Informatik II - SS 2018

(Algorithmen & Datenstrukturen)

Vorlesung 17b (20.6.2018)

Dynamische Programmierung I

Fabian Kuhn

Algorithmen und Komplexität

Dynamische Programmierung ($\underline{D}$P)

- Wichtige Algorithmenentwurf-Technik!
- Einfache, aber oft sehr effektive Idee

- Viele Probleme, welche naiv exponentielle Zeit benötigen, können mit dynamischer Programmierung in polynomieller Zeit gelöst werden.
  - Das gilt insbesondere für Optimierungsprobleme (min / max)

DP ≈ vorsichtige / optimierte Brute-Force-Lösung

DP ≈ Rekursion + Wiederverwendung

DP: Geschichte

- Woher kommt der Name?
- DP wurde durch Richard E. Bellman in den 1940er/1950er Jahren entwickelt. In seiner Autobiographie steht:

  "I spent the Fall quarter (of 1950) at RAND. My first task was to find a name for multistage decision processes. … The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word research. … His face would suffuse, he would turn red, and he would get violent if people used the term research in his presence. You can imagine how he felt, then, about the term mathematical. … Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose? In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word "programming". I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying. … It also has a very interesting property as an adjective, and that it's impossible to use the word dynamic in a pejorative sense. … Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. …"

Fibonacci Zahlen

Definition der Fibonacci Zahlen $\boldsymbol{F_0, F_1, F_2, \ldots}$:

$$F_0 = 0, \qquad F_1 = F_2 = 1$$
$$F_n = F_{n-1} + F_{n-2}$$

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

Ziel: Berechne F_n

```
fib(n)
    falls $n \leq 1$
        return n
    sonst
        return fib(n-1) + fib(n-2)
```

Naiver Algorithmus

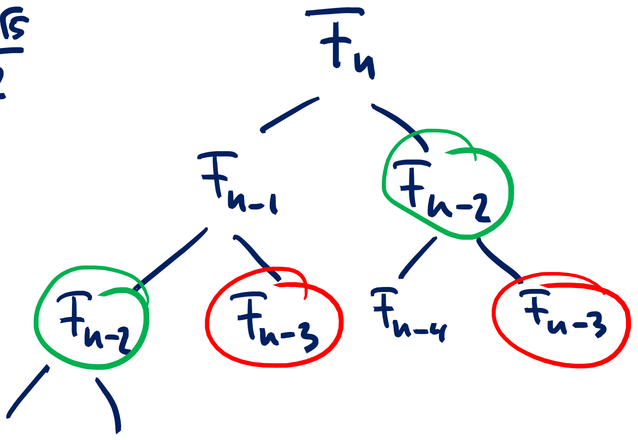
Rekursiver Algorithmus (basierend auf der rek. Definition):

```
fib(n):
    if n <= 1:
        f = n
    else:
        f = fib(n-1) + fib(n-2)
    return f
```

$\phi = \frac{1+\sqrt{5}}{2}$

$T(n) = T(n-1) + T(n-2) + c \geq F_n \approx \phi^n$

$\geq 2T(n-2) + c \geq 2^{n/2}$



Algorithmus mit Memoization

Memoization: Man merkt sich schon berechnete Werte (auf einem Notizzettel = memo)

```
memo = {} ←
fib(n):
    if n in memo: return memo[n]
    if n <= 1:
        f = n
    else:
        f = fib(n-1) + fib(n-2)
    memo[n] = f
    return f
```

fib(k) führt Rek. nur einmal aus (pro k)

insgesamt: nur n+1 Aufrufe, welche nicht den Wert aus memo nehmen

Laufzeit: O(1) Zeit für jeden Aufruf (Rek. ignoriert)

↳ Laufzeit: $O(n)$

DP: Etwas Genauer …

DP $\approx$ Rekursion + Memoization

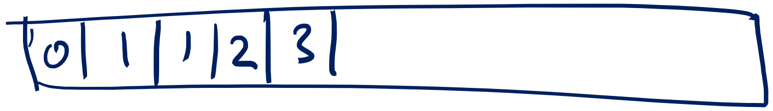
Memoize: *Speichere* Lösungen zu *Teilproblemen*, verwende die gespeicherten Lösungen, falls das gleiche Teilproblem wieder auftaucht.

- Bei den Fibonacci-Zahlen sind die Teilprobleme $\underline{F_1}, \underline{F_2}, \underline{F_3}, \ldots$

$$\text{Laufzeit} = \underbrace{\text{\#Teilprobleme}} \cdot \underbrace{\text{Zeit pro Teilproblem}}_{\uparrow \text{ Rek. ignoriert}}$$

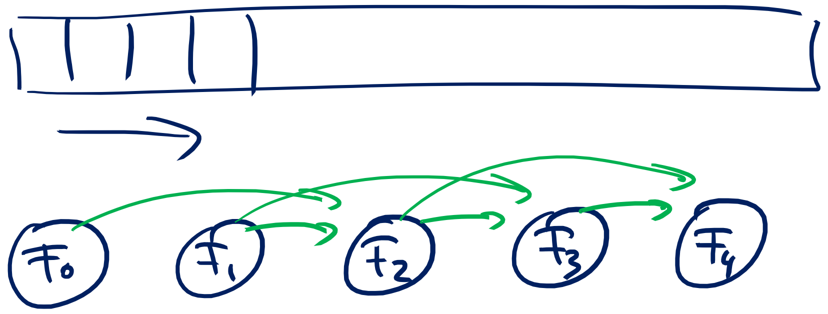
Fibonacci: Bottom-Up

```
fib(n):
    fn = {}   0,
    for k in [1, 2, …, n]:
        if k <= 1:
            f = k
        else:
            f = fn[k-1] + fn[k-2]
        fn[k] = f
    return fn[n]
```



Fibonacci: Bottom-Up

```
fib(n):
    fn = [None] * (n + 1)
    for k in [0, 1, 2, …, n]:
        if k <= 1:
            f = k
        else:
            f = fn[k-1] + fn[k-2]
        fn[k] = f
    return fn[n]
```


F_0
F_1
F_2
F_3
F_4