1. Для сортировки данного массива заведем два указателя (бегунка).

Первый сначала будет указывать на первый элемент, второй – на последний.

Затем будем последовательно двигать бегунки: если первый указывает на 0 элемент, то двигаем дальше вперед, если на 1 – останавливаем и движение начинает второй бегунок.

Если второй бегунок указывает на 1, то двигаем его назад по массиву (влево), иначе (если на 0), то останавливаем и меняем местами элементы, на которые указывают первый и второй бегунок.

Остановкой алгоритма является встреча двух бегунков.

Корректность:

Алгоритм проходит все элементы массива, правый бегунок при обнаружении 1 меняет его с левым, указывающим на первый 0, считая от левого конца. Таким образом, на выходе массив будет упорядоченным. Условие остановки всегда достижимо, даже в вырожденном случае: когда массив состоит из одинаковых элементов, тогда бегунки встретятся на краях.

Сложность:

Сложность данного алгоритма линейная: $\mathbf{O(n)}$. В ходе него будет не более, чем $n-1$ сдвигов и $\frac{n}{2}$ обменов элементов.

2.

Мысленно представим, что координаты скобок упорядочены, в алгоритме это использоваться не будет, просто для наглядности.

Для нахождения количества точек, необходимых по условию задачи, для начала найдем границы тех отрезков, где они заключены.

Картинка вложенных отрезков будет симметричная, рассмотрим ситуацию слева (где скобки открывающиеся). Точки, лежащие между первой и второй скобками покрыты одним отрезком, второй и третьей – двумя и т.д. Точки, удовлетворяющие условию находятся между $\frac{2n}{3}$ и $\frac{2n}{3}+1$. Аналогичная ситуация справа: только считать необходимо от левой скобки, так что от первой до второй скобки точки покрыты $n-1$ скобками, далее – $n-2$ и т.д. Точки, необходимые по условию, находятся между $n-\frac{2n}{3}=\frac{n}{3}$ и $\frac{n}{3}+1$ скобками.

Количество целых точек между двумя скобками можно найти следующим образом: $n=\lfloor b\rfloor-\lceil a\rceil+1$, где a и b - начало и конец соответствующего отрезка.

Таким образом, алгоритм выглядит следующим образом:

1. Формируем массив координат левых границ отрезков: $l[n]$ и правых: $r[n]$
2. Находим $\frac{2n}{3}$ и $\frac{2n}{3}+1$-статистики в массиве $l[n]$ - массив левых концов.
Обозначим $a_1=l[\frac{2n}{3}], b_1=l[\frac{2n}{3}+1]$, тогда число подходящих точек: $n_1=\lfloor b_1\rfloor-\lceil a_1\rceil+1$.
3. Находим $\frac{n}{3}$ и $\frac{n}{3}+1$-статистики в массиве $l[n]$ - массив левых концов.
Обозначим $a_2=l[\frac{n}{3}], b_2=l[\frac{n}{3}+1]$, тогда число подходящих точек: $n_2=\lfloor b_2\rfloor-\lceil a_2\rceil+1$.
4. Общее число подходящих точек $n=n_1+n_2$

Корректность:

Так как отрезки строго вложены друг в друга неоднозначности в поиске k-статистик не возникает. Количество точек считается и слева, и справа (то есть, между открыва-

ющимися и закрывающими скобками), таким образом, в ответ войдут всевозможные точки.

Сложность:

Сложность поиска k-статистик, согласно лекции, $O(n)$. В алгоритме 4 раза ищется k-статистика, так что общая сложность алгоритма $\mathbf{O(n)}$.

3.

1. Рассмотрим худший случай для быстрой сортировки: полностью упорядоченный массив: $1, 2, 3, ..., n$ (в качестве опорного элемента берем последний).

Каждое разделение даёт два подмассива размерами 1 и $n-1$. То есть, будет **n** рекурсивных вызовов.

$T(n) = T(n-1) + ... + T(1)$

2. Для того чтобы не было переполнения стека, можно сделать следующее: устранить одну ветвь рекурсии, то есть, рекурсивный вызов делать только для меньшего подмассива, а больший обрабатывать в цикле в пределах этого же вызова процедуры (хвостовая рекурсия).

Таким образом, глубина рекурсии гарантированно не превысит $\log_2 n$, а в худшем случае и вовсе будет не более 2 — вся обработка пройдёт в цикле первого уровня рекурсии.

4.

Докажем для начала, что s, на котором достигается минимум – это медиана массива a. Заметим, что число элементов массива $2n+1$ заведомо нечетное, так что медиана – это строго $n+1$-статистика массива.

Доказательство.

