МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭКОНОМИЧЕСКИЙ УНИВЕРСИТЕТ»

Кафедра прикладной математики и
экономико-математических методов

ОТЧЕТ
по дисциплине «Методы оптимизации»

студента Дудниченко Максима Юрьевича
(Ф.И.О. полностью)
Курс 4 Группа Э-1715
Форма обучения очная

Форма представления на кафедру выполненных заданий:
отчет в электронной форме

Оценка по результатам текущего
контроля (КТ3)

_____ _____
(подпись преподавателя)

Санкт-Петербург
2020 г.

В [1]:

```
import random
import numpy as np
import pandas as pd
from functools import lru_cache

import pulp
from scipy.optimize import linprog

import requests
import networkx as nx

import tensorflow as tf

from IPython.display import Image
import plotly.express as px
import plotly.graph_objects as go
import plotly.io as pio
pio.templates.default = "plotly_white"
%matplotlib notebook
```

1 Методы одномерной оптимизации

Методы сравнивались по двум основным метрикам:

1. Число итераций
2. Количество вызовов функции

Для методов, использующих производные, вызов производной для простоты считался как один вызов функции.

Все методы сравнивались в одинаковых условиях:

1. Функция $x^4 + 4x^3 + 7x^2 + 2x + 1$
2. Начальный промежуток $[-3, 2]$
3. Точность $[10^{-1}, 10^{-2}, \ldots, 10^{-8}]$

Для удобства функции возвращают не оптимальное значение, а число итераций

В [2]:

```
# Каждый вызов функции увеличивает счетчик на 1
def func(x):
    func.n_calls += 1
    return x**4 + 4*x**3 + 7*x**2 + 2*x + 1
```

В [3]:

```
def der(x):
    func.n_calls += 1
    return 4*x**3 + 12*x**2 + 14*x + 2
```

В [4]:

```
def der2(x):
    func.n_calls += 1
    return 12*x**2 + 24*x + 14
```

В [5]:

```
a = -3
b = 2
epsilon_list = [0.1**i for i in range(1, 9)]
```

Для более удобного подсчета числа вызовов функции был использован декоратор

В [6]:

```
def count_calls(function):
    def wrapper(*args):
        # Обнуляем счетчик
        func.n_calls = 0
        # Возвращаем результат выполнения функции
        # вместе с числом вызовов функции
        return function(*args), func.n_calls
    return wrapper
```

Метод перебора значений

Самый простой метод оптимизации:

1. В заданном промежутке равномерно расставляем точки
2. Для каждой из них вычисляем значение функции
3. Выбираем точку с максимальным / минимальным значениям

К достоинствам метода можно отнести то, что он подходит для любых функций (негладких, с разрывами, ...)

К недостаткам:

1. Требуемое число вызовов функции гораздо больше, чем для других методов
2. При увеличении размерности число вызовов возрастает экспоненциально

В [7]:

```
@count_calls
def brute_force(func, a, b, epsilon):
    # Вычисляем необходимое число точек
    n = (b-a)/epsilon
    n = np.floor(n)
    # Начинаем в точке a
    current_x = a
    # Записываем лучшие значения
    best_x = a
    best_y = func(a)
    while current_x < b:
        # Переходим в следующую точку
        current_x += (b-a)/n
        current_y = func(current_x)
        # Проверяем, лучше ли новая точка
        if current_y < best_y:
            best_y = current_y
            best_x = current_x

    return n
```

В [8]:

```
brute_force(func, a, b, 0.1)
```

Out[8]:

```
(50.0, 51)
```

Метод перебора не был включен в итоговое сравнение, поскольку число итераций гораздо больше, чем у остальных методов.

Метод бинарного поиска

Если известно, что функция обладает свойством унимодальности, можно существенно уменьшить число вычислений.

Метод двоичного поиска:

1. Проверить, возрастает ли функция в середине промежутка
    A. Если да: убрать правую половину
    B. Если нет: убрать левую половину
2. Проверить, не достигнута ли требуемая точность
3. Перейти к шагу 1.

В [9]:

```
@count_calls
def binary_search(func, a, b, epsilon):
    i = 0
    while abs(a - b) > epsilon:
        i += 1
        delta = epsilon / 2
        x1 = (a + b - delta) / 2
        x2 = (a + b + delta) / 2
        if func(x1) < func(x2):
            b = x2
        else:
            a = x1
    return i
```

В [10]:

```
binary_result = [binary_search(func, a, b, epsilon)
                 for epsilon in epsilon_list]
binary_iterations = [t[0] for t in binary_result]
binary_calls = [t[1] for t in binary_result]
```

Метод секущих

Метод основан на методе касательных и приближенном вычислении производной:

$$f'(x^k) \approx \frac{f(x^k) - f(x^{k-1})}{x^k - x^{k-1}}$$

Получаем формулу для итерационного процесса:

$$x^{k+1} = x^k - f(x^k)\frac{x^k - x^{k-1}}{f(x^k) - f(x^{k-1})}$$

В [11]:

```
@count_calls
def secant_method(func, der, a, b, epsilon):
    x_prev, x_curr = a, b
    i = 0
    while abs(der_x_curr := der(x_curr)) > epsilon:
        x_prev, x_curr = \
        x_curr, x_curr - (der_x_curr*(x_curr-x_prev)) / (der_x_curr-der(x_prev))
        i += 1
    return i
```

В [12]:

```
secant_result = [secant_method(func, der, a, b, epsilon)
                 for epsilon in epsilon_list]
secant_iterations = [t[0] for t in secant_result]
secant_calls = [t[1] for t in secant_result]
```

Метод Ньютона

В [13]:

```
@count_calls
def Newton_method(func, der, der2, a, b, epsilon):
    x_prev, x_curr = a, b
    i = 0
    while abs( der_x_curr := der(x_curr)) > epsilon:
        x_prev, x_curr = x_curr, x_curr - der_x_curr / der2(x_curr)
        i += 1
    return i
```

В [14]:

```
Newton_result = [Newton_method(func, der, der2, a, b, epsilon)
                 for epsilon in epsilon_list]
Newton_iterations = [t[0] for t in Newton_result]
Newton_calls = [t[1] for t in Newton_result]
```

Золотое сечение

В [15]:

```
@count_calls
def golden_ratio(func, a, b, epsilon):
    T = (3 - 5**0.5) / 2
    x1 = a + (b-a)*T
    x2 = a + b - x1
    f1 = func(x1)
    f2 = func(x2)
    i = 0
    while abs(a - b) > epsilon:
        if f1 <= f2:
            b = x2
            x2 = x1
            f2 = f1
            x1 = a + b - x2
            f1 = func(x1)
        else:
            a = x1
            x1 = x2
            f1 = f2
            x2 = a + b - x1
            f2 = func(x2)
        i += 1

    return i
```

В [16]:

```
golden_result = [golden_ratio(func, a, b, epsilon)
                 for epsilon in epsilon_list]
golden_iterations = [t[0] for t in golden_result]
golden_calls = [t[1] for t in golden_result]
```

Метод Фибоначчи

В [17]:

```
# Вспомогательная функция
# Возвращает список чисел Фибоначчи, которые <= n
def fibonacci_list(n):
    fib = [0, 1]
    while fib[-1] < n:
        fib.append( fib[-2] + fib[-1] )
    return fib
```

В [18]:

```
@count_calls
def fibonacci(func, a, b, epsilon):
    F = fibonacci_list( np.floor((b-a)/epsilon) )
    n = len(F) - 1
    lambda_ = a + (b-a) * F[n-2] / F[n]
    mu = a + (b-a) * F[n-1] / F[n]
    k = 1
    while k != n - 2:
        if func(lambda_) > func(mu):
            a = lambda_
            lambda_ = mu
            mu = a + (b-a) * F[n - k - 1] / F[n - k]
        else:
            b = mu
            mu = lambda_
            lambda_ = a + (b-a) * F[n - k - 2] / F[n - k]
        k += 1
    return k
```

В [19]:

```
fibonacci_result = [fibonacci(func, a, b, epsilon)
                    for epsilon in epsilon_list]
fibonacci_iterations = [t[0] for t in fibonacci_result]
fibonacci_calls = [t[1] for t in fibonacci_result]
```

Итоговое сравнение

В [20]:

```
names = ['Золотое сечение', 'Фибоначчи',
         'Бинарный', 'Метод секущих', 'Метод Ньютона']
iterations = [golden_iterations, fibonacci_iterations,
              binary_iterations, secant_iterations, Newton_iterations]
calls = [golden_calls, fibonacci_calls,
         binary_calls, secant_calls, Newton_calls]
```

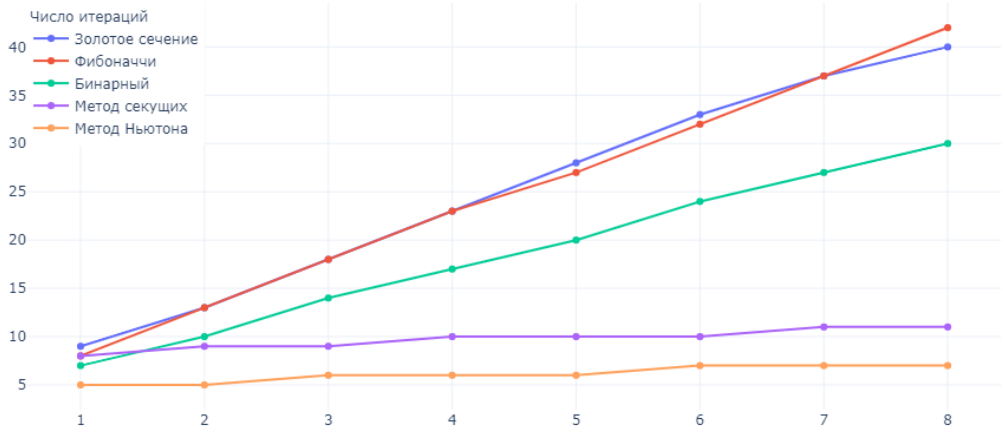
В [21]:

```
fig = go.Figure()
for name, iteration in zip(names, iterations):
    fig.add_trace(go.Scatter(x=[i for i in range(1, 9)],
                             y=iteration,
                             mode='lines+markers',
                             name=name))

fig.update_layout(legend=dict(
    xanchor="left", yanchor="top",
    x=0, y=1, title_text='Число итераций'
))

fig.show()
```

