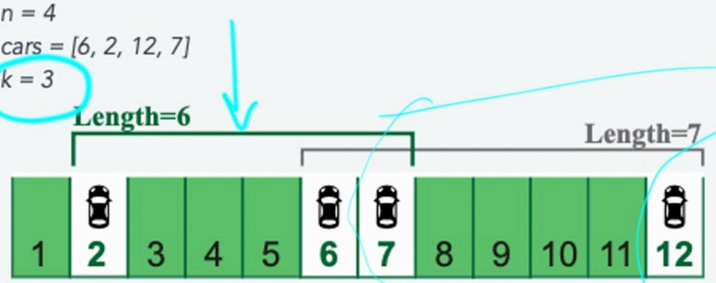
2. Parking Dilemma

There are many cars parked in a parking lot. The parking lot is a straight line with a parking spot for every meter. There are *n* cars currently parked and a roofer wants to cover them with a roof. The requirement is that at least *k* cars are covered by the roof. Determine the minimum length of the roof that will cover *k* cars.

Example

n = 4
cars = [6, 2, 12, 7]
k = 3



Two roofs that cover three cars are possible: one covering spots *2* through *7* with a length of *6*, and another covering slots *6* through *12* with a length of *7*. The shortest roof that meets the requirement is of length *6*.

Function Description

Complete the function *carParkingRoof* in the editor below.

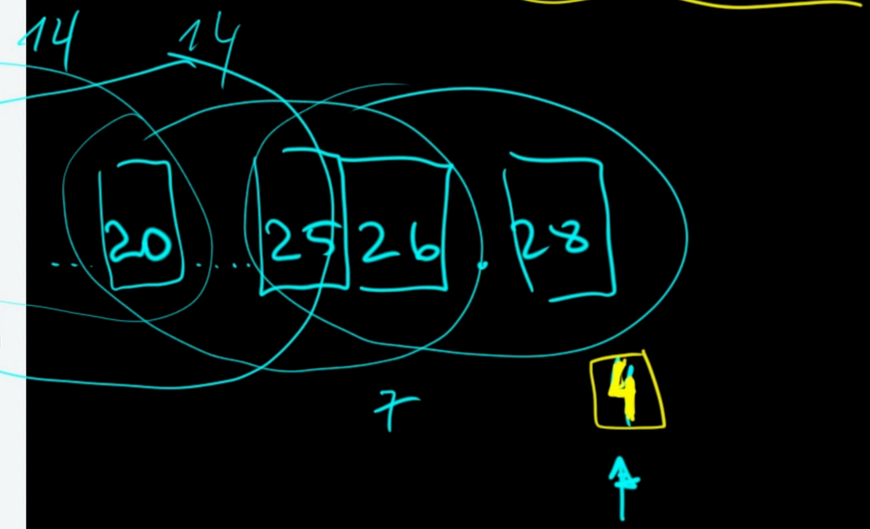
carParkingRoof has the following parameter(s):
int cars[n]: the parking spots where cars are parked
int k:the number of cars that have to be covered by the roof
Returns:
int: the minimum length of a roof that can cover *k* cars

Constraints

- $1 \le n \le 10^5$
- $1 \le k \le n$
- $1 \le cars[i] \le 10^{14}$
- All spots taken by cars are unique

```
fn car_parking_roof1(cars: &[u32], k: usize) -> u32
```

```
[6, 2, 12, 7, 25, 20, 26, 28];
```




```
let mut cars:Vec<u32> = cars.to_vec();
cars.sort_unstable();

println!("{:?}", cars);
```

```
[2, 6, 7, 12, 20, 25, 26, 28]
```

2. Parking Dilemma

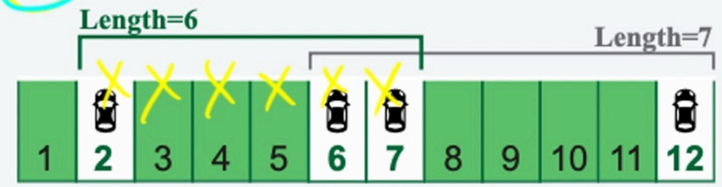
There are many cars parked in a parking lot. The parking lot is a straight line with a parking spot for every meter. There are *n* cars currently parked and a roofer wants to cover them with a roof. The requirement is that at least *k* cars are covered by the roof. Determine the minimum length of the roof that will cover *k* cars.

