Algorithmen & Datenstrukturen

Prof. Dr. Christian Sohler Oliver Filla

Sommersemester 2023

Contents

Disclaimer **3**

1. Definitionen **3**
Informatik . . . 3
Algorithmus . . . 3
Lernziele . . . 3

2. Entwicklung von Algorithmen **3**
Methode: Teile und Herrsche . . . 3
Beispiele . . . 4
Unterscheidungen . . . 4
Laufzeit . . . 4
Methode: Dynamische Programmierung . . . 4
Beispiele . . . 4
Methode: Gierige Algorithmen . . . 4
Beweise . . . 5
Beispiele . . . 5
Rekursion . . . 5
Laufzeit . . . 5
Optimierung . . . 6
Kostenfunktion . . . 6

3. wichtige Algorithmen **6**
Rekursionsalgorithmen . . . 6
Insertion Sort . . . 6
deskriptiver Pseudocode . . . 6
Merge Sort . . . 6
deskriptiver Pseudocode . . . 7
BinarySearch . . . 7
deskriptiver Pseudocode . . . 7
n-Ziffer-Integer Multiplikation . . . 7
Algorithmus von Strassen (Matrixmultiplikation) . . . 7
Dynamische Programmierung . . . 8
Fibbonacci-Zahlen . . . 8
primitiver rekursiver Algorithmus . . . 8
dynamischer Algorithmus . . . 8

SearchMax . . . 9
rekursiv . . . 9
dynamisch . . . 9
Partition . . . 9
SubsetSum . . . 10
Entwicklung des Algorithmus . . . 10
Indikatorfunktion . . . 10
Rekursive Beschreibung . . . 10
Pseudocode . . . 10
Korrektheitsbeweis . . . 11
Optimierungsprobleme . . . 11
Rucksackproblem . . . 11
Wechselgeldrückgabe . . . 11
Korrektheit . . . 11
Interval-Scheduling . . . 11
Notation . . . 12
Kompatible Intervalle . . . 12
Gieriger Algorithmus . . . 12
Interval-Scheduling mit Deadlines . . . 12
Notation . . . 12
Pseudocode . . . 13
Leerlauf . . . 13
Inversion . . . 13

4. wichtige Datenstrukturen **13**
Graphen . . . 13

5. Speicher und Datentypen **13**
Speichermodell . . . 13
Elementare Datentypen . . . 14
ganze Zahlen . . . 14
reale Zahlen . . . 14
Zeichen . . . 14
Zeiger / Referenz . . . 14
Nicht-Elementare Datentypen . . . 14
Felder . . . 14
Verbunddaten . . . 14
Speicherbedarf . . . 15

6. Pseudocode **15**
Kommentare . . . 15
Verbunddatentypen . . . 15
Felder . . . 15
Zuweisung . . . 15
Typ 1 . . . 15
Typ 2 . . . 16
Bedingte Verzweigungen . . . 16
Schleifen . . . 16
for . . . 16
while . . . 16

repeat 17
Prozeduren 17

Rechentricks / -regeln **24**
Vollständige Induktion 24
Landau-Notation 24

Literatur **24**

Disclaimer

Dies ist eine *inoffizielle* Mitschrift aus der Vorlesung zu Algorithmen & Datenstrukturen von Prof. Dr. Christian Sohler. Ich habe Prof. Sohler's Erlaubnis, dies zu publizieren. Dies bedeutet jedoch nicht, dass irgendjemand Korrekturgelesen hätte. Fehler, Ungenauigkeiten etc. sind demnach zu erwarten und mir zuzuschreiben.

1. Definitionen

Informatik

Informatik ist die Disziplin der automatischen Verarbeitung von Information.[1]

Algorithmus

Ein Algorithmus ist eine wohldefinierte Handlungsvorschrift, die einen Wert oder eine Menge von Werten als Eingabe erhält und als Ausgabe einen Wert oder eine Menge von Werten liefert.[2]

Lernziele

- Methoden zur Entwicklung von Algorithmen
- Bewertung der Qualität von Algorithmen
  - Korrektheit
  - Ressourcen, insbesondere Laufzeit
- Lernen grundlegender Algorithmen und Datenstrukturen

2. Entwicklung von Algorithmen

Methode: Teile und Herrsche

1. Teile die Eingabe in mehrere gleich große Teile auf.
2. Löse das Problem rekursiv auf den einzelnen Teilen.
3. Füge die Teile zu einer Lösung des Gesamtproblems zusammen.

Diese Algorithmen sind oft für teilbare Daten geeignet, beispielsweise für Felder und geometrische Daten.

[1] https://gi.de/fileadmin/GI/Hauptseite/Themen/was-ist-informatik-kurz.pdf
[2] (Cormen et al. 2022)

Beispiele

- MergeSort
- BinäreSuche
- n-Ziffer-Integer Multiplikation
- Matrixmultiplikation (Algorithmus von Strassen)

Unterscheidungen

Teile-und-Herrsche-Algorithmen unterscheiden sich durch... * die Anzahl der Teilprobleme. * die Größe der Teilprobleme. * den Algorithmus für das Zusammensetzen der Teilprobleme. * den Rekursionsabbruch.

Laufzeit

Die Laufzeit kann durch eine Laufzeitanalyse vorhergesagt werden:

- $T(1) \in \mathcal{O}(1)$
- $T(n) = aT(\frac{n}{b}) + f(n)$
  - a: Anzahl der Teilprobleme
  - b: Größe der Teilprobleme, bestimmt die Höhe des Rekursionsbaums
  - $f(n)$: Aufwand für Aufteilen und Zusammenfügen

Methode: Dynamische Programmierung

- Beschreibe optimale Lösung einer gegebenen Instanz durch optimale Lösungen „kleinerer“ Instanzen.
- Beschreibe Rekursionsabbruch.
- Löse die Rekursion “bottom-up” durch schrittweises Ausfüllen einer Tabelle der benötigten Teillösungen.

Dies ist schneller als die rekursive Methode, wenn 1. die “Rekursionstiefe” klein ist. 2. die normale Rekursion viele Mehrfachausführungen hat.

Hinweise: * Wenn wir es mit Mengen zu tun haben, können wir eine Ordnung der Elemente einführen und die Rekursion durch Zurückführen der optimalen Lösung für i Elemente auf die Lösung für $i-1$ Elemente erhalten. * Benötigt wird dabei der Wert der optimalen Lösung für $i-1$ Elemente. * Die Lösung selbst kann nachher aus der Tabelle rekonstruiert werden.

Dynamische Programmierung kann genutzt werden, um Optimierungsprobleme zu lösen.