В [22]:

```
Image('n_iterations.png')
```

Out[22]:



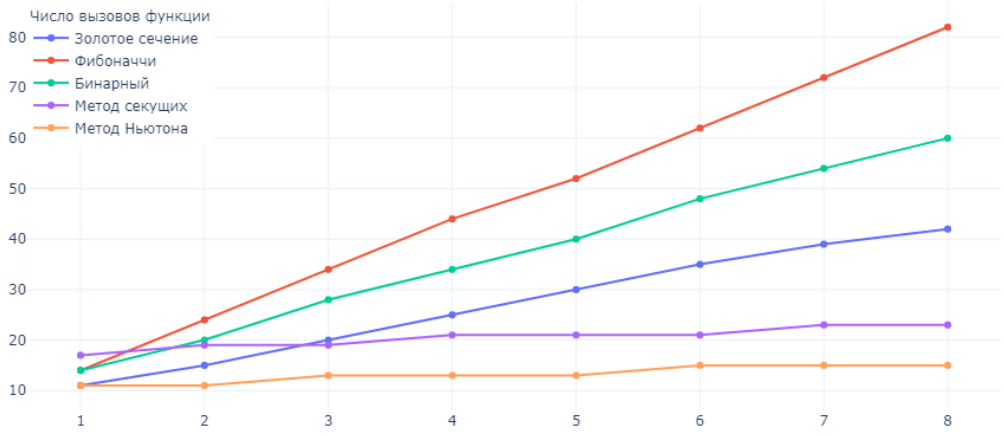
В [23]:

```
fig = go.Figure()
for name, call in zip(names, calls):
    fig.add_trace(go.Scatter(x=[i for i in range(1, 9)],
                             y=call,
                             mode='lines+markers',
                             name=name))

fig.update_layout(legend=dict(
    xanchor="left", yanchor="top",
    x=0, y=1, title_text='Число вызовов функции'
))

fig.show()
```

В [24]:

```
Image('n_calls.png')
```

Out[24]:



Методы многомерной оптимизации

Градиентный спуск

Метод основан на свойстве градиента функции: он показывает направление наискорейшего возрастания функции.

Алгоритм:

1. Начать в случайной точке
2. Вычислить градиент функции

3. Сделать шаг в этом направлении
4. Перейти к шагу 2.

В [25]:

```
# Функция для теста
def f(x):
    return 2*x[0]**2 + 5*x[1]**2 + x[0]*x[1] - 3*x[0] - x[1] + 5
```

В [26]:

```
# Градиент функции
def grad(x):
    return np.array([4*x[0] + x[1] - 3, x[0] + 10*x[1] - 1])
```

В [27]:

```
def gradient_descent(grad, x0=[2,2], ALPHA=0.05, epsilon=0.001):
    x = np.array(x0)
    history = [x]
    for i in range(10):
        # Вычисляем градиент
        gradient = grad(x)
        if np.linalg.norm(gradient) <= epsilon:
            break
        # Делаем шаг в сторону улучшения
        x = x - gradient * ALPHA
        history.append(x)
    return history
```

В [28]:

```
history = gradient_descent(grad)
```

В [29]:

```
xx = [i[0] for i in history]
yy = [i[1] for i in history]
zz = [f(i) for i in history]
```

Визуализация

B [30]:

```
N = 20
X_MIN = np.min(xx) - 0.5
X_MAX = np.max(xx) + 0.5
x = np.linspace(X_MIN, X_MAX, N)

Y_MIN = np.min(yy) - 0.5
Y_MAX = np.max(yy) + 0.5
y = np.linspace(Y_MIN, Y_MAX, N)

z = [[f([x,y]) for x in x] for y in y]
z = np.array(z)
```

B [31]:

```
blue_gradient = [(0, 'blue'), (1, 'cyan')]
surface = go.Surface(x=x, y=y, z=z,
                     opacity=0.25,
                     showscale=False,
                     colorscale=blue_gradient)

scatter = go.Scatter3d(x=xx, y=yy, z=zz,
                       mode='markers',
                       marker_color='black',
                       marker_size=3,
                       opacity=1)
```

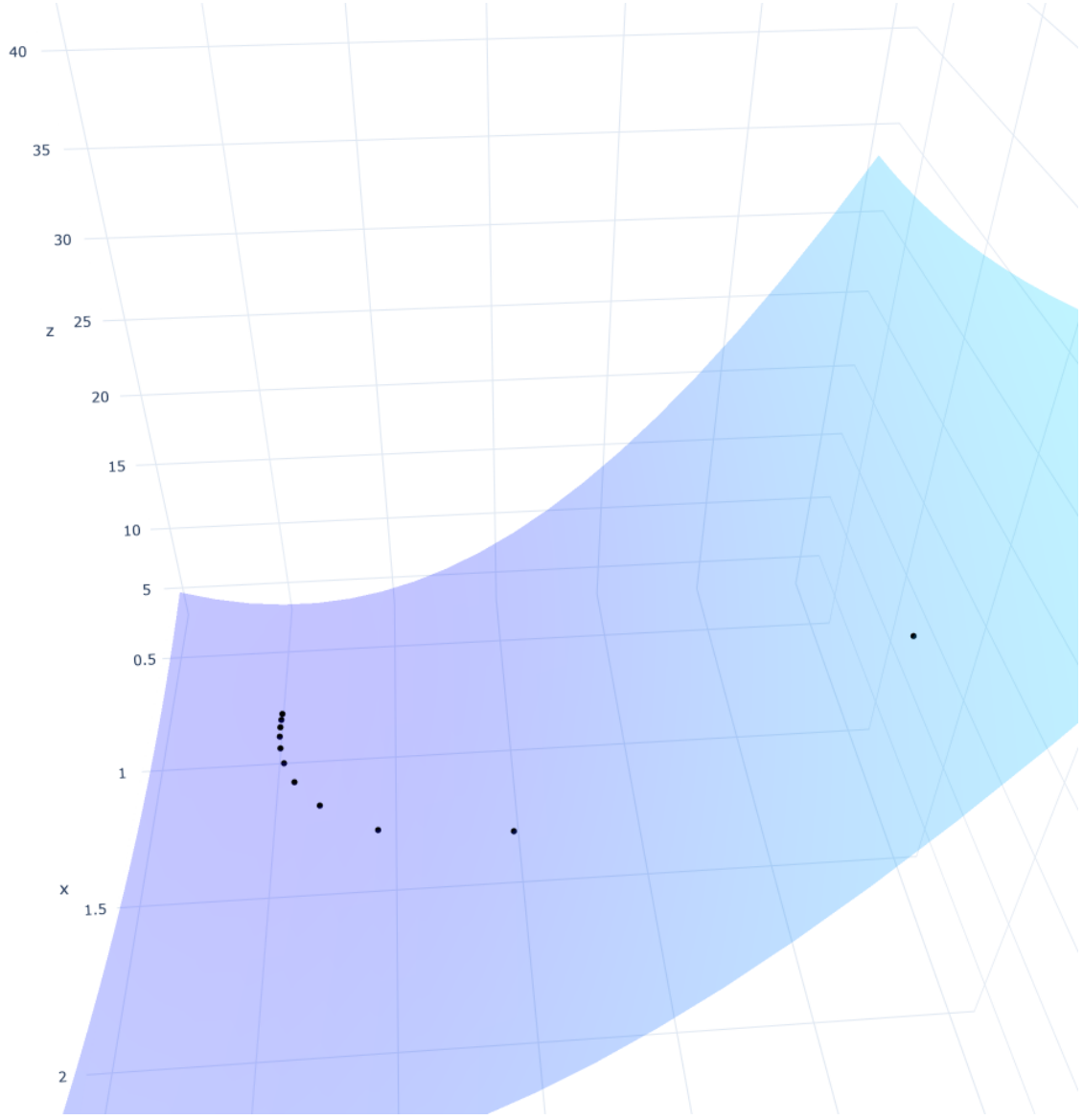
B [32]:

```
fig = go.Figure(data=[surface,scatter])
fig.update_layout(height=950, width=950)
fig.update_layout(margin=dict(l=0, r=0, b=0, t=0))
fig.show()
```

B [33]:

```
Image('gradient_descent.png')
```

Out[33]:



Градиентный спуск с автоматическим взятием производной

Библиотека TensorFlow используется для обучения нейронных сетей, но также может использоваться для взятия производных.

