Студент: Александр Никулин
Дата: 21 сентября 2020 г.

Задача 1. *Есть m стойл с координатами $(x_1, \ldots, x_m)$ и n коров. Расставить коров по стойлам (не более одной в стойло) так, чтобы минимальное расстояние между коровами было максимально.* $\mathcal{O}\left(m\left(\log m + \log x_{\max}\right)\right)$.

Решение. Предположим, что нужно решить обратную задачу и нам дано минимальное расстояние и надо проверить, можем ли мы расставить n коров. Проверить это очень просто, достаточно просто идти с шагом в это расстояние, расставляя коров, и если отрезок кончился, а коровы еще нет - то не можем. Функция количство коров, таким образом, невозрастающая - при малом расстоянии коров можно расставить много, а при большом мало. Это значит, что мы можем использовать бинарный поиск по количеству коров.

Минимальным расстоянием может быть 1, максимальным $x_{max} - x_{min}$. Нам также необходимо отсортировать координаты, чтобы можно было расставлять коров по порядку. Теперь, нам достаточно выбрать среднее расстояние и проверить, можем ли мы расставить n коров, если можем, то надо обновить максимальное расстояние и идти влево, если нет - то вправо. Асимптотика соответственно $\mathcal{O}(m \log m + \log(x_{max} - x_{min}))$

Решение на python:

```python
def if_possible(min_dist, arr, k):
    count, last = 1, arr[0]

    for i in range(1, len(arr)):
        if arr[i] - last >= min_dist:
            count += 1
            last = arr[i]

            if count == k:
                return True
    return False

def max_min_dist(arr: List[int], k: int) -> int:
    sorted_arr = sorted(arr)
    left, right = sorted_arr[0], sorted_arr[-1]

    best_dist = 0
    while left <= right:
        min_dist = (left + right) // 2

        if if_possible(min_dist, sorted_arr, k):
            best_dist = max(best_dist, min_dist)
            left = min_dist + 1
        else:
            right = min_dist - 1

    return best_dist
```

.

Задача 2. *Даны два сортированных массива длины n, которые нельзя модифицировать. Найдите k-ю порядковую статистику в объединении массивов (то есть элемент, находившийся бы на k -ой позиции если бы массивы слили), используя $\mathcal{O}(1)$ дополнительной памяти.*

(a) *За* $\mathcal{O}(\log^2 n)$
(b) *За* $\mathcal{O}(\log n)$

Решение.

(a) Нам нужно найти элемент такой, что в итоговом отсортированном массиве будет $k-1$ элементов меньше его. Минимальным может быть значние $min(arr_1[0], arr_2[0])$, а максимальным $max(arr_1[-1], arr_2[-1])$. Можно также заметить, что количство элементов меньше данного является неубывающей функцей т.к. оба масссива отсортированы. А значит, мы можем использовать бинарный поиск. Для поиска количества элементов меньше данного также будем использовать бинарный поиск. Итоговая асимптотика $\mathcal{O}(\log x_{max} \log n)$.

Решение на python:

```python
def k_smallest_naive(arr1: List[int], arr2: List[int], k: int) -> int:
    left = min(arr1[0], arr2[0])
    right = max(arr1[-1], arr2[-1])

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

        smaller = bisect_right(arr1, mid) + bisect_right(arr2, mid)

        if smaller == k:
            return mid
        elif smaller < k:
            left = mid + 1
        else:
            right = mid - 1

    return left
```

.

(b) Можно получить результат лучше, если искать сразу границы, а не k-ю порядковкую статистику напрямую. Предположим у нас есть два массива $\{x_i\}$ и $\{y_i\}$:

$$x_1, x_2, x_3, x_4, x_5, x_6$$

$$y_1, y_2, y_3, y_4, y_5, y_6$$

Если мы разобьем первый массив по x_3, то нам останется взять из второго массива только $k-3$ элементов, чтобы в сумме получить k. Для того чтобы данное разбиение дало ответ, нужно чтобы все элементы из первого подмассива $\leq$ элементов из второго подмассива. Что эквивалентно такому условию:

$$\begin{cases} x_i \leq y_{k-1-i} \\ y_{k-2-i} \leq x_{i+1} \end{cases}$$

Если же одно из этих условий не выполняется, то нам нужно двигать границу либо влево, либо вправо.

Таким образом мы можем получить $\mathcal{O}(\log^2 n)$, если будем искать бин. поиском оба разбиения для первого массива и для второго, но достаточно и только одного, то есть $\mathcal{O}(\log n)$.

