THE UNIVERSITY OF MELBOURNE

COMP90038
Algorithms and Complexity

Lecture 18: Dynamic Programming
(with thanks to Harald Søndergaard & Michael Kirley)

Casey Myers
Casey.Myers@unimelb.edu.au
David Caro Building (Physics) 274

Review from Lecture 17: Hashing

- If we have a hash table of size m and keys are integers, we may define

$$h(n) = n \bmod m.$$

- But keys may be other things, such as strings of characters, and the hash function should apply to these and still be easy (cheap) to compute.

- We need to choose m so that it is large enough to allow efficient operations, without taking up excessive memory.

- The hash function should distribute keys evenly along the cells of the hash table.

Review from Lecture 17: Hashing of Strings

$char$	M	Y	K	E	Y
s_i	12	24	10	4	24
$bin(s_i)$	01100	11000	01010	00100	11000
i	0	1	2	3	4

- Now concatenate the binary string:

$$M\ Y\ K\ E\ Y \longmapsto 01100110000101000010011000\ (=13379736)$$
$$13379736 \bmod 101 = 64$$

- So 64 is the position of string of string $M\ Y\ K\ E\ Y$ in the hash table.

- We deliberately chose m to be prime.

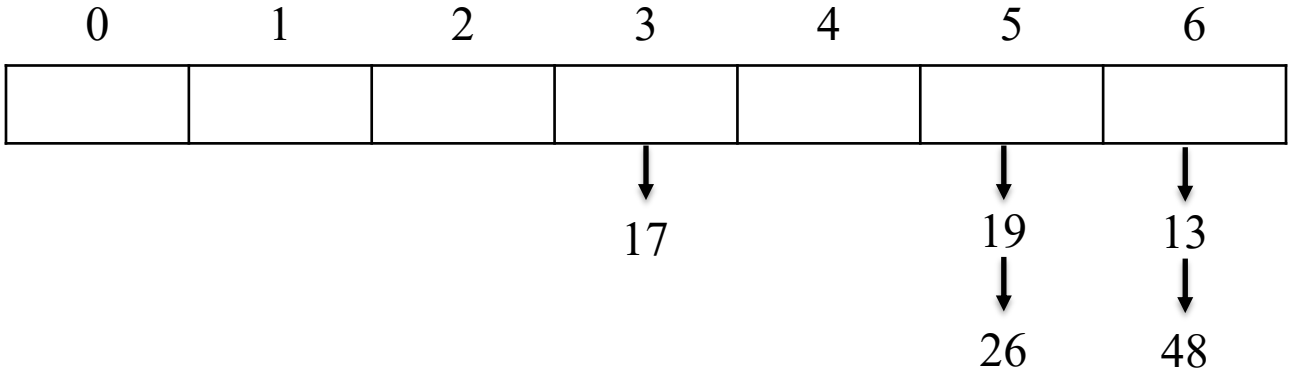
$$13379736 = 12 \times 32^4 + 24 \times 32^3 + 10 \times 32^2 + 4 \times 32^1 + 24 \times 32^0$$

- With m = 32, the hash value of any key is the last character's value!

Review from Lecture 17: Collision Handling

- Consider $h(k) = k \ mod \ 7$. Draw the resulting hash tables after inserting $19, 26, 13, 48, 17$ (in this order).

- Separate chaining



- Linear probing

0	1	2	3	4	5	6
13	48		17		19	26

- Double hashing, using $s(k) = 5 - (k \ mod \ 5)$ offset

0	1	2	3	4	5	6
	48	26	17		19	13

Review from Lecture 17: Rabin-Karp String Search

- Repeatedly hashing strings of length m seems like a bad idea. However, the hash values can be calculated incrementally. The hash value of the length-m substring s that starts at position j is:

$$hash(s,j) = \sum_{i=0}^{m-1} chr(s_{j+i}) \times a^{m-i-1},$$

where a is the alphabet size. From that we we can get the next hash value, for the substring that starts at position $j+1$, quite cheaply:

$$hash(s,j+1) = \left(hash(s,j) - a^{m-1} chr(s_j)\right) \times a + chr(s_{j+m})$$

modulo m.

- The first substring "the" $= t \cdot (26)^2 + h \cdot (26) + e$
- If we have "the", can we compute "her"?

$$\begin{aligned} \text{"her"} &= h \cdot (26)^2 + e \cdot (26) + r \\ &= 26 \cdot (h \cdot (26) + e) + r \\ &= 26 \cdot (t.(26)^2 + h \cdot (26) + e - t.(26)^2) + r \\ &= 26 \cdot (\text{"the"} - t.(26)^2) + r \end{aligned}$$

Dynamic Programming

- Dynamic programming is an algorithm design technique that is sometimes applicable when we want to solve a recurrence relation and the recursion involves overlapping instances.

- In Lecture 16 we achieved a spectacular performance improvement in the calculation of Fibonacci numbers by switching from a naïve top-down algorithm to one that solved, and tabulated, smaller sub-problems.

