21. Dynamic Programming II

Subset sum problem, knapsack problem, greedy algorithm vs dynamic programming [Ottman/Widmayer, Kap. 7.2, 7.3, 5.7, Cormen et al, Kap. 15,35.5]

Task

Partition the set of the “item” above into two set such that both sets have the same value.

A solution:

Subset Sum Problem

Consider $n \in \mathbb{N}$ numbers $a_1, \ldots, a_n \in \mathbb{N}$.
Goal: decide if a selection $I \subseteq \{1, \ldots, n\}$ exists such that

$$\sum_{i \in I} a_i = \sum_{i \in \{1,\ldots,n\} \setminus I} a_i.$$

Naive Algorithm

Check for each bit vector $b = (b_1, \ldots, b_n) \in \{0, 1\}^n$, if

$$\sum_{i=1}^{n} b_i a_i \stackrel{?}{=} \sum_{i=1}^{n} (1 - b_i) a_i$$

Worst case: n steps for each of the 2^n bit vectors b. Number of steps: $\mathcal{O}(n \cdot 2^n)$.

Algorithm with Partition

- Partition the input into two equally sized parts $a_1, \dots, a_{n/2}$ and $a_{n/2+1}, \dots, a_n$.
- Iterate over all subsets of the two parts and compute partial sum $S_1^k, \dots, S_{2^{n/2}}^k$ $(k = 1, 2)$.
- Sort the partial sums: $S_1^k \le S_2^k \le \dots \le S_{2^{n/2}}^k$.
- Check if there are partial sums such that $S_i^1 + S_j^2 = \frac{1}{2}\sum_{i=1}^n a_i =: h$
  - Start with $i = 1, j = 2^{n/2}$.
  - If $S_i^1 + S_j^2 = h$ then finished
  - If $S_i^1 + S_j^2 > h$ then $j \leftarrow j - 1$
  - If $S_i^1 + S_j^2 < h$ then $i \leftarrow i + 1$

Example

Set $\{1, 6, 2, 3, 4\}$ with value sum 16 has 32 subsets.

Partitioning into $\{1, 6\}$, $\{2, 3, 4\}$ yields the following 12 subsets with value sums:

$\{1,6\}$				$\{2,3,4\}$							
$\{\}$	$\{1\}$	$\{6\}$	$\{1,6\}$	$\{\}$	$\{2\}$	$\{3\}$	$\{4\}$	$\{2,3\}$	$\{2,4\}$	$\{3,4\}$	$\{2,3,4\}$
0	$\boxed{1}$	6	7	0	2	3	4	5	6	$\boxed{7}$	9

$\Leftrightarrow$ One possible solution: $\{1, 3, 4\}$

Analysis

- Generate partial sums for each part: $\mathcal{O}(2^{n/2} \cdot n)$.
- Each sorting: $\mathcal{O}(2^{n/2} \log(2^{n/2})) = \mathcal{O}(n2^{n/2})$.
- Merge: $\mathcal{O}(2^{n/2})$

Overal running time

$$\mathcal{O}\left(n \cdot 2^{n/2}\right) = \mathcal{O}\left(n\left(\sqrt{2}\right)^n\right).$$

Substantial improvement over the naive method – but still exponential!

