

# Лабораторная работа №1

Выполнил: Чернышев Андрей, МСУ – 201

**Тема:** «Задача о рюкзаке (0, 1) методом динамического программирование на языке Python с использованием параллельных вычислений».

## План работы:

- 1) Решить задачу о рюкзаке полным перебором (brute force), замерить время выполнение программы.
- 2) Оптимизировать решение задачи с помощью методов динамического программирования.
- 3) Использовать параллельные вычисления для оптимизированного решения, замерить время, построить графики зависимости ускорения от числа процессов.

Все решения приводится на моем гитхабе.

Проверка решения с ответами осуществляется с помощью `numpy.testing.assert_allclose()`

Время работы программы указаны для 26 тестов

## 1. Решение задачи о рюкзаке полным перебором

Для решения данной задачи использовалась библиотека `itertools`, метод `combinations`, который возвращает кортежи перебираемых значений в лексикографическом порядке. Сложность алгоритма:  $2^n$

`537.6914975643158`

, то есть почти 9 минут.

### 1.1. Решение задачи перебором с помощью рекурсии.

Для решения данной задачи была написана функция `best_value`, которая рекурсивно находит максимальное значение, начиная с конца.

Время выполнения программы:

Time: 303.16358280181885, примерно 6 минут, то есть скорость увеличилось примерно в 1.5 раз

### 1.2. Ускорение с помощью библиотеки `numba`

Занимательный пост на Хабре: «Python (+numba) быстрее Си — серьёзно?!»:

<https://habr.com/ru/post/484136/>

`Numba` — это JIT-компилятор с открытым исходным кодом, который переводит Python и NumPy в быстрый машинный код с помощью LLVM через пакет `llvmlite` Python. Он предлагает широкий спектр вариантов распараллеливания кода Python для процессоров и графических процессоров, часто с незначительными изменениями кода.

Воспользуемся декоратором `@njit()` перед рекурсивной функцией.

Time: 9.040167808532715 сек, скорость без параллельных вычислений произошло примерно в **33** раза.

## 2. Метод динамического программирования

**Динамическое программирование** в теории управления и теории вычислительных систем — способ решения сложных задач путём разбиения их на более простые подзадачи. Он применим к задачам с оптимальной подструктурой, выглядящим как набор перекрывающихся подзадач, сложность которых чуть меньше исходной. В этом случае время вычислений, по сравнению с «наивными» методами, можно значительно сократить.

В общем виде задачу можно сформулировать так: из заданного множества предметов со свойствами «стоимость» и «вес» требуется отобрать подмножество с максимальной полной стоимостью, соблюдая при этом ограничение на суммарный вес.

Сложность алгоритма:  $n * capacity$

Time: 3.492619276046753 , без numba посчитал в 3 раза быстрее.

Проверял вместе с numba, получилось примерно 0.78 секунд

Алгоритм, взятый из википедии:

Рекуррентные соотношения:

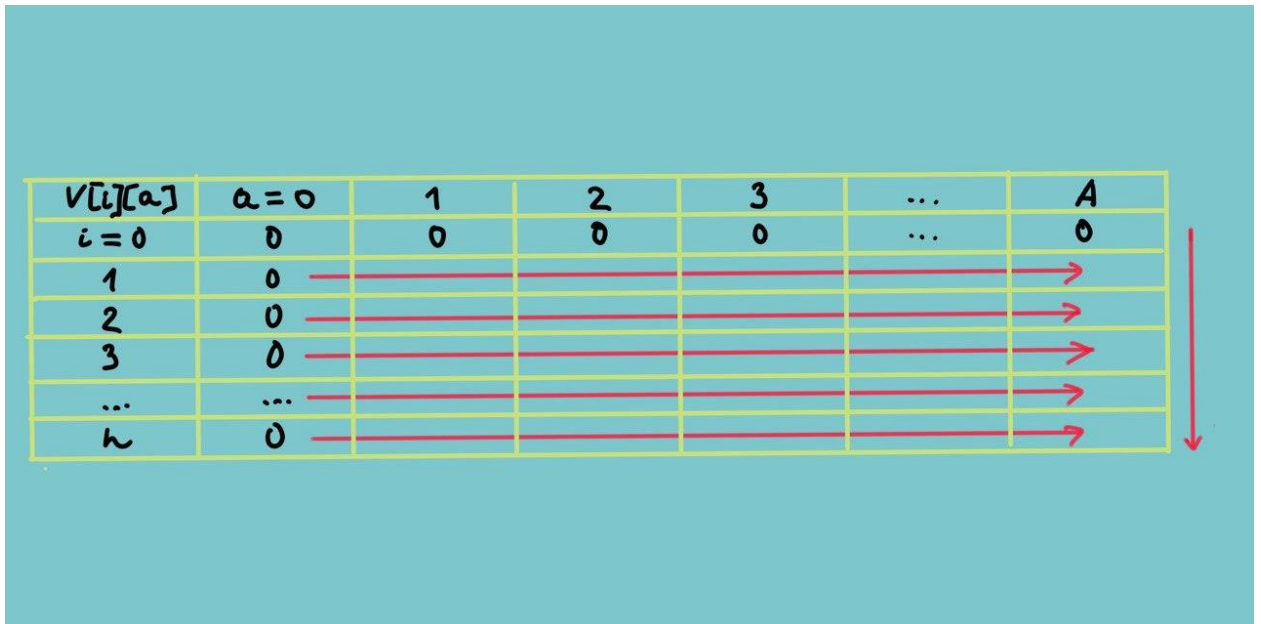
- $m[0, w] = 0$
- $m[i, w] = m[i - 1, w]$ , если  $w_i > w$
- $m[i, w] = \max(m[i - 1, w], m[i - 1, w - w_i] + v_i)$ , если  $w_i \leq w$

Вычисляя  $m[n, W]$ , можно найти точное решение. Если массив  $m[i, w]$  помещается в памяти машины, то данный алгоритм, вероятно, является одним из наиболее эффективных<sup>[12]</sup>.

Val	Wt	Item	Max Weight							
			0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0	0	0
1	1	1	0	1	1	1	1	1	1	1
4	3	2	0	1	1	4	5	5	5	5
5	4	3	0	1	1	4	5	6	6	9
7	5	4	0	1	1	4	5	7	8	9

Поскольку это значение берется сверху (показано серой стрелкой), элемент в этой строке не включается. Идем вертикально вверх по таблице, не кладя его в рюкзак. Теперь это значение  $K[n-1][W]$ , которое равно 9, не исходит сверху, что означает, что элемент в этой строке включен и идет вертикально вверх, а затем влево по весу включенного элемента (показано черной стрелкой). Продолжая этот процесс, положим в рюкзак предметы с весом 3 и 4 - общей стоимостью 9.

Вначале создаётся и заполняется таблица в виде вложенных списков. Нулевую строку и нулевой столбец заполняем нулями. Это базовый случай: когда площадь или количество элементов равны нулю, значение ячейки равно нулю. Далее таблица значений  $V$  будет заполняться строка за строкой слева направо и сверху вниз:



$V[i][a]$	$a=0$	1	2	3	...	A
$i=0$	0	0	0	0	...	0
1	0					
2	0					
3	0					
...	...					
n	0					

Если площадь текущего элемента меньше или равна площади (номеру столбца) текущей ячейки, вычисляем значение ячейки следуя правилу:

То есть выбираем максимальное из двух значений:

Сумма ценности текущего предмета  $value[i-1]$  и величины элемента из предыдущей строки  $i-1$  с площадью, меньшей на величину площади текущего предмета  $area[i-1]$ . : Нужно помнить, что элементы в таблице отличаются по нумерации на единицу из-за нулевой строки.

