Problem 1

Because every time we push an integer `i` into the stack, we will pop out the element that is larger than `i`, and put `i` on the top of the stack, which promises that the top of the stack is always the largest integer of the stack, so the stack is always in decreasing order from the top to the bottom.

Worst Time Analyze

Push

The worst-case of push operation appear when inserting a number larger than all of the element in the stack, it will iterate the whole of the stack, which takes $O(n)$ time.

```
function push:
  for each element in stack:
    if element > i:
      stack pop out element
  push i into the stack
```

- Line 2 to 4 take $O(n)$.
- Line 5 take $O(1)$.
- $O(n) + O(1) = O(n)$

Pop

The worst-case of pop operation is $O(1)$, it only operate the top element on the stack.

Amortized running time

Push

By observation, for every operation that takes $O(i)$, there must exist at least i operations that take $O(1)$ before it (otherwise there will be not enough element for i iterations which takes $O(i)$), assume the `n`th operation takes $O(n)$:

$$nO(f(n)) = \sum_{i<n} O(1) + O(n) = nO(1) + O(n) = O(n)$$

For each push operation that takes $O(n)$, it's time can be amortized to `n` operation:

$$O(f(n)) = \frac{O(n)}{n} = O(1)$$

Pop

Every pop operation takes $O(1)$ so among n operation:

$$nO(f(n)) = \sum_{i<n} O(1) = O(n)$$

The time complexity over n operation can be amortized to `n` operation:

$$O(f(n)) = \frac{O(n)}{n} = O(1)$$

Problem 2

Main idea: Use an array-based lists to store the element, and record the index of the head and tail of the list.

partition a)

List structure:

- The list is an array that is able to contain k elements.
- Use four integers to help implement the list:
  1. `head`, always indicating the first element of the list.
  2. `tail`, always indicating the last element of the list.
  3. `size`, recording how much element is in the list.
  4. `maxCapacity`, recording the largest amount of element this list can hold.
- At the beginning, both `head` and `tail` are pointing at the first index of the array.

ADDTOFRONT

- If the list is full, do nothing.
- If the list is not full, decrease the index of the head by 1, place the element.

ADDTOBACK

- If the list is full, do nothing.
- If the list is not full, increase the index of the tail by 1, place the element.

REMOVEFROMFRONT

- If the list is empty, do nothing.
- If the list is not empty, increase the index of the head by 1.

REMOVEFROMBACK

- If the list is empty, do nothing.
- If the list is not empty, decrease the index of the tail by 1.

SETELEMENT

- If the element index is larger than list size, which means it is out of bound, do nothing.
- If not, set the element.

Partition b)

1. A list with max capacity k can contain k elements.
2. The head index of the list doesn't have to be the `list[0]`, the tail index of the list doesn't have to be the `list[maxCapacity-1]`.
3. To obtain an element after a specific element with index `i`, there are two cases:

   - When `i` is smaller than `maxCapacity-2`, the next element index is `i+1`.
   - When `i` is exactly `maxCapacity-1`, the next element index is `0`.

   Both two cases can be represented as `(i+1)%maxCapacity`, it is easy to deduce that `j` element after `i` can be represented as `(i+j)%maxCapacity`.

4. From paragraph 3, it is also clear that the index of the element before a specific element with index `i`, is `(i-1)%maxCapacity`.
5. To insert an element at the front of the list, we can just put the element into the position before the current first element, set it as the head. Other functions use the same logic.

Partition C)

1.
```
def AddToFront(e)
  if size == max capability
    return "array is full"
  else
    head = (head - 1) % maxCapacity
    list[head] = e
    size+=1
```

   - If statement take $O(1)$.
   - Assignments and return in row 3, 5, 6 and 7 takes $O(1)$.
   - This function takes $O(1)$ in total.

2.
```
def AddToBack(e)
  if size == max capability
    return "array is full"
  else
    tail = (tail + 1) % maxCapacity
    list[tail] = e
    size+=1
```

   - Same as `AddToFront`.

3.
```
def RemoveFromFront():
  if size == 0:
    return "array is empty"
  else
    result = list[head]
    head = (head + 1) % maxCapacity
    size-=1
    return result
```

   - If statement take $O(1)$.
   - Assignments and return in row 3, 5, 6, 7 and 8 takes $O(1)$.
   - This function takes $O(1)$ in total.

4.
   ```
   def RemoveFromBack():
     if size == 0:
       return "array is empty"
     else
       result = list[tail]
       tail = (tail - 1) % maxCapacity
       size-=1
       return result
   ```

   - Same as `RemoveFromFront`.

5.
   ```
   def SetElement(i, e):
     if i > size or i < 0:
       return "out of bound"
     else
       targetIndex = (i + head) % maxCapacity
       list[targetIndex] = e
   ```

   - If statement take $O(1)$.
   - Assignments and return in row 3, 5 and takes 6 $O(1)$.
   - This function takes $O(1)$ in total.

