ALGORITHMS IN AI & DATA SCIENCE 1 (AKIDS 1)

Dynamic Programming: Some Problems

Prof. Dr. Goran Glavaš

4.12.2023

Content

- DP Recap
- Knapsack Problem
- Minimal Edit Distance

Dynamic Programming

- **Dynamic programming** solves **problems** with following properties:
  - **Divisible** into subproblems of the same type as the original problem
  - **Solution to the subproblem** is part of the solution to the whole problem

+

  - **Subproblems repeat a lot**: storing solutions of solved subproblems crucial

- Commonly applied to **(discrete) optimization problems**
  - Problems that have many possible solutions
  - Solutions have values or costs associated to them
  - We want to find the **optimal solution** – one with max value or min cost
    - There can be more than one optimal solution!

Example DP Problem: **Rod cutting**

Rod-cutting problem

A company buys **long steel rods** and **cuts** them into **shorter rods** which it then sells. For each rod of length *i* (in inches), we know the corresponding price p_i the company sells it for. Given a rod of length *n* inches and a table of prices p_i for $i = 1, 2, ..., n$ determine the **maximal revenue** r_n the company can have from that rod and find a cutting **s** that achieves that maximal revenue. Not that, in principle, if p_n is large enough the optimal revenue may be achieved without any cutting at all.

Length *i*	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Price p_i	1	5	8	9	10	17	17	20	24	26

Dynamic programming

- Developing a DP algorithm involves the following steps:

1. **Characterize the structure** of an optimal solution
2. **Recursively define** the value of an optimal solution
3. **Compute the value** of an optimal solution
    - **Top-down** manner (recursively)
    - **Bottom-up** manner (without recursion)
    - **Memorize the solutions to solved subproblems!!!**
4. **Construct** an optimal solution
    - Not enough just to know what the optimal (max or min) value is
    - Need to know which „sequence of steps” leads to that optimal solution
        - For example, in **rod-cutting –** where exactly to cut (sub-rods of which lengths to sell)

Dynamic programming: **solution memorization**

- Avoid the exponential complexity of the recursive solution **$O(2^n)$**
- As usual**,** the **solution** is to **trade some space for time**

- **Memorization**
  - Once the **solution for a subproblem** is computed the first time, **store it**
  - When you encounter **the same subproblem**, simply **retrieve its solution**
  - Obviously, this only helps in case of **repeating subproblems**

```
cut_rod(p, n)
   r = {}  # empty solutions hashtable
   return cut_rod_rec(p,n,r)

cut_rod_rec(p, n, r)
   if n in r # lookup key in hashtable
     return r[n]

   q = -inf
   if n == 0
     q = 0
   else
     for i in 1 to n
       t = p[i] + cut_rod_rec(p, n-i,r)
       if t > q
         q = t
   r[n] = q # store when computed 1st time
   return r
```

Dynamic programming: **bottom up**

- **Bottom-up DP** – the idea is that of **iterative induction**:
  - We know the solution of the **smallest** subproblem
  - For rod-cutting, this is when n = 0
  - If I know r_{k-i} – the solution for **all** $i < k$, then I know how to compute the solution for r_k – the solution for k
- **Q: Runtime** of bottom-up approach?
  - Clearly **O(n^2)**

```
cut_rod_iter(p, n)
  r = {}
  r[0] = 0
  r[1] = p[1]

  for i in 2 to n
    r[i] = -inf

    # we’re looking for the max
    for j in 1 to i
      q = p[j] + r[i-j]
      if q > r[i]
        r[i] = q

  return r[n]
```

Rod-cutting: complete example

Length *i*	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Price p_i	1	5	8	9	10	17	17	20	24	26

