6.1. Доказать, что операции Insert и Extract_Min в двоичной куче работают (амортизированно) за O(logN) и O(1) соответственно.

Смотри картинку.

6.2. Двоичные счетчики и операции Inc, Sum и Double – все за O(1).

Очевидно, что удвоение счетчика надо делать двоичным сдвигом влево, но так как в условии сказано, что стоимость сложения равна числу операций сложения в столбик, а стоимость удвоения равна единице, то просто так сдвиг не сделаешь – ему требуется n операций (перенос) + одно очищение бита.

Можно хранить число в «обратном» порядке: нулевой элемент битового массива есть старший бит числа, n-ый элемент – младший бит, а также будем отдельно хранить индекс младшего бита. Тогда **удвоение** выглядит просто: сдвигаем указатель на младший бит вправо и ставим туда нолик.

Обнуление (как часть суммы) тоже просто: обнуляем индекс млашего бита и ставим туда ноль.

Инкремент делается почти также, как в обычном счетчике: начинаем с младшего бита и идем влево: пока единички, ставим туда ноль, как только встретилась первый нолик – пишем туда единичку. Если нолик не встретился вообще а мы уже дошли до конца числа (т.е. до начала массива), пишем в старший бит единицу и удваиваем число: т.е. переносим указатель младшего бита вправо и пишем туда ноль.

Писать ноль нужно, потому что иногда там могут быть мусорные значения: например было число **11111** и мы его занулили: на самом деле лишь перенесли индекс младшего бита в первый элемент массива и занулили то, куда он указывает – получилось число **0**1111: справа остался мусор.

Сумма делается тоже несложно: почти как инкремент, только с запоминанием “переполнения” и соответствующим подсчетом. Будет одна небольшая проблема: если в первом счетчике лежит маленькое число, а во втором (который должен быть обнулен) – большое. Тогда можно перед подсчетом суммы найти максимум их длин, переместить указатель младшего бита первого счетчика направо до позиции максимума и начать заполнять число оттуда (сохраняя при этом копию “старого” указателя, это нужно для суммирования). В таком случае, придется сделать еще максимум одно смещение индекса младшего бита.

Покажем, что дав достаточно большой (постоянный) кредит каждой установке бита в 1, можно будет «расплатиться» за все другие операции и не уйти в минус. Таким образом, амортизационная стоимость каждой операции будет O(1).

Известно, что если стоимость установки бита в 1 стоит 1 монету, то плата за операцию Inc должна стоить 2 (одна пойдет за установку хотя бы одного бита в 1, вторая – как кредит за будущее обнуление). Также будем платить по одной монете за обновление индекса младшего бита, и, если он изменился, то еще одну монету (как кредит для будущего обнуления, но это обнуление уже вызовет функция reset (часть sum) или double, а не inc). И установим стоимость Reset в 1 (для сброса индекса младшего бита).

Удвоение будет стоить 1 монету: перенос младшего бита. За обнуление бита платить не нужно: либо он уже там был, либо там стояла единица, а, значит, был когда-то инкремент, а значит в ней есть кредит на сброс для операции «reset» - но так как Reset не делает с числом в нашем алгоритме ничего – эта одна монета пойдет на обнуление нового «младшего» бита при удвоении.

Сумма работает время, пропорциональное числу единиц в числах (точнее, максимуму единиц) – она состоит из переноса младшего бита (за это нужно еще доплатить, к уже существующим монетам), суммы (0+0 ничего не меняет, 1+0 тоже не меняет, 0 + 1 стоит 1 смену – за это заплатит «вторая» единица, 1+1 стоит тоже одну смену, за это заплатит, пусть «первая» единица), возможного переполнения (а оно бывает только в 1+1 – нужно и за это доплатить) и возможного сдвига младшего бита. Итог: нужно доплатить еще 4

монеты для каждой установки бита в единицу, чтобы покрыть все расходы на сумму и убедиться, что амортизационная стоимость равна 0.

Итак, будем платить по 8 монет для инкремента (так как инкремент устанавливает не больше одного бита), по одной монете для сброса счетчика, по одной за удвоение – стоимость каждой операции O(1).

6.3. Найти (за линию) в массиве различных чисел (возможно, неотсортированном) k ближайших к медиане элементов по двум метрикам: по позиции в отсортированном массиве и по абсолютному значению.

a) По позиции в отсортированном массиве

Предположим, для простоты, что n нечетное, а k четное. Тогда если бы массив был отсортирован, то его медиана была бы на месте с индексом n/2, а ближайшие k элементов к медиане находились бы по индексам от (n-k)/2 до (n+k)/2.

