CH08-320201 Algorithms and Data Structures

Lecture 19/20 — 24 Apr 2018

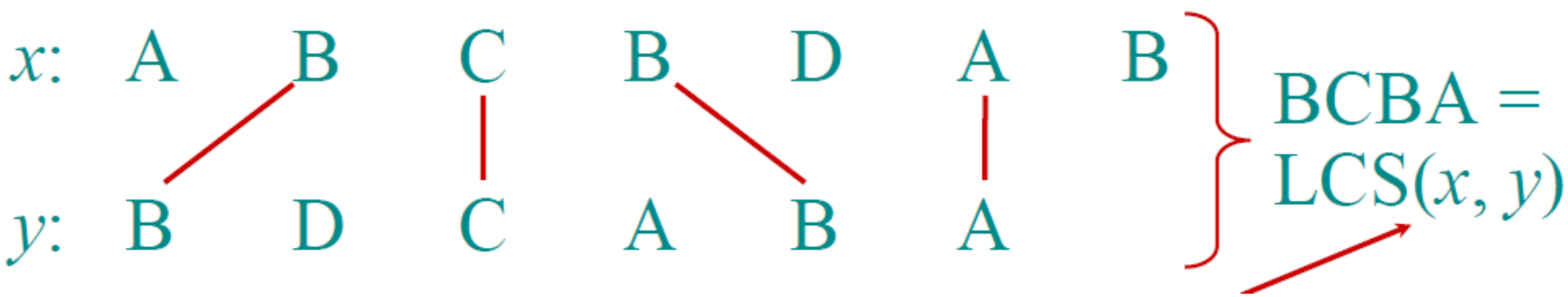
Prof. Dr. Michael Sedlmair

Jacobs University
Spring 2018

4.2 Dynamic programming

Problem

- Given two sequences x[1..m] and y[1..n], find a longest subsequence common to both of them.
- Example:



Brute-force algorithm

- Check every subsequence of x[1..m] to see if it is also a subsequence of y[1..n].

Analysis:

- Checking per subsequence is done in O(n).
- As each bit-vector of m determines a distinct subsequence of x, x has 2^m subsequences.
- Hence, the worst-case running time is O(n 2^m), i.e., it is exponential.

Strategy

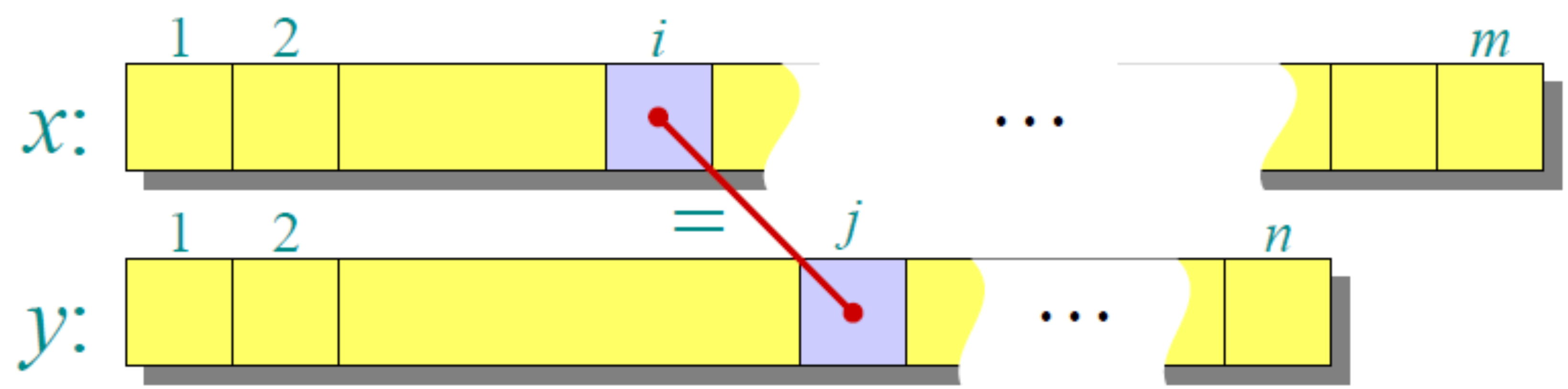
- Look at length of longest-common subsequence.
- Let |s| denote the length of a sequence s.
- To find LCS(x,y), consider **prefixes** of x and y (i.e. we go from right to left)
- Definition: c[i,j] = |LCS(x[1..i], y[1..j])|.
- In particular, c[m,n] = |LCS(x,y)|.

