Динамическое программирование. Реализация

Задача о рюкзаке (revisited)

- **Дано:**
  - Набор товаров (n штук), каждый из которых характеризуется весом q_i и стоимостью p_i.
  - Рюкзак, загрузка которого не должна превышать Q_{max}
- **Требуется:**
  - Выбрать набор вещей, обладающих наибольшей стоимостью при условии выполнения требования по максимальной загрузке

Задача о рюкзаке. Уравнение Беллмана

- Описание процесса:
  - Этапы – предметы (*n*).
  - Выигрыш – стоимость предметов в рюкзаке.
  - Управление – решение о том, брать или не брать предмет ($u_i \in \{0,1\}$).
  - Состояние – остаточная вместимость рюкзака ($S_i \in \{0,..,Q_{max}\}$).
- Уравнение Беллмана:

$$W_i(S_i) = \max_{u_i \in \{u | u \in \{0,1\}, u q_i \leq S_i\}} \{p_i u_i + W_{i+1}(S_i - q_i u_i)\}$$

Цена *i*-того предмета

Вес *i*-того предмета

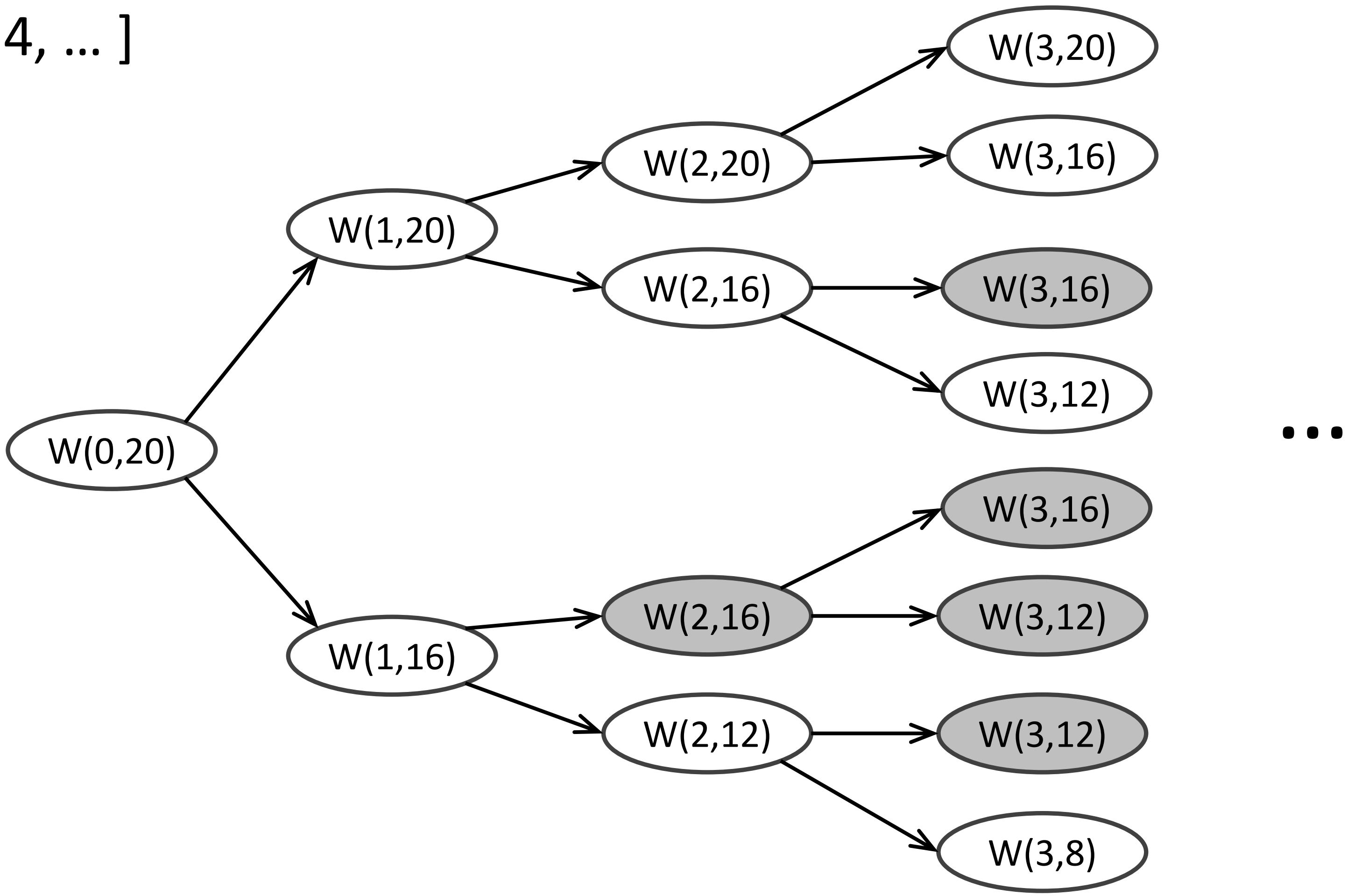
Рекурсивная реализация (наивная)

```
class NaiveKnapsackSolver:
    def __init__(self, weights, prices, capacity):
        self.q = weights
        self.p = prices
        self.c = capacity

    def W(self, stage, state):
        if stage >= len(self.p):
            return (0, None)

        best_w = None
        best_u = None
        for u in (0, 1):
            if u * self.q[stage] <= state:
                wi = self.p[stage] * u + self.W(stage + 1, state – self.q[stage] * u)[0]
                if best_w is None or wi > best_w:
                    best_w = wi
                    best_u = u
        return (best_w, best_u)

    def solve(self):
        return self.W(0, self.c)
```