Beispiele

- Fibbonacci-Zahlen
- SearchMax (keine Laufzeitverkürzung möglich)
- Rucksackproblem

Methode: Gierige Algorithmen

Gierige Algorithmen sind dazu gedacht, Optimierungsprobleme zu lösen. Sie lösen das Problem schrittweise, wobei bei jedem Schritt ein lokales Kriterium

optimiert wird.

Üblicherweise sind diese Algorithmen einfach zu implementieren. Die Korrektheit sicherzustellen ist dagegen schwieriger. Da immer ein lokales Kriterium optimiert wird, ist nicht sichergestellt, dass das globale Kriterium dabei optimal werden kann. Es kann also sein, dass keine oder eine suboptimale Lösung gefunden wird.

Manchmal kann eine optimale Lösung gefunden werden, manchmal kann aber nur eine approximative Lösung gefunden werden. Üblicherweise lassen sich diese Algorithmen in polynomieller Laufzeit implementieren.

Eine Idee zum Entwickeln kann sein, nur bestimmte Ereignisse zu überprüfen. Beispielsweise bei der Verteilung von (Zeit-)Intervallen sind nur die Zeitpunkte betrachten, zu denen mindestens ein Interval beginnt oder endet.

Beweise

Zu einem bestimmten Zeitpunkt im Algorithmus muss gezeigt werden, dass der gierige Algorithmus mindestens so gut wie die optimale Lösung ist.

Beispiele

- Wechselgeldrückgabe
- IntervalScheduling
- LatenessScheduling: Interval-Scheduling mit Deadlines

Rekursion

Eine rekursive Methode ruft sich selbst mit veränderten Parametern auf. Hierzu ist zu Beginn der Methode eine Abbruchbedingung notwendig, die den einfachsten Fall des Problems löst. Ansonsten kommt es zu einer Endlosrekursion.

Zur Entwicklung von neuen Algorithmen ist Rekursion oft hilfreich, wenn man ein Problem auf eine kleinere Stufe desselben Problems runterbrechen kann. Allerdings sind manche rekursive Methoden ineffizient,[3] daher sollte ein solcher Algorithmus oft verbessert / angepasst werden.

Laufzeit

Die Laufzeit kann durch eine Laufzeitanalyse vorhergesagt werden.

- $T(1) \in \mathcal{O}(1)$
- $T(n) = aT(\frac{n}{b}) + f(n)$
  - a: Anzahl der Teilprobleme
  - n/b: Größe der Teilprobleme, bestimmt die Höhe des Rekursionsbaums
  - $f(n)$: Aufwand für Aufteilen und Zusammenfügen
- Die Laufzeit beträgt normalerweise $T(n) \in \mathcal{O}(f(n) \cdot \log_b(n))$
  - $\log_b(n)$ ist die Höhe des Rekursionsbaums
  - meistens ist $b = 2$, also gilt meist $T(n) \in \mathcal{O}(f(n) \cdot \log_2(n))$

[3] Beispielsweise die Berechnung von Fibbonacci-Zahlen ist rekursiv extrem ineffizient, so lange keine Ergebnisse zwischengespeichert werden.

– Auf der letzten Rekursionsstufe gibt es n Teilprobleme der Größe 1. Es gilt $b^h = n$, wobei $h = \log_b n$ die Rekursionshöhe beschreibt.

Optimierung

Kostenfunktion

Für eine Eingabe I sei $S(I)$ die Menge der möglichen Lösungen. Für $L \in S(I)$ sei cost(L) eine *Kostenfunktion*. Gesucht ist nun die Lösung L mit minimalen Kosten cost(L).

Alternativ zu dieser Methode kann man auch eine *Wertefunktion* maximieren.

3. wichtige Algorithmen

Rekursionsalgorithmen

Insertion Sort

```
InsertionSort(A, n) \\ Feld A der Länge n wird übergeben
    for i=2 to n do
    x = A[i]
    j = i -1
    while j>0 and A[j]>x do
        A[j+1] = A[j]
        j = j-1
    A[j+1]=x
```

Die Worst-Case-Laufzeit von InsertionSort ist $\Theta(n^2)$.

deskriptiver Pseudocode

```
InsertionSort(A, n) \\ Feld A der Länge n wird übergeben
    if n=1 return \\ n=1 ist sortiert
    x = A[n] \\ speichere das letzte Element
    InsertionSort(A,n-1) \\ sortiere das Feld bis auf die letzte Stelle
    Füge x an die korrekte Stelle in A ein
```

Merge Sort

MergeSort sortiert erst beide Hälften eines Feldes seperat, bevor es sie zusammenfügt. Dadurch wird das Feld rekursiv sortiert.

- Erster Aufruf: MergeSort$(A, 1, n)$ mit einem Feld A der Länge n.
- Worst-Case-Laufzeit: $T(n) \leq \begin{cases} 1 \Leftrightarrow n = 1 \\ 2T(\frac{n}{2}) + n : \text{sonst} \end{cases} \Rightarrow T(n) = \mathcal{O}(n \log_2 n)$

Satz: Der Algorithmus MergeSort(A, p, r) sortiert das Feld $A[p..r]$ korrekt. Satz: Der Algorithmus MergeSort$(A, 1, n)$ hat eine Laufzeit von $\mathcal{O}(n \log_2 n)$.

deskriptiver Pseudocode

```
MergeSort(A,p,r) \\ Sortiert A[p..r]
    if p<r then \\ Rekursionsabbruch, wenn p=r
        int q = (p+r)/2 \\ Berechne die Mitte (Gaußklammer)
        MergeSort(A,p,q) \\ Sortiere linke Teilhälfte
        MergeSort(A,q+1,r) \\ Sortiere rechte Teilhälfte
        Merge(A,p,q,r) \\ Füge die Teile zusammen
```

BinarySearch

BinarySearch sucht erst in beiden Hälften eines Feldes seperat, die Ergebnisse vergleicht. Dadurch wird das Feld rekursiv durchsucht.

Satz: Die Laufzeit von BinäreSuche(A, x, p, r) ist $\mathcal{O}(\log_2 n)$, wobei $n = r - p + 1$ die Größe des zu durchsuchenden Bereichs ist. Satz: Der Algorithmus BinäreSuche(A, x, p, r) findet den Index einer Zahl x in einem sortierten Feld $A[p..r]$, sofern x in $A[p..r]$ vorhanden ist.

deskriptiver Pseudocode

```
BinarySearch(A,x,p,r) \\ Finde Zahl x in sortiertem Feld A[p..r]
    if p=r then return p \\ sofern vorhanden
    else \\ Ausgabe: Index der gesuchten Zahl
    int q = (p+r)/2 \\ Berechne die Mitte (Gaußklammer)
    if x <= A[q] then return BinarySearch(A,x,p,q)
    else return BinarySearch(A,x,q+1,r)
```

n-Ziffer-Integer Multiplikation

Für große Zahlen nehmen wir an, dass jede *Ziffer* eine Speicherzelle benötigt. Zwei n-Ziffer-Zahlen können wir in der Laufzeit $\Theta(n)$ berechnen. Ein n-Ziffer können wir in Laufzeit $\Theta(n + k)$ mit 10^k multiplizieren.

