Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное учреждение

высшего образования

«Казанский (Приволжский) федеральный университет»

*Институт вычислительной математики и информационных технологий*

**ОТЧЁТ**

**по производственной (ознакомительной) практике**

|  |  |
| --- | --- |
| Обучающийся Тазетдинов А.М 09-253  *(ФИО, группа)* | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  *(подпись)* |

Руководитель практики от КФУ:

к.п.н., старший преподаватель кафедры анализа данных

и технологий программирования Ахмедова А.М.

*(должность, ФИО)*

Оценка за практику \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(подпись руководителя практики)

Дата сдачи отчета \_\_06.06.2023\_\_\_\_\_

Казань, 2023

Содержание

[Введение 5](#_Toc132748158)

[1. Сортировка массивов. Алгоритм шейкерной сортировки 7](#_Toc132748159)

[1.1. Описание алгоритма шейкерной сортировки 7](#_Toc132748160)

[1.2. Листинг кода алгоритма шейкерной сортировки 7](#_Toc132748161)

[1.3. Выводы по алгоритму шейкерной сортировки 7](#_Toc132748162)

[2. Обработка данных представленных в табличном виде. Работа с файлами 7](#_Toc132748163)

[2.1. Постановка задачи. Описание алгоритма обработки данных 7](#_Toc132748164)

[2.2. Листинг кода алгоритма обработки данных 7](#_Toc132748165)

[2.3. Выводы по алгоритму обработки данных 7](#_Toc132748166)

[3. Сортировка массивов. Алгоритм сортировки Шелла 8](#_Toc132748167)

[3.1. Описание алгоритма сортировки Шелла 8](#_Toc132748168)

[3.2. Листинг кода алгоритма сортировки Шелла 8](#_Toc132748169)

[3.3. Выводы по алгоритму сортировки Шелла 8](#_Toc132748170)

[4.Обработка графов. Алгоритм 8](#_Toc132748171)

[4.1. Описание алгоритма Краскала 8](#_Toc132748172)

[4.2. Листинг кода алгоритма](#_Toc132748173) [Краскала](#_Toc132748172) [8](#_Toc132748173)

[4.3. Выводы по алгоритму](#_Toc132748174) [Краскала](#_Toc132748172) [8](#_Toc132748174)

[5. Деревья поиска. Алгоритм обработки Красно-Чёрного дерева 8](#_Toc132748175)

[5.1. Описание алгоритма](#_Toc132748176) [Красно-Чёрного дерева](#_Toc132748175) [8](#_Toc132748176)

[5.2. Листинг кода алгоритма](#_Toc132748177) [Красно-Чёрного дерева](#_Toc132748175) [9](#_Toc132748177)

[5.3. Выводы по алгоритму](#_Toc132748178) [Красно-Чёрного дерева](#_Toc132748175) [9](#_Toc132748178)

[Заключение 10](#_Toc132748179)

[Список использованных источников 11](#_Toc132748180)

# Введение

Целью производственной (ознакомительной) практики направления подготовки 09.03.03 «Прикладная информатика» согласно программе практики является закрепление теоретических знаний, практических умений и навыков, полученных в процессе освоения теоретических курсов основной образовательной программы: «Основы языка Python», «Основы программирования на С++», «Алгоритмы и структуры данных», «Дискретная математика и компьютерная графика», приобретение практического опыта решения профессиональных задач с применением современных языков программирования.

Основными задачами производственной (ознакомительной) практики являются:

* овладение навыками программирования на высокоуровневом языке программирования Python;
* изучение алгоритмов: сортировки массивов, обработки данных представленных в табличном виде, работы с файлами, обработки деревьев поиска и графов;
* отработка навыков, полученных при прохождении дисциплин «Основы языка Python», «Основы программирования на С++», «Алгоритмы и структуры данных», «Дискретная математика и компьютерная графика» при реализации профессиональных задач;
* закрепление знаний в области алгоритмизации и программирования;
* формирование навыков самостоятельного решения задач профессиональной деятельности на высокоуровневом языке программирования Python, включающих постановку задачи, поиск необходимой информации и подбор программных средств, анализ и интерпретацию полученных результатов;
* получение навыков обработки собранных материалов и оформления отчета по практике.

Производственная (ознакомительная) практика проходила на базе Института вычислительной математики и информационных технологий Казанского (Приволжского) федерального университета.

