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) Let the input array be A.

begin algorithm
create an array 'index_arr' to store the index of the previous greater element for each element in array

create a new empty stack 'st'

for every element A[i] in the input array 'A':
while stack is not empty and current element A[i] is greater than or equal to element stored at index equal to top element of stack i.e, A[st.top()]
pop the top element of stack
end while

if stack is empty:
push index of current element in stack 'st'
push -1 in 'index_arr'

else:
insert top element of stack into array at current index
push index of current element onto stack
end if

end for
end algorithm

Note : -1 is inserted if there is no lighthouse that is greater than the current lighthouse to it's left.

B)

Proof of correctness:

For each element in array, indices having smaller value or equal to the current element are popped off the stack and when it encounters a greater element or the stack becomes empty, the while loop breaks meaning there are no indices in the stack that refer to a smaller value than the current element. Therefore, for each element in the input array, we can only encounter a index having greater value than the current element. Hence the program works correctly.

C)

Time complexity analysis:

At first, the program looks to run in O(n^2) time because of the 2 nested loops but the inner loop runs for a total of n times at max. Since, we are inserting each index only once in the stack, there can be a maximum of n pop operation on the stack. Therefore, the outer loop runs for n times and the inner loop also runs for a maximum of n times in total. Hence the program can runs in O(2*n) at max which is also O(n).

Example:

suppose the array is sorted in ascending order (worst case) :

for each element in the array, the stack will have only index of only one element. The element of stack can never be greater than the current element therefore, top element of the stack is always popped. i.e, n pop operations.

Therefore, the program runs in O(n).