**МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ**

Федеральное государственное автономное образовательное учреждение

высшего образования

**«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ**

**ТОМСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»**

Институт кибернетики

Кафедра программной инженерии

Направление «Программная инженерия»

Отчет по лабораторной работе №1 по дисциплине

«Алгоритмы и структуры данных»

**Представление алгоритмов и их сложность**

Выполнил:

Студент группы 8К51 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ И.С. Ванюшин

Проверил:

Доцент кафедры ОСУ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ О.Б. Фофанов

Томск 2017

**Задание:**

1. Разработать алгоритм решения соответствующего варианта задания на псевдокоде

2. На основе описания алгоритма вывести формулу определения числа операций в зависимости от размерности исходных данных

3. Построить графики зависимости времени выполнения от размерности исходных данных

4. Написать программы тестирования алгоритмов и построить графики зависимости времени выполнения от размерности исходных данных на основе вычислительных экспериментов

5. Провести сравнительный анализ графиков по пунктам 3-4

**Вариант № 5**

1. Заменить минимальный элемент побочной диагонали на среднюю сумму отрицательных элементов матрицы P[n,n].
2. В матрицах A[m,n], B[m,n] найти и поменять местами максимальный элемент среди нечётных столбцов и чётных строк соответственно
3. Из положительных элементов массива Х1, Х2 , ... , Хn, расположенных правее минимального элемента, сформировать новый массив.

**Задание 1.**

Заменить минимальный элемент побочной диагонали на среднюю сумму отрицательных элементов матрицы P[n,n].

Псевдокод:

Input: массив arr, содержащий целые положительные числа

Output: массив arr с замененным в соответствии с заданием элементом

diag = GetDiagonal(arr)

min\_elem\_index = IndexOfMin(diag)

avrg\_sum = Mean(GetNegativeNumbers(arr))

min\_elem\_row = min\_elem\_index

min\_elem\_column = size(arr) - min\_elem\_index - 1

arr[min\_elem\_row, min\_elem\_column] = avrg\_sum

Код на языке Python (3.6.2):

**import** numpy **as** np  
  
np.set\_printoptions(precision**=**2)  
  
arr **=** np.random.randint(**-**30, 30, (5, 5))  
diag **=** np.diag(np.fliplr(arr))  
min\_elem\_index **=** diag.argmin()  
avrg\_sum **=** arr[arr **<** 0].mean()  
min\_elem\_row, min\_elem\_columns **=** min\_elem\_index, arr.shape[1] **-** min\_elem\_index **-** 1  
  
print(**'Default array:'**)  
print(arr)  
print(**'Index of minimal: {0}x{1}'**.format(min\_elem\_row, min\_elem\_columns))  
print(**'Negative numbers:'**, arr[arr **<** 0])  
print(**'Average negative value: '**, avrg\_sum)  
arr **=** arr.astype(float)  
arr[min\_elem\_row, min\_elem\_columns] **=** avrg\_sum  
print(**'Result array:'**)  
print(arr)

Пример результата выполнения программы:

Default array:

[[ 1 -18 25 20 -28]

[-21 -24 11 29 17]

[ 8 1 -14 13 -2]

[ 8 -30 11 27 -26]

[ 16 -25 17 19 -8]]

Index of minimal: 3x1

Negative numbers: [-18 -28 -21 -24 -14 -2 -30 -26 -25 -8]

Average negative value: -19.6

Result array:

[[ 1. -18. 25. 20. -28. ]

[-21. -24. 11. 29. 17. ]

[ 8. 1. -14. 13. -2. ]

[ 8. -19.6 11. 27. -26. ]

[ 16. -25. 17. 19. -8. ]]

Для того, чтобы асимптотически оценить сложность алгоритма, рассмотрим сложность отдельно взятых операций, описанных в коде программы. Для массива размером NxN большинство операций выполняется за время, приблизительно равное O(NlogN), с увеличение до O(N2) в наихудших условиях. Одной из таких операций является операция нахождения индекса минимального элемента массива. Поскольку массив двумерный, данная операция относительно легко возрастать по сложности до O(N2). Следовательно, исходя из того, что отдельные операции алгоритма по сложности достигают O(N2), весь алгоритм в целом будет по сложности ограничен сверху функцией O(N2).

