# 1 Problem List 2

Задача 1.1. The input is an integer array $a$ of $n$ elements. It is required to find the number of inversions in $a$, i.e., the pairs of indices $(i, j)$ such that $i < j$ and $a[i] > a[j]$. Construct an $O(n \log n)$ algorithm that finds the number of inversions in $a$.

Доказательство. Будем решать задачу методом, схожим с Merge sort, разделяя список на две равных (или почти равных) половины пока не будет достигнут базовый случай Введем функцию Merge которая подсчитывает число инверсий при объединении двух половин следующим образом: Если $a[i] > a[j]$ (для $i$ - индекс элемента из первой половины, $j$ - из второй) то в списке хотя бы $(\text{mid} - i)$ инверсий (mid - индекс последнего элемента из первой половины), так как $a[k] > a[j]$ для всех $k > i$ Базовым случаем будет являться деление множества на два одноэлементных множества (в нем число инверсий подсчитывается путем сравнения элемента 1го множества с элементом 2го, если первый больше то есть 1 инверсия, иначе 0) Шагом в данном случае будет являться объединение 2 множеств, использование Merge для подсчета инверсий элементов первого множества относительно второго и суммирование результата с имеющимся числом инверсий для каждого из множеств.

Примерный код:

```
int MergeSort(int arr[], int arraySize){
    int temp[arraySize];
    return _MergeSort(arr, temp, 0, arraySize - 1);
}


int MergeSort(int arr[], int temp[], int left, int right){
    int mid, Inv = 0;
    if (right > left) {
        mid = (right + left) / 2;
        Inv += _MergeSort(arr, temp, left, mid);
        Inv += _MergeSort(arr, temp, mid + 1, right);
        Inv += Merge(arr, temp, left, mid + 1, right);
    }
    return Inv;
}


int Merge(int arr[], int temp[], int left, int mid, int right){
    int i, j, k;
    int Inv = 0;

    i = left;
    j = mid;
    k = left;
    while ((i <= mid - 1) && (j <= right)) {
        if (arr[i] <= arr[j]) {
            temp[k++] = arr[i++];
        } else {
            temp[k++] = arr[j++];
            Inv = Inv + (mid - i);}
    }

    while (i <= mid - 1)
        temp[k++] = arr[i++];

    while (j <= right)
        temp[k++] = arr[j++];

    for (i = left; i <= right; i++)
        arr[i] = temp[i];

    return Inv;
}
```

□

Задача 1.2. RAM stores $k$ arrays $A^1, A^2, \ldots, A^k$, each of which stores integers from 1 to $n$, and the sum of array lengths (total number of elements) is also equal to $n$. Build an algorithm that sorts all arrays in $O(n)$.

Доказательство. Введем $n$ стеков - каждый будет соответствовать числу от 1 до $n$. Далее пройдем поочередно по каждому из массивов (в любом порядке), если в массиве $A_i$ встретилось число $j$ - положим в стек $j$ число $i$ (если в $A_i$ число $j$ встретилось $x$ раз, положим в стек $i$ $x$ раз в стек $j$). Таким образом в стеке $i$ лежат номера массивов, где встречается число $i$ и если $i$ встречается в каком-то массиве несколько раз то его номер такое же число раз встречается в стеке. Теперь мы можем пройтись поочередно по каждому стеку (от 1 к $n$), расставляя числа обратно - то есть сперва мы расставим 1 в соответствии с тем, какие номера лежат в первом стеке, далее проведем аналогичные действия для $2, 3, \ldots$ стеков. Таким образом мы обратились к каждому числу массива 1 раз, а потом в соответствии со стеком просто расставили числа, также обращаясь к каждому 1 раз, сделав таким образом $2n$ действий (то есть сложность $O(n)$) □

Задача 1.3. Let the integer array $a[1], \ldots, a[n]$ be strictly unimodal to the maximum. It means that there exists $t$ such that

$$a[1] < a[2] < \cdots < a[t] > a[t+1] > \cdots > a[n-1] > a[n], \quad 1 \leqslant t \leqslant n.$$

It is allowed to retrieve a single array's element per move (the input of the query is $i$ the output is $a[i]$). Prove that it is possible to find the maximum element $a[t]$ in at most $O(\log n)$ moves.