- The bottom-up approach used the tabulated results, rather than solving overlapping sub-problems repeatedly.

- That was a particularly simple example of dynamic programming.

Review from Lecture 16: Fibonacci Numbers with Tabulation

- We assume that, from the outset, all entries of the table F are 0.

```
function FIB(n)
    if n = 0 or n = 1 then
        return 1
    result ← F[n]
    if result = 0 then
        result ← FIB(n − 1) + FIB(n − 2)
        F[n] ← result
    return result
```

Initial	$F = 0,0,\cdots,0$
$n = 2$	result=FIB(1)+FIB(0) =1+1=2 $F = 0,0,2,0,\cdots,0$
$n = 2$	result=FIB(2)+FIB(1) =2+1=3 $F = 0,0,2,3,0,\cdots,0$
$n = 4$	result=FIB(3)+FIB(2) =3+2=5 $F = 0,0,2,3,5,0,\cdots,0$

- $F[0,\cdots,n]$ is an array that stores partial results, initialised to 0.
- Base cases FIB(0) = 1 & FIB(1) = 1.

Dynamic Programming and Optimisation

- Optimisation problems sometimes allow for clever dynamic programming solutions.
  - the objective is to find the best possible combination: the one with the lowest cost, or higher profit, subject to some constraints.
- For dynamic programming to be useful, the optimality principle must hold:
  - *An optimal solution to a problem is composed of optimal solutions to its subproblems.*
- While not always, this principle often holds.

Example 1: The Coin-Row Problem

- Given a row of coins, pick the largest possible sum, subject to this constraint: no two adjacent coins can be picked.

Example 1: The Coin-Row Problem

- Given a row of coins, pick the largest possible sum, subject to this constraint: no two adjacent coins can be picked.

- For example, consider the coins: 20 10 20 50 20 10 20

Example 1: The Coin-Row Problem

- Given a row of coins, pick the largest possible sum, subject to this constraint: no two adjacent coins can be picked.

- For example, consider the coins: 20 10 20 50 20 10 20

- We cannot take these two coins, since they are neighbours.

Example 1: The Coin-Row Problem

- Given a row of coins, pick the largest possible sum, subject to this constraint: no two adjacent coins can be picked.

- For example, consider the coins: 20 10 20 50 20 10 20

- We about this combination, is 80 the maximum profit?

Example 1: The Coin-Row Problem

- Given a row of coins, pick the largest possible sum, subject to this constraint: no two adjacent coins can be picked.
- Think of the problem recursively.
- Let the values of the coins be $v_1, v_2, \cdots, v_n$.
- Let $S(i)$ be the sum that can be gotten by picking optimally from the first i coins.
- Either the ith coin (with value v_i) is part of the solution or it is not.
- If we choose to pick the ith coin, we cannot also pick its neighbour on the left, so the best we can achieve is $\mathrm{S}(i-2) + v_i$.
- Otherwise we can leave it, and the best we can achieve is $\mathrm{S}(i-1)$.

Example 1: The Coin-Row Problem

- We can say the same thing formally, as a recurrence relation:

$$S(i) = max\{S(i-1), S(i-2) + v_i\}$$

- This holds for $i > 1$.

- We need two base cases: $S(0) = 0$ and $S(1) = v_1$.

Example 1: The Coin-Row Problem

- We can say the same thing formally, as a recurrence relation:

$$S(i) = max\{S(i-1), S(i-2) + v_i\}$$

- This holds for $i > 1$.
- We need two base cases: $S(0) = 0$ and $S(1) = v_1$.
- You can code this algorithm directly like that in your favourite programming language.
- However, the solutions suffers the same problem as the naïve Fibonacci program: lots of repetition of identical sub-computations.

Example 1: The Coin-Row Problem

- Since all values S(1) to S(n) need to be found anyway, we may as well proceed from the bottom up, storing intermediate results in an array S as we go.

- Given an array C that holds the coin values, the recurrence relation tells us what to do:

function CoinRow($C[1..n]$)
    $S[0] \leftarrow 0$
    $S[1] \leftarrow C[1]$
    for $i \leftarrow 2$ to n **do**
        $S[i] \leftarrow max\{S[i-1], S[i-2] + C[i]\}$
    return $S[n]$

