

CS61B Lecture #2

- Please make sure you have obtained an account and run 'register' by *Friday*.
- Pick up readers at Vick Copy (there are three).
- This week's lab is due Tuesday at 9:00AM.
- *No discussion sections* this week.
- Please read through chapter 4 of *Head First Java* by Friday. Also, please look through Chapter 2 of *Assorted Materials on Java* reader.

Thinking Recursively

Understand and check `isDivisible(13,2)` by *tracing one level*.

```
/** True iff X is divisible by
 * some number >=K and < X,
 * given K > 1. */
boolean isDivisible (int x, int k) {
    if (k >= x)
        return false;
    else if (x % k == 0)
        return true;
    else
        return isDivisible (x, k+1);
}
```

- Call assigns $x=13$, $k=2$
- Body has form 'if ($k \geq x$) S_1 else S_2 '.
- Since $2 < 13$, we evaluate the first else.
- Check if $13 \bmod 2 = 0$; it's not.
- Left with `isDivisible(13,3)`.
- Rather than tracing it, instead use the *comment*:
- Since 13 is *not* divisible by any integer in the range 3..12 (and $3 > 1$), `isDivisible(13,3)` must be *false*, and we're done!
- Sounds like that last step begs the question. Why doesn't it?

Lesson: Comments aid understanding. Make them *count*!

Iteration

- `isDivisible` is *tail recursive*, and so creates an *iterative process*.
- Traditional “Algol family” production languages have special syntax for iteration. Four equivalent versions of `isDivisible`:

```
if (k >= x)                while (k < x) { // ! (k >= x)
    return false;          if (x % k == 0)
else if (x % k == 0)       return true;
    return true;          k = k+1;
else                        // or k += 1, or k++ (yuch).
    return isDivisible (x, k+1);}
                           return false;
```

```
int k1 = k;                for (int k1 = k; k1 < x; k1 += 1) {
while (k1 < x) {           if (x % k1 == 0)
    if (x % k1 == 0)       return true;
        return true;
    k1 += 1;
}
return false;             return false;
```

More Iteration: Sort an Array

Problem. Print out the command-line arguments in order:

```
% java sort the quick brown fox jumped over the lazy dog  
brown dog fox jumped lazy over quick the the
```

Plan.

```
class sort {  
    public static void main (String[] words) {  
        sort (words, 0, words.length-1);  
        print (words);  
    }  
  
    /** Sort items A[L..U], with all others unchanged. */  
    static void sort (String[] A, int L, int U) { /* TOMORROW */ }  
  
    /** Print A on one line, separated by blanks. */  
    static void print (String[] A) { /* TOMORROW */ }  
}
```

Selection Sort

```
/** Sort items A[L..U], with all others unchanged. */
static void sort (String[] A, int L, int U) {
    if (L < U) {
        int k = /*( Value, p, s.t. A[p] is largest in A[L], ..., A[U] )*/;
        /*{ swap A[k] with A[U] }*/;
        /*{ Sort items L to U-1 of A. }*/;
    }
}
```

Selection Sort

```
/** Sort items A[L..U], with all others unchanged. */
static void sort (String[] A, int L, int U) {
    if (L < U) {
        int k = indexOfLargest (A, L, U);
        /*{ swap A[k] with A[U] }*/;
        /*{ Sort items L to U-1 of A. }*/;
    }
}
```

Selection Sort

```
/** Sort items A[L..U], with all others unchanged. */
static void sort (String[] A, int L, int U) {
    if (L < U) {
        int k = indexOfLargest (A, L, U);
        /*{ swap A[k] with A[U] }*/;
        sort (A, L, U-1);      // Sort items L to U-1 of A
    }
}
```

Selection Sort

```
/** Sort items A[L..U], with all others unchanged. */
static void sort (String[] A, int L, int U) {
    if (L < U) {
        int k = indexOfLargest (A, L, U);
        String tmp = A[k];  A[k] = A[U]; A[U] = tmp;
        sort (A, L, U-1);    // Sort items L to U-1 of A
    }
}
```


Selection Sort

```
/** Sort items A[L..U], with all others unchanged. */
static void sort (String[] A, int L, int U) {
    if (L < U) {
        int k = indexOfLargest (A, L, U);
        String tmp = A[k];  A[k] = A[U]; A[U] = tmp;
        sort (A, L, U-1);    // Sort items L to U-1 of A
    }
}
```

Iterative version:

```
while (L < U) {
    int k = indexOfLargest (A, L, U);
    String tmp = A[k];  A[k] = A[U]; A[U] = tmp;
    U -= 1;
}
```

And we're done! Well, OK, not quite.

Really Find Largest

```
/** Value k, I0<=k<=I1, such that V[k] is largest element among
 * V[I0], ... V[I1]. Requires I0<=I1. */
static int indexOfLargest (String[] V, int i0, int i1) {
    if (i0 >= i1)
        return i1;
    else /* if (i0 < i1) */ {
        int k = indexOfLargest (V, i0+1, i1);
        return (V[i0].compareTo (V[k]) > 0) ? i0 : k;
        // or if (V[i0].compareTo (V[k]) > 0) return i0; else return k;
    }
}
```

Iterative:

```
int i, k;
k = i1;    // Deepest iteration
for (i = i1-1; i >= i0; i -= 1)
    k = (V[i].compareTo (V[k]) > 0) ? i : k;
return k;
```

Finally, Printing

```
/** Print A on one line, separated by blanks. */
static void print (String[] A) {
    for (int i = 0; i < A.length; i += 1)
        System.out.print (A[i] + " ");
    System.out.println ();
}
```

```
/* Looking ahead: There's a brand-new syntax for the for
 * loop here (as of J2SE 5): */
    for (String s : A)
        System.out.print (s + " ");
/* Use it if you like, but let's not stress over it yet! */
```