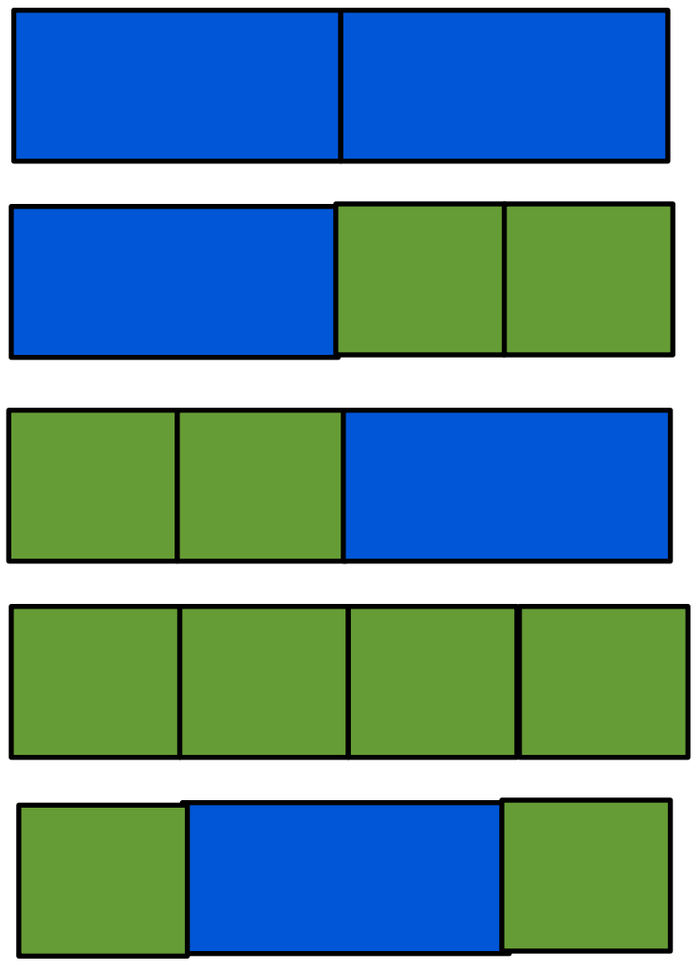
Prepared by Julian García

Lecture 25
Dynamic programming

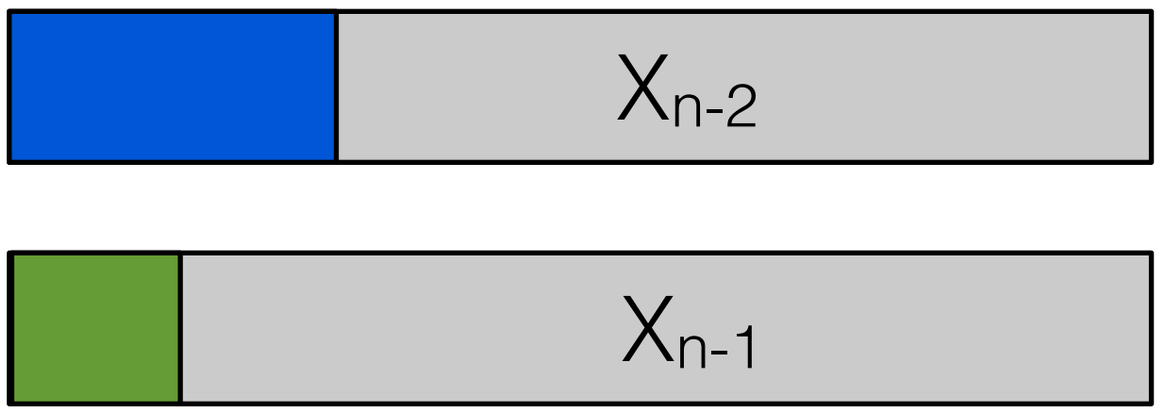
FIT 1008
Introduction to Computer Science