Example 1: The Coin-Row Problem

- Let us run the algorithm on the seven coins 20, 10, 20, 50, 20, 10, 20:

function CoinRow($C[1..n]$)
  $S[0] \leftarrow 0$
  $S[1] \leftarrow C[1]$
  for $i \leftarrow 2$ to n **do**
    $S[i] \leftarrow max\{S[i-1], S[i-2] + C[i]\}$
  return $S[n]$

index	0	1	2	3	4	5	6	7
C		20	10	20	50	20	10	20
S	0							

- $i = 0$
- $S[0] = 0$

Example 1: The Coin-Row Problem

- Let us run the algorithm on the seven coins 20, 10, 20, 50, 20, 10, 20:

function CoinRow($C[1..n]$)
  $S[0] \leftarrow 0$
  $S[1] \leftarrow C[1]$
  for $i \leftarrow 2$ to n **do**
    $S[i] \leftarrow max\{S[i-1], S[i-2] + C[i]\}$
  return $S[n]$

index	0	1	2	3	4	5	6	7
C		20	10	20	50	20	10	20
S	0	20						

- $i = 1$
- $S[1] = 20$

Example 1: The Coin-Row Problem

- Let us run the algorithm on the seven coins 20, 10, 20, 50, 20, 10, 20:

function CoinRow($C[1..n]$)
  $S[0] \leftarrow 0$
  $S[1] \leftarrow C[1]$
  for $i \leftarrow 2$ to n **do**
    $S[i] \leftarrow max\{S[i-1], S[i-2] + C[i]\}$
  return $S[n]$

index	0	1	2	3	4	5	6	7
C		20	10	20	50	20	10	20
S	0	20	20					

- $i = 2$
- $S[2] = max(\ S[1] = 20,\ S[0] + 10 = 0 + 10\ ) = 20$

Example 1: The Coin-Row Problem

- Let us run the algorithm on the seven coins 20, 10, 20, 50, 20, 10, 20:

function CoinRow($C[1..n]$)
  $S[0] \leftarrow 0$
  $S[1] \leftarrow C[1]$
  for $i \leftarrow 2$ to n **do**
    $S[i] \leftarrow max\{S[i-1], S[i-2] + C[i]\}$
  return $S[n]$

index	0	1	2	3	4	5	6	7
C		20	10	20	50	20	10	20
S	0	20	20	40				

- $i = 3$
- $S[3] = max(\, S[2] = 20,\ S[1] + 20 = 20 + 20\,) = 40$

Example 1: The Coin-Row Problem

- Let us run the algorithm on the seven coins 20, 10, 20, 50, 20, 10, 20:

function CoinRow($C[1..n]$)
  $S[0] \leftarrow 0$
  $S[1] \leftarrow C[1]$
  for $i \leftarrow 2$ to n **do**
    $S[i] \leftarrow max\{S[i-1], S[i-2] + C[i]\}$
  return $S[n]$

index	0	1	2	3	4	5	6	7
C		20	10	20	50	20	10	20
S	0	20	20	40	70			

- $i = 4$
- $S[4] = max(\, S[3] = 40,\ S[2] + 50 = 20 + 50\,) = 70$

Example 1: The Coin-Row Problem

- Let us run the algorithm on the seven coins 20, 10, 20, 50, 20, 10, 20:

function CoinRow($C[1..n]$)
  $S[0] \leftarrow 0$
  $S[1] \leftarrow C[1]$
  for $i \leftarrow 2$ to n **do**
    $S[i] \leftarrow max\{S[i-1], S[i-2] + C[i]\}$
  return $S[n]$

index	0	1	2	3	4	5	6	7
C		20	10	20	50	20	10	20
S	0	20	20	40	70	70		

- $i = 5$
- $S[5] = max(\ S[4] = 70,\ S[3] + 20 = 40 + 20\ ) = 70$

Example 1: The Coin-Row Problem

- Let us run the algorithm on the seven coins 20, 10, 20, 50, 20, 10, 20:

function CoinRow($C[1..n]$)
  $S[0] \leftarrow 0$
  $S[1] \leftarrow C[1]$
  for $i \leftarrow 2$ to n **do**
    $S[i] \leftarrow max\{S[i-1], S[i-2] + C[i]\}$
  return $S[n]$

index	0	1	2	3	4	5	6	7
C		20	10	20	50	20	10	20
S	0	20	20	40	70	70	80	

- $i = 6$
- $S[6] = max(\, S[5] = 70,\ S[4] + 10 = 70 + 10\,) = 80$

Example 1: The Coin-Row Problem

- Let us run the algorithm on the seven coins 20, 10, 20, 50, 20, 10, 20:

function CoinRow($C[1..n]$)
  $S[0] \leftarrow 0$
  $S[1] \leftarrow C[1]$
  for $i \leftarrow 2$ to n **do**
    $S[i] \leftarrow max\{S[i-1], S[i-2] + C[i]\}$
  return $S[n]$

index	0	1	2	3	4	5	6	7
C		20	10	20	50	20	10	20
S	0	20	20	40	70	70	80	90

- $i = 7$
- $S[7] = max(\, S[6] = 80,\ S[5] + 20 = 70 + 20\,) = 90$

Example 1: The Coin-Row Problem

- Let us run the algorithm on the seven coins 20, 10, 20, 50, 20, 10, 20:

function CoinRow($C[1..n]$)
  $S[0] \leftarrow 0$
  $S[1] \leftarrow C[1]$
  for $i \leftarrow 2$ to n **do**
    $S[i] \leftarrow max\{S[i-1], S[i-2] + C[i]\}$
  return $S[n]$

