**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: TimSort**

|  |  |  |
| --- | --- | --- |
| Студент гр. 3342 |  | Песчатский С. Д. |
| Преподаватель |  | Иванов Д.В. |

Санкт-Петербург

2024

## **Цель работы**

Разработка сортировки TimSort, которая сортирует по убыванию модуля и выводит промежуточные результаты. Исследовать время работы алгоритма для лучшего, среднего и худшего случаев с заданным количеством элементов.

## **Задание**

Реализация

Имеется массив данных для сортировки int arr[] размера n

Необходимо отсортировать его алгоритмом сортировки TimSort по убыванию модуля.  
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.

Кратко алгоритм сортировки можно описать так:

Вычисление min\_run по размеру массива n (для упрощения отладки n уменьшается, пока не станет меньше 16, а не 64)

Разбиение массива на частично-упорядоченные (в т.ч. и по убыванию) блоки длины не меньше min\_run

Сортировка вставками каждого блока

Слияние каждого блока с сохранением инварианта и использованием галопа (галоп начинать после 3-х вставок подряд)

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.  
Для исследования используйте стандартный алгоритм вычисления min\_run и начинайте галоп после 7-ми вставок подряд.

Примечание:

Нельзя пользоваться готовыми библиотечными функциями для сортировки, нужно сделать реализацию сортировки вручную.  
Сортировка должна быть устойчивой.

Обратите внимание на пример.

## **Выполнение работы**

Разработан класс Stack, который для объединения отсортированных блоков, с помощью метода ending возвращает отсортированный массив.

Метод push(item) добавляет элемент в голову стека, и так же при условии того, что в стеке находится не менее двух элементов начинает проверку инвариантов, и их объединение при необходимости.

Метод \_\_len\_\_() возвращает количество элементов в стеке.

Метод top() возвращает последний элемент стека.

Метод pop() удаляет последний элемент стека.

Метод \_\_merge () проверяет условия инварианта, и при необходимости вызывает функцию, которая соединяет два массива.

Метод merge\_arr(arr1,arr2,gallop\_start) Метод, который объединяет два поданных в него массива с использованием галопа и возвращает результат.

Метод binary\_search(orig\_arr,target) Метод, который возвращет индекс искомого элемента в предоставленном миссиве.

Метод final\_merge() Метод, который вызывается, если в стеке после добавления всех элементов, осталось более двух элементов. Объединяет оставшиеся элементы.

Функция insertion\_sort(arr) реализует сортировку вставками.

Функция calculate\_min\_run(n) подсчитывает оптимальный размер minrun.

Функция is\_sorted\_abs(arr) исследует последовательность элементов в массиве и возвращает две переменных типа bool, первая – возрастает массив или нет, вторая – убывает массив или нет.

Функция separate\_arr() возвращает массив блоков, разделенных в соответствии с min\_run.

Функция tim\_sort(orig\_arr) возвращает отсортированный массив, с помощью TimSort.

Разработанный программный код см. в приложении А.

## **Тестирование**

Тесты для проверки корректности работы реализованного алгоритма TimSort находятся в файле tests.py.

## **Выводы**

В ходе исследования был разработан и подробно изучен алгоритм сортировки TimSort. Анализ времени выполнения показал, что как в среднем, так и в худшем случае алгоритм работает за время порядка n\*log(n) ,что согласуется с теоретическими ожиданиями. Однако в лучшем случае алгоритм демонстрирует значительное ускорение, работая намного быстрее.

# **Приложение А Исходный код программы**

Название файла: main.py

class Stack:  
 def \_\_init\_\_(self):  
 self.\_\_arr = []  
 self.gallops\_cnt = 0  
 self.merge\_cnt = 0  
  
 def \_\_len\_\_(self):  
 return len(self.\_\_arr)  
  
 def top(self):  
 if len(self.\_\_arr) == 0:  
 return None  
 else:  
 return self.\_\_arr[-1]  
  
 def push(self, item):  
 self.\_\_arr.append(item)  
 if len(self.\_\_arr) >= 2:  
 self.\_\_merge()  
  
 def pop(self):  
 if len(self.\_\_arr) == 0:  
 return  
 else:  
 self.\_\_arr.pop(-1)  
  