Dynamic Programming

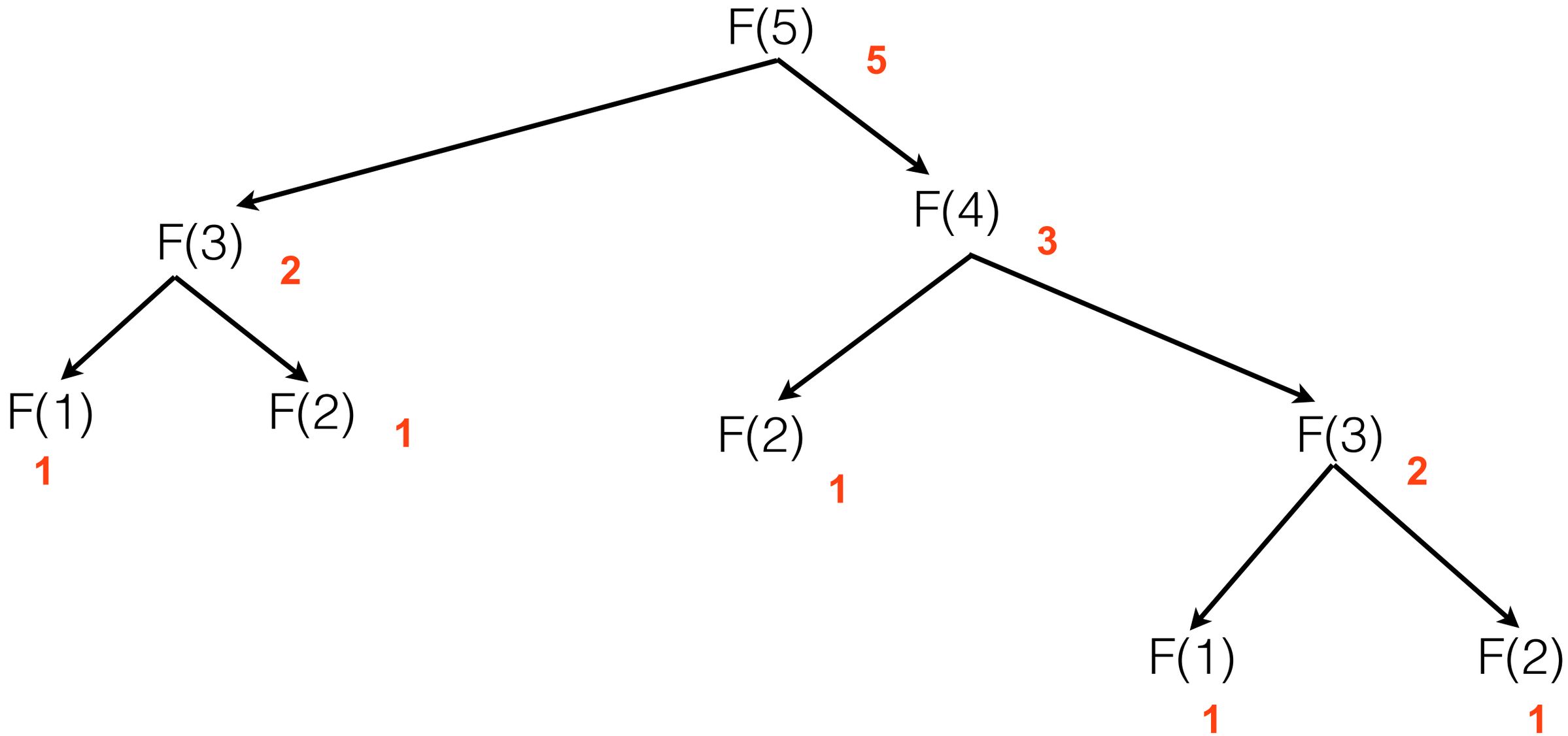
Knapsack

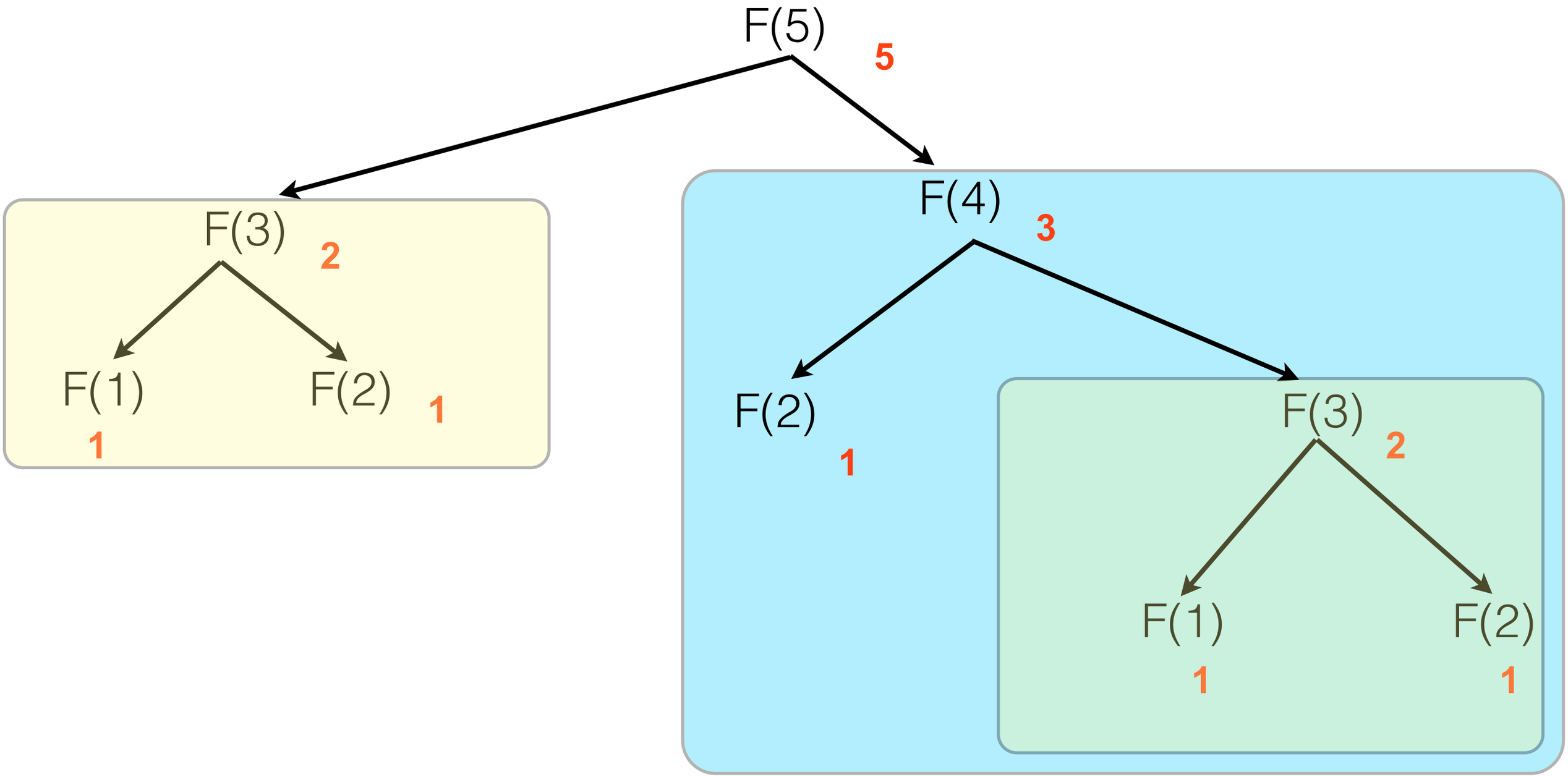


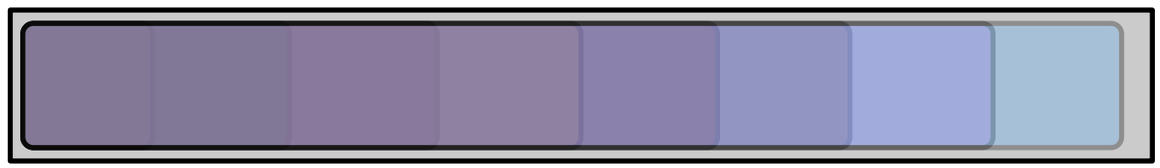
Number of possible tilings = X_n



$$X_n = X_{n-1} + X_{n-2}$$

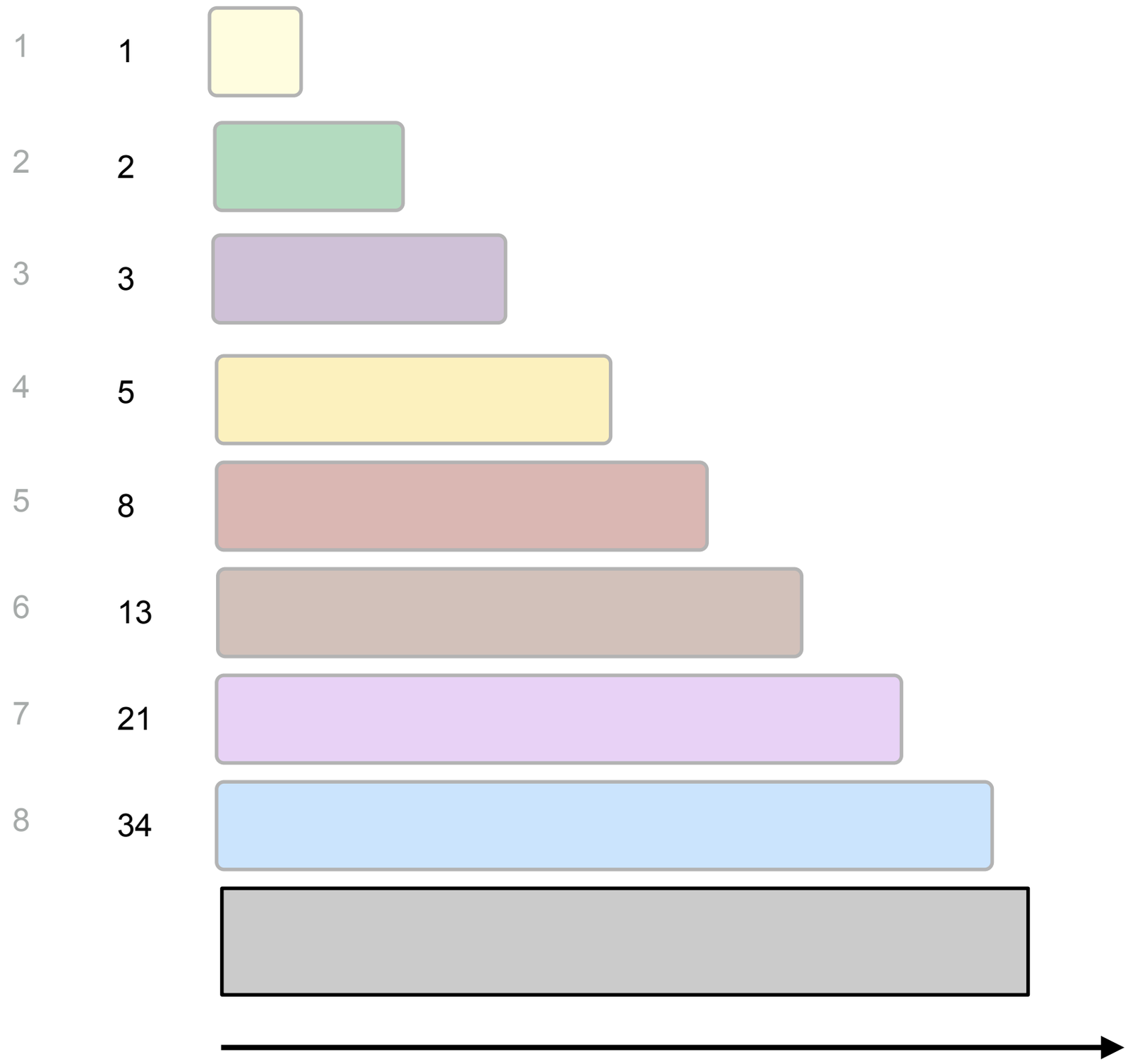

F(5)
5
F(3)
2
F(4)
3
F(1)
1
F(2)
1
F(2)
1
F(3)
2
F(1)
1
F(2)
1


F(5)
5
F(3)
2
F(1)
1
F(2)
1
F(4)
3
