Домашнее задание №3

Томинин Ярослав, 778 группа

11 марта 2018 г.

1.

Воспользуемся пройденным алгроитмом. Запишем слова в столбик. Далее сравним возьмем их последние буквы и по ним отсортируем наши слова за O(n). Поступать будем следующим образом: обозначим за z количество латинских букв, созадим массив a и за O(n) операций запишем в него количество повторений каждой буквы. Теперь создадим массив к, в которм на i месте будет стоять сумма всех предыдущих повторений. Теперь отсортируем сам массив. Посмотрим на 1 букву. Найдем значение цифру, сопоставленную этой букве в массиве к, после этого запишем ее на это место и увеличим цифру в массиве к на 1. Так будем делать и дальше. Потом отсортируем 2 стобик из букв и т.д. Тогда, как мы знаем сложность этого алгоритма O(n), где n- длина входа.

2.

1)Каждый раз будем выбирать средний $\frac{n}{2}$ элемент и спрашивать: действительно ли $a[i] < a[i+1]$?(то есть будем узнавать значене двух элементов) Если это так то будем далее задавать тот же вопрос, но уже для массива $\frac{n}{2}+1, n$. Тогда длина массива каждый раз будет уменьшаться в 2 раза. Следовательно мы будем делать 2 действия $\log n$ раз. А это и есть $(\log n)$.

Корректность алгоритма: Допустим, что так получилось и мы потеряли наш максимум, это означет, что в какойто момент он был не включен в массив. Но что это может означать? Это означает то, что $a[i] < a[i+1]$, но наш элемент лежит слева от $\frac{n}{2}+1$ или же $a[i] > a[i+1]$, но наш элемент лежит справа от $\frac{n}{2}$. В силу нашего условия и той и другой ситууации быть не может.

2)Разделим наш отрезок следующим обращом.

$$\frac{b-a}{b-x_1} = \frac{b-a}{x_2-a} = \Phi$$

где $\Phi = \frac{1+\sqrt[2]{5}}{2}$(золотое сечение)

Тогда по св-ву золотого сечения x_1 делит отрезок $[a, x_2]$ в отношении золотого сечения. Теперь будем сравнивать $f(x_1), f(x_2)$. Будем выбирать большее из них и один из отрезков $[a, x_2]; [x_1, b]$, которому это большее принадлежит. По свойству золотого сечения у нашего отрезка будет уже выбрана одна точка(мы знаем ее значение), поэтому нам остается спросить значение второй точки. Так алгоритм будет работать до тех пор, пока мы не доберемся до наибольшего элемента нашего массива(массив каждый раз уменьшается в Ф раз). Следовательно мы спросим значение $\log_{\frac{1+\sqrt[2]{5}}{2}} n + c$ раз

Корректность алгоритма: В силу того, что наша функция унимодальна, мы можем утверждать, что наше максимальное значение будет всегда в рассматриваемом массиве. Так как если $f(x_1) < f(x_2)$, то это означает, что справа от x нет максимума, так как значение элементов слева от x_1 меньше значения x_1. Аналогично, если $f(x_1) > f(x_2)$. Тогда значения справа от x_2 будут меньше значения x_2. Следовательно мы можем безнаказанно откинуть эту часть массива. Каждый раз массив уменьшается в Ф раз, следовательно мы доберемся до нашего максимального элемента.
3) Докажем от обратного. Допустим, что она не унимодальна. Тогда существует либо слева от максимума точка, в которой функция строго не убывает, либо справа от максимума, в которой функция строго не возрастает. Тогда это означает что первая производная в первом случае ≥ 0, а во втором случае ≤ 0. Но необходимое условие выпуклости- вторая производная всегда неотрицательна. Следовательно наша первая производная непрерывна и всегда возрастает. Это означает, что в первом и во втором случаях был участок, на которм производная точно не возрастала(так как в минимуме она обращается в 0). А такого быть не может. Второе доказательства прямо следует из курса математического анализа, потому что условие $f''(x) > 0$ является достаточным для строгой выпуклости. Выходит оно из следующего уравнения(модифицировванное условие выпуклости):

$$f(x) - l_{\alpha,\beta}(x) = \frac{(x-\alpha)(\beta-x)f''(\Theta)(\epsilon-\nu)}{\beta-\alpha}$$

