Recursion

- Recursion as problem solving technique
- Learning to think recursively
- Knowing when to use recursion
- Coding recursively in Java
- Limitations and efficiency of recursive programming

Thinking Recursively

- Our usual approach to understanding complexity is to break things into smaller, more understandable pieces
  - For example, we divide a program into classes and methods
  - The program is then executed sequentially
  - We may have repetitions which we can model with loops
- This is a form of "Divide and Conquer"
  - We reduce the problem to more manageable pieces
- Recursion is another form of reduction
  - The difference is that with recursion, we don't divide into pieces
  - Instead, we reduce the complexity to a smaller version of the same thing
Normally, we would think that defining something with the term being defined can not be a useful definition

- But consider this definition of a list of integers

  A list of integers is either a single integer or it is one integer followed by a comma and a list of integers.

- Do you believe that this properly defines a list?
  - Certainly the definition works for a list of one

- What about a list of two integers?
  - There is a first integer, and after it is the second, which by the definition is a list of integers
  - So this fits the definition

- What about a list of three integers?
  - There is a first integer, and after it are two integers, which we just noted fits the definition
  - So now this fits the definition, too

- We could keep going
  - We are defining a list as the first item, and then a smaller list

**Recursive Definitions**

- Definition of factorial as you might find in a math text:

  \[ N! = \begin{cases} 
  1 & \text{if } N \leq 1 \\ 
  N \times (N-1)! & \text{if } N > 1 
  \end{cases} \]

  - base case
  - recursive part

- Notice that the definition of factorial uses the definition of factorial

- Use the definition to determine the value of 5!

  \[ 5! = 5 \times 4! \]
  \[ 4! = 4 \times 3! \]
  \[ 3! = 3 \times 2! \]
  \[ 2! = 2 \times 1! \]
  \[ 1! = 1 \]

  \[ 5! = 5 \times 4 \times 3 \times 2 \times 1 = 120 \]
Recursion for Problem Solving

- In general, problem solving is done by decomposing a problem into smaller parts which are easier to solve
  - We solve all the pieces, and together they solve the problem
- A recursive approach decomposes a problem into smaller problems of the same type as the original
  - Solving the smaller problem allows us to solve the larger one
  - Eventually we get to a problem that is trivial to solve (the base case)
- Not all problems should be approached with recursion
  - Solving the smaller problem must be relevant to solving the larger problem
  - For example, knowing the square root of 1 is 1 does not easily help us solve the square root of 2

Recursive Solution Example

- A palindrome is a string that reads the same forward or backwards
  - Problem: Determine if a string is a palindrome.
- Recursive approach: If the first and last characters match and the "middle" is a palindrome, then we have a palindrome
  - The "middle" is the smaller problem (a shorter string to determine whether it is a palindrome)
  - The base case is a string with no "middle" (0 or 1 chars), which is a palindrome
  - Knowing the middle is a palindrome is relevant to knowing the whole string is a palindrome if we also know the first and last characters match
- Recursive algorithm
  - If the string is length zero or one, it is trivially a palindrome
  - If the first and last characters match, and the remaining characters (from the second through the next to last) form a palindrome, then the whole string is a palindrome
  - Otherwise, the string is not a palindrome
Programming Recursive Solutions

- In Java, a method can call any other method, including itself
  - We can use this to design a recursive method
- When a method calls itself, there is a new invocation of the method
  - The new invocation has its own environment (parameters, local variables)
  - When that invocation calls itself, there is now a third invocation
  - If this keeps happening, when will we ever return?
- For a recursive method to work correctly, there must be a base case
  - A condition to determine that the method does not call itself again
- To implement recursion, there must be a recursive case of the method
  - A condition to determine that the method should call itself
  - The recursive call should make progress toward the base case
    - Just as the iteration in a loop progresses toward the terminating condition

Tracing Recursive Calls

```java
boolean isPalindrome(String s) {
    boolean ret;
    int end = s.length()-1;
    if (end <= 0) {
        ret = true;
    } else if (s.charAt(0)==s.charAt(end)) {
        String mid = s.substring(1,end);
        ret = isPalindrome(mid);
    } else {
        ret = false;
    }
    return ret;
}
```

Example:
```
isPalindrome("racecar") (1)
  isPalindrome("aceca") (2)
    isPalindrome("cec") (3)
      isPalindrome("e") (4)
```

PalindromChecker.java
Recursive Programming

- A danger for recursive methods is that the recursion never ends
  - Base case might have been omitted in error
  - Logic to determine base case might be wrong
  - Values that are supposed to get closer to base case might be incorrectly growing away
- What happens if recursion does not end?
  - The method call stack keeps growing
  - Eventually memory will be exhausted
  - A Stack Overflow exception will terminate the program
- Important to design recursion very carefully
  - Errors are usually harder to see than in loops

