: Algorithmus

```
1 Input: Array K von n Knoten, Kostenfunktion d(i,j)
2 Output: minimaler Spannbaum B
3
4 Glasfasernetz(K, d)
5     B = K[1];
6     while (B nicht Spannbaum)
7         suche billigste Kante aus B;
8         fuege Kante und Knoten zu B hinzu;
```

Komplexität naiver Implementation:  $O(n^3)$

⇒ geht besser, (später in der Vorlesung)

# Algorithmenmuster: Brute Force

## Brute Force:

- ▶ erzeuge all in Frage kommenden Lösungskandidaten
- ▶ überprüfe für jeden Kandidaten ob es eine zulässige Lösung ist
- ▶ ggfs. (bei Optimierungsproblemen) bestimme zulässige Lösung mit minimalen Kosten/maximalem Profit

## Eigenschaften:

- ▶ sehr einfach zu implementieren
- ▶ häufig sehr schlechte Laufzeit

# Algorithmenmuster: Brute Force

```
1 Input: Problem P
2 Output: zulaessige Loesung fuer P
3
4 BruteForce(P)
5     S = first(P)
6     while (S != NULL)
7         if (S valid for P)
8             return S
9     S = next(P)
```

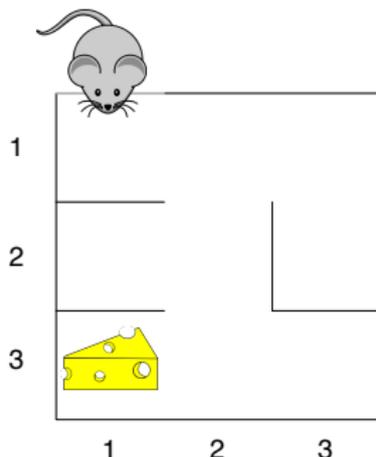
# Algorithmenmuster: Backtracking

**Backtracking:** systematische Suchtechnik, um vorgegebenen Lösungsraum vollständig abzuarbeiten

# Algorithmenmuster: Backtracking

**Backtracking:** systematische Suchtechnik, um vorgegebenen Lösungsraum vollständig abzuarbeiten

**Paradebeispiel:** Labyrinth. Wie findet Maus den Käse?

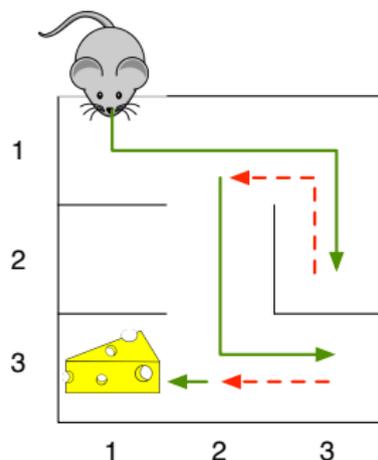


# Backtracking: Labyrinth I

**Problem:** Wie findet Maus den Käse?

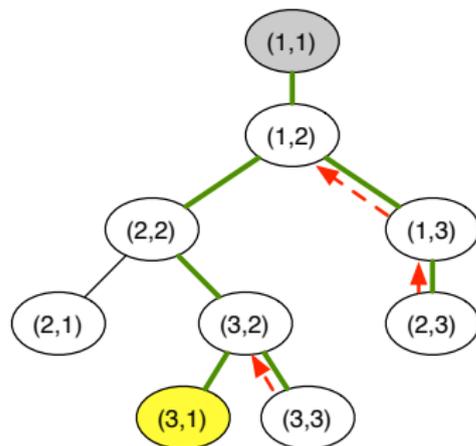
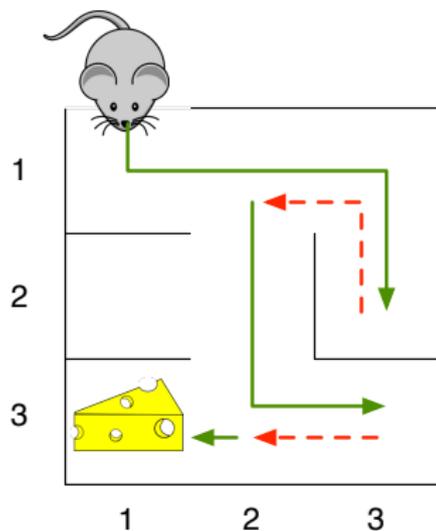
**Lösung:**