Чтобы проверить это утверждение экспериментально, напишем отдельную программу, где данный алгоритм будет представлен функцией. Для корректности замеров времени исполнения алгоритма без учета сторонних действий, добавим в функцию только сам алгоритм в чистом виде, без вспомогательных команд, таких как print(). Опишем массив значений величины N, и будем в цикле замерять время выполнения программы на различных величинах размерности массива N. Таким образом, мы получим массив, который представляет собой все замеры времени выполнения алгоритма на различных размерах данных. На основании этих данных построим графики, а также для наглядности добавим рядом графики, соответствующие линейной и квадратичной асимптоте.

Итоговый код замера производительности алгоритма представлен ниже.

Код измерения времени выполнения программы:

**def** solve(a):

diag = np.diag(np.fliplr(a))

min\_elem\_index = diag.argmin()

avrg\_sum = a[a < 0].mean()

min\_elem\_row, min\_elem\_columns = min\_elem\_index, a.shape[1] - min\_elem\_index - 1

a = a.astype(float)

a[min\_elem\_row, min\_elem\_columns] = avrg\_sum

size\_values = list(range(100, 900, 200)) + list(range(1000, 19000, 1000))

timings = []

**for** size **in** tqdm\_notebook(size\_values):

arr = np.random.randint(-10000, 10000, (size, size))

start\_time = time.time()

solve(arr)

end\_time = time.time()

timings.append(end\_time - start\_time)

Код построения графика:

output\_notebook()

p = figure(title='Lab 1 measurements', x\_axis\_label='Array size', y\_axis\_label='Time')

p.line(size\_values, timings, legend='Result', line\_width=2, line\_color='DarkGrey')

p.circle(size\_values, timings, size=5, color='Black')

linear = [x\*0.0001 **for** x **in** size\_values]

p.line(size\_values, linear, legend='Linear function', line\_width=1,

line\_dash='dashed', line\_color='CornflowerBlue')

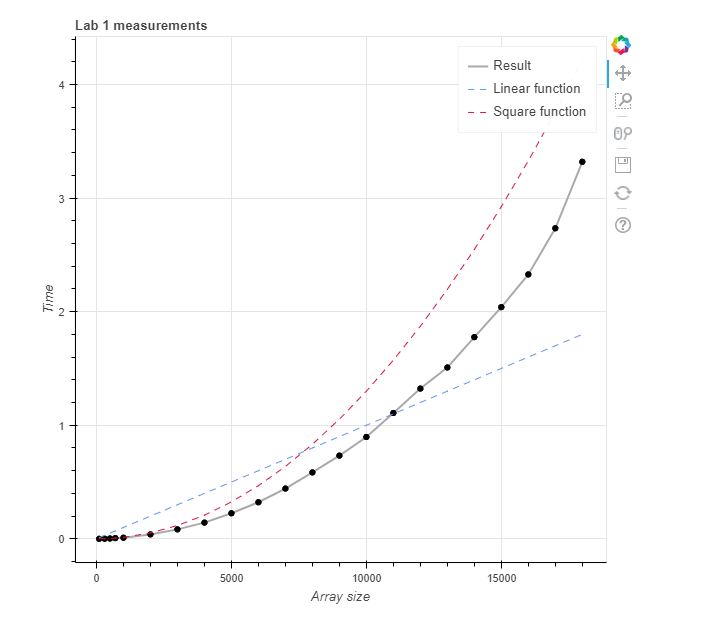
square = [1.3e-8\*(x\*\*2) **for** x **in** size\_values]

p.line(size\_values, square, legend='Square function', line\_width=1,

line\_dash='dashed', line\_color='Crimson')

show(p)

Итоговый график:



Весь итоговый код программы в тетрадке Jupyter Notebook доступен по ссылке:

<http://nbviewer.jupyter.org/github/iwouldnot/TPU_Algo_labs/blob/master/Lab%201/Lab1_1.ipynb>

**Задание 2.**

В матрицах A[m,n], B[m,n] найти и поменять местами максимальный элемент среди нечётных столбцов и чётных строк соответственно

Псевдокод:

Input: массивы A и B, размерностью [MxN], заполненные случайными числами

Output: массивы A и B, размерностью [MxN], измененные в соответствии с установленным заданием

A\_temp = A

B\_temp = B

GetEvenColumns(A\_temp) = INT\_MIN

GetOddRows(B\_temp) = INT\_MIN

A\_max\_ind = IndexOfMax(A\_temp)

B\_max\_ind = IndexOfMax(B\_temp)

swap(A[A\_max\_ind], B[B\_max\_ind])

Код на языке Python (3.6.2):

**import** numpy **as** np  
  
A **=** np.random.randint(0, 50, (3, 5))  
B **=** np.random.randint(0, 50, (3, 5))  
print(A)  
print(B)  
  
A\_temp **=** np.array(A)  
A\_temp[**:**, **::**2] **=** np.iinfo(A\_temp.dtype).min *# обращаем четные столбцы в минимально возможное числа*B\_temp **=** np.array(B)  
B\_temp[1**::**2, **:**] **=** np.iinfo(A\_temp.dtype).min *# обращаем нечетные строки в минимально возможно числа*A\_max\_ind **=** np.unravel\_index(A\_temp.argmax(), A\_temp.shape)  
B\_max\_ind **=** np.unravel\_index(B\_temp.argmax(), B\_temp.shape)  
  
print(**'\nSwapping {0} in {1} and {2} in {3}...\n'**.format(A[A\_max\_ind], A\_max\_ind, B[B\_max\_ind], B\_max\_ind))  
  
A[A\_max\_ind], B[B\_max\_ind] **=** B[B\_max\_ind], A[A\_max\_ind]  
print(A)  
print(B)

Пример результата выполнения программы:

[[ 3 7 19 31 28]

[29 20 39 6 9]

[35 9 38 20 45]]

[[18 0 17 4 22]

[37 27 43 33 36]

[41 17 28 35 43]]

Swapping 31 in (0, 3) and 43 in (2, 4)...

[[ 3 7 19 43 28]

[29 20 39 6 9]

[35 9 38 20 45]]

[[18 0 17 4 22]

[37 27 43 33 36]

[41 17 28 35 31]]

Чтобы сделать вывод о вычислительной сложности алгоритма, обратим внимание на те вычисления, которые необходимы в алгоритме. Легко заметить, что различие в размерности относительно величины по столбцам или по строкам несущественно: алгоритм будет работать абсолютно одинаково на массивах [M, N]; [M. N], так и на массивах [N, M], [N, M]. По этой причине для упрощения анализа алгоритма мы будем рассматривать асимптотику алгоритма относительно одной величины, N, имея в виду под этим, что алгоритм работает с двумя массивами размерностью [N, N].

Как и в случае с предыдущим алгоритмом, основная вычислительная сложность заключается в нахождении минимальных элементов, сложность которых может возрастать до O(N2) операций. Следовательно, сложность алгоритма также ограничивается O(N2) асимптотой. Докажем это утверждение экспериментально.

Код измерения времени выполнения программы:

**def** solve(A, B):

A\_temp = np.array(A)

A\_temp[:, ::2] = -1 *# обращаем четные столбцы в минимально возможное числа*

B\_temp = np.array(B)

B\_temp[1::2, :] = -1 *# обращаем нечетные строки в минимально возможно числа*

A\_max\_ind = np.unravel\_index(A\_temp.argmax(), A\_temp.shape)

B\_max\_ind = np.unravel\_index(B\_temp.argmax(), B\_temp.shape)

