|  |  |
| --- | --- |
| для прик эмбл | |
| МИНОБРНАУКИ РОССИИ | |
| Федеральное государственное бюджетное образовательное учреждение  высшего профессионального образования  **"Московский технологический университет"**  **МИРЭА** | |
| Институт информационных технологий (ИТ) | |
|  | |

|  |  |
| --- | --- |
| **ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ** | |
| **по дисциплине** | |
| **«**Анализ сложности алгоритмов**»** | |
|  | |
| Выполнил студент группы ИКБО-05-16 | Пронина Е.Р. |
| Принял преподаватель | Мирабо Е.И. |

|  |  |  |
| --- | --- | --- |
| Лабораторные работы выполнены | «\_\_»\_\_\_\_\_\_\_201\_\_ г. |  |
|  |  |  |
| «Зачтено» | «\_\_»\_\_\_\_\_\_\_201\_\_ г. |  |

Москва 2017

Оглавление

[Постановка задачи 1](#_Toc483509008)

[Теоретические сведения 2](#_Toc483509009)

[Сортировки 2](#_Toc483509010)

[Поиск 3](#_Toc483509011)

[Ход работы 4](#_Toc483509012)

[Написание программы на C в среде CodeBlocks 4](#_Toc483509013)

[Тестирование программы 4](#_Toc483509014)

[Обработка данных 5](#_Toc483509015)

[Построение диаграмм 5](#_Toc483509016)

[Вывод 7](#_Toc483509017)

[Приложение 7](#_Toc483509018)

[Таблицы 7](#_Toc483509019)

[Листинг 9](#_Toc483509020)

# Постановка задачи

1. Ознакомиться с понятием сложность алгоритма
2. Ознакомиться с методами оценки сложности алгоритмов
3. Изучить алгоритмы сортировок
   1. Пузырьковая сортировка
   2. Быстрая сортировка
   3. Сортировка вставками
   4. Сортировка деревом
4. Изучить алгоритмы поиска
   1. Линейный поиск
   2. Поиск по дереву
   3. Интерполяционный поиск
   4. Быстрый линейный поиск
5. Используя приобретенные знания, написать программу для подсчета временной сложности с учетом
   * Времени выполнения
   * Количества сравнений
   * Количества обращений к памяти
6. Построить по полученным данным таблицы
7. Построить графики и по ним оценить сложность алгоритмов
8. Сделать вывод

# Теоретические сведения

### Сортировки

#### Быстрая

"Быстрая сортировка", хоть и была разработана более 40 лет назад, является наиболее широко применяемым и одним их самых эффективных алгоритмов.

Метод основан на подходе "разделяй-и-властвуй". Общая схема такова:

1. из массива выбирается некоторый опорный элемент a[i],
2. запускается процедура разделения массива, которая перемещает все ключи, меньшие, либо равные a[i], влево от него, а все ключи, большие, либо равные a[i] - вправо,
3. теперь массив состоит из двух подмножеств, причем левое меньше, либо равно правого,  
   http://algolist.manual.ru/sort/gif/22.gif
4. для обоих подмассивов: если в подмассиве более двух элементов, рекурсивно запускаем для него ту же процедуру.

В конце получится полностью отсортированная последовательность.

#### Деревом

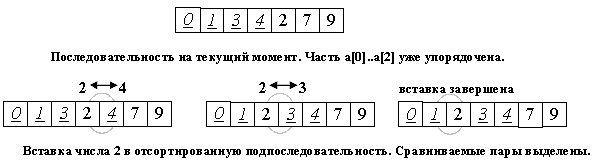
Сортировка с помощью двоичного дерева (сортировка двоичным деревом, сортировка деревом, древесная сортировка, сортировка с помощью бинарного дерева, англ. tree sort) — универсальный алгоритм сортировки, заключающийся в построении двоичного дерева поиска по ключам массива (списка), с последующей сборкой результирующего массива путём обхода узлов построенного дерева в необходимом порядке следования ключей.

#### Пузырьком

Алгоритм состоит из повторяющихся проходов по сортируемому массиву. За каждый проход элементы последовательно сравниваются попарно и, если порядок в паре неверный, выполняется обмен элементов. Проходы по массиву повторяются {\displaystyle N-1}n-1 раз или до тех пор, пока на очередном проходе не окажется, что обмены больше не нужны, что означает — массив отсортирован. При каждом проходе алгоритма по внутреннему циклу, очередной наибольший элемент массива ставится на своё место в конце массива рядом с предыдущим «наибольшим элементом», а наименьший элемент перемещается на одну позицию к началу массива («всплывает» до нужной позиции, как пузырёк в воде, отсюда и название алгоритма).

#### Вставками

На (i+1)-м каждом шаге алгоритма берем a[i+1] и вставляем на нужное место в готовую часть массива.   
Поиск подходящего места для очередного элемента входной последовательности осуществляется путем последовательных сравнений с элементом, стоящим перед ним.  
В зависимости от результата сравнения элемент либо остается на текущем месте(вставка завершена), либо они меняются местами и процесс повторяется.



Таким образом, в процессе вставки мы "просеиваем" элемент x к началу массива, останавливаясь в случае, когда

1. Hайден элемент, меньший x или
2. Достигнуто начало последовательности.

### Поиск

#### Интерполяционный

Данный алгоритм производит предсказание местонахождения элемента: поиск происходит подобно двоичному поиску, но вместо деления области поиска на две части, интерполирующий поиск производит оценку новой области поиска по расстоянию между ключом и текущим значением элемента. Другими словами, бинарный поиск учитывает лишь знак разности между ключом и текущим значением, а интерполирующий ещё учитывает и модуль этой разности и по данному значению производит предсказание позиции следующего элемента для проверки. В среднем интерполирующий поиск производит log(log(N)) операций, где N есть число элементов, среди которых производится поиск. Число необходимых операций зависит от равномерности распределения значений среди элементов. В плохом случае (например, когда значения элементов экспоненциально возрастают) интерполяционный поиск может потребовать до O(N) операций.