$$W_i(S_i) = \max_{u_i \in \{u | u \in \{0,1\}, u q_i \le S_i\}} \{p_i u_i + W_{i+1}(S_i - q_i u_i)\}$$

Проблемы наивной реализации

- Пусть:
  - q_i = [4, 4, 4, …]
  - Q_{max} = 20



Проблемы наивной реализации

- Очень неэффективно
  - В первую очередь, из-за многократного пересчета одних и тех же значений
- Нет возможности восстановить оптимальное управление

Рекурсивная реализация (наивная)

```python
def W(self, stage, state):

    if stage >= len(self.p):
        return (0, [])

    best_w = None
    best_u = None
    for u in (0, 1):
        if u * self.q[stage] <= state:
            w1, u1 = self.W(stage + 1, state - u * self.q[stage])
            wi = self.p[stage] * u + w1
            if best_w is None or wi > best_w:
                best_w = wi
                best_u = [u] + u1
    return (best_w, best_u)
```

$$\boxed{W_i(S_i) = \max_{u_i \in \{u | u \in \{0,1\}, u q_i \le S_i\}} \{p_i u_i + W_{i+1}(S_i - q_i u_i)\}}$$

Рекурсивная реализация (с мемоизацией)

```
class StandardKnapsackSolver:
    def __init__(self, weights, prices, capacity):
        self.q = weights
        self.p = prices
        self.c = capacity
        self.W_cache = [{} for _ in self.q]

    def W(self, stage, state):
        if stage >= len(self.p):
            return (0, None)

        if state in self.W_cache[stage]:
            return self.W_cache[stage][state]

        best_w = None
        best_u = None
        for u in (0, 1):
            if u * self.q[stage] <= state:        # условие допустимости управления
                wi = self.p[stage] * u + self.W(stage + 1, state - self.q[stage] * u)[0]
                if best_w is None or wi > best_w:
                    best_w = wi
                    best_u = u
        self.W_cache[stage][state] = (best_w, best_u)
        return (best_w, best_u)
```

$$\boxed{W_i(S_i) = \max_{u_i \in \{u | u \in \{0,1\}, u q_i \le S_i\}} \{p_i u_i + W_{i+1}(S_i - q_i u_i)\}}$$

Рекурсивная реализация (с мемоизацией)

…продолжение

```
def restore_optimal(self):
    control = []
    state = self.c
    for stage in range(len(self.q)):
        u = self.W(stage, state)[1]
        control.append(u)
        state = state - u * self.q[stage]
    return control

def solve(self):
    return (self.W(0, self.c)[0], self.restore_optimal())
```