где $a < \alpha < \epsilon < \Theta < \nu < \beta < b$
4) Воспользуемся полученными знаниями, введем функцию суммы расстояний от выбранной точки x_0 она равна

$$R = \sum_{i=1}^{k} \sqrt[2]{(x_i - x_0)^2 + y_i^2}$$

Возьмем первую и вторую производные

$$R'(x) = \sum_{i=1}^{k} \frac{(x_0 - x_i)}{\sqrt{(x_0 - x_i)^2 + y_i^2}}$$

$$R''(x) = \sum_{i=1}^{k} \frac{\sqrt{(x_0 - x_i)^2 + y_i^2} - \frac{(x_0 - x_i)}{\sqrt{(x_0 - x_i)^2 + y_i^2}}(x_0 - x_i)}{(x_0 - x_i)^2 + y_i^2} = \sum_{i=1}^{k} \frac{y_i^2}{((x_0 - x_i)^2 + y_i^2)^{\frac{3}{2}}} > 0$$

Отсюда следует, что наша функция унимодальна. Поэтому мы можем устроить бинарный поиск минимума. Каждый раз будем вычислять две суммы для точек $x_0-\epsilon, x_0, x_0+\epsilon$. И будем выбирать левую часть массива, если $R(x-\epsilon) < R(x)$; правую, если $R(x+\epsilon) < R(x)$. В противном случае мы уже найдем минимум. Каждый раз мы будем делать O(k) действий, а повторять мы это будем $\log \frac{N}{\epsilon}$ раз. Следовательно сложность алгоритма равна $O(k \log \frac{N}{\epsilon})$

3.

Будем делить монеты на три кучки и взвешивать произвольные две из них. При равенстве дальнейшие действия будем проводить с 3 кучкой, в противном случае будем брать меньшуу из кучек. Так как искомая монета легче, то она всегда будет лежать в более легкой кучке. Будем повторять эти действия до тех пор, пока не останется одной монеты.

Сложность алгоритма: Наш алгоритм делит кучки на три до тех пор, пока не останется 1 монета, после каждого деления производится одна операция взвешивания. Следовательно всего операций будет $\log_3 n + c$. Что и требовалось доказать.

Корректность алгоритма: Допустим, что мы нашли не ту монету, следовательно в какой-то момент мы выбрали кучку в которой не лежала наша искомая монета. Следовательно наша кучка весила больше кучки, в которой лежит искомая монета, но такого не может быть, так как мы выбираем всегда более легкую кучку.

4.

Допустим, что можно найти монетку меньше, чем за $\log_3 n + c$ взвешиваний. Выберем после первого взвешивания кучку(у нас будет 3 кучки :(та, которая взвешивалась на 1 чаше; та, которая взвешивалась на 2 чаше и которая вообще не взвешивалась)), в которой наибольшее количество монет и положим туда нашу фальшивую монету(мы играем против нашего алгоритма). В дальнейших взвешиваниях мы будем следить за нашей кучкой. Самое интересное, что после первого взвешивания мы не знаем о взаимоотношениях монет между друг другом(любая может быть фальшивой). Информацию о монетах мы можем узнать, только ести какая-то часть из них будет учавствовать в взвешивании. После второго взвешивания наша кучка(за которй мы следим) опять поделится на 3 кучки(та, которая взвешивалась на 1 чаше; та, которая взвешивалась на 2 чаше и которая вообще не взвешивалась). Может оказаться так, что некоторые из них будут пустыми, но это не мешает нашему алгоритму. Выберем наибольшую кучку. Мы опять не знаем о взаимоотношениях монет в ней. Это означает, что чтобы точно понять где лежит монета нам надо не меньше $\log_3 n$ взвешиваний. Следовательно такого алгоритма не су-

ществует.

5.

Обозначим два массива a и b.(где a[n]<b[n]) Поймем, что если $a[i]$ меньше, чем b[n-i], то справа от a[i] находится больше, чем n элементов. Если же $a[i]$ больше, чем b[n-i], то если b[n-i+1]< a[i], то справа от a больше, чем n элементов. В этих случаях мы будем выбирать другое a. Если же b[n-i+1]> a[i], то a- наша медиана. Далее построим следующий алгоритм бинарного поиска по a. Выберем середину массива a и применим предыдущие рассуждения, если слева от a[i] больше, чем n элементов, то выберем медиану массива, состоящего из элементов, меньших a[i]. Если же справа от a[i] больше, чем n элементов, то выберем медиану из элементов, больших чем a[i]. Мы считаем, что по определению при четных длинах это $\frac{L}{2}-1$, а все элементы массива a не могут быть меньше этой медианы(так как элементов n) и не могут быть больше(так как a[n]<b[n]). Будем проделывать действия пока у нас не останется массива длины 1 или пока мы не найдем медиану. Если же мы не нашли медиану, то это означает, что медиана в массиве b, тогда проделаем те же действия для массива b.

