Chapter 3
Dynamic Programming

Algorithm Theory
WS 2014/15

Fabian Kuhn

Dynamic Programming

„Memoization“ for increasing the efficiency of a recursive solution:

- Only the *first time* a sub-problem is encountered, its solution is computed and then stored in a table. Each subsequent time that the subproblem is encountered, the value stored in the table is simply looked up and returned
  (without repeated computation!).

- *Computing the solution*: For each sub-problem, store how the value is obtained (according to which recursive rule).

Dynamic Programming

Dynamic programming / memoization can be applied if

- Optimal solution contains optimal solutions to sub-problems (recursive structure)

- Number of sub-problems that need to be considered is small

Knapsack

- n items $1, \dots, n$, each item has weight w_i and value v_i
- Knapsack (bag) of capacity W

- Goal: pack items into knapsack such that total weight is at most W and total value is maximized:

$$\max \sum_{i \in S} v_i$$

$$\text{s.t. } S \subseteq \{1, \dots, n\} \text{ and } \sum_{i \in S} w_i \leq W$$

- E.g.: jobs of length w_i and value v_i, server available for W time units, try to execute a set of jobs that maximizes the total value

Recursive Structure?

- Optimal solution: $\mathcal{O}$
- If $n \notin \mathcal{O}$: $\mathrm{OPT}(n) = \mathrm{OPT}(n-1)$
- What if $n \in \mathcal{O}$?
  - Taking n gives value v_n
  - But, n also occupies space w_n in the bag (knapsack)
  - There is space for $W - w_n$ total weight left!

$$\mathrm{OPT}(n) = w_n + \text{optimal solution with first } n-1 \text{ items}$$
$$\text{and knapsack of capacity } W - w_n$$

A More Complicated Recursion

$\mathbf{OPT}(\boldsymbol{k}, \boldsymbol{x})$: value of optimal solution with items $1, \dots, k$
and knapsack of capacity x

Recursion:

Dynamic Programming Algorithm

Set up table for all possible $\mathrm{OPT}(k, x)$-values

- Assume that all weights w_i are integers!

	$\mathbf{1}$	$\mathbf{2}$	$\mathbf{3}$			$\cdots$						$\boldsymbol{W}$
$\mathbf{1}$												
$\mathbf{2}$												
$\mathbf{3}$												
$\vdots$												
$\boldsymbol{n}$												

Row $\boldsymbol{i}$, column $\boldsymbol{j}$:

$\boldsymbol{OPT(i, j)}$

Example

- 8 items: (3,2), (2,4), (4,1), (5,6), (3,3), (4,3), (5, 4), (6,6)
  Knapsack capacity: 12

  weight value

- $OPT(k,x) = \max\{OPT(k-1,x), OPT(k-1, x-w_k) + v_k\}$

	1	2	3	4	5	6	7	8	9	10	11	12
1												
2												
3												
4												
5												
6												
7												
8												

Running Time of Knapsack Algorithm

- **Size of table:** $O(n \cdot W)$
- Time per table entry: $O(1)$ → **overall time: $\boldsymbol{O(nW)}$**
- Computing solution (set of items to pick):
  Follow $\leq n$ arrows → $O(n)$ time (after filling table)
- Note: Time depends on W → can be exponential in n…
- And it is problematic if weights are not integers.

String Matching Problems

Edit distance:

- For two given strings A and B, efficiently compute the **edit distance $D(A, B)$** (# edit operations to transform A into B) as well as a minimum sequence of edit operations that transform A into B.

- **Example:** mathematician → multiplication:

$$\texttt{m u} \underbrace{\;}_{\texttt{l}} \texttt{t i p l} \underbrace{\;}_{\texttt{i c}} \texttt{a t i o } \cancel{\texttt{i}}\ \cancel{\texttt{a}}\ \texttt{n}$$

String Matching Problems

Edit distance $\boldsymbol{D(A,B)}$ (between strings A and B):

```
m a – t h e m - - a t i c i a n
m u l t i p l i c a t i o - - n
```

Approximate string matching:

For a given text *T*, a pattern *P* and a distance *d*, find all

substrings P' of T with $D(P,P') \leq d$.

Sequence alignment:

Find optimal alignments of DNA / RNA / ... sequences.

```
G A G C A - C T T G G A T T C T C G G
- - - C A C G T G G - A - A C T - - -
```

Edit Distance

Given: Two strings $A = a_1 a_2 \dots a_m$ and $B = b_1 b_2 \dots b_n$

Goal: Determine the minimum number $D(A, B)$ of edit operations required to transform A into B

Edit operations:

a) **Replace** a character from string A by a character from B

b) **Delete** a character from string A

c) **Insert** a character from string B into A

```
m a – t h e m - - a t i c i a n
m u l t i p l i c a t i o - - n
```

Edit Distance – Cost Model

- Cost for **replacing** character a by b: $\boldsymbol{c(a,b) \geq 0}$

- Capture insert, delete by allowing $a = \varepsilon$ or $b = \varepsilon$:
  - Cost for **deleting** character a: $\boldsymbol{c(a,\varepsilon)}$
  - Cost for **inserting** character b: $\boldsymbol{c(\varepsilon,b)}$

- **Triangle inequality**:

$$c(a,c) \leq c(a,b) + c(b,c)$$

→ each character is changed at most once!

- **Unit cost model**: $c(a,b) = \begin{cases} 1, & \text{if } a \neq b \\ 0, & \text{if } a = b \end{cases}$

Recursive Structure

- Optimal "alignment" of strings (unit cost model)

  `bbcadfagikccm` and `abbagflrgikacc`:

```
- b b c a g f a – g i k - c c m
a b b – a d f l r g i k a c c –
```

- Consists of optimal "alignments" of sub-strings, e.g.:

```
-bbcagfa        –gik-ccm
abb-adfl   and  rgikacc-
```

- Edit distance between $A_{1,m} = a_1 \dots a_m$ and $B_{1,n} = b_1 \dots b_n$:

$$D(A,B) = \min_{k,\ell}\left\{D\left(A_{1,k}, B_{1,\ell}\right) + D\left(A_{k+1,m}, B_{\ell+1,n}\right)\right\}$$

Computation of the Edit Distance

Let $A_k := a_1 \dots a_k$, $B_\ell := b_1 \dots b_\ell$, and

$$D_{k,\ell} := D(A_k, B_\ell)$$