Dynamic Programming

- A paradigm to design algorithms.

- We want to compute the *optimal solution* of a problem.

- We divide the problem into sub-problems of smaller size.

- We calculate *optimal solutions* of sub-problems, which *will be pieces* of the *optimal solution* of the original problem

---

- Sub-problems may overlap! They may appear again and again… Think about the Fibonacci problem F(n) = F(n-1) + F(n-2)

- Let’s keep a table of solutions to store *optimal solutions* of sub-problems. (In this way, we calculate the solution of a sub-problem just once!)

Knapsack Problem

Item:					
Weight:	10	2	4	3	11
Value:	20	8	14	13	35

...and we have a knapsack with (max. weight) capacity 10

Which items should we insert in the knapsack to get
The maximum possible value without exceeding the capacity?

Knapsack Problem

Item:					
Weight:	10	2	4	3	11
Value:	20	8	14	13	35

...and we have a knapsack with (max. weight) capacity 10

Which items should we insert in the knapsack to get
The maximum possible value without exceeding the capacity?

Two variants of this problem!

Unbounded knapsack → assume infinite copies of items

0/1 Knapsack → assume only one copy of each item

Knapsack Problem

Item:	🐢	💡	🍉	🌮	🚒
Weight:	10	2	4	3	11
Value:	20	8	14	13	35

...and we have a knapsack with (max. weight) capacity 10

Which items should we insert in the knapsack to get
The maximum possible value without exceeding the capacity?

Two variants of this problem!

Unbounded knapsack → assume infinite copies of items

weight=10
value=42

0/1 Knapsack → assume only one copy of each item

Knapsack Problem

Item:	🐢	💡	🍉	🌮	🚒
Weight:	10	2	4	3	11
Value:	20	8	14	13	35

...and we have a knapsack with (max. weight) capacity 10

Which items should we insert in the knapsack to get
The maximum possible value without exceeding the capacity?

Two variants of this problem!

Unbounded knapsack → assume infinite copies of items

weight=10
value=42

0/1 Knapsack → assume only one copy of each item

weight=9
value=35

(Unbounded) Knapsack Problem

Let us assume n classes of items, and a knapsack of capacity x.

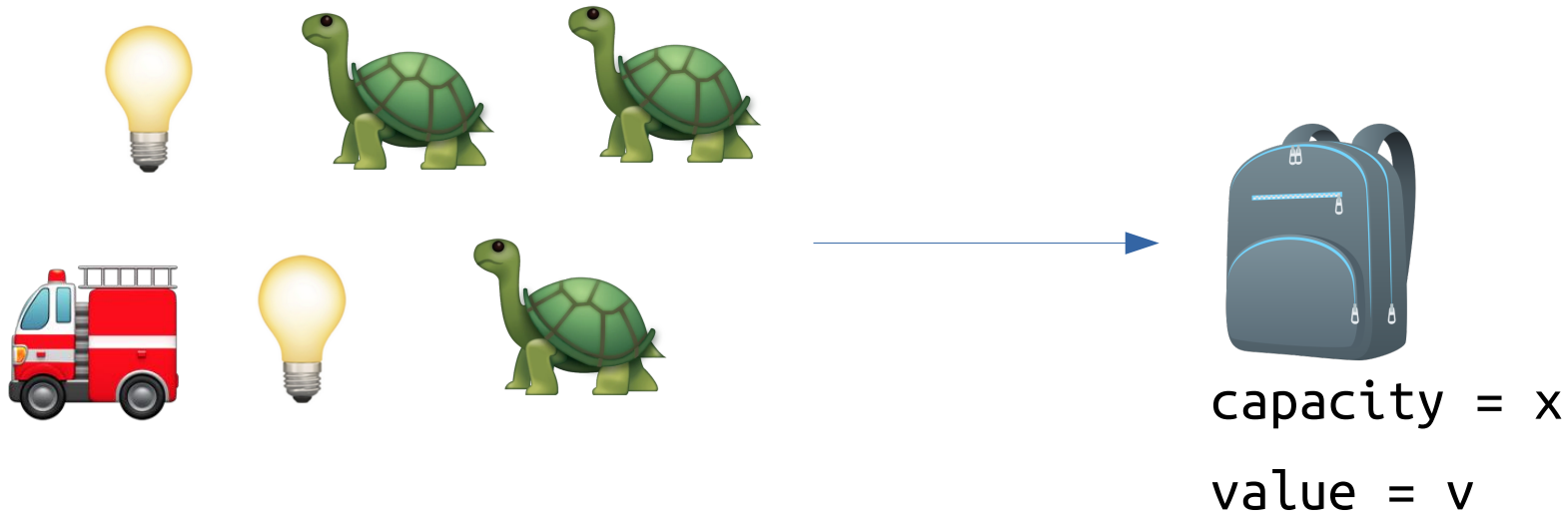
Item of class i has weight w_i and value v_i

(Unbounded) Knapsack Problem

Let us assume n classes of items, and a knapsack of capacity x.

Item of class i has weight w_i and value v_i

Suppose this is the optimal solution for capacity x which has at least one copy of item i



(Unbounded) Knapsack Problem

Let us assume n classes of items, and a knapsack of capacity x.

Item of class i has weight w_i and value v_i

Then, this is the optimal solution for capacity x - w_i which has at least one copy of item i

capacity = x - w_i

value = v - v_i

(Unbounded) Knapsack Problem

Let K(x) be the *optimal value* for a knapsack with capacity x

$K(x) = \max_i($  +  $)$

Optimal value for a *smaller* knapsack with capacity $x-w_i$

Value of item i

where item i is the item whose value maximizes this formula, such that its weight $w_i \leq x$

(Unbounded) Knapsack Problem

Let K(x) be the *optimal value* for a knapsack with capacity x

$K(x) = \max_i($  $+$  $)$

Optimal value for a *smaller* knapsack with capacity $x-w_i$

Value of item i

where item i is the item whose value maximizes this formula, such that its weight $w_i \leq x$

$$K(x) = \max_i(K(x-w_i) + v_i)$$

and K(x)=0 if there is no item i such that its weight $w_i \leq x$

(Unbounded) Knapsack Problem – pseudocode

K[x]: optimal value for a knapsack of capacity x

```
unboundedKnapsack(w, n, weights, values)
{
    K[0] = 0;

    for x = 1,...,w:
        K[x] = 0;
        for i = 1,...,n:
            if(w_i <= x):
                K[x] = max(K[x],K[x-w_i] + v_i);

    return K[w];
}
```

(Unbounded) Knapsack Problem – pseudocode

K[x]: optimal value for a knapsack of capacity x

```
unboundedKnapsack(w, n, weights, values)
{
    K[0] = 0;
    ITEMS[0] = ∅;

    for x = 1,...,w:
        K[x] = 0;
        ITEMS[x] = ∅;
        for i = 1,...,n:
            if(w_i <= x):
                K[x] = max(K[x],K[x-w_i] + v_i);
            if(K[x] was updated):
                ITEMS[x] = ITEMS[x-w_i] U { item i };

    return K[w];
}
```