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:

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

Цена *i*-того предмета

Вес *i*-того предмета

Рекурсивная реализация (наивная)

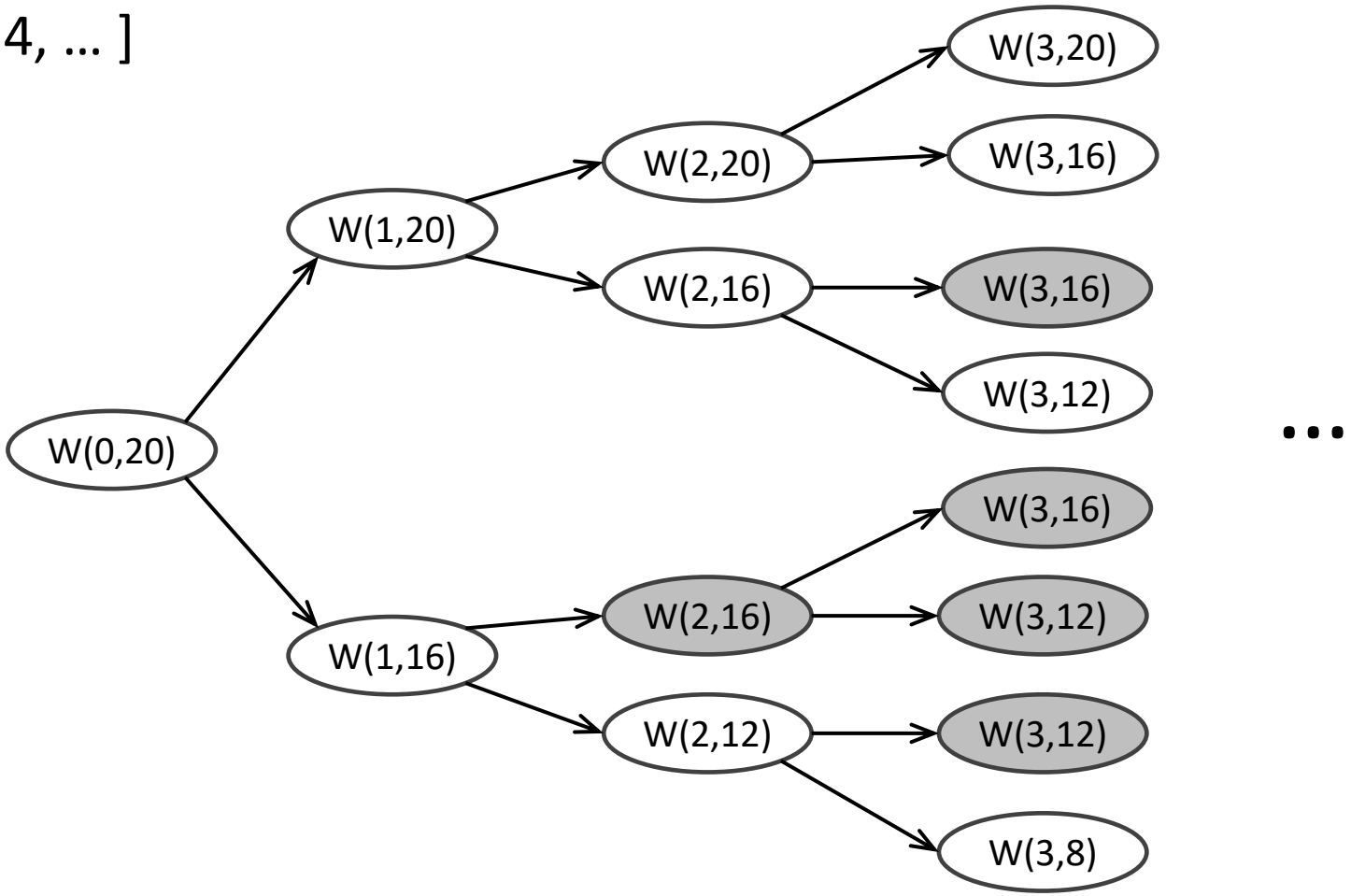
```
1.   class NaiveKnapsackSolver:
2.       def __init__(self, weights, prices, capacity):
3.           self.q = weights
4.           self.p = prices
5.           self.c = capacity

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

9.           best_w = None
10.          best_u = None
11.          for u in (0, 1):
12.              if u * q[stage] <= state:
13.                  wi = self.p[stage] * u + self.W(stage + 1, state - q[stage] * u)[0]
14.                  if best_w is None or wi > best_w:
15.                      best_w = wi
16.                      best_u = u
17.          return (best_w, best_u)
18.
19.      def solve(self):
20.          return self.W(0, self.c)
```

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

НЕ ИСПОЛЬЗОВАТЬ!

