COMP90038
Algorithms and Complexity

Lecture 18: Dynamic Programming

(with thanks to Harald Søndergaard & Michael Kirley)

Andres Munoz-Acosta

munoz.m@unimelb.edu.au

Peter Hall Building G.83

On the previous lecture

- We talked about **hashing**, a standard approach to implement a dictionary

On the previous lecture

- We talked about **hashing**, a standard approach to implement a dictionary

- Each record in a **hash table** is identified by a **key**. They key is the input to a **hash function** which generates the **address** of the record in the table

On the previous lecture

- We talked about **hashing**, a standard approach to implement a dictionary

- Each record in a **hash table** is identified by a **key**. They key is the input to a **hash function** which generates the **address** of the record in the table

- The challenges in implementing a hash table are:
  - Designing a **robust** hash function
  - Handling of **collisions**, i.e., when two different records have the same address

On the previous lecture

- We talked about **hashing**, a standard approach to implement a dictionary

- Each record in a **hash table** is identified by a **key**. They key is the input to a **hash function** which generates the **address** of the record in the table

- The challenges in implementing a hash table are:
  - Designing a **robust** hash function
  - Handling of **collisions**, i.e., when two different records have the same address

- We described **Horner's rule**, a simple trick to simplify polynomial calculations

On the previous lecture

- We talked about **hashing**, a standard approach to implement a dictionary

- Each record in a **hash table** is identified by a **key**. They key is the input to a **hash function** which generates the **address** of the record in the table

- The challenges in implementing a hash table are:
  - Designing a **robust** hash function
  - Handling of **collisions**, i.e., when two different records have the same address

- We described **Horner's rule**, a simple trick to simplify polynomial calculations

On the previous lecture

- We talked about **hashing**, a standard approach to implement a dictionary
- Each record in a **hash table** is identified by a **key**. They key is the input to a **hash function** which generates the **address** of the record in the table
- The challenges in implementing a hash table are:
  - Designing a **robust** hash function
  - Handling of **collisions**, i.e., when two different records have the same address
- We described **Horner’s rule**, a simple trick to simplify polynomial calculations
- We also discussed the **Rabin-Karp algorithm**, a string matching method that uses hashing to identify matches

Which one of the following statements is true:

The load factor in separate chaining tends to be less than one

The load factor on linear probing must be close to one to guarantee efficiency

Clustering is a common problem in double hashing

Deletion in linear probing is much easier than in separate chaining

Today’s lecture

- **Dynamic programming** is a bottom-up problem solving technique, somewhat similar to divide-and-conquer

Today’s lecture

- **Dynamic programming** is a bottom-up problem solving technique, somewhat similar to divide-and-conquer

- We divide the problem into **smaller**, albeit **interdependent**, problems

Today’s lecture

- **Dynamic programming** is a bottom-up problem solving technique, somewhat similar to divide-and-conquer

- We divide the problem into **smaller**, albeit **interdependent**, problems
  - In divide-and-conquer, the subproblems are **independent** of each other

Today's lecture

- **Dynamic programming** is a bottom-up problem solving technique, somewhat similar to divide-and-conquer

- We divide the problem into **smaller**, albeit **interdependent**, problems
  - In divide-and-conquer, the subproblems are **independent** of each other

- Because of their dependencies, **intermediate results are stored** and used to find the complete solution

Today’s lecture

- **Dynamic programming** is a bottom-up problem solving technique, somewhat similar to divide-and-conquer

- We divide the problem into **smaller**, albeit **interdependent**, problems
  - In divide-and-conquer, the subproblems are **independent** of each other

- Because of their dependencies, **intermediate results are stored** and used to find the complete solution
  - That does not happen in divide-and-conquer
  - For example, think about MergeSort for a moment. Do you keep the solution from one branch to be re-used in another?

Dynamic programming

- For example, in **Lecture 16** we examined this algorithm that used tabulated results to find the Fibonacci numbers

$$
\begin{array}{l}
\textbf{function } \textsc{Fib}(n) \\
\quad \textbf{if } n = 0 \textbf{ or } n = 1 \textbf{ then} \\
\quad\quad \textbf{return } 1 \\
\quad x \leftarrow F[n] \\
\quad \textbf{if } x = 0 \textbf{ then} \\
\quad\quad x \leftarrow \textsc{Fib}(n-1) + \textsc{Fib}(n-2) \\
\quad\quad F[n] \leftarrow x \\
\quad \textbf{return } x
\end{array}
$$

Dynamic programming

- For example, in **Lecture 16** we examined this algorithm that used tabulated results to find the Fibonacci numbers

- Note that:
  - F[0...n] is an **array** that stores **partial** results, initialized to zero

$$
\begin{array}{l}
\textbf{function } \textsc{Fib}(n) \\
\quad \textbf{if } n = 0 \textbf{ or } n = 1 \textbf{ then} \\
\quad\quad \textbf{return } 1 \\
\quad x \leftarrow F[n] \\
\quad \textbf{if } x = 0 \textbf{ then} \\
\quad\quad x \leftarrow \textsc{Fib}(n-1) + \textsc{Fib}(n-2) \\
\quad\quad F[n] \leftarrow x \\
\quad \textbf{return } x
\end{array}
$$

Dynamic programming

- For example, in **Lecture 16** we examined this algorithm that used tabulated results to find the Fibonacci numbers

- Note that:
  - F[0...n] is an **array** that stores **partial** results, initialized to zero
  - If F[n]=0, then this partial result has not been calculated yet, hence follow the **recursion**

$$
\begin{array}{l}
\textbf{function } \textsc{Fib}(n) \\
\quad \textbf{if } n = 0 \textbf{ or } n = 1 \textbf{ then} \\
\quad\quad \textbf{return } 1 \\
\quad x \leftarrow F[n] \\
\quad \textbf{if } x = 0 \textbf{ then} \\
\quad\quad x \leftarrow \textsc{Fib}(n-1) + \textsc{Fib}(n-2) \\
\quad\quad F[n] \leftarrow x \\
\quad \textbf{return } x
\end{array}
$$

Dynamic programming

- For example, in **Lecture 16** we examined this algorithm that used tabulated results to find the Fibonacci numbers

- Note that:
  - F[0...n] is an **array** that stores **partial** results, initialized to zero
  - If F[n]=0, then this partial result has not been calculated yet, hence follow the **recursion**
  - If F[n]≠0, then this value has been calculated and we can use it.

$$
\begin{array}{l}
\textbf{function } \textsc{Fib}(n) \\
\quad \textbf{if } n = 0 \textbf{ or } n = 1 \textbf{ then} \\
\quad\quad \textbf{return } 1 \\
\quad x \leftarrow F[n] \\
\quad \textbf{if } x = 0 \textbf{ then} \\
\quad\quad x \leftarrow \textsc{Fib}(n-1) + \textsc{Fib}(n-2) \\
\quad\quad F[n] \leftarrow x \\
\quad \textbf{return } x
\end{array}
$$

Dynamic programming

- Dynamic programming is often used on **combinatorial optimization** problems
  - The objective is to find the **best** possible **combination**, i.e., the one with the lowest cost or highest profit, subject to some **constraints**

Dynamic programming

- Dynamic programming is often used on **combinatorial optimization** problems
  - The objective is to find the **best** possible **combination**, i.e., the one with the lowest cost or highest 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 true, this principle holds often

Dynamic programming

- Constructing DP algorithms is often **tricky**. Hence, they are best developed in stages:

1. **Formulate** the problem recursively
    - This is often the hard part

Dynamic programming

- Constructing DP algorithms is often **tricky**. Hence, they are best developed in stages:

1. **Formulate** the problem recursively
   - This is often the hard part

2. Build solutions to your **recurrence** from the bottom up
   - Write an algorithm that **starts with the base cases** and works the way up the recursion to the final solution

Dynamic programming

- Constructing DP algorithms is often **tricky**. Hence, they are best developed in stages:

1. **Formulate** the problem recursively
   - This is often the hard part

2. Build solutions to your **recurrence** from the bottom up
   - Write an algorithm that **starts with the base cases** and works the way up the recursion to the final solution

- These stages can be further divided in smaller steps

Dynamic programming

- Stage 1: Formulation
  - **a) Describe the problem** that you want to solve recursively in coherent and precise language

Dynamic programming

- Stage 1: Formulation
  a) **Describe the problem** that you want to solve recursively in coherent and precise language
  b) Construct a **recursive formula** for the whole problem in terms of answers to smaller instances of **exactly** the same problem

Dynamic programming

- Stage 1: Formulation
  a) **Describe the problem** that you want to solve recursively in coherent and precise language
  b) Construct a **recursive formula** for the whole problem in terms of answers to smaller instances of **exactly** the same problem

- Stage 2: Algorithm development
  a) **Identify the subproblems**: What are all the different ways that your recursive algorithm can call itself, starting with an initial input?

Dynamic programming

- Stage 1: Formulation
  a) **Describe the problem** that you want to solve recursively in coherent and precise language
  b) Construct a **recursive formula** for the whole problem in terms of answers to smaller instances of **exactly** the same problem

- Stage 2: Algorithm development
  a) **Identify the subproblems**: What are all the different ways that your recursive algorithm can call itself, starting with an initial input?
  b) **Choose a memoization data structure**: Find a data structure that can store the solution to every subproblem identified before

Dynamic programming

- Stage 1: Formulation
  a) **Describe the problem** that you want to solve recursively in coherent and precise language
  b) Construct a **recursive formula** for the whole problem in terms of answers to smaller instances of **exactly** the same problem

- Stage 2: Algorithm development
  a) **Identify the subproblems**: What are all the different ways that your recursive algorithm can call itself, starting with an initial input?
  b) **Choose a memoization data structure**: Find a data structure that can store the solution to every subproblem identified before
  c) **Identify dependencies**: Which problems depend on other subproblems?

Dynamic programming

- Stage 2 (continuation)
  - **d) Find a good evaluation order**: Order the subproblems so each one comes after the subproblem it depends on

Dynamic programming

- Stage 2 (continuation)
  - **d) Find a good evaluation order**: Order the subproblems so each one comes after the subproblem it depends on
  - **e) Analyse space and running time**: To compute the total running time, add up the running times of all possible subproblems, assuming that deeper recursive calls are already memoized

Dynamic programming

- Stage 2 (continuation)
  - **d) Find a good evaluation order**: Order the subproblems so each one comes after the subproblem it depends on
  - **e) Analyse space and running time**: To compute the total running time, add up the running times of all possible subproblems, assuming that deeper recursive calls are already memoized
  - **f) Write down the algorithm**

Dynamic programming

- Stage 2 (continuation)
  - **d) Find a good evaluation order**: Order the subproblems so each one comes after the subproblem it depends on
  - **e) Analyse space and running time**: To compute the total running time, add up the running times of all possible subproblems, assuming that deeper recursive calls are already memoized
  - **f) Write down the algorithm**

- We will observe some of these steps while we work through some example problems
  - The coin row problem
  - The knapsack problem
  - Message passing in a tree problem

The coin row problem

- You are shown a **group of coins** of different denominations ordered in a row

The coin row problem

- You are shown a **group of coins** of different denominations ordered in a row
- **You can keep some of them**, as long as you **do not pick two adjacent ones**

The coin row problem

- You are shown a **group of coins** of different denominations ordered in a row
- **You can keep some of them**, as long as you **do not pick two adjacent ones**
  - Your objective is to **maximize your profit** , i.e., you want to take the largest amount of money

The coin row problem

- Let's visualize the problem. Our coins are [20 10 20 50 20 10 20]

The coin row problem

- We cannot take these two.

The coin row problem

- We cannot take these two.
  - It does not fulfil our constraint (We cannot pick adjacent coins)

The coin row problem

- We could take all the 20s (Total of 80).

The coin row problem

- We could take all the 20s (Total of 80).
  - Is that the maximum profit? Is this a greedy solution?

The coin row problem

- Let’s think of a **recursion** that help us solve this problem? What is the smallest problem possible?

The coin row problem

- Let’s think of a **recursion** that help us solve this problem? What is the smallest problem possible?

- If instead of a row of seven coins we only had one coin
  - We have only one choice

The coin row problem

- Let’s think of a **recursion** that help us solve this problem? What is the smallest problem possible?

- If instead of a row of seven coins we only had one coin
  - We have only one choice

- What about if we had a row of two?
  - We either pick the first or second coin

The coin row problem

- If we have a row of three, we can pick the middle coin or the two in the sides. Which one is the optimal?

The coin row problem

- If we had a row of four, there are sixteen combinations

The coin row problem

- If we had a row of four, there are sixteen combinations
- For simplicity, I represent these combinations as binary strings:
  - ‘0’ = leave the coin
  - ‘1’ = pick the coin

0	0000	
1	0001	
2	0010	
3	0011	
4	0100	
5	0101	
6	0110	
7	0111	
8	1000	
9	1001	
10	1010	
11	1011	
12	1100	
13	1101	
14	1110	
15	1111	

The coin row problem

- If we had a row of four, there are sixteen combinations
- For simplicity, I represent these combinations as binary strings:
  - ‘0’ = leave the coin
  - ‘1’ = pick the coin
- Eight of them are not valid (in optimization lingo **unfeasible**), one has the worst profit (0)

0	0000	PICK NOTHING (NO PROFIT)
1	0001	
2	0010	
3	0011	UNFEASIBLE
4	0100	
5	0101	
6	0110	UNFEASIBLE
7	0111	UNFEASIBLE
8	1000	
9	1001	
10	1010	
11	1011	UNFEASIBLE
12	1100	UNFEASIBLE
13	1101	UNFEASIBLE
14	1110	UNFEASIBLE
15	1111	UNFEASIBLE

The coin row problem

- If we had a row of four, there are sixteen combinations
- For simplicity, I represent these combinations as binary strings:
  - ‘0’ = leave the coin
  - ‘1’ = pick the coin
- Eight of them are not valid (in optimization lingo **unfeasible**), one has the worst profit (0)
- Picking one coin will always lead to lower profit (in optimization lingo **suboptimal**)

0	0000	PICK NOTHING (NO PROFIT)
1	0001	SUBOPTIMAL
2	0010	SUBOPTIMAL
3	0011	UNFEASIBLE
4	0100	SUBOPTIMAL
5	0101	
6	0110	UNFEASIBLE
7	0111	UNFEASIBLE
8	1000	SUBOPTIMAL
9	1001	
10	1010	
11	1011	UNFEASIBLE
12	1100	UNFEASIBLE
13	1101	UNFEASIBLE
14	1110	UNFEASIBLE
15	1111	UNFEASIBLE

The coin row problem

- Let's give the coins values $[c_1\ c_2\ c_3\ c_4]$, and focus on the **feasible** combinations:
  - Our choice is to pick two coins $[c_1\ 0\ c_3\ 0]$ $[0\ c_2\ 0\ c_4]$ $[c_1\ 0\ 0\ c_4]$

The coin row problem

- Let's give the coins values $[c_1\ c_2\ c_3\ c_4]$, and focus on the **feasible** combinations:
  - Our choice is to pick two coins $[c_1\ 0\ c_3\ 0]$ $[0\ c_2\ 0\ c_4]$ $[c_1\ 0\ 0\ c_4]$

- If the coins arrived in sequence, when we reach c_4, the best that we can do is either:
  - Take a solution at step 3 $[c_1\ 0\ c_3\ 0]$
  - Add to one of the solutions at step 2 the new coin: $[0\ c_2\ 0\ c_4]$ $[c_1\ 0\ 0\ c_4]$

The coin row problem

- Let's give the coins values $[c_1\ c_2\ c_3\ c_4]$, and focus on the **feasible** combinations:
  - Our choice is to pick two coins $[c_1\ 0\ c_3\ 0]$ $[0\ c_2\ 0\ c_4]$ $[c_1\ 0\ 0\ c_4]$

- If the coins arrived in sequence, when we reach c_4, the best that we can do is either:
  - Take a solution at step 3 $[c_1\ 0\ c_3\ 0]$
  - Add to one of the solutions at step 2 the new coin: $[0\ c_2\ 0\ c_4]$ $[c_1\ 0\ 0\ c_4]$

- Generally, we can express this as the recurrence:

$$S(n) = \max\left(c_n + S\left(n-2\right), S\left(n-1\right)\right) \text{ for } n > 1$$

$$S(1) = c_1$$

$$S(0) = 0$$

The coin row problem

- Given that we have to backtrack to $S(0)$ and $S(1)$, we store these results in an array

- Then the algorithm is:

$$
\begin{array}{l}
\textbf{function } \textsc{CoinRow}(C[\cdot], n) \\
\quad S[0] \leftarrow 0 \\
\quad S[1] \leftarrow C[1] \\
\quad \textbf{for } i \leftarrow 2 \text{ to } n \textbf{ do} \\
\quad\quad S[i] \leftarrow \max(S[i-1], S[i-2] + C[i]) \\
\quad \textbf{return } S[n]
\end{array}
$$

The coin row problem

- Lets run our algorithm in the example. $i=0$

- $S[0] = 0$

The coin row problem

- i=1

- $S[1] = 20$

The coin row problem

- i=2

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

The coin row problem

- i=3

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

The coin row problem

- i=4

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

The coin row problem

- At i=5, we can pick between:
  - **$S[4] = 70$**
  - $S[3] + 20 = 60$

i	0	1	2	3	4	5	6	7
$C[.]$	0	20	10	20	50	20	10	20
$S[.]$	0							
	0	20						
	0	20	20					
	0	20	20	40				
	0	20	20	40	70			
	0	20	20	40	70	70		

The coin row problem

- At i=5, we can pick between:
  - ***S*[4] = 70**
  - $S[3] + 20 = 60$
- At i=6, we can pick between:
  - $S[5] = 70$
  - ***S*[4] + 10 = 80**

i	0	1	2	3	4	5	6	7
$C[.]$	0	20	10	20	50	20	10	20
$S[.]$	0							
	0	20						
	0	20	20					
	0	20	20	40				
	0	20	20	40	70			
	0	20	20	40	70	70		
	0	20	20	40	70	70	80	

The coin row problem

- At i=5, we can pick between:
  - **$S[4] = 70$**
  - $S[3] + 20 = 60$
- At i=6, we can pick between:
  - $S[5] = 70$
  - **$S[4] + 10 = 80$**
- At i=7, we can pick between:
  - $S[6] = 80$
  - **$S[5] + 20 = 90$**

i	0	1	2	3	4	5	6	7
$C[.]$	0	20	10	20	50	20	10	20
$S[.]$	0							
	0	20						
	0	20	20					
	0	20	20	40				
	0	20	20	40	70			
	0	20	20	40	70	70		
	0	20	20	40	70	70	80	
	0	20	20	40	70	70	80	90

The coin row problem

- In a sense, DP allows us to review our solutions **considering newly arrived information**

The coin row problem

- In a sense, DP allows us to review our solutions **considering newly arrived information**
  - If we used a brute-force approach such as **exhaustive search**, we had to test 33 feasible combinations

The coin row problem

- In a sense, DP allows us to review our solutions **considering newly arrived information**
  - If we used a brute-force approach such as **exhaustive search**, we had to test 33 feasible combinations
  - Instead we tested 5 combinations

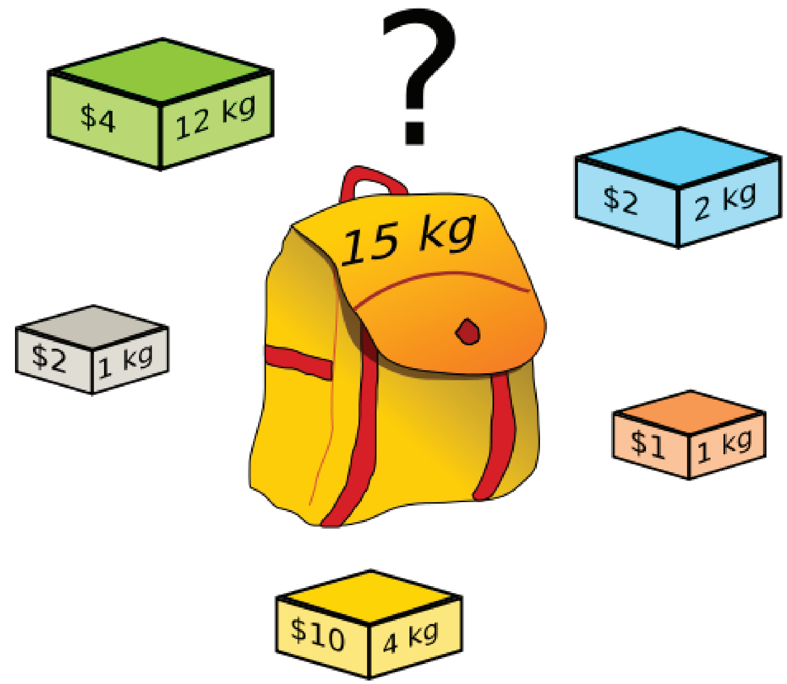
The knapsack problem

- In Lecture 5 you encountered the **knapsack problem**:

- Given a list of n items with:
  - Weights $\{w_1, w_2, \ldots, w_n\}$
  - Values $\{v_1, v_2, \ldots, v_n\}$
- and a knapsack (container) of capacity W

- Find the **combination** of items with the **highest value** that would **fit into the knapsack**

- All variables are positive integers



The knapsack problem

- This is another combinatorial optimization problem:
  - In both the coin row and knapsack problems, we are **maximizing profit**
  - Unlike the coin row problem which had **one variable** <coin value>, we now have **two variables** <item weight, item value>

The knapsack problem

- The critical step is answer to the question: **what is the smallest version of the problem that I could solve?**

The knapsack problem

- The critical step is answer to the question: **what is the smallest version of the problem that I could solve?**
  - If I have a knapsack of capacity 1, and an item of weight 2. **Does it fit?**

The knapsack problem

- The critical step is answer to the question: **what is the smallest version of the problem that I could solve?**
  - If I have a knapsack of capacity 1, and an item of weight 2. **Does it fit?**
  - If the capacity was 2 and the weight 1. Does it fit? **Do I have capacity left?**

The knapsack problem

- The critical step is answer to the question: **what is the smallest version of the problem that I could solve?**
  - If I have a knapsack of capacity 1, and an item of weight 2. **Does it fit?**
  - If the capacity was 2 and the weight 1. Does it fit? **Do I have capacity left?**

- Given that we have **two variables**, the recurrence relation is formulated over **two parameters**:
  - the **sequence of items considered so far** $\{1, 2, \dots i\}$, and
  - the **remaining capacity** $w \leq W$.

The knapsack problem

- The critical step is answer to the question: **what is the smallest version of the problem that I could solve?**
  - If I have a knapsack of capacity 1, and an item of weight 2. **Does it fit?**
  - If the capacity was 2 and the weight 1. Does it fit? **Do I have capacity left?**

- Given that we have **two variables**, the recurrence relation is formulated over **two parameters**:
  - the **sequence of items considered so far** $\{1, 2, \ldots i\}$, and
  - the **remaining capacity** $w \leq W$.

- Let $K(i,w)$ be the value of the best choice of items amongst the first i using knapsack capacity *w*.
  - Then we are after $K(n,W)$.

The knapsack problem

- By focusing on $K(i,w)$ we can express a recursive solution

- Once a new item i arrives, we can either pick it or not.

The knapsack problem

- By focusing on $K(i,w)$ we can express a recursive solution

- Once a new item i arrives, we can either pick it or not.
  - **Excluding *i*** means that the solution is $K(i\text{-}1,w)$, that is, which items were selected before i arrived with the same knapsack capacity.

  - **Including *i*** means that the solution also includes the subset of previous items **that will fit into a bag of capacity $\boldsymbol{w\text{-}w_{\mathrm{i}}}$** ≥ 0, i.e., $K(i\text{-}1,w\text{-}w_i) + v_i$.

The knapsack problem

- Let us express this as a recursive function

The knapsack problem

- Let us express this as a recursive function

- First the base **state**:

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

The knapsack problem

- Let us express this as a recursive function

- First the base **state**:

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

- Otherwise:

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

The knapsack problem

- This results in a correct, but inefficient algorithm
  - It fills systematically a **two-dimensional table** of n+1 rows and W+1 columns
  - As result it has both time and space complexity of $O(nW)$
  - This is known as a **bottom-up** solution

$$
\begin{array}{l}
\textbf{for } i \leftarrow 0 \text{ to } n \textbf{ do} \\
\quad K[i,0] \leftarrow 0 \\
\textbf{for } j \leftarrow 1 \text{ to } W \textbf{ do} \\
\quad K[0,j] \leftarrow 0 \\
\textbf{for } i \leftarrow 1 \text{ to } n \textbf{ do} \\
\quad \textbf{for } j \leftarrow 1 \text{ to } W \textbf{ do} \\
\quad\quad \textbf{if } j < w_i \textbf{ then} \\
\quad\quad\quad K[i,j] \leftarrow K[i-1,j] \\
\quad\quad \textbf{else} \\
\quad\quad\quad K[i,j] \leftarrow \max(K[i-1,j], K[i-1,j-w_i]+v_i) \\
\textbf{return } K[n,W]
\end{array}
$$

The knapsack problem

- Lets look at the algorithm, step-by-step

- The data is:
  - The knapsack capacity $W = 8$
  - The values are $\{42, 12, 40, 25\}$
  - The weights are $\{7, 3, 4, 5\}$

The knapsack problem

- On the first **for loop**:

			j	0	1	2	3	4	5	6	7	8
v	w	i										
		0		0								
42	7	1		0								
12	3	2		0								
40	4	3		0								
25	5	4		0								

$$\boxed{\begin{array}{l}\textbf{for } i \leftarrow 0 \text{ to } n \textbf{ do}\\ \quad K[i,0] \leftarrow 0\end{array}}$$

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 $K[n,W]$

The knapsack problem

- On the second **for loop**:

			j	0	1	2	3	4	5	6	7	8
v	w	i										
		0		0	0	0	0	0	0	0	0	0
42	7	1		0								
12	3	2		0								
40	4	3		0								
25	5	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 $K[n, W]$

The knapsack problem

- Now we advance row by row:

			j	0	1	2	3	4	5	6	7	8
v	w	i										
		0		0	0	0	0	0	0	0	0	0
42	7	1		0								
12	3	2		0								
40	4	3		0								
25	5	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 $K[n, W]$

The knapsack problem

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

			j	0	1	2	3	4	5	6	7	8
v	w	i										
		0		0	0	0	0	0	0	0	0	0
42	7	1		0	?							
12	3	2		0								
40	4	3		0								
25	5	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 $K[n, W]$

The knapsack problem

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

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

			j	0	1	2	3	4	5	6	7	8
v	w	i										
		0		0	0	0	0	0	0	0	0	0
42	7	1		0	0	0	0	0	0	0		
12	3	2		0								
40	4	3		0								
25	5	4		0								

- $i = 1$
- $j = 7$

The knapsack problem

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

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 $K[n, W]$

			j	0	1	2	3	4	5	6	7	8
v	w	i										
		0		0	0	0	0	0	0	0	0	0
42	7	1		0	0	0	0	0	0	0		
12	3	2		0								
40	4	3		0								
25	5	4		0								

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

The knapsack problem

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

			j	0	1	2	3	4	5	6	7	8
v	w	i										
		0		0	0	0	0	0	0	0	0	0
42	7	1		0	0	0	0	0	0	0		
12	3	2		0								
40	4	3		0								
25	5	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 $K[n, W]$

- $i = 1$
- $j = 7$
- $K[1\text{-}1,7] = K[0,7] = 0$
- $K[1\text{-}1,7\text{-}7] + 42 = K[0,0] + 42 = 0 + 42 = 42$

The knapsack problem

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

			j	0	1	2	3	4	5	6	7	8
v	w	i										
		0		0	0	0	0	0	0	0	0	0
42	7	1		0	0	0	0	0	0	0	42	
12	3	2		0								
40	4	3		0								
25	5	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

The knapsack problem

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

			j	0	1	2	3	4	5	6	7	8
v	w	i										
		0		0	0	0	0	0	0	0	0	0
42	7	1		0	0	0	0	0	0	0	42	42
12	3	2		0								
40	4	3		0								
25	5	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]
```

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

The knapsack problem

- Next row. We won't have enough capacity until j=3

$$
\begin{array}{l}
\textbf{for } i \leftarrow 0 \text{ to } n \textbf{ do} \\
\quad K[i,0] \leftarrow 0 \\
\textbf{for } j \leftarrow 1 \text{ to } W \textbf{ do} \\
\quad K[0,j] \leftarrow 0 \\
\textbf{for } i \leftarrow 1 \text{ to } n \textbf{ do} \\
\quad \textbf{for } j \leftarrow 1 \text{ to } W \textbf{ do} \\
\quad\quad \textbf{if } j < w_i \textbf{ then} \\
\quad\quad\quad K[i,j] \leftarrow K[i-1,j] \\
\quad\quad \boxed{\begin{array}{l}\textbf{else} \\ \quad K[i,j] \leftarrow \max(K[i-1,j], K[i-1,j-w_i]+v_i)\end{array}} \\
\textbf{return } K[n,W]
\end{array}
$$

			j	0	1	2	3	4	5	6	7	8
v	w	i										
		0		0	0	0	0	0	0	0	0	0
42	7	1		0	0	0	0	0	0	0	42	42
12	3	2		0	0	0	12					
40	4	3		0								
25	5	4		0								

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

The knapsack problem

- But at j=7, it is better to pick 42

			j	0	1	2	3	4	5	6	7	8
v	w	i										
		0		0	0	0	0	0	0	0	0	0
42	7	1		0	0	0	0	0	0	0	42	42
12	3	2		0	0	0	12	12	12	12	42	
40	4	3		0								
25	5	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 $K[n, W]$

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

The knapsack problem

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

$$
\begin{array}{l}
\textbf{for } i \leftarrow 0 \text{ to } n \textbf{ do} \\
\quad K[i,0] \leftarrow 0 \\
\textbf{for } j \leftarrow 1 \text{ to } W \textbf{ do} \\
\quad K[0,j] \leftarrow 0 \\
\textbf{for } i \leftarrow 1 \text{ to } n \textbf{ do} \\
\quad \textbf{for } j \leftarrow 1 \text{ to } W \textbf{ do} \\
\quad\quad \textbf{if } j < w_i \textbf{ then} \\
\quad\quad\quad K[i,j] \leftarrow K[i-1,j] \\
\quad\quad \boxed{\begin{array}{l}\textbf{else} \\ \quad K[i,j] \leftarrow \max(K[i-1,j], K[i-1,j-w_i]+v_i)\end{array}} \\
\textbf{return } K[n,W]
\end{array}
$$

			j	0	1	2	3	4	5	6	7	8
v	w	i										
		0		0	0	0	0	0	0	0	0	0
42	7	1		0	0	0	0	0	0	0	42	42
12	3	2		0	0	0	12	12	12	12	42	42
40	4	3		0	0	0	12	40				
25	5	4		0								

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

The knapsack problem

- What would happen at j=7?

			j	0	1	2	3	4	5	6	7	8
v	w	i										
		0		0	0	0	0	0	0	0	0	0
42	7	1		0	0	0	0	0	0	0	42	42
12	3	2		0	0	0	12	12	12	12	42	42
40	4	3		0	0	0	12	40	40	40	52	
25	5	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 $K[n,W]$

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

The knapsack problem

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

```
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]
```

			j	0	1	2	3	4	5	6	7	8
v	w	i										
		0		0	0	0	0	0	0	0	0	0
42	7	1		0	0	0	0	0	0	0	42	42
12	3	2		0	0	0	12	12	12	12	42	42
40	4	3		0	0	0	12	40	40	40	52	52
25	5	4		0	0	0	12	40	40	40	52	52

- $i = 4$
- $j = 8$
- $K[4\text{-}1,8] = K[3,7] = 52$
- $K[4\text{-}1,8\text{-}5] + 25 = K[3,3] + 25 = 12 + 25 = 37$
- $K[4,8] = \max(52,37) = 52$

Memoing

- This bottom-up (table-filling) solution is overkill:
  - It solves **every conceivable sub-instance**, most of which are unnecessary

Memoing

- This bottom-up (table-filling) solution is overkill:
  - It solves **every conceivable sub-instance**, most of which are unnecessary
- A top-down approach with **memoing** is preferable
  - There are many implementations of the memo table
  - We will examine a simple array type implementation

Memoing

- This bottom-up (table-filling) solution is overkill:
  - It solves **every conceivable sub-instance**, most of which are unnecessary

- A top-down approach with **memoing** is preferable
  - There are many implementations of the memo table
  - We will examine a simple array type implementation

- And, yes, **memoing** is correctly spelled…

Memoing

- Lets look at this algorithm, step-by-step

- The data is again:
  - The knapsack capacity $W = 8$
  - The values are $\{42, 12, 40, 25\}$
  - The weights are $\{7, 3, 4, 5\}$

- $K[1..n, 1..W]$ is initialized to -1

```
function MFKnap(i, j)
    if i < 1 or j < 1 then
        return 0
    if K[i, j] < 0 then
        if j < w_i then
            K[i, j] ← MFKnap(i − 1, j)
        else
            K[i, j] ← max(MFKnap(i − 1, j), MFKnap(i − 1, j − w_i) + v_i)
    return K[i, j]
```

Memoing

- We start with i=4 and j=8

function $\text{MFK}\textsc{nap}(i, j)$
    if $i < 1$ **or** $j < 1$ **then**
        return 0
    if $K[i, j] < 0$ **then**
        if $j < w_i$ **then**
            $K[i, j] \leftarrow \text{MFK}\textsc{nap}(i-1, j)$
        else
            $K[i, j] \leftarrow \max(\text{MFK}\textsc{nap}(i-1, j), \text{MFK}\textsc{nap}(i-1, j-w_i) + v_i)$
    return $K[i, j]$

Memoing

- We start with i=4 and j=8

function $\text{MFK}\textsc{nap}(i, j)$
  if $i < 1$ **or** $j < 1$ **then**
    return 0
  if $K[i, j] < 0$ **then** K[4,8] = -1
    if $j < w_i$ **then**
      $K[i, j] \leftarrow \text{MFK}\textsc{nap}(i-1, j)$
    else
      $K[i, j] \leftarrow \max(\text{MFK}\textsc{nap}(i-1, j), \text{MFK}\textsc{nap}(i-1, j-w_i) + v_i)$
  return $K[i, j]$

Memoing

- We start with i=4 and j=8

function MFKNAP(i, j)
  if $i < 1$ **or** $j < 1$ **then**
    return 0
  if $K[i, j] < 0$ **then**
    if $j < w_i$ **then** ← **There is enough capacity to place w_4**
      $K[i, j] \leftarrow \text{MFKNAP}(i-1, j)$
    else
      $K[i, j] \leftarrow \max(\text{MFKNAP}(i-1, j), \text{MFKNAP}(i-1, j-w_i) + v_i)$
  return $K[i, j]$

Memoing

- We start with i=4 and j=8

			j	1	2	3	4	5	6	7	8
v	w	i									
42	7	1		-1	-1	-1	-1	-1	-1	-1	-1
12	3	2		-1	-1	-1	-1	-1	-1	-1	-1
40	4	3		-1	-1	-1	-1	-1	-1	-1	-1
25	5	4		-1	-1	-1	-1	-1	-1	-1	-1

```
function MFKNAP(i, j)
    if i < 1 or j < 1 then
        return 0
    if K[i, j] < 0 then
        if j < w_i then
            K[i, j] ← MFKNAP(i − 1, j)
        else
            K[i, j] ← max(MFKNAP(i − 1, j), MFKNAP(i − 1, j − w_i) + v_i)
    return K[i, j]
```

- $i = 4$
- $j = 8$

Memoing

- We start with i=4 and j=8

			j	1	2	3	4	5	6	7	8
v	w	i									
42	7	1		-1	-1	-1	-1	-1	-1	-1	-1
12	3	2		-1	-1	-1	-1	-1	-1	-1	-1
40	4	3		-1	-1	-1	-1	-1	-1	-1	-1
25	5	4		-1	-1	-1	-1	-1	-1	-1	-1

```
function MFKnap(i, j)
    if i < 1 or j < 1 then
        return 0
    if K[i, j] < 0 then
        if j < w_i then
            K[i, j] ← MFKnap(i − 1, j)
        else
            K[i, j] ← max(MFKnap(i − 1, j), MFKnap(i − 1, j − w_i) + v_i)
    return K[i, j]
```

- $i = 4$
- $j = 8$
- $K[4\text{-}1,8] = K[3,8]$

Memoing

- We start with i=4 and j=8

			j	1	2	3	4	5	6	7	8
v	w	i									
42	7	1		-1	-1	-1	-1	-1	-1	-1	-1
12	3	2		-1	-1	-1	-1	-1	-1	-1	-1
40	4	3		-1	-1	-1	-1	-1	-1	-1	-1
25	5	4		-1	-1	-1	-1	-1	-1	-1	-1

```
function MFKnap(i, j)
    if i < 1 or j < 1 then
        return 0
    if K[i, j] < 0 then
        if j < w_i then
            K[i, j] ← MFKnap(i − 1, j)
        else
            K[i, j] ← max(MFKnap(i − 1, j), MFKnap(i − 1, j − w_i) + v_i)
    return K[i, j]
```

- $i = 4$
- $j = 8$
- K[4-1,8] = K[3,8]
- K[4-1,8-5] + 25 = K[3,3] + 25

Memoing

- We start with i=4 and j=8

			j	1	2	3	4	5	6	7	8
v	w	i									
42	7	1		-1	-1	-1	-1	-1	-1	-1	-1
12	3	2		-1	-1	-1	-1	-1	-1	-1	-1
40	4	3		-1	-1	-1	-1	-1	-1	-1	-1
25	5	4		-1	-1	-1	-1	-1	-1	-1	-1

```
function MFKnap(i, j)
    if i < 1 or j < 1 then
        return 0
    if K[i, j] < 0 then
        if j < w_i then
            K[i, j] ← MFKnap(i − 1, j)
        else
            K[i, j] ← max(MFKnap(i − 1, j), MFKnap(i − 1, j − w_i) + v_i)
    return K[i, j]
```

We take this branch of the recursion

- $i = 4$
- $j = 8$
- K[4-1,8] = K[3,8]
- K[4-1,8-5] + 25 = K[3,3] + 25

Memoing

- Next is i=3 and j=8

			j	1	2	3	4	5	6	7	8
v	w	i									
42	7	1		-1	-1	-1	-1	-1	-1	-1	-1
12	3	2		-1	-1	-1	-1	-1	-1	-1	-1
40	4	3		-1	-1	-1	-1	-1	-1	-1	-1
25	5	4		-1	-1	-1	-1	-1	-1	-1	-1

function $\text{MFK}\textsc{nap}(i, j)$
    if $i < 1$ **or** $j < 1$ **then**
        return 0
    if $K[i, j] < 0$ **then**
        if $j < w_i$ **then**
            $K[i, j] \leftarrow \text{MFK}\textsc{nap}(i-1, j)$
        else
            $K[i, j] \leftarrow \max\left(\boxed{\text{MFK}\textsc{nap}(i-1, j)}, \boxed{\text{MFK}\textsc{nap}(i-1, j-w_i) + v_i}\right)$
    return $K[i, j]$

- $i = 3$
- $j = 8$
- $K[3\text{-}1,8] = K[2,8]$
- $K[3\text{-}1,8\text{-}4] + 40 = K[2,4] + 40$

Memoing

- Next is i=3 and j=8

			j	1	2	3	4	5	6	7	8
v	w	i									
42	7	1		-1	-1	-1	-1	-1	-1	-1	-1
12	3	2		-1	-1	-1	-1	-1	-1	-1	-1
40	4	3		-1	-1	-1	-1	-1	-1	-1	-1
25	5	4		-1	-1	-1	-1	-1	-1	-1	-1

function MFKNAP(i, j)
  if $i < 1$ **or** $j < 1$ **then**
    return 0
  if $K[i, j] < 0$ **then**
    if $j < w_i$ **then**
      $K[i, j] \leftarrow \text{MFKNAP}(i-1, j)$
    else
      $K[i, j] \leftarrow \max\left(\boxed{\text{MFKNAP}(i-1, j)}, \boxed{\text{MFKNAP}(i-1, j-w_i) + v_i}\right)$
  return $K[i, j]$

We continue with this branch of the recursion

- $i = 3$
- $j = 8$
- $K[3\text{-}1,8] = K[2,8]$
- $K[3\text{-}1,8\text{-}4] + 40 = K[2,4] + 40$

Memoing

- Next is i=2 and j=8

			j	1	2	3	4	5	6	7	8
v	w	i									
42	7	1		-1	-1	-1	-1	-1	-1	-1	-1
12	3	2		-1	-1	-1	-1	-1	-1	-1	-1
40	4	3		-1	-1	-1	-1	-1	-1	-1	-1
25	5	4		-1	-1	-1	-1	-1	-1	-1	-1

function MFKNAP(i, j)
  if $i < 1$ **or** $j < 1$ **then**
    return 0
  if $K[i, j] < 0$ **then**
    if $j < w_i$ **then**
      $K[i, j] \leftarrow \text{MFKNAP}(i-1, j)$
    else
      $K[i, j] \leftarrow \max(\boxed{\text{MFKNAP}(i-1, j)}, \boxed{\text{MFKNAP}(i-1, j-w_i) + v_i})$
  return $K[i, j]$

- $i = 2$
- $j = 8$
- $K[2\text{-}1,8] = K[1,8]$
- $K[2\text{-}1,8\text{-}3] + 12 = K[1,5] + 12$

Memoing

- Next is i=2 and j=8

			j	1	2	3	4	5	6	7	8
v	w	i									
42	7	1		-1	-1	-1	-1	-1	-1	-1	-1
12	3	2		-1	-1	-1	-1	-1	-1	-1	-1
40	4	3		-1	-1	-1	-1	-1	-1	-1	-1
25	5	4		-1	-1	-1	-1	-1	-1	-1	-1

function MFKNAP(i, j)
  if $i < 1$ **or** $j < 1$ **then**
    return 0
  if $K[i, j] < 0$ **then**
    if $j < w_i$ **then**
      $K[i, j] \leftarrow \text{MFKNAP}(i-1, j)$
    else
      $K[i, j] \leftarrow \max\left(\boxed{\text{MFKNAP}(i-1, j)}, \boxed{\text{MFKNAP}(i-1, j-w_i) + v_i}\right)$
  return $K[i, j]$

We continue with this branch of the recursion

- $i = 2$
- $j = 8$
- $K[2\text{-}1,8] = K[1,8]$
- $K[2\text{-}1,8\text{-}3] + 12 = K[1,5] + 12$

Memoing

- At i=1 and j=8, we reach the bottom of this branch

			j	1	2	3	4	5	6	7	8
v	w	i									
42	7	1		-1	-1	-1	-1	-1	-1	-1	-1
12	3	2		-1	-1	-1	-1	-1	-1	-1	-1
40	4	3		-1	-1	-1	-1	-1	-1	-1	-1
25	5	4		-1	-1	-1	-1	-1	-1	-1	-1

```
function MFKNAP(i, j)
    if i < 1 or j < 1 then
        return 0
    if K[i, j] < 0 then
        if j < w_i then
            K[i, j] ← MFKNAP(i − 1, j)
        else
            K[i, j] ← max(MFKNAP(i − 1, j), MFKNAP(i − 1, j − w_i) + v_i)
    return K[i, j]
```

- $i = 1$
- $j = 8$

Memoing

- At i=1 and j=8, we reach the bottom of this branch

			j	1	2	3	4	5	6	7	8
v	w	i									
42	7	1		-1	-1	-1	-1	-1	-1	-1	-1
12	3	2		-1	-1	-1	-1	-1	-1	-1	-1
40	4	3		-1	-1	-1	-1	-1	-1	-1	-1
25	5	4		-1	-1	-1	-1	-1	-1	-1	-1

function MFKNAP(i, j)
    if $i < 1$ **or** $j < 1$ **then**
        return 0
    if $K[i, j] < 0$ **then**
        if $j < w_i$ **then**
            $K[i, j] \leftarrow$ MFKNAP$(i-1, j)$
        else
            $K[i, j] \leftarrow \max\left(\boxed{\text{MFKNAP}(i-1, j)}, \text{MFKNAP}(i-1, j-w_i) + v_i\right)$
    return $K[i, j]$

- $i = 1$
- $j = 8$
- $K[1\text{-}1,8] = K[0,8] = 0$

Memoing

- At i=1 and j=8, we reach the bottom of this branch

			j	1	2	3	4	5	6	7	8
v	w	i									
42	7	1		-1	-1	-1	-1	-1	-1	-1	-1
12	3	2		-1	-1	-1	-1	-1	-1	-1	-1
40	4	3		-1	-1	-1	-1	-1	-1	-1	-1
25	5	4		-1	-1	-1	-1	-1	-1	-1	-1

function $\text{MFK}\textsc{nap}(i, j)$
  if $i < 1$ **or** $j < 1$ **then**
    return 0
  if $K[i, j] < 0$ **then**
    if $j < w_i$ **then**
      $K[i, j] \leftarrow \text{MFK}\textsc{nap}(i-1, j)$
    else
      $K[i, j] \leftarrow \max\left(\boxed{\text{MFK}\textsc{nap}(i-1, j)}, \boxed{\text{MFK}\textsc{nap}(i-1, j-w_i) + v_i}\right)$
  return $K[i, j]$

- $i = 1$
- $j = 8$
- $K[1\text{-}1,8] = K[0,8] = 0$
- $K[1\text{-}1,8\text{-}7] + 42 = K[0,1] + 42 = 0 + 42 = 42$

Memoing

- At i=1 and j=8, we reach the bottom of this branch

			j	1	2	3	4	5	6	7	8
v	w	i									
42	7	1		-1	-1	-1	-1	-1	-1	-1	42
12	3	2		-1	-1	-1	-1	-1	-1	-1	-1
40	4	3		-1	-1	-1	-1	-1	-1	-1	-1
25	5	4		-1	-1	-1	-1	-1	-1	-1	-1

function $\text{MFK}\textsc{nap}(i, j)$
  if $i < 1$ **or** $j < 1$ **then**
    return 0
  if $K[i, j] < 0$ **then**
    if $j < w_i$ **then**
      $K[i, j] \leftarrow \text{MFK}\textsc{nap}(i-1, j)$
    else
      $K[i, j] \leftarrow \max\left(\boxed{\text{MFK}\textsc{nap}(i-1, j)}, \boxed{\text{MFK}\textsc{nap}(i-1, j-w_i) + v_i}\right)$
  return $K[i, j]$

- $i = 1$
- $j = 8$
- 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

Memoing

- At i=1 and j=8, we reach the bottom of this branch
- We go back to i=2 and j=8 to continue

			j	1	2	3	4	5	6	7	8
v	w	i									
42	7	1		-1	-1	-1	-1	-1	-1	-1	42
12	3	2		-1	-1	-1	-1	-1	-1	-1	-1
40	4	3		-1	-1	-1	-1	-1	-1	-1	-1
25	5	4		-1	-1	-1	-1	-1	-1	-1	-1

```
function MFKnap(i, j)
    if i < 1 or j < 1 then
        return 0
    if K[i, j] < 0 then
        if j < w_i then
            K[i, j] ← MFKnap(i − 1, j)
        else
            K[i, j] ← max(MFKnap(i − 1, j), MFKnap(i − 1, j − w_i) + v_i)
    return K[i, j]
```

We continue with this branch of the recursion

- $i = 2$
- $j = 8$
- $K[2\text{-}1,8] = K[1,8] = 42$
- $K[2\text{-}1,8\text{-}3] + 12 = K[1,5] + 12$

Memoing

- At i=1 and j=5, we also reach the bottom of this branch

			j	1	2	3	4	5	6	7	8
v	w	i									
42	7	1		-1	-1	-1	-1	0	-1	-1	42
12	3	2		-1	-1	-1	-1	-1	-1	-1	-1
40	4	3		-1	-1	-1	-1	-1	-1	-1	-1
25	5	4		-1	-1	-1	-1	-1	-1	-1	-1

function $\text{MFK}\textsc{nap}(i, j)$
    if $i < 1$ **or** $j < 1$ **then**
        return 0
    if $K[i, j] < 0$ **then**
        if $j < w_i$ **then**
            $K[i, j] \leftarrow \text{MFK}\textsc{nap}(i-1, j)$
        else
            $K[i, j] \leftarrow \max\left(\boxed{\text{MFK}\textsc{nap}(i-1, j)}, \boxed{\text{MFK}\textsc{nap}(i-1, j-w_i) + v_i}\right)$
    return $K[i, j]$

- $i = 1$
- $j = 5$
- K[1-1,5] = K[0,5] = 0
- K[1-1,5-8] = 0
- K[1,5] = max(0,0) = 0

Memoing

- At i=1 and j=5, we also reach the bottom of this branch
- We continue the algorithm, until we find our solution

			j	1	2	3	4	5	6	7	8
v	w	i									
42	7	1		-1	-1	-1	-1	0	-1	-1	42
12	3	2		-1	-1	-1	-1	-1	-1	-1	-1
40	4	3		-1	-1	-1	-1	-1	-1	-1	-1
25	5	4		-1	-1	-1	-1	-1	-1	-1	-1

```
function MFKnap(i, j)
    if i < 1 or j < 1 then
        return 0
    if K[i, j] < 0 then
        if j < w_i then
            K[i, j] ← MFKnap(i − 1, j)
        else
            K[i, j] ← max(MFKnap(i − 1, j), MFKnap(i − 1, j − w_i) + v_i)
    return K[i, j]
```

- $i = 1$
- $j = 5$
- $K[1\text{-}1,5] = K[0,5] = 0$
- $K[1\text{-}1,5\text{-}8] = 0$
- $K[1,5] = \max(0,0) = 0$

Memoing

- The states visited (11) are shown in the table

			j	1	2	3	4	5	6	7	8
v	w	i									
42	7	1		0	-1	0	0	0	-1	-1	42
12	3	2		-1	-1	12	12	-1	-1	-1	42
40	4	3		-1	-1	12	-1	-1	-1	-1	52
25	5	4		-1	-1	-1	-1	-1	-1	-1	52

Memoing

- The states visited (11) are shown in the table
  - Unlike the bottom-up approach, in which we visited all the states (40)

			j	1	2	3	4	5	6	7	8
v	w	i									
42	7	1		0	-1	0	0	0	-1	-1	42
12	3	2		-1	-1	12	12	-1	-1	-1	42
40	4	3		-1	-1	12	-1	-1	-1	-1	52
25	5	4		-1	-1	-1	-1	-1	-1	-1	52

Memoing

- The states visited (11) are shown in the table
  - Unlike the bottom-up approach, in which we visited all the states (40)
- There are a lot of never used places in the table. Hence, the algorithm is less space-efficient

			j	1	2	3	4	5	6	7	8
v	w	i									
42	7	1		0	-1	0	0	0	-1	-1	42
12	3	2		-1	-1	12	12	-1	-1	-1	42
40	4	3		-1	-1	12	-1	-1	-1	-1	52
25	5	4		-1	-1	-1	-1	-1	-1	-1	52

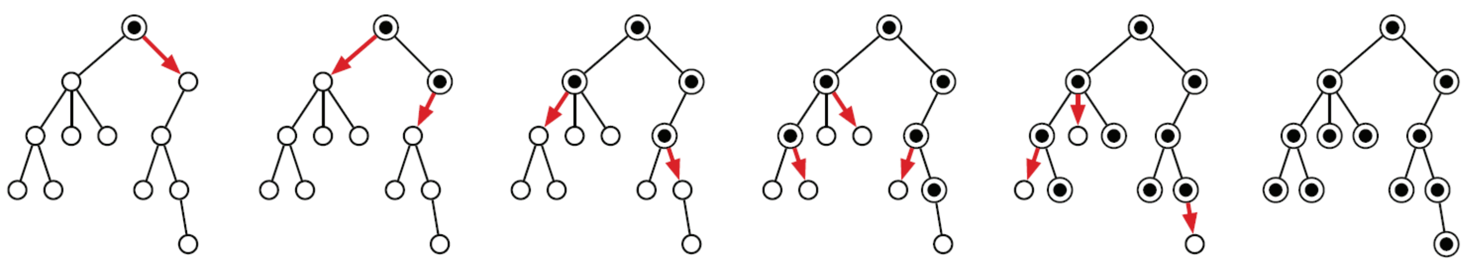
Memoing

- The states visited (11) are shown in the table
  - Unlike the bottom-up approach, in which we visited all the states (40)
- There are a lot of never used places in the table. Hence, the algorithm is less space-efficient
  - Can you think of a way to improve the space efficiency?

			j	1	2	3	4	5	6	7	8
v	w	i									
42	7	1		0	-1	0	0	0	-1	-1	42
12	3	2		-1	-1	12	12	-1	-1	-1	42
40	4	3		-1	-1	12	-1	-1	-1	-1	52
25	5	4		-1	-1	-1	-1	-1	-1	-1	52

Message passing in a tree

- Suppose we need to broadcast a message to all the nodes in a n-ary tree.
  - Initially, only the root node knows the message.
  - In a single round, any node that knows the message can forward it to at most one of its children.
  - What would be the minimum number of rounds required for the message to reach all the nodes?



Message passing in a tree

- To solve this problem, we should first answer a few questions:

Message passing in a tree

- To solve this problem, we should first answer a few questions:
  - What is the **base case**?

Message passing in a tree

- To solve this problem, we should first answer a few questions:
  - What is the **base case**?

If the node has no children, then the number of rounds required is zero

Message passing in a tree

- To solve this problem, we should first answer a few questions:
  - What is the **base case**?

    If the node has no children, then the number of rounds required is zero

  - Which **child** should receive the message **first** (second, third…)?

Message passing in a tree

- To solve this problem, we should first answer a few questions:
  - What is the **base case**?

    If the node has no children, then the number of rounds required is zero

  - Which **child** should receive the message **first** (second, third…)?

    If the node has the largest tree, then it should receive the message first

Message passing in a tree

- To solve this problem, we should first answer a few questions:
  - What is the **base case**?

    If the node has no children, then the number of rounds required is zero

  - Which **child** should receive the message **first** (second, third…)?

    If the node has the largest tree, then it should receive the message first

  - How do we **accumulate the rounds** for each parent?

Message passing in a tree

- To solve this problem, we should first answer a few questions:
  - What is the **base case**?

    If the node has no children, then the number of rounds required is zero

  - Which **child** should receive the message **first** (second, third…)?

    If the node has the largest tree, then it should receive the message first

  - How do we **accumulate the rounds** for each parent?

    To the largest child, the rounds increase by one, for the second largest by two…

Message passing in a tree

- This results in the following recursive relationship:

$$v = \begin{cases} 0 & \text{if } n = 0 \\ \max\left\{i = 1, \ldots, n | v_{(i)\downarrow} + i\right\} & \text{if } n > 0 \end{cases}$$

Message passing in a tree

This is notation indicates the results are ordered

- This results in the following recursive relationship:

$$v = \begin{cases} 0 & \text{if } n = 0 \\ \max\left\{i = 1, \ldots, n \middle| v_{(i)\downarrow} + i\right\} & \text{if } n > 0 \end{cases}$$

Message passing in a tree

- This results in the following recursive relationship:

$$v = \begin{cases} 0 & \text{if } n = 0 \\ \max\left\{i = 1, \ldots, n | v_{(i)\downarrow} + i\right\} & \text{if } n > 0 \end{cases}$$

This is notation indicates a child

Message passing in a tree

- This results in the following recursive relationship:

$$v = \begin{cases} 0 & \text{if } n = 0 \\ \max\left\{i = 1, \ldots, n | v_{(i)\downarrow} + i\right\} & \text{if } n > 0 \end{cases}$$

- Which we translate in the following algorithm:

function MinNumberOfRounds(T)
  if $T.n = 0$ **then**
    return 0
  $T.v\,[1, \ldots, T.n] \leftarrow$ NULL
  for $i \leftarrow 1$ to $T.n$ **do**
    $T.v\,[i] \leftarrow$ MinNumberOfRounds $(T.child\,[i])$
  $A \leftarrow$ SortDescending $(T.v)$
  for $i \leftarrow 1$ to $T.n$ **do**
    $A\,[i] \leftarrow A\,[i] + i$
  return $\max(A)$

Message passing in a tree

- This results in the following recursive relationship:

$$v = \begin{cases} 0 & \text{if } n = 0 \\ \max\left\{i = 1, \ldots, n | v_{(i)\downarrow} + i\right\} & \text{if } n > 0 \end{cases}$$

- Which we translate in the following algorithm:

This is the base case, return 0

function $\textsc{MinNumberOfRounds}(T)$
  if $T.n = 0$ **then**
    return 0
  $T.v\,[1, \ldots, T.n] \leftarrow \textsc{null}$
  for $i \leftarrow 1$ to $T.n$ **do**
    $T.v\,[i] \leftarrow \textsc{MinNumberOfRounds}\,(T.child\,[i])$
  $A \leftarrow \textsc{SortDescending}\,(T.v)$
  for $i \leftarrow 1$ to $T.n$ **do**
    $A\,[i] \leftarrow A\,[i] + i$
  return $\max\,(A)$

Message passing in a tree

- This results in the following recursive relationship:

$$v = \begin{cases} 0 & \text{if } n = 0 \\ \max\left\{i = 1, \ldots, n | v_{(i)\downarrow} + i\right\} & \text{if } n > 0 \end{cases}$$

- Which we translate in the following algorithm:

Our memoing structure is an array stored on each node

function $\textsc{MinNumberOfRounds}(T)$
    if $T.n = 0$ **then**
        return 0
    $T.v\,[1, \ldots, T.n] \leftarrow \textsc{null}$
    for $i \leftarrow 1$ to $T.n$ **do**
        $T.v\,[i] \leftarrow \textsc{MinNumberOfRounds}\,(T.child\,[i])$
    $A \leftarrow \textsc{SortDescending}\,(T.v)$
    for $i \leftarrow 1$ to $T.n$ **do**
        $A\,[i] \leftarrow A\,[i] + i$
    return $\max\,(A)$

Message passing in a tree

- This results in the following recursive relationship:

$$v = \begin{cases} 0 & \text{if } n = 0 \\ \max\left\{i = 1, \ldots, n | v_{(i)\downarrow} + i\right\} & \text{if } n > 0 \end{cases}$$

- Which we translate in the following algorithm:

We follow the recursion over each child

function $\textsc{MinNumberOfRounds}(T)$
    if $T.n = 0$ **then**
        return 0
    $T.v\,[1, \ldots, T.n] \leftarrow \textsc{null}$
    for $i \leftarrow 1$ to $T.n$ **do**
        $T.v\,[i] \leftarrow \textsc{MinNumberOfRounds}\,(T.child\,[i])$
    $A \leftarrow \textsc{SortDescending}\,(T.v)$
    for $i \leftarrow 1$ to $T.n$ **do**
        $A\,[i] \leftarrow A\,[i] + i$
    return $\max\,(A)$

Message passing in a tree

- This results in the following recursive relationship:

$$v = \begin{cases} 0 & \text{if } n = 0 \\ \max\left\{i = 1, \ldots, n | v_{(i)\downarrow} + i\right\} & \text{if } n > 0 \end{cases}$$

- Which we translate in the following algorithm:

We sort the results from the largest to the smallest

function $\textsc{MinNumberOfRounds}(T)$
    if $T.n = 0$ **then**
        return 0
    $T.v\,[1, \ldots, T.n] \leftarrow \textsc{null}$
    for $i \leftarrow 1$ to $T.n$ **do**
        $T.v\,[i] \leftarrow \textsc{MinNumberOfRounds}\,(T.child\,[i])$
    $A \leftarrow \textsc{SortDescending}\,(T.v)$
    for $i \leftarrow 1$ to $T.n$ **do**
        $A\,[i] \leftarrow A\,[i] + i$
    return $\max\,(A)$

Message passing in a tree

- This results in the following recursive relationship:

$$v = \begin{cases} 0 & \text{if } n = 0 \\ \max\left\{i = 1, \ldots, n | v_{(i)\downarrow} + i\right\} & \text{if } n > 0 \end{cases}$$

- Which we translate in the following algorithm:

We increase the number of rounds

$$
\begin{array}{l}
\textbf{function } \textsc{MinNumberOfRounds}(T) \\
\quad \textbf{if } T.n = 0 \textbf{ then} \\
\quad\quad \textbf{return } 0 \\
\quad T.v\left[1, \ldots, T.n\right] \leftarrow \textsc{null} \\
\quad \textbf{for } i \leftarrow 1 \text{ to } T.n \textbf{ do} \\
\quad\quad T.v\left[i\right] \leftarrow \textsc{MinNumberOfRounds}\left(T.child\left[i\right]\right) \\
\quad A \leftarrow \textsc{SortDescending}\left(T.v\right) \\
\quad \textbf{for } i \leftarrow 1 \text{ to } T.n \textbf{ do} \\
\quad\quad A\left[i\right] \leftarrow A\left[i\right] + i \\
\quad \textbf{return } \max\left(A\right)
\end{array}
$$

Message passing in a tree

- This results in the following recursive relationship:

$$v = \begin{cases} 0 & \text{if } n = 0 \\ \max\left\{i = 1, \ldots, n | v_{(i)\downarrow} + i\right\} & \text{if } n > 0 \end{cases}$$

- Which we translate in the following algorithm:

$$
\begin{array}{l}
\textbf{function } \textsc{MinNumberOfRounds}(T) \\
\quad \textbf{if } T.n = 0 \textbf{ then} \\
\quad\quad \textbf{return } 0 \\
\quad T.v\,[1, \ldots, T.n] \leftarrow \textsc{null} \\
\quad \textbf{for } i \leftarrow 1 \text{ to } T.n \textbf{ do} \\
\quad\quad T.v\,[i] \leftarrow \textsc{MinNumberOfRounds}\,(T.child\,[i]) \\
\quad A \leftarrow \textsc{SortDescending}\,(T.v) \\
\quad \textbf{for } i \leftarrow 1 \text{ to } T.n \textbf{ do} \\
\quad\quad A\,[i] \leftarrow A\,[i] + i \\
\quad \textbf{return } \max\,(A)
\end{array}
$$

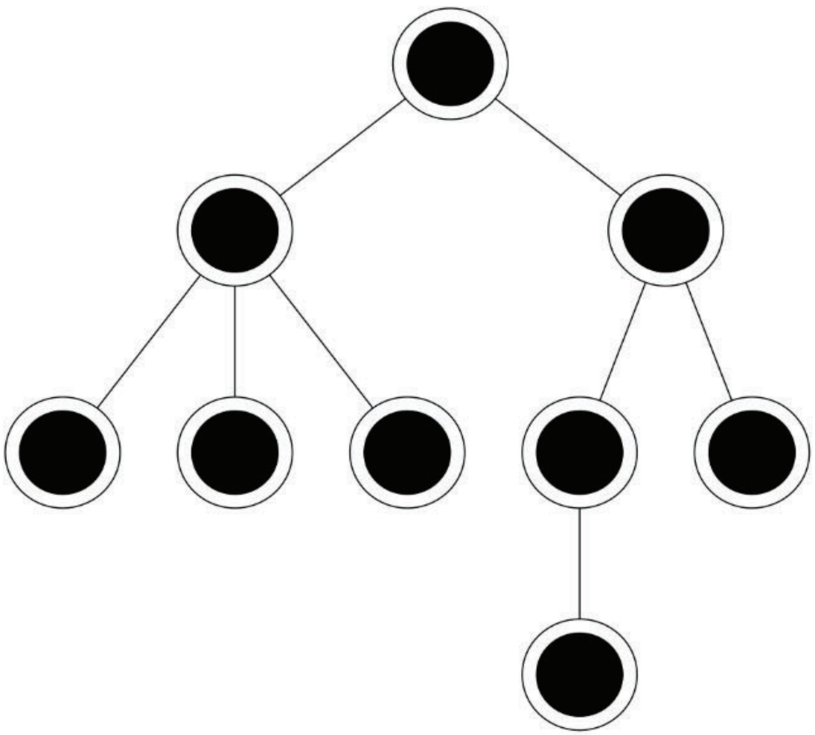
We return the maximum as the result

Dynamic programming

- DP algorithms, when well constructed, are usually efficient and very elegant
  - However, it is easy to get the **design wrong** if the **recurrence relationship** or the **evaluation order** are mistaken

  - **Don't write any code before you are sure that the recursion is correct!!!**

How many rounds would take to broadcast the message for the tree in the figure?



2

4

5

3

Next lecture

- Dynamic programming on graphs (Levitin Section 8.4)
  - Warshall's algorithm for transitive closure
  - Floyd's algorithm for all-pairs shortest-paths