F(2)
1
F(3)
2
F(1)
1
F(2)
1



Overlapping subproblems

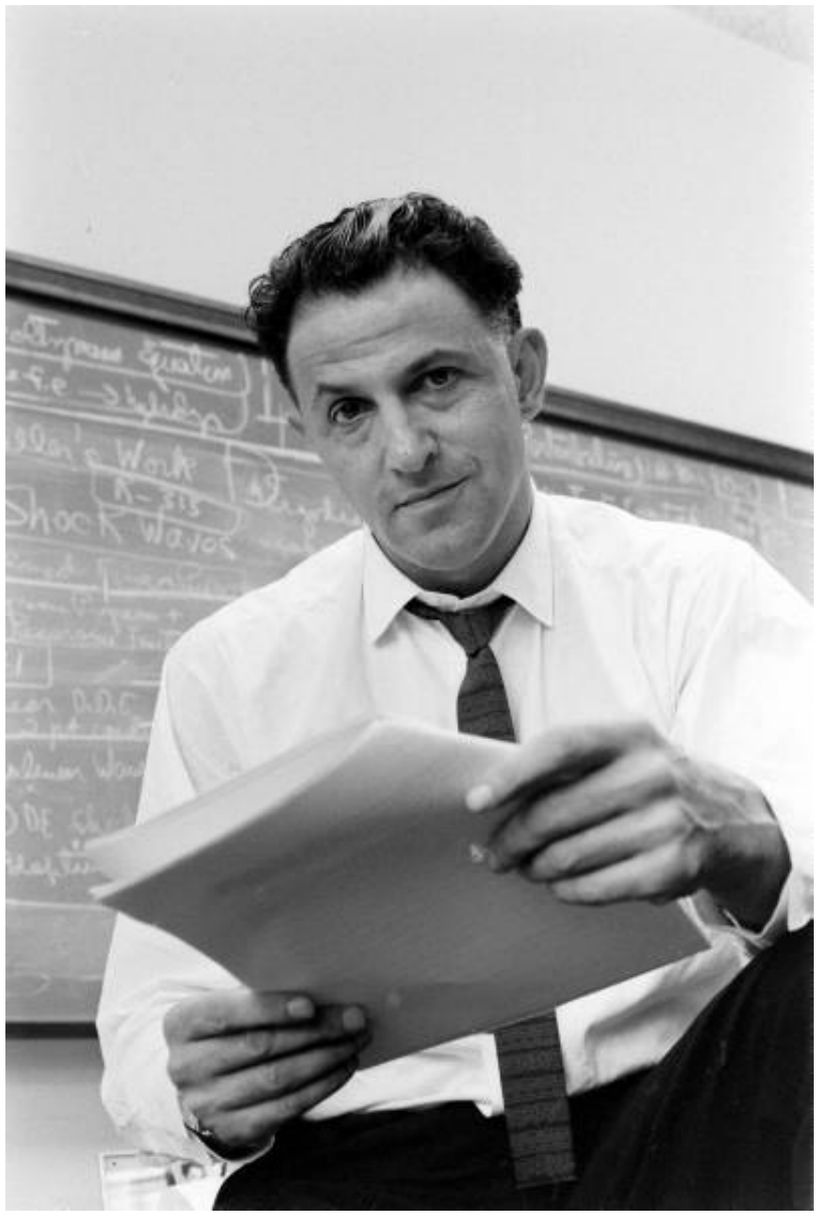
Optimal substructure


1 1
2 2
3 3
4 5
5 8
6 13
7 21
8 34

Dynamic Programming

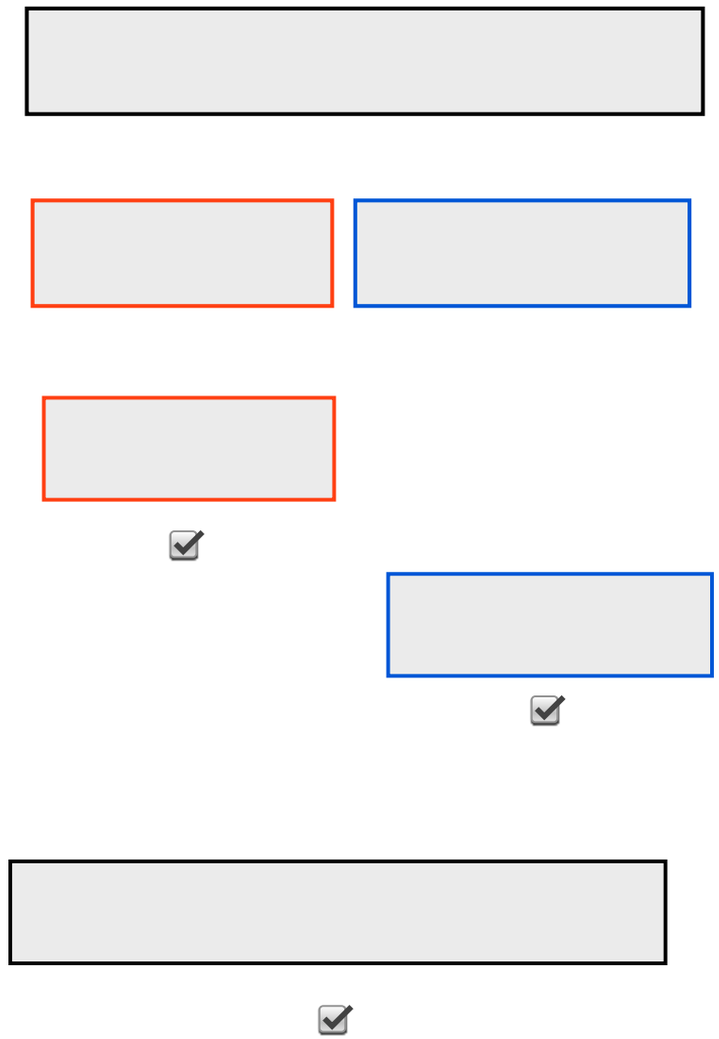
- Solves problems by **solving subproblems**.
- Each subproblem is solved only once, storing relevant information.
- Subproblems may **overlap**.