index	0	1	2	3	4	5	6	7
C		20	10	20	50	20	10	20
S	0	20	20	40	70	70	80	90

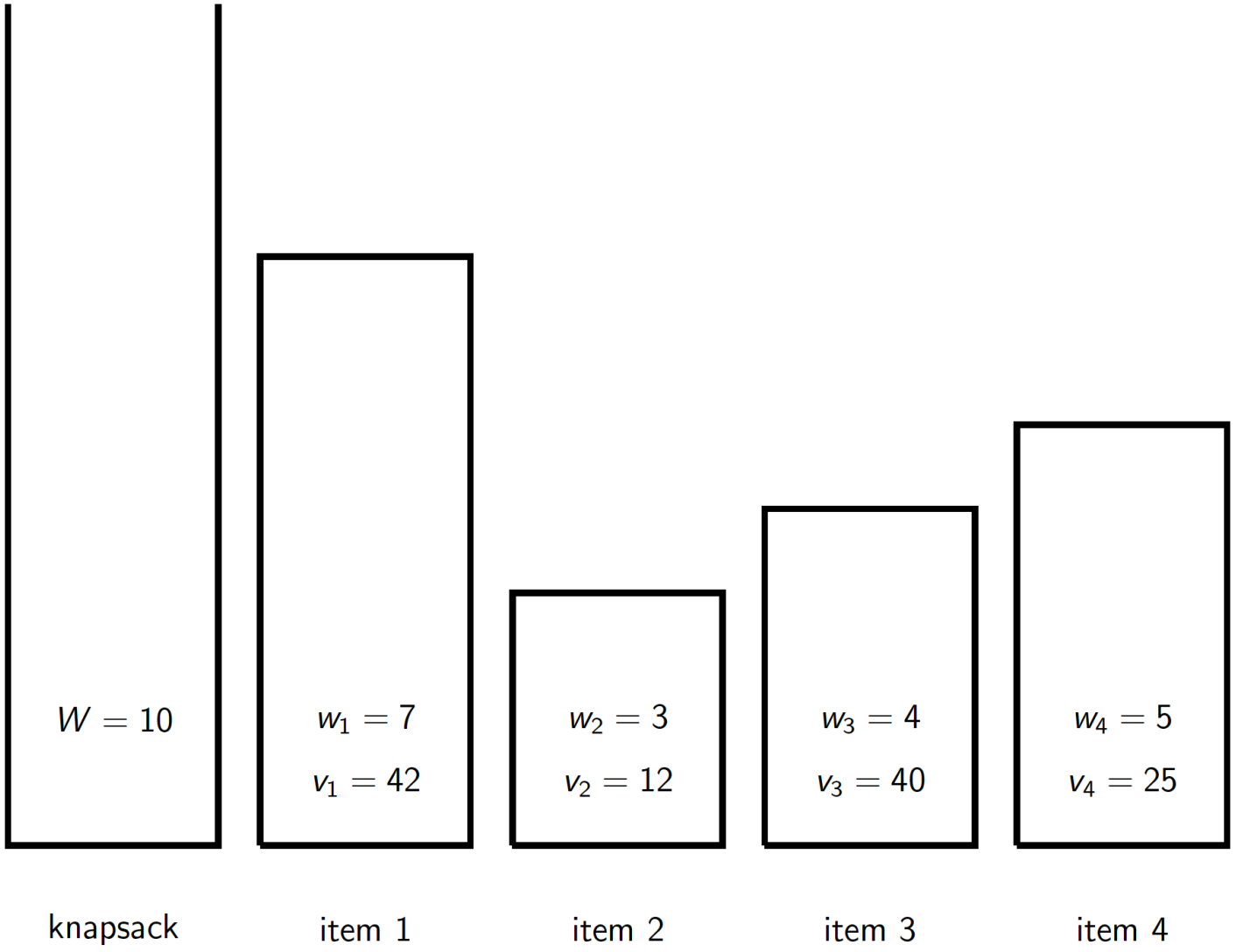
Using		1	1	1	1	1	1	1
				3	4	4	4	4
							6	7

- Keeping trach of the (indices of the) coins used in an optimal solution is an easy extension to the algorithm.

Example 2: The Knapsack Problem

- In Lecture 5 we looked at the knapsack problem.
- Given n items with
  - weights: $w_1, w_2, \cdots, w_n$
  - values: $v_1, v_2, \cdots, v_n$
  - knapsack of weight capacity W.
- Find the most valuable selection of items that will fit in the knapsack.
- We assume that all entities involved are positive integers.

Example 2: The Knapsack Problem



Example 2: The Knapsack Problem

- We previously devised a brute-force algorithm, but dynamic programming may give is a better solution.
- The critical step is to find a good answer to the question "what is the sub-problem?"
- In this case, the trick is to formulate the recurrence relation over two parameters, namely the sequence 1, 2, ⋯, i of item considered so far, and the remaining capacity $w \leq W$.

