# 1 Advanced Designs

## 1.1 Dynamische Programmierung

**Anwendung**
Anwendung, wenn sich Teilprobleme überlappen:

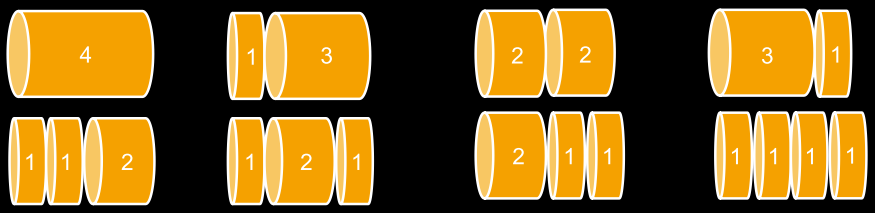
1. Wir charakterisieren die Struktur einer optimalen Lösung
2. Wir definieren den Wert einer optimalen Lösung rekursiv
3. Wir berechnen den Wert einer optimalen Lösung (meist bottom-up Ansatz)
4. Wir konstruieren eine zugehörige optimale Lösung aus berechneten Daten

### 1.1.1 Stabzerlegungsproblem

**Ausgangsproblem:** Stangen der Länge $n$ cm sollen so zerschnitten werden, dass der Erlös $r_n$ maximal ist, indem die Stange in kleinere Stäbe geschnitten wird.

| Länge $i$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Preis $p_i$ | 0 | 1 | 5 | 8 | 9 | 10 | 17 | 17 | 20 | 24 | 30 |

**Beispiel:** Gesamtstange hat Länge 4. Welchen Erlös kann man max. erhalten?




Optimaler Erlös: zwei 2cm lange Stücke ($5 + 5 = 10$)

**Aufteilung der Stange**

- Stange mit Länge $n$ kann auf $2^{n-1}$ Weisen zerlegt werden
- Position $i$: Distanz vom linken Ende der Stange
- Aufteilung in $k$ Teilstäbe ($1 \leq k \leq n$)
- optimale Zerlegung: $n = i_1 + i_2 + ... + i_k$
- maximaler Erlös: $r_n = p_{i_1} + p_{i_2} + ... + p_{i_k}$
- z.B.: $r_4 = 10$ (siehe oben)

### Rekursive Top-Down Implementierung

CUT-ROD(p,n) // p Preis-Array, n Stangenlänge

```
IF n == 0
    return 0;
q = −∞;
FOR i = 1 TO n  // nicht Start bei 0, sonst kein Rekursionsschritt
    q = max(q, p[i] + CUT-ROD(p, n - i));
return q;
```

### Stabzerlegung via Dynamischer Programmierung:

**Ziel** Mittels dynamischer Programmierung wollen wir CUT-ROD in einen effizienten Algorithmus verwandeln.

**Bemerkung** Naiver rekursiver Ansatz ist **ineffizient**, da dieser immer wieder diesselben Teilprobleme löst.

**Ansatz** Jedes Teilproblem nur einmal lösen. Falls die Lösung eines Teilproblems nochmal benötigt wird, schlagen wir diese nach.

- Reduktion von exponentieller auf polynomielle Laufzeit.
- Dynamische Programmierung wird zusätzlichen Speicherplatz benutzen um Laufzeit einzusparen.

### Rekursiver Top-Down-Ansatz mit Memoisation:

**Idee — Rekursiver Top-Down-Ansatz mit Memoisation** Speicherung der Lösungen der Teilprobleme

Laufzeit: $\Theta(n^2)$

MEMOIZED-CUT-ROD(p, n)

```
Let r[0...] be new array
FOR i = 0 TO n
    r[i] = −∞
return MEMOIZED-CUT-ROD-AUX(p, n, r)
```

MEMOIZED-CUT-ROD-AUX(p, n, r) // r new Array

```
IF r[n] ≥ 0                          // Abfrage ob vorhanden
    return r[n]
IF n == 0
    q = 0
ELSE
    q = −∞
    FOR i = 1 to n
    q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n - i, r))
r[n] = q                              // Abspeichern
return q
```

**Bottom-Up Ansatz:**

- Laufzeit: $\Theta(n^2)$
- Sortieren der Teilprobleme nach ihrer Größe und lösen in dieser Reihenfolge
- Alle Teilprobleme kleiner als das momentane Problem sind bereits gelöst

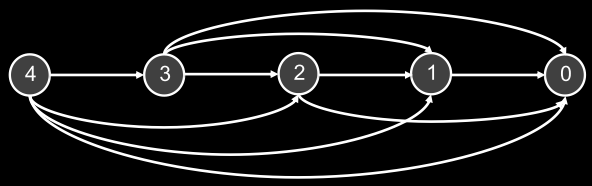
BOTTOM-UP-CUT-ROD(p, n)

```
1  Let r[0...n] be a new array
2  r[0] = 0
3  FOR j = 1 TO n
4      q = −∞
5      FOR i = 1 TO j
6          q = max(q, p[i] + r[j - i])
7      r[j] = q
8  return r[n]
```

EXTENDED-BOTTOM-UP-CUT-ROD(p, n)

```
1  Let r[0...n] and s[0...n] be  new arrays
2  r[0] = 0, s[0] = 0
3  FOR j = 1 TO n
4      q = −∞
5      FOR i = 1 TO j
6          IF q < p[i] + r[j-i]
7              q = p[i] + r[j - i]
8              s[j] = i
9      r[j] = q
10 return r and s
```

Teilproblemgraph ($i \to j$ bedeutet, dass Berechnung von $r_i$ den Wert $r_j$ benutzt)