Доказательство. Введем переменные $L = \frac{1}{2}$ и $H = n + \frac{1}{2}$, а также $M = \left[\frac{H+L}{2}\right] + \frac{1}{2}$. Шаг алгоритма: рассмотрим элементы $a\left[M - \frac{1}{2}\right]$ и $a\left[M + \frac{1}{2}\right]$ и сравним их, если $a\left[M - \frac{1}{2}\right] < a\left[M + \frac{1}{2}\right]$ то $L = M$, если $a\left[M - \frac{1}{2}\right] > a\left[M + \frac{1}{2}\right]$ то $H = M$, $a\left[M - \frac{1}{2}\right] = a\left[M + \frac{1}{2}\right]$ не достигается так как мы рассматриваем два соседних элемента в strictly unimodal списке. На каждом шагу расстояние $L - H$ уменьшается и соответственно когда-то мы достигнем случая когда это будут соседние числа (из которых мы просто возьмем наибольшее). Расстояние $L - H$ на каждом шагу уменьшается примерно в два раза, а число обращений к массиву и сравнений на каждом шагу - константа, соответственно сложность $O(\log n)$. □

Задача 1.4. There are $n$ coins, the one of which is fake, and a pan balance that can be used to find which of any two given coins is heavier. Real coins all have the same weight, while a fake coin is lighter. You can put any number of coins on each pan. Prove that the fake coin can be found in $\log_3 n + c$ weighings.

Доказательство. Назовем множество монет однообразным если в нем нет фальшивых монет и уникальным если есть. Обозначи множество как $C = \{c_1, \ldots, c_n\}$ для $n \geq 2$, обозначим число монет как $|A|$ и их массу как $||A||$. Алгоритм будет состоять из деления всех монет на 3 примерно равных множества $C_1, C_2, C_3$ (хотя бы 2 из них в любом случае будут равны), после чего мы будем сравнивать 2 равных (без ограничения общности 1 и 2), если $||C_1|| = ||C_2||$, то на следующем шаге рассматриваем $C_3$ в случае неравенства рассматриваем множество с меньшей массой, заметим, что данный алгоритм находит монету за $\log_3(n) + c$. □

Задача 1.5. Prove that under the conditions of the previous problem, finding the fake coin requires $\log_3 n + c$ weighings.

Доказательство. Заметим что если представлять результаты взвешивания как ориентированное decision tree то из каждой вершины будет выходить не более 3 ребер (случаи $>$, $=$, $<$), каждый возможный исход представлен в дереве, то есть число вершин не меньше чем число исходов, то есть для дерева глубины $a$ выполнено

$n \leq$ число вершин $\leq \sum_{i=0}^{a} 3^i = \frac{3^{a+1}-1}{2} < 3^{a+1}$, откуда $\log_3(n) < a+1$ и $a > \log_3(n) - 1$, а глубина дерева и является числом взвешиваний. □

Задача 1.6. There are two sorted arrays of length $n$ of different elements. Propose $O\left(\log^2 n\right)$ algorithm for finding the median in the array consisting of all given $2n$ elements. Prove the correctness of the algorithm and estimate its complexity (number of comparisons). In this problem, retrieving an element is performed in O(1). Assume that both arrays are stored in RAM (so it is not needed to read the input).

Доказательство. Заметим что взятие медианы отсортированного списка занимает $O(1)$ - так как мы можем просто взять элемент, находящийся в середине списка за $O(1)$.
Для поиска медианы всех элементов за $O(\log n)$ будем следовать следующим правилам:

(1) Если ($\text{length}(A) \leq 2$ или $\text{length}(B) \leq 2$) или ($A_{\text{last}} \leq B_{\text{first}}$ или $B_{\text{last}} \leq A_{\text{first}}$) сразу вычислим медиану.