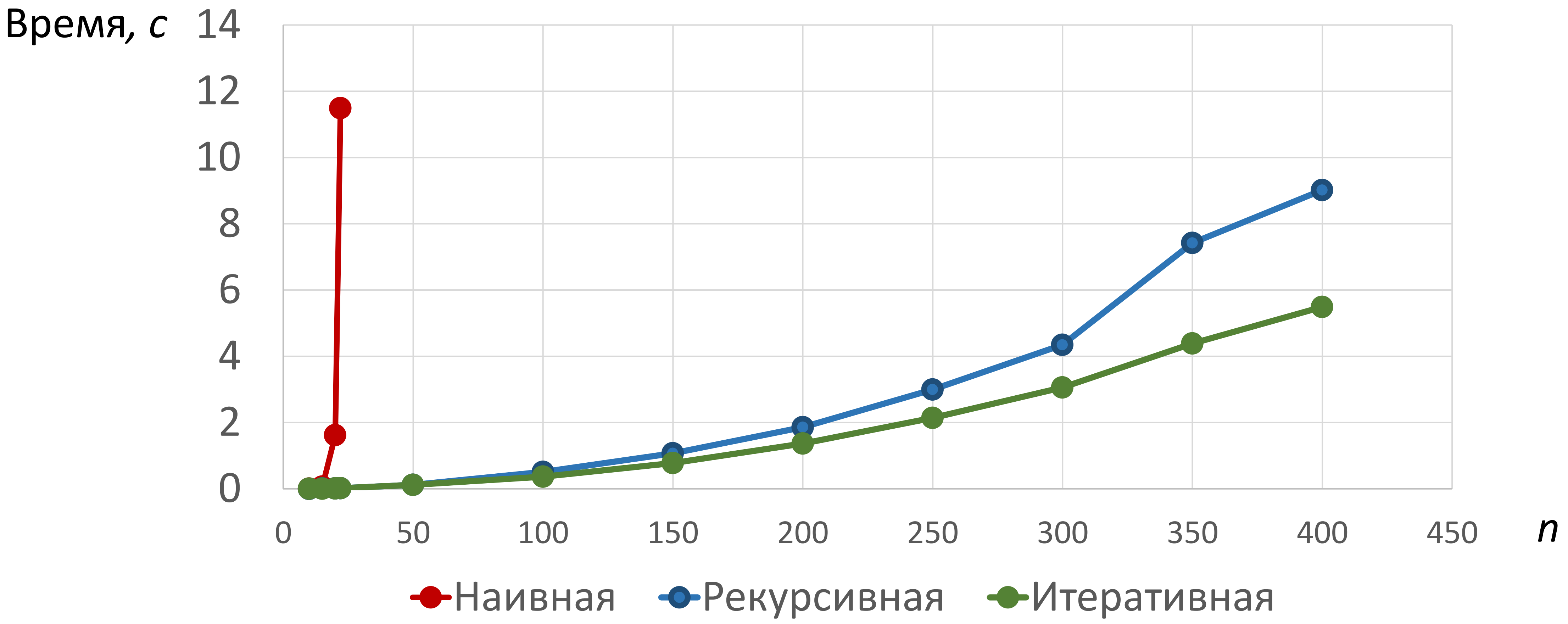
Итеративная реализация

```python
def calculate(self):
    self.W_cache = [{} for _ in self.q]

    # Фиктивный шаг
    self.W_cache.append({})
    for state in range(0, self.c+1):
        self.W_cache[len(self.q)][state] = (0, None)

    for stage in reversed(range(len(self.q))):
        for state in range(0, self.c+1):
            best_w = None
            best_u = None
            for u in (0, 1):
                phi = state - self.q[stage] * u
                if phi >= 0:
                    wi = self.p[stage] * u + self.W_cache[stage + 1][phi][0]
                    if best_w is None or wi > best_w:
                        best_w = wi
                        best_u = u
            self.W_cache[stage][state] = (best_w, best_u)
```

$$W_i(S_i) = \max_{u_i \in \{u | u \in \{0,1\}, uq_i \leq S_i\}} \{p_i u_i + W_{i+1}(S_i - q_i u_i)\}$$

Сравнение времени выполнения

Вес и стоимость предмета ~U(5; 50), грузоподъемность рюкзака ~1/2 суммарного веса предметов.



