Basic Java Syntax (2)

(and some algorithms as well)

Prof. Chris Jermaine
cmj4@cs.rice.edu

So Far…

- We have seen:
  - For, if, arrays, comments, new, code blocks, method calls, return statements
- What are we missing?
  - While loops
  - Ex: say we want to compute ceil of log (base b) of an int n
  - Simple alg: keep multiplying by b until you reach/exceed n

Writing a While Loop

```
/**
 *  This method computes the log_b (n) in a silly way
 */
public int logBaseB (int b, int n) {
    // accum is always b^pow
    int accum = 1, pow = 0;

    // repeatedly multiply accum by another b until
    // we hit n
    while (accum < n) {
        pow++;
        accum *= b;
    }
    return pow;
}
```

Stuff to Notice

```
/**
 *  This method computes the log_b (n) in a silly way
 */
public int logBaseB (int b, int n) {
    // accum is always b^pow
    int accum = 1, pow = 0;

    // repeatedly multiply accum by another b until
    // we hit n; accum is always b^pow
    while (accum < n) {
        pow++;
        accum *= b;
    }
    return pow;
}
```

Can declare, init. multiple vars/refs at once

This is the classic C notation for “eval. then increment”

C notation for “accum = accum * b”

Stuff to Notice

```
/**
 *  This method computes the log_b (n) in a silly way
 */
public int logBaseB (int b, int n) {
    // accum is always b^pow
    int accum = 1, pow = 0;

    // repeatedly multiply accum by another b until
    // we hit n; accum is always b^pow
    while (accum < n) {
        pow++;
        accum *= b;
    }
    return pow;
}
```

Can declare, init. multiple vars/refs at once

This is the classic C notation for “eval. then increment”

C notation for “accum = accum * b”

ALGORITHMS ALERT!

- Question: can we do better than linear in the return value?

Repeated Squaring

- Every power of two is the product of some subset of $\{2^1, 2^2, 2^4, 2^8, 2^{16}...\}$
  - Ex: $128 = 2^7 = 2^1 \times 2^2 \times 2^4$ (note you can compute this greedily!)
- Want to compute $x = \log_2(n)$
- Say $n = 128$
  - Then $2^{1+2+4} = n$
  - Thus (by definition of log) $\log_2(n) = 7$
- Suggests an algorithm to compute the $\log_b(n)$...
  - Find a set of powers of b that multiply together to get n
  - And where each power is itself a power of 2
  - In other words, form n by multiplying items from the set $\{b^1, b^2, b^4, b^8, b^{16}...\}$

Repeated Squaring (cont'd)

- How to get $\{b^1, b^2, b^4, b^8, b^{16}...\}$
  - — Repeated squaring!
- Then, once you have this, greedily figure out how to compute n
  - — Ex: want $\log_3 4321211$
  - — First compute $\{3^1 = 3, 3^2 = 9, 3^4 = 81, 3^8 = 6561, 3^{16} = 43046721\}$
  - — Then multiply values greedily as long as you don't exceed 4321211

Repeated Squaring (cont'd)

- How to get $\{b^1, b^2, b^4, b^8, b^{16}...\}$
  - — Repeated squaring!
- Then, once you have this, greedily figure out how to compute n
  - — Ex: want $\log_3 4321211$
  - — First compute $\{3^1 = 3, 3^2 = 9, 3^4 = 81, 3^8 = 6561, 3^{16} = 43046721\}$
  - — Then multiply values greedily as long as you don't hit or exceed 4321211
  - — Try 43046721 * 1... total would be 43046721... too big
  - — Try 6561 * 1... OK! total is 6561
  - — Try 81 * 6561... OK! total is 81 * 6561 = 531441
  - — Try 9 * 531441... total would be 9 * 531441 = 4782969... too big
  - — Try 3 * 531441... OK! total is 3 * 531441 = 1594323
  - — So then $\log_3 4321211$ is 8 + 4 + 1 + 1 = 14 (need an extra 1 since we want ceiling)