Richard Bellman

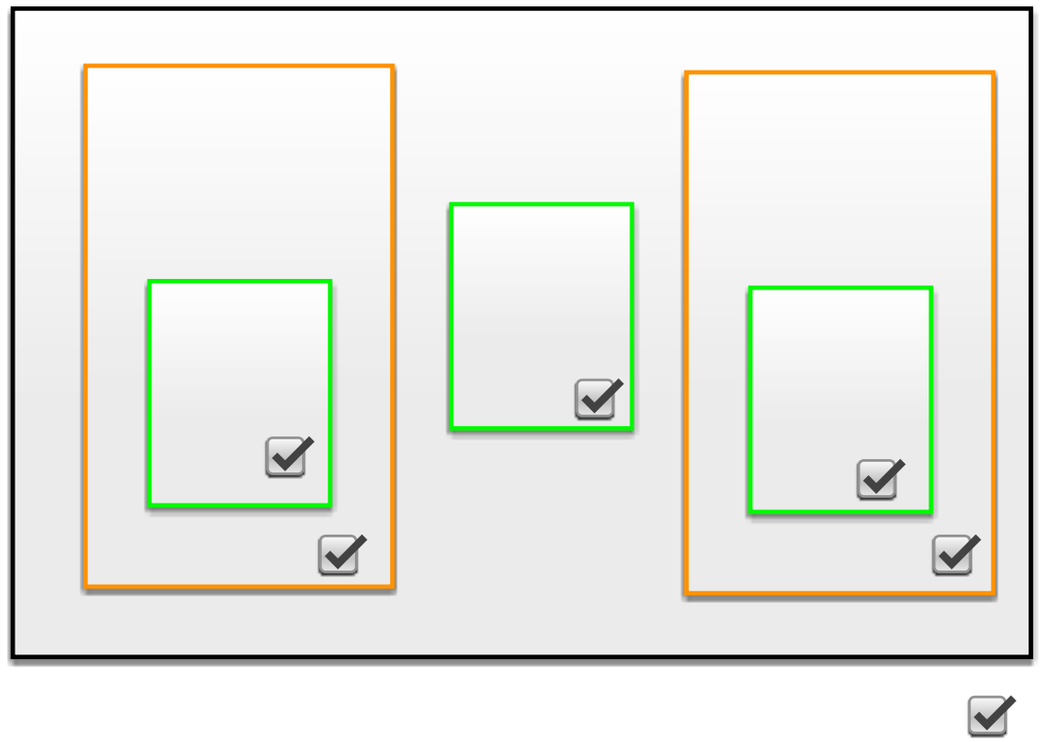


source: http://lifesun.info/

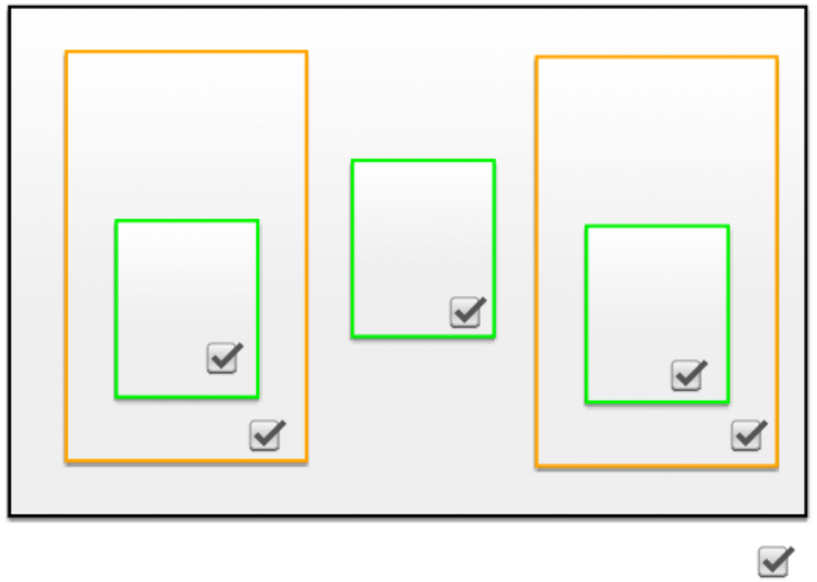
Solves problems by **solving subproblems**.



Divide and conquer



Dynamic programming



Subproblems may **overlap**.

20 kg max

Knapsack

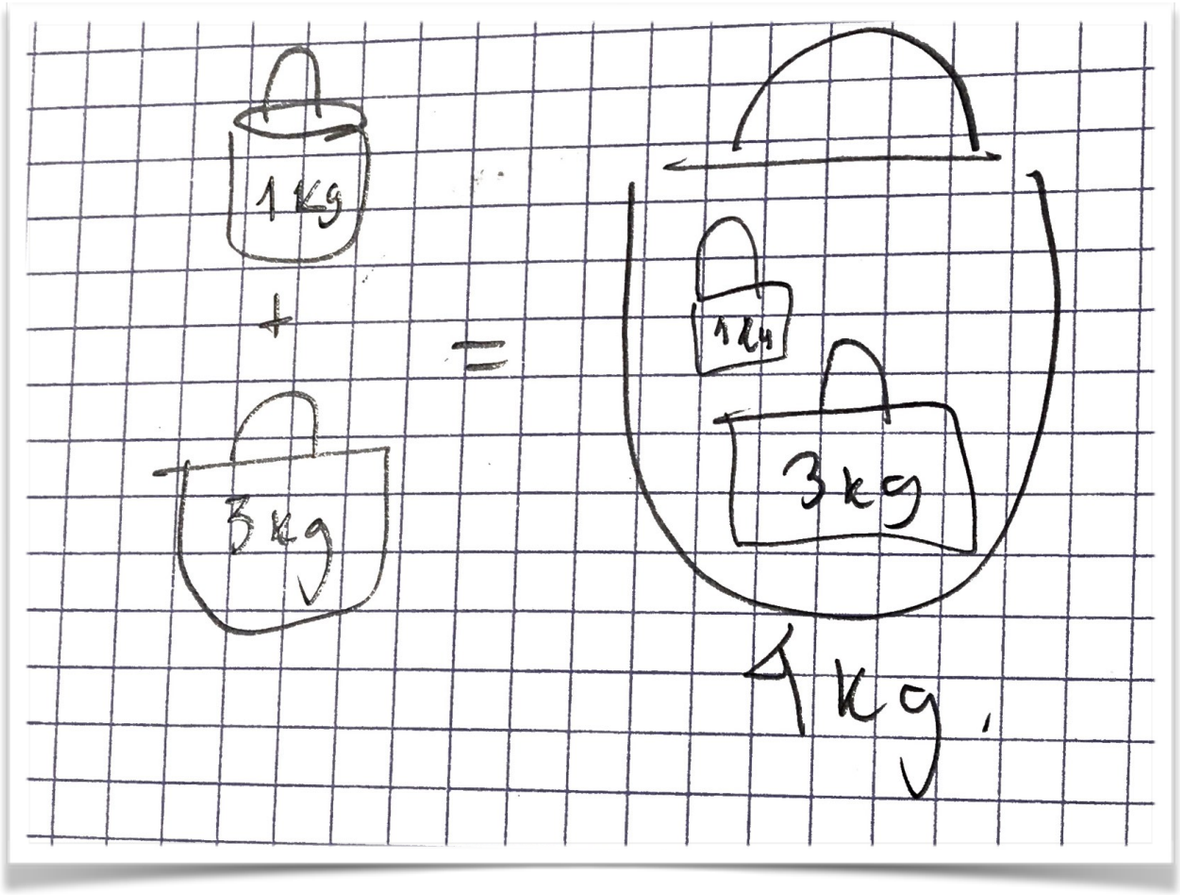
Knapsack

Suppose you are in a treasure cave which contains 6 precious items, with the following weights and monetary value.

Item	**1**	**2**	**3**	**4**	**5**	**6**
Weight	20kg	10kg	9kg	4kg	2kg	1kg
Value	$4000	$3500	$1800	$400	$1000	$200

You want to take as much treasure as you can carry. However, you can only carry up to 20kg. Which items do you take?

What is the maximum value you can take?



Overlapping subproblems

Optimal substructure

Subproblems

Item	1	2	3	4	5	6
Weight	20kg	10kg	9kg	4kg	2kg	1kg
Value	$4000	$3500	$1800	$400	$1000	$200

MV[i, j]

maximum value in a problem with the first **i** items and capacity **j**

$0 \leq i \leq 6$

$0 \leq j \leq 20$

assume: weights and capacities are integers.