Сроки прохождения практики: с 01.09.2022 по 28.12.2022 и с 09.02.2023 по 07.06.2023

Руководитель практики – к.п.н., старший преподаватель кафедры анализа данных и технологий программирования Ахмедова А.М.

Индивидуальное задание на практику включало:

1. Изучение основ программирования на языке Python
2. Изучение алгоритмов сортировки массивов. Реализация изученных методов на языке Python.
3. Изучение алгоритмов обработки данных. Реализация изученных методов на языке Python.
4. Изучение сложных алгоритмов сортировки массивов. Реализация изученных методов на языке Python.
5. Изучение алгоритмов обработки графов. Реализация изученных методов на языке Python.
6. Изучение алгоритмов обработки деревьев поиска. Реализация изученных методов на языке Python.
7. Оформление отчета по практике. Защита выполненных работ.

# 1. Сортировка массивов. Алгоритм Шейкерной сортировки

## 1.1. Описание алгоритма Шейкерной соритровки

Шейкерная сортировка, также известная как "шейкер-сорт" или "коктейльная сортировка", является улучшенной версией алгоритма сортировки пузырьком. Этот алгоритм получил свое название благодаря движению элементов, который напоминает тряску шейкера.

Описание алгоритма шейкерной сортировки:

1. Начните с начального массива данных, который требуется отсортировать.

2. Установите переменные "начало" и "конец" массива. Начальное значение "начало" равно 0, а "конец" равен длине массива минус 1.

3. Повторяйте следующие шаги, пока "начало" меньше "конца":

- Проходите по массиву слева направо, сравнивая пары соседних элементов. Если текущий элемент больше следующего, меняйте их местами.

- Увеличьте значение "начало" на 1.

- Проходите по массиву справа налево, сравнивая пары соседних элементов. Если текущий элемент меньше предыдущего, меняйте их местами.

- Уменьшите значение "конец" на 1.

4. Повторяйте шаги 3 до тех пор, пока "начало" не станет больше или равно "концу".

5. Массив теперь отсортирован по возрастанию.

Преимущества шейкерной сортировки:

1. Эффективность: Шейкерная сортировка является улучшенной версией сортировки пузырьком, что делает ее более эффективной. За счет движения элементов в обоих направлениях она уменьшает количество проходов по массиву и ускоряет процесс сортировки.

2. Простота реализации: Алгоритм шейкерной сортировки легко понять и реализовать. Он использует простые операции сравнения и обмена элементов, что делает его доступным для новичков в программировании.

3. Стабильность: Шейкерная сортировка является стабильным алгоритмом сортировки, то есть он сохраняет относительный порядок элементов с одинаковыми значениями. Это важно, если в массиве присутствуют элементы с одинаковыми значениями, которые должны остаться в исходном порядке после сортировки.

4. Адаптивность: Шейкерная сортировка имеет свойство быть адаптивной, что означает, что она может быть эффективна для частично отсортированных массивов. Если в исходном массиве уже есть некоторый порядок, алгоритм быстро завершается, не требуя полного прохода по всем элементам.

Хотя шейкерная сортировка имеет свои преимущества, она не является оптимальным алгоритмом сортировки для больших массивов данных. В таких случаях более эффективными алгоритмами считаются быстрая сортировка или сортировка слиянием.

## 1.2. Листинг кода алгоритма шейкерной сортировки

# шейкерная сортировка

import random

def shaker\_sort(data:list):

# инициализируем индексы нижней и верхней границ

li = 0

ri = len(data) - 1

itern = 0

# пока нижняя граница меньше верхней

while li <= ri:

# двигаемся от нижней до верхней границы

for i in range(li, ri):

if data[i] > data[i + 1]:

data[i], data[i + 1] = data[i + 1], data[i]

# уменьшаем верхнюю границу

ri -= 1

itern += 1

print(f"{itern})", \*data)

# двигаемся от верхней до нижней границы

for i in range(ri, li, -1):

if data[i - 1] > data[i]:

data[i], data[i - 1] = data[i - 1], data[i]