#### По дереву

Двоичное дерево поиска похоже на дерево из примера выше, но строится по определенным правилам:

1. У каждого узла не более двух детей.
2. Любое значение меньше значения узла становится левым ребенком или ребенком левого ребенка.
3. Любое значение больше или равное значению узла становится правым ребенком или ребенком правого ребенка.

Средняя сложность поиска по такому дереву O(log n).

#### Линейный

Работа алгоритма заключается в том, что элементы масси-ва, начиная с первого, последовательно сравнивается с искомым элементом. Сравнение элементов продолжается до тех пор, пока не будут просмотрены все элементы или очередной элемент мас-сива не равен искомому.

#### Быстрый линейный

Любой алгоритм поиска содержит блок проверки на окончание массива. В алгоритме линейного поиска эта проверка осуществляется каждый раз перед обращением очередному элементу. Однако проверка на окончание массива может осуществляться не при каждом сравнении. Для этого в конец массива включается (N+1)-й элемент, равный искомому. Тогда проверка на окончание массива осуществляется лишь при совпадении очередного элемента с искомым. Если этот элемент находится внутри массива, то поиск заканчивается удачно и элемент считается найденным. Если же этот элемент оказался (N+1)-ым, то искомого элемента в массиве нет.

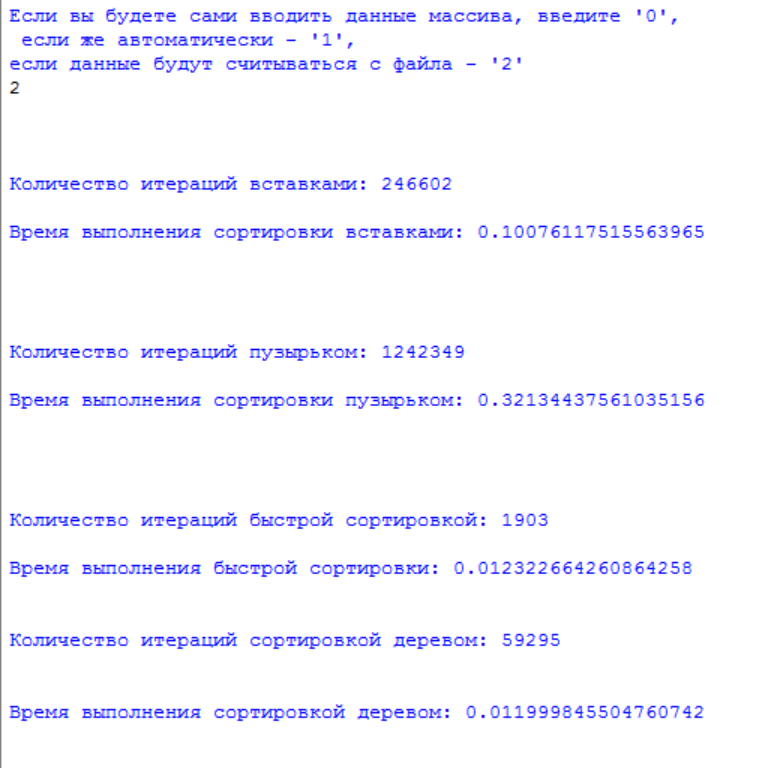
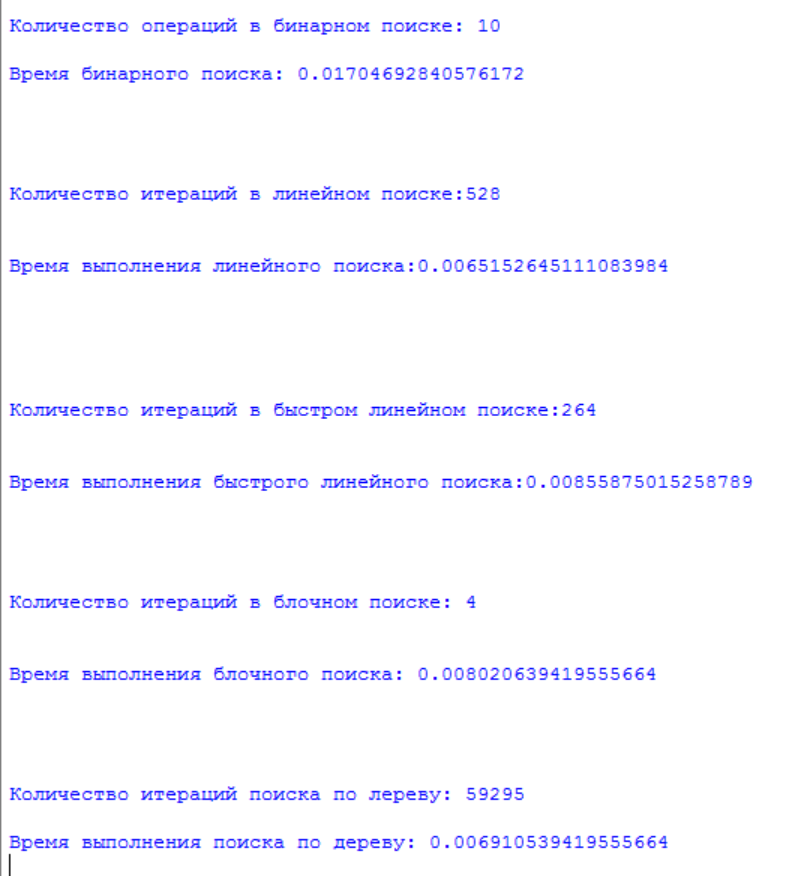
Применяя алгоритм быстрого линейного поиска для поиска элемента в упорядоченном массиве, поиск можно прекратить, если очередной элемент массива будет больше искомого. Это будет означать, что искомого элемента в массиве нет. В случае поиска элемента, который есть в массиве, этот алгоритм аналогичен алгоритму быстрого линейного поиска в неупорядоченном массиве.