A[A\_max\_ind], B[B\_max\_ind] = B[B\_max\_ind], A[A\_max\_ind]

size\_values = list(range(100, 900, 200)) + list(range(1000, 14000, 1000))

timings = []

**for** size **in** tqdm\_notebook(size\_values):

b = np.random.randint(0, 10000, (size, size))

a = np.random.randint(0, 10000, (size, size))

start\_time = time.time()

solve(a, b)

end\_time = time.time()

timings.append(end\_time - start\_time)

Код построения графика:

output\_notebook()

p = figure(title='Lab 2 measurements', x\_axis\_label='Array size', y\_axis\_label='Time in sec')

p.line(size\_values, timings, legend='Result', line\_width=2, line\_color='DarkGrey')

p.circle(size\_values, timings, size=5, color='Black')

linear = [x\*0.0001 **for** x **in** size\_values]

p.line(size\_values, linear, legend='Linear function', line\_width=1,

line\_dash='dashed', line\_color='CornflowerBlue')

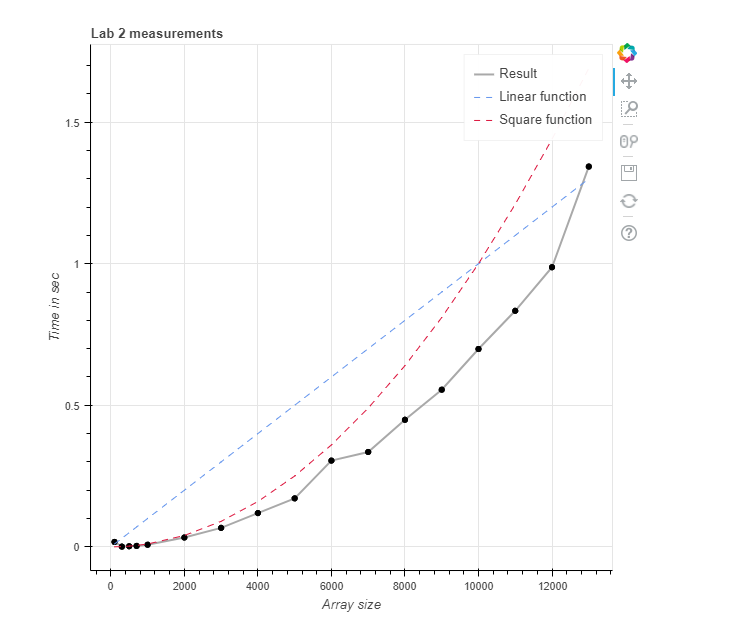
square = [1e-8\*(x\*\*2) **for** x **in** size\_values]

p.line(size\_values, square, legend='Square function', line\_width=1,

line\_dash='dashed', line\_color='Crimson')

show(p)

Итоговый график:



Весь итоговый код программы в тетрадке Jupyter Notebook доступен по ссылке:

<http://nbviewer.jupyter.org/github/iwouldnot/TPU_Algo_labs/blob/master/Lab%201/Lab1_2.ipynb>

**Задание 3.**

Из положительных элементов массива Х1, Х2 , ... , Хn, расположенных правее минимального элемента, сформировать новый массив.

Псевдокод:

Input: массив X, заполненный целыми числами

Output: массив X, содержащий положительные числа, найденными в соответствии с заданием

min\_index = IndexOfMin(X)

result = X.GetRange(min\_index + 1, X.Last())

result = GetPositiveNumbers(result)

Код на языке Python (3.6.2):

**import** numpy **as** np  
  
X **=** np.random.randint(**-**50, 50, 25)  
print(X)  
min\_index **=** X.argmin()  
print(**'Index of minimal value: {0}, minimal value: {1}'**.format(min\_index, X[min\_index]))  
result **=** X[min\_index**+**1**:**]  
result **=** result[result **>** 0]  
print(**'All positives after {0}: {1}'**.format(X[min\_index], result))

Пример результата выполнения программы:

[ 11 -35 39 20 4 -31 -17 -46 -24 12 -29 20 32 -15 -24 42 -10 21

-40 30 -31 -32 -4 -27 42]

Index of minimal value: 7, minimal value: -46

All positives after -46: [12 20 32 42 21 30 42]

В отличие от предыдущих двух заданий, в данном задании все алгоритм обрабатывает лишь одномерный массив, поэтому асимптотика алгоритма будет оптимально равна O(logN) с падением производительности до O(N). Поскольку О-нотация обязывает рассматривать наихудший случай, мы оцениваем производительность этого алгоритма как линейный, то есть сложность равна O(N). Докажем это экспериментально.