 def \_\_merge(self):  
 ok = True  
 while ok:  
 if len(self.\_\_arr) < 2:  
 break  
 y = self.\_\_arr[-2]  
 x = self.\_\_arr[-1]  
 if len(self.\_\_arr) > 2:  
 z = self.\_\_arr[-3]  
 if not (len(z) > len(x) + len(y) and len(y) > len(x)):  
 if len(z) < len(x):  
 self.\_\_arr[-1] = self.merge\_arr(self.\_\_arr[-1], y, 3)  
 print(f"Gallops {self.merge\_cnt}:", self.gallops\_cnt)  
 self.gallops\_cnt = 0  
 print(f"Merge {self.merge\_cnt}:", \*self.\_\_arr[-1])  
 self.merge\_cnt += 1  
 self.\_\_arr.pop(-2)  
 else:  
 self.\_\_arr[-1] = self.merge\_arr(self.\_\_arr[-1], y, 3)  
 print(f"Gallops {self.merge\_cnt}:", self.gallops\_cnt)  
 self.gallops\_cnt = 0  
 print(f"Merge {self.merge\_cnt}:", \*self.\_\_arr[-1])  
 self.merge\_cnt += 1  
 self.\_\_arr.pop(-2)  
 else:  
 ok = False  
 else:  
 if not (len(y) > len(x)):  
 self.\_\_arr[-1] = self.merge\_arr(self.\_\_arr[-1], y, 3)  
 print(f"Gallops {self.merge\_cnt}:", self.gallops\_cnt)  
 self.gallops\_cnt = 0  
 print(f"Merge {self.merge\_cnt}:", \*self.\_\_arr[-1])  
 self.merge\_cnt += 1  
 self.\_\_arr.pop(-2)  
 else:  
 ok = False  
  
 def merge\_arr(self, arr1, arr2, gallop\_start):  
 rez = []  
 first\_cnt = 0  
 second\_cnt = 0  
 first\_ind = 0  
 second\_ind = 0  
 while len(rez) < len(arr1) + len(arr2):  
 if first\_cnt == gallop\_start:  
 found\_ind = self.binary\_search(arr1[first\_ind:], arr2[second\_ind]) + first\_ind  
 rez.extend(arr1[first\_ind:found\_ind])  
 first\_ind = found\_ind  
 first\_cnt = 0  
 self.gallops\_cnt += 1  
 if second\_cnt == gallop\_start:  
 found\_ind = self.binary\_search(arr2[second\_ind:], arr1[first\_ind]) + second\_ind  
 rez.extend(arr2[second\_ind:found\_ind])  
 second\_ind = found\_ind  
 second\_cnt = 0  
 self.gallops\_cnt += 1  
 if first\_ind == len(arr1):  
 rez.extend(arr2[second\_ind:])  
 break  
 if second\_ind == len(arr2):  
 rez.extend(arr1[first\_ind:])  
 break  
 if abs(arr1[first\_ind]) > abs(arr2[second\_ind]):  
 rez.append(arr1[first\_ind])  
 first\_ind += 1  
 first\_cnt += 1  
 second\_cnt = 0  
 else:  
 rez.append(arr2[second\_ind])  
 second\_ind += 1  
 second\_cnt += 1  
 first\_cnt = 0  
 return rez  
  
 def binary\_search(self, orig\_arr, target):  
 left, right = 0, len(orig\_arr) - 1  
 result\_index = len(orig\_arr)  
 while left <= right:  
 mid = (left + right) // 2  
 mid\_val = orig\_arr[mid]  
 if abs(mid\_val) < abs(target):  
 result\_index = mid  
 right = mid - 1  
 else:  
 left = mid + 1  
 return result\_index  
  
 def final\_merge(self):  
 while len(self.\_\_arr) >= 2:  
 y = self.\_\_arr[-2]  
 x = self.\_\_arr[-1]  
 if len(self.\_\_arr) > 2:  
 z = self.\_\_arr[-3]  
 if len(z) < len(x):  
 self.\_\_arr[-3] = self.merge\_arr(self.\_\_arr[-3], y, 3)  
 print(f"Gallops {self.merge\_cnt}:", self.gallops\_cnt)  
 self.gallops\_cnt = 0  
 print(f"Merge {self.merge\_cnt}:", \*self.\_\_arr[-3])  
 self.merge\_cnt += 1  
 self.\_\_arr.pop(-2)  
 else:  
 self.\_\_arr[-1] = self.merge\_arr(self.\_\_arr[-1], y, 3)  
 print(f"Gallops {self.merge\_cnt}:", self.gallops\_cnt)  
 self.gallops\_cnt = 0  
 print(f"Merge {self.merge\_cnt}:", \*self.\_\_arr[-1])  
 self.merge\_cnt += 1  
 self.\_\_arr.pop(-2)  
 else:  
 self.\_\_arr[-1] = self.merge\_arr(self.\_\_arr[-1], y, 3)  
 print(f"Gallops {self.merge\_cnt}:", self.gallops\_cnt)  
 self.gallops\_cnt = 0  
 print(f"Merge {self.merge\_cnt}:", \*self.\_\_arr[-1])  
 self.merge\_cnt += 1  
 self.\_\_arr.pop(-2)  
  
def calculate\_min\_run(n):  
 r = 0  
 while n >= 16:  
 r |= n & 1  
 n >>= 1  
 return n + r  
  
def insertion\_sort(orig\_arr):  
 for i in range(1, len(orig\_arr)):  
 key = orig\_arr[i]  
 j = i - 1  
 while j >= 0 and abs(orig\_arr[j]) < abs(key):  
 orig\_arr[j + 1] = orig\_arr[j]  
 j -= 1  
 orig\_arr[j + 1] = key  
 return orig\_arr  
  
