Алгоритмы. Домашнее задание №3

Горбунов Егор Алексеевич

2 октября 2015 г.

Задача №1 (про слегка перемешавшиеся патроны)

(a) [сортировка за $\mathcal{O}(nk)$] Ясно, что т.к. 1-ый по порядку патрон не мог оказаться дальше, чем на k-ой позиции, то за $\mathcal{O}(k)$ операций легко найти его, пробежавшись по первым k элементам и поменять местами с текущим патроном на первой позиции. Теперь на первой позиции нужный патрон. Тогда аналогично для 2-ого по порядку патрона: он точно не дальше, чем на $k+1$ позиции. Тогда опять за $\mathcal{O}(k)$ операций находим его и сажаем на 2-ую позицию. И т.д. мы после i-ого шага будем иметь первые i патронов в отсортированном порядке, причём операций затрачено $\mathcal{O}(ki)$. Итого в конце будем иметь отсортированный массив, за $\mathcal{O}(kn)$ ■

(b) [сортирока за $\mathcal{O}(n+I)$] Для $i=0$ число инверсий в массиве $\leq k-1$. Аналогично для остальных i. Но тогда инверсий в перестановке патронов $\mathcal{O}(n(k-1)) = \mathcal{O}(nk-n)$. Т.е. $I \leq (nk-n)$. У нас есть алгоритм из предыдущего пункта, который работает за $\mathcal{O}(nk)$. Но $\mathcal{O}(nk) = \mathcal{O}(n+(nk-n)) = \mathcal{O}(n+I)$ ■

(c) Предположим, что можем отсортировать патроны быстрее, чем за $\Omega n \log k$, но тогда, если $k=n$, то можем отсортировать обычный массив, без доп. условий на элементы, быстрее, чем за $\mathcal{O}(n \log n)$, что невозможно ■

(d) Рассмотрим алгоритм из пункта (a). На каждом i-ом шаге этого алгоритма мы ищем минимум среди элементов с номерами $i, \ldots, i+k$. Заметим, что это можно реализовать используя кучу: добавим первые k элементов массива в кучу, с операцией $extractMin$, извлечём минимум, положим его на место первого элемента массива (тут куча не могла сломаться, если что), а потом добавим в кучу $k+1$ элемент и снова извлечём минимум. Таким образом всего операций с кучей $\mathcal{O}(n)$, а высота кучи всегда $\mathcal{O}(k)$, а значит суммарная сложность алгоритма $\mathcal{O}(n \log k)$ ■

Задача №2 (p_i-ые порядковые статистики)

Задача №3 (про перестановку p максимизирующую сумму $a_{p(i)}b_i$)

Очевидно, что максимальная такая сумма равна:

$$\sum_{i=1}^{n} a_{sort_a(i)} b_{sort_b(i)}$$

Действительно, пускай в сумме $\sum_{i=1}^{n} a_i b_i$ нет слагаемого $a_{max} b_{max}$ (a_{max} — максимальный элемент в a, b_{max} аналогично, но вместо него есть слагаемые $a_{max} b_i + b_{max} a_j$. Покажем тогда, что $a_{max} b_{max} + a_j b_i > a_{max} b_i + b_{max} a_j$, т.е. произведение максимальных брать выгоднее:

$$(a_{max} b_i + b_{max} a_j) - (a_{max} b_{max} + a_j b_i) = a_{max}(b_i - b_{max}) + a_j(b_{max} - b_i) = (a_{max} - a_j)(b_i - b_{max}) \le 0$$

Тут $sort_a$ - перестановка, соответствующая сортировке элементов массива a по убыванию (или по возрастанию). Аналогично $sort_b(i)$ Тогда перестановку p нужно задать таким образом, чтобы, $a_{p(i)}$ был элементом, стоящим на той же позиции в a после его сортировки, что и b_i в массиве b после сортировки. Ясно, что такая перестановка это:

$$p = sort_a^{-1} \circ sort_b$$

Тогда соответственно:

$$p(i) = sort_a^{-1} \circ sort_b(i) = \overbrace{sort_a^{-1}(\overbrace{sort_b(i)}^{\text{индекс } b_i \text{ в отсортированном массиве } b})}^{\text{индекс элемента в массиве } a\text{, что после сортировки попал бы на позицию } i}$$

тут $sort_a^{-1}$ — обратная перестановка. ■

Задача №4 (про кучу)

Построим такую кучу, что при сортировке число вызовов операции `SiftDown` максимально. В данной задаче рассматривается max-куча. Чтобы максимизировать число вызовов `SiftDown` при сортировке, будем строить кучу так, чтобы число вызовов `SiftUp` было максимальным, а это будет тогда, когда элементы поднимаются с самого конца кучи. Собственно алгоритм будет «обратен» алгоритму сортировке:

1. Добавить в кучу элемент i (в начале $i = 1$, потом 2 и т. д.):
2. Заменить его с 1 в куче
3. Выполнить для элемента i `SiftUp`
4. Вставить 1 в конец кучи

Элемент 1 после каждого добавления находится в конце кучи, что обеспечивает нам максимум вызовов операции `SiftUp`. n вызовов операции `SiftUp` дают временную сложность $\mathcal{O}(n \log n)$. Алгоритм обратен алгоритму сортировки, а значит число вызовов `SiftDown` будет так же максимально при сортировке, как и число вызовов `SiftUp`. ■