Решение на python: его нет, я запутался в индексах и сдох. Есть пример для медианы.

```python
def merged_median(arr1, arr2):
    part_size = (len(arr1) + len(arr2)) // 2

    left, right = 0, len(arr1)
    while left <= right:
        mid = (left + right) // 2

        left_x, left_y = arr1[mid - 1], arr2[part_size - mid - 1]
        right_x, right_y = arr1[mid], arr2[part_size - mid]

        if left_x <= right_y and left_y <= right_x:
            break
        elif left_x > right_y:
            right = mid - 1
```

```
        else:
            left = mid + 1

    return max(left_x, left_y)
```

.

Задача 3. *В свободное время Анка-пулемётчица любит сортировать патроны по серийным номерам. Вот и сейчас она только разложила патроны на столе в строго отсортированном порядке, как Иван Васильевич распахнул дверь с такой силой, что все патроны на столе подпрыгнули и немного перемешались. Оставив ценные указания, Иван Васильевич отправился восвояси. Как оказалось, патроны перемешались не сильно. Каждый патрон отклонился от своей позиции не более чем на k. Всего патронов n. Помогите Анке отсортировать патроны.*

(a) *Отсортируйте патроны за $\mathcal{O}(nk)$.*
(b) *Отсортируйте патроны за $\mathcal{O}(n + I)$, где $I-$ число инверсий.*
(c) *Докажите нижнюю оценку на время сортировки $\Omega(n \log k)$.*
(d) *Отсортируйте патроны за $\mathcal{O}(n \log k)$*

Решение.

(c) Из общего свойства сортировок, основанных на сравнениях, мы знаем, что нижняя граница определяется как $\Omega(n \log n)$. Тогда, отсортируем первые $2k$ элементов. Из условия мы знаем, что так на своих местах окажутся ровно первые k элементов, в то время как остальная половина все еще может быть в k от исходной позиции. Отсотируем [k:3k], [2k:4k] и т.д. Всего таких отрезков $\frac{n}{2k}$. Получаем время работы равное $\Omega(2k \log 2k) * \frac{n}{2k} \Rightarrow \Omega(\frac{n}{2k} 2k \log 2k) \Rightarrow \Omega(n \log k)$

(a) Воспользуемся простым алгоритмом сортировки - insertion sort. Предположим, что $a_1, \cdots, a_n$ отсортированы, тогда, чтобы добавить a_{n+1} нужно сравнить его с $a_n, a_{n-1}, \cdots$, найдя его место, после чего сдвинуть все после него. В нашем же случае, нам придется сравнить только с k предыдущих элементов. На первом шаге последовательность отсортирована.

```
def insertion_sort(arr, k):
    for i in range(len(arr)):
        j = i

        while j > 0 and arr[j] <= arr[j - 1]:
            arr[j], arr[j - 1] = arr[j - 1], arr[j]
            j = j - 1
    return arr
```

.

Нам нужно сравнить каждый элемент с k предыдущих, итого $\mathcal{O}(nk)$.

(b) Код аналогичен предыдущему пункту. Достаточно заметить, что каждый обмен во внутреннем цикле уменьшает количество инверсий на 1. Всего обменов будет столько же, сколько в сумме инверсий в данной последовательности. Более формально, пусть $a_1, a_2, \cdots, a_n$ последовательность, а $v_1, v_2, \cdots, v_n$ последовательность инверсий, где $v_i = |\{(i,j) \mid i < j \wedge a_i > a_j\}|$ - количество элементов левее a_i, которые больше его. Тогда на каждой итерации, чтобы вернуть a_{i+1} на свое место, нужно исправить все инверсии, в которых есть $i+1$, то есть v_i. В сумме: $\sum_0^n v_i = I$.

(d) Воспользуемся кучей. Мы можем за $\mathcal{O}(k)$ создать кучу на первых k элементах. Далее, доставать минимальный и добавлять новый из массива. Добавление и минимум за $\mathcal{O}(\log k)$. В итоге $\mathcal{O}(k + (n-k)\log k) \Rightarrow \mathcal{O}(n \log k)$

```
def inversion_heap_sort(arr, k):
    sorted_arr = []
    heap = arr[:k+1]

    heapify(heap)

    for i in range(k + 1, len(arr)):
```

```python
        sorted_arr.append(heappop(heap))
        heappush(heap, arr[i])

    while heap:
        sorted_arr.append(heappop(heap))

    return sorted_arr
```

.