def separate\_arr(array, min\_run):  
 runs = [[]]  
 for i in range(len(array)):  
 if len(runs[-1]) < min\_run:  
 runs[-1].append(array[i])  
 if i == len(array) - 1:  
 insertion\_sort(runs[-1])  
 else:  
 ascending\_order, descending\_order = is\_sorted\_abs(runs[-1])  
 if ascending\_order and not descending\_order:  
 if abs(array[i]) > abs(runs[-1][-1]):  
 runs[-1].append(array[i])  
 else:  
 insertion\_sort(runs[-1])  
 runs.append([array[i]])  
 continue  
 if not ascending\_order and descending\_order:  
 if abs(array[i]) < abs(runs[-1][-1]):  
 runs[-1].append(array[i])  
 else:  
 insertion\_sort(runs[-1])  
 runs.append([array[i]])  
 continue  
 if not ascending\_order and not descending\_order:  
 insertion\_sort(runs[-1])  
 runs.append([array[i]])  
 continue  
 if ascending\_order and descending\_order:  
 if abs(array[i]) == abs(runs[-1][-1]):  
 runs[-1].append(array[i])  
 continue  
 insertion\_sort(runs[-1])  
 runs.append([array[i]])  
 return runs  
  
def is\_sorted\_abs(arr):  
 ascending = all(abs(arr[i]) <= abs(arr[i + 1]) for i in range(len(arr) - 1))  
 descending = all(abs(arr[i]) >= abs(arr[i + 1]) for i in range(len(arr) - 1))  
 return ascending, descending  
  
def tim\_sort(orig\_arr):  
 min\_run = calculate\_min\_run(len(orig\_arr))  
 runs = separate\_arr(orig\_arr, min\_run)  
 for i in range(len(runs)):  
 print(f"Part {i}:", \*runs[i])  
 stack = Stack()  
 for i in runs:  
 stack.push(i)  
 stack.final\_merge()  
 return stack.top()  
  
n = int(input())  
inp = [int(x) for x in input().split()]  
print("Answer:", \*tim\_sort(inp))

Название файла: tests.py

import random

from main import \*

def test\_insertion\_sort():

arr = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]

sorted\_arr = insertion\_sort(arr)

assert sorted\_arr == [9, 6, 5, 5, 5, 4, 3, 3, 2, 1, 1], f"Error: {sorted\_arr}"

print("test\_insertion\_sort passed")

def test\_separate\_arr():

arr = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]

min\_run = 3

runs = separate\_arr(arr, min\_run)

assert runs == [[4, 3, 1], [9, 5, 1], [6, 5, 2], [5, 3]], f"Error: {runs}"

print("test\_separate\_arr passed")

def test\_is\_sorted\_abs():

arr1 = [1, 2, 3, 4]

arr2 = [4, 3, 2, 1]

arr3 = [1, 3, 2, 4]

assert is\_sorted\_abs(arr1) == (True, False), f"Error: {is\_sorted\_abs(arr1)}"

assert is\_sorted\_abs(arr2) == (False, True), f"Error: {is\_sorted\_abs(arr2)}"

assert is\_sorted\_abs(arr3) == (False, False), f"Error: {is\_sorted\_abs(arr3)}"

print("test\_is\_sorted\_abs passed")

def test\_merge\_arr():

stack = Stack()

arr1 = [1, 3, 5]

arr2 = [2, 4, 6]

merged\_arr = stack.merge\_arr(arr1, arr2, 3)

assert merged\_arr == [2, 4, 6, 1, 3, 5], f"Error: {merged\_arr}"

print("test\_merge\_arr passed")

def test\_1():

assert tim\_sort(

[10, -10, 10, -10, -10, -10, 10, 10, 10, 11, 10, -10, 10, -10, -10, 10, -10, 25, -15, 13, -12, -11, 11, -11,

-25, 15, -13, 12, 11, -11, 11, 10]) == [25, -25, -15, 15, 13, -13, -12, 12, 11, -11, 11, -11, 11, -11, 11, 10, -10, 10, -10, -10, 10, -10, 10, -10, 10, -10, -10, -10, 10, 10, 10, 10]

def test\_timsort\_empty\_list():

assert tim\_sort([]) == []

def test\_timsort\_single\_element():

assert tim\_sort([42]) == [42]

def test\_timsort\_already\_sorted():

assert tim\_sort([5, -5, 4, 3, -2]) == [5, -5, 4, 3, -2]

def test\_timsort\_reverse\_sorted():

assert tim\_sort([1, 2, 3, 4, 5]) == [5, 4, 3, 2, 1]

def run\_tests():

test\_insertion\_sort()

test\_separate\_arr()

test\_is\_sorted\_abs()

test\_merge\_arr()

test\_1()

test\_timsort\_empty\_list()

test\_timsort\_single\_element()

test\_timsort\_already\_sorted()

test\_timsort\_reverse\_sorted()

print("All tests passed")

if \_\_name\_\_ == "\_\_main\_\_":

run\_tests()