Item	1	2	3	4
Weight	20kg	10kg	9kg	4kg
Value	$4000	$3500	$1800	$400

Item	1	2
Weight	20kg	10kg
Value	$4000	$3500

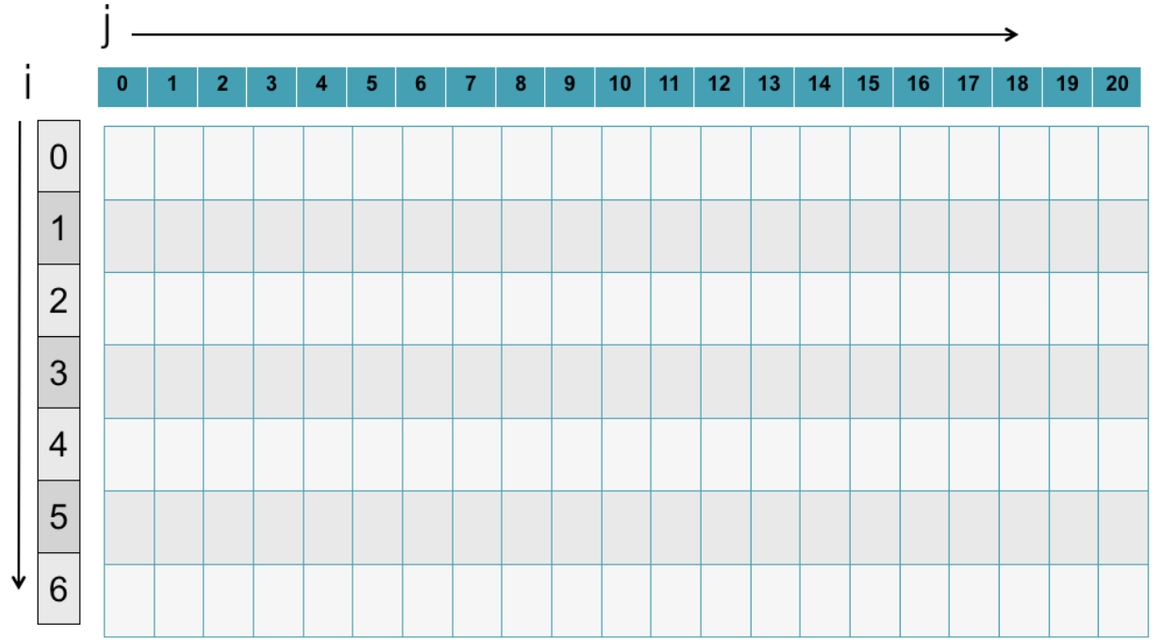
Fewer Items

Less capacity

Item	1	2
Weight	20kg	10kg
Value	$4000	$3500

Item	1	2	3	4
Weight	20kg	10kg	9kg	4kg
Value	$4000	$3500	$1800	$400

Item	1	2	3	4	5	6
Weight	20kg	10kg	9kg	4kg	2kg	1kg
Value	$4000	$3500	$1800	$400	$1000	$200



j

i

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

0
1
2
3
4
5
6

M[c, n]

Find how subproblems are related

$$M[0, j] = 0$$

No items, no value no matter how large the knapsack

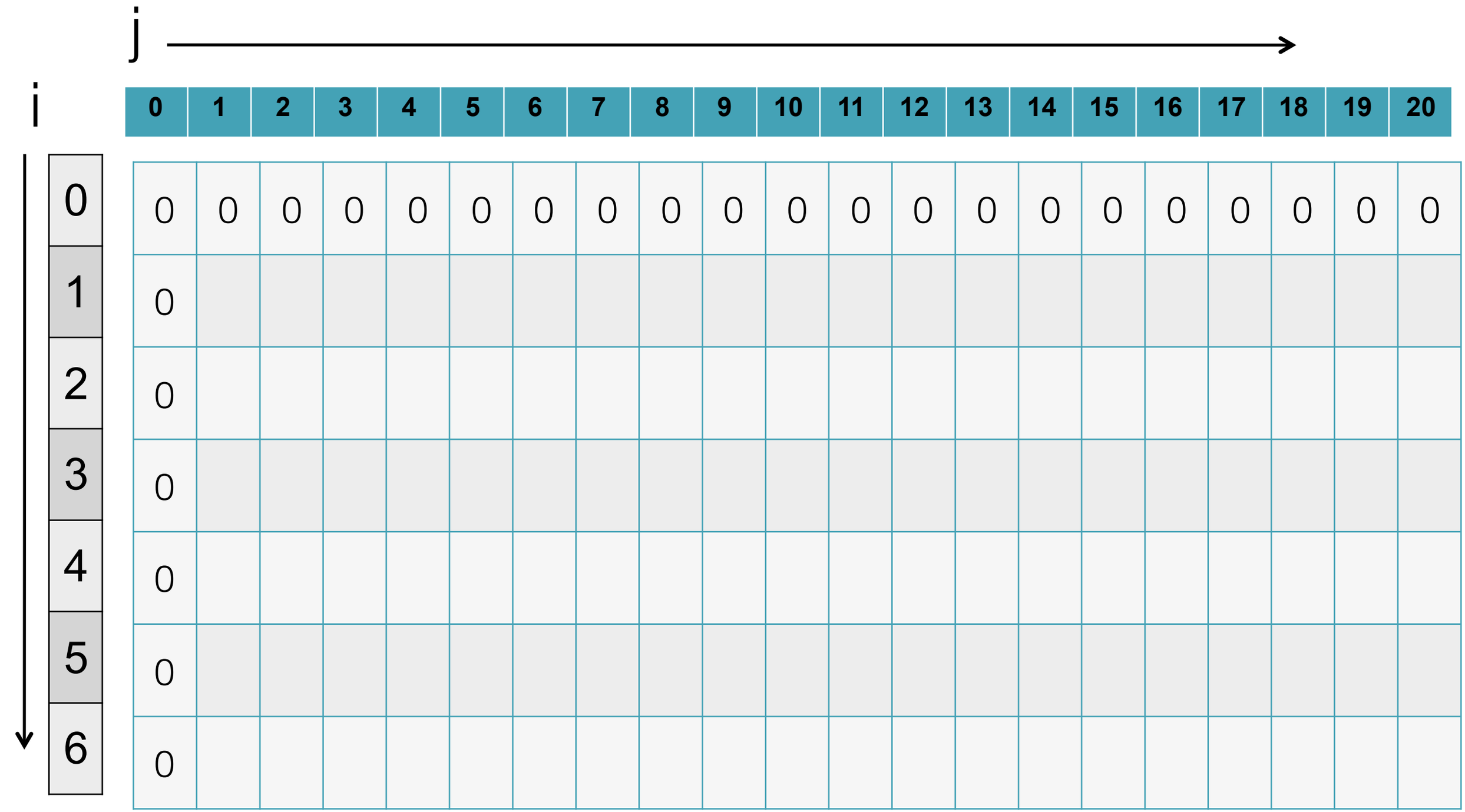
$$M[i, 0] = 0$$

No knapsack, no value no matter how many items

Base case

MV[0, j] = 0

MV[i, 0] = 0



Item	1	2	3	4	5	6
Weight	20kg	10kg	9kg	4kg	2kg	1kg
Value	$4000	$3500	$1800	$400	$1000	$200

j →

i ↓

i \ j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0																				
2	0																				
3	0																				
4	0																				
5	0																				
6	0																				

Item	1	2	3	4	5	6
Weight	20kg	10kg	9kg	4kg	2kg	1kg
Value	$4000	$3500	$1800	$400	$1000	$200

j →

i ↓

i \ j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0																				
2	0																				
3	0																				
4	0																				
5	0																				
6	0																				

Item	1	2	3	4	5	6
Weight	20kg	10kg	9kg	4kg	2kg	1kg
Value	$4000	$3500	$1800	$400	$1000	$200

j →

i ↓

i \ j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
2	0																				
3	0																				
4	0																				
5	0																				
6	0																				

Item	1	2	3	4	5	6
Weight	20kg	10kg	9kg	4kg	2kg	1kg
Value	$4000	$3500	$1800	$400	$1000	$200

j →

i ↓

i \ j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
2	0																				
3	0																				
4	0																				
5	0																				
6	0																				

Item	1	2	3	4	5	6
Weight	20kg	10kg	9kg	4kg	2kg	1kg
Value	$4000	$3500	$1800	$400	$1000	$200

j →

i ↓

i \ j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4000
2	0																				
3	0																				
4	0																				
5	0																				
6	0																				

