# МИНОБРНАУКИ РОССИИ

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

# «ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)

**Кафедра МО ЭВМ**

# ОТЧЕТ

**по лабораторной работе №2**

# по дисциплине «Алгоритмы и структуры данных» Тема: Реализация и исследование алгоритма разработки TimSort

|  |  |
| --- | --- |
| Студент гр. 2384 | Кузьминых Е.М |
| Преподаватель | Иванов Д.В. |

Санкт-Петербург 2023

# Цель работы.

Написать реализацию алгоритма TimSort на языке программирования Python.

# Задачи.

Реализация

Имеется массив данных для сортировки int arr[] размера n.

Необходимо отсортировать его алгоритмом сортировки Timsort по следующему критерию: по наименьшему значению квадрата элемента (в случае равенства значений элементов в квадрате - сортировка происходит по убыванию).

Так как Timsort - это гибридный алгоритм, содержащий в себе сортировку слиянием и сортировку вставками, то вам предстоит использовать оба этих алгоритма. Поэтому нужно выводить разделённые блоки, которые уже отсортированы сортировкой вставками.

Исследование

После успешного решения задачи в рамках курса проведите исследование данной сортировки на различных размерах данных (10/1000/100000), сравнив полученные результаты с теоретической оценкой (для лучшего, среднего и худшего случаев), и разного размера min\_run. Результаты исследования предоставьте в отчете.

Примечание:

Нельзя пользоваться готовыми библиотечными функциями для сортировки, нужно сделать реализацию сортировки вручную.

0 < n < 65

Обратите внимание на пример. (min\_run = 32)

# Выполнение работы.

Данный код реализует алгоритм TimSort, модификацию стабильного сортировочного алгоритма, объединяющего сортировку вставками и сортировку слиянием.

Функция sorted\_insertion(items, start, end):

Эта функция выполняет сортировку вставками на подмассиве items[start:end+1] основного списка items. Она сортирует элементы так, чтобы их квадраты были упорядочены в возрастающем порядке. Сначала функция определяет "ключевой" элемент key\_item, а затем сравнивает его со всеми предыдущими элементами. Если квадрат ключевого элемента меньше квадрата сравниваемого элемента, или если квадраты равны, но ключевой элемент больше, тогда она "сдвигает" сравниваемые элементы к концу списка до тех пор, пока не найдет подходящее место для key\_item (или не достигнет начала списка).

Функция combine\_parts(arr, left, middle, right):

Эта функция реализует слияние двух подмассивов arr[left:middle+1] и arr[middle+1:right+1] в один отсортированный подмассив, применяя аналогичное сравнение, как и sorted\_insertion. Выбирается один элемент из каждого подмассива и сравниваются их квадраты. Меньший (или больший при равных квадратах) элемент добавляется во временный массив temp\_result, а указатель на выбранный подмассив увеличивается на 1. Эта операция повторяется до тех пор, пока оба подмассива не будут полностью обработаны. После этого содержимое temp\_result копируется обратно в arr. 3.

Функция run\_tim\_sort(items):

Эта функция осуществляет реализацию алгоритма TimSort на входном списке items. Он начинает с создания отсортированных сегментов фиксированного размера (32 элемента), используя функцию sorted\_insertion, и выводит обработанные части на экран. Затем он объединяет эти сегменты вдвое большим размером с помощью функции combine\_parts. Процесс повторяется до тех пор, пока весь список не станет отсортированным.

После программа запрашивает у пользователя число элементов и список этих элементов, которые необходимо сортировать, и запускает на нем функцию run\_tim\_sort. После завершения сортировки выводится отсортированный список.

