Порядок выполнения работы:

1. Нахождение числа фиббоначи:

Динамическое программирование назад:

Инициализация

$F[0 \ldots n] = [-1, -1, \ldots, -1]$

Функция FibTD(n)

если $F[n] = -1$:
  если $n \leq 1$:
    $F[n] \leftarrow n$
  иначе:
    $F[n] \leftarrow \text{FibTD}(n-1) + \text{FibTD}(n-2)$
вернуть $F[n]$

Рисунок 1 – Алгоритм нахождения числа Фибоначчи

```
5   def fib(n, calculation_method=0):
6       """
7       Функции вычисления числа фибоначчи.
8       """
9       def fib_td(n):
10          """
11          Динамическое программирование назад.
12          """
13          if n <= 1:
14              f[n] = n
15          else:
16              f[n] = fib_td(n - 1) + fib_td(n-2)
17          return f[n]
18
```

Рисунок 2 – Фрагмент кода файла fib.py

Динамическое программирование вперед:

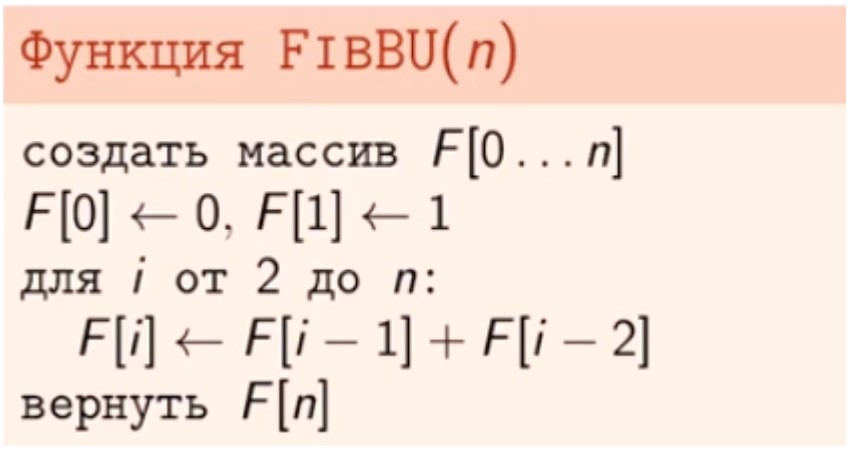


Рисунок 3 – Алгоритм нахождения числа Фибоначчи

```
19  def fib_bu(n):
20      """
21      Динамическое программирование вперед.
22      """
23      f = [-1] * (n+1)
24      f[0] = 0
25      f[1] = 1
26
27      for i in range(2, n + 1):
28          f[i] = f[i - 1] + f[i - 2]
29      return f[n]
```

Рисунок 4 – Фрагмент кода файла fib.py

Уменьшение количества потребления памяти:

```
Функция FibBUImproved(n)
если n ≤ 1:
  вернуть n

prev ← 0
curr ← 1

повторить (n − 1) раз:
  next ← prev + curr
  prev ← curr
  curr ← next
вернуть curr
```

Рисунок 5 – Алгоритм нахождения числа Фибоначчи

```python
def fib_bu_imroved(n, calculation_method):
    """
    Уменьшенным памяти.
    """
    if n <= 1:
        return n

    prev, curr = 0, 1

    for _ in range(n - 1):
        prev, curr = curr, prev + curr
    return curr

```

Рисунок 6 – Фрагмент кода файла fib.py

Результат:

```
Число Фибоначчи(10):
fib(N, 0) = 55
fib(N, 1) = 55
fib(N, 2) = 55
```

Рисунок 7 – Результат выполнения программы fib.py

2. Нахождение в списка НВП и самой НВП:

Поиск длины наибольшей возрастающей подпоследовательности

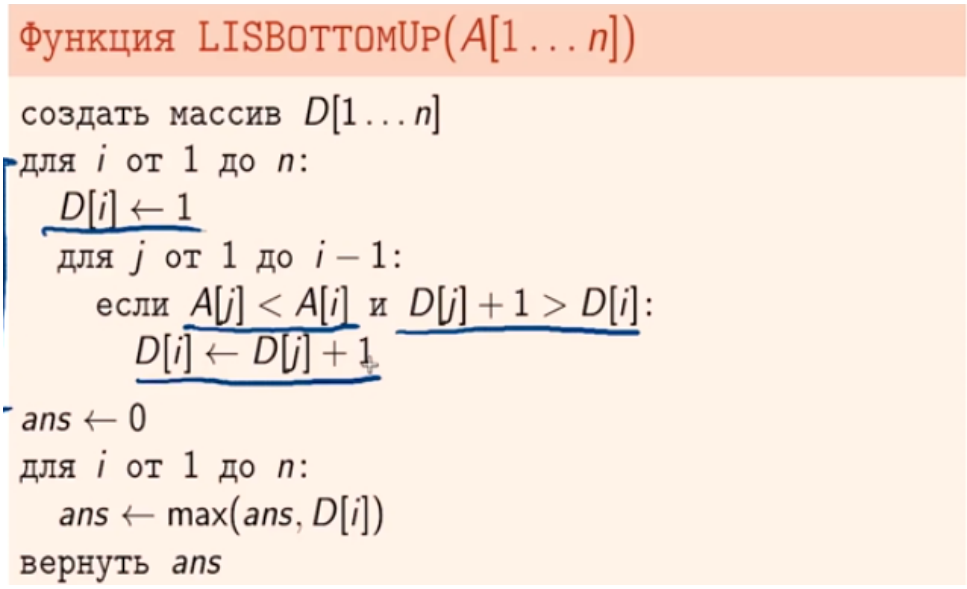


Рисунок 8 – Алгоритм нахождения длинны НВП

```python
def list_bottom_up(a):
    """
    Поиск длины наибольшей возрастающей подпоследовательности.
    """
    n = len(a)
    D = []

    for i in range(n):
        D.append(1)
        for j in range(i):
            if a[j] < a[i] and D[j] + 1 > D[i]:
                D[i] = D[j]+1

    ans = max(D)

    return ans

```

Рисунок 9 – Фрагмент кода файла (list.py)

Восстановление НВП с помощью списка prev:

Восстановление ответа

создать массив $L[1 \ldots ans]$ {индексы НВП}
$k \leftarrow 1$
для i от 2 до n:
  если $D[i] > D[k]$:
    $k \leftarrow i$
$j \leftarrow ans$
пока $k > 0$:
  $L[j] \leftarrow k$
  $j \leftarrow j - 1$
  $k \leftarrow \text{prev}[k]$

Рисунок 10 – Алгоритм нахождения НВП

```
23  def using_prev(prev, m_index):
24      """
25      Восстановление НВП с помощью списка prev
26      """
27      l = []
28      while True:
29          l.append(m_index)
30          if prev[m_index] == -1:
31              break
32          m_index = prev[m_index]
33
34      l.reverse()
35      return l
```

Рисунок 11 – Фрагмент кода файла (list.py)

Восстановление НВП без помощи списка prev:

```
Функция LISBottomUp2(A[1...n])
создать массивы D[1...n] и prev[1...n]
для i от 1 до n:
  D[i] ← 1, prev[i] ← −1
  для j от 1 до i − 1:
    если A[j] < A[i] и D[j] + 1 > D[i]:
      D[i] ← D[j] + 1, prev[i] ← j
ans ← 0
для i от 1 до n:
  ans = max(ans, D[i])
вернуть ans
```

Рисунок 12 – Алгоритм нахождения НВП

```python
def without_prev(d, ans, m_index):
    """
    Восстановление НВП без помощи списка prev
    """
    l = []
    while True:
        l.append(m_index)
        if ans == 1:
            break
        ans -= 1
        while True:
            m_index -= 1
            if d[m_index] == ans and a[m_index] < a[l[-1]]:
                break
    l.reverse()

    return l

```

Рисунок 13 – Фрагмент кода файла (list.py)

