МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №1

по дисциплине «Алгоритмы и структуры данных»

Тема: Реализация и исследование алгоритма сортировки TimSort.

Студент гр. 3388 _____ Снигирев А.А.

Преподаватель _____ Шалагинов И.В.

Санкт-Петербург

2024

Цель работы

Исследовать методы оптимизации сортировки Timsort, такие как слияние массивов приблизительно равного размера с применением галопа, а также использование уже отсортированных подмассивов в исходном массиве.

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.

Выполнение работы

`bin_search(array,number)` - Функция выполняет бинарный поиск в массиве array, чтобы найти индекс первого числа, которое по модулю превосходит number. Это нужно, чтобы совершить слияние до этого индекса.

`insertion_sort(array, increase, sorted_end)` - сортирует массив array с использованием сортировки вставками.

`calculate_optimal_minrun(num, min_num=16)` – Вычисляет минимальную длину run для использования в Timsort.

`find_runs(array)` - Делит массив массив на отсортированные подмассивы. Если длина подмассива меньше minrun, в него добавляется нужное число элементов и используется сортировка вставками.

`stack_merge(stack, merges_number, debug, indexes)` - Сливает два отсортированных массива left_arr и right_arr в один. Использует метод галопа для улучшения производительности слияния.

`runs_merge(runs, debug = False)` - Сливает список отсортированных блоков runs в один отсортированный массив. Использует стек для управления слиянием.

`TimSort(array, debug=False)` - Основная функция, реализующая алгоритм Timsort. Сначала разбивает массив на подмассивы, затем сливает их в один отсортированный массив.

Исследование

Алгоритм сортирует массивы разных размеров данное количество времени:

Сортировка	10	1000	10000
Вставками	0.000041	0.027519	2.5
Подсчетом	0.001604	0.002052	0.004099
Быстрая	0.000038	0.002301	0.021556
Командная	0.000055	0.004510	0.066082

Как можно заметить, qsort и counting_sort работают быстрее, но и памяти они расходуют намного больше. Сортировка вставками же, которая также не использует доп память, сильно проигрывает тимсорт на больших данных.

В целом, Тимсорт эффективен. Он использует мало дополнительной памяти и довольно быстр. На небольших же данных сортировка вставками будет эффективнее.

Оценка сложности TimSort:

Лучший случай: O(n) (выбирается один большой отсортированный блок)

Средний случай: O(n*log(n))

Худший случай: O(n*log(n))

Вывод

Были реализованы функции-прелюдии и сам Timsort, а также проведены сравнения с некоторыми другими сортировками.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.py

```
from modules.timsort import TimSort
from datetime import datetime
from modules.constants import Constants

n = int(input())
array = [int(number) for number in input().split()]
a = datetime.now()
array = TimSort(array,debug=Constants.DEBUG)
print("Answer:",*TimSort(array,debug=Constants.DEBUG))
b = datetime.now()
print(b-a)
```