# Ход работы

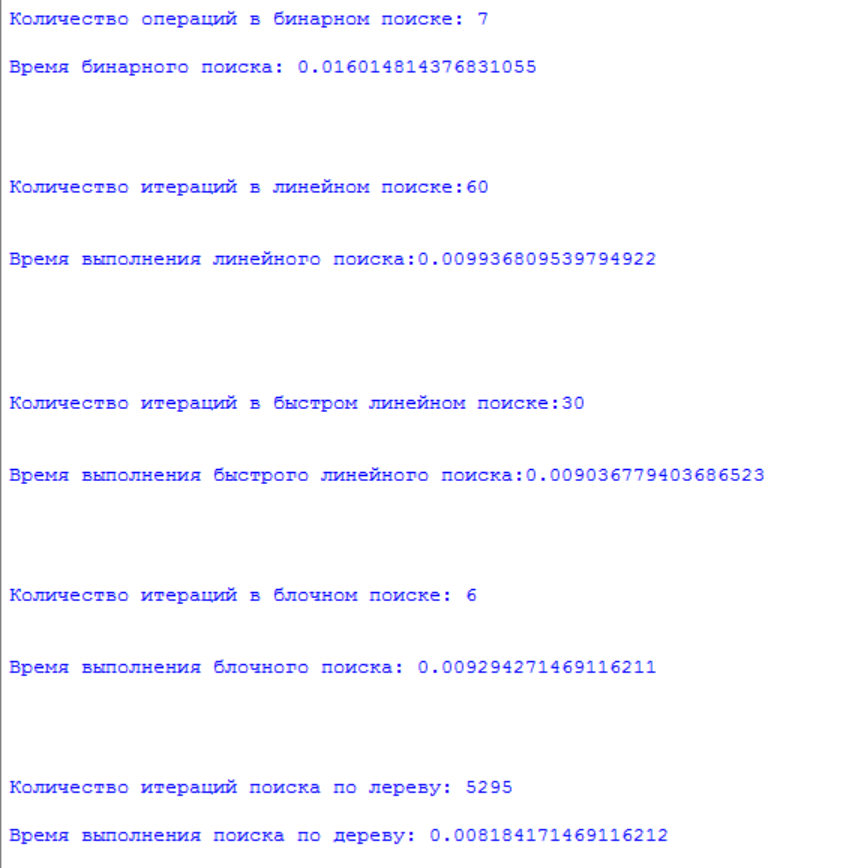
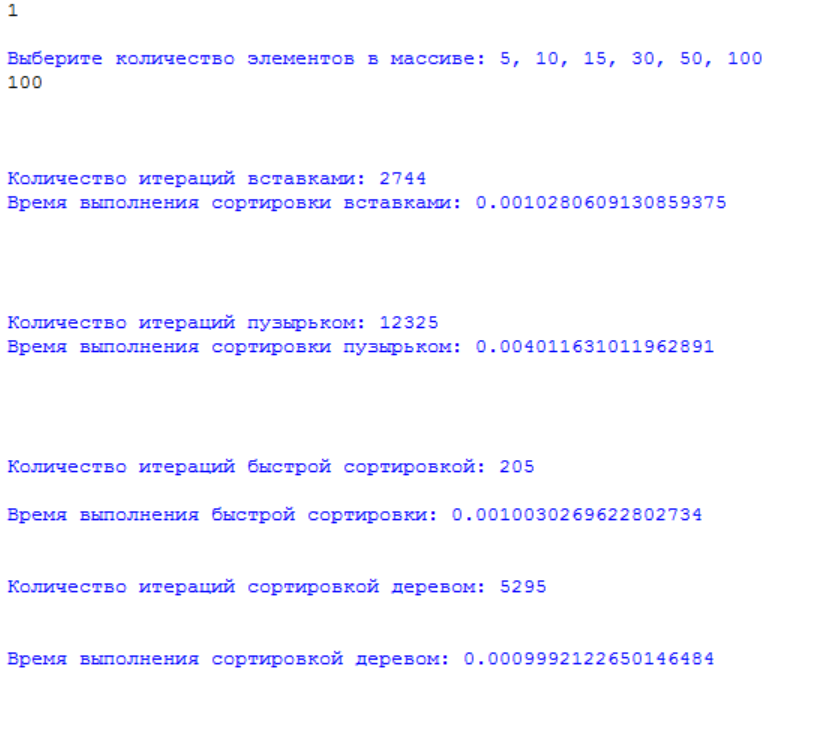
### Написание программы на Python 3 в IDLE

