Cashier problem (T2.2)

The cashier problem is specified as follows. The function solving the problem has two arguments:

1) first, a natural number, given in the smallest currency unit (*e.g.*, pence), representing an **amount of money** that is to be paid out;
2) second, a sorted list with the **values of the existing coin types**, in the same currency unit (we assume that “1” is always among these values).

As the function’s return value, we expect a **list containing coin values** that add up to the requested amount; this must be the **shortest possible list**, *i.e.*, we want to use as few coins as possible.

Note that as a precondition it is assumed that the list passed as the function's second argument is already sorted.

Cashier problem (T2.2)

greedy algorithm

```
def cashier(amount, coin_types):
    coins = []
    remainder = amount

    while remainder >= coin_types[0]:

        for i in range(len(coin_types)-1, -1, -1):
            if remainder >= coin_types[i]:
                coins.append(coin_types[i])
                remainder -= coin_types[i]
                break
    return coins
```

amount = 12,
coin_types = [1, 2, 5, 10]

↓

remainder = 12,
coins = []

↓

remainder = 2,
coins = [10]

↓

remainder = 0,
coins = [10, 2]

↓

return [10, 2]

Cashier problem (T2.2)

Condition for the greedy algorithm:

The shortest sum containing coin *x* never consists of more coins than the shortest equivalent sum containing only coins < *x*.

```
def cashier(amount, coin_types):
    coins = []
    remainder = amount
    while remainder >= coin_types[0]:
        for i in range(len(coin_types)-1, -1, -1):
            if remainder >= coin_types[i]:
                coins.append(coin_types[i])
                remainder -= coin_types[i]
                break
    return coins
```

amount = 12,
coin_types = [1, 4, 9, 16]

↓

remainder = 12,
coins = []

↓

remainder = 3,
coins = [9]

↓

remainder = 2,
coins = [9, 1]

↓

remainder = 1,
coins = [9, 1, 1]

↓

remainder = 0,
coins = [9, 1, 1, 1]

↓

return [9, 1, 1, 1]

compare
[4, 4, 4]

Cashier problem (T2.2)

greedy algorithm

Let *n* = amount and *k* = len(**coin_types**).

```
def cashier(amount, coin_types):
    coins = []
    remainder = amount

    while remainder >= coin_types[0]:

        for i in range(len(coin_types)-1, -1, -1):
            if remainder >= coin_types[i]:
                coins.append(coin_types[i])
                remainder -= coin_types[i]
                break
    return coins
```

O(1) instructions

Upper bound: O(*n*) iterations

- Upper bound: O(*k*) iterations
  - O(1) time on average, for a well-managed **dyn. array**. O(1) time worst case if we were using a **linked list**.

Cashier problem (T2.2)

greedy algorithm

Let n = amount and k = len(**coin_types**).

```
def cashier(amount, coin_types):
    coins = []
    remainder = amount

    while remainder >= coin_types[0]:

        for i in range(len(coin_types)-1, -1, -1):
            if remainder >= coin_types[i]:
                coins.append(coin_types[i])
                remainder -= coin_types[i]
                break
    return coins
```

O(1) instructions

Upper bound: $O(n)$ iterations

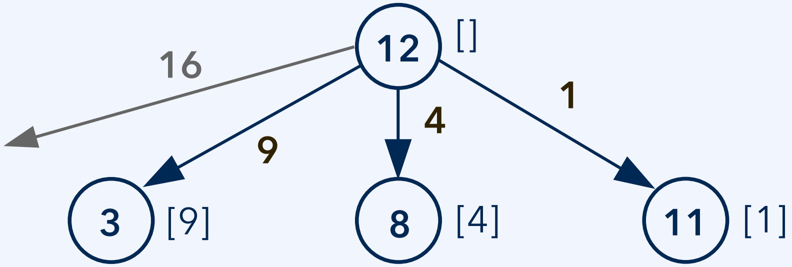
- Upper bound: $O(k)$ iterations
  - O(1) time* per iteration [*rigorous with a linked list]

$O(kn)$ time efficiency

that is $O(n)$ if k is treated as constant

Cashier problem (T2.2)