- Theorem (recursive formulation):

$$c[i,j] = \begin{cases} c[i-1,j-1]+1 & \text{if } x[i]=y[j], \\ \max\{c[i-1,j],\, c[i,j-1]\} & \text{otherwise.} \end{cases}$$

Proof

Case x[i] = y[j]:



Let z[1..k] = LCS(x[1..i], y[1..j]) with c[i,j] = k.

Then, z[k] = x[i] = y[j] (else z could be extended).

Thus, z[1..k-1] is CS of x[1..i-1] and y[1..j-1].

Claim: z[1..k-1] = LCS(x[1..i-1], y[1..j-1]).

- Assume w is a longer CS of x[1..i-1] and y[1..j-1], i.e., |w| > k-1.
- Then the concatenation w++z[k] is a CS of x[1..i] and y[1..j] with length > k.
- This contradicts |LCS(x[1..i], y[1..j])| = k.
- Hence, the assumption was wrong and the claim is proven.

Hence, c[i-1,j-1] = k-1, i.e., c[i,j] = c[i-1,j-1] + 1.

Proof

Case x[i] $\neq$ y[j]:

Then, z[k] $\neq$ x[i] or z[k] $\neq$ y[j].

- z[k] $\neq$ x[i]:

  Then, z[1..k] = LCS(x[1..i-1], y[1..j]).

  Thus, c[i-1,j] = k = c[i,j].

- z[k] $\neq$ y[j]:

  Then, z[1..k] = LCS(x[1..i], y[1..j-1]).

  Thus, c[i,j-1] = k = c[i,j].

In summary, c[i,j] = max{c[i-1,j], c[i,j-1]}.

Dynamic programming concept

Step 1: Optimal substructure.

An optimal solution to a problem contains optimal solutions to subproblems.

Example:

If z = LCS(x,y), then any prefix of z is an LCS of a prefix of x and a prefix of y.

