AI Masters Algorithms **Name:** *Денис Грачев*

Homework 7

30 октября 2022 г.

Task 1

Насколько я понял в данном задании ключи = значения. Опишем функцию которая позволяет обойти двоичное дерево таким образом, чтобы вывести все ключи в порядке возрастания за $O(n)$.

def *Printer(node: int)***:**

```
    if node.left ≠ Nil then
|       Printer(node.left)
    end
    print(node.value)
    if node.right ≠ Nil then
|       Printer(node.right)
    end
```

И запустим эту функцию из корня.

Доказательство корректности.

Докажем по индукции. Для дерева высотой 1 корректность очевидна.

Пусть функция корректно работает для дерева высотой n, докажем что она корректно работает для дерева высотой $n+1$. Сначала мы распечатает все значения по возрастанию находящиеся в левом поддереве корня, из свойств двоичного дерева поиска все эти значения будут меньше корня.

Затем мы распечатаем сам корень дерева.

И после этого распечатаем все значения правого поддерева корня, из свойств двоичного дерева поиска все эти значения будут больше корня.

Таким образом мы выведем все значения в порядке возрастания.

Так как на каждом вызове мы печатаем одну вершину, и все вершины мы печатаем ровно 1 раз, то сложность выполнения этой функции $O(n)$.

Task 2

Семейство H будет универсальным если $\forall x \neq y : \mathbb{P}(\hat{f}(x) = \hat{f}(y) \mid \hat{f} \in H,\ x,\ y) = \frac{1}{m} = \frac{1}{3}$.

$$\begin{aligned}\mathbb{P}(\hat{f}(x) = \hat{f}(y) \mid \hat{f} \in H,\ x,\ y) &= \mathbb{P}(\hat{f} = f)\mathbb{P}(f(x) = f(y)) \\ &+ \mathbb{P}(\hat{f} = g)\mathbb{P}(g(x) = g(y)) \\ &+ \mathbb{P}(\hat{f} = h)\mathbb{P}(h(x) = h(y)) \\ &= \mathbb{P}(\hat{f} = f)\mathbb{1}(f(x) = f(y)) \\ &+ \mathbb{P}(\hat{f} = g)\mathbb{1}(g(x) = g(y)) \\ &+ \mathbb{P}(\hat{f} = h)\mathbb{1}(h(x) = h(y))\end{aligned}$$

Возьмем $x = 0, y = 3$, тогда

$$\begin{aligned}\mathbb{P}(\hat{f}(x) = \hat{f}(y) \mid \hat{f} \in H,\ x,\ y) &= \mathbb{P}(\hat{f} = f)\mathbb{P}(f(x) = f(y)) \\ &+ \mathbb{P}(\hat{f} = g)\mathbb{P}(g(x) = g(y)) \\ &+ \mathbb{P}(\hat{f} = h)\mathbb{P}(h(x) = h(y)) \\ &= \frac{1}{3}\mathbb{1}(f(0) = f(3)) \\ &+ \frac{1}{3}\mathbb{1}(g(0) = g(3)) \\ &+ \frac{1}{3}\mathbb{1}(h(0) = h(3)) \\ &= \frac{1}{3} * 0 + \frac{1}{3} * 1 + \frac{1}{3} * 1 \neq \frac{1}{3}\end{aligned}$$

Следовательно семейство H не является универсальным.

Task 3

Уточнение: когда $b \geq \lceil \frac{n}{2} \rceil$ то устанавливаем значение $b = 0$ и переносим всю осмысленную часть массива в начало.

Доказательство амортизационной сложности.

Заведем для удобства еще один указатель c - конец нашей очереди. Будем считать, что добавление стоит условные 4 монеты.

Добавление в массив

Пусть 1 монету стоит добавление в текущий массив если в нем есть место.

Еще 1 монета нужна для копирования самого элемента в новый массив вдвое большего размера, если место закончилось.

Еще 1 монета нужна для копирования элемента под номером $i - \frac{n}{2}$ в массив двое большено размера, если место закончилось

Еще 1 монету отложим для переноса. Корректность.

Докажем по индукции.

База индукции очевидна.

Пусть в массива размера n закончилось место, тогда у правой половины массива есть 3 свободные монеты, то есть $3\frac{n}{2}$ монет. Создадим новый массив вдвое большего размера и скопируем туда текущий это будет стоить n монет, следовательно у монеты в правой заполненной части останется по свободной монете.

При добавлении новых элементов у каждого из них будет 3 свободные монеты и когда место снова закончится, у правой половины снова будет 3 свободные монеты которые смогут оплатить переезд.

Следовательно добавление при амортизационном анализе работает за $O(1)$.

Удаление

Тогда 1 монету стоит само добавление, а на 1 монету мы *оплатим* перенос элемента в первую половину, если такой состоится. Любой элемент мы переносим в начало только 1 раз (так как n только растет), если уж элемент оказался в какой то момент в левой половине, то этот индекс уже не сможет попасть в правую половину массива. Следовательно удаление при амортизационном анализе работает за $O(1)$.

И мы использовали не более чем $2n$ памяти.

Task 4

Заведем HashMap ключом которого будет ключ, значением - приоритет.

Тогда добавление нового ключа с приоритетом будет стоить $O(1)$, установка приоритета так же $O(1)$, так как добавление и изменение элементов в HashMap осуществляются за $O(1)$.