- $r_1 = p_1 = \mathbf{1}$
- $r_2 = \max(p_1 + r_1, p_2) = \max(1+1, 5) = \mathbf{5}, s_2 = 2$
- $r_3 = \max(p_1 + r_2, p_2 + r_1, p_3) = \max(1+5, 5+1, 8) = \mathbf{8}, s_3 = 3$
- $r_4 = \max(p_1 + r_3, p_2 + r_2, p_3 + r_1, p_4) = \max(1+8, 5+5, 8+1, 9) = \mathbf{10}, s_4 = 2$
- $r_5 = \max(1+10, 5+8, 8+5, 9+1, 10) = \mathbf{13}, s_5 = 2$
- $r_6 = \max(1+13, 5+10, 8+8, 9+5, 10+1, 17) = \mathbf{17}, s_6 = 6$
- $r_7 = \max(1+16, 5+5, 8+10, 9+8, 10+5, 17+1, 17) = \mathbf{18}, s_7 = 3$
- $r_8 = \max(1+18, 5+17, 8+13, 9+10, 10+8, 17+5, 18+1, 20) = \mathbf{22}, s_8 = 6$
- $r_9 = \max(1+22, 5+18, 8+17, 9+13, 10+10, 17+8, 17+5, 22+1, 24) = \mathbf{25}, s_9 = 3$
- $r_{10} = \max(1+25, 5+22, 8+18, 9+17, 10+13, 17+10, 17+8, 20+5, 24+1, 26) = \mathbf{27}, s_{10} = 6$

```
get_solution_rod_iter(p, n)
  r, s = cut_rod_iter(p, n)
  print(r[n]) # max price
  i = n
  while i > 0
    print(s[i]) # cuts / lengths
    i = i-s[i]
```

- **Reconstruction** of the solution: $s_{10} = 6 \rightarrow s_{4(=10-6)} = 2 \rightarrow s_{2(=4-2)} = 2 \rightarrow s_{0(=2-2)}$ **end**!
  - Optimal solution is to cut **only once**, after the **6th inch**. We sell two rods, 6 and 4 inch long.

Content

- DP Recap
- Knapsack Problem
- Minimal Edit Distance

The Knapsack Problem

(Binary) knapsack problem

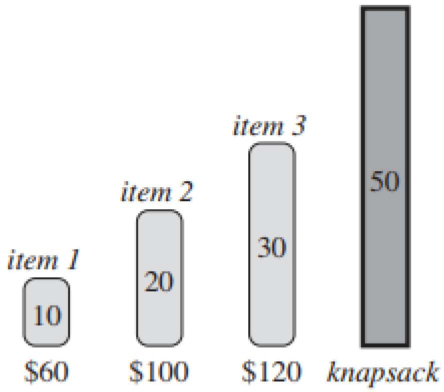
The **binary** (or **0-1**) **knapsack problem** is given as follows. A thief is robbing a store and finds n items in the store. The i-th item has a weight w_i and value v_i. The thief wants to steal the most valuable possible load, but he's limited with the maximal weight W his knapsack can carry. What is the **maximal value** the thief can steal (and which set of items are to be stolen)?

- Binary knapsack problem: each item is **either taken or not**
  - Cannot take a part/fraction of an item
- There exists also the fractional knapsack problem – thief can take any **fraction** (real number between 0 and 1) of an item
  - Easier problem, can be solved **greedily**

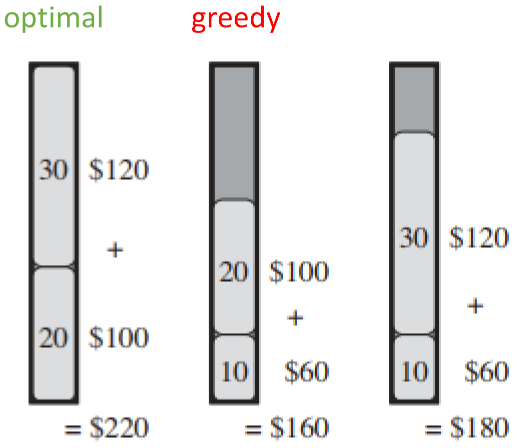
Knapsack problem – **greedy** approach?

