4.1 Про Анку и массив пуль длины N, где каждая пуля отскочила не более чем на K от правильной позиции. Надо отсортировать массив.

a) Алгоритм, решающий задачу за O(NK).

Можно решать совсем в лоб: возьмем первые K+1 элемент массива A[0..K+1]. В них, по условию, находится минимальный элемент исходного массива. Найдем его за O(K). Поставим на место. Сократили задачу до размеров T(N-1). Сдвинемся на один элемент вправо и возьмем отрезок A[1..K+2], проделаем с ним то же самое. В конце придется брать отрезки длиной меньше чем k, потому что закончится массив – продолжаем алгоритм пока длины рассматриваемых отрезков больше 1. Итоговое время – (n-1)*O(K) = **O(NK)**.

b) Алгоритм, решающий задачу за O(N+I), где I – число инверсий

Эту задачу решает простой алгоритм сортировки вставками. Идея алгоритма: на каждом шаге выбираем элемента key (с индексом key), называемый ключевым и поддерживаем следующий инвариант цикла: после каждого шага все элементы A[0..key] оказываются отсортированными. Что происходит на каждом шаге – мы смотрим какие элементы слева от ключевого больше его и двигаем их направо. Как только мы встречаем элемент не больше ключевого – останавливаемся и ставим наш ключевой на его место.

Нужно заметить, что та часть алгоритма (на самом деле это будет внутренний цикл while), где мы выбираем куда же поставить очередной ключевой элемент, будет работать пропорционально количеству инверсий слева от ключевого элемента – ведь что значит мы ищем место для элемента? Это значит, что он образует инверсию со вновь рассматриваемым элементом слева от него. Как только мы поставим ключевой элемент на место, число инверсий сократится на число, равное числу сдвигов во внутреннем цикле. Всего элементов – N, мы гарантированно тратим на каждый элемент по одному сравнению (строго говоря не на каждый – мы не сравниваем ни с чем самый первый элемент, значит гарантированно делаем N-1 сравнение), плюс число инверсий – то число перестановок, которое мы совершаем. Итоговое время работы алгоритма – **O(N+I)**, где I – число инверсий в массиве.

c) Алгоритм, решающий задачу за O(N*logK)

Эту задачу решает куча.

Построим MinHeap от первых K+1 элементов. Мы знаем, что это делается за линейное от размеров время, т.е. за O(K). Найдем в этой куче минимум (это вообще за O(1) делается – минимум лежит на вершине кучи), сделаем ExtractMin, положим элемент на первое место и добавим в кучу еще один (K+2) элемент. Снова найдем минимум и т.д. Корректность очевидна – мы постоянно находим минимальный элемент в «окне» шириной K+1 – а он там обязательно есть и равен элементу, лежащему в исходном отсортированном массиве на месте i, где i – начало нашего «окна». Время работы – O(k) на построение, O(logK) на удаление минимума из кучи и столько же на добавление нового элемента. Всего мы добавим (N-K) элементов, значит общее время работы есть O(k)+((N-K)*logK) = O(K+NlogK – KlogK) = **O(NlogK)**

d) Доказательство асимптотической оптимальности третьего алгоритма.

Применим тот же метод, что и на лекции: построим дерево сравнений. Отличие будет только в том, что число листьев в дереве будет не n!, а меньше, ведь нам позволительны не все перестановки, а только те, в которых каждый элемент уехал от своего места на расстояние не больше K. Я попытался найти точную оценку, но дальше чисел Стирлинга первого рода не ушел. Поэтому грубая оценка: разобьем массив на группы по K элементов и позволим в каждой группе любые перестановки: очевидно, что получившиеся перестановки коррекны – никакой элемент не сможет уехать более чем на K позиций от своего места. (вообще говоря можно было усилить оценку и взять окна шириной даже K+1). Но с другой стороны понятно, что мы много перестановок потеряли: мы не учил те, где элемент переходит из одной группы в другую, но не более чем на K. Поэтому это явно

нижняя оценка. Посчитаем число таких перестановок: $K!^{\left\lfloor \frac{N}{K} \right\rfloor}$. А число листьев в дереве не меньше числа допустимых перестановок. Аналогично рассуждениям в лекции устанавливаем, что время работы алгоритма есть высота дерева сравнений, то есть логарифм нижней оценки числа листьев. Отсюда получаем: $T(n) = \Omega\left(\log K!^{\left\lfloor \frac{N}{K} \right\rfloor}\right) = \Omega(\left\lfloor \frac{N}{K} \right\rfloor \log K!) = \Omega(N \log K)$.

Получилось, что нижняя оценка времени работы совпадает с верхней, поэтому представленный в третьем пункте алгоритм асимптотически оптимален.

С другой стороны это ожидаемо: чтобы поставить элемент на место нам надо не logN бит информации, а logK, ибо по условию элемент находится не дальше чем за K мест от своего верного положения.

4.2 Дано 2n-1 коробок с черными и белыми шарами. В i-ой коробке находится wi белых и bi черных шаров. Требуется выбрать n коробок, чтобы суммарное число белых и черных шаров в выбранных коробках было не меньше половины.