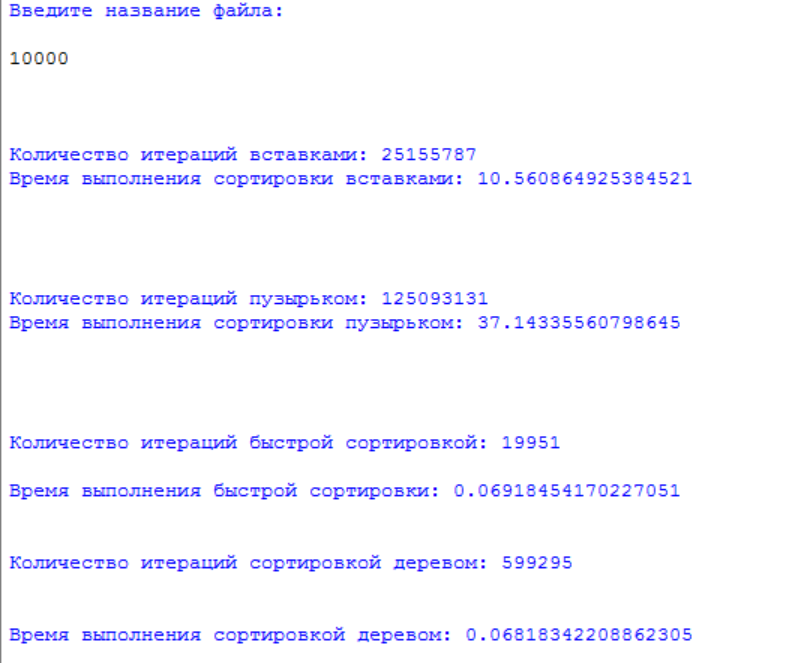
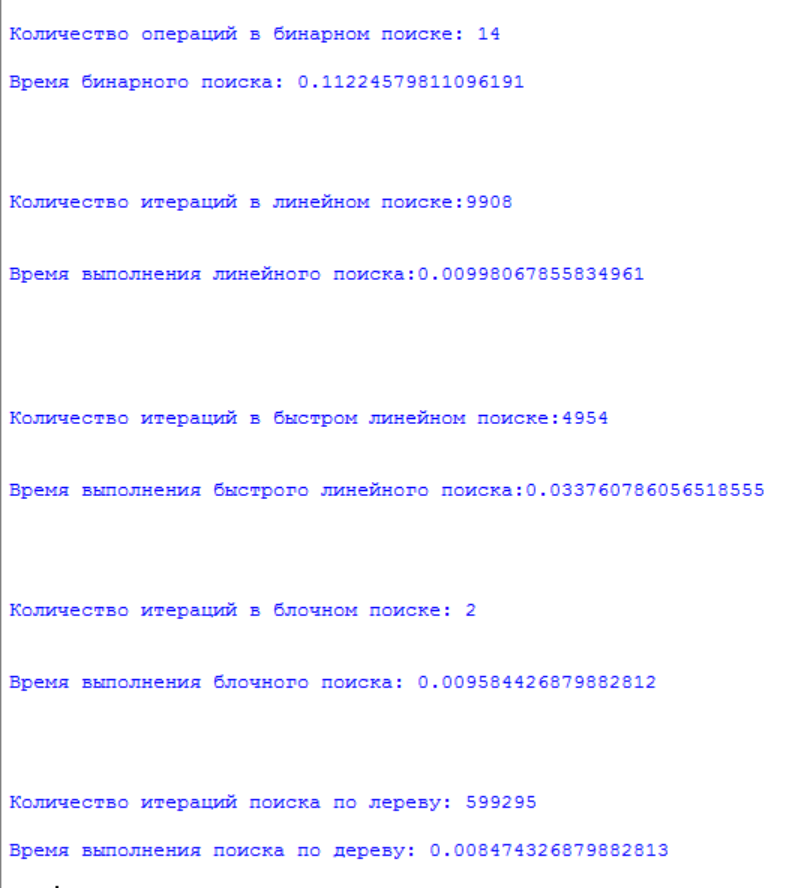
В программе предусмотрены функции для получения данных (из файла, создание случайного набора данных, ввод с клавиатуры), отдельные функции для каждого поиска и сортировки. При запуске программа просит выбрать способ ввода. Прочитав данные, она последовательно выполняет все сортировки и поиски, выводя промежуточные результаты. Статистика выводится о затраченном на выполнение алгоритма времени и количестве операций (сравнения, вхождения в циклы).

