Rheinische Friedrich-Wilhelms-Universität Bonn
Mathematisch-Naturwissenschaftliche Fakultät

Algorithmen und Berechnungskomplexität I

Skript WS 2015/2016

Nach Aufzeichnungen von Rolf Klein, Elmar Langetepe und Heiko Röglin

Bonn, Oktober 2015

I

Inhaltsverzeichnis

Kapitel 1

Einführung

In dieser Vorlesung wird die Komplexität von Berechnungsproblemen untersucht und wir werden die dafür notwendigen theoretischen Konzepte entwickeln. Für viele Problemstellungen werden Algorithmen (Lösungspläne) vorgestellt und analysiert.

1.1 Algorithmen

Formal gesehen ist ein *Algorithmus* ein Lösungsplan, der für eine *Probleminstanz* eines definierten Berechnungsproblems die enstprechende *Ausgabe* liefert.

Beispiel: **Sortierproblem**

Eingabe: Eine Folge von n reellen Zahlen $(a_1, a_2, \ldots, a_n)$.

Ausgabe: Eine Permutation π, so dass $(a_{\pi(1)}, a_{\pi(2)}, \ldots a_{\pi(n)})$ eine sortierte Folge der Eingabe ergibt, d.h., $a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}$.

Konkrete Eingabefolgen werden *Instanzen* genannt.

Bezüglich der Algorithmen und der Problemstellung werden wir uns insgesamt zunächst mit drei Begriffen auseinandersetzen:

- Korrektheit: Wir wollen für die Algorithmen formal beweisen, dass zu jeder Instanz das richtige Ergebnis berechnet wird.
- Laufzeitabschätzung: Wie ist die Laufzeit des Algorithmus in Bezug auf die Eingabegröße n und wie effizient ist der Algorithmus im Vergleich zu anderen möglichen Algorithmen?
- Speicherplatzabschätzung: Wieviel Speicher in Bezug auf die Eingabegröße n benötigt der Algorithmus und wie effizient ist das im Vergleich zu anderen möglichen Algorithmen?

Die hier betrachteten Algorithmen sollen mithilfe eines Computers ihre Aufgabe erfüllen, deshalb werden wir uns an gängigen Programmiersprachen orientieren und dazu *Pseudocode* verwenden.

1.1.1 Anwendungsgebiete

Algorithmen spielen in verschiedenen Anwendungsgebieten eine große Rolle. Die Problemstellungen werden durch mathematische Modellierung so formuliert, dass sie durch einen Rechner bearbeitet werden können.

- Human Genome Project: Datenanalyse, DNA-Vergleiche, Matchingalgorithmen, ...
- Internet: Routing, Datenhaltung, Suchmaschinen, Routenplaner, Verschlüsselung, ...
- Industrie: Beladen von Containern, Optimieren von Arbeitsabläufen, ...
- Infrastruktur: Optimieren von Netzen, Umwege minimieren, Kürzeste Wege berechnen/approximieren, ...
- ...

Ein Beispiel aus unserer aktuellen algorithmischen Forschung in Zusammenarbeit mit dem FKIE Wachtberg.

Wir betrachten ein Gebiet, in dem es einen Industrie-Unfall gegeben hat und das von einer Menge von Robotern beobachtet werden soll. Das Gebiet ist durch den Menschen nicht mehr betretbar. Die Roboter haben einen beschränkten Senderadius und sollen den Kontakt untereinander und zur Basisstation halten. Von der Basisstation aus sollen die Roboter zu vorher festgelegten Positionen gefahren werden. Die Positionen und die Wege der einzelnen Positionen zueinander konnten berechnet werden. Damit die Roboter den Kontakt untereinander nicht verlieren, ist es manchmal erforderlich, mehrere Roboter von Position A zu Position B zu bewegen. Für eine Bewegung von A nach B sind somit zum Beispiel mindestens 10 Roboter notwendig.

Die Frage lautet, wieviel Roboter brauchen wir, um von einer Basisstation alle gegebenen Positionen zu besetzen und bei der Bewegung zu gewährleisten, dass die Agenten alle in Kontakt bleiben?

Modellbildung durch einen Graphen $G = (V, E)$ mit ganzzahligen Knotengewichten w_v für $v \in V$ und ganzzahligen Kantengewichten w_e für $e \in E$, ein Startknoten v_s; siehe Abbildung 1.1.

Folgende Bedingungen:

- Eine Kante e darf nur mit $k \geq w_e$ Agenten traversiert werden.
- Ein Knoten v darf nur mit $k \geq w_v$ Agenten besucht werden.
- Beim ersten Besuch eines Knotens v bleiben w_v Agenten am Knoten zurück.

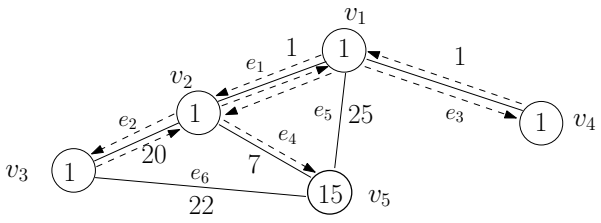

Abbildung 1.1: Falls die Agenten am Knoten v_1 starten, kann eine Tour mit einer minimalen Anzahl an Agenten wie folgt beschrieben werden. 23 Agenten in einer einzelnen Gruppe besuchen Knoten in der Reihenfolge v_1,v_2,v_3,v_4 and v_5. Die Kanten e_5 and e_6 werden nicht besucht.

Frage: Wieviele Agenten werden benötigt, um alle Knoten v mit w_v Agenten zu füllen? Welchen Weg können die Agenten dabei zurücklegen? Lässt sich ein effizienter Algorithmus formulieren, der dieses Problem für beliebige Instanzen löst?

Die Abbildung zeigt ein Beispiel: Vom festgelegten Basisknoten v_1 aus, kommt man mit 23 Agenten aus. Beginnend am Knoten v_1 bleibt zunächst ein Agent zurück und 22 Agenten bewegen sich über die Kante e_1 nach v_2. Hier wird ein Agent abgelegt und mit 21 Agenten geht es zum Knoten v_3 über die Kante e_2. Nachdem wir dort einen Agenten abgelegt haben, können wir mit den verbleibenden 20 Agenten gerade noch über die Kante zurück. Hier stellt sich quasi heraus, dass es mit weniger als 23 gar nicht gehen kann!

Mit den 20 Agenten besuchen wir v_4 (über e_1 und e_3) und lassen einen Agenten dort. Dann geht es mit den verbleibenden über e_3, e_1 nnd e_4 zu v_5. Hier werden 15 Agenten benötigt und wir sind fertig.

Es stellt sich heraus, dass dieses Problem für allgemeine Graphen wirklich *schwer* zu lösen ist. Für einfache Graphen (Bäume) gibt es effiziente Algorithmen, durch die die optimale Lösung in best-möglicher Laufzeit (auch durch die Verwendung effizienter Datenstrukturen (Heaps)) ermittelt werden kann. Daraus resultiert auch eine Approximationslösung für allgemeine Graphen. Aus dem Graphen wird ein Baum mit guter Verbindungseigenschaft generiert und darauf der Baumalgorithmus angewendet.

Ein formaler Beweis zeigt jeweils die Güte der Approximation, die Optimalität des Baumalgorithmus und die Schwere des Problems im Allgemeinen.

Dieses Beispiel enthält somit viele Aspekte, die in der Vorlesung vermittelt werden sollen.

1.1.2 Maschinenmodell

Die Laufzeitabschätzungen der Algorithmen soll unabhängig von der jeweils aktuellen Hardware-Rechnerausstattung (z.B. Taktfrequenz, Verarbeitungsbreite etc.) sein, deshalb zählen wir die Anzahl der ausgeführten *Elementaroperationen* in unserem jeweiligen Maschinenmodell. Der Einfachheit halber veranschlagen wir für jede Operation in etwa die gleichen Kosten.

Wir verwenden das Modell einer sogenannten **REAL RAM**, eine *Random Access Maschine*, die mit reellen Zahlen rechnet. In diesem Modell werden Anweisungen nacheinander (sequentiell) ausgeführt. Das REAL RAM Modell orientiert sich an realen Rechnern und hat folgende Spezifikation:

- Abzählbar unendlich viele Speicherzellen, die mit den natürlichen Zahlen adressiert werden.
- Jede Speicherzelle kann eine beliebige reelle oder natürliche Zahl enthalten. Direkter oder indirekter Zugriff ist möglich.
- Elementare Rechenoperationen: Addieren, Subtrahieren, Multiplizieren, Dividieren, Restbilden, Abrunden, Aufrunden.
- Elementare Relationen: kleiner gleich, größer gleich, gleich, $\vee$, $\wedge$.
- Kontrollierende Befehle: Verzweigung, Aufruf von Unterroutinen, Rückgabe.
- Datenbewegende Befehle: Laden, Speichern, Kopieren.

Alle angegebenen Operationen können in konstanter Zeit ausgeführt werden.

Wir nehmen an, dass Integer-Zahlen der Größe m mit $c \log m$ Bits für eine Konstante $c \geq 1$ dargestellt werden können. Reelle Zahlen werden gemäß des IEEE Standards im Rechner durch Binärzahlen approximiert. Innerhalb des Darstellungsbereiches können arithmetische Operationen mit reellen Zahlen im Rechner sehr effizient durchgeführt werden.

1.1.3 Algorithmische Paradigmen

Algorithmen lassen sich häufig bezüglich ihrer Heransgehensweise an das jeweilige Problem kategorisieren. Zunächst seien hier einige davon kurz erwähnt. Es geht hier im Wesentlichen darum, wie das Problem gelöst wird.

- Brute-Force (Naive Methode)
- Inkrementelle Konstruktion (Schrittweise Vergrößerung der Eingabe)
- Divide-and-Conquer (Aufteilen und Zusammenfügen)
- Greedy (*gierig*, schnelle Verbesserungen)
- Dynamische Programmierung (Tabellarische Auflistung und Verwertung von Teillösungen)
- Sweep (geometrische Probleme, Dimensionsreduktion)
- ...

Die Methoden können auch kombiniert auftreten, zum Beispiel ein Divide-and-Conquer Algorithmus, der im Merge-Schritt einen Sweep verwendet. Auf die einzelnen Paradigmen werden wir später gezielt für konkrete Problemstellungen eingehen.

1.1.4 Analysemethode

Sei P eine Probleminstanz eines Problems Π und sei A ein Algorithmus mit RAM-Anweisungen, der für jede Instanz $P \in \Pi$ eine Lösung liefert. Im Sprachgebrauch wird eine Instanz P auch oft als Problem bezeichnet, wenn keine Verwechslung zu befürchten ist.

Die Anzahl der Elementaroperationen eines Algorithmus A bei einer Eingabegröße $|P| = n \in \mathbb{N}$ wird durch eine *Kostenfunktion* $T_A : \mathbb{N} \mapsto \mathbb{R}^+$ beschrieben. Die Instanz P könnte beispielsweise eine Menge von n Zahlen sein.

Die exakte Bestimmung dieser Funktion kann vernachlässigt werden, insbesondere, da auf unterschiedlichen Rechnern verschiedene Konstanten für die Elementaroperationen beachtet werden müssten. Also sind wir bei der Analyse nur an der Größenordung der Funktion T_A in Abhängigkeit der Eingabegröße $|P| = n$ interessiert. Ebenso verhält es sich mit der Eingabegröße selbst, ob wir es mit n Punkten in der Ebene zu tun haben oder mit $2n$ Koordinaten, soll beispielsweise keine Rolle bei der Beschreibung der Größenordnung spielen.

Die Anzahl der Elementaroperationen eines Algorithmus mit Eingabegröße $n \in \mathbb{N}$ kann auch von der Zusammenstellung der Eingabe selbst abhängen. Deshalb können wir verschiedene Kennzahlen verwenden:

- Worst-case: Maximal mögliche Anzahl an Elementaroperationen

- Average-case: Mittlere Anzahl der Elementaroperationen, gemittelt über alle möglichen Eingaben der entsprechenden Größenordnung

Die Analyse kleiner Eingabegrößen ist generell wenig aussagekräftig, da wir in diesem Fall die Summe aller Elementaroperationen unter einer großen Konstante subsumieren können. Deshalb untersuchen wir das *asymptotische Verhalten* der Kostenfunktion.

Das führt zu folgenden Notationen für Klassen von Funktionen:

Definition 1 (O-,Ω-,Θ-Notation)

$$\begin{aligned} g \in O(f) \quad &:\Leftrightarrow \quad \text{es existiert ein } n_0 \geq 0 \text{ und ein } C > 0, \\ &\qquad \text{so dass } g(n) \leq C\, f(n) \text{ für alle } n \geq n_0\} \end{aligned}$$

$$\begin{aligned} g \in \Omega(f) \quad &:\Leftrightarrow \quad f \in O(g) \\ &:\Leftrightarrow \quad \text{es existiert ein } n_0 \geq 0 \text{ und ein } C > 0, \\ &\qquad \text{so dass } f(n) \leq C\, g(n) \text{ für alle } n \geq n_0. \end{aligned}$$

$$g \in \Theta(f) \quad :\Leftrightarrow \quad g \in O(f) \text{ und } g \in \Omega(f)$$

Wir erlauben also mit $n \geq n_0$ endliche viele Ausnahmestellen.

Die obige Notation wird in dem Sinne gebraucht, dass $f \in O(g)$ eine *obere Schranke* an die Funktion f durch g liefert, währenddessen $f \in \Omega(g)$, eine *untere Schranke* für f durch g darstellt.

Beispiele:

$3n+2 \in O(n)$, da für alle $n \geq 2$ und für $C = 4$ gilt: $3n+2 \leq 4n$. Desgleichen gilt $3n + 2 \in \Omega(n)$, da für alle $n \geq 0$ bereits $n \leq 3n + 2$ gilt. Somit gilt $3n + 2 \in \Theta(n)$.

Für $g(n) = 2n^3 - 18n^2 - \sin(n) + 16n + 3$ führt zunächst die grobe Abschätzung

$$\begin{aligned} g(n) &\leq 2n^3 + 16n + 3 \\ &\leq (2 + 16 + 3)n^3 \text{ für alle } n \geq 1 \end{aligned}$$

zur Aussage $g \in O(n^3)$. Es gilt aber auch

$$\begin{aligned} g(n) &\geq 2n^3 - 18n^2 \\ &= n^3 + (n - 18)n^2 \\ &\geq n^3 \text{ für alle } n \geq 15 \end{aligned}$$

und deshalb gilt hier ebenfalls $g \in \Omega(n^3)$ und $g \in \Theta(n^3)$. Wir sagen auch, dass die Funktion g in der *Größenordnung* n^3 wächst.

Eine weitere hilfreiche Notationskonvention ist, dass wir die obigen Symbole der Einfachheit halber bei einer asymptotischen Betrachtung auch in Formeln verwenden möchten. So drückt $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$ aus, dass wir uns um die Funktion $3n + 1$ in der Größenordnung $\Theta(n)$ nicht genau kümmern möchten. Diese Konvention kann hilfreich sein, wenn beispielsweise eine Laufzeitabschätzung *rekursiv* durch eine Formel $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$ beschrieben werden darf. Wahlweise kann hier auch $T(n) = 2T\left(\frac{n}{2}\right) + C \cdot n$ für ein nicht näher bestimmtes C verwendet werden.