- ▶ systematisches Abgehen des Labyrinths
- ▶ Zurückgehen falls Sackgasse (daher: **Backtracking**)  
⇒ “trial and error”



# Backtracking: Labyrinth II

Mögliche Wege repräsentiert als **Baum**:



# Algorithmenmuster: Backtracking

## Voraussetzungen:

- ▶ Lösungs(teil)raum repräsentiert als **Konfiguration  $K$**
- ▶  $K_0$  ist Startkonfiguration
- ▶ jede Konfiguration  $K_i$  kann **direkt erweitert** werden
- ▶ für jede Konfiguration ist entscheidbar, ob Lösung

```
1 Input: Konfiguration K
2
3 Backtrack(K)
4     if (K Loesung)
5         gib K aus;
6     else
7         foreach (Erweiterung  $K'$  von K)
8             Backtrack( $K'$ )
```

# Algorithmenmuster: Backtracking

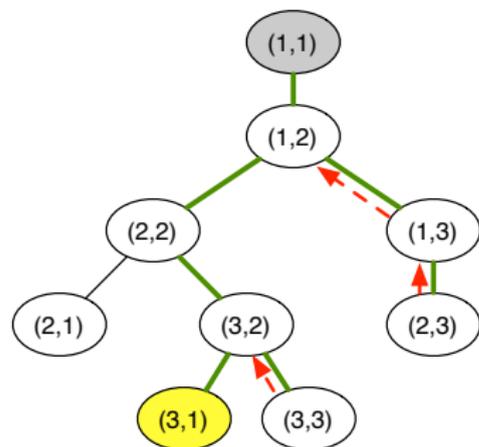
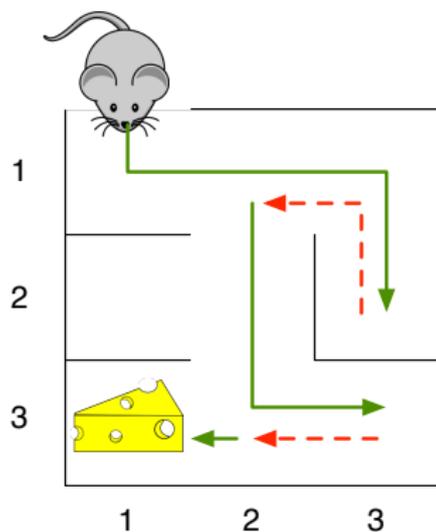
## Voraussetzungen:

- ▶ Lösungs(teil)raum repräsentiert als **Konfiguration  $K$**
- ▶  $K_0$  ist Startkonfiguration
- ▶ jede Konfiguration  $K_i$  kann **direkt erweitert** werden
- ▶ für jede Konfiguration ist entscheidbar, ob Lösung

```
1 Input: Konfiguration K
2
3 Backtrack(K)
4     if (K Loesung)
5         gib K aus;
6     else
7         foreach (Erweiterung  $K'$  von K)
8             Backtrack( $K'$ )
```

⇒ **initialer Aufruf** mittels  $\text{Backtrack}(K_0)$

# Backtracking: Konfigurationen



**Konfiguration** z.B. repräsentiert als **Pfad** im Baum

# Backtracking: Eigenschaften

**Terminierung** von Backtracking:

- ▶ nur wenn Lösungsraum **endlich**
- ▶ nur wenn sichergestellt dass Konfigurationen **nicht wiederholt getestet** werden

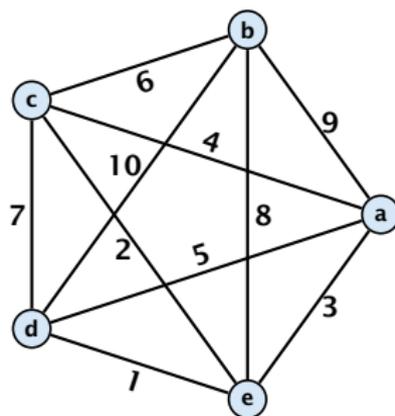
**Komplexität** von Backtracking:

- ▶ direkt abhängig von Größe des Lösungsraums
- ▶ meist **exponentiell**, also  $O(2^n)$ , oder schlimmer!
- ▶ nur für kleine Probleme wirklich anwendbar

# Backtracking Beispiel: Traveling Salesman

## Traveling Salesman Problem:

- ▶  $n$  Städte
- ▶ finde **kürzeste Rundreise**, die alle Städte exakt einmal besucht (ausser Start- und Zielort (identisch))



⇒ Lösung z.B. mit Algorithmenmuster Backtracking

# Traveling Salesman Problem: Algorithmus mit Backtracking

```
1 Input: n Staedte, Rundreise trip
2
3 TSP(trip)
4   if (trip besucht jede Stadt)
5       erweitere trip um Reise zum Standort;
6       gebe trip und Kosten aus;
7   else
8       foreach (bislang unbesuchte Stadt s)
9           trip' = trip erweitert um s
10          TSP(trip');
```

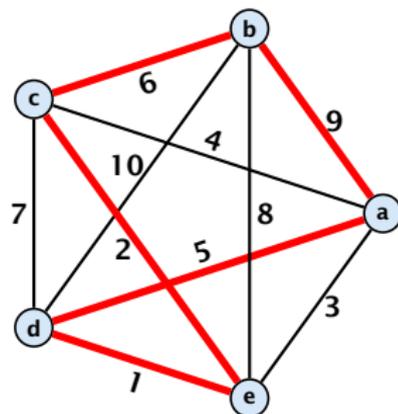
# Traveling Salesman Problem: Beispiel

Bei  $n$  Städten mit fixiertem Start-/Zielort gibt es  $(n - 1)!$  Rundreisen (hier: 5 Städte  $\Rightarrow 4! = 24$  Rundreisen).

Laufzeit von **TSP** hier ist  $\mathcal{O}((n - 1)!)$

hier: kürzeste Rundreise hat Länge 23

- ▶ z.B. über Route  $a \rightarrow b \rightarrow c \rightarrow e \rightarrow d \rightarrow a$



# Backtracking Beispiel: Acht-Damen-Problem

## Acht-Damen-Problem:

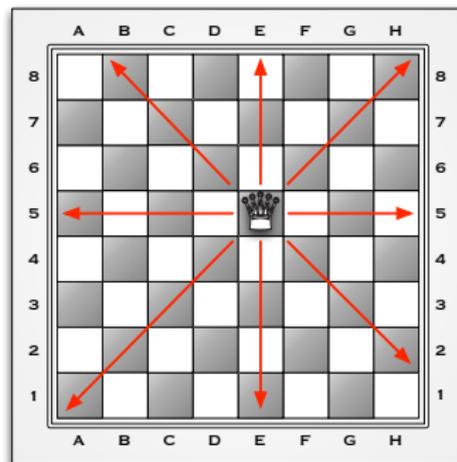
- ▶ suche alle Konfigurationen von 8 Damen auf Schachbrett
- ▶ so dass keine Dame eine andere bedroht

# Backtracking Beispiel: Acht-Damen-Problem

## Acht-Damen-Problem:

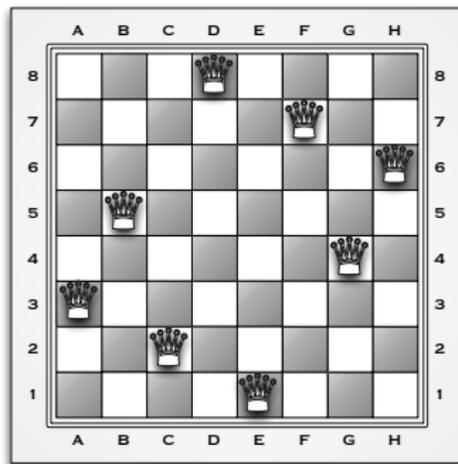
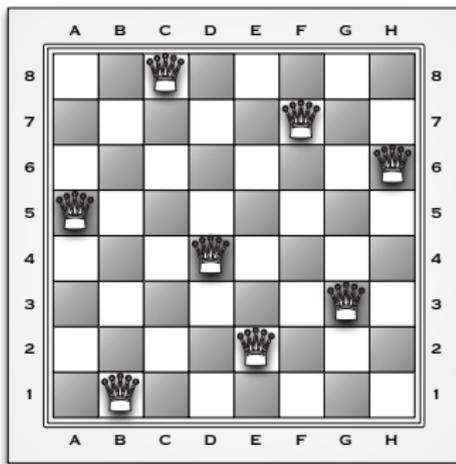
- ▶ suche alle Konfigurationen von 8 Damen auf Schachbrett
- ▶ so dass keine Dame eine andere bedroht