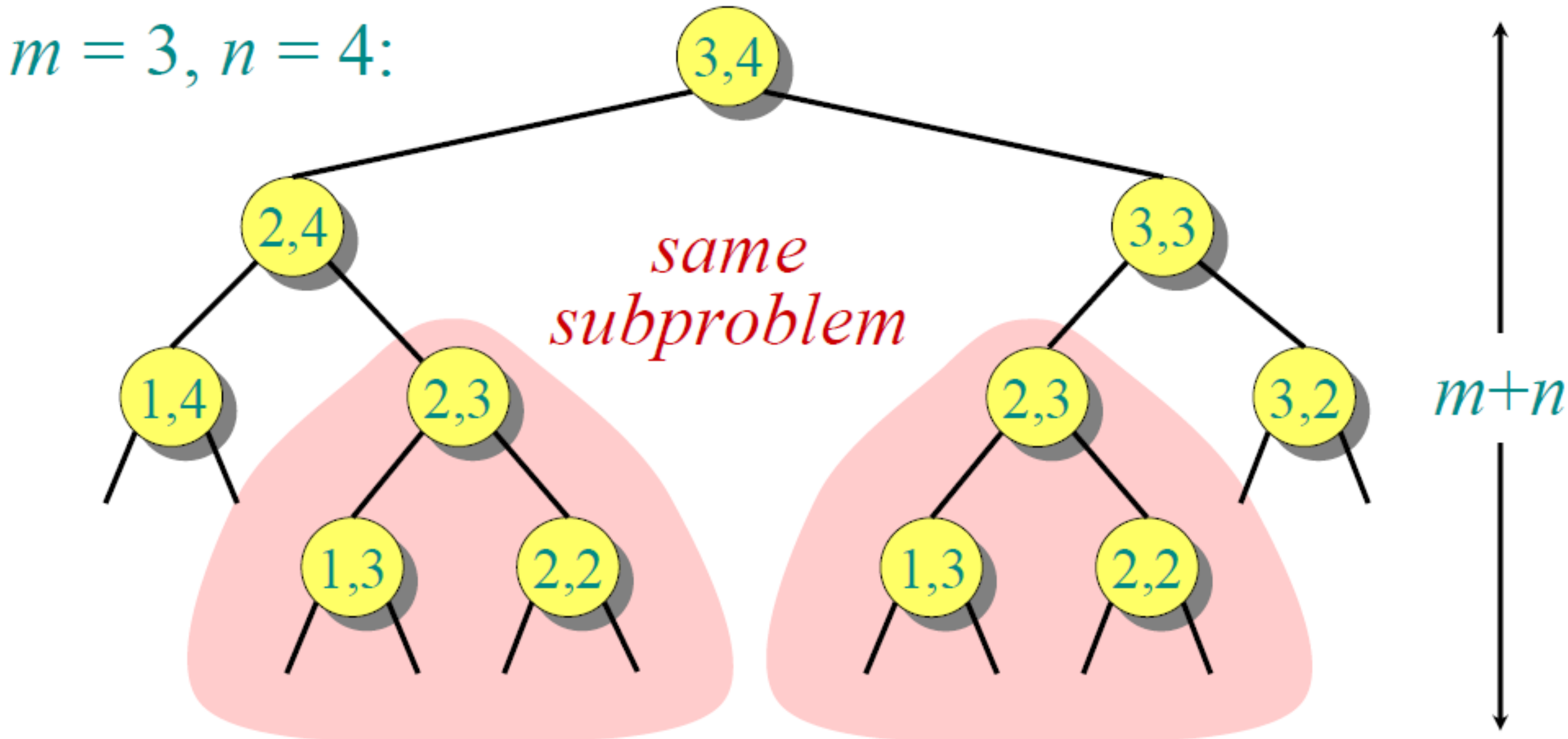
Recursive algorithm

- Computation of the length of LCS:

```
LCSlength(x,y,i,j):
   if i=0 or j=0
      return 0
   else if x[i] = y[j]
      return LCSlength(x,y,i-1,j-1)+1
   else return max {LCSlength(x,y,i-1,j),
                    LCSlength(x,y,i,j-1)}
```

- Remark: if $x[i] \neq y[j]$, the algorithm evaluates two subproblems that are very similar.

Recursive tree



Height $= m + n \Rightarrow$ work potentially exponential, but we're solving subproblems already solved!

Dynamic programming concept

Step 2: Overlapping subproblems.

A recursive solution contains a „small" number of distinct subproblems repeated many times.

Example:

The number of distinct LCS subproblems for two prefixes of lengths m and n is only mn.

Memoization algorithm

Memoization:

After computing a solution to a subproblem, store it in a table.

Subsequent calls check the table to avoid repeating the same computation.

Recursive algorithm with memoization

Computation of the length of LCS:

```
LCSlength'(x,y,i,j):
   if c[i,j] = NIL
      then if i=0 or j=0
              c[i,j] = 0
      else if x[i] = y[j]
              c[i,j] = LCSlength'(x,y,i-1,j-1)+1
      else c[i,j] = max {LCSlength'(x,y,i-1,j),
                         LCSlength'(x,y,i,j-1)}
return c[i,j]
```

Dynamic programming

- Compute the table bottom-up:

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0	1	2	2	3	3	4	4

Complexity

- Time complexity:
  $T(m,n) = \Theta(mn)$.
- Space complexity:
  $S(m,n) = \Theta(mn)$

Reconstructing LCS

- Trace backwards:

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0	1	2	2	3	3	4	4

- Time complexity = O(m+n).