Представлена реализация метода градиентного спуска с использованием **tf.GradientTape**

В [34]:

```
# Создаем переменную
x = tf.Variable([1.0, 1.0])
alpha = tf.constant(0.05)

for i in range(10):
    with tf.GradientTape() as g:
        g.watch(x)
        y = 2*x[0]**2 + 5*x[1]**2 + x[0]*x[1] - 3*x[0] - x[1] + 5
        # Вычисляем производную
        dy_dx = g.gradient(y, x)
    # Делаем шаг в направлении
    x = x - dy_dx * alpha
```

Метод Хука-Дживса

Метод не использует производную функции

В [35]:

```
def conf_func(x):
    return 2*x[0]**2 + 5*x[1]**2 + x[0]*x[1] - 3*x[0] - x[1] + 5
```

В [36]:

```
def configurations(func,
                   N_VARS=2,
                   h=0.5,
                   lambda_=0.2,
                   x_start=[5,-5],
                   n_iterations=5):

    x_curr = np.array(x_start)
    points = []

    for _ in range(n_iterations):
        points.append(x_curr)

        # Улучшилось ли?
        got_better = False

        # Для каждой переменной
        for curr_var in range(N_VARS):

            # Первую точку записываем
            if curr_var == 0:
                x_prev = x_curr

            step = np.zeros(N_VARS)
            step[curr_var] = h
            # Вперед
            if func(x_curr+step) < func(x_curr):
                x_next = x_curr + step
                x_curr = x_next
                got_better = True
            # Назад
            elif func(x_curr-step) < func(x_curr):
                x_next = x_curr - step
                x_curr = x_next
                got_better = True

            points.append(x_curr)

        # Если ни по одной переменной не улучшилось
        if not got_better:
            h /= 2

        points.append(x_curr)
        # Делаем шаг в сторону улучшения
        x_curr = x_curr + lambda_*(x_curr - x_prev)
        points.append(x_curr)

    return points
```

B [37]:

```
points = configurations(conf_func)
xx = [point[0] for point in points]
yy = [point[1] for point in points]
```

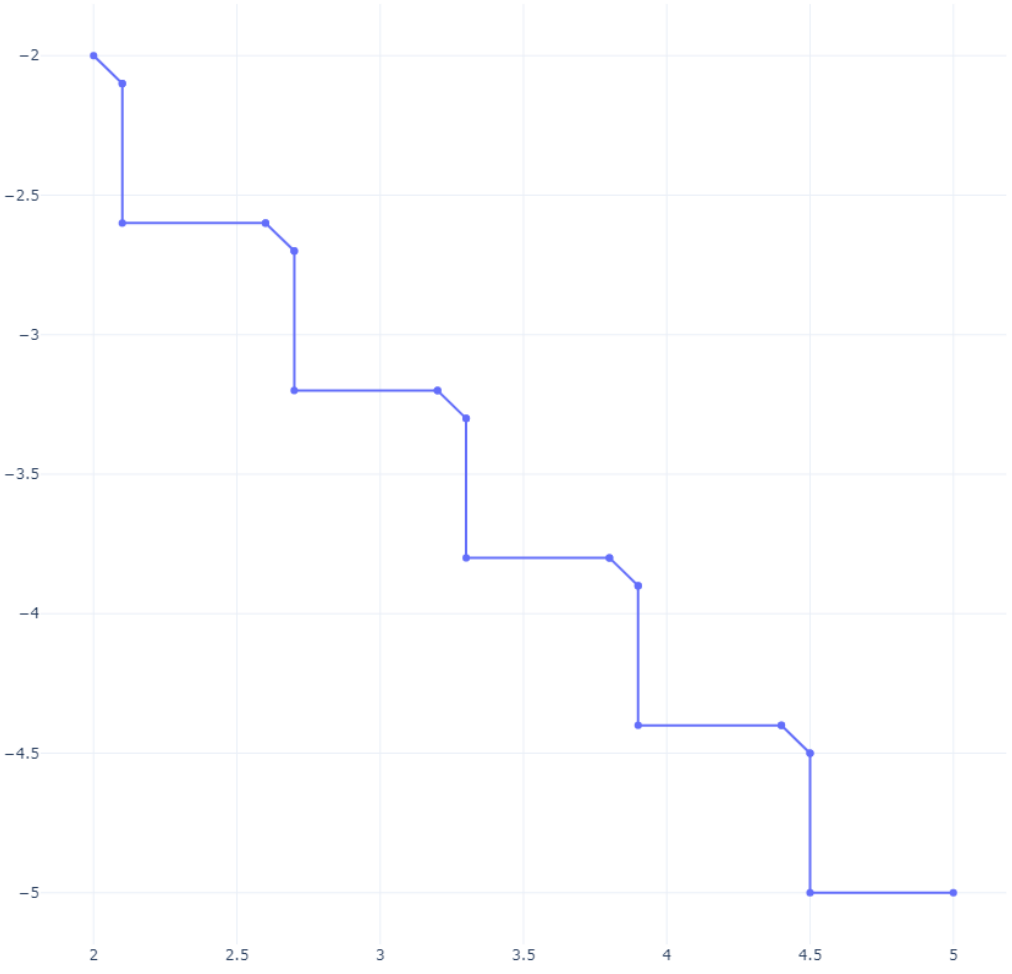
B [38]:

```
fig = go.Figure(go.Scatter(
    x=xx,
    y=yy,
    mode='lines+markers'
))
fig.update_layout(height=950, width=950)
fig.show()
```

B [39]:

```
Image('Hooke_Jeeves.png')
```

Out[39]:



Метод наискорейшего спуска

Одна из проблем градиентного спуска - выбор параметра α

При слишком высоких значениях можно пропустить оптимум.

При слишком низких потребуется больше вычислений.

Метод наискорейшего спуска делает α переменной.

B [40]:

```
def f(x):
    return 2*x[0]**2 + 5*x[1]**2 + x[0]*x[1] - 3*x[0] - x[1] + 5

def grad(x):
    return np.array([4*x[0] + x[1] - 3, x[0] + 10*x[1] - 1])
```

B [41]:

```
epsilon = 0.01
alpha_max = 2
```

B [42]:

```
x = np.array([-5, 5])
points = []
candidates = []

while np.linalg.norm( grad_x := grad(x) ) > epsilon:

    # <Бинарный поиск>
    phi = lambda alpha: f(x - alpha * grad_x)
    a = 0
    b = alpha_max

    # while abs(a - b) > epsilon:
    for _ in range(10):
        delta = epsilon / 2
        x1 = (a + b - delta) / 2
        x2 = (a + b + delta) / 2
        points.append(x)
        candidates.append(x - grad_x*(a+b)/2)
        if phi(x1) < phi(x2):
            b = x2
        else:
            a = x1
    alpha_star = (a+b)/2
    # </Бинарный поиск>

    x = x - alpha_star * grad_x
```

Линейное программирование

Были использованы две реализации:

1. Функция linprog из библиотеки scipy.optimize
2. Библиотека pulp

linprog принимает аргументы:

1. c - вектор коэффициентов целевой функции ($\rightarrow max$)
2. A_ub, c_ub - коэффициенты ограничений ($\leq$)
3. A_eq, c_eq - коэффициенты ограничений (равенство)

Библиотека pulp удобнее, поскольку ограничения и целевую функцию можно записать в более удобном виде.

Витамины

Необходимо выбрать компоненты так, чтобы число активных веществ попадало в заданый диапазон, а вес при этом не превышал лимит.

Цель - минимизация стоимости.

В [43]:

```
A_ub = [
    [-0.2, -0.1, 0, -0.2],
    [0.2, 0.1, 0, 0.2],

    [-0.34, -0.25, -0.07, 0],
    [0.34, 0.25, 0.07, 0],

    [-0.02, -0.03, -0.28, -0.05],
    [0.02, 0.03, 0.28, 0.05],

    [-0.0008, 0, -0.0014, -0.003],
    [0.0008, 0, 0.0014, 0.003],

    [1, 1, 1, 1]
]
```

В [44]:

```
b_ub = [-0.028, 0.03, -0.054, 0.06, -0.036, 0.04, -0.00033, 0.00035, 0.28]
```

В [45]:

```
c = [0.8, 0.35, 0.5, 0.3]
```

В [46]:

```
solution = linprog(c, A_ub, b_ub, method='simplex')
```

В [47]:

```
# Решение
solution.x
```

Out[47]:

```
array([0.12309091, 0.01418182, 0.12290909, 0.01981818])
```

Доставка

Решение транспортной задачи с использованием linprog не очень удобно, поскольку необходимо переводить матрицы в векторы.

В [48]:

```
c = -np.array([
    345,340,360,360,350,355,335,340,
    335,360,355,355,345,345,350,355,
    350,340,340,345,350,345,350,345,
    350,335,350,340,360,360,365,360
])
```

В [49]:

```
b_ub = [45,78,63,62]
b_eq = [26,14,28,17,13,18,34,54]
```

В [50]:

```
def ones_column(i):
    a = np.zeros([4, 8])
    a[:,i] = np.ones(4)
    return a.flatten()
```

В [51]:

```
def ones_row(i):
    a = np.zeros([4, 8])
    a[i,:] = np.ones(8)
    return a.flatten()
```

В [52]:

```
A_ub = [ones_row(i) for i in range(4)]
A_eq = [ones_column(i) for i in range(8)]
```

В [53]:

```
solution = linprog(c,
                   A_ub=A_ub, b_ub=b_ub,
                   A_eq=A_eq, b_eq=b_eq,
                   method='simplex')
```

В [54]:

```
solution.x
```

Out[54]:

```
array([ 0.,  0., 28., 17.,  0.,  0.,  0.,  0.,  0., 14.,  0.,  0.,
0.,
        0.,  0., 54., 26.,  0.,  0.,  0.,  3.,  0.,  0.,  0.,  0.,
0.,
        0.,  0., 10., 18., 34.,  0.])
```

В [55]:

```
solution.fun
```

Out[55]:

```
-73050.0
```

Задача об аудиторах

Необходимо назначить аудиторов для заказов, при этом опыт работы в разных сферах разный.

Требуется найти распределение, минимизирующее время на подготовку.

Если аудитор не может быть назначен на заказ, считаем затраты очень большими.