Dazu multiplizieren wir schriftlich, wobei A, B, C, D n-Ziffern sind: $AB \cdot CD = 100AC + 10(AD + BC) + BD$. Dies sind 4 Multiplikationen von n-Ziffern. Dies hat allerdings eine Laufzeit von $T(n) = 4T(\frac{n}{2}) + cn$, daher gilt $T(n) \in \Theta(n^2)$.

Effizienter wird es, wenn wir die Identität $(A+B)(C+D) = AC+BC+AD+BD$ verwenden. Damit können wir die Summe $BC + AD$ durch $(A + B)(C + D) - AC - BD$ ausdrücken, die Werte AC und BD müssen wir ohnehin berechnen. Dadurch kann man sich eine Multiplikation sparen und man erhält die Laufzeit von $T(n) = 3T(\frac{n}{2}) + cn$, daher gilt $T(n) \in \Theta(n)$.

Algorithmus von Strassen (Matrixmultiplikation)

Falls wir das Produkt von zwei $n \times n$-Matrizen berechnen wollen, können wir diese in je 4 Teilmatrizen der Größe $\frac{n}{2} \times \frac{n}{2}$ aufteilen. Dann multiplizieren wir 8 $\frac{n}{2} \times \frac{n}{2}$-Matrizen und addieren 4 $\frac{n}{2} \times \frac{n}{2}$-Matrizen.

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} \cdot \begin{pmatrix} E & F \\ G & H \end{pmatrix} = \begin{pmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{pmatrix}$$

Die Laufzeit ist dann $T(n) \in \mathcal{O}(n^{\log_2 8}) \subseteq \mathcal{O}(n^3)$:

$$T(n) = \begin{cases} c & n = 1 \\ 8T(\frac{n}{2}) + cn^2 & n > 1 \end{cases}$$

Wir können folgende Rechentricks nutzen:

- $P_1 = A(F - H)$
- $P_2 = (A + B)H$
- $P_3 = (C + D)E$
- $P_4 = D(G - E)$
- $P_5 = (A + D)(E + H)$
- $P_6 = (B - D)(G + H)$
- $P_7 = (A - C)(E + F)$

Damit können wir eine Matrixmultiplikation sparen: * $AE + BG = P_4 + P_5 + P_6 \breve{} P_2$ * $AF + BH = P_1 + P_2$ * $AF + BH = P_1 + P_2$ * $AF + BH = P_1 + P_5 \breve{} P_3 \breve{} P_7$

Auf diese Weise können wir zwei $n \times n$-Matrizen in der $\mathcal{O}(n^{\log_2 7}) \approx \mathcal{O}(n^{1.81})$ berechnen.

Dynamische Programmierung

Fibbonacci-Zahlen

primitiver rekursiver Algorithmus

```
FibRecursive(n)
    if n=1 then return 1
    if n=2 then return 1
    return Fib2(n-1) + Fib2(n-2)
```

FibRecursive hat eine Laufzeit von $T(n) \in \Omega(2^n)$, da für jede Rekursionsebene 2-mal der komplette Rekursionsbaum aufgerufen werden muss. Beispielsweise wird FibRecursive(6) dreimal FibRecursive(3) aufrufen.

$$T(n) = \begin{cases} 2 & n = 1 \\ 3 & n = 2 \\ T(n-1) + T(n-2) + 1 & n > 2 \end{cases}$$

dynamischer Algorithmus Ein besserer Algorithmus speichert Zwischenergebnisse, um doppelte Berechnungen zu vermeiden. Dies gehört zur Dynamischen Programmierung.

Für jedes $m > 0$ gilt, dass FibDynamicCalc(m) maximal zweimal aufgerufen wird. Daher ist die Laufzeit von FibDynamic(n) linear $T(n) \in \mathcal{O}(n)$.

```
FibDynamic(n)
    F = new array [1..n]
    for i=1 to n do
        F[i]=0
    F[1] = 1
```

```
    F[2] = 1

    return FibDynamicCalc(F, n)

FibDynamicCalc(F, n)
    if F[n] > 0 then return F[n]
    else
        F[n] = FibDynamicCalc(F,n-1) + FibDynamicCalc(F,n-2)
    return F[n]
```

Vereinfacht:

```
Fib1(n)
    F = new array[1..n]
    F[1] = 1
    F[2] = 1
    for i=3 to n do
        F[i] = F[i-1] + F[i-2]
    return F[n]
```

SearchMax

Suche das Maximum der Werte in einem Feld A der Länge n. In diesem Fall bringt die Dynamisch Programmierung keine Laufzeitverkürzung.

rekursiv Rekursiver Algorithmus.

```
SearchMaxRecursive(A, n)
    if n=1 then return A[1]
    prev_max = SearchMaxRecursive(A, n-1)
    return max{prev_max, A[n]}
```

dynamisch Algorithmus nach Dynamischer Programmierung.

```
MaxSucheDP(A,n)
    Max = new array [1...n]
    Max[1] = A[1]
    for i=2 to n do
        Max[i] = max{Max[i-1], A[i]}
    return Max[n]
```

Partition

Sei eine Menge natürlicher Zahlen $M \subset \mathbb{N}$ gegeben. Nun soll festgestellt werden, ob M in zwei Mengen L, R aufgeteilt werden kann, sodass die Summe aller Elemente in den Teilmengen gleich ist.

$$\sum_{x \in L} x = \sum_{y \in R} y$$

- Das Partitionsproblem ist NP-vollständig.

- Die Frage, ob man Partition in *polynomieller* Laufzeit lösen kann, ist äquivalent zur Frage ob P gleich NP ist.
- Sei $W = \sum_{x \in M} x$, so kann man die zwei Teilmengen L, R genau dann finden, wenn es eine Teilmenge L mit $\sum_{x \in L} x = \frac{W}{2}$ gibt.

SubsetSum

SubsetSum löst eine verallgemeinerte Fragestellung aus der Partitionsfrage. Gibt es für ein gegebenes U eine Teilmenge $L \subseteq M$, für die $U = \sum_{x \in L} x$ gilt?

- Sei $M = \{x_1, \ldots, x_n\}$ eine Menge, deren Elemente eine Reihenfolge haben.
- Definiere Indikatorfunktion $\mathrm{Ind}(U, m)$.