Значение элемента предыдущей строки с той же площадью, то есть из того же столбца, что текущая ячейка. То же значение устанавливается в случае, если площадь текущей ячейки меньше, чем площадь текущего элемента

За счёт такого подхода при одной и той же суммарной площади элементов происходит максимизация суммарной ценности.

```
elif weights[i - 1] <= j:
    best_costs[i][j] = max(costs[i - 1] + best_costs[i - 1][j - int(weights[i - 1])], best_costs[i - 1][j])
```

Забираем нужные элементы из последней строки таблицы

Найдём набор предметов с максимальной суммарной ценностью для указанной возможной площади. Начнём с нижнего правого угла таблицы с максимальным значением, и соберём предметы:

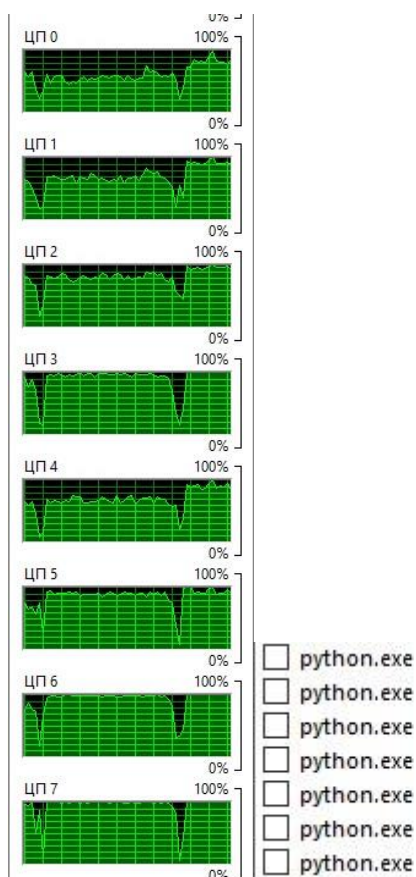
```
for i in range(n, 0, -1):
    if round(result) != round(best_costs[i - 1][int(cap_copy)]):
        best_combination[i - 1] = 1
        result -= costs[i - 1]
        cap_copy -= weights[i - 1]
```

### 3. Параллельные вычисления

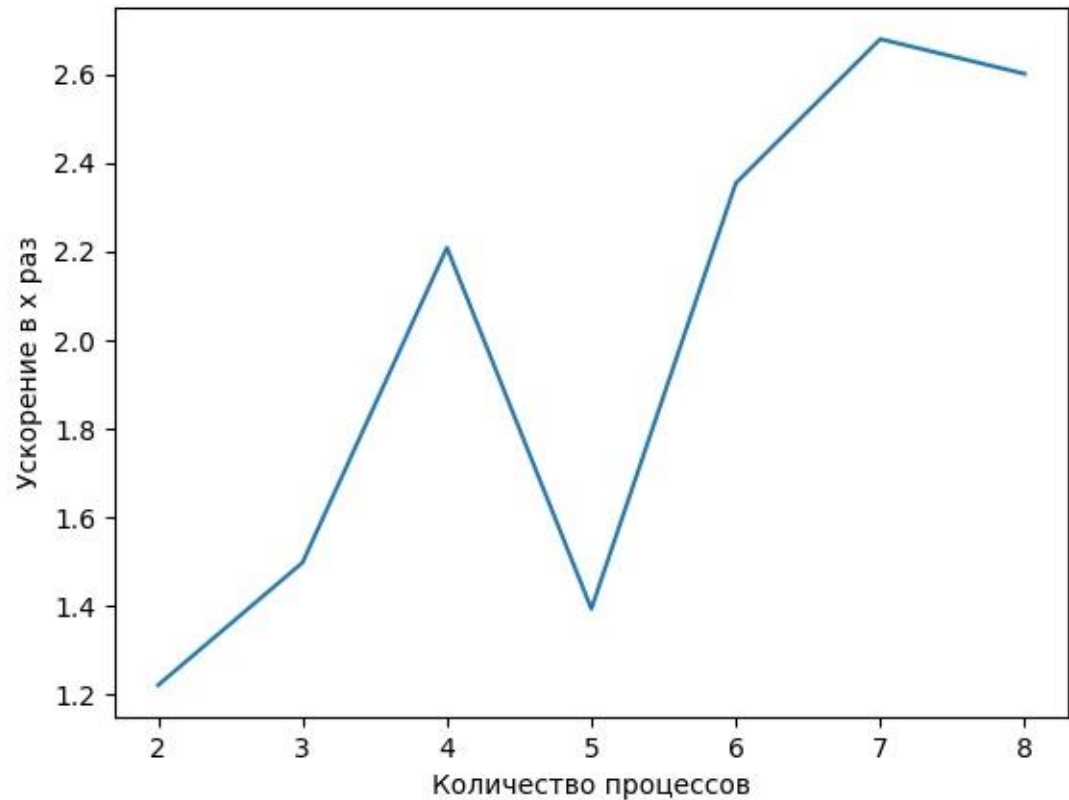
Использую библиотеку joblib

```
start = time()
result = Parallel(n_jobs=job)(delayed(dynamic_parallel_backpack)(d) for d in test_files)
delta = time() - start
```

