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 сортировки TimSort

Студентка гр. 3342 _____ Смирнова Е.С.

Преподаватель _____ Иванов Д.В.

Санкт-Петербург

2024

Цель работы

Исследование алгоритма сортировки TimSort и его реализация на языке Python или C++.

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.

Формат ввода

Первая строка содержит натуральное число n - размерность массива, следующая строка содержит элементы массива через пробел.

Формат вывода

Выводятся разделённые блоки для сортировки в формате "Part i: *отсортированный разделённый массив*"

Затем для каждого слияния выводится количество вхождений в режим галопа и получившийся массив в формате

"Gallops i: *число вхождений в галоп*

Merge i: *итоговый массив после слияния*"

Последняя строчка содержит финальный результат сортировки массива с надписью "Answer: "

Пример #1 (min_run = 10)

Ввод

20

1 -2 3 -4 5 6 -7 -8 9 -10 11 -10 -9 8 7 -7 -6 6 5 4

Вывод

Part 0: 11 -10 9 -8 -7 6 5 -4 3 -2 1

Part 1: -10 -9 8 7 -7 -6 6 5 4

Gallops 0: 0

Merge 0: 11 -10 -10 9 -9 -8 8 -7 7 -7 6 -6 6 5 5 -4 4 3 -2 1

Answer: 11 -10 -10 9 -9 -8 8 -7 7 -7 6 -6 6 5 5 -4 4 3 -2 1

Пример #2 (min_run = 8)

Ввод

16

-1 2 3 4 5 -6 7 8 -8 -8 7 -7 7 6 -5 4

Вывод

Part 0: 8 7 -6 5 4 3 2 -1

Part 1: -8 -8 7 -7 7 6 -5 4

Gallops 0: 1

Merge 0: 8 -8 -8 7 7 -7 7 6 -6 5 -5 4 4 3 2 -1

Answer: 8 -8 -8 7 7 -7 7 6 -6 5 -5 4 4 3 2 -1

Для C++ #include <iostream>, <vector>, <stack> уже добавлены

Выполнение работы

Реализован класс Stack для хранения отсортированных подмассивов и объединения их с помощью модифицированного алгоритма – “галоп”. В рамках данного класса реализованы следующие методы:

1. pop(index) – реализует удаление последнего элемента.
2. top() – возвращает последний элемент в стеке.
3. push(item) – добавляет подмассив в стек и проверяет условия для слияния двух подмассивов, если они выполнены, реализует слияние.
4. gallop_search(array, arr, end) – модернизированный алгоритм бинарного поиска.
5. merge() – алгоритм для определения подмассивов для слияния.
6. merge_arr(arr1, arr2) – реализует слияние двух подмассивов.
7. end_merge() – завершение слияния.

В ходе выполнения лабораторной работы были написаны следующие функции: calculate_minrun, insertion_sort, split_arr, tim_sort.

В функции calculate_minrun вычисляется размер массива minrun.

В функции insertion_sort осуществляется сортировка вставками массива, поданного на вход.

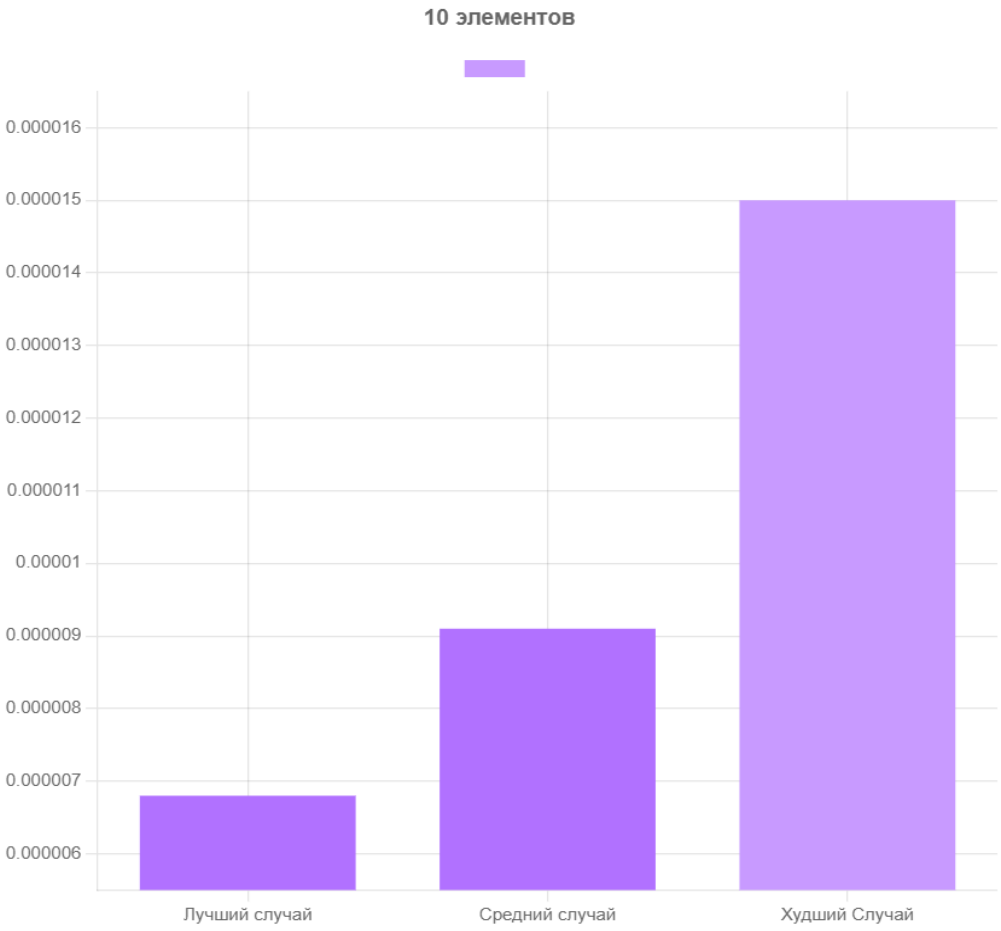
В функции split_arr происходит разбиение исходного массива на подмассивы.

