DP Subsetsum

SOI Workshop 2017, Zürich

November 5, 2017

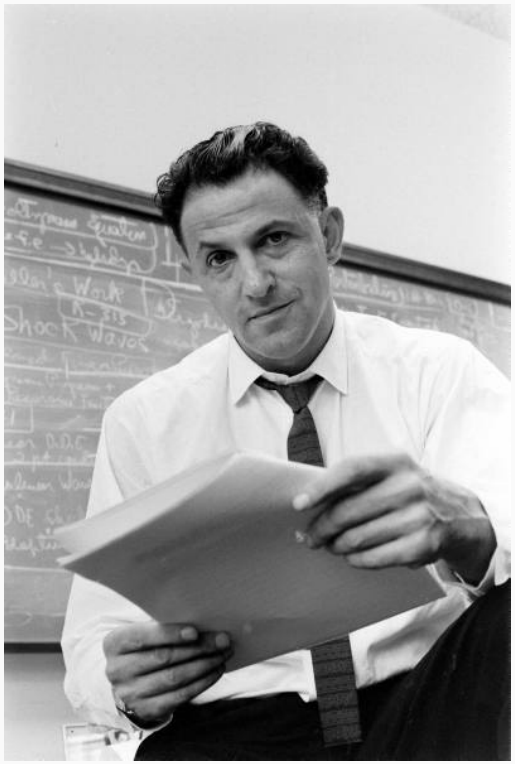
Einführung

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

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 = ...
def subsets(n):
    if n == 0: return [[]]
    last = a[n-1]
    skip = subsets(a[:-1])
    take = [subset + [last] for subset in skip]
    return skip + take

if any(sum(sub) == s for sub 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 = ...
def subsets(n):
    if n == 0: return [[]]
    last = a[n-1]
    skip = subsets(a[:-1])
    take = [subset + [last] for subset in skip]
    return skip + take

if any(sum(sub) == s for sub in subsets(n)):
    print("Ja.")
else:
    print("Nein.")
```

Wie machen wir das schneller?

Kleinere Teilprobleme desselben Typs

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

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

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

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

Wie machen wir das schneller?

Memoization

Dasselbe nicht mehr als einmal Ausrechnen

```
memo = [[[None]*(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] is not None: return memo[n][s]
    last = a[n-1]
    skip = check(n-1, s)
    take = check(n-1, s - last)
    result = skip or take
    memo[n][s] = result
    return result
```

Dasselbe nicht mehr als einmal Ausrechnen

```
memo = [[[None]*(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] is not None: return memo[n][s]
    last = a[n-1]
    skip = check(n-1, s)
    take = check(n-1, s - last)
    result = skip or take
    memo[n][s] = result
    return result
```

Laufzeitanalyse

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

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

Bottom-Up: Tabelle ausfüllen

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

DP=[[False]*(s+1) for _ in range(n+1)]
DP[0][0] = True
for i, x in enumerate(a, 1):
    for j in range(s+1):
        skip = DP[i-1][j]
        take = False if x > j else DP[i-1][j-x]
        DP[i][j] = skip or take
print("Ja." if DP[n][s] else "Nein.")
```