Здесь test\_files - это набор параметров, которые мы передаем в функцию



### 1) Случай для 26 тестов



#### Комментарий:

Так как программа и так быстро считает столько тестов (1.36 секунд). Параллельные вычисления не особо эффективно решают данную задачу.

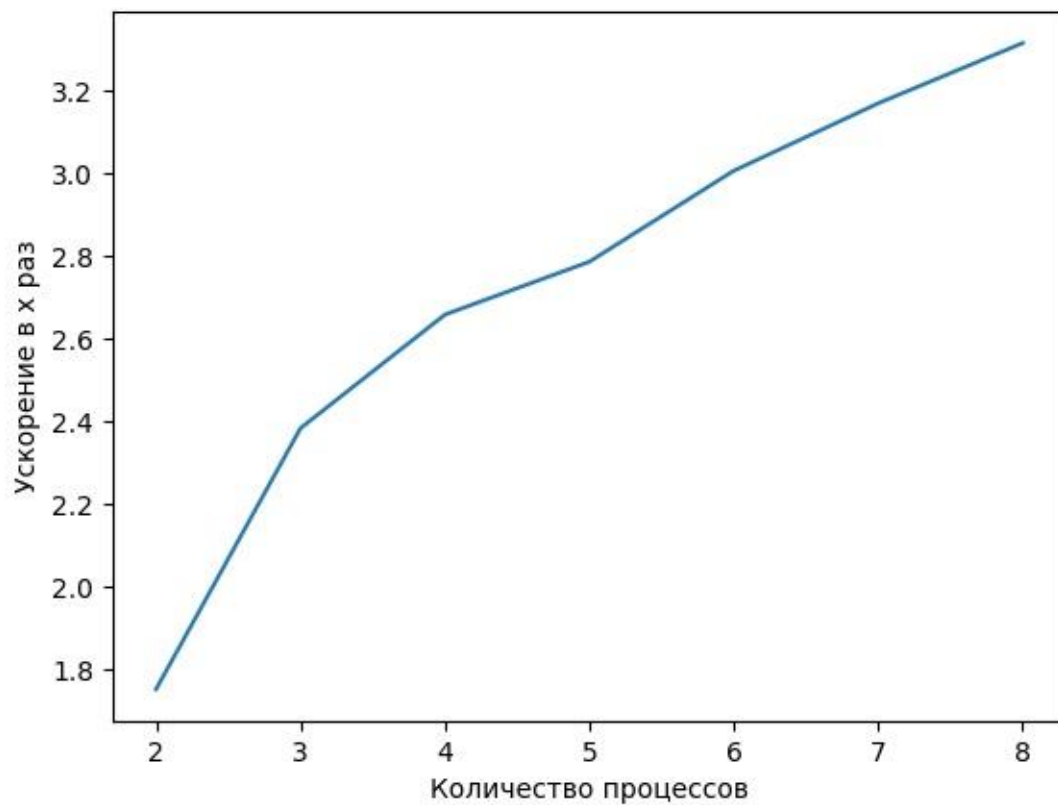
#### Время в зависимости от процессов (нумерация от 1 до 8):

```
[1.3630380630493164, 1.0842390060424805, 0.9293453693389893, 0.6157541275024414, 0.9534847736358643, 0.6425454616546631, 0.5785915851593018, 0.5643596649169922]
```