Сложность алгоритма: Каждое действие состоит из 2 сравнений и выбора 2 элементов- это O(1). Эти действия мы повторяем $\log n$ раз. Следовательно сложность $O(\log n)$.

Корректность алгоритма: В силу того, что мы рассматриваем 2 массива, то мы не нарушая общности можем сказать, что медиана содержится в массиве a. Допустим, что мы пропустили медиану, это означает, что в какой-то момент мы выбрали массив, в котором она не содержжалась, но у нас есть два случая: либо a[i]-медиана(тогда все ок), либо справа от a[i] находится больше, чем n элементов(тогда медиана точно не лежит слева от a, так как там уже меньше, чем n-1 элемент), аналогично для случая, когда слева от a[i] находится больше, чем n элементов. Противоречие. Следовательно наш алгоритм работает корректно.

6.

Посмотрим на a_0. Запишем все его делители в массив k- это будут кандидаты на решение. Почему нет других возможных решений? Потому что, если x_0 решение, то многочлен расскладывается на произведение многочленов, один из которых$x - x_0$. Если мы посмотрим на свободный член после перемножения двух многочленов, то увидим, что он равен $x_0 * \alpha = a_0$. Теперь продставим каждый элемент из массива к в нашу функцию, если она равна 0, то мы нашли решение. В противном случае элемент не является решением.

Сложность алгоритма: Для начала разложим наше число на простые делители, для этого преберем все простые числа от 2 до $\sqrt[2]{a_0}$. Это займет $O(\sqrt[2]{n})$. Далее будем подставлять в функцию все наши числа. Это займет $O(2n^2 + 2(n-1)^2 + ... + 4) = O(n^3)$. Следовательно общая сложность $O(n^4)$

Корректность алгоритма: Допустим, что мы не разобрали какой-то случай. Если x_0 решение, то многочлен расскладывается на произведение многочленов, один из которых$x - x_0$. Если мы посмотрим на свободный член после перемножения двух многочленов, то увидим, что он равен $x_0 * \alpha = a_0$. Противоречие.

7.

1)Устроим турнир между монетами по швейцарской системе. Тогда самую тяжелую монету мы найдем через

$$\sum_{i}^{\log_2 n} \frac{n}{2^i} = n$$

После этого посмотрим на проигравших на первом этапе. Понятно, что наименьшая монета среди них. Теперь устроим такой же турнир, только победителем теперь уже будет более легкая монета. Это займет

$$\sum_{i}^{\log_2 \frac{n}{2}} \frac{n}{2^{i+1}} = \frac{n}{2}$$

Следовательно наш алгоритм будет работать за $\frac{3n}{2} + c$.

2). Обозначим за MC(max cond)- те монеты, которые сыграли не меньше одного раза и все выйграли, за N(neutral)-монеты котрые не сыграли ни одного раза. за L(lose)-те, которые сыграли не меньше одного раза и все проиграли. За NON(non-priority)-оставшиеся монеты

Разберем взможные случаи

1) MC vs MC
2)MC vs N
3)MC vs L
4)N vs L
5)N vs N
6)L vs L
7)MC vs NON
8)N vs NON
9)L vs NON

Правила игры будут следующими: MC всегда выигрывает N,L: N всегда выигрывает L. NON проигрывает MC и N, но выигрывает L. Тогда будет

следующая таблица изменения кол-ва монет в кучках.

NUM	MC	N	L
1	-1	0	0
2	0	-1	1
3	0	0	0
4	1	-1	0
5	1	-2	1
6	0	0	-1
7	0	0	0
8	1	-1	0
9	0	0	0

Изначально у нас вектор (0, n, 0). В конце (1,0,1). Поймем, что N мы можем изменять либо на 1 либо на 2, причем эти монеты перейдут либо в MC, либо в L. Из каждого из них мы умеем убирать только 1 монету за раз. Следовательно мы перекачаем не меньше, чем за $\frac{n}{2}$ действий монеты из N в кучки MC, L(иначе мы не умеем), а из MC и L мы их уберем только за n действий, так как мы умеем убирать только 1 монету за раз. Следовательно не сушествует алгоритма, делающего меньше, чем $\frac{3n}{2} + c$ действий.