Problem 3

Partition a)

1. For every sensor, there is a list `connect[i][]` contains all of the censors after `i` can communicate with sensor `i`.
2. For an arbitrary point p, if there exist points a and b where $a, b > p$ that $|p - a| < r$ and $|p - b| < r$, then we have $|a - b| < r$.

While `i` is smaller than the size of `sensorList` do:

- Let `cursor` equal to `i`:
  1. If `sensorList[cursor+1]` is within the radius of `sensorList[cursor]`, add `sensorList[cursor+1]` into `connect[n][]` where n is all number from `i` to `cursor`, let `cursor` equal to `cursor+1`, back to step 1.
  2. If `sensorList[cursor+1]` is without the radius of `sensorList[cursor]`, let `i` equal to `cursor+1`.

Partition b)

Correctness of `connect[][]`

If `a` can communicate with `b`, `a` can communicate with sensors that can communicate with `b`.

If sensors `(a, b)` are connected, there are two cases:

- The distance of `a` and `b` is smaller than radius `r`.
- There exist intermediate sensors $i_1, i_2 ... i_n$ that distance of `a` and i_1, i_1 and i_2, i_2 and i_3, ... , i_n and `b` are all within `r`.

Since `connect[i][]` only contains sensors after `i` can communicate with `i`, for pair look like (i, j), it will be stored in `connect[i][]`, for pair look like (j, i), it will be stored in `connect[j][]`, so if the algorithm works, `connect[][]` should contain all pairs of sensors which can communicate with each other.

Correctness of algorithm

As shown above, if `a` can communicate with `b`, `b` can communicate with `c`, then `a` can communicate with `c`. Similarly, if `a` can communicate with `b`, but `b` cannot communicate with `c`, then `a` can't communicate with `c`.

So when we find sensors that can communicate after `a`, we just need to find sensors that can communicate with the sensor which is after `a`. Once we find the next sensor cannot communicate with the previous one, it means this sensor cannot communicate with sensors that can communicate with the previous one.

If we find such a cut-off, it means we already found all pairs before the cut-off can communicate with each other.

So if the cut-off is at the end of the list of sensors, then we find all pairs of sensors that can communicate with each other.

Partition c)

```
def findPair(sensorList, r){
  sensor_i = 0
  while sensor_i < size of sensorList do
    cursor = sensor_i
    while sensorList[cursor + 1] can communicate with sensorList[cursor] do
      i = sensor_i
      while i <= cursor do
        add sensorList[cursor + 1] to connect[i][]
      cursor = cursor + 1
    sensor_i = cursor + 1
  return connect
}
```

Assumption

Although there are three while loop, the time complexity is still $O(n + k)$, consider two boundary situation of the algorithm.

1. There is no sensor that can communicate with another sensor:

   In this situation, the algorithm will not run into the second while loop, `sensor_i` will go through the whole `sensorList`, so the time complexity of this situation is $O(n)$.

2. All of the sensors can communicate with each other:

   In this situation, the algorithm will be stuck in the second while loop, the time complexity of the loop is the sum of the time complexity of the third while loop.

   It is clear that the sum of the time we enter the third loop is equal to the number of pairs of sensors that can communicate with each other, so if all of the sensors can communicate with each other, the time complexity of the second while loop is $O(k)$.

So the time complexity of the algorithm will be $O(n)$ or $O(k)$, depends on which is larger. $O(n)$ or $O(k)$ can also be represented as $O(n+k)$.

Proof these two situations is the worst case

From assumption, let $x \in \{x|0,1,2,\dots,N\}$ be the number of sensors that can communicate with others, y is the time complexity:

- When $x=0$, the time complexity of the first loop is $O(N)$, when $x=N$, the time complexity of the first loop is $O(1)$.
    - For the first loop, we can have an equation:
    $$y1 = \frac{1-N}{N}x + N$$
- When $x=0$, the time complexity of the second $O(1)$, when $x=N$, the time complexity of the second loop is $O(N)$.
    - For the second loop, we can have an equation:
    $$y2 = \frac{k-1}{N}x + 1$$
- For arbitrary x, we can calculate the time complexity as:
$$\begin{aligned} y1 + y2 &= \frac{1-N}{N}x + N + \frac{k-1}{N}x + 1 \\ &= \frac{k-N}{N}x + N + 1 \end{aligned}$$
    1. If $\frac{k-N}{N} > 0$, that is $k > N$, the maximum of the equation is $K+1$ when $x=N$.
    2. If $\frac{k-N}{N} < 0$, that is $k < N$, the maximum of the equation is $N+1$ when $x=0$.

So two situations in the assumption are the worst case, the time complexity is $O(N+k)$.