HPI Hasso Plattner Institut
Digital Engineering · Universität Potsdam

MOIA

KIT
Karlsruhe Institute of Technology

Algorithm Bootcamp

July 2024

Day 3: Dynamic Programming

Dr. Christopher Weyand
Optimization Expert
Fleet Optimization

David Stangl
Software Engineer
Fleet Optimization

Greedy vs. Dynamic Programming

Greedy vs. Dynamic Programming

- Greedy algorithms always choose the next best option

Greedy vs. Dynamic Programming

- Greedy algorithms always choose the next best option
- Greedy algorithms look temptingly correct

Greedy vs. Dynamic Programming

- Greedy algorithms always choose the next best option
- Greedy algorithms look temptingly correct ... but are mostly plain wrong!

Greedy vs. Dynamic Programming

- Greedy algorithms always choose the next best option
- Greedy algorithms look temptingly correct ... but are mostly plain wrong!

> Never trust a greedy algorithm!

Greedy vs. Dynamic Programming

- Greedy algorithms always choose the next best option
- Greedy algorithms look temptingly correct ... but are mostly plain wrong!

Never trust a greedy algorithm!

- better: Use Dynamic Programming instead!

Greedy vs. Dynamic Programming

- Greedy algorithms always choose the next best option
- Greedy algorithms look temptingly correct ... but are mostly plain wrong!

Never trust a greedy algorithm!

- better: Use Dynamic Programming instead!

When is Dynamic Programming the right choice?

Greedy vs. Dynamic Programming

- Greedy algorithms always choose the next best option
- Greedy algorithms look temptingly correct ... but are mostly plain wrong!

Never trust a greedy algorithm!

- better: **Use Dynamic Programming instead!**

When is Dynamic Programming the right choice?

- optimal solution consists of optimal solutions for sub-problems

"Bellman's Optimality Principle"

Greedy vs. Dynamic Programming

- Greedy algorithms always choose the next best option
- Greedy algorithms look temptingly correct ... but are mostly plain wrong!

Never trust a greedy algorithm!

- better: Use Dynamic Programming instead!

When is Dynamic Programming the right choice?

- optimal solution consists of optimal solutions for sub-problems

  "Bellman's Optimality Principle"

- many overlapping sub-problems

  if no overlap: use Divide & Conquer

Dynamic Programming

Never trust a greedy algorithm!

Use Dynamic Programming instead!

Dynamic Programming

Never trust a greedy algorithm!

Use Dynamic Programming instead!

Idea: use clever recursion to solve your problem

Dynamic Programming

Never trust a greedy algorithm!

Use Dynamic Programming instead!

Idea: use clever recursion to solve your problem

use a recursive algorithm

Dynamic Programming

Never trust a greedy algorithm!

Use Dynamic Programming instead!

Idea: use clever recursion to solve your problem

store solutions to sub-problems
to avoid re-computation

use a recursive algorithm

Dynamic Programming

Never trust a greedy algorithm!

Use Dynamic Programming instead!

Idea: use clever recursion to solve your problem

store solutions to sub-problems
to avoid re-computation

use a recursive algorithm

3 steps for doing it:

Dynamic Programming

Never trust a greedy algorithm!

Use Dynamic Programming instead!

Idea: use clever recursion to solve your problem

store solutions to sub-problems to avoid re-computation

use a recursive algorithm

3 steps for doing it:

1.) come up with a recursive approach to solve your problem

Dynamic Programming

Never trust a greedy algorithm!

Use Dynamic Programming instead!

Idea: use clever recursion to solve your problem

store solutions to sub-problems
to avoid re-computation

use a recursive algorithm

3 steps for doing it:

1.) come up with a recursive approach to solve your problem

2.) store the solutions of sub-problems to avoid future recursive calls

Dynamic Programming

Never trust a greedy algorithm!

Use Dynamic Programming instead!

Idea: use clever recursion to solve your problem

store solutions to sub-problems to avoid re-computation

use a recursive algorithm

3 steps for doing it:

1.) come up with a recursive approach to solve your problem

2.) store the solutions of sub-problems to avoid future recursive calls

3.) optional: generate solutions to sub-problems in a bottom-up fashion to get an iterative algorithm

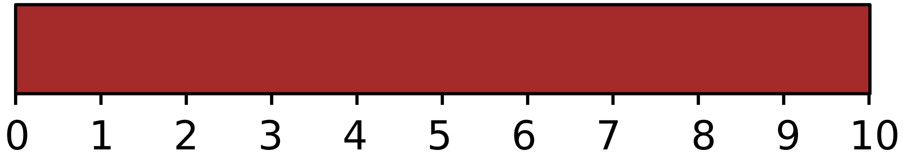
Example 1: Rod Cutting

Task: Rod Cutting
Input: rod of length n, prices for different lengths of the rod
Task: find the optimal way to cut the rod to maximize the profit

Example 1: Rod Cutting

Task: Rod Cutting
Input: rod of length n, prices for different lengths of the rod
Task: find the optimal way to cut the rod to maximize the profit

Input:

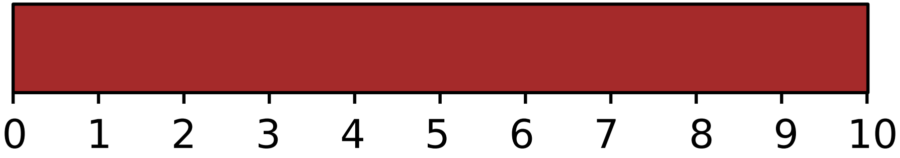


length	1	2	3	4	5	6	7	8	9	10
price	0	1	3	5	6	3	7	9	7	10

Example 1: Rod Cutting

Task: Rod Cutting
Input: rod of length n, prices for different lengths of the rod
Task: find the optimal way to cut the rod to maximize the profit

Input:



length	1	2	3	4	5	6	7	8	9	10
price	0	1	3	5	6	3	7	9	7	10

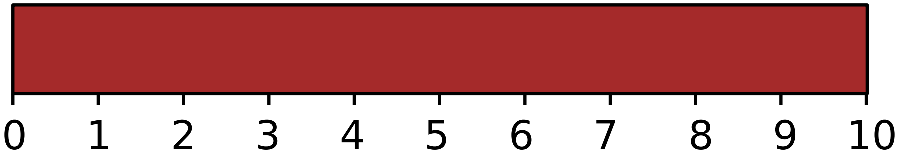
naïve approach:

- try all combinations of cuts

Example 1: Rod Cutting

Task: Rod Cutting
Input: rod of length n, prices for different lengths of the rod
Task: find the optimal way to cut the rod to maximize the profit

Input:



length	1	2	3	4	5	6	7	8	9	10
price	0	1	3	5	6	3	7	9	7	10

naïve approach:

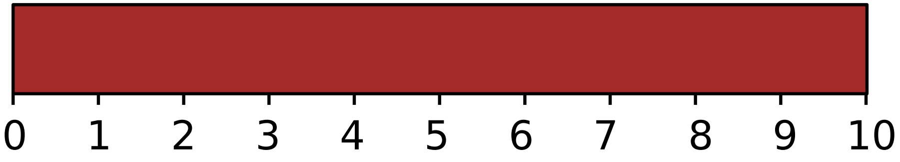
- try all combinations of cuts
- do this recursively:

recursiveCut(n):
    for $1 \leq i \leq n$:
        cut piece of length i
        recursiveCut($n - i$)

Example 1: Rod Cutting

Task: Rod Cutting
Input: rod of length n, prices for different lengths of the rod
Task: find the optimal way to cut the rod to maximize the profit

Input:



length	1	2	3	4	5	6	7	8	9	10
price	0	1	3	5	6	3	7	9	7	10

naïve approach:

- try all combinations of cuts
- do this recursively:

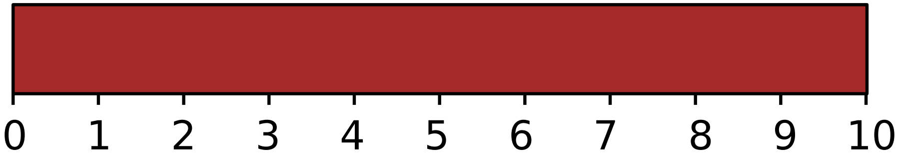
  recursiveCut(n):
  for $1 \leq i \leq n$:
  cut piece of length i
  recursiveCut($n - i$)

- output best solution

Example 1: Rod Cutting

Task: Rod Cutting
Input: rod of length n, prices for different lengths of the rod
Task: find the optimal way to cut the rod to maximize the profit

Input:



length	1	2	3	4	5	6	7	8	9	10
price	0	1	3	5	6	3	7	9	7	10

naïve approach:

- try all combinations of cuts
- do this recursively:

```
recursiveCut(n):
    for 1 ≤ i ≤ n:
        cut piece of length i
        recursiveCut(n − i)
```

- output best solution

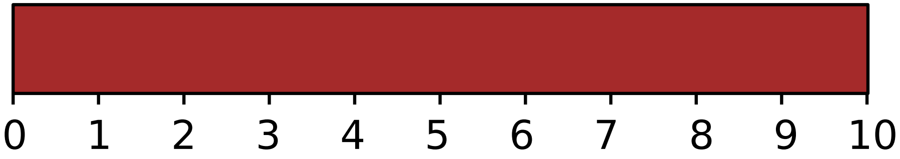
better recursive approach:

- try all possible first cuts, then use the optimal solutions for smaller rod lengths

Example 1: Rod Cutting

Task: Rod Cutting
Input: rod of length n, prices for different lengths of the rod
Task: find the optimal way to cut the rod to maximize the profit

Input:



length	1	2	3	4	5	6	7	8	9	10
price	0	1	3	5	6	3	7	9	7	10

naïve approach:

- try all combinations of cuts
- do this recursively:

```
recursiveCut(n):
    for 1 ≤ i ≤ n:
        cut piece of length i
        recursiveCut(n − i)
```

- output best solution

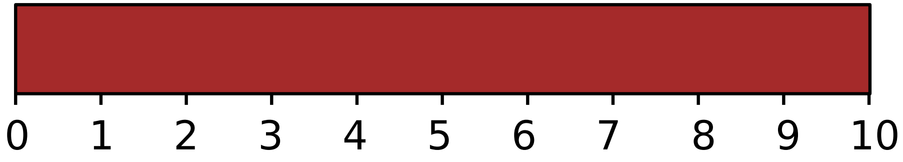
better recursive approach:

- try all possible first cuts, then use the optimal solutions for smaller rod lengths

let $optCut(\ell)$ be the highest possible profit for a rod of length ℓ

Example 1: Rod Cutting

Task: Rod Cutting
Input: rod of length n, prices for different lengths of the rod
Task: find the optimal way to cut the rod to maximize the profit



length	1	2	3	4	5	6	7	8	9	10
price	0	1	3	5	6	3	7	9	7	10

naïve approach:

- try all combinations of cuts
- do this recursively:

```
recursiveCut(n):
    for 1 ≤ i ≤ n:
        cut piece of length i
        recursiveCut(n − i)
```

- output best solution

better recursive approach:

- try all possible first cuts, then use the optimal solutions for smaller rod lengths

let $optCut(\ell)$ be the highest possible profit for a rod of length ℓ

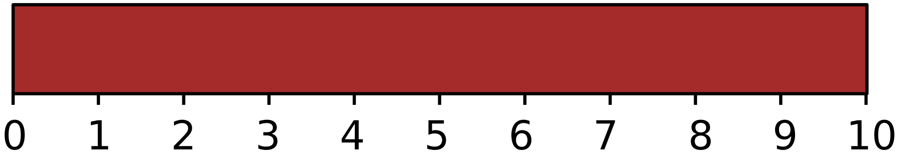
$$optCut(\ell) = \max_{1 \leq i \leq \ell} \left(profit[i] + optcut(\ell - i)\right)$$

$$optCut(0) = 0$$

Example 1: Rod Cutting

Task: Rod Cutting
Input: rod of length n, prices for different lengths of the rod
Task: find the optimal way to cut the rod to maximize the profit

Input:



length	1	2	3	4	5	6	7	8	9	10
price	0	1	3	5	6	3	7	9	7	10

naïve approach:

- try all combinations of cuts
- do this recursively:

```
recursiveCut(n):
    for 1 ≤ i ≤ n:
        cut piece of length i
        recursiveCut(n − i)
```