Implementing This…

```
public int logBaseB (int b, int n) {
    // vals[i] will store b^(2^i)
    int [] vals = new int[30];
    // pows[i] will store 2^i
    int [] pows = new int[30];

    // repeatedly square accum until we exceed n; in this
    // way, accum is always b^(2^i), curPow is always 2^i
    int i = 0, curPow = 1, accum = b;
    while (true) {
        vals[i] = accum;
        pows[i] = curPow;
        if (accum > n)
            break;
        accum *= accum;
        curPow *= 2;
        i++;
    }
```

Implementing This…

```
public int logBaseB (int b, int n) {
    // vals[i] will store b^(2^i)
    int [] vals = new int[30];
    // pows[i] will store 2^i
    int [] pows = new int[30];

    // repeatedly square accum until we exceed n; in this
    // way, accum is always b^(2^i), curPow is always 2^i
    int i = 0, curPow = 1, accum = b;
    while (true) {
        vals[i] = accum;
        pows[i] = curPow;
        if (accum >= n)
            break;
        accum *= accum;
        curPow *= 2;
        i++;
    }
```

Trick so loop never exits here

Tells Java to exit the loop

Implementing This…

```
public int logBaseB (int b, int n) {
    ...
    // now take the powers we have computed and construct n;
    // it will always hold that totSoFar = b^powSoFar
    int totSoFar = 1, powSoFar = 0;
    do {
        if (totSoFar * vals[i] < n) {
            totSoFar *= vals[i];
            powSoFar += pows[i];
        }
        i--;
    } while (i >= 0);

    // we now have the largest power of b that does not
    // reach n, so add one to it and return
    return powSoFar + 1;
}
```

This algorithm takes on the order of log (log n) steps! Fast!

Implementing This…

```
public int logBaseB (int b, int n) {
    ...
    // now take the powers we have computed and construct n;
    // it will always hold that totSoFar = b^powSoFar
    int totSoFar = 1, powSoFar = 0;
    do {
        if (totSoFar * vals[i] < n) {
            totSoFar *= vals[i];
            powSoFar += pows[i];
        }
        i--;
    } while (i >= 0);

    // we now have the largest power of b that does not
    // reach n, so add one to it and return
    return powSoFar + 1;
}
```

This is a “do-while” loop; stoppage check is executed *after* the body of the loop

What would this look like with a *for* loop instead?

Many Ways To Write Same Loop!

```
public int logBaseB (int b, int n) {
    ...
    // now take the powers we have computed and construct n;
    // it will always hold that totSoFar = b^powSoFar
    int totSoFar = 1, powSoFar = 0;
    for (; i >= 0; i--) {
        // don't use this factor if we'd meet/exceed n
        if (totSoFar * vals[i] >= n)
            continue;
        totSoFar *= vals[i];
        powSoFar += pows[i];
    }

    // we now have the largest power of b that does not
    // reach n, so add one to it and return
    return powSoFar + 1;
}
```

Many Ways To Write Same Loop!

```
public int logBaseB (int b, int n) {
    ...
    // now take the powers we have computed and construct n;
    // it will always hold that totSoFar = b^powSoFar
    int totSoFar = 1, powSoFar = 0;
    for (; i >= 0; i--) {
        // don't use this factor if we'd meet/exceed n
        if (totSoFar * vals[i] >= n)
            continue;
        totSoFar *= vals[i];
        powSoFar += pows[i];
    }

    // we now have the largest power of b that does not
    // reach n, so add one to it and return
    return powSoFar + 1;
}
```

Note use of blank initialization statement

"continue" means to skip rest of loop body

OK, What Have We Missed?

- "Switch"
  - — Generally pretty useless, in my opinion, so won't talk about it much
  - — Look it up in the book
  - — In a nutshell: alternative to specific form of "if":

```
if (a == 1) {
    // do something
} else if (a == 2) {
    // do something else
} else if (a == 3) {
    // do another thing
} else {
    // do this
}
```

  - — Switch is a bit more efficient, but easier to screw up
  - — I try to avoid it...

About Does It for Control Flow, Basic Syntax