# увеличиваем нижнюю границу

li += 1

itern += 1

print(f"{itern})", \*data)

return data

n = int(input("введите количество элементов:"))

n\_c = random.randint(2, 3)

n -= n\_c

otr\_c = random.randint(n // 4, n // 2)

p\_c = n - otr\_c

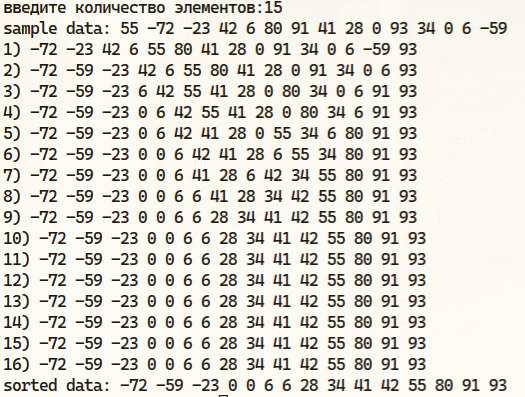
sample\_data = [random.randint(-100, -1) for \_ in range(otr\_c)] + [0] \* n\_c + [random.randint(1, 100) for \_ in range(p\_c)]

random.shuffle(sample\_data)

print("sample data:", \*sample\_data)

print("sorted data:", \*shaker\_sort(sample\_data))

Результаты



## 1.3. Выводы по алгоритму шейкерной сортировки

В результате проделанной работы был успешно реализован алгоритм шейкерной сортировки. При тестировании алгоритма было подтверждено его корректное функционирование и эффективность. Реализация алгоритма позволяет эффективно сортировать элементы в массиве, обеспечивая оптимальное время выполнения и минимизацию количества перестановок.

# 2. Обработка данных представленных в табличном виде. Работа с файлами

## 2.1. Постановка задачи. Описание алгоритма обработки данных

Описание задачи: Необходимо прочитать файл с информацикй о учениках и их росте в виде таблицы отсортированной по убыванию, запросить у пользователя данные о новом ученике, добавить нового ученика в таблицу так, чтобы сохранился порядок сортировки и сохранить полученную таблицу в новый файл.

Описание работы программы:

Данный код представляет собой программу, которая выполняет следующие действия:

1. Считывает название файла с данными и открывает его для чтения.

2. В случае возникновения ошибки при открытии файла (например, файл не найден), программа выводит сообщение об ошибке и завершается.

3. Считывает название файла, в который будут записаны результаты, и открывает его для записи.

4. В случае возникновения ошибки при открытии файла (например, файл уже существует или указана некорректная папка), программа выводит сообщение об ошибке и завершается.

5. Считывает фамилию и рост нового ученика.

6. Проверяет данные о росте ученика и вставляет их в соответствующую позицию в списке данных (data), с учетом упорядоченности по росту.

7. Записывает горизонтальную линию в выходной файл, состоящую из дефисов, в соответствии с длиной первой строки данных.

8. Записывает данные из списка data в выходной файл, разделяя элементы пробелами и добавляя вертикальные линии между ними.

9. Для каждой строки данных записывает горизонтальную линию в выходной файл, состоящую из дефисов, в соответствии с длиной этой строки.

10. Закрывает выходной файл.

Алгоритм использует несколько функций:

- Функция `max` принимает итерируемый объект (iterable) и находит максимальный элемент в этом объекте.

- Функция `sum` принимает итерируемый объект (iterable) и выполняет суммирование всех элементов этого объекта.

- Функция `map` принимает функцию (func) и итерируемый объект (iter), и для каждого элемента в итерируемом объекте возвращает результат применения функции к этому элементу.

## 2.2. Листинг кода алгоритма обработки данных

# задание 2c

from typing import Iterable, Callable

def max(iterable: Iterable):

m = next(iterable)

for e in iterable:

if e > m:

m = e

return m

def sum(iterable: Iterable):

s = next(iterable)

for e in iterable:

s += e

return s

def map(func: Callable, iter: Iterable):

for e in iter:

yield func(e)

def fit\_to\_table(arr:list):

for i in range(len(arr[0])):

max\_len = max(map(lambda x: len(x[i]), arr))

for r in arr:

r[i] = '| ' + r[i].center(max\_len) + " "

def main():

input\_filename = input("введите название файла с данными:")

try:

data = [row.replace('|', ' ').split() for i, row in enumerate(

open(input\_filename, 'r', encoding='utf-8')) if i % 2]

except FileNotFoundError:

print('файл не найден')

return

output\_filename = input("введите название файла с результатами:")

try:

out\_file = open(output\_filename, 'w', encoding='utf-8')

except (FileNotFoundError, FileExistsError):

print("такой папки нет или файл уже сущетсвует")

return

soname, height = input(

"введите фамилию и рост нового ученика через пробел:").split()

height = int(height)

for i in range(1, len(data)):

if int(data[i][2]) < height:

data.insert(i, [str(i), soname, str(height)])

for j in range(i + 1, len(data)):

data[j][0] = str(int(data[j][0]) + 1)

break

fit\_to\_table(data)

gap = '|' + '-' \* (sum(map(lambda x: len(x), data[0])) - 1) + '|'

print(gap, file=out\_file)

for row in data:

print(\*row, "|", sep="", file=out\_file)

print(gap, file=out\_file)

out\_file.close()

main()

