CS11313 - Spring 2023

Design & Analysis of Algorithms

0-1 Knapsack

Ibrahim Albluwi

0 – 1 Knapsack

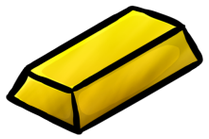
Input. N items, each of weight $w_i > 0$ and value $v_i > 0$. and a knapsack of weight capacity $W > 0$.

Goal. Find the *maximum* value the knapsack can hold.

Assumptions.

- Taking a fraction of an item is not possible.
- W, w_i and v_i are integers.

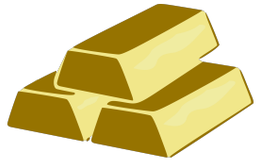
capacity = 13 kg



weight = 4 kg
value = $3000

weight = 12 kg
value = $8000

weight = 5 kg
value = $4000



weight = 6 kg
value = $6000

weight = 1 kg
value = $1000

0 – 1 Knapsack

Input. N items, each of weight $w_i > 0$ and value $v_i > 0$. and a knapsack of weight capacity $W > 0$.

Goal. Find the *maximum* value the knapsack can hold.

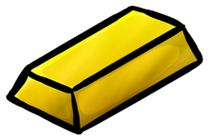
Assumptions.

- Taking a fraction of an item is not possible.
- W, w_i and v_i are integers.

Examples.

- `$8000` (12 kg) + `$1000` (1 kg) = `$9000` (13 kg)

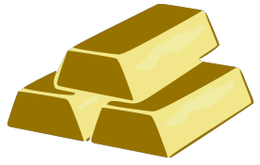
capacity = 13 kg



weight = 4 kg
value = $3000

weight = 12 kg
value = $8000

weight = 5 kg
value = $4000



weight = 6 kg
value = $6000

weight = 1 kg
value = $1000

0–1 Knapsack

Input. N items, each of weight $w_i > 0$ and value $v_i > 0$. and a knapsack of weight capacity $W > 0$.

Goal. Find the *maximum* value the knapsack can hold.

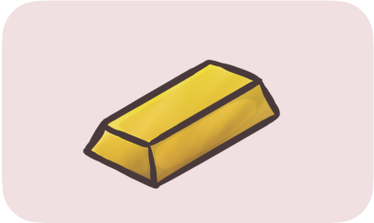
Assumptions.

- Taking a fraction of an item is not possible.
- W, w_i and v_i are integers.

Examples.

- `$8000` (12 kg) + `$1000` (1 kg) = `$9000` (13 kg)
- `$6000` (6 kg) + `$3000` (4 kg) + `$1000` (1 kg) = `$10000` (11 kg)

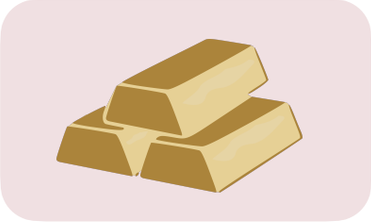
capacity = 13 kg



weight = 4 kg
value = $3000

weight = 12 kg
value = $8000

weight = 5 kg
value = $4000



weight = 6 kg
value = $6000

weight = 1 kg
value = $1000

0–1 Knapsack

Input. N items, each of weight $w_i > 0$ and value $v_i > 0$. and a knapsack of weight capacity $W > 0$.

Goal. Find the *maximum* value the knapsack can hold.

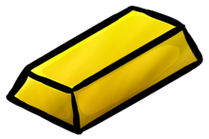
Assumptions.

- Taking a fraction of an item is not possible.
- W, w_i and v_i are integers.

Examples.

- `$8000` (12 kg) + `$1000` (1 kg) = `$9000` (13 kg)
- `$6000` (6 kg) + `$3000` (4 kg) + `$1000` (1 kg) = `$10000` (11 kg)
- `$6000` (6 kg) + `$4000` (5 kg) + `$1000` (1 kg) = `$11000` (12 kg)

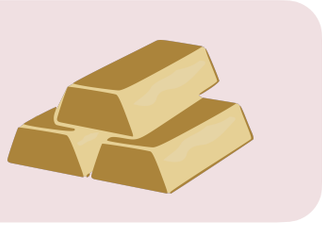
capacity = 13 kg



weight = 4 kg
value = $3000

weight = 12 kg
value = $8000

weight = 5 kg
value = $4000



weight = 6 kg
value = $6000

weight = 1 kg
value = $1000

0-1 Knapsack

Applications. Many!

0-1 Knapsack

Applications. Many!

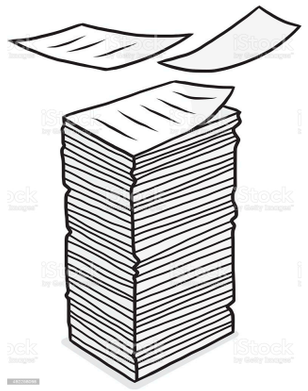
Example 1. Given a set of daily *jobs* each having an *income* and a required number of *hours*.

Which subset of jobs can be done in a day (< 24 *hours*) and provide the *maximum* total income?

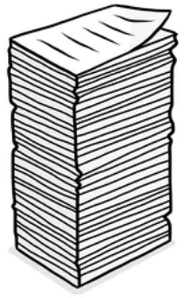
capacity = 24 hours

weight = 6 hours
value = $300

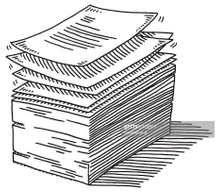
weight = 12 hours
value = $800



weight = 10 hours
value = $400



weight = 8 hours
value = $600



weight = 5 hours
value = $100

0 – 1 Knapsack

Applications. Many!

Example 1. Given a set of daily *jobs* each having an *income* and a required number of *hours*.

Which subset of jobs can be done in a day (< *24 hours*) and provide the *maximum* total income?

Example 2. Given a set of *available beds* in a hotel and a set of *teams* each with a certain *number of players* and a *cost per player* they are willing to pay.

Which teams should the hotel admit such that the total number of players admitted < the number of available beds and the return is maximized?

capacity = 15 beds

players = 3
value = $3000

players = 12
value = $8000

players = 7
value = $4000

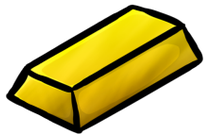
players = 5
value = $6000

players = 4
value = $1000

0-1 **Knapsack** (Brute Force)

Check all possibilities. For each item either take it or don't take it.

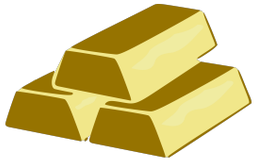
capacity = 13 kg



weight = 4 kg
value = $3000

weight = 12 kg
value = $8000

weight = 5 kg
value = $4000



weight = 6 kg
value = $6000

weight = 1 kg
value = $1000

How long will this take?

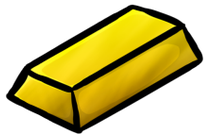
0-1 **Knapsack** (Brute Force)

Check all possibilities. For each item either take it or don't take it.

Item 1	Item 2	Item 3	Item 4	Item 5
0	0	0	0	0

don't take any item

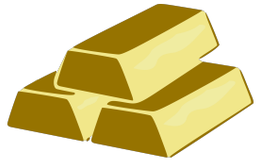
capacity = 13 kg



weight = 4 kg
value = $3000

weight = 12 kg
value = $8000

weight = 5 kg
value = $4000



weight = 6 kg
value = $6000

weight = 1 kg
value = $1000

How long will this take?

0-1 **Knapsack** (Brute Force)

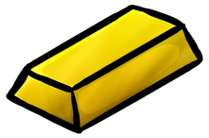
Check all possibilities. For each item either take it or don't take it.

Item 1	Item 2	Item 3	Item 4	Item 5
0	0	0	0	0
0	0	0	0	**1**

take only item 5

How long will this take?

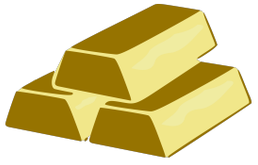
capacity = 13 kg



weight = 4 kg
value = $3000

weight = 12 kg
value = $8000

weight = 5 kg
value = $4000



weight = 6 kg
value = $6000

weight = 1 kg
value = $1000

0-1 **Knapsack** (Brute Force)

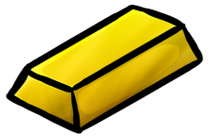
Check all possibilities. For each item either take it or don't take it.

Item 1	Item 2	Item 3	Item 4	Item 5
0	0	0	0	0
0	0	0	0	**1**
0	0	0	**1**	0

take only item 4

How long will this take?

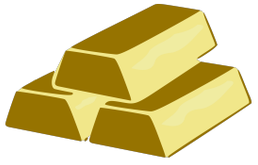
capacity = 13 kg



weight = 4 kg
value = $3000

weight = 12 kg
value = $8000

weight = 5 kg
value = $4000



weight = 6 kg
value = $6000

weight = 1 kg
value = $1000

0-1 **Knapsack** (Brute Force)

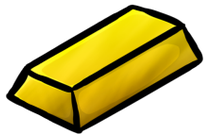
Check all possibilities. For each item either take it or don't take it.

Item 1	Item 2	Item 3	Item 4	Item 5
0	0	0	0	0
0	0	0	0	**1**
0	0	0	**1**	0
0	0	0	**1**	**1**

take item 4 and 5

How long will this take?

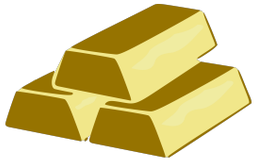
capacity = 13 kg



weight = 4 kg
value = $3000

weight = 12 kg
value = $8000

weight = 5 kg
value = $4000



weight = 6 kg
value = $6000

weight = 1 kg
value = $1000

0-1 **Knapsack** (Brute Force)

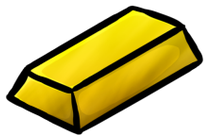
Check all possibilities. For each item either take it or don't take it.

Item 1	Item 2	Item 3	Item 4	Item 5
0	0	0	0	0
0	0	0	0	**1**
0	0	0	**1**	0
0	0	0	**1**	**1**
0	0	**1**	0	0
...	...	...	...	...
...	...	...	...	...
1	**1**	**1**	**1**	**1**

take all items

How long will this take?

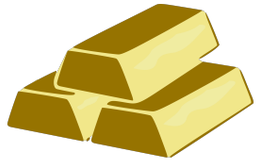
capacity = 13 kg



weight = 4 kg
value = $3000

weight = 12 kg
value = $8000

weight = 5 kg
value = $4000



weight = 6 kg
value = $6000

weight = 1 kg
value = $1000

0–1 **Knapsack** (Brute Force)

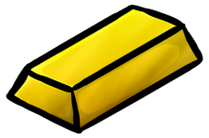
Check all possibilities. For each item either take it or don't take it.

Item 1	Item 2	Item 3	Item 4	Item 5
0	0	0	0	0
0	0	0	0	**1**
0	0	0	**1**	0
0	0	0	**1**	**1**
0	0	**1**	0	0
...	...	...	...	...
...	...	...	...	...
1	**1**	**1**	**1**	**1**

There are 2^N different possibilities!

Exponential!

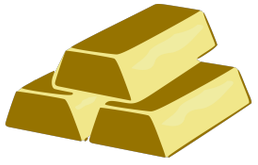
capacity = 13 kg



weight = 4 kg
value = $3000

weight = 12 kg
value = $8000

weight = 5 kg
value = $4000



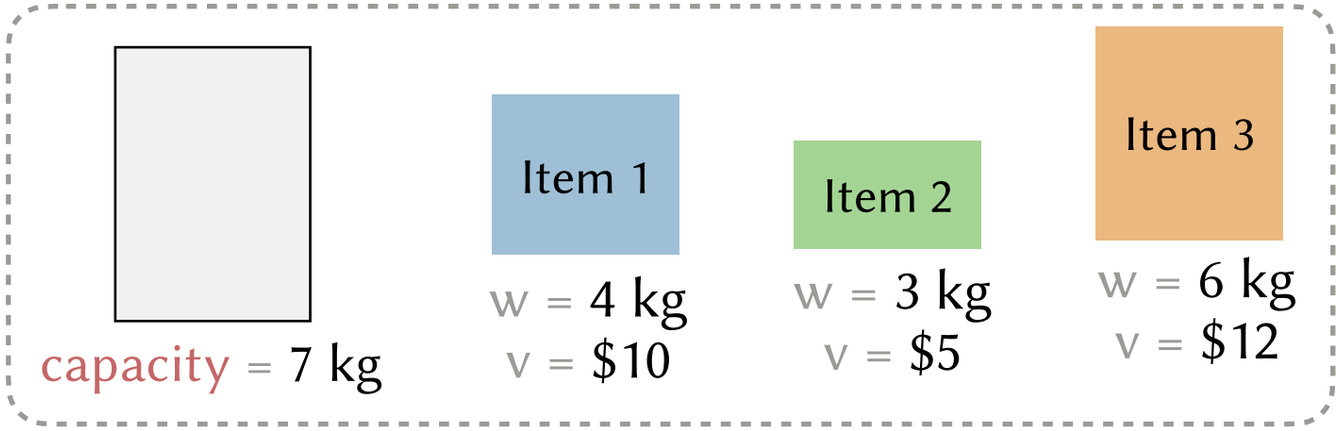
weight = 6 kg
value = $6000

weight = 1 kg
value = $1000

0-1 **Knapsack** (Optimal Substructure)

Assume:

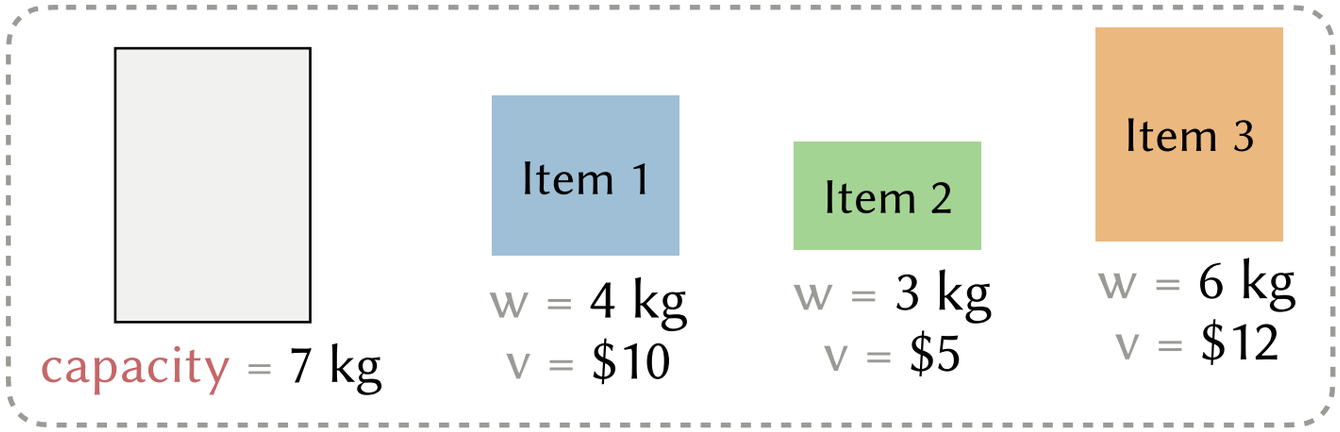
- Items are numbered `1` to `N`.
- **`w`** is an array of weights, where **`w`**`[i]` = *weight* of item `i`.
- **`v`** is an array of values, where **`v`**`[i]` = *value* of item `i`.



0-1 **Knapsack** (Optimal Substructure)

Assume:

- Items are numbered `1` to `N`.
- **`w`** is an array of weights, where **`w`**`[i]` = *weight* of item `i`.
- **`v`** is an array of values, where **`v`**`[i]` = *value* of item `i`.
- **`Knapsack`**`(i, W)`: Solve the knapsack problem for *items* `1` to `i`, assuming the capacity is `W`.

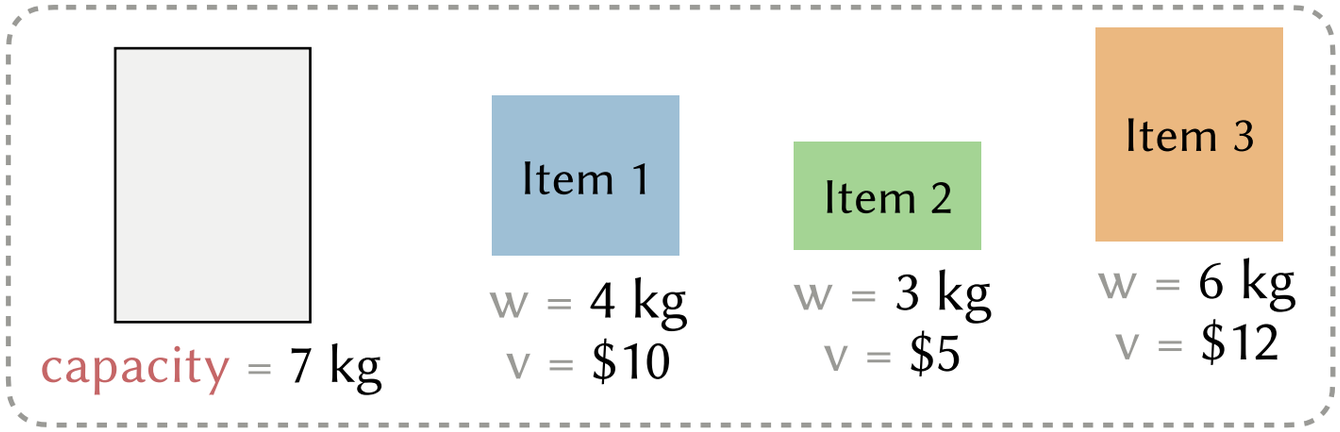


`Knapsack``(3, 7)`

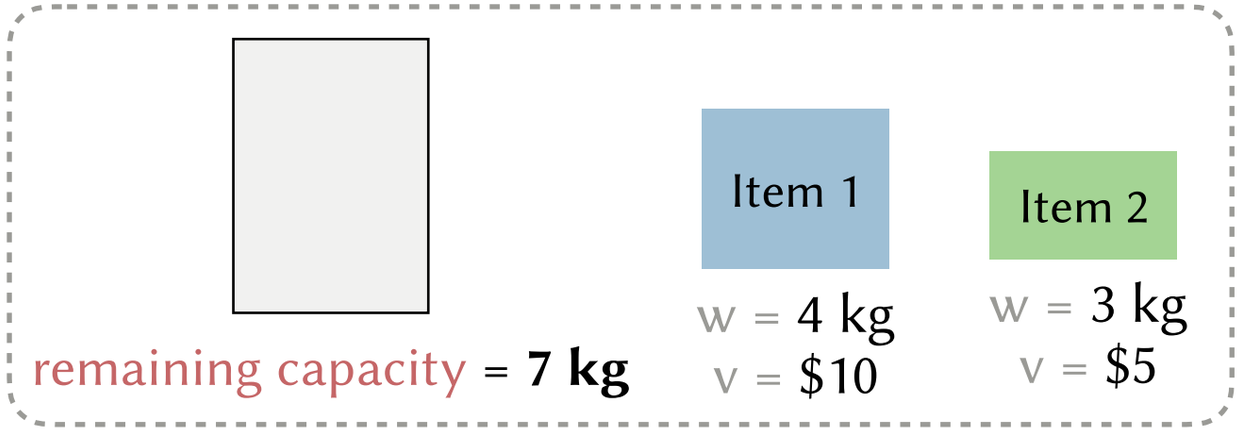
0-1 **Knapsack** (Optimal Substructure)

Assume:

- Items are numbered `1` to `N`.
- **`w`** is an array of weights, where **`w`**`[i]` = *weight* of item `i`.
- **`v`** is an array of values, where **`v`**`[i]` = *value* of item `i`.
- **`Knapsack`**`(i, W)`: Solve the knapsack problem for *items* `1` to `i`, assuming the capacity is `W`.