Для того чтобы совершить extractmax, необходимо обойти все пары ключ, значение запоминая максимальное значение и соответсвующий ему ключ. Тогда мы найдем ключ с максимальным значением за $O(n)$. После этого удаление этого ключа стоит $O(1)$, итоговая сложность extractmax $O(n)$.

Task 5

Заведем HashMap в котором ключом будет ключ, значением - количество повторений этого ключа.

Соотвественно нам нужно пройтись по массиву, если ключ уже присутсвует в HashMap то увеличить значение на 1, иначе создать со значением 1. Это занимает $O(n)$, так как элементов n, вставка и получение значения по ключу $O(1)$.

После этого создадим новый массив, в котором хранятся пары ключ, значение в произ-

вольном порядке. Это так же занимает $O(n)$ так как нам необходимо обойти HashMap размер которого не больше n.

После этого мы можем отсортировать данных массив с помощью CountingSort в убывающем порядке, сравнивая значения в нашем массиве (то есть количество повторений). Так как количество повторений минимум 1, максимум n, то сложность CountingSort $O(n)$.

Итого мы получили все ключи с количеством их повторений, в порядке убывания количества повторений за $O(n)$.

Task 6

Обозначим данный массив за a. Заведем 3 счетчика left, right, counter и HashSet currentnumbers, будем поддерживать следующий инвариант.

- left < right.
- currentnumbers = HashSet(a[left: right])
- counter - количество различных подмассивов, в которых все элементы различны, в промежутке $a[: \textit{right}]$.
- все элементы $a[\textit{left} : \textit{right}]$ - различны.
- $\nexists$ left': left' < left, все элементы $a[\textit{left'} : \textit{right}]$ - различны

Изначально left = 1, right = 2, counter = 1, currentnumbers = HashSet([a[1]]).

Если элемент right не встречался в currentnumbers, то увеличим right на 1 и доабвим этот элемент к текущему множеству.

Иначе уберем крайний левый элемент из текущих.

Из построения легко видеть, что все перечисленные свойства выполняются (кроме counter).

Заметим, что при увеличении right на 1, количество непрерывных подмассивов с различными числами, увеличивается на right - left = len(currentnumbers). Так как добавляются

подмассивы a[right: right + 1], a[right - 1: right + 1], ... , a[left: right + 1].

Следовательно для поддержания инварианта counter += len(currentnumbers). Таким образом мы посчитали все подмассивы состоящие из различных чисел, которые в какой то момент находились внутри a[left, right]. Но это и есть все возможные подмассивы с различными числами, так как если предположить противное, то в момент когда right находилась в правом крае подобного массива, а left по предположению была правее левого края подобного массива, то это противоречит последнему инварианту.

Так как на каждой итерации увеличивается left или right, а каждая итерация стоит $O(1)$, то итоговая сложность $O(n)$.

Псевдокод приведен ниже.

```
left = 1
right = 2
counter = 1
currentnumbers = HashSet([a[1]])
while left ≠ n + 1 or right ≠ n + 1 do
    if right ≠ n + 1 and a[right] not in currentnumbers then
        right += 1
        currentnumbers.add(a[right])
        counter += len(currentnumbers)
    else
        left += 1
        currentnumbers.remove(a[left - 1])
    end
end
```

Task 7

1

Будем осуществлять поиск *лесенкой* начиная с нижнего правого угла.

Обозначим текущие координаты i, j. Пусть левый сосед больше искомого элемента, тогда в подтаблице $\hat{i}, \hat{j} : \hat{i} \geq i - 1, \hat{j} \geq j$ искомых элементов нет, так как она упорядочена.

Аналогично если верхний сосед больше искомого элемента.

Таким образом каждый раз мы можем уменьшать одну из координат на 1 и либо найдем искомый элемент, либо если дошли до края таблицы или верхний и левый соседи меньше искомого, значит искомого элемента нет.

2

Аналогично предыдущему пункту.

Вставим в правый нижний угол, там находится ∞. Посчитаем количество инверсий во всех столбцах и строках. Далее будем сравнивать с левым и верхним соседом. Заметим, что если поменять местами текущий элемент и наибольшего из соседей, то количество инверсий уменьшится (если текущий элемент его меньше), если же текущий элемент больше этих соседей то кол-во инверсий равно 0.

Доказательство.

Пусть текущий элемент $a_{i,j}$ и мы поменяли его с $a_{i,j-1}$. Тогда $a_{i,j-1} < a_{i+1,j-1} < a_{i+1,j}$, $a_{i,j-1} > a_{i-1,j-1} > a_{i-1,j}$. Следовательно в j-ом столбце инверсий не будет. А в i-ом столбце количество инверсий уменьшилось на 1.

Аналогично, если $a_{i-1,j} > a_{i,j-1}$ и $a_{i-1,j} > a_{i,j}$.

Отсюдо легко видеть, что если оба соседа меньше $a_{i,j}$ то инверсий нет. Так как одна из координат каждый раз уменьшается на 1, то сложность $O(m+n)$.

3

Аналогично предыдущему пункты но наоборот.

Меняем нужное значение на ∞. Сравниваем с правым и нижним соседом. Меняем местами ∞ и наименьшего из соседей. Тогда порядок в предыдущих столбцах и строках не нарушится и каждый раз одна из координат бесконечностей увеличивается на 1. Значит в какой то момент она окажется в нижнем правом углу а все строки и столбцы останутся упорядоченными. Сложность алгоритма так же будет $O(m+n)$