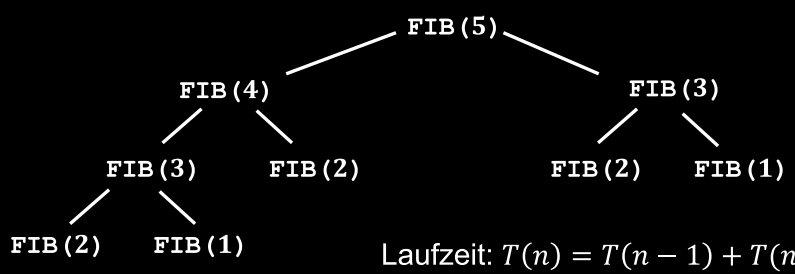

**Fibonacci-Zahlen**

- $F_1 = F_2 = 1$
- $F_n = F_{n-1} + F_{n-2}$

*Naiver rekursiver Algorithmus:*

FIB(n)

```
1  IF n ≤ 2
2      f = 1;
3  ELSE
4      f = FIB(n-1) + FIB(n-2);
5  return f;
```




Laufzeit: $T(n) = T(n-1) + T(n-2) + \Theta(1)$
$\Longrightarrow \Theta(2^n)$

Gleiche Teilprobleme werden wieder mehrmals gelöst

*Rekursiver Algorithmus mit Memoisation:*

- Wieder Abspeichern von Teilproblemen um Laufzeit einzusparen
- Laufzeit: $\Theta(n)$

MEMOIZED-FIB(n)

```
Let m[0...n-1] be a new array
FOR i = 0 TO n - 1
    m[i] = 0
return MEMOIZED-FIB-AUX(n, m)
```

MEMOIZED-FIB-AUX(n, m)

```
IF m[n-1] != 0
    return m[n-1];      // Auslesen von gespeicherten Werten
IF n ≤ 2
    f = 1;
ELSE
    f = MEMOIZED-FIB-AUX(n-1, m) + MEMOIZED-FIB-AUX(n-2, m);
m[n-1] = f;
return f;
```

*Bottom-Up Algorithmus:*

Hier wieder Berechnen aller Teilprobleme von unten beginnend

BOTTOM-UP-FIB(n)

```
Let m[0...n] be a new array
FOR i = 1 TO n
    IF i ≤ 2
        f = 1;
    ELSE
        f = m[i-1] + m[i-2];
    m[i] = f;
return m[n];
```

## 1.2 Greedy-Algorithmus

**Idee — Greedy-Algorithmus**

- Trifft stets die Entscheidung, die in diesem Moment am besten erscheint
- Trifft **lokale** optimale Entscheidung (evtl. nicht global die Beste)

### 1.2.1 Aktivitäten-Auswahl-Problem

**Definition — Aktivitäten-Auswahl-Problem**

- 11 anstehende Aktivitäten $S = \{a_1, ..., a_{11}\}$
- Startzeit $s_i$ und Endzeit $f_i$, wobei $0 \leq s_i < f_i < \infty$
- Aktivität $a_i$ findet im halboffenen Zeitintervall $[s_i, f_i)$ statt
- Zwei Aktivititäten sind kompatibel, wenn sich deren Zeitintervalle nicht überlappen

| $i$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $s_i$ | 1 | 3 | 0 | 5 | 3 | 5 | 6 | 8 | 8 | 2 | 12 |
| $f_i$ | 4 | 5 | 6 | 7 | 9 | 9 | 10 | 11 | 12 | 14 | 16 |

Aktivitäten: $\{a_3, a_9, a_{11}\}$
Aktivitäten: $\{a_1, a_4, a_8, a_{11}\}$
Aktivitäten: $\{a_2, a_4, a_9, a_{11}\}$

#### Ansatz mittels dynamischer Programmierung

- Menge von Aktivitäten, die starten nachdem $a_i$ endet und enden, bevor $a_j$ startet
  $S_{ij} = \{a \in S, a = (s, f) : s \geq f_i, f < s_j\}$
- Definiere maximale Menge $A_{ij}$ von paarweise kompatiblen Aktivitäten in $S_{ij}$.
  $c[i, j] = |A_{ij}|$
- Optimale Lösung für Menge $S_{ij}$ die Aktivitäten $a_k$ enthält:
  $c[i, j] = max_{a_k \in S_{ij}}\{c[i, k] + c[k, j] + 1\}$ (0, falls $S_{ij} = \emptyset$)

#### Greedy-Wahl

- lokal die beste Wahl
- Auswahl der Aktivität mit geringster Endzeit (möglichst viele freie Ressourcen)
- Also hier Teilprobleme, die nach $a_1$ starten
- $S_k = \{a_i \in S : s_i \geq f_k\}$: Menge an Aktivitäten, die starten, nachdem $a_k$ endet
- *Optimale-Teilstruktur-Eigenschaft*
  Wenn $a_1$ in optimaler Lösung enthalten ist, dann besteht optimale Lösung zu ursprünglichem Problem aus Aktivität $a_1$ und allen Aktivitäten zur einer optimalen Lösung des Teilproblems $S_1$

### Rekursiver Greedy-Algorithmus

> **Voraussetzung**: Aktivitäten sind monoton steigend nach der Endzeit sortiert

Laufzeit: $\Theta(n)$

RECURSIVE-ACTIVITY-SELECTOR(s,f,k,n)

```
// s Anfangszeitenarray, f Endzeitenarray,
// k Index von Teilproblem, n Größe Anfangsproblem
m = k + 1;
WHILE m ≤ n and s[m] < f[k]  // Suche nach erster Kompatibilität
    m = m + 1;
IF m ≤ n
    // Ausgabe des Elements und Berechnung weiterer Aktivitäten
    return {a_m} ∪ RECURSIVE-ACTIVITY-SELECTOR(s,f,m,n)
ELSE
    return ∅
```