Außerdem kann es sinnvoll sein, die Analyse der Laufzeiten *ausgabesensitiv* vorzunehmen, um die Ausgabekomplexität nicht der algorithmischen Komplexität anzulasten. Falls wir beispielsweise die Menge aller Schnittpunkte von n Liniensegmenten suchen, so ist klar, dass es im worst-case $\binom{n}{2}$ viele Schnittpunkte geben kann, jeder Algorithmus also mindestens diese Laufzeit im worst-case benötigt. Eine ausgabesensitive Analyse könnte zu einem Ergebnis von $O((n + k) \log n)$ führen, wobei k die Anzahl der tatächlichen Schnittpunkte angibt.

1.2 Ein einfaches Beispiel: Insertionsort

Die obigen Begriffe wollen wir zunächst auf das einfache Beispiel des Sortierens anwenden. Wir nehmen an, die n zu sortierenden Zahlen sind in einem *Array* A gespeichert.

Eingabe: n reelle Zahlen $A[1], A[2], \ldots, A[n]$.

Ausgabe: Ein Array B der Zahlen aus A mit $B[1] \leq B[2] \leq \cdots \leq B[n]$.

Beachte, dass wir hier eine etwas andere Ausgabe als zu Beginn erwarten (Permutation π). Die Aufgabenstellung ist aber äquivalent.

Eine *Brute-Force* Vorgehensweise wäre, das kleinste Element des Arrays zu ermitteln, dieses zu entfernen und mit dem restlichen Array fortzufahren.

Wir betrachten einen *inkrementellen* Ansatz. Sukzessive wird ein immer größerer Teil des Arrays A in B sortiert. Anders ausgedrückt, falls wir die ersten $j - 1$ Elemente des Arrays A in B bereits sortiert haben, nehmen wir uns danach das j-te Element vor und sortieren es in B ein.

Die Korrektheit des Algorithmus wird durch eine *Schleifeninvariante* gezeigt. Wir zeigen formal, dass für jeden Beginn der FOR-Schleife die fogende Invariante gilt: *Für den aktuellen Wert j gilt, dass B bis zum Index $j - 1$ eine sortierte Folge der ersten $j - 1$ Einträge von A ist und dass A und B ab dem Index j übereinstimmen*

Formal gesehen handelt es sich hier um eine Induktion, die bei einem be-

Algorithmus 1 InsertionSort(A)
Array A (nichtleer) sortiert in Array B ausgeben

```
1: B[1] := A[1];
2: for j = 2 to length(A) do
3:    key := A[j]; i := j − 1;
4:    while i > 0 und B[i] > key do
5:       B[i + 1] := B[i]; i := i − 1;
6:    end while
7:    B[i + 1] := key;
8: end for
9: RETURN B
```

stimmten Parameter terminiert.

Induktionsanfang: $j = 2$. Die Eigenschaft ist erfüllt, durch die erste Anweisung.

Induktionsschritt: Wir nehmen an, dass die Aussage bereits für $j-1 \geq 2$ gilt. Jetzt werden in der WHILE-Schleife im Array B solange die Werte um einen Index nach hinten geschoben, bis das Element $key := A[j]$ kleiner gleich dem i-ten Element von B ist. Dann wird key als $(i+1)$-tes Element in B eingetragen. Falls das bis $i = 0$ nicht eintritt, wird key als 1-tes Element von B eingetragen. Insgesamt ist B danach bis zum Index j ein sortiertes Array der ersten j-Elemente von A.

Terminierung: Falls $j = length(A)+1$ wird, gilt zunächst die Invariante, dass für den aktuellen Wert j gilt, dass B bis zum Index $j-1 = length(A)$ eine sortierte Folge der ersten $length(A)$ Einträge von A ist. Die FOR-Schleife bricht ab und das Ergebnis wird korrekt in B ausgegeben.

Übungsaufgabe: Streng genommen muss die WHILE-Schleife genauso analysiert werden. Welche Schleifeninvariante muss hier gewählt werden?

Für eine Laufzeitabschätzung analysieren wir die einzelnen Programmschritte. Dabei sei t_j die Anzahl der Durchläufe der WHILE-Schleife wenn die FOR-Schleife für j aufgerufen wird. Sei $length(A) = n$.

Es ergibt sich die folgende Tabelle:

Anweisungen in Algorithmus 1	Kosten	Häufigkeit
1:	c_1	1
3:	c_2	$n-1$
5:	c_3	$\sum_{j=2}^{n} t_j$
7:	c_4	$n-1$

Insgesamt ergibt sich nach unseren Konventionen eine Laufzeit $T(n) = c_3 \sum_{j=2}^{n} t_j + O(n)$.

Die Laufzeit hängt also von der Größe von t_j ab. Im schlechtesten Fall ist das Array A absteigend sortiert, alle Elemente aus B müssen verschoben werden und das j-te Element wird vorne eingefügt. In diesem Fall gilt $t_j = j$ und wir haben.

$$T(n) \leq c_3 \sum_{j=1}^{n} j + O(n) = c_3 \frac{n(n+1)}{2} + O(n) \in O(n^2)$$

Die worst-case Analyse ist im allgemeinen sinnvoll, weil ...

- ... wir eine Laufzeit für jede beliebige Eingabe garantieren.
- ... der worst-case in vielen Anwendungsfeldern relativ häufig vorkommen kann.
- ... die Analyse der mittleren Laufzeit nicht unbedingt bessere Ergebnisse erzielt.

Beispielsweise können wir bei der Analyse einer mittleren Laufzeit die Frage stellen, an welcher Stelle im Mittel das j-te Element eingeordnet werden muss. Im Mittel ist die Hälfte der Elemente größer als $A[j]$ und die Hälfte der Elemente kleiner als $A[j]$ und somit könnten wir $t_j = \frac{j}{2}$ betrachten. Das führt asymptotisch gesehen exakt zur gleichen Laufzeit.

Anders kann es aussehen, wenn wir aus der Eingabe durch *Würfeln* zufällig gleichverteilt die Elemente aus A ziehen.

Solche *probabilistische Analysen* werden wir später für sogenannte *randomisierte* Algorithmen durchführen. Im Gegensatz dazu haben wir hier zunächst einen *deterministischen* Algorithmus betrachtet. Der Lösungsplan ist von vorneherein vollständig festgelegt und nicht vom Zufall abhängig.

Kapitel 2

Divide-and-Conquer

In diesem Kapitel behandeln wird die Entwurfsmethode *Divide-and-Conquer* (übersetzt: Teile und herrsche) und eine zugehörige allgemeine Analysemethode. Gleichzeitig lernen wir dadurch das Prinzip der *Rekursion* kennen, das heißt wir rufen bestimmte Routinen unseres Algorithmus immer wieder auf, um Teilprobleme zu lösen. Am Ende werden die Lösungen der Teilprobleme zur Lösung des Gesamtproblems zusammengefügt bzw. zusammengemischt.

Das allgemeine Schema der Methode für ein Problem P der Größe n funktioniert wie folgt:

1. Divide	$\begin{cases} n > c \\ \\ n \leq c \end{cases}$	:	Teile das Problem in $k \geq 2$ Teilprobleme der Größen $n_1, n_2, \ldots, n_k$, gehe zu 2.
		:	Löse das Problem direkt
2. Conquer	Löse die k Teilprobleme auf dieselbe Art (rekursiv)		
3. Merge	Füge die k berechneten Teillösungen zu einer Gesamtlösung zusammen		

2.1 Sortieren von Zahlenketten, Mergesort

Hier betrachten wir $k = 2$ und $n_1, n_2 \approx \frac{n}{2}$. Für eine konkrete Zahlenfolge ergibt sich das folgende Aufrufschema eines Divide-and-Conquer Algorithmus:

Eingabe	:	$(3,2,1,7,3,4,9,2)$
Divide I.	:	$(3,2,1,7)(3,4,9,2)$
(Conquer)Divide II.	:	$(3,2)(1,7)(3,4),(9,2)$
(Conquer)Divide III.	:	$(3)(2)(1)(7)(3)(4)(9)(2)$
Merge III.	:	(2,3)(1,7)(3,4)(2,9)
Merge II.	:	(1,2,3,7)(2,3,4,9)
Merge I.	:	(1,2,2,3,3,4,7,9)

Im Pseudocode und unter Verwendung von Arrays könnte eine Implementierung wie folgt aussehen. Wir verwenden ein nichtleeres Array A von Index 1 bis $n = \text{length}(A)$ und rufen zur vollständigen Sortierung MergeSort$(A, 1, n)$ auf.

Algorithmus 2 MergeSort(A,p,r)
Array A wird zwischen Index p und r sortiert

```
1: if p < r then
2:     q := ⌊(p + r)/2⌋;
3:     MergeSort(A, p, q);
4:     MergeSort(A, q + 1, r);
5:     Merge(A, p, q, r);
6: end if
```

Die eigentliche Arbeit (bis auf die Anzahl der rekursiven Aufrufe) wird im Merge-Schritt vollzogen. Dafür kopieren wir die bereits von Index p bis q und von Index $q+1$ bis r sortierten Abschnitte in zwei Arrays L und R und fügen diese dann wieder gemeinsam sortiert in A von Index p bis r ein. Für ein Abbruchkriterium des Merge-Schrittes fügen wir einen Dummywert ∞ am Ende von L und R ein.

Danach werden die beiden Teilarrays gemischt, indem in einem gleichzeitigen Durchlauf jeweils das kleinste momentan vorderste Element von L und R in A eingefügt wird.

2.1.1 Laufzeitanalyse

Für einen einzelnen Merge-Schritt entsteht offensichtlich ein Aufwand von $O(|L| + |R|)$; siehe Prozedur 3. In jedem Schritt der finalen FOR-Schleife wird ein Element von L oder R nicht mehr betrachtet. Deshalb kann die Schleife nicht mehr als $(|L| + |R|)$ Mal aufgerufen werden.

Zur Laufzeitanalyse verwenden wir eine Rekursionsgleichung. Eine klassische Vereinfachung ist dabei, dass wir annehmen, dass n eine Zeierpotenz ist, also

Prozedur 3 Merge(A,p,q,r)
Array A vorsortiert zwischen Index p und q und Index $q+1$ und r wird zwischen p und r sortiert

```
1: n_1 := q − p + 1; n_2 := r − q;
2: for i = 1 to n_1 do
3:    L[i] := A[p + i − 1];
4: end for
5: for j = 1 to n_2 do
6:    R[j] := A[q + j];
7: end for   // erzeuge Arrays L[1 ... (n_1 + 1)] und R[1 ... (n_2 + 1)]
8: L[n_1 + 1] := ∞; R[n_2 + 1] := ∞;
9: i := 1; j := 1;
10: for k = p to r do
11:    if L[i] ≤ R[j] then
12:       A[k] := L[i]; i := i + 1;
13:    else
14:       A[k] := R[j]; j := j + 1;
15:    end if
16: end for
```

$n = 2^k$. Wir können die Kostenfunktion dann für den obigen Algorithmus wie folgt beschreiben.

$$T(n) = \begin{cases} c & \text{falls } n = 1 \\ 2T\left(\frac{n}{2}\right) + cn & \text{falls } n > 1 \end{cases} \tag{2.1}$$

In der obigen Gleichung haben wir eine einzelne Konstante c verwendet. Das ist prinzipiell erlaubt, da wir für die Konstanten für das Problem der Größe 1 und den Merge-Schritt einfach die größte der beiden Konstanten wählen können. Die asymptotische Laufzeit ändert sich nicht.

Man beachte, dass in den Kosten cn auch das Aufteilen der Aufgabenstellung in zwei Teilprobleme subsumiert ist. Hier musste lediglich ein Index in der Mitte in konstanter Zeit berechnet werden.

Die Rekursionsgleichung läßt sich nun leicht inbesondere auch wegen der Annahme $n = 2^k$ durch einen Rekursionsbaum abschätzen, siehe Abbildung 2.1. Die allerunterste Ebene des Baumes enthält exakt n Aufrufe für die Arrays der Größe 1. In jeder darüberliegenden Ebene werden jeweils zwei Mengen der darunterliegenden Ebene im Merge-Schritt vereinigt. Die Kosten sind jeweils *linear* zur Summe der Größen der darunterliegenden Arrays. Somit sind die Kosten auf jeder Ebene linear zur Gesamtzahl der Elemente überhaupt. Der Baum hat $k = \log_2 n$ viele Ebenen, da sich die Anzahl der Knoten von unten nach oben jeweils halbiert. Wir können also $T(n) = O(n \log n)$ und sogar $T(n) = \Theta(n \log n)$ folgern. Streng genommen hätten

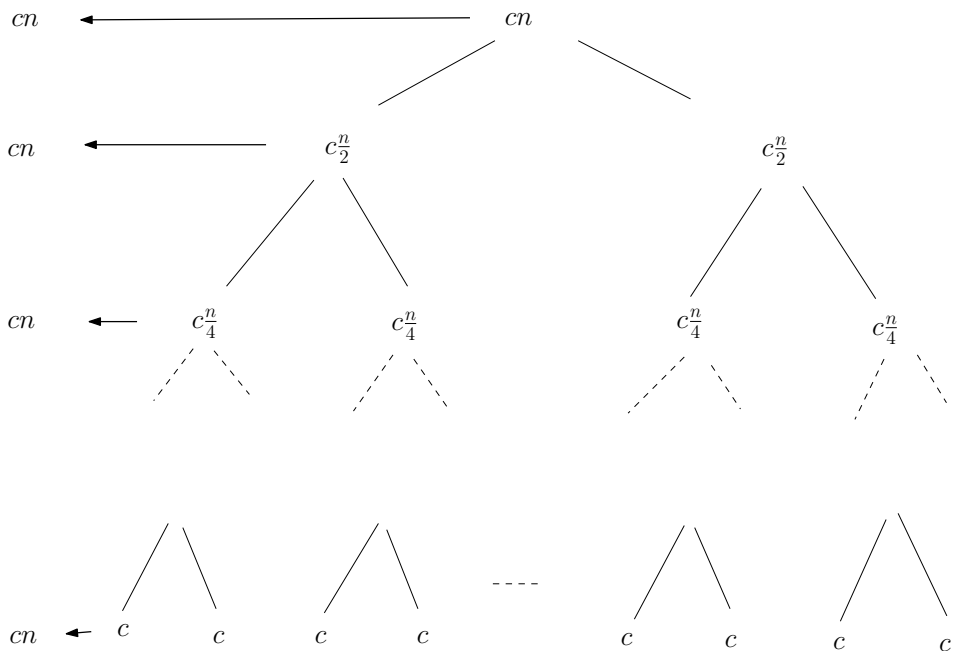

Abbildung 2.1: Nach $k = \log_2 n$ Ebenen sind nur noch einzelne Elemente vorhanden. Auf jeder Ebene des Rekursionsbaumes fällt ein Aufwand von cn für die einzelnen Merge-Schritte an.

wir die Rekursionsgleichnung (2.1) wie folgt formulieren müssen.

$$T(n) = \begin{cases} c & \text{falls } n = 1 \\ T\left(\lfloor \frac{n}{2} \rfloor\right) + T\left(\lceil \frac{n}{2} \rceil\right) + cn & \text{falls } n > 1 \end{cases} \tag{2.2}$$

In jedem Fall müssen wir uns beim Aufstellen und Lösen der Rekursionsformeln Gedanken darüber machen, ob die Einschränkung $n = 2^k$ nicht das asysmptotischen Verhalten der Laufzeit beinflusst. Im obigen Fall ist das offensichtlich nicht der Fall, der zugehörige Rekursionsbaum wäre zwar nicht so ausgeglichen, wie in Abbildung 2.1. Für jedes n könnten wir aber beispielsweise die nächste Zweierpotenz oberhalb von n nehmen, um eine obere Schranke zu erzielen, sowie die nächste Zweierpotenz unterhalb von n für eine untere Schranke der Laufzeit. Der Algorithmus hat dann entsprechend mehr oder entsprechend weniger Aufrufe zu absolvieren. Darüber müssen wir uns aber Gedanken machen.

