12. Dynamic Programming

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

[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

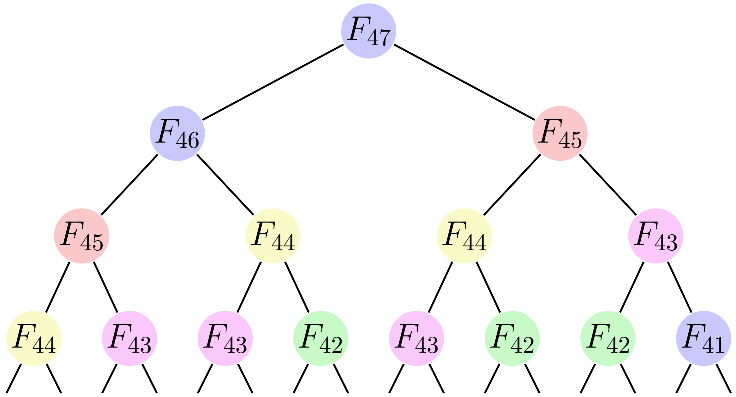
$T(n)$: Number executed operations.

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

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

Algorithm is *exponential* in n.

Reason (visual)



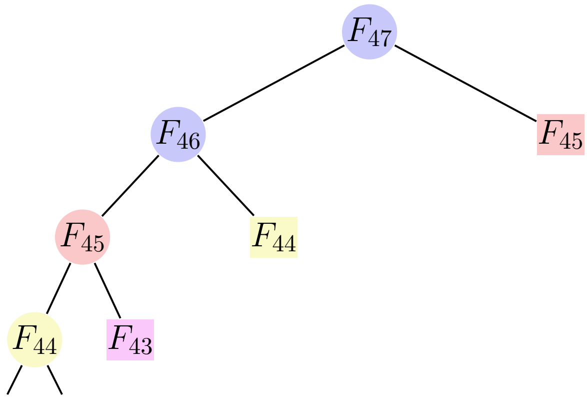
Nodes with same values are evaluated (too) often.

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.[19]

[19] 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,... 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 $F[n]$

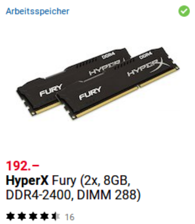
Dynamic Programming: Idea

- 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. nth entry contains nth Fibonacci number.
2. **Which entries do not depend on other entries?**
   Values F_1 and F_2 can be computed easily and independently.
3. **What is the execution order such that required entries are always available?**
   F_i with increasing i.
4. **Wie kann sich Lösung aus der Tabelle konstruieren lassen?**
   F_n ist die n-te Fibonacci-Zahl.

Dynamic Programming = Divide-And-Conquer ?

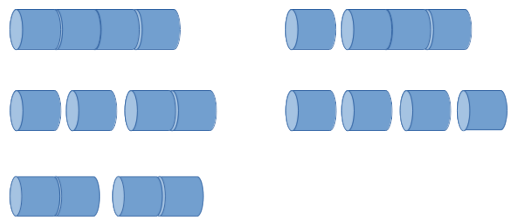
- In both cases the original problem can be solved (more easily) by utilizing the solutions of sub-problems. The problem provides *optimal substructure*.
- Divide-And-Conquer algorithms (such as Mergesort): sub-problems are independent; their solutions are required only once in the algorithm.
- DP: sub-problems 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*

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

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)

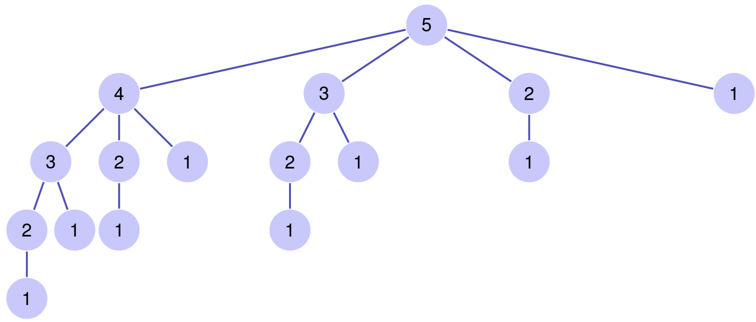
Input: $n \geq 0$, Prices v
Output: best value

$q \leftarrow 0$
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 $T(n) = \sum_{i=0}^{n-1} T(i) + c \quad \Rightarrow$[20] $\quad T(n) \in \Theta(2^n)$

