**Федеральное агентство связи  
Ордена Трудового Красного Знамени федеральное государственное бюджетное образовательное учреждение высшего образования   
«Московский технический университет связи и информатики»**

Кафедра «Математическая кибернетика и информационные технологии»

**Курсовая работа**

По дисциплине «Структуры и алгоритмы обработки данных»   
Вариант №6

Выполнила: Гердт Таисия

Группа: БСТ1955

Руководитель: Кутейников Иван Алексеевич

## Лабораторная работа №1

Реализовать пирамидальный метод сортировки строк числовой матрицы. Добавить реализацию быстрой сортировки (quicksort). Оценить время работы каждого алгоритма сортировки и сравнить его со временем стандартной функции сортировки, используемой в выбранном языке программирования.

В качестве языка для реализации алгоритмов был выбран язык Python.

Код программы:

#функция преобразования бинарного дерева в кучу(сортировка)

def heapify(a, n, i):

mah = i

l = 2\*i+1

r = 2\*i+2

if l<n and a[i]<a[l]:

mah = l

if r<n and a[mah]<a[r]:

mah = r

if mah!=i:

a[i],a[mah]=a[mah],a[i]

heapify(a, n, mah)

#функция сортировки кучи

def heapSort(a):

n = len(a)

for i in range(n, -1, -1):

heapify(a, n, i)

for i in range (n-1, 0, -1):

a[i], a[0] = a[0], a[i]

heapify(a, i, 0)

#функция сортировки отельной части массива

def partition(arr, low=0, high=None):

if high == None: high = len(arr)

i = (low-1)

pivot = arr[high]

for j in range(low, high):

if arr[j] <= pivot:

i = i+1

arr[i], arr[j] = arr[j], arr[i]

arr[i+1], arr[high] = arr[high], arr[i+1]

return (i+1)

#функция быстрой сортировки («прыжками» в разные стороны)

def quickSort(arr, low=0, high=None):

if high == None: high = len(arr)-1

if len(arr) == 1:

return arr

if low < high:

pi = partition(arr, low, high)

quickSort(arr, low, pi-1)

quickSort(arr, pi+1, high)

#сортировка каждой строки массива

def matrixSort(a, func):

for i in range(len(a)):

func(a[i])

#основная функция

import random, time

mat = [[random.randint(0, 100000) for x in range(100)] for k in range(1000)]

copy = mat

#print("Generated array:")

#for row in mat: print(row)

t = time.time()

matrixSort(mat, heapSort)

t = time.time() - t

print(f"HeapSort execution time: {1000\*t} milliseconds")

#print("\nSorted array by heapSort:")

#for row in mat: print(row)

mat = copy

t = time.time()

matrixSort(mat, quickSort)

t = time.time() - t

print(f"QuickSort execution time: {1000\*t} milliseconds")

#print("\nSorted array by quickSort:")

#for row in mat: print(row)

mat = copy

t = time.time()

matrixSort(mat, sorted)

t = time.time() - t

print(f"Sorted execution time: {1000\*t} milliseconds")

#print("\nSorted array by Python sorted():")

#for row in mat: print(row)

Выполнение программы:

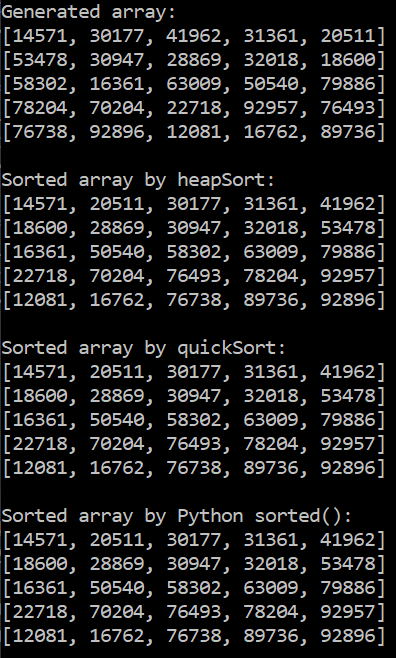


Рисунок 1. Выполнение программы для матрица размерности 5х5

Составим таблицу времени выполнения сортировок:

Таблица 1. Время выполнения сортировок

|  |  |  |  |
| --- | --- | --- | --- |
| **Сортировка\размер** | **10x10** | **100x100** | **1000x100** |
| **HeapSort** | 24.9324 | 517.6156 | 343.1122 |
| 15.9569 | 382.0088 | 432.8136 |
| 16.9826 | 445.8110 | 420.8791 |
| **Среднее** | 19.2906 | 448.4785 | 398.9350 |
| **QuickSort** | 22.9383 | 1207.7992 | 1077.1315 |
| 20.9448 | 1145.9339 | 1067.1463 |
| 18.9505 | 1470.0372 | 1096.0648 |
| **Среднее** | 20.9445 | 1274.5901 | 1080.1142 |
| **Sorted** | 0.9978 | 1.9660 | 1.9543 |
| 0.9983 | 0.9704 | 1.9953 |
| 0.9701 | 1.9953 | 1.9665 |
| **Среднее** | 0.9887 | 1.6439 | 1.9720 |

При быстрой сортировке матрицы размером 100х1000 достигается максимальная глубина рекурсии.

**Вывод:** Лучший результат выполнения сортировки массива показал встроенный метод sorted(), худший – QuickSort.

## Лабораторная работа №2

Реализовать заданный метод поиска в соответствии с индивидуальным заданием. Организовать генерацию начального набора случайных данных. Для всех вариантов добавить реализацию добавления, поиска и удаления элементов. Оценить время работы каждого алгоритма поиска и сравнить его со временем работы стандартной функции поиска, используемой в выбранном языке программирования.

В качестве языка для реализации алгоритмов был выбран язык Python.

Код программы:

#класс – узел дерева, хранит в себе значение текущего элемента и указание на левый(меньший) и правый(больший) элемент

class Node():

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

self.left = None

self.right = None

self.val = key

#метод поиска значение в узлах дерева

def search(root, key):

if root is None or root.val == key:

return root

if root.val < key:

return search(root.right,key)

return search(root.left,key)

#метод поиска узла дерева с минимальным значением

def min(root):

if root.left is None: return root

else: return min(root.left)

#метод удаления узла из дерева

def delete(root, key):

if root is None:

return root

if key < root.val:

root.left = delete(root.left, key)

elif(key > root.val):

root.right = delete(root.right, key)

else:

if root.left is None:

temp = root.right

root = None

return temp

elif root.right is None:

temp = root.left

root = None

return temp

temp = min(root.right)

root.val = temp.val

root.right = delete(root.right, temp.val)

return root

#метод вставки значения в дерево

def insert(root, key):

if root is None:

return Node(key)

else:

if root.val == key:

return root

elif root.val < key:

root.right = insert(root.right, key)

else:

root.left = insert(root.left, key)

return root

#метод – вывод узлов дерева в порядке возрастания

def inorder(root):

if root:

inorder(root.left)

print(root.val)

inorder(root.right)

import random, time

a = random.sample(range(-2147483648, 2147483647), 100000)

#print(f"Generated array:")

#for i in a: print(i)

b = Node(a[0]) #первый сгенерированного массива задаем корнем дерева

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

b = insert(b, a[i]) #каждый следующий элемент массива вставляем в дерево

#print(f"\nBinary tree:")

#inorder(b)

k = a[-1]

t = time.time()

search(b, k)

t = (time.time() - t)\*1000

print(f"Binary search tree execution time: {t\*1000} milliseconds")

t = time.time()

a.index(k)

t = (time.time() - t)\*1000

print(f"Python search execution time: {t\*1000} milliseconds")

Выполнение программы:

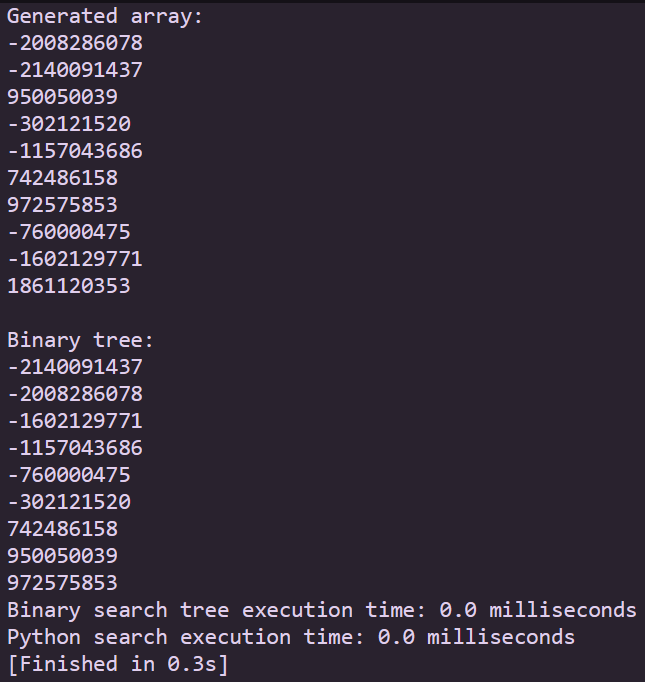


Рисунок 1. Результат выполнения программы с массивом размера 10

Составим таблицу времени выполнения поиска:

Таблица 1. Время выполнения поиска

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Поиск\размер** | **1000** | **10000** | **100000** | **100000** |
| **BinarySort** | 0.0000 | 0.0000 | 0.0000 | 0.0000 |
| 0.0000 | 0.0000 | 0.0000 | 997.5433 |
| 0.0000 | 0.0000 | 0.0000 | 0.0000 |
| **Среднее** | 0.0000 | 0.0000 | 0.0000 | 332.5144 |
| **Index** | 997.5433 | 997.5433 | 2993.3453 | 19947.5288 |
| 0.0000 | 996.8281 | 2993.1068 | 18949.7471 |
| 0.0000 | 998.0202 | 2989.7690 | 20946.0258 |
| **Среднее** | 332.5144 | 997.4639 | 2992.0737 | 19947.7673 |

Так как для генерации бинарного дерева требуется большое количество времени, выполнение самой программы заметно увеличивается с увеличением размера массива:

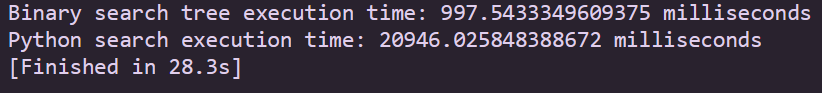


Рисунок 2. Сравнение времени выполнения поиска и основной программы

В данном случае, программа выполнялась без вывода массивов на экран, потому временем, затраченным на выполнение остальных команд, можно пренебречь: для генерации дерева потребовалось около 7 секунд, для поиска элемента в нем – менее секунды, для поиска в массиве – около 21 секунды.

**Вывод:** Поиск в бинарном дереве показал значительное преобладание в скорости над стандартной функцией поиска особенно с увеличением количества данных в массиве, но требует время на генерацию бинарного дерева.

## Лабораторная работа №3

Реализовать метод Кнута-Морриса-Пратта поиска подстроки в строке в соответствии с индивидуальным заданием. Для всех вариантов добавить реализацию добавления строк, ввода подстроки и поиска подстроки. Предусмотреть возможность существования пробела. Ввести опцию чувствительности / нечувствительности к регистру. Оценить время работы каждого алгоритма поиска и сравнить его со временем работы стандартной функции поиска, используемой в выбранном языке программирования.

Код программы:

#функция КМП

def kmp(pat, txt):

M = len(pat)

N = len(txt)

lps = [0]\*M

j = 0

lsp(pat, M, lps)

i = 0

flag = 0

while i < N:

if pat[j] == txt[i]:

i += 1

j += 1

if j == M:

print ("KMP: Found pattern at index " + str(i-j))

j = lps[j-1]

flag += 1

elif i < N and pat[j] != txt[i]:

if j != 0:

j = lps[j-1]

else:

i += 1

if flag == 0:

print("KMP: Pattern is not found")

#функция нахождения массива длиннейших префиксов являющихся суффиксами

def lsp(pat, M, lps):

len = 0

lps[0]

i = 1

while i < M:

if pat[i]== pat[len]:

len += 1

lps[i] = len

i += 1

else:

if len != 0:

len = lps[len-1]

else:

lps[i] = 0

i += 1

import time, random, string

flag = input("Enter 1 for case-sensitive search\nEnter 2 for non-case sensetive search\nEnter 0 for quit\n")

while(True):

if flag == "1":

#with open("verylongstr.txt", mode="r", encoding="utf-8") as f:

# for line in f:

# txt = line

txt = input("Enter string:\n")

pat = input("Enter pattern:\n")#"myUniquePattern"

t = time.time()

kmp(pat, txt)

t = time.time() - t

print(f"KMP execurtion time: {t\*1000} milliseconds")

t = time.time()

m = txt.index(pat)

t = time.time() - t

print(".index(): Found pattern at index ", m)

print(f".index() execurtion time: {t\*1000} milliseconds")

elif flag == "2":

txt = input("Enter string:\n").lower()

pat = input("Enter pattern:\n").lower()

t = time.time()

kmp(pat, txt)

t = time.time() - t

print(f"KMP execurtion time: {t\*1000} milliseconds")

t = time.time()

m = txt.index(pat)

t = time.time() - t

print(".index(): Found pattern at index ", m)

print(f".index() execurtion time: {t\*1000} milliseconds")

elif flag == "0":