Алгоритм становится прост: запустим линейный поиск трижды и найдем *левый* элемент ((n-k)/2-я порядковая статистика), центральный (медиану, n/2-статистику) и *правый* элемент((n+k)/2-я порядковая статистика). Затем пробежимся по массиву и найдем все элементы большие *левого*, но меньшие *правого*, и одновременно не равные медиане: это и будет ответом. Линейность очевидна: трижды линейный поиск + один просмотр массива.

b) По значению.

Найдем за линию медиану массива. Дальше заведем новый массив B, каждый элемент которого будет равен модулю разности между (каждым) элементом из A и его медианой: B[i] = abs(A[i] – median). Очевидно, что k ближайших к медиане элементов по модулю есть k наименьших элементов массива B. Однако массив B неотсортирован: не беда, ищем в B k-ю порядковую статистику: левая часть разбиения (плюс статистика) будет соответствовать тем элементам, которые ближе всего к медиане в исходном массиве. Их можно искать проходом по массиву и сопоставляя расстояние до медианы, а можно хранить в B индексы элементов, для которых считалось расстояние (в нашем случае это i) – на асимптотику это не влияет: два линейных поиска порядковых статистик + просмотр массива дают, очевидно, линейную сложность.

6.4. K-квантили.

Будем искать способом, очень похожим на алгоритм быстрой сортировки: выберем из квантилей ближайший к середине и запустим поиск соответствующей порядковой статистики. Массив поделится на две почти равные части, причем их рассматривать можно независимо (так происходит поиск статистики). Поэтому, запустим алгоритм рекурсивно на каждой из подчастей с параметром k/2. Время работы можно оценить как сумму ряда n+2*n/2 + 4*n/4+…+k*n/k = O(n*logK) или по рекуррентной формуле: T(n,k) = 2*T(n/2, k/2) + O(n)

T(n,k) = O(n*logK)

6.5*. Разбиение неотсортированных постов на Хабрахабре.

На новостном сервере Тыбротыбр произошла неудача. Все посты перемешались и теперь находятся в неотсортированном порядке. У каждого поста осталась запись, когда он был создан, и известно, что все посты были созданы в разное время.

Пользователям Тыбратыбра (тыбровчанам) посты демонстрируют постранично. На каждой стра- нице находятся k постов. Первая страница должна содержать самые свежие, следующая – чуть старее и так далее. На каждой странице посты должны быть отсортированы. **Пусть тыбровчанен хочет посмотреть *<u>все</u>* посты Табратыбра за раз**. Нужно придумать такой алгоритм, что каждая новая страница генерировалась за O(n + k log k), а суммарное время на создание всех страниц было O(n log n).

Примечание: Попробуйте сначала достигнуть каждой оценки по отдельности.

a) Оценка на каждую страницу $O(N + k \log k)$

Найдем в массиве k-ю порядковую статистику. Затем пробежимся по всему массиву и найдем все посты за временем, не превышающим время создания k найденной статистики

Отсортируем полученный массив длины k. Время генерации страницы есть $O(N) + O(N) + O(k \log k) = O(N + k \log k)$.

Вторые и последующие генерации страниц будут работать это же время: найдем k-ю статистику в массиве размером N-k, найдем все элементы не больше ее, отсортируем и так далее.

Суммарное время будет равно суммарному времени сортировок $O(N \log k)$ плюс суммарному времени линейных просмотров: но так как получится арифметическая прогрессия, ее сумма даст квадрат, даже без точных подсчетов, что, конечно, не равно $O(N \log N)$. *Мы достигли первой оценки, но не достигли второй.*

b) Оценка на суммарное время создания всех страниц $O(N \log N)$

Перед отправкой первой страницы отсортируем все страницы по убыванию времен создания постов. Затем отправим первые k постов. Запрос второй страницы вернет вторые k постов и т.д. Итог: время создания первой страницы $O(k + N \log N) = O(N \log N)$, а всех остальные страниц - $O(k)$. Суммарное же время создания всех страниц есть $O(N + N \log N) = O(N \log N)$. *Мы достигли второй оценки, но не достигли первой.*

Я правильно понимаю, что условия примечания полностью выполнены и оценены правильно? И теперь цель придумать такой алгоритм, который одновременно достигает **обеих** оценок?