Entwicklung des Algorithmus

1. Sei $x_n \in L$
   - Es gilt $L = \{x_n\} \cup L \backslash \{x_n\}$.
   - Sei $U' = U - x_n$
   - Gesucht wird eine Menge $L' \subseteq L \backslash \{x_n\} : U' = \sum_{x \in L} x$
2. Sei $x_n \notin L$
   - Gesucht wird eine Menge $L' \subseteq L \backslash \{x_n\} : U = \sum_{x \in L} x$

Indikatorfunktion

$$\mathrm{Ind}(U, m) = \begin{cases} \text{true} & \exists L \subseteq M : U = \sum_{x \in L} x \\ \text{false} & \nexists L \subseteq M : U = \sum_{x \in L} x \end{cases}$$

Rekursive Beschreibung

```
Ind(A, U, m)
    if n=1
    then
        if U>0 \\ Ind(0, 1)
        then return true
        else \\ Ind(U, 1)
        if A[1]=U
        then return true
        else return false
    if U>=x and Ind(A, U-x, n-1) = true then return true
    return Ind(A, U, n-1)
```

Pseudocode

```
SubsetSum(A, U, n)
    \\ initalisiere Indikator
    Ind = new array [0..U][1..n]
    for j=1 to n do
        Ind[j,1] = false
    Ind[0,1] = true \\ leere Menge
    Ind[A[1],1] = true \\ Menge {A[1]}
```

```
\\ suche nach Teilmenge
for i=2 to n do
    for u=0 to U do
        Ind[u,i] = false
        if Ind[u,i-1] = true then Ind[u,i] = true
        if u>=A[i] und Ind[u-A[i], i-1] = true then Ind[u,i] = true
return Ind[U,n]
```

SubsetSum(A, U, n) hat eine Laufzeit von $T(n) = \mathcal{O}(nU)$.

Korrektheitsbeweis Der Korrektheitsbeweis nutzt die Schleifeninvariante $\text{Ind}[u, i] = \text{true}$ genau dann, wenn es eine Teilmenge der ersten i Zahlen aus A gibt, die sich zu u aufsummieren.

Optimierungsprobleme

Rucksackproblem

Es gibt einen Rucksack mit begrenzter Kapazität, in den Objekte mit verschiedenen Größen und verschiedenen Werten gepackt werden sollen. Ziel ist es, den Rucksack mit dem größtmöglichen Wert zu befüllen.

Dazu hat man eine Menge $M = \{1, \ldots, n\}$ an Objekten, die jeweils eine Größe und einen Wert haben. Dies kann man auch durch getrennte Felder für die Werte w_i, die Gewichte g_i und die Rucksackgröße G darstellen.

Dies ist ein Optimierungsproblem.[4]

Wechselgeldrückgabe

Ein eingegebener Centbetrag soll mit möglichst wenig Münzen zurückgegeben werden. Dies wird mit einem gierigen Algorithmus gelöst.

Sei B der Centbetrag. Ein gieriger Algorithmus wählt zunächst die größte verfügbare Münze M mit $M \leq B$ aus und sucht die optimale Rückgabe für den Restbetrag $B - M$.

Korrektheit Angenommen, die Menge der Münzen sei $\{50, 10, 5, 1\}$, dann funktioniert der Algorithmus.

Falls die Menge der Münzen aber $\{50, 10, 7, 5, 1\}$ ist, löst der Algorithmus das Problem nicht: Sei $B = 14$, so liefert der Algorithmus $(1 \times 10, 4 \times 1)$ als Ergebnis. Die optimale Lösung wäre aber (2×7).

Interval-Scheduling

Ziel ist es, eine Ressource möglichst effektiv zu nutzen. Dies bedeutet, dass die Ressource möglichst wenig genutzt wird oder immer möglichst schnell wieder freigegeben wird.

[4] siehe Kapitel *Optimierungsprobleme*

Notation Sei die Eingabe eine Menge von Intervallen. In Pseudocode kann dies durch die Anzahl n, sowie Felder mit den Anfangswerten A und den Endwerten E dargestellt werden.

Gesucht sei die Menge $S \subseteq \{1, \ldots, n\}$, sodass die Anzahl der Elemente maximiert wird, wenn sich die verschiedenen Intervalle nicht überlappen. $\forall i \in S : \exists i \neq j \in S : E[i] \leq A[j] \vee E[j] \leq A[i]$.

Kompatible Intervalle Zwei Intervalle heißen kompatibel, wenn sie sich nicht teilweise überlappen. D.h. mit Feldern der Anfangswerte A und Feldern der Endwerte E gilt $\forall i \in S : \exists i \neq j \in S : E[i] \leq A[j] \vee E[j] \leq A[i]$.

Gieriger Algorithmus

1. Wähle ein Interval i_j geschickt und füge es in die Ergebnismenge S ein.
2. Entferne alle Intervalle, die nicht mit i_j kompatibel sind.
3. Gehe zu 1.

Die Schwierigkeit liegt in Schritt 1. Sowohl die Wahl des erstmöglichen Intervals als auch die Wahl des kürzesten Intervals liefert nicht immer das gewünschte Ergebnis. Da die Ressource immer möglichst früh freigegeben werden soll, kann man immer das Interval nehmen, das am frühesten endet.

Der Algorithmus IntervalSchedule berechnet in Laufzeit $\mathcal{O}(n)$ eine optimale Lösung, wenn die Eingabe nach Endzeit der Intervalle sortiert ist. Die Sortierung kann in $\mathcal{O}(n \log n)$ Zeit berechnet werden.

```
IntervalScheduling(A,E,n) \\ Voraussetzung: Die Intervalle sind nach Endzeitpunkt sortiert
    S = {1}
    j = 1
    for i=2 to n do
        if A[i] \ge E[j] then
            S = S + {i} \\ Vereinigungsmenge
            j = i
    return S
```

Interval-Scheduling mit Deadlines

Wie beim Interval-Scheduling soll eine Ressource bestmöglich genutzt werden. Im Unterschied zu den dortigen Bedingungen sollen hier aber bestimmte Aufgaben gelöst werden, die die Ressource für eine gewisse Dauer in Anspruch nehmen. Zudem hat jede Aufgabe eine Deadline, zu der sie erfüllt sein soll.

Wird eine Aufgabe z Zeiteinheiten nach der Deadline erfüllt, hat sie eine Verzögerung von z. Wird eine Aufgabe innerhalb der Deadline beendet, hat sie eine Verzögerung von 0.

In diesem Fall wird so optimiert, dass die maximale Verzögerung minimiert wird. Hierzu müssen die Aufgaben in der Reihenfolge ihrer Deadlines bearbeitet werden.