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 (с мемоизацией)

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

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

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

12.           best_w = None
13.           best_u = None
14.           for u in (0, 1):
15.               if u * self.q[stage] <= state:        # условие допустимости управления
16.                   wi = self.p[stage] * u + self.W(stage + 1, state - self.q[stage] * u)[0]
17.                   if best_w is None or wi > best_w:
18.                       best_w = wi
19.                       best_u = u
20.           self.W_cache[stage][state] = (best_w, best_u)
21.           return (best_w, best_u)
```

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

Рекурсивная реализация (с мемоизацией)

…продолжение

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

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

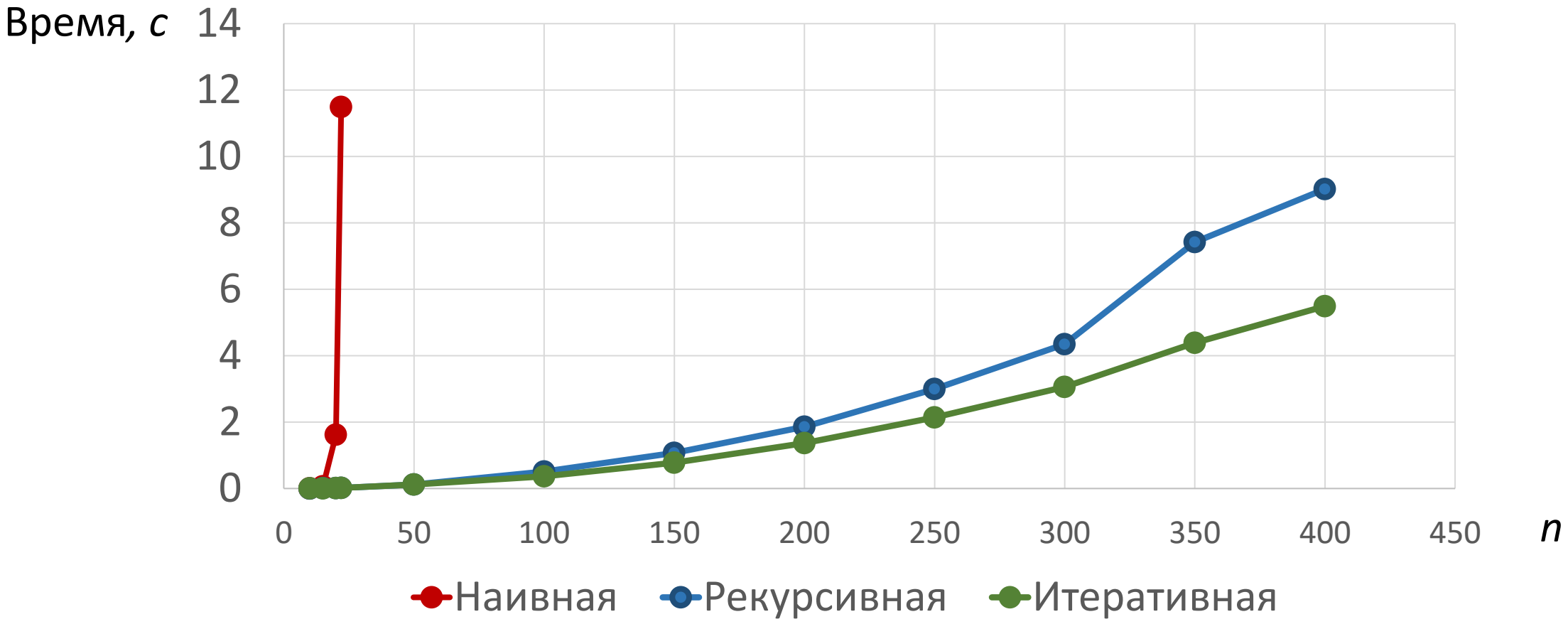
Итеративная реализация

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

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

7.       for stage in reversed(range(len(self.q))):
8.           for state in range(0, self.c+1):
9.               best_w = None
10.              best_u = None
11.              for u in [0, 1]:
12.                  phi = state - self.q[stage] * u
13.                  if phi >= 0:
14.                      wi = self.p[stage] * u + self.W_cache[stage + 1][phi][0]
15.                      if best_w is None or wi > best_w:
16.                          best_w = wi
17.                          best_u = u
18.              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:

```
1.  def W(…):
2.          …
3.          for (u, val, phi) in problem.valid_controls(stage, s):
4.              wi = val + self.W(stage + 1, phi)
5.          …
```

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