Example 2: The Knapsack Problem

- We previously devised a brute-force algorithm, but dynamic programming may give is a better solution.
- The critical step is to find a good answer to the question "what is the sub-problem?"
- In this case, the trick is to formulate the recurrence relation over two parameters, namely the sequence 1, 2, ⋯, i of item considered so far, and the remaining capacity $w \leq W$.
- Let $K(i, w)$ by the value of the best choice of items amongst the first i using the knapsack capacity w.
- Then we are after $K(n, W)$.

Example 2: The Knapsack Problem

- The reason we focus on $K(i, w)$ is that we can express a solution to that recursively.
- Amongst the first i items we either pick item i or we don't.

Example 2: The Knapsack Problem

- The reason we focus on $K(i, w)$ is that we can express a solution to that recursively.
- Amongst the first i items we either pick item i or we don't.
- For a solution that excludes the item i, the value of an optimal subset is simply $\mathrm{K}(i-1, w)$.

Example 2: The Knapsack Problem

- The reason we focus on $K(i, w)$ is that we can express a solution to that recursively.
- Amongst the first i items we either pick item i or we don't.
- For a solution that excludes the item i, the value of an optimal subset is simply $\mathrm{K}(i-1, w)$.
- For a solution that includes item i, apart from that item, an optimal solution contains an optimal subset of the first $i-1$ items that will fit into a bag of capacity $\mathrm{w} - w_i$. The value of such a subset is $K(i-1, w-w_i) + v_i$.

Example 2: The Knapsack Problem

- The reason we focus on $K(i, w)$ is that we can express a solution to that recursively.
- Amongst the first i items we either pick item i or we don't.
- For a solution that excludes the item i, the value of an optimal subset is simply $\mathrm{K}(i-1, w)$.
- For a solution that includes item i, apart from that item, an optimal solution contains an optimal subset of the first $i-1$ items that will fit into a bag of capacity $\mathrm{w} - w_i$. The value of such a subset is $K(i-1, w-w_i) + v_i$.
- ⋯ provided item i fits, that is, provided $\mathrm{w} - w_i \geq 0$.

Example 2: The Knapsack Problem

- Now it is easy to express the solution recursively:

$$K(i, w) = 0 \text{ if } i = 0 \text{ or } w = 0$$

- Otherwise:

$$K(i,w) = \begin{cases} max(K(i-1,w), K(i-1,w-w_i) + v_i) & \text{if } w - w_i \geq 0 \\ K(i-1,w) & \text{if } w - w_i < 0 \end{cases}$$

Example 2: The Knapsack Problem

- Now it is easy to express the solution recursively:

$$K(i, w) = 0 \text{ if } i = 0 \text{ or } w = 0$$

- Otherwise:

$$K(i,w) = \begin{cases} max(\, K(i-1,w), K(i-1,w-w_i) + v_i) & \text{if } w \geq w_i \\ K(i-1,w) & \text{if } w < w_i \end{cases}$$

Example 2: The Knapsack Problem

- Now it is easy to express the solution recursively:

$$K(i, w) = 0 \text{ if } i = 0 \text{ or } w = 0$$

- Otherwise:

$$K(i,w) = \begin{cases} max(\, K(i-1,w), K(i-1,w-w_i) + v_i) & \text{if } w \geq w_i \\ K(i-1,w) & \text{if } w < w_i \end{cases}$$

- That gives a correct algorithm for the problem, albeit an inefficient one, if we take it literally.
- For a bottom-up solution we need to write the code that systematically fills a two-dimensional table.
- The table will have $n + 1$ rows and $W + 1$ columns.

Example 2: The Knapsack Problem

- First fill the leftmost column and top row, then proceed row by row:

```
for i ← 0 to n do
    K[i, 0] ← 0
for j ← 1 to W do
    K[0, j] ← 0
for i ← 1 to n do
    for j ← 1 to W do
        if j < w_i then
            K[i, j] ← K[i − 1, j]
        else
            K[i, j] ← max(K[i − 1, j], K[i − 1, j − w_i] + v_i)
return K[n, W]
```

- The algorithm has time (and space) complexity $\Theta(nW)$.

Example 2: The Knapsack Problem

- Here is a concrete example—like before, expect let $W = 8$.

$v_1 = 42 \quad w_1 = 7$
$v_2 = 12 \quad w_2 = 3$
$v_3 = 40 \quad w_3 = 4$
$v_4 = 25 \quad w_4 = 5$

i\j	0	1	2	3	4	5	6	7	8
0									
1									
2									
3									
4									

Example 2: The Knapsack Problem

- Here is a concrete example—like before, expect let $W = 8$.

