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.

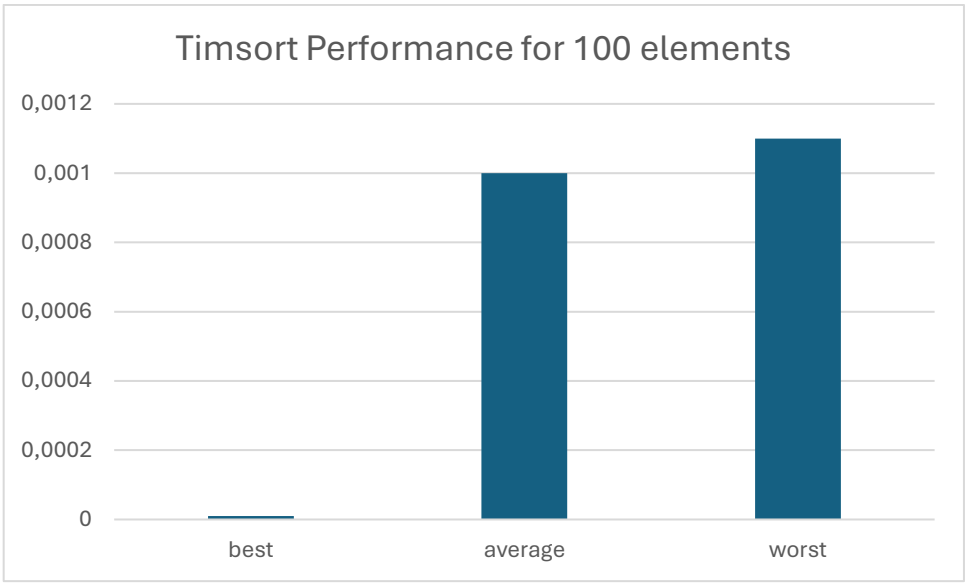
Функция separate_arr() возвращает массив блоков, разделенных в соответствии с min_run.

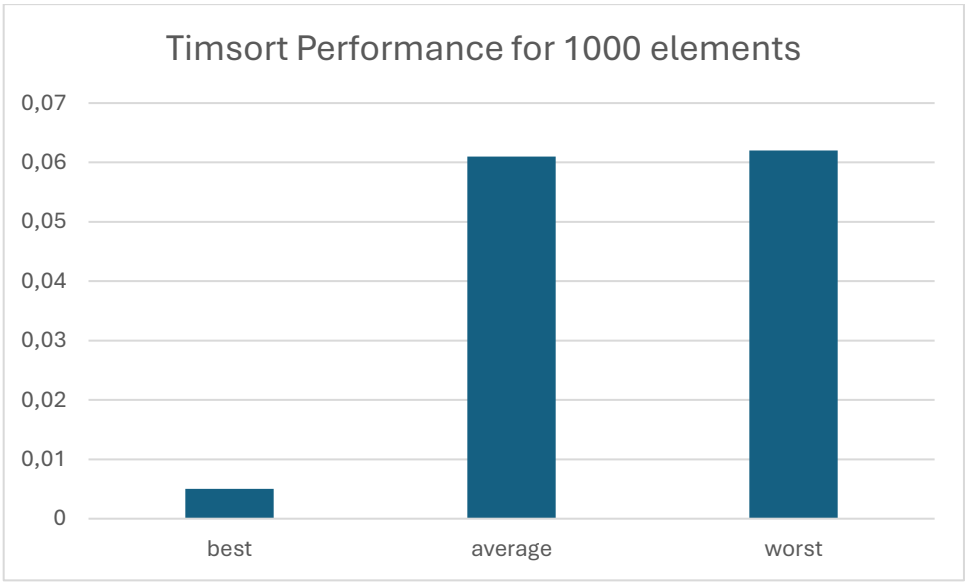
Функция tim_sort(orig_arr) возвращает отсортированный массив, с помощью TimSort.

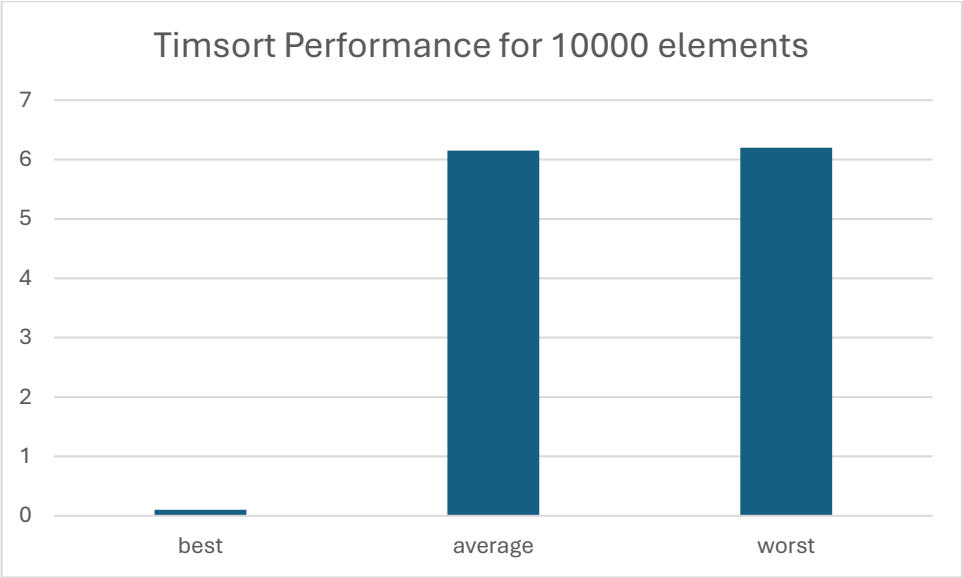
Разработанный программный код см. в приложении А.

Тестирование

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






Timsort Performance for 10000 elements
7
6
5
4
3
2
1
0
best
average
worst

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()
```