Binary Search

- In many applications, it is important to have the capability to quickly search through a collection of elements, i.e. to find a specific element within the larger collection, or to determine that element does not exist in the collection.

- It is possible, using the data structures we've seen up to now (i.e. linear data structures like dynamic arrays and linked lists), to search for specific elements.

- However, doing so is fairly expensive, since, for these data structures, our only choice at this point is to search by iterating through the data structure's elements one by one until we either find the element we're looking for or reach the end of the collection (at which point we know the element doesn't exist in the collection).

- This ***linear search*** algorithm, where we must iterate through all of the elements in a collection, has *O(n)* complexity, a cost that becomes more and more prohibitive as our collection grows.

- Here, we'll explore a different kind of search algorithm known as ***binary search***, which works by iterating through an *ordered* (i.e. sorted) array and, at each iteration, eliminating *half* of the remaining array from consideration.

- Recall that, as we've already explored, algorithms like this that cut a problem of size *n* in half until the problem size is 0 have *O(log n)* runtime complexity.

- This means that binary search is quite a bit faster than linear search, particularly for large values of *n*. Indeed, the differences in speed can be striking:
    - For an array of size *n* = 1,000,000, linear search would have to perform on the order of 1,000,000 comparisons to find a specific element, on average. Binary search, on the other hand, would have to perform on the order of *log(*1,000,000*)* ≈ 20 comparisons, on average.
    - For an array of size *n* = 4,000,000,000, linear search would have to perform on the order of 4,000,000,000 comparisons, on average, while binary search would have to perform on the order of *log(*4,000,000,000*)* ≈ 32 comparisons, on average.

- At a high level, here's how binary search works. At each iteration it:

    - Compares the ***query value*** (i.e. the value it's searching for) to the value at the exact midpoint of the array.
    - If this value matches the query value, halt and return some value to indicate that the query value has been found.
    - Otherwise:
        - If the query value is *less than* the value at the array's midpoint, repeat, focusing only on the "lower" half of the array, i.e. the half of the array before the midpoint.
        - If the query value is *greater than* the value at the array's midpoint, repeat, focusing only on the "upper" half of the array, i.e. the half of the array after the midpoint.
    - If the array under consideration has size 0, halt. The query value does not exist in the array in this case. Here we can either return some value to indicate that the query value hasn't been found (e.g. -1), or we can return the index in the array at which is should be inserted to maintain the sorted order of the array.

- In actual C code, binary search looks like this:

```
int binary_search(int q, int* array, int n) {
  int mid, low = 0, high = n - 1;
  while (low <= high) {
    mid = (low + high) / 2;
    if (array[mid] == q) {
      return mid;
    } else if (array[mid] < q) {
      low = mid + 1;
    } else {
      high = mid - 1;
    }
  }
  return low;
}
```

- Here, `low` represents the first index of the sub-array we're currently focusing on, and `high` represents the last index of that array. Initially, these are 0 and *n-1*, indicating that we start out by focusing on the entire array.

- The value `mid` represents the index of the midpoint of the aub-array we're currently focusing on, i.e. the index of the value that will be compared to the query value at the current iteration.

- At each iteration of the loop in the code above, if the query value isn't found (i.e. it doesn't match `array[mid]`), one of the bounds of the sub-array is adjusted to eliminate half that array from consideration.
    - If the query value is *greater than* the value at the midpoint, `low` is moved to the index *immediately after* the midpoint, thus eliminating the *lower* half of the sub-array from consideration.
    - Otherwise, if the query value is *less than* the value at the midpoint, `high` is moved to the index *immediately before* the midpoint, thus eliminating the *upper* half of the sub-array from consideration.

- The iteration continues until `low` is greater than `high`, at which point the sub-array under consideration has size 0. This, in turn, means that the query value was not found in the array.
    - Note that if the loop terminates this way, the binary search function returns `low`. Because of the way the function works, this is guaranteed to be the index at which we'd need to insert the query value into the array in order to maintain its sorted order.

- Note that the function above can easily modified to operate recursively. The recursive version of binary search works by searching in either the lower half or the upper half of the array (depending on whether the query value was, respectively, less than or greater than the value at the midpoint) via a recursive call to the binary search function in which the endpoints of the array are modified appropriately.

A note on ordered arrays

- It is important to emphasize that binary search can only work within an ordered array (i.e. sorted). In particular, at each iteration, it relies on all values before the midpoint of the array being less than the value at the midpoint and all values after the midpoint of the array being greater than the midpoint:

- This is the assumption that allows binary search to eliminate half of the array at each iteration.

- This means, though, that if we want to be able to use binary search, we need to ensure that the array we're using is ordered. There are two options here, each of which is most appropriate in specific situations.

Ordering an array using a sorting algorithm

- One way we can order the array in which we're using binary search is to use a sorting algorithm, like quicksort.

- Because the best general-purpose sorting algorithms have *O(n log n)* runtime complexity, it is best if we limit the number of times we run them.

- For example, in the worst case, if our array contains *n* elements, we might call a sorting algorithm *n* times, leading to an overall complexity of $O(n^2 \log n)$.

- Thus, using a sorting algorithm to ensure an ordered array might not be the best choice if we expect new elements to be inserted into the array frequently, and binary search might be called on the array after any given insertion into it.

- Sorting algorithms are best used in this context when our data does not change, e.g. all of the data is loaded at the beginning of the program's execution. In this situation, we can run the sorting algorithm just a single time, and the array will be ordered and ready for binary search thereafter.

Ordering an array using binary search

- If our data is frequently changing, e.g. new elements are frequently being inserted into the array, and we might need to run binary search after any given insertion, a better option for maintaining an ordered array is to use binary search.

- Recall specifically that the `binary_search()` function above is set up to return the index at which an element should be inserted into the array, if it isn't already present there, in order to maintain the array's ordering.

- Thus, for a cost of *O(log n)*, we can identify the index at which a new element must be inserted into the array, and we can insert it with an additional cost of *O(n)* (since we need to shift subsequent elements back one spot in the array to make space for the new element).

- In other words, using this approach, the total cost to insert a new element at the correct index in the array is *O(n)* (the *O(log n)* term is dropped from the overall complexity because it is dominated by the *O(n)* term). Thus, the total cost to insert *n* elements into the array and maintain sorted order using this approach is $O(n^2)$.

- Of course, having done this analysis, we'd likely be better off using a ***binary search tree*** to store our data for an application in which we need to frequently search within data that's constantly changing. We'll explore binary search trees in the coming weeks.