$v_1 = 42 \quad w_1 = 7$
$v_2 = 12 \quad w_2 = 3$
$v_3 = 40 \quad w_3 = 4$
$v_4 = 25 \quad w_4 = 5$

i\j	0	1	2	3	4	5	6	7	8
0									
1									
2									
3									
4									

for $i \leftarrow 0$ to n **do**
    $K[i, 0] \leftarrow 0$
for $j \leftarrow 1$ to W **do**
    $K[0, j] \leftarrow 0$
for $i \leftarrow 1$ to n **do**
    for $j \leftarrow 1$ to W **do**
        if $j < w_i$ **then**
            $K[i, j] \leftarrow K[i-1, j]$
        else
            $K[i, j] \leftarrow max(K[i-1, j], K[i-1, j-w_i] + v_i)$
return $K[n, W]$

Example 2: The Knapsack Problem

- On the first for loop:

$v_1 = 42 \quad w_1 = 7$

$v_2 = 12 \quad w_2 = 3$

$v_3 = 40 \quad w_3 = 4$

$v_4 = 25 \quad w_4 = 5$

i\j	0	1	2	3	4	5	6	7	8
0	0								
1	0								
2	0								
3	0								
4	0								

```
for i ← 0 to n do
    K[i, 0] ← 0
for j ← 1 to W do
    K[0, j] ← 0
for i ← 1 to n do
    for j ← 1 to W do
        if j < w_i then
            K[i, j] ← K[i − 1, j]
        else
            K[i, j] ← max(K[i − 1, j], K[i − 1, j − w_i] + v_i)
return K[n, W]
```

Example 2: The Knapsack Problem

- On the second for loop:

$v_1 = 42 \quad w_1 = 7$

$v_2 = 12 \quad w_2 = 3$

$v_3 = 40 \quad w_3 = 4$

$v_4 = 25 \quad w_4 = 5$

i\j	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0								
2	0								
3	0								
4	0								

```
for i ← 0 to n do
    K[i, 0] ← 0
for j ← 1 to W do
    K[0, j] ← 0
for i ← 1 to n do
    for j ← 1 to W do
        if j < w_i then
            K[i, j] ← K[i − 1, j]
        else
            K[i, j] ← max(K[i − 1, j], K[i − 1, j − w_i] + v_i)
return K[n, W]
```

Example 2: The Knapsack Problem

- Now we advance row by row:

$v_1 = 42 \quad w_1 = 7$

$v_2 = 12 \quad w_2 = 3$

$v_3 = 40 \quad w_3 = 4$

$v_4 = 25 \quad w_4 = 5$

i\j	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0								
2	0								
3	0								
4	0								

```
for i ← 0 to n do
    K[i, 0] ← 0
for j ← 1 to W do
    K[0, j] ← 0
for i ← 1 to n do
    for j ← 1 to W do
        if j < w_i then
            K[i, j] ← K[i − 1, j]
        else
            K[i, j] ← max(K[i − 1, j], K[i − 1, j − w_i] + v_i)
return K[n, W]
```

Example 2: The Knapsack Problem

- Is the current capacity (j=1) sufficient to fit the first item (i=1)?

$v_1 = 42 \quad w_1 = 7$

$v_2 = 12 \quad w_2 = 3$

$v_3 = 40 \quad w_3 = 4$

$v_4 = 25 \quad w_4 = 5$

i\j	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	?							
2	0								
3	0								
4	0								

for $i \leftarrow 0$ to n **do**
    $K[i, 0] \leftarrow 0$
for $j \leftarrow 1$ to W **do**
    $K[0, j] \leftarrow 0$
for $i \leftarrow 1$ to n **do**
    for $j \leftarrow 1$ to W **do**
        if $j < w_i$ **then**
            $K[i, j] \leftarrow K[i-1, j]$
        else
            $K[i, j] \leftarrow max(K[i-1, j], K[i-1, j-w_i] + v_i)$
return $K[n, W]$

Example 2: The Knapsack Problem

- We won't have enough capacity until j=7:

$v_1 = 42 \quad w_1 = 7$
$v_2 = 12 \quad w_2 = 3$
$v_3 = 40 \quad w_3 = 4$
$v_4 = 25 \quad w_4 = 5$

i\j	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0		
2	0								
3	0								
4	0								

```
for i ← 0 to n do
    K[i, 0] ← 0
for j ← 1 to W do
    K[0, j] ← 0
for i ← 1 to n do
    for j ← 1 to W do
        if j < w_i then
            K[i, j] ← K[i − 1, j]
        else
            K[i, j] ← max(K[i − 1, j], K[i − 1, j − w_i] + v_i)
return K[n, W]
```

- i=1
- j=7

Example 2: The Knapsack Problem

- We won't have enough capacity until j=7:

$v_1 = 42 \quad w_1 = 7$
$v_2 = 12 \quad w_2 = 3$
$v_3 = 40 \quad w_3 = 4$
$v_4 = 25 \quad w_4 = 5$

i\j	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0		
2	0								
3	0								
4	0								

```
for i ← 0 to n do
    K[i, 0] ← 0
for j ← 1 to W do
    K[0, j] ← 0
for i ← 1 to n do
    for j ← 1 to W do
        if j < w_i then
            K[i, j] ← K[i − 1, j]
        else
            K[i, j] ← max(K[i − 1, j], K[i − 1, j − w_i] + v_i)
return K[n, W]
```