Упорядочим массив (для алгоритма этого не нужно, только для доказательства факта выше): $a_1 \leq a_2 \leq ... \leq a_{2n+1}$

Для любых трех чисел x_1, x_2, x выполняется неравенство $|x_2 - x_1| \leq |x - x_1| + |x - x_2|$ в силу неравенства треугольника, равенство достигается только в случае: $x_1 \leq x \leq x_2$. Следовательно, для любого числа s:

$|s - a_1| + |s - a_2| + ... + |s - a_{2n}| + |s - a_{2n+1}| = (|s - a_1| + s - a_{2n+1}) + (|s - a_2| + s - a_{2n}) + (|s - a_n| + s - a_{n+2}) + |s - a_{n+1}| \geq |a_1 - a_{2n+1}| + |a_2 - a_{2n}| + ... + |x_n - x_{n+2}| + |s - a_{n+1}|$

Так как медиана a_{n+1} находится между каждой парой значений a_1 и a_{2n+1}, a_2 и a_{2n}, ..., a_n и a_{n+2}, то при $s = a_{n+1}$ предыдущее неравенство обращается в равенств, иначе будет строгое неравенство. Следовательно данная сумма принимает минимальное значение при s – $n+1$-статистика, то есть, когда s – медиана массива.

Алгоритм состоит в нахождении $n+1$-статистики массива a. Данный алгоритм был предложен на лекции за $O(n)$.

Корректность:

Корректность того, что минимум достигается на s – медиане массива, было доказано выше. Алгоритм нахождения k-статистики и его корректность была доказана на лекции.

Сложность:

Сложность алгоритма $\mathbf{O(n)}$, так как нахождение статистики вычисляется за $O(n)$.

5.

Для начала упорядочим массив a быстрой сортировкой с рандомизированным выбором опорного элемента.

Затем заведем два указателя: первый установим на первый элемент, а второй – на последний.

Будем последовательно складывать значения элементов, расположенные по этим указателям, если сумма их больше заданного x, то двигаем влево (назад) правый указатель, если меньше – то двигаем вправо (вперед) левый указатель, если достигнуто равенство – то заканчиваем программу и оповещаем о том, что такие элементы есть. Условием остановки алгоритма является встреча двух указателей.

Корректность:

В ходе алгоритма мы пройдем в худшем случае обязательно все элементы, так что пропуска быть не может. Если накопленная сумма становится больше, то движение правого указателя назад обусловленно тем, что там элементы заведомо меньше, что и нужно для дальнейшего поиска, аналогичная ситуация с другим указателем, только наоборот. Алгоритм всегда заканчивает работу: в случае успеха при найденных двух элементах. В противном случае при встрече двух указателей, которая обязательно произойдет, ведь указатели двуижутся навстречу друг другу.

Сложность:

Мы делаем быструю сортировку с рандомизированным выбором опорного элемента, это $O(n \log n)$ (было доказано на лекции) и проходим в худшем случае по всем элементам упорядоченного массива: $O(n)$. Таким образом, итоговая сложность: $\mathbf{O}(\mathbf{n} \log \mathbf{n})$.

6.

Пусть a – заданный массив. Нумеровать будем с 1.

Создадим алгоритм с асимптотикой $O(n)$.

Заведем четыре массива: $min\_start[n], min\_end[n], max\_start[n], max\_end[n]$, которые будут последовательно содержать элементы, являющиеся максимальными и минимальными при приходе вперед и назад, ниже будет подробно показано.

Зададим им начальные значения:

$min\_start[1] = a[1]$,

$min\_end[n] = a[n]$,

$max\_start[1] = a[1]$,

$max\_end[n] = a[n]$.

Затем в цикле, начиная со второго элемента, пройдем все элементы массива и последовательно будем присваивать значения этим четырем массивам.

$min\_start[i] = min(a[i], min\_start[i-1])$,

$min\_end[n-i] = min(a[n-i], min\_end[n-i])$,

$max\_start[i] = max(a[i], max\_start[i-1])$,

$max\_end[n-i] = max(a[n-i], max\_end[n-i])$

Далее, заведем два счетчика n_dec – число убывающих подмассивов, n_inc – число возрастающих подмассивов. Присвоим начальные значения: $n\_dec = 1, n\_inc = 1$.

Затем пройдемся вновь по циклу от первого до предпоследнего элемента и посчитаем, сколько получилось таких возрастающих и убывающих подмассивов.

То есть, если максимальная левая граница не больше правой минимальной, то подмассив будет возрастающим, а, если минимальная левая граница будет не меньше максимальной правой – убывающим.

Таким образом, наращиваем n_inc, если $max\_start[i] \leq min\_end[i]$ и наращиваем n_dec, если $min\_start[i] \geq max\_end[i]$.

Результатом, то есть максимальным количеством подмассивов, после сортировки которых массив станет отсортированным, будет $res = max(n\_inc, n\_dec)$, ведь это два независимых разбиения, из которых мы выбираем лучшее.

