# 16. Dynamic Programming

Memoization, Optimal Substructure, Overlapping Sub-Problems, Dependencies, General Procedure. Examples: Rod Cutting, Rabbits

[Ottman/Widmayer, Kap. 7.1, 7.4, Cormen et al, Kap. 15]

# Fibonacci Numbers

 (again)

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

Analysis: why ist the recursive algorithm so slow?

# Algorithm FibonacciRecursive($n$)

**Input:** $n \geq 0$
**Output:** $n$-th Fibonacci number

**if** $n < 2$ **then**
  $f \leftarrow n$
**else**
  $f \leftarrow$ FibonacciRecursive$(n - 1) +$ FibonacciRecursive$(n - 2)$
**return** $f$

# Analysis

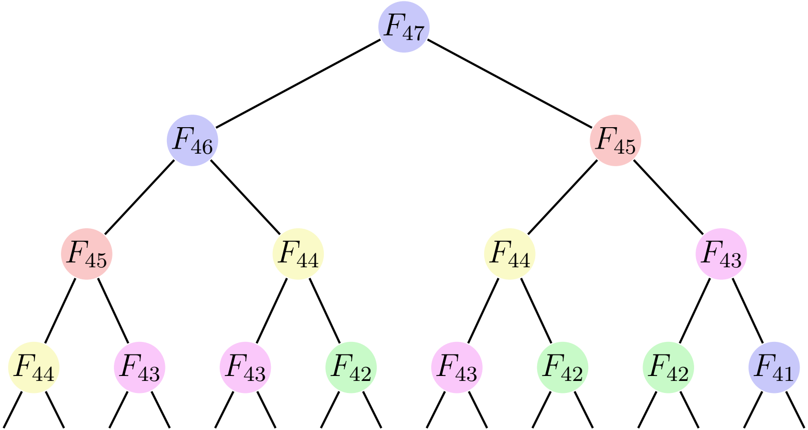
$T(n)$: Number executed operations.

- $n = 0, 1$: $T(n) = \Theta(1)$
- $n \geq 2$: $T(n) = T(n-2) + T(n-1) + c$.

$$T(n) = T(n-2) + T(n-1) + c \geq 2T(n-2) + c \geq 2^{n/2}c' = (\sqrt{2})^n c'$$

Algorithm is **exponential** in $n$.

# Reason (visual)



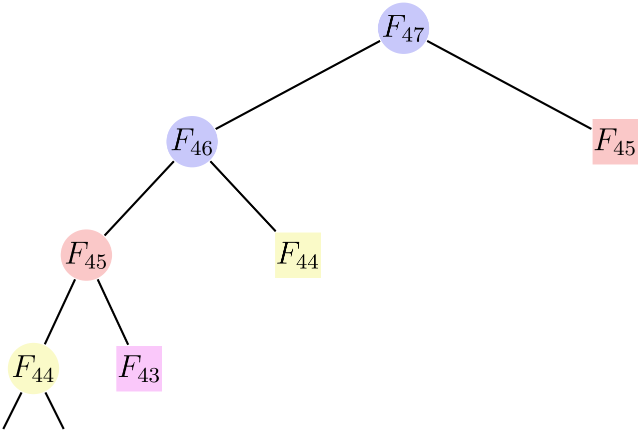

Nodes with same values are evaluated (too) often.

# Memoization

**Memoization** (sic) saving intermediate results.

- Before a subproblem is solved, the existence of the corresponding intermediate result is checked.
- If an intermediate result exists then it is used.
- Otherwise the algorithm is executed and the result is saved accordingly.

# Memoization with Fibonacci




Rechteckige Knoten wurden bereits ausgewertet.

# Algorithm FibonacciMemoization($n$)

**Input:** $n \geq 0$
**Output:** $n$-th Fibonacci number

**if** $n \leq 2$ **then**
  $f \leftarrow 1$
**else if** $\exists$memo$[n]$ **then**
  $f \leftarrow$ memo$[n]$
**else**
  $f \leftarrow$ FibonacciMemoization$(n-1)$ + FibonacciMemoization$(n-2)$
  memo$[n] \leftarrow f$

**return** $f$

# Analysis

Computational complexity:

$$T(n) = T(n-1) + c = ... = \mathcal{O}(n).$$

because after the call to $f(n-1)$, $f(n-2)$ has already been computed.