Using Recursion

- Recursion can be a very useful way of finding solutions and understanding the nature of problems
  - A recursive solution is often simpler to state and comprehend
- But a recursive implementation may not be the best programming solution
  - Recursion can be expensive because of the stack use
  - Recursion can be dangerous because of subtle errors
- For example, computing factorial is probably best done with a loop rather than a recursive method
- Important to choose the best implementation for a solution
- But, some programming languages do not have loop constructs and only have recursion
  - And some languages do not permit recursive methods
  - Even so, thinking recursively can help understand and solve problems
Another Recursion Example

- Question: how many subsets are there for a set with N elements?
- Some observations:
  - The empty set \( \{ \} \) has **one** subset, namely itself
  - A set with one element \( \{ x \} \) has **two** subsets, namely \( \{ \} \) and \( \{ x \} \)
  - A set with two elements \( \{ x, y \} \) has **four** subsets, namely \( \{ \} \), \( \{ x \} \), \( \{ y \} \), and \( \{ x, y \} \)
- So \( \text{numSubsets}(0) \) is 1, \( \text{numSubsets}(1) \) is 2, \( \text{numSubsets}(2) \) is 4
- We can see a pattern and base cases, but how do we think recursively about this problem?

Subset Recursion Example

- Suppose we have a set \( S \) with \( N \) elements
- Can we relate the subsets of that set to knowledge about subsets of a smaller set? I.e., can we relate \( \text{numSubsets}(N) \) to smaller values?
- Certainly each subset of a subset of \( S \) is also a subset of \( S \).
- Let’s remove one element (call it \( x \)) from \( S \), and call the resulting set \( Q \)
  - \( Q \) has \( N-1 \) elements and \( \text{numSubsets}(N-1) \) subsets
- Now think about each subset of \( S \).
  - Either it contains the element \( x \), or it does not.
  - The subsets that **do not** contain \( x \) are the subsets of \( Q \), so there are \( \text{numSubsets}(N-1) \) of them.
  - For each subset of \( S \) that **does** contain \( x \), the rest of the elements are a subset of \( Q \), so there are \( \text{numSubsets}(N-1) \) of them.
Subset Recursion Example

- Putting this together as a recursive formula:
  - Base case:
    - If N == 0, numSubsets(N) is 1
  - Recursive case:
    - If N > 1, numSubsets(N) is 2 * numSubsets(N-1)
- We have a recursive formula, and from this can see that the number of subsets is $2^N$

Towers of Hanoi

- Solving the Towers of Hanoi puzzle is a good example of recursion
- The puzzle has three pegs and disks of different sizes that slide on to the pegs
- Starting configuration is to have all disks on one peg, ordered by size, with largest at bottom of pile
- Goal is to move the entire pile to another peg
- Rules
  - Only one disk can be moved at a time
  - It is illegal to ever place a larger disk on top of a smaller one
Towers of Hanoi

- Goal: move stack of disks from first peg to last peg
- Can only move one disk at a time
- Cannot place a larger disk on a smaller one
- Disks have to be on a peg (except the moving one)

Towers of Hanoi

- Four disk puzzle
- Naive approach: just move the disks to the middle peg, one by one, till we get biggest to move to last peg

  - First move top disk to peg 2
  - Then move next disk to peg 2

  Oops! A bigger disk is on a smaller one
Towers of Hanoi

- Let's think recursively...
- Suppose we knew how to move a stack of three disks from one peg to another
- Then we could get to this:

![Diagram of Towers of Hanoi]

And now we can move the big disk from the first peg to the last (do you think this is our base case?)
- So we could get to this:

- If only we knew we could move a stack of three...
  - Gee, that sounds familiar...
Towers of Hanoi

- Putting it together: moving the top 3 requires first moving the top 2 which requires moving the top 1 ...

Recursive Java method moveDisks to move a stack of disks of a given size
- Starting peg and destination peg are given
- Remaining peg is used as a spare

Base case
- Stack has only one disk, just move it

Recursive case
- Use method to move smaller stack to spare (all but one disk)
- Then move the one disk to destination
- Then move the other disks on top
- Note recursive method is called twice in the recursion

Recursion provides a small and elegant solution to the problem
- Iterative solution is possible, but much more complex and difficult

TowersOfHanoi.java
Recursion

- Recursion is often the best way to think about and solve certain types of problems
  - Finding shortest paths in graphs
  - Locating items in hierarchical tree structure
  - Exploring possible paths in maze or graph
  - Drawing fractals – figures which have smaller copies of themselves
- Choice of recursion versus iteration should be based on nature of problem and limitations of implementation
- The best solution is the one that feels most intuitive and is easiest to understand