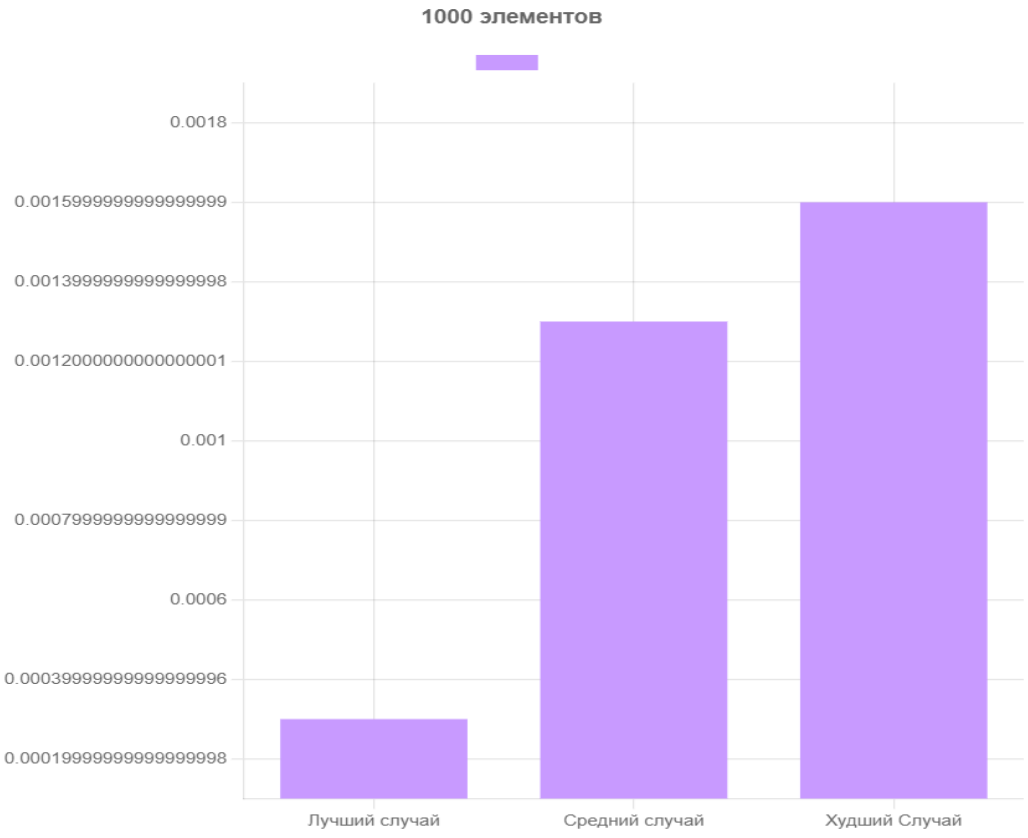
В функции tim_sort осуществляется реализация алгоритма TimSort.