Notation Sei die Eingabe eine Menge von Intervallen. In Pseudocode kann dies durch die Anzahl n, sowie Felder mit den Laufzeiten t und den Deadlines d.

Es sollen die Startzeitpunkte der jeweiligen Aufgaben zurückgegeben werden.

Pseudocode Unter der Annahme, dass die Aufgaben in Reihenfolge ihrer Deadlines nicht-absteigend sortiert sind, löst der folgende Algorithmus das Problem in Laufzeit $T(n) \in \mathcal{O}(n)$ optimal und ohne Leerlauf.

```
LatenessScheduling(t,d,n)
    A = new array [1..n]
    z=0
    for i=1 to n do
        A[i] = z
        z = z + t[i]
    return A
```

Leerlauf Alle Lösungen ohne Leerlauf, bei denen die Aufgaben nicht-absteigend nach Deadline sortiert sind, haben dieselbe maximale Verzögerung. Es gibt immer eine optimale Lösung ohne Leerlauf, bei der die Aufgaben nicht-absteigend nach Deadline sortiert sind.

Inversion Eine Reihenfolge von Aufgaben hat eine Inversion (i, j), wenn Aufgabe i vor Aufgabe j in der Reihenfolge auftritt, aber die Deadline $d[i]$ von Aufgabe i größer ist als die Deadline $d[j]$ von Aufgabe j.

Eine Reihenfolge ohne Inversionen ist nicht-absteigend sortiert.

Gibt es in einer Reihenfolge von Aufgaben eine Inversion (i, j), dann gibt es auch eine Inversion zweier in der Reihenfolge benachbarter Aufgaben und man kann Aufgabe i und j vertauschen, ohne die Lösung zu verschlechtern.

4. wichtige Datenstrukturen

Graphen

Bestehen aus *Knoten* und *Kanten*. Kanten können *gerichtet* sein.

Beispielsweise das “Pageranking” von Google war ein *Graphalgorithmus*, der Google die Vorherrschaft auf dem Suchmaschinenmarkt einbrachte: Das Ranking einer Website wurde aus der Anzahl von Verweisen auf ebendiese Website ermittelt.

5. Speicher und Datentypen

Speichermodell

- Beliebig viele Speicherzellen (abstrahiert)
- Durchnummeriert, beginnend mit 1
- *Elementare Datentypen* brauchen jeweils eine Speicherzelle
  - In der Realität ist das nicht exakt der Fall, z.B. bei Kommazahlen

Details von Hardwareimplementierungen werden in diesem Modell vernachlässigt. Diese haben zwar Einfluss, aber üblicherweise in konstanten Größenordnungen.

Elementare Datentypen

Im Vereinfachten RAM-Modell gehen wir davon aus, dass jeder elementare Datentyp eine Speicherzelle belegt.

Bei einer Zuweisung an eine (andere) Variable werden Elementare Datentypen kopiert. Dies nennt man *copy by value*. Im Gegensatz dazu wird bei Nicht-Elementare Datentypen nur der Zeiger darauf kopiert. Dies nennt man *copy by reference*.

ganze Zahlen

reale Zahlen

Zeichen

Zeiger / Referenz

Eine ganze Zahl, die eine Speicherzelle bezeichnet, er kann 0 bzw. NIL sein, das bedeutet dann "kein Wert."

Eine Referenz wird z.B. benutzt, um auf größere Datentypen oder Verbundobjekte zu verweisen. In diesem Fall wird immer auf die erste Speicherzelle verwiesen.

Nicht-Elementare Datentypen

Nicht-Elementare Datentypen sind aus mehreren Elementaren Datentypen zusammengesetzt.

Felder

Felder sind zusammenhängende Speicherbereiche, die denselben elementaren Datentyp enthalten. In einer Variable wird eine Referenz auf die erste Speicherzelle gespeichert.

```
li = new array[n]
li[1] = 4
```

Verbunddaten

Elementare Datentypen können als Verbund organisiert werden. In einer Variable wird eine Referenz auf die erste Speicherzelle gespeichert.

```
Verbund list_item:
    previous
    number
    next

li = new list_item
previous[li] = NIL
number[li] = 5
next[li] = NIL
```

Speicherbedarf

- *Elementare Datentypen*: 1 Zelle
- Felder / Verbunddaten: Summe aller Elemente
- Speicherbedarf Algorithmus
  - Summe *aller* belegten Zellen (inkl. Parameter)
  - kann von Parametern abhängen

6. Pseudocode

- Datentyp wird i.A. nicht explizit angegeben
  - nutzen hier nur elementare Datentypen
- eine Anweisung braucht 1 Rechenschritt
- Variablen im Befehlsblock sichtbar
  - durch Einrückung gekennzeichnet

Kommentare

```
\* Kommentar \*
\\ Kommentar
```

Verbunddatentypen

Laufzeit der Initialisierung: entspricht reserviertem Speicherplatz

```
Verbund list_item:
    previous
    number
    next

li = new list_item
previous[li] = NIL
number[li] = 5
next[li] = NIL
```

Felder

Laufzeit der Initialisierung: entspricht reserviertem Speicherplatz * Initialisierung: `x = new <_Verbundtyp_>` * Zugriff auf das i-te Feldelement: `x[i]` * Index beginnt bei 1

Zuweisung

Typ 1

Es wird eine Kopie von `Y` in `X` gespeichert. Variablen müssen definiert sein.

```
X = Y
```

Typ 2

Ein *konstant großer* mathematischer Ausdruck wird in `X` gespeichert. Variablen müssen definiert sein.

```
X = 10
Y = 2
X = X*Y
```

Nicht konstant groß ist z.B. $\sum_{i=1}^{N} i$. Dies hätte Laufzeit N. Die Summe $\sum_{i=1}^{8} i$ ist dagegen konstant groß. Ggf. wird eine Variable

Bedingte Verzweigungen

lazy evaluation: Bei *UND*-Verknüpfungen wird nach dem ersten *False*-Ergebnis abgebrochen.

```
X = 10
Y = 20
if X > Y then output << Y
else output << X
```

Schleifen

for

Annahmen: * Die Laufvariable i wird am Ende des Schleifenrumpfs erhöht. * Nach dem letzten Durchlauf wird die Laufvariable dennoch erhöht. * Zur Initialisierung wird die Laufvariable i auf den Startwert gesetzt. * Deswegen wird das Schleifenkonstrukt einmal mehr als der Schleifenrumpf aufgerufen. * Die Laufzeitbestimmung zählt hierbei nur die Aufrufe des Schleifenkonstrukts. * Laufzeitanalyse: $1 + (n + 1) + n + 1 = 2n + 3$

