

Dynamische Programmierung

Hallo Welt für Fortgeschrittene

Frank Blendinger

`fb@intoxicatedmind.net`

31. Mai 2005

1 Motivation

- Was ist Dynamische Programmierung?
- Geschichtliches
- Einführendes Beispiel: Fibonacci-Zahlen

2 Grundlagen

- Memoization
- Optimale Teilprobleme
- Top-down vs. bottom-up
- Ein weiteres Beispiel: Binomialkoeffizienten

3 Praktische Anwendung

- Entwurf eines DP-Algorithmus
- Das Rucksackproblem

4 Fazit und Ausblick

- Zusammenfassung
- Probleme
- Einsatzgebiete

5 Literatur

Um was geht es?

Definition

Dynamische Programmierung (DP) ist ein algorithmisches Muster, das meist zur Lösung von **Optimierungsproblemen** eingesetzt wird.

- typischer Einsatz: Suche nach einer ein **Maximum oder Minimum** erzielenden Kombination
- nicht eine, sondern die beste aus vielen möglichen Lösungen interessiert
- auch häufig gesucht: die **Anzahl aller Lösungen** zu einem Problem

Wie wird es gemacht?

- „dumme“ Herangehensweise wäre, alle Möglichkeiten einfach durchzuprobieren
- hierbei werden oftmals Teilkombinationen wieder und wieder probiert
- diesen Aufwand spart man sich beim DP, indem man **Zwischenergebnisse speichert** und auf diese bei Bedarf zurückgreift

Wer hat's erfunden?

Wer hat's erfunden?

...nein, nicht die Schweizer! ;)

Wer hat's erfunden?

... nein, nicht die Schweizer! ;)



- Begriff in den 1940er Jahren vom amerikanischen Mathematiker RICHARD BELLMAN (1920–1984) geprägt
- eigentlich ein mathematisches Verfahren, der Begriff „Programmierung“ bezieht sich auf das Ausfüllen einer Tabelle mit Zwischenergebnissen

Alte Bekannte: die Fibonacci-Zahlen

Definition

Fibonacci-Zahlen

$$fib(1) = fib(2) = 1$$

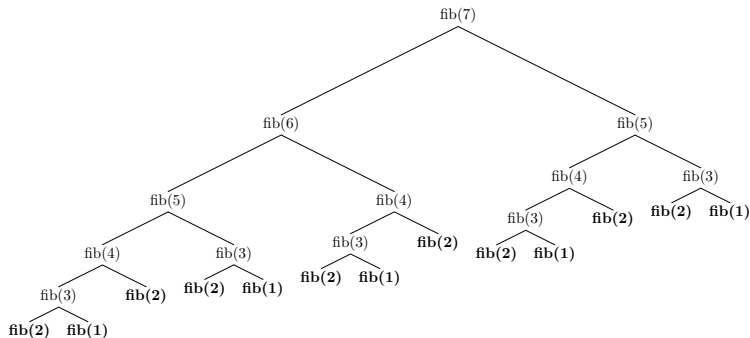
$$fib(n) = fib(n-2) + fib(n-1) \quad \forall n > 2$$

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

- rekursive Implementierung naheliegend
- aber: wie sieht es mit dem Aufwand aus?

Fibonacci rekursiv

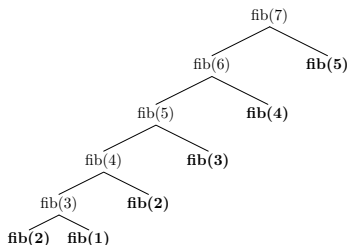
Rekursionsbaum für die Berechnung von `fib(7)`:



- **mehrfache Berechnung** von Zwischenergebnissen
- exponentieller Aufwand: $\mathcal{O}(2^n)$
(präziser: $\Omega(1.6^n)$)

Fibonacci rekursiv - aber diesmal mit Hirn!

