Студент: Кирилл Яценко
Группа: Яндекс
Дата: 10 октября 2023 г.

Задание 1

None

Задание 2

```
def get_sorted_sums(a, b):
    n = len(a)
    sums = []

    for ai in a:
        for bj in b:
            sums.append(ai + bj)

    sums.sort()

    return sums

#Пример использования:
a = [1, 3, 5]
b = [2, 4, 6]
sorted_sums = get_sorted_sums(a, b)
print(sorted_sums)
```

Задание 3

Алгоритм решения задачи можно описать следующим образом:

```
def can_place_cows(x, n, m, distance):
    placed_cows = 1
    last_cow = x[0]

    for i in range(1, n):
        if x[i] - last_cow >= distance:
            placed_cows += 1
            last_cow = x[i]

            if placed_cows == m:
                return True

    return False

def find_max_min_distance(x, n, m):
```

```
    x.sort()
    left = 1
    right = x[n - 1] - x[0]
    max_min_distance = -1

    while left <= right:
        distance = (left + right) // 2

        if can_place_cows(x, n, m, distance):
            max_min_distance = max(max_min_distance, distance)
            left = distance + 1
        else:
            right = distance - 1

    return max_min_distance

# Пример использования:
x = [1, 2, 4, 8, 9]
n = len(x)
m = 3

max_min_distance = find_max_min_distance(x, n, m)
print(max_min_distance)
```

В данной программе решается задача расстановки коров в стойлах. Алгоритм работает следующим образом:

1. Функция can_place_cows проверяет, можно ли расставить m коров в стойла с координатами x с заданным минимальным расстоянием distance между ними. Она последовательно просматривает координаты стоек, и если между двумя соседними есть достаточно места для коровы, то увеличивает счетчик расставленных коров.

2. Функция $find\_max\_min\_distance$ работает по принципу двоичного поиска. Сначала стойла сортируются по координатам. Затем устанавливаются переменные, отвечающие за левую и правую границы для поиска максимального минимального расстояния. В цикле выполняется бинарный поиск минимального расстояния. На каждой итерации проверяется, можно ли расставить коров с заданным расстоянием, и если да, то обновляется максимальное значение минимального расстояния. Если расстановка невозможна, то изменяется правая граница поиска. Поиск продолжается, пока левая граница не превысит правую.

3. В результате программа выводит максимальное минимальное расстояние, при котором можно разместить m коров в стойлах с координатами x.

Таким образом, программа находит решение задачи за время $O(m(logm + \log\max(x)))$.

Задание 4

(a) Для слияния k отсортированных массивов можно использовать heapq.merge из модуля heapq. Эта функция позволяет сливать несколько отсортированных итерируемых объектов в один отсортированный итератор.

```
import heapq

def merge_k_sorted_arrays(arrays):
    # Создаем кучу для хранения текущих минимальных элементов
    min_heap = []
```

```
    # Заполняем кучу из первых элементов входных массивов
    for i, arr in enumerate(arrays):
        if arr:
            heapq.heappush(min_heap, (arr[0], i, 0))

    merged = []
    while min_heap:
        val, arr_idx, idx = heapq.heappop(min_heap)
        merged.append(val)

        if idx + 1 < len(arrays[arr_idx]):
            # Если текущий входной массив еще не закончился, добавляем следующий элемент из него в кучу
            heapq.heappush(min_heap, (arrays[arr_idx][idx + 1], arr_idx, idx + 1))

    return merged
```

(b) Время работы сортировки слиянием, разбивающей каждый раз массив на k частей, можно оценить так:

На каждом уровне рекурсии мы разбиваем массив на k частей, что требует времени O(n) для создания каждой из них. Количество уровней рекурсии будет logk n.

На каждом уровне рекурсии происходит k слияний, каждое из которых работает за O(n), так как необходимо пройти через все элементы сливаемых массивов.

Таким образом, общее время работы сортировки слиянием, разбивающей каждый раз массив на k частей, составляет O(k logk n).

Задание 6

Доказательство:

Предположим, что для поиска максимума в массиве из n различных чисел требуется m сравнений, где $m < n - 1$.

Рассмотрим ситуацию, когда первые $m - 1$ чисел были сравнены друг с другом и максимальное число еще не найдено. Это означает, что максимальное число находится среди оставшихся $n - m + 1$ чисел.

Так как предположение состоит в том, что m < n - 1, мы можем сделать вывод, что $n - m + 1 > 2$.

Теперь рассмотрим ситуацию, когда мы сравниваем первое число с оставшимися $n - m + 1$ числами. Мы получаем информацию о том, какое из этих чисел является большим, и у нас остается n-m чисел, из которых нужно найти максимум.

Заметим, что мы можем повторить этот процесс еще $(n - m - 1)$ раз, сравнивая каждый раз максимальное число с оставшимися числами.

После каждого сравнения количество оставшихся чисел уменьшается на 1.

Поэтому минимальное количество сравнений, необходимых для поиска максимума, равно $(m - 1) + (n - m) + (n - m - 1) + ... + 1 = (n - 1) + (n - 2) + ... + 1 = (n - 1)n/2$.

Следовательно, для поиска максимума в массиве различных чисел потребуется как минимум n - 1 сравнений.

Задание 6

Найти второй максимум в массиве за $n + O(logn)$ сравнений.

Решение:

Для нахождения второго максимума в массиве можно применить модифицированный алгоритм сортировки слиянием, который будет выполнять только необходимые сравнения.

Алгоритм:

1. Разделим массив на две части и отсортируем каждую часть рекурсивно с использованием того же алгоритма.

2. Найдем максимальный элемент в каждой части массива.

3. Сравним два найденных максимальных элемента и определим, какой из них больше.

- Если левый максимальный элемент больше, то второй максимум будет находиться в правой части массива. В этом случае рекурсивно применим алгоритм для правой части массива.

- Если правый максимальный элемент больше, то второй максимум будет находиться в левой части массива. В этом случае рекурсивно применим алгоритм для левой части массива.

4. В конце алгоритма мы найдем второй максимум в массиве.

Пример реализации на Python:

```
def find_second_maximum(arr):
    def merge_sort(arr):
        if len(arr) <= 1:
            return arr

        mid = len(arr) // 2
        left = merge_sort(arr[:mid])
        right = merge_sort(arr[mid:])

        return merge(left, right)

    def merge(left, right):
        merged = []
        i = j = 0
        while i < len(left) and j < len(right):
            if left[i] > right[j]:
                merged.append(left[i])
                i += 1
            else:
                merged.append(right[j])
                j += 1

        while i < len(left):
            merged.append(left[i])
            i += 1

        while j < len(right):
            merged.append(right[j])
            j += 1

        return merged

    sorted_arr = merge_sort(arr)
    return sorted_arr[1]
```

Время работы алгоритма состоит из двух частей: время выполнения сортировки слиянием (O(n log n)) и одного сравнения поиска второго максимума. Общее время работы будет O(n log n) + O(1) = O(n log n).

Таким образом, мы можем найти второй максимум в массиве за n + O(log n) сравнений.

Задание 7

none