Стилистические улучшения

- Разделение **описания задачи**:
  - Количество этапов
  - Множество состояний на каждом из этапов
  - Множество допустимых управлений в каждом из состояний
  - «Эффекты» управления
- …и обобщенного кода, реализующего схему ДП в соответствующих терминах:

```
def W(…):
    …
    for (u, val, phi) in problem.valid_controls(stage, s):
        wi = val + self.W(stage + 1, phi)
    …
```

Тонкости при работе с непрерывным состоянием

- Простейший вариант – дискретизация
- Использование вещественного ключа в словаре

```
phi = … # некоторая функция от state и u
wi = … + self.W_cache[stage + 1][phi][0]
```

```
2 == math.sqrt(2) ** 2          -> False!
0.1 + 0.1 + 0.1 - 0.3 == 0      -> False!
```

```
a = {}
a[2] = 'Two'
a[math.sqrt(2) ** 2] = 'Two‘
```

```
{2: 'Two',
2.0000000000000004: 'Two'}
```

Контролируйте размер словаря!

Тонкости при работе с непрерывным состоянием

- Использование вещественного ключа в словаре. Возможное (но не единственное!) решение – округление:

```
phi = … # некоторая функция от state и u
wi = … + self.W_cache[stage + 1][round(phi, 2)][0]
```

```
round(2, 2) == round(math.sqrt(2) ** 2, 2)          -> True
round(0.1 + 0.1 + 0.1 - 0.3, 2) == round(0, 2)    -> True
```

```
a = {}
a[round(2, 2)] = 'Two'
a[round(math.sqrt(2) ** 2, 2)] = 'Two‘
```

```
{2: 'Two'}
```

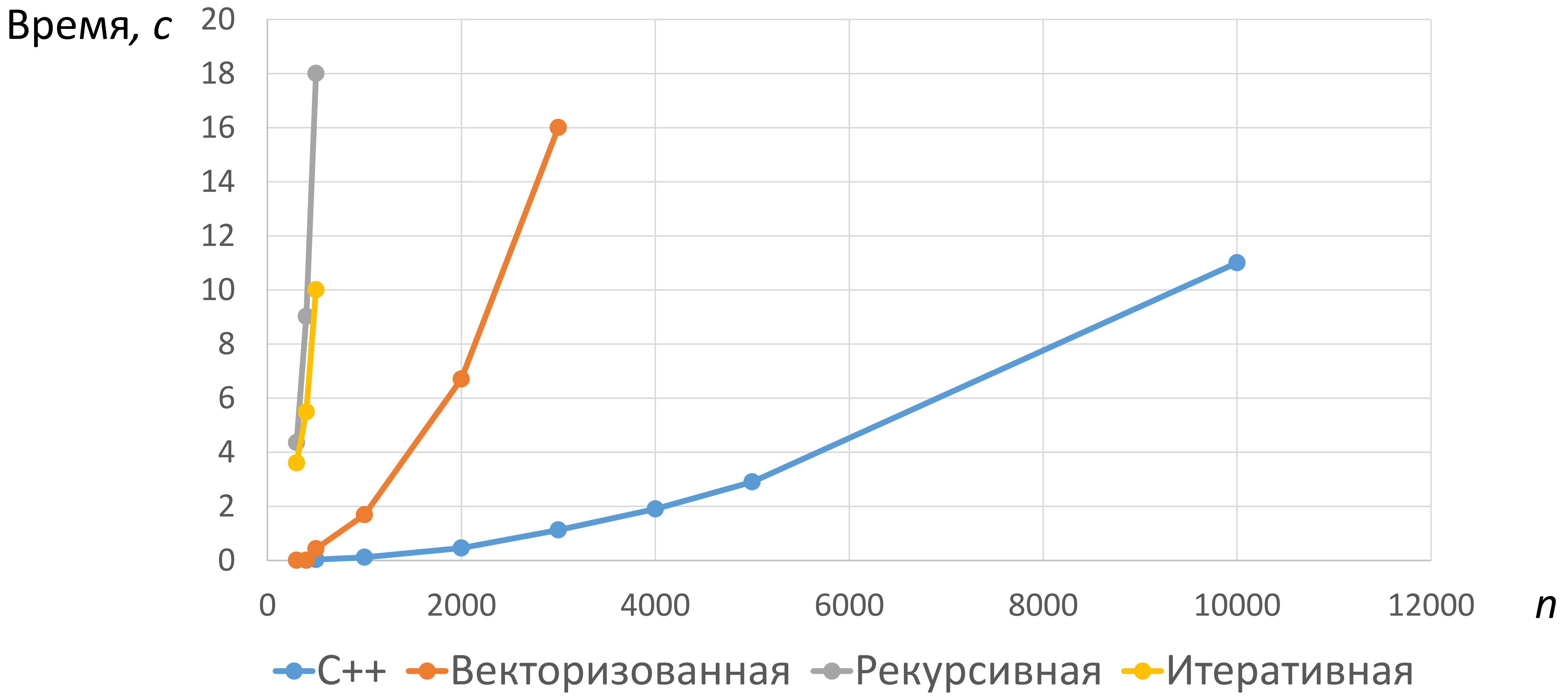
Тонкости при работе с непрерывным состоянием

```
class FloatAwareDict:
    def __init__(self, precision):
        self.precision = precision
        self.dict = {}
    def __setitem__(self, k, v):
        self.dict[round(k, self.precision)] = v
    def __getitem__(self, k):
        return self.dict[round(k, self.precision)]
    def __iter__(self):
        return self.dict.__iter__()

fad = FloatAwareDict(2)
fad[0.1 + 0.1 + 0.1 - 0.3] = 'Zero'
fad[0] -> 'Zero'
fad[0.0000001] -> 'Zero'
```