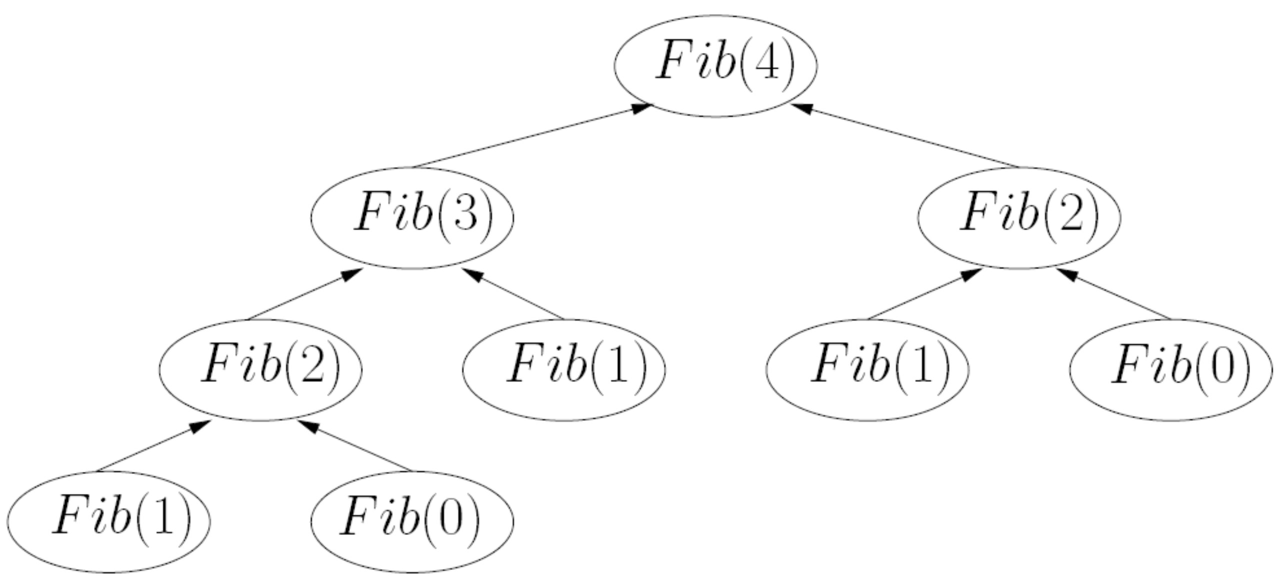
Fibonacci numbers revisited

Recall:

- Recursive definition:

$$F_n = \begin{cases} 0 & \text{if } n = 0; \\ 1 & \text{if } n = 1; \\ F_{n-1} + F_{n-2} & \text{if } n \geq 2. \end{cases}$$

- Recursion tree of brute-force implementation:

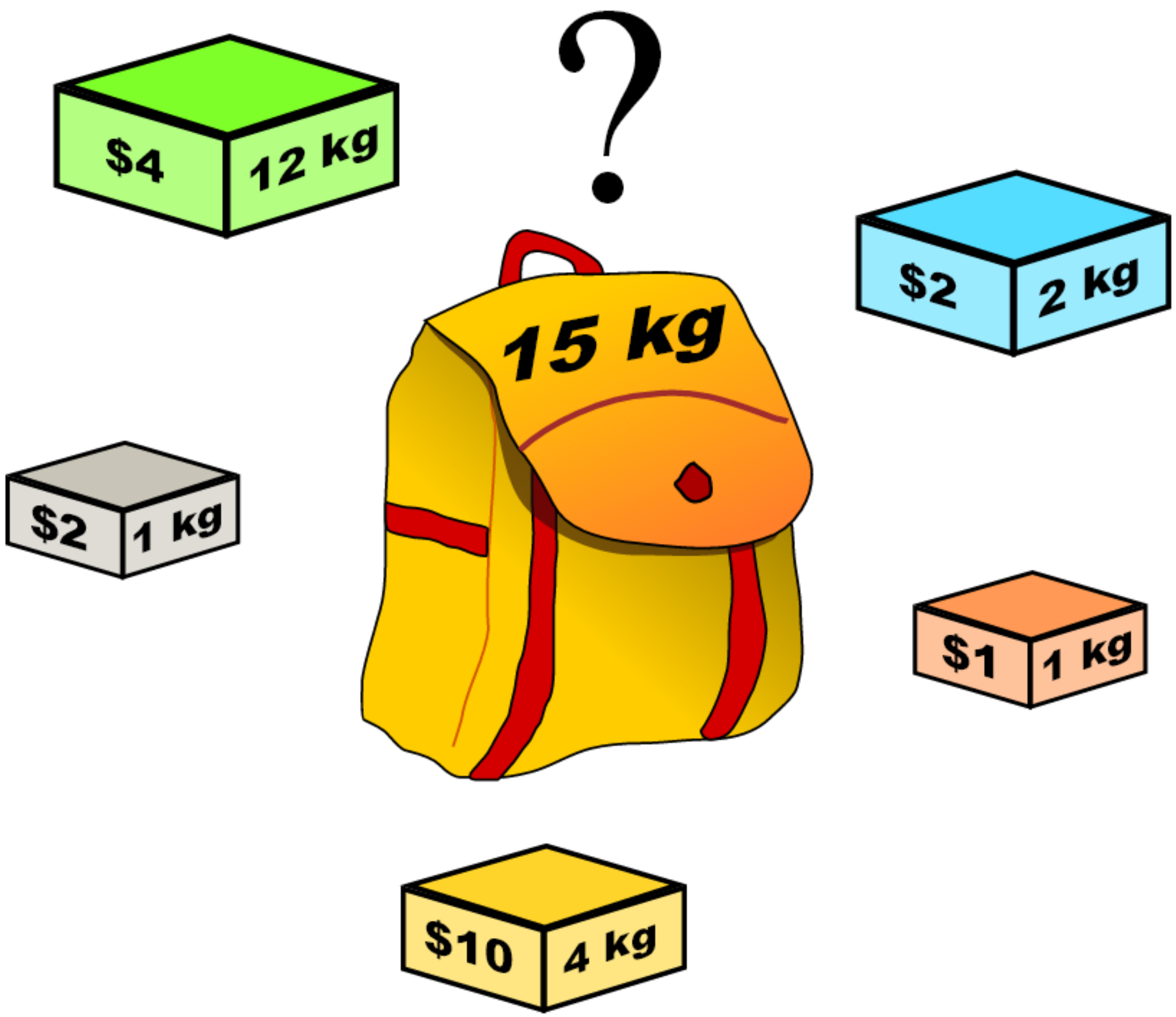


Fibonacci numbers revisited

Dynamic programming solution:

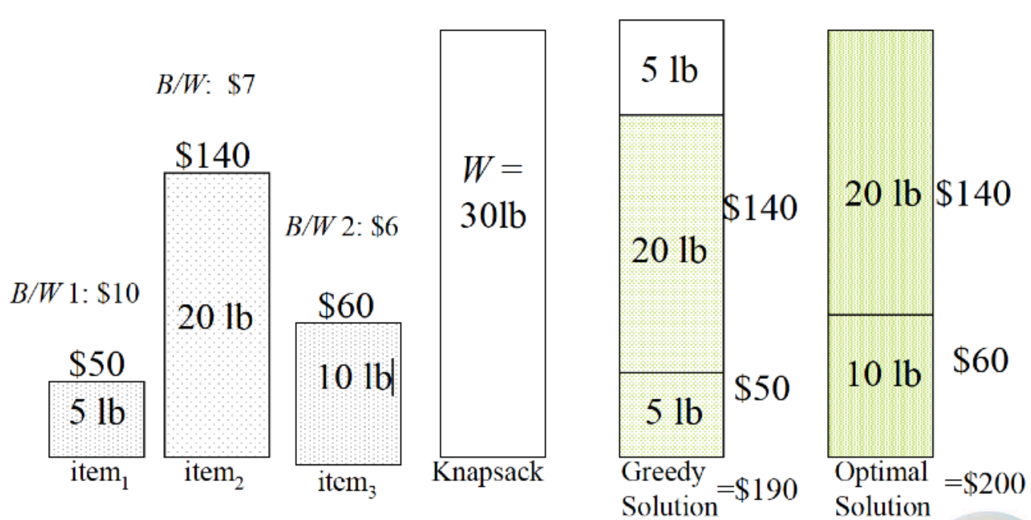
- Avoid re-computations of same terms.
- Store results of subproblems in a table.
- Thus, Fib(k) is computed exactly once for each k.
- This basically leads to the previously discussed bottom-up approach.
- Computation time is $T(n) = \Theta(n)$.

Knapsack problem (revisited)



Greedy algorithm

- Greedy approaches make a locally optimal choice.
- There is no guarantee that this will lead to a globally optimal solution.
- In the 0-1 Knapsack problem it did not.



Dynamic-programming approach

- Let us try a dynamic-programming approach.
- We need to carefully identify the subproblems
- If items are labeled *1..n*, then a subproblem would be to find an optimal solution for S_k = *{items labeled 1, 2,..., k}*.

Dynamic-programming approach

Max weight: W = 20

$w_1=2$ $b_1=3$	$w_2=4$ $b_2=5$	$w_3=5$ $b_3=8$	$w_4=3$ $b_4=4$	

For S_4:

Total weight: 14

Maximum benefit: 20

$w_1=2$ $b_1=3$	$w_2=4$ $b_2=5$	$w_3=5$ $b_3=8$	$w_5=9$ $b_5=10$

For S_5:

