Recursion

Solving problems by recursion

- How can you solve a complex problem?
  - Devise a complex solution
  - Break the complex problem into simpler problems

- Sometimes, the simpler problem is similar to (but smaller than) the original problem
  - Example: factorials
    - It’s hard to calculate 24! directly, but we could just calculate 23! and multiply by 24…

- This technique is called recursion
  - Often involves a method calling itself!
  - May call itself one or more times…
  - Must have an ending point!

Dictionary Search

- Want to look up a word in the dictionary
  - Could start at beginning until I find the word
  - Could look in the middle, and then decide which half to look in

- First approach is called a linear search
- Second approach is called a binary search:

```java
search( dictionary, myWord ) {
  if the word in the middle is myWord then done
  else if ( myWord < word )
    search( first half of dictionary, myWord )
  else
    search( second half of dictionary, myWord )
}
```

Efficiency

- How much does it ‘cost’ to find the word using each algorithm?
- We can think of cost as the number of operations required to perform an algorithm.
  - Linear search – Dictionary size / 2 on average
  - Binary search – log₂ ( Dictionary Size )

- We can use Big ‘O’ notation describe the ‘order’ of an algorithm
  - Linear search – O( n ) ‘order n’
  - Binary search – O( log₂ n ) ‘order log n’

Four Questions for Constructing Recursive Solutions

- How can you define the problem in terms of a smaller problem of the same type?
- How does each recursive call diminish the size of the problem?
- What instance of the problem can serve as the base case?
- As the problem size diminishes, will you reach the base case?

Factorials by recursion

- Easy way to calculate n!
  - n! = 1 if n == 1
  - Otherwise, n! = n * (n-1)!

- There are two important things here
  - Each step reduces the size of the problem
  - There is a definite stopping point
    - This point must be reached!

- For this example, there are more efficient ways to do this problem…

```java
factorial(n) {
  if n == 0 or n == 1 then return 1
  else return n * factorial(n-1)
}
```
**Some 'facts' about rabbits**

- Rabbits never die
- A rabbit can start reproducing at the beginning of its third month of life
- Rabbits are always born in male/female pairs. At the beginning of each month, every mature female rabbit gives birth to one male/female pair of rabbits.
- So, if we start with 1 pair of newborn rabbits:
  - Month 1: 1 pair
  - Month 2: 1 pair
  - Month 3: 2 pairs – the original pair gave birth
  - Month 4: 3 pairs – the original pair gave birth again
  - Month 5: 5 pairs – the original pair and its first offspring both gave birth
  - Month 6: 8 pairs – the 3 pairs alive in Month 4 gave birth

**Fibonacci numbers**

- Definition
  - $F_0 = 0$
  - $F_1 = 1$
  - $F_n = F_{n-1} + F_{n-2}$
- How can we solve this recursively?
  - Calculate fibonacci(n-1)
  - Add the two together
- This works!
  - Problem reduced to smaller problem
  - Definite stopping point (always reached)
  - Again, there are more efficient ways to do this...

---

**How can a function ‘exist’ many times?**

- Example: fibonacci(n) calls fibonacci(…) twice
  - How does the computer keep track of this?
  - Solution: keep information on a stack
  - Done automatically by the compiler / run-time system
  - Stack grows in length as needed
  - Each method's local information is kept in an activation record of method that called this one
  - Activation record cleaned up after method returns

---

**Solving Towers of Hanoi with recursion**

- Two cases
  - Tower has exactly one disk
    - Move it from source to destination
  - Tower has more than one disk
    - Move the top $n-1$ disks to the holding peg
    - Move the bottom disk
    - Move the top $n-1$ disks from holding peg to destination
- Important!
  - Holding peg is same at start and end of each move
  - Top disk on holding peg is larger than any disk in tower being moved

---

**Towers of Hanoi**

- Problem: move “tower” from one peg to another
  - May only move one disk at a time
  - May only place a disk on top of a larger disk
  - May use (currently) empty peg as a “holding area”
  - Solve the problem for an n-disk tower by
    - Moving the n-1 top disks from left to middle peg
    - Moving the bottom (largest) disk from left to right peg
    - Moving the n-1 tower from middle to right peg
  - Can we do this?

---

**What else can we use recursion for?**

- Generic problem type: divide-and-conquer
  - Break a problem into 2 (or more) smaller pieces
  - Each piece is similar to the original problem, but smaller
  - Solve each piece individually
  - Combine the results
  - Divide-and-conquer is best implemented with recursion
  - Relatively small code
  - Low code complexity: let the system stack handle all of the details
  - May be easy to convert some forms of recursion, though…
Converting tail recursion into a loop

- Factorials can be solved with recursion
  - $N! = (N-1)! \times N$
  - This is recursive, but can be rewritten as $N \times (N-1)!$
- The last thing the method does is call itself!
  - This is called tail recursion
  - Tail recursion can (usually) be converted into a loop easily
  - By the time the tail recursion occurs, none of the variables in the method are needed
  - This means they can be overwritten!
  - Rather than calling the method recursively, simply return to the top with a new "parameter"

Example: tail recursion for factorials

- Multiplications are done in the order 2, 3, 4, ...
  - Calls are made in the reverse order
  - Multiplication comes after the call
  - "Straighten out" the recursion to make it more efficient

Using recursion

- Many recursive routines may be simplified, but...
- Many cannot be made simpler
  - It’s very hard to do Towers of Hanoi without recursion
  - Doing Towers of Hanoi non-recursively requires several stacks—why not let the system manage them?
- General rule of thumb:
  - If it’s obvious how to do it non-recursively, do it that way
  - If recursion makes programming easier, take advantage of it
- Example: Fibonacci sequence
  - This can be "easily" straightened by simply keeping track of the previous two Fibonacci numbers
  - This is a (very) simple stack, and easy to implement
- Many sorting and tree algorithms use recursion for ease of implementation and understanding (more on this in the next week)