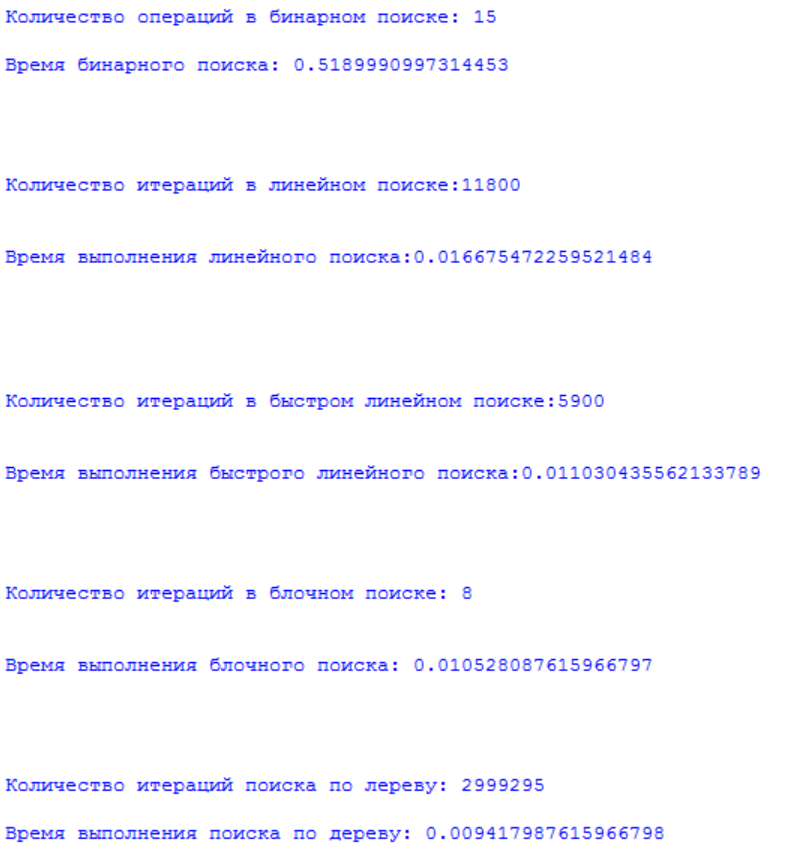
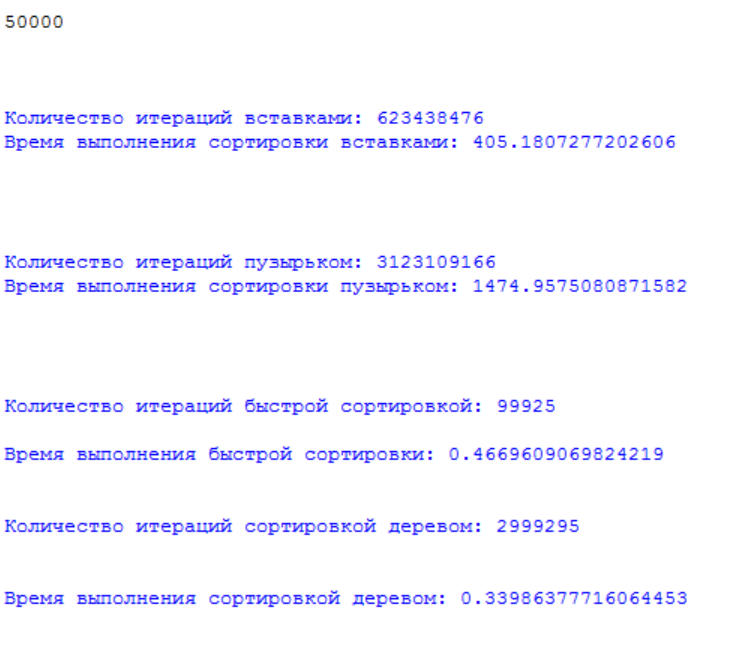
### Тестирование программы

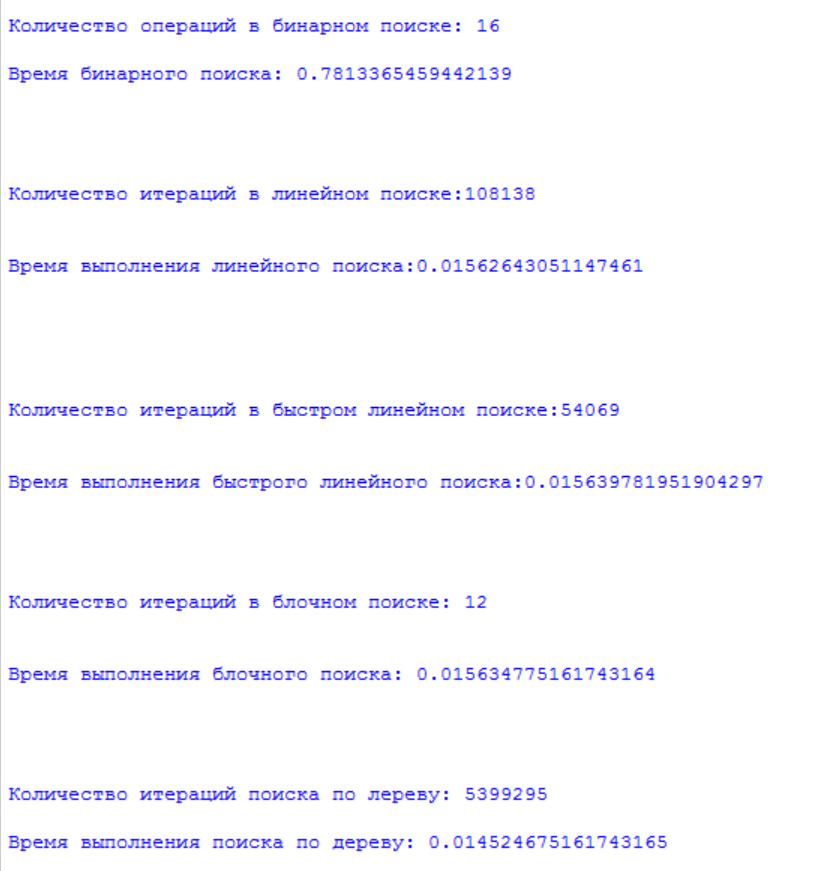
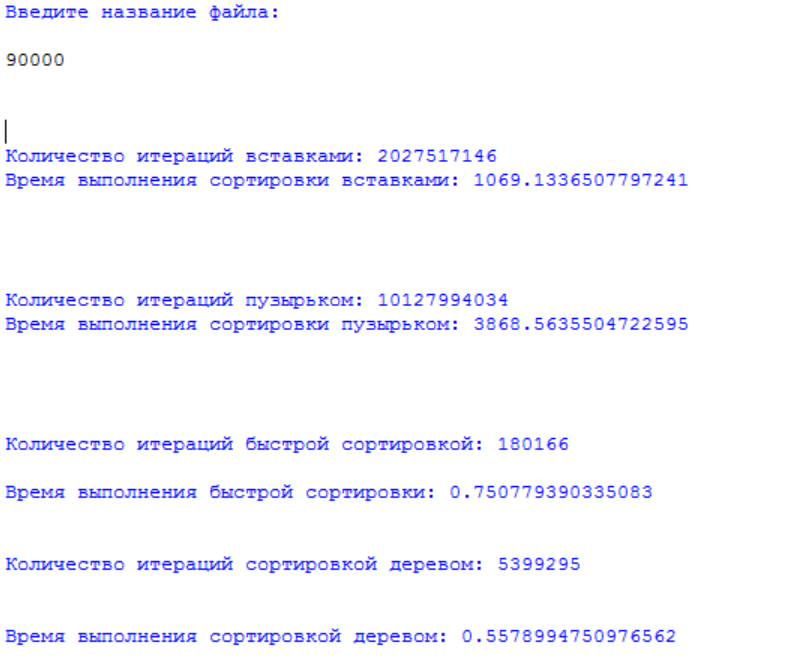
Работа на 1 000 элементах:

Работа на 100 элементах:

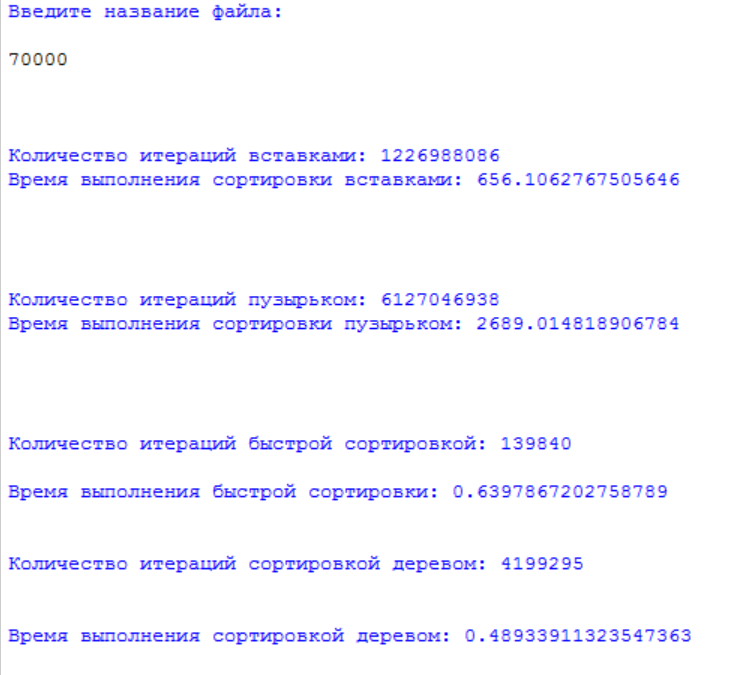
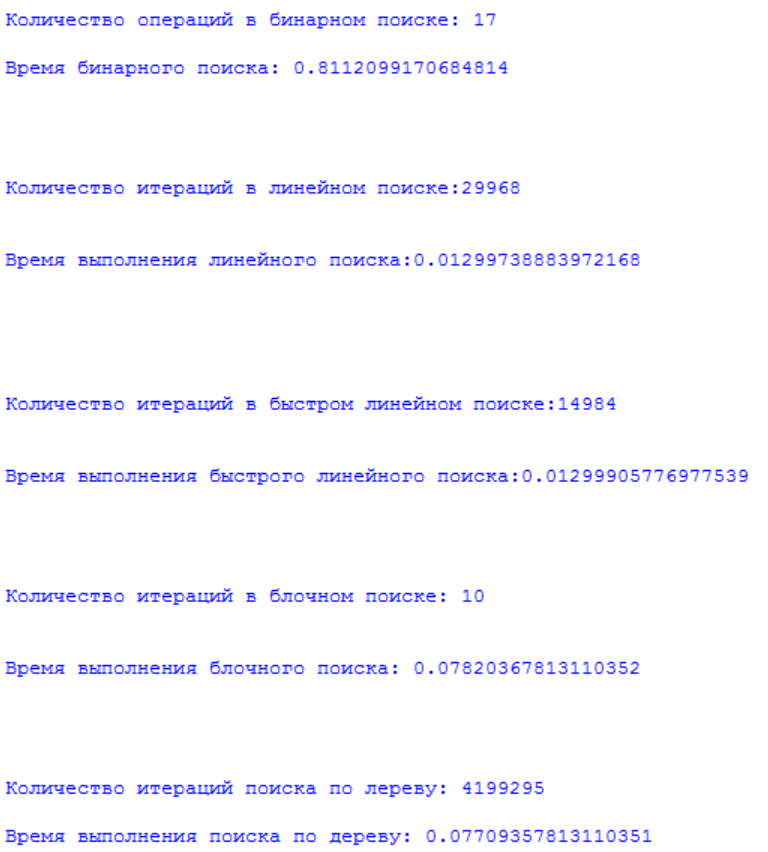


Работа на 10 000 элементов

Работа на 50 000 элементов

Работа на 90 000 элементов

Работа на 70 000 элементов



### Обработка данных

Для каждого значения n (количество элементов) проведено три замера.

### Построение графиков

На основе информации о затраченном времени каждой сортировки и каждого поиска я построила графики зависимости время выполнения программы от количества элементов в ней.

Также я построила графики зависимости времени от количества итераций

# Вывод

В ходе данной работы опытным путем была получена временная сложность наиболее часто используемых алгоритмов.

На небольшом количестве входных значений все алгоритмы работают одинаково эффективно, но для массивов, включающих более 20-30 тысяч элементов имеет смысл выбирать наиболее эффективную сортировку.

В ходе данной работы наиболее быстродействующими сортировками оказались быстрая сортировка Хоара и сортировка по дереву. Их использование позволяет ускорить выполнение программы и избежать лишних вычислений. Самой медленной сортировкой оказалась сортировка пузырьком.