Название файла timsort.py:

```
from modules.constants import Constants


def find_runs(array):
    if len(array) <= 1:
        return [array]
    min_run = calculate_optimal_minrun(len(array))
    run_start = 0
    runs = []
    while run_start < len(array) - 1:
        end_idx = run_start
                  increase_flag  =  abs(array[run_start  +  1])  >
abs(array[run_start])
        if increase_flag:
          while end_idx < len(array) - 1 and abs(array[end_idx + 1])
> abs(array[end_idx]):
                end_idx += 1
        else:
          while end_idx < len(array) - 1 and abs(array[end_idx + 1])
<= abs(array[end_idx]):
                end_idx += 1

        if (end_idx - run_start + 1) < min_run:
            sorted_idx = end_idx
            end_idx = min(run_start + min_run - 1,len(array) - 1)
            current_run = array[run_start:end_idx+1]
          current_run = insertion_sort(current_run, increase_flag,
sorted_end=sorted_idx - run_start)
        else:
            current_run = array[run_start:end_idx + 1]
            if increase_flag:
                current_run = current_run[::-1]
        runs.append(current_run)
        run_start += len(current_run)

    return runs
```

```
    def runs_merge(runs, debug = False):
        stack = []
        merges_number = 0
        yx_mode = (-2,-1)
        zy_mode = (-3,-2)
        if debug:
            for i in range(len(runs)):
                print(f"Part {i}:", *runs[i])

        for run in runs:
            stack.append(run)
            while len(stack) >= 2:
                X = len(stack[-1])
                Y = len(stack[-2])
                if len(stack) > 2:
                    Z = len(stack[-3])
                    if Z > X + Y or Y > X:
                        break
                    if X < Z:
                      stack = stack_merge(stack, merges_number, debug,
indexes=yx_mode)
                        merges_number += 1
                    else:
                      stack = stack_merge(stack, merges_number, debug,
indexes=zy_mode)
                        merges_number += 1
                if X < Y:
                    break
                    stack = stack_merge(stack, merges_number, debug,
indexes=yx_mode)
                merges_number += 1

        if len(stack) >= 2:
            X = len(stack[-1])
            Z = None
            if len(stack) > 2:
                Z = len(stack[-3])
            if Z and X > Z:
                    stack = stack_merge(stack, merges_number, debug,
indexes=zy_mode)
                merges_number += 1
            else:
                    stack = stack_merge(stack, merges_number, debug,
indexes=yx_mode)
                merges_number += 1

        return stack[0]


    def stack_merge(stack, merges_number, debug, indexes):
        second, first = indexes
             stack[-2] = gallop_merge(stack[second], stack[first],
merges_number=merges_number, debug=debug)
        if debug:
```

```
            print(f"Merge {merges_number}:", *stack[second])
        stack.pop(first)
        return stack


    def    gallop_merge(left_array,right_array,    gallop_start    =
Constants.GALLOP_N, merges_number = -1, debug = Constants.DEBUG):
        gallops_num = 0
        res_array = []
        gallop_buffer = 0
        left = 1
        right = 2
        gallop_array = left
        while len(left_array) > 0 and len(right_array) > 0:
            merge_idx = 1
            if abs(left_array[0]) >= abs(right_array[0]):
                if gallop_array == left:
                    gallop_buffer += 1
                    if gallop_buffer == gallop_start:
                        merge_idx = bin_search(left_array, right_array[0])
                        gallop_buffer = 0
                        gallops_num+=1
                else:
                    gallop_array = left
                    gallop_buffer = 1
                res_array.extend(left_array[:merge_idx])
                left_array = left_array[merge_idx:]
            else:
                if gallop_array == right:
                    gallop_buffer += 1
                    if gallop_buffer == gallop_start:
                        merge_idx = bin_search(right_array, left_array[0])
                        gallop_buffer = 0
                        gallops_num += 1
                else:
                    gallop_array = right
                    gallop_buffer = 1
                res_array.extend(right_array[:merge_idx])
                right_array = right_array[merge_idx:]
        if debug and merges_number >= 0:
            print(f"Gallops {merges_number}: {gallops_num}")
        return res_array + left_array + right_array


    def bin_search(array,number):
        left = 0
        right = len(array)
        mid = (left + right)//2
        while right > left:
            mid = (left + right)//2
            if abs(array[mid]) < abs(number):
                right = mid
            else:
                left = mid+1
        return left
```

```
def insertion_sort(array, increase = False, sorted_end = 0):
    if increase == True:
        array = array[:sorted_end+1][::-1] + array[sorted_end+1:]
    for i in range(sorted_end+1,len(array)):
        array.insert(bin_search(array[:i],array[i]),array.pop(i))
    return array


def calculate_optimal_minrun(num, min_num = Constants.MIN_RUN):
    flag = 0
    while num >= min_num:
        flag = num%2
        num //= 2
    return num + flag


def TimSort(array, debug = False):
    runs = find_runs(array)
    return runs_merge(runs,debug=debug)
```