Отсортируем коробки по убыванию количества белых шаров в них, причем нужно обратить особое внимание на случай, когда количество белых шаров в сравниваемых коробках равны – тогда бОльший приоритет имеет коробка, в которой больше черных.

Сделаем копию массива и отсортируем аналогично, только теперь по убыванию количества черных шаров, причем аналогичное замечание распространяется и на эту сортировку. Далее будем действовать жадно: выберем самую большую белую коробку и, из другого массива, самую большую черную коробку. Может так оказаться, что это одна и та же коробка – ее и выбираем. Таким образом мы уменьшили задачу, взяв самые большие коробки, которые только возможно, а, значит, лучшего варианта нет.

Затем на оставшемся массиве действуем также, пока n не станет равным 1 или 0. Если ноль – задача решена, если 1 – то нужно пробежаться по массиву еще не взятых ящиков и взять первый, который решает задачу. (предварительно можно за линию посчитать сумму белых и черных шаров).

Выходит, что мы дважды выбираем n самых больших чисел из массива длины 2n-1. Рассмотрим наихудший случай: когда все числа равны. Но тогда очевидно, что любые n чисел в сумме дадут не меньше половины общей суммы. А если случай будет не худшим (то есть вместо равных числе будут неравные) – то задача даже упростится, потому что мы берем наибольшие из возможных. То есть хуже ситуация уже быть не может.

Сложность же очевидна – две сортировки и два линейных просмотра массива в сумме дают O(NlogN)

4.3* Суть вкратце – дано n, построить worst-case для HeapSort массива из n элементов. Сложностью будет число свопов родителя и максимального ребенка.

Я эту задачу уже решал – она была на каком-то старом NEERC – не старше 2006 года.

Предположим, у нас уже построен наихудший случай для массива длины n. Покажем, как построить для случая n+1. Запихнем новый элемент, который как раз равен n+1 на предпоследнее место в массиве. Теперь поменяем этот максимальный элемент с его родителем, затем опять с родителем и так далее – то есть пропихнем его на самую верхушку пирамиды. Очевидно, что свойство кучи не нарушится. Действительно: в куче изменится лишь положение последнего элемента (который по построению у нас всегда будет 1) – который ничего не меняет и сдвинется вниз вся «родительская» ветка предпоследнего элемента + этот самый предпоследний элемент станет первым. Ясно, что сдвиг ветки вниз тоже не меняет свойства кучи – так как мы на место каждого элемента ставим его родителя – а родитель не меньше элемента.

Утверждается, что это и будет ответом. Проверим это: убедимся, что в результате применения одного шага алгоритма мы вернемся к worst-case для массива размером n.

Применим алгоритм сортировки к нашей новой куче. Меняем максимум (n+1) и минимум (он равен 1) местами, вычеркиваем последний элемент (n+1). Далее заметим, что эта наша единичка пойдет именно по той ветке, которую мы только что двигали вниз – причина в том, что начало этой ветки есть **максимум** предыдущего шага. А второй элемент

введения нового элемента. Получили worst-case для n. А движений (свопов, т.е. степень сложности кучи) мы сделали максимум из возможного числа – длину наибольшей ветки =logN.

Теперь почему это все сработает за O(NlogN). Просто запишем рекуррентное соотношение: T(N) = logN + T(N-1). (так как вставка в дерево занимает O(logN) времени). Решая это несложное соотноешние получим, что T(N) = logN + log (N-1) + … + log 1 = log (N!) = O(NlogN).

Ответы для первых n будут такими:

N=1: 1
N=2: 2 1
N=3: 3 2 1
N=4: 4 2 3 1
N=5: 5 4 3 2 1
N=6: 6 5 3 2 4 1
N=7: 7 5 6 2 4 3 1

4.4 Разработать PersistentList.

Заведем односвязный список и будем делать копию всего списка, стоящего перед указателем на текущий элемент (который будем хранить отдельно) при каждом запросе на добавление или изменение. Также будем хранить указатель на начало.

Для персистентности будем хранить указатели на начало и на текущий элемент в массиве-истории, индексами которого будут моменты времени.

Begin(S) за O(1) – очевидно.

Next(S) за O(1) – очевидно, просто двигаемся по списку вперед.

At(S) за O(1) – очевидно.

Restore(S) за O(1) – очевидно, просто достаем характеристику списка (указатель на начало и на текущий элемент) из массива-истории.

Insert за O(d) – двигаемся по списку с самого начала и клонируем его до тех пор, пока не встанем на «текущий элемент». Потом создаем новый элемент и устанавливаем указатель с нового на «текущий».

Remove за O(d) – почти то же самое, что в предыдущем пункте, за исключением создания нового элемента – там мы запоминаем еще и “пред-предыдущий” элемент и как только встретим “текущий” – ставим указатель с “пред-предыдущего” на текущий, не копируя “предыдущий” – таким образом он оказывается удаленным.

4.4 Разработать Persistent2List. (Все операции, кроме At и Restore, которые за O(1), делаются за O(log|S|) + есть операция Prev)

Пока нет – видимо тут будет дерево версий, раз почти все операции за логарифм…