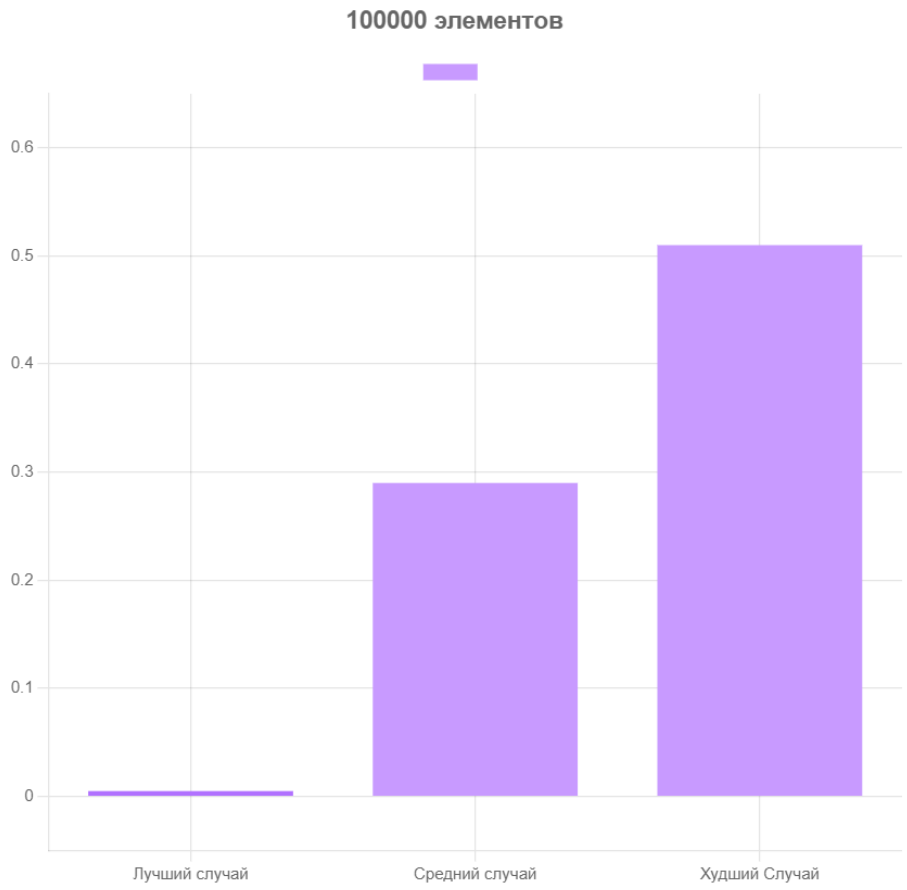
Разработанный программный код см. в приложении А.

Результаты исследования работы см. в приложении Б.

Тестирование






100000 элементов
0.6
0.5
0.4
0.3
0.2
0.1
0
Лучший случай
Средний случай
Худший Случай