Код измерения времени выполнения программы:

**def** solve(A):

min\_index = A.argmin()

result = X[min\_index+1:]

**return** result[result > 0]

size\_values = list(range(100000, 9000000, 500000))

timings = []

**for** size **in** tqdm\_notebook(size\_values):

arr = np.random.randint(-10000, 10000, size)

start\_time = time.time()

solve(arr)

end\_time = time.time()

timings.append(end\_time - start\_time)

Код построения графика:

output\_notebook()

p = figure(title='Lab 2 measurements', x\_axis\_label='Array size', y\_axis\_label='Time in sec')

p.line(size\_values, timings, legend='Result', line\_width=2, line\_color='DarkGrey')

p.circle(size\_values, timings, size=5, color='Black')

linear = [x\*1.5e-9 **for** x **in** size\_values]

p.line(size\_values, linear, legend='Linear function', line\_width=1,

line\_dash='dashed', line\_color='CornflowerBlue')

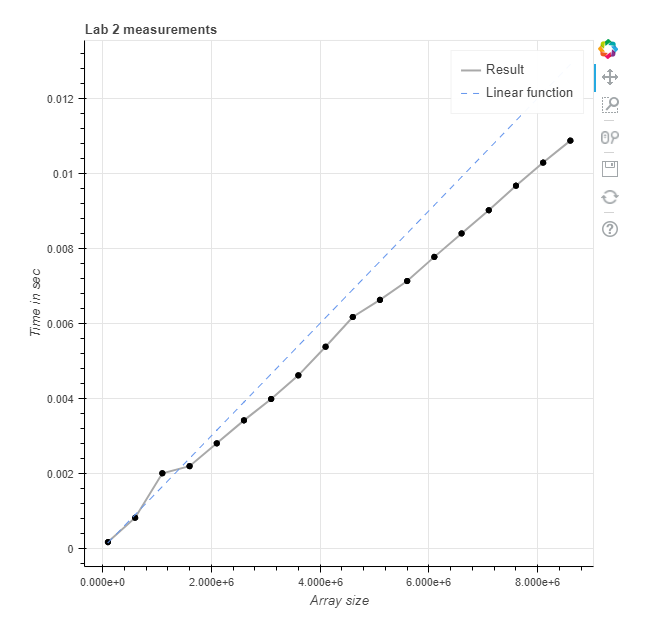
*# square = [1e-10\*(x\*\*2) for x in size\_values]*

*# p.line(size\_values, square, legend='Square function', line\_width=1,*

*# line\_dash='dashed', line\_color='Crimson')*

show(p)

Итоговый график:



Весь итоговый код программы в тетрадке Jupyter Notebook доступен по ссылке: <http://nbviewer.jupyter.org/github/iwouldnot/TPU_Algo_labs/blob/master/Lab%201/Lab1_3.ipynb>

**Выводы**

Были построены различные алгоритмы, подходящие для различных заданий. Определено, что для каждой конкретно заданной задачи возможно определить оптимальный алгоритм, однако оптимальность алгоритма для одной задачи не гарантирует оптимальность для любой другой подобной задачи. Получена практика определения алгоритмической сложности алгоритма, а также осуществления выбора походящего алгоритма на основе поставленной задачи.

Код всей лабораторной работы вместе с иходными кодами программ, а также тетрадками Jupyter Notebook с построенными графиками в них доступен в репозитории GitHub, доступном по ссылке <https://github.com/iwouldnot/TPU_Algo_labs>.