A different argument: $f(n)$ is computed exactly once recursively for each $n$.
Runtime costs: $n$ calls with $\Theta(1)$ costs per call $n \cdot c \in \Theta(n)$. The recursion vanishes from the running time computation.

Algorithm requires $\Theta(n)$ memory.[22]

[22] But the naive recursive algorithm also requires $\Theta(n)$ memory implicitly.

# Looking closer ...

... the algorithm computes the values of $F_1$, $F_2$, $F_3$,$\ldots$ in the **top-down** approach of the recursion.

Can write the algorithm **bottom-up**. This is characteristic for **dynamic programming**.

# Algorithm FibonacciBottomUp(n)

**Input:** $n \geq 0$
**Output:** $n$-th Fibonacci number

$F[1] \leftarrow 1$
$F[2] \leftarrow 1$
**for** $i \leftarrow 3, \ldots, n$ **do**
  $F[i] \leftarrow F[i-1] + F[i-2]$

**return** $F[n]$

# Dynamic Programming: Idea

- Divide a complex problem into a reasonable number of sub-problems
- The solution of the sub-problems will be used to solve the more complex problem
- Identical problems will be computed only once

# Dynamic Programming Consequence

Identical problems will be computed only once

$\Rightarrow$ Results are saved


We trade spee against memory consumption

# Dynamic Programming: Description

1. Use a **DP-table** with information to the subproblems.
   Dimension of the entries? Semantics of the entries?
2. Computation of the **base cases**
   Which entries do not depend on others?
3. Determine **computation order**.
   In which order can the entries be computed such that dependencies are fulfilled?
4. Read-out the **solution**
   How can the solution be read out from the table?

Runtime (typical) = number entries of the table times required operations per entry.

# Dynamic Programing: Description with the example

1. Dimension of the table? Semantics of the entries?

   $n \times 1$ table. $n$th entry contains $n$th Fibonacci number.

2. Which entries do not depend on other entries?

   Values $F_1$ and $F_2$ can be computed easily and independently.

3. Computation order?

   $F_i$ with increasing $i$.

4. Reconstruction of a solution?

   $F_n$ ist die $n$-te Fibonacci-Zahl.

# Dynamic Programming = Divide-And-Conquer ?

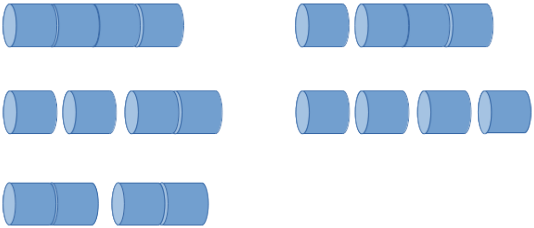
- In both cases the original problem can be solved (more easily) by utilizing the solutions of sub-problems. The problem provides **optimal substructure**.
- Divide-And-Conquer algorithms (such as Mergesort): sub-problems are independent; their solutions are required only once in the algorithm.
- DP: sub-problems are dependent. The problem is said to have **overlapping sub-problems** that are required multiple-times in the algorithm.
- In order to avoid redundant computations, results are tabulated. For **sub-problems there must not be any circular dependencies**.

# Rod Cutting

- Rods (metal sticks) are cut and sold.
- Rods of length $n \in \mathbb{N}$ are available. A cut does not provide any costs.
- For each length $l \in \mathbb{N}$, $l \leq n$ known is the value $v_l \in \mathbb{R}^+$
- Goal: cut the rods such (into $k \in \mathbb{N}$ pieces) that

$$\sum_{i=1}^{k} v_{l_i} \text{ is maximized subject to } \sum_{i=1}^{k} l_i = n.$$

# Rod Cutting: Example



Possibilities to cut a rod of length 4 (without permutations)

| Length | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| Price | 0 | 2 | 3 | 8 | 9 |

$\Rightarrow$ Best cut: 3 + 1 with value 10.

