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 Мохамед .М.Х.

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.

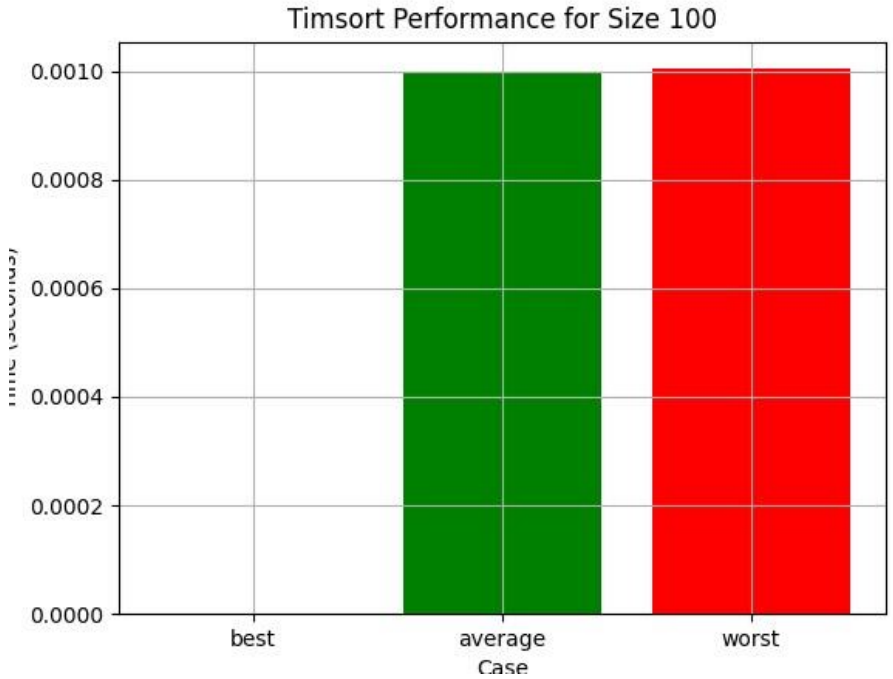
Функция separate_arr() возвращает массив блоков, разделенных в соответствии с min_run.

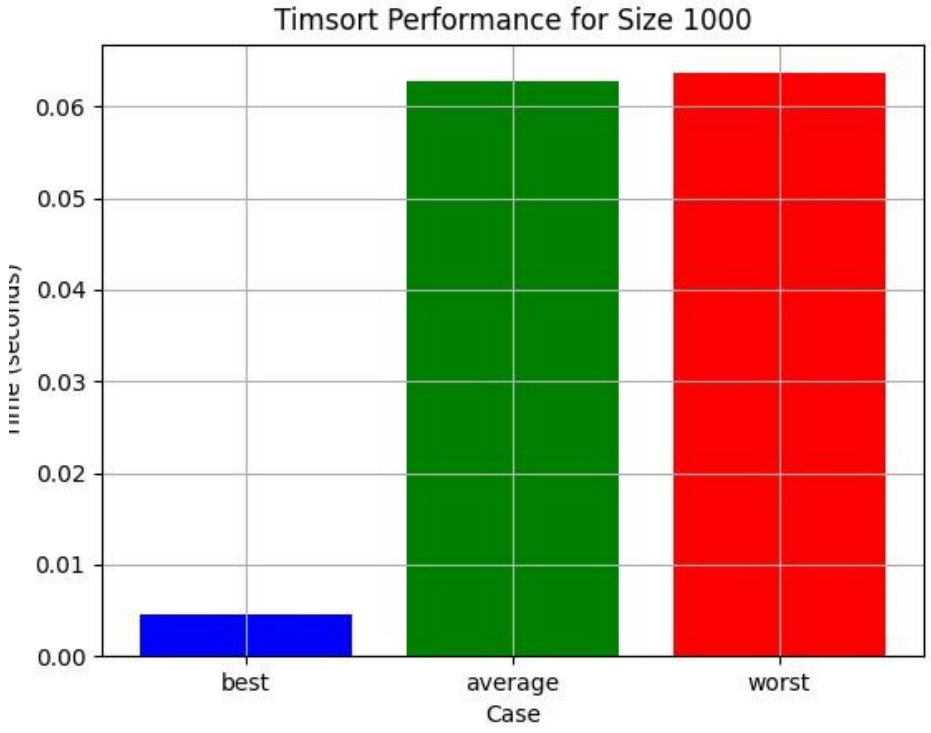
Функция tim_sort(orig_arr) возвращает отсортированный массив, с помощью TimSort.