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

Recursion Tree



Algorithm RodCutMemoized(m, v, n)

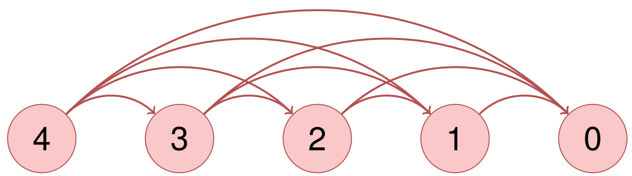
Input: $n \geq 0$, Prices v, Memoization Table m
Output: best value

$q \leftarrow 0$
if $n > 0$ **then**
  if $\exists\ m[n]$ **then**
    $q \leftarrow m[n]$
  else
    for $i \leftarrow 1, \ldots, n$ **do**
      $q \leftarrow \max\{q, v_i + \text{RodCutMemoized}(m, v, n-i)\}$;
    $m[n] \leftarrow q$
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

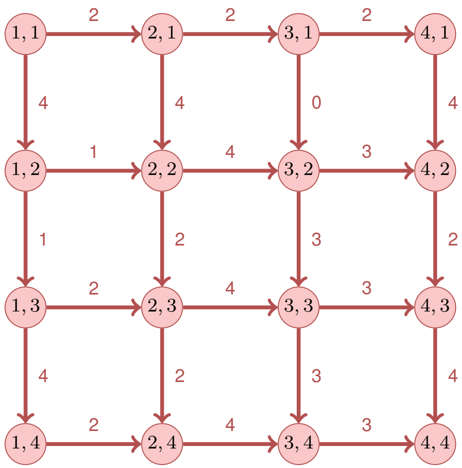
- During the (recursive) computation of the optimal solution for each $k \le n$ the recursive algorithm determines the optimal length of the first rod
- Store the lenght of the first rod in a separate table of length n

Bottom-up Description with the example

1. **Dimension of the table? Semantics of the entries?**
   $n \times 1$ table. nth 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. **What is the execution order such that required entries are always available?**
   r_i, $i = 1, \ldots, n$.
4. **Wie kann sich Lösung aus der Tabelle konstruieren lassen?**
   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?

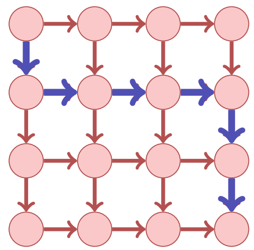


Rabbit!

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

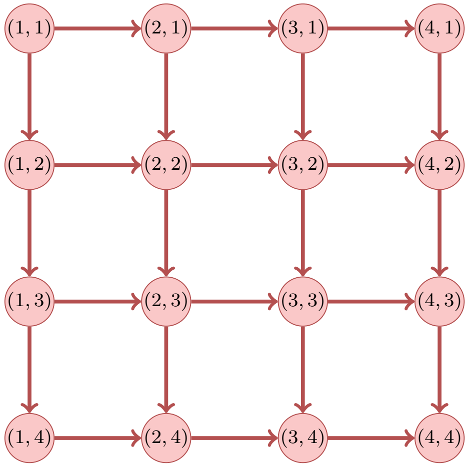
Wanted: $T_{0,0}$ = *maximal number carrots from* $(0,0)$ *to* (n,n).

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))

