Dynamische Programmierung

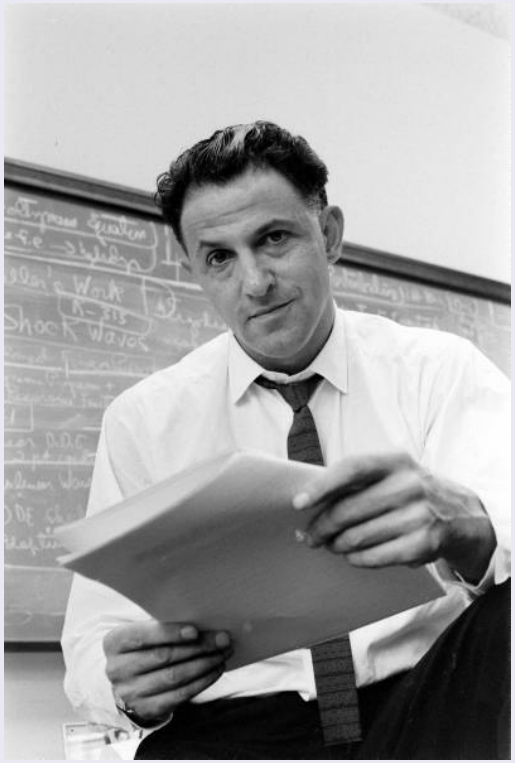
Timon Gehr

Entstehung der Dynamischen Programmierung

Entstehung

- Erstmal *Bellman* 1940-50 als Forscher bei RAND (**R**esearch **AN**d **D**evelopment).
- *Dynamisch* im Sinne von mehrstufig.
- *Programmierung* im Sinne von Planung (z.B. von militärischen Abläufen).

Richard Bellman



Subset-Sum

Problem: Gegeben ist eine Liste $a_1, a_2 \ldots, a_n$ von nicht-negativen ganzen Zahlen, sowie eine ganze Zahl S.
Bestimme, ob es möglich ist, einige der Zahlen zu wählen, so dass ihre Summe S ist.

Subset-Sum

Problem: Gegeben ist eine Liste $a_1, a_2 \ldots, a_n$ von nicht-negativen ganzen Zahlen, sowie eine ganze Zahl S.
Bestimme, ob es möglich ist, einige der Zahlen zu wählen, so dass ihre Summe S ist.

Beispiele:
Eingabe: $a = (5, 10, 20, 50, 100, 200, 500), S = 390$.
Ausgabe:

Subset-Sum

Problem: Gegeben ist eine Liste $a_1, a_2 \ldots, a_n$ von nicht-negativen ganzen Zahlen, sowie eine ganze Zahl S.
Bestimme, ob es möglich ist, einige der Zahlen zu wählen, so dass ihre Summe S ist.

Beispiele:
Eingabe: $a = (5, 10, 20, 50, 100, 200, 500), S = 390$.
Ausgabe: Nein.

Subset-Sum

Problem: Gegeben ist eine Liste $a_1, a_2 \ldots, a_n$ von nicht-negativen ganzen Zahlen, sowie eine ganze Zahl S.
Bestimme, ob es möglich ist, einige der Zahlen zu wählen, so dass ihre Summe S ist.

Beispiele:

Eingabe: $a = (5, 10, 20, 50, 100, 200, 500), S = 390$.
Ausgabe: Nein.

Eingabe: $a = (5, 10, 20, 50, 100, 200, 500), S = 875$.
Ausgabe:

Subset-Sum

Problem: Gegeben ist eine Liste $a_1, a_2 \ldots, a_n$ von nicht-negativen ganzen Zahlen, sowie eine ganze Zahl S.
Bestimme, ob es möglich ist, einige der Zahlen zu wählen, so dass ihre Summe S ist.

Beispiele:

Eingabe: $a = (5, 10, 20, 50, 100, 200, 500), S = 390$.
Ausgabe: Nein.

Eingabe: $a = (5, 10, 20, 50, 100, 200, 500), S = 875$.
Ausgabe: Ja.

Brute-force

Idee: Alle Möglichkeiten ausprobieren.
Sei S_i die Menge aller Möglichkeiten, einige Zahlen aus den ersten i zu wählen.

Beispiel:
$a = (5, 10, 20)$.
$S_0 = \{()\}$,
$S_1 = \{(), (5)\}$,
$S_2 = \{(), (5), (10), (5, 10)\}$,
$S_3 = \{(), (5), (10), (5, 10), (20), (5, 20), (10, 20), (5, 10, 20)\}$.

Rekursive Struktur: 5 10 20
Entweder wird das letzte Element gewählt, oder nicht.

Brute-force: Implementierung

```
a = map(int, input.split())
n = len(a)
s = int(input())

def subsets(n):
    if n==0: return [[]];
    last = a[n-1]
    withoutLast = subsets(n-1)
    withLast = [X + [last] for X in withoutLast]
    return withoutLast + withLast

if any([sum(X) == s for X in subsets(n)]):
    print('Ja.')
else: print('Nein.')
```

Laufzeitanalyse

Laufzeit von subsets(n): Mindestens 2^n Operationen.