Example

n = 4

cars = [6, 2, 12, 7]

k = 3



Two roofs that cover three cars are possible: one covering spots *2* through *7* with a length of *6*, and another covering slots *6* through *12* with a length of *7*. The shortest roof that meets the requirement is of length *6*.

Function Description

Complete the function *carParkingRoof* in the editor below.

carParkingRoof has the following parameter(s):

int cars[n]: the parking spots where cars are parked

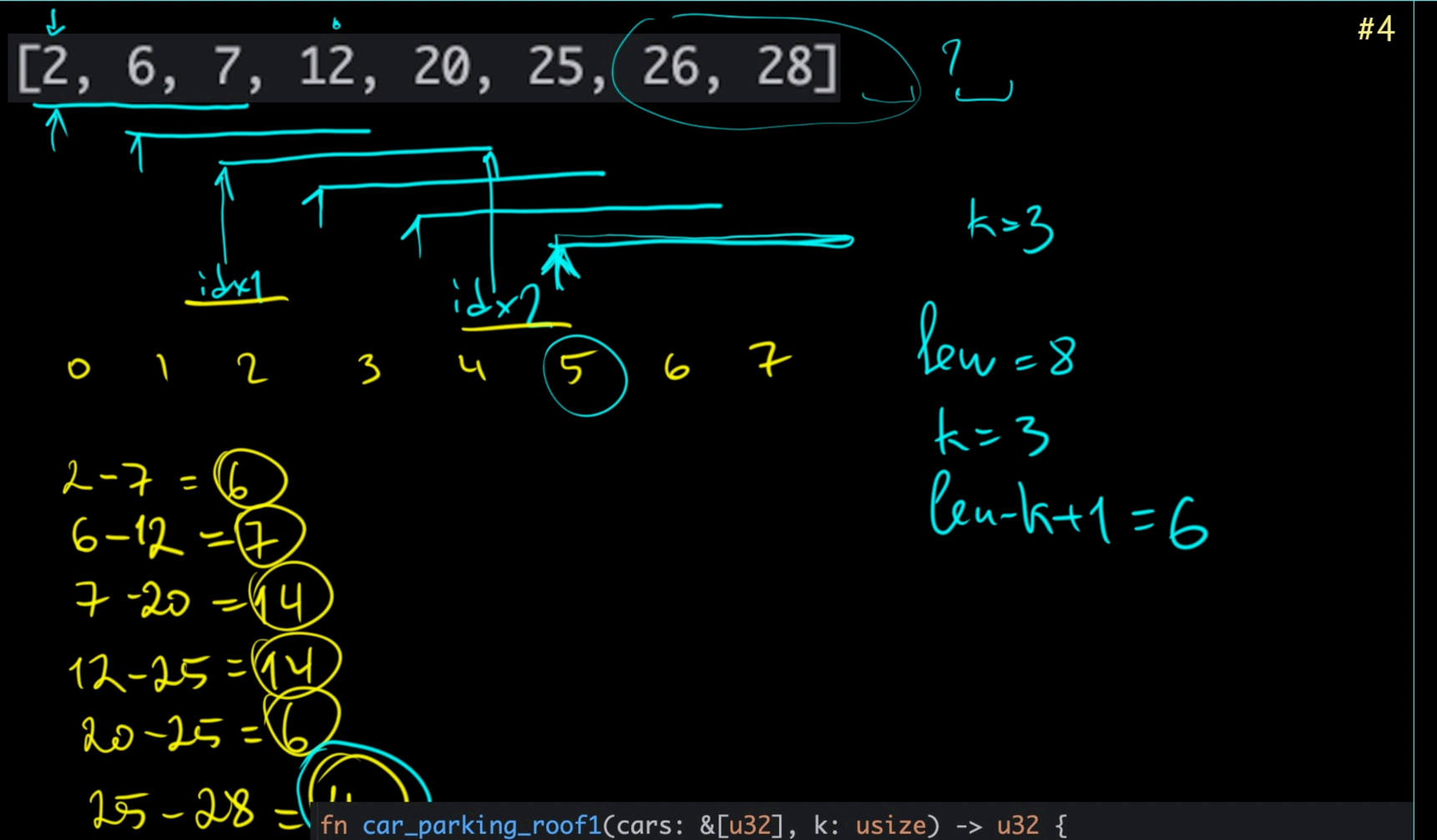
int k:the number of cars that have to be covered by the roof