В [56]:

```
no = 10**6
```

В [57]:

```
C = [
    [ 8, 21, 15, 13,  9, 17, 18,  7, 26,  9],
    [14, 18, 17, 19, 12,  6, no, 15, 24, 13],
    [ 9, 15, 18, 16, 16, 15, 11, 13, 21, 19],
    [11, no, 14,  7, 23,  9,  6, 18, no,  7],
]
```

В [58]:

```
needs = [4, 9, 2, 12, 7, 6, 9, 3, 18, 5]
employees = [35, 20, 25, 10]
```

В [59]:

```
# Создаем модель, tf -> min
model = pulp.LpProblem('Accounting', pulp.LpMinimize)
```

В [60]:

```
# Переменные задачи:
# Сколько сотрудников из i конторы назначено на заявку j
X = [[pulp.LpVariable(f'x_{i}_{j}',
                      lowBound=0,
                      cat='Integer')
      for i in range(10)]
      for j in range(4)]
```

В [61]:

```
# Целевая функция: сумма затрат*количество
model += sum(X[i][j]*C[i][j] for i in range(4) for j in range(10))
```

В [62]:

```
# Сумма по строкам <= числу сотрудников
for i in range(4):
    model += sum(X[i]) <= employees[i]

# Сумма по столбцам == числу заявок
for j in range(10):
    model += sum(X[i][j] for i in range(4)) == needs[j]
```

В [63]:

```
model.solve();
```

В [64]:

```
pulp.LpStatus[model.status]
```

Out[64]:

```
'Optimal'
```

В [65]:

```
# Значение целевой функции
print(pulp.value(model.objective))
```

```
950.0
```

В [66]:

```
# Ответ
answer = [[X[i][j].varValue for j in range(10)] for i in range(4)]
answer
```

Out[66]:

```
[[4.0, 0.0, 2.0, 11.0, 7.0, 0.0, 0.0, 3.0, 0.0, 5.0],
 [0.0, 0.0, 0.0, 0.0, 0.0, 6.0, 0.0, 0.0, 2.0, 0.0],
 [0.0, 9.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 16.0, 0.0],
 [0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 9.0, 0.0, 0.0, 0.0]]
```

Задача об аудиторах с дополнительным ограничением

Дополнительное ограничение: **Не назначать компаниям аудиторов только из одной конторы**

В [67]:

```
# Создаем модель, tf -> min
model = pulp.LpProblem('Accounting2', pulp.LpMinimize)
```

В [68]:

```
# Переменные задачи:
# Сколько сотрудников из i конторы назначено на заявку j
X = [[pulp.LpVariable(f'x_{i}_{j}',
                      lowBound=0,
                      cat='Integer')
      for i in range(10)] for j in range(4)]
```

В [69]:

```
# Целевая функция: сумма затрат*количество
model += sum(X[i][j]*C[i][j] for i in range(4) for j in range(10))
```

В [70]:

```
# Сумма по строкам <= числу сотрудников
for i in range(4):
    model += sum(X[i]) <= employees[i]

# Сумма по столбцам >= числу заявок
for j in range(10):
    model += sum(X[i][j] for i in range(4)) == needs[j]

# Не назначать компаниям аудиторов только из одной конторы
# То есть Xij < заявки[j] для всех i
for i in range(4):
    for j in range(10):
        model += X[i][j] <= needs[j] - 1
```

В [71]:

```
model.solve();
```

В [72]:

```
pulp.LpStatus[model.status]
```

Out[72]:

```
'Optimal'
```

В [73]:

```
# Значение целевой функции
# Из-за дополнительного ограничения оно больше
print(pulp.value(model.objective))
```

```
982.0
```

В [74]:

```
# Ответ
answer = [[X[i][j].varValue for j in range(10)] for i in range(4)]
answer
```

Out[74]:

```
[[3.0, 0.0, 1.0, 11.0, 6.0, 0.0, 0.0, 2.0, 0.0, 4.0],
 [0.0, 3.0, 1.0, 0.0, 1.0, 5.0, 0.0, 1.0, 1.0, 1.0],
 [1.0, 6.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 17.0, 0.0],
 [0.0, 0.0, 0.0, 1.0, 0.0, 1.0, 8.0, 0.0, 0.0, 0.0]]
```

Задача о назначениях

На каждый заказ необходимо 2 сотрудника: финансист и бухгалтер.

Необходимо составить пары таким образом, чтобы неприязнь в парах была минимальной.

В [75]:

```
C = [
    [ 3,  4,  9, 18,  9,  6],
    [16,  8, 12, 13, 20,  4],
    [ 8,  6, 13,  1,  6,  9],
    [16,  9,  6,  8,  1, 11],
    [ 8, 12, 17,  5,  3,  5],
    [ 2,  9,  1, 10,  5, 17]
]
```

В [76]:

```
model = pulp.LpProblem('Psychology', pulp.LpMinimize)
```

В [77]:

```
# Бинарные переменные
X = [[pulp.LpVariable(f'x_{i}_{j}',
                      lowBound=0,
                      upBound=1,
                      cat='Integer')
      for i in range(6)]
      for j in range(6)]
```

В [78]:

```
# Целевая функция
model += sum(X[j][i]*C[j][i] for i in range(6) for j in range(6))
```

В [79]:

```
for i in range(6):
    model += sum(X[i]) == 1
```

В [80]:

```
for j in range(6):
    model += sum(X[i][j] for i in range(6)) == 1
```

В [81]:

```
model.solve();
```

В [82]:

```
print(pulp.value(model.objective))
```

```
19.0
```

В [83]:

```
answer = [[X[i][j].varValue for i in range(6)] for j in range(6)]
answer
```

Out[83]:

```
[[1.0, 0.0, 0.0, 0.0, 0.0, 0.0],
 [0.0, 1.0, 0.0, 0.0, 0.0, 0.0],
 [0.0, 0.0, 0.0, 0.0, 0.0, 1.0],
 [0.0, 0.0, 1.0, 0.0, 0.0, 0.0],
 [0.0, 0.0, 0.0, 1.0, 0.0, 0.0],
 [0.0, 0.0, 0.0, 0.0, 1.0, 0.0]]
```

В [84]:

```
namesA = ['Red', 'Blue', 'Cyan', 'Black', 'Orange', 'Green']
namesB = ['Purple', 'Lime', 'White', 'Yellow', 'Pink', 'Brown']
```

В [85]:

```
# Вывод в более удобном для пользователя формате
for i in range(6):
    for j in range(6):
        if X[i][j].varValue == 1:
            print(f'Пара: {namesA[i]}\t{namesB[j]}')
```

```
Пара: Red       Purple
Пара: Blue      Lime
Пара: Cyan      Yellow
Пара: Black     Pink
Пара: Orange    Brown
Пара: Green     White
```

Дополнительное ограничение

Максимальная неприязнь друг к другу должна не превышать лимит.

В [86]:

```
model = pulp.LpProblem('Psychology2', pulp.LpMinimize)
```

В [87]:

```
X = [[pulp.LpVariable(f'x_{i}_{j}',
                      lowBound=0,
                      upBound=1,
                      cat='Integer')
      for i in range(6)]
      for j in range(6)]
```

В [88]:

```
model += sum(X[i][j]*C[i][j] for i in range(6) for j in range(6))
```

В [89]:

```
for i in range(6):
    model += sum(X[i]) == 1

for j in range(6):
    model += sum(X[i][j] for i in range(6)) == 1
```

В [90]:

```
# Дополнительное ограничение
for i in range(6):
    for j in range(6):
        model += X[i][j]*C[i][j] <= 7
```

В [91]:

```
model.solve();
```

В [92]:

```
pulp.LpStatus[model.status]
```

Out[92]:

```
'Optimal'
```

В [93]:

```
# Вывод в более удобном для пользователя формате
for i in range(6):
    for j in range(6):
        if X[i][j].varValue == 1:
            print(f'Пара: {namesA[i]}\t{namesB[j]}')
```

```
Пара: Red       Purple
Пара: Blue      Brown
Пара: Cyan      Lime
Пара: Black     Pink
Пара: Orange    Yellow
Пара: Green     White
```

В [94]:

```
answer = [[X[i][j].varValue for i in range(6)] for j in range(6)]
answer
```

Out[94]:

```
[[1.0, 0.0, 0.0, 0.0, 0.0, 0.0],
 [0.0, 0.0, 1.0, 0.0, 0.0, 0.0],
 [0.0, 0.0, 0.0, 0.0, 0.0, 1.0],
 [0.0, 0.0, 0.0, 0.0, 1.0, 0.0],
 [0.0, 0.0, 0.0, 1.0, 0.0, 0.0],
 [0.0, 1.0, 0.0, 0.0, 0.0, 0.0]]
```

Алгоритмы на графах

Матрица расстояний

Для того, чтобы узнать, в какие точки можно перейти ровно за **k** шагов, можно использовать логическое умножение матрицы на себя.

В [95]:

```
M = np.array([
    [0, 1, 1, 0],
    [1, 0, 0, 1],
    [1, 0, 0, 1],
    [0, 1, 1, 0]
])
```

В [96]:

```
def logic_multiply(M1, M2):
    n = len(M)
    M_new = np.zeros((n,n))
    for i in range(n):
        array1 = M1[i]
        for j in range(n):
            array2 = M2[:, j]
            M_new[i,j] = any(array1*array2)

    return M_new
```

В [97]:

```
logic_multiply(M, M)
```

Out[97]:

```
array([[1., 0., 0., 1.],
       [0., 1., 1., 0.],
       [0., 1., 1., 0.],
       [1., 0., 0., 1.]])
```

Для нескольких ходов используется возведение в степень.