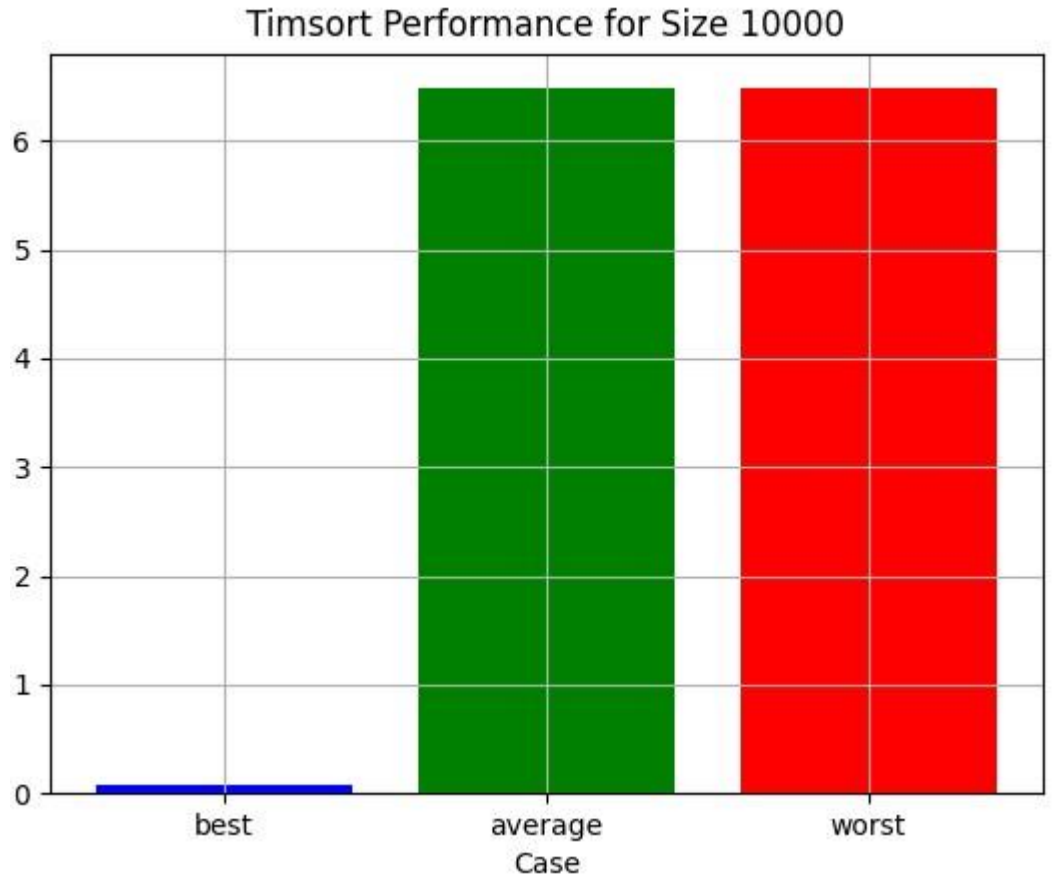
Разработанный программный код см. в приложении А.

