Solving Problems Recursively

- Recursion is an indispensable tool in a 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 concepts: call on the clone and use the result

Print words entered, but backwards

- Can use an ArrayList, store all the words and print in reverse order
  - Probably the best approach, recursion works too

```
public void printReversed(Scanner s){
     if (s.hasNext()){             // reading succeeded?
         String word = s.next(); // store word
         printReversed(s);       // print rest
        System.out.println(word);// print the word
     }
}
```

- The function `printReversed` reads a word, prints the word only after the clones finish printing in reverse order
  - 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?
    - What about when ArrayList<String> used?

Exponentiation

- Computing x^n means multiplying n numbers (or does it?)
  - What's the easiest value of n to compute x^n?
  - If you want to multiply only once, what can you ask a clone?

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

- What about an iterative version?

Faster exponentiation

- How many recursive calls are made to computer 2^{1024}?
  - How many multiplies on each call? Is this better?

```
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?

Back to Recursion

- Recursive functions have two key attributes
  - There is a *base case*, sometimes called the *exit case*, which does <u>not</u> make a recursive call
    - See print reversed, exponentiation
  - All other cases make a recursive call, with some parameter or other measure that decreases or moves towards the base case
    - Ensure that sequence of calls eventually reaches the base case
    - "Measure" can be tricky, but usually it's straightforward

- Example: sequential search in an array
  - If first element is search key, done and return
  - Otherwise look in the "rest of the array"
  - How can we recurse on "rest of array"?

Thinking recursively

- Problem: find the largest element in an array
  - Iteratively: loop, remember largest seen so far
  - Recursive: find largest in [1..n), then compare to 0th element

```
public static double max(double[] a){
    double maxSoFar = a[0];
    for(int k=1; k < a.length; k++) {
        maxSoFar = Math.max(maxSoFar,a[k]);
    }
    return maxSoFar;
}
```

  - In a recursive version what is base case, what is measure of problem size that decreases (towards base case)?

Recursive Max

```
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`?
- We can use `recMax` to implement `arrayMax` as follows

```
return recMax(a,0);
```

Recognizing recursion:

```
public static void change(String[] a, int first, int last){
    if (first < last) {
        String temp = a[first];  // swap a[first], a[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?

More recursion recognition

```
public static int value(int[] a, int index){
    if (index < a.length) {
        return a[index] + value(a,index+1);
    }
    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 array of doubles, does anything change?

Analysis: Algorithms and Data Structures

- We need a vocabulary to discuss performance and to reason about alternative algorithms and implementations
  - It's faster! It's more elegant! It's safer! It's cooler!

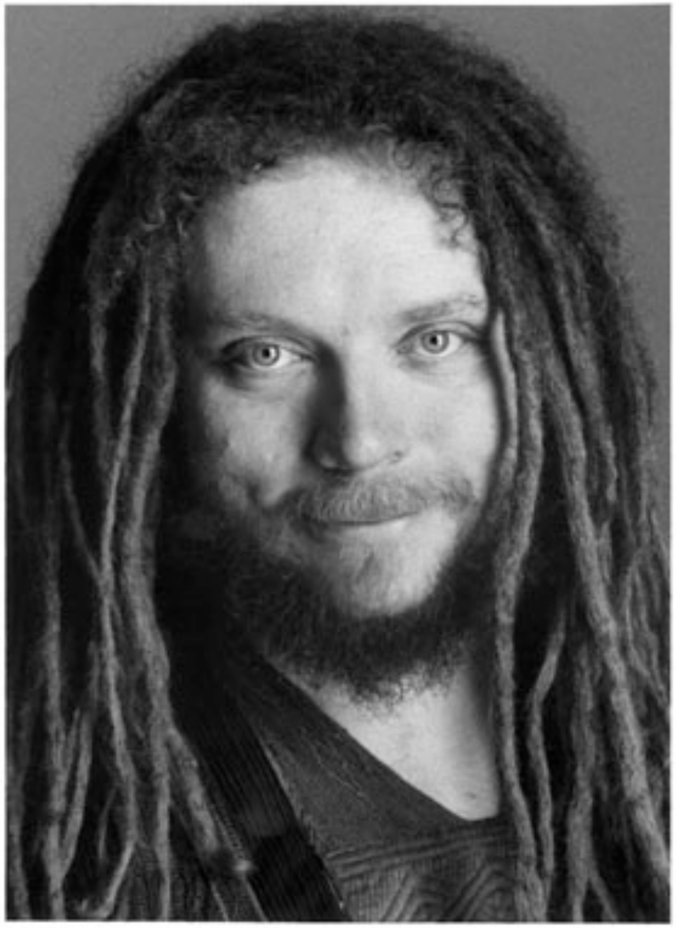
- We need empirical tests and analytical/mathematical tools
  - Given two methods, which is better? Run them to check.
    - 30 seconds vs. 3 seconds, easy. 5 hours vs. 2 minutes, harder
    - What if it takes two weeks to implement the methods?
  - Use mathematics to analyze the *algorithm*,
  - The implementation is another matter, cache, compiler optimizations, OS, memory,…