### Iterativer Greedy-Algorithmus

> **Voraussetzung**: Aktivitäten sind monoton steigend nach der Endzeit sortiert

Laufzeit: $\Theta(n)$

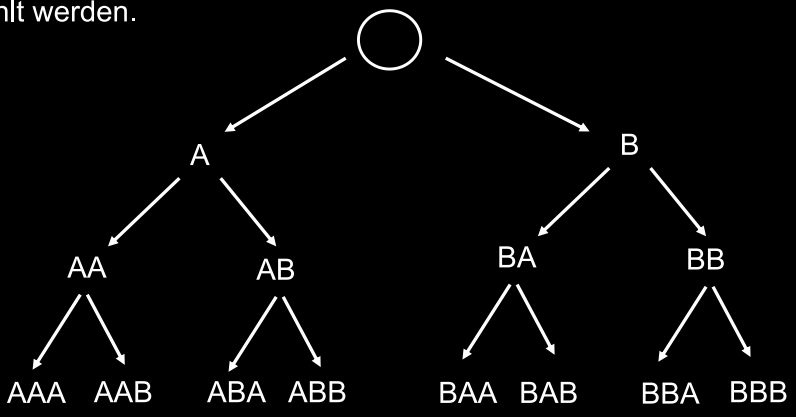
GREEDY-ACTIVITY-SELECTOR(s,f)

```
n = s.length;
A = {a_1};
k = 1;          // Index des zuletzt hinzugefügten Elements in A
FOR m = 2 TO n
    IF s[m] ≥ f[k]       // Findet zuerst endende Aktivität in Menge
        A = A ∪ {a_m};   // Fügt diese hinzu, falls kompatibel
        k = m;
return A;
```

## 1.3 Backtracking

**Suchbaum - Baum der Möglichkeiten**
Darstellung aller für ein Problem bestehenden Möglichkeiten

Problem: Aus den Buchstaben A, B soll dreimal nacheinander einer gewählt werden.




Der Suchraum ist die Menge aller für ein Problem bestehende Möglichkeiten.

**Idee — Backtracking**

- Lösung finden via Trial and error
- Schrittweises Herantasten an die Gesamtlösung
- Falls Teillösung inkorrekt → Gehe einen Schritt zurück und probiere eine andere Möglichkeit
- Voraussetzung:
  - Lösung setzt sich aus Komponenten zusammen (Sudoku, Labyrinth,..)
  - Mehrere Wahlmöglichkeiten für jede Komponente
  - Teillösung kann auf Korrektheit getestet werden

**Allgemeiner Backtracking-Algorithmus**

BACKTRACKING(A, s)

```
IF alle Komponenten richtig gesetzt
    return true;
ELSE
    WHILE auf aktueller Stufe gibt es Wahlmöglichkeiten
        wähle einen neuen Teillösungsschritt
        Teste Lösungsschritt gegen vorliegende Einschränkungen
        IF keine Einschränkung THEN
            setze die Komponente
        ELSE
            Auswahl(Komponente) rückgängig machen
        BACKTRACKING(A, s + 1)
```

**Damenproblem**
Auf einem Schachbrett der Größe $n \cdot n$ sollen $n$ Damen so positioniert werden, dass sie sich gegenseitig nicht schlagen können. Wie viele Möglichkeiten gibt es, $n$ Damen so aufzustellen, dass keine Damen eine andere schlägt.

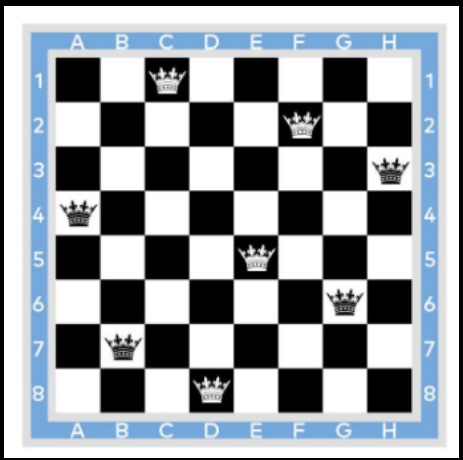



Abbildung 1: Beispielhafte Darstellung des Damenproblems

- $n = 8$ : 4 Milliarden Positionierungen
- Optimierte Suche: In jeder Zeile/Spalte nur eine Dame
- Reduziert Problem auf 40.000 Positionierungen (ohne Diagonale)

PLACE-QUEENS(Q,r) // Q Array von Damenpositionen, r Index der ersten leeren Zeile

```
IF r == n
    return Q;
ELSE
    FOR j = 0 TO n - 1 // Mögliche Positionierungen
        legal = true;
        FOR i = 0 TO r - 1  // Evaluation der mgl. Bedrohungen
            IF (Q[i] == j) OR (Q[i == j + r - i]) OR (Q[i] == j - r + i)
                legal = false;
        IF legal == true
            Q[r] = j;
            PLACE-QUEENS(Q, r + 1)
```