Для поисков же особой разницы по времени обнаружено не было(кроме поиска по дереву). Для входного массива, размер которого не превышает 100 тысяч, выбор алгоритма поиска мало влияет на быстродействие программы. Но следует отметить, что поиск по дереву сильно уступает своим конкурентам из-за необходимости предварительного построения бинарного дерева.

Листинг программы

import csv

import random

import time

############################## Сортировки

def BubbleSort(A): # Сортировка пузырьком

start = time.time()

it = 0

for j in range(len(A) - 1):

it += 1

for i in range(len(A) - 1):

it += 1

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

it +=1

A[i], A[i + 1] = A[i + 1], A[i]

stop = time.time()

sec = stop - start

print("\nКоличество итераций пузырьком: " + str(it))

return sec

global iterationOfQS

iterationOfQS = 0

global BlenItem

BlenItem = 0

def sort2(array2): #быстрая сортировка

global iterationOfQS

iterationOfQS +=1

array = array2

less = []

equal = []

greater = []

if len(array) > 1:

pivot = array[0][1]

for x in array:

if x[1] < pivot:

less.append(x)

if x[1] == pivot:

equal.append(x)

if x[1] > pivot:

greater.append(x)

iterationOfQS +=1

return sort2(less)+equal+sort2(greater)

else:

return array

global iterS

iterS = iterationOfQS

global it

it = 132

def InsertionSort(A): # Cортировка вставками

it = 0

start1 = time.time()

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

it += 1

new\_elem = A[i]

j = i - 1

while j >= 0 and A[j] > new\_elem:

A[j + 1] = A[j]

j -= 1

it +=1

A[j + 1] = new\_elem

stop1 = time.time()

sec1 = stop1 - start1

print("\nКоличество итераций вставками: " + str(it))

return sec1

################ поиски

start1 = time.time()

stop1 = time.time()

sec1 = stop1 - start1

def BinSearchVirt(li, x): # бинарный поиск для отсортированного массива

it = 0

i = 0

j = len(li) - 1

while i < j:

it += 1

m = int((i+j)/2)

if x > li[m][1]:

i = m + 1

else:

j = m

if li[j][1] == x:

print("\nКоличество операций в бинарном поиске: " + str(it) + "\n")

return j

else:

print("\nКоличество операций в бинарном поиске: " + str(it) + "\n")

return 0

def linearSearch(lst, x): # Линейный поиск

it = 0

i = 0

startLin = time.time()

while (i < len(lst) and lst[i][1] != x):

it += 2

i += 1

# stopLin = time.time()

print("\nКоличество итераций в линейном поиске:" + str(it)+'\n')

if i < len(lst):

stopLin = time.time()

secLin = stopLin - startLin

print("\nВремя выполнения линейного поиска:" + str(secLin) +'\n')

return i

else:

stopLin = time.time()

secLin = stopLin - startLin

print("\nВремя выполнения линейного поиска:" + str(secLin) +'\n')

return 0

def linearQSearch(lst, x1): # Быстрый Линейный поиск

it = 0

x = ['0.1']

startLinQ = time.time()

x.append(x1)

lst.append(x)

i = 0

while int(lst[i][1]) != x1:

i += 1

it += 1

print("\nКоличество итераций в быстром линейном поиске:" + str(it)+'\n')

if i < len(lst):

stopLinQ = time.time()

secLinQ = stopLinQ - startLinQ

print("\nВремя выполнения быстрого линейного поиска:" + str(secLinQ) +'\n')

return 1

else:

stopLinQ = time.time()

secLinQ = stopLinQ - startLinQ

print("\nВремя выполнения быстрого линейного поиска:" + str(secLinQ) +'\n')

return 0

def interSearch(lst, x): #Интерполяционный поиск для отсортированного массива

itIS = 0

f = '0'

l = 1

r = len(lst) - 1

i = 0

while (r>=1)and(i<=len(lst)-1):

itIS +=1

i = l+((r-l)\*(x-lst[l][1]))//((lst[r][1]-lst[l][1]))

if x != lst[i][1]:

itIS += 1

l = i + 1

else:

itIS += 1

print("\nКоличество итераций в блочном поиске: " + str(itIS) + "\n")

return 1

######################################## ДЕРЕВО

class TreeNode:

def \_\_init\_\_(self, key, val,left=None,right=None,

parent=None):

self.payload = val

self.key = key

self.leftChild = left

self.rightChild = right

self.parent = parent

def hasLeftChild(self):

return self.leftChild

def hasRightChild(self):

return self.rightChild

def isLeftChild(self):

return self.parent and self.parent.leftChild == self

def isRightChild(self):

return self.parent and self.parent.rightChild == self

def isRoot(self):

return not self.parent

def isLeaf(self):

return not (self.rightChild or self.leftChild)

def hasAnyChildren(self):

return self.rightChild or self.leftChild

def hasBothChildren(self):

return self.rightChild and self.leftChild

def replaceNodeData(self,value,lc,rc):

self.payload = value

self.leftChild = lc

self.rightChild = rc

if self.hasLeftChild():

self.leftChild.parent = self

if self.hasRightChild():

self.rightChild.parent = self

class BinarySearchTree:

def \_\_init\_\_(self):

self.root = None

self.size = 0

def length(self):

return self.size

def \_\_len\_\_(self):

return self.size

def \_\_iter\_\_(self):

return self.root.\_\_iter\_\_()

def put(self,key,val):

if self.root:

self.\_put(key,val,self.root)

else:

self.root = TreeNode(key,val)

self.size = self.size + 1

def \_put(self,key,val,currentNode):