`Knapsack``(3, 7)`

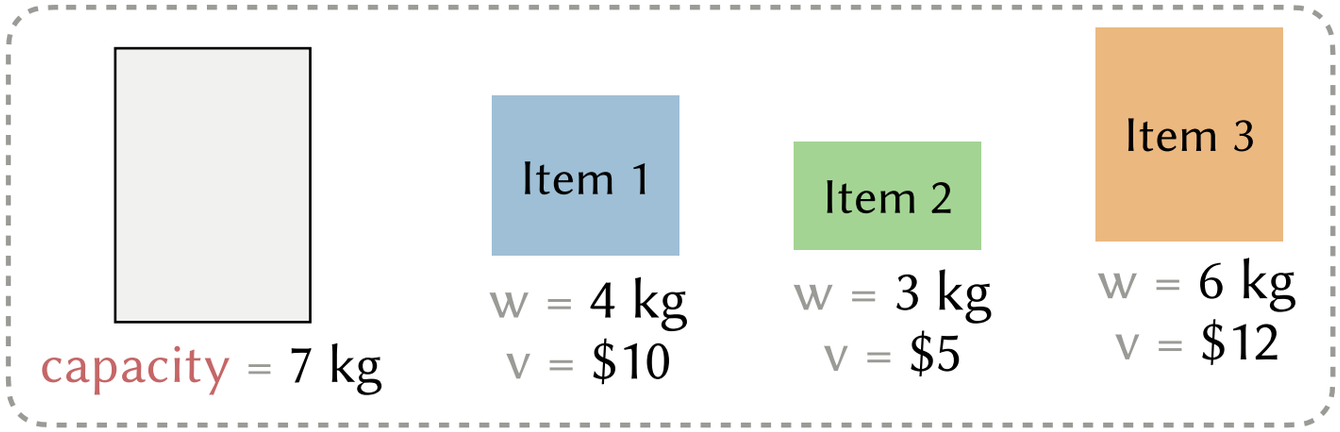


`Knapsack``(2, 7)`

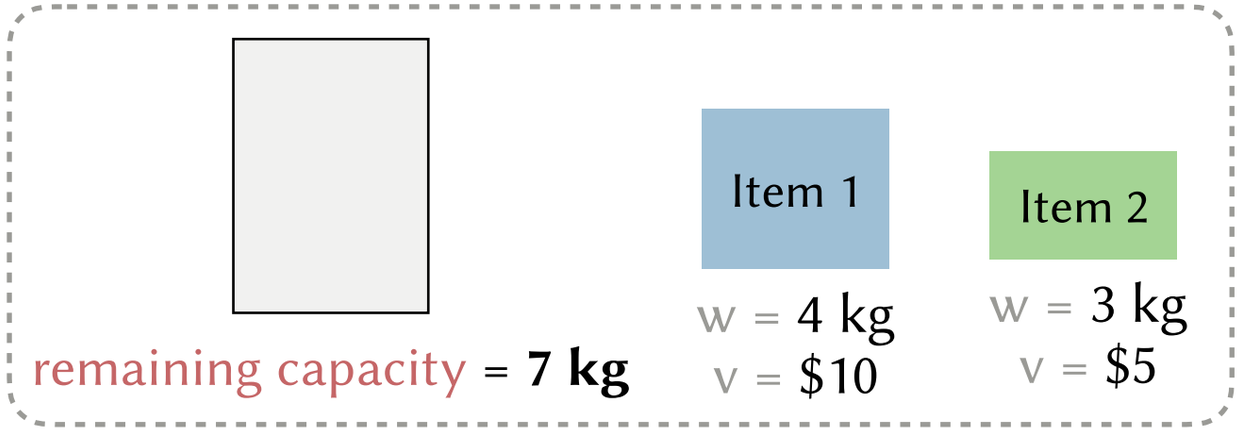
0-1 **Knapsack** (Optimal Substructure)

Assume:

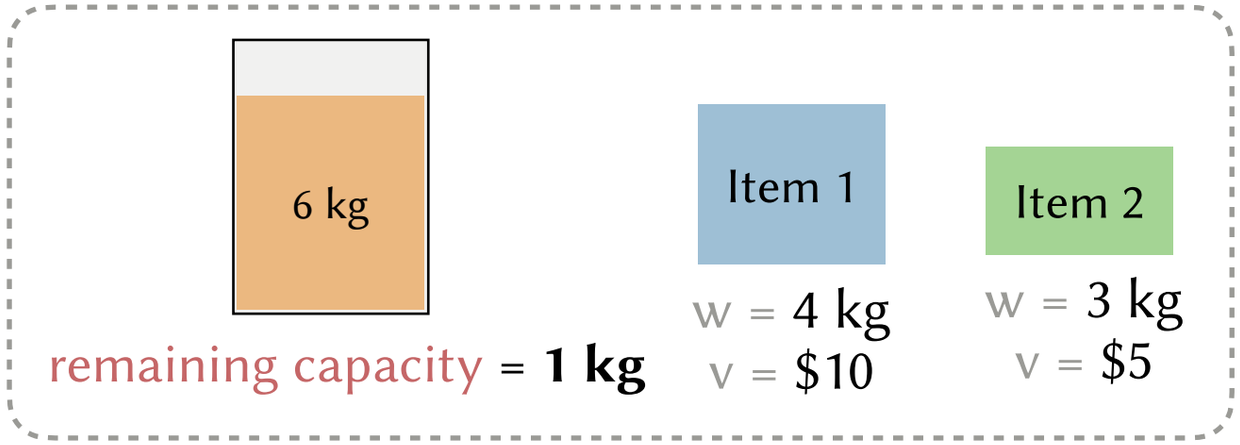
- Items are numbered `1` to `N`.
- **`w`** is an array of weights, where **`w`**`[i]` = *weight* of item `i`.
- **`v`** is an array of values, where **`v`**`[i]` = *value* of item `i`.
- **`Knapsack`**`(i, W)`: Solve the knapsack problem for *items* `1` to `i`, assuming the capacity is `W`.



`Knapsack(3, 7)`



`Knapsack(2, 7)`



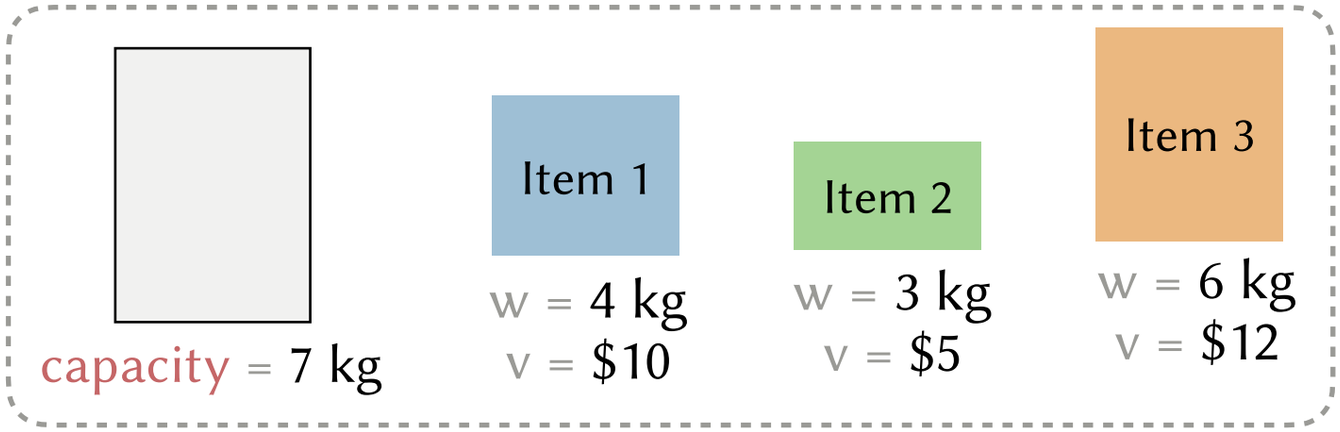
`Knapsack(2, 1)`

0-1 **Knapsack** (Optimal Substructure)

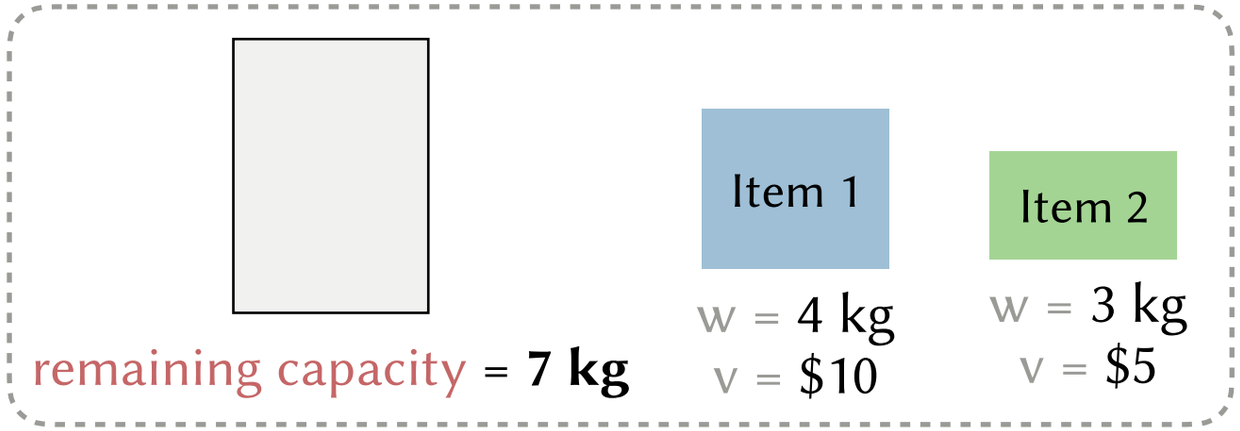
Assume:

- Items are numbered `1` to `N`.
- **`w`** is an array of weights, where **`w`**`[i]` = *weight* of item `i`.
- **`v`** is an array of values, where **`v`**`[i]` = *value* of item `i`.
- **`Knapsack`**`(i, W)`: Solve the knapsack problem for *items* `1` to `i`, assuming the capacity is `W`.

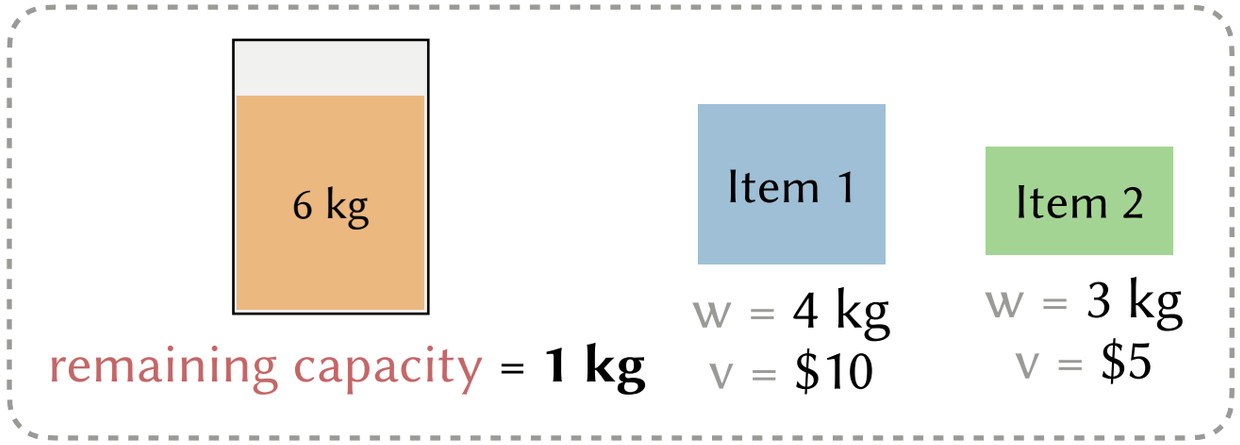
We are interested in solving **`Knapsack`**`(N, W)`.



`Knapsack``(3, 7)`



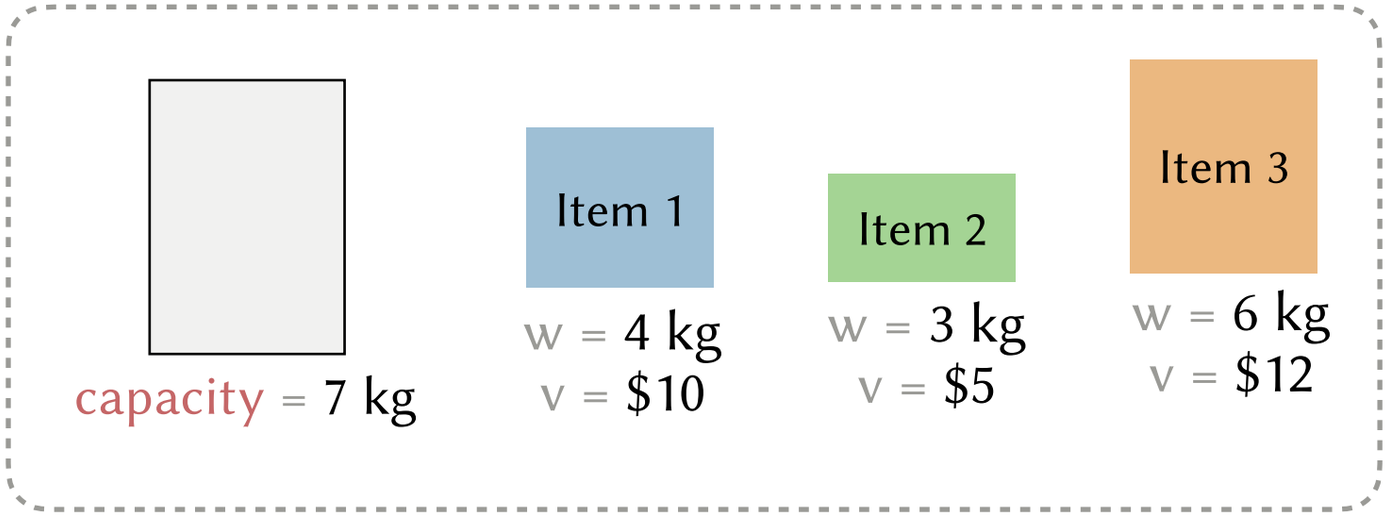
`Knapsack``(2, 7)`



`Knapsack``(2, 1)`

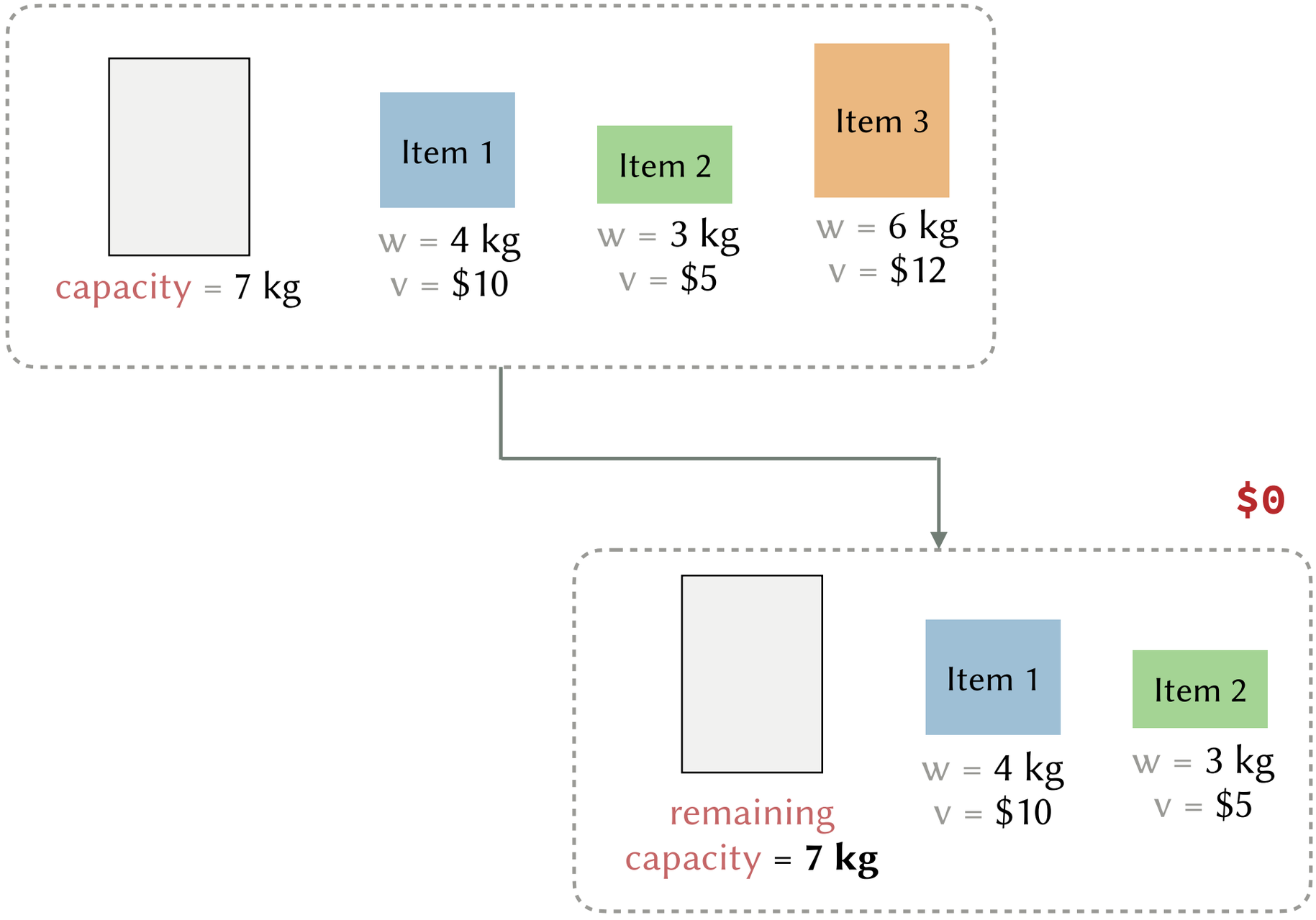
0-1 **Knapsack** (Optimal Substructure)

Two possible choices each leading to a new smaller problem!



0-1 **Knapsack** (Optimal Substructure)

Two possible choices each leading to a new smaller problem!

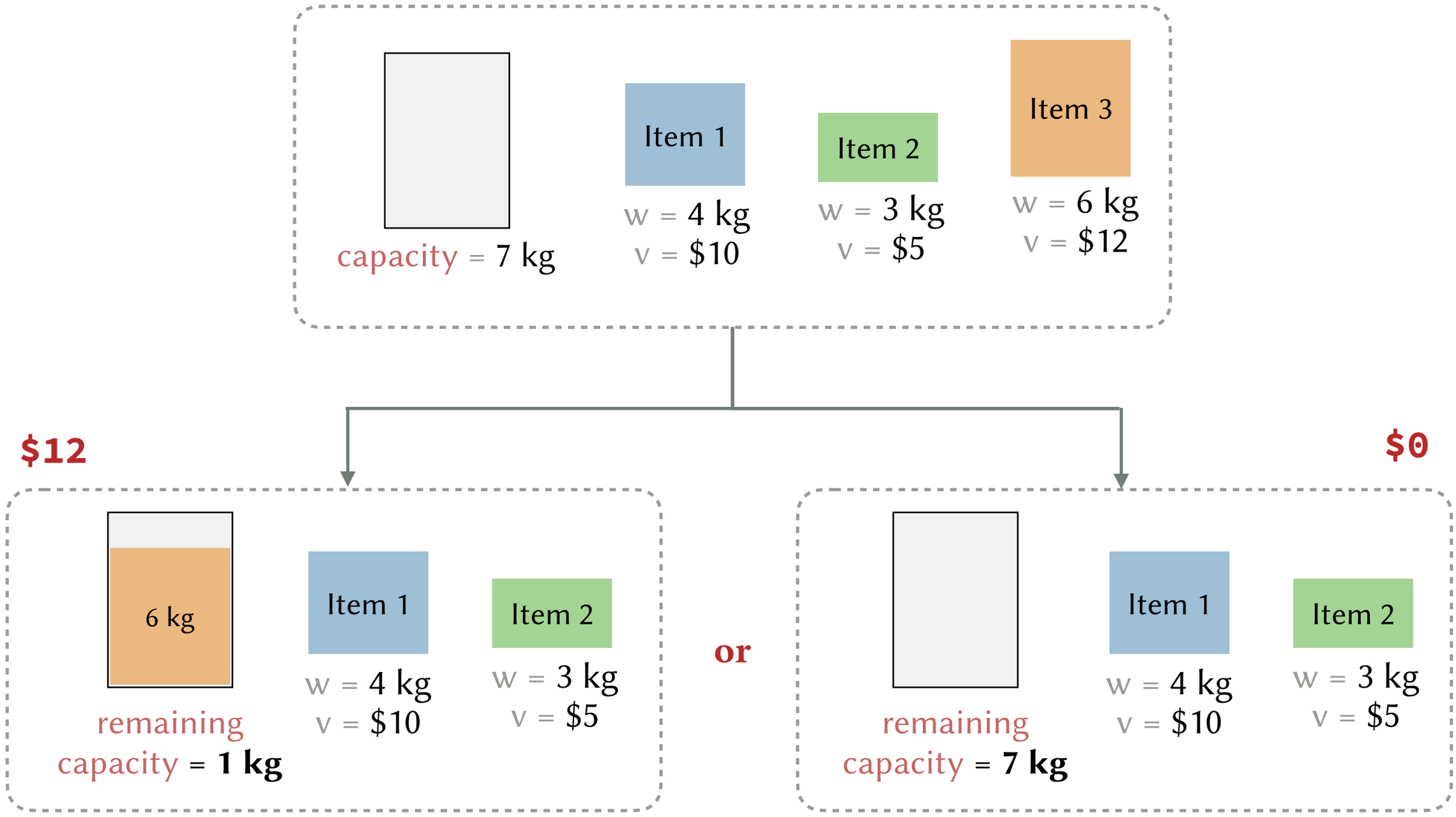


don't take **item 3**

capacity is the same but the number of items decreased

0-1 **Knapsack** (Optimal Substructure)

Two possible choices each leading to a new smaller problem!