- output best solution

better recursive approach:

- try all possible first cuts, then use the optimal solutions for smaller rod lengths

let $optCut(\ell)$ be the highest possible profit for a rod of length ℓ

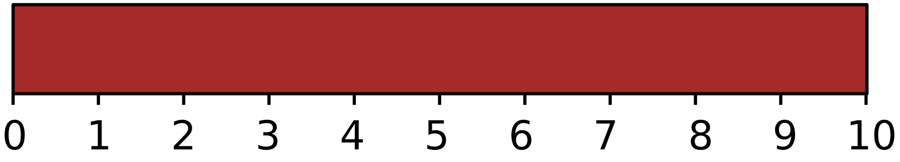
Idea: store $optCut(\ell)$ once it is computed

$$optCut(\ell) = \max_{1 \leq i \leq \ell} \left(profit[i] + optcut(\ell - i) \right)$$

$$optCut(0) = 0$$

Example 1: Rod Cutting

Task: Rod Cutting
Input: rod of length n, prices for different lengths of the rod
Task: find the optimal way to cut the rod to maximize the profit

Input: 

length	1	2	3	4	5	6	7	8	9	10
price	0	1	3	5	6	3	7	9	7	10

naïve approach:

- try all combinations of cuts
- do this recursively:

  recursiveCut(n):
  for $1 \leq i \leq n$:
  cut piece of length i
  recursiveCut($n - i$)
- output best solution

exponential time

better recursive approach:

- try all possible first cuts, then use the optimal solutions for smaller rod lengths

let $optCut(\ell)$ be the highest possible profit for a rod of length ℓ

Idea: store $optCut(\ell)$ once it is computed

$$optCut(\ell) = \max_{1 \leq i \leq \ell} \left(profit[i] + optcut(\ell - i) \right)$$

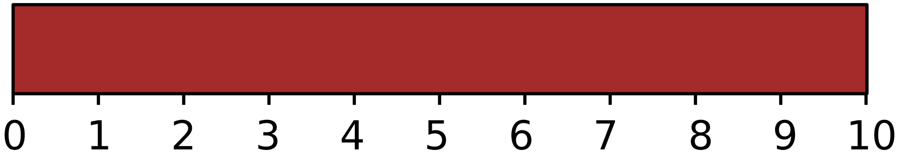
$$optCut(0) = 0$$

$\mathcal{O}(n^2)$ time

Example 1: Rod Cutting

Task: Rod Cutting
Input: rod of length n, prices for different lengths of the rod
Task: find the optimal way to cut the rod to maximize the profit

Input:



length	1	2	3	4	5	6	7	8	9	10
price	0	1	3	5	6	3	7	9	7	10

naïve approach:

- try all combinations of cuts
- do this recursively:

  recursiveCut(n):
      for $1 \leq i \leq n$:
          cut piece of length i
          recursiveCut($n - i$)
- output best solution

exponential time

why?

better recursive approach:

- try all possible first cuts, then use the optimal solutions for smaller rod lengths

let $optCut(\ell)$ be the highest possible profit for a rod of length ℓ

Idea: store $optCut(\ell)$ once it is computed

$$optCut(\ell) = \max_{1 \leq i \leq \ell} \left(profit[i] + optcut(\ell - i)\right)$$

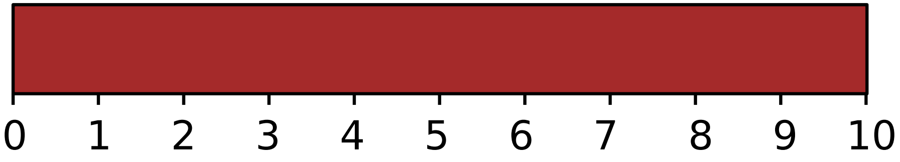
$$optCut(0) = 0$$

$\mathcal{O}(n^2)$ time

Example 1: Rod Cutting

Task: Rod Cutting
Input: rod of length n, prices for different lengths of the rod
Task: find the optimal way to cut the rod to maximize the profit

Input:



length	1	2	3	4	5	6	7	8	9	10
price	0	1	3	5	6	3	7	9	7	10

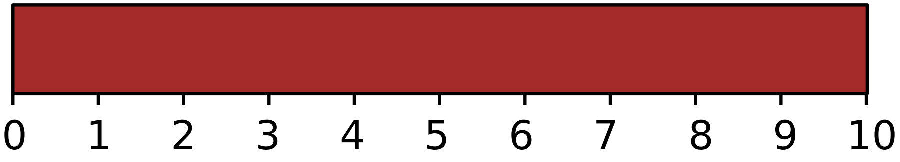
naïve approach:

better recursive approach:

Example 1: Rod Cutting

Task: Rod Cutting
Input: rod of length n, prices for different lengths of the rod
Task: find the optimal way to cut the rod to maximize the profit

Input:



length	1	2	3	4	5	6	7	8	9	10
price	0	1	3	5	6	3	7	9	7	10

naïve approach:

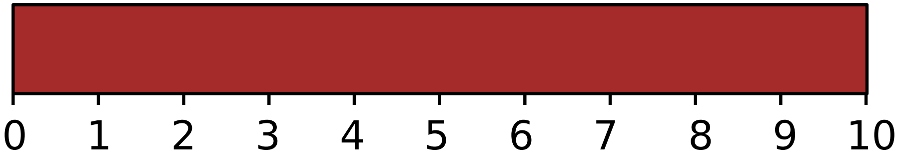
better recursive approach:

(n)

Example 1: Rod Cutting

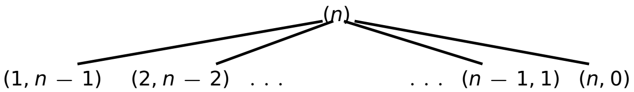
Task: Rod Cutting
Input: rod of length n, prices for different lengths of the rod
Task: find the optimal way to cut the rod to maximize the profit

Input:



length	1	2	3	4	5	6	7	8	9	10
price	0	1	3	5	6	3	7	9	7	10

naïve approach:

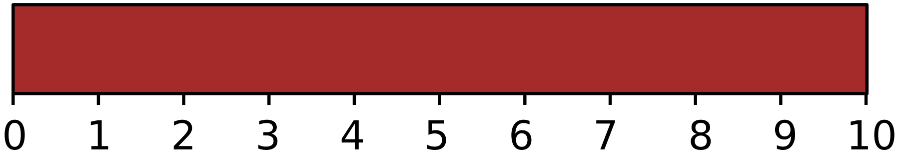


better recursive approach:

Example 1: Rod Cutting

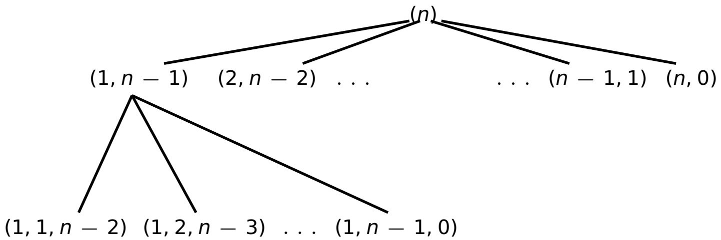
Task: Rod Cutting
Input: rod of length n, prices for different lengths of the rod
Task: find the optimal way to cut the rod to maximize the profit

Input:



length	1	2	3	4	5	6	7	8	9	10
price	0	1	3	5	6	3	7	9	7	10

naïve approach:

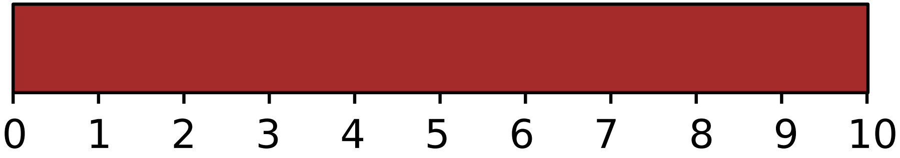


better recursive approach:

Example 1: Rod Cutting

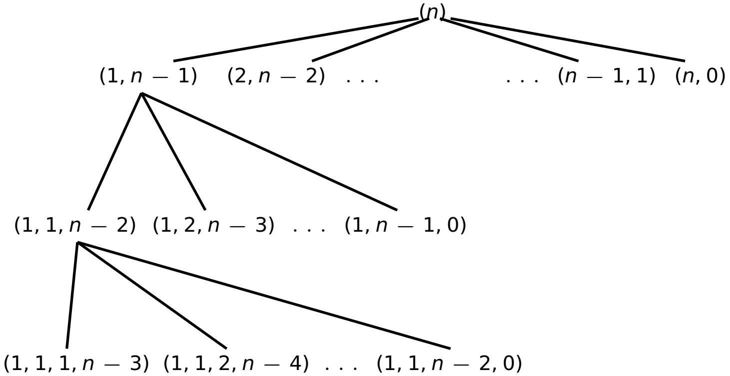
Task: Rod Cutting
Input: rod of length n, prices for different lengths of the rod
Task: find the optimal way to cut the rod to maximize the profit

Input:



length	1	2	3	4	5	6	7	8	9	10
price	0	1	3	5	6	3	7	9	7	10

naïve approach:

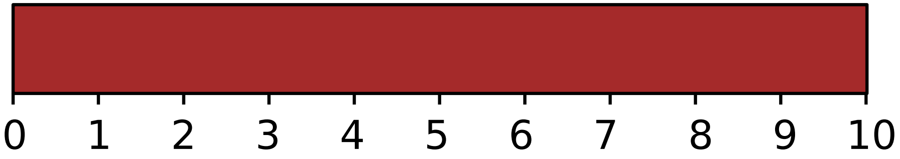


better recursive approach:

Example 1: Rod Cutting

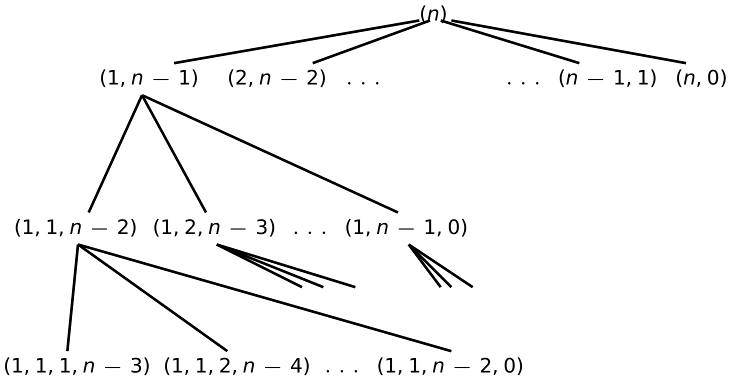
Task: Rod Cutting
Input: rod of length n, prices for different lengths of the rod
Task: find the optimal way to cut the rod to maximize the profit

Input:



length	1	2	3	4	5	6	7	8	9	10
price	0	1	3	5	6	3	7	9	7	10

naïve approach:

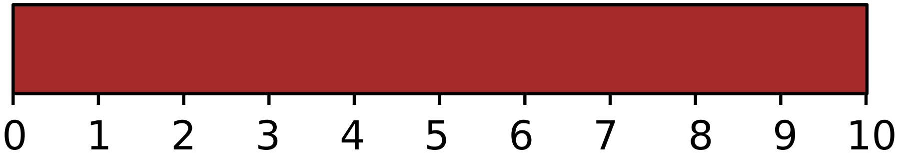


better recursive approach:

Example 1: Rod Cutting

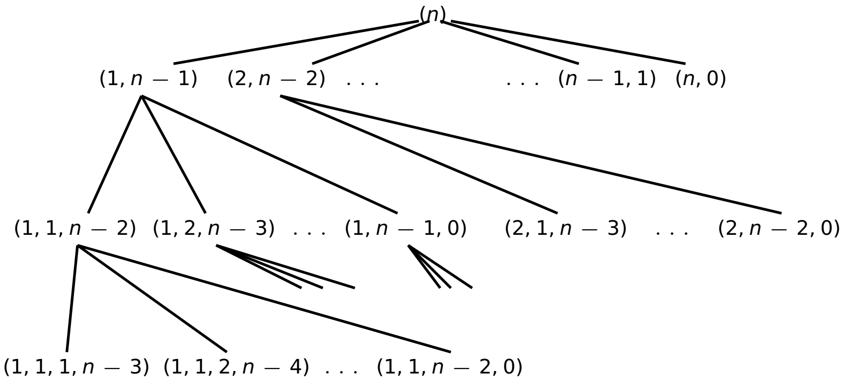
Task: Rod Cutting
Input: rod of length n, prices for different lengths of the rod
Task: find the optimal way to cut the rod to maximize the profit