**Исследование работы алгоритма TimSort:**

В ходе исследования была замерена скорость работы алгоритма TimSort с разными значениями min\_run.

|  |  |  |  |
| --- | --- | --- | --- |
| Знач.min\_run/Кол-во элементов | 10 | 1000 | 100000 |
| 4 | 0 | 0.010608911514282227 | 2.0430123805999756 |
| 8 | 0 | 0.012396812438964844 | 1.9237785339355469 |
| 16 | 0 | 0.011246919631958008 | 1.9720726013183594 |
| 32 | 0 | 0.011102437973022461 | 2.0293829441070557 |
| 64 | 0 | 0.014513015747070312 | 2.497995615005493 |

Таблица 1 – время сортировки массивов

# Тестирование.

В ходе написания лабораторной работы для тестирования был создан файл *tests.py* с функциями, проверяющими работоспособность программы. Тесты проводились с помощью *pytest.*

|  |  |  |  |
| --- | --- | --- | --- |
| № | Входные данные | Выходные данные | Комментарий |
| 1 | def test\_short\_array():  array = [0,-1,1]  run\_tim\_sort(array)  assert array==[0,1,-1] | [0,1,-1]== [0,1,-1] | Ответ верный |
| 2 | def test\_len(x = 1000):  array = []  for i in range(x):  array.append(random.randint(-100,100))  run\_tim\_sort(array)  assert len(array) == x | 1000==1000 | Ответ верный |

**Вывод.**

В ходе выполнения лабораторной работы был реализован алгоритм сортировки TimSort. Произведен анализ скорости сортировки алгоритма в зависимости от размера значения min\_run, было произведено тестирование программы с помощью pytest.

# Приложение А.

**Исходный код программы.**

**Main.py**

def combine\_parts(arr, left, middle, right):

pointers = [left, middle + 1]

temp\_result = []

while pointers[0] <= middle and pointers[1] <= right:

if arr[pointers[0]] \*\* 2 < arr[pointers[1]] \*\* 2 or (

arr[pointers[0]] \*\* 2 == arr[pointers[1]] \*\* 2 and arr[pointers[0]] > arr[pointers[1]]):

temp\_result.append(arr[pointers[0]])

pointers[0] += 1

else:

temp\_result.append(arr[pointers[1]])

pointers[1] += 1

while pointers[0] <= middle:

temp\_result.append(arr[pointers[0]])

pointers[0] += 1

while pointers[1] <= right:

temp\_result.append(arr[pointers[1]])

pointers[1] += 1

for i, item in enumerate(temp\_result):

arr[left + i] = item

def sorted\_insertion(items, start, end):

for pos in range(start + 1, end + 1):

key\_item = items[pos]

j = pos - 1

while j >= start and (

key\_item \*\* 2 < items[j] \*\* 2 or (key\_item \*\* 2 == items[j] \*\* 2 and key\_item > items[j])):

items[j + 1] = items[j]

j -= 1

items[j + 1] = key\_item

def run\_tim\_sort(items):

n = len(items)

min\_run = 32

for start in range(0, n, min\_run):

end = min(start + min\_run - 1, n - 1)

sorted\_insertion(items, start, end)

print(f"Part {start // min\_run}: {' '.join(map(str, items[start:end + 1]))}")

size = min\_run

while size < n:

for start in range(0, n, 2 \* size):

mid = min(n - 1, start + size - 1)

end = min(n - 1, start + 2 \* size - 1)

if mid < end:

combine\_parts(items, start, mid, end)

size \*= 2

if \_\_name\_\_ == '\_\_main\_\_':

items\_amount = int(input())

items = list(map(int, input().split()))

run\_tim\_sort(items)

print(f"Answer: {' '.join(map(str, items))}")

**Tests.py**

import random

from main import run\_tim\_sort

def test\_from\_moevm():

array = [-4, 7, 5, 3, 5, -4, 2, -1, -9, -8, -3, 0, 9, -7, -4, -10, -4, 2, 6, 1, -2, -3, -1, -8, 0, -8, -7, -3, 5, -1, -8, -8, 8, -1, -3, 3, 6, 1, -8, -1, 3, -9, 9, -6]

run\_tim\_sort(array)

assert array == [0, 0, 1, 1, -1, -1, -1, -1, -1, 2, 2, -2, 3, 3, 3, -3, -3, -3, -3, -4, -4, -4, -4, 5, 5, 5, 6, 6, -6, 7, -7, -7, 8, -8, -8, -8, -8, -8, -8, 9, 9, -9, -9, -10]

def test\_short\_array():

array = [0,-1,1]

run\_tim\_sort(array)

assert array==[0,1,-1]

def test\_len(x = 1000):

array = []

for i in range(x):

array.append(random.randint(-100,100))

run\_tim\_sort(array)

assert len(array) == x