From Recursion to Self Reference

```java
public int calc(int n){
    return n*calc(n-1);
}
```

- **What is the Internet?**
  - A network of networks.
  - Or ...

- **What is recursive DNS?**
  - What IP is fxyzt1.com?

- **What is a linked list?**
  - Where is it used?

Quota Exceeded: coping with storage

- **You’re running out of disk space**
  - Buy more
  - Compress files
  - Delete files

- **How do you find your “big” files?**
  - What’s big?
  - How do you do this?

BlobCount or edge detection or …

- **How do we find images? Components? Paths?**
  - Create information from data

Tools: Solving Computational Problems

- **Algorithmic techniques and paradigms**
  - Brute-force/exhaustive, greedy algorithms, dynamic programming, divide-and-conquer, ...
  - Transcend a particular language
  - Designing algorithms, may change when turned into code

- **Programming techniques and paradigms**
  - Recursion, memo-izing, compute-once/lookup, tables, ...
  - Transcend a particular language
  - Help in making code work
    - Cope with correctness and maintenance
    - Cope with performance problems
Tools: Solving Computational Problems

- **Java techniques**
  - java.util.*, Comparator, Priority Queue, Map, Set, ...
  - These aren’t really Java-specific, but realized in Java
  - Map, Comparator, Set: C++, Python, ....
  - We learn idioms in a language and talk about abstractions

- **Analysis of algorithms and code**
  - Mathematical analysis, empirical analysis
  - We need a language and techniques for discussion
  - Theory and practice, real problems and in-the-limit issues

- “In theory there is no difference between theory and practice, but in practice there is.”
  (attributed to many)

Recursive structure matches code

```java
public static long THRESHOLD = 1000000L; // one million bytes

public static void findBig(File dir, String tab) {
    File[] dirContents = dir.listFiles();
    System.out.println(tab + "**:" + dir.getPath());
    for (File f : dirContents) {
        if (f.isDirectory()) {
            findBig(f, tab + "\t" );
        } else {
            if (f.length() > THRESHOLD) {
                System.out.println("%s%s%8d", tab, f.getName(), f.length());
            }
        }
    }
}
```

Solving Problems Recursively

- **Recursion: indispensable in programmer’s toolkit**
  - Allows many complex problems to be solved simply
  - Elegance and understanding in code often leads to better programs: easier to modify, extend, verify (and sometimes more efficient!!)
  - Sometimes recursion isn’t appropriate, when it’s bad it can be very bad—every tool requires knowledge and experience in how to use it

- The basic idea is to get help solving a problem from coworkers (clones) who work and act like you do
  - Ask clone to solve a simpler but similar problem
  - Use clone’s result to put together your answer

- Need both: call on the clone and use the result

Print words read, but print backwards

- **Could store words and print in reverse order, but ...**
  - Probably the best approach, recursion works too

```java
public void printReversed(Scanner s) {
    if (s.hasNext()) {
        String word = s.next();
        printReversed(s);
        System.out.println(word);
    }
}
```

- The function printReversed reads a word, only prints word after the clones finish printing
  - Each clone has own version of the code, own word variable
  - Who keeps track of the clones?
  - How many words are created when reading N words?
    - Can we do better?
Exponentiation

- Computing $x^n$ means multiplying $n$ numbers
  - Does it require $n$ multiplies?
  - What’s the simplest value of $n$ when computing $x^n$?
  - If you want to multiply only once, what can you ask a clone?

```java
public static double power(double x, int n){
    if (n == 0){
        return 1.0;
    }
    return x * power(x, n-1);
}
```

- Number of multiplications?
  - Note base case: no recursion, no clones
  - Note recursive call: moves toward base case (unless …)

Faster exponentiation

- Recursive calls made to compute $2^{1024}$?
  - How many multiplies on each call? Is this better?

```java
public static double power(double x, int n){
    if (n == 0) return 1.0;
    double semi = power(x, n/2);
    if (n % 2 == 0) return semi*semi;
    return x * semi * semi;
}
```

- What about an iterative version of this function?
  - Why might we want such a version?

Back to Recursion

- Recursive functions have two key attributes
  - There is a base case, aka exit case: no recursion!
    - See print reversed, exponentiation
  - All other cases make a recursive call, with some measure (e.g., parameter value) that decreases towards the base case
    - Ensure that sequence of calls eventually reaches the base case
    - “Measure” can be tricky, but usually it’s straightforward

- Example: finding large files in a directory (on a hard disk)
  - Why is this inherently recursive?
  - How is this different from exponentiation?