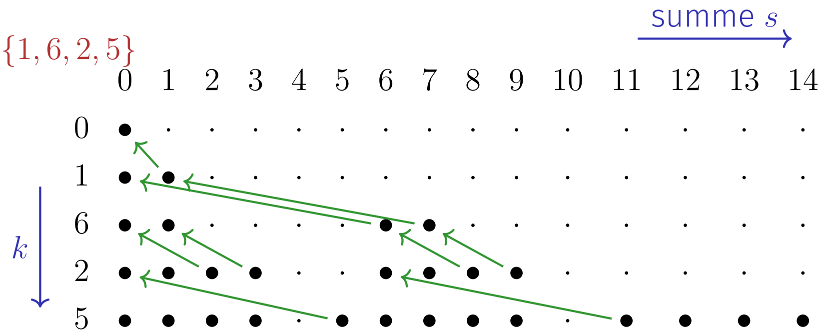
Dynamic programming

Task: let $z = \frac{1}{2}\sum_{i=1}^{n} a_i$. Find a selection $I \subset \{1,\dots,n\}$, such that $\sum_{i\in I} a_i = z$.
DP-table: $[0,\dots,n] \times [0,\dots,z]$-table T with boolean entries. $T[k,s]$ specifies if there is a selection $I_k \subset \{1,\dots,k\}$ such that $\sum_{i\in I_k} a_i = s$.
Initialization: $T[0,0] =$ true. $T[0,s] =$ false for $s > 1$.
Computation:

$$T[k,s] \leftarrow \begin{cases} T[k-1,s] & \text{if } s < a_k \\ T[k-1,s] \vee T[k-1,s-a_k] & \text{if } s \geq a_k \end{cases}$$

for increasing k and then within k increasing s.

Example



Determination of the solution: if $T[k, s] = T[k-1, s]$ then a_k unused and continue with $T[k-1, s]$, otherwise a_k used and continue with $T[k-1, s-a_k]$.

That is mysterious

The algorithm requires a number of $\mathcal{O}(n \cdot z)$ fundamental operations.

What is going on now? Does the algorithm suddenly have polynomial running time?

Explained

The algorithm does not necessarily provide a polynomial run time. z is an **number** and not a **quantity**!

Input length of the algorithm $\cong$ number bits to *reasonably* represent the data. With the number z this would be $\zeta = \log z$.

Consequently the algorithm requires $\mathcal{O}(n \cdot 2^\zeta)$ fundamental operations and has a run time exponential in ζ.

If, however, z is polynomial in n then the algorithm has polynomial run time in n. This is called **pseudo-polynomial**.

NP

It is known that the subset-sum algorithm belongs to the class of **NP**-complete problems (and is thus *NP-hard*).

P: Set of all problems that can be solved in polynomial time.

NP: Set of all problems that can be solved Nondeterministically in Polynomial time.

Implications:

- NP contains P.
- Problems can be **verified** in polynomial time.
- Under the not (yet?) proven assumption[36] that NP $\neq$ P, there is **no algorithm with polynomial run time** for the problem considered above.

[36]The most important unsolved question of theoretical computer science.

The knapsack problem

We pack our suitcase with ...

- toothbrush
- dumbell set
- coffee machine
- uh oh – too heavy.

- Toothbrush
- Air balloon
- Pocket knife
- identity card
- dumbell set
- Uh oh – too heavy.

- toothbrush
- coffe machine
- pocket knife
- identity card
- Uh oh – too heavy.

Aim to take as much as possible with us. But some things are more valuable than others!

Knapsack problem

Given:

- set of $n \in \mathbb{N}$ items $\{1, \ldots, n\}$.
- Each item i has value $v_i \in \mathbb{N}$ and weight $w_i \in \mathbb{N}$.
- Maximum weight $W \in \mathbb{N}$.
- Input is denoted as $E = (v_i, w_i)_{i=1,\ldots,n}$.

Wanted:

a selection $I \subseteq \{1, \ldots, n\}$ that maximises $\sum_{i \in I} v_i$ under $\sum_{i \in I} w_i \leq W$.

Greedy heuristics

Sort the items decreasingly by value per weight v_i/w_i: Permutation p with $v_{p_i}/w_{p_i} \geq v_{p_{i+1}}/w_{p_{i+1}}$

Add items in this order ($I \leftarrow I \cup \{p_i\}$), if the maximum weight is not exceeded.

That is fast: $\Theta(n \log n)$ for sorting and $\Theta(n)$ for the selection. But is it good?

