



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, ..., 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:

$$\max \sum_{i \in S} v_i$$
 s.t. $S \subseteq \{1, ..., n\}$ and $\sum_{i \in S} w_i \le W$

• 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 $n \notin \mathcal{O}$: OPT(n) = 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!

```
OPT(n) = w_n + optimal solution with first <math>n - 1 items and knapsack of capacity W - w_n
```

A More Complicated Recursion



OPT(k, x): value of optimal solution with items 1, ..., k and knapsack of capacity x

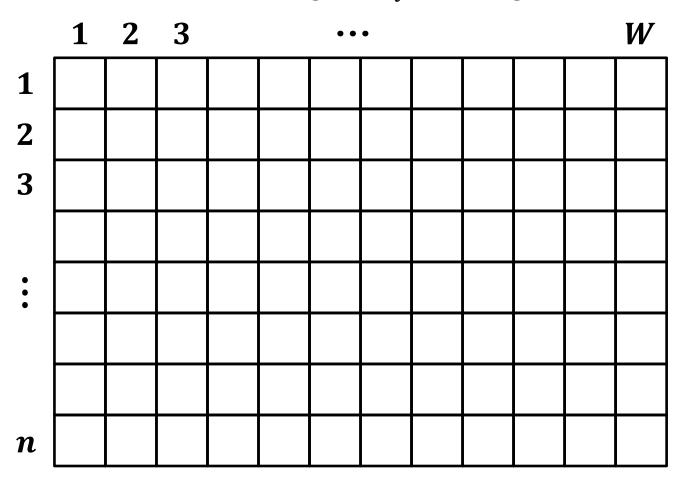
Recursion:

Dynamic Programming Algorithm



Set up table for all possible OPT(k, x)-values

• Assume that all weights w_i are integers!



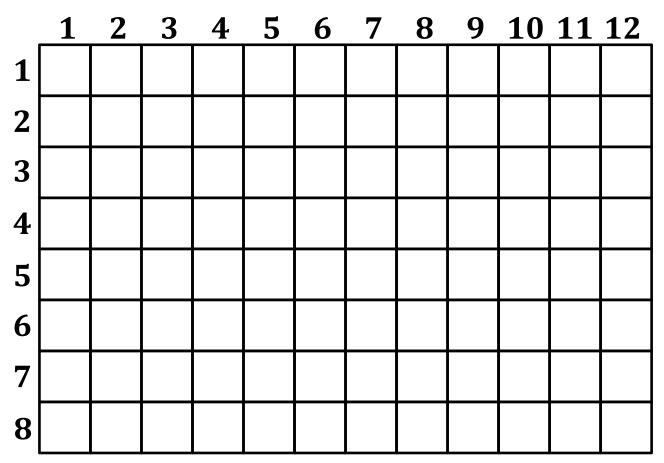
Row *i*, column *j*:

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



Running Time of Knapsack Algorithm



- Size of table: $O(n \cdot W)$
- Time per table entry: $O(1) \rightarrow$ overall time: O(nW)
- Computing solution (set of items to pick): Follow $\leq n$ arrows $\rightarrow O(n)$ time (after filling table)
- Note: Time depends on $W \rightarrow$ 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:

String Matching Problems



Edit distance D(A, B) (between strings A and B):

$$ma-them--atician$$

 $multiplicatio--n$

Approximate string matching:

For a given text T, a pattern P and a distance d, find all substrings P' of T with $D(P, P') \le d$.

Sequence alignment:

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

Edit Distance



Given: Two strings $A=a_1a_2\dots a_m$ and $B=b_1b_2\dots b_n$

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

```
ma-them--atician
multiplicatio--n
```

Edit Distance – Cost Model



- Cost for **replacing** character a by b: $c(a, b) \ge 0$
- Capture insert, delete by allowing $a = \varepsilon$ or $b = \varepsilon$:
 - Cost for **deleting** character $a: c(a, \varepsilon)$
 - Cost for **inserting** character b: $c(\varepsilon, b)$
- Triangle inequality:

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

→ 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:

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

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

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

Computation of the Edit Distance



Let
$$A_k\coloneqq a_1\dots a_k$$
 , $B_\ell\coloneqq b_1\dots b_\ell$, and
$$D_{k,\ell}\coloneqq D(A_k,B_\ell)$$



B