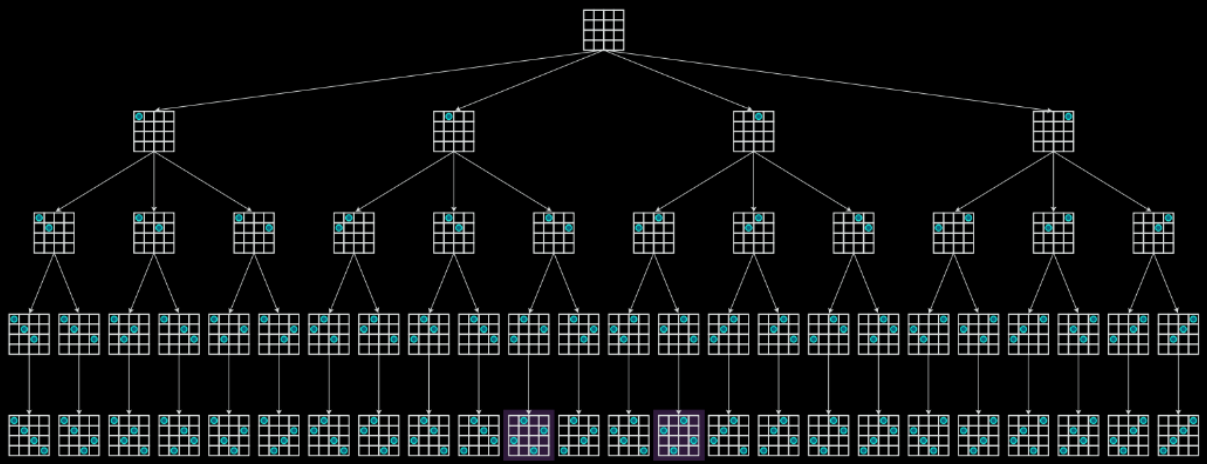


Abbildung 2: Mögliche Pfade von Place-Queens

## 1.4 Metaheuristiken

### Optimierungsproblem

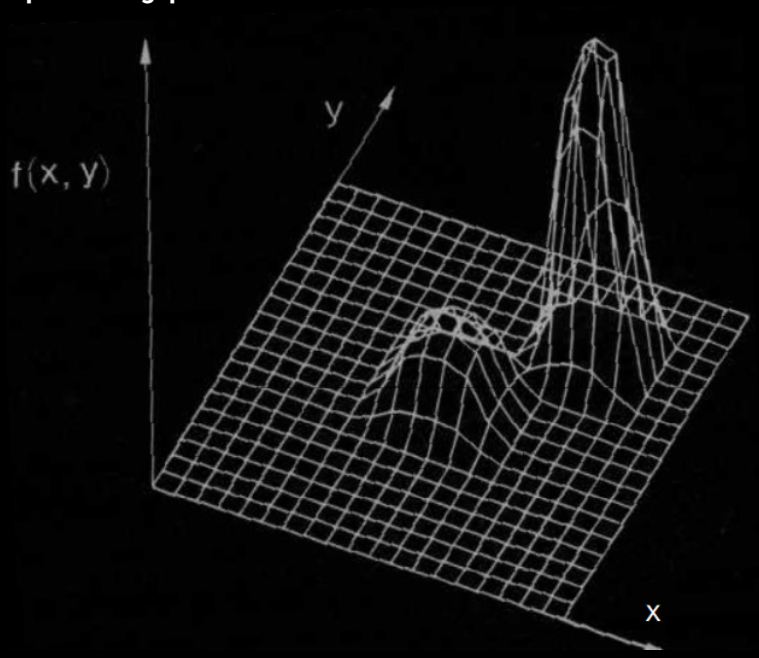



Abbildung 3: Beispiel Optimierungsproblem

- Lösungsstrategien:
  - Exakte Methode
  - Approximationsmethode
  - Heuristische Methode
- Einschränkungen
  - Antwortzeit
  - Problemgröße

  ⇒ exkludieren oft exakte Methoden

### Heuristik

- Technik um Suche zur Lösung zu führen
- Metaheuristik (Higher-Level-Strategie)
  - soll z.B. Hängenbleiben bei lokalem Maxima verhindern
- *Leiten einer Suche*
  1. Finde eine Lösung (z.B. mit Greedy-Algorithmus)
  2. Überprüfe die Qualität der Lösung
  3. Versuche eine bessere Lösung zu finden
     - Herausfinden in welcher Richtung bessere Lösung evtl. liegt
     - ggf. Wiederholung dieses Prozesses
- *Finden einer besseren Lösung*
  - Modifikation der Lösung durch erlaubte Operationen
  - Dadurch erhalten wir Nachbarschaftslösungen

    ⇒ Suche nach besseren Lösungen in der Nachbarschaft

### Rucksackproblem

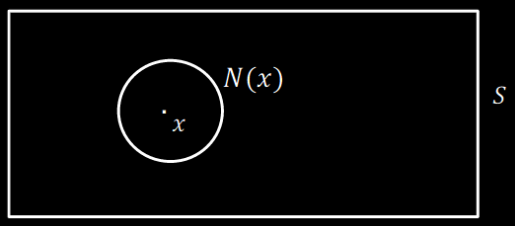
> **Ziel:** Höchster Wert der Gegenstände im Rucksack

Beispiel:

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| Wert | 79 | 32 | 47 | 18 | 26 | 85 | 33 | 40 | 45 |
| Größe | 85 | 26 | 48 | 21 | 22 | 95 | 43 | 45 | 55 |

Abbildung 4: Beispielgegenstände für Rucksackproblem

- Rucksack hat eine Kapazität von 101, 9 verschiedene Gegenstände
- Beispiellösung: Gegenstand 3 + 5 (Wert 73, Größe 70)
- Nachbarschaftslösungen:
  - Gegenstände 2,3 und 5: Wert 105, Größe 96
  - Gegenstände 1,3 und 5: Wert 152, Größe 155 (Gewichtsüberschreitung problematisch)
  - Gegenstand 3: Wert 47, Größe 48




**Nachbarschaft:**

- Suchraum $S$ kann sehr groß sein
- Einschränkung des Suchraums in der Nähe der Startlösung x
- Distanzfunktion $d : SxS \rightarrow \mathbb{R}$
- Nachbarschaft: $N(x) = \{y \in S : d(x, y) \leq \epsilon\}$