В [98]:

```
def logic_power(M, k):
    M_new = M.copy()
    for _ in range(k-1):
        M_new = logic_multiply(M, M_new)
    return M_new
```

В [99]:

```
logic_power(M, 4)
```

Out[99]:

```
array([[1., 0., 0., 1.],
       [0., 1., 1., 0.],
       [0., 1., 1., 0.],
       [1., 0., 0., 1.]])
```

Чтобы узнать, куда можно попасть за **k** шагов или меньше, достаточно сложить полученные матрицы

В [100]:

```
sum(
    logic_power(M, i)
    for i in range(2, 4)
)
```

Out[100]:

```
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.]])
```

Минимальное остовное дерево

Задача: в взвешенном графе найти подграф с минимальной суммой весов ребер, покрывающий все вершины

Для данной задачи жадный алгоритм дает оптимальное решение

В [101]:

```
N = 20
# Генерируем случайную матрицу
matrix = np.random.randint(1, 100, [N,N])
matrix = matrix.astype('object')

for i in range(N):
    for j in range(N):
        if np.random.rand() < 0.5:
            matrix[i][j] = np.inf
```

Алгоритм Прима

В [102]:

```
def find_minimum_tree(matrix, start=0):
    V = len(matrix)
    edges = []
    total = 0
    # Список True/False
    is_covered = [False]*V
    is_covered[start] = True

    for _ in range(V - 1):

        # Находим минимальное расстояние и индесы
        # Между посещенными и непосещенными вершинами

        min_dist = np.inf
        # Для выходящих из посещенных
        for i in range(V):
            if is_covered[i]:

                # Для входящих в непосещенные
                for j in range(V):
                    if not is_covered[j]:

                        # Если дистанция меньше прошлых - обновить
                        if matrix[i][j] < min_dist:
                            min_from = i
                            min_to = j
                            min_dist = matrix[i][j]

        total += min_dist
        is_covered[min_to] = True
        edges.append((min_from, min_to))

    return edges, total
```

В [103]:

```
edges, total = find_minimum_tree(matrix)
total
```

Out[103]:

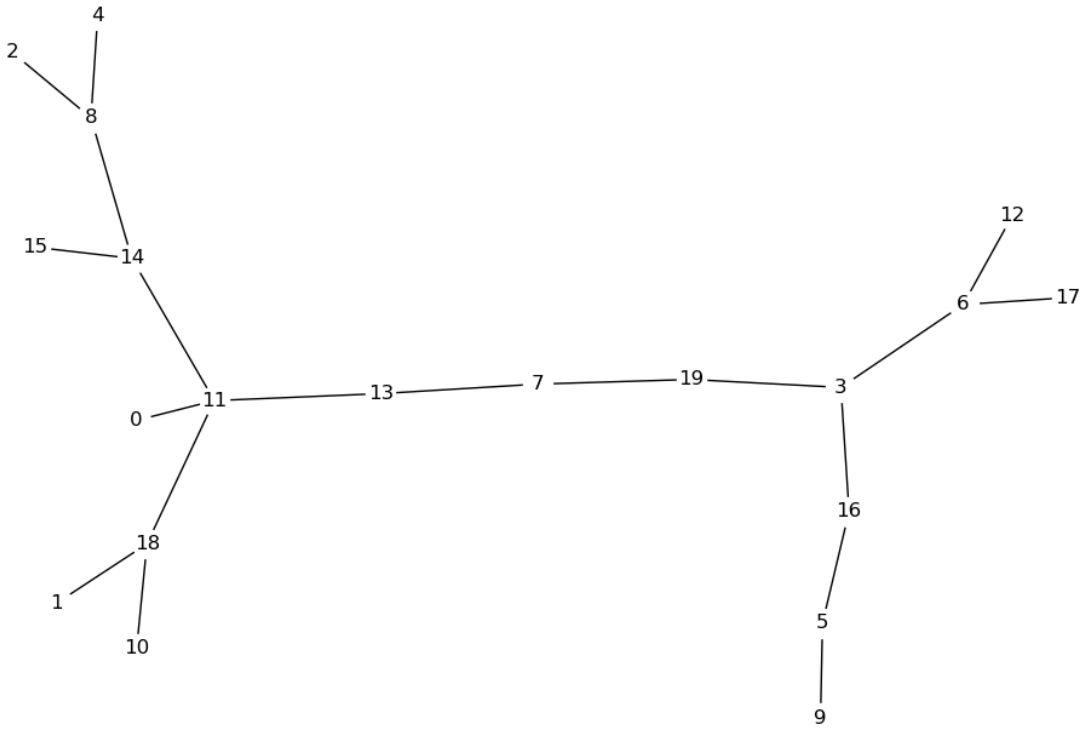
```
269
```

В [104]:

```
g = nx.Graph()
for i in range(N):
    for j in range(N):
        if (i,j) in edges:
            g.add_edge(i, j)
```

В [105]:

```
nx.draw(g, with_labels=True, node_color='white')
```



Алгоритм Краскала

В [106]:

```
# Переводим ребра в формат (вершина1, вершина2*, вес)
edges_Kraskal = [(i, j, matrix[i][j]) for i in range(N) for j in range(N)]
# Сортируем
edges_Kraskal = sorted(edges_Kraskal, key=lambda x: x[2])
```

В [107]:

```
def Kraskal(edges):
    # Покрытые вершины
    tree = set()
    # Использованные ребра
    edges = []
    total_weight = 0
    for node1, node2, weight in edges_Kraskal:
        # Если не создает цикл
        if not(node1 in tree and node2 in tree):
            # Добавляем ребро в дерево
            tree.add(node1); tree.add(node2)
            edges.append([node1, node2])
            total_weight += weight
    return edges, total
```

В [108]:

```
edges, total = Kraskal(edges_Kraskal)
```

В [109]:

```
total
```

Out[109]:

269

Алгоритм Дейкстры

В [110]:

```
inf = np.inf
def Dijkstra(matrix, start):
    # Число вершин
    V = len(matrix)
    # Дистанции
    dist = [inf] * V
    dist[start] = 0
    # Обработана ли вершина
    is_visited = [False] * V
    # Для записи путей
    prev = [None] * V


    # Для каждой вершины
    for _ in range(V):

        # Находим не посещенную вершину с минимальной дистанцией
        min_dist = inf
        for v in range(V):
            if dist[v] < min_dist and (not is_visited[v]):
                min_dist = dist[v]
                min_index = v

        # Обновляем пути, если возможно
        for v in range(V):
            alt_route = dist[min_index] + matrix[min_index][v]
            if (not is_visited[v]) and dist[v] > alt_route:
                dist[v] = alt_route
                prev[v] = min_index

        # Вычеркиваем вершину
        is_visited[min_index] = True

    return dist, prev
```

В [111]:

```
dist, prev = Dijkstra(matrix, 0)
```

В [112]:

```
dist
```

Out[112]:

```
[0, 38, 43, 16, 47, 61, 21, 34, 25, 51, 41, 15, 24, 19, 17, 37, 38,
29, 35, 35]
```

В [113]:

```
# Делаем путь
def path(prev, vertex):
    history = [vertex]
    while vertex is not None:
        vertex = prev[vertex]
        history.append(vertex)
    # Разворачиваем и убираем None
    return history[::-1][1::]

def pairs(path):
    return [(path[i], path[i+1]) for i in range(len(path)-1)]
```

В [114]:

```
shortest_path = path(prev, 8)
shortest_path
```

Out[114]:

```
[0, 11, 14, 8]
```

Муравьиный алгоритм для задачи коммивояжера

Муравьиный алгоритм основан на поведении колоний муравьев при поиске кратчайших путей

Основные шаги алгоритма:

1. Муравьи проходят по графу
    A. Выбирают следующую вершину в зависимости от расстояния и феромонов
    B. Посещенные вершины запоминаются и больше не выбираются
2. Муравьи наносят феромоны в зависимости от длины пути
3. Происходит обновление и испарение феромонов

Также существует множество модификаций алгоритма

В [115]:

```
# Число городов
N = 20
# Гиперпараметры
ALPHA = 0.3
BETA = 0.8
Q = 50
p = 0.5

cities_id = [i for i in range(N)]
```

Генерируем случайные данные

В [116]:

```
# Вычисляем видимость
distance_matrix = np.random.uniform(1, 100, [N,N])
np.fill_diagonal(distance_matrix, np.inf)
vision_matrix = 1 / distance_matrix
vision_matrix.shape
```

Out[116]:

```
(20, 20)
```

В [117]:

```
# Феромоны на первом шаге расставляем случайно
feromone_matrix = np.random.uniform(0.01, 0.02, (N, N))
np.fill_diagonal(feromone_matrix, 0)
feromone_matrix.shape
```

Out[117]:

```
(20, 20)
```

В [118]:

```
# Вспомогательная функция
# Переводит путь в список ребер
def path_to_edges(order):
    return [order[i:i+2] for i in range(len(order)-1)]
```

В [119]:

```
# Один проход муравья
def walk(start_point, vision_matrix, feromone_matrix):
    # Первая точка
    current_point = start_point
    # Можно ли перейти в вершину?
    allow_list = [True for i in range(N)]
    allow_list[current_point] = False
    # Порядок, в котором посещаем города
    order = [current_point]
    total_distance = 0

    # Пока не посетим все города
    # == пока будут вершины
    while sum(allow_list) != 0:
        # Выбираем видимость и феромоны
        # Если уже посещены, вероятность = 0
        current_point_vision = np.where(
            allow_list,
            vision_matrix[current_point], 0)
        current_point_feromones = np.where(
            allow_list,
            feromone_matrix[current_point], 0)
        # Вычисляем вероятности перехода
        probabilities = \
        (current_point_vision**ALPHA) * (current_point_feromones**BETA)
        probabilities = probabilities / np.sum(probabilities)
        # Выбираем следующий город
        next_city = np.random.choice(cities_id, p=probabilities)
        # Добавляем дистанцию
        total_distance += distance_matrix[current_point][next_city]
        # Переходим в точку, убираем ее из доступных, добавляем в список
        current_point = next_city
        allow_list[current_point] = False
        order.append(current_point)
    # Возвращаем длину пути и порядок
    return total_distance, order
```

