Exercise 1

Set implementation using ordered-array.

Implement the set data type using an ordered array. That is, you maintain the n keys in the set in ascending order in an array. Implement the following operations with the specified worst-case time complexity:

`add(key)`	*add the key to the set*	$\Theta(n)$
`contains(key)`	*is the key in the set?*	$\Theta(\log n)$
`ceiling(key)`	*smallest key in set* $\geq$ *given key*	$\Theta(\log n)$
`rank(key)`	*number of keys in set* $<$ *given key*	$\Theta(\log n)$
`select(i)`	*i-th largest key in the set*	$\Theta(1)$
`min()`	*minimum key in the set*	$\Theta(1)$
`remove(key)`	*delete the given key from the set*	$\Theta(n)$

Note:

- You may use `std::vector` for resizing support.
- Do not use binary search from standard library. Implement your own binary search for `contains`, `ceiling`, and `rank` operations.
- For `add` and `remove`, you may need to shift the elements in the array. Do not use vector's `insert` and `erase` functions.
- Make sure that set contains unique keys.

Exercise 2

Exam room (Leet code 855).

There is an exam room with n seats in a single row labeled from 0 to $n-1$. When a student enters the room, they must sit in the seat that maximizes the distance to the closest person. If there are multiple such seats, they sit in the seat with the lowest number. If no one is in the room, then the student sits at seat number 0.

Implement the `ExamRoom` class that simulates the mentioned exam room.

- `ExamRoom(int n)`
  Initializes the object of the exam room with the number of the seats n.

- `int seat()`

  Returns the label of the seat at which the next student will set.

- `void leave(int p)`

  Indicates that the student sitting at seat p will leave the room. It is guaranteed that there will be a student sitting at seat p.

Example:

```
ExamRoom examRoom(10);
examRoom.seat(); // return 0, no one is in the room, then the student sits at seat number 0.
examRoom.seat(); // return 9, the student sits at the last seat number 9.
examRoom.seat(); // return 4, the student sits at the last seat number 4.
examRoom.seat(); // return 2, the student sits at the last seat number 2.
examRoom.leave(4);
examRoom.seat(); // return 5, the student sits at the last seat number 5.
```

Hint: The problem requires us to seat students in such a way that they are seated as far as possible from the already seated students. The challenge lies in efficiently finding the optimal seat given potentially many occupied seats while ensuring that we can update our data structure when a student leaves. The key observation is that we can treat the empty seats as segments/gaps and determine the best seat to occupy in those segments based on the distance from the occupied seats.

We use a set from previous exercise or `std::set` from the standard library to maintain a list of occupied seats. This allows us to quickly check which seats are taken and to remove seats efficiently when students leave.

Seating Logic:

- If the room is empty, the first student sits at seat 0.
- If there are already occupied seats, we:
  - Calculate the distance from the start of the row (seat 0) to the first occupied seat.
  - Check the gaps between each pair of occupied seats to find the maximum distance.
  - Finally, calculate the distance from the last occupied seat to the end of the row (seat $n-1$).
- The seat that provides the maximum distance from the occupied seats among all empty segments is chosen .

Leaving Logic: When a student leaves, we simply remove their seat from the set of occupied seats.