Input:



length	1	2	3	4	5	6	7	8	9	10
price	0	1	3	5	6	3	7	9	7	10

naïve approach:

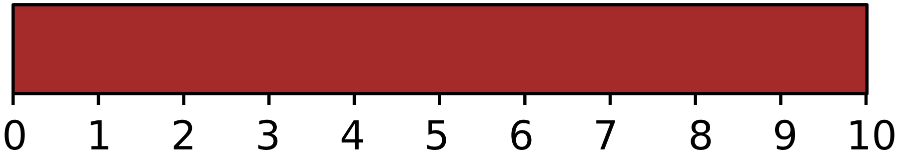


better recursive approach:

Example 1: Rod Cutting

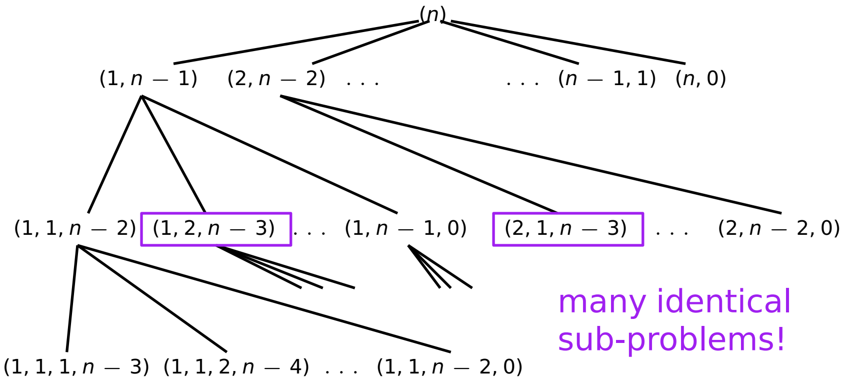
Task: Rod Cutting
Input: rod of length n, prices for different lengths of the rod
Task: find the optimal way to cut the rod to maximize the profit

Input:



length	1	2	3	4	5	6	7	8	9	10
price	0	1	3	5	6	3	7	9	7	10

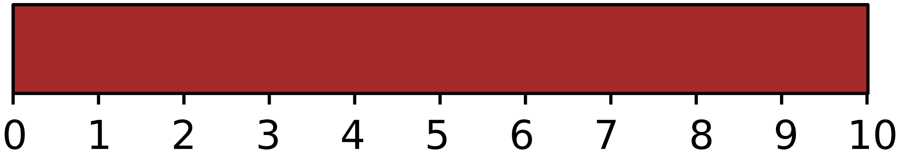
naïve approach:



better recursive approach:

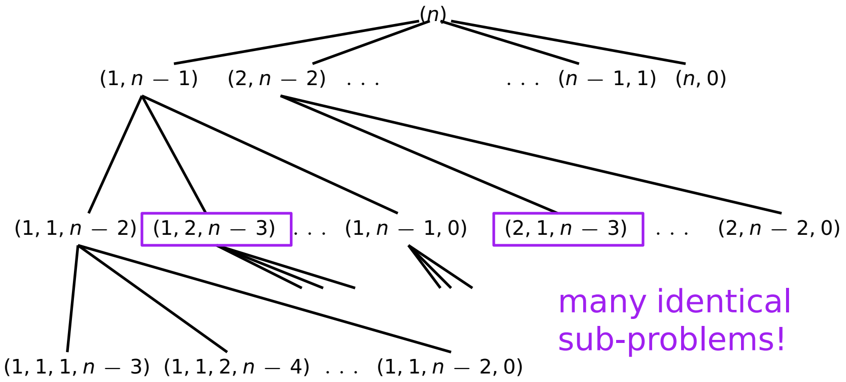
Example 1: Rod Cutting

Task: Rod Cutting
Input: rod of length n, prices for different lengths of the rod
Task: find the optimal way to cut the rod to maximize the profit

Input: 

length	1	2	3	4	5	6	7	8	9	10
price	0	1	3	5	6	3	7	9	7	10

naïve approach:



better recursive approach:

let $optCut(\ell)$ be the highest possible profit for a rod of length ℓ

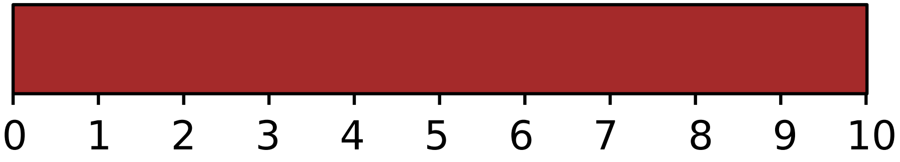
$$optCut(\ell) = \max_{1 \le i \le \ell} \left(profit[i] + optcut(\ell - i) \right)$$

$$optCut(0) = 0$$

Example 1: Rod Cutting

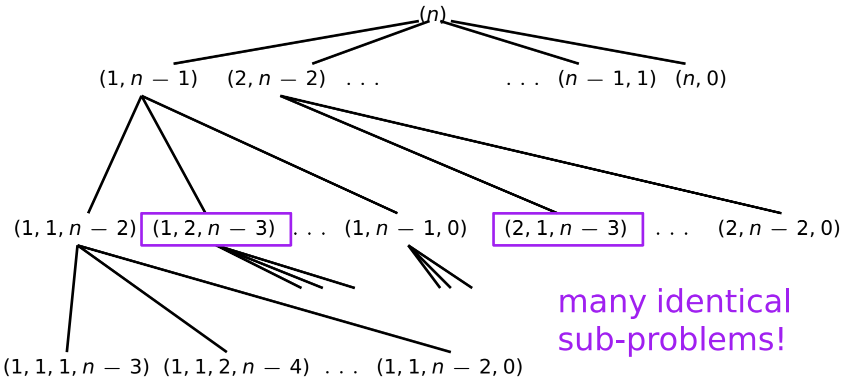
Task: Rod Cutting
Input: rod of length n, prices for different lengths of the rod
Task: find the optimal way to cut the rod to maximize the profit

Input:



length	1	2	3	4	5	6	7	8	9	10
price	0	1	3	5	6	3	7	9	7	10

naïve approach:



many identical sub-problems!

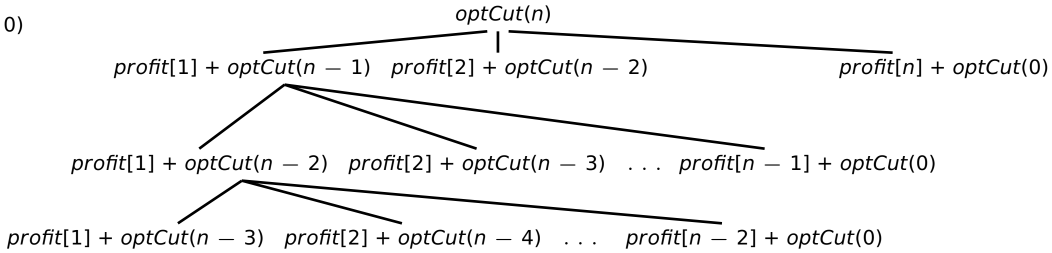
better recursive approach:

let $optCut(\ell)$ be the highest possible profit for a rod of length ℓ

$$optCut(\ell) = \max_{1 \le i \le \ell} \left(profit[i] + optcut(\ell - i) \right)$$

$$optCut(0) = 0$$

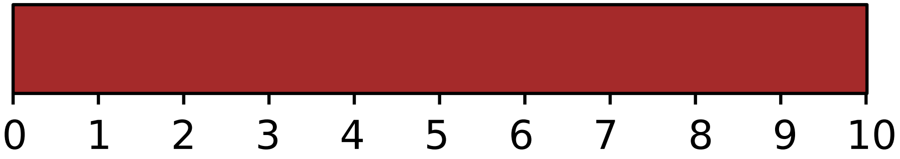
Idea: store $optCut(\ell)$ once it is computed



Example 1: Rod Cutting

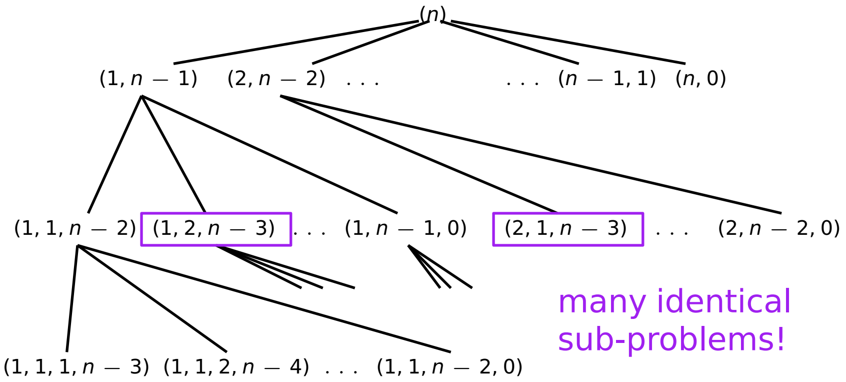
Task: Rod Cutting
Input: rod of length n, prices for different lengths of the rod
Task: find the optimal way to cut the rod to maximize the profit

Input:



length	1	2	3	4	5	6	7	8	9	10
price	0	1	3	5	6	3	7	9	7	10

naïve approach:



many identical sub-problems!

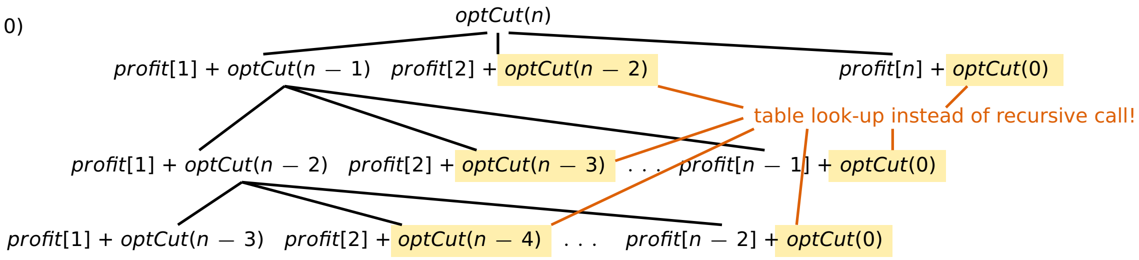
better recursive approach:

let $optCut(\ell)$ be the highest possible profit for a rod of length ℓ

$$optCut(\ell) = \max_{1 \le i \le \ell} \left(profit[i] + optcut(\ell - i) \right)$$

$$optCut(0) = 0$$

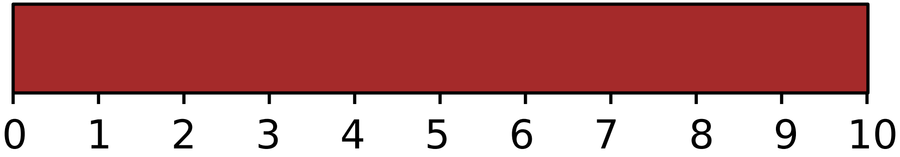
Idea: store $optCut(\ell)$ once it is computed



Example 1: Rod Cutting

Task: Rod Cutting
Input: rod of length n, prices for different lengths of the rod
Task: find the optimal way to cut the rod to maximize the profit

Input:



length	1	2	3	4	5	6	7	8	9	10
price	0	1	3	5	6	3	7	9	7	10

iterative bottom-up approach: ⟵ better recursive approach:

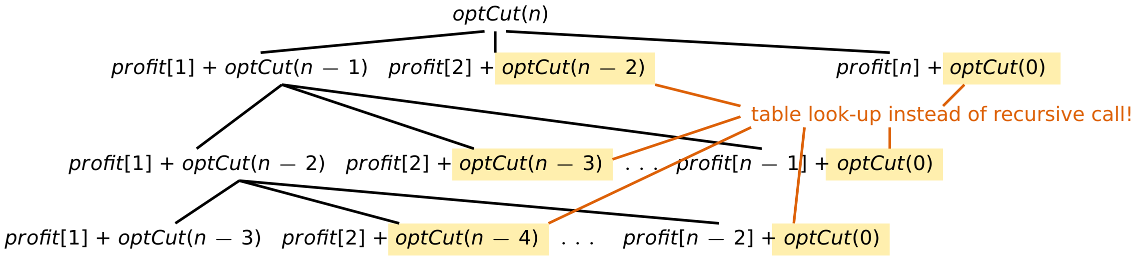
```
optCut = [0]*(n+1)
for l in range(n+1):
    for i in range(1,l+1):
        optCut[l] = max(optCut[l],
            profit[i] + optCut[l-i])
```

let $optCut(\ell)$ be the highest possible profit for a rod of length ℓ

$$optCut(\ell) = \max_{1 \leq i \leq \ell} (profit[i] + optcut(\ell - i))$$

$$optCut(0) = 0$$

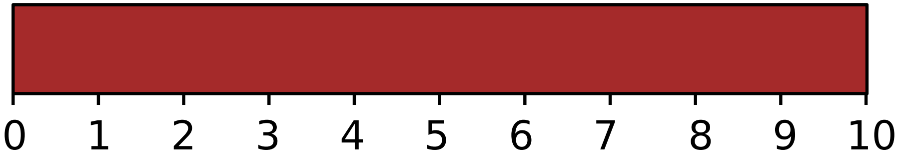
Idea: store $optCut(\ell)$ once it is computed



Example 1: Rod Cutting

Task: Rod Cutting
Input: rod of length n, prices for different lengths of the rod
Task: find the optimal way to cut the rod to maximize the profit

Input:



length	1	2	3	4	5	6	7	8	9	10
price	0	1	3	5	6	3	7	9	7	10

iterative bottom-up approach: ← better recursive approach:

let $optCut(\ell)$ be the highest possible profit for a rod of length ℓ

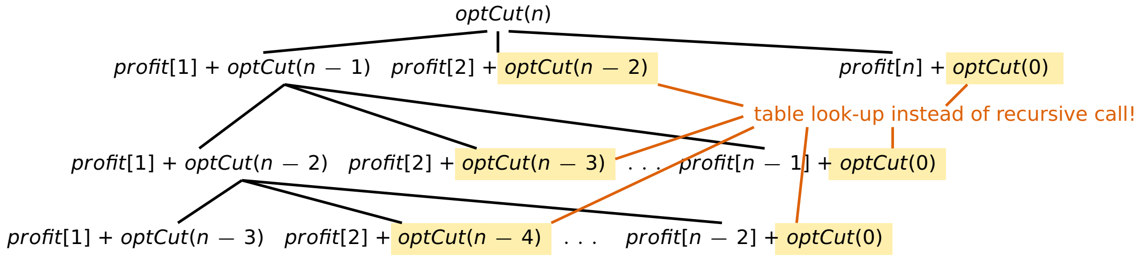
```
optCut = [0]*(n+1)
for l in range(n+1):
    for i in range(1,l+1):
        optCut[l] = max(optCut[l],
            profit[i] + optCut[l-i])
```

table look-up

$$optCut(\ell) = \max_{1 \leq i \leq \ell} (profit[i] + optcut(\ell - i))$$

$$optCut(0) = 0$$

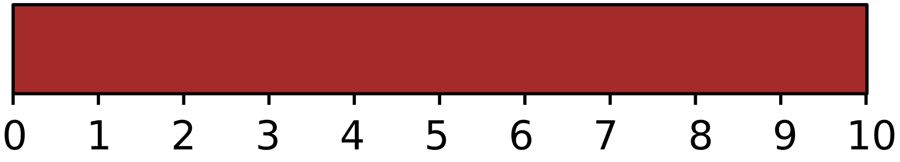
Idea: store $optCut(\ell)$ once it is computed



Example 1: Rod Cutting

Task: Rod Cutting
Input: rod of length n, prices for different lengths of the rod
Task: find the optimal way to cut the rod to maximize the profit

Input:



length	1	2	3	4	5	6	7	8	9	10
price	0	1	3	5	6	3	7	9	7	10

iterative bottom-up approach: ← better recursive approach:

let $optCut(\ell)$ be the highest possible profit for a rod of length ℓ

$$optCut(\ell) = \max_{1 \leq i \leq \ell} (profit[i] + optcut(\ell - i))$$

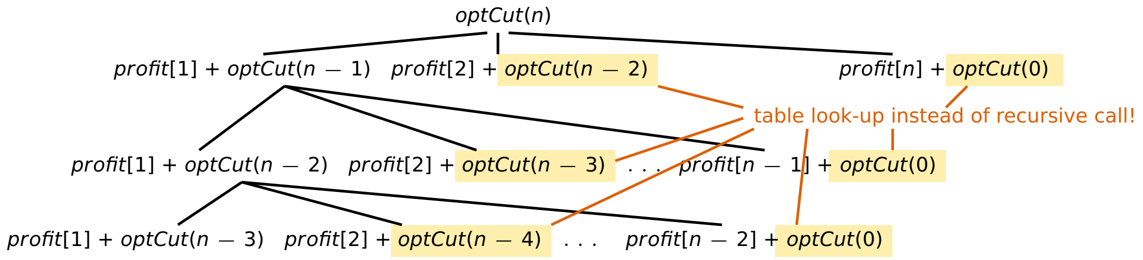
$$optCut(0) = 0$$

Idea: store $optCut(\ell)$ once it is computed

```
optCut = [0]*(n+1)
for l in range(n+1):
    for i in range(1,l+1):
        optCut[l] = max(optCut[l],
            profit[i] + optCut[l-i])
```

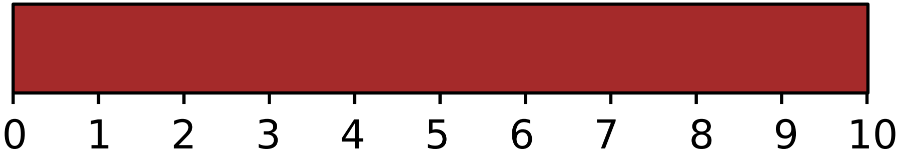
table look-up

length	0	1	2	3	4	5	6	7	8	9	10
optCut	0										



Example 1: Rod Cutting

> **Task: Rod Cutting**
> Input: rod of length n, prices for different lengths of the rod
> Task: find the optimal way to cut the rod to maximize the profit

Input: 

length	1	2	3	4	5	6	7	8	9	10
price	0	1	3	5	6	3	7	9	7	10

iterative bottom-up approach: ← better recursive approach:

```
optCut = [0]*(n+1)
for l in range(n+1):
    for i in range(1,l+1):
        optCut[l] = max(optCut[l],
            profit[i] + optCut[l-i])
```

table look-up

length	0	1	2	3	4	5	6	7	8	9	10
optCut	0	0									

let $optCut(\ell)$ be the highest possible profit for a rod of length ℓ

$$optCut(\ell) = \max_{1 \leq i \leq \ell} (profit[i] + optcut(\ell - i))$$

$$optCut(0) = 0$$

Idea: store $optCut(\ell)$ once it is computed

$optCut(n)$

$profit[1] + optCut(n-1)$ $profit[2] + optCut(n-2)$ $profit[n] + optCut(0)$

table look-up instead of recursive call!

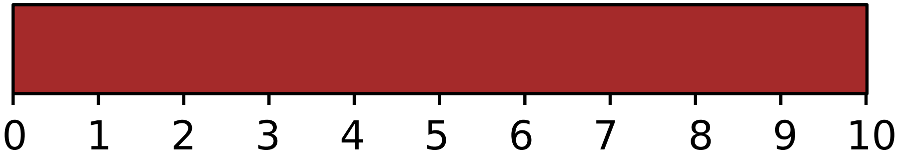
$profit[1] + optCut(n-2)$ $profit[2] + optCut(n-3)$ ... $profit[n-1] + optCut(0)$

$profit[1] + optCut(n-3)$ $profit[2] + optCut(n-4)$ ... $profit[n-2] + optCut(0)$

Example 1: Rod Cutting

Task: Rod Cutting
Input: rod of length n, prices for different lengths of the rod
Task: find the optimal way to cut the rod to maximize the profit

Input:



length	1	2	3	4	5	6	7	8	9	10
price	0	1	3	5	6	3	7	9	7	10

iterative bottom-up approach: ← better recursive approach:

let $optCut(\ell)$ be the highest possible profit for a rod of length ℓ

$$optCut(\ell) = \max_{1 \le i \le \ell} (profit[i] + optcut(\ell - i))$$

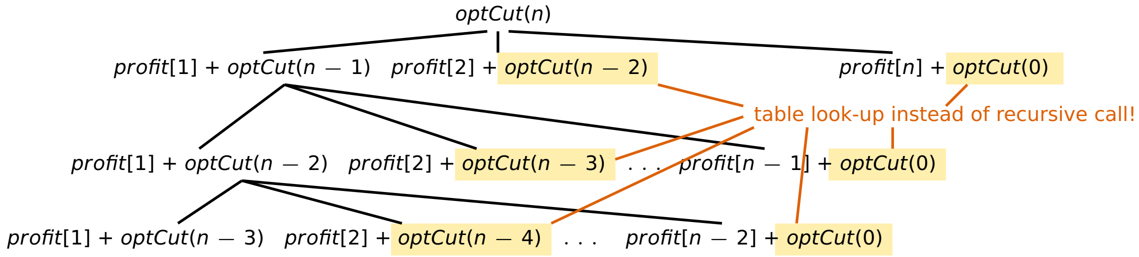
$$optCut(0) = 0$$

Idea: store $optCut(\ell)$ once it is computed

```
optCut = [0]*(n+1)
for l in range(n+1):
    for i in range(1,l+1):
        optCut[l] = max(optCut[l],
            profit[i] + optCut[l-i])
```

table look-up

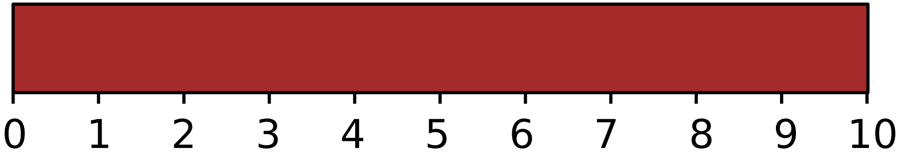
length	0	1	2	3	4	5	6	7	8	9	10
optCut	0	0	1								



Example 1: Rod Cutting

Task: Rod Cutting
Input: rod of length n, prices for different lengths of the rod
Task: find the optimal way to cut the rod to maximize the profit

Input:



length	1	2	3	4	5	6	7	8	9	10
price	0	1	3	5	6	3	7	9	7	10

iterative bottom-up approach: ← better recursive approach:

let $optCut(\ell)$ be the highest possible profit for a rod of length ℓ

```
optCut = [0]*(n+1)
for l in range(n+1):
    for i in range(1,l+1):
        optCut[l] = max(optCut[l],
            profit[i] + optCut[l-i])
```

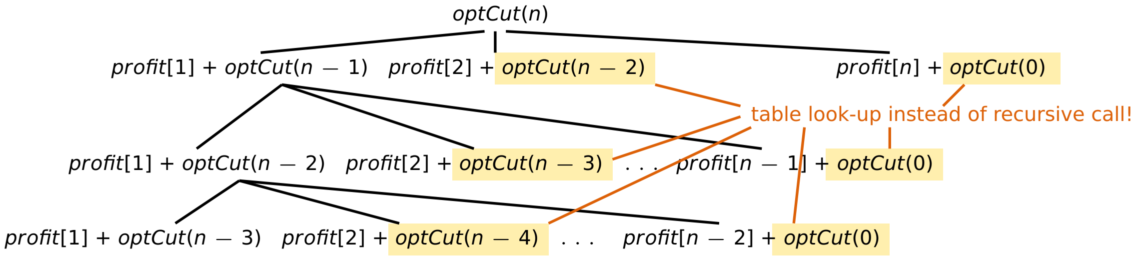
table look-up

$$optCut(\ell) = \max_{1 \le i \le \ell} (profit[i] + optcut(\ell - i))$$

$$optCut(0) = 0$$

Idea: store $optCut(\ell)$ once it is computed

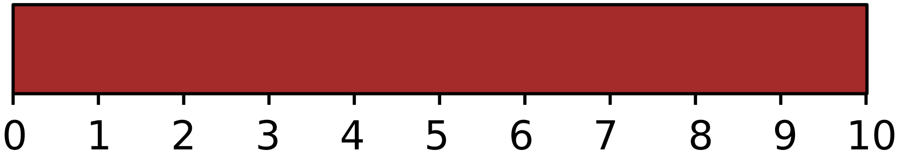
length	0	1	2	3	4	5	6	7	8	9	10
optCut	0	0	1	3							