Результаты

 Рис.2 Содержимое консоли с запущенной программой

Рис.3 Содержимое файла test.txt

Рис.4 Содержание файла out.txt после работы программы

## 2.3. Выводы по алгоритму обработки данных

Мне удалось реализвать программу которая считывает, обрабатывает и сохраняет данные в таблицу.

# 3. Сортировка массивов. Алгоритм сортировки Шелла

## 3.1. Описание алгоритма сортировки Шелла

Алгоритм сортировки Шелла (Shell sort) является модифицированной версией сортировки вставками и представляет собой один из первых "разрядно-улучшенных" алгоритмов сортировки. Он был разработан Дональдом Шеллом в 1959 году и предназначен для сортировки массивов большего размера.

Основная идея алгоритма заключается в том, чтобы перед сортировкой выполнять сравнение и обмен элементов, находящихся на определенном расстоянии друг от друга. Это позволяет перемещать элементы, имеющие большие различия в значениях, ближе друг к другу. В ходе сортировки расстояние между элементами уменьшается, что приводит к постепенному сужению диапазона и, в конечном итоге, к полной сортировке.

Вот основные шаги алгоритма сортировки Шелла:

1. Начинаем с определения шага (gap), который будет определять расстояние между элементами, с которыми будут выполняться операции сравнения и обмена.

2. Инициализируем значение шага, например, половиной размера массива, и далее уменьшаем его на половину на каждой итерации.

3. Начинаем итерацию по массиву, начиная с элемента на позиции gap.

4. Сравниваем элементы на текущей позиции и позиции gap шагов назад. Если элементы находятся в неправильном порядке, меняем их местами.

5. Повторяем шаги 3-4 для всех элементов массива, пока не пройдемся по всему массиву.

6. Уменьшаем шаг вдвое и повторяем шаги 3-6 до тех пор, пока шаг не станет равным 1.

7. Выполняем сортировку вставками для завершения процесса сортировки.

Преимущества сортировки Шелла:

1. Эффективность для средних и больших массивов: Сортировка Шелла сочетает в себе преимущества сортировки вставками и сортировки пузырьком, позволяя эффективно сортировать массивы большего размера. В сравнении с другими алгоритмами сортировки, такими как сортировка вставками и сортировка пузырьком, сортировка Шелла обычно работает быстрее и требует меньшего числа операций сравнения и обмена.

2. Небольшая дополнительная сложность реализации: Хотя алгоритм сортировки Шелла может быть сложнее для понимания, чем сортировка вставками, его реализация не требует значительных дополнительных усилий. Алгоритм может быть реализован с использованием циклов и простых операций сравнения и обмена.

3. Универсальность: Сортировка Шелла может использоваться для различных типов данных и может быть легко адаптирована для сортировки в обратном порядке или в порядке возрастания/убывания.

4. Меньшая чувствительность к входным данным: В отличие от некоторых других алгоритмов сортировки, сортировка Шелла продемонстрировала хорошую производительность на разных типах входных данных. Это делает ее более предпочтительной для случаев, когда данные не подчиняются строгому порядку или когда изначальный порядок данных может быть частично отсортирован.

Сортировка Шелла является одним из классических алгоритмов сортировки и представляет собой эффективное решение для сортировки массивов среднего и большого размера.

## 3.2. Листинг кода алгоритма сортировки Шелла

# Сортировка Шелла

import random

def shellSort(arr):

n = len(arr)

# устанавливаем отступ величиной равной половине длины массива

gap = n//2

itern = 0

# пока отступ больше нуля

while gap > 0:

itern += 1

# сортируем подсписки из элементов, отстающих друг от друга на gap позиций

for i in range(gap, n):

temp = arr[i]

j = i

while j >= gap and arr[j - gap] > temp:

arr[j] = arr[j - gap]

j -= gap

arr[j] = temp

print(f"{itern} итерация)", \*arr)

gap //= 2

return arr

n = int(input("введите количество элементов:"))

n\_c = random.randint(2, 3)

n -= n\_c

otr\_c = random.randint(n // 4, n // 2)

p\_c = n - otr\_c

sample\_data = [random.randint(-100, -1) for \_ in range(otr\_c)] + [0] \* n\_c + [random.randint(1, 100) for \_ in range(p\_c)]