Counterexample

$$
\begin{array}{lll}
v_1 = 1 & w_1 = 1 & v_1/w_1 = 1 \\
v_2 = W - 1 & w_2 = W & v_2/w_2 = \frac{W-1}{W}
\end{array}
$$

Greed algorithm chooses $\{v_1\}$ with value 1.
Best selection: $\{v_2\}$ with value $W - 1$ and weight W.

Greedy heuristics can be arbitrarily bad.

Dynamic Programming

Partition the maximum weight.
Three dimensional table $m[i, w, v]$ ("doable") of boolean values.
$m[i, w, v] =$ true if and only if

- A selection of the first i parts exists ($0 \leq i \leq n$)
- with overal weight w ($0 \leq w \leq W$) and
- a value of at least v ($0 \leq v \leq \sum_{i=1}^{n} v_i$) .

Computation of the DP table

Initially

- $m[i, w, 0] \leftarrow$ true für alle $i \geq 0$ und alle $w \geq 0$.
- $m[0, w, v] \leftarrow$ false für alle $w \geq 0$ und alle $v > 0$.

Computation

$$m[i, w, v] \leftarrow \begin{cases} m[i-1, w, v] \vee m[i-1, w-w_i, v-v_i] & \text{if } w \geq w_i \text{ und } v \geq v_i \\ m[i-1, w, v] & \text{otherwise.} \end{cases}$$

increasing in i and for each i increasing in w and for fixed i and w increasing by v.

Solution: largest v, such that $m[i, w, v] =$ true for some i and w.

Observation

The definition of the problem obviously implies that

- for $m[i, w, v] = \text{true}$ it holds:
  $m[i', w, v] = \text{true}\ \forall i' \geq i$,
  $m[i, w', v] = \text{true}\ \forall w' \geq w$,
  $m[i, w, v'] = \text{true}\ \forall v' \leq v$.
- fpr $m[i, w, v] = \text{false}$ it holds:
  $m[i', w, v] = \text{false}\ \forall i' \leq i$,
  $m[i, w', v] = \text{false}\ \forall w' \leq w$,
  $m[i, w, v'] = \text{false}\ \forall v' \geq v$.

This strongly suggests that we do not need a 3d table!

2d DP table

Table entry $t[i, w]$ contains, instead of boolean values, the largest v, that can be achieved[37] with

- items $1, \ldots, i$ $(0 \leq i \leq n)$
- at maximum weight w $(0 \leq w \leq W)$.

[37] We could have followed a similar idea in order to reduce the size of the sparse table.

Computation

Initially

- $t[0, w] \leftarrow 0$ for all $w \geq 0$.

We compute

$$t[i, w] \leftarrow \begin{cases} t[i-1, w] & \text{if } w < w_i \\ \max\{t[i-1, w], t[i-1, w-w_i] + v_i\} & \text{otherwise.} \end{cases}$$

increasing by i and for fixed i increasing by w.

Solution is located in $t[n, w]$

Example

$E = \{(2,3),(4,5),(1,1)\}$

$\xrightarrow{w}$

$i \downarrow$		0	1	2	3	4	5	6	7
	$\emptyset$	0	0	0	0	0	0	0	0
	$(2,3)$	0	0	3	3	3	3	3	3
	$(4,5)$	0	0	3	3	5	5	8	8
	$(1,1)$	0	1	3	4	5	6	8	9

Reading out the solution: if $t[i,w] = t[i-1,w]$ then item i unused and continue with $t[i-1,w]$ otherwise used and continue with $t[i-1, s-w_i]$.

Analysis

The two algorithms for the knapsack problem provide a run time in $\Theta(n \cdot W \cdot \sum_{i=1}^{n} v_i)$ (3d-table) and $\Theta(n \cdot W)$ (2d-table) and are thus both pseudo-polynomial, but they deliver the best possible result.

The greedy algorithm is very fast butmight deliver an arbitrarily bad result.

Now we consider a solution between the two extremes.