- **Greedy strategy**: choice that (locally) most increases the value
- Binary vs. Fractional knapsack problem
  - **Binary**: greedy (per „kg” of weight) doesn’t work (not an optimal solution)
  - **Fractional**: greedy works (per „kg” of weight)

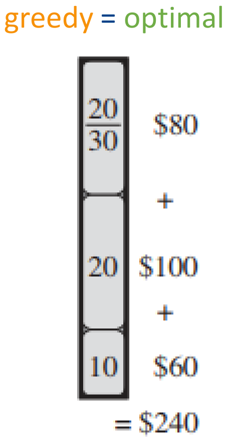
Item 1: 6$/kg
Item 2: 5$/kg
Item 3: 4$/kg



Knapsack problem



Solutions to binary problem



Greedy solution to fractional problem

Knapsack problem: **DP solution**

- Developing a DP algorithm involves the following steps:

1. **Characterize the structure** of an optimal solution
    - Quite analogous to structure of the solution for the rod-cutting problem
    - We start with an empty knapsack and n items we could put in it
    - n choices for the first item
    - Place an i-th item (v_i, w_i) → the remaining knapsack capacity is $W - w_i$
    - Let **I** = {i_1, ..., i_n} be the set of **all items**
    - Assume an **optimal solution** consists of K items $i_1, i_2, \ldots i_K$
        - The value of the optimal load: $v = v_{i1} + v_{i2} + \ldots + v_{iK}$
        - The weight of the optimal load: $w = w_{i1} + w_{i2} + \ldots + w_{iK} \leq W$

Knapsack problem: **DP solution**

- Developing a DP algorithm involves the following steps:

1. **Characterize the structure** of an optimal solution
    - Let **I** = $\{i_1, \ldots, i_n\}$ be the set of **all items**
    - Assume an **optimal solution** consists of K items $i_1, i_2, \ldots i_K$
        - The value of the optimal load: $v = v_{i1} + v_{i2} + \ldots + v_{iK}$
        - The weight of the optimal load: $w = w_{i1} + w_{i2} + \ldots + w_{iK} \leq W$

    - If we remove any item i_k, the remaining $k-1$ items represent an optimal solution for the **subproblem**:
        - **items**: **I** / $\{i_k\}$, **knapsack capacity**: $W - w_{ik}$

Knapsack problem: **DP solution**

- Developing a DP algorithm involves the following steps:

2. **Recursively define** the value of an optimal solution
   - If we remove any item i_k, the remaining $k-1$ items represent an optimal solution for the **subproblem**:
     - **items**: $\mathbf{I} / \{i_k\}$, **knapsack capacity**: $W - w_{ik}$
   - How many items can we remove from the solution?
     - Since we don't know how many items an optimal solution has, we need to consider every item in $\mathbf{I}$

$$\mathit{val}(\mathbf{I}, W) = \mathbf{max}_{\,ik \in \mathbf{I}}\ \mathit{val}(\mathbf{I} / \{i_k\}, W - w_k) + v_k$$

     - If we assume that i_k is part of the **optimal solution**, then the rest of the knapsack must be filled with the items that are an **optimal solution for the remaining capacity** $W - w_k$