Beispielsweise kann eine Rekursionsformel der Form

$$T(n) = \begin{cases} n & \text{falls } n \text{ gerade} \\ n^2 & \text{falls } n \text{ ungerade} \end{cases} \tag{2.3}$$

nicht mit dem Ansatz $n = 2^k$ betrachtet werden.

2.1.2 Korrektheit

Die Korrektheit des obigen Divide-and-Conquer Algorithmus ergibt sich direkt aus der Korrektheit der Merge-Prozedur. Sobald diese gemäß ihrer Spezifikation funktioniert, erhalten wir die richtigen Zwischenergebnisse rekursiv für weitere Aufrufe.

Zunächst ist klar, dass die Merge-Prozedur durch die ersten beiden FOR-Schleifen die Einträge des Arrays A von Index p bis Index q und von Index $q + 1$ bis r in die Kopien L und R der Größen $n_1 := q - p + 1$ $n_2 := r - q$ einträgt. Streng genommen müssen auch diese Behauptungen durch eine Schleifeninvariante bewiesen werden.

Wir beschränken uns hier auf die dritte FOR-Schleife und formulieren eine geeignete Schleifeninvariante.

Zu Beginn jeder Iteration der FOR-Schleife enthält das Array $A[p \dots k-1]$ die $(k-p)$ kleinsten Einträge aus L und R in sortierter Reihenfolge. $L[i]$ und $R[j]$ sind die kleinsten Einträge der jeweiligen Arrays, die noch nicht in A zurückkopiert wurden.

Induktionsanfang: $k = p$. Das Array $A[p \dots k-1]$ enthält die $k - p = 0$ kleinsten Elemente von L und R. $L[1]$ und $R[1]$ sind die kleinsten Elemente der Arrays, die noch nicht zurückkopiert wurden.

Induktionsschritt: Wir nehmen an, die Aussage gelte bereits für $k = p+l$ und wollen zeigen, dass die Aussage nach dem Durchlauf der FOR-Schleife für $k' = p+l+1$ erhalten bleibt. Die Schleife wird mit $k = p+l$ ausgeführt. Es werden die aktuellen Köpfe der beiden Arrays L und R verglichen und das kleinste der beiden Elemente als $k = (p + l)$-tes Element in A eingetragen. Also besteht danach $A[p \dots k'-1]$ für $k' = p+l+1$ aus den $(k'-p)$ kleinsten Einträge aus L und R in sortierter Reihenfolge. Je nachdem, ob $L[i] \leq R[j]$ oder $L[i] > R[j]$ gilt, wird i oder j um eins erhöht und die Aussage gilt auch für $k' = p + l + 1$ und das neue j oder i.

Terminierung: Beim Abbruch gilt $k = r + 1$ und $A[p \dots r]$ enthält die $r+1-p$ kleinsten Elemente von L und R in sortierter Reihenfolge. Zusammen enthalten L und R insgesamt $n_1 + n_2 + 2 = r - p + 3$ Elemente. Also muss sowohl i am Index $n_1 + 1$ und j am Index $n_2 + 1$ angekommen sein. Alle Werte außer den Dummywerten ∞ sind in A eingetragen worden.

2.1.3 Untere Schranke von Sortieralgorithmen

Mergesort kann also n Zahlen $(a_1, a_2, \dots, a_n)$ in Zeit $O(n \log n)$ sortieren. Dabei greift der Algorithmus auf die Elemente der Eingabefolge nur durch *paarweise Vergleiche* zu (siehe Zeile 11 der Merge-Prozedur): Es wird getestet, ob zum Beispiel $a_i < a_j$ gilt, und der Ausgang dieses Tests entscheidet, was der Algorithmus als nächstes macht. "Gerechnet" wird mit den a_i aber nicht.

Mit solchen vergleichsbasierten Sortierverfahren kann man deswegen nicht nur Zahlen sortieren, sondern Objekte aus beliebigen vollständig geordneten Mengen, vorausgesetzt, es steht ein Größenvergleichstest zur Verfügung.

Interessanterweise ist Mergesort sogar optimal.

Theorem 2 *Jeder vergleichsbasierte Sortieralgorithmus benötigt im worst-case $\Omega(n \log n)$ viele Vergleiche, um n Objekte zu sortieren.*

Beweis. Sei A ein vergleichsbasierter Sortieralgorithmus. Wir nehmen der Einfachheit halber an, dass A deterministisch ist; das Theorem gilt aber auch für randomisierte Verfahren. Außerdem setzen wir voraus, dass die a_i paarweise verschieden sind. Weil A den Input $(a_1, a_2, \dots, a_n)$ zunächst nicht kennt, wird stets derselbe Test $a_i < a_j$? als erster ausgeführt. In Abhängigkeit vom Ergebnis verzweigt der Algorithmus. Für alle Eingaben, bei denen $a_i < a_j$ gilt, wird jetzt stets derselbe Test $a_v < a_w$? als nächster ausgeführt. Ebenso kommt für alle Inputs mit $a_i > a_j$ als nächstes stets derselbe Test $a_r < a_s$? an die Reihe, und so fort. So entsteht der Entscheidungsbaum B_A von Algorithmus A, ein Binärbaum, der mindestens so viele

Blätter enthält, wie es Permutationen gibt; denn ein korrekter Sortieralgorithmus darf für unterschiedlich permutierte Eingabefolgen nicht zum selben Ergebnis kommen. Also gilt

$$\begin{aligned} \text{Höhe}(B_A) &\geq \log_2(n!) = \log_2\left(\left(\frac{n}{2}\right)^{\frac{n}{2}}\right) \geq \frac{n}{2}\log_2\left(\frac{n}{2}\right) \\ &\geq \frac{1}{3}n\log_2(n) \end{aligned}$$

für alle $n \geq 8$. Dabei gilt die erste Ungleichung, weil es in $n!$ mindestens $n/2$ viele Faktoren gibt, die mindestens so groß sind wie $n/2$. Es gibt also im Baum B_A mindestens einen Pfad der Länge $\Omega(n \log n)$, und wenn die Eingabe so permutiert ist, wie es den Testergebnissen längs dieses Pfades entspricht, muss Algorithmus A alle diese Vergleiche ausführen. □

Wenn die zu sortierenden Objekte aber Zahlen sind, mit denen man "rechnen" darf, gilt die untere Schranke von Theorem 2 nicht. Zum Beispiel geht der Algorithmus Radixsort folgendermaßen vor, um n k-stellige Zahlen zu sortieren: Zunächst werden leere 10 Listen $L_0, L_1, \ldots, L_9$ angelegt, je eine für jede Ziffer. Nun werden abwechselnd k Verteilungs- und Sammelphasen ausgeführt. Man beginnt mit der letzten der k Stellen, durchläuft die Eingabereihenfolge und hängt die Zahl a_i an diejenige Liste an, die zur letzten Ziffer von a_i gehört. Anschließend werden die Zahlen aus den Listen wieder eingesammelt, in der Reihenfolge $L_0, L_1, \ldots, L_9$ und unter Beibehalt der Reihenfolgen innerhalb der Listen. Mit der neuen Zahlenfolge verfährt man ebenso, verwendet aber nun die vorletzte Stelle, und so fort. Wenn man schließlich die Zahlen nach der ersten Stelle verteilt und wieder eingesammelt hat, sind sie sortiert (!).

Offensichtlich führt Radixsort nur $O(k\,n)$ viele Schritte aus. Wenn die Stellenzahl k als konstant betrachtet wird (was bei Standard-Zahlentypen meist der Fall ist), so ergibt sich die Laufzeit $O(n)$.

2.2 Closest Pair von n Punkten

Wir betrachten die folgende geometrische Aufgabenstellung. Für einer Menge S von n Punkten $p_1, p_2, \ldots, p_n$ in der Ebene, soll das Punktepaar mit kleinstem Euklidischen Abstand $d(\cdot,\cdot)$ berechnet werden.

Wir suchen also den Wert

$$d_S = \min_{1 \leq i,j \leq n, i \neq j} d(p_i, p_j).$$

Der Brute-Force Ansatz für dieses Problem vergleicht systematisch alle $O(n^2)$ vielen Distanzen und benötigt eine Laufzeit von $\Theta(n^2)$.

Für einen Divide-and-Conquer Algorithmus sortieren wir die Punkte zunächst nach X-Koordinaten und teilen die Punktmenge rekursiv in zwei gleichgroße Mengen anhand der X-Koordinaten auf.

1. Divide	Teile das Problem in Teilmengen S_r und S_l gleicher Größe anhand der X-Koordinaten auf.
2. Conquer	Bestimme rekursiv d_{S_l} und d_{S_r}.
3. Merge	Berechne d_S durch die Verwendung von d_{S_l} und d_{S_r}.

Wir wollen im Merge-Schritt wiederum möglichst wenig Aufwand für die Bestimmung von d_S verwenden. Sei $d := \min\{d_{S_l}, d_{S_r}\}$. Wir brauchen für den Merge nur Punkte betrachten die von S_l zu Punkten aus S_r einen Abstand $< d$ haben. Nur die Punkte innerhalb des Streifens der Breite $2d$ symmetrisch entlang der *Splitvertikalen* können einen solchen Abstand $< d$ zueinander haben; siehe Abbildung 2.2. Im Merge-Schritt wird folgendes

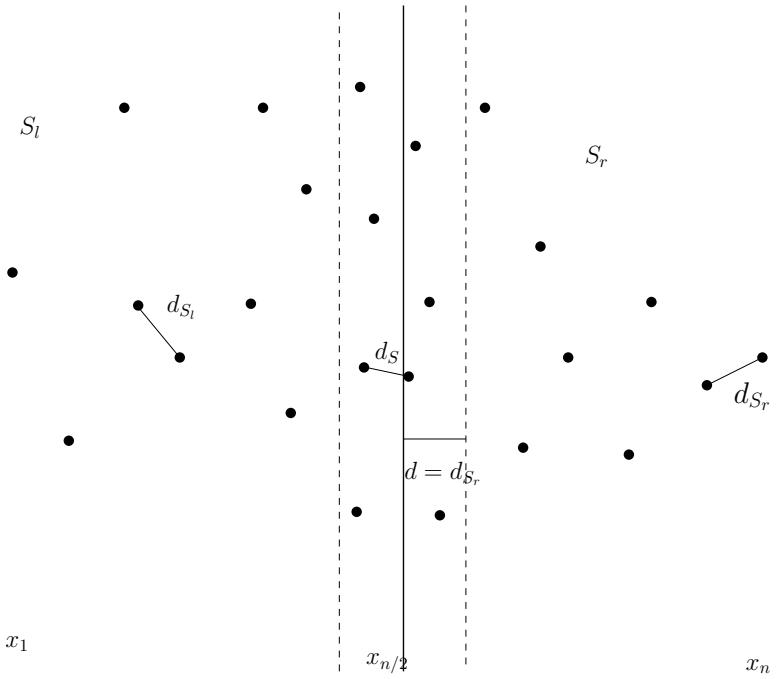

Abbildung 2.2: Für den Merge-Schritt müssen nur die Punkte der beiden Teilstreifen der Breite d entlang der Splitvertikalen betrachtet werden.

Verfahren angewendet:

- Bestimme die Punkte in den zugehörigen rechten und linken Teilstreifen der Breite d und sortiere diese jeweils nach Y-Koordinaten;

- gehe die Punkte im linken Teilstreifen nach fallenden Y-Koordinaten durch
- für jeden solchen Punkt p betrachte alle Punkte aus dem rechten Teilstreifen deren; Y-Koordinaten im Bereich $[p_y - d, p_y + d]$ liegen
- der zugehörige Punktbereich auf der rechten Seite wandert sukzessive nach *unten*.

Behauptung: Zu jedem Zeitpunkt sind nicht mehr als 10 Punkte im zugehörigen Bereich.

Beweis. (Behauptung) Nehmen wir an, dass p und q ein dichtestes Punktepaar mit Abstand $< d$ sind. Dann können wir oBdA annehmen, dass p im linken und q im rechten Teilstreifen liegt. Der Punkt q kann nicht außerhalb des Rechtecks $R(p)$ der Breite d und Höhe $2d$ im rechten Teilstreifen liegen; siehe Abbildung 2.3. Außerdem haben alle Punkte in $R(p)$ einen Abstand von mindestens d zueinander. Dann sind die Kreisumgebungen $U_{\frac{d}{2}}(q)$ in $R(p)$ disjunkt. Für jeden Punkt q in $R(p)$ ist mindestens ein Viertel dieser Kreisfäche in $R(p)$ enthalten. $R(p)$ kann dann höchstens

$$\frac{2d^2}{\frac{\pi}{4}\left(\frac{d}{2}\right)^2} = \frac{32}{\pi} < 11$$

viele Punkte enthalten. □

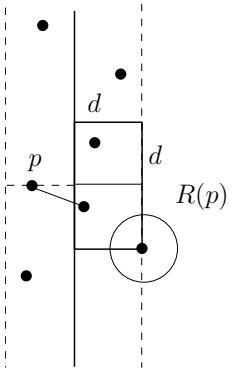

Abbildung 2.3: Für jeden Punkt p des linken Teilstreifens müssen nur konstant viele Punkte im rechten Teilstreifen getestet werden. Die zu testenden Punkte haben Y-Koordinaten, die im Bereich $[p_y - d, p_y + d]$ liegen.

Zur Analyse der Laufzeit betrachten wir nun die Einzelschritte des beschriebenen Algorithmus.

- In einem ersten Schritt werden die Punkte nach X-Koordinaten sortiert. Das kostet $O(n \log n)$ Zeit.
- Der Divide-Schritt kostet dann stets lineare Zeit.
- Für den Merge Schritt ergibt sich folgender Aufwand:
  1. Es müssen die Punkte mit X-Koordinaten aus den Teilstreifen ermittelt werden, das kostet $O(n)$ Zeit.
  2. Danach werden die Punkte in den jeweiligen Bereichen nach Y-Koordinate sortiert. Das kostet $O(n \log n)$ Zeit.
  3. In einem linearen Durchlauf wird für jeden Punkt p zu einer konstanten Anzahl (≤ 10) von Punkten q der Abstand ermittelt und der aktuell kleinste gespeichert. Insgesamt liegt der Aufwand in $O(n)$ für diesen Schritt.