Example 1: Rod Cutting

Task: Rod Cutting
Input: rod of length *n*, prices for different lengths of the rod
Task: find the optimal way to cut the rod to maximize the profit

Input:



length	1	2	3	4	5	6	7	8	9	10
price	0	1	3	5	6	3	7	9	7	10

iterative bottom-up approach: ⟵ better recursive approach:

let $optCut(\ell)$ be the highest possible profit for a rod of length ℓ

```
optCut = [0]*(n+1)
for l in range(n+1):
    for i in range(1,l+1):
        optCut[l] = max(optCut[l],
            profit[i] + optCut[l-i])
```

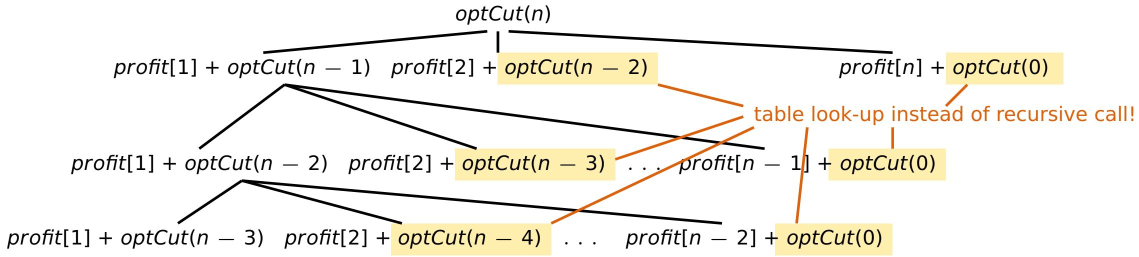
table look-up

length	0	1	2	3	4	5	6	7	8	9	10
optCut	0	0	1	3	5						

$$optCut(\ell) = \max_{1 \le i \le \ell} (profit[i] + optcut(\ell - i))$$

$$optCut(0) = 0$$

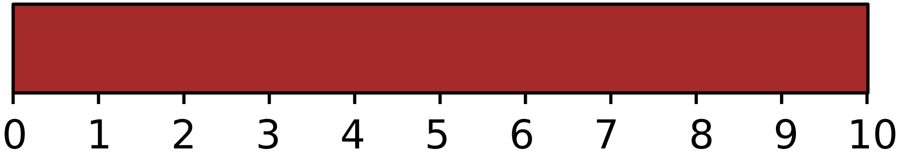
Idea: store $optCut(\ell)$ once it is computed



Example 1: Rod Cutting

Task: Rod Cutting
Input: rod of length n, prices for different lengths of the rod
Task: find the optimal way to cut the rod to maximize the profit

Input:



length	1	2	3	4	5	6	7	8	9	10
price	0	1	3	5	6	3	7	9	7	10

iterative bottom-up approach: ← better recursive approach:

let $optCut(\ell)$ be the highest possible profit for a rod of length ℓ

```
optCut = [0]*(n+1)
for l in range(n+1):
    for i in range(1,l+1):
        optCut[l] = max(optCut[l],
            profit[i] + optCut[l-i])
```

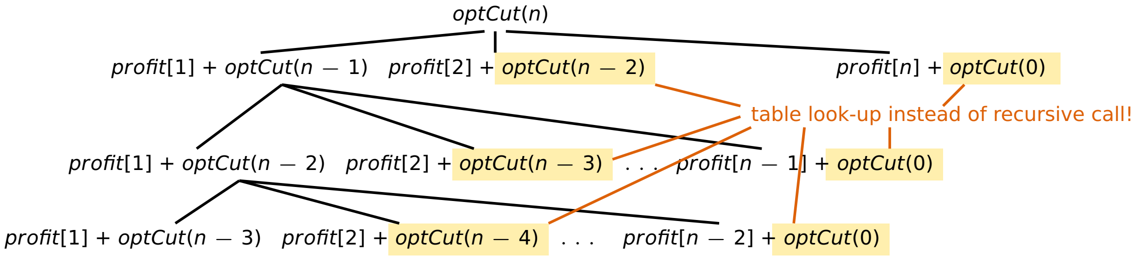
table look-up

$$optCut(\ell) = \max_{1\le i\le \ell} (profit[i] + optcut(\ell - i))$$

$$optCut(0) = 0$$

Idea: store $optCut(\ell)$ once it is computed

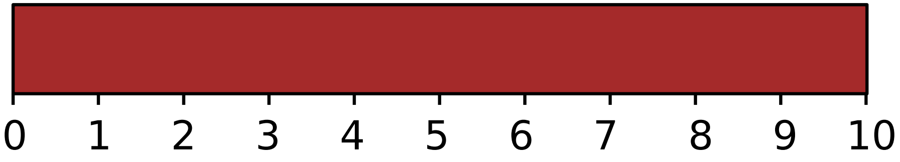
length	0	1	2	3	4	5	6	7	8	9	10
optCut	0	0	1	3	5	6					



Example 1: Rod Cutting

Task: Rod Cutting
Input: rod of length n, prices for different lengths of the rod
Task: find the optimal way to cut the rod to maximize the profit

Input:



length	1	2	3	4	5	6	7	8	9	10
price	0	1	3	5	6	3	7	9	7	10

iterative bottom-up approach: ← better recursive approach:

let $optCut(\ell)$ be the highest possible profit for a rod of length ℓ

$$optCut(\ell) = \max_{1 \le i \le \ell} \left(profit[i] + optcut(\ell - i)\right)$$

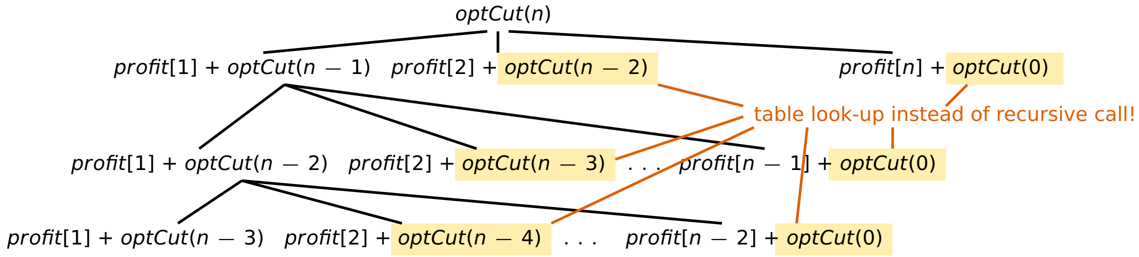
$$optCut(0) = 0$$

Idea: store $optCut(\ell)$ once it is computed

```python
optCut = [0]*(n+1)
for l in range(n+1):
    for i in range(1,l+1):
        optCut[l] = max(optCut[l],
            profit[i] + optCut[l-i])
```

table look-up

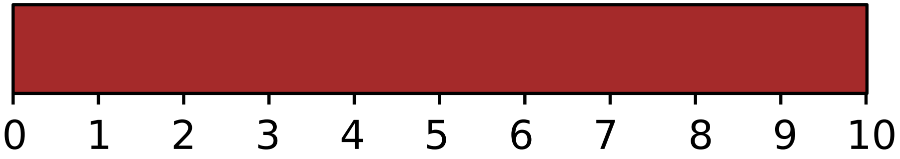
length	0	1	2	3	4	5	6	7	8	9	10
optCut	0	0	1	3	5	6	6				



Example 1: Rod Cutting

Task: Rod Cutting
Input: rod of length n, prices for different lengths of the rod
Task: find the optimal way to cut the rod to maximize the profit

Input:



length	1	2	3	4	5	6	7	8	9	10
price	0	1	3	5	6	3	7	9	7	10

iterative bottom-up approach: ⟵ better recursive approach:

let $optCut(\ell)$ be the highest possible profit for a rod of length ℓ

$$optCut(\ell) = \max_{1 \le i \le \ell} \left(profit[i] + optcut(\ell - i)\right)$$

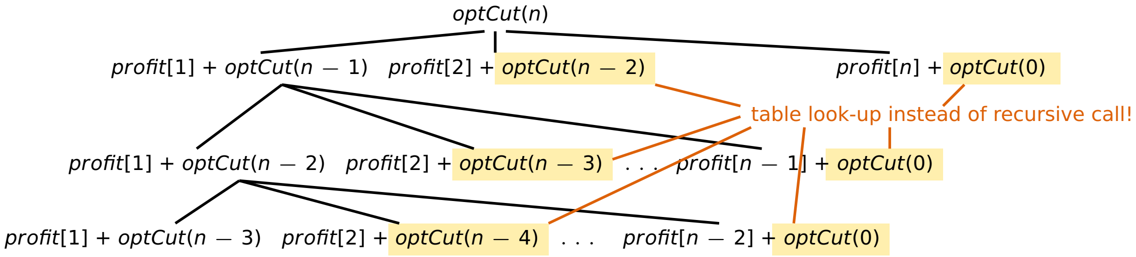
$$optCut(0) = 0$$

Idea: store $optCut(\ell)$ once it is computed

```
optCut = [0]*(n+1)
for l in range(n+1):
    for i in range(1,l+1):
        optCut[l] = max(optCut[l],
            profit[i] + optCut[l-i])
```

table look-up

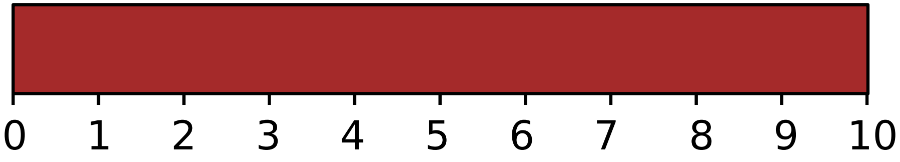
length	0	1	2	3	4	5	6	7	8	9	10
optCut	0	0	1	3	5	6	6	8			



Example 1: Rod Cutting

Task: Rod Cutting
Input: rod of length n, prices for different lengths of the rod
Task: find the optimal way to cut the rod to maximize the profit

Input:



length	1	2	3	4	5	6	7	8	9	10
price	0	1	3	5	6	3	7	9	7	10

iterative bottom-up approach: ← better recursive approach:

let $optCut(\ell)$ be the highest possible profit for a rod of length ℓ

```
optCut = [0]*(n+1)
for l in range(n+1):
    for i in range(1,l+1):
        optCut[l] = max(optCut[l],
            profit[i] + optCut[l-i])
```

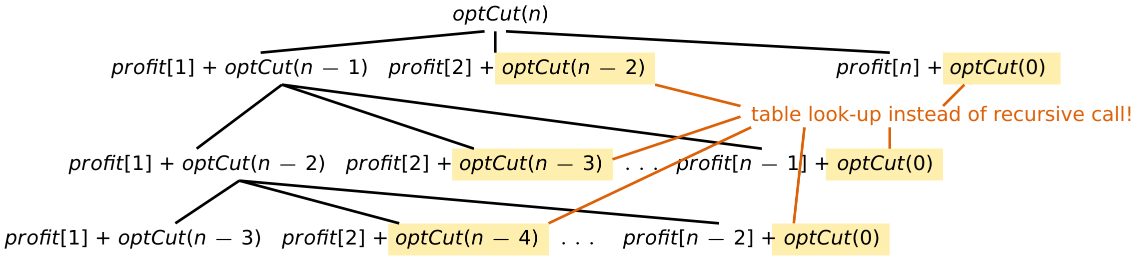
table look-up

$$optCut(\ell) = \max_{1\leq i\leq \ell} (profit[i] + optcut(\ell - i))$$

$$optCut(0) = 0$$

Idea: store $optCut(\ell)$ once it is computed

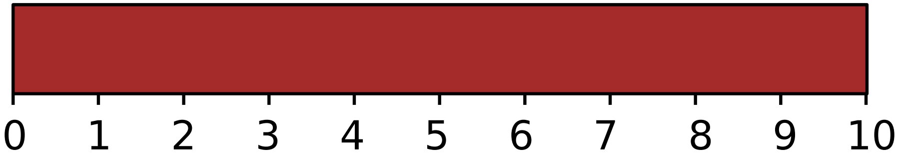
length	0	1	2	3	4	5	6	7	8	9	10
optCut	0	0	1	3	5	6	6	8	10		