Das bedeutet, dass die Laufvariable beim Eintritt in den Schleifenrumpf schon den Wert für den folgenden Schleifendurchlauf hat. Dies ist für die Betrachtung von Schleifeninvarianten relevant.

```
j=0  \* 1 \*
for i=1 to n do  \* Schleifenkonstrukt n+1 \*
    \* Schleifenrumpf \*
    j = j + i  \* n \*
output << j  \* 1 \*
```

```
j=0  \* 1 \*
for i=n downto 1 do  \* Schleifenkonstrukt n+1 \*
    \* Schleifenrumpf \*
    j = j + i  \* n \*
output << j  \* 1 \*
```

while

Der Schleifenrumpf kann 0-mal durchlaufen werden.

```
i=n  \* 1 \*
j=0  \* 1 \*
```

```
while i>0 do  \* n+1 \*
    j=j+i  \* n \*
    i=i-1  \* n \*
output << j  \* 1 \*
```

repeat

Der Schleifenkörper wird mindestens 1-mal durchlaufen

```
i=n  \* 1 \*
j=0  \* 1 \*
repeat  \* 1 \*
    j=j+i  \* n \*
    i=i-1  \* n \*
until = 0  \* n \*
output << j  \* 1 \*
```

Prozeduren

- jede Variable wird als Kopie übergeben (*call by value*)
- der Aufruf einer Prozedur kostet einen Zeitschritt
  - die Zuweisung des Ergebnisses kostet einen weiteren Zeitschritt
  - dazu kommt die Zeit für die Prozedur selbst “‘ beispiel(j) j=j-10 return j

j=100 * 1 * x=7+beispiel(j) * 2+ Zeit für Prozedur * output « j * 1 * output « x * 1 *