Insgesamt erstellen wir daraus die folgende Rekursionsgleichung.

$$T(n) = \begin{cases} c & \text{falls } n = 1 \\ 2T\left(\frac{n}{2}\right) + c \cdot n \log n & \text{falls } n > 1 \end{cases} \tag{2.4}$$

Wie wir solche Rekursionsgleichungen im Allgemeinen lösen ist Gegenstand des nächsten Kapitels. Eine Übungsaufgabe besteht dann darin, die obige Gleichung abzuschätzen.

Kapitel 3

Lösen von Rekursionsgleichungen

Für die Analyse von Algorithmen, die das Prinzip der Rekursion verwenden, kann häufig eine Rekursionsgleichung aufgestellt werden.

Zunächst erwähnen wir einige Konventionen. Obwohl unsere Kostenfunktionen auf den natürlichen Zahlen definiert sind ignorieren wir diesen Sachverhalt und nehmen an, dass Funktionen wie $T(n)$ zwischen den ganzzahligen Argumenten monoton interpoliert werden. Deshalb können wir zum Beispiel auch für ungerade Zahlen n eine Formel wie $T(\frac{n}{2})$ hinschreiben. Außerdem können wir statt der genauen Laufzeit $U(n)$ eines Algorithmus, die manchmal nur mit umständlichen Fallunterscheidungen beschrieben werden kann, eine obere Schranke $T(n) \geq U(n)$ verwenden. Wenn wir dann eine Rekursionsgleichung für $T(n)$ aufstellen und lösen, haben wir auch eine Abschätzung von $U(n)$ nach oben. Schließlich lassen wir die Abschätzungen für die Anfangswerte von n oft weg, wenn es sich um Konstanten handelt, und konzentrieren uns auf die Rekursionsformel.

Wir wollen nun einige Beispiele für Rekursionsgleichungen betrachten.

Bei der *binären Suche* ist ein aufsteigend sortiertes Array $A = (a_1, a_2, \ldots, a_n)$ von Objekten a_i aus einer vollständig geordneten Menge M gegeben. Es soll festgestellt werden, ob ein Objekt $x \in M$ in diesem Array vorkommt. Zu diesem Zweck wird x zunächst mit dem mittleren Element a_{mitte} von A verglichen, wobei

$$\text{mitte} \;=\; \left\lfloor \frac{1+n}{2} \right\rfloor$$

gewählt wird. Gilt $x = a_{\text{mitte}}$, so hat man x gefunden und ist fertig. Ist $x < a_{\text{mitte}}$, so braucht man die Objekte rechts von a_{mitte} nicht mehr zu betrachten, denn das Array A ist ja aufsteigend sortiert. Man setzt die binäre Suche also rekursiv auf dem linken Teilstück $(a_1, a_2, \ldots, a_{\text{mitte}\,-1})$ von A fort.

Ganz analog wird die binäre Suche auf dem rechten Teil $(a_{\text{mitte}+1}, \ldots, a_n)$ fortgesetzt, falls $x > a_{\text{mitte}}$ gilt. Natürlich endet die Rekursion mit der Meldung "nicht gefunden", sobald das verbleibende Array keine Objekte mehr enthält.

Egal, ob n gerade oder ungerade ist, das verbleibende Array hat höchstens die Länge $\frac{n}{2}$. Deshalb erhalten wir als Laufzeitabschätzung für binäres Suchen die Rekursionsgleichung

$$T(n) = T\left(\frac{n}{2}\right) + c, \tag{3.1}$$

wobei die Konstante c den Aufwand für die Berechnung von a_{mitte} und den Vergleich mit x abschätzt.

Für das Sortierverfahren *Mergesort* hatten wir in Kapitel 2.1 schon die Formel

$$T(n) = 2\,T\left(\frac{n}{2}\right) + c\,n \tag{3.2}$$

aufgestellt, während wir für den Divide-and-Conquer Algorithmus für das Closest-Pair-Problem in Kapitel 2.2 die Gleichung

$$T(n) = 2\,T\left(\frac{n}{2}\right) + c\,n\log n \tag{3.3}$$

erhalten hatten.

Eine wichtige und immer wieder auftretende Aufgabe ist die *Multiplikation von Matrizen.* Berechnet man bei der Multiplikation

$$\begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & \cdots & b_{1n} \\ \vdots & \ddots & \vdots \\ b_{n1} & \cdots & b_{nn} \end{pmatrix} = \begin{pmatrix} c_{11} & \cdots & c_{1n} \\ \vdots & \ddots & \vdots \\ c_{n1} & \cdots & c_{nn} \end{pmatrix}$$

nach der Schulmethode alle Koeffizienten $c_{i,j}$ einzeln nach der Formel

$$c_{i,j} \;=\; \sum_{k=1}^{n} a_{ik}\, b_{kj},$$

so ergeben sich n^3 viele Multiplikationen von Zahlen. Ob es mit Divide-and-Conquer schneller geht?

Für gerade Zahlen n können wir die Matrizen durch jeweils vier Blockmatrizen der Größe $\frac{n}{2} \times \frac{n}{2}$ darstellen:

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}.$$

Jede der vier Blockmatrizen C_{ij} ergibt sich wegen

$$C_{ij} \;=\; A_{i1} \cdot B_{1j} \;+\; A_{i2} \cdot B_{2j}$$

durch Multiplikation von zwei Matrizen der Größe $\frac{n}{2} \times \frac{n}{2}$. Also erhalten wir die Rekursionsgleichung

$$T(n) = 8\,T\left(\frac{n}{2}\right) + c\,n^2, \tag{3.4}$$

bei der $c\,n^2$ die Kosten für die insgesamt vier Additionen von $\frac{n}{2} \times \frac{n}{2}$ - Matrizen abschätzt. *V. Strassen* entdeckte 1969 eine Möglichkeit, eine der acht Matrixmultiplikationen einzusparen. Er definierte sieben Hilfsmatrizen

$$\begin{aligned}
M_1 &:= (A_{11} + A_{22}) \cdot (B_{11} + B_{22}) \\
M_2 &:= (A_{21} + A_{22}) \cdot B_{11} \\
M_3 &:= A_{11} \cdot (B_{12} - B_{22}) \\
M_4 &:= A_{22} \cdot (B_{21} - B_{11}) \\
M_5 &:= (A_{11} + A_{12}) \cdot B_{22} \\
M_6 &:= (A_{21} - A_{11}) \cdot (B_{11} + B_{12}) \\
M_7 &:= (A_{12} - A_{22}) \cdot (B_{21} + B_{22}),
\end{aligned}$$

deren Berechnung jeweils eine Multiplikation von zwei $\frac{n}{2} \times \frac{n}{2}$ - Matrizen erfordert, und konnte nun die C_{ij} folgendermaßen daraus gewinnen:

$$\begin{aligned}
C_{11} &= M_1 + M_4 - M_5 + M_7 \\
C_{12} &= M_3 + M_5 \\
C_{21} &= M_2 + M_4 \\
C_{22} &= M_1 - M_2 + M_3 + M_6.
\end{aligned}$$

Bei diesem Ansatz werden also in der Rekursion nur 7 Multiplikationen benötigt, allerdings 18 Additionen und Subtraktionen. Somit ergibt sich die Rekursionsgleichung

$$T(n) = 7\,T\left(\frac{n}{2}\right) + c\,n^2, \tag{3.5}$$

mit einer etwas größeren Konstante c als in (3.4).

Die Beispiele (3.1) bis (3.5) sind Spezialfälle der allgemeinen Rekursionsgleichung

$$T(n) \;=\; a\,T\left(\frac{n}{b}\right) \;+\; f(n). \tag{3.6}$$

Die Konstante a beschreibt die Anzahl der Teile, in die das Problem zerlegt wird, und $\frac{n}{b}$ die Größe der Teilprobleme. Die Funktion $f(n)$ gibt den Gesamtaufwand beim Teilen und Zusammensetzen eines Problems der Größe

n an. Durch iterierte Substitution ergibt sich

$$\begin{aligned} T(n) &= a\,T\left(\frac{n}{b}\right) + f(n) \\ &= a\left(a\,T\left(\frac{n}{b^2}\right) + f\left(\frac{n}{b}\right)\right) + f(n) \;=\; a^2\,T\left(\frac{n}{b^2}\right) + a\,f\left(\frac{n}{b}\right) + f(n) \\ &= a^2\left(a\,T\left(\frac{n}{b^3}\right) + f\left(\frac{n}{b^2}\right)\right) + a\,f\left(\frac{n}{b}\right) + f(n) \\ &= a^3\,T\left(\frac{n}{b^3}\right) + a^2\,f\left(\frac{n}{b^2}\right) + a\,f\left(\frac{n}{b}\right) + f(n), \end{aligned}$$

und nun errät man leicht, dass in der i-ten Substitution

$$T(n) \;=\; a^i\,T\left(\frac{n}{b^i}\right) + \sum_{j=0}^{i-1} a^j\,f\left(\frac{n}{b^j}\right)$$

herauskommt. Strenggenommen muss man diese Gleichung durch vollständige Induktion über i beweisen; der Beweis ist aber so einfach, dass wir ihn weglassen. Nun nehmen wir an, dass $n = b^k$ eine Potenz von b ist und setzen $i = k$ ein. Wegen

$$a^k \;=\; a^{\log_b n} \;=\; n^{\log_b a}$$

ergibt sich

$$T(n) \;=\; n^{\log_b a}\,T(1) + \sum_{j=0}^{\log_b n-1} a^j\,f\left(\frac{n}{b^j}\right). \tag{3.7}$$

Mit dieser Formel können wir nun die Rekursionsgleichungen lösen, die wir oben aufgestellt hatten.

Beim *binären Suchen* (3.1) wird in Formel (3.7) $a = 1, b = 2$ und $f(n) = c$ gesetzt, und wir erhalten

$$T(n) \;=\; T(1) + \sum_{j=0}^{\log n-1} c \quad\in\; \Theta(\log n),$$

wobei $\log n$ stets $\log_2 n$ bedeutet. Für das Sortierverfahren *Mergesort* (3.2) wird in (3.7) $a = b = 2$ und $f(n) = cn$ gesetzt, und es ergibt sich

$$T(n) \;=\; n\,T(1) + \sum_{j=0}^{\log n-1} 2^j\,c\frac{n}{2^j} \quad\in\; \Theta(n \log n),$$

was wir auch Betrachtung des Rekursionsbaums schon herausgefunden hatten. Beim *Closest-Pair-Problem* (3.3) ist ebenfalls $a = b = 2$ aber $f(n) = cn \log n$. Hier erhalten wir

$$\begin{aligned} T(n) &= n\,T(1) + \sum_{j=0}^{\log n-1} 2^j\,c\frac{n}{2^j}\log\left(\frac{n}{2^j}\right) \\ &= n\,T(1) + cn\left(\log^2 n - \frac{(\log n - 1)\log n}{2}\right) \quad\in\; \Theta(n \log^2 n), \end{aligned}$$

wobei wir die Formel $\sum_{j=0}^{m} j = \frac{m(m+1)}{2}$ für $m = \log n - 1$ benutzt haben. Wir werden aber später sehen, dass das Problem *Closest Pair* sich noch effizienter lösen lässt.

Betrachten wir abschließend die *Matrixmultiplikation*. Zur Lösung der Rekursion (3.4) setzen wir in Formel (3.7) die Werte $a = 8, b = 2$ und $f(n) = n^2$ ein und bekommen

$$\begin{aligned} T(n) &= n^3\, T(1) + \sum_{j=0}^{\log n - 1} 8^j\, c \left(\frac{n}{2^j}\right)^2 \\ &= n^3\, T(1) + cn^2 \sum_{j=0}^{\log n - 1} 2^j \;\in\; \Theta(n^3), \end{aligned}$$

denn nach der Formel $\sum_{j=0}^{m} q^j = \frac{q^{m+1}-1}{q-1}$ ist der Wert der Summe in $\Theta(n)$. Die (naive) Rekursion auf 8 Produkte von Teilmatrizen ist also nicht besser als die Schulmethode. Verwendet man aber den Ansatz von Strassen (3.5), so hat a nur den Wert 7, und es folgt aus (3.7)

$$\begin{aligned} T(n) &= n^{\log 7}\, T(1) + \sum_{j=0}^{\log n - 1} 7^j\, c \left(\frac{n}{2^j}\right)^2 \\ &= n^{\log 7}\, T(1) + cn^2 \sum_{j=0}^{\log n - 1} \left(\frac{7}{4}\right)^j \;\in\; \Theta(n^{2.807\ldots}), \end{aligned}$$

denn es ist ja $\left(\frac{7}{4}\right)^{\log n} = \frac{n^{\log 7}}{n^2}$ und $\log 7 = 2.8073549\ldots$. Dieses Verfahren ist also deutlich schneller als die Schulmethode. Bis heute ist nicht bekannt, was der kleinstmögliche Exponent ω ist, für den zwei $n \times n$ - Matrizen sich in Zeit $O(n^\omega)$ multiplizieren lassen. Der Rekord liegt gegenwärtig bei $\omega = 2.37\ldots$.

Man hätte die Lösungen der obigen Rekursionsgleichungen auch durch Untersuchung der Rekursionsbäume erhalten können, wie anfangs am Beispiel von Mergesort vorgeführt. Welchem Verfahren man den Vorzug gibt, mag vom Geschmack abhängen; die Rekursionsbäume sind recht anschaulich, die Substitutionsmethode ist mathematisch präziser.

In der Literatur findet sich eine Diskussion der Formel (3.7) unter der Bezeichnung "Mastertheorem". Zum Beweis verwendet man ganz ähnliche Argumente wie wir sie gerade benutzt haben; siehe z.B. [1].

Theorem 3 *Seien $a \geq 1$ und $b > 1$ Konstanten und $f(n)$ eine nichtnegative Funktion. Dann gilt für die Rekursionsgleichung (3.6):*

1. *Wenn $f(n) \in O\left(n^{\log_b a - \epsilon}\right)$ für $\epsilon > 0$, dann gilt $T(n) \in \Theta\left(n^{\log_b a}\right)$.*

2. *Wenn* $f(n) \in \Theta\left(n^{\log_b a}\right)$, *dann gilt* $T(n) \in \Theta\left(n^{\log_b a} \log n\right)$.
3. *Wenn* $f(n) \in \Omega\left(n^{\log_b a+\epsilon}\right)$ *für* $\epsilon > 0$ *und a* $f(n/b) \leq c \cdot f(n)$ *für* $c < 1$ *und für hinreichend große* n, *dann gilt* $T(n) \in \Theta\left(f(n)\right)$.