***take* item 3**

capacity and number of items decreased

***don't take* item 3**

capacity is the same but the number of items decreased

0-1 **Knapsack** (Optimal Substructure)

Bellman Equation.

```
Knapsack(i, W) = MAX(choice 1,
                     choice 2
```

0-1 **Knapsack** (Optimal Substructure)

Bellman Equation.

```
Knapsack(i, W) = MAX(take item i,
                     don't take item i)
```

0-1 **Knapsack** (Optimal Substructure)

Bellman Equation.

```
Knapsack(i, W) = MAX(take item i,
                     Knapsack(i-1, W       ) + 0)
```

capacity stays the *same*

nothing is added to the total *value*

0-1 **Knapsack** (Optimal Substructure)

we *lost* some of the *capacity*

we *gained* some *value*

Bellman Equation.

```
Knapsack(i, W) = MAX(Knapsack(i-1, W - w[i]) + v[i],
                     Knapsack(i-1, W       ) + 0)
```

capacity stays the *same*

nothing is added to the total *value*

0-1 **Knapsack** (Optimal Substructure)

Bellman Equation.

```
Knapsack(i, W) = MAX(Knapsack(i-1, W - w[i]) + v[i],
                     Knapsack(i-1, W       ) + 0)
```

Special Cases:

- `Knapsack(i, W)= 0` `if i = 0` ← no more items!

0-1 **Knapsack** (Optimal Substructure)

Bellman Equation.

```
Knapsack(i, W) = MAX(Knapsack(i-1, W - w[i]) + v[i],
                     Knapsack(i-1, W       ) + 0)
```

Special Cases:

- `Knapsack(i, W)= 0` `if i = 0` ← no more items!
- `Knapsack(i, W)= Knapsack(i-1, W)` `if w[i] > W` ← knapsack can't take item *i*

↑ consider only the second choice (don't take the item)

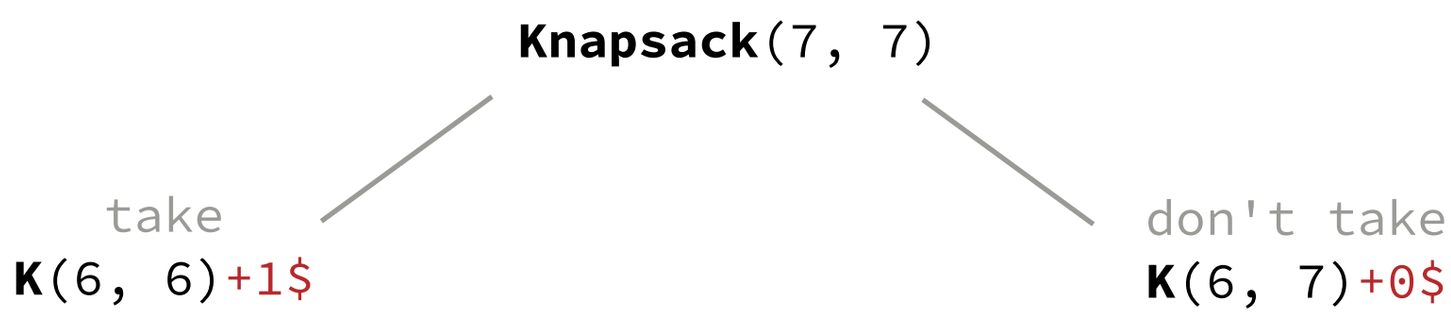
0-1 **Knapsack** (Optimal Substructure)

Trace. Assume **all** items have *value* 1 and *weight* 1 and that $W = 7$.

Knapsack(7, 7)

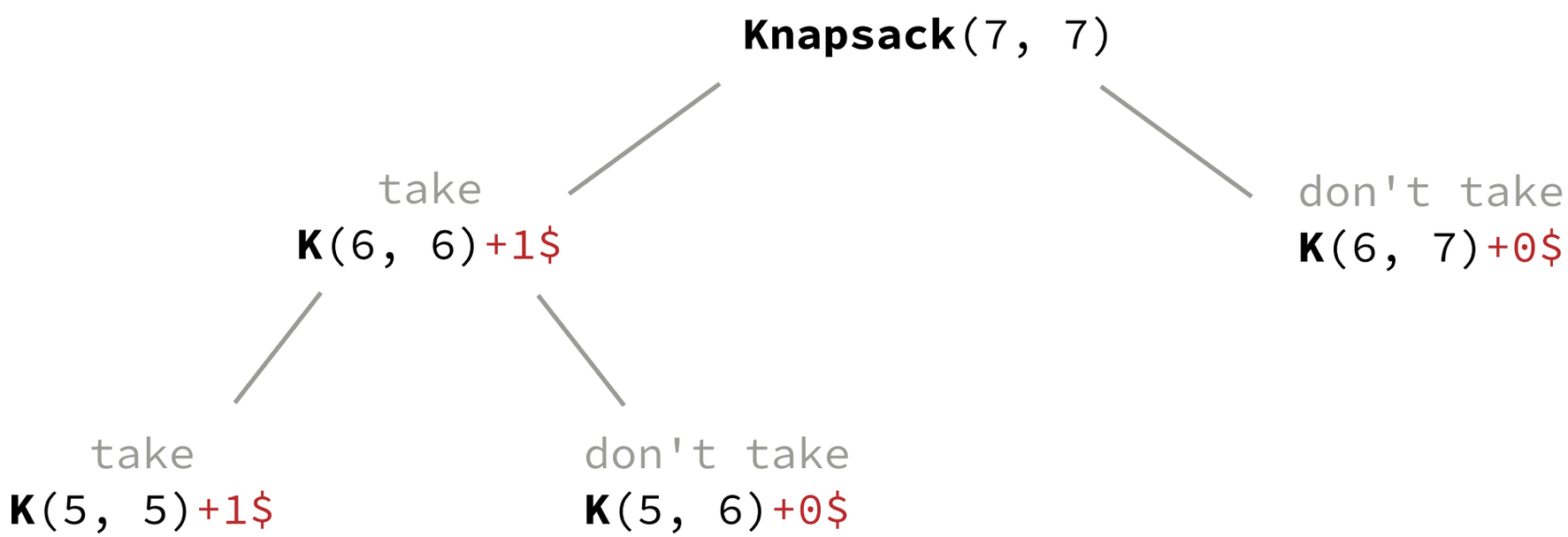
0-1 **Knapsack** (Optimal Substructure)

Trace. Assume **all** items have *value* 1 and *weight* 1 and that $W = 7$.



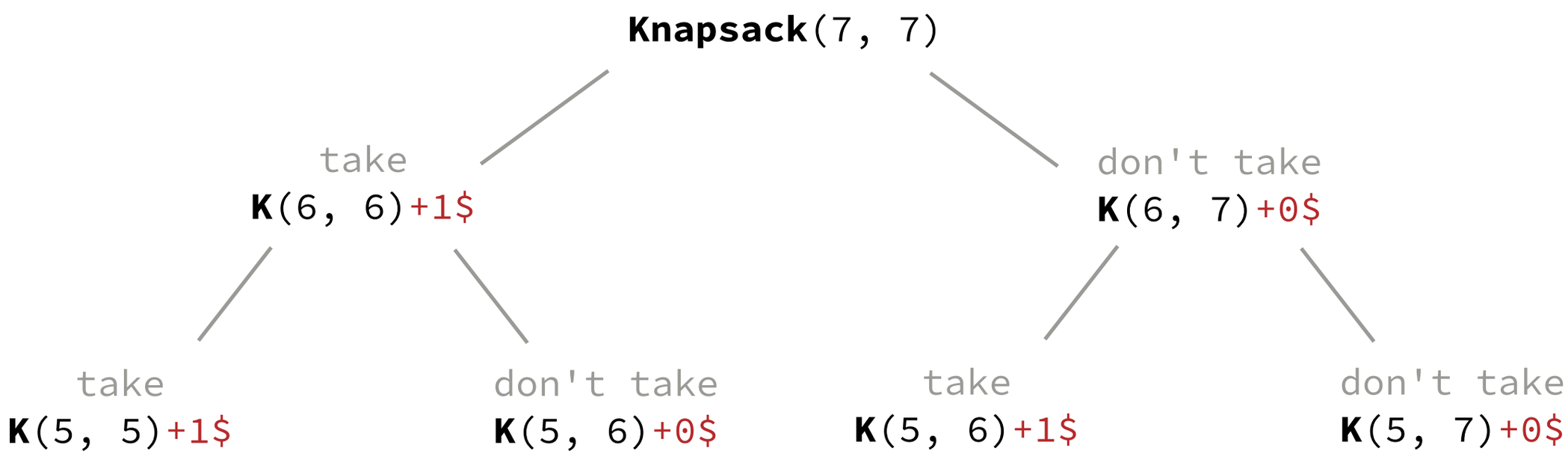
0-1 **Knapsack** (Optimal Substructure)

Trace. Assume **all** items have *value* 1 and *weight* 1 and that $W = 7$.



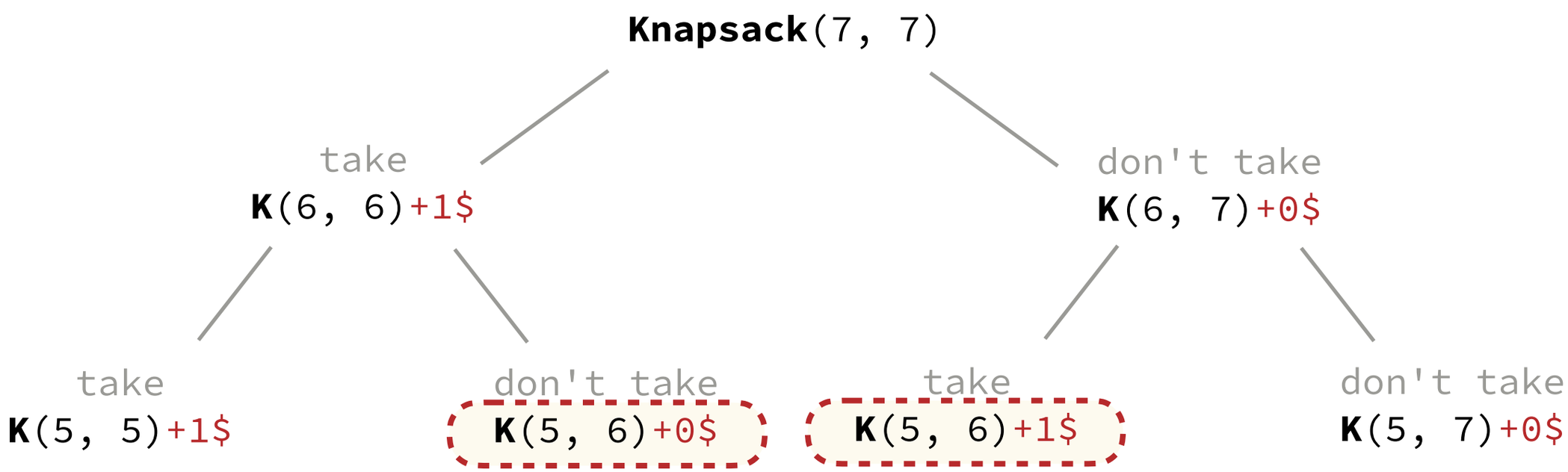
0-1 **Knapsack** (Optimal Substructure)

Trace. Assume **all** items have *value* 1 and *weight* 1 and that $W = 7$.



0-1 **Knapsack** (Optimal Substructure)

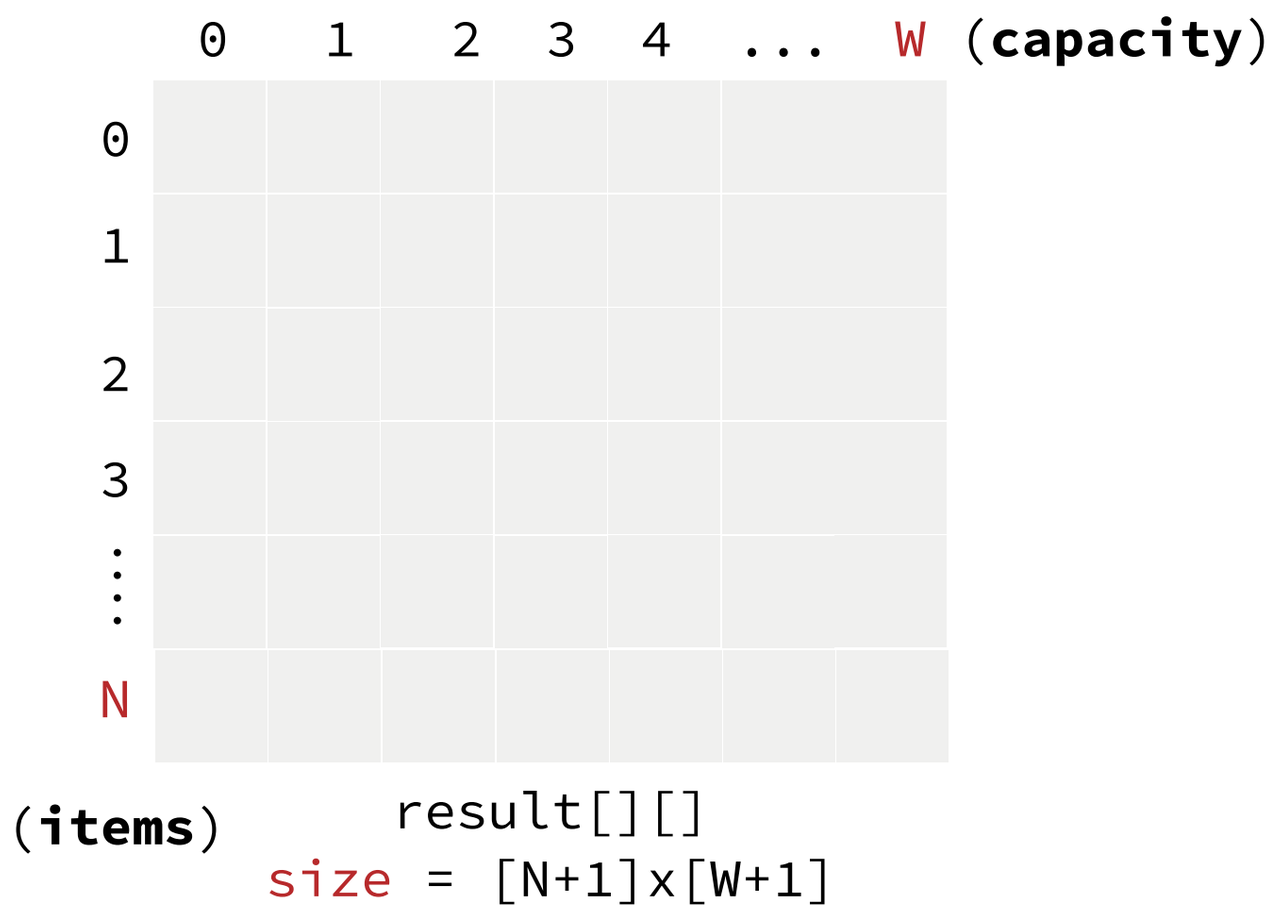
Trace. Assume **all** items have *value* 1 and *weight* 1 and that $W = 7$.



overlapping subproblems!

0-1 Knapsack (Memoized Solution)

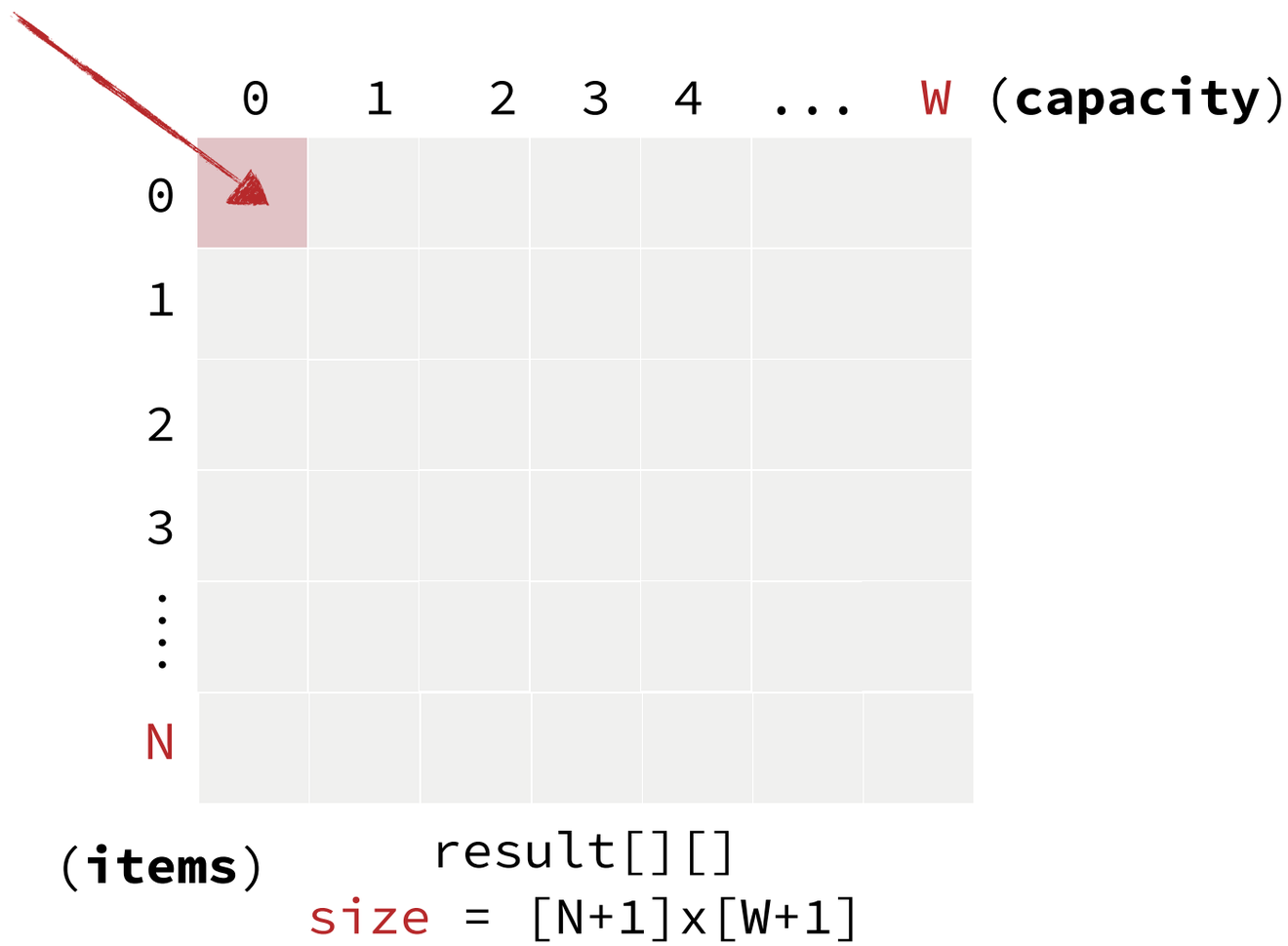
Use a table to store solutions for subproblems.



0-1 Knapsack (Memoized Solution)

Use a table to store solutions for subproblems.

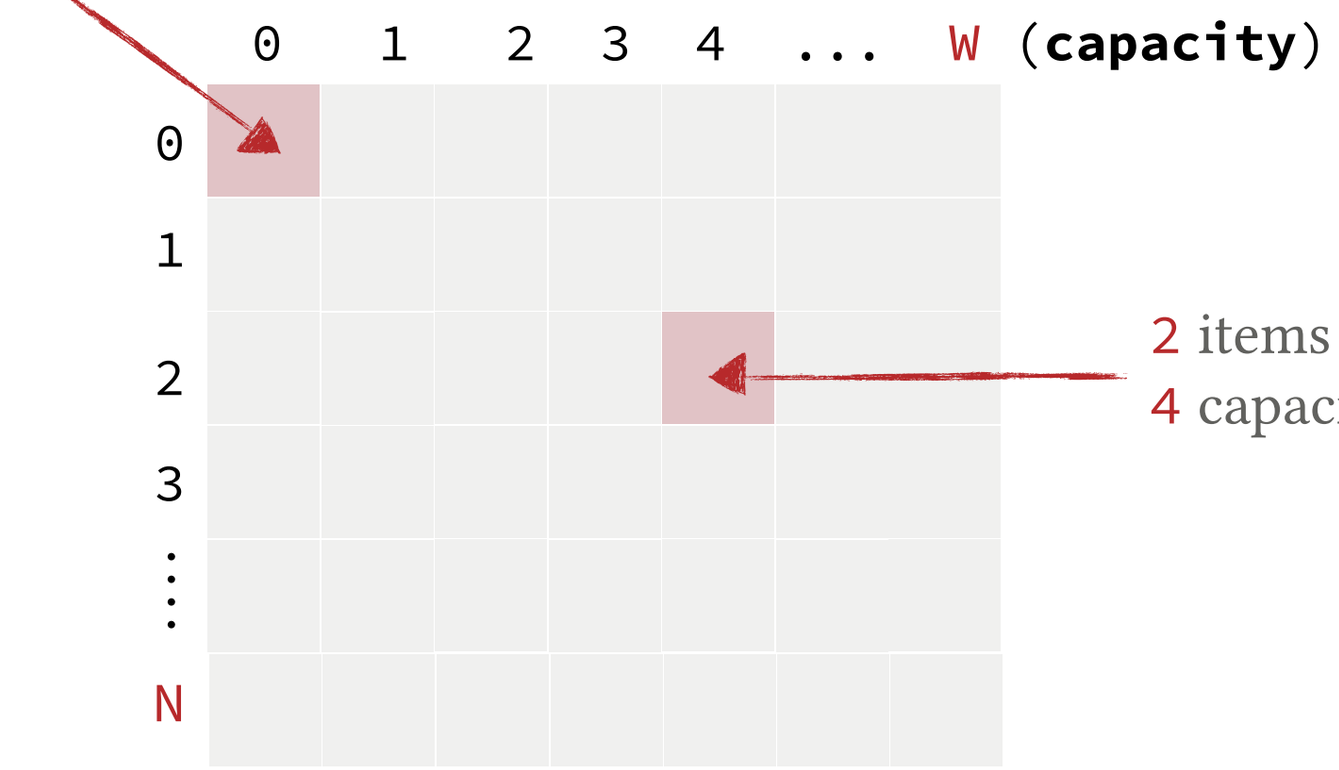
0 items
0 capacity



0-1 Knapsack (Memoized Solution)

Use a table to store solutions for subproblems.

