Problem One: Array Rotation

Here is one possible O(n)-time, O(1)-space solution:

```
void rotateArray(int* array, size_t n, size_t amount) {
   reverse(array, 0, n);
   reverse(array, 0, n - amount);
   reverse(array, n - amount, n);
}

void reverse(int* array, size_t start, size_t end) {
   size_t n = end - start;
   for (size_t i = 0; i < n / 2; i++) {
      size_t lhs = start + i;
      size_t rhs = start + n - 1 - i;

      int temp = array[lhs];
      array[lhs] = array[rhs];
      array[rhs] = temp;
   }
}
```

This approach works by reversing the entire array, then selectively re-reversing two ranges in a way that will effectively end in a rotation. Try it out on some inputs!

Problem Two: Searching a Rotated Array

This approach works by using a modified version of binary search. At each step, we see if the front half of the array or the back half of the array is sorted (one of them will be). If the value to search for lies between the start and endpoint of the sorted range, we explore that half. Otherwise, we explore the other.

```
bool searchRotatedArray(int* array, size_t n, int key) {
  size_t lhs = 0;
  size_t rhs = n;

  while (lhs < rhs) {
    size_t mid = (rhs - lhs - 1) / 2 + lhs;

    if (array[mid] == key) return true;

    /* See if the first half of the array is sorted. */
    if (array[lhs] <= array[mid]) {
      /* If we're within the bounds of that half, continue to explore that
       * half. Otherwise, explore the other.
       */
      if (array[lhs] <= key && key < array[mid]) rhs = mid;
      else lhs = mid + 1;
    }
    /* Otherwise, the second half is sorted. */
    else {
      if (array[mid] < key && key <= array[rhs-1]) lhs = mid + 1;
      else rhs = mid;
    }
  }

  return false;
}
```