3.1 Typische Schwierigkeiten

Bei der Lösung von Rekursionsgleichungen können u.a. typischerweise die folgenden Probleme auftreten:

1. Der Induktionsbeweis ist aufgrund der Konstanten fehlerhaft.
2. Die Lösungsform ist zu ungenau.
3. Die Lösungsform erfordert eine Variablentransformation.

Wir geben hier Beispiele für die obigen Probleme an.

Korrekte Konstanten: Für

$$T(n) \quad = \quad 2T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n$$

könnten wir

$$\begin{aligned} T(n) &\leq \left(c\left\lfloor \frac{n}{2} \right\rfloor\right) + n \\ &\leq c \cdot n + n \\ &= O(n) \qquad \text{Falsch!!} \end{aligned}$$

folgern, da c eine Konstante ist. Der Fehler liegt darin, dass wir nicht exakt $T(n) \leq cn$ bewiesen haben. Wir müssen die Konstante genau angeben.

Ein ganz ähnliches Problem kann auftreten, wenn man die O - Notation leichtfertig zur Behandlung von Summen nicht-konstanter Länge verwendet. Sind zum Beispiel alle Funktionen $f_i(n)$ in $O(g(n))$, so folgt daraus im allgemeinen *nicht*, dass die Funktion $F(n) = \sum_{i=0}^{n} f_i(n)$ in $O(ng(n))$ liegt. Beispiel: Alle Funktionen $f_i(n) = i$ sind in $O(1)$, aber $F(n)$ ist quadratisch und nicht linear.

Lösungsform ist zu ungenau: Für

$$T(n) \quad = \quad \begin{cases} 1 & \text{falls } n = 1 \\ 2T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + 1 & \text{falls } n > 1 \end{cases}$$

vermuten wir $T(n) \leq cn$. Die Induktionsannahme ist für $c > 1$ erfüllt. Leider führt

$$\begin{aligned} T(n) &\leq c\left\lfloor \frac{n}{2} \right\rfloor + c\left\lceil \frac{n}{2} \right\rceil + 1 \\ &= cn + 1 > cn \end{aligned}$$

im Induktionsschluss durch Substitution nicht zum Ziel. Wir müssen hier $T(n) \leq cn - 1$ wählen und dann für die Induktionsannahme $c > 2$.

Lösung durch Variablentransformation:

Für

$$T(n) \quad = \quad \begin{cases} 1 & \text{falls } n = 1 \\ 2T\left(\lfloor\sqrt{n}\rfloor\right) + \log n & \text{falls } n > 1 \end{cases}$$

können wir eine geeignete Umformung vornehmen. Durch $m = \log n$ gelangen wir vereinfacht zu

$T(2^m) = 2T\left(2^{m/2}\right) + m$ und erhalten durch Umbenennung die Gleichung $S(m) = 2S(m/2) + m$. Da, wie bereits gezeigt $S(m) = O(m \log m)$ gilt erhalten wir

$$T(n) = T(2^m) = S(m) \in O(m \log m) \in O(\log n \log \log n).$$

Kapitel 4

Dynamische Programmierung

Das Prinzip der dynamischen Programmierung kann dann angewendet werden, wenn eine rekursive Problemzerlegung an vielen Stellen die gleiche Teillösung verlangt. In diesem Fall ist es ratsam, die Lösungen dieser Teilprobleme zu speichern und gegebenenfalls darauf zurückzugreifen.

Falls die Teillösungen immer wieder neu berechnet werden, kann der Rechenaufwand exponentiell steigen. Grundsätzlich besteht die dynamische Programmierung aus einem rekursiven Aufruf für Teilprobleme und aus einer Tabellierung von Zwischenergebnissen die, wiederverwendet werden sollen.

Wir betrachten zunächst ein einfaches klassisches Beispiel, das diese wesentlichen Bestandteile bereits charakterisiert. Generell werden durch die dynamischen Programmierung meistens Optimierungsprobleme gelöst. In solche Fällen wird dann versucht, zu jedem Teilproblem eine optimale Lösung zu finden.

Dynamische Programmierung ist dann sinnvoll, wenn das sogenannte *Optimalitätsprinzip von Bellman* vorliegt:

Eine optimale Lösung hat stets eine Zerlegung in optimale Teilprobleme.

Ein solches Beispiel wird in Abschnitt 4.2 vorgestellt.

4.1 Fibonacci Zahlen

Wir betrachten die Fibonacci-Zahlen, die rekursiv wie folgt definiert werden.

$$\begin{aligned} \text{fib}(0) &= 1 && (1)\\ \text{fib}(1) &= 1 && (2)\\ \text{fib}(n) &= \text{fib}(n-1) + \text{fib}(n-2) && (3) \end{aligned}$$

Wollen wir nun beispielsweise fib(5) berechnen, so ergibt sich folgender Rekursionsbaum für die Aufrufe von fib; siehe Abbildung 4.1(I). Dieser Baum hat für fib(n) eine Größe von $\Omega\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$, wie eine Übungsaufgabe zeigt.

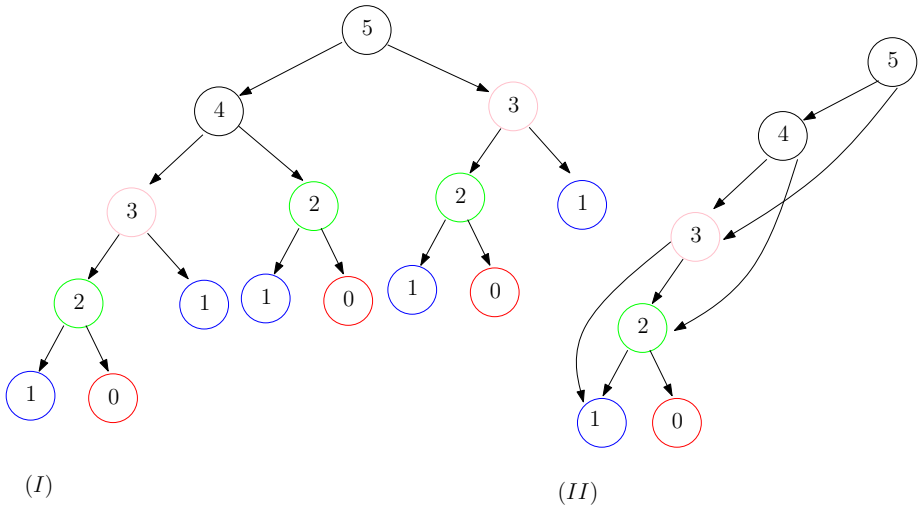

Abbildung 4.1: (I) Der vollständige Rekursionsbaum für fib(5) enthält einige Redundanzen. (II) Die rekursiven Abhängigkeiten können einfach dargestellt werden.

Offensichtlich ist es ratsam, einige der Ergebnisse zu speichern, die rekursiven Abhängigkeiten können einfacher dargestellt werden, wie in Abbildung 4.1(II). Wir wollen zum Beispiel fib(2) nicht dreimal wieder aus den Vorgängern neu berechnen, deshalb speichern wir die Werte ab. Es ist hier so, dass sich Teilprobleme überlappen und diese effizient gespeichert werden sollen. Der Abhängigkeitsgraph in Abbildung 4.1(II) hat nur eine Größe von $\Theta(n)$. Das Prinzip der Speicherung von Zwischenergebnissen wird auch *Memoisation* genannt.

Wenn wir nun die Abhängigkeiten nutzen wollen und von oben nach unten (Top-Down) die benötigten Werte fib(i) bestimmen wollen, könnten wir algorithmisch wie in Algorithmus 4 vorgehen, wobei FibMemo die memoisierende Prozedur 5 ist.

Algorithmus 4 FibTopDown(n)

```
1: F[1] := 1; F[0] := 1;
2: for i = 2 to n do
3:     F[i] := 0;
4: end for
5: RETURN FibMemo(n, F)
```

Hierbei ist FibMemo die folgende memoisierende Prozedur.

Prozedur 5 FibMemo(n,F)

```
1: if F[n] > 0 then
2:    RETURN F[n]
3: end if
4: F[n] := FibMemo(n − 1, F) + FibMemo(n − 2, F);
5: RETURN F[n]
```

In Algorithmus 4 wird das Feld F zunächst mit Nullen initialisiert, und dann beginnt den Aufruf der rekursiven Prozedur 5. Falls der Wert von $F[n]$ bekannt ist, geben wir diesen hier zurück, sonst wird er rekursiv gemäß der Rekursionsformel berechnet.

Laufzeitabschätzung: Wir benötigen im Algorithmus 4 $O(n)$ Zeit für die Initialisierung. Danach wird FibMemo(n, F) aufgerufen. Jeder Aufruf von FibMemo(m, F) in Prozedur 5 generiert höchstens einmal die Aufrufe FibMemo$(m-1, F)$ und FibMemo$(m-2, F)$. Für jedes $l < n$ wird somit FibMemo(l, F) höchstens zweimal aufgerufen. Ohne die rekursiven Aufrufe wird nur $O(1)$ Zeit in Prozedur 5 benötigt. Insgesamt gilt für die Laufzeit $T(n) \in O(n)$.

Wir machen nun die folgende Beobachtung. Obwohl im obigen Algorithmus $F[n]$ in einer Top-Down Vorgehensweise errechnet wird, werden die eigentlichen Werte von unten nach oben berechnet. Erst wenn die Rekursion bei $F[1]$ und $F[0]$ angekommen ist, werden neue Felder belegt und die Memoisation setzt ein. Wenn wir beispielsweis $F[5]$ berechnen, rufen wir sukzessive FibMemo$(5, F)$, FibMemo$(4, F)$, FibMemo$(3, F)$, FibMemo$(2, F)$ und FibMemo$(1, F)$ auf bis hier zum ersten Mal ein Wert $F[1]$ zurückgegeben wird. Dann wird wieder FibMemo$(0, F)$ aufgerufen und $F[2]$ belegt. Dann wird FibMemo$(1, F)$ nochmal aufgerufen und $F[3]$ belegt. Danach rufen wir FibMemo$(2, F)$ auf und belegen $F[4]$. Dann FibMemo$(3, F)$ und $F[5]$ wird belegt und ausgegeben.

Deshalb ist es offensichtlich sinnvoller gleich die Werte von unten nach oben (Bottom-Up) zu berechnen. Da für die Berechnung von $F[n]$ nur zwei Werte aus $F[0], \ldots, F[n-1]$ benötigt werden, sparen wir außerdem Speicherplatz. Die obige Memoisation und die Top-Down vorgehensweise ist hier bezüglich des Speicherplatzes nicht effizient.

Insgesamt erhalten wir folgende effiziente dynamische Programmierungslösung.

- Es müssen insgesamt nur $\Theta(n)$ viele Berechnungen durchgeführt werden.
- Die Bottom-Up Berechnung stellt sicher, dass die benötigten Werte stets vorliegen.

Algorithmus 6 FibDynProg(n)

```
1: F_letzt := 1; F_vorletzt := 1; F := 1;
2: for i = 2 to n do
3:    F := F_vorletzt + F_letzt;
4:    F_vorletzt := F_letzt; F_letzt := F;
5: end for
6: RETURN F
```

- Der Speicherplatzbedarf liegt in $O(1)$.

Beachte: Manchmal kann eine Top-Down Vorgehensweise auch sinnvoller sein, wenn zum Beispiel gar nicht alle Werte verwendet werden, die wir Bottom-Up berechnen würden. Die Prinzipien sind also nicht *dogmatisch* zu interpretieren. Es kann auch sinnvoll sein, die Zwischenergebnisse für mehrfache Aufrufe des Algorithmus hintereinander zu speichern.

Grundprinzip der dynamischen Programmierung:

- Formuliere das Problem insgesamt rekursiv. Stelle fest, dass sich Teillösungen überlappen und rekursiv voneinander abhängen.
- Stelle eine Rekursionsgleichnung (Top-Down) auf.
- Löse das Problem (Bottom-Up) durch das Ausfüllen von Tabellen mit Teillösungen (Memoisation).
- Ziel: Die Zahl der Tabelleneinträge soll klein gehalten werden.
- Anwendung: Bei einer direkten Ausführung der Rekursion ist mit vielen Mehrfachberechnungen zu rechnen.

Die Korrektheit der dynamischen Programmierung ergibt sich in der Regel dadurch, dass die rekursive Problembeschreibung korrekt durchgeführt wurde. Die Laufzeitanalyse ist in der Regel nicht schwer, da die Algorithmen typischerweise aus geschachtelten FOR-Schleifen bestehen. Das eigentliche Problem der dynamischen Programmierung ist es, eine geeignete rekursive Formulierung des Problems zu finden. Wir werden uns deshalb ein signifikantes Optimierungsbeispiel ansehen in dem auch das *Optimalitätsprinzip von Bellman* zum tragen kommt.

4.2 Matrixmultiplikation

Wir betrachten das Multiplizieren von reellwertigen Matrizen. Seien $A \in \mathbb{R}^{i \times j}$ und $B \in \mathbb{R}^{j \times k}$ Matrizen mit i Zeilen und j Spalten bzw. j Zeilen und

k Spalten, dann beschreibt $A \cdot B$ die Multiplikation dieser beiden Matrizen. Insgesamt sind zur Berechnung von $A \cdot B = C \in \mathbb{R}^{i \times k}$ $i \cdot k \cdot j$ Floating-Point Berechnungen notwendig. Genauer: Die Matrixeinträge von C werden durch

$$c_{st} = \sum_{l=1}^{j} a_{sl} \cdot b_{lt}$$

für alle $s = 1, \ldots i$ und alle $t = 1, \ldots, k$ berechnet, also $i \cdot j \cdot k$ Multiplikationen und Additionen.

Falls wir nun mehrere Matrizen multiplizieren wollen, also $E = A_1 \cdot A_2 \cdot \ldots \cdot A_n$ dann müssen auch hier die entsprechenden Matrizen gemäß der Größen *zusammenpassen*. Die Anzahl der Spalten von A_i muss der Anzahl der Zeilen von A_{i+1} entsprechen für $i = 1, \ldots, n-1$.

Die Matrixmultiplikation ist *assoziativ*, für drei Matrizen $A \in \mathbb{R}^{i \times j}$, $A \in \mathbb{R}^{j \times k}$ und $C \in \mathbb{R}^{k \times l}$ gilt also $(A \cdot B) \cdot C = A \cdot (B \cdot C) \in \mathbb{R}^{i \times l}$. Deshalb lassen sich auch allgemeine (zusammenpassende) Ketten beliebig Klammern. Diese Klammerung hat Einfluss auf die Anzahl der Floating-Point Operationen.

Beispiel: Für $A \in \mathbb{R}^{10 \times 50}$, $B \in \mathbb{R}^{50 \times 10}$ und $C \in \mathbb{R}^{10 \times 50}$, benötigt $(A \cdot B) \cdot C$ zuerst $10 \times 50 \times 10$ Operationen für $(A \cdot B)$ und danach $10 \times 10 \times 50$ Operationen für die Multiplikation des Ergebnisses mit C also insgesamt 10000 Operationen. Dahingegen benötigt $A \cdot (B \cdot C)$ zuerst $50 \times 10 \times 50$ Operationen für $(B \cdot C)$ und danach $10 \times 50 \times 50$ Operationen für die Multiplikation des Ergebnisses mit A also insgesamt 50000 Operationen. Die Klammerung $(A \cdot B) \cdot C$ ist also deutlich günstiger.