0 items
0 capacity



2 items
4 capacity

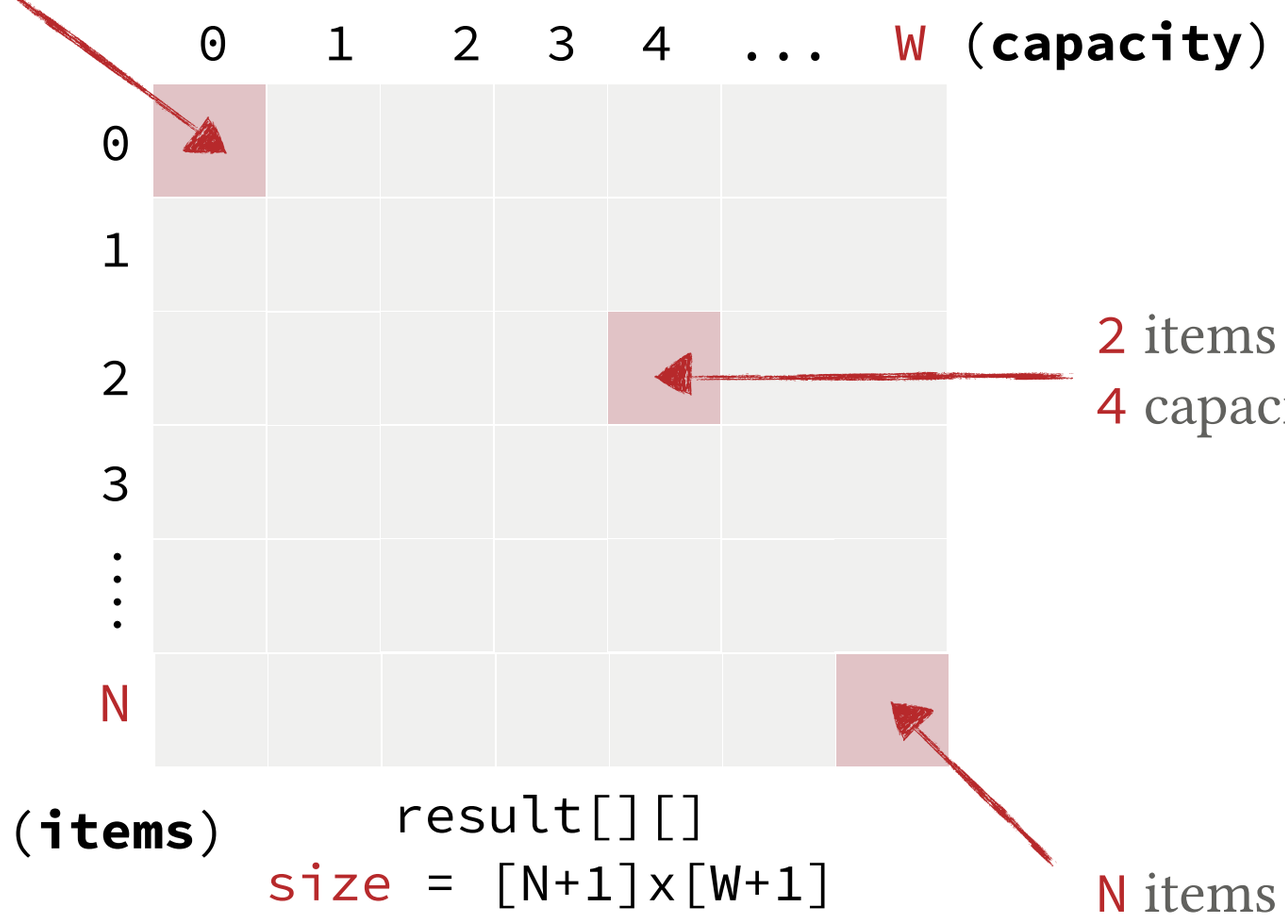
(items)

```
result[][]
size = [N+1]x[W+1]
```

0-1 Knapsack (Memoized Solution)

Use a table to store solutions for subproblems.

0 items
0 capacity



2 items
4 capacity

N items
W capacity

Knapsack(N, W)
this is the main problem
we interested in solving

0-1 Knapsack (Memoized Solution)

Initialize the table.

	0	1	2	3	4	...	W (**capacity**)
0	0	0	0	0	0	0	0
1	-1	-1	-1	-1	-1	-1	-1
2	-1	-1	-1	-1	-1	-1	-1
3	-1	-1	-1	-1	-1	-1	-1
⋮	-1	-1	-1	-1	-1	-1	-1
N	-1	-1	-1	-1	-1	-1	-1

(**items**)

base case
(the best we can get if there are 0 items is 0)

result[][]
size = [N+1]x[W+1]

0-1 Knapsack (Memoized Solution)

SOLVE-KNAPSACK()

```
Initialize result[][] to -1
Initialize result[0][] to 0
```

assuming
`W`, `N`, `w[]`, `v[]`
and `result[][]`
are globally accessible

(**capacity**) ⟶

(**items**) ↑	0	1	2	3	4	...	W
0	0	0	0	0	0	0	0
1	-1	-1	-1	-1	-1	-1	-1
2	-1	-1	-1	-1	-1	-1	-1
3	-1	-1	-1	-1	-1	-1	-1
⋮	-1	-1	-1	-1	-1	-1	-1
N	-1	-1	-1	-1	-1	-1	-1

`result[][]`

0-1 Knapsack (Memoized Solution)

SOLVE-KNAPSACK()

```
Initialize result[][] to -1
Initialize result[0][] to 0

KNAPSACK(N, W)

return result[N][W]
```

fill the table

assuming
`W`, `N`, `w[]`, `v[]`
and `result[][]`
are globally accessible

(**capacity**) ⟶

(**items**) ↑	0	1	2	3	4	...	W
0	0	0	0	0	0	0	0
1	-1	-1	-1	-1	-1	-1	-1
2	-1	-1	-1	-1	-1	-1	-1
3	-1	-1	-1	-1	-1	-1	-1
⋮	-1	-1	-1	-1	-1	-1	-1
N	-1	-1	-1	-1	-1	-1	-1

`result[][]`

0-1 Knapsack (Memoized Solution)

```
SOLVE-KNAPSACK()

Initialize result[][] to -1
Initialize result[0][] to 0

KNAPSACK(N, W)

return result[N][W]
```

```
KNAPSACK(i, W)

if (result[i][W] != -1):
   return result[i][W]
```

problem was solved previously

(capacity) ⟶

(items) ↑	0	1	2	3	4	...	W
0	0	0	0	0	0	0	0
1	-1	-1	-1	-1	-1	-1	-1
2	-1	-1	-1	-1	-1	-1	-1
3	-1	-1	-1	-1	-1	-1	-1
⋮	-1	-1	-1	-1	-1	-1	-1
N	-1	-1	-1	-1	-1	-1	-1

result[][]

0-1 Knapsack (Memoized Solution)

```
SOLVE-KNAPSACK()

Initialize result[][] to -1
Initialize result[0][] to 0

KNAPSACK(N, W)

return result[N][W]
```

```
KNAPSACK(i, W)

if (result[i][W] != -1):
   return result[i][W]

choice1 = 0
if (w[i] <= W)
   choice1 = KNAPSACK(i-1, W-w[i]) + v[i]
```

take the item
only if possible

(capacity) ⟶

(items) ↑	0	1	2	3	4	...	W
0	0	0	0	0	0	0	0
1	-1	-1	-1	-1	-1	-1	-1
2	-1	-1	-1	-1	-1	-1	-1
3	-1	-1	-1	-1	-1	-1	-1
⋮	-1	-1	-1	-1	-1	-1	-1
N	-1	-1	-1	-1	-1	-1	-1

result[][]

0-1 Knapsack (Memoized Solution)

```
SOLVE-KNAPSACK()

Initialize result[][] to -1
Initialize result[0][] to 0

KNAPSACK(N, W)

return result[N][W]
```

```
KNAPSACK(i, W)

if (result[i][W] != -1):
   return result[i][W]

choice1 = 0
if (w[i] <= W)
   choice1 = KNAPSACK(i-1, W-w[i]) + v[i]

choice2 = KNAPSACK(i-1, W)
```

it is always possible
not to take the item!

(capacity) ⟶

(items) ↑	0	1	2	3	4	...	W
0	0	0	0	0	0	0	0
1	-1	-1	-1	-1	-1	-1	-1
2	-1	-1	-1	-1	-1	-1	-1
3	-1	-1	-1	-1	-1	-1	-1
⋮	-1	-1	-1	-1	-1	-1	-1
N	-1	-1	-1	-1	-1	-1	-1

result[][]

0-1 Knapsack (Memoized Solution)

```
SOLVE-KNAPSACK()

Initialize result[][] to -1
Initialize result[0][] to 0

KNAPSACK(N, W)

return result[N][W]
```

```
KNAPSACK(i, W)

if (result[i][W] != -1):
   return result[i][W]

choice1 = 0
if (w[i] <= W)
   choice1 = KNAPSACK(i-1, W-w[i]) + v[i]

choice2 = KNAPSACK(i-1, W)

result[i][W] = MAX(choice1, choice2)
```

pick the better choice and store the result

(capacity) ⟶

(items) ↑	0	1	2	3	4	...	W
0	0	0	0	0	0	0	0
1	-1	-1	-1	-1	-1	-1	-1
2	-1	-1	-1	-1	-1	-1	-1
3	-1	-1	-1	-1	-1	-1	-1
⋮	-1	-1	-1	-1	-1	-1	-1
N	-1	-1	-1	-1	-1	-1	-1

result[][]

0-1 Knapsack (Memoized Solution)

```
SOLVE-KNAPSACK()

Initialize result[][] to -1
Initialize result[0][] to 0

KNAPSACK(N, W)

return result[N][W]
```

```
KNAPSACK(i, W)

if (result[i][W] != -1):
   return result[i][W]

choice1 = 0
if (w[i] <= W)
   choice1 = KNAPSACK(i-1, W-w[i]) + v[i]

choice2 = KNAPSACK(i-1, W)

result[i][W] = MAX(choice1, choice2)

return result[i][W]
```

(capacity) ⟶

(items) ↑	0	1	2	3	4	...	W
0	0	0	0	0	0	0	0
1	-1	-1	-1	-1	-1	-1	-1
2	-1	-1	-1	-1	-1	-1	-1
3	-1	-1	-1	-1	-1	-1	-1
⋮	-1	-1	-1	-1	-1	-1	-1
N	-1	-1	-1	-1	-1	-1	-1

result[][]

0-1 Knapsack (Memoized Solution Tracing)

items \ capacity	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	-1	-1	-1	-1	-1	-1	-1	-1
2	-1	-1	-1	-1	-1	-1	-1	-1
3	-1	-1	-1	-1	-1	-1	-1	-1

result[][]

0-1 Knapsack (Memoized Solution Tracing)

items \ capacity	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	-1	-1	-1	-1	-1	-1	-1	-1
2	-1	-1	-1	-1	-1	-1	-1	-1
3	-1	-1	-1	-1	-1	-1	-1	-1

result[][]

K(3,7)

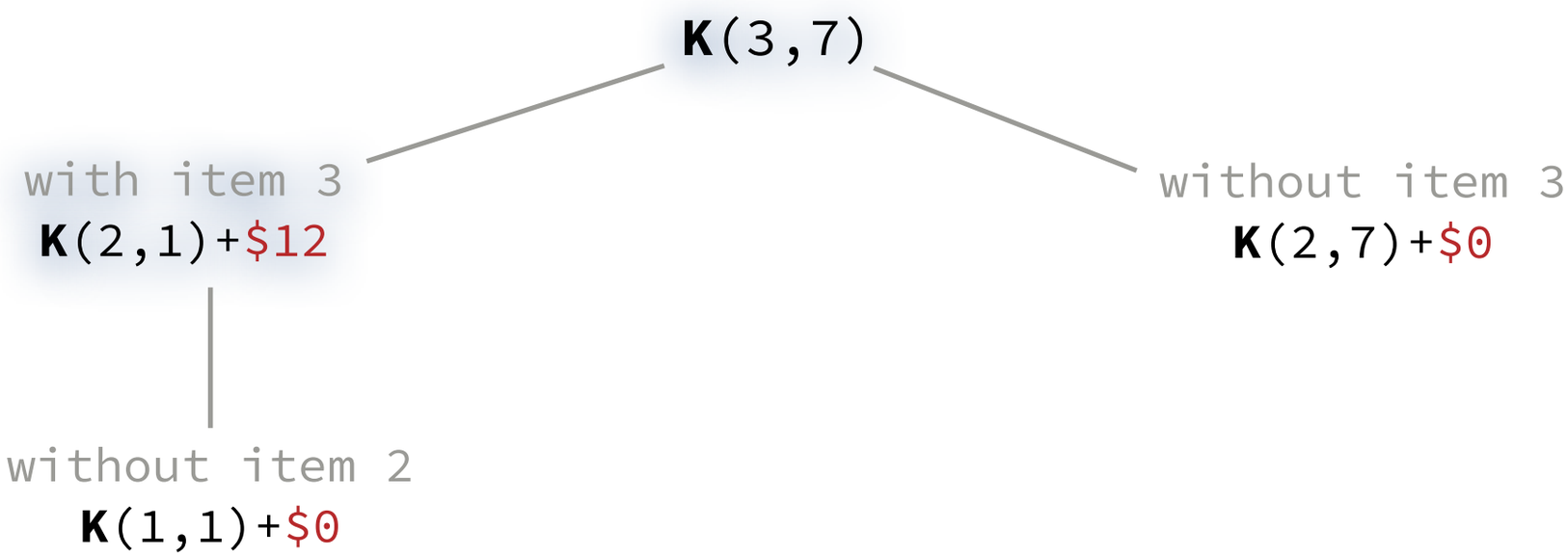
with item 3
K(2,1)+$12

without item 3
K(2,7)+$0

0-1 Knapsack (Memoized Solution Tracing)

items \ capacity	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	-1	-1	-1	-1	-1	-1	-1	-1
2	-1	-1	-1	-1	-1	-1	-1	-1
3	-1	-1	-1	-1	-1	-1	-1	-1

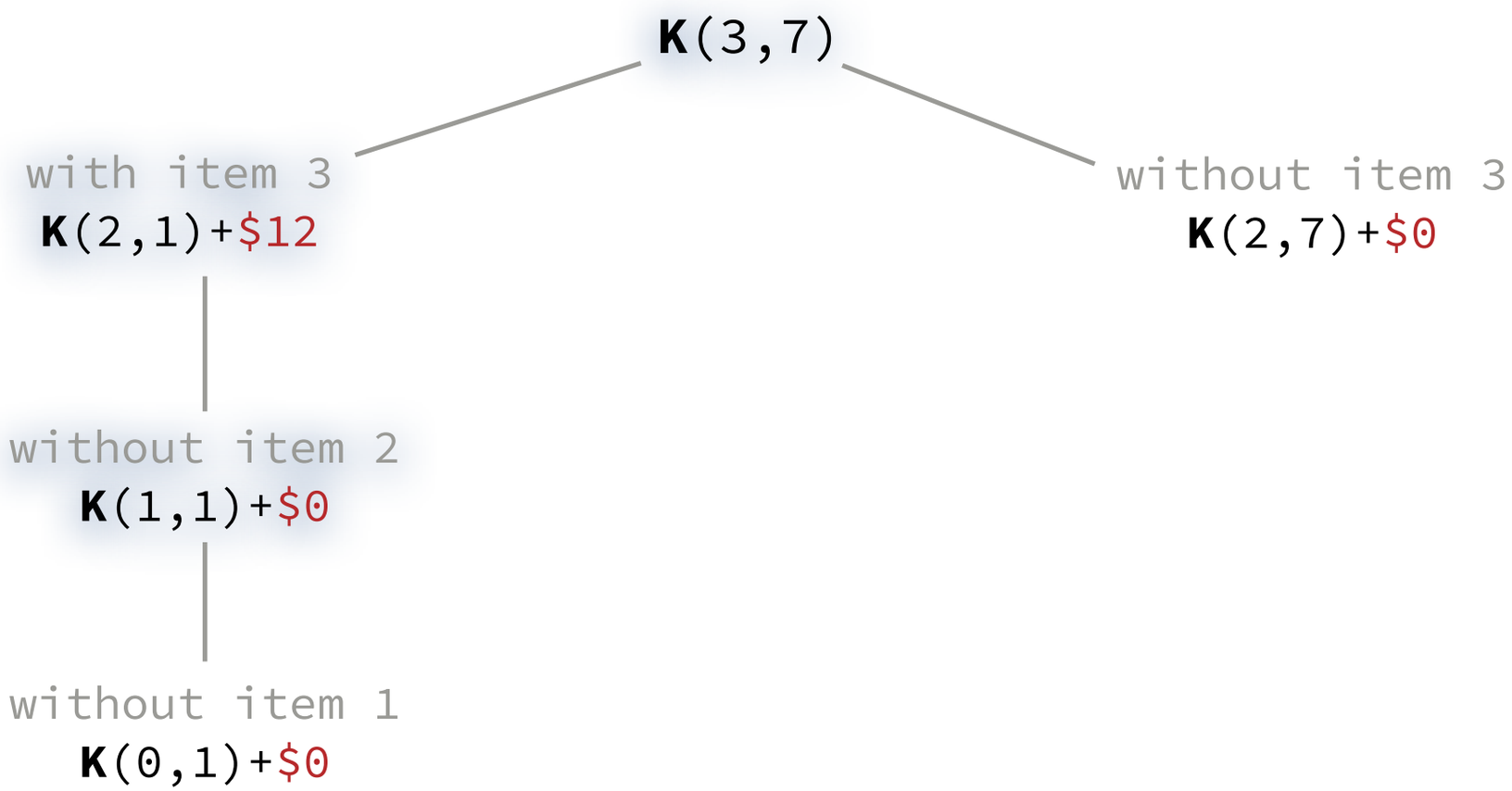
result[][]



0-1 Knapsack (Memoized Solution Tracing)

items \ capacity	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	-1	-1	-1	-1	-1	-1	-1	-1
2	-1	-1	-1	-1	-1	-1	-1	-1
3	-1	-1	-1	-1	-1	-1	-1	-1

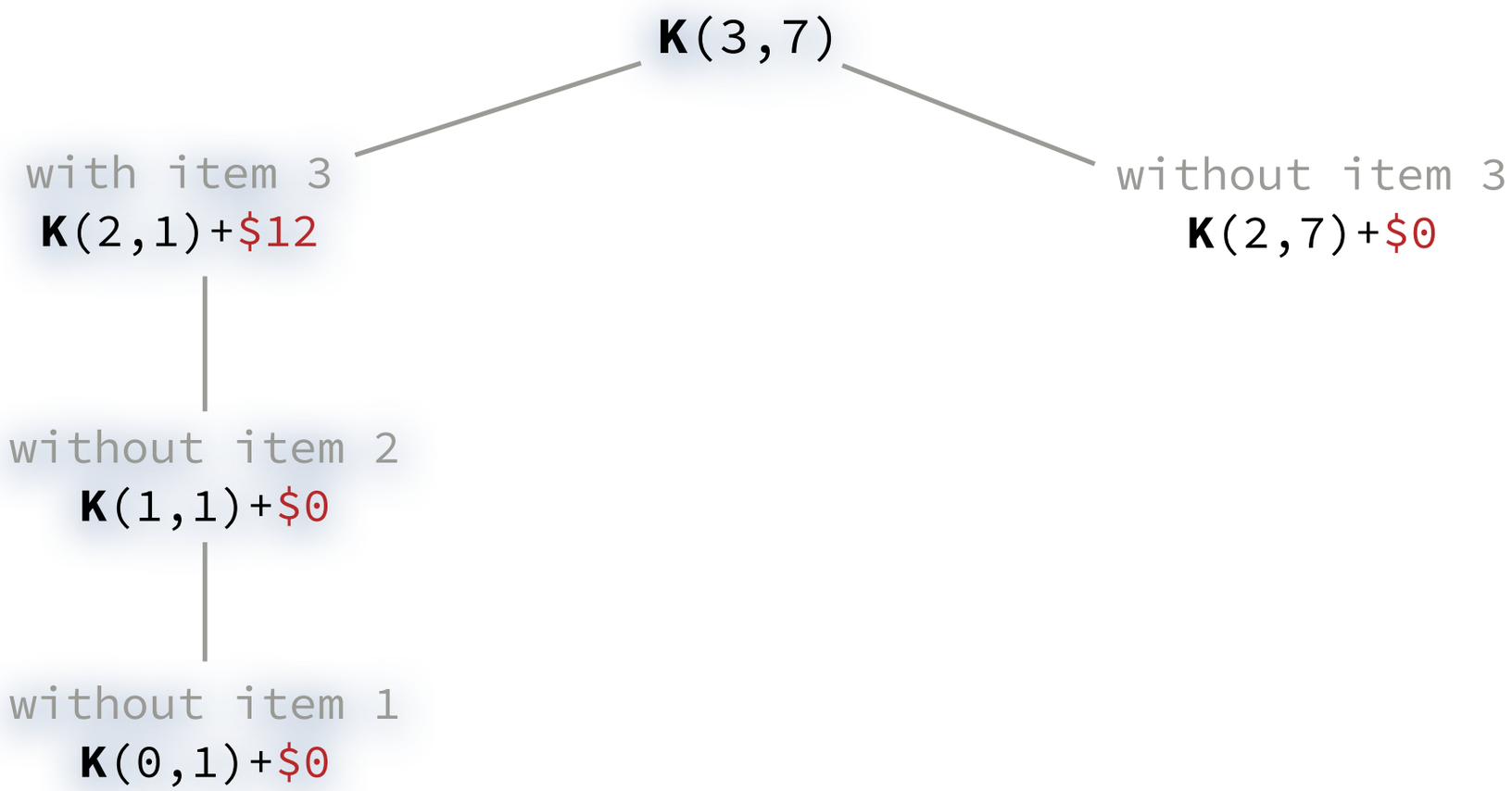
result[][]