global BlenItem

global iterS

iterS += 1

sortTreeC = True

if BlenItem > 15:

iterS +=1

sortTreeC = False

if key < currentNode.key:

iterS += 1

if currentNode.hasLeftChild():

iterS +=1

self.\_put(key,val,currentNode.leftChild)

else:

iterS += 1

currentNode.leftChild = TreeNode(key,val,parent=currentNode)

else:

iterS += 1

if currentNode.hasRightChild():

iterS += 1

self.\_put(key,val,currentNode.rightChild)

else:

iterS += 1

if sortTreeC == True:

iterS +1

currentNode.rightChild = TreeNode(key,val,parent=currentNode)

else: return None

def \_\_setitem\_\_(self,k,v):

global BlenItem

BlenItem +=1

self.put(k,v)

def get(self,key):

if self.root:

res = self.\_get(key,self.root)

if res:

return res.payload

else:

return None

else:

return None

def \_get(self,key,currentNode):

if not currentNode:

return None

elif currentNode.key == key:

return currentNode

elif key < currentNode.key:

return self.\_get(key,currentNode.leftChild)

else:

return self.\_get(key,currentNode.rightChild)

def findEl(self, value):

current = self

global it

while current.hasLeftChild():

it += 12

current = current.leftChild

print(it)

self.\_\_iter\_\_()

return current

def \_\_iter\_\_(self):

global it

if self:

it += 2

if self.hasLeftChild():

it += 1

for elem in self.leftChiLd:

it += 1

if self.payload == value:

it += 1

return self.key

yield elem

yield self.key

if self.hasRightChild():

it += 10

if self.paylosd == value:

it += 10

return self.key

for elem in self.rightChild:

yield elem

def getFinditem(self, value):

self.\_\_iter\_\_()

return 0

def getSortitem(self,key):

return self.get(key)

def \_\_contains\_\_(self,key):

if self.\_get(key,self.root):

return True

else:

return False

########################################

def WriteArray(arr):

f = open('massive.txt', 'w')

for i in range(0, len(arr)-1):

f.write(str(arr[i][0]) + '\n' + str(arr[i][1]) + '\n')

f.close()

def ReadArray():

f = open('massive.txt', 'r')

arr = []

b1 = [0, 0]

arr1 = []

arr1 = [line.strip() for line in f]

for i in range(0, len(arr1), 2):

arr.append(arr1[i:i+2])

for i in range(0, len(arr)):

arr[i][0] = float(arr[i][0])

arr[i][1] = int(arr[i][1])

return arr

def RandomEl(a):

b = 0

while (b < 5):

b = (random.randint(0, len(a)-1))

element = a[b][1]

print("индекс: " + str(b) + "\n")

return element

a = []

m =[]

# Генерация массива

y = input("Если вы будете сами вводить данные массива, введите '0',\n если же автоматически - '1',\nесли данные будут считываться с файла - '2'\n")

if y == '0':

x = input("Введите количество элементов массива\n")

for i in range(int(x)):

b = [0.1]

text = input("\nВведите элемент\n")

b.append(int(text))

a.append(b)

m.append(b)

b.pop()

elif y == '1':

x = input("\nВыберите количество элементов в массиве: 5, 10, 15, 30, 50, 100\n")

for i in range(int(x)):

b = [0.1]

b.append(random.randint(1, 16000))

a.append(b)

m.append(b)

elif y == '2':

print("Введите название файла: \n")

text = input()

with open(text+".csv") as f:

reader = csv.reader(f)

for row in reader:

a.append(row)

m.append(row)

else:

print("\nНеправильно\n")

exit(0)

WriteArray(a)

print("\n")

print("Время выполнения сортировки вставками: " + str(InsertionSort(a)) + "\n")

a = []

a = ReadArray()

print("\n")

print("Время выполнения сортировки пузырьком: " + str(BubbleSort(a)) + "\n")

a = []

a = ReadArray()

print("\n")

start6 = time.time()

sort2(a)

stop6 = time.time()

print("\nКоличество итераций быстрой сортировкой: " + str(iterationOfQS))

print("\nВремя выполнения быстрой сортировки: "+str(stop6-start6) + "\n")

a = []

a = ReadArray()

myTree = BinarySearchTree()

for i in range(0, len(a)-1):

myTree.\_\_setitem\_\_(i, a[i][1])

StartTS = time.time()

for i in range(0, len(a)-1):

myTree.getSortitem(i)

StopTS = time.time()

secTs = StopTS - StartTS

print("\nКоличество итераций сортировкой деревом: " + str(iterS) + "\n")

print("\nВремя выполнения сортировкой деревом: " + str(secTs) + "\n")

print("\n\nПроверка поисков\n")

el = RandomEl(a)

print("Значение искомое: "+str(el) + "\n")

print('\n\n')

startBin = time.time()

ses = 0.0011101

sort2(a)

BinSearchVirt(a, el)

stopBin = time.time()

secBin = stopBin - startBin

print("Время бинарного поиска: " + str(secBin) + "\n")

print('\n')

linearSearch(a, el)

print('\n\n')

sort2(a)

linearQSearch(a, el)

print("\n")

sort2(a)

start = 0

stop = 0

b = sort2(a)

start = time.time()

interSearch(b, el)

stop = time.time()

sec = stop - start

print("\nВремя выполнения блочного поиска: " + str(sec) + "\n")

print("\n")

sec -= ses

start = 0

stop = 0

start = time.time()

myTree.getFinditem(el)

stop = time.time()

ses = stop - start

print("\nКоличество итераций поиска по лереву: " + str(iterS))

print("\nВремя выполнения поиска по дереву: " + str(sec) + "\n")