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

Студент гр. 3384 _____ Рудаков А.Л.

Преподаватель _____ Шестопалов Р.П.

Санкт-Петербург

2024

Цель работы.

Изучить, что из себя представляет алгоритм сортировки TimSort, и реализовать данный алгоритм. Также необходимо провести исследование его работы.

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.

Вначале работы был написан класс *Stack*, элементы которого являются массивами. Он используется при слиянии подмассивов.

Далее была создана функция *def merge_block(value_stack, galops, merges, flag=0)*, которая используется для слияния подмассивов в стеке. Для его реализации используется оптимизация слияния блоков в определенном порядке, описанном в методических материалах. Внутри метода из стека достаются подмассивы, по их длинам происходит слияние, если оно не нужно, то они записываются обратно в стек, если нужно, то вызывается функция *merge()*, после чего полученное значение записывается в стек.

Далее реализована функция неточного бинпоиска *def bin_find(data, element)*, который ищет место, куда в data вставить новый элемент element.

Далее идет функция слияния двух упорядоченных массивов *def merge(first, second, flag=3).* Внутри оно идет по массивам и сравнивает элементы, подходящий добавляет в новый массив. Если взято *flag* элементов подряд из одного из массивов, то включается режим галопа (вызов функции *bin_find*). Если в одном из массивов заканчиваются элементы, то оставшиеся элементы другого массива добавляются в новый.

Далее реализована функция сортировки вставками *def InsertionSort(data, flag).* Если *flag == True*, то сортируется в порядке невозрастание, иначе – в порядке неубывания.

Далее идет функция *def push_and_merge(value_stack, data, galops, merges),* вызывающая метод *push()* и функцию *merge_block()* соответственно.

Далее идет функция *def final_merge(value_stack, galops, merges),* выполняющая слияние всех оставшихся элементов в стеке, и возвращает отсортированный массив.

Также реализована функция *def minrun_length(n)*, которая считает размер минрана.

Далее реализована функция *def TimSort(data)*, в которой происходит пробег по массиву, разделение его на упорядоченные части >= минарну, вызов функций *InsertionSort()* для сортировки этих небольших частей и *push_and_merge()* для помещения в стек. В конце вызывается функция *final_merge(),* выводятся дополнительные значения, требуемые в задании и возвращается остортированный массив.

Описание пайплайна.

Дан массив, вызывается функция *TimSort(),* которая определяет размер минара при помощи функции *minrun_length(),* после чего разделяет массив на упорядоченные подмассивы длины >= минрану, упорядочивая их при помощи функции *InsertionSort().* После этого закидывает их в стек при помощи функции *push_and_merge(),* которая в свою очередь вызывает функцию *merge_block()* для объединения подмассивов в стеке, объединение происходит при помощи функции *merge(),* которая в режиме галопа активирует функцию *bin_find().* После прохода по всему массиву вызывается функция *final_merge()* для слияния оставшихся частей в стеке и возвращается результат.

Анализ полученных значений.

Также было проведено исследование. Замерено время работы алгоритма на небольшом, среднем и большом наборах данных в лучших, средних и худших случаях.

Единицы измерения 10^{-6} с.

Таблица 1 – Результаты исследования

	Маленький	Средний	Большой
Лучший	1	4	477
Средний	1	5	803
Худший	2	117	19422

Рисунок 1 – Время сортировки

Разработанный программный код см. в приложении А.

Результаты тестирования см. в приложении Б.

Выводы.