### Zufällige Suche

**Idee — Zufällige Suche**

- Suche nach globalem Optimum
- Anwenden der Technik auf **aktuelle** Lösung im Suchraum
- Wahl einer neuen zufälligen Lösung in jeder Iteration
- Falls die neue Lösung besseren Wert liefert ⇒ als neue **aktuelle** Lösung setzen
- Terminierung, falls keine weiteren Verbesserungen auffindbar oder Zeit vorbei

*Code:*

RANDOM-SEARCH

```
best <- irgendeine initiale zufällige Lösung
REPEAT
    S <- zufällige Lösung // von "best" unabhängig
    IF (Quality(S) > Quality(best)) THEN
        best <- S
UNTIL best ist die ideale Lösung oder Zeit ist vorbei
return best
```

Nachteile
- Potentiell lange Laufzeit
- Laufzeit abhängig von der initialien Konfiguration

Vorteile
- Algorithmus **kann** beim globalen Optimum terminieren

### Bergsteigeralgorithmus

**Idee — Bergsteigeralgorithmus**

- Nutzung einer iterativen Verbesserungstechnik
- Anwenden der Technik auf **aktuelle** Lösung im Suchraum
- Auswahl einer neuen Lösung aus Nachbarschaft in jeder Iteration
- Falls diese besseren Wert liefert, überschreiben der **aktuellen** Lösung
- Falls nicht, Wahl einer anderen Lösung aus Nachbarschaft
- Terminierung, falls keine weiteren Verbesserungen auffindbar oder Zeit vorbei

*Code*

HILL-CLIMBER

```
T <- Distribution von möglichen Zeitintervallen
S <- irgendeine initiale zufällige Lösung
best <- S
REPEAT
    time <- zufälliger Zeitpunkt in der Zukunft aus T
    REPEAT
        wähle R aus der Nachbarschaft von S
        IF Quality(R) > Quality(S) THEN
            S <- R
    UNTIL S ist ideale Lösung oder time ist erreicht oder totale Zeit erreicht
    IF Quality(S) > Quality(best) THEN
        best <- S
    S <- irgendeine zufällige Lösung
UNTIL best ist die ideale Lösung oder totale Zeit erreicht
return best
```

Nachteile
- Algorithmus terminiert in der Regel bei lokalem Optimum
- Keine Auskunft, inwiefern sich lokale Lösung von Globaler unterscheidet
- Optimum abhängig von Initialkonfiguration

Vorteile
- Einfach anzuwenden

### Iterative lokale Suche

**Idee — Iterative lokale Suche**

- Suche nach anderen lokalen Optima bei Fund eines lokalen Optimas
- Lösungen nur in der Nähe der "Homebase"
- Entscheidung, ob neue oder alte Lösung
- Bergsteigeralgo zu Beginn, danach aber großen Sprung um anderes Optimum zu finden

*Code*

ITERATIVE-LOCAL-SEARCH

```
T <- Distribution von möglichen Zeitintervallen
S <- irgendeine initiale zufällige Lösung
H <- S      // Wahl des Homebasepunktes
best <- S
REPEAT
    time <- zufälliger Zeitpunkt in der Zukunft aus T
    REPEAT
        wähle R aus der Nachbarschaft von S
        IF Quality(R) > Quality(S) THEN
            S <- R
    UNTIL S ist ideale Lösung oder time ist erreicht oder totale Zeit erreicht
    IF Quality(S) > Quality(best) THEN
        best <- S
    H <- NewHomeBase(H,S)
    S <- Perturb(H)
UNTIL best ist die ideale Lösung oder totale Zeit erreicht
return best
```

**Perturb**

- ausreichend weiter Sprung (außerhalb der Nachbarschaft)
- Aber nicht soweit, dass es eine zufällige Wahl ist

**NewHomeBase**

- wählt die neue Startlösung aus
- Annahme neuer Lösungen nur, wenn die Qualität besser ist

### Simulated Annealing

**Idee — Simulated Annealing**

- Wenn neue Lösung besser, dann wird diese immer gewählt
- Wenn neue Lösung schlechter, wird diese mit gewisser Wahrscheinlichkeit gewählt: $Pr(R, S, t) = e^{\frac{Quality(R)-Quality(S)}{t}}$
- Der Bruch ist negativ, da $R$ schlechter ist als $S$

SIMULATED-ANNEALING

```
t <- Temperatur, initial eine hohe Zahl
S <- irgendeine initiale zufällige Lösung
best <- S
REPEAT
    wähle R aus der Nachbarschaft von S
    IF Quality(R) > Quality(S) oder zufälliges
                Z ∈ [0,1] < e^((Quality(R)-Quality(S))/t) THEN
            S <- R
    dekrementiere t
    IF Quality(S) > Quality(best) THEN
        best <- S
UNTIL best ist die ideale Lösung oder Temperatur ≤ 0
return best
```

### Tabu-Search

**Idee — Tabu-Search**

- Speichert alle bisherigen Lösungen und Liste und nimmt diese nicht nochmal
- Kann sich jedoch wieder von der optimalen Lösung entfernen
- Tabu List hat maximale Größe, falls voll, werden älteste Lösungen gelöscht

TABU-SEARCH

```
l <- maximale Größe der Tabu List
n <- Anzahl der zu betrachtenden Nachbarschaftslösungen
S <- irgendeine initiale zufällige Lösung
best <- S
L <- { } Tabu List der Länge l
Füge S in L ein
REPEAT
    IF Length(L) > l THEN
        Entferne ältestes Element aus L
    wähle R aus Nachbarschaft von S
    FOR n - 1 mal DO
        Wähle W aus Nachbarschaft von S
        IF W ∉ L und (Quality(W) > Quality(R)) oder R ∈ L) THEN
            R <- W
    IF R ∉ L THEN
        S <- R
        Füge R in L ein
    IF Quality(S) > Quality(best) THEN
        best <- S
UNTIL best ist die ideale Lösung oder totale Zeit erreicht
return best
```