#### Ускорение:

```
[1.2571380068906253, 1.4666647169273546, 2.2136076758071135, 1.4295331197075418, 2.121309921852476, 2.355786184954687, 2.415193976078706]
```

## 2) Случай 50 тестов



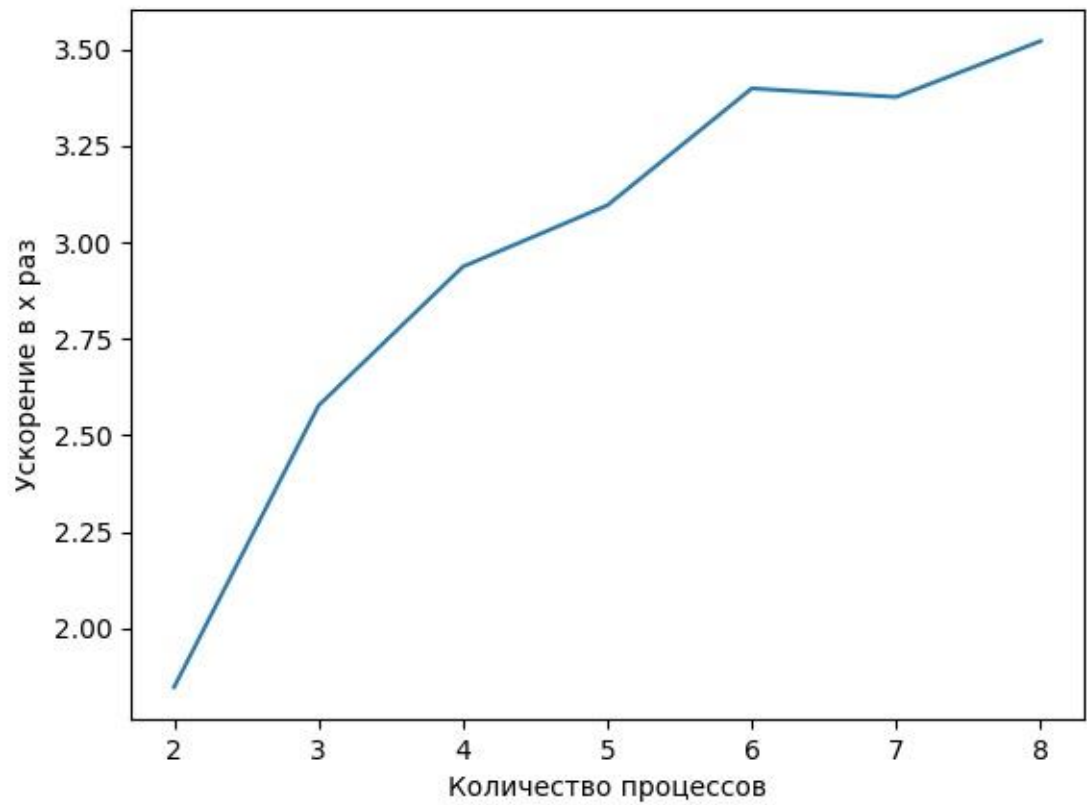
### Время

```
[13.85809063911438, 7.903631687164307, 5.811343669891357, 5.210963487625122, 4.9717583656311035, 4.607759714126587, 4.370750427246094, 4.17869758605957]
```

### Ускорение

```
[1.7533826458057582, 2.3846620379574723, 2.659410428037782, 2.7873620598524935, 3.0075549722412593, 3.170643318531009, 3.3163660096739105]
```