- i=1
- j=7
- $K[1-1,7]=K[0,7]=0$

Example 2: The Knapsack Problem

- We won't have enough capacity until j=7:

$v_1 = 42 \quad w_1 = 7$
$v_2 = 12 \quad w_2 = 3$
$v_3 = 40 \quad w_3 = 4$
$v_4 = 25 \quad w_4 = 5$

i\j	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0		
2	0								
3	0								
4	0								

```
for i ← 0 to n do
    K[i, 0] ← 0
for j ← 1 to W do
    K[0, j] ← 0
for i ← 1 to n do
    for j ← 1 to W do
        if j < w_i then
            K[i, j] ← K[i − 1, j]
        else
            K[i, j] ← max(K[i − 1, j], K[i − 1, j − w_i] + v_i)
return K[n, W]
```

- i=1
- j=7
- $K[1-1,7]=K[0,7]=0$
- $K[1-1,7-7]+42=K[0,0]+42=0+42=42$

Example 2: The Knapsack Problem

- We won't have enough capacity until j=7:

$v_1 = 42 \quad w_1 = 7$
$v_2 = 12 \quad w_2 = 3$
$v_3 = 40 \quad w_3 = 4$
$v_4 = 25 \quad w_4 = 5$

i\j	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	42	
2	0								
3	0								
4	0								

for $i \leftarrow 0$ to n **do**
  $K[i,0] \leftarrow 0$
for $j \leftarrow 1$ to W **do**
  $K[0,j] \leftarrow 0$
for $i \leftarrow 1$ to n **do**
  for $j \leftarrow 1$ to W **do**
    if $j < w_i$ **then**
      $K[i,j] \leftarrow K[i-1,j]$
    else
      $K[i,j] \leftarrow max(K[i-1,j], K[i-1,j-w_i] + v_i)$
return $K[n,W]$

- i=1
- j=7
- $K[1-1,7]=K[0,7] = 0$
- $K[1-1,7-7] + 42=K[0,0] + 42 = 0 + 42 = 42$
- $K[1,7] = \max(0,42) = 42$

Example 2: The Knapsack Problem

- There are no more items to pack, then K[1,8] = K[1,7]:

$v_1 = 42 \quad w_1 = 7$
$v_2 = 12 \quad w_2 = 3$
$v_3 = 40 \quad w_3 = 4$
$v_4 = 25 \quad w_4 = 5$

i\j	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	42	42
2	0								
3	0								
4	0								

for $i \leftarrow 0$ to n **do**
  $K[i,0] \leftarrow 0$
for $j \leftarrow 1$ to W **do**
  $K[0,j] \leftarrow 0$
for $i \leftarrow 1$ to n **do**
  for $j \leftarrow 1$ to W **do**
    if $j < w_i$ **then**
      $K[i,j] \leftarrow K[i-1,j]$
    else
      $K[i,j] \leftarrow max(K[i-1,j], K[i-1,j-w_i] + v_i)$
return $K[n,W]$

- i=1
- j=8
- $K[1-1,8]=K[0,8] = 0$
- $K[1-1,8-7] + 42=K[0,1] + 42 = 0 + 42 = 42$
- $K[1,8] = max(0,42) = 42$

Example 2: The Knapsack Problem

- Next row, we won't have enough capacity until j = 3:

$v_1 = 42 \quad w_1 = 7$
$v_2 = 12 \quad w_2 = 3$
$v_3 = 40 \quad w_3 = 4$
$v_4 = 25 \quad w_4 = 5$

i\j	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	42	42
2	0	0	0	12					
3	0								
4	0								

for $i \leftarrow 0$ to n **do**
  $K[i, 0] \leftarrow 0$
for $j \leftarrow 1$ to W **do**
  $K[0, j] \leftarrow 0$
for $i \leftarrow 1$ to n **do**
  for $j \leftarrow 1$ to W **do**
    if $j < w_i$ **then**
      $K[i, j] \leftarrow K[i-1, j]$
    else
      $K[i, j] \leftarrow max(K[i-1, j], K[i-1, j-w_i] + v_i)$
return $K[n, W]$

- i=2
- j=3
- $K[2-1,3]=K[1,3] = 0$
- $K[2-1,3-3] + 12=K[1,0] + 12 = 0 + 12 = 12$
- $K[2,3] = \max(0,12) = 12$

Example 2: The Knapsack Problem

- At j = 7, it is better to pick 42:

$v_1 = 42 \quad w_1 = 7$
$v_2 = 12 \quad w_2 = 3$
$v_3 = 40 \quad w_3 = 4$
$v_4 = 25 \quad w_4 = 5$

i\j	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	42	42
2	0	0	0	12	12	12	12	42	
3	0								
4	0								

```
for i ← 0 to n do
    K[i, 0] ← 0
for j ← 1 to W do
    K[0, j] ← 0
for i ← 1 to n do
    for j ← 1 to W do
        if j < w_i then
            K[i, j] ← K[i − 1, j]
        else
            K[i, j] ← max(K[i − 1, j], K[i − 1, j − w_i] + v_i)
return K[n, W]
```