Total weight: 20

Maximum benefit: 26

S_4

S_5

Item #	Weight w_i	Benefit b_i
1	2	3
2	4	5
3	5	8
4	3	4
5	9	10

Solution for S_4 is not part of the solution for S_5

Dynamic-programming approach

- Re-define the subproblem by also considering the weight that is given to the subproblem.
- The subproblem then will be to compute V[k,w], i.e., to find an optimal solution for S_k = *{items labeled 1, 2, .. k}* in a knapsack of size w, with w ≤ W.
- V[k,w] denotes the overall benefit of the solution.
- Question: Assuming we know V[i,j] for i = 0,1,2,…,k-1 and j = 0,1,2,…,w, how can we derive V[k,w]?
- Answer:

$$V[k,w] = \begin{cases} V[k-1,w] & \text{if } w_k > w \\ \max\{V[k-1,w], V[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

Dynamic-programming approach

Explanation of

$$V[k,w] = \begin{cases} V[k-1,w] & \text{if } w_k > w \\ \max\{V[k-1,w], V[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

- The best subset of S_k that has the total weight $\leq w$, either contains item k or not.
- First case: $w_k > w$.
  Item k cannot be part of the solution, since if it was, the total weight would be $> w$, which is unacceptable.
- Second case: $w_k \leq w$.
  Then the item k can be in the solution, and we choose *the case with greater value*.

Dynamic-programming approach

Dynamic-programming algorithm:

Input: $S_n = \{(w_i,b_i): i = 1,\ldots,n\}$ *and maximum weight* W

```
for w = 0 to W
   V[0,w] = 0
for i = 1 to n
   V[i,0] = 0
for i = 1 to n
   for w = 0 to W
      if (wi > w) // item i cannot be part of the solution
            V[i,w] = V[i-1,w]
      else          // wi <= w
            if (V[i-1,w] > bi + V[i-1,w-wi])
                  V[i,w] = V[i-1,w]
            else
                  V[i,w] = bi + V[i-1,w-wi]
```

Dynamic-programming approach

Computation time:

O(W)

O(n)

O(nW)

```
for w = 0 to W
   V[0,w] = 0
for i = 1 to n
   V[i,0] = 0
for i = 1 to n
   for w = 0 to W
      if (wi > w)
            V[i,w] = V[i-1,w]
      else
            if (V[i-1,w] > bi + V[i-1,w-wi])
                  V[i,w] = V[i-1,w]
            else
                  V[i,w] = bi + V[i-1,w-wi]
```

Overall time complexity is **O(nW)**

Dynamic-programming approach

Example:

- n = 4 (# of elements)
- W = 5 (maximum weight)
- Elements (weight, benefit):
  (2,3), (3,4), (4,5), (5,6)

Dynamic-programming approach

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1						
2						
3						
4						

```
for w = 0 to W
  V[0,w] = 0
```

Dynamic-programming approach

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					

```
for i = 1 to n
  V[i,0] = 0
```

Dynamic-programming approach

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	**0**				
2	0					
3	0					
4	0					

$i=1$

$b_i=3$

$w_i=2$

$w=1$

$w-w_i=-1$

Items:

$\boxed{1: (2,3)}$

2: (3,4)

3: (4,5)

4: (5,6)

```
if wi <= w // item i can be part of the solution
   if bi + V[i-1,w-wi] > V[i-1,w]
      V[i,w] = bi + V[i-1,w-wi]
   else
      V[i,w] = V[i-1,w]
else V[i,w] = V[i-1,w] // wi > w
```

Dynamic-programming approach

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	**3**			
2	0					
3	0					
4	0					

$i=1$

$b_i=3$

$w_i=2$

$w=2$

$w-w_i=0$

Items:

$\boxed{1: (2,3)}$

2: (3,4)

3: (4,5)

4: (5,6)

```
if wi <= w // item i can be part of the solution
   if bi + V[i-1,w-wi] > V[i-1,w]
      V[i,w] = bi + V[i-1,w-wi]
   else
      V[i,w] = V[i-1,w]
else V[i,w] = V[i-1,w] // wi > w
```

Dynamic-programming approach

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	**3**		
2	0					
3	0					
4	0					

$i=1$

$b_i=3$

$w_i=2$

$w=3$

$w-w_i=1$

Items:

$\boxed{1: (2,3)}$

2: (3,4)

3: (4,5)

4: (5,6)

```
if w_i <= w // item i can be part of the solution
  if b_i + V[i-1,w-w_i] > V[i-1,w]
    V[i,w] = b_i + V[i-1,w-w_i]
  else
    V[i,w] = V[i-1,w]
else V[i,w] = V[i-1,w] // w_i > w
```

Dynamic-programming approach

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	**3**	
2	0					
3	0					
4	0					