### 3) Случай 100 тестов



Время

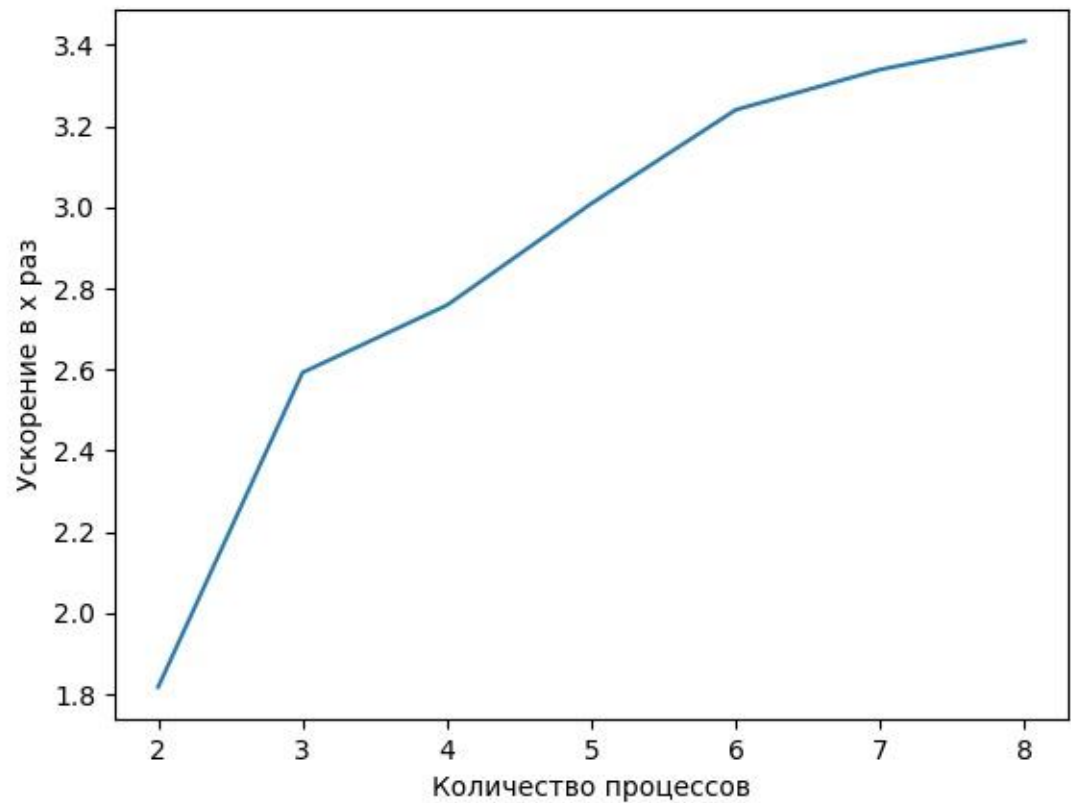
```
[122.93346619606018, 64.56454873085022, 46.88298773765564, 40.08579397201538, 40.586421966552734, 40.128251791000366, 41.71762490272522, 40.268686056137085]
```

Ускорение

```
[1.9040397340734472, 2.622133787291079, 3.066758919179255, 3.0289308650407674, 3.0635141255675307, 2.946799260089936, 3.052830331356806]
```



#### 4) Случай 200 тестов



Время:

```
[1033.6492791175842, 568.6699864864349, 398.6573054790497, 374.67862153053284, 343.4703767299652, 318.9902672767639, 309.51445603370667, 303.1466062068939]
```

Ускорение

```
[1.817661040112306, 2.5928266330789835, 2.758762362515395, 3.00942773859719, 3.240378736134806, 3.3395832051379792, 3.40973396354677]
```

Проверка решений:

```
OK
OK
OK
OK
OK
OK
OK
OK
OK
OK
OK
OK
OK
OK
```

#### **4. Что не получилось**

- 1) Как распараллелить рекурсию
- 2) Библиотека multiprocessing
- 3) Numba p.range

## 5. Список литературы.

<https://ru.wikipedia.org/wiki/%D0%97%D0%B0%D0%B4%D0%B0%D1%87%D0%B0%D0%BE%D1%80%D1%8E%D0%BA%D0%B7%D0%B0%D0%BA%D0%B5>

<https://proglib.io/p/python-i-dinamicheskoe-programmirovani-na-primere-zadachi-o-ryukzake-2020-02-04>

<https://dev.to/downey/solving-the-knapsack-problem-with-dynamic-programming-4hce>

<https://hal.archives-ouvertes.fr/hal-01152223/document>