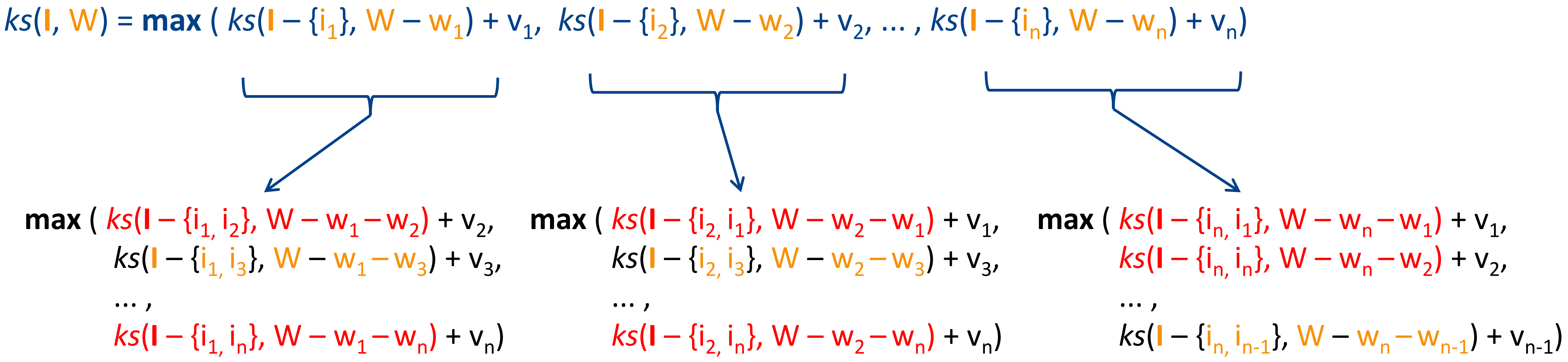
Knapsack problem: **DP solution**

- Recursive implementation (no memorization)

```
knapsack(W, I, v, w)
  val = 0
  if len(I) == 0 or W == 0 # no items or knapsack capacity left
    return 0
  for i in I
    if w[i] ≤ W
      q = knapsack(W - w[i], I / {i}, v, w) + v[i]
      if q > val
        val = q
  return val
```

Knapsack problem: **DP solution**

- **Q:** Where are the repeating **subproblems**?
- Set of all items: $\mathbf{I} = \{i_1, \ldots, i_n\}$



$$ks(\mathbf{I}, W) = \mathbf{max}\ (\ ks(\mathbf{I} - \{i_1\}, W - w_1) + v_1,\ \ ks(\mathbf{I} - \{i_2\}, W - w_2) + v_2,\ \ldots,\ ks(\mathbf{I} - \{i_n\}, W - w_n) + v_n)$$

$$\begin{aligned}\mathbf{max}\ (\ & ks(\mathbf{I} - \{i_1, i_2\}, W - w_1 - w_2) + v_2,\\ & ks(\mathbf{I} - \{i_1, i_3\}, W - w_1 - w_3) + v_3,\\ & \ldots,\\ & ks(\mathbf{I} - \{i_1, i_n\}, W - w_1 - w_n) + v_n)\end{aligned}$$

$$\begin{aligned}\mathbf{max}\ (\ & ks(\mathbf{I} - \{i_2, i_1\}, W - w_2 - w_1) + v_1,\\ & ks(\mathbf{I} - \{i_2, i_3\}, W - w_2 - w_3) + v_3,\\ & \ldots,\\ & ks(\mathbf{I} - \{i_2, i_n\}, W - w_2 - w_n) + v_n)\end{aligned}$$

$$\begin{aligned}\mathbf{max}\ (\ & ks(\mathbf{I} - \{i_n, i_1\}, W - w_n - w_1) + v_1,\\ & ks(\mathbf{I} - \{i_n, i_n\}, W - w_n - w_2) + v_2,\\ & \ldots,\\ & ks(\mathbf{I} - \{i_n, i_{n-1}\}, W - w_n - w_{n-1}) + v_{n-1})\end{aligned}$$

- Without memorization of subsolution problems
  - Exponential runtime complexity, **$O(2^n)$**

Dynamic programming

- Developing a DP algorithm involves the following steps:

1. **Characterize the structure** of an optimal solution
2. **Recursively define** the value of an optimal solution
3. **Compute the value** of an optimal solution
    - **Bottom-up** manner (without recursion)
    - Memorize the solutions to solved subproblems!!!
4. **Construct** an optimal solution
    - Not enough just to know what the optimal (max or min) value is
    - Need to know which „sequence of steps” leads to that optimal solution
        - For example, in **rod-cutting –** where exactly to cut (sub-rods of which lengths to sell)

Knapsack problem: iterative DP solution

- We could add memorization to the recursive computation
- **DP**: **iterative solution** more common (with memorization)

- We will iterate over **increasing subsets of items**
  - In each iteration, we will consider only up to first *k* items, for different capacities of the knapsack
  - `val[k, W']`: **maximal value** if considering only first *k* items, for capacity W'

  - By the time we compute `val[k, W']` we compare
    - `val[k-1, W']` and `# this means not including the k-th item`
    - $val[k-1, W' - w_k] + v_k$ `# this means adding k-th item`
      - 0 if $W' - w_k < 0$