$i=1$

$b_i=3$

$w_i=2$

$w=4$

$w\text{-}w_i=2$

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

```
if wi <= w // item i can be part of the solution
   if bi + V[i-1,w-wi] > V[i-1,w]
      V[i,w] = bi + V[i-1,w-wi]
   else
      V[i,w] = V[i-1,w]
else V[i,w] = V[i-1,w] // wi > w
```

Dynamic-programming approach

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	**3**
2	0					
3	0					
4	0					

$i=1$
$b_i=3$
$w_i=2$
$w=5$
$w-w_i=3$

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

```
if wi <= w // item i can be part of the solution
   if bi + V[i-1,w-wi] > V[i-1,w]
      V[i,w] = bi + V[i-1,w-wi]
   else
      V[i,w] = V[i-1,w]
else V[i,w] = V[i-1,w] // wi > w
```

Dynamic-programming approach

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	**0**				
3	0					
4	0					

$i=2$

$b_i=4$

$w_i=3$

$w=1$

$w-w_i=-2$

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

```
if wi <= w // item i can be part of the solution
   if bi + V[i-1,w-wi] > V[i-1,w]
      V[i,w] = bi + V[i-1,w-wi]
   else
      V[i,w] = V[i-1,w]
else V[i,w] = V[i-1,w] // wi > w
```

Dynamic-programming approach

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	**3**			
3	0					
4	0					

$i=2$

$b_i=4$

$w_i=3$

$w=2$

$w-w_i=-1$

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

```
if wi <= w // item i can be part of the solution
   if bi + V[i-1,w-wi] > V[i-1,w]
      V[i,w] = bi + V[i-1,w-wi]
   else
      V[i,w] = V[i-1,w]
else V[i,w] = V[i-1,w] // wi > w
```

Dynamic-programming approach

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	**4**		
3	0					
4	0					

$i=2$

$b_i=4$

$w_i=3$

$w=3$

$w-w_i=0$

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

```
if w_i <= w // item i can be part of the solution
   if b_i + V[i-1,w-w_i] > V[i-1,w]
      V[i,w] = b_i + V[i-1,w-w_i]
   else
      V[i,w] = V[i-1,w]
else V[i,w] = V[i-1,w] // w_i > w
```

Dynamic-programming approach

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	**4**	
3	0					
4	0					

$i=2$

$b_i=4$

$w_i=3$

$w=4$

$w-w_i=1$

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

```
if wi <= w // item i can be part of the solution
   if bi + V[i-1,w-wi] > V[i-1,w]
      V[i,w] = bi + V[i-1,w-wi]
   else
      V[i,w] = V[i-1,w]
else V[i,w] = V[i-1,w] // wi > w
```

Dynamic-programming approach

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	**7**
3	0					
4	0					

$i=2$

$b_i=4$

$w_i=3$

$w=5$

$w-w_i=2$

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

```
if wi <= w // item i can be part of the solution
   if bi + V[i-1,w-wi] > V[i-1,w]
      V[i,w] = bi + V[i-1,w-wi]
   else
      V[i,w] = V[i-1,w]
else V[i,w] = V[i-1,w] // wi > w
```

Dynamic-programming approach

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	**0**	**3**	**4**		
4	0					

$i=3$

$b_i=5$

$w_i=4$

$w=1..3$

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

```
if wi <= w // item i can be part of the solution
   if bi + V[i-1,w-wi] > V[i-1,w]
     V[i,w] = bi + V[i-1,w-wi]
   else
     V[i,w] = V[i-1,w]
else V[i,w] = V[i-1,w] // wi > w
```

Dynamic-programming approach

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	**5**	
4	0					

$i=3$

$b_i=5$

$w_i=4$

$w=4$

$w-w_i=0$

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

```
if wi <= w // item i can be part of the solution
   if bi + V[i-1,w-wi] > V[i-1,w]
      V[i,w] = bi + V[i-1,w-wi]
   else
      V[i,w] = V[i-1,w]
else V[i,w] = V[i-1,w] // wi > w
```

Dynamic-programming approach

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	**7**
4	0					

$i=3$

$b_i=5$

$w_i=4$

$w=5$

$w-w_i=1$

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

```
if wi <= w // item i can be part of the solution
   if bi + V[i-1,w-wi] > V[i-1,w]
     V[i,w] = bi + V[i-1,w-wi]
   else
     V[i,w] = V[i-1,w]
else V[i,w] = V[i-1,w] // wi > w
```