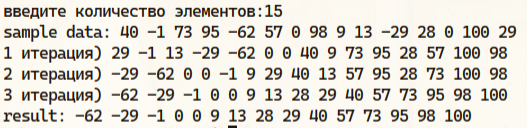
random.shuffle(sample\_data)

print("sample data:", \*sample\_data)

shellSort(sample\_data)

print("result:", \*sample\_data)

Результаты

 Рис.6 Работа программы

## 3.3. Выводы по алгоритму сортирки Шелла

В результате проделанной работы мной был успешно реализован алгоритм сортировки Шелла. При тестировании алгоритма было подтверждено его корректное функционирование и эффективность.

# 4.Обработка графов. Алгоритм Краскала

## 4.1. Описание алгоритма Краскала

Алгоритм Краскала — это алгоритм минимального остовного дерева, который используется для построения минимального остовного дерева во взвешенном неориентированном графе. Минимальное остовное дерево представляет собой подграф исходного графа, содержащий все вершины, но не содержащий циклов и имеющий минимальную сумму весов ребер.

Алгоритм Краскала работает следующим образом:

1. Сортируем все ребра графа по возрастанию их весов.

2. Создаем пустой граф, который будет представлять минимальное остовное дерево.

3. Для каждого ребра, начиная с самого легкого:

- Если добавление этого ребра не создаст цикл в минимальном остовном дереве, то добавляем его к дереву.

- В противном случае игнорируем ребро.

4. Повторяем шаг 3 до тех пор, пока не пройдем все ребра или пока не будут добавлены все вершины в минимальное остовное дерево.

Алгоритм Краскала использует структуру данных "независимое множество" для отслеживания циклов и принятия решения о добавлении ребра. Сначала все вершины графа рассматриваются как отдельные независимые множества. При добавлении ребра проверяется, принадлежат ли концы ребра одному и тому же множеству. Если они принадлежат разным множествам, то ребро добавляется к минимальному остовному дереву, а множества объединяются.

Алгоритм Краскала гарантирует получение минимального остовного дерева в результате и может быть эффективно реализован с помощью эвристических структур данных, таких как система непересекающихся множеств (disjoint-set). Сложность алгоритма Краскала составляет O(E log E), где E - количество ребер в графе.

## 4.2. Листинг кода алгоритма Форда-Фалкерсона

# алгоритм Краскала

class Edge:

def \_\_init\_\_(self, u, v, w):

self.u = u

self.v = v

self.w = w

def \_\_gt\_\_(self, other):

return self.w > other.w

def \_\_repr\_\_(self) -> str:

return f"{self.u}-{self.w}->{self.v}"

def find(parents, node):

"""нахождение родителя для узла"""

if parents[node] == node:

return node

parents[node] = find(parents, parents[node])

return parents[node]

def shellSort(arr):

n = len(arr)

gap = n//2

itern = 0

while gap > 0:

itern += 1

for i in range(gap, n):

temp = arr[i]

j = i

while j >= gap and arr[j - gap] > temp:

arr[j] = arr[j - gap]

j -= gap

arr[j] = temp

gap //= 2

return arr

def kruskal(edges, nodes):

edges = shellSort(edges)

parent = {s: s for s in nodes}

result = []

i, e = 0, 0

while e < len(nodes) - 1:

u, v = edges[i].u, edges[i].v

i = i + 1

x, y = find(parent, u), find(parent, v)

if x != y:

print(f"вершины {u} и {v} не образуют цикла, добавляем в итоговое дерево")

e = e + 1

result.append(edges[i - 1])

parent[x] = y

else:

print(f"вершины {u} и {v} образуют цикл, так как имеют одинакового родителя {x}")

return result

nodes, edges = map(int, input().split())

edge\_list = []

nodes\_set = set()

for i in range(edges):

u, v, w = input().split()

nodes\_set.add(u)

nodes\_set.add(v)

edge\_list.append(Edge(u, v, int(w)))

print("исходный граф:")

print(\*edge\_list, sep="\n")