(2) Пусть $A_m = \text{median}(A), B_m = \text{median}(B)$, сравним их. Если $A_m = B_m$ вернем результат. Если $A_m < B_m$ то выкинем первую половину $A$ и это же число элементов с конца $B$. Иначе если $A_m > B_m$ выкинем вторую половину $A$ и то же число элементов начиная с 1 из $B$

(3) Повторяем (2) пока какое-либо из условий (1) не начнет выполняться

Данный алгоритм работает $O(\log n)$ - уменьшая на каждом шагу длину списков в 2 раза (и прерываясь не позднее чем через $O(\log n)$ если $A_m = B_m$) □

Задача 1.7. Determine whether the given number is the value of the given polynomial with natural (positive integer) coefficients at the natural point. Natural numbers $n, a_0, \ldots, a_n$, and $y$ are the problem's input. It is necessary to determine whether there exists a natural number $x$ such that

$$y = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$$

Construct an $O(n \log y)$ algorithm that solves this problem. Assume that arithmetic operations cost $O(1)$.

Доказательство. Заметим, что так как все коэффициенты функции - натуральные числа, то функция строго возрастающая. Тогда, если рассмотреть $f(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + (a_0 - y)$ то натуральные корни будут лежать между 0 и $y - a_0$ (так как $n \geq 1$, $a_1 \geq 1$ то функция растет не медленнее линейной, а следовательно у нее не может быть корней больше модуля свободного члена (и если $a_0 \geq y$ то натуральных корней нет). Тогда построим алгоритм бин поиска в котором мы будем сравнивать значение функции в точке $0 \leq x \leq y - a_0$ с 0, пытаясь найти $f(x) = 0$. Заметим, что таким образом мы получим алгоритм сложности $O(\log(y)) +$ □

Задача 1.8. You have a 100-story building and a couple of marbles. You must identify the lowest floor for which a marble will break if you drop it from this floor. How fast can you find this floor if you are given an infinite supply of marbles? What if you have only two marbles? If the marble had not broken after the drop, it is as solid as before the drop, i.e. the number of flow had not changed.

Доказательство. Если шариков бесконечное количество, то мы можем провести бинарный поиск этажа за $O(\log_2 n) + c$ (так как по факту это поиск числа, в данном случае этажа, среди чисел $\{1, \ldots, 100\}$). Заметим что быстрее найти этаж невозможно, так как по факту бросок шарика имеет 2 исхода (шарик разбился или не разбился), соответственно если представлять результаты бросания шара как ориентированное decision tree то из каждой вершины будет выходить не более 2 ребер (шар разбился или нет), каждый возможный исход представлен в дереве, то есть число вершин не меньше чем число исходов, то есть для дерева глубины $a$ выполнено $n \leq$ число вершин $\leq \sum_{i=0}^{a} 2^i = \frac{2^{a+1}-1}{2} < 2^{a+1}$, откуда $\log_2(n) < a+1$ и $a > \log_2(n) - 1$, а глубина дерева и является числом бросков.

Если же шариков всего 2 то можно заметить что если рассматривать действия как дерево, то у нас степень каждой вершины также будет не больше 2, однако если шарик разбился, то все вершины случая “шарик разбился” поддерева данного случая - остовные и, соответственно, если шарик разбился на этаже $i$ и самый высокий проверенный этаж до $i$ - $j$, то мы должны будем проверить все этажи между $i$ и $j$ чтобы найти самый низкий этаж на котором разбивается шарик, то есть если мы проверяем каждый $k$й этаж до того как первый шарик разбился, мы проведем $\frac{n}{k} + k - 1$ проверок, при $n = 100$ данная функция достигает минимума при $k = 10$, то есть мы будем проверять каждый этаж, номер которого кратен 10, пока хотя бы один шарик не разобьется, после чего поочередно снизу вверх проверим 9 этажей под этижем, на котором шарик разбился. □