Был изучен и реализован алгоритм сортировки TimSort. В реализации использовалась работа со стеком а также алгоритм сортировки вставками для небольших объемов данных.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.py

```
class Node:
    def __init__(self, data=[]):
        self.data = data
        self.prev_element = None


class Stack:
    def __init__(self):
        self.__tail = None
        self.__length = 0

    def __len__(self):
        return self.__length

    def push(self, data):
        current = Node(data)
        new_tail = self.__tail
        current.prev_element = new_tail
        self.__tail = current
        self.__length += 1

    def pop(self):
        if not self.isEmpty():
            data = self.__tail.data
            self.__tail = self.__tail.prev_element
            self.__length -= 1
            return data
        else:
            return -1

    def isEmpty(self):
        if self.__tail != None:
            return False
        else:
            return True


def merge_block(value_stack, galops, merges, flag=0):
    while len(value_stack) >= 2:
        if len(value_stack) >= 3 and flag == 0:
            X_data = value_stack.pop()
            Y_data = value_stack.pop()
            Z_data = value_stack.pop()
            X_len,  Y_len,  Z_len  =  len(X_data),  len(Y_data),
len(Z_data)
            if Y_len >= X_len and Z_len >= X_len + Y_len:
                value_stack.push(Z_data)
                value_stack.push(Y_data)
                value_stack.push(X_data)
                break
```

```
                else:
                    if X_len < Z_len:
                        galop_number,   merged_data   =   merge(Y_data,
X_data)
                        galops.append(galop_number)
                        merges.append(merged_data)
                        value_stack.push(Z_data)
                        value_stack.push(merged_data)
                    else:
                        galop_number,   merged_data   =   merge(Z_data,
Y_data)
                        galops.append(galop_number)
                        merges.append(merged_data)
                        value_stack.push(merged_data)
                        value_stack.push(X_data)
            else:
                X_data = value_stack.pop()
                Y_data = value_stack.pop()
                X_len, Y_len = len(X_data), len(Y_data)
                if Y_len <= X_len or flag == 1:
                    galop_number, merged_data = merge(Y_data, X_data)
                    galops.append(galop_number)
                    merges.append(merged_data)
                    value_stack.push(merged_data)
                else:
                    value_stack.push(Y_data)
                    value_stack.push(X_data)
                    break
        return galops, merges, value_stack


    def bin_find(data, element):
        left = 0
        right = len(data)
        if abs(element) > abs(data[0]):
            return [], 0
        while right - left != 1:
            center = left + (right - left) // 2
            if abs(data[center]) >= abs(element):
                left = center
            elif abs(data[center]) < abs(element):
                right = center
        return data[1:right], right


    def merge(first, second, flag=3):
        first_idx, second_idx = 0, 0
        first_count, second_count = 0, 0
        count_galops = 0
        new_data = []
        while first_idx < len(first) and second_idx < len(second):
            if abs(first[first_idx]) >= abs(second[second_idx]):
                new_data.append(first[first_idx])
                first_idx += 1
                first_count += 1
                second_count = 0
            elif abs(first[first_idx]) < abs(second[second_idx]):
                new_data.append(second[second_idx])
```

```
                second_idx += 1
                second_count += 1
                first_count = 0

            if first_count == flag:
                data, new_idx = bin_find(first[first_idx - 1:],
second[second_idx])
                first_idx += new_idx - 1
                first_count = 0
                new_data.extend(data)
                count_galops += 1
            elif second_count == flag:
                data, new_idx = bin_find(second[second_idx - 1:],
first[first_idx])
                second_idx += new_idx - 1
                second_count = 0
                new_data.extend(data)
                count_galops += 1

        if first_idx >= len(first):
            new_data.extend(second[second_idx:])
        elif second_idx >= len(second):
            new_data.extend(first[first_idx:])
        return count_galops, new_data


    def InsertionSort(data, flag):
        for i in range(len(data)):
            for j in range(i, 0, -1):
                if flag == True:
                    if abs(data[j]) > abs(data[j - 1]):
                        temp = data[j]
                        data[j] = data[j - 1]
                        data[j - 1] = temp
                else:
                    if abs(data[j]) < abs(data[j - 1]):
                        temp = data[j]
                        data[j] = data[j - 1]
                        data[j - 1] = temp
        return data


    def push_and_merge(value_stack, data, galops, merges):
        value_stack.push(data)
        galops, merges, value_stack = merge_block(value_stack, galops,
merges, 0)
        return galops, merges, value_stack


    def final_merge(value_stack, galops, merges):
        while len(value_stack) > 1:
            galops, merges, value_stack = merge_block(value_stack,
galops, merges, 1)
        return value_stack.pop()

    def minrun_length(n):
        flag = 0
        while n >= 16:
```

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

    def TimSort(data):
        minrun = minrun_length(len(data))
        parts = []
        galops = []
        merges = []
        if minrun != len(data):
            idx = 0
            current_idx = 0
            value_stack = Stack()
            direction = True  # True == ">=", False == "<"
            for i in range(len(data)):
                if idx != 0:
                    if idx == 1:
                        if current_idx + idx < len(data) and
abs(data[current_idx]) < abs(data[current_idx + idx]):
                            direction = False
                        else:
                            direction = True

                    elif idx >= minrun - 1:
                        flag = 0
                        if idx == minrun - 1:
                            add_idx = idx + 1
                        else:
                            add_idx = idx
                        if direction == True:
                            if current_idx + idx == len(data) or
abs(data[current_idx + idx - 1]) < abs(
                                    data[current_idx + idx]):
                                new_data =
InsertionSort(data[current_idx: current_idx + add_idx], True)
                                flag = 1
                        elif current_idx + idx == len(data) or
abs(data[current_idx + idx - 1]) >= abs(
                                data[current_idx + idx]):
                            new_data = InsertionSort(data[current_idx:
current_idx + add_idx], False)
                            flag = 1

                        if flag == 1:
                            if direction == False:
                                new_data = InsertionSort(new_data,
True)
                            data = data[:current_idx] + new_data +
data[current_idx + add_idx:]
                            parts.append(new_data)

                            galops, merges, value_stack =
push_and_merge(value_stack, new_data, galops, merges)
                            current_idx += add_idx
                            idx = 0

                    else:
```

```
                        new_data    =    InsertionSort(data[current_idx:
current_idx + idx], direction)
                        data   =   data[:current_idx]   +   new_data   +
data[current_idx + idx:]
                idx += 1

            if current_idx < len(data):
                new_data = InsertionSort(data[current_idx:], True)
                parts.append(new_data)
                galops,          merges,          value_stack          =
push_and_merge(value_stack, new_data, galops, merges)
            answer_data = final_merge(value_stack, galops, merges)

        else:
            answer_data = InsertionSort(data, True)
            parts.append(data)
        for i in range(len(parts)):
            print(f"Part {i}:", ' '.join(map(str, parts[i])))
        if len(galops) != 0:
            for i in range(len(galops)):
                print(f"Gallops {i}:", galops[i])
                print(f"Merge {i}:", ' '.join(map(str, merges[i])))
        return answer_data


    n = int(input())
    data = list(map(int, input().split()))

    data = TimSort(data)

    print("Answer:", ' '.join(map(str, data)))
```

ПРИЛОЖЕНИЕ Б

ТЕСТИРОВАНИЕ

Таблица Б.2 - Примеры тестовых случаев (функция check)

№ п/п	Входные данные	Выходные данные	Комментарии
1.	0 8 6 7 1 2 3 90 87 120 11 13 16 178 32 456 12 0 98 34 -1 -4 -32 87	Part 0: 178 120 90 87 16 13 11 8 7 6 3 2 1 0 Part 1: 456 98 87 34 32 -32 12 -4 -1 0 Gallops 0: 3 Merge 0: 456 178 120 98 90 87 87 34 32 -32 16 13 12 11 8 7 6 -4 3 2 1 -1 0 0 Answer: 456 178 120 98 90 87 87 34 32 -32 16 13 12 11 8 7 6 -4 3 2 1 -1 0 0	Ok