- i=2
- j=7
- $K[2-1,7]=K[1,7]=42$
- $K[2-1,7-3]+12=K[1,4]+12=0+12=12$
- $K[2,7]=\max(42,12)=42$

Example 2: The Knapsack Problem

- Next row, at j = 4, it is better to pick 40:

$v_1 = 42 \quad w_1 = 7$
$v_2 = 12 \quad w_2 = 3$
$v_3 = 40 \quad w_3 = 4$
$v_4 = 25 \quad w_4 = 5$

i\j	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	42	42
2	0	0	0	12	12	12	12	42	42
3	0	0	0	12	40				
4	0								

```
for i ← 0 to n do
    K[i, 0] ← 0
for j ← 1 to W do
    K[0, j] ← 0
for i ← 1 to n do
    for j ← 1 to W do
        if j < w_i then
            K[i, j] ← K[i − 1, j]
        else
            K[i, j] ← max(K[i − 1, j], K[i − 1, j − w_i] + v_i)
return K[n, W]
```

- i=3
- j=4
- $K[3-1,4]=K[2,4]=12$
- $K[3-1,4-4]+40=K[2,0]+40=0+40=40$
- $K[3,4]=\max(12,40)=40$

Example 2: The Knapsack Problem

- What happens at j = 7?

$v_1 = 42 \quad w_1 = 7$
$v_2 = 12 \quad w_2 = 3$
$v_3 = 40 \quad w_3 = 4$
$v_4 = 25 \quad w_4 = 5$

i\j	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	42	42
2	0	0	0	12	12	12	12	42	42
3	0	0	0	12	40	40	40	52	
4	0								

for $i \leftarrow 0$ to n **do**
  $K[i, 0] \leftarrow 0$
for $j \leftarrow 1$ to W **do**
  $K[0, j] \leftarrow 0$
for $i \leftarrow 1$ to n **do**
  for $j \leftarrow 1$ to W **do**
    if $j < w_i$ **then**
      $K[i, j] \leftarrow K[i-1, j]$
    else
      $K[i, j] \leftarrow max(K[i-1, j], K[i-1, j-w_i] + v_i)$
return $K[n, W]$

- i=3
- j=7
- $K[3-1,7]=K[2,7] = 42$
- $K[3-1,7-4] + 40=K[2,3] + 40 = 12 + 40 = 52$
- $K[3,7] = max(42,52) = 52$

Example 2: The Knapsack Problem

- At the end, the best solution found is K[4,8]=52:

$v_1 = 42 \quad w_1 = 7$
$v_2 = 12 \quad w_2 = 3$
$v_3 = 40 \quad w_3 = 4$
$v_4 = 25 \quad w_4 = 5$

i\j	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	42	42
2	0	0	0	12	12	12	12	42	42
3	0	0	0	12	40	40	40	52	52
4	0	0	0	12	40	40	40	52	52

for $i \leftarrow 0$ to n **do**
  $K[i,0] \leftarrow 0$
for $j \leftarrow 1$ to W **do**
  $K[0,j] \leftarrow 0$
for $i \leftarrow 1$ to n **do**
  for $j \leftarrow 1$ to W **do**
    if $j < w_i$ **then**
      $K[i,j] \leftarrow K[i-1,j]$
    else
      $K[i,j] \leftarrow max(K[i-1,j], K[i-1,j-w_i] + v_i)$
return $K[n,W]$

- i=4
- j=8
- $K[4-1,8]=K[3,8] = 52$
- $K[4-1,8-5] + 25=K[3,3] + 25 = 12 + 25 = 37$
- $K[4,8] = max(52,37) = 52$

Solving the Knapsack Problem with MEmoing

- To some extent the bottom-up (table-filling) solution is overkill: it finds the solution to every conceivable sub-instance.
- Most of the table entries cannot actually contribute to a solution. For a clearer example of this, let all the weights in the problem be multiplied by 1000.
- In this situation, a top-down approach, with memoing, is preferable.
- To keep the memo table small, make it a hash table.

Solving the Knapsack Problem with Memoing

- The hash table uses keys (i, j). These are stored together with their corresponding values k = K(i, j).

```
function KNAP(i,j)
    if i = 0 or j = 0 then
        return 0
    if key (i, j) is in hashtable then
        return the corresponding value (that is, K(i, j))
    if j < w_i then
        k ← KNAP(i − 1, j)
    else
        k ← max(KNAP(i − 1, j), KNAP(i − 1, j − w_i) + v_i)
    insert k into hashtable, with key (i, j)
    return k
```

Coming Up Next

- We apply dynamic programming to two graph problems (transitive closure and all-pairs shortest-paths); the resulting algorithms are known as Warshall's and Floyd's.