Item	1	2	3	4	5	6
Weight	20kg	10kg	9kg	4kg	2kg	1kg
Value	$4000	$3500	$1800	$400	$1000	$200

j →

i ↓

i \ j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4000
2	0																				
3	0																				
4	0																				
5	0																				
6	0																				

Item	1	2	3	4	5	6
Weight	20kg	10kg	9kg	4kg	2kg	1kg
Value	$4000	$3500	$1800	$400	$1000	$200

j →

i ↓

i \ j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4000
2	0																				
3	0																				
4	0																				
5	0																				
6	0																				

Adding item 2 as a possibility makes no difference when j < 10

Item	1	2	3	4	5	6
Weight	20kg	10kg	9kg	4kg	2kg	1kg
Value	$4000	$3500	$1800	$400	$1000	$200

j →

i ↓

i \ j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4000
2	0	0	0	0	0	0	0	0	0	0											
3	0																				
4	0																				
5	0																				
6	0																				

Adding item 2 as a possibility makes no difference when j < 10

Item	1	2	3	4	5	6
Weight	20kg	10kg	9kg	4kg	2kg	1kg
Value	$4000	$3500	$1800	$400	$1000	$200

j →

i ↓

i \ j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4000
2	0	0	0	0	0	0	0	0	0	0											
3	0																				
4	0																				
5	0																				
6	0																				

Do not put item 2 in the knapsack: 0

Put item 2 in the knapsack: 3500 + 0

Item	1	2	3	4	5	6
Weight	20kg	10kg	9kg	4kg	2kg	1kg
Value	$4000	$3500	$1800	$400	$1000	$200

j →

i ↓

i \ j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4000
2	0	0	0	0	0	0	0	0	0	0	3500										
3	0																				
4	0																				
5	0																				
6	0																				

Do not put item 2 in the knapsack: 0

Put item 2 in the knapsack: 3500 + 0

Item	1	2	3	4	5	6
Weight	20kg	10kg	9kg	4kg	2kg	1kg
Value	$4000	$3500	$1800	$400	$1000	$200

j →

i ↓

i \ j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4000
2	0	0	0	0	0	0	0	0	0	0	3500										
3	0																				
4	0																				
5	0																				
6	0																				

Do not put item 2 in the knapsack: 0

Put item 2 in the knapsack: 3500 + 0

Item	1	2	3	4	5	6
Weight	20kg	10kg	9kg	4kg	2kg	1kg
Value	$4000	$3500	$1800	$400	$1000	$200

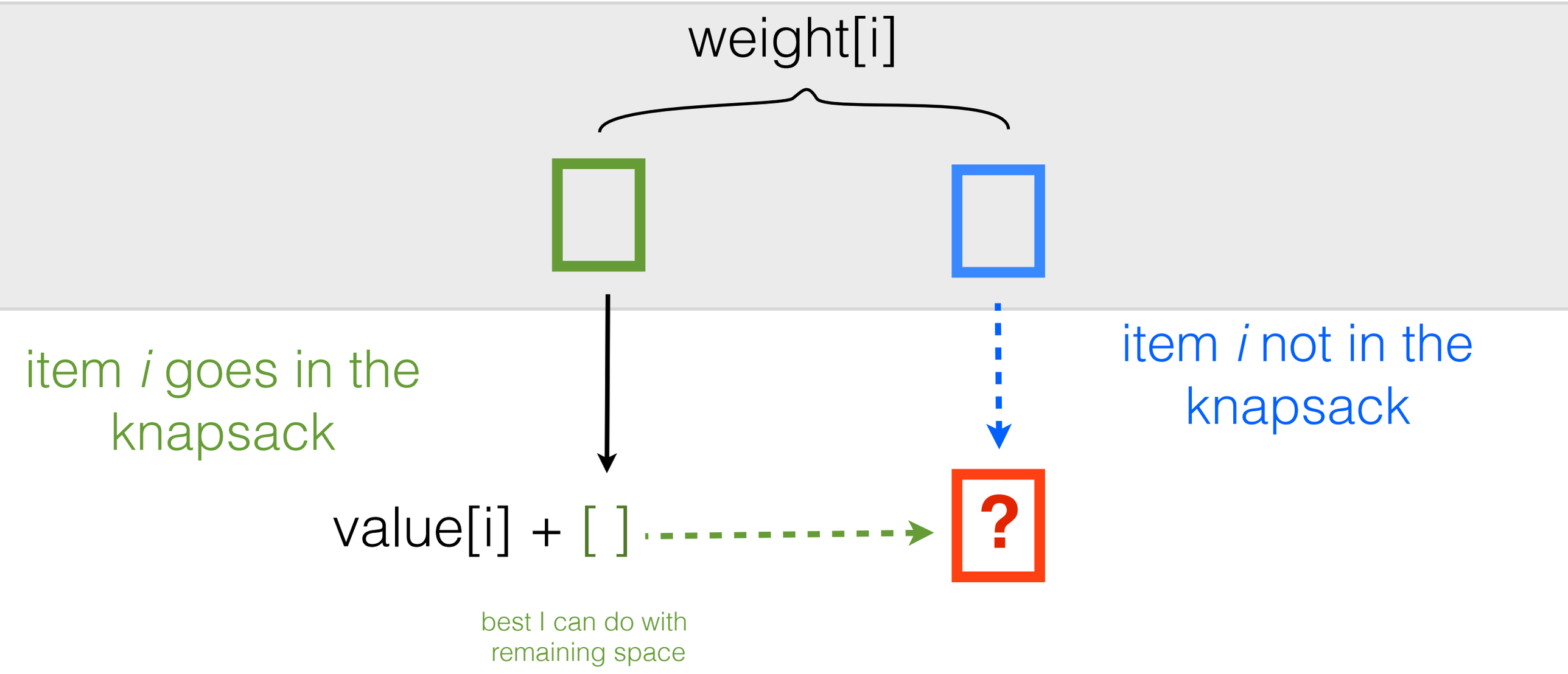
j →

i ↓

i \ j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4000
2	0	0	0	0	0	0	0	0	0	0	3500	3500									
3	0																				
4	0																				
5	0																				
6	0																				

Do not put item 2 in the knapsack: 0

Put item 2 in the knapsack: 3500 + 0


weight[i]
item *i* goes in the knapsack
item *i* not in the knapsack
value[i] + [ ]
?
best I can do with remaining space

Item	1	2	3	4	5	6
Weight	20kg	10kg	9kg	4kg	2kg	1kg
Value	$4000	$3500	$1800	$400	$1000	$200

j →

i ↓

i \ j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4000
2	0	0	0	0	0	0	0	0	0	0	3500	3500	3500	3500	3500	3500	3500	3500	3500	3500	4000
3	0	0	0	0	0	0	0	0	0	1800	3500	3500	3500	3500	3500	3500	3500	3500	3500	5300	5300
4	0	0	0	0	400	400	400	400	400	1800	3500	3500	3500	3500	3900	3900	3900	3900	3900	5300	5300
5	0	0	1000	1000	1000	1000	1400	1400	1400	1800	3500	3500	4500	4500	4500	4500	4900	4900	4900	5300	5300
6	0	200	1000	1200	1200	1200	1400	1600	1600	1800	3500	3700	4500	4700	4700	4700	4900	5100	5100	5300	**5500**

Knapsack

M[0, j] = 0

M[i, 0] = 0

base case

Item *i* does not fit

M(i, j) =

M[i-1, j] **if** i.weight > j

Max { M[i-1, j] **,** i.value + M[i-1, j - i.weight] **}**

otherwise

items

capacity

item *i* not in the knapsack

item *i* goes in the knapsack

j

i

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0																				
1																				
2																				
3																				
4																				
5																				
6																				