- weiterhin Rekursion
- berechnete Fibonacci-Zahlen in Tabelle speichern
- vor Rekursionsabstieg erst Blick in die Tabelle, ob Ergebnis nicht bereits vorliegt



- Ergebnis: nur noch **linearer Aufwand** $\mathcal{O}(n)$

Fibonacci rekursiv - aber diesmal mit Hirn!

```
long fib(int n) {  
  
    if (n == 1 || n == 2) return 1;  
  
    return fib(n-1) + fib(n-2);  
  
}
```

Fibonacci rekursiv - aber diesmal mit Hirn!

```
long memo[MAX_N];  
memset(memo, UNSET, MAX_N * sizeof(long));  
memo[1] = 1; memo[2] = 1;  
  
long fib(int n) {  
  
    if (n == 1 || n == 2) return 1;  
  
    return fib(n-1) + fib(n-2);  
  
}
```

Fibonacci rekursiv - aber diesmal mit Hirn!

```
long memo[MAX_N];  
memset(memo, UNSET, MAX_N * sizeof(long));  
memo[1] = 1; memo[2] = 1;  
  
long fib(int n) {  
  
    if (n == 1 || n == 2) return 1;  
  
    return fib(n-1) + fib(n-2);  
  
}
```

Fibonacci rekursiv - aber diesmal mit Hirn!

```
long memo[MAX_N];  
memset(memo, UNSET, MAX_N * sizeof(long));  
memo[1] = 1; memo[2] = 1;  
  
long fib(int n) {  
    if (memo[n] != UNSET) return memo[n];  
    if (n == 1 || n == 2) return 1;  
  
    return fib(n-1) + fib(n-2);  
  
}
```

Fibonacci rekursiv - aber diesmal mit Hirn!

```
long memo[MAX_N];  
memset(memo, UNSET, MAX_N * sizeof(long));  
memo[1] = 1; memo[2] = 1;  
  
long fib(int n) {  
    if (memo[n] != UNSET) return memo[n];  
    if (n == 1 || n == 2) return 1;  
  
    return fib(n-1) + fib(n-2);  
  
}
```

Fibonacci rekursiv - aber diesmal mit Hirn!

```
long memo[MAX_N];  
memset(memo, UNSET, MAX_N * sizeof(long));  
memo[1] = 1; memo[2] = 1;  
  
long fib(int n) {  
    if (memo[n] != UNSET) return memo[n];  
  
    return fib(n-1) + fib(n-2);  
  
}
```

Fibonacci rekursiv - aber diesmal mit Hirn!

```
long memo[MAX_N];  
memset(memo, UNSET, MAX_N * sizeof(long));  
memo[1] = 1; memo[2] = 1;  
  
long fib(int n) {  
    if (memo[n] != UNSET) return memo[n];  
  
    memo[n] = fib(n-1) + fib(n-2);  
    return fib(n-1) + fib(n-2);  
  
}
```

Fibonacci rekursiv - aber diesmal mit Hirn!

```
long memo[MAX_N];  
memset(memo, UNSET, MAX_N * sizeof(long));  
memo[1] = 1; memo[2] = 1;  
  
long fib(int n) {  
    if (memo[n] != UNSET) return memo[n];  
  
    memo[n] = fib(n-1) + fib(n-2);  
  
    return memo[n];  
}
```

Grundlage eines jeden DP-Algorithmus: Memoization

Definition

Unter **Memoization** versteht man allgemein die **Speicherung der Ergebnisse** einer Funktion, um zu einem späteren Zeitpunkt auf diese zurückgreifen zu können, anstatt sie neu berechnen zu müssen.

Memoization: wie implementieren?

- Tabelle je nach Anwendungsfall ein- oder mehrdimensionales Array
- meist kann (oder muss) die Tabelle mit einer Anzahl von Grundfällen vorinitialisiert werden
- Rest der Tabelle wird auf einen speziellen Wert außerhalb des Lösungsraums gesetzt, der für „noch nicht berechnet“ steht
- erste Anweisung bei jedem Funktionsaufruf ist ein Tabellenlookup
 - **Wert bereits in Tabelle:**
zurückliefern
 - **Wert noch nicht in Tabelle:**
berechnen, in Tabelle speichern, zurückliefern

Aus eins mach viele: optimale Teilprobleme

- ursprüngliches Problem wird in eine **Kombination aus kleineren Teilproblemen** zerlegt
- Teilprobleme sind ihrerseits optimal
- im Gegensatz zu Divide-and-Conquer-Algorithmen müssen die Teilprobleme nicht unabhängig voneinander sein
- wir erhalten eine **Rekursionsgleichung** für das Problem
- Basisfälle (z.B. $n = 1$) aufstellen

