CS 4: Lecture 18
Wednesday, March 22, 2006

Binary search

Recall that when a method calls itself recursively, Java’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. Assume i is 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 check the middle element of that half, and so on, cutting the possibilites 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) {
    bsearch(i, 0, i.length - 1, findMe);
  }
```

```
STACK    bsearch    left [4]             |
                   right [4]  findMe [1] |
                     mid [4]       i [.]-+---------\
         --------------------------------|         |
         bsearch    left [4]             |         |
                   right [6]  findMe [1] |         |
                     mid [5]       i [.]-+---------|
         --------------------------------|         |
         bsearch    left [0]             |         |
                   right [6]  findMe [1] |         |
                     mid [3]       i [.]-+---------|
         --------------------------------|         |
         bsearch  findMe [1]       i [.]-+---------|   -------------------
         --------------------------------|         \-->| -3 -2 0 0 1 5 5 |
         main                   args [.]-+->["in"]     ------------------- HEAP
```

How long does binary search take? Suppose the array has n elements. In one call to bsearch, we can eliminate 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.

Stack Size

Most operating systems give a program enough stack space for a few thousand stack frames. If you use a recursive procedure to perform a million-iteration loop, 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 would be very deep.

However, our recursive binary search method does not have this problem. Most modern microprocessors cannot access 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 size of a problem) is safe, whereas recursion to a depth of roughly n is not.

Mutual Recursion

Mutual recursion occurs when two or more methods create a cycle by calling each other.

```
  static long gcd(long a, long b) {
    if (b == 0) {
      return a;
    } else {
      return gcdHelper(a, b);
    }
  }

  static long gcdHelper(long a, long b) {
    return gcd(b, a % b);
  }
```

The gcd method can call the gcdHelper method, which calls gcd again. There’s no reason to write a GCD algorithm this way. But mutual recursion is useful for some tasks, like parsing English sentences. In English, a sentence can have an object, which can be modified by a subordinate clause, which can have an object of its own, and so on.

  Dorothy killed the wicked witch, who flew on a broomstick, which I used to sweep my house, which is painted green.

For this reason, computer programs for parsing often use mutual recursion.

```
  void parseClause(String[] clause) {  |  void parseObject(String[] object) {
    ...                                |    ...
    if (clauseHasObject) {             |    if (objectHasClause) {
      parseObject(object);             |      parseClause(clause);
    }                                  |    }
    ...                                |    ...
  }                                    |  }
```

You can also have a method w that calls x, which calls y, which calls z, which calls x. The bottom line is that recursion is happening if one method has two or more stack frames on the stack at the same time.

The Hardest Midterm Question

What is the output of this program?

```
public class What {
  public long n;

  public void increment() {
    n++;
  }

  public static void reset(What w) {
    w.increment();
    w = new What();
    w.n = 0;
  }

  public static void main(String[] args) {
    What w = new What();
    w.n = 7;
    reset(w);
    System.out.println("The number is " + w.n);
  }
}
```

Just after the line "n++;" executes, memory looks like this.

```
STACK                                         |                            HEAP
                                              |
method call      parameters & local variables |
----------------------------------------------|
                                     ---      |
What.increment                  this |.+---------------------\
                                     ---      |              |
----------------------------------------------|              |
                                     ---      |              |
What.reset                         w |.+---------------\     |
                                     ---      |        |     |
----------------------------------------------|        v     v
                            ---      ---      |       ----------
What.main             args |?|    w |.+------------->|  n | 8 | What object
                            ---      ---      |       ----------
----------------------------------------------|
```

Then, reset() creates a new object and sets its "n" field to zero.

```
----------------------------------------------|
                                     ---      |       ----------
What.reset                         w |.+------------->|  n | 0 |
                                     ---      |       ----------
----------------------------------------------|
                            ---      ---      |       ----------
What.main             args |?|    w |.+------------->|  n | 8 |
                            ---      ---      |       ----------
----------------------------------------------|
```

However, this has no effect on the first "What" object. So the output is 8.