### Populationsbasierte Methode

- Bisher: Immer nur Betrachtung einer einzigen Lösung
- Hier: Betrachtung einer Stichprobe von möglichen Lösungen
- Bei der Bewertung der Qualität spielt die Stichprobe die Hauptrolle
- z.B. Evolutionärer Algorithmus

### Evolutionärer Algorithmus

**Idee — Evolutionärer Algorithmus**

- Algorithmus aus der Klasse der Evolutionary Computation
- *generational Algorithmus:* Aktualisierung der gesamten Stichprobe pro Iteration
- *steady-state Algorithmus:* Aktualisierung einzelner Kandidaten der Probe pro Iteration
- *Resampling-Technik:* Generierung neuer Strichproben basierend auf vorherigen Resultaten

*Abstrakter Code (Allgemeiner Breed und Join):*

ABSTRACT-EVOLUTIONARY-ALGORITHM

```
1  P <- generiere initiale Population
2  best <- ⊡ // leere Menge
3  REPEAT
4      AssesFitness(P)
5      FOR jedes individuelle P_i ∈ P DO
6          IF best = ⊡ oder Fitness(P_i) > Fitness(best) THEN
7              best <- P_i
8      P <- Join(P, Breed(P))
9  UNTIL best ist die ideale Lösung oder totale Zeit erreicht
10 return best
```

**Breed** Erstellung neuer Stichprobe mithilfe Fitnessinformation

**Join** Fügt neue Population der Menge hinzu

*Initialisierung der Population*

- Initialisierung durch zufälliges Wählen der Elemente
- Beeinflussung der Zufälligkeit bei Vorteilen möglich
- Diversität der Population (alle Elemente in Population einzigartig)
- Falls neue zufällige Wahl eines Individuums
  - Entweder Vergleich mit allen bisherigen Individuen ($O(n^2)$)
  - Oder besser: Nutzen eines Hashtables zur Überprüfung auf Einzigartigkeit ($O(n)$)

**Idee — Evolutionsstrategie**

- Generiere Population zufällig
- Beurteile Qualität jedes Individuums
- Lösche alle bis auf die $\mu$ besten Individuen
- Generie $\frac{\lambda}{\mu}$-viele Nachfahren pro bestes Individuum
- Join Funktion: Die Nachfahren ersetzen die Individuen

*Algorithmus der Evolutionsstrategie*

$(\mu, \lambda)$-EVOLUTION-STRATEGY

```
μ <- Anzahl der Eltern (initiale Lösung)
λ <- Anzahl der Kinder
P <- {}
FOR λ-oft DO
    P <- {neues zufälliges Individuum}
best <- ⊡
REPEAT
    FOR jedes individuelle P_i ∈ P DO
        AssesFitness(P_i)
        IF best = ⊡ oder Fitness(P_i) > Fitness(best) THEN
            best <- P_i
    Q <- die μ Individueen deren Fitness() am Größten ist
    P <- {}
    FOR jedes Element Q_j ∈ Q DO
        FOR λ/μ-oft DO
            P <- P ∪ {MUTATE(Q_j)}
UNTIL best ist die ideale Lösung oder totale Zeit erreicht
return best
```

## 1.5 Amortisierte Analyse

### Kosten von Operationen

- Bisher: Betrachtung von Algorithmen, die Folge von Operationen auf Datenstrukturen ausführen
- Abschätzung der Kosten von $n$ Operationen im Worst-Case
- Dies liefert die obere Schranke für die Gesamtkosten der Operationenfolge
- Nun: **Amortisierte Analyse**: Genauere Abschätzung des Worst Case
- Voraussetzung: Nicht alle Operationen in der Operationenfolge gleich teuer
- z.B. eventuell abhängig vom aktuellen Zustand der Datenstruktur
- Amortisierte Analyse garantiert die mittlere Performanz jeder Operation im Worst-Case

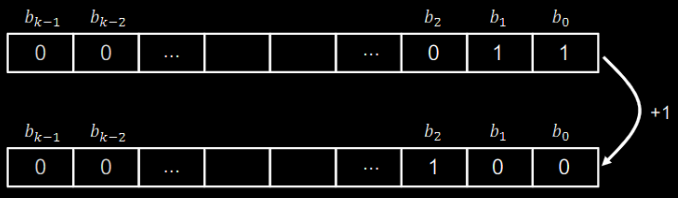
### Beispiel Binärzähler

**Eigenschaften**

- $k$-Bit Binärzähler hier als Array
- Codierung der Zahl als $x = \sum_{i=0}^{k-1} 2^i b_i$
- Initialer Array für $x = 0$:

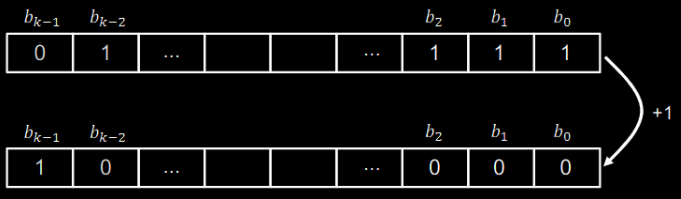

**Inkrementieren eines Binärzählers**

- Erhöhe $x$ um 1
- Beispiel: $x = 3$
- `INCREMENT` kostet 3, da sich drei Bitpositionen ändern