## Dame auf Schachbrett:



# Acht-Damen-Problem

Zwei der möglichen Lösungen:



**Beobachtung:** jeweils nur eine Dame pro Zeile/Spalte  
→ Lösung z.B. mit Algorithmenmuster Backtracking

# Acht-Damen-Problem: Algorithmus mit Backtracking

```
1 Input: Zeilenindex i
2
3 AchtDamen(i)
4   if (i == 9)
5     gib Loesung aus;
6     return;
7   for h=1 to 8
8     if (Feld in Zeile i, Spalte h nicht bedroht)
9       setze Dame auf Feld (i,h);
10      AchtDamen(i+1);
11      entferne Dame von Feld (i,h);
```

# Acht-Damen-Problem

- ▶ es gibt **92 Lösungen** für das Acht-Damen-Problem
- ▶ das Problem lässt sich auf  $n$  Damen auf einem  $n \times n$  Schachbrett ausweiten
  - ▶ Anzahl Lösungen wächst stark
  - ▶ z.B. für  $n = 13$  gibt es **73712** Lösungen
- ▶ ähnliche Spiele, wie z.B. **Sudoku**, lassen sich entsprechend lösen

## Dynamisches Programmieren

- ▶ einsetzbar für Probleme, deren optimale Lösung sich aus optimalen Lösungen von Teilproblemen zusammensetzt (z.B. Rekursion)

### Prinzip:

- ▶ statt Rekursion berechnet man vom kleinsten Teilproblem **“aufwärts”**
- ▶ Zwischenergebnisse werden in **Tabellen** gespeichert

# Beispiel: Fibonacci Zahlen

## Fibonacci Folge

Die **Fibonacci Folge** ist eine Folge natürlicher Zahlen  $f_1, f_2, f_3, \dots$ , für die gilt

$$f_n = f_{n-1} + f_{n-2} \quad \text{für } n \geq 3$$

mit Anfangswerten  $f_1 = 1, f_2 = 1$ .

- ▶ eingesetzt von Leonardo Fibonacci zur Beschreibung von Wachstum einer Kaninchenpopulation
- ▶ Folge lautet: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...
- ▶ berechenbar z.B. via Rekursion



# Beispiel: Fibonacci Funktion

**Input:** Index  $n$  der Fibonaccifolge

**Output:** Wert  $f_n$

```
fib(n)
  if (n == 1 || n == 2) {
    return 1;
  }
  else {
    // rekursiver Aufruf
    return fib(n-1) + fib(n-2);
  }
```

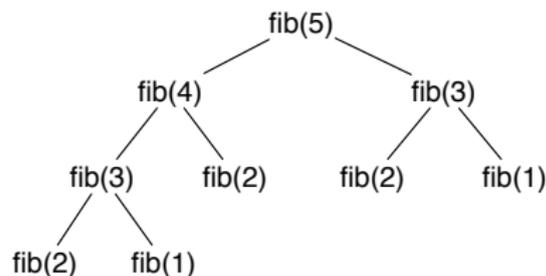
# Beispiel: Fibonacci Funktion

Input: Index  $n$  der Fibonaccifolge

Output: Wert  $f_n$

```
fib(n)
  if (n == 1 || n == 2) {
    return 1;
  }
  else {
    // rekursiver Aufruf
    return fib(n-1) + fib(n-2);
  }
```

Aufrufstruktur für fib(5):



# Fibonacci Funktion: dynamisch programmiert

```
1 Input: Index n der Fibonaccifolge
2 Output:  $F_n$ 
3
4 FibDyn(n)
5     fib = new long[n+1];
6     fib[1] = 1;
7     fib[2] = 1;
8     for (k=3; k <= n; k++)
9         fib[k] = fib[k-1] + fib[k-2];
10    res = fib[n];
11    delete fib;
12    return fib[n];
```

- Komplexität dynamisch programmiert:  $O(n)$