Thinking recursively: recursive Max

```java
public static double recMax(double[] a, int index){
    if (index == a.length-1){ // last element, done
        return a[index];
    }
    double maxAfter = recMax(a,index+1);
    return Math.max(a[index],maxAfter);
}
```

- What is base case (conceptually)?
  - Do we need variable `maxAfter`?

- Use `recMax` to implement `arrayMax` as follows:
  - Introduce auxiliary variable from `arrayMax(a)` to …
  - return `recMax(a,0)`; [Recursive methods sometimes use extra parameters; helper methods set this up]
Recognizing recursion:

```java
public void change(String[] a, int first, int last) {
    if (first < last) {
        String temp = a[first]; // swap first/last
        a[first] = a[last];
        a[last] = temp;
        change(a, first+1, last-1);
    }
}
// original call (why?): change(a, 0, a.length-1);
```

- What is base case? (no recursive calls)
- What happens before recursive call made?
- How is recursive call closer to the base case?

Recursive methods sometimes use extra parameters; helper methods set this up

More recursion recognition

```java
public static int value(int[] a, int index) {
    if (index < a.length) {
        return a[index] + value(a, index+1);
    } else {
        return 0;
    }
}
// original call:  int v = value(a, 0);
```

- What is base case, what value is returned?
- How is progress towards base case realized?
- How is recursive value used to return a value?
- What if a is an array of doubles, does anything change?

Recursive methods sometimes use extra parameters; helper methods set this up

From programming techniques to Java

- Is recursion a language independent concept?
  - Do all languages support recursion?
  - What are the issues in terms of computer/compiler/runtime support?

- We use a language and its libraries, study them?
  - Should we know how to implement ArrayList
    - What are the building blocks, what are our tools
  - Should we know how to implement different sorts
    - Should we know how to call existing sorts

Fran Allen

- IBM Fellow, Turing Award
  - Optimizing compilers

- Taught high school for two years, then Master’s degree and IBM
  - Teachers excited me to learn

I’ve always felt that theory without practice is maybe nice and maybe pretty, but it’s not going to influence computing as much as the practice side. But the practice has to be backed up with the ability to talk about it, reason about it, and formulate it so that it can be reproduced.
Blob Counting, Flood Fill

- Flood a region with color
  - Erase region, make transparent, ..
  - How do find the region?

- Finding regions, blobs, edges, ..
  - See blob counting code
  - What is a blob?

- Recursion helps, but necessary?
  - Performance, clarity, ...
  - Ease of development

Analyzing Algorithms

- Consider two solutions to SortByFreqs
  - Sort, then scan looking for changes
  - Insert into Set, then count each unique string
  - Use a Map (TreeMap or HashMap)

- What about iterative flood fill?
  - Is it faster in theory? In practice?

- We want to discuss trade-offs of these solutions
  - Ease to develop, debug, verify
  - Runtime efficiency
  - Vocabulary for discussion

More on O-notation, big-Oh

- Big-Oh hides/obscures some empirical analysis, but is good for general description of algorithm
  - Allows us to compare algorithms in the limit
    - 20N hours vs N² microseconds: which is better?

- O-notation is an upper-bound, this means that \( N \) is \( O(N) \), but it is also \( O(N^2) \) we try to provide tight bounds. Formally:
  - A function \( g(N) \) is \( O(f(N)) \) if there exist constants \( c \) and \( n \) such that \( g(N) < cf(N) \) for all \( N > n \)

<table>
<thead>
<tr>
<th>( N )</th>
<th>( O(log N) )</th>
<th>( O(N) )</th>
<th>( O(N log N) )</th>
<th>( O(N^2) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.000003</td>
<td>0.00001</td>
<td>0.000033</td>
<td>0.0001</td>
</tr>
<tr>
<td>100</td>
<td>0.000007</td>
<td>0.00010</td>
<td>0.000664</td>
<td>0.1000</td>
</tr>
<tr>
<td>1,000</td>
<td>0.000010</td>
<td>0.00100</td>
<td>0.010000</td>
<td>1.0</td>
</tr>
<tr>
<td>10,000</td>
<td>0.000013</td>
<td>0.01000</td>
<td>0.132900</td>
<td>1.7 min</td>
</tr>
<tr>
<td>100,000</td>
<td>0.000017</td>
<td>0.10000</td>
<td>1.661000</td>
<td>2.78 hr</td>
</tr>
<tr>
<td>1,000,000</td>
<td>0.000020</td>
<td>1.0</td>
<td>19.9</td>
<td>11.6 day</td>
</tr>
<tr>
<td>1,000,000,000</td>
<td>0.000030</td>
<td>16.7 min</td>
<td>18.3 hr</td>
<td>318 centuries</td>
</tr>
</tbody>
</table>