0-1 Knapsack (Memoized Solution Tracing)

items \ capacity	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	-1	-1	-1	-1	-1	-1	-1	-1
2	-1	-1	-1	-1	-1	-1	-1	-1
3	-1	-1	-1	-1	-1	-1	-1	-1

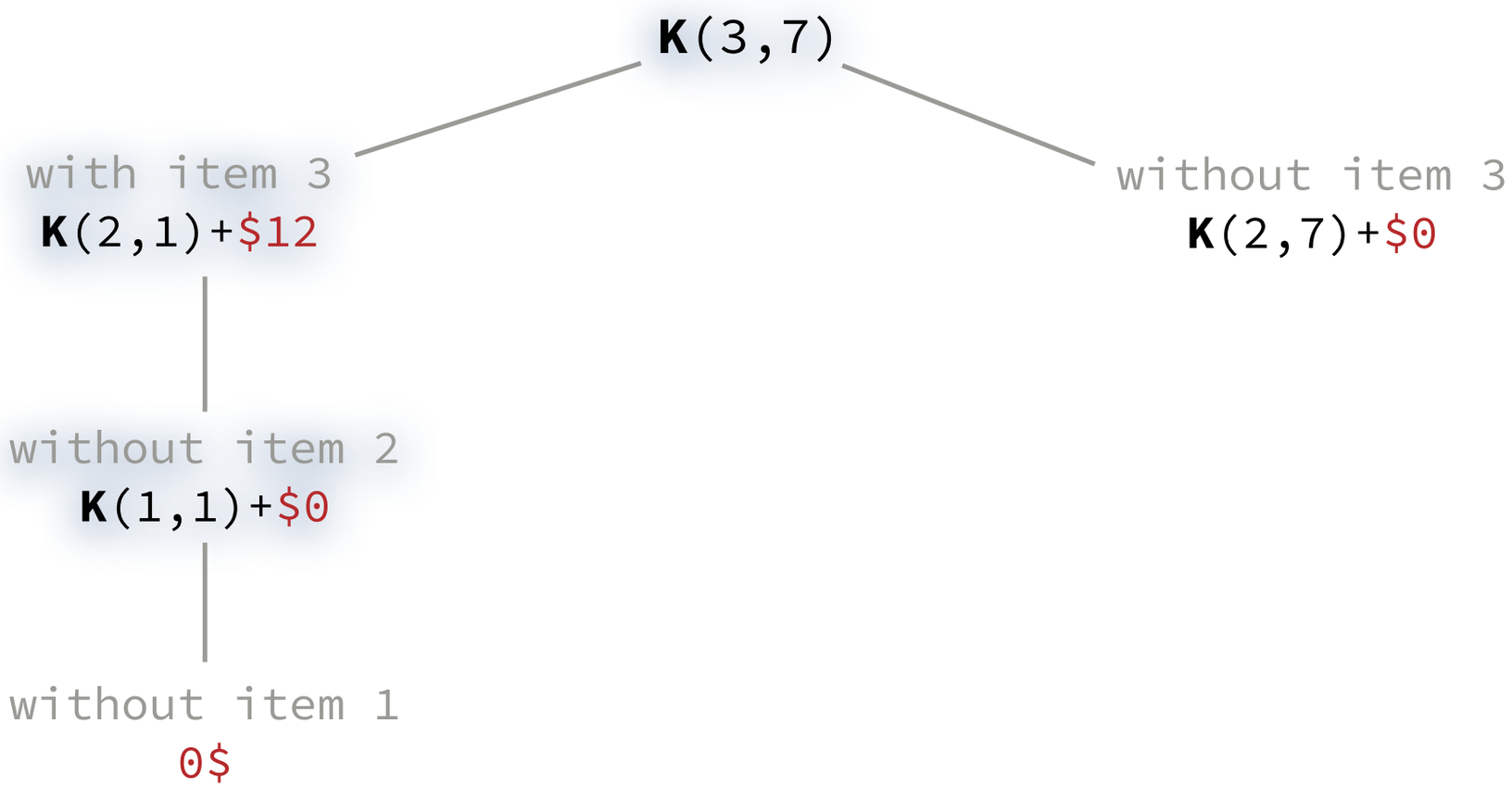
result[][]



0-1 Knapsack (Memoized Solution Tracing)

items \ capacity	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	-1	-1	-1	-1	-1	-1	-1	-1
2	-1	-1	-1	-1	-1	-1	-1	-1
3	-1	-1	-1	-1	-1	-1	-1	-1

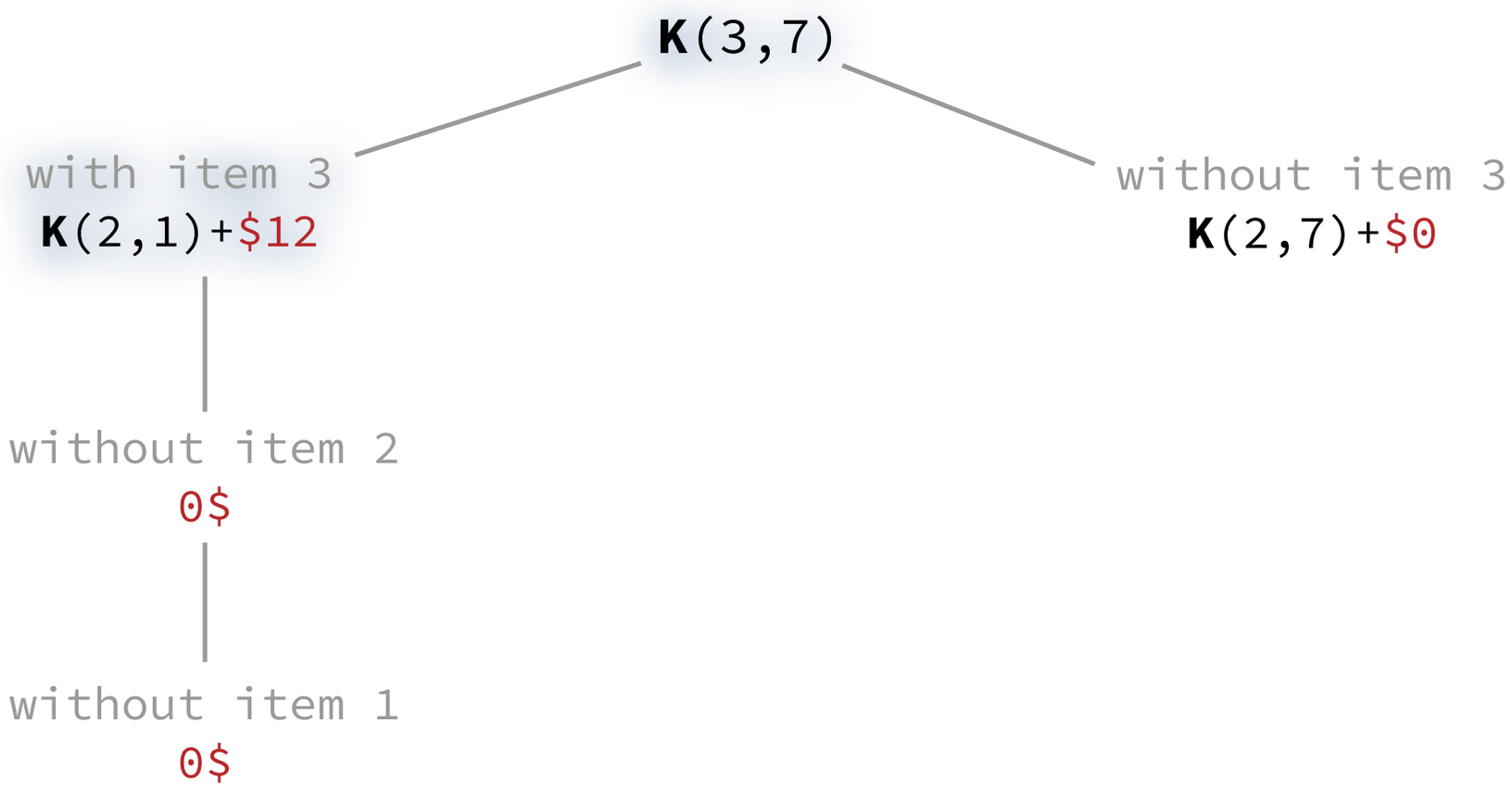
result[][]



0-1 Knapsack (Memoized Solution Tracing)

items \ capacity	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	-1	0	-1	-1	-1	-1	-1	-1
2	-1	-1	-1	-1	-1	-1	-1	-1
3	-1	-1	-1	-1	-1	-1	-1	-1

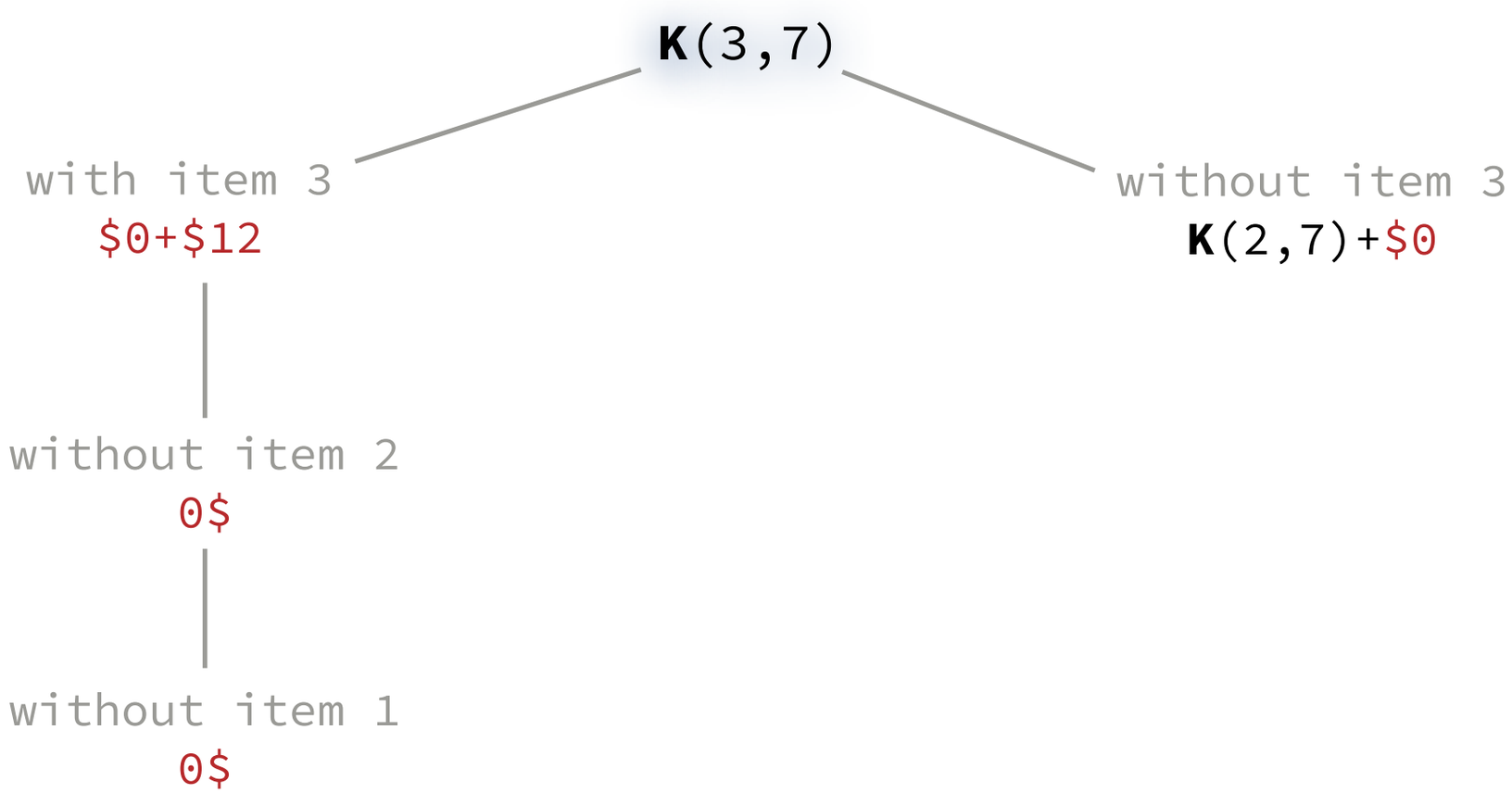
result[][]



0-1 Knapsack (Memoized Solution Tracing)

items \ capacity	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	-1	0	-1	-1	-1	-1	-1	-1
2	-1	0	-1	-1	-1	-1	-1	-1
3	-1	-1	-1	-1	-1	-1	-1	-1

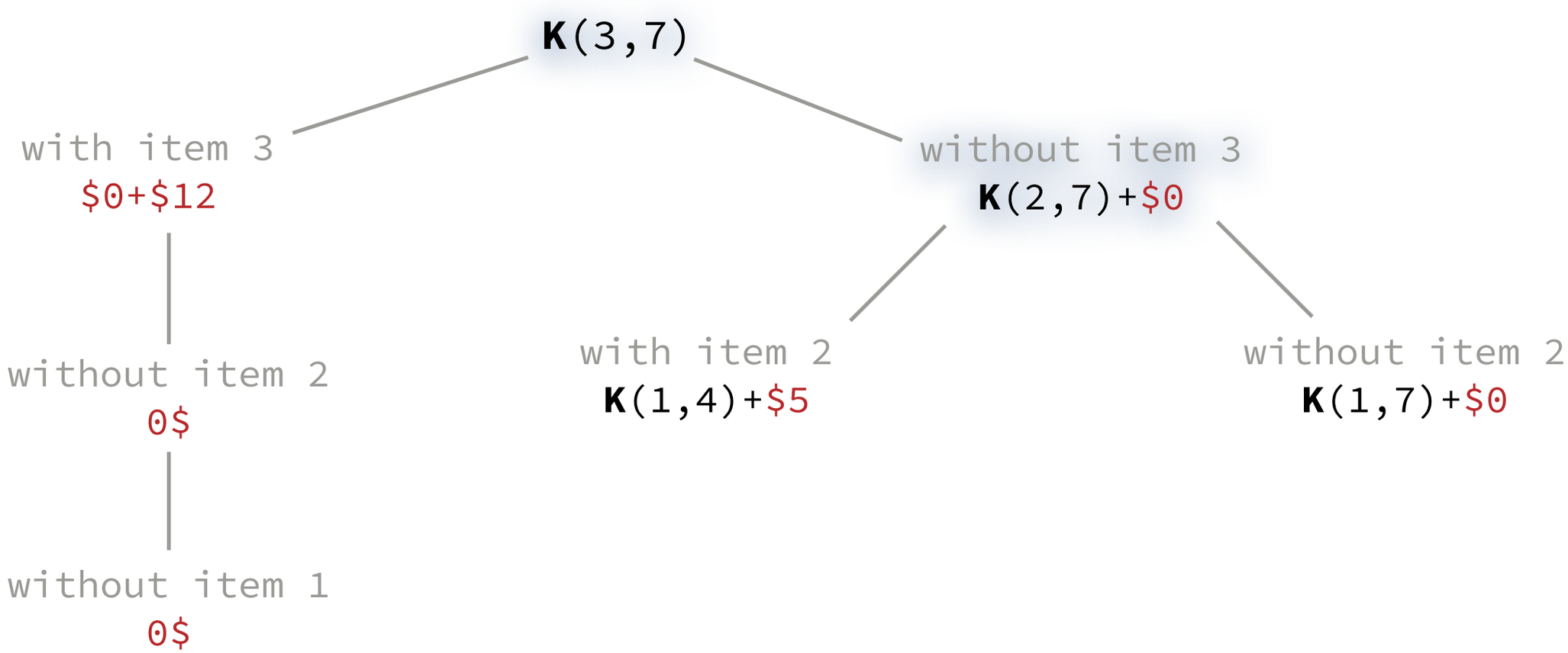
result[][]



0-1 Knapsack (Memoized Solution Tracing)

items \ capacity	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	-1	0	-1	-1	-1	-1	-1	-1
2	-1	0	-1	-1	-1	-1	-1	-1
3	-1	-1	-1	-1	-1	-1	-1	-1

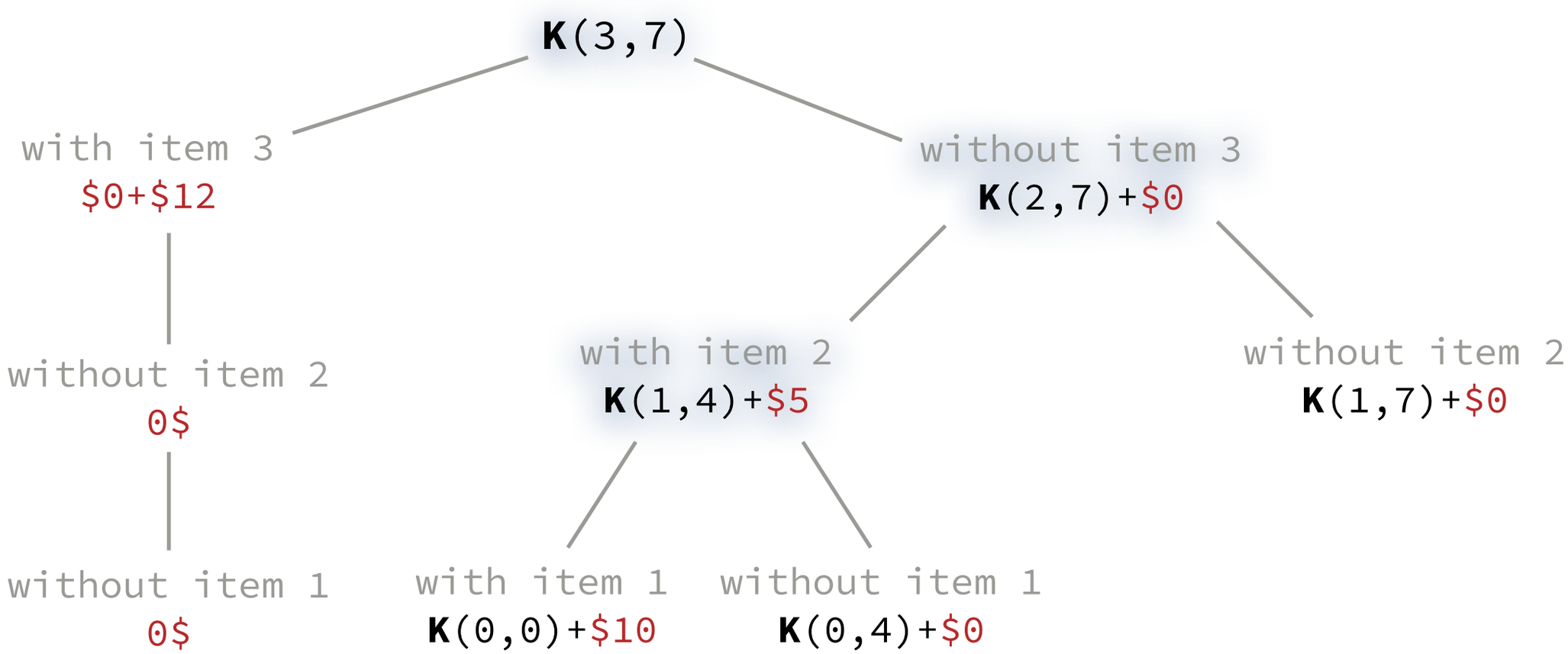
result[][]