Knapsack problem: iterative DP solution

- 4 items: i_1, i_2, i_3, i_4 , knapsack capacity $W = 8$
- Weights: $w_1 = 3$, $w_2 = 4$, $w_3 = 6$, $w_4 = 5$
- Values: $v_1 = 2$, $v_2 = 3$, $v_3 = 1$, $v_4 = 4$
- Initialization: set 0s for row i_0 and column $W' = 0$

Items (w, v)	**W' = 0**	**W' = 1**	**W' = 2**	**W' = 3**	**W' = 4**	**W' = 5**	**W' = 6**	**W' = 7**	**W' = 8**
i_0 = no item	0	0	0	0	0	0	0	0	0
i_1 (3, 2)	0								
i_2 (4, 3)	0								
i_3 (6, 1)	0								
i_4 (5, 4)	0								

Knapsack problem: iterative DP solution

Items (w, v)	W' = 0	W' = 1	W' = 2	W' = 3	W' = 4	W' = 5	W' = 6	W' = 7	W' = 8
i_0 = no item	0	0	0	0	0	0	0	0	0
i_1 (3, 2)	0	0	0	2	2	2	2	2	2
i_2 (4, 3)	0								
i_3 (6, 1)	0								
i_4 (5, 4)	0								

- $\mathit{val}[i = 1, W' = x] = \mathbf{max}(\mathit{val}[i = 0, W' = x],$
  $\mathit{val}[i = 0, W' - w_1] + v_k \text{ if } W' - w_1 \geq 0 \text{ else } 0)$

Knapsack problem: iterative DP solution

Items (w, v)	W' = 0	W' = 1	W' = 2	W' = 3	W' = 4	W' = 5	W' = 6	W' = 7	W' = 8
i_0 = no item	0	0	0	0	0	0	0	0	0
i_1 (3, 2)	0	0	0	2	2	2	2	2	2
i_2 (4, 3)	0	0	0	2	3	3	3	5	5
i_3 (6, 1)	0								
i_4 (5, 4)	0								

- $\mathit{val}[i = 2, W' = x] = \textbf{max}(\ \mathit{val}[i = 1, W' = x],$

  $\mathit{val}[i = 1, W' - w_2] + v_2 \text{ if } W' - w_2 \geq 0 \text{ else } 0\ )$

Knapsack problem: iterative DP solution

Items (w, v)	W’ = 0	W’ = 1	W’ = 2	W’ = 3	W’ = 4	W’ = 5	W’ = 6	W’ = 7	W’ = 8
i_0 = no item	0	0	0	0	0	0	0	0	0
i_1 (3, 2)	0	0	0	2	2	2	2	2	2
i_2 (4, 3)	0	0	0	2	3	3	3	5	5
i_3 (6, 1)	0	0	0	2	3	3	3	5	5
i_4 (5, 4)	0								

- *val*[i = 3, W’ = x] = **max**(*val*[i = 2, W’ = x],
  val[i = 2, W’ – w_3] + v_3 if W’ – w_3 ≥ 0 else 0)

Knapsack problem: iterative DP solution

Items (w, v)	**W’ = 0**	**W’ = 1**	**W’ = 2**	**W’ = 3**	**W’ = 4**	**W’ = 5**	**W’ = 6**	**W’ = 7**	**W’ = 8**
i_0 = no item	0	0	0	0	0	0	0	0	0
i_1 (3, 2)	0	0	0	2	2	2	2	2	2
i_2 (4, 3)	0	0	0	2	3	3	3	5	5
i_3 (6, 1)	0	0	0	2	3	3	3	5	5
i_4 (5, 4)	0	0	0	2	3	4	4	5	**6**

- *val*[i = 4, W’ = x] = **max**(*val*[i = 3, W’ = x],

  val[i = 3, W’ – w_4] + v_4 if W’ – $w_4 \geq 0$ else 0)