Illustration: Dynamic programming algorithm for the cashier problem



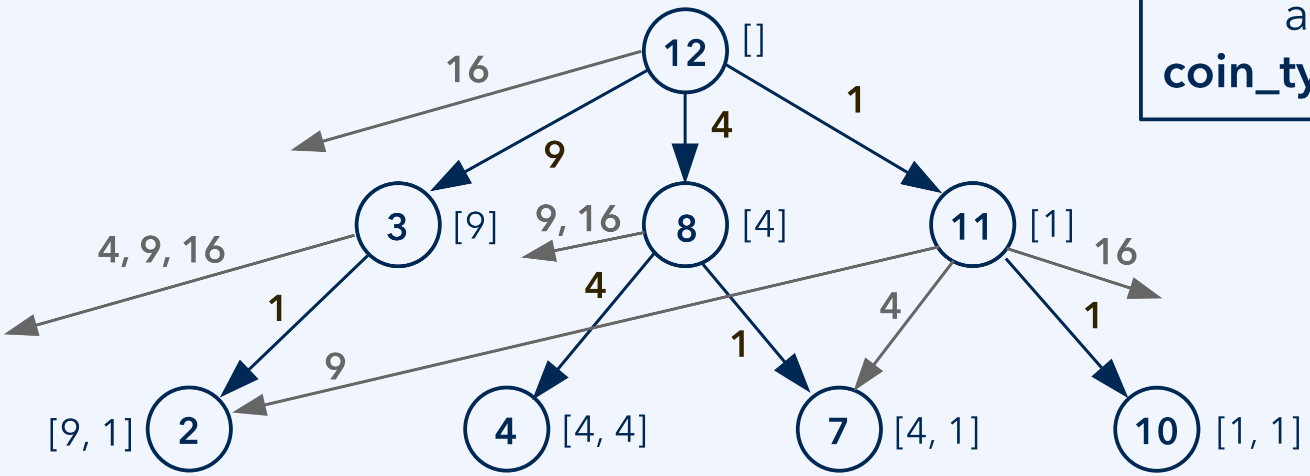
amount = 12,
coin_types = [1, 4, 9, 16]

If we return a 9-valued coin, the remainder reduces to 3.
If we return a 4-valued coin, the remainder reduces to 8.
If we return a 1-valued coin, the remainder reduces to 11.

Cashier problem (T2.2)

Illustration: Dynamic programming algorithm for the cashier problem

amount = 12,
coin_types = [1, 4, 9, 16]