Allgemeine Problembeschreibung: Bestimme für die Berechnung von $E = A_1 \cdot A_2 \cdot \ldots \cdot A_n$ mit Dimensionen $d_0 \times d_1, d_1 \times d_2, \ldots, d_{n-1} \times d_n$ die optimale Klammerung für die minimale Anzahl an Operationen.

Dieses Problem wollen wir durch dynamische Programmierung lösen. Deshalb muss zunächst die Lösung rekursiv durch Teillösungen angegeben werden. Falls $M(n)$ alle Möglichkeiten beschreibt, wie man n Matrizen klammert, dann ist $M(n) = \sum_{k=1}^{n-1} M(k) \cdot M(n-k)$ mit $M(1) = 1$. Diese Rekursionsgleichung liegt in $\Omega(2^n)$. Es ist also nicht zweckmäßig, alle Möglichkeiten auszuprobieren.

Wir wollen zunächst herausfinden, wie groß die Anzahl $M(n)$ der Klammerungen von n Faktoren ist, und dabei auch unsere Kenntnisse über die Lösung von Rekursionen erweitern. Ein Trick besteht darin, die unbekannten Zahlen $M(n)$ als Koeffizienten einer formalen Potenzreihe

$$f(X) \;=\; \sum_{n=0}^{\infty} M(n)\, X^n$$

anzusehen und aus der Rekursionsgleichung für die $M(n)$ eine algebraische Gleichung für die sogenannte *erzeugende Funktion* $f(X)$ zu gewinnen. Der

Bequemlichkeit halber setzen wir $M(0) := 0$ und erhalten damit für alle $n \geq 2$ die Rekursionsgleichung

$$M(n) = \sum_{k=0}^{n} M(k)\, M(n-k).$$

Genau solche Summen (Faltungen oder Cauchy-Produkte genannt) treten als Koeffizienten von X^n auf, wenn man die Reihe $f(X)$ quadriert:

$$\begin{aligned} f(X)^2 &= \sum_{n=0}^{\infty} \Big(\sum_{k=0}^{n} M(k)\, M(n-k) \Big) X^n && (4.1) \\ &= 0 \cdot X^0 + 0 \cdot X^1 + \sum_{n=2}^{\infty} M(n)\, X^n && (4.2) \\ &= f(X) - 1 \cdot X^1. && (4.3) \end{aligned}$$

Die Summation in Gleichung (4.2) darf erst ab $n = 2$ beginnen, denn erst ab da gilt die Rekursionsgleichung. Für $n = 0$ und $n = 1$ hat die innere Summe in (4.1) den Wert 0 wegen $M(0) = 0$.

Lösen der quadratischen Gleichung

$$f(X)^2 - f(X) + X = 0$$

ergibt

$$f(X) = \frac{1 \pm \sqrt{1 - 4X}}{2}, \tag{4.4}$$

und die Wurzel hat die Reihendarstellung

$$\sqrt{1 - 4X} = \sum_{n=0}^{\infty} \binom{2n}{n} \frac{1}{1 - 2n}\, X^n; \tag{4.5}$$

man kommt darauf, indem man zum Beispiel in der Taylorreihenentwicklung für $\sqrt{1 + X}$ die Variable X durch $-4X$ ersetzt. Der Nenner in (4.5) wird für $n \geq 1$ negativ. Deshalb muss in (4.4) das Minuszeichen verwendet werden, und man erhält

$$\begin{aligned} f(X) &= \frac{1}{2} - \frac{1}{2} \sum_{n=0}^{\infty} \binom{2n}{n} \frac{1}{1 - 2n}\, X^n \\ &= \frac{1}{2} \sum_{n=1}^{\infty} \binom{2n}{n} \frac{1}{2n - 1}\, X^n, \end{aligned}$$

weil die Summe für $n = 0$ den Koeffizienten 1 hat. Durch Koeffizientenvergleich ergibt sich jetzt

$$M(n) = \frac{1}{2} \binom{2n}{n} \frac{1}{2n - 1}.$$

Zum Beispiel kommt für $n = 4$ der Wert $M(4) = 5$ heraus, und tatsächlich gibt es genau 5 Arten, um 4 Faktoren zu beklammern, nämlich

$$((ab)c)d, (a(bc))d, (ab)(cd), a((bc)d), a(b(cd)).$$

Die Zahl $C_{n-1} := M(n)$ wird die $n-1$-te *Catalan'sche Zahl* genannt. Um ihre Größe abzuschätzen, können wir die *Stirling-Formel*

$$n! \sim \sqrt{2\pi n}\, \frac{n^n}{e^n}$$

als asymptotische Näherung für die Fakultät anwenden (was bedeutet, dass der Quotient der beiden Seiten gegen 1 konvergiert,) und erhalten

$$\binom{2n}{n} \sim \sqrt{4\pi n}\, \frac{(2n)^{2n}}{e^{2n}}\, \frac{1}{2\pi n}\, \frac{(e^n)^2}{(n^n)^2} \quad \in \quad \Theta\left(\frac{2^{2n}}{\sqrt{n}}\right).$$

Also ist $M(n) \in \Theta\left(\frac{4^n}{n\sqrt{n}}\right)$ eine exponentiell wachsende Größe.

Wir wollen nun optimale Teillösungen wiederverwenden. Sei nun $K(l,k)$ der minimale Aufwand bzw. die optimale Klammerung für $A_l \cdot A_{l+1} \cdot \cdots \cdot A_k$. Zum Beispiel zerlegt sich zu Beginn ein erster Multiplikationsschritt für die Stelle s, d.h. der minimale Aufwand für $(A_1 \cdot A_2 \cdot \cdots \cdot A_s) \cdot (A_{s+1} \cdot A_{s+2} \cdot \cdots \cdot A_n)$, rekursiv in die Kosten

$$K(1,s) + K(s,n) + d_0 \cdot d_s \cdot d_n \tag{4.6}$$

Das Ziel ist nun, das optimale s zu finden, und das wollen wir rekursiv beschreiben.

- Offensichtlich ist $K(l,l) = 0$ für $l = 1, \ldots, n$
- Die Dimension eine Teilkette $A_l \cdot A_{l+1} \cdot \cdots \cdot A_k$ ist $d_{l-1} \times d_k$.
- Die Kosten für ein Teilen bei s sind $K(l,s) + d_{l-1} \cdot d_s \cdot d_k + K(s+1,k)$

Da wir $K(1,n)$ minimieren wollen, erhalten wir insgesamt:

$$K(l,k) = \begin{cases} 0 & \text{falls } l = k \\ \min_{l \le s \le k-1} K(l,s) + d_{l-1} \cdot d_s \cdot d_k + K(s+1,k) & \text{falls } l < k \end{cases}. \tag{4.7}$$

Wir stellen fest, dass dynamische Programmierung geeignet ist:

- Wiederverwendung: Hier werden viele Teilprobleme mehrfach benutzt, zum Beispiel $K(1,2)$ wird verwendet von $K(1,3), K(1,4), \ldots, K(1,n)$.

- Beschränkte Anzahl: Für alle $1 \leq l < k \leq n$ gibt es genau ein Teilproblem, also insgesamt $\Theta(n^2)$ viele.
- Bearbeitungsreihenfolge: Falls alle Werte $K(l, s)$ und $K(s + 1, k)$ bekannt sind, kann $K(l, k)$ in Zeit $O(k - l) \in O(n)$ berechnet werden.

Die geschickte Auswertung läßt sich sehr gut durch eine Matrix beschreiben. Wir verwenden ein Array $K[1..n, 1..n]$.

Zuerst werden die Diagonalen eingetragen, dann berechnen wir sukzessive die Nebendiagonalen von rechts unten nach links oben, zu jedem Zeitpunkt sind für $K(l, k)$ alle Werte $K(l, s)$ und alle Werte $K(s + 1, k)$ für $l \leq s$ und $s + 1 \leq k$ berechnet worden; siehe Abbildung 4.2.

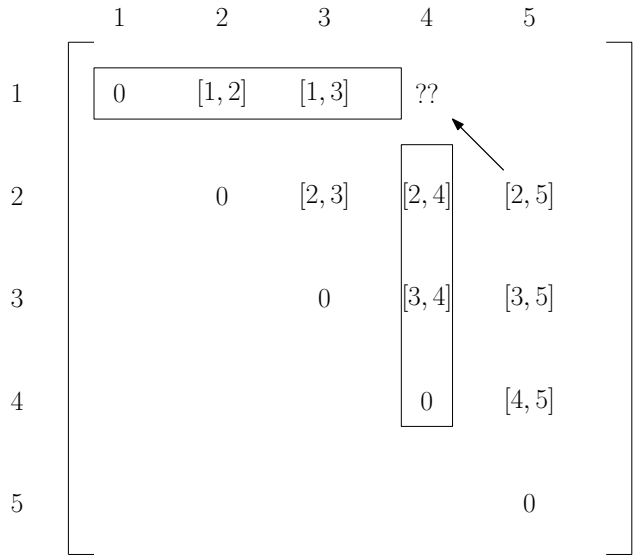

Abbildung 4.2: Die Nebendiagonalen können von unten nach oben berechnet werden. Nachdem zwei Nebendiagonalen bereits berechnet wurden beginnen wir die Berechnung der dritten Nebendiagonalen. Die benötigten Werte liegen bereits vor.

Korrektheit: Am Ende der Auswertung wird der Wert $[1, n]$ die minimale Anzahl an Operationen enthalten, wir haben den Wert rekursiv genauso festgelegt. Wir wollen uns natürlich auch die optimale Klammerung merken. Dazu müssen wir uns nur für jeden berechneten Matrixeintrag den jeweils optimalen Index $s[l, k]$ merken. Falls beispielsweise der optimale Index für die Berechnung von $[1, 5]$ bei $s = 2$ liegt, ist die Klammerung $(A_1A_2)(A_3A_4A_5)$ rekursiv die beste und wir speichern den Wert $s[1, 5] = 2$. Die optimale Klammerung von (A_1A_2) zur Berechnung von $[1, 2]$ ist trivialerweise $s = 1$ also $s[1, 2] = 1$. Rekursiv haben wir auch die optimale Klammerung aus

$[3,5]$ gespeichert, diese kann $s = 3$ oder $s = 4$ sein, somit ist $s[3,5] \in \{3,4\}$. Insgesamt erhalten wir so die optimale Klammerung.

Laufzeitanalyse und Speicherplatz: Insgesamt wird Speicherplatz von $O(n^2)$ für beide Matrizen (Einträge $K[l,k]$ und $s[l,k]$) verwendet. Für die Berechnung eines Eintrages $K[l,k]$ und der Klammerung $s[l,k]$ wird ein Aufwand von $O(k-l) \in O(n)$ benötigt. Da wir $O(n^2)$ viele Aufrufe haben, ergibt sich ein Gesamtaufwand von $O(n^3)$.

Übungsaufgabe: Formulieren Sie einen Pseudocode, der den obigen Algorithmus beschreibt.

4.3 Rucksackproblem

Wir betrachten das Problem des optimalen Befüllens eines Rucksackes mit Gesamtgewichtskapazität G. Dazu sei eine Menge von n Gegenständen a_i mit Gewicht $g_i > 0$ und Wert w_i gegeben. Wir wollen den Rucksack *optimal* füllen.

Aufgabenstellung: Bestimme eine Teilmenge $\{a_{i_1}, a_{i_2}, \ldots, a_{i_k}\}$ der Elemente aus $\{a_1, \ldots, a_n\}$, so dass $\sum_{j=1}^{k} g_{i_j} \leq G$ gilt und $\sum_{j=1}^{k} w_{i_j}$ maximal ist. Bestimme also den Wert:

$$w_{\max} = \max_{\{a_{i_1}, a_{i_2}, \ldots, a_{i_k}\} \subseteq \{a_1, \ldots, a_n\}} \left\{ \sum_{j=1}^{k} w_{i_j} \,\middle|\, \sum_{j=1}^{k} g_{i_j} \leq G \right\} .$$

Wenn wir dieses Problem rekursiv lösen wollen, können wir naiv einen Rekursionsbaum angeben, der in jeder Ebene i die Entscheidung für den i-ten Gegenstand berücksichtigt; siehe Abbildung 4.3. Dadurch werden alle möglichen Fälle beschrieben. Das ist aber nicht effizient, da der Baum eine exponentielle Größe.

Wir betrachten deshalb die folgende Vereinfachung und wollen Teilergebnisse verwenden. Sei dazu $W(i,j)$ der optimale Wert falls nur die ersten i Gegenstände betrachten werden und der Rucksack die Kapazität j hat. Am Ende soll $w_{\max} = W(n,G)$ berechnet werden.

Wir können $W(i,j)$ rekursiv wie folgt angeben. Die Frage stellt sich, ob bei Lösungen mit den ersten $i-1$ Gegenständen der i-te Gegenstand hinzugenommen werden soll oder nicht. Falls nicht, bleibt der Wert bei $W(i-1,j)$ stehen. Falls ja, kommt der Wert w_i hinzu. Allerdings müssen wir zur Berechnung von $W(i,j)$ dann rekursiv auf $W(i-1, j-g_i)$ zurückgreifen, $W(i-1, j-g_i) + w_i$ ist dann die Lösung. Der maximale Wert aus diesen beiden Fällen ergibt den Wert $W(i,j)$.

Falls j negativ wird, ist das Ergebnis ungültig. Das drücken wir durch

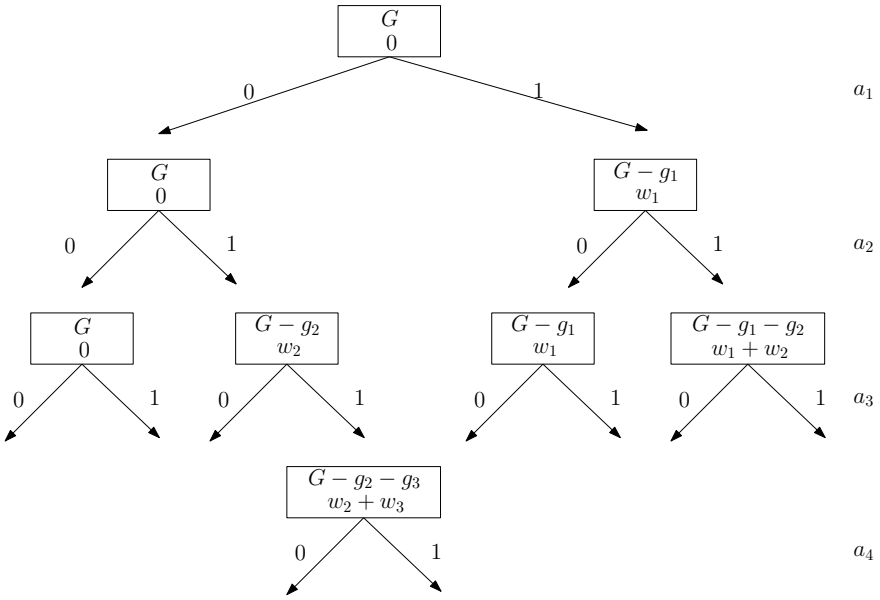

Abbildung 4.3: In Ebene i wird die Entscheidung für den i-ten Gegenstand getroffen. Der Baum hat exponentielle Größe.

