Prob 2:
a)

```
struct stack {
   unsigend short size (2)
   T minimum (sizeof T)
   T data[maxSize] (sizeof T * maxSize)
 }
```

Following type T, the struct stack need size that sizeof (T * (max size + 1) + 4) bytes. Size of stack always be n >= 0. It couldn't be negative. If max size is bigger then 65535, size could be int or more bigger type.

In initializing, size = 0, minimum = MAX_VALUE_OF_TYPE is required.

```
PUSH(T) {
   stack.data[size] = T;
   if (T < stack.minimum) T = stack.minimum;
}
```

-> add T at index 'size' of array data of stack. if size is 0, minimum is T, and size + 1. else T is smaller then minimum in stack, minimum is T. Or not ignore.

```
POP() {
   if (!isEmpty()) {
      T t = stack.data[size];
      size -= 1;
      return t;
   } else
      return -1; // Means stack is empty.
   }
}
```

-> return T at index 'size' of array data of stack. And size -1. If size is 0, return error type of T.

```
TOP() {
   if (!isEmpty()) return stack.data[size];
   return -1; // Means stack is empty.
}
```

-> return T at index 0 of array data of stack. If size is 0, return error type of T.

```
SIZE() {
   return stack.size;
}
```

-> return size of stack.

```
isEmpty() {
   return SIZE();
}
```

-> return size of stack. In programming, 0 means false, else means true.
getMinimum() -> return minimum of stack.

b) The correctness follows size of stack. Need to handle 0 size case and max size case. We re-define the max size of stack or expand max size of stack with dynamic array if there is more data then max size of stack.

c) PUSH(T) always add a values in last runs in O(1) time, POP() always remove and return value in last runs in O(1) time, TOP() return last value when size is no 0 runs in O(1) time, SIZE() always constant value of stack runs in O(1) time, isEmpty() always return size of stack runs in O(1) time, getMinimum() always return constant T of stack runs in O(1) time follow a), need size that sizeof (T * (max size + 1) + 4) bytes.

Prob 3:
a) Traverse from last index to first index. On each data(height of lighthouse) traverse current index of data of index 0, start with data -1 and if data[k] is bigger than data[i], data - 1, desl escape inner traverse and so on.

b)

```
int [] func(int arr[]) {
        for (int k = arr.size() - 1; k >=0; k—) {
            if ( k == 0 ) {
                arr[k] = -1;
                break;
            }
            for( int i = k-1; i >= 0; i—) {
                arr[k]—;

                if (arr[k] > arr[i]) arr[k]—;
                else break;
            }
        }
    return arr;
}
```

b)
func() travels the array only once from bottom to top(index 0) runs in O(n). Though inner travels from current index of data to top. If met bigger one, it will be done. In this case, best one runs in O(1) when meet bigger one at first every single inner traverse. but the Worst one runs in O(log n) when in every traverse from current index to 0. So it takes time complexity n to n log n. As the Big-O time complexity, the time complexity of algorithm is O(n log n).