Векторизованная реализация

```
class VectorizedIterativeNPKnapsackSolver:
    def __init__(self, weights, prices, capacity):
        self.q = weights
        self.p = prices
        self.c = capacity
        self.W_cache = [np.zeros(self.c+1) for _ in self.q]
        self.U_cache = [np.zeros(self.c+1, dtype=np.int32) for _ in self.q]

    def calculate(self):
        # Фиктивный шаг
        self.W_cache.append(np.zeros(self.c + 1))
        self.U_cache.append(np.zeros(self.c + 1, dtype=np.int32))

        NINF = -1000
        # Инициализация векторов управления и состояния
        x = np.array([0, 1])
        s = np.arange(self.c + 1)
        # Преобразование их в вид, подходящий для вычислений
        xx = np.broadcast_to(x.reshape(1, -1), (self.c + 1, len(x)))
        ss = np.broadcast_to(s.reshape(-1, 1), (self.c + 1, len(x)))

        for stage in reversed(range(len(self.q))):
            # Вычисление новых состояний
            new_state = ss - self.q[stage] * xx
            valid_state = new_state >= 0
            new_state[~valid_state] = 0

            tmp_w = self.p[stage] * xx + self.W_cache[stage + 1][new_state]
            tmp_w[~valid_state] = NINF

            u = np.argmax(tmp_w, axis=1, keepdims=True)
            W = np.take_along_axis(tmp_w,
                                   u,
                                   axis=-1)

            self.W_cache[stage] = W.ravel()
            self.U_cache[stage] = u.ravel()
```

Еще о времени выполнения



Резюме

- Существует два базовых способа программной реализации применения метода динамического программирования
  - Итеративный
  - Рекурсивный
- Итеративный
  - (-) Возможно, требует больше памяти
  - (-) Чуть сложнее в реализации
  - (+) Как правило, работает несколько быстрее
- Рекурсивный
  - (+) Может сэкономить память
  - (+) Чуть проще в реализации
  - (-) Как правило, работает чуть медленнее
  - (-) Можно «упереться» в ограничения стека