$W(i,j) = -\infty$ aus. Falls i null wird, setzen wir den Wert $W(i,j)$ auf null, da kein Gegenstand verwendet wird.

Wir kommen zu folgender Rekursionsgleichung, die wir danach Bottom-Up lösen wollen.

$$W(i,j) = \begin{cases} 0 & \text{falls } i = 0 \\ -\infty & \text{falls } j < 0 \\ \max\{W(i-1,j), W(i-1,j-g_i) + w_i\} & \text{sonst} \end{cases} \quad (4.8)$$

Nun wollen wir wiederum die Werte $W(i,j)$ in einem Array $W[0..n, 0..G]$ berechnen und initialisieren $W[0,j]$ und $W[i,0]$ mit 0. Aus (4.8) ergibt sich offensichtlich, dass wir die Werte am besten Zeilenweise von links nach rechts und mit steigender Zeilenzahl berechnen. Zur Berechnung von $W(i,j)$ muss $W(i-1,j)$ und $W(i-1,j-g_i)$ vorliegen.

Beispiel: Für $n = 5$ notieren $a_i = (g_i, w_i)$ mit $a_1 = (6,4)$ $a_2 = (1,2)$, $a_3 = (2,3)$, $a_4 = (5,8)$ und $a_5 = (9,10)$ mit $G = 12$.

Wir beschreiben einen Teilausschnitt der zugehörigen Matrix während der Berechnung. Zur Berechnung von $W(2,7)$ wird $W(1,7) = 4$ (ohne a_2) und $W(1,7-g_2) + w_2 = W(1,6) + 2 = 4 + 2 = 6$ (mit a_2) verglichen. $W(1,7)$

ist also gleich 6. Zeilenweise wird die Matrix von links nach rechts gefüllt.

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	4	4	4	4	4	4	4
2	0	2	2	2	2	2	4	$W(2,7)$?	x	x	x	x	x
3	0	x	x	x	x	x	x	x	x	x	x	x	x
4	0	x	x	x	x	x	x	x	x	x	x	x	x
5	0	x	x	x	x	x	x	x	x	x	x	x	x

Auch für diesen Fall wollen wir uns am Ende die optimale Lösung generieren können. Dazu merken wir uns in einer zweiten Matrix $L[0..n, 0..G]$, woraus $W(i,j)$ entstanden ist. Zum Beispiel merken wir uns für $W(1,7)$ das a_2 benutzt wird und der Plan aus $L(1,6)$. Diese Information wird in $L(2,7)$ gespeichert, zum Beispiel durch $L[2,7] := (a_2, [1,6])$. In $L(1,6)$ steht dann wiederum, dass a_1 benutzt wird.

Korrektheit: Offensichtlich läßt sich nach Berechnung von $W(n,G)$ und $L(n,G)$ der Lösungsplan rekursiv angeben, und das optimale Packen ist berechnet worden. Die Lösung ist rekursiv genauso formuliert worden.

Wir geben hier nochmal explizit die Formulierung im Pseudocode in Algorithmus 7 an und analysieren dann die Korrektheit und die Laufzeit.

Algorithmus 7 RucksackDynProg($a_i = (g_i, w_i), i = 1, \ldots n$, Gesamtgew.: G)

```
1:  for i = 0 to G do
2:     W[0, i] := 0;
3:  end for
4:  for j = 0 to n do
5:     W[j, 0] := 0;
6:  end for
7:  for i = 1 to n do
8:     for j = 1 to G do
9:        if (j − g_i) < 0 then
10:          W[i, j] := W[i − 1, j];    // a_i passt gar nicht rein
11:       else
12:          W[i, j] := max{W[i − 1, j], W[i − 1, j − g_i] + w_i};
13:       end if
14:    end for
15: end for
16: RETURN W[n, G]
```

Laufzeit und Speicherplatz: In Algorithmus 7 werden nach der Initialisierung der Werte $W[i,0]$ und $W[0,j]$ zwei FOR-Schleifen geschachtelt,

wobei die Äußere von 1 bis n läuft und die Innere von 1 bis G. Im Inneren ist der Aufwand $\Theta(1)$. Also ist der Aufwand insgesamt in $O(n \cdot G)$ und für das Array benötigen wir $O(n \cdot G)$ Speicherplatz.

Übungsaufgabe: Füllen Sie die Matrizen $W[0..5, 0..12]$ und $L[0..5, 0..12]$ für das obige Beispiel. Formulieren Sie die algorithmische Lösung im Pseudocode auch für die Belegung der Matrix L zur Rekonstruktion der optimalen Lösung.

Bemerkungen: Der obige Algorithmus ist für ganzzahlige Gewichte entworfen worden, und seine Laufzeit hängt polynomiell von der Anzahl n der Gegenstände und der Größe G des Rucksacks ab. Während n linear in der Größe der Probleminstanz ist, gilt dies für G nicht, denn die Zahl G wird ja in der Eingabe durch $\log_2 G$ viele Bits dargestellt. Deshalb ist der oben vorgestellte Algorithmus *keine* polynomielle Lösung für das Rucksackproblem, und man vermutet, dass es keine gibt. Darauf werden wir im kommenden Semester noch ausführlicher eingehen.

Das Rucksackproblem wird deutlich einfacher, wenn die Gegenstände beliebig teilbar sind (Zucker, Mehl, ...). Dazu mehr im nächsten Kapitel.

Kapitel 5

Greedy Algorithmen

Neben den bereits behandelten klassischen Entwurfsmethoden der Algorithmik gibt es eine weitere Methode, die sich auf die intuitive Idee stütz, dass lokal optimale Berechnungsschritte auch global optimal sind. Es geht also auch in diesem Kapitel um Optimierungsprobleme. Der lokal optimale Berechnungsschritt soll dabei möglichst nicht wieder rückgängig gemacht werden. Außerdem werden keine Alternativen ausprobiert. In jedem Schritt wird der nächste lokal optimale Schritt berechnet und tatsächlich ausgeführt. In diesem Sinne nennt man diese Vorgehensweise auch *iterativ*. Die Kosten für einen *Iterationsschritt* sollen klein gehalten werden. Das ist meistens möglich, da wir zunächst nicht auf die Gesamtlösung achten. Zur Bestimmung des nächsten Schrittes wird allerdings häufig eine Liste von Kandidaten für den nächsten Schritt vorgehalten. Diese Liste wird nach dem Schritt aktualisiert. Greedy-Verfahren sind dennoch in der Regel einfach zu beschreiben und benötigen wenig Rechenaufwand.

Dieses *Greedy*-Prinzip wird leider nicht immer die optimale Lösung erzielen, drei Szenarien sind denkbar:

1. Wir erhalten die optimale Lösung.
2. Wir erhalten nicht die optimale Lösung, aber wir können zumindest beweisen, dass die Lösung im Vergleich zum Optimum nicht beliebig schlecht ist.
3. Die Lösung ist im Vergleich zum Optimum beliebig schlecht.

Zusammengefasst ergibt sich folgendes Grundprinzip der Greedy-Vorgehensweise:

- Lokal optimale Schritte werden iterativ ausgeführt.
- Eine Kandidatenliste wird dafür bereithalten und ggf. aktualisiert.

- Es werden keine Alternativen ausprobiert oder Zwischergebnisse tabellarisch abgespeichert.
- Die Kosten für den nächsten Schritt sind nicht sehr hoch.
- Die Lösung kann je nach Aufgabenstellung optimal, beliebig schlecht oder approximativ sein.

5.1 Rucksackproblem

Das bereits bekannte Rucksackproblem aus dem letzten Abschnitt läßt sich leicht in Greedy-Manier formulieren. Lokal optimal erscheint dabei, dass wir den Gegenstand mit dem jeweiligen besten Preis/Gewichtverhältnis als erstes Einfügen.

Beispiel: Für $n = 5$ notieren $a_i = (g_i, w_i)$ mit $a_1 = (6, 4)$ $a_2 = (1, 2)$, $a_3 = (2, 3)$, $a_4 = (5, 8)$ und $a_5 = (9, 10)$ mit $G = 12$. Die optimale Lösung ist hier offensichtlich (a_5, a_2, a_3) mit einem Wert von 15.

Zur Erinnerung: die Gegenstände haben ein Gewicht von g_i und einen Wert von w_i. Deshalb bestimmen wir zunächst eine Kandidatenliste gemäß des Verhältnisses $v_i = \frac{w_i}{g_i}$. Wir haben $v_1 = \frac{4}{6} = \frac{2}{3}$, $v_2 = \frac{2}{1} = 2$, $v_3 = \frac{3}{2} = 1.5$, $v_4 = \frac{8}{5}$ und $v_5 = \frac{10}{9}$. Das ergibt folgende Sortierung $v_2 > v_4 > v_3 > v_5 > v_1$. Ein Greedy Algorithmus wählt sukzessive den aktuell besten Kandidaten und revidiert diese Entscheidung nicht. Für den Algorithmus 8 nehmen wir zur einfachen Beschreibung an, dass die Gegenstände nach dem Preis/Gewichtsverhältnis sortiert vorliegen, also $v_1 \geq v_2 \geq \ldots \geq v_n$.

Algorithmus 8 RucksackGreedy($v_i = \frac{w_i}{g_i}$ nach Größe sortiert, Gewicht: G)

```
1: for i = 1 to n do
2:    if g_i ≤ G then
3:       λ_i := 1; G := G − g_i;
4:    else
5:       λ_i := 0;
6:    end if
7: end for
8: RETURN Σ_{i_1}^{n} λ_i w_i
```

Mit einer entsprechenden Umbenennung bedeutet das in unserem Beispiel, dass der Algorithmus sukzessive a_2 und a_4 und a_3 auswählt und dann die Kapazität $12 - 1 - 5 - 2 = 4$ verbleibt. Nun kann a_5 und a_1 nicht mehr verwendet werden. Insgesamt ergibt sich ein Gesamtwert von $2 + 8 + 3 = 13$. Der Greedy-Algorithmus liefert hier also keine optimale Lösung.

Schlimmer noch, wir können ein ganz einfaches Beispiel angeben, indem der Greedy-Algorithmus beliebig schlecht wird.

Worst-Case Beispiel: Sei $a_1 = (1, 1)$ und $a_2 = (G, G-1)$. Dann ist $v_1 = 1 > \frac{G-1}{G} = v_1$. Der Greedy Algorithmus legt $\lambda_1 = 1$ und $\lambda_2 = 0$ fest und das Ergebnis ist 1. Optimal wäre $\lambda_1 = 0$ und $\lambda_2 = 1$ mit Ergebnis $G-1$. Da G beliebig groß werden kann, wird das Verhältnis zwischen dem Wert der optimalen Lösung und dem Wert der Greedy-Lösung, $\frac{G-1}{1}$, beliebig groß.

Die Situation ändert sich grundlegend, wenn die zu packenden Gegenstände beliebig teilbar sind. Dann können wir von jedem $a_i = (g_i, w_i)$ einen beliebigen Anteil $\lambda_i \in [0, 1]$ in den Rucksack packen, der $\lambda_i \cdot g_i$ wiegt und den Wert $\lambda_i \cdot w_i$ beiträgt. In Algorithmus 8 wird dann in Zeile 5 die Zahl λ_i nicht auf null gesetzt sondern auf G/g_i. Die Gegenstände $a_1, \ldots, a_{i-1}$ werden also "ganz" eingepackt, von a_i aber nur so viel, wie gerade noch in den Rucksack hineinpasst.

Bei obigem Beispiel wird also zunächst $a_1 = (1, 1)$ ganz verpackt und von $a_2 = (G, G-1)$ der Anteil $\lambda_2 = \frac{G-1}{G}$. Damit ist der Rucksack voll und hat den Wert $1 + \lambda_2(G-1)$, also mehr als zuvor. Dass dieses gute Verhalten des Greedy-Verfahrens beim Rucksackproblem kein Einzelfall ist, zeigt folgendes Theorem.

Theorem 4 *Bei beliebig teilbaren Gegenständen findet der Greedy-Algorithmus in Zeit $O(n \log n)$ stets eine optimale Lösung für das Rucksackproblem.*

Beweis. Es wird zunächst Zeit $O(n \log n)$ gebraucht, um die Gegenstände $a_i = (g_i, w_i)$ nach absteigendem Nutzen $v_i = \frac{w_i}{g_i}$ zu sortieren; danach läuft Algorithmus 8 in linearer Zeit. Zum Nachweis der Optimalität stellen wir uns vor, wir hätten einen Gewichtsanteil γ des Rucksacks mit a_i oder a_j zu füllen, und es sei

$$\frac{w_i}{g_i} = v_i > v_j = \frac{w_j}{g_j}.$$

Nehmen wir a_j, so passt der Anteil $\frac{\gamma}{g_j}$ hinein und liefert den Wert $\frac{\gamma}{g_j} \cdot w_j$. Von a_i können wir den Anteil $\frac{\gamma}{g_i}$ einpacken und erhalten den Wert

$$\frac{\gamma}{g_i} \cdot w_i > \frac{\gamma}{g_j} \cdot w_j.$$

Also sollten stets Gegenstände mit maximalem Nutzen hinzugenommen werden. □

Es gibt also Fälle, in denen der Greedy-Ansatz versagt, und andere, in denen er optimale Lösungen findet. Manchmal kommt er dem Optimum sehr nahe, ohne es ganz zu erreichen. Um die *Approximationsgüte* für eine Problemklasse Π und die Instanzen $P \in \Pi$ zu messen, betrachten wir folgende Abschätzungen.

Sei Π ein Optimierungsproblem. Sei $Greed(P)$ der Gewinn der Greedy-Lösung und $OPT(P)$ der Gewinn der optimalen Lösung fr eine Probleminstanz $P \in \Pi$. Gilt für ein

Maximierungsproblem Π:

$$Greed(P) \;\geq\; \frac{1}{C} \cdot OPT(P) \quad - \quad A$$

Minimierungsproblem Π:

$$Greed(P) \;\leq\; C \cdot OPT(P) \quad + \quad A$$

für alle Instanzen $P \in \Pi$, so sagen wir, dass der Greedy-Algorithmus eine *C-Approximation* der optimalen Lösung liefert. Wichtig ist der *Approximationsfaktor* C, der nur von Π, nicht aber von P abhängen darf. Die additive Konstante A ist dazu da, konstanten Mehraufwand abzuschätzen; dazu gleich ein Beispiel beim Bepacken von Behältern.

Diese Definitionen kann man nicht nur für den Greedy-Algorithmus, sondern für beliebige Lösungsverfahren treffen. Solche *Approximationsalgorithmen* spielen eine wichtige Rolle bei der Lösung schwerer Probleme.