Knapsack problem: DP solution

- Iterative solution

```
knapsack_iter(I, W, w, v)
  val = array[|I|+1][W+1] # 2-D array, assume 0-indexing for both dimensions
  # initialization
  for i in 0 to len(I)
    val[i][0] = 0
  for w’ in 0 to W
    val[0][w’] = 0

  for i in 1 to len(I)
    for w’ in 1 to W
      v’ = 0
      if w’- w[i] ≥ 0
        v’ = val[i-1, w’- w[i]] + v[i]
      val[i, w’] = max(val[i-1, w’], v’)

  return val[|I|][W]
```

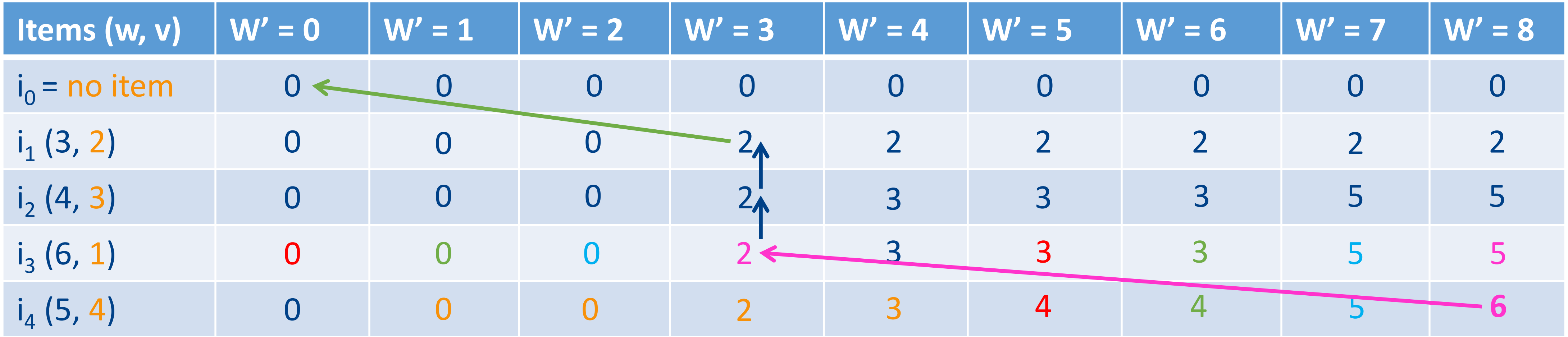
Q: Runtime?

Dynamic programming

- Developing a DP algorithm involves the following steps:

1. **Characterize the structure** of an optimal solution
2. **Recursively define** the value of an optimal solution
3. **Compute the value** of an optimal solution
    - **Bottom-up** manner (without recursion)
    - **Memorize the solutions to solved subproblems!!!**
4. **Construct** an optimal solution
    - Not enough just to know what the optimal (max or min) value is
    - Need to know which „sequence of steps” leads to that optimal solution
        - For example, in **rod-cutting –** where exactly to cut (sub-rods of which lengths to sell)

Knapsack problem: iterative DP solution



Items (w, v)	W' = 0	W' = 1	W' = 2	W' = 3	W' = 4	W' = 5	W' = 6	W' = 7	W' = 8
i_0 = no item	0	0	0	0	0	0	0	0	0
i_1 (3, 2)	0	0	0	2	2	2	2	2	2
i_2 (4, 3)	0	0	0	2	3	3	3	5	5
i_3 (6, 1)	0	0	0	2	3	3	3	5	5
i_4 (5, 4)	0	0	0	2	3	4	4	5	6

- $val[i = 4, W' = x] = \textbf{max}(\, val[i = 3, W' = x],$

  $val[i = 3, W' - w_4] + v_4 \text{ if } W' - w_4 \geq 0 \text{ else } 0\,)$

- **Q:** Which **items were taken**, and which not?
  - When the „backward” path crosses columns, item was added, when it stays in the same column, it was omitted