Item	1	2
Weight	20kg	10kg
Value	$4000	$3500

10 Kg

j

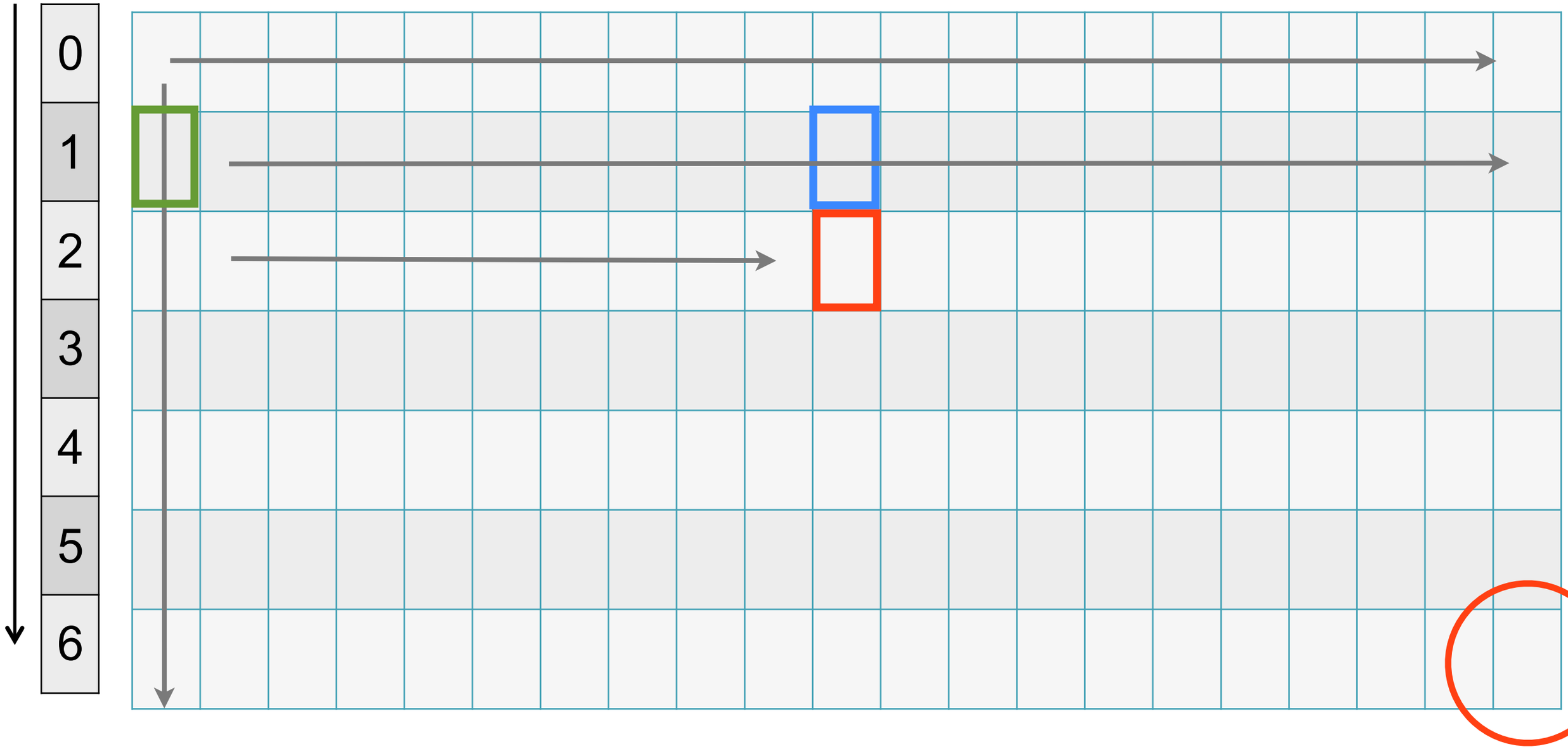
i

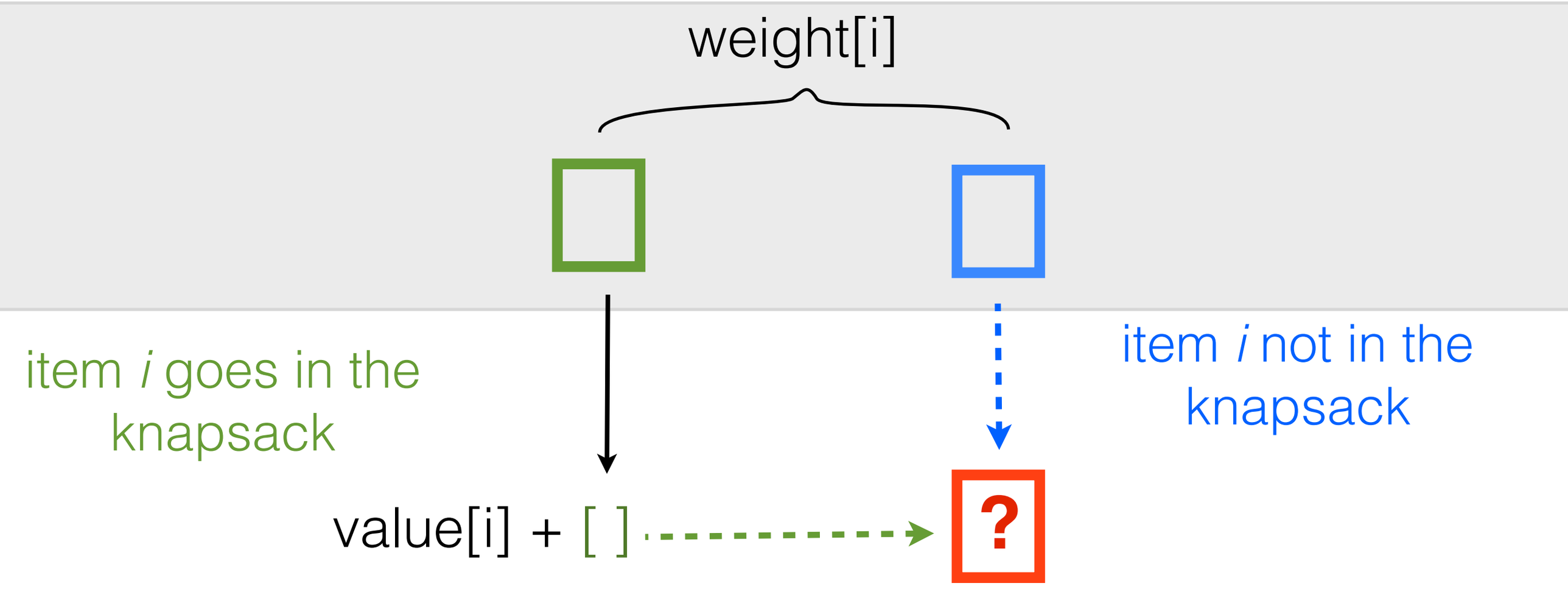
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0																					
1																					
2																					
3																					
4																					
5																					
6																					

j

i

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20




weight[i]
item *i* goes in the knapsack
item *i* not in the knapsack
value[i] + [ ]
?