Продемонстрируем работу алгоритма на примере.

Пусть $a = 6, 7, 4, 5, 1, 3, 8$ (в принципе могут быть повторяющиеся элементы, на работе алгоритма это не скажется, неравенства в условиях нестрогие).

Создаем $min\_start[n], min\_end[n], max\_start[n], max\_end[n]$

min_start

Первое число – индекс, второе – значение.

$1 : 6$ – начальное значение

$2 : 6$ – так как $6 < 7$

$3 : 4$ – так как $4 < 6$

$4 : 4$

$5 : 1$

$6 : 1$

$7 : 1$

Аналогично:

min_end

Идем с конца, поэтому:

$7 : 8$ – начальное значение

$6 : 3$ – так как $3 < 8$

$5 : 1$ – так как $1 < 3$

$4 : 1$

$3 : 1$

$2 : 1$

$1 : 1$

max_start

$1 : 6$

$2 : 7$

$3 : 7$

$4 : 7$

$5 : 7$

$6 : 7$

$7 : 8$

max_end

$7 : 8$

$6 : 8$

$5 : 8$

$4 : 8$

3 : 8
2 : 8
1 : 8

Затем считаем количество $max\_start[i] \leq min\_end[i]$ и наращиваем n_inc, получим: $n\_inc = 2$

Считаем $min\_start[i] \geq max\_end[i]$ и наращиваем n_dec, получим: $n\_dec = 1$

Таким образом, результатом будет: $res = 2$.

Корректность:

Данный алгоритм на первом шаге в цикле запоминает последовательно максимумы и минимумы предполагаемых левых и правых границ массива, а затем пытается разбивать массив на два подмассива, эти два – еще на два и т.д., если соответственно все числа из второй части не меньше, чем все числа из первой, но чтобы все числа не сравнивать, используются заранее подготовленные максимумумы и минимумы. Таким образом, всегда сохраняется условие упорядоченности между границами всех подмассивов, что делает возможным их последующую сортированную конкатенацию в полностью отсортированный массив.

Сложность:

Алгоритм состоит из двух циклов, так что итоговая его сложность: $\mathbf{O(n)}$.

7.

Заметим, что массив F содержит число инверсий $1, 2, 3, ..., n$ в массиве A.

Создадим для начала массив размера $a[n]$ из одних 0. Затем необходимо поставить $1, 2, 3, ..., i, ..., n$ на место в массиве $f_i + 1$, ведь f_i – число инверсий данного элемента $1, 2, 3, ..., i, ..., n$, то есть, количество чисел, которых он больше. Таким образом, шаг за шагом устанавливая значения в массив, на выходе мы получим полностью восстановленный.

Сложность данного алгоритма квадратичная (каждому из n необходимо найти место в массиве, в худшем случае n-ое), чтобы его упростить, необходимо использовать структру данных – дерево отрезков – для нахождения $f_i + 1$-го нуля, то есть, $f_i + 1$-го свободного места в массиве. В дереве обрезков будем хранить количество нулей, встречающихся в отрезках массива. Чтобы найти $f_i + 1$-ую свободную позицию, нужно спускаться (начиная с корня) в левое поддерев, если сумма в нём больше, чем $f_i + 1$, и в правое дерево иначе. Таким образом, сложность понизится до $O(n \log n)$, подробнее будет описано ниже про сложность.

Корректность:

Массив F последовательно содержит число элементов больших, чем $1, 2, 3, ..., i, ..., n$ и стоящих левее элемента i, что соответствует числу инверсий данного элемента. В восстановленном массиве это число i устанавливается строго на такую позицию, что перед ней есть f_i свободных мест для этих чисел, которых оно меньше.

Структура данных – дерево отрезков – позволяет быстро искать позицию, на которую надо поставить элемент i, то есть, ищет $f_i + 1$ свободное место. Для этого производиться спуск по дереву отрезков, начиная с корня, и переходя каждый раз в левого или правого потомка в зависимости от того, в каком из отрезков находится искомый $f_i + 1$-ый ноль.

Действительно, чтобы понять, в какого потомка надо переходить, достаточно посмотреть на значение, записанное в левом потомке: если оно больше либо равно $f_i + 1$, то переходить надо в левого потомка (потому что в его отрезке есть как минимум $f_i + 1$ нулей), а иначе — переходить в правого потомка.

Сложность:

Сложность алгоритма без использования структуры данных – дерева отрезков – $O(n^2)$ (было показано выше). Использование дерева отрезков позволяет находить $f_i + 1$ ноль за $O(\log n)$, ведь в худшем случае мы спустимся по всему дереву, высота которого $O(\log n)$, что позволяет расставить все элементы за $\mathbf{O(n \log n)}$.