

Binary search

When a method calls itself recursively, the JVM's internal stack holds two or more stack frames connected with that method. Only the top one can be accessed.

Here's a recursive method that searches a sorted array of ints for a particular int. Let *i* be an array of ints sorted from least to greatest--for instance, {-3, -2, 0, 0, 1, 5, 5}. We want to search the array for the value "findMe". If we find "findMe", we return its array index; otherwise, we return FAILURE.

We could simply check every element of the array, but that would be slow. A better strategy is to check the middle array element first. If findMe is lesser, we know it can only be in the left half of the array; if findMe is greater, we know it can only be in the right half. Hence, we've eliminated half the possibilities with one comparison. We still have half the array to check, so we recursively check the middle element of that half, and so on, cutting the possibilities in half each time. Suppose we search for 1.

```

-----
| -3 -2 0 0 1 5 5 |
-----^-----
compare with 0 |
               v
             -----
             | 1 5 5 |
             -----^-----
               |
               | compare with 5
               v
             -----
             | 1 |
             -----

```

The recursion has two base cases.

- (1) If findMe equals the middle element, return its index; in the example above, we return 4.
- (2) If we try to search a subarray of length zero, the array does not contain "findMe", and we return FAILURE.

```

public static final int FAILURE = -1;

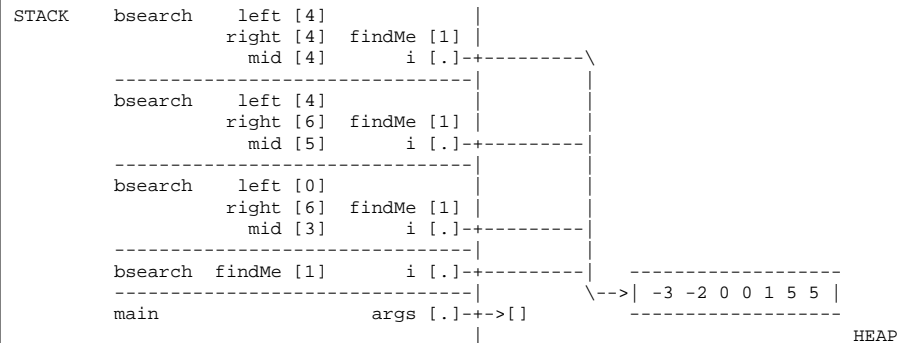
private static int bsearch(int[] i, int left, int right, int findMe) {
    if (left > right) {
        return FAILURE; // Base case 2: subarray of size zero.
    }
    int mid = (left + right) / 2; // Halfway between left and right.
    if (findMe == i[mid]) {
        return mid; // Base case 1: success!
    } else if (findMe < i[mid]) {
        return bsearch(i, left, mid - 1, findMe); // Search left half.
    } else {
        return bsearch(i, mid + 1, right, findMe); // Search right half.
    }
}

public static int bsearch(int[] i, int findMe) {
    return bsearch(i, 0, i.length - 1, findMe);
}

```

How long does binary search take? Suppose the array has *n* elements. In one call to bsearch, we eliminate at least half the elements from consideration. Hence, it takes $\log_2 n$ (the base 2 logarithm of *n*) bsearch calls to pare down

the possibilities to one. Binary search takes time proportional to $\log_2 n$. If you're not comfortable with logarithms, please review Goodrich & Tamassia Sections 4.1.2 & 4.1.7.



The stack frames appear at right in the figure above. There are three different local variables named "left" on the stack, three named "right", three named "mid", four named "i", and four named "findMe". While the current invocation of bsearch() is executing, only the topmost copy of "left" is in scope, and likewise for "right" and "mid". The other copies are hidden and cannot be accessed or changed until the current invocation of bsearch() terminates.

Most operating systems give a program enough stack space for a few thousand stack frames. If you use a recursive procedure to walk through a million-node list, Java will try to create a million stack frames, and the stack will run out of space. The result is a run-time error. You should use iteration instead of recursion when the recursion will be very deep.

However, our recursive binary search method does not have this problem. Most modern microprocessors cannot address more than 2^{64} bytes of memory. Even if an array of bytes takes this much space, we will only have to cut the array in half 64 times to run a binary search. There's room on the stack for 64 stack frames, with plenty to spare. In general, recursion to a depth of roughly $\log n$ (where *n* is the number of items in a data structure) is safe, whereas recursion to a depth of roughly *n* is not.

Unfortunately, binary search can't be used on linked lists. Think about why.

Scope and Recursion

The `_scope_` of a variable is the portion of the program that can access the variable. Here are the scoping rules we've learned so far. (More complicated scoping rules will be added when we learn about inheritance.)

- (1) Class variables (static fields) are in scope everywhere in the class, except when overridden by a local variable or parameter of the same name.
- (2) Fully qualified class variables ("System.out", rather than "out") are in scope everywhere in the class, and cannot be overridden. If they're public, they're in scope in `_all_` classes.
- (3) Instance variables (non-static fields) are in scope in non-static methods of the class, except when overridden.
- (4) Fully qualified instance variables ("kayla.name", "this.i") are in scope everywhere in the class, and cannot be overridden. If they're public, they're in scope in all classes.
- (5) Local variables and parameters are in scope only inside the method that declares them, and only for the topmost stack frame.