one coin required

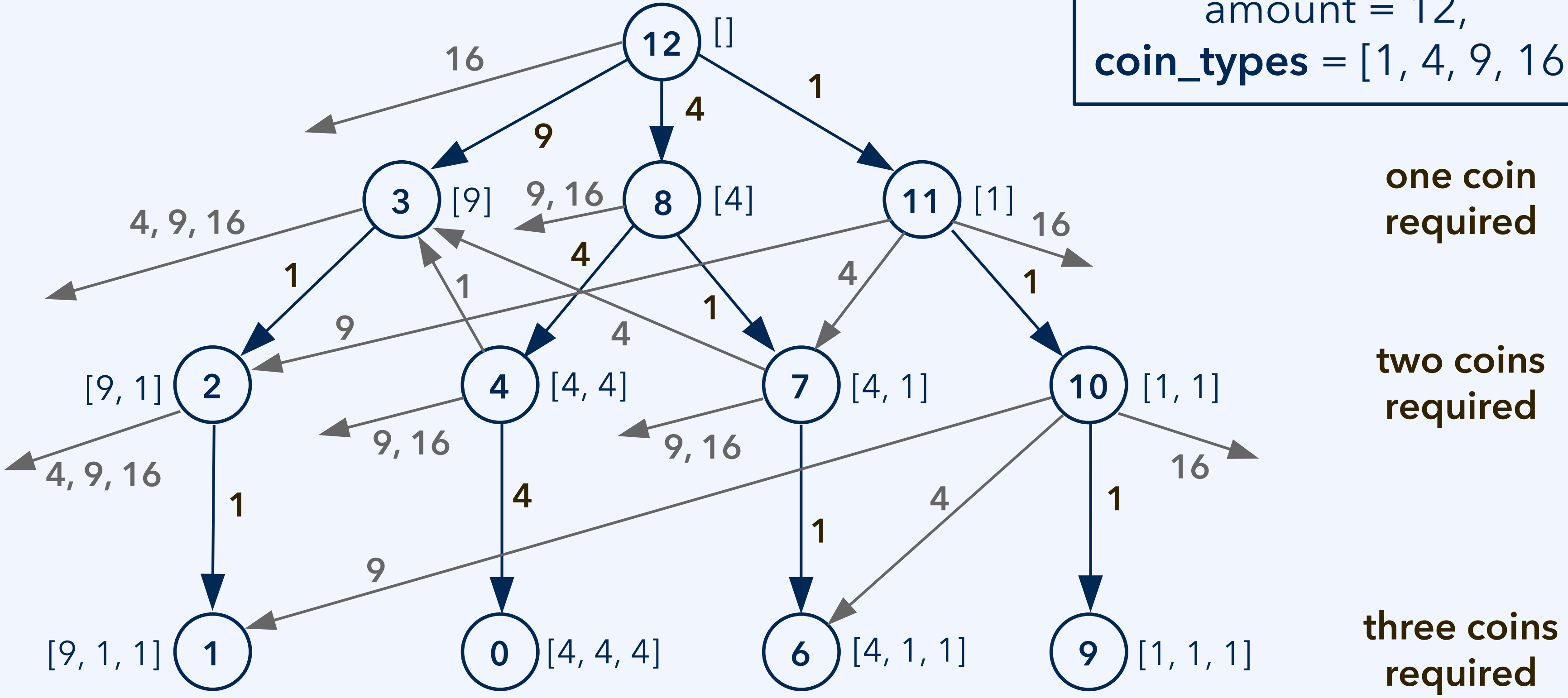
two coins required

A remainder of 12 currency units can be reached using zero coins.
A remainder of 3, 8, or 11 can be reached using one coin.
A remainder of 2, 4, 7, or 10 can be reached using two coins.

Cashier problem (T2.2)

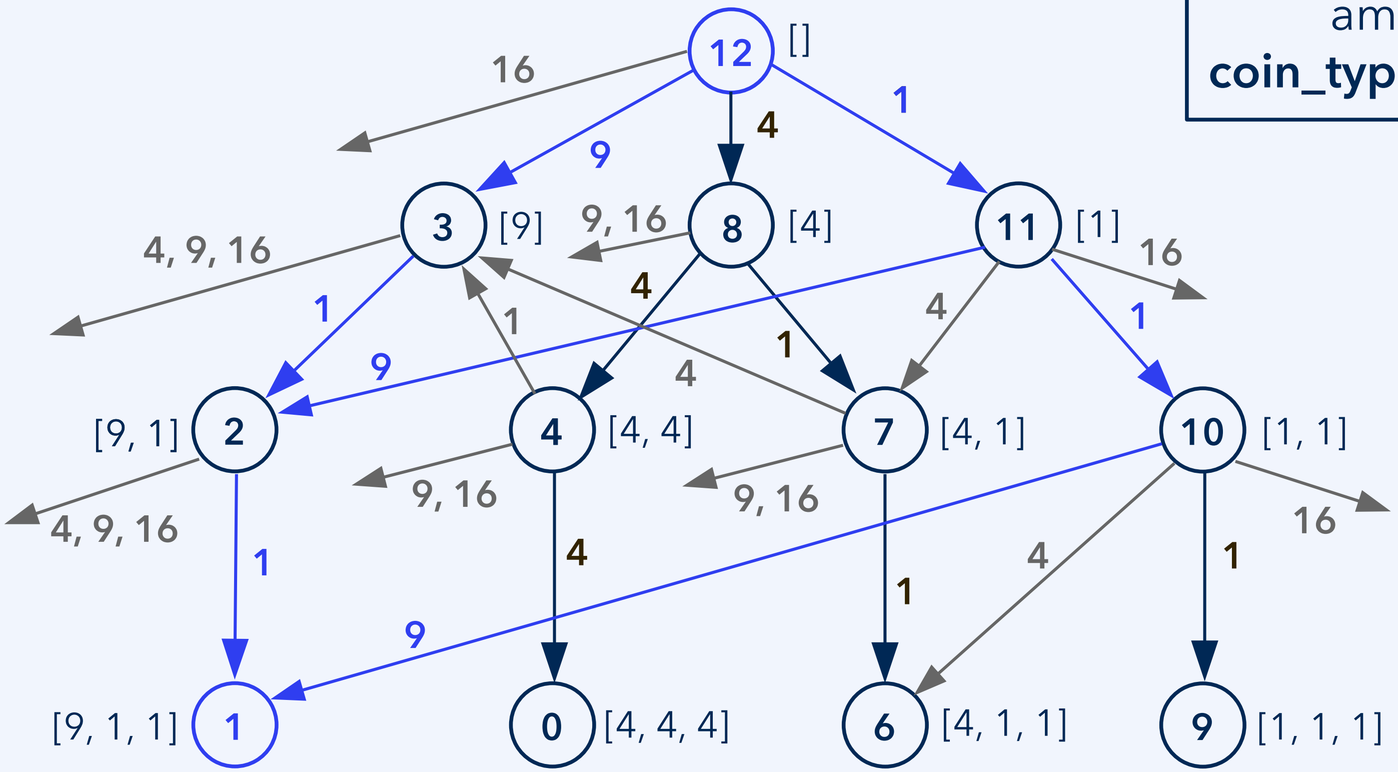
Illustration: Dynamic programming algorithm for the cashier problem

