Recursion

Recursion Basics

- It's possible for code in the body of a method to call other methods.
  - eg. in `Quadratic`, `root1` calls `discrim`.
- There's no reason a method can't call itself.
  - eg. from within a method `myFunc`, make a call to `myFunc`.
  - A function that calls itself is called "recursive".
- Each function call has separate parameters and local variables.

Why?

- There are many problems that are easy to solve with recursive algorithms.
- A problem that can be solved in pieces is a candidate for a recursive algorithm.
  - Chop the problem into one or more smaller parts.
  - Recursively solve each part.
  - Combine for the whole solution.
- eg. reversing a string: reverse characters 1…end; result is that + character 0.

Example

```
class StringRev {
    public static String reverse(String s) {
        if ( s.length() == 0 ) {
            return s;
        } else {
            return reverse(s.substring(1)) + s.charAt(0);
        }
    }
    public static void main(String[] args) {
        System.out.println( reverse("CMPT") );
    }
}
```

- Output:

```
TPMC
```

Huh?

```
reverse("CMPT")   returns   reverse("MPT") + 'C'
reverse("MPT")    returns   reverse("PT") + 'M'
reverse("PT")     returns   reverse("T") + 'P'
reverse("T")      returns   reverse("") + 'T'
reverse("")       returns   ""
```

Oh!

"TPMC"

reverse("CMPT") returns ~~reverse("MPT")~~ **"TPM"** + 'C'

reverse("MPT") returns ~~reverse("PT")~~ **"TP"** + 'M'

reverse("PT") returns ~~reverse("T")~~ **"T"** + 'P'

reverse("T") returns ~~reverse("")~~ "" + 'T'

reverse("") returns ""

Understanding Recursion

- …but you probably shouldn't worry too much about those details.
- When trying to understand a recursive algorithm,
  - assume the recursive call(s) return the right thing;
  - look at how that result is used to build the whole result.

Creating Recursion

- The idea:
  1. take the original problem,
  2. find smaller subproblem(s),
  3. solve subproblem(s) recursively,
  4. combine for a full solution.

1. `"CMPT"`
2. chars 1… `== "MPT"`
3. `reverse("MPT") == "TPM"`
4. `"TPM" + 'C' == "TPMC"`

- Find this structure in the problem, and the recursion is (almost) done.
  - Again, don't worry about the recursive call details.
- Important words: "smaller subproblem"

“Smaller”

- If the subproblem isn't strictly smaller, you end up with infinite recursion:
- The recursive step **must** decrease the problem size.

```
reverse("CMPT")
    ↓
reverse("CMPT")
    ↓
reverse("CMPT")
    ↓
reverse("CMPT")
    ↓
reverse("CMPT")
    ↓
reverse("CMPT")
    ↓
reverse("CMPT")
    ↓
```

“Subproblem”

- You must be able to split the problem to make a recursive call.
  - If the subproblems are always getting smaller, this can’t continue forever. (good: program will stop)
  - Eventually, we can’t split any further: `reverse("")`
- This is when the recursion stops
  - the “base case”

Base Case

- There are typically one or two tiny cases of the problem that can't be split for recursion
  - The answer in these cases should be obvious.
  - eg. `reverse("") == ""`
- These are handled as special cases in the recursive function
  - if (base case): return the result.
  - else: do recursive thing.
- Every input **must** end with a base case.

More Examples

- Calculate x^y for integer y, $y \geq 0$
- factorial ($n!$)
- find prime factorization
- insert spaces between characters in a string
- sum an array
- linear search an array (later)
- sort an array (later)

Recursive Algorithms

- There could be several ways to "split" to make a recursive call.
  - … or other non-recursive ways to solve a problem.
- This can have a big effect on efficiency.
- eg. calculating powers…

Powers, version 1

- subproblem: $x^y \rightarrow x \cdot x^{y\text{-}1}$
- base case: $x^0 \rightarrow 1$

```
public static long pow(long x, long y) {
    // return x to the power of y
    if ( y == 0 ) {
        return 1;
    } else {
        return x * pow(x,y-1);
    }
}
```

Powers, version 2

- subproblem: $x^y \rightarrow x^{y/2} \cdot x^{y/2}$, base case: $x^0 \rightarrow 1$

```
public static long pow(long x, long y) {
    long half;
    if ( y == 0 ) {
        return 1;
    } else if ( y%2 == 0 ) {      // y even
        half = pow(x, y/2);
        return half*half;
    } else {                      // y odd
        half = pow(x, (y-1)/2);
        return half*half*x;
    }
}
```

Comparison

- `Power1` takes y recursive steps to calculate x^y.
- `Power2` takes about $\log_2(y)$ steps.
  - much faster as y gets large
- Running time comparison:

	`Power1`	`Power2`
`pow(1,` 10^6`)`	0.7 s	0.2 s
`pow(1,` 10^7`)`	5.0 s	0.2 s
`pow(1,` 10^8`)`	out of memory after 2 min	0.2 s
`pow(1,` 10^9`)`	didn't attempt	0.2 s