Поиск длины и самой НВП:

```python
def list_bottom_up_2(a):
    """
    Поиск длины и самой НВП.
    """
    n = len(a)
    d, prev = [], []
    for i in range(n):
        d.append(1)
        prev.append(-1)
        for j in range(i):
            if a[j] < a[i] and d[j] + 1 > d[i]:
                d[i] = d[j]+1
                prev[i] = j

    ans, max_index = 0, 0
    for i, item in enumerate(d):
        if ans < item:
            ans, max_index = item, i

    list_using_prev = using_prev(prev, max_index)
    list_without_prev = without_prev(d, ans, max_index)

    return ans, (list_using_prev, list_without_prev)

```

Рисунок 14 – Фрагмент кода файла (list.py)

Результат:

```
5
(5, ([1, 3, 5, 9, 11], [2, 3, 5, 10, 11]))
```

Рисунок 15 – Результат выполнения кода (list.py)

3. Поиск максимальной стоимости предметов в рюкзаке

Предметы могут повторяться:

Функция KNAPSACKWITHREPSBU($W, w_1, \ldots, w_n, c_1, \ldots, c_n$)

создать массив $D[0 \ldots W] = [0, 0, \ldots, 0]$
для w от 1 до W:
  для i от 1 до n:
    если $w_i \leq w$:
      $D[w] \leftarrow \max(D[w], D[w - w_i] + c_i)$
вернуть $D[W]$

Рисунок 16 – Алгоритм поиска максимальной стоимости предметов в рюкзаке

```python
def knapsack_with_reps(W, weight, cell):
    """
    Поиск максимальной стоимости предметов в рюкзаке.
    Предметы могут повторяться.
    """
    d = [0] * (W+1)
    for w in range(1, W+1):
        for weight_i, cell_i in zip(weight, cell):
            if weight_i <= w:
                d[w] = max(d[w], d[w - weight_i] + cell_i)
    return d[W]
```

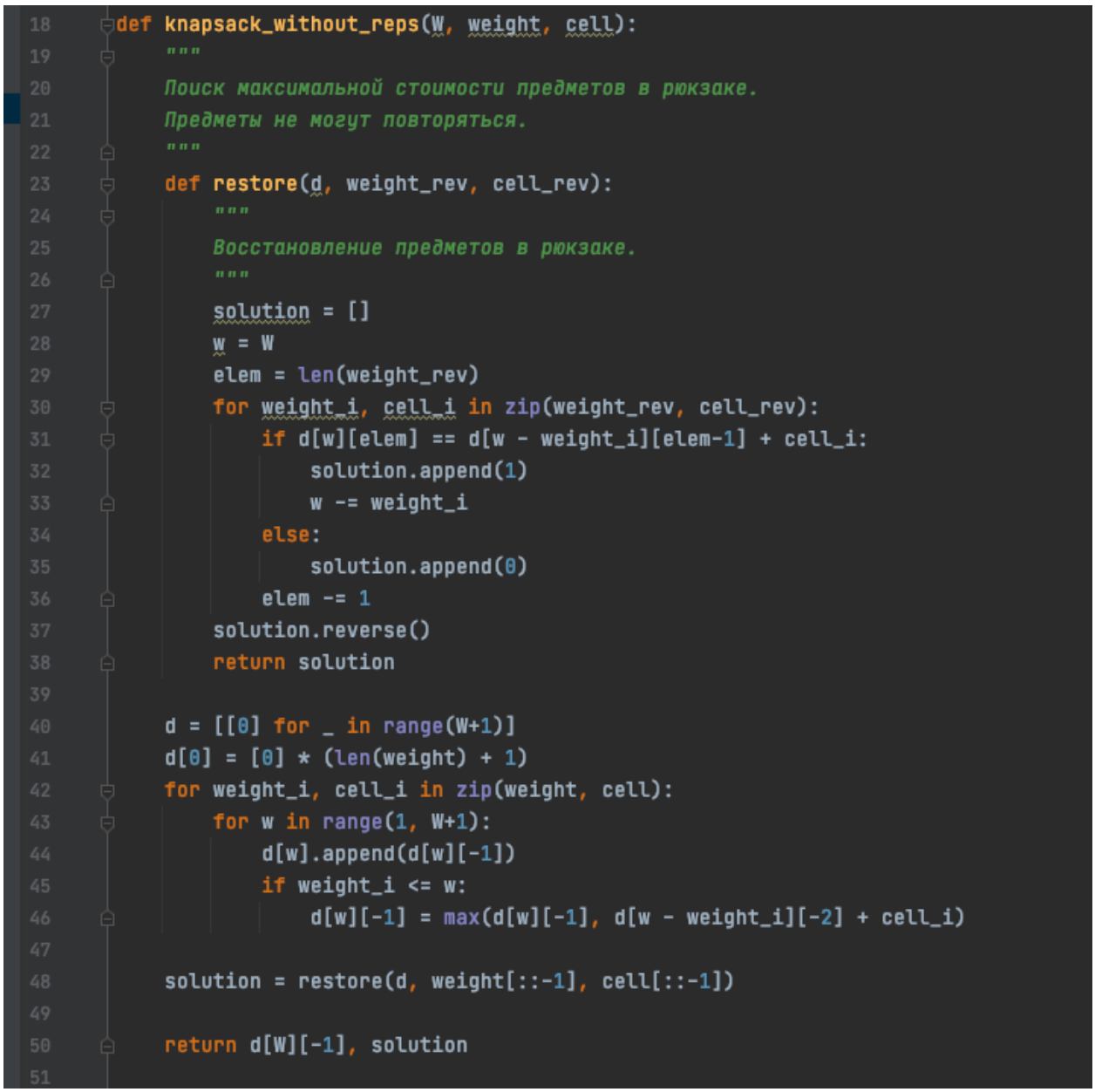
Рисунок 17 – Фрагмент кода файла (knapsack.py)