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

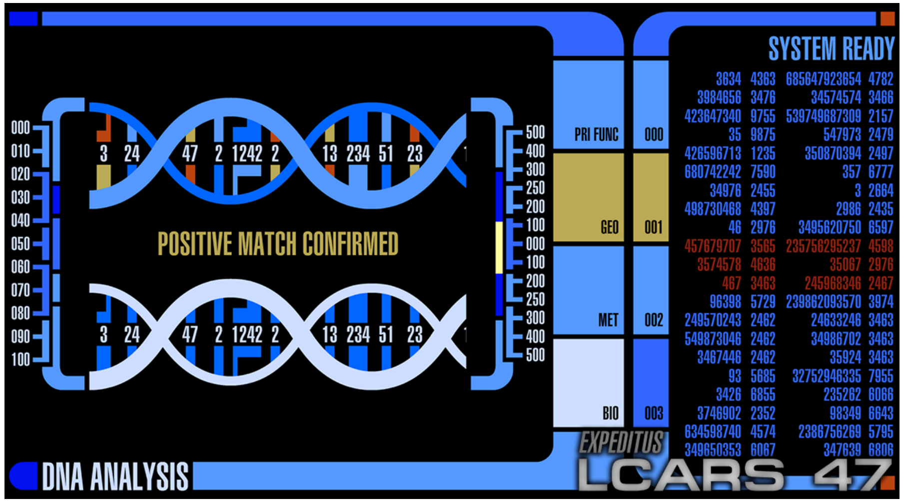
Graph of Subproblem Dependencies



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. **What is the execution order such that required entries are always available?**
   $T_{i,j}$ 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. **Wie kann sich Lösung aus der Tabelle konstruieren lassen?**
   $T_{1,1}$ provides the maximal number of carrots.

DNA - Comparison (Star Trek)



DNA - Comparison

- DNA consists of sequences of four different nucleotides **A**denine **G**uanine **T**hymine **C**ytosine
- DNA sequences (genes) thus can be described with strings of A, G, T and C.
- Possible comparison of two genes: determine the **longest common subsequence**

The longest common subsequence problem is a special case of the minimal edit distance problem. The following slides are therefore not presented in the lectures.

[Longest common subsequence]

Subsequences of a string:

Subsequences(KUH): (), (K), (U), (H), (KU), (KH), (UH), (KUH)

Problem:

- Input: two strings $A = (a_1, \ldots, a_m)$, $B = (b_1, \ldots, b_n)$ with lengths $m > 0$ and $n > 0$.
- Wanted: Longest common subsequecnes (LCS) of A and B.

[Longest Common Subsequence]

Examples:

LGT(IGEL,KATZE)=E, LGT(TIGER,ZIEGE)=IGE

Ideas to solve?

T	I		G	E	R
Z	I	E	G	E	

[Recursive Procedure]

Assumption: solutions $L(i, j)$ known for $A[1, \ldots, i]$ and $B[1, \ldots, j]$ for all $1 \leq i \leq m$ and $1 \leq j \leq n$, but not for $i = m$ and $j = n$.

T	I		G	E	R
Z	I	E	G	E	

Consider characters a_m, b_n. Three possibilities:

1. A is enlarged by one whitespace. $L(m, n) = L(m, n-1)$
2. B is enlarged by one whitespace. $L(m, n) = L(m-1, n)$
3. $L(m, n) = L(m-1, n-1) + \delta_{mn}$ with $\delta_{mn} = 1$ if $a_m = b_n$ and $\delta_{mn} = 0$ otherwise

[Recursion]

$$L(m,n) \leftarrow \max\{L(m-1,n-1)+\delta_{mn}, L(m,n-1), L(m-1,n)\}$$

for $m,n>0$ and base cases $L(\cdot,0)=0$, $L(0,\cdot)=0$.

	∅	Z	I	E	G	E
∅	0	0	0	0	0	0
T	0	0	0	0	0	0
I	0	0	1	1	1	1
G	0	0	1	1	2	2
E	0	0	1	2	2	3
R	0	0	1	2	2	3

[Dynamic Programming algorithm LCS]

1 **Dimension of the table? Semantics?**
Table $L[0,\dots,m][0,\dots,n]$. $L[i,j]$: length of a LCS of the strings $(a_1,\dots,a_i)$ and $(b_1,\dots,b_j)$

2 **Computation of an entry**
$L[0,i] \leftarrow 0\ \forall 0\le i\le m$, $L[j,0]\leftarrow 0\ \forall 0\le j\le n$. Computation of $L[i,j]$ otherwise via $L[i,j]=\max(L[i-1,j-1]+\delta_{ij}, L[i,j-1], L[i-1,j])$.

[Dynamic Programming algorithm LCS]

3 **Computation order**
Rows increasing and within columns increasing (or the other way round).

4 **Reconstruct solution?**
Start with $j=m$, $i=n$. If $a_i=b_j$ then output a_i and continue with $(j,i)\leftarrow(j-1,i-1)$; otherwise, if $L[i,j]=L[i,j-1]$ continue with $j\leftarrow j-1$ otherwise, if $L[i,j]=L[i-1,j]$ continue with $i\leftarrow i-1$. Terminate for $i=0$ or $j=0$.