amount = 12,
coin_types = [1, 4, 9, 16]



Cashier problem (T2.2)

Overlapping subproblems, *e.g.*, equivalence of [9, 1, 1], [1, 9, 1], and [1, 1, 9]



amount = 12,
coin_types = [1, 4, 9, 16]

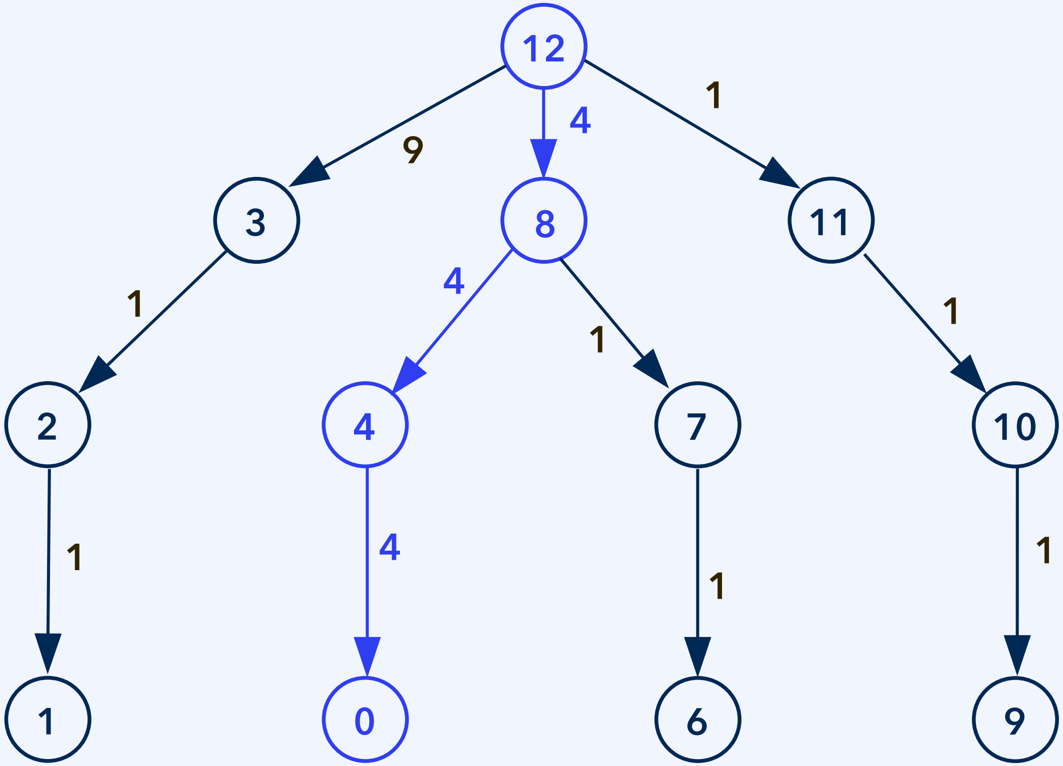
one coin required

two coins required

three coins required

Cashier problem (T2.2)

Illustration: Dynamic programming algorithm for the cashier problem



This reduces to computing a **BFS spanning tree**,

over a graph with at most $n+1$ nodes,

with node out-degree upper bound of k.

Time efficiency O(kn), same as for the greedy algorithm.