Knapsack problem: DP solution

- Save the „path” – for each cell remember where the max came from

```
knapsack_iter(I, W, w, v)
  val = array[|I|+1][W+1]
  pred = array[|I|+1][W+1]
  for i in 0 to len(I)
    val[i][0] = 0
    pred[i][0] = null
  for w’ in 0 to W
    val[0][w’] = 0
    pred[0][w’] = null

  for i in 1 to len(I)
    for w’ in 1 to W
      v’ = 0
      if w’- w[i] ≥ 0
        v’ = val[i-1, w’- w[i]] + v[i]

      if val[i-1, w’] ≥ v’
        val[i, w’] = val[i-1, w’]
        pred[i, w’] = (i-1, w’)
      else
        val[i, w’] = v’
        pred[i, w’] = (i-1, w’- w[i])

  return val, pred
```

```
get_items(I, W, w, v)
  val, pred = knapsack_iter(I, W, w, v)
  print(val[|I|][W]) # optimal value

  i = |I|
  w = W
  while i > 0
    j, w’ = pred[i][w]
    if w’ != w
      print(i)
    i = j
    w = w’
```

Content

- DP Recap
- Knapsack Problem
- Minimal Edit Distance

Minimal Edit Distance

Minimal Edit Distance

The **minimal edit distance problem** asks to determine the minimal number of **atomic/unit operations** that **convert one string to another**. The minimal edit distance is used in many applications (e.g., information retrieval, bioinformatics) as a measure of proximity between sequences of symbols (commonly characters). Most often, the three atomic operations being counted are: (character) **insertion**, **deletion**, and **replacement**

- Often also called just Edit distance or Levenshtein distance
- **Q:** How many **unit operations** do we need to convert
  - „ailgorthm” into **„algorithm”**?
  - „intrligenece” into **„intelligence”**?

Dynamic programming

- Developing a DP algorithm involves the following steps:

1. **Characterize the structure** of an optimal solution
2. **Recursively define** the value of an optimal solution
3. **Compute the value** of an optimal solution
    - **Bottom-up** manner (without recursion)
    - **Memorize the solutions to solved subproblems!!!**
4. **Construct** an optimal solution
    - Not enough just to know what the optimal (max or min) value is
    - Need to know which „sequence of steps" leads to that optimal solution
        - For example, in **rod-cutting –** where exactly to cut (sub-rods of which lengths to sell)

Minimal Edit Distance: **Problem Structure**

- We have two sequence of characters
  - **x** = $x_1, x_2, \ldots, x_m$
  - **y** = $y_1, y_2, \ldots, y_n$

- **Edit distance** between whole strings: *dist*(m, n)
  - *dist*(i, j) indicates the distances between substrings $\mathbf{x_i} = x_1, x_2, \ldots, x_i$ and $\mathbf{y_j} = y_1, y_2, \ldots, y_j$

- **Q:** Express the edit distance between **x** and **y** (m and n) in terms of the edit distances between their substrings?

Minimal Edit Distance: **Problem Structure**

- **Q:** Express the edit distance between **x** and **y** (m and n) in terms of the edit distances between their substrings?

- Edit distance of converting **x** to **y** is the **smallest** of the following:

  1. edit distance between $\mathbf{x}_{m-1}$ and **y** **+ 1** (**deletion** of x_m)
  2. edit distance between **x** and $\mathbf{y}_{n-1}$ **+ 1** (**insertion** of y_n)
  3. edit distance between $\mathbf{x}_{m-1}$ and $\mathbf{y}_{n-1}$ **+** {**1** if $x_m \neq y_n$ else **0**} (**replacement**)

- The same holds for any i and j, not just whole strings (m and n)

Dynamic programming

- Developing a DP algorithm involves the following steps:

1. **Characterize the structure** of an optimal solution
2. **Recursively define** the value of an optimal solution
3. **Compute the value** of an optimal solution
   - **Bottom-up** manner (without recursion)
   - **Memorize the solutions to solved subproblems!!!**
4. **Construct** an optimal solution
   - Not enough just to know what the optimal (max or min) value is
   - Need to know which „sequence of steps” leads to that optimal solution
     - For example, in **rod-cutting –** where exactly to cut (sub-rods of which lengths to sell)