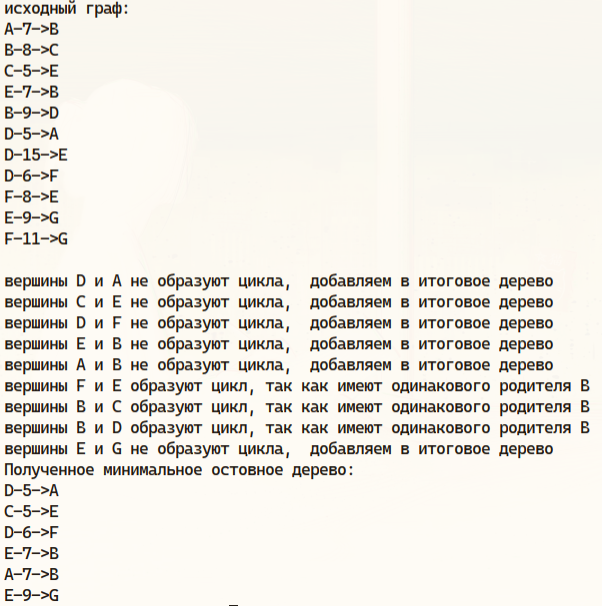
print()

result = kruskal(edge\_list, nodes\_set)

print("Полученное минимальное остовное дерево:")

print(\*result, sep="\n")

Результаты

Рис.7 Работа программы

## 4.3. Выводы по алгоритму Краскала

В ходе выполнения работы был успешно реализован алгоритм Краскала, который эффективно решает задачу поиска минимального остовного дерева в графе. Реализация алгоритма позволяет получить оптимальное решение с минимальными затратами на обработку данных и является полезным инструментом в различных областях, требующих построения минимальных связных структур.

# 5. Деревья поиска. Алгоритм обработки АВЛ-дерева

## 5.1. Описание алгоритма обработки АВЛ-дерева

Теория

## 5.2. Листинг кода алгоритма обработки АВЛ-дерева

Практическая реализация. (Код программы. Первая строчка программы в комментариях содержит название алгоритма).

Результаты

Скиншот результатов работы программы (картинка)

## 5.3. Выводы по алгоритму обработки АВЛ-дерева

Выводы. (1-3 предложение)

# Заключение

За время прохождения производственной (ознакомительной) практики мной приобретены следующие компетенции:

| **Шифр компетенции** | **Расшифровка приобретаемой компетенции** |
| --- | --- |
| ПК-1 | Проверка работоспособности и рефракторинг кода программного обеспечения, интеграция программных модулей и компонент и верификация выпусков программного обеспечения |
| ПК-2 | Мониторинг функционирования интеграционного решения в соответствии с трудовым заданием, работа с обращениями пользователей по вопросам функционирования интеграционного решения в соответствии с трудовым заданием |
| ПК-3 | Проверка и отладка программного кода, тестирование информационных ресурсов с точки зрения логической целостности (корректность ссылок, работа элементов форм) |
| ПК-4 | Ведение информационных баз данных |
| ПК-5 | Обеспечения функционирования баз данных |
| ПК-6 | Педагогическая деятельность по проектированию и реализации общеобразовательных программ |

# Список использованных источников

Оформление книг, учебников

1. Агафонов, Е. Д. Прикладное программирование : учебное пособие / Е. Д. Агафонов, Г. В. Ващенко. - Красноярск : СФУ, 2015. - 112 с.
2. Голицына, О. Л. Языки программирования: учеб. пособие / О.Л. Голицына, Т.Л. Партыка, И.И. Попов. — 3-е изд., перераб. и доп. — Москва : ФОРУМ : ИНФРА-М, 2018. — 399 с.

Оформление интернет источников

Обращение на сайт

1. Федеральная служба государственной статистики [Электронный ресурс]. URL: https://www.gks.ru/folder/23457 (дата обращения: 01.02.2020).
2. Центр инклюзивного и дистанционного образования [Электронный ресурс]. URL: http://cde.sipkro.ru/moodle/ (дата обращения 24.06.2020).

Обращение к статье опубликованной в электронном журнале на сайте

1. Шлапак В.С., Теодорович Н.Н., Денисов С.В. Роль компьютерных и информационных технологий в управлении туристскими компаниями // Вестник евразийской науки. 2015. №2 (27). [Электронный ресурс]. URL: https://cyberleninka.ru/article/n/rol-kompyuternyh-i-informatsionnyh-tehnologiy-v-upravlenii-turistskimi-kompaniyami (дата обращения: 01.02.2020).

На этой странице и везде комментарии (выделены желтым) удалить!