Dynamic Programming 0-1 Knapsack Problem

Design and Analysis of Algorithms

zeshan.khan@nu.edu.pk

0-1 Knapsack Problem

- A bin packing problem
- Similar to fair teams problem from recursion assignment
- You have a set of items
- Each item has a weight and a value
- You have a knapsack with a weight limit
- Goal: Maximize the value of the items you put in the knapsack without exceeding the weight limit

0-1 Knapsack Problem Formally

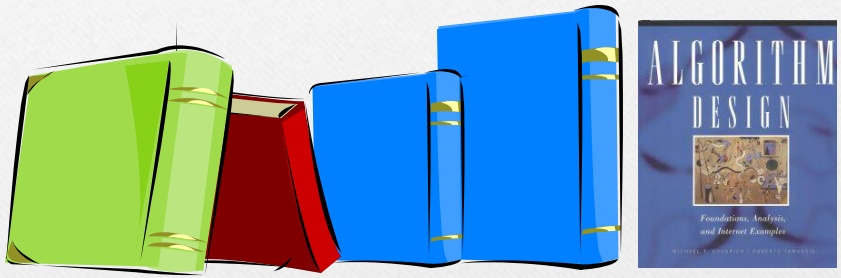
- Given: A set S of n items, with each item i having
  - w_i - a positive weight
  - b_i - a positive benefit
- Goal: Choose items with maximum total benefit but with weight at most W.
- If we are **not** allowed to take fractional amounts, then this is the **0/1 knapsack problem**.
  - In this case, we let T denote the set of items we take
  - Objective: maximize $\sum_{i \in T} b_i$
  - Constraint $\sum_{i \in T} w_i \leq W$

Knapsack

- Given: A set S of n items, with each item i having
  - b_i - a positive "benefit"
  - w_i - a positive "weight"
- Goal: Choose items with maximum total benefit but with weight at most W.

Knapsack

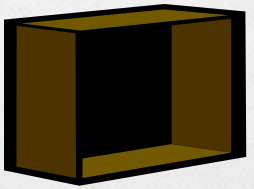
Items:



	1	2	3	4	5
Weight:	4 in	2 in	2 in	6 in	2 in
Benefit:	$20	$3	$6	$25	$80

- From Lectures of Prof. Roger Crawfis

"knapsack"



box of width 9 in

Solution:

- item 5 ($80, 2 in)
- item 3 ($6, 2in)
- item 1 ($20, 4in)

zeshan.khan@nu.edu.pk

Greedy Algorithm: Knapsack

No	Weight	Value	Ratio
1	1	6	6.0
2	2	11	5.5
3	4	1	0.25
4	4	12	3.0
5	6	19	3.167
6	7	12	1.714

- Weight limit = W = 8
- Greedy Solution: Take the highest ratio item that will fit: (1, 6), (2, 11), and (4, 12)
- Total value = 6 + 11 + 12 = 29
- Is this optimal?
  - No
  - Yes

Knapsack DP Solution

- Select maximum from these two below:
  - Select and item and calculate the value
  - Reject an item and calculate the value
- If there is no capacity in the bin return the value computed till now
- $$A(k,w) = \begin{cases} A(k-1,W) & if \quad w_k > W \\ \max(A(k-1,W), A(k-1,W-w_k)+v_k) & if \quad w_k \leq W \end{cases}$$

Knapsack Example

Knapsack of capacity $W = 5$

$w_1 = 2, v_1 = 12$ $w_2 = 1, v_2 = 10$

$w_3 = 3, v_3 = 20$ $w_4 = 2, v_4 = 15$

item	weight	value
1	2	$12
2	1	$10
3	3	$20
4	2	$15

Max item allowed	Max Weight					
	0	**1**	**2**	**3**	**4**	**5**
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	**37**

Algorithm

Since B[k,w] is defined in terms of B[k−1,*], we can use two arrays of instead of a matrix.

Running time is **O**(nW).

Not a polynomial-time algorithm since W may be large.

Called a pseudo-polynomial time algorithm.

Algorithm *01Knapsack*(*S*, *W*):

Input: set *S* of *n* items with benefit b_i and weight w_i; maximum weight *W*

Output: benefit of best subset of *S* with weight at most *W*

let *A* and *B* be arrays of length $W + 1$

for $w \leftarrow 0$ to *W* do

$B[w] \leftarrow 0$

for $k \leftarrow 1$ to *n* do

copy array *B* into array *A*

for $w \leftarrow w_k$ to *W* do

if $A[w-w_k] + b_k > A[w]$ then

$B[w] \leftarrow A[w-w_k] + b_k$

return *B*[*W*]

Longest Common Subsequence

Dynamic Programming

LCS

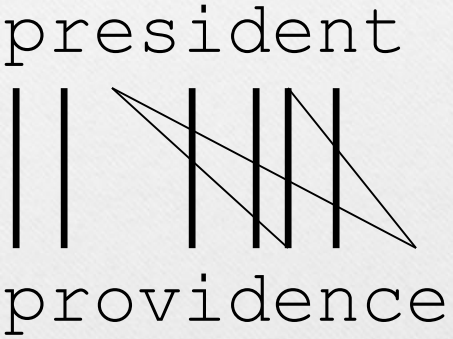
- A subsequence of a sequence/string *S* is obtained by deleting zero or more symbols from *S*. For example, the following are some subsequences of "president": pred, sdn, predent. In other words, the letters of a subsequence of S appear in order in *S*, but they are not required to be consecutive.
- The longest common subsequence problem is to find a maximum length common subsequence between two sequences.