Example 1: Rod Cutting

Task: Rod Cutting
Input: rod of length n, prices for different lengths of the rod
Task: find the optimal way to cut the rod to maximize the profit

Input:



length	1	2	3	4	5	6	7	8	9	10
price	0	1	3	5	6	3	7	9	7	10

iterative bottom-up approach: ← better recursive approach:

let $optCut(\ell)$ be the highest possible profit for a rod of length ℓ

$$optCut(\ell) = \max_{1 \leq i \leq \ell} (profit[i] + optcut(\ell - i))$$

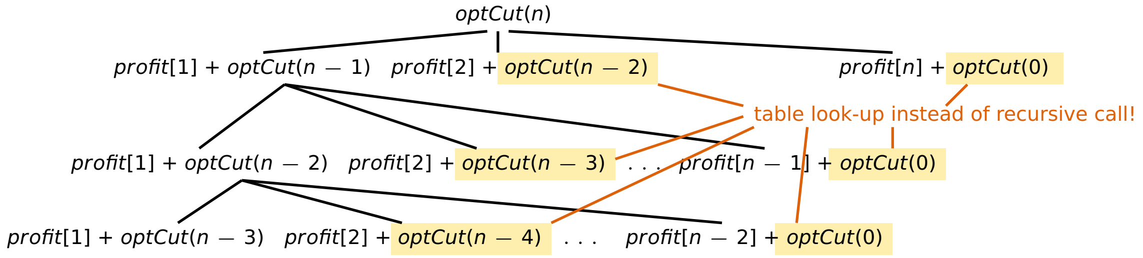
$$optCut(0) = 0$$

Idea: store $optCut(\ell)$ once it is computed

```
optCut = [0]*(n+1)
for l in range(n+1):
    for i in range(1,l+1):
        optCut[l] = max(optCut[l],
            profit[i] + optCut[l-i])
```

table look-up

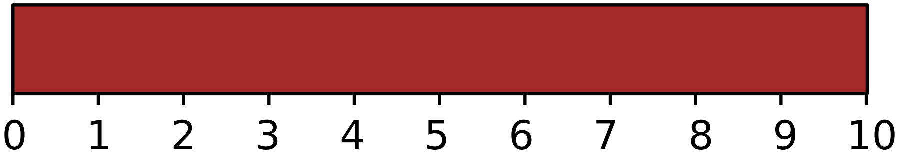
length	0	1	2	3	4	5	6	7	8	9	10
optCut	0	0	1	3	5	6	6	8	10	11	



Example 1: Rod Cutting

Task: Rod Cutting
Input: rod of length n, prices for different lengths of the rod
Task: find the optimal way to cut the rod to maximize the profit

Input:



length	1	2	3	4	5	6	7	8	9	10
price	0	1	3	5	6	3	7	9	7	10

iterative bottom-up approach: ← better recursive approach:

let $optCut(\ell)$ be the highest possible profit for a rod of length ℓ

```
optCut = [0]*(n+1)
for l in range(n+1):
    for i in range(1,l+1):
        optCut[l] = max(optCut[l],
            profit[i] + optCut[l-i])
```

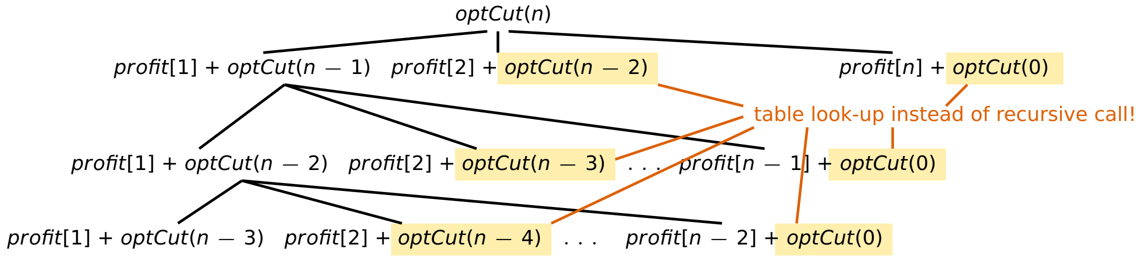
table look-up

$$optCut(\ell) = \max_{1 \le i \le \ell} (profit[i] + optcut(\ell - i))$$

$$optCut(0) = 0$$

Idea: store $optCut(\ell)$ once it is computed

length	0	1	2	3	4	5	6	7	8	9	10
optCut	0	0	1	3	5	6	6	8	10	11	12



Example 2: Subset Sum

Task: Subset Sum
Input: natural numbers $a_1, \dots, a_n, s$
Task: find subset of $\{a_1, \dots, a_n\}$ that sums up to exactly s

Example 2: Subset Sum

Task: Subset Sum
Input: natural numbers $a_1, \ldots, a_n, s$
Task: find subset of $\{a_1, \ldots, a_n\}$ that sums up to exactly s

1.) idea for recursion: sum of k while choosing from the first i numbers is possible, if:

- sum k possible while choosing from the first $i-1$ numbers (= don't choose a_i)
- sum $k - a_i$ possible while choosing from the first $i-1$ numbers (= choose a_i)

Example 2: Subset Sum

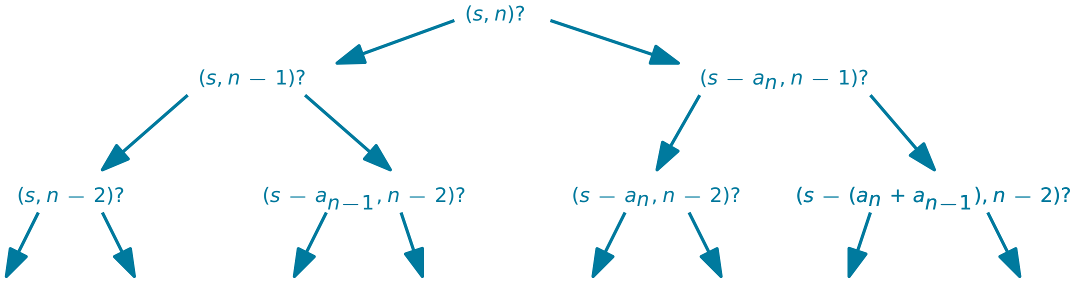
Task: Subset Sum
Input: natural numbers $a_1, \dots, a_n, s$
Task: find subset of $\{a_1, \dots, a_n\}$ that sums up to exactly s

1.) idea for recursion: sum of k while choosing from the first i numbers is possible, if:

sub-problem (k, i)

- sum k possible while choosing from the first $i-1$ numbers (= don't choose a_i)
- sum $k - a_i$ possible while choosing from the first $i-1$ numbers (= choose a_i)

2.) recursion + save solutions for sub-problems (k, i)



Example 2: Subset Sum

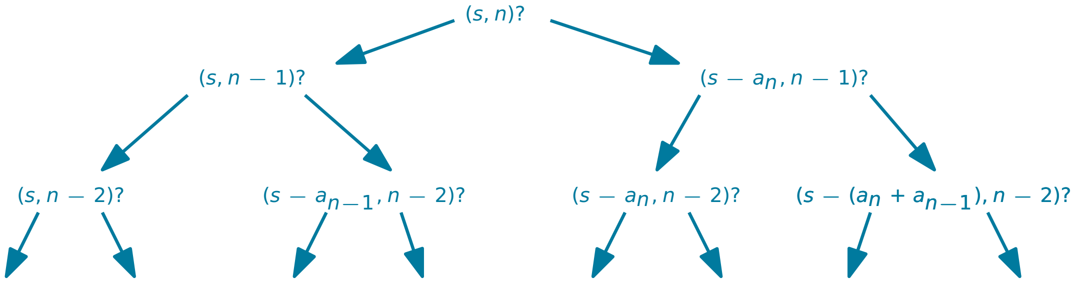
Task: Subset Sum
Input: natural numbers $a_1, \dots, a_n, s$
Task: find subset of $\{a_1, \dots, a_n\}$ that sums up to exactly s

1.) idea for recursion: sum of k while choosing from the first i numbers is possible, if:

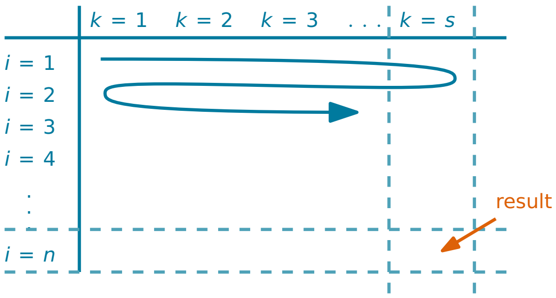
- sum k possible while choosing from the first $i-1$ numbers (= don't choose a_i)
- sum $k - a_i$ possible while choosing from the first $i-1$ numbers (= choose a_i)

sub-problem (k, i)

2.) recursion + save solutions for sub-problems (k, i)



3.) iterative bottom-up:



Example 2: Subset Sum

Task: Subset Sum
Input: natural numbers $a_1, \ldots, a_n, s$
Task: find subset of $\{a_1, \ldots, a_n\}$ that sums up to exactly s

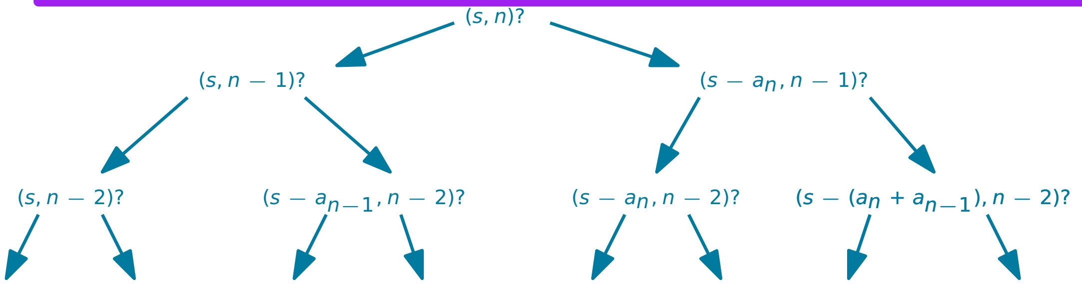
1.) idea for recursion: sum of k while choosing from the first i numbers is possible, if:

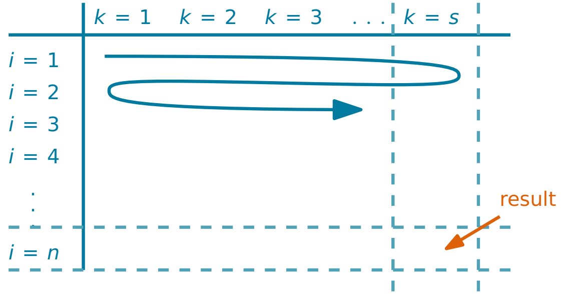
- sum k possible while choosing from the first $i-1$ numbers (= don't choose a_i)

sub problem (k, i)

Why not something like: sumPossible(k) if for some $1 \leq i \leq n$: sumPossible($k - a_i$)
for $1 \leq i \leq n$: sumPossible(a_i) = TRUE

2.)