`Ausgabe:`

100 97

```
# 7. Laufzeitanalyse
In der Realität spielen Hardware sowie Software (z.B. OS, Compiler(-optionen)) eine Rolle.

Unser Rechenmodell besagt, dass eine Pseudocodeoperartion einen Zeitschritt benötigt. Wir

Hierbei will man für eine gegebene _Eingabegröße_ $n$ eine obere Schranke für die Laufzeit
Üblicherweise benutzt man eine Worst Case Analyse, auch wenn es auch die Average Case Anal

## Worst Case Analyse
Die Worst-Case Laufzeit $T(n) = \max[\text{Laufzeit}]$ ist die längste Laufzeit für alle m
Dies ist der Standard, normalerweise ist diese Analyse gemeint, wenn man von "Laufzeitanal

## Average Case Analyse
Die Worst-Case Laufzeit $T(n) = \mathrm{avg}[\text{Laufzeit}]$ ist die längste Laufzeit fü

## Master-Theorem
Seien $a\ge 1$ und $b\ge 1$ ganzzahlige Konstanten und $f: \mathbb N\rightarrow \mathbb N$
$$
    T(n) \le
        \begin{cases}
            n=1: f(n) \\
```

```
            n>1: a\cdot T(\frac{n}{b}) + f(n)
        \end{cases}
$$
Es gebe ein $\gamma$, sodass gilt:

$$
\begin{aligned}
    1. &&
        \gamma=1:&&
        f(n) = \gamma af\left(\frac{n}{b}\right)
        &&\Rightarrow&&
        T(n) &\in\mathcal O{\large(}f(n)\cdot\log_b(n){\large)} \\
    2. &&
        \gamma>1:&&
        f(n) \ge \gamma af\left(\frac{n}{b}\right)
        &&\Rightarrow&&
        T(n) &\in\mathcal O{\large(}f(n){\large)} \\
    3. &&
        0<\gamma<1:&&
        f(n) \le \gamma af\left(\frac{n}{b}\right)
        &&\Rightarrow&&
        T(n) &\in\mathcal O{\large(}a^{\log_b(n)}{\large)} \\
\end{aligned}
$$

### Merkhilfen
Die folgenen Erklärungen sind nicht zwangsweise mathematisch korrekt, daher sind sie eher

1. Der Aufwand $f(n)$ ist in jeder Rekursionsebene _gleichartig_ (z.B. linear, konstant).
2. Der Aufwand $f(n)$ _wächst_ in jeder Rekursionsebene (abhängig von $n$). Daher dominier
3. Der Aufwand $f(n)$ _sinkt_ in jeder Rekursionsebene (abhängig von $n$). Hier fließt in

### Alternative Formulierung
Es gibt noch andere Formulierungen. Die folgende Formulierung ist gängiger.[^5]
Seien $a\ge 1$ und $b\ge 1$ ganzzahlige Konstanten und $f: \mathbb N\rightarrow \mathbb N$
$$
    T(n) \le
        \begin{cases}
            n=1: f(n) \\
            n>1: a\cdot T(\frac{n}{b}) + f(n)
        \end{cases}
$$

Seien $\varepsilon >0$, $k<1$, $n_0\in\mathbb N$, dann gilt:

$$
\begin{aligned}
    1. &&
        f(n) = O(n^{\log_b(a) - \varepsilon})
        &&\Rightarrow&&
```

```
        T(n) &\in \Theta(n^{\log_b(a)}) \\
    2. &&
        f(n) = \Theta(n^{\log_b(a)}\log_{10}^k(n))
        &&\Rightarrow&&
        T(n) &\in \Theta(n^{\log_b(a)} \log_{10}^{k+1}(n)) \\
    3.
       &&
        f(n) = \Omega(n^{\log_b a + \varepsilon})
        \land \forall n\ge n_0: a f\left(\frac{n}{b}\right) \le kf(n))
        &&\Rightarrow&&
        T(n) &\in \Theta(f(n))
\end{aligned}
$$

[^5]: _Theorem 4.1_ [@AlgorithmsCormen2022, p. 103]

# 8. Landau-Notation
Die detaillierte Laufzeitanalyse hat einige Schwachstellen: Konstante Faktoren werden durc

Die Landau-Notation nutzt eine _asymptotischen Analyse_ für große Eingabemengen $n\rightar

Im Folgenden werden einige Annahmen getroffen:
* Die Funktionen $f$ und $g$ haben den Definitionsbereich $\mathbb N_0$ und sollten für gr
* Die Worst-Case-Laufzeit wird asymptotisch angenährt.

Bei rekursiven Funktionen muss man mit dem Abschätzen der $\Omega$- und $\mathcal O$-Notat

## Beweise
Für Beweise, z.B. mittels Vollständiger Induktion, sollte man die Landau-Notationen nur vo

## Schranken
* Die Schranken $\mathcal O(g(n))$ und $\Omega(g(n))$ geben an, wie stark die analysierte
* Die Schranke $\Theta(g(n))$ gibt dagegen an, dass die Funktion bei großen $n$  in exakt
* Die Schranken $o(g(n))$ und $\omega(g(n))$ geben dagegen an, dass die Funktion immer sch

## $\mathcal O$-Notation
Mit der $\mathcal O$-Notation wird die _obere Schranke_ angenähert.

$f(n)\in\mathcal O (g(n))$ bedeutet, $f$ wächst höchstens so stark wie $g$. Dazu m
$$
    \mathcal O (g(n)) = \{
        \text{Funktion } f(n) |
        \exists c\in\mathbb R_+:\exists n_0\in\mathbb N:
        \forall\mathbb N \ni n\ge n_0: 0 \le f(n) \le c\cdot g(n)
    \}
$$

### Hierarchien
Seien $b, \varepsilon\in\mathbb R$, sodass $b\ge 2$ und $\varepsilon > 0$.
```

```
1. $\mathcal O(\log n) \subseteq\mathcal O(\log^2 n) \subseteq\mathcal O(\log^b n)$
2. $\mathcal O(\log^bn) \subseteq\mathcal O(n^\varepsilon)$
3. $\forall \varepsilon<1:\mathcal O(n^\varepsilon) \subseteq\mathcal O(n) \subseteq\mathc

### Erweiterte $\mathcal O$-Notation
Man kan die $\mathcal O$-Notation auf Funktionen erweitern, die von mehreren Parametern $n

$$
    \mathcal O(g(n,m)) = \{
        \text{Funktion } f(n) |
        \exists c\in\mathbb R_+:\exists n_0, m_0\in\mathbb N:
        \forall\mathbb N \ni n\ge n_0, N \ni m\ge m_0:
        f(n,m) \le c\cdot g(n,m)
    \}
$$

* Diese Definition kann in konstruierten Fällen zu ungewünschten Aussagen führen!
    * z.B. $g(1,m) = m^2$ und $\forall n>1: g(n,m)=m$

## $\Omega$-Notation
Die $\Omega$-Notation liefert eine _untere Schranke_ für die Laufzeit.
$f(n)\in\Omega (g(n))$ bedeutet, $f$ wächst mindestens so stark wie $g$.

$$
    \Omega (g(n)) = \{
        \text{Funktion } f(n) |
        \exists c\in\mathbb R_+:\exists n_0\in\mathbb N:
        \forall n\ge n_0: 0 \le c\cdot g(n) \le f(n)
    \}
$$

## Zusammenhänge

$$
\begin{aligned}
    f(n) &\in \mathcal O(g(n)) &&\Leftrightarrow& g(n) \in \Omega(f(n)) \\
    f(n) &\in o(g(n)) &&\Leftrightarrow& g(n) \in \omega(g(n)) \\
    f(n) &\in \Theta(g(n)) &&\Leftrightarrow& f(n) \in \mathcal O(g(n)) \land f(n) \in \O
\end{aligned}
$$

## $\Theta$-Notation
$f(n)\in\Theta(n)$ bedeutet, dass $f$ für große $n$ ($n\rightarrow\infty$) genauso stark w

$$
    f(n) \in \Theta(g(n)) \Leftrightarrow f(n) \in \mathcal O(g(n)) \land f(n) \in \Omega(
$$

## $o$-Notation
$f(n)\in o (g(n))$ bedeutet, $f$ wächst weniger stark als $g$.
```

```
$$
    o (g(n)) = \{
        \text{Funktion } f(n) |
        \forall c\in\mathbb R_+:\exists n_0\in\mathbb N:
        \forall\mathbb N \ni n\ge n_0: 0 \le f(n) \le c\cdot g(n)
    \}
$$

## $\omega$-Notation
$f(n)\in \omega(g(n))$ bedeutet, $f$ wächst stärker als $g$.

$$
    \omega(g(n)) = \{
        \text{Funktion } f(n) |
        \forall c\in\mathbb R_+:\exists n_0\in\mathbb N:
        \forall\mathbb N \ni n\ge n_0: 0 \le c\cdot g(n) \le f(n)
    \}
$$

# 9. Korrektheitsbeweise
* Elemente eines Korrektheitsbeweise können zur Überprüfung der Funktionsweise während der
* Aus Korrektheitsbeweisen lassen sich häufig gute Kommentare herleiten.
* Ein Korrektheitsbeweis hält letztlich die Überlegungen fest, die ein Entwickler sowieso
* Korrektheitsbeweise helfen dabei, sich dieser Überlegungen bewusst zu werden, und somit

## Definitionen
### Korrektheitsbeweis
Ein Korrektheitsbeweis ist eine formale Argumentation, dass ein Algorithmus korrekt arbeit

### Problembeschreibung
Definiert für eine Menge von zulässigen Eingaben die zugehörigen gewünschten Ausgaben.

### Korrektheit
Wir bezeichnen einen Algorithmus für eine vorgegebene Problembeschreibung als _korrekt_, w

##  Methoden
### Nachvollziehen der Befehle
Ohne Schleifen und Rekursion reicht es, den Ablauf der Befehle zu überprüfen.

### Schleifeninvarianten
Sei $A(n)$ eine Aussage über den Zustand des Algorithmus vor dem $n$-ten Eintritt in den S

Der Beweis für die Korrektheit erfolgt über Vollständige Induktion. Hierbei ist wesentlich

Für _for_-Schleifen werden hierbei folgende Annahmen getroffen:[^6]
* Die Laufvariable $i$ wird am Ende des Schleifenrumpfs erhöht.
* Zur Initialisierung wird die Laufvariable $i$ auf den Startwert gesetzt.
* Die Invariante kann von dem Laufparameter $i$ abhängen.
```

```
Lemma: $A(i)$ ist eine korrekte Schleifeninvariante.

[^6]: siehe Pseudocode/for-Schleife

### Rekursion
* Der Rekursionsabbruch entspricht dem Anfang der Vollständigen Induktion.
* Der Rekursionsaufruf entspricht dem Induktionsschritt.

## $P$ vs. $NP$
Das Problem $P$ _vs._ $NP$ ist eines der wichtigsten Probleme der theoretischen Informatik

Es gibt die Frage, ob die Menge der Probleme, die _schnell_ lösbar sind ($P$), und die Men

* $P$ ist die Menge der Probleme, die in _polynomieller Laufzeit_ zu berechnen sind.
* $NP$ ist die Menge der Probleme, die in _nichtdeterministisch polynomieller Laufzeit_ zu
* Es gilt $P\subset NP$.

# 10. Optimierungsprobleme
Bei einem Optimierungsproblem wird nach einer _optimalen Lösung_ gesucht. Dies kann z.B. d

Ein klassisches Optimierungsproblem ist das Rucksackproblem.

## Rucksackproblem
Es gibt einen Rucksack mit begrenzter Kapazität, in den Objekte mit verschiedenen Größen u

Dazu hat man eine Menge $M=\{1,\dots,n\}$ an Objekten, die jeweils eine Größe und einen We

Dann suchen wir eine Teilmenge $S\subseteq M$, für die $w(S) = \sum_{x\in S} w(x)$ maximie

## Zulässige Lösungen
Eine Lösung $S\in M^\prime$ heißt _zulässig_ für einen Rucksack der Größe $j$, wenn $g(S)\

* Ist $S\subseteq \{1,\dots, i-1\}$ eine zulässige Lösung für einen Rucksack der Größe $j-
* Ist $S\subseteq \{1,\dots, i-1\}$ eine zulässige Lösung für einen Rucksack der Größe $j$
* $S=\{\}$ ist eine zulässige Lösung für _jeden_ Rucksack der Größe $j\ge 0$.

## Optimale Lösungen
Eine zulässige Lösung $S\in M^\prime$ heißt _optimal_ für einen Rucksack der Größe $j$, we

Sei $O\subseteq M^\prime$ eine optimale Lösung für Objekte aus $M^\prime$ und einen Rucksa

1. Ist das Objekt $i\in O$, so ist $O\backslash \{i\}$ eine optimale Lösung mit Objekten a
2. Ist Objekt $i\notin O$ enthalten, so ist $O$ eine optimale Lösung mit Objekten aus $\{1

Weiterhin gilt:

1. $\forall j\ge g[1]: \mathrm{Opt}(1,j)= w[1]$
2. $\forall j<g[1]: \mathrm{Opt}(1,j)=0$
3. $\forall i>1,g[i]>j: \mathrm{Opt}(i,j) = \mathrm{Opt}(i-1,j)$
4. $\forall i>1, g[i]\le j: \mathrm{Opt}(i,j) = \max\{\mathrm{Opt}(i-1,j), w[i] + \mathrm{
```

```
## Methode: Dynamische Programmierung
Dynamische Programmierung kann genutzt werden, um Optimierungsprobleme zu lösen

1. Führe dadurch das Problem auf optimale Teillösungen zurück.
2. Entwerfe eine rekursive Methode zur Bestimmung des _Wertes_ einer optimalen Lösung.
3. Transformiere diese Methode in eine iterative Methode zur Bestimmung des Wertes einer o
4. Bestimmen aus dem Wert einer optimalen Lösung und in der iterativen Methode berechneten

### Finde optimale Teillösungen
Sei $O\in M^\prime$ eine optimale Lösung für einen Rucksack der Größe $j$. Sei $\mathrm{Op

Seien $i=1$, die Eingabemenge $\{1,\dots,i\}=\{1\}$ und die Größe des Rucksacks $j$ gegebe
* Gilt $j\ge g[1]$, dann ist $O=\{1\}$ eine optimale Lösung mit Wert $\mathrm{Opt}(1,j) =
* Ist $j<g[1]$, dann ist $O=\{\}$ eine optimale Lösung mit Wert $\mathrm{Opt}(1,j) = 0$.

### Finde den Wert der optimalen Lösung iterativ
```

