# COMP90038 Algorithms and Complexity

## Dynamic Programming

Douglas Pires

Lecture 18

Semester 1, 2021

## Dynamic Programming

Dynamic programming is an algorithm design technique which sometimes is 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 naive 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.

## Dynamic Programming and Optimization

Optimization problems sometimes allow for clever dynamic programming solutions.

A prerequisite for the application here is a certain preservation of optimality: An optimal solution to an instance can be obtained by combining optimal solutions to the sub-instances.

## Example 1: The Coin-Row Problem

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

Think of the problem recursively.

## Example 1: The Coin-Row Problem

Given a row of coins, pick up 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 $v_1, v_2, \ldots v_n$.

Let $S(i)$ be the sum that can be gotten by picking optimally from the first $i$ coins.

Either the $i$th coin (with value $v_i$) is part of the solution or it is not.

If we choose to pick it up then we cannot also pick its neighbour on the left, so the best we can achieve is $S(i-2) + v_i$.

Otherwise we leave it, and the best we can achieve is $S(i-1)$.

## Example 1: The Coin-Row Problem

Here is saying the same thing formally, as a recurrence relation:

$$S(n) = max\{S(n-1), S(n-2) + v_n\}$$

This holds for $n > 1$.

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

## Example 1: The Coin-Row Problem

Here is saying the same thing formally, as a recurrence relation:

$$S(n) = max\{S(n-1), S(n-2) + v_n\}$$

This holds for $n > 1$.

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 solution suffers the same problem as the naive 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:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| index: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| $C$ : | | 20 | 10 | 20 | 50 | 20 | 10 | 20 |
| $S$ : | 0 | 20 | | | | | | |

## Example 1: The Coin-Row Problem

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

| index: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| $C$ : | | 20 | 10 | 20 | 50 | 20 | 10 | 20 |
| $S$ : | 0 | 20 | 20 | | | | | |

## Example 1: The Coin-Row Problem

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

| index: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| $C$ : |  | 20 | 10 | 20 | 50 | 20 | 10 | 20 |
| $S$ : | 0 | 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:

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

## Example 1: The Coin-Row Problem

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

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

## Example 1: The Coin-Row Problem

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

| 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 |  |

## Example 1: The Coin-Row Problem

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

| 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 |

## Example 1: The Coin-Row Problem

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

| 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 track 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.

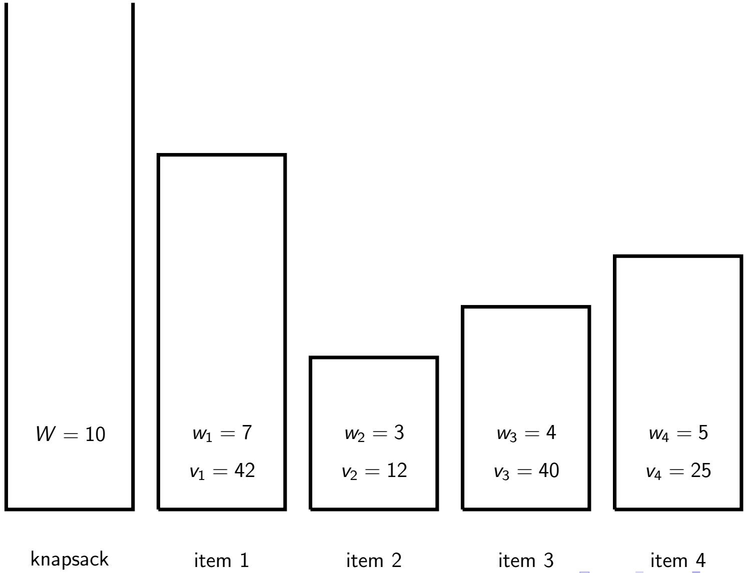
Given $n$ items with

- weights: $w_1, w_2, \ldots, w_n$
- values: $v_1, v_2, \ldots, v_n$
- knapsack of 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: Knapsack



## Example 2: The Knapsack Problem

We previously devised a brute-force algorithm, but dynamic programming may give us 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 $1, 2, \dots i$ of items 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 us 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 $1, 2, \ldots i$ of items considered so far, 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 $K(n, W)$.

## Example 2: The Knapsack Problem

The reason why 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 why 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 item $i$, the value of an optimal subset is simply $K(i - 1, w)$.

## Example 2: The Knapsack Problem

The reason why 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 item $i$, the value of an optimal subset is simply $K(i-1, w)$.

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 $w - w_i$. The value of such a subset is $K(i-1, w-w_i) + v_i$.

## Example 2: The Knapsack Problem

The reason why 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 item $i$, the value of an optimal subset is simply $K(i-1, w)$.

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 $w - w_i$. The value of such a subset is $K(i-1, w-w_i) + v_i$.

. . . provided item $i$ fits, that is, provided $w - w_i \geq 0$.

## 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:

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

## 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 leftmost column and top row, then proceed row by row:

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

## Example 2: The Knapsack Problem

Here is a concrete example—like before, except we let $W = 8$.

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

| $i \backslash j$ | 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 | | |
| 4 | 0 | | | | | | | | |

## Example 2: The Knapsack Problem

Here is a concrete example—like before, except we let $W = 8$.

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

| $i \backslash j$ | 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 | | |
| 4 | 0 | | | | | | | | |

# Example 2: The Knapsack Problem

Here is a concrete example—like before, except we let $W = 8$.

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

| $i \backslash j$ | 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 |

We conclude that the optimal value is 52.

We can find the optimal combination by back-tracing through the computation of the table entries.

## 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 hashtable 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
        $k \leftarrow \text{KNAP}(i - 1, j)$
    else
        $k \leftarrow max(\text{KNAP}(i - 1, j), \text{KNAP}(i - 1, j - w_i) + v_i)$
    insert $k$ into hashtable, with key $(i, j)$
    return $k$
```

## 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.