# Wie findet man den DP Algorithms

0. Exact formulation of the wanted solution
1. Define sub-problems (and compute the cardinality)
2. Guess / Enumerate (and determine the running time for guessing)
3. Recursion: relate sub-problems
4. Memoize / Tabularize. Determine the dependencies of the sub-problems
5. Solve the problem
   Running time = #sub-problems $\times$ time/sub-problem

# Structure of the problem

0. **Wanted:** $r_n$ = maximal value of rod (cut or as a whole) with length $n$.
1. **sub-problems**: maximal value $r_k$ for each $0 \leq k < n$
2. **Guess** the length of the first piece
3. **Recursion**

$$\begin{aligned} r_k &= \max\{v_i + r_{k-i} : 0 < i \leq k\}, \quad k > 0 \\ r_0 &= 0 \end{aligned}$$

4. **Dependency:** $r_k$ depends (only) on values $v_i$, $1 \leq i \leq k$ and the optimal cuts $r_i$, $i < k$
5. **Solution** in $r_n$

# Algorithm RodCut($v$,$n$)

**Input:** $n \geq 0$, Prices $v$
**Output:** best value

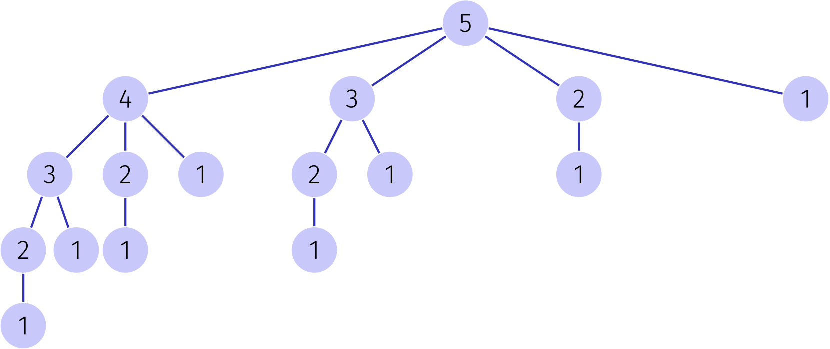
$q \leftarrow 0$
**if** $n > 0$ **then**
  **for** $i \leftarrow 1, \ldots, n$ **do**
    $q \leftarrow \max\{q, v_i + \text{RodCut}(v, n-i)\}$;
**return** $q$

Running time $T(n) = \sum_{i=0}^{n-1} T(i) + c \quad \Rightarrow$[23] $\quad T(n) \in \Theta(2^n)$

---

[23] $T(n) = T(n-1) + \sum_{i=0}^{n-2} T(i) + c = T(n-1) + (T(n-1) - c) + c = 2T(n-1) \quad (n > 0)$

# Recursion Tree



# Algorithm RodCutMemoized($m, v, n$)

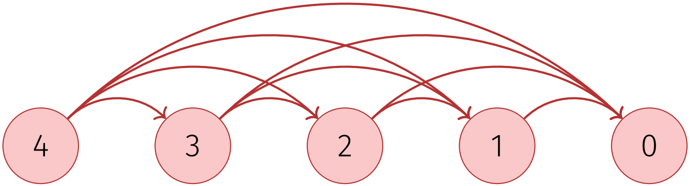
**Input:** $n \geq 0$, Prices $v$, Memoization Table $m$
**Output:** best value

```
q ← 0
if n > 0 then
    if ∃ m[n] then
        q ← m[n]
    else
        for i ← 1, ..., n do
            q ← max{q, v_i + RodCutMemoized(m, v, n − i)};
        m[n] ← q
return q
```

Running time $\sum_{i=1}^{n} i = \Theta(n^2)$

# Subproblem-Graph

Describes the mutual dependencies of the subproblems




and must not contain cycles

# Construction of the Optimal Cut

- During the (recursive) computation of the optimal solution for each $k \leq n$ the recursive algorithm determines the optimal length of the first rod
- Store the lenght of the first rod in a separate table of length $n$

# Bottom-up Description with the example

1. Dimension of the table? Semantics of the entries?

   $n \times 1$ table. $n$th entry contains the best value of a rod of length $n$.