Jaron Lanier (http://www.advanced.org/jaron)

Jaron Lanier is a computer scientist, composer, visual artist, and author. He coined the term 'Virtual Reality' … he co-developed the first implementations of virtual reality applications in surgical simulation, vehicle interior prototyping, virtual sets for television production, and assorted other areas

"What's the difference between a bug and a variation or an imperfection? If you think about it, if you make a small change to a program, it can result in an enormous change in what the program does. If nature worked that way, the universe would crash all the time."

Lanier has no academic degrees



Recursion and recurrences

- Why are some functions written recursively?
  - Simpler to understand, but ...
  - Mt. Everest reasons
- Are there reasons to prefer iteration?
  - Better optimizer: programmer/scientist v. compiler
  - Six of one? Or serious differences
    - "One person's meat is another person's poison"
    - "To each his own", "Chacun a son gout", ...
- Complexity (big-Oh) for iterative and recursive functions
  - How to determine, estimate, intuit

What’s the complexity of this code?

```
// first and last are integer indexes, list is List
int lastIndex = first;
Comparable pivot = list.get(first);
for(int k=first+1; k <= last; k++){
    Comparable ko = list.get(k);
    if (ko.compareTo(pivot) <= 0){
        lastIndex++;
        Collections.swap(list,lastIndex,k);
    }
}
```

- What is big-Oh cost of a loop that visits *n* elements of a vector?
  - Depends on loop body, if body `O(1)` then …
  - If body is `O(n)` then …
  - If body is `O(k)` for k in [0..n) then …

FastFinder.findHelper

```
private Object findHelper(ArrayList<Comparable> list,
                          int first, int last, int kindex){
    int lastIndex = first;
    Comparable pivot = list.get(first);
    for(int k=first+1; k <= last; k++){
        Comparable ko = list.get(k);
        if (ko.compareTo(pivot) <= 0){
            lastIndex++;
            Collections.swap(list,lastIndex,k);
        }
    }
    Collections.swap(list,lastIndex,first);
    if (lastIndex == kindex) return list.get(lastIndex);
    if (kindex < lastIndex)
        return findHelper(list,first,lastIndex-1,kindex);
    return findHelper(list,lastIndex+1,last,kindex);
}
```

Different measures of complexity

- Worst case
  - Gives a good upper-bound on behavior
  - Never get worse than this
  - Drawbacks?

- Average case
  - What does average mean?
  - Averaged over all inputs? Assuming uniformly distributed random data?
  - Drawbacks?

- Best case
  - Linear search, useful?

Multiplying and adding big-Oh

- Suppose we do a linear search then we do another one
  - What is the complexity?
  - If we do 100 linear searches?
  - If we do n searches on a vector of size n?

- What if we do binary search followed by linear search?
  - What are big-Oh complexities? Sum?
  - What about 50 binary searches? What about n searches?

- What is the number of elements in the list (1,2,2,3,3,3)?
  - What about (1,2,2, …, n,n,…,n)?
  - How can we reason about this?