Edit distance: recursive definition of optimal value

$$
\text{dist}(i, j) \begin{cases} \max(i, j) \quad \textbf{if} \quad \min(i, j) = 0 & \text{\# if one of the strings is empty} \\ \min \begin{cases} \text{dist}(i-1, j) + 1 & \text{\# deletion of } x_i \\ \text{dist}(i, j-1) + 1 & \text{\# insertion of } y_j \\ \text{dist}(i-1, j-1) + 1\{x_i \neq y_j\} & \text{\# replacement of } x_i \text{ with } y_j \\ & \quad \text{if not the same} \end{cases} \end{cases}
$$

- **Q:** Write the recursive algorithm for solving the edit distance
- **Q:** Where are the repetitive subproblems?

Edit Distance: **Recursively** (no memorization)

- For the example, we will follow only one thread of recursion (first subproblem)
- „sany” vs. „sam”
  - min(dist(„san”, „sam”) + 1, dist(„sany”, „sa”) + 1, dist(„san”, „sa”) + 1)
- „san” vs. „sam”
  - min(dist(„sa”, „sam”) + 1, dist(„san”, „sa”) + 1, dist(„sa”, „sa”) + 1)
- „sa” vs. „sam”
  - min(dist(„s”, „sam”) + 1, dist(„sa”, „sa”) + 1, dist(„s”, „sa”) + 1)
- „s” vs. „sam”
  - min(dist(„”, „sam”) + 1, dist(„s”, „sa”) + 1, dist(„”, „sa”) + 1)
- „” vs. „sam”
  - return 3

Dynamic programming

- Developing a DP algorithm involves the following steps:

1. **Characterize the structure** of an optimal solution
2. **Recursively define** the value of an optimal solution
3. **Compute the value** of an optimal solution
   - **Bottom-up** manner (without recursion)
4. **Construct** an optimal solution
   - Not enough just to know what the optimal (max or min) value is
   - Need to know which „sequence of steps” leads to that optimal solution
     - For example, in **rod-cutting –** where exactly to cut (sub-rods of which lengths to sell)

Edit Distance: **Iterative solution**

- Create a matrix of size m+1, n+1 (+1 for empty string)
- Initialize [0, j] with j and [i, 0] with i

- Fill the table cell by cell (same as in knapsack)
  - [i, j] = **min**([i-1, j] **+ 1**,
    [i, j-1] **+ 1**,
    [i-1, j-1] **+ 1** *if* $x_i \neq y_j$, **0** *otherwise*)

- **Q:** Write the iterative algorithm for solving the Edit Distance

Example – Levenshtein non-recursively

	_	s	a	m
_	0	1	2	3
s	1	0	1	2
a	2	1	0	1
n	3	2	1	1
y	4	3	2	**2**

Dynamic programming

- Developing a DP algorithm involves the following steps:

1. **Characterize the structure** of an optimal solution
2. **Recursively define** the value of an optimal solution
3. **Compute the value** of an optimal solution
    - **Bottom-up** manner (without recursion)
4. **Construct** an optimal solution
    - Not enough just to know what the optimal (max or min) value is
    - Need to know which „sequence of steps” leads to that optimal solution
        - For example, in **rod-cutting –** where exactly to cut (sub-rods of which lengths to sell)

Edit distance: **solution reconstruction**

- For edit distance, we normally just want the minimal value
- But if we wanted, we could reconstruct the actual edit operations
    - **Q**: How?
    - **A:** Analogous to how we did it for knapsack – for each cell, remember from which of the three possibilities the **min value came**

- **Q:** write the iterative algorithm for solving edit distance that allows for the **reconstruction of the optimal solution**

- **Q**: write the **function that reconstructs** and prints the optimal solution (i.,e., the sequence of edit operations)

Questions?

Awọn ibeere?

ਸਵਾਲ?

Küsimusi?

Turite klausimų?

Pitanja?

Sorusu olan?

Dúvidas?

質問は？

有问题吗？

Fragen?

¿Preguntas?

Frågor?

Questions?

Pytania?

Domande?

Питання?

Vragen?

Ερωτήσεις;

Porandukuéra?

คำถาม?

أسئلة؟