В [120]:

```
# Колония муравьев
class AntColony:

    def __init__(self, vision_matrix, fenomone_matrix):
        self.vision_matrix = vision_matrix
        self.feromone_matrix = feromone_matrix
        # Записываем длины лучшего и среднего пути
        self.history_best = []
        self.history_mean = []

    # Вычисляет, сколько будет добавлено феромонов
    # Не модифицирует матрицу феромонов
    # Также собирает статистику
    def get_feromones(self):

        best_path = np.inf
        total_path = 0

        total_feromone = np.zeros([N, N])
        for i in range(N):

            start = i
            path_length, path = walk(start,
                                     self.vision_matrix,
                                     self.feromone_matrix)
            edges = path_to_edges(path)
            feromone_amount = Q / path_length

            for node1, node2 in edges:
                total_feromone[node1, node2] += feromone_amount

            total_path += path_length
            if path_length < best_path:
                best_path = path_length

        mean_path = total_path / N
        self.history_best.append(best_path)
        self.history_mean.append(mean_path)

        return total_feromone

    # Модифицирует список
    def update_feromones(self):
        # Моделируем испарение феромонов, добавляем сумму феромонов
        self.feromone_matrix = self.get_feromones() + \
                               self.feromone_matrix * p
```

В [121]:

```
# Создаем колонию
colony = AntColony(vision_matrix, feromone_matrix)
# Делаем 50 итераций
for _ in range(50):
    colony.update_feromones()
```

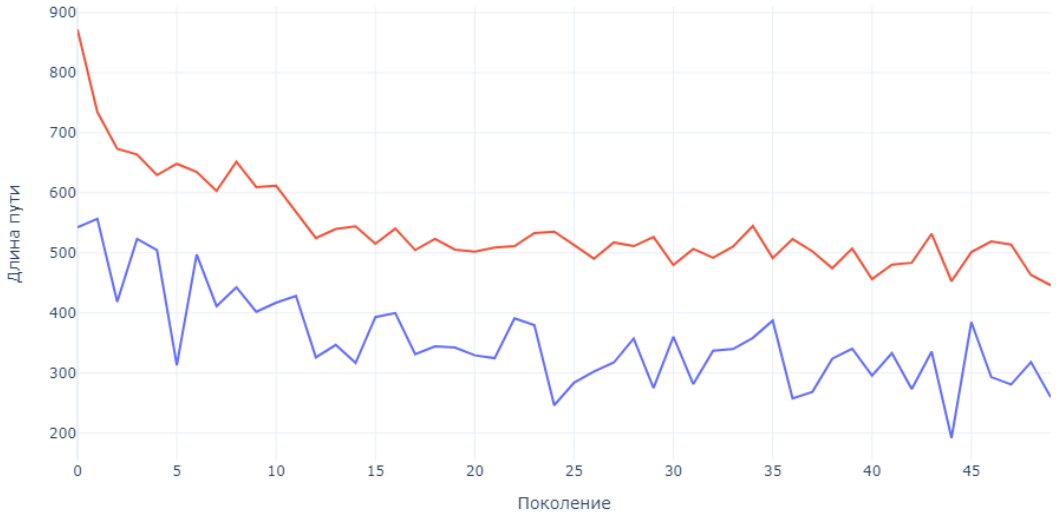
В [122]:

```
fig = px.line(y=[colony.history_best, colony.history_mean])
fig.update_layout(showlegend=False,
                  xaxis_title='Поколение',
                  yaxis_title='Длина пути')
fig.show()
```

В [123]:

```
Image('ants.png')
```

Out[123]:



Работа с графами социальных сетей

Одно из применений теории графов - анализ социальных сетей.

Рассмотрим построение графа друзей на примере ВКонтакте.

В [124]:

```
v = '5.126'
```

В [125]:

```
# Для доступа к API необходим токен
with open('token.txt') as file:
    token = file.read()
```

В [126]:

```
def create_request(user_id):
    return "https://api.vk.com/method/friends.get?&" + \
           f"user_id={user_id}&v={v}&access_token={token}"
```

В [127]:

```
# Получить список друзей
# Возвращает словарь {id: список id друзей}
def get_friends(user_id):
    try:
        r = requests.get(create_request(user_id))
        friends = r.json()['response']['items']
        return friends
    except:
        pass
```

В [136]:

```
friends = {f: get_friends(f) for f in get_friends(157624383)}
```

В [137]:

```
# Убираем None, означающий ошибку
friends = {k:v for k,v in friends.items() if (k is not None) and (v is not None)}
```

В [138]:

```
# Создаем граф
g = nx.from_dict_of_lists(friends)
```

В [139]:

```
# Граф получается очень большой
len(g)
```

Out[139]:

```
24152
```

В [140]:

```
# Чтобы сделать граф чуть меньше,
# уберем пользователей, у которых меньше 2 связей
deg = g.degree()
to_remove = [k for k,v in deg if v < 2]
```

В [141]:

```
g.remove_nodes_from(to_remove)
```

В [142]:

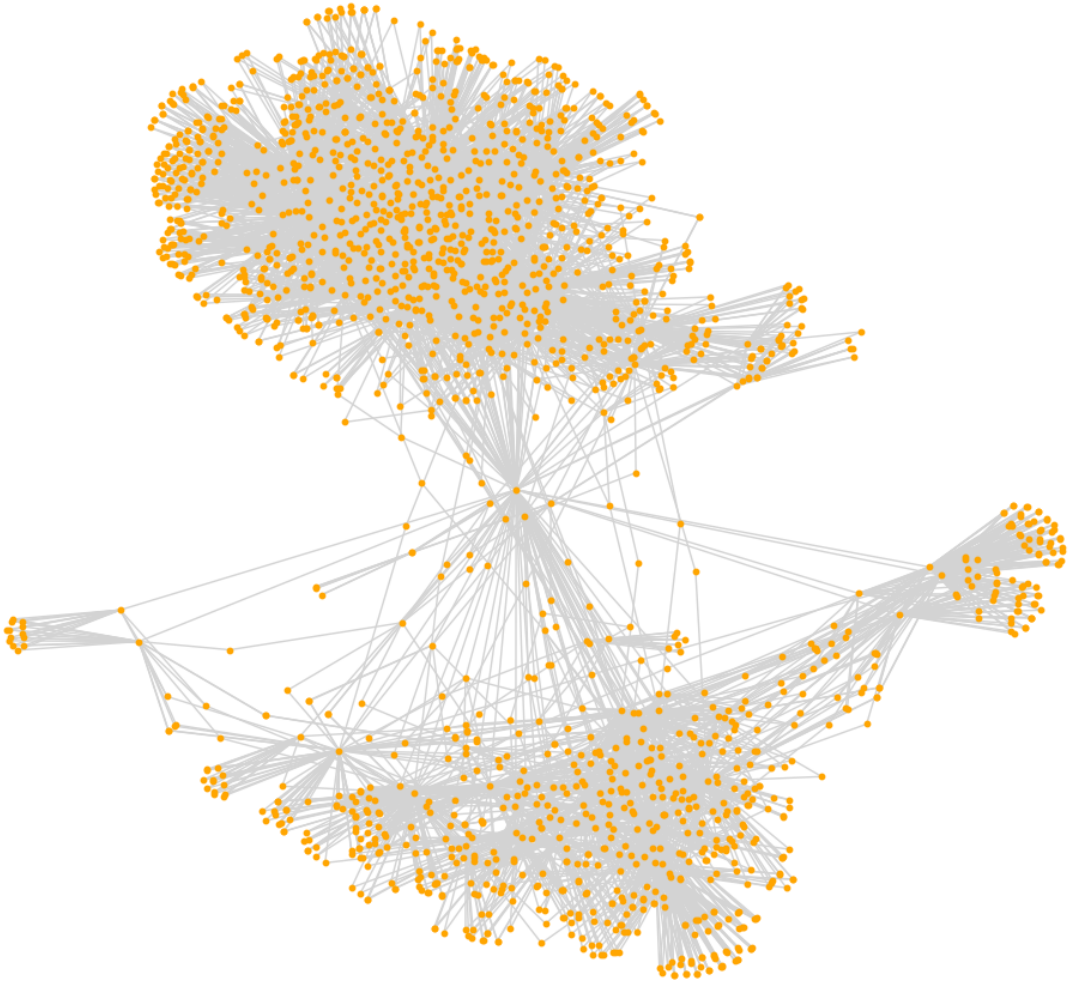
```
len(g)
```

Out[142]:

1892

В [144]:

```
# Рисуем граф
nx.draw(g, node_size=10, edge_color='lightgrey', node_color='orange')
```



Динамическое программирование

Распределение инвестиций

В [145]:

```
# Загружаем исходные данные
df = pd.read_csv("data1.csv", index_col="X")
df
```

Out[145]:

X	F1	F2	F3
0	0	0	0
100	40	30	40
200	50	80	50
300	90	80	100
400	110	150	120
500	170	190	180
600	180	200	210

В [146]:

```
# Число фирм
N = df.shape[1]
N
```

Out[146]:

```
3
```

В [147]:

```
# Объем средств
S = df.index.max()
S
```

Out[147]:

```
600
```

В [148]:

```
X = df.index
```

В [149]:

```
# Переводим в список словарей для удобства
F = [df[i].to_dict() for i in df.columns]
F
```

Out[149]:

```
[{0: 0, 100: 40, 200: 50, 300: 90, 400: 110, 500: 170, 600: 180},
 {0: 0, 100: 30, 200: 80, 300: 80, 400: 150, 500: 190, 600: 200},
 {0: 0, 100: 40, 200: 50, 300: 100, 400: 120, 500: 180, 600: 210}]
```