**Teuerste INCREMENT-Operation**

- `INCREMENT` flippt $k - 1$ Bits von 1 zu 0 und 1 Bit von 0 auf 1
- Kosten nicht konstant, stark abhängig von Datenstruktur




**Traditionelle Worst-Case Analyse**

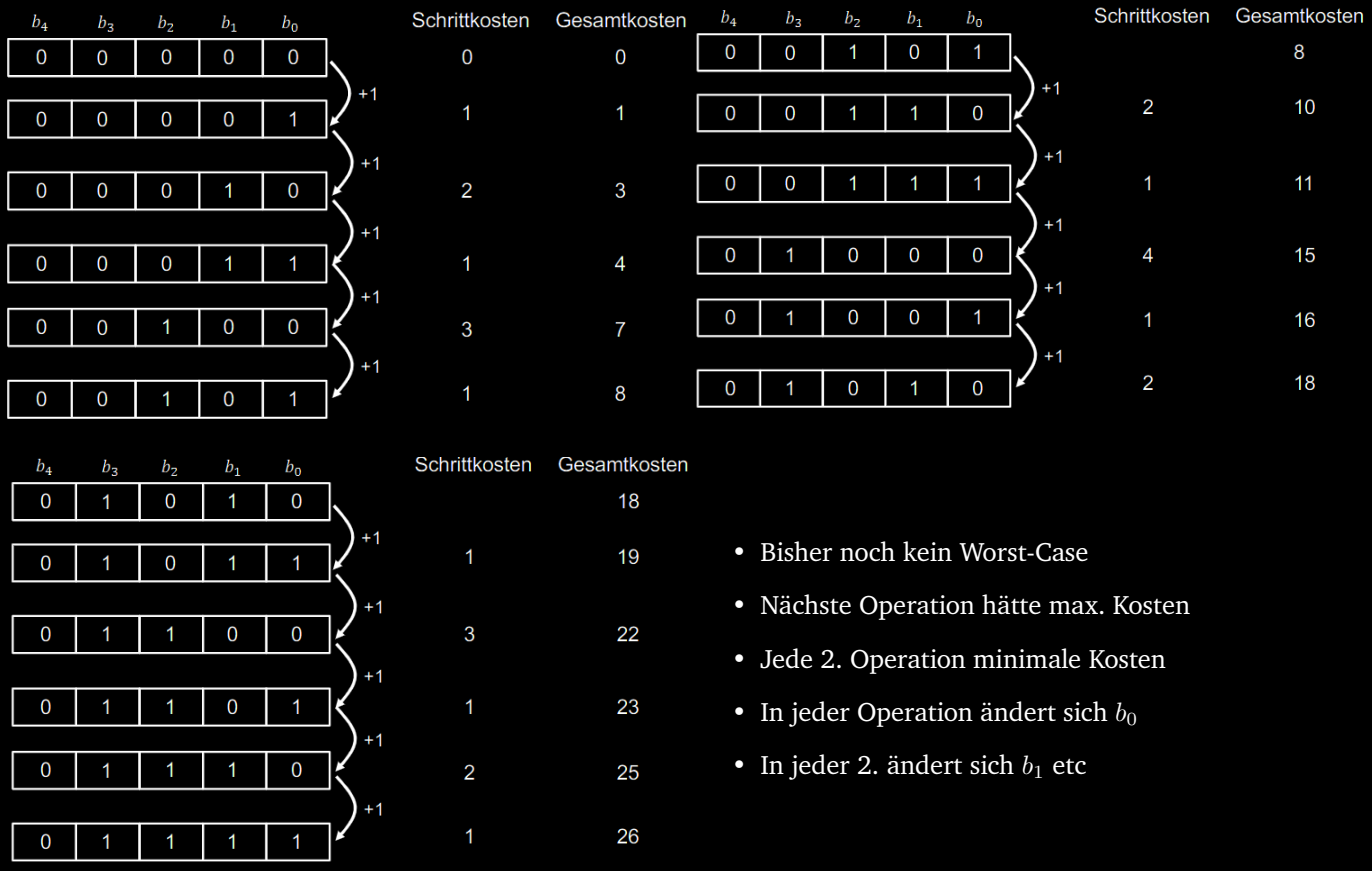
- Worst-Case Kosten von $n$ `INCREMENT`-Operationen auf $k$-Bit Binärzähler
- Anfangswert $x = 0$
- Schlimmster Kostenfall: $INCREMENT$-Operation hat $k$ Bitflips
- $n$-mal inkrementieren sorgt für Kosten: $T(n) \leq n \cdot k \in O(kn)$

### Aggregat Methode - Beispiel Binärzähler

*Eigenschaften:*

- Methode für Amortisierte Analyse
- Sequenz von $n$-Operationen kostet Zeit $T(n)$
- Durchschnittliche Kosten pro Operation $\frac{T(n)}{n}$
- Ziel: $T(n)$ genau berechnen, **ohne** jedes Mal Worst-Case anzunehmen
- Ansatz: Aufsummation der **tatsächlich** anfallenden Kosten aller Operationen

*Durchführung:*




- Bisher noch kein Worst-Case
- Nächste Operation hätte max. Kosten
- Jede 2. Operation minimale Kosten
- In jeder Operation ändert sich $b_0$
- In jeder 2. ändert sich $b_1$ etc