Schwarze Schafe: überlappende Teilprobleme

- bei Berechnung der Teilprobleme kann es zu **Überschneidungen** kommen
- man spricht von sog. „überlappenden Teilproblemen“
- diese werden mittels **Memoization** erschlagen

Memoization + Teilprobleme = DP

Top-down: vom Problem zur Lösung

- bisher Effizienzgewinn durch Memoization
- aber noch immer großer **Overhead durch Funktionsaufrufe** insbesondere bei hohen Rekursionstiefen
- bei extrem großen Eingaben kann sogar der Speicherbedarf problematisch werden – es drohen **Stack Overflows!**
- rekursive Berechnung ist ein **top-down** Ansatz

Bottom-up: von der Lösung zum Problem

- oft ist es geschickter einen **bottom-up** Ansatz zu verfolgen
- hierbei wird die Tabelle sukzessiv mit den Teillösungen aufgefüllt
- am Ende kann im Idealfall die Lösung direkt aus der Tabelle abgelesen werden
- im Gegensatz zur top-down Variante (Rekursion erweitert um Memoization) ist der bottom-up Ansatz meist nicht so leicht zu finden
- kann aber nochmals beachtliche Performanzsteigerung zur Folge haben

Ein Klassiker der Kombinatorik: Binomialkoeffizienten

- Anzahl der Möglichkeiten k (numerierte) Bierflaschen aus einem Kasten mit n solcher erfrischenden Getränke zu entnehmen

Die Lösung:

$$\binom{n}{k} = \frac{n!}{(n-k)!k!}$$

- gilt natürlich auch für vergleichbare, aber weniger aufregende Problemstellungen

- Wie macht es der gemeine Informatiker? Klar: **rekursiv!**

Rekursionsgleichung

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

- Ein Blick auf das PASCALSche Dreieck offenbart uns, warum das funktioniert:

				1						
				1		1				
			1		2		1			
		1		3		3		1		
	1		4		6		4		1	
1		5		10		10		5		1

In der $(n+1)$ -ten Zeile finden wir die Werte $\binom{n}{i}$ für $0 \leq i \leq n$.

Konstruktion einer bottom-up Lösung

- Um den Binomialkoeffizienten $\binom{n}{m}$ zu berechnen, können wir also einfach eine Dreiecksmatrix mit einem PASCALSchen Dreieck mit $n + 1$ Zeilen ausfüllen.
- Die „Ränder“ des Dreiecks, also die Einträge $[0] [0]$ bis $[0] [n]$ sowie $[n] [0]$ bis $[n] [n]$, werden mit Einsen initialisiert.
- Die „Innenfläche“ wird zeilenweise abgelaufen, zur Berechnung kann immer auf zwei bereits besetzte Einträge zurückgegriffen werden.
- An Position $[n] [m]$ steht schließlich der gesuchte Wert.

Optimierungsmöglichkeit

- keine teureren Funktionsaufrufe nötig, da Rekursion vermieden wurde
- aber: einige der berechneten Werte wären gar nicht notwendig gewesen
- für den Wert $x_{i,j}$ sind $x_{i-1,j-1}$ und $x_{i-1,j}$ relevant
- zur Berechnung von $\binom{n}{m} = x_{n,m}$ werden also keine $x_{i,j}$ mit $j > m$ gebraucht
- es reicht daher beim Ausfüllen der Tabelle in jeder Zeile bis zum m -ten Wert zu gehen

Schritte beim Entwurf eines DP-Algorithmus

Der Entwurf eines DP-Algorithmus erfolgt üblicherweise in vier Schritten:

- 1 Struktur einer optimalen Lösung analysieren
- 2 Rekursionsgleichung für eine optimale Lösung aufstellen
- 3 wenn möglich Umwandlung des top-down Ansatzes in eine bottom-up Variante und Berechnung einer optimalen Lösung
- 4 Rekonstruktion des Lösungsweges zum zuvor bestimmten Ergebnis

Ich packe meinen Rucksack, und ich nehme mit ...

- Rucksack mit der Kapazität c
- N verschiedene Typen von Gegenständen (jeweils in unbegrenzter Anzahl)
- jeder Typ n hat einen spezifischen Wert v_n sowie einen Platzbedarf s_n
- wir wollen unseren Rucksack füllen
 - ohne die Kapazität c zu überschreiten
 - dabei den Gesamtwert des Inhalts maximieren
- typisches **Optimierungsproblem**

Beispiel: Einkauf im Supermarkt

n	Artikel	Platz s_n	Wert v_n
0	Bratwurst	3	4
1	Bier	4	5
2	Cola	7	10
3	Chips	8	11
4	Melone	9	13

Was packen wir in unseren Rucksack der Größe 17 um später maximalen Genuß zu erhalten?

