Binary Search

The binary search algorithm is a mainstay in computer science. Given a ***sorted*** array or vector of items, the algorithm is used to test whether or not a candidate item (the ***key***) is in the array or not. Often the algorithm is written in such a way that it returns the position (index) within the array at which the key is located, rather than only a Boolean value indicating whether key was found or not.

Recall that the linear search algorithm solves this same problem, but does not assume the array is sorted. By beginning with a sorted array, binary search usually runs much faster than linear search.

Binary search begins by identifying the item in the middle of the array and comparing it against the key. If the middle item matches the key, then the position of the middle item is returned. If the middle item is larger than the key, the algorithm repeats itself on the sub-array to the left of the middle item; if the middle item is smaller than the key, the algorithm repeats on the sub-array to the right of the middle item. If the remaining sub-array to be searched ever becomes empty, we know the key is not in the array.

The "repetition" part of the algorithm can be implemented using iteration (a loop) or recursion. In either case, the algorithm maintains two variables to keep track of the current upper and lower bounds for the portion of the array that could potentially contain the key.

We present the algorithm as a search over a sorted vector of integers, though any data type can be used as long as the vector is sorted in some fashion.

1. We are given
    a. an array `A` of size `n`, indexed from `0` to `n-1`
    b. an integer `key` to look for in the array
    c. an integer `low` that is the lowest index in the array that could contain the key
    d. an integer `high` that is the highest index in the array that could contain the key
2. If low > high, then the item is not found (return `-1`)
3. Compute the middle position in the array.
4. If the item at the middle position is the key, return that position.
5. If the item at the middle position is greater than the key, repeat from step 2, on the left sub-array.
6. If the item at the middle position is less than the key, repeat from step 2 on the right sub-array.

For steps 5 and 6, if using recursion, the "repeat" part is done by calling your binary search function with new argument values for low or high.

We will write a recursive version of binary search.

1. Get the starter code and paste it into a new project.
2. Find the 4-argument `binary_search` function. Notice that the arguments: the vector to search, the key to look for, the lower bound, and the upper bound.
3. Edit this function to implement binary search correctly. Hint: Use lots of cout statements as you are writing the code. It will help you understand the recursion better if you can see exactly what the function is doing. For instance, try printing out the arguments at the top of the function, printing out "base case" or "recursive case" when those cases are entered into, and printing out "returning" whenever the function returns.
4. Change the code so you can control the starting size of the array. Experiment with various sizes of the array, and see if you can discover how many calls will be made to binary_search in the worst case, for an arbitrary sorted array.