Für $n = 100$:

Laufzeitanalyse

Laufzeit von subsets(n): Mindestens 2^n Operationen.

Für $n = 100$:

$2^{100} \geq 1000000000000000000000000000000$ Operationen.

≥ 1000000000000000000 Sekunden.

≥ 1000000000000000 Stunden.

≥ 100000000000000 Tage.

≥ 100000000000 Jahre.

Wie machen wir das schneller?

```
a = map(int, input.split())
n = len(a)
s = int(input())

def subsets(n):
    if n==0: return [[]];
    last = a[n-1]
    withoutLast = subsets(n-1)
    withLast = [X + [last] for X in withoutLast]
    return withoutLast + withLast

if any([sum(X) == s for X in subsets(n)]):
    print('Ja.')
else: print('Nein.')
```

Wie machen wir das schneller?

Kleinere Teilprobleme desselben Typs

```
a = map(int, input().split())
n = len(a)
s = int(input())

def check(n, s):
    if s==0: return True
    if s<0 or n==0: return False
    last = a[n-1]
    withoutLast = check(n-1, s)
    withLast = check(n-1, s-last)
    return withoutLast or withLast

if check(n,s): print('Ja.')
else: print('Nein.')
```

Weniger Overhead, aber nach wie vor:

Laufzeitanalyse

Laufzeit von check(n, s): Im schlimmsten Fall mindestens 2^n Operationen.

Für $n = 100$:

Weniger Overhead, aber nach wie vor:

Laufzeitanalyse

Laufzeit von check(n, s): Im schlimmsten Fall mindestens 2^n Operationen.

Für $n = 100$:

$2^{100} \geq 1000000000000000000000000000000$ Operationen.

≥ 1000000000000000000 Sekunden.

≥ 1000000000000000 Stunden.

≥ 100000000000000 Tage.

≥ 100000000000 Jahre.

Kleinere Teilprobleme desselben Typs

```
a = map(int, input().split())
n = len(a)
s = int(input())

def check(n, s):
    if s==0: return True
    if s<0 or n==0: return False
    last = a[n-1]
    withoutLast = check(n-1, s)
    withLast = check(n-1, s-last)
    return withoutLast or withLast

if check(n,s): print('Ja.')
else: print('Nein.')
```

Wie machen wir das schneller?

Idee: Dasselbe nicht mehr als einmal Ausrechnen

```
memo = [[(s+1)*["?"]] for _ in range(n+1)]
def check(n, s):
    if s==0: return True
    if s<0 or n==0: return False
    if memo[n][s]!="?": return memo[(n, s)]
    last = a[n-1]
    withoutLast = check(n-1, s)
    withLast = check(n-1, s-last)
    result = withoutLast or withLast
    memo[n][s] = result
    return result
```

Idee: Dasselbe nicht mehr als einmal Ausrechnen

```
memo = [[(s+1)*["?"]] for _ in range(n+1)]
def check(n, s):
    if s==0: return True
    if s<0 or n==0: return False
    if memo[n][s]!="?": return memo[(n, s)]
    last = a[n-1]
    withoutLast = check(n-1, s)
    withLast = check(n-1, s-last)
    result = withoutLast or withLast
    memo[n][s] = result
    return result
```

Laufzeitanalyse

Maximal $n \cdot s$ verschiedene Eingaben zu “check”.

Für jede Eingabe konstanter Aufwand: Laufzeit ist in $O(n \cdot s)$!

Für $n = 100, s = 100'000$: $\approx 10'000'000$ Operationen.

Rekonstruieren der Lösung

```
def reconstruct(n, s):
    if s==0: return []
    last = a[n-1]
    if check(n-1, s):
        return reconstruct(n-1, s)
    l = reconstruct(n-1, s-last)
    l.append(n);
    return l

if check(n,s):
    print('YES')
    print(' '.join(map(str, reconstruct(n, s))))
    else: print('NO')
```

Dynamische Programmierung

Voraussetzungen an das Problem

- Lösung kann effizient durch Lösungen von Teilproblemen des gleichen Typs ausgedrückt werden.
- Die Anzahl Teilprobleme ist nicht zu gross.

Dynamische Programmierung

Voraussetzungen an das Problem

- Lösung kann effizient durch Lösungen von Teilproblemen des gleichen Typs ausgedrückt werden.
- Die Anzahl Teilprobleme ist nicht zu gross.

Die 4 DP-Schritte [Cormen et al.]

1. Die Struktur einer optimalen Lösung beschreiben.
2. Rekursive Formulierung einer optimalen Lösung.
3. Berechnen des Wertes einer optimalen Lösung.
4. Rekonstruieren der optimalen Lösung.

Jetzt: Praktische Übungen

W-lan: public

`enter.ethz.ch`, login

`https://pages.soi.ch/stiu17/`