CS416

Introduction to Computer Science II

Spring 2018

6a Recursion

- Recursion
  - A way of thinking about problems (and solutions)
  - It’s a form of *control structure* - how flow of control proceeds in your program; an alternative to *iteration* (looping)
  - It’s a way of thinking about *data structures*
- Examples
- Analysis

What is Recursion?

- In mathematics
  - when the definition of a function uses the function itself

factorial(0