Тестирование

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







Выводы

В ходе данного исследования был проведен и проанализирован алгоритм сортировки TimSort. Результаты, связанные со временем, показывают, что как средний, так и наихудший случаи занимают примерно одно и то же время, что согласуется с теорией о том, что временная сложность равна n*log(n). Однако в лучшем случае все гораздо быстрее.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.py

```
class CustomStack:
    def __init__(self):
        self._elements = []
        self.gallops_count = 0
        self.merge_count = 0

    def __len__(self):
        return
len(self._elements)

    def top(self):
        return self._elements[-
1] if self._elements else None

    def push(self, item):

self._elements.append(item)
        if len(self._elements)
>= 2:
            self._merge()

    def pop(self):
        if self._elements:
            self._elements.pop()

    def _merge(self):
        valid = True

        while valid:
            if
len(self._elements) < 2:
                break

            y = self._elements[-
2]
            x = self._elements[-
1]

            if
len(self._elements) > 2:
                z =
self._elements[-3]
```

```
                if not (len(z) >
len(x) + len(y) and len(y) >
len(x)):
                    if len(z) <
len(x):

self._elements[-1] =
self._merge_arrays(self._elemen
ts[-1], y, 3)

self._log_merge()

self._elements.pop(-2)
                    else:

self._elements[-1] =
self._merge_arrays(self._elemen
ts[-1], y, 3)

self._log_merge()

self._elements.pop(-2)
                else:
                    valid =
False
            else:
                if not (len(y) >
len(x)):

self._elements[-1] =
self._merge_arrays(self._elemen
ts[-1], y, 3)

self._log_merge()

self._elements.pop(-2)
                else:
                    valid =
False

    def _merge_arrays(self,
arr1, arr2, gallop_start):
        result = []
        first_count,
second_count = 0, 0
        first_index,
second_index = 0, 0
```

```
        while len(result) <
len(arr1) + len(arr2):
            if first_count ==
gallop_start:
                found_index = (

self._binary_search(arr1[first_
index:], arr2[second_index]) +
first_index
                )

result.extend(arr1[first_index:
found_index])
                first_index =
found_index
                first_count = 0

self.gallops_count += 1

            if second_count ==
gallop_start:
                found_index = (

self._binary_search(arr2[second
_index:], arr1[first_index]) +
second_index
                )

result.extend(arr2[second_index
:found_index])
                second_index =
found_index
                second_count = 0

self.gallops_count += 1

            if first_index ==
len(arr1):

result.extend(arr2[second_index
:])
                break
            if second_index ==
len(arr2):

result.extend(arr1[first_index:
])
                break
```

```
            if
abs(arr1[first_index]) >
abs(arr2[second_index]):

result.append(arr1[first_index]
)
                first_index += 1
                first_count += 1
                second_count = 0
            else:

result.append(arr2[second_index
])
                second_index +=
1
                second_count +=
1
                first_count = 0

        return result

    def _binary_search(self,
original_array, target):
        left, right = 0,
len(original_array) - 1
        result_index =
len(original_array)

        while left <= right:
            mid = (left + right)
// 2
            mid_value =
original_array[mid]

            if abs(mid_value) <
abs(target):
                result_index =
mid
                right = mid - 1
            else:
                left = mid + 1

        return result_index

    def _log_merge(self):
        print(f"Gallops
{self.merge_count}:",
self.gallops_count)
        self.gallops_count = 0
```

```
        print(f"Merge
{self.merge_count}:",
*self._elements[-1])
        self.merge_count += 1

    def final_merge(self):
        while
len(self._elements) >= 2:
            y = self._elements[-
2]
            x = self._elements[-
1]

            if
len(self._elements) > 2:
                z =
self._elements[-3]
                if len(z) <
len(x):

self._elements[-3] =
self._merge_arrays(self._elemen
ts[-3], y, 3)

self._log_merge()

self._elements.pop(-2)
                else:

self._elements[-1] =
self._merge_arrays(self._elemen
ts[-1], y, 3)

self._log_merge()

self._elements.pop(-2)
            else:
                self._elements[-
1] =
self._merge_arrays(self._elemen
ts[-1], y, 3)

self._log_merge()

self._elements.pop(-2)

def calculate_min_run(n):
    r = 0
    while n >= 16:
```

```
        r |= n & 1
        n >>= 1
    return n + r

def
insertion_sort(original_array):
    for i in range(1,
len(original_array)):
        key = original_array[i]
        j = i - 1

        while j >= 0 and
abs(original_array[j]) <
abs(key):
            original_array[j +
1] = original_array[j]
            j -= 1

        original_array[j + 1] =
key

    return original_array

def separate_array(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,
descending =
_is_sorted_abs(runs[-1])

            if ascending and not
descending:
                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 and
descending:
                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 and
not descending:

insertion_sort(runs[-1])

runs.append([array[i]])
                continue

            if ascending and
descending:
                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(original_array):
    min_run =
calculate_min_run(len(original_
array))
    runs =
separate_array(original_array,
min_run)

    for i, run in
enumerate(runs):
        print(f"Part {i}:",
*run)

    stack = CustomStack()

    for run in runs:
        stack.push(run)

    stack.final_merge()

    return stack.top()

n = int(input())
input_data = [int(x) for x in
input().split()]

print("Answer:",
*tim_sort(input_data))
```