Helpful formulae

- We always mean base 2 unless otherwise stated
  - What is log(1024)?
  - $\log(xy)$ $\log(x^y)$ $\log(2^n)$ $2^{(\log n)}$
    - $\log(x) + \log(y)$
    - $y \log(x)$
    - $n\log(2) = n$
    - $2^{(\log n)} = n$
- Sums (also, use sigma notation when possible)
  - $1 + 2 + 4 + 8 + \ldots + 2^k = 2^{k+1} - 1 = \sum_{i=0}^{k} 2^i$
  - $1 + 2 + 3 + \ldots + n = n(n+1)/2 = \sum_{i=1}^{n} i$
  - $a + ar + ar^2 + \ldots + ar^{n-1} = a(r^n - 1)/(r-1) = \sum_{i=0}^{n-1} ar^i$

Recursion Review

- Recursive functions have two key attributes
  - There is a *base case*, sometimes called the *exit case*, which does not make a recursive call
  - All other cases make recursive call(s), the results of these calls are used to return a value when necessary
    - Ensure that every sequence of calls reaches base case
    - Some measure decreases/moves towards base case
    - “Measure” can be tricky, but usually it’s straightforward

- Example: sequential search in an ArrayList
  - If first element is search key, done and return
  - Otherwise look in the “rest of the list”
  - How can we recurse on “rest of list”?

Sequential search revisited

- What is complexity of sequential search? Of code below?

```
boolean search(ArrayList<Object> list,
               int first, Object target) {
    if (first >= list.size()) return false;
    else if (list.get(first).equals(target))
         return true;
    else return search(list,first+1,target);
}
```

- Why are there three parameters? Same name good idea?

```
boolean search(ArrayList list, Object target){
  return search(list,0,target);
}
```

Why we study recurrences/complexity?

- **Tools to analyze algorithms**
- **Machine-independent measuring methods**
- **Familiarity with good data structures/algorithms**

- **What is CS person: programmer, scientist, engineer?**

  scientists build to learn, engineers learn to build

- **Mathematics is a notation that helps in thinking, discussion, programming**

Recurrences

- Summing Numbers

```
int sum(int n)
{
    if (0 == n) return 0;
    else return n + sum(n-1);
}
```

- What is complexity? justification?
- T(n) = time to compute sum for n

```
T(n) = T(n-1) + 1
T(0) = 1
```

- instead of 1, use O(1) for *constant time*
  - independent of n, the measure of problem size

Solving recurrence relations

- plug, simplify, reduce, guess, verify?

```
T(n) = T(n-1) + 1
T(0) = 1
                    T(n-1) = T(n-1-1) + 1

T(n) = [T(n-2) + 1] + 1 = T(n-2)+2
                    T(n-2) = T(n-2-1) + 1

T(n) = [(T(n-3) + 1) + 1] + 1 = T(n-3)+3

T(n) = T(n-k) + k
```

find the pattern!

```
Now, let k=n, then T(n) = T(0)+n = 1+n
```

- get to base case, solve the recurrence: O(n)

Complexity Practice

- What is complexity of *Build*? (what does it do?)

```
ArrayList<Integer> build(int n)
{
    if (0 == n) return new ArrayList<Integer>(); // empty
    ArrayList<Integer> list = build(n-1);
    for(int k=0;k < n; k++){
        list.add(n);
    }
    return list;
}
```

- Write an expression for T(n) and for T(0), solve.

Recognizing Recurrences

- Solve once, re-use in new contexts
  - T must be explicitly identified
  - n must be some measure of size of input/parameter
    - T(n) is the time for quicksort to run on an n-element vector

`T(n) = T(n/2) + O(1)`	binary search	`O( log n )`
`T(n) = T(n-1) + O(1)`	sequential search	`O( n )`
`T(n) = 2T(n/2) + O(1)`	tree traversal	`O( n )`
`T(n) = 2T(n/2) + O(n)`	quicksort	`O(n log n)`
`T(n) = T(n-1) + O(n)`	selection sort	O(n^2)

- Remember the algorithm, re-derive complexity

Eugene (Gene) Myers

- Lead computer scientist/software engineer at Celera Genomics (now at Berkeley,now at ...?)

- "What really astounds me is the architecture of life. The system is extremely complex. It's like it was designed." ... "There's a huge intelligence there."