5.2 Das optimale Bepacken von Behältern

Das Problem des optimalen Bepackens von mehreren Behältern gleicher Gewichtskapazität G mit verschiedenen Paketen (Binpacking) gestaltet sich wie folgt. Die Pakete p haben keinen Preis sondern lediglich verschiedene Gewichte g. Insgesamt haben wir zunächst n Behälter b_j für $j = 1, \ldots, n$ mit Kapazität G und n Gegenstände p_i mit Gewichten g_i für $i = 1, \ldots, n$. Wir können $g_i \leq G$ annehmen, sonst brauchen wir einen Gegenstand gar nicht zu betrachten.

Ziel der Aufgabenstellung ist es, die Anzahl der benötigten Behälter zu minimieren. Im Prinzip bestimmen wir eine Zuordnung $Z : \{1, \ldots, n\} \mapsto \{1, \ldots, k\}$ die jedem p_i einen Behälter b_j über $Z(i) = j$ zuweist. Wir suchen das minimale k, für das es eine gültige Zuordnung Z gibt. Gültig heißt, dass die Summe aller g_i mit $Z(i) = j$ nicht größer als G ist.

Aufgabenstellung: Minimiere k, so dass eine Zuordnung $Z : \{1, \ldots, n\} \mapsto \{1, \ldots, k\}$ existiert mit

$$\sum_{Z(i)=j} g_i \leq G \text{ für alle } i \in \{1, \ldots, n\}\,. \tag{5.1}$$

Für eine Greedystrategie benutzen wir hier keine besondere Kandidatenliste, wir nehmen die Pakete so wie sie gegeben sind in der Reihenfolge $a_1, a_2, \ldots, a_n$. Die Annahme ist wirklich sinnvoll, da wir uns vorstellen können, dass die Pakete so vom Band kommen oder so geliefert werden und zugeordnet werden müssen.

Wir berücksichtigen nicht, welche Pakete in der Zukunft auftreten. Die einzige Entscheidung die wir treffen, ist also, in welchen Behälter b_j legen wir das Paket a_i nachdem bereits die Pakete $a_1, a_2, \ldots, a_{i-1}$ einsortiert wurden. Der maximale Index j soll klein gehalten werden. Naheliegend sind die folgenden beiden Vorgehensweisen. Angenommen, wir haben bereits $a_1, a_2, \ldots, a_{i-1}$ Pakete zugeordnet. Platziere nun a_i.

First-Fit: Unter allen Behältern wähle den ersten aus, in den a_i noch hineinpasst. Genauer: Finde kleinstes j, so dass $g_i + \sum_{1 \leq l \leq (i-1), Z(l)=j} g_l \leq G$ gilt.

Best-Fit: Packe a_i in die aktuell vollste Kiste, in die es noch hinein passt. Genauer: Finde ein j, so dass $g_i + \sum_{1 \leq l \leq (i-1), Z(l)=j} g_l \leq G$ gilt und dabei $\sum_{1 \leq l \leq (i-1), Z(l)=j} g_l$ maximal ist.

Beispiel: Betrachten wir Behälter der Kapazität $G = 10$ und 7 Gegenstände der Größen $g_1 = 2$, $g_2 = 5$, $g_3 = 4$, $g_4 = 7$, $g_5 = 1$, $g_6 = 3$ und $g_7 = 8$ dann belegt First-Fit die Behälter wie in Abbildung 5.1 und benötigt 4 Behälter. Optimal wäre es gewesen, drei Behälter jeweils mit (g_1, g_7), (g_2, g_3, g_5) und (g_4, g_6) zu belegen. Beide Algorithmen lassen sich mit einer Laufzeit von

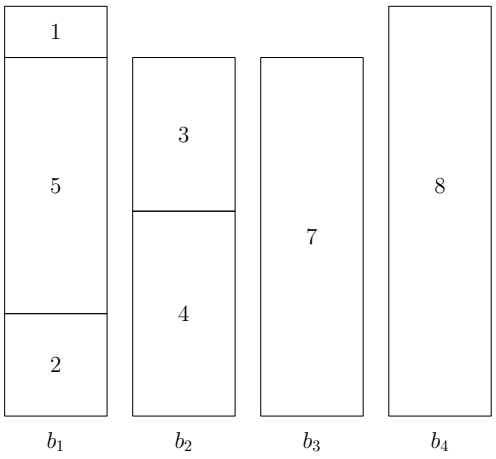

Abbildung 5.1: First-Fit für $g_1 = 2$, $g_2 = 5$, $g_3 = 4$, $g_4 = 7$, $g_5 = 1$, $g_6 = 3$ und $g_7 = 8$ benötigt einen Behälter zu viel.

$O(n^2)$ realisieren. Beispielsweise wird bei First-Fit für alle Behälter j der aktuelle Belegungswert gespeichert und wir überprüfen sukzessive von $j = 1, \ldots, n$ in welchen Behälter der Gegenstand a_i noch hineinpasst. Dieses Verfahren wird wiederum sukzessive für $i = 1, \ldots, n$ für die Gegenstände a_i durchgeführt. Insgesamte werden zwei FOR-Schleifen geschachtelt, die von $j = 1, \ldots, n$ respektive von $i = 1, \ldots, n$ laufen. Ähnlich kann bei Best-Fit vorgegangen werden.

Wir wollen nun zeigen, dass beide Algorithmen eine 2-Approximation der optimalen Anzahl der benötigten Behälter liefern. Wir gehen auch hier der Einfachheit halber davon aus, dass die Gegenstände ganzzahliges Gewicht haben.

Lemma 5 *Für das Binpacking Problem liefern die Algorithmen First-Fit und Best-Fit eine 2-Approximation für die optimale (minimale) Anzahl an benötigten Behältern.*

Beweis. Wir betrachten zunächst den Algorithmus First-Fit. Angenommen, er benötigt k Behälter, während die optimale Lösung mit $k_{\min}$ Behältern auskommt. Wesentlich ist nun die folgende

Beobachtung: Alle bis auf höchstens einen der von First-Fit benutzten Behälter sind mindestens halb voll.

Denn angenommen, es gäbe am Schluss zwei höchstens halb volle Behälter b_i und b_j mit $i < j$. Sei a der erste Gegenstand, der in b_j abgelegt wurde. Er hat höchstens das Gewicht $G/2$ (weil er in dem höchstens halb vollen Behälter b_j liegt) und hätte deshalb in b_i noch Platz gehabt (weil b_i selbst am Ende höchstens halb voll ist). Das ist ein Widerspruch zu $i < j$, denn FirstFit hätte ja erst testen müssen, ob a in b_i hineinpasst.

Aus der Beobachtung folgt:

$$\text{Gesamtgewicht} \;\geq\; (k-1) \cdot \frac{G}{2},$$

also

$$k \;\leq\; 2\,\frac{\text{Gesamtgewicht}}{G} \;+\; 1 \;\leq\; 2\,k_{\min} \;+\; 1,$$

denn auch die optimale Lösung benötigt mindestens so viele Behälter wie das Gesamtgewicht aller Gegenstände, geteilt durch die Behälterkapazität. Man sieht, dass hier tatsächlich eine additive Konstante in der Abschätzung auftritt.

Für BestFit wird die obige Beobachtung fast genauso bewiesen: Angenommen, Behälter b_i ist schon benutzt worden, wenn der erste Gegenstand a in b_j abgelegt wird. Dann wäre für a auch in b_i oder in einem anderen, noch volleren Behälter Platz gewesen. □

Durch schärfere Analyse lässt sich zeigen, dass FirstFit und BestFit sogar $\frac{17}{10}$ - Approximationen liefern, aber nichts Besseres. Wir zeigen jetzt, wie man die beiden Verfahren so "ärgern" kann, dass sie $\frac{5}{3}$ mal so viele Behälter brauchen wie die optimale Lösung.

Lemma 6 *Für das Binpacking Problem und die Algorithmen First-Fit und Best-Fit gibt es Beispiel-Sequenzen, die beliebig lang sein können und für die die Lösung aus den beiden Algorithmen niemals besser sein kann als $\frac{5}{3}$ mal die optimale Lösung.*

Beweis. Sei dazu $n = 18m$ und $G = 131$. Gegenstände mit Gewicht 20 plus Gewicht 45 plus Gewicht 66 füllen genau einen Behälter.

Insgesamt betrachten wir $18m$ Gegenstände, wobei in der Sequenz der Gegenstände

1. ... die ersten $6m$ ein Gewicht $g_i = 20$ für $i = 1, \ldots, 6m$,
2. ... die nächsten $6m$ ein Gewicht von $g_i = 45$ für $i = 6m + 1, \ldots, 12m$,
3. ... und die letzten $6m$ ein Gewicht von $g_i = 66$ für $i = 12m+1, \ldots, 18m$

besitzen. Wir haben also drei Sequenzblöcke 1., 2. und 3. und exakt $6m$ Behälter reichen aus.

Wir zeigen, dass für diese Sequenz First-Fit und Best-Fit jeweils $10m$ Behälter befüllen. Beide Algorithmen füllen für den ersten Sequenzblock sukzessive die ersten m Behälter mit Gewichten $6 \times 20 = 120$. Jeder Block hat eine Restkapazität von 11. Danach werden für den zweiten Block $3m$ Behälter mit jeweils 2×45 Gewicht und Restkapazität von 41. Danach werden $6m$ Behälter mit Gewicht 66 gefüllt und Restkapazität jeweils 65.

Insgesamt ergibt sich in beiden Fällen ein Quotient von $\frac{Greedy(P)}{OPT(P)} = \frac{10m}{6m} = \frac{5}{3}$. □

Wenn die Sequenzen aus dem obigen Beweis in absteigend sortierter Reihenfolge vorgelegen hätten, dann hätten beide Algorithmen die optimale Belegung mit $6m$ Behältern erzielt.

5.3 Aktivitätenauswahl

Wir betrachten das Problem, dass eine Ressource für verschiedene zeitlich beschränkte Aktivitäten benutzt werden soll. Die Aktivitäten können die Resource nicht zeitgleich benutzen. Es sollen aber soviel wie möglich Aktivitäten eingeplant werden. Beispielsweise soll die Belegung eines Konzertsaals für verschiedene Veranstaltungen geplant werden. Allgemein gibt es

eine Menge von n Aktivitäten $a_i = (s_i, t_i)$ die jeweils durch einen Starttermin, s_i, und einen Endtermin, t_i, mit $s_i < t_i$ bestimmt sind.

Zwei Aktivitäten a_i und a_j sind kompatibel, wenn die Intervalle $[s_i, t_i)$ und $[s_j, t_j)$ nicht überlappen.

Aufgabenstellung Aktivitäten-Auswahl: Für eine Menge von n Aktivitäten $a_i = (s_i, t_i)$ für $i = 1, \ldots, n$ finde die maximale Menge gegenseitig kompatibler Aktivitäten. Mit maximal ist hier die Anzahl der einplanbaren Aktivitäten gemeint.

Zunächst ist klar, dass wir die Aktivitäten zwischen dem kleinsten Startzeitpunkt $S = \min\{s_i | i = 1, \ldots, n\}$ und dem größten Endzeitpunkt $T = \max\{t_i | i = 1, \ldots, n\}$ einplanen müssen. Eine naheliegende Greedy-Idee ist, die bislang nicht verwendete Zeit zu maximieren. Deshalb sortieren wir die Aktivitäten nach den Endzeitpunkten t_i. Nehmen wir an, dass wir die Folge a_i bereits so vorliegen haben, das heißt $t_1 \leq t_2 \leq \cdots \leq t_n$.

Greedy-Strategie: Wähle stets die Aktivität, die den frühesten Endzeitpunkt hat und noch legal eingeplant werden kann.

Beispiel:

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
t_i	4	5	6	7	8	9	10	11	12	13	14

Hier sind zum Beispiel $\{a_3, a_9, a_{11}\}$ kompatibel, $\{a_1, a_4, a_8, a_{11}\}$ ist eine maximale kompatible Teilmenge, die durch die Greedy Strategie ausgewählt wird. Auch $\{a_2, a_4, a_9, a_{11}\}$ ist optimal bezüglich der Anzahl der einplanbaren Aktivitäten.

Algorithmisch beschreiben wir die Greedy-Strategie ganz einfach wie in Algorithmus 9.

Algorithmus 9 AktivitätenGreedy($a_i = (s_i, t_i)$ nach t_i sortiert)

```
1: A_1 := {a_1};
2: last := 1;    // last ist der Index der zuletzt eingefügten Aktivität
3: for i = 2 to n do
4:    if s_i ≥ t_last then
5:       A_i := A_{i-1} ∪ {a_i}; last := i;
6:    else
7:       A_i := A_{i-1};
8:    end if
9: end for
10: RETURN A_n
```

Laufzeit: Es müssen lediglich vorab die Endzeitpunkte sortiert werden. Das kann in $O(n \log n)$ durchgeführt werden. Danach wird die Liste in $O(n)$ Zeit

in der FOR-Schleife abgearbeitet.

Korrektheit: Der Algorithmus ist einfach und liefert eine Lösung, das ist ein Grundprinzip der Greedy-Algorithmen. Die Analyse, dass die Lösung optimal ist, kann manchmal sehr schwer sein. Im Folgenden versuchen wir die Optimalität formal zu beweisen.

Theorem 7 *Für das Problem der Aktivitäten-Auswahl berechnet der Greedy-Algorithmus 9 in Zeit $O(n \log n)$ eine optimale Lösung.*

Beweis. Am Anfang müssen die Aktivitäten nach Endzeitpunkten sortiert werden; danach kann Algorithmus 9 in linearer Zeit laufen. Die Optimalität lässt sich folgendermaßen beweisen. Angenommen, das Greedy-Verfahren wählt k Aktivitäten $a_{i_j} = (s_{i_j}, t_{i_j})$ aus. Für jedes j sei T_j die Menge aller Aktivitäten aus der Eingabe, die den Endpunkt t_{i_j} enthalten. Alle Intervalle in T_j überlappen sich; also kann auch eine optimale Lösung OPT aus jeder Menge T_j höchstens *ein* Intervall enthalten. Das sind insgesamt höchstens k viele.

Angenommen, in OPT kommt ein weiteres Intervall $a = (s, t)$ vor, das in keiner Menge T_j auftaucht, weil es keinen der Endpunkte t_{i_j} enthält. Dieses Intervall a kann dann nur komplett vor einem Intervall a_{i_j} liegen, oder dessen Startpunkt s_{i_j} enthalten, den Endpunkt t_{i_j} aber nicht. Dann hätte das Greedy-Verfahren aber Intervall a ausgewählt, weil es sich mit $a_{i_{j-1}}$ nicht überlappt und sein Endzeitpunkt t vor t_{i_j} kommt. Widerspruch!

Daraus folgt, dass OPT nicht mehr Intervalle enthalten kann, als das Greedy-Verfahren auswählt. □

Insgesamt haben wir drei Greedy Beispiele kennengelernt, in denen entweder eine optimale Lösung, eine gute Approximation oder eine beliebig schlechte Lösung erzielt wird.

Literaturverzeichnis

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. Introduction to Algorithms. MIT Press, 3. Auflage, 2009.

[2] Norbert Blum. Algorithmen und Datenstrukturen: Eine anwendungsorientierte Einfhrung. Oldenbourg Verlag, 2004.