Предметы не могут повторяться:

KNAPSACKWITHOUTREPSBU($W, w_1, \ldots, w_n, c_1, \ldots, c_n$)

создать массив $D[0 \ldots W, 0 \ldots n]$
для w от 0 до W:
  $D[w, 0] \leftarrow 0$
для i от 0 до n:
  $D[0, i] \leftarrow 0$
для i от 1 до n:
  для w от 1 до W:
    $D[w, i] \leftarrow D[w, i-1]$
    если $w_i \leq w$:
      $D[w, i] = \max(D[w, i], D[w - w_i, i-1] + c_i)$
вернуть $D[W, n]$

Рисунок 18 – Алгоритм поиска максимальной стоимости предметов в рюкзаке



```python
def knapsack_without_reps(W, weight, cell):
    """
    Поиск максимальной стоимости предметов в рюкзаке.
    Предметы не могут повторяться.
    """
    def restore(d, weight_rev, cell_rev):
        """
        Восстановление предметов в рюкзаке.
        """
        solution = []
        w = W
        elem = len(weight_rev)
        for weight_i, cell_i in zip(weight_rev, cell_rev):
            if d[w][elem] == d[w - weight_i][elem-1] + cell_i:
                solution.append(1)
                w -= weight_i
            else:
                solution.append(0)
            elem -= 1
        solution.reverse()
        return solution

    d = [[0] for _ in range(W+1)]
    d[0] = [0] * (len(weight) + 1)
    for weight_i, cell_i in zip(weight, cell):
        for w in range(1, W+1):
            d[w].append(d[w][-1])
            if weight_i <= w:
                d[w][-1] = max(d[w][-1], d[w - weight_i][-2] + cell_i)

    solution = restore(d, weight[::-1], cell[::-1])

    return d[W][-1], solution
```

Рисунок 19 – Фрагмент кода файла (knapsack.py)

Результат:

```
with_rep_bu = 48
without_rep_bu = (46, [1, 0, 1, 0])
```

Рисунок 20 – Результат выполнения кода (knapsack.py)

Вывод: в процессе работы мы ознакомились с методами динамического программирования, такими как нахождение чисел Фибоначчи, поиск наибольшей возрастающей подпоследовательности (НВП) и алгоритм расчета максимальной стоимости предметов в рюкзаке.