Dynamic-programming approach

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	**0**	**3**	**4**	**5**	

$i=4$

$b_i=6$

$w_i=5$

$w=1..4$

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

```
if wi <= w // item i can be part of the solution
   if bi + V[i-1,w-wi] > V[i-1,w]
      V[i,w] = bi + V[i-1,w-wi]
   else
      V[i,w] = V[i-1,w]
else V[i,w] = V[i-1,w] // wi > w
```

Dynamic-programming approach

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

$i=4$

$b_i=6$

$w_i=5$

$w=5$

$w-w_i=0$

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

```
if w_i <= w // item i can be part of the solution
   if b_i + V[i-1,w-w_i] > V[i-1,w]
      V[i,w] = b_i + V[i-1,w-w_i]
   else
      V[i,w] = V[i-1,w]
else V[i,w] = V[i-1,w] // w_i > w
```

Dynamic-programming approach

- This algorithm only finds the maximally possible value that can be carried in the knapsack, i.e., the value of V[n,W].
- To know the items that are put together to reach this maximum value, an addition to this algorithm is necessary that is based on traversing the table in a post-processing step.
- Algorithm:

```
i=n, k=W
while (i > 0 and k > 0)
    if (V[i,k] ≠ V[i-1,k])
        add item i to knapsack
        i = i-1, k = k-wi
    else // item i is not in the knapsack
        i = i-1
```

Dynamic-programming approach

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

$i=4$
$k=5$
$b_i=6$
$w_i=5$
$V[i,k]=7$
$V[i-1,k]=7$

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

```
i=n, k=W
while (i > 0 and k > 0)
    if (V[i,k] ≠ V[i−1,k])
        mark the ith item as in the knapsack
        i = i-1, k = k-wi
    else
        i = i-1
```

Dynamic-programming approach

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

$i=4$
$k=5$
$b_i=6$
$w_i=5$
$V[i,k]=7$
$V[i-1,k]=7$

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

```
i=n, k=W
while (i > 0 and k > 0)
    if (V[i,k] ≠ V[i−1,k])
        mark the ith item as in the knapsack
        i = i-1, k = k-wi
    else
        i = i-1
```

Dynamic-programming approach

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

i=3
k=5
b_i=5
w_i=4
V[i,k]=7
V[i-1,k]=7

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

```
i=n, k=W
while (i > 0 and k > 0)
    if (V[i,k] ≠ V[i−1,k])
        mark the ith item as in the knapsack
        i = i-1, k = k-wi
    else
        i = i-1
```

Dynamic-programming approach

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

$i=2$
$k=5$
$b_i=4$
$w_i=3$
$V[i,k]=7$
$V[i-1,k]=3$
$k-w_i=2$

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

```
i=n, k=W
while (i > 0 and k > 0)
    if (V[i,k] ≠ V[i-1,k])
        mark the ith item as in the knapsack
        i = i-1, k = k-wi
    else
        i = i-1
```

Dynamic-programming approach

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	**3**	3	3	3
2	0	0	3	4	4	**7**
3	0	0	3	4	5	7
4	0	0	3	4	5	7

$i=1$

$k=2$

$b_i=3$

$w_i=2$

$V[i,k]=3$

$V[i-1,k]=0$

$k-w_i=0$

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

```
i=n, k=W
while (i > 0 and k > 0)
    if (V[i,k] ≠ V[i−1,k])
        mark the ith item as in the knapsack
        i = i-1, k = k-wi
    else
        i = i-1
```

Dynamic-programming approach

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	**3**	3	3	3
2	0	0	3	4	4	**7**
3	0	0	3	4	5	7
4	0	0	3	4	5	7

i=0

k=0

The optimal knapsack should contain {1,2}

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

```
i=n, k=W
while (i > 0 and k > 0)
    if (V[i,k] ≠ V[i-1,k])
        mark the ith item as in the knapsack
        i = i-1, k = k-wi
    else
        i = i-1
```

4.3 Summary

Summary

We have discussed 3 algorithmic concepts:

1. Divide & Conquer
   - Splits problem into multiple subproblems, solves them recursively, and combines the solutions.
2. Greedy Algorithms
   - Makes a locally best choice to reduce the problem to a subproblem and iteratively solves the subproblem in the hope to find a globally best solution.
3. Dynamic Programming
   - Computes subproblems in a bottom-up fashion and stores (intermediate) solutions to subproblems in a table.