[Analysis LCS]

- Number table entries: $(m+1)\cdot(n+1)$.
- Constant number of assignments and comparisons each. Number steps: $\mathcal{O}(mn)$
- Determination of solition: decrease i or j. Maximally $\mathcal{O}(n+m)$ steps.

Runtime overal:

$$\mathcal{O}(mn).$$

Minimal Editing Distance

Editing distance of two sequences $A_n = (a_1, \ldots, a_m)$, $B_m = (b_1, \ldots, b_m)$.

Editing operations:

- Insertion of a character
- Deletion of a character
- Replacement of a character

Question: how many editing operations at least required in order to transform string A into string B.

TIGER ZIGER ZIEGER ZIEGE

Minimal Editing Distance

Wanted: cheapest character-wise transformation $A_n \to B_m$ with costs

operation	Levenshtein	LCS[21]	general
Insert c	1	1	ins(c)
Delete c	1	1	del(c)
Replace $c \to c'$	$\mathbb{1}(c \neq c')$	$\infty \cdot \mathbb{1}(c \neq c')$	repl(c, c')

Beispiel

T I G E R	T I _ G E R	T→Z +E -R
Z I E G E	Z I E G E _	Z→T -E +R

[21] Longest common subsequence – A special case of an editing problem

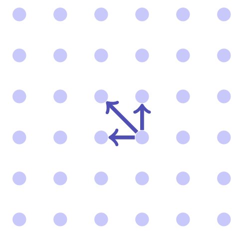
DP

0. $E(n, m)$ = mimimum number edit operations (ED cost) $a_{1\ldots n} \to b_{1\ldots m}$
1. Subproblems $E(i, j)$ = ED von $a_{1\ldots i}$. $b_{1\ldots j}$. #SP $= n \cdot m$
2. Guess Costs$\Theta(1)$
   - $a_{1..i} \to a_{1\ldots i-1}$ (delete)
   - $a_{1..i} \to a_{1\ldots i} b_j$ (insert)
   - $a_{1..i} \to a_{1\ldots i_1} b_j$ (replace)
3. Rekursion

$$E(i, j) = \min \begin{cases} \text{del}(a_i) + E(i-1, j), \\ \text{ins}(b_j) + E(i, j-1), \\ \text{repl}(a_i, b_j) + E(i-1, j-1) \end{cases}$$

DP

4. Dependencies



⇒ Computation from left top to bottom right. Row- or column-wise.

5. Solution in $E(n, m)$

Example (Levenshtein Distance)

$$E[i,j] \leftarrow \min\left\{E[i-1,j]+1, E[i,j-1]+1, E[i-1,j-1]+\mathbb{1}(a_i \neq b_j)\right\}$$

	∅	Z	I	E	G	E
∅	0	1	2	3	4	5
T	1	1	2	3	4	5
I	2	2	1	2	3	4
G	3	3	2	2	2	3
E	4	4	3	2	3	2
R	5	5	4	3	3	3

Editing steps: from bottom right to top left, following the recursion.
Bottom-Up description of the algorithm: exercise

Bottom-Up DP algorithm ED]

1 **Dimension of the table? Semantics?**

Table $E[0,\dots,m][0,\dots,n]$. $E[i,j]$: minimal edit distance of the strings $(a_1,\dots,a_i)$ and $(b_1,\dots,b_j)$

2 **Computation of an entry**

$E[0,i] \leftarrow i \;\forall 0 \leq i \leq m$, $E[j,0] \leftarrow i \;\forall 0 \leq j \leq n$. Computation of $E[i,j]$ otherwise via $E[i,j] =$
$\min\{\text{del}(a_i) + E(i-1,j), \text{ins}(b_j) + E(i,j-1), \text{repl}(a_i,b_j) + E(i-1,j-1)\}$

Bottom-Up DP algorithm ED

3 **Computation order**

Rows increasing and within columns increasing (or the other way round).

4 **Reconstruct solution?**

Start with $j = m$, $i = n$. If $E[i,j] = \text{repl}(a_i,b_j) + E(i-1,j-1)$ then output $a_i \rightarrow b_j$ and continue with $(j,i) \leftarrow (j-1,i-1)$; otherwise, if $E[i,j] = \text{del}(a_i) + E(i-1,j)$ output del(a_i) and continue with $j \leftarrow j-1$ otherwise, if $E[i,j] = \text{ins}(b_j) + E(i,j-1)$, continue with $i \leftarrow i-1$.
Terminate for $i = 0$ and $j = 0$.