Example

Sequence 1: president

Sequence 2: providence

Its LCS is priden.

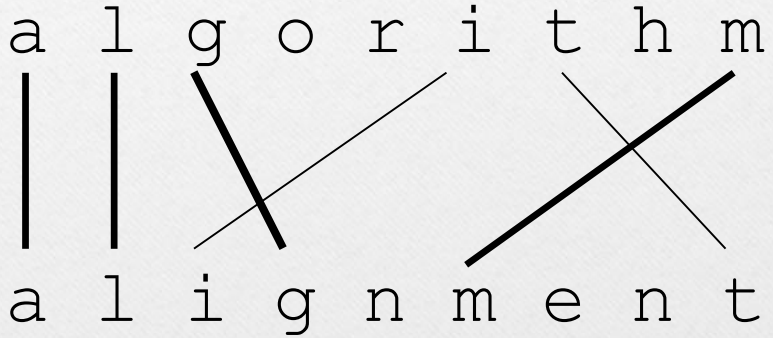


Example

Sequence 1: algorithm

Sequence 2: alignment

One of its LCS is algm



Algorithm 1

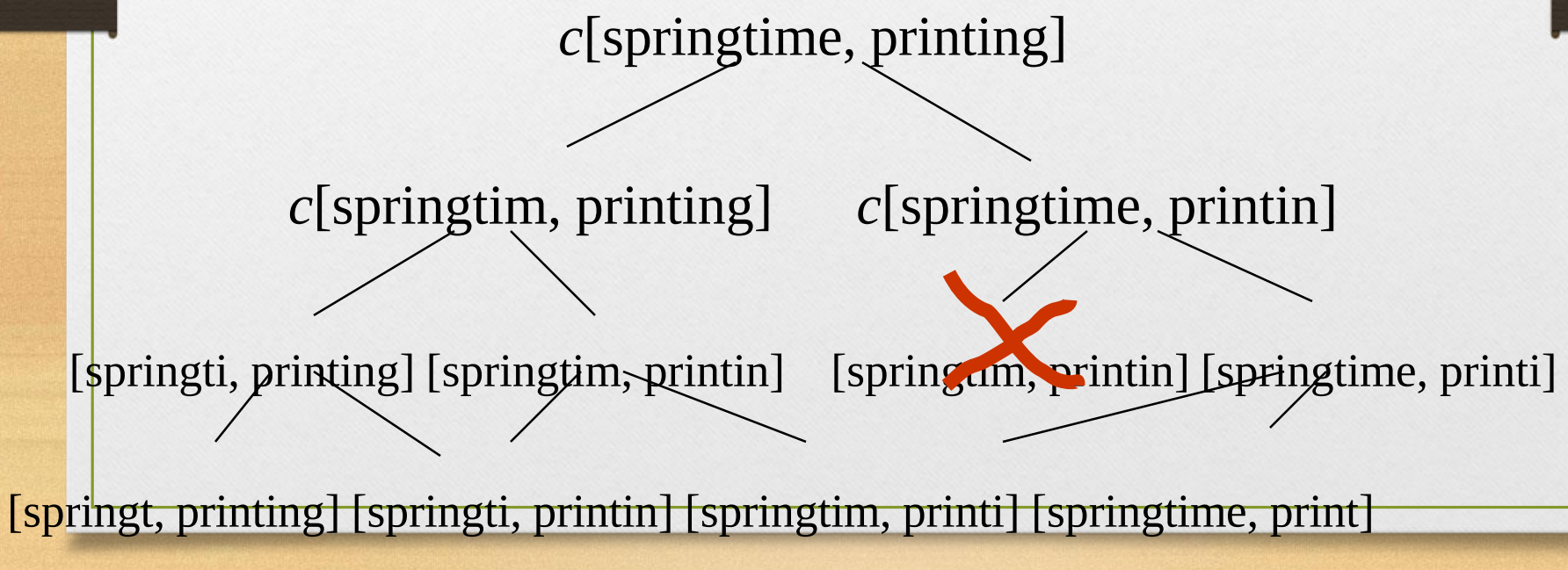
- For every subsequence of X, check whether it's a subsequence of Y.
- Time: $\Theta(n2^m)$.
  - 2^m subsequences of X to check.
  - Each subsequence takes $\Theta(n)$ time to check: scan Y for first letter, for second, and so on.