Название файла: tests.py

```
from main import tim_sort


def test_empty_list():
    input_list = []
    sorted_result =
tim_sort(input_list)
    assert sorted_result == [],
f"Expected [], got
{sorted_result}"


def test_single_item_list():
```

```
    input_list = [96]
    sorted_result               =
tim_sort(input_list)
    assert sorted_result == [96],
f"Expected       [96],       got
{sorted_result}"


def test_sorted_list():
    input_list = [5, 4, 3, 2, 1]
    sorted_result               =
tim_sort(input_list)
    assert sorted_result == [5,
4, 3, 2, 1], f"Expected [5, 4, 3,
2, 1], got {sorted_result}"


def test_reverse_sorted_list():
    input_list = [1, 2, 3, 4, 5]
    sorted_result               =
tim_sort(input_list)
    assert sorted_result == [5,
4, 3, 2, 1], f"Expected [5, 4, 3,
2, 1], got {sorted_result}"


def test_random_list():
    input_list = [3, 1, 4, 1, 5,
9, 2, 6, 5, 3, 5]
    sorted_result               =
tim_sort(input_list)
    assert sorted_result == [9,
6, 5, 5, 5, 4, 3, 3, 2, 1, 1],
f"Expected [9, 6, 5, 5, 5, 4, 3,
3, 2, 1, 1], got {sorted_result}"


def test_duplicates_in_list():
    input_list = [3, 1, 4, 1, 5,
9, 2, 6, 5, 3, 5]
    sorted_result               =
tim_sort(input_list)
    assert sorted_result == [9,
6, 5, 5, 5, 4, 3, 3, 2, 1, 1],
f"Expected [9, 6, 5, 5, 5, 4, 3,
3, 2, 1, 1], got {sorted_result}"
```

```
def
test_negative_numbers_in_list()
:
    input_list = [-3, 1, -4, 1,
5, -9, 2, 6, -5, 3, 5]
    sorted_result              =
tim_sort(input_list)
    assert sorted_result == [-9,
6, 5, -5, 5, -4, -3, 3, 2, 1, 1],
f"Expected [-9, 6, 5, -5, 5, -4,
-3,   3,   2,   1,   1],   got
{sorted_result}"


def
test_identical_elements_in_list
():
    input_list = [7, 7, 7, 7, 7]
    sorted_result              =
tim_sort(input_list)
    assert sorted_result == [7,
7, 7, 7, 7], f"Expected [7, 7, 7,
7, 7], got {sorted_result}"


def
test_large_numbers_in_list():
    input_list  =  [1,  10,  100,
1000, 10000, 1000000]
    sorted_result              =
tim_sort(input_list)
    assert   sorted_result   ==
[1000000, 10000, 1000, 100, 10,
1], f"Expected [1000000, 10000,
1000,   100,   10,   1],   got
{sorted_result}"


if __name__ == '__main__':
    test_empty_list()
    test_single_item_list()
    test_sorted_list()
    test_reverse_sorted_list()
    test_random_list()
    test_duplicates_in_list()

test_negative_numbers_in_list()
```

```
test_identical_elements_in_list
()
    test_large_numbers_in_list()

    print("All    tests    passed
successfully!")
```