Returns:

int: the minimum length of a roof that can cover *k* cars

Constraints

- $1 \le n \le 10^5$
- $1 \le k \le n$
- $1 \le cars[i] \le 10^{14}$
- All spots taken by cars are unique




```rust
fn car_parking_roof1(cars: &[u32], k: usize) -> u32 {
    let mut cars: Vec<u32> = cars.to_vec();
    cars.sort_unstable();
    let roof_size: fn(...) -> ... = |i1: usize, i2: usize| cars[i2] - cars[i1] + 1;

    let mut mn: u32 = u32::MAX;
    for idx: usize in 0..=cars.len() - k {
        let size: u32 = roof_size(idx, i2: idx + (k - 1));
        mn = min(mn, size);
    }
    mn
}
```

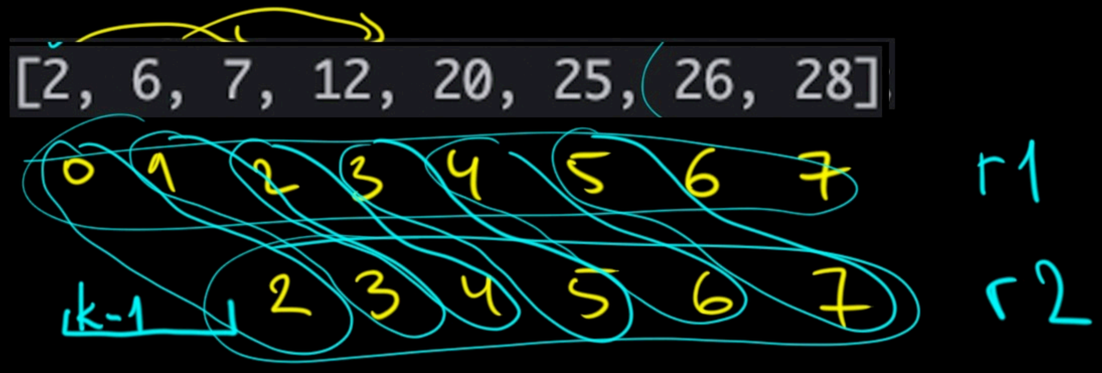
```rust
fn car_parking_roof1(cars: &[u32], k: usize) -> u32 {
    let mut cars: Vec<u32> = cars.to_vec();
    cars.sort_unstable();
    let roof_size: fn(...) -> ... = |i1: usize, i2: usize| cars[i2] - cars[i1] + 1;

    let mut mn: u32 = u32::MAX;
    for idx: usize in 0..=cars.len() - k {
        let size: u32 = roof_size(idx, i2: idx + (k - 1));
        if size < mn {
            mn = size;
        }
    }
    mn
}
```

```rust
fn car_parking_roof2(cars: &[u32], k: usize) -> u32 {
    let mut cars: Vec<u32> = cars.to_vec();
    cars.sort_unstable();
    let roof_size: fn(usize) -> u32 = |idx: usize| cars[idx + k - 1] - cars[idx] + 1;

    (0..=cars.len() - k)
        .map(|i: usize| roof_size(i)): impl Iterator<Item=u32>
        .min(): Option<u32>
        .unwrap()
}
```

roof_size(idx, i2: idx + (k - 1))



0 → 2 7-2+1 = 6

1 → 3 12-6+1 = 7

5 → 7 28-25+1 = 4

r1.zip(r2)

(0,2) (1,3) (2,4)

```rust
fn car_parking_roof2(cars: &[u32], k: usize) -> u32 {
    let mut cars: Vec<u32> = cars.to_vec();
    cars.sort_unstable();
    let roof_size: fn(usize) -> u32 = |idx: usize| cars[idx + k - 1] - cars[idx] + 1;

    (0..=cars.len() - k)
        .map(|i: usize| roof_size(i)): impl Iterator<Item=u32>
        .min(): Option<u32>
        .unwrap()
}

fn car_parking_roof3(cars: &[u32], k: usize) -> u32 {
    let mut cars: Vec<u32> = cars.to_vec();
    cars.sort_unstable();
    let roof_size: fn(...) -> u32 = |i1: usize, i2: usize| cars[i1].abs_diff(other: cars[i2]) + 1;
    let r1: Range<usize> = 0..cars.len();
    let r2: Range<usize> = (k - 1)..cars.len();
    r1.zip(r2): impl Iterator<Item=(...)>
        .into_iter(): impl Iterator<Item=(...)>
        .map(|(i1: usize, i2: usize)| roof_size(i1, i2)): impl Iterator<Item=u32>
        .min(): Option<u32>
        .unwrap()
}
```