Example 2: Subset Sum

Task: Subset Sum
Input: natural numbers $a_1, \ldots, a_n, s$
Task: find subset of $\{a_1, \ldots, a_n\}$ that sums up to exactly s

1.) idea for recursion: sum of k while choosing from the first i numbers is possible, if:

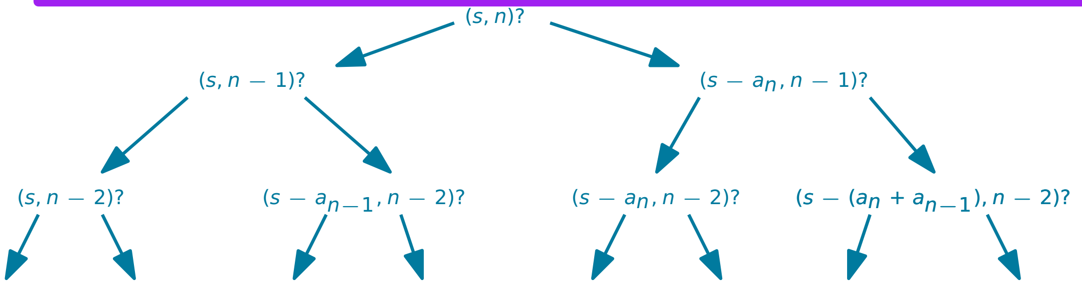
- sum k possible while choosing from the first $i-1$ numbers (= don't choose a_i)

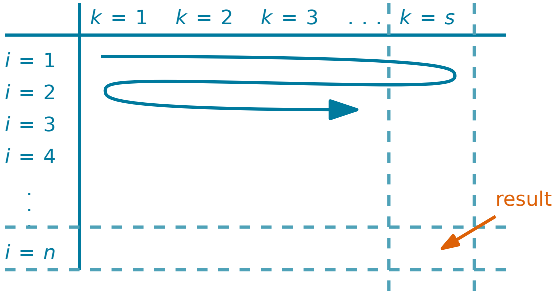
sub problem (k, i)

Why not something like: sumPossible(k) if for some $1 \leq i \leq n$: sumPossible($k - a_i$)
for $1 \leq i \leq n$: sumPossible(a_i) = TRUE

Answer: numbers could be chosen more than once!

2.)





Example 2: Subset Sum

Task: Subset Sum
Input: natural numbers $a_1, \ldots, a_n, s$
Task: find subset of $\{a_1, \ldots, a_n\}$ that sums up to exactly s

1.) idea for recursion: sum of k while choosing from the first i numbers is possible, if:

- sum k possible while choosing from the first $i-1$ numbers (= don't choose a_i)

sub problem (k, i)

Why not something like: sumPossible(k) if for some $1 \leq i \leq n$: sumPossible($k - a_i$)
for $1 \leq i \leq n$: sumPossible(a_i) = TRUE
Answer: numbers could be chosen more than once!

2.)

(s, n)?

Why not something like:

sumPossible(k, S) if for some $1 \leq i \leq n$: sumPossible($k - a_i, S \setminus a_i$)
for all $S' \subseteq S$: sumPossible($0, S'$) = TRUE

Example 2: Subset Sum

Task: Subset Sum
Input: natural numbers $a_1, \ldots, a_n, s$
Task: find subset of $\{a_1, \ldots, a_n\}$ that sums up to exactly s

1.) idea for recursion: sum of k while choosing from the first i numbers is possible, if:

- sum k possible while choosing from the first $i-1$ numbers (= don't choose a_i)

sub problem (k, i)

2.)

Why not something like: sumPossible(k) if for some $1 \leq i \leq n$: sumPossible($k - a_i$)
for $1 \leq i \leq n$: sumPossible(a_i) = TRUE
Answer: numbers could be chosen more than once!

Why not something like:
sumPossible(k, S) if for some $1 \leq i \leq n$: sumPossible($k - a_i, S \setminus a_i$)
for all $S' \subseteq S$: sumPossible($0, S'$) = TRUE
Answer: exponentially many subsets $S' \subseteq S$

Example 2: Subset Sum

Task: Subset Sum
Input: natural numbers $a_1, \ldots, a_n, s$
Task: find $\boxed{\text{subset of } \{a_1, \ldots, a_n\}}$ that sums up to exactly s

1.) idea for recursion: sum of k while choosing from the first i numbers is possible, if:

- sum k possible while choosing from the first $i-1$ numbers (= don't choose a_i)

sub-problem (k, i)

2.)

Why not something like: sumPossible(k) if for some $1 \le i \le n$: sumPossible($k - a_i$)
for $1 \le i \le n$: sum
Answer: numbers could be

Clever idea of (k, i) approach:
Fixed order of adding elements to solution restricts possibilities!

Why not something like:

sumPossible(k, S) if for some $1 \le i \le n$: sumPossible($k - a_i, S \setminus a_i$)
for all $S' \subseteq S$: sumPossible($0, S'$) = TRUE

Answer: exponentially many subsets $S' \subseteq S$

Example 3: Coin Change

Task: Coin Change
Input: value v and k coin types $c_1, \ldots, c_k$ with values $v_1 < v_2 < \cdots < v_k$
(infinitely many coins per type)
Task: find the smallest amount of coins to change value v (= coin values sum to v)

Example 3: Coin Change

Task: Coin Change
Input: value v and k coin types $c_1, \dots, c_k$ with values $v_1 < v_2 < \dots < v_k$
(infinitely many coins per type)
Task: find the smallest amount of coins to change value v (= coin values sum to v)

recursive approach:

$$mincoins(0) = 0$$

$$mincoins(v) = \min_{1 \leq i \leq k \text{ and } v_i \leq v} \{1 + mincoins(v - v_i)\}$$

Example 3: Coin Change

Task: Coin Change
Input: value v and k coin types $c_1, \ldots, c_k$ with values $v_1 < v_2 < \cdots < v_k$
(infinitely many coins per type)
Task: find the smallest amount of coins to change value v (= coin values sum to v)

recursive approach:

$$mincoins(0) = 0$$

$$mincoins(v) = \min_{1 \leq i \leq k \text{ and } v_i \leq v} \{1 + mincoins(v - v_i)\}$$

Task: Coin Change Combinations
Input: value v and k coin types $c_1, \ldots, c_k$ with values $v_1 < v_2 < \cdots < v_k$
(infinitely many coins per type)
Task: compute the number of possibilities to change value v (= sum to v, ignore order)

Example 3: Coin Change

Task: Coin Change
Input: value v and k coin types $c_1, \ldots, c_k$ with values $v_1 < v_2 < \cdots < v_k$
(infinitely many coins per type)
Task: find the smallest amount of coins to change value v (= coin values sum to v)

recursive approach:

$$mincoins(0) = 0$$

$$mincoins(v) = \min_{1 \leq i \leq k \text{ and } v_i \leq v} \{1 + mincoins(v - v_i)\}$$

Task: Coin Change Combinations
Input: value v and k coin types $c_1, \ldots, c_k$ with values $v_1 < v_2 < \cdots < v_k$
(infinitely many coins per type)
Task: compute the number of possibilities to change value v (= sum to v, ignore order)

recursive approach:

$count(\ell, j)$ = number of possibilities to change value ℓ using the first j coin types

$$count(0, i) = 1 \text{ for } 1 \leq i \leq k$$

$$count(v, k) = count(v, k-1) + count(v - v_k, k)$$

Example 3: Coin Change

Task: Coin Change
Input: value v and k coin types $c_1, \ldots, c_k$ with values $v_1 < v_2 < \cdots < v_k$
(infinitely many coins per type)
Task: find the smallest amount of coins to change value v (= coin values sum to v)

recursive approach:

$$mincoins(0) = 0$$

$$mincoins(v) = \min_{1 \leq i \leq k \text{ and } v_i \leq v} \{1 + mincoins(v - v_i)\}$$

Task: Coin Change Combinations
Input: value v and k coin types $c_1, \ldots, c_k$ with values $v_1 < v_2 < \cdots < v_k$
(infinitely many coins per type)
Task: compute the number of possibiliti

Why do we need the second dimension?

recursive approach:

$count(\ell, j)$ = number of possibilities to change value ℓ using the first j coin types

$$count(0, i) = 1 \text{ for } 1 \leq i \leq k$$

$$count(v, k) = count(v, k-1) + count(v - v_k, k)$$

Example 3: Coin Change

Task: Coin Change
Input: value v and k coin types $c_1, \ldots, c_k$ with values $v_1 < v_2 < \cdots < v_k$
(infinitely many coins per type)
Task: find the smallest amount of coins to change value v (= coin values sum to v)

recursive approach:

$$mincoins(0) = 0$$

$$mincoins(v) = \min_{1 \leq i \leq k \text{ and } v_i \leq v} \{1 + mincoins(v - v_i)\}$$

Task: Coin Change Combinations
Input: value v and k coin types $c_1, \ldots, c_k$ with values $v_1 < v_2 < \cdots < v_k$
(infinitely many coins per type)
Task: compute the number of possibiliti

Why do we need the second dimension?

Answer: to avoid double-counting solutions

recursive approach:

$count(\ell, j)$ = number of possibilities to change value ℓ using the first j coin types

$$count(0, i) = 1 \text{ for } 1 \leq i \leq k$$

$$count(v, k) = count(v, k-1) + count(v - v_k, k)$$

DP Checklist

3 steps for doing it:

DP Checklist

3 steps for doing it:

1.) come up with a recursive approach to solve your problem

- describe the exact problem you want to solve recursively
- give a recursive formula for your problem that involves calls to smaller sub-problems of exatly the same problem

DP Checklist

3 steps for doing it:

1.) come up with a recursive approach to solve your problem

- describe the exact problem you want to solve recursively
- give a recursive formula for your problem that involves calls to smaller sub-problems of exatly the same problem

2.) store the solutions of sub-problems to avoid future recursive calls

- identify which sub-problems will be involved
- choose a data structure for storing the results (usually: multi-dim. array)

DP Checklist

3 steps for doing it:

1.) come up with a recursive approach to solve your problem

- describe the exact problem you want to solve recursively
- give a recursive formula for your problem that involves calls to smaller sub-problems of exatly the same problem

2.) store the solutions of sub-problems to avoid future recursive calls

- identify which sub-problems will be involved
- choose a data structure for storing the results (usually: multi-dim. array)

3.) optional: generate solutions to sub-problems in a bottom-up fashion to get an iterative algorithm

- identify dependencies among the sub-problems
- find a good evaluation order for computing the sub-problems (all depending sub-problems must be computed already)

DP Checklist

3 steps for doing it:

1.) come up with a recursive approach to solve your problem

- describe the exact problem you want to solve recursively
- give a recursive formula for your problem that involves calls to smaller sub-problems of exatly the same problem

2.) store the solutions of sub-problems to avoid future recursive calls

- identify which sub-problems will be involved
- choose a data structure for storing the results (usually: multi-dim. array)

3.) optional: generate solutions to sub-problems in a bottom-up fashion to get an iterative algorithm

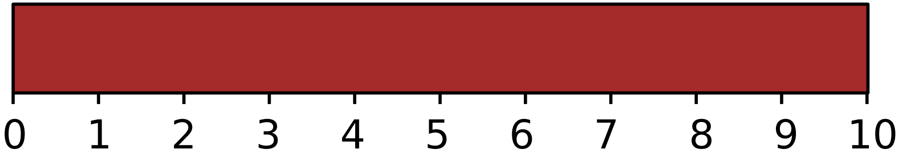
- identify dependencies among the sub-problems
- find a good evaluation order for computing the sub-problems (all depending sub-problems must be computed already)

Why optional?