В [150]:

```
# Значения phi
phi = [None] * N
# Рекурсия
for i in range(N):
    phi[i] = {x: max(
        F[i][x_] + (phi[i-1][x - x_] if i != 0 else 0)
        for x_ in X if x_ <= x
    ) for x in X}

phi[-1][S]
```

Out[150]:

```
230
```

В [151]:

```
# Восстанавливаем путь
camefrom = [None] * N
for i in range(N):
    camefrom[i] = {x: X[np.argmax([
        F[i][x_] + (phi[i-1][x - x_] if i != 0 else 0)
        for x_ in X if x_ <= x
    ])] for x in X}
```

В [152]:

```
remaining = S
for i in reversed(range(N)):
    print(f"{i+1}: {camefrom[i][remaining]}")
    remaining -= camefrom[i][remaining]
```

```
3: 0
2: 500
1: 100
```

Замена оборудования

В [153]:

```
profit = [80, 75, 65, 60, 60, 55]
costs = [20, 25, 30, 35, 45, 55]
price = 40
n_years = len(profit) - 1
```

В [154]:

```
@lru_cache(maxsize=None)
def max_profit(k, t):
    return max(
        profit[t] - costs[t] + \
        (max_profit(k+1, t+1) if k != n_years else 0),

        profit[0] - costs[0] - price + \
        (max_profit(k+1, 1) if k != n_years else 0)
    )
```

В [155]:

```
max_profit(1, 0)
```

Out[155]:

```
215
```

В [160]:

```
def print_solution(k, t):

    save = profit[t] - costs[t] + \
           (max_profit(k+1, t+1) if k != n_years else 0)

    change = profit[0] - costs[0] - price + \
             (max_profit(k+1, 1) if k != n_years else 0)

    print(f"Подзадача: {k, t}, {save} | {change}")
    if save > change:
        print(f"  Год: {k}, решение: сохранить")
        return (print_solution(k+1, t+1) if k != n_years else 0)

    else:
        print(f"  Год: {k}, решение: заменить")
        return (print_solution(k+1, 1) if k != n_years else 0)
```

В [161]:

```
print_solution(1, 0)
```

```
Подзадача: (1, 0), 215 | 175
  Год: 1, решение: сохранить
Подзадача: (2, 1), 155 | 140
  Год: 2, решение: сохранить
Подзадача: (3, 2), 105 | 105
  Год: 3, решение: заменить
Подзадача: (4, 1), 85 | 70
  Год: 4, решение: сохранить
Подзадача: (5, 2), 35 | 20
  Год: 5, решение: сохранить
```

Out[161]:

```
0
```

Задача о рюкзаке

Дано множество предметов, у каждого есть вес и ценность.

Также есть максимальный вес предметов.

Необходимо найти подмножнство предметов:

1. С наибольшей ценностью
2. Вес предметов не превышает лимит

Генерация случайных данных

В [162]:

```
n = 100
weights = np.random.randint(1, 100, n)
values = np.random.randint(1, 100, n)
knapsack_weight = 500
```

В [163]:

```
class Item:
    def __init__(self, value, weight):
        self.value = value
        self.weight = weight
        self.density = value / weight
```

B [164]:

```
items = [Item(value, weight)
         for value, weight in zip(values, weights)]
```

Динамическое программирование

Сверху вниз

B [165]:

```
@lru_cache(maxsize=None)
def dp(i, S):
    if i == n:
        return 0

    return max(
        dp(i+1, S),
        (dp(i+1, S-weights[i]) + values[i]) if weights[i] <= S else 0
    )
```

B [166]:

```
%%time
dp(0, knapsack_weight)
```

```
Wall time: 64.8 ms
```

Out[166]:

```
1998
```

Снизу вверх

B [167]:

```
def bottom_up(values, weights, knapsack_weight):

    matrix = np.zeros([len(values)+1, knapsack_weight+1])

    for i in reversed(range(n)):
        for w in range(knapsack_weight+1):
            matrix[i, w] = max(
                matrix[i+1, w],
                (matrix[i+1, w - weights[i]] + values[i])
                if weights[i] <= w else 0
            )

    return matrix[0, -1]
```

B [168]:

```
%%time
dp_solution = bottom_up(values, weights, knapsack_weight)
dp_solution
```

```
Wall time: 90.8 ms
```

Out[168]:

```
1998.0
```

Линейное программирование

B [169]:

```
%%time

x = [pulp.LpVariable(f'x_{i}',
                     lowBound=0,
                     upBound=1,
                     cat='Integer')
     for i in range(n)]

model = pulp.LpProblem('Knapsack', pulp.LpMaximize)
model += sum(x[i] * values[i] for i in range(n))
model += sum(x[i] * weights[i] for i in range(n)) <= knapsack_weight

model.solve();
```

```
Wall time: 67.8 ms
```

Out[169]:

```
1
```

B [170]:

```
pulp.LpStatus[model.status]
```

Out[170]:

```
'Optimal'
```

B [171]:

```
pulp.value(model.objective)
```

Out[171]:

```
1998.0
```

Жадный алгоритм

B [172]:

```
def greedy_algorithm(items, knapsack_weight):
    sorted_items = sorted(items,
                          key=lambda item: item.density,
                          reverse=True)

    value = 0
    remaining_capacity = knapsack_weight
    for item in sorted_items:
        if item.weight <= remaining_capacity:
            value += item.value
            remaining_capacity -= item.weight

    return value
```

B [173]:

```
greedy_solution = greedy_algorithm(items, knapsack_weight)
greedy_solution
```

Out[173]:

```
1993
```

B [174]:

```
def greedy_skipping_algorithm(items,
                              knapsack_weight,
                              probability=0.9):
    items = [item for item in items if np.random.rand() <= probability]
    return greedy_algorithm(items, knapsack_weight)
```

B [175]:

```
max(
    greedy_skipping_algorithm(items, knapsack_weight, probability=0.9)
    for _ in range(100)
)
```

Out[175]:

```
1993
```

Генетический алгоритм

В [176]:

```
class Genome:

    # Инициализация - в Population
    def __init__(self, genome):
        self.genome = genome
        self.n_genes = len(genome)
        self.fitness = self.fitness_function()

    # Возвращает новый геном
    def mutation(self, mutation_percent=0.01):
        changes = np.random.choice([0, 1],
                                   self.n_genes,
                                   p=[1-mutation_percent,
                                      mutation_percent])
        new_genome = self.genome + changes
        new_genome = new_genome % 2
        return Genome(new_genome)

    def crossover(self, other):
        choice = np.random.choice([0, 1], self.n_genes)
        new_genome = np.where(choice, self.genome, other.genome)
        return Genome(new_genome)

    def fitness_function(self):
        max_items = np.searchsorted((self.genome*weights).cumsum(),
                                    knapsack_weight,
                                    side='right')
        max_value = (self.genome*values)[:max_items].sum()
        return max_value
```

В [177]:

```
class Population:

    # Инициализируем нулями?
    def __init__(self, n):
        self.n = n
        self.first_generation_size = n
        self.species = [Genome(np.zeros(len(values))) for _ in range(n)]

    # Модифицирует изначальный список
    def update_population(self, k_mutations, k_crossovers):
        to_mutation = random.choices(self.species,
                                     k=k_mutations)
        to_crossover = [random.choices(self.species, k=2)
                        for k in range(k_crossovers)]

        self.species = self.species + [i.mutation() for i in to_mutation]
        self.species = self.species + [a.crossover(b) for a,b in to_crossover]
        self.n += k_mutations + k_crossovers

    def selection(self, p=0.7):
        species_sorted = sorted(self.species, key=lambda x: x.fitness, reverse=True
        self.species = species_sorted[:self.first_generation_size]

        # self.species = sorted(self.species, key=lambda x: x.fitness, reverse=True
        # probabilities = [p*(1-p)**i for i in range(self.n)]
        # self.species = random.choices(self.species, weights=probabilities, k=self
        # self.n = self.first_generation_size

    def get_best_fitness(self):
        return self.species[0].fitness

    def get_mean_fitness(self):
        fitnesses = [i.fitness for i in self.species]
        return sum(fitnesses) / len(fitnesses)
```

В [178]:

```
history_best = []
history_mean = []

p = Population(500)
for i in range(200):
    p.update_population(100, 100)
    p.selection()
    history_best.append(p.get_best_fitness())
    history_mean.append(p.get_mean_fitness())
```

Задача о 2 рюкзаках

Генерация случайных данных

B [179]:

```
n = 100

weights = np.random.randint(1, 100, n)
values = np.random.randint(1, 100, n)
# values = weights + np.random.randint(0, 50, n)

knapsack_weights = [230, 270]
```

B [180]:

```
class Item:

    def __init__(self, value, weight):
        self.value = value
        self.weight = weight
        self.density = value / weight

items = [Item(value, weight) for value, weight in zip(values, weights)]
```

Динамическое программирование

Сверху вниз

B [181]:

```
@lru_cache(maxsize=None)
def knapsack(i, X1, X2):

    if i == n:
        return 0

    return max(
        knapsack(i+1, X1, X2),
        (knapsack(i+1, X1 - weights[i], X2) + values[i]) if weights[i] <= X1 else 0
        (knapsack(i+1, X1, X2 - weights[i]) + values[i]) if weights[i] <= X2 else 0
    )
```

B [182]:

```
%%time
dp_solution = knapsack(0, *knapsack_weights)
dp_solution
```

```
Wall time: 16 s
```