DP Algorithm

- If
  - The current alphabet of first string is equal to the current alphabet of second string increment LCS length
- Else find maximum by skipping the current alphabet of each string
- $$C(i,j) = \begin{cases} 0 & if \quad i == 0 \; or \; j == 0 \\ c(i-1, j-1) + 1 & if \quad i,j > 0 \; and \; x_i = y_i \\ \max(c(i, j-1), c(i-1, j)) & if \quad i,j > 0 \; and \; x_i \neq y_i \end{cases}$$

Recursive Solution

$$c[\alpha,\beta] = \begin{cases} 0 & \text{if } \alpha \text{ empty or } \beta \text{ empty,} \\ c[\,prefix\,\alpha, prefix\,\beta] + 1 & \text{if } \text{end}(\alpha) = \text{end}(\beta), \\ \max(c[\,prefix\,\alpha, \beta], c[\alpha, prefix\,\beta]) & \text{if } \text{end}(\alpha) \neq \text{end}(\beta). \end{cases}$$



$$c[\alpha,\beta] = \begin{cases} 0 & \text{if } \alpha \text{ empty or } \beta \text{ empty,} \\ c[prefix\,\alpha, prefix\,\beta] + 1 & \text{if } \text{end}(\alpha) = \text{end}(\beta), \\ \max(c[prefix\,\alpha, \beta], c[\alpha, prefix\,\beta]) & \text{if } \text{end}(\alpha) \neq \text{end}(\beta). \end{cases}$$

- Keep track of $c[\alpha,\beta]$ in a table of *nm* entries:
  - top/down
  - bottom/up

		p	r	i	n	t	i	n	g
s									
p									
r									
i									
n									
g									
t									
i									
m									
e									

LCS Algorithm

procedure *LCS-Length(A, B)*

1. **for** $i \leftarrow 0$ **to** m **do** $len(i,0) = 0$
2. **for** $j \leftarrow 1$ **to** n **do** $len(0,j) = 0$
3. **for** $i \leftarrow 1$ **to** m **do**
4. **for** $j \leftarrow 1$ **to** n **do**
5. **if** $a_i = b_j$ **then** $\begin{cases} len(i,j) = len(i-1, j-1) + 1 \\ prev(i,j) = \text{"} \nwarrow \text{"} \end{cases}$
6. **else if** $len(i-1, j) \geq len(i, j-1)$
7. **then** $\begin{cases} len(i,j) = len(i-1, j) \\ prev(i,j) = \text{"} \uparrow \text{"} \end{cases}$
8. **else** $\begin{cases} len(i,j) = len(i, j-1) \\ prev(i,j) = \text{"} \leftarrow \text{"} \end{cases}$
9. **return** *len* and *prev*

i		*j*	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
				p	*r*	*o*	*v*	*i*	*d*	*e*	*n*	*c*	*e*
0			0	0	0	0	0	0	0	0	0	0	0
1	*p*		0	1	1	1	1	1	1	1	1	1	1
2	*r*		0	1	2	2	2	2	2	2	2	2	2
3	*e*		0	1	2	2	2	2	2	3	3	3	3
4	*s*		0	1	2	2	2	2	2	3	3	3	3
5	*i*		0	1	2	2	2	3	3	3	3	3	3
6	*d*		0	1	2	2	2	3	4	4	4	4	4
7	*e*		0	1	2	2	2	3	4	5	5	5	5
8	*n*		0	1	2	2	2	3	4	5	6	6	6
9	*t*		0	1	2	2	2	3	4	5	6	6	6

Running time and memory: O(*mn*) and O(*mn*).

The Backtracking Algorithm

procedure *Output-LCS(A, prev, i, j)*

1. **if** *i = 0* **or** *j = 0* **then return**
2. **if** *prev(i, j)=”↖“* **then**
   Output-LCS(A, prev, i-1, j-1)
   Print A[i];
3. **else if** *prev(i, j)=”↑“* **then**
   Output-LCS(A, prev, i-1, j)
4. **else** *Output-LCS(A, prev, i, j-1)*

Example

i		*j*	0	1	2	3	4	5	6	7	8	9	10
				p	*r*	*o*	*v*	*i*	*d*	*e*	*n*	*c*	*e*
0			0	0	0	0	0	0	0	0	0	0	0
1	*p*		0	1	1	1	1	1	1	1	1	1	1
2	*r*		0	1	2	2	2	2	2	2	2	2	2
3	*e*		0	1	2	2	2	2	2	3	3	3	3
4	*s*		0	1	2	2	2	2	2	3	3	3	3
5	*i*		0	1	2	2	2	3	3	3	3	3	3
6	*d*		0	1	2	2	2	3	4	4	4	4	4
7	*e*		0	1	2	2	2	3	4	5	5	5	5
8	*n*		0	1	2	2	2	3	4	5	6	6	6
9	*t*		0	1	2	2	2	3	4	5	6	6	6

Output: *priden*