- ausprobieren undenkbar
- aber: wie dann?

Entwurf einer Lösung für das Rucksackproblem

1 Wie sieht eine optimale Lösung aus?

- Liste von k Gegenstandstypen n :

$$L_c = (n_1, n_2, \dots, n_k)$$

(die einzelnen Einträge n_i müssen nicht zwingenderweise paarweise verschieden sein)

- maximaler Wert des so gepackten Rucksacks ist die Summe der Werte all seiner Gegenstände:

$$w_c = \sum_{i=1}^k v_i$$

Entwurf einer Lösung für das Rucksackproblem

② Rekursionsgleichung aufstellen

- wir gehen davon aus, dass L_c für die Kapazität c eine optimale Lösung ist
- wenn wir nun ein beliebiges Element, sagen wir das letzte, aus der Liste entfernen, so ist die Restliste eine optimale Lösung für die Kapazität $c - s_{n_k}$
- beim Weg zurück von der Teillösung ist eine **Auswahl** zu treffen: füge weiteres Element hinzu, so dass
 - die Kapazität nach wie vor unter der Gesamtkapazität liegt
 - der erhöhte Gesamtwert maximal ist

Entwurf einer Lösung für das Rucksackproblem

Ein funktionierender, rekursiver Ansatz:

```
int sack (int cap) {
    int i, space, max, t;
    for (i = 0, max = 0; i < N; i++)
        if ((space = cap - items[i].size) >= 0)
            if ((t = sack(space) + items[i].value) > max)
                max = t;
    return max;
}
```

- aber Vorsicht: **Finger weg davon!**
- extrem ineffizient schon bei sehr kleinen Werten!

Entwurf einer Lösung für das Rucksackproblem

Laufzeiten für die rekursive Implementierung (verwendete Daten waren die fünf Artikeltypen aus dem Supermarkt-Beispiel):

c	Zeit
100	0.9 sec
110	4.3 sec
120	23 sec
125	51 sec
130	1 min 57 sec
135	4 min 28 sec
140	10 min 10 sec
150	52 min 48 sec

- exponentielle Laufzeit

Entwurf einer Lösung für das Rucksackproblem

- ③ **Umwandeln in top-down DP**
 - das muss natürlich besser gehen!
 - wir bauen **Memoization** ein

Entwurf einer Lösung für das Rucksackproblem

```
int sack (int cap) {  
    int i, space, max, t;  
  
    for (i = 0, max = 0; i < N; i++)  
        if ((space = cap - items[i].size) >= 0)  
            if ((t = sack(space) + items[i].val) > max) {  
                max = t;  
            }  
  
    return max;  
}
```

Entwurf einer Lösung für das Rucksackproblem

```
int sack (int cap) {  
    int i, space, max, t;  
    if (maxKnown[cap] != UNSET) return maxKnown[cap];  
    for (i = 0, max = 0; i < N; i++)  
        if ((space = cap - items[i].size) >= 0)  
            if ((t = sack(space) + items[i].val) > max) {  
                max = t;  
            }  
  
    return max;  
}
```

Entwurf einer Lösung für das Rucksackproblem

```
int sack (int cap) {  
    int i, space, max, t;  
    if (maxKnown[cap] != UNSET) return maxKnown[cap];  
    for (i = 0, max = 0; i < N; i++)  
        if ((space = cap - items[i].size) >= 0)  
            if ((t = sack(space) + items[i].val) > max) {  
                max = t;  
            }  
    maxKnown[cap] = max;  
    return max;  
}
```

Entwurf einer Lösung für das Rucksackproblem

Wir werfen erneut einen Blick auf die Laufzeiten (Daten wie gehabt):

c	Zeit
1.000	5 ms
50.000	20 ms
100.000	25 ms
500.000	80 ms
1.000.000	145 ms

- immense **Performanzsteigerung!**
- **lineare Laufzeit**
- notwendige Änderung umfasste gerade mal zwei Zeilen Code