Rod Cutting Again

Task: Rod Cutting
Input: rod of length n, prices for different lengths of the rod
Task: find the optimal way to cut the rod to maximize the profit

Input:

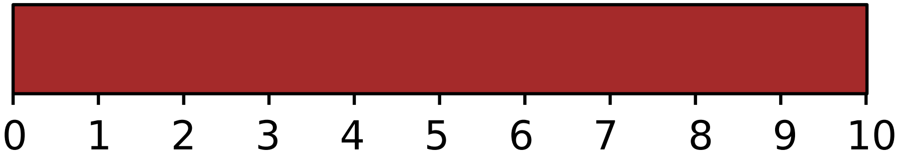


length	1	2	3	4	5	6	7	8	9	10
price	0	1	3	5	6	3	7	9	7	10

Rod Cutting Again

Task: Rod Cutting
Input: rod of length n, prices for different lengths of the rod
Task: find the optimal way to cut the rod to maximize the profit

Input:



length	1	2	3	4	5	6	7	8	9	10
price	0	1	3	5	6	3	7	9	7	10

recursive approach:

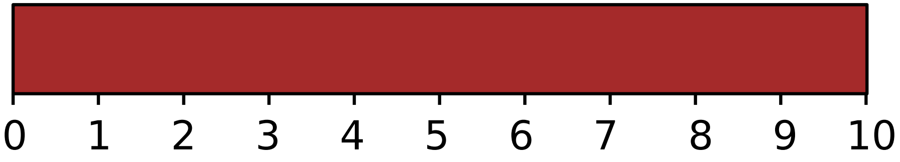
$$optCut(0) = 0$$

$$optCut(\ell) = \max_{1 \le i \le \ell} \big(profit[i] + optcut(\ell - i)\big)$$

Rod Cutting Again

Task: Rod Cutting
Input: rod of length n, prices for different lengths of the rod
Task: find the optimal way to cut the rod to maximize the profit

Input:



length	1	2	3	4	5	6	7	8	9	10
price	0	1	3	5	6	3	7	9	7	10

recursive approach:

$$optCut(0) = 0$$

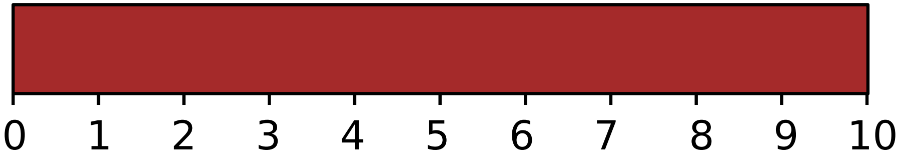
$$optCut(\ell) = \max_{1 \le i \le \ell} \left(profit[i] + optcut(\ell - i) \right)$$

python approach:

```
def optCut(l):
    if l==0:
        return 0
    return max(price[i] + optCut(l-i)
        for i in range(1,l+1))
```

Rod Cutting Again

Task: Rod Cutting
Input: rod of length n, prices for different lengths of the rod
Task: find the optimal way to cut the rod to maximize the profit

Input: 

length	1	2	3	4	5	6	7	8	9	10
price	0	1	3	5	6	3	7	9	7	10

recursive approach:

$$optCut(0) = 0$$

$$optCut(\ell) = \max_{1 \le i \le \ell} \left(profit[i] + optcut(\ell - i) \right)$$

python approach:

```
from functools import *
# just @cache in python 3.9+
@lru_cache(maxsize=None)
def optCut(l):
    if l==0:
        return 0
    return max(price[i] + optCut(l-i)
        for i in range(1,l+1))
```

Dynamic Programming on DAGs

- many problems that are hard in general can be solved on DAGs with DP

Dynamic Programming on DAGs

- many problems that are hard in general can be solved on DAGs with DP

  Eg: Longest Path

Input: DAG $G = (V, E)$.
Task: Find length of longest path.

Dynamic Programming on DAGs

- many problems that are hard in general can be solved on DAGs with DP

  Eg: Longest Path

Input: DAG $G = (V, E)$.
Task: Find length of longest path.

recursive idea:

Dynamic Programming on DAGs

- many problems that are hard in general can be solved on DAGs with DP

  Eg: Longest Path

Input: DAG $G = (V, E)$.
Task: Find length of longest path.

recursive idea:

$$opt(v) = \max_{vu \in E}(opt(u) + 1)$$

Dynamic Programming on DAGs

- many problems that are hard in general can be solved on DAGs with DP

  Eg: Longest Path

Input: DAG $G = (V, E)$.
Task: Find length of longest path.

recursive idea:

$$opt(v) = \max_{vu \in E}(opt(u) + 1)$$

- compute this for all vertices

Dynamic Programming on DAGs

- many problems that are hard in general can be solved on DAGs with DP

  Eg: Longest Path

Input: DAG $G = (V, E)$.
Task: Find length of longest path.

recursive idea: $$opt(v) = \max_{vu \in E}(opt(u) + 1)$$

- compute this for all vertices
- remember result for later recurisve calls (memorization)

Dynamic Programming on DAGs

- many problems that are hard in general can be solved on DAGs with DP

  Eg: Longest Path

Input: DAG $G = (V, E)$.
Task: Find length of longest path.

Can this DP be done bottom-up / iteratively?

recursive idea: $$opt(v) = \max_{vu \in E}(opt(u) + 1)$$

- compute this for all vertices
- remember result for later recurisve calls (memorization)

Dynamic Programming on DAGs

- many problems that are hard in general can be solved on DAGs with DP

  Eg: Longest Path

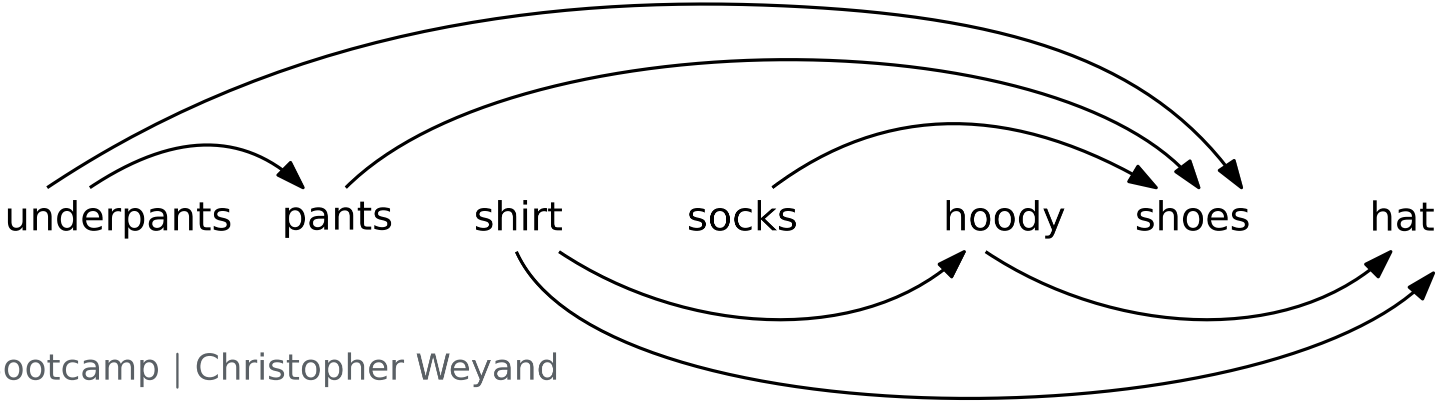
Input: DAG $G = (V, E)$.
Task: Find length of longest path.

recursive idea: $$opt(v) = \max_{vu \in E}(opt(u) + 1)$$

- compute this for all vertices
- remember result for later recurisve calls (memorization)

Can this DP be done bottom-up / iteratively?

Reverse Topological Order!



Dynamic Programming on Trees

- many problems that are hard in general can be solved on Trees with DP

Dynamic Programming on Trees

- many problems that are hard in general can be solved on Trees with DP

  Eg: Maximum Independent Set (MIS)

Input: Tree $T = (V, E)$.
Task: Find largest set $I \subseteq V$ with no edge between them.

Dynamic Programming on Trees

- many problems that are hard in general can be solved on Trees with DP

  Eg: Maximum Independent Set (MIS)

Input: Tree $T = (V, E)$.
Task: Find largest set $I \subseteq V$ with no edge between them.

recursive idea: If $u \in I$, then no neighbor of u is in I.

Dynamic Programming on Trees

- many problems that are hard in general can be solved on Trees with DP

Eg: Maximum Independent Set (MIS)

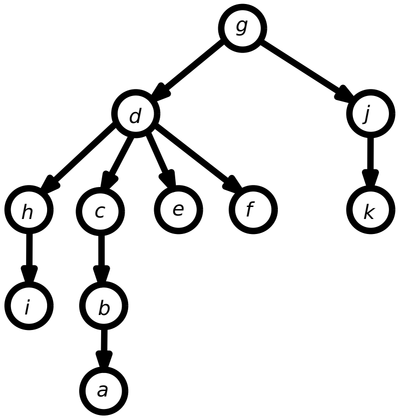
Input: Tree $T = (V, E)$.
Task: Find largest set $I \subseteq V$ with no edge between them.

recursive idea: If $u \in I$, then no neighbor of u is in I.

trick: root T at arbitrary vertex!

Per Subtree T_x with root x:

- recursively compute MIS of T_x with $x \in I$
- recursively compute MIS of T_x with $x \notin I$
- remember result for later recurisve calls (memorization)



Dynamic Programming on Trees

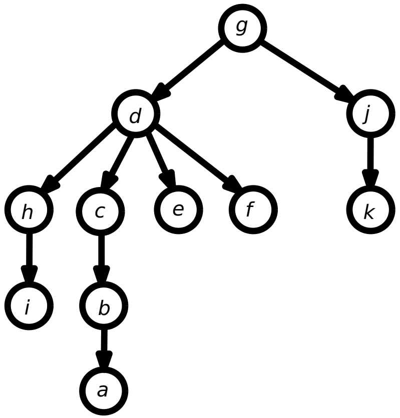
- many problems that are hard in general can be solved on Trees with DP

Eg: Maximum Independent Set (MIS)

Input: Tree $T = (V, E)$.
Task: Find largest set $I \subseteq V$ with no edge between them.

recursive idea: If $u \in I$, then no neighbor of u is in I.

trick: root T at arbitrary vertex!



Per Subtree T_x with root x:

- recursively compute MIS of T_x with $x \in I$
- recursively compute MIS of T_x with $x \notin I$
- remember result for later recurisve calls (memorization)

Bottom-Up?

Dynamic Programming on Trees

- many problems that are hard in general can be solved on Trees with DP

Eg: Maximum Independent Set (MIS)

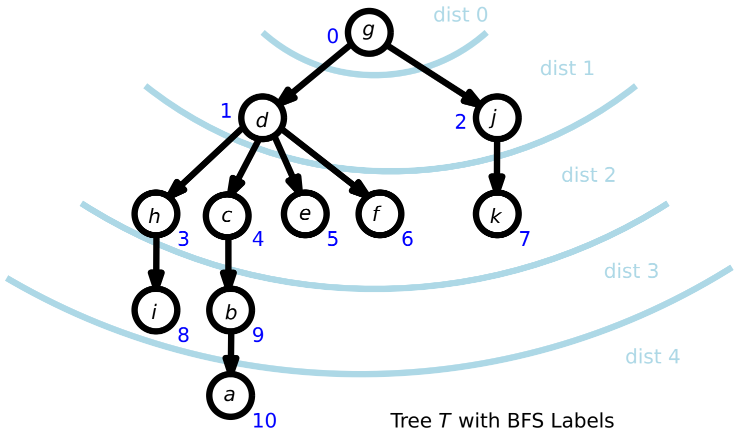
Input: Tree $T = (V, E)$.
Task: Find largest set $I \subseteq V$ with no edge between them.

recursive idea: If $u \in I$, then no neighbor of u is in I.

trick: root T at arbitrary vertex!

Per Subtree T_x with root x:

- recursively compute MIS of T_x with $x \in I$
- recursively compute MIS of T_x with $x \notin I$
- remember result for later recurisve calls (memorization)



Tree T with BFS Labels

Bottom-Up?

Reverse BFS Order!

Structure of a Tree DP

Structure of a Tree DP

TODO:

- input
- write custom treeDFS
- create dp-array of length n
- call **treeDFS**(graph, 0, -1, dp-array)
- output dp-array[0]

Structure of a Tree DP

TODO:

- input
- write custom treeDFS
- create dp-array of length n
- call **treeDFS**(graph, 0, -1, dp-array)
- output dp-array[0]

treeDFS(graph, node, parent, dp-array)

- **return**

Structure of a Tree DP

TODO:

- input
- write custom treeDFS
- create dp-array of length n
- call **treeDFS**(graph, 0, -1, dp-array)
- output dp-array[0]

treeDFS(graph, node, parent, dp-array)

- // a leaf should be handled here
- **return**

Structure of a Tree DP

TODO:

- input
- write custom treeDFS
- create dp-array of length n
- call **treeDFS**(graph, 0, -1, dp-array)
- output dp-array[0]

treeDFS(graph, node, parent, dp-array)

- // a leaf should be handled here
- **for** each neighbor x of node
  - **if** x ≠ parent **then** recursive call
- **return**

Structure of a Tree DP

TODO:
- input
- write custom treeDFS
- create dp-array of length n
- call **treeDFS**(graph, 0, -1, dp-array)
- output dp-array[0]

treeDFS(graph, node, parent, dp-array)
- // a leaf should be handled here
- **for** each neighbor x of node
    - **if** x $\neq$ parent **then** recursive call
- // compute dp-array[node]
- **return**