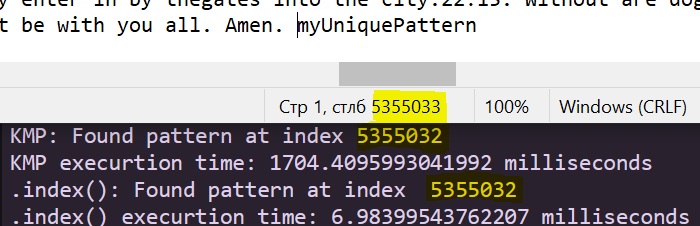
quit()

else:

print("Invalid command")

flag = input("Enter 1 for case-sensitive search\nEnter 2 for non-case sensetive search\nEnter 0 for quit\n")

Для проверки правильности работы программы использовали текстовый файл в 5355032 символа, в конце которого единожды встречается «myUniquePattern»:



Так в программе первый символ имеет нулевой индекс, индексы найденного паттерна отличаются на 1.

Составим таблицу времени выполнения поиска:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Поиск\размер** | **44661** | **531160** | **1275785** | **2579515** | **5355047** |
| **KMP** | 18.9614 | 174.5324 | 375.0100 | 740.0553 | 1704.4096 |
| 20.9434 | 169.5590 | 412.9248 | 749.0301 | 1893.9054 |
| 11.9879 | 190.5267 | 416.8835 | 730.0839 | 1642.6046 |
| **Среднее** | 17.2976 | 178.2060 | 401.6061 | 739.7231 | 1746.9732 |
| **Скорость обработки** | 2581.9210 | 2980.5947 | 3176.7074 | 3487.1358 | 3065.3287 |
| **Index** | 0.0000 | 0.0000 | 0.9630 | 2.9881 | 6.9840 |
| 0.0000 | 0.0000 | 0.9682 | 2.9600 | 4.9877 |
| 0.0000 | 0.9990 | 0.9983 | 1.9584 | 5.0151 |
| **Среднее** | 0.0000 | 0.3330 | 0.9765 | 2.6355 | 5.6623 |
| Скорость обработки | #ДЕЛ/0!(быстро!) | 1595116.83 | 1306510 | 978765.2 | 945740.07 |

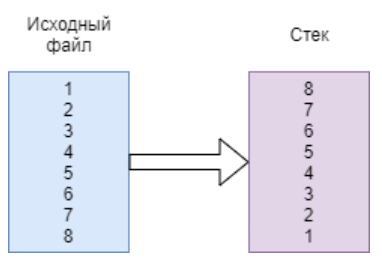
**Вывод:** Стандартная функция поиска подстроки в строке показала лучшие результаты, чем КМП алгоритм.

## Лабораторная работа №4

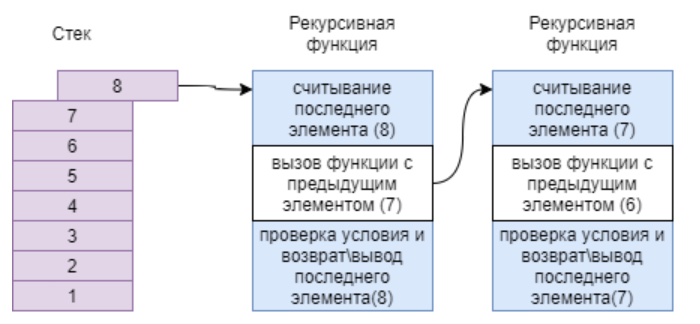
Используя технологию модульного программирования разработать программу обработки данных, содержащихся в заранее подготовленном файле, в соответствии с индивидуальным заданием. Применить динамическую структуру указанного в задании вида: стек, очередь или дек. Программа должна включать модуль, содержащий набор всех необходимых средств (типов, подпрограмм и т.д.) для решения поставленной задачи.

Вариант 6. Дан файл из вещественных чисел. Используя стек за один просмотр файла напечатать сначала все числа, меньшие a, затем все числа из интервала [a,b], и, наконец, все остальные числа, сохраняя исходный порядок в каждой группе.

