1 Обязательная часть

1. **Параллельный минимум и максимум.**

   Разобьем весь массив мысленно на n пар. Сначала сравним элементы внутри этих пар и, если нужно, поменяем так, чтобы первый эл-т в паре (левый) был меньшим, а второй — большим. Это будет n сравнений. Теперь заведем переменные min и max, в которые изначально запишем первый и второй эл-ты первой пары соответственно. А теперь пробежимся по оставшимся $n-1$ парам, сравная наш текущий min с первый эл-том рассматриваемой пары, а max — со вторым. Если окажется, что эл-т в паре меньше min — обновляем min. Аналогично для max. Таким образом для каждой из оставшихся $n-1$ пары мы сделаем по два сравнения. Итог: $n + 2 \cdot (n-1) = 3 \cdot n - 2$, что и требовалось в условии

2. **Поиск статистики.**

   - За $\mathcal{O}(m + k \cdot \log m)$.

     Возьмем по первому элементу каждого из массивов и построим из них кучу с минимумом в вершине. Она строится за $\mathcal{O}(m)$. А теперь будем делать как бы сортировку кучей, но не менять вершину с последним элементом, а подставлять в нее каждый раз следующий эл-т того массива, которому она принадлежит.

     Подробнее: изначально в вершине кучи будет лежать самый маленький эл-т, то есть первая порядковая статистика. Возьмем из массива, которому наша вершина принадлежит, следующий за ней эл-т и положим в вершину кучи. Затем запустим SiftDown. За $\mathcal{O}(\log m)$ операций куча снова придет в нормальное состояние, в вершине ее будет лежать новый минимальный эл-т, то есть вторая порядковая статистика. Нам нужна k-я. То есть через $k-1$ таких операций в вершине кучи окажется k-я порядковая статистика. Время работы алгоритма $\mathcal{O}(m + k \cdot \log m)$, что и требовалось.

   - За $\mathcal{O}(m \cdot \log MAX \cdot \log n)$.

     Максимальный возможный ответ - Max эл-т массивов. Запускаем бинпоиск по ответу. Пусть у предполагается ответ ans. В каждом массиве бинпоиском находим первый эл-т, больший или равный ans и смотрим на его индекс (не строго больший, так как ans может оказаться в нескольких массивах, а нам нужно его первое в объединении вхождение). Если сумма по всем массивам (индекс - 1) будет равна k - ответ найден.

     Время работы: $\mathcal{O}(\log MAX)$ - бинпоиск по ответу. На каждом его этапе в каждом массиве мы запускаем обычный бинпоиск. То есть на каждом этапе происходит $\mathcal{O}(m \cdot \log n)$ операций. Итого k-ю статистику мы найдем за $\mathcal{O}(\log MAX \cdot m \cdot \log n)$, что и требовалось.

3. **Ближайший по значению.**

   Сначала за $\mathcal{O}(n \cdot \log n)$ отсортируем a и b. Теперь поставим по указателю на начало каждого массива. До тех пор, пока рассматриваемый в b эл-т не больше рассматриваемого эл-та a — сдвигаем указатель на b на один вправо. Как только нашли больший эл-т, смотрим, кто ближе к эл-ту a — расматриваемый сейчас или предыдущий эл-т b. Победителя выводим. Затем сдвигаем указатель в массиве a на один вправо, а указатель в b на один влево (чтобы не писать if для случаев, кто был ближе к предыдущему эл-ту a, текущий или предыдущий из b). И продолжаем действовать по тому же алгоритму.

   Почему корректно. Пусть указатель на $a[i]$. Для него по алгоритму мы находим первое $b[j]$, большее $a[i]$. Тогда $b[j-1]$ либо равен $a[i]$, либо меньше. Сравним $b[j] - a[i]$ и $a[i] - b[j-1]$. У кого разност меньше, тот и ближе к $a[i]$, тот и победил. Теперь указатель передвигается на $a[i+1]$. Он не меньше $a[i]$, следовательно, ближайшее к нему число не меньше нашего предыдущего победителя. Но, чтобы не писать лишний if, можем просто считать, что всегда побеждает $b[j-1]$. Если это неправда, то мы просто сделаем одну лишнюю операцию, в сумме сделаем n лишних операций. А так как метод указателей отработает за $\mathcal{O}(n)$, то эти лишние сдвиги нам ничего не испортят. Время работы алгоритма — $\mathcal{O}(n \cdot \log n)$, то есть самая долгая получилась сортировка. На отсортированных сработал бы за $\mathcal{O}(n)$.