*Genauere Kostenanalyse:*

- Nun in der Lage $T(n)$ genau auszurechnen
- Bei $n$ Operationen ändert sich das Bit $b_i$ genau $\lfloor \frac{n}{2^i} \rfloor$-mal
- Bits $b_i$ mit $i > log_2\ n$ ändern sich nie
- Über alle $k$ Bits aufsummieren liefert:
  $T(n) = \sum_{i=0}^{k-1} \lfloor \frac{n}{2^i} \rfloor = n \sum_{i=0}^{k-1} \frac{1}{2^i} < n \sum_{i=0}^{\infty} \frac{1}{2^i} \leq 2n \in O(n)$
- Obere Schranke: $T(n) \leq 2n$
- Kosten jeder `INCREMENT`-Operation im Durchschnitt: $\frac{2n}{n} = 2 \in O(1)$

**Account Methode - Beispiel Binärzähler**

*Eigenschaften:*

- Besteuerung einiger Operationen, so dass sie Kosten anderer Operationen mittragen
- Zuweisung von höherern Kosten (Amortisierte Kosten), als ihre tatsächlichen Kosten sind
- **Guthaben**: Differenz zwischen amortisierten und tatsächlichen Kosten
- Nutzung dieses Guthabens für Operationen bei denen amortisiert < tatsächlich gilt
- Guthaben darf nicht negativ werden:

  Summe amortisierte Kosten > Summe tatsächliche Kosten

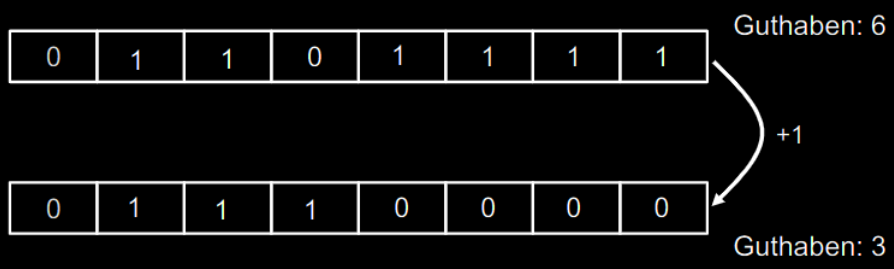
*Wahl der Amortisierten Kosten - Binärzähler:*

- Setzen eines Bits von 0 $\rightarrow$ 1 zahlt 2 Einheiten ein / Bezeichnung $f_i$
- Setzen eines Bits von 1 $\rightarrow$ 0 zahlt 0 Einheiten ein / Bezeichnung $e_i$
- Tatsächliche Kosten $t_i$: Anzahl der Bitflips bei der $i-ten$ `INCREMENT`-Operation

  $t_i = e_i + f_i$
- Amortisierte Kosten betragen: $a_i = 0 \cdot e_i + 2 \cdot f_i$

*Kostenbeispiel:*

- Jede Bitflip Operation kostet zusätzlich 1 Einheit
- Setzen Bit 0 $\rightarrow$ 1: Zahlt 2 ein, kostet aber 1 $\rightarrow$ +1 Guthaben
- Setzen Bit 1 $\rightarrow$ 0: Zahlt 0 ein, kostet aber 1 $\rightarrow$ -1 Guthaben




*Obere Schranken der Kosten:*

- Guthaben auf dem Konto entspricht der Anzahl der auf 1 gesetzten Bits
- Kosten: $T(n) \sum_{i=1}^{n} t_i \leq v \sum_{i=1}^{n} a_i$, für ein konstantes $v$
- Nun Abschätzung dieser Formel zum Erhalten einer oberen Schranke
- Beobachtung: Bei jeder `INCREMENT` höchstens ein neues Bit von 0 auf 1
- Für alle $i$ gilt damit $f_i \leq 1$
- Amortisierte Kosten jeder Operation höchstens $2 \cdot f_i \leq 2$
- Insgesamt: $T(n) = \sum_{i=1}^{n} t_i \leq \sum_{i=1}^{n} a_i \leq 2n \in O(n)$

### Potential-Methode - Beispiel Binärzähler

*Eigenschaften:*

- Betrachtung welchen Einfluss die Operationen auf die Datenstruktur haben
- Potentialfunktion $\phi(i)$: Hängt vom aktuellen Zustand der Datenstruktur nach $i$-ter Operation ab
- Ausgangspotential sollte vor jeglicher Operation nicht negativ sein: $\phi(0) \geq 0$

*Amortisierte Kosten:*

- Amortisierte Kosten der $i$-ten Operation: (Summe tatsächliche Kosten + Potentialänderung)

  $a_i = t_i + \phi(i) - \phi(i-1)$
- Summe der amortisierten Kosten:

  $\sum_{i=1}^{n} a_i = \sum_{i=1}^{n}(t_i + \phi(i) - \phi(i-1)) = \sum_{i=1}^{n} t_i + \phi(n) - \phi(0)$
- Wenn für jedes $i$ gilt $\phi(i) \geq \phi(0)$:

  Summe der amor. Kosten ist gültige obere Schranke an Summe der tatsächlichen Kosten

*Potential-Methode anhand des Binärzählers:*

- $\phi(i)$: Anzahl der 1-en im Array nach $i$-ter `INCREMENT`-Operation

  $\rightarrow \phi(i)$ nie negativ und $\phi(0) = 0$
- Angenommen $i$-te Operation setzt $e_i$ Bits von 1 auf 0, dann hat diese Operation Kosten $t_i \leq e_i + 1$
- Neues Potential: $\phi(i) \leq \phi(i-1) - e_i + 1 \Leftrightarrow \phi(i) - \phi(i-1) \leq e_i$
- Amortisierte Kosten der $i$-ten `INCREMENT`-Operation:

  $a_i = t_i + \phi(i) - \phi(i-1) \leq e_i + 1 + 1 - e_i = 2$
- Insgesamt: $T(n) = \sum_{i=1}^{n} t_i \leq \sum_{i=1}^{n} a_i \leq 2n \in O(n)$