MV[0, j] = 0

MV[i, 0] = 0

base case

$$MV(i, j) = \begin{cases} MV(i-1, j) & \textbf{if } weights[i] > j \\ \textbf{Max } \{ MV(i-1, j), values[i] + MV(i-1, j - weights[i]) \} \end{cases}$$

items **capacity**

Item	1	2	3	4	5	6
Weight	20kg	10kg	9kg	4kg	2kg	1kg
Value	$4000	$3500	$1800	$400	$1000	$200

j →

i ↓

i \ j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4000
2	0	0	0	0	0	0	0	0	0	0	3500	3500	3500	3500	3500	3500	3500	3500	3500	3500	4000
3	0	0	0	0	0	0	0	0	0	1800	3500	3500	3500	3500	3500	3500	3500	3500	3500	5300	5300
4	0	0	0	0	400	400	400	400	400	1800	3500	3500	3500	3500	3900	3900	3900	3900	3900	5300	5300
5	0	0	1000	1000	1000	1000	1400	1400	1400	1800	3500	3500	4500	4500	4500	4500	4900	4900	4900	5300	5300
6	0	200	1000	1200	1200	1200	1400	1600	1600	1800	3500	3700	4500	4700	4700	4700	4900	5100	5100	5300	**5500**

Knapsack

M[0, j] = 0

M[i, 0] = 0

base case

Item *i* does not fit

M(i, j) =

M[i-1, j] **if** i.weight > j

Max { M[i-1, j] **,** i.value + M[i-1, j - i.weight] **}**

otherwise

items

capacity

item *i* not in the knapsack

item *i* goes in the knapsack

Knapsack

M[0, j] = 0

M[i, 0] = 0

→ **base case**

M(i, j) =

M[i-1, j] **if** i.weight > j

Max { M[i-1, j] **,** i.value + M[i-1, j - i.weight] **}**

otherwise

items

capacity

item *i* not in the knapsack

item *i* goes in the knapsack

A matrix object using **numpy**

```
import numpy as np
```

```
np.zeros((4, 6))
```

```
array([[ 0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.]])
```

```
my_table = np.zeros((4, 6))
my_table[2, 3] = -1
my_table
```

```
array([[ 0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0., -1.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.]])
```

enumerate

```
a_list_of_things = ['A', 'B', 'C', 'D', 'E']
```

```
for i, thing in enumerate(a_list_of_things):
    print(i, end=", ")
    print(thing)
```

```
0, A
1, B
2, C
3, D
4, E
```

```
for i, thing in enumerate(a_list_of_things, start=1):
    print(i, end=", ")
    print(thing)
```

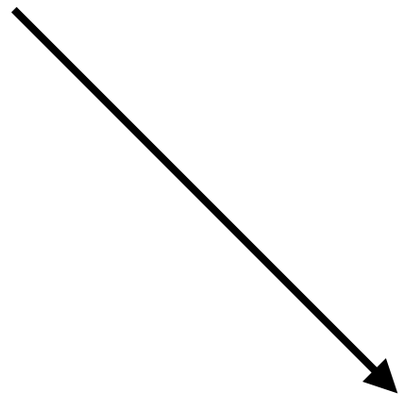
```
1, A
2, B
3, C
4, D
5, E
```

```python
class Item:

    def __init__(self, value=0, weight=0):
        assert type(weight) == int
        self.value = value
        self.weight = weight

    def __str__(self):
        # str is meant to be readable, possibly ambiguous
        return "(v={}: w={})".format(self.value, self.weight)

def __repr__(self):
    # __repr__ is supposed to be unambiguous
    return str(self)
```

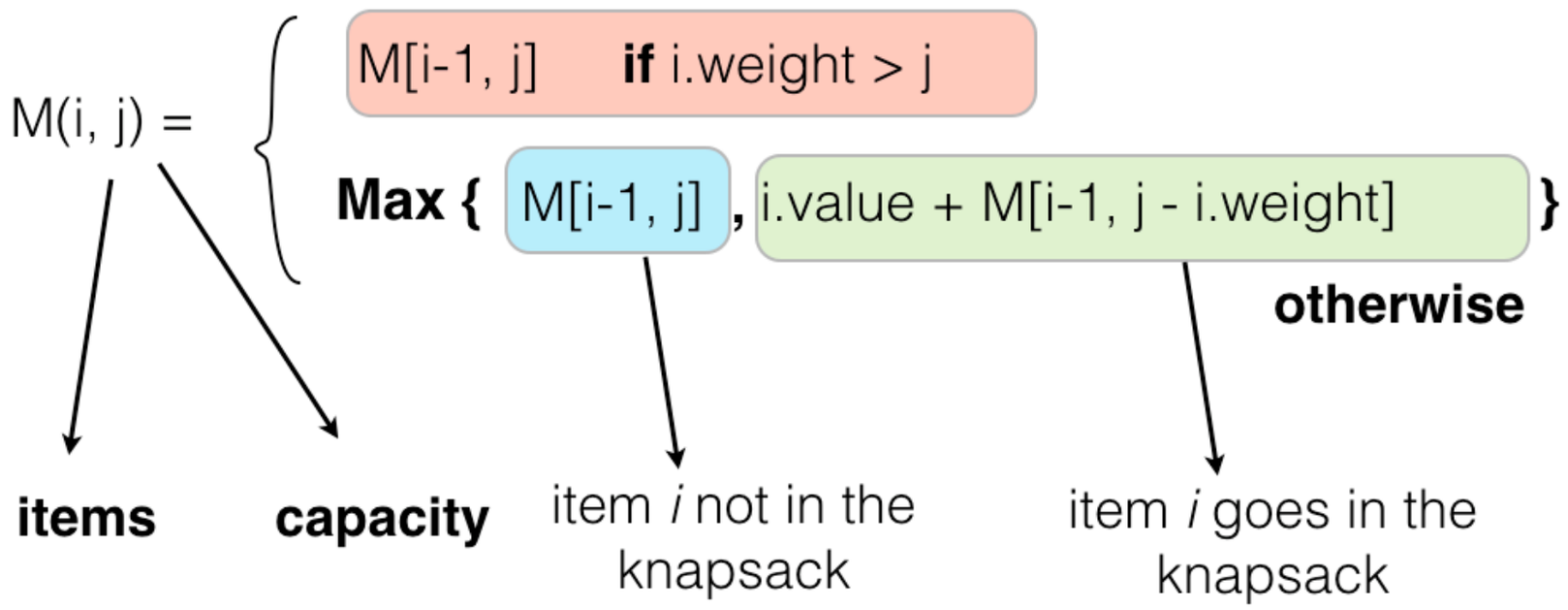


The *str* of the standard python list defaults to *__repr__* for printing its objects

```python
def knapsack_value(list_of_items, knapsack_capacity):
    assert len(list_of_items) > 0, "No items"
    assert knapsack_capacity > 0, "No space to store anything?"
    assert type(knapsack_capacity) == int

    table = np.zeros(shape=(len(list_of_items) + 1, knapsack_capacity + 1))

    for i, item in enumerate(list_of_items, start=1):
        for j in range(1, knapsack_capacity+1):
            if item.weight > j:
                table[i, j] = table[i-1, j]
            else:
                table[i, j] = max(table[i-1, j], item.value + table[i-1, j - item.weight])
```



```
def knapsack_value(list_of_items, knapsack_capacity):
    assert len(list_of_items) > 0, "No items"
    assert knapsack_capacity > 0, "No space to store anything?"
    assert type(knapsack_capacity) == int

    table = np.zeros(shape=(len(list_of_items) + 1, knapsack_capacity + 1))

    for i, item in enumerate(list_of_items, start=1):
        for j in range(1, knapsack_capacity+1):
            if item.weight > j:
                table[i, j] = table[i-1, j]
            else:
                table[i, j] = max(table[i-1, j], item.value + table[i-1, j - item.weight])

return table[-1, -1]
```

Find list of Items

j →

i ↓

i \ j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4000
2	0	0	0	0	0	0	0	0	0	0	3500	3500	3500	3500	3500	3500	3500	3500	3500	3500	4000
3	0	0	0	0	0	0	0	0	0	1800	3500	3500	3500	3500	3500	3500	3500	3500	3500	5300	5300
4	0	0	0	0	400	400	400	400	400	1800	3500	3500	3500	3500	3900	3900	3900	3900	3900	5300	5300
5	0	0	1000	1000	1000	1000	1400	1400	1400	1800	3500	3500	4500	4500	4500	4500	4900	4900	4900	5300	5300
6	0	200	1000	1200	1200	1200	1400	1600	1600	1800	3500	3700	4500	4700	4700	4700	4900	5100	5100	5300	**5500**

Questions

Can you add items whose weight is less than 1 kg?

Can you take fractions of items using this technique?

Dynamic Programming

- Solves problems by **solving subproblems**.
- Each subproblem is solved only once, storing relevant information.
- Subproblems may **overlap**.