0-1 Knapsack (Memoized Solution Tracing)

items \ capacity	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	-1	0	-1	-1	-1	-1	-1	-1
2	-1	0	-1	-1	-1	-1	-1	-1
3	-1	-1	-1	-1	-1	-1	-1	-1

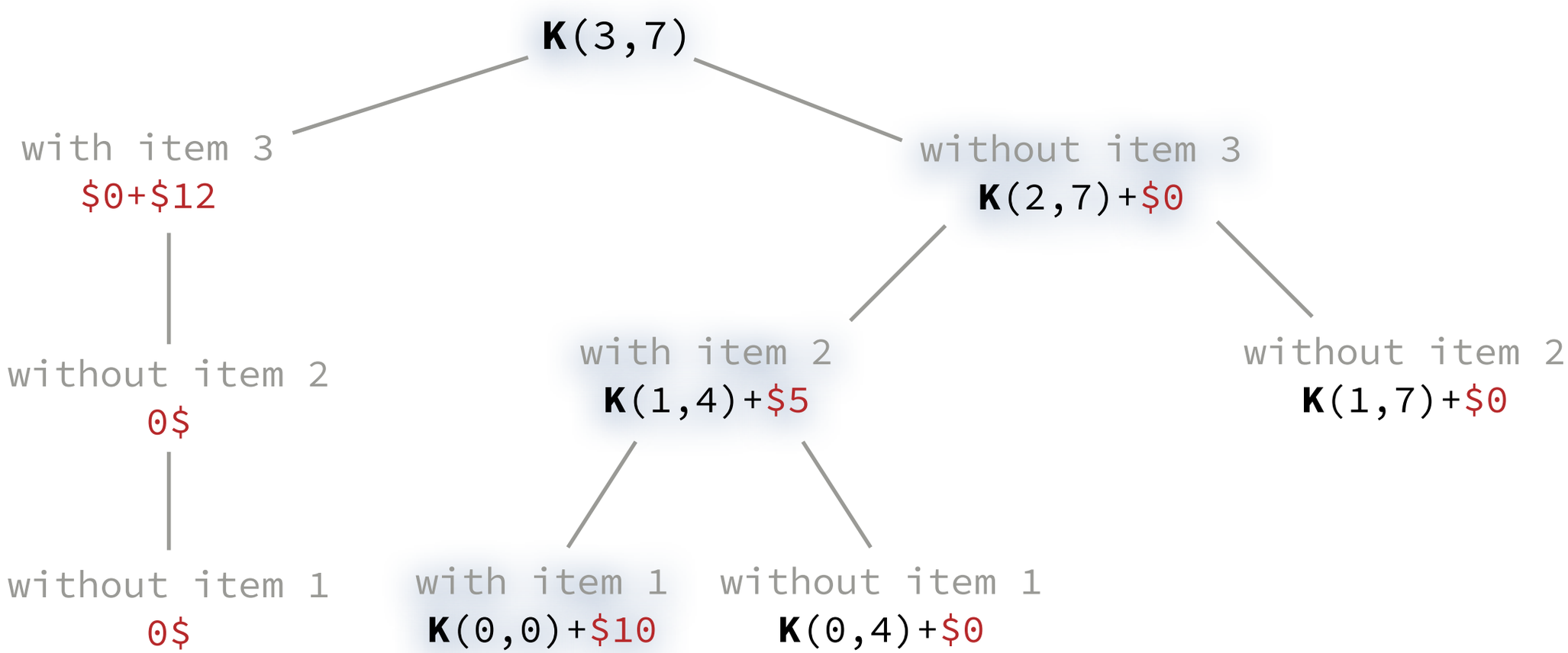
result[][]



0-1 Knapsack (Memoized Solution Tracing)

items \ capacity	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	-1	0	-1	-1	-1	-1	-1	-1
2	-1	0	-1	-1	-1	-1	-1	-1
3	-1	-1	-1	-1	-1	-1	-1	-1

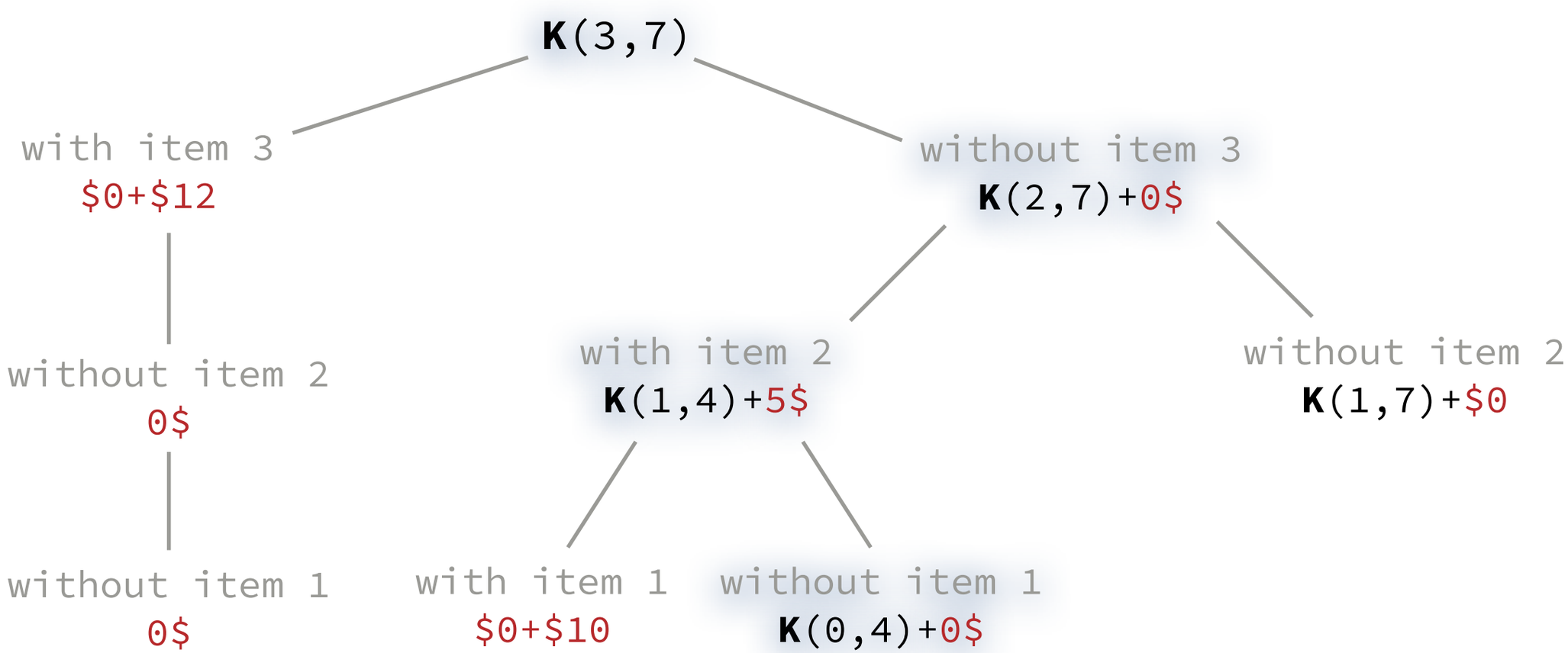
result[][]



0-1 Knapsack (Memoized Solution Tracing)

items \ capacity	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	-1	0	-1	-1	-1	-1	-1	-1
2	-1	0	-1	-1	-1	-1	-1	-1
3	-1	-1	-1	-1	-1	-1	-1	-1

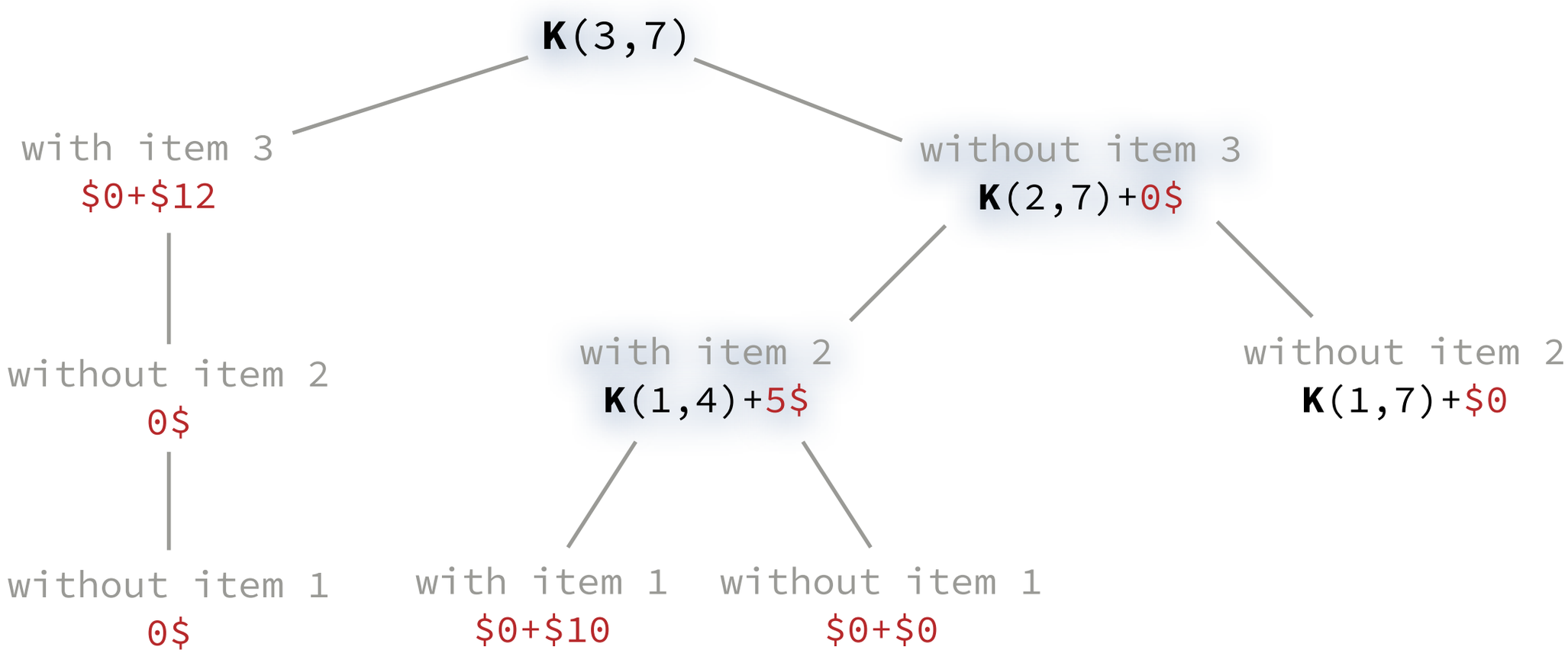
result[][]



0-1 Knapsack (Memoized Solution Tracing)

items \ capacity	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	-1	0	-1	-1	-1	-1	-1	-1
2	-1	0	-1	-1	-1	-1	-1	-1
3	-1	-1	-1	-1	-1	-1	-1	-1

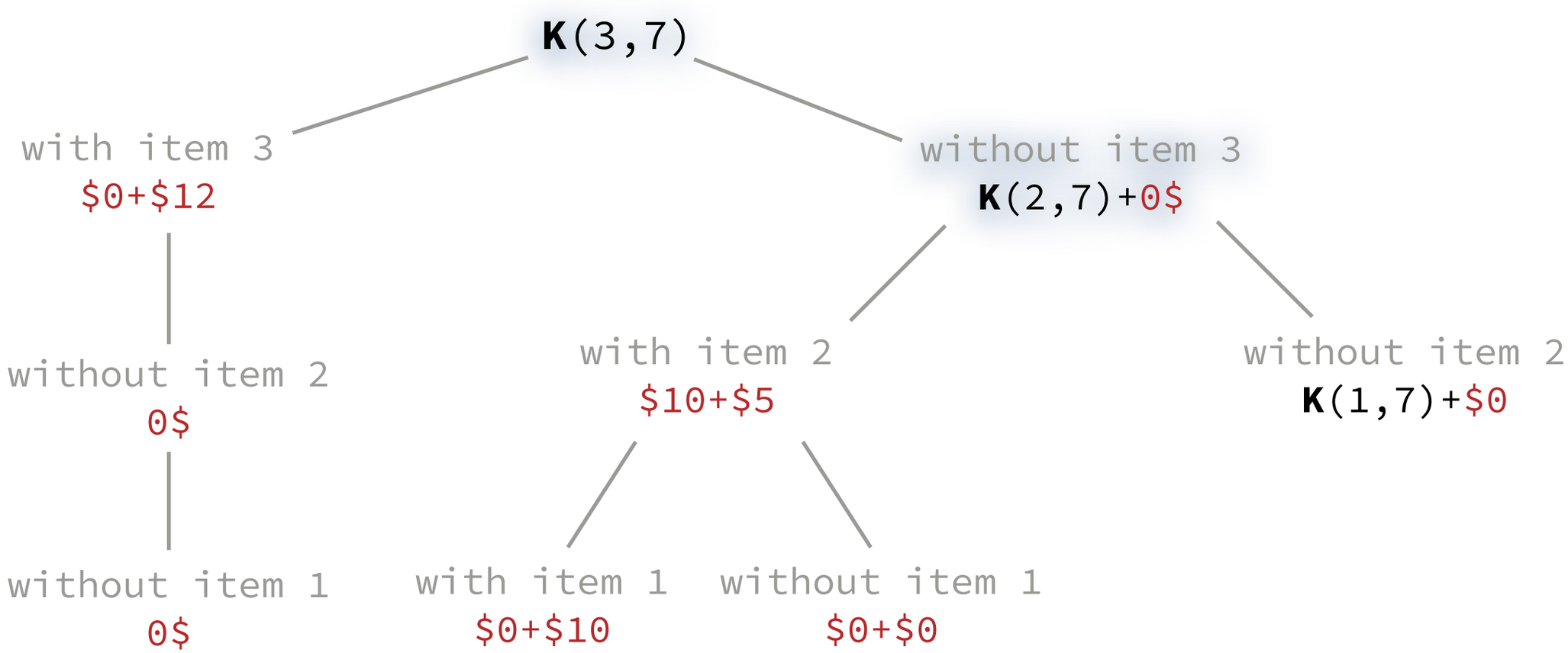
result[][]



0-1 Knapsack (Memoized Solution Tracing)

items \ capacity	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	-1	0	-1	-1	10	-1	-1	-1
2	-1	0	-1	-1	-1	-1	-1	-1
3	-1	-1	-1	-1	-1	-1	-1	-1

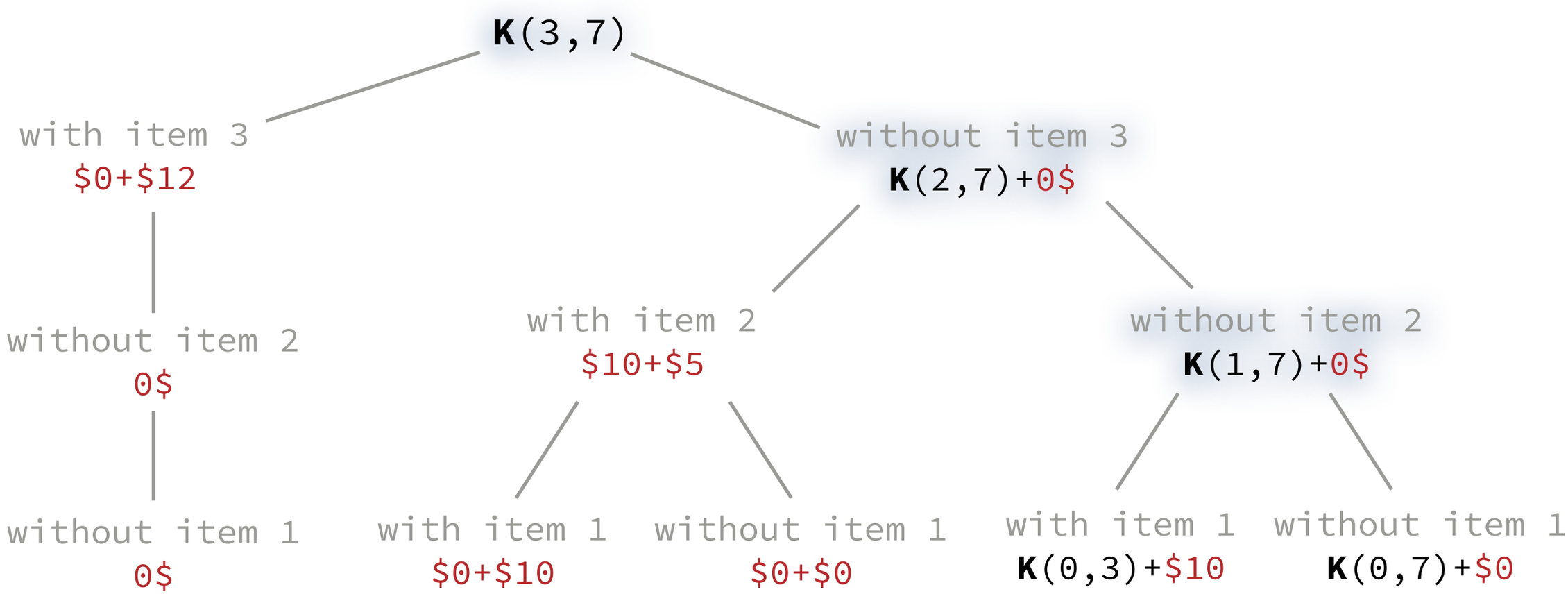
result[][]



0-1 Knapsack (Memoized Solution Tracing)

items \ capacity	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	-1	0	-1	-1	10	-1	-1	-1
2	-1	0	-1	-1	-1	-1	-1	-1
3	-1	-1	-1	-1	-1	-1	-1	-1

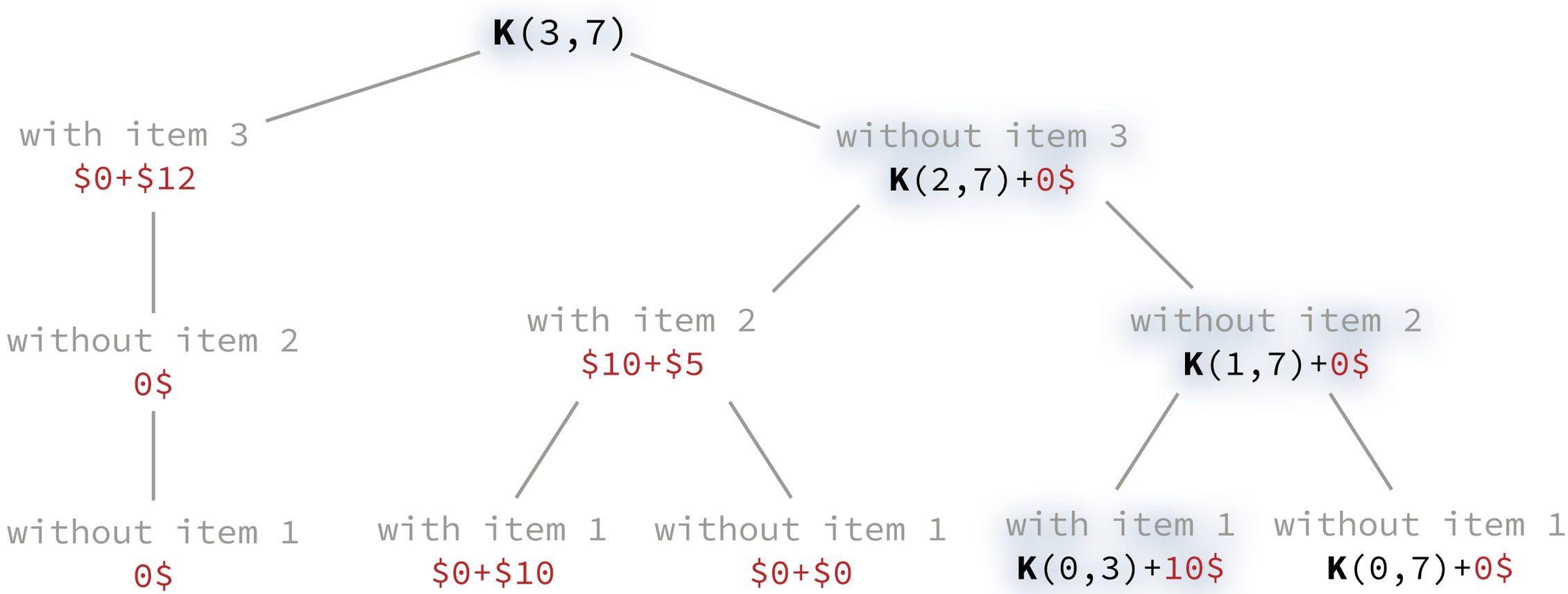
result[][]



0-1 Knapsack (Memoized Solution Tracing)

items \ capacity	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	-1	0	-1	-1	10	-1	-1	-1
2	-1	0	-1	-1	-1	-1	-1	-1
3	-1	-1	-1	-1	-1	-1	-1	-1