Выводы

Был исследован алгоритм сортировки TimSort и была реализована программа, осуществляющая сортировку введённого пользователем массива по убыванию модулей. В ходе исследования было выяснено, что скорость сортировки совпадает с теоретической с точностью до константы.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.py

```
'''LB2 TimSort'''
class Stack:
    '''Stack'''
    def __init__(self):
        '''Initialization'''
        self.arr = []
        self.gallops = 0
        self.cnt = 0

    def __len__(self):
        '''Length'''
        return len(self.arr)

    def top(self):
        '''Last value'''
        return self.arr[-1]

    def push(self, item):
        '''Adding an element to the end'''
        self.arr.append(item)
        if len(self.arr) >= 2:
            self.merge()

    def pop(self):
        '''Deleting the last element'''
        self.arr.pop(-1)

    def gallop_search(self, arr, end):
        '''Searching for an index using a gallop'''
        left = 0
        right = len(arr) - 1
        index = len(arr)
        while left <= right:
            mid = (left + right) // 2
            if abs(arr[mid]) < abs(end):
                index = mid
                right = mid - 1
            else:
                left = mid + 1
        return index

    def merge(self):
        '''Merging'''
        while len(self.arr) >= 2:
            if len(self.arr) > 2:
                z = self.arr[-3]
                y = self.arr[-2]
                x = self.arr[-1]
                if not (len(z) > len(x) + len(y) and len(y) > len(x)):
                    if len(z) < len(x):
```

```
                        self.arr[-3] = self.merge_arr(z, y)
                        print(f"Gallops {self.cnt}:", self.gallops)
                        print(f"Merge {self.cnt}:", *self.arr[-1])
                        self.gallops = 0
                        self.cnt += 1
                        self.arr.pop(-2)
                    else:
                        self.arr[-1] = self.merge_arr(x, y)
                        print(f"Gallops {self.cnt}:", self.gallops)
                        print(f"Merge {self.cnt}:", *self.arr[-1])
                        self.gallops = 0
                        self.cnt += 1
                        self.arr.pop(-2)
                else:
                    break
            else:
                y = self.arr[-2]
                x = self.arr[-1]
                if not len(y) > len(x):
                    self.arr[-1] = self.merge_arr(x, y)
                    print(f"Gallops {self.cnt}:", self.gallops)
                    print(f"Merge {self.cnt}:", *self.arr[-1])
                    self.gallops = 0
                    self.cnt += 1
                    self.arr.pop(-2)
                else:
                    break

    def merge_arr(self, arr1, arr2):
        '''Merging two arrays'''
        res = []
        i = j = cnt_i = cnt_j = 0
        while i < len(arr1) and j < len(arr2):
            if abs(arr1[i]) > abs(arr2[j]):
                res.append(arr1[i])
                i += 1
                cnt_i += 1
                cnt_j = 0
            else:
                res.append(arr2[j])
                j += 1
                cnt_j += 1
                cnt_i = 0

            if cnt_i == 3:
                ind = self.gallop_search(arr1[i:], arr2[j]) + i
                res += arr1[i:ind]
                i = ind
                self.gallops += 1
                cnt_i = 0

            elif cnt_j == 3:
                ind = self.gallop_search(arr2[j:], arr1[i]) + j
                res += arr2[j:ind]
                j = ind
                self.gallops += 1
                cnt_j = 0
        while i < len(arr1):
```

```
                res.append(arr1[i])
                i += 1
            while j < len(arr2):
                res.append(arr2[j])
                j += 1
            return res

        def end_merge(self):
            '''Completion of the merger'''
            while len(self.arr) >= 2:
                y = self.arr[-2]
                x = self.arr[-1]
                if len(self.arr) > 2:
                    self.arr.merge()
                else:
                    y = self.arr[-2]
                    x = self.arr[-1]
                    self.arr[-1] = self.merge_arr(x, y)
                    print(f"Gallops {self.cnt}:", self.gallops)
                    print(f"Merge {self.cnt}:", *self.arr[-1])
                    self.gallops = 0
                    self.cnt += 1
                    self.arr.pop(-2)

    def calculate_minrun(num):
        '''Minrun calculation function'''
        flag = 0
        while num >= 16:
            flag |= num & 1
            num >>= 1
        return num + flag

    def insertion_sort(arr):
        '''Insertion sorting function'''
        for i in range(1, len(arr)):
            key = arr[i]
            j = i - 1
            while j >= 0 and abs(arr[j]) < abs(key):
                arr[j + 1] = arr[j]
                j -= 1
            arr[j + 1] = key
        return arr

    def split_arr(array_, minrun):
        '''Function of dividing an array into parts'''
        cur = []
        runs = []
        is_increasing, is_decreasing = False, False
        for element in array_:
            if cur:
                if (abs(element) > abs(cur[-1]) and not is_increasing)
or (abs(element) <= abs(cur[-1]) and not is_decreasing):
                    if abs(element) > abs(cur[-1]):
                        is_decreasing = True
                    if abs(element) <= abs(cur[-1]):
                        is_increasing = True
                    cur.append(element)
                elif len(cur) < minrun:
```

```
                cur.append(element)
                is_increasing = True
                is_decreasing = True
            else:
                insertion_sort(cur)
                runs.append(cur)
                cur = [element]
                is_increasing = False
                is_decreasing = False
        else:
            cur.append(element)
    if cur:
        insertion_sort(cur)
        runs.append(cur)
        for i, item in enumerate(runs):
            print(f"Part {i}: {' '.join(map(str, item))}")
    return runs

def tim_sort(arr, m):
    '''Timsort function'''
    minrun = calculate_minrun(m)
    runs = split_arr(arr, minrun)
    stack = Stack()
    for i in runs:
        stack.push(i)
    stack.end_merge()
    return stack.top()

n = int(input())
array = [int(x) for x in input().split()]
print("Answer:", *tim_sort(array, n))
```

ПРИЛОЖЕНИЕ Б

ТЕСТИРОВАНИЕ

Название файла: tests.py

```
'''tests LB2 timsort'''
import main

def test_base_case():
    '''Standard sorting'''
    assert main.timsort([1, -2, 3, -4, 5, 6, -7, -8, 9, -10, 11, -
10, -9, 9, 7, -7, -6, 6, 5, 4], 20) ==
    [11, -10, -10, 9, -9, -8, 8, -7, 7, -7, 6, -6, 6, 5, 5, -4, 4,
3, -2, 1]

def test_timsort_single_element():
    '''Sorting one element'''
    assert main.timsort([43], 1) == [43]

def test_timsort_empty_list():
    '''Sorting empty list'''
    assert main.timsort([], 0) == []

def test_timsort_already_sorted():
    '''Sorting sorted list'''
    assert main.timsort([-19, 17, 15, 2, 1], 5) == [-9, 8, 5, 2, 1]

def test_timsort_reverse_sorted():
    '''Sorting reverse sorted list'''
    assert main.timsort([1, 2, 3, 4, 5, 6, 7, 8, 9, 10], 10) == [10,
9, 8, 7, 6, 5, 4, 3, 2, 1]
```