20. 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]

Quiz Solution

- **$n \times n$ Table**
- Entry at row $i$ and column $j$: height of highest possible stack formed from maximally $i$ boxes and basement box $j$.

<table>
<thead>
<tr>
<th>$w \times d$</th>
<th>$[1 \times 2]$</th>
<th>$[1 \times 3]$</th>
<th>$[2 \times 3]$</th>
<th>$[3 \times 4]$</th>
<th>$[3 \times 5]$</th>
<th>$[4 \times 5]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$h$</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>9</td>
<td>8</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>9</td>
<td>8</td>
<td>12</td>
</tr>
</tbody>
</table>

Determination of the table: $\Theta(n^3)$, for each entry all entries in the row above must be considered. Computation of the optimal solution by traversing back, worst case $\Theta(n^2)$

Quiz Alternative Solution

- $1 \times n$ Table, topologically sorted according to half-order stackability
- Entry at index $j$: height of highest possible stack with basement box $j$.

<table>
<thead>
<tr>
<th>$w \times d$</th>
<th>$[1 \times 2]$</th>
<th>$[1 \times 3]$</th>
<th>$[2 \times 3]$</th>
<th>$[3 \times 4]$</th>
<th>$[3 \times 5]$</th>
<th>$[4 \times 5]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$h$</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>9</td>
<td>8</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>9</td>
<td>8</td>
<td>12</td>
</tr>
</tbody>
</table>

Topological sort in $\Theta(n^2)$. Traverse from left to right in $\Theta(n)$, overall $\Theta(n^2)$. Traversing back also $\Theta(n^2)$

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

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

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

Algorithm with Partition

- Partition the input into two equally sized parts $a_1, \ldots, a_{n/2}$ and $a_{n/2+1}, \ldots, a_n$.
- Iterate over all subsets of the two parts and compute partial sum $S^k_1, \ldots, S^k_{2^{n/2}} (k = 1, 2)$.
- Sort the partial sums: $S^k_1 \leq S^k_2 \leq \cdots \leq S^k_{2^{n/2}}$.
- Check if there are partial sums such that $S^1_i + S^2_j = \frac{1}{2} \sum_{i=1}^n a_i =: h$

  - Start with $i = 1, j = 2^{n/2}$.
  - If $S^1_i + S^2_j = h$ then finished
  - If $S^1_i + S^2_j > h$ then $j \leftarrow j - 1$
  - If $S^1_i + S^2_j < h$ 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:

\[
\begin{array}{ccc}
\{1, 6\} & \{1\} & \{6\} & \{1, 6\} & \{2\} & \{3\} & \{4\} & \{2, 3\} & \{2, 4\} & \{3, 4\} & \{2, 3, 4\} \\
0 & 1 & 6 & 7 & 0 & 2 & 3 & 4 & 5 & 6 & 7 & 9
\end{array}
\]

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

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

Overall running time

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

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 $I \subset \{1, \ldots, n\}$, such that $\sum_{i \in I} a_i = z$.

**DP-table:** $[0, \ldots, n] \times [0, \ldots, z]$-table $T$ with boolean entries. $T[k, s]$ specifies if there is a selection $I_k \subset \{1, \ldots, k\}$ such that $\sum_{i \in I_k} a_i = s$.

**Initialization:** $T[0, 0] = true$. $T[0, s] = false$ for $s > 1$.

**Computation:**

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

for increasing $k$ and then within $k$ increasing $s$.

Example

$\{1, 6, 2, 5\}$

Summe $s$

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 $O(n \cdot z)$ 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 \textit{number} and not a \textit{quantity}!

Input length of the algorithm \( \approx \) number bits to \textit{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 \( \zeta \).

If, however, \( z \) is polynomial in \( n \) then the algorithm has polynomial run time in \( n \). This is called \textit{pseudo-polynomial}.


NP

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

\textit{P}: Set of all problems that can be solved in polynomial time.

\textit{NP}: Set of all problems that can be solved \textit{Nondeterministically} in Polynomial time.

Implications:

\begin{itemize}
  \item NP contains P.
  \item Problems can be \textit{verified} in polynomial time.
  \item Under the not (yet?) proven assumption\footnote{The most important unsolved question of theoretical computer science.} that \( \text{NP} \neq \text{P} \), there is no \textit{algorithm with polynomial run time} for the problem considered above.
\end{itemize}


The knapsack problem

We pack our suitcase with ...

\begin{itemize}
  \item toothbrush
  \item dumbell set
  \item coffee machine
  \item uh oh – too heavy.
\end{itemize}

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

Knapsack problem

Given:

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

Wanted:

\begin{itemize}
  \item a selection \( I \subseteq \{1, \ldots, n\} \) that maximises \( \sum_{i \in I} v_i \) under \( \sum_{i \in I} w_i \leq W \).
\end{itemize}
**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**

$v_1 = 1 \quad w_1 = 1 \quad v_1/w_1 = 1$

$v_2 = W - 1 \quad w_2 = W \quad v_2/w_2 = \frac{W-1}{W}$

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

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

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:
  - \( m[i', w, v] = \text{true} \forall i' \geq i \),
  - \( m[i, w', v] = \text{true} \forall w' \geq w \),
  - \( m[i, w, v'] = \text{true} \forall v' \leq v \).

- for \( m[i, w, v] = \text{false} \) 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\(^{33}\) with

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

\(^{33}\) 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

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

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 but might deliver an arbitrarily bad result.
Now we consider a solution between the two extremes.