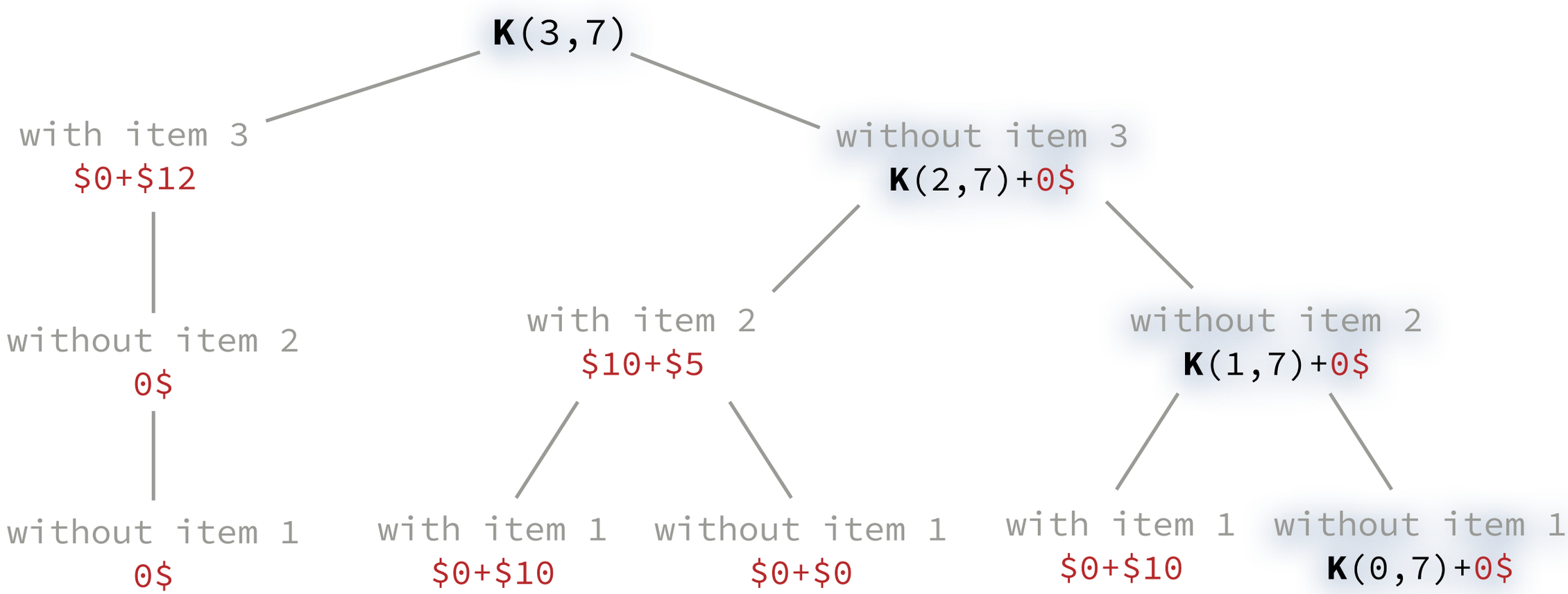
result[][]



0-1 Knapsack (Memoized Solution Tracing)

items \ capacity	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	-1	0	-1	-1	10	-1	-1	-1
2	-1	0	-1	-1	-1	-1	-1	-1
3	-1	-1	-1	-1	-1	-1	-1	-1

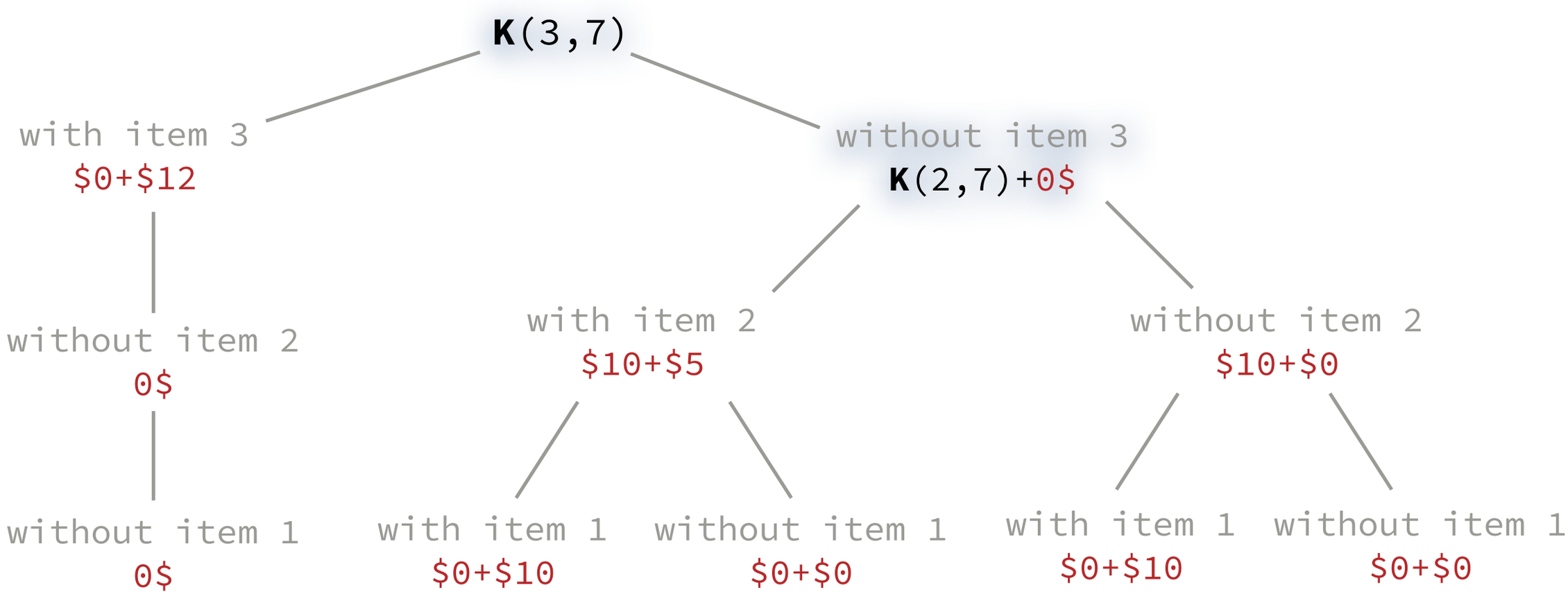
result[][]



0-1 Knapsack (Memoized Solution Tracing)

items \ capacity	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	-1	0	-1	-1	10	-1	-1	10
2	-1	0	-1	-1	-1	-1	-1	-1
3	-1	-1	-1	-1	-1	-1	-1	-1

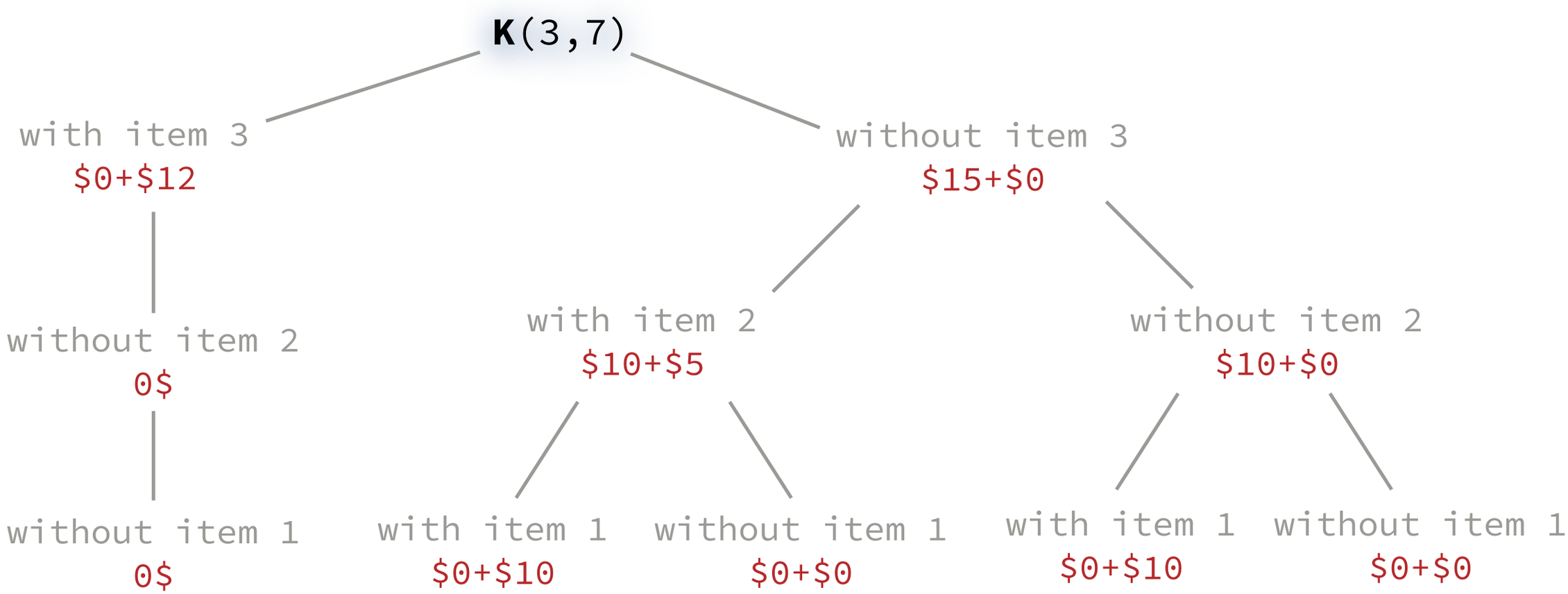
result[][]



0-1 Knapsack (Memoized Solution Tracing)

items \ capacity	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	-1	0	-1	-1	10	-1	-1	10
2	-1	0	-1	-1	-1	-1	-1	15
3	-1	-1	-1	-1	-1	-1	-1	-1

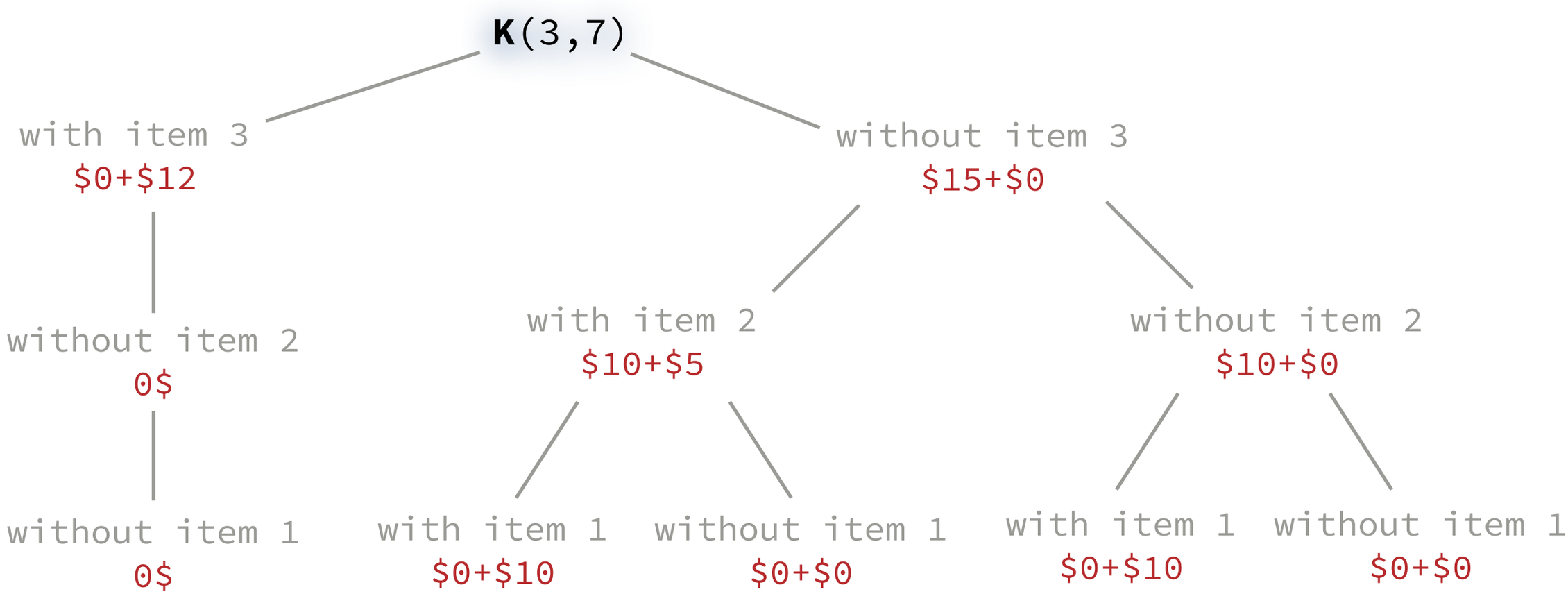
result[][]



0-1 Knapsack (Memoized Solution Tracing)

items \ capacity	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	-1	0	-1	-1	10	-1	-1	10
2	-1	0	-1	-1	-1	-1	-1	15
3	-1	-1	-1	-1	-1	-1	-1	15

result[][]



0-1 Knapsack (Bottom-up Solution)

items \ capacity	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	-1	-1	-1	-1	-1	-1	-1	-1
2	-1	-1	-1	-1	-1	-1	-1	-1
3	-1	-1	-1	-1	-1	-1	-1	-1

result[][]

0-1 Knapsack (Bottom-up Solution)

items \ capacity	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	-1	-1	-1	-1	-1	-1	-1	-1
2	-1	-1	-1	-1	-1	-1	-1	-1
3	-1	-1	-1	-1	-1	-1	-1	-1

result[][]

Note that:

- The entries needed for any cell `result[i][j]` are:
  - `result[i-1][j]` If we don't take the item. This is the cell right *above* `result[i][j]`.
  - `result[i-1][h<j]` If we take the item. This is *above* `result[i][j]` and to its *left*.
- Cells below or to the right of `result[i][j]` are never requested!

0-1 Knapsack (Bottom-up Solution)

items \ capacity	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	-1	-1	-1	-1	-1	-1	-1	-1
2	-1	-1	-1	-1	-1	-1	-1	-1
3	-1	-1	-1	-1	-1	-1	-1	-1

result[][]

Note that:

- The entries needed for any cell `result[i][j]` are:
  - `result[i-1][j]` If we don't take the item. This is the cell right *above* `result[i][j]`.
  - `result[i-1][h<j]` If we take the item. This is *above* `result[i][j]` and to its *left*.
- Cells below or to the right of `result[i][j]` are never requested!

Filling the table row-by-row or column-by-column guarantees that a cell in the table will not be `-1` when it is needed!

0-1 Knapsack (Bottom-up Solution)

KNAPSACK()

```
Initialize result[0][] to 0
```

	0	1	2	3	...	W
0	0	0	0	0	0	0
1						
2						
3						
⋮						
N						

result[][]

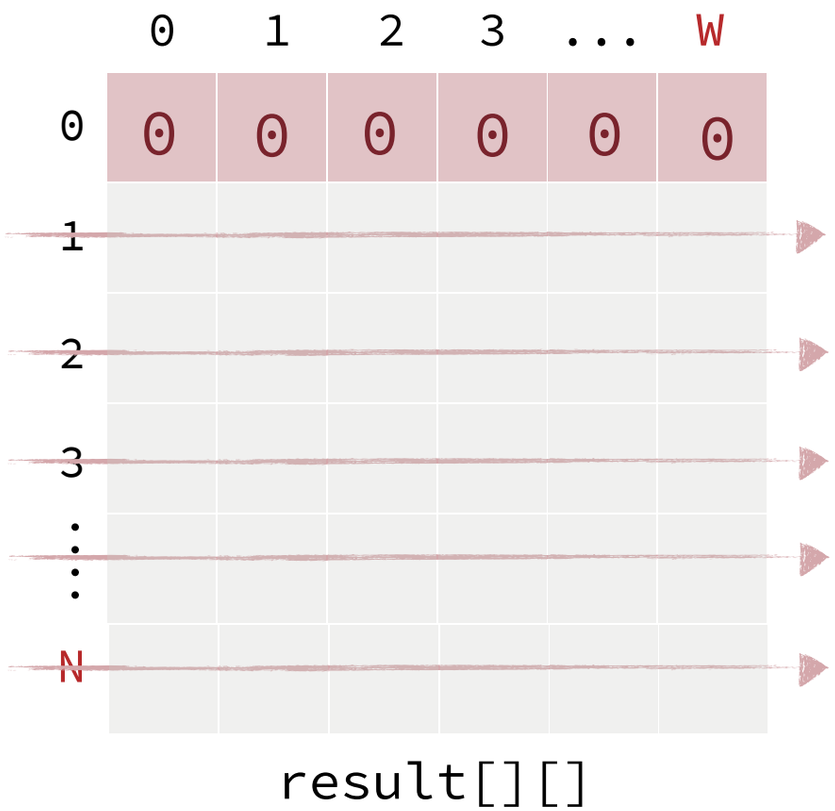
assuming
W, N, w[], v[]
and result[][]
are globally accessible

0-1 Knapsack (Bottom-up Solution)

KNAPSACK()

```
Initialize result[0][] to 0

for (i = 1 ⟶ N):
     for (j = 0 ⟶ W):
```



0-1 Knapsack (Bottom-up Solution)

KNAPSACK()

```
Initialize result[0][] to 0

for (i = 1 ⟶ N):
     for (j = 0 ⟶ W):
```

	0	1	2	3	...	W
0	0	0	0	0	0	0
1						
2						
3						
⋮						
N						

result[][]

0-1 Knapsack (Bottom-up Solution)

KNAPSACK()

```
Initialize result[0][] to 0

for (i = 1 ⟶ N):
    for (j = 0 ⟶ W):
        choice1 = 0
        if (w[i] <= j):
            choice1 = result[i-1][j-w[i]] + v[i]
```

j represents the capacity in the subproblem currently considered

	0	1	2	3	...	W
0	0	0	0	0	0	0
1						
2						
3						
⋮						
N						

result[][]

0-1 Knapsack (Bottom-up Solution)

KNAPSACK()

```
Initialize result[0][] to 0

for (i = 1 ⟶ N):
    for (j = 0 ⟶ W):
        choice1 = 0
        if (w[i] <= j):
            choice1 = result[i-1][j-w[i]] + v[i]

        choice2 = result[i-1][j]
```

	0	1	2	3	...	W
0	0	0	0	0	0	0
1						
2						
3						
⋮						
N						

result[][]

0-1 Knapsack (Bottom-up Solution)

KNAPSACK()

```
Initialize result[0][] to 0

for (i = 1 ⟶ N):
    for (j = 0 ⟶ W):
        choice1 = 0
        if (w[i] <= j):
            choice1 = result[i-1][j-w[i]] + v[i]

        choice2 = result[i-1][j]

        result[i][j] = MAX(choice1, choice2)
```

	0	1	2	3	...	W
0	0	0	0	0	0	0
1						
2						
3						
⋮						
N						

result[][]

0-1 Knapsack (Bottom-up Solution)

KNAPSACK()

```
Initialize result[0][] to 0

for (i = 1 ⟶ N):
    for (j = 0 ⟶ W):
        choice1 = 0
        if (w[i] <= j):
            choice1 = result[i-1][j-w[i]] + v[i]

        choice2 = result[i-1][j]

        result[i][j] = MAX(choice1, choice2)

return result[N][W]
```

	0	1	2	3	...	W
0	0	0	0	0	0	0
1						
2						
3						
⋮						
N						

result[][]

0-1 Knapsack (Bottom-up Solution Tracing)

items \ capacity	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1								
2								
3								

result[][]

result[1][0] = **result**[0][0]

item 1 does not fit in capacity 0

0-1 Knapsack (Bottom-up Solution Tracing)

items \ capacity	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0							
2								
3								

result[][]

result[1][1] = **result**[0][1]

item 1 does not fit in capacity 1

0-1 Knapsack (Bottom-up Solution Tracing)

items \ capacity	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	0	0				
2								
3								

result[][]

result[1][3] = **result**[0][3]

item 1 does not fit in capacity 3

0-1 Knapsack (Bottom-up Solution Tracing)

items \ capacity	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	0	0				
2								
3								

result[][]

```
result[1][4] = MAX(result[0][0]+$10, result[0][4])
```

item 1 fits in capacity 4

0-1 Knapsack (Bottom-up Solution Tracing)

items \ capacity	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	0	0	10			
2								
3								

result[][]

```
result[1][5] = MAX(result[0][1]+$10, result[0][5])
```

item 1 fits in capacity 5

0-1 Knapsack (Bottom-up Solution Tracing)

items \ capacity	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	0	0	10	10		
2								
3								

result[][]

```
result[1][6] = MAX(result[0][2]+$10, result[0][6])
```

item 1 fits in capacity 6

0-1 Knapsack (Bottom-up Solution Tracing)

items \ capacity	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	0	0	10	10	10	10
2								
3								

result[][]

```
result[1][7] = MAX(result[0][3]+$10, result[0][7])
```

item 1 fits in capacity 7

0-1 Knapsack (Bottom-up Solution Tracing)

items \ capacity	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	0	0	10	10	10	10
2	0	0	0					
3								

result[][]

items 1 and 2 do not fit in capacities 0-2

0-1 Knapsack (Bottom-up Solution Tracing)

items \ capacity	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	0	0	10	10	10	10
2	0	0	0					
3								

result[][]

```
result[2][3] = MAX(result[1][0]+$5, result[1][3])
```

item 2 fits in capacity 3

0-1 Knapsack (Bottom-up Solution Tracing)

items \ capacity	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	0	0	10	10	10	10
2	0	0	0	5				
3								

result[][]

```
result[2][3] = MAX(result[1][1]+$5, result[1][4])
```

item 2 fits in capacity 4

0-1 Knapsack (Bottom-up Solution Tracing)

items \ capacity	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	0	0	10	10	10	10
2	0	0	0	5	10	10	10	
3								

result[][]

```
result[2][7] = MAX(result[1][4]+$5, result[1][4])
```

take both item 1 and 2

take item 1 only

0-1 Knapsack (Bottom-up Solution Tracing)

items \ capacity	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	0	0	10	10	10	10
2	0	0	0	5	10	10	10	15
3								

result[][]

result[2][7] = MAX(**result**[1][4]+$5, **result**[1][4])

take both item 1 and 2

take item 1 only

0-1 Knapsack (Bottom-up Solution Tracing)

items \ capacity	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	0	0	10	10	10	10
2	0	0	0	5	10	10	10	15
3	0	0	0	5	10	10		

result[][]

```
result[3][6] = MAX(result[2][0]+$12, result[2][6])
```

item 3 fitst in capacity 6

0-1 Knapsack (Bottom-up Solution Tracing)

items \ capacity	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	0	0	10	10	10	10
2	0	0	0	5	10	10	10	15
3	0	0	0	5	10	10	12	

result[][]

```
result[3][7] = MAX(result[2][1]+$12, result[2][7])
```

take item 3 only

take items 1 and 2

0-1 Knapsack (Bottom-up Solution Tracing)

items \ capacity	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	0	0	10	10	10	10
2	0	0	0	5	10	10	10	15
3	0	0	0	5	10	10	12	15

result[][]

result[3][7] = MAX(**result**[2][1]+$12, **result**[2][7])

take item 3 only

take items 1 and 2

Running Time Quiz

How quickly does the running time of the dynamic programming solution grow when the input size grows?