Стек (от англ. stack — стопка) — структура данных, представляющая из себя упорядоченный набор элементов, в которой добавление новых элементов и удаление существующих производится с одного конца, называемого вершиной стека (по принципу LIFO, от англ. last in — first out, «последним пришёл — первым вышел»).



Так как задание требуется выполнить за один просмотр, данные в стеке будут храниться в обратном порядке. Для того, чтобы выводить данные в прямом порядке, воспользуемся рекурсивной функцией:



Для реализации стека создадим новый класс, хранящий массив данных, имеющий методы добавления и получения элементов, метод проверки на пустоту. В рекурсивную функцию будем отправлять стек и лямбду-функцию, задающую проверку выполнения условия.

Код программы:

# новый класс для стека

class Stack():

#инициализация стека - создание пустого массива для хранения значений

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

self.array = []

#метод стека – проверка на пустоту массива внутри класса

def empty(self):

if len(self.array) == 0:

return True

else:

return False

#метод стека – добавление нового элемента к массиву

def put(self, item):

self.array.append(item)

#метод стека – вывод последнего элемента и его удаление в массиве

def get(self):

return self.array.pop()

#метод – вывод всех элементов массива из stack, удовлетворяющих условию func (с последнего элемента до первого, при помощи рекурсии)

def pop(stack, func):

if not stack.empty():

num = stack.get()

pop(stack, func)

if func(num):

print(num)

else:

stack.put(num)

mystack = Stack()

a = float(input("Enter a:\n"))

b = float(input("Enter b:\n"))

#чтение чисел из файла без переноса строки и пробелов и их запись в стек

with open("file.txt", mode="r", encoding="utf-8-sig") as f:

for lines in f:

for l in lines.rstrip('\n').split(' '):

mystack.put(float(l))

print(f"\nLess than {a}:")

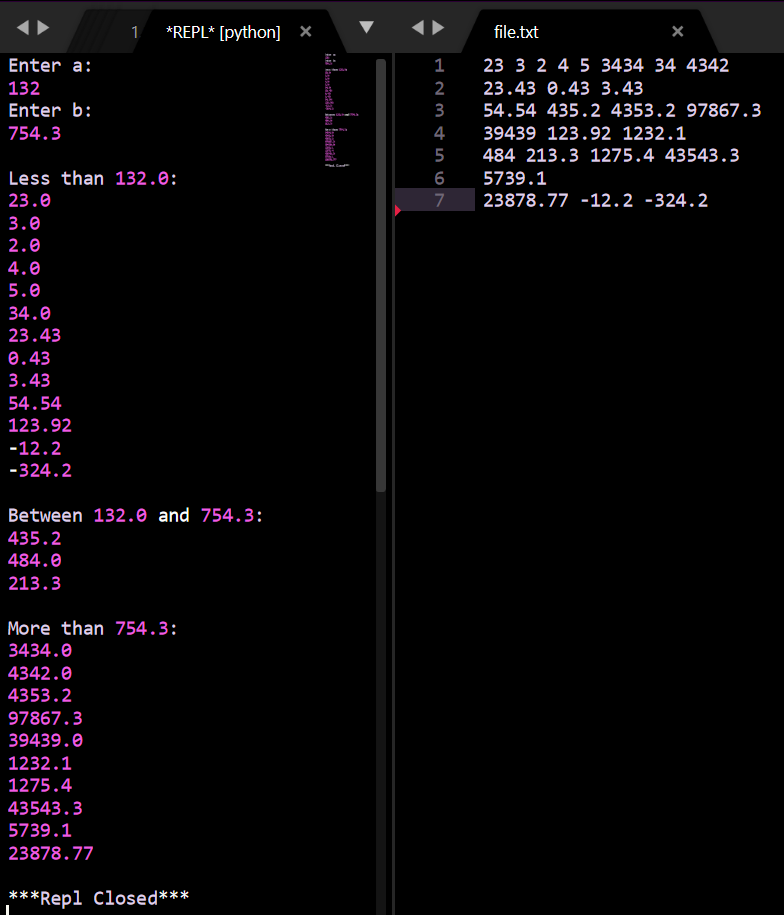
pop(mystack, lambda x: x < a)

print(f"\nBetween {a} and {b}:")

pop(mystack, lambda x: x <= b)

print(f"\nMore than {b}:")

pop(mystack, lambda x: x > b)

Результаты выполнения программы:

**Вывод**: Реализация стека осуществляется с помощью рекурсии; с помощью стека за один просмотр возможно осуществить вывод элементов, удовлетворяющих заданным условиям.