Rucksack(n,g,w,G) \ finde die Werte der optimalen Lösungen Opt = new array[1,..,n][0,..,G] for j = 0 to G do if j<g[1] \ Lösungen für 1-elementige Lösungen then Opt[1,j] = 0 else Opt[1,j] = w[1]

```
for i = 2 to n do
    for j = 0 to G do
        if g[i]>j
        then Opt[i,j] = Opt[i-1,j] \\ Objekt i passt nicht in den Rucksack
        else Opt[i,j] = max{Opt[i-1,j], w[i] + Opt[i-1,j-g[i]]} \\ finde optimale Lösung
return Opt[n,G]
```

```
Die Laufzeit ist $T(n)\in \mathcal O(nG)$. Sei $R$ der Wert einer optimalen Lösung für Obj

### finde den Weg der optimalen Lösung
Wir gehen davon aus, dass das Feld $\mathrm{Opt}$ auch nach dem Aufruf von $\mathrm{Rucksa

* Falls das $i$-te Objekt in einer optimalen Lösung für Objekte $1$ bis $i$ und Rucksackgr
* Ansonsten fahre mit Objekt $i-1$ und Rucksackgröße $j$ fort.
```

RucksackLösung(Opt,g,w,i,j) if i=0 return {} if g[i]>j then return RucksackLösung(Opt,g,w,i-1,j)

```
if Opt[i,j]=w[i] + Opt[i-1,j-g[i]]
then return {i} + RucksackLösung(Opt,g,w,i-1,j-g[i]) \\ +: Bilde Vereinigungsmenge
else return RucksackLösung(Opt,g,w,i-1,j)
```

“‘

Nach der Berechnung der Tabelle Opt in der Funktion Rucksack wird RucksackLösung(Opt, $g, w, i = n, j = G$).

Hat die optimale Lösung für Objekte aus M' und Rucksackgröße j den Wert Opt(i, j), so berechnet Algorithmus RucksackLösung eine Teilmenge $S \subseteq M'$, so dass $g(S) \leq j$ und $w(S) = \text{Opt}(i, j)$ ist.

Mit Hilfe der Algorithmen Rucksack und RucksackLösung kann man in der Laufzeit $\mathcal{O}(nG)$ eine optimale Lösung für das Rucksackproblem berechnen, wobei n die Anzahl der Objekte ist und G die Größe des Rucksacks.

Rechentricks / -regeln

- Satz von Gauß: $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$
- Gauß-Klammer: $\lfloor n/2 \rfloor$: Gauss-Klammer: Abgerundet auf ganze Zahl

Vollständige Induktion

Es soll bewiesen werden, dass eine Aussage $A(n)$ für alle $n \in \mathbb{N}$ gilt. 1. Induktionsvoraussetzung: Beweise für $n = 1$ 2. Induktionsschritt: Beweise: Wenn n gilt, dann gilt auch $n + 1$ ("$n \Rightarrow n + 1$") * n gilt ist die Induktionsannahme * auch $n - 1 \Rightarrow n$ ist eine gültige Induktionsannahme * für manche Beweise braucht man auch $n - 1 \Rightarrow n + 1$

Landau-Notation

Für Beweise mittels Vollständiger Induktion darf man die Landau-Notationen nicht verwenden. Bei dieser muss es konkrete Konstanten c geben, die für alle $n \geq n_0$ gelten. Nutzt man während eines Beweises eine Landau-Notation, kann man verschleiern, dass c immer wieder geändert wird.

Literatur

1. (Cormen et al. 2022)

Cormen, Thomas, Charles Leiserson, Ronald Rivest, and Clifford Stein. 2022. *Introduction to Algorithms*. 4th ed. The MIT Press. https://mitpress.mit.edu/9780262046305/introduction-to-algorithms.