4. **Код на языке C++.**

   - Множество-разность.

```cpp
int p = 0;
int q = 0;
vector <int> ans;

while (p != a.size() && q != b.size()){

    if (a[p] == b[q]){
        p++;
        q++;
    }else if (a[p] > b[q]){
        q++;
    }else{
        ans.push_back(a[p]);
        p++;
    }

}

for (int i = p; i < a.size(); i++)
    ans.push_back(a[i]);
```

- Множество-объединение.

```
int p = 0;
int q = 0;
vector <int> ans;

while (p != a.size() && q != b.size()){

    if (a[p] == b[q]){
        ans.push_back(a[p]);
        p++;
        q++;
    }else if (a[p] > b[q]){
        ans.push_back(b[q]);
        q++;
    }else{
        ans.push_back(a[p]);
        p++;
    }

}

for (int i = p; i < a.size(); i++)
    ans.push_back(a[i]);
for (int i = q; i < b.size(); i++)
    ans.push_back(b[i]);
```

5. **Ближайший по координате.**
Считываем координаты точек сразу в два массива: X — в нем для точки сначала идет координата x, а затем y, и в массив Y — в нем сначала y, затем x. Отсортируем эти два массива. Теперь в X точки идут по порядку слева направо, а те, у которых одинаковая координата по x идут снизу вверх. В Y же они идут снизу вверх, а для равных y слева направо.

Теперь на каждый запрос будем отвечать при помощи бинпоиска. Когда для заданной нам точки ищем ближайшие точки слева и справа с одинаковым Y: запускаем бинпоиск на массиве Y сразу по двум координатам (приоритет у y). Если существует ближайший справа с такой же координатой по $0y$, то в конце работы бинпоиска мы получим индекс на один меньший, чем у этого эл-та. Если существует ближайший слева - на один больший. При существовании обоих, очевидно, бинпоиск укажет нам на место между ними. Если же нет точек с токй же координатой по $0y$, мы просто будем стоять между эл-там с другими y, ближайшими к нашему. Но это неважно, просто надо каждый раз проверять равенство между данной нам координатой и той, что указал бинпоиск.

Точно также находим ближайшие сверху и снизу с таким же x, но теперь запускаем бинпоиск на массиве X. Время работы алгоритма: $\mathcal{O}(n \cdot \log n)$ на сортировку массивов и $\mathcal{O}(\log n)$ на поиск ближайших по одному запросу. Итого: $\mathcal{O}(min(n \cdot \log n, q \cdot \log n))$.

2 Дополнительная часть.

2. **Коробки с предметами.**

   Отсортируем наши предметы по весу с максимального до минимального и запишем их в список. Поставим два указателя l и r: на начало и конец списка. Будем действовать жадно. Ищем для $w[l]$ такой эл-т с максимальным весом, который еще можно положить с ним в коробку.

   Поиск происходит при помощи указателей. Сдвигаем r влево до тех пор, пока $w[r] + w[l] \leq W$. Тут три варианта:

   - Если в какой-то момент r сравнялся с l, значит, любой эл-т можно положить в коробку с нашим. А так как $w[l]$ — максимальный, следовательно, можем сложить в одну коробку любую пару предметов. Складываем все предметы попарно, начиная с максимального. Мы выполнили задание
   - Если неравенство $w[r]+w[l] \leq W$ перестало выполняться, и $l! = r$. Мы дошли до первого такого предмета, который в паре с текущим будет давать перевес. Но предыдщий $w[r-1]$ равенство выполнял. Складываем его в коробку с $w[l]$, сдвигаем l на один вправо, удаляем $w[r-1]$ (в списках это делается за $\mathcal{O}(1)$). Указатель r сбрасывать не нужно. Ведь новый $w[l]$ меньше старого, следовательно, все предметы с индексом меньше r тоже можно к нему сложить. Просто оставляем r на месте и продолжаем действовать по алгоритму.
   - Неравенство ни разу не выполнилось. То есть даже эл-т с минимальным весом нельзя сложить в одну коробку с $w[l]$. Тогда кладем $w[l]$ в отдельную коробку, l увеличиваем на единицу и снова действуем по алгоритму.

   Когда l дойдет до конца списка, все эл-ты будут разложены по коробкам. А так как старались класть предметы с весом как можно большим в паре с максимально доступными, то коробок затрачено минимальное кол-во.

   Время работы алгоритма: $\mathcal{O}(n \cdot \log n)$ на сортировку, $\mathcal{O}(n)$ на метод указателей (каждый предмет мы рассматривали не больше одного раза). Итого: $\mathcal{O}(n \cdot \log n)$.