A. The growth is *linear* in the input size.

B. The growth is *quadratic* in the input size.

C. The growth is *exponential* in the input size.

D. $i\hbar\frac{\partial}{\partial t}\Psi(\mathbf{r},t) = \hat{H}\Psi(\mathbf{r},t)$

Running Time Quiz

How quickly does the running time of the dynamic programming solution grow when the input size grows?

A. The growth is *linear* in the input size.

B. The growth is *quadratic* in the input size.

C. The growth is *exponential* in the input size.

D. $i\hbar\frac{\partial}{\partial t}\Psi(\mathbf{r},t) = \hat{H}\Psi(\mathbf{r},t)$

Running Time

	0	1	2	3	4	5	6	W
0	0	0	0	0	0	0	0	0
1	0	0	0	0	10	10	10	10
2	0	0	0	5	10	10	10	15
3	0	0	0	5	10	10	12	15

Bottom-up

	0	1	2	3	4	5	6	W
0	0	0	0	0	0	0	0	0
1	-1	0	-1	-1	10	-1	-1	10
2	-1	0	-1	-1	-1	-1	-1	15
N	-1	-1	-1	-1	-1	-1	-1	15

Memoized

Running Time

	0	1	2	3	4	5	6	W
0	0	0	0	0	0	0	0	0
1	0	0	0	0	10	10	10	10
2	0	0	0	5	10	10	10	15
3	0	0	0	5	10	10	12	15

Bottom-up

$\Theta(NW)$ to create and initialize the array.

	0	1	2	3	4	5	6	W
0	0	0	0	0	0	0	0	0
1	-1	0	-1	-1	10	-1	-1	10
2	-1	0	-1	-1	-1	-1	-1	15
N	-1	-1	-1	-1	-1	-1	-1	15

Memoized

$\Theta(NW)$ to create and initialize the array.

Running Time

	0	1	2	3	4	5	6	W
0	0	0	0	0	0	0	0	0
1	0	0	0	0	10	10	10	10
2	0	0	0	5	10	10	10	15
3	0	0	0	5	10	10	12	15

Bottom-up

$\Theta(NW)$ to create and initialize the array.

	0	1	2	3	4	5	6	W
0	0	0	0	0	0	0	0	0
1	-1	0	-1	-1	10	-1	-1	10
2	-1	0	-1	-1	-1	-1	-1	15
N	-1	-1	-1	-1	-1	-1	-1	15

Memoized

$\Theta(NW)$ to create and initialize the array.

Think. Can we reduce the amount of used space?

Running Time

	0	1	2	3	4	5	6	W
0	0	0	0	0	0	0	0	0
1	0	0	0	0	10	10	10	10
2	0	0	0	5	10	10	10	15
3	0	0	0	5	10	10	12	15

Bottom-up

$\Theta(NW)$ to create and initialize the array.

$\Theta(NW)$ to fill the array.

	0	1	2	3	4	5	6	W
0	0	0	0	0	0	0	0	0
1	-1	0	-1	-1	10	-1	-1	10
2	-1	0	-1	-1	-1	-1	-1	15
N	-1	-1	-1	-1	-1	-1	-1	15

Memoized

$\Theta(NW)$ to create and initialize the array.

Running Time

	0	1	2	3	4	5	6	W
0	0	0	0	0	0	0	0	0
1	0	0	0	0	10	10	10	10
2	0	0	0	5	10	10	10	15
3	0	0	0	5	10	10	12	15

Bottom-up

$\Theta(NW)$ to create and initialize the array.

$\Theta(NW)$ to fill the array.

	0	1	2	3	4	5	6	W
0	0	0	0	0	0	0	0	0
1	-1	0	-1	-1	10	-1	-1	10
2	-1	0	-1	-1	-1	-1	-1	15
N	-1	-1	-1	-1	-1	-1	-1	15

Memoized

$\Theta(NW)$ to create and initialize the array.

$O(NW)$ to fill the array. ← all items have weight = 1

$\Omega(N)$ to fill the array.

↑ all items have weights greater than the available capacity

Running Time

	0	1	2	3	4	5	6	W
0	0	0	0	0	0	0	0	0
1	0	0	0	0	10	10	10	10
2	0	0	0	5	10	10	10	15
3	0	0	0	5	10	10	12	15

Bottom-up

$\Theta(NW)$ to create and initialize the array.

$\Theta(NW)$ to fill the array.

This does not look *exponential* to me !!

	0	1	2	3	4	5	6	W
0	0	0	0	0	0	0	0	0
1	-1	0	-1	-1	10	-1	-1	10
2	-1	0	-1	-1	-1	-1	-1	15
N	-1	-1	-1	-1	-1	-1	-1	15

Memoized

$\Theta(NW)$ to create and initialize the array.

$O(NW)$ to fill the array.

$\Omega(N)$ to fill the array.

Running Time

	0	1	2	3	4	5	6	W
0	0	0	0	0	0	0	0	0
1	0	0	0	0	10	10	10	10
2	0	0	0	5	10	10	10	15
3	0	0	0	5	10	10	12	15

Bottom-up

$\Theta(NW)$ to create and initialize the array.

$\Theta(NW)$ to fill the array.

This does not look *exponential* to me !!

	0	1	2	3	4	5	6	W
0	0	0	0	0	0	0	0	0
1	-1	0	-1	-1	10	-1	-1	10
2	-1	0	-1	-1	-1	-1	-1	15
N	-1	-1	-1	-1	-1	-1	-1	15

Memoized

$\Theta(NW)$ to create and initialize the array.

$O(NW)$ to fill the array.

$\Omega(N)$ to fill the array.

Running Time

	0	1	2	3	4	5	6	W
0	0	0	0	0	0	0	0	0
1	0	0	0	0	10	10	10	10
2	0	0	0	5	10	10	10	15
3	0	0	0	5	10	10	12	15

Bottom-up

$\Theta(NW)$ to create and initialize the array.

$\Theta(NW)$ to fill the array.

This does not look *exponential* to me !!

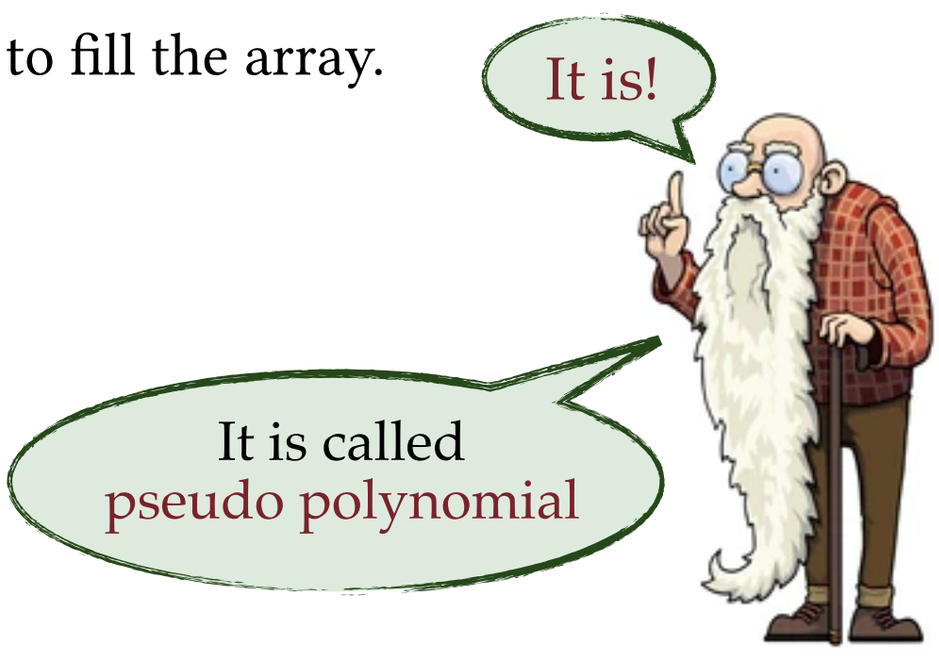
	0	1	2	3	4	5	6	W
0	0	0	0	0	0	0	0	0
1	-1	0	-1	-1	10	-1	-1	10
2	-1	0	-1	-1	-1	-1	-1	15
N	-1	-1	-1	-1	-1	-1	-1	15

Memoized

$\Theta(NW)$ to create and initialize the array.

$O(NW)$ to fill the array.

$\Omega(N)$ to fill the array.



Pseudo-Polynomial-Time Algorithms

```
SUM(a[], n)
  result = 0

  FOR (i = 0 ⟶ n-1)
      result += a[i]

  RETURN result
```

```
SUM(n)
  result = 0

  FOR (i = 0 ⟶ n-1)
      result += i

  RETURN result
```

Pseudo-Polynomial-Time Algorithms

```
SUM(a[], n)
  result = 0

  FOR (i = 0 ⟶ n-1)
      result += a[i]

  RETURN result
```

Number of addition operations = n

```
SUM(n)
  result = 0

  FOR (i = 0 ⟶ n-1)
      result += i

  RETURN result
```

Number of addition operations = n

Pseudo-Polynomial-Time Algorithms

```
SUM(a[], n)
  result = 0

  FOR (i = 0 ⟶ n-1)
      result += a[i]

  RETURN result
```

Number of addition operations = n

Input size = $\sim kn$ bits.
(assuming each number needs k bits to represent)

```
SUM(n)
  result = 0

  FOR (i = 0 ⟶ n-1)
      result += i

  RETURN result
```

Number of addition operations = n

Pseudo-Polynomial-Time Algorithms

```
SUM(a[], n)
  result = 0

  FOR (i = 0 ⟶ n-1)
      result += a[i]

  RETURN result
```

Number of addition operations = n

Input size = $\sim kn$ bits.
(assuming each number needs k bits to represent)

operations = Θ(input size)

If we double the input size, the running time doubles!

Adding 1 element to the array, adds 1 addition operation to the loop

```
SUM(n)
  result = 0

  FOR (i = 0 ⟶ n-1)
      result += i

  RETURN result
```

Number of addition operations = n

Pseudo-Polynomial-Time Algorithms

```
SUM(a[], n)
  result = 0

  FOR (i = 0 ⟶ n-1)
      result += a[i]

  RETURN result
```

Number of addition operations = n

Input size = $\sim kn$ bits.
(assuming each number needs k bits to represent)

operations = Θ(input size)

If we double the input size, the running time doubles!

Adding 1 element to the array, adds 1 addition operation to the loop

```
SUM(n)
  result = 0

  FOR (i = 0 ⟶ n-1)
      result += i

  RETURN result
```

Number of addition operations = n

Input size = $\sim \log_2(n)$ bits
(representing an integer requires $\lfloor \log_2(n) \rfloor + 1$ bits)

Pseudo-Polynomial-Time Algorithms

```
SUM(a[], n)
  result = 0

  FOR (i = 0 ⟶ n-1)
      result += a[i]

  RETURN result
```

Number of addition operations = n

Input size = $\sim kn$ bits.
(assuming each number needs k bits to represent)

operations = Θ(input size)

If we double the input size, the running time doubles!

Adding 1 element to the array, adds 1 addition operation to the loop

```
SUM(n)
  result = 0

  FOR (i = 0 ⟶ n-1)
      result += i

  RETURN result
```

Number of addition operations = n

Input size = $\sim \log_2(n)$ bits
(representing an integer requires $\lfloor \log_2(n) \rfloor + 1$ bits)

operations = $\Theta(2^{\log n}) = \Theta(2^{\text{input size}})$

Pseudo-Polynomial-Time Algorithms

```
SUM(a[], n)
  result = 0

  FOR (i = 0 ⟶ n-1)
      result += a[i]

  RETURN result
```

Number of addition operations = n

Input size = $\sim kn$ bits.
(assuming each number needs k bits to represent)

operations = Θ(input size)

If we double the input size, the running time doubles!

Adding 1 element to the array, adds 1 addition operation to the loop

```
SUM(n)
  result = 0

  FOR (i = 0 ⟶ n-1)
      result += i

  RETURN result
```

Number of addition operations = n

Input size = $\sim \log_2(n)$ bits
(representing an integer requires $\lfloor \log_2(n) \rfloor + 1$ bits)

operations = $\Theta(2^{\log n}) = \Theta(2^{\text{input size}})$

If we increase the input size by 1 bit:

- The running time doubles!
  $2^{\log n+1} = 2 \times 2^{\log n}$
- `10000` in binary = `16` in decimal
  `100000` in binary = `32` in decimal

Pseudo-Polynomial-Time Algorithms

Polynomial Time Algorithm

The running time is polynomial in the *size* of the input.

Pseudo-Polynomial Time Algorithm

The running time is polynomial in the *value* of the input.

Pseudo-Polynomial-Time Algorithms

Polynomial Time Algorithm
The running time is polynomial in the *size* of the input.

Pseudo-Polynomial Time Algorithm
The running time is polynomial in the *value* of the input.

Pseudo-Polynomial Time Quiz

Which of the following has pseudo-polynomial time?

A. Searching an array using Linear Search.

B. Dynamic Programming solution for the Collecting Apples problem.

C. Dynamic Programming solution for Fibonacci.

D. Checking if a number n is prime (by testing every number $i < n$ to see if n divides i).

Pseudo-Polynomial-Time Algorithms

Polynomial Time Algorithm
The running time is polynomial in the *size* of the input.

Pseudo-Polynomial Time Algorithm
The running time is polynomial in the *value* of the input.

Pseudo-Polynomial Time Quiz

Which of the following has pseudo-polynomial time?

A. Searching an array using Linear Search.

B. Dynamic Programming solution for the Collecting Apples problem.

C. Dynamic Programming Solution for Fibonacci.

D. Checking if a number n is prime (by testing every number $i < n$ to see if n divides i).

Pseudo-Polynomial-Time Algorithms

0-1 Knapsack.

Given N and W, the input size is:

- kN bits to store the weights and values of the items.
- $\sim \log_2 W$ bits for the capacity.

The algorithm runs in $\Theta(NW) = \Theta(N2^{\log_2 W})$

Pseudo-Polynomial-Time Algorithms

0-1 Knapsack.

Given N and W, the input size is:

- kN bits to store the weights and values of the items.
- $\sim \log_2 W$ bits for the capacity.

The algorithm runs in $\Theta(NW) = \Theta(N2^{\log_2 W})$

Doubling N:

- $2kN$ bits are needed for the weights and values.
- Doubles the running time.

$N = 32$
(size = $32 \times k$ bits)

$N = 64$
(size = $64 \times k$ bits)

Pseudo-Polynomial-Time Algorithms

0-1 Knapsack.

Given N and W, the input size is:

- kN bits to store the weights and values of the items.
- $\sim \log_2 W$ bits for the capacity.

The algorithm runs in $\Theta(NW) = \Theta(N2^{\log_2 W})$

Doubling N:

- $2kN$ bits are needed for the weights and values.
- Doubles the running time.

$N = 32$
(size = $32 \times k$ bits)

$N = 64$
(size = $64 \times k$ bits)

Doubling W:

- Increases the input size by 1 bit.
- Doubles the running time!

$W = 32$ (size = 100000 in binary: 6 bits)

$W = 64$ (size = 1000000 in binary: 7 bits)

Which items to pick?

	0	1	2	3	4	5	6	7 = W
0	0	0	0	0	0	0	0	0
1		0			10			10
2		0						15
N=3								15

Memoized

Which items to pick?

	0	1	2	3	4	5	6	7 = W
0	0	0	0	0	0	0	0	0
1		0			10			10
2		0						15
N=3								15

Memoized

since `result[3][7]` is the
same as `result[2][7]`,
we know that item **3** was **not** taken.

capacity remains the same

Which items to pick?

	0	1	2	3	4	5	6	7 = W
0	0	0	0	0	0	0	0	0
1		0			10			10
2		0						15
N=3								15

Memoized

since `result[2][7]` is **not** the same as `result[1][7]`, we know that item **2** was taken.

next capacity 7 − 3 = 4

Which items to pick?

	0	1	2	3	4	5	6	7 = W
0	0	0	0	0	0	0	0	0
1		0			10			10
2		0						15
N=3								15

Memoized

since `result[1][4]` is **not** the same as `result[0][4]`, we know that item **1** was taken.

next capacity `4 - 4 = 0`

Which items to pick?

	0	1	2	3	4	5	6	7 = W
0	0	0	0	0	0	0	0	0
1		0			10			10
2		0						15
N=3								15

Memoized

No more work to be done!

Which items to pick?

	0	1	2	3	4	5	6	W
0	0	0	0	0	0	0	0	0
1	-1	0	-1	-1	10	-1	-1	10
2	-1	0	-1	-1	-1	-1	-1	15
N	-1	-1	-1	-1	-1	-1	-1	15

Memoized

```
FIND_ITEMS(result[][])

i = N, j = W

while (i > 0 and j > 0):
```

assuming
`W`, `N`, `w[]` *and* `v[]`
are globally accessible

Which items to pick?

	0	1	2	3	4	5	6	W
0	0	0	0	0	0	0	0	0
1	-1	0	-1	-1	10	-1	-1	10
2	-1	0	-1	-1	-1	-1	-1	15
N	-1	-1	-1	-1	-1	-1	-1	15

Memoized

```
FIND_ITEMS(result[][])

i = N, j = W

while (i > 0 and j > 0):
    if (result[i][j] > result[i-1][j]):
        pick Item i
```

assuming
`W`, `N`, `w[]` *and* `v[]`
are globally accessible

Which items to pick?

	0	1	2	3	4	5	6	W
0	0	0	0	0	0	0	0	0
1	-1	0	-1	-1	10	-1	-1	10
2	-1	0	-1	-1	-1	-1	-1	15
N	-1	-1	-1	-1	-1	-1	-1	15

Memoized

```
FIND_ITEMS(result[][])

i = N, j = W

while (i > 0 and j > 0):
    if (result[i][j] > result[i-1][j]):
        pick Item i
        j -= w[i]
```

assuming
`W`, `N`, `w[]` *and* `v[]`
are globally accessible

Which items to pick?

	0	1	2	3	4	5	6	W
0	0	0	0	0	0	0	0	0
1	-1	0	-1	-1	10	-1	-1	10
2	-1	0	-1	-1	-1	-1	-1	15
N	-1	-1	-1	-1	-1	-1	-1	15

Memoized

```
FIND_ITEMS(result[][])

i = N, j = W

while (i > 0 and j > 0):
    if (result[i][j] > result[i-1][j]):
        pick Item i
        j -= w[i]
    i -= 1
```

assuming
`W`, `N`, `w[]` *and* `v[]`
are globally accessible

Which items to pick?

	0	1	2	3	4	5	6	W
0	0	0	0	0	0	0	0	0
1	-1	0	-1	-1	10	-1	-1	10
2	-1	0	-1	-1	-1	-1	-1	15
N	-1	-1	-1	-1	-1	-1	-1	15

Memoized

```
FIND_ITEMS(result[][])

i = N, j = W

while (i > 0 and j > 0):

    if (result[i][j] > result[i-1][j]):
        pick Item i
        j -= w[i]

    i -= 1
```

Running Time. $O(N)$