Entwurf einer Lösung für das Rucksackproblem

- ④ **Rekonstruktion des Lösungsweges**
 - wenn uns nur der Wert unseres perfekt gefüllten Rucksacks interessiert, sind wir nun fertig
 - falls auch die Bestückung von Interesse ist, so muss man sich die gewählten Gegenstände merken

Entwurf einer Lösung für das Rucksackproblem

- bei jedem Schritt kommt ein neuer Gegenstand hinzu, der die verbleibende Kapazität verkleinert
- dieser einmal gewählte Gegenstand bleibt für diesen Schritt fest und wird im weiteren Verlauf nicht mehr geändert
- es gibt also zu jedem Kapazitätswert, auf den wir auf unserem Weg stoßen, genau einen eindeutigen Gegenstand
- wir merken uns daher die Zuordnung Kapazität \rightarrow Index des gewählten Gegenstandes in einem Array `itemKnown`

Entwurf einer Lösung für das Rucksackproblem

- bei der Rekonstruktion beginnen wir bei der Gesamtkapazität c
- der Index unseres ersten Gegenstands befindet sich daher in `itemKnown[c]`
- wurde dieser Gegenstand in den Rucksack aufgenommen so verbleibt noch die Kapazität $c - \text{itemKnown}[c].\text{size}$
- der nächste Index findet sich also bei `itemKnown[c - itemKnown[c].size]` usw.

Entwurf einer Lösung für das Rucksackproblem

```
int sack (int cap) {  
    int i, space, max,          t;  
    if (maxKnown[cap] != UNSET) return maxKnown[cap];  
    for (i = 0, max = 0; i < N; i++)  
        if ((space = cap - items[i].size) >= 0)  
            if ((t = sack(space) + items[i].val) > max) {  
                max = t;  
            }  
    maxKnown[cap] = max;  
    return max;  
}
```

Entwurf einer Lösung für das Rucksackproblem

```
int sack (int cap) {  
    int i, space, max, max_i, t;  
    if (maxKnown[cap] != UNSET) return maxKnown[cap];  
    for (i = 0, max = 0; i < N; i++)  
        if ((space = cap - items[i].size) >= 0)  
            if ((t = sack(space) + items[i].val) > max) {  
                max = t; max_i = i;  
            }  
    maxKnown[cap] = max;  
    return max;  
}
```

Entwurf einer Lösung für das Rucksackproblem

```
int sack (int cap) {  
    int i, space, max, max_i, t;  
    if (maxKnown[cap] != UNSET) return maxKnown[cap];  
    for (i = 0, max = 0; i < N; i++)  
        if ((space = cap - items[i].size) >= 0)  
            if ((t = sack(space) + items[i].val) > max) {  
                max = t; max_i = i;  
            }  
    maxKnown[cap] = max; itemKnown[cap] = items[max_i];  
    return max;  
}
```

Wann bietet es sich an, DP einzusetzen?

- Suche nach einem Optimum
- jeder Schritt zum Ziel benötigt eine Auswahl
- optimales Ergebnis lässt sich in Teilergebnisse aufteilen, die ihrerseits optimal sind
- es treten bei der Zerstückelung häufig Überlappungen auf

Wann ist DP kein geeigneter Ansatz?

- wenn die Anzahl der möglichen Funktionswerte, die benötigt werden, zu groß wird
- Speicherplatz für Zwischenspeicherung (top-down) bzw. Vorausberechnung (bottom-up) dann nicht ausreichend
- insbesondere dann, wenn reelle Zahlen als Funktionsparameter auftreten
- ... und natürlich Finger weg von DP, wenn ein effizienterer Lösungsweg (Formel o.ä.) bekannt ist! :)

Zum Schluss ein kleiner Blick über den Tellerrand

DP wird *nicht nur* für das Lösen von ACM-Aufgaben benutzt.
Einsatzgebiete „in freier Wildbahn“ sind beispielsweise:





- **Bioinformatik**

- Sequenzierung von Genen und Proteinen
- sehr ähnliche Problemstellung wie bei Stringvergleichen

- **Linguistik**

- CYK-Algorithmus

Literaturempfehlungen

-  T. Cormen et al.
Introduction To Algorithms
MIT Press, 1992
-  R. Sedgewick
Algorithmen
Addison-Wesley, 1992
-  R. Sedgewick
Algorithmen in C++
Pearson Studium, 2002
-  S. Skiena, M. Revilla
Programming Challenges
Springer, 2003

That's all, folks!