Out[182]:

```
1845
```

Снизу вверх

B [183]:

```
def bottom_up(values, weights, knapsack_weight_1, knapsack_weight_2):

    matrix = np.zeros([n+1, knapsack_weight_1+1, knapsack_weight_2+1])

    for i in reversed(range(n)):
        for w1 in range(knapsack_weight_1+1):
            for w2 in range(knapsack_weight_2+1):
                matrix[i, w1, w2] = max(
                    matrix[i+1, w1, w2],
                    (matrix[i+1, w1 - weights[i], w2] + values[i])
                    if weights[i] <= w1 else 0,

                    (matrix[i+1, w1, w2 - weights[i]] + values[i])
                    if weights[i] <= w2 else 0
                )

    return matrix[0, -1, -1]
```

B [184]:

```
%%time
bottom_up(values, weights, *knapsack_weights)
```

```
Wall time: 17.1 s
```

Out[184]:

```
1845.0
```

Линейное программирование

B [185]:

```
x = [
    [pulp.LpVariable(f'x_{i}_{j}', lowBound=0,
                     upBound=1,
                     cat='Integer')
     for i in range(len(values))]
    for j in [0, 1]
]
```

B [186]:

```
model = pulp.LpProblem('Knapsack', pulp.LpMaximize)
```

B [187]:

```
model += sum(x[knapsack][i] * values[i] for i in range(len(values)) for knapsack in
```

B [188]:

```
for k in [0, 1]:
    model += sum(x[k][i] * weights[i] for i in range(len(values))) <= knapsack_weig
```

B [189]:

```
for i in range(n):
    model += sum(x[k][i] for k in [0,1]) <= 1
```

B [190]:

```
%%time
model.solve();
```

```
Wall time: 267 ms
```

Out[190]:

```
1
```

B [191]:

```
pulp.LpStatus[model.status]
```

Out[191]:

```
'Optimal'
```

B [192]:

```
pulp.value(model.objective)
```

Out[192]:

```
1845.0
```

Жадный алгоритм

B [193]:

```
def greedy_algorithm(items, knapsack_weights):
    sorted_items = sorted(items, key=lambda item: item.density, reverse=True)

    value = 0
    remaining_capacity = knapsack_weights
    for item in sorted_items:

        remaining_capacity = sorted(remaining_capacity, reverse=True)
        if item.weight <= remaining_capacity[0]:
            value += item.value
            remaining_capacity[0] -= item.weight

    return value
```

B [194]:

```
greedy_solution = greedy_algorithm(items, knapsack_weights)
greedy_solution
```

Out[194]:

1826

Генетический алгоритм

В [195]:

```
class Genome:

    # Инициализация - в Population
    def __init__(self, genome):
        self.genome = genome
        self.n_genes = len(genome)
        self.fitness = self.fitness_function()

    # Возвращает новый геном
    def mutation(self, mutation_percent=0.01):
        new_genome = self.genome + np.random.choice([0, 1, 2],
                                                    self.n_genes,
                                                    p=[1-mutation_percent,
                                                       mutation_percent/2,
                                                       mutation_percent/2])
        new_genome = new_genome % 3
        return Genome(new_genome)

    def crossover(self, other):
        choice = np.random.choice([0, 1], self.n_genes)
        new_genome = np.where(choice, self.genome, other.genome)
        return Genome(new_genome)

    def fitness_function(self):
        is_in_first_knapsack = (self.genome == 1)
        is_in_second_knapsack = (self.genome == 2)

        max_items_1 = np.searchsorted((is_in_first_knapsack*weights).cumsum(),
                                      knapsack_weights[0],
                                      side='right')
        max_items_2 = np.searchsorted((is_in_second_knapsack*weights).cumsum(),
                                      knapsack_weights[1],
                                      side='right')

        max_value = (is_in_first_knapsack*values)[:max_items_1].sum() + \
                    (is_in_second_knapsack*values)[:max_items_2].sum()
        return max_value
```

В [196]:

```
class Population:

    # Инициализируем нулями
    def __init__(self, n):
        self.n = n
        self.first_generation_size = n
        self.species = [Genome(np.zeros(len(values))) for _ in range(n)]

    # Модифицирует изначальный список
    def update_population(self, k_mutations, k_crossovers):
        to_mutation = random.choices(self.species, k=k_mutations)
        to_crossover = [random.choices(self.species, k=2) for k in range(k_crossove

        self.species = self.species + [i.mutation() for i in to_mutation]
        self.species = self.species + [a.crossover(b) for a,b in to_crossover]
        self.n += k_mutations + k_crossovers

    def selection(self, p=0.7):
        species_sorted = sorted(self.species, key=lambda x: x.fitness, reverse=True
        self.species = species_sorted[:self.first_generation_size]

        # self.species = sorted(self.species, key=lambda x: x.fitness, reverse=True
        # probabilities = [p*(1-p)**i for i in range(self.n)]
        # self.species = random.choices(self.species, weights=probabilities, k=self
        # self.n = self.first_generation_size

    def get_best_fitness(self):
        return self.species[0].fitness

    def get_mean_fitness(self):
        fitnesses = [i.fitness for i in self.species]
        return sum(fitnesses) / len(fitnesses)
```

В [197]:

```
history_best = []
history_mean = []

p = Population(200)
for i in range(500):
    p.update_population(100, 100)
    p.selection()
    history_best.append(p.get_best_fitness())
    history_mean.append(p.get_mean_fitness())
```

Интервальное планирование

Дано множество заказов

У каждого заказа есть:

1. Время начала
2. Время завершения
3. Прибыль

Необходимо найти подмножество работ, такое что:

1. Прибыль максимальна
2. Никакие две работы не пересекаются

Генерация случайных данных

B [198]:

```
n = 500

starts = np.random.randint(1, 1000, n)
durations = np.random.binomial(400, 0.5, n)
finishes = starts + durations
values = np.random.randint(100, 1000, n)
```

B [199]:

```
class Job:

    def __init__(self, start, finish, value):
        self.start = start
        self.finish = finish
        self.value = value
        self.density = (finish - start) / value

    # Пересекается ли работа с другой работой?
    def is_overlapping(self, other):
        # Если заканчивается раньше, чем начинается другая
        # или начинается после того, как закончилась другая, то нет
        if self.finish <= other.start or self.start >= other.finish:
            return 0
        return 1
```

B [200]:

```
# Создаем работы
jobs = [Job(start, finish, value)
        for start, finish, value in zip(starts, finishes, values)]
# Создаем работы по времени завершения
jobs = sorted(jobs, key=lambda x: x.finish)
# Добавляем работам индесы
for i in range(n):
    jobs[i].index = i
```

Динамическое программирование

B [201]:

```
def p_function(i):
    not_overlapping_jobs = [job for job in jobs if job.finish <= jobs[i].start]
    if len(not_overlapping_jobs) == 0:
        return -1
    return max(i.index for i in not_overlapping_jobs)
```

B [202]:

```
@lru_cache(maxsize=None)
def dynamic_jobs(j):
    if j == -1:
        return 0

    return max(
        values[j] + dynamic_jobs(p[j]),
        dynamic_jobs(j-1)
    )
```

B [203]:

```
%%time
p = [p_function(i) for i in range(n)]
dynamic_jobs(n-1)
```

```
Wall time: 63.8 ms
```

Out[203]:

```
4962
```

Жадный алгоритм

B [204]:

```
def greedy_algorithm(jobs):
    jobs = sorted(jobs, key=lambda x: x.density, reverse=True)
    total_value = 0
    while len(jobs) > 0:
        best_job = jobs[0]
        total_value += best_job.value
        jobs = [job for job in jobs if not best_job.is_overlapping(job)]
    return total_value
```

B [205]:

```
greedy_algorithm(jobs)
```

Out[205]:

```
453
```

В [206]:

```
def greedy_skipping_algorithm(jobs, probability=0.9):
    jobs = [job for job in jobs if np.random.rand() <= probability]
    return greedy_algorithm(jobs)
```

В [207]:

```
max(
    greedy_skipping_algorithm(jobs, probability=0.9)
    for _ in range(100)
)
```

Out[207]:

```
850
```

Расстановка переносов в тексте

Дан список чисел (длины слов) и ширина страницы.

Необходимо расставить переносы, чтобы число пробелов было минимальным.

Обычно используют сумму квадратов или кубов.

В [208]:

```
power = 2
page_width = 80
```

Предварительная обработка текства

В [209]:

```
with open('text.txt', encoding='utf-8') as file:
    text = file.readlines()[0]
```

В [210]:

```
words = [len(word) for word in text.split(' ')]
n = len(words)
n
```

Out[210]:

```
202
```

Динамическое программирование

B [211]:

```
def metric(i, j, power=3):
    width = sum(words[i:j]) + len(words[i:j]) - 1
    if width > page_width:
        return float('inf')
    return (page_width - width)**power
```

Сверху вниз

B [212]:

```
@lru_cache(maxsize=None)
def dp(i):
    if i == n:
        return 0
    return min(
        metric(i, j, power=power) + dp(j)
        for j in range(i+1, n+1)
    )
```

B [213]:

```
dp(0)
```

Out[213]:

```
1012
```

Снизу вверх

B [214]:

```
def bottom_up(words):
    n = len(words)
    dp = [0 for i in range(n+1)]

    for i in reversed(range(n)):
        dp[i] = min(
            metric(i, j, power=power) + dp[j]
            for j in range(i+1, n+1)
        )

    return dp[0]
```

B [215]:

```
bottom_up(words)
```

Out[215]:

```
1012
```

Жадный алгоритм

B [216]:

```
def greedy(words):
    total_badness = 0
    line_width = words[0]
    words = words[1:]

    for word in words:

        if line_width + 1 + word <= page_width:
            line_width = line_width + 1 + word

        else:
            total_badness += (page_width - line_width)**power
            line_width = word

    total_badness += (page_width - line_width)**power

    return total_badness
```

B [217]:

```
greedy(words)
```

Out[217]:

```
4724
```