2. Which entries do not depend on other entries?

   Value $r_0$ is $0$

3. Computation order?

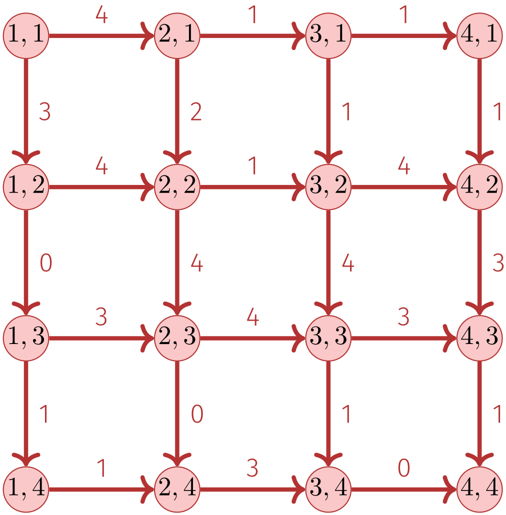
   $r_i, i = 1, \ldots, n$.

4. Reconstruction of a solution?

   $r_n$ is the best value for the rod of length $n$.

# Rabbit!

A rabbit sits on cite $(1,1)$ of an $n \times n$ grid. It can only move to east or south. On each pathway there is a number of carrots. How many carrots does the rabbit collect maximally?

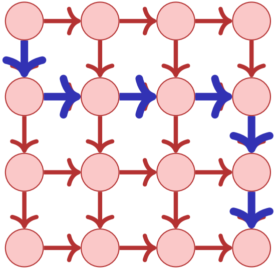


# Rabbit!

Number of possible paths?

- Choice of $n-1$ ways to south out of $2n-2$ ways overal.
- $$\binom{2n-2}{n-1} \in \Omega(2^n)$$

$\Rightarrow$ No chance for a naive algorithm



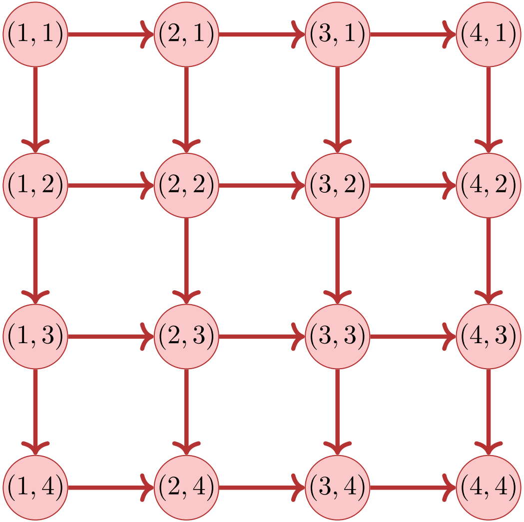
The path $100011$
(1:to south, 0: to east)

# Recursion

Wanted: $T_{0,0}$ **= maximal number carrots from** $(0,0)$ **to** $(n,n)$**.**
Let $w_{(i,j)-(i',j')}$ number of carrots on egde from $(i,j)$ to $(i',j')$.
Recursion (maximal number of carrots from $(i,j)$ to $(n,n)$

$$T_{ij} = \begin{cases} \max\{w_{(i,j)-(i,j+1)} + T_{i,j+1}, w_{(i,j)-(i+1,j)} + T_{i+1,j}\}, & i<n, j<n \\ w_{(i,j)-(i,j+1)} + T_{i,j+1}, & i=n, j<n \\ w_{(i,j)-(i+1,j)} + T_{i+1,j}, & i<n, j=n \\ 0 & i=j=n \end{cases}$$

# Graph of Subproblem Dependencies



# Bottom-up Description with the example

1. Dimension of the table? Semantics of the entries?

   Table $T$ with size $n \times n$. Entry at $i, j$ provides the maximal number of carrots from $(i, j)$ to $(n, n)$.

2. Which entries do not depend on other entries?

   Value $T_{n,n}$ is $0$

3. Computation order?

   $T_{i,j}$ with $i = n \searrow 1$ and for each $i$: $j = n \searrow 1$, (or vice-versa: $j = n \searrow 1$ and for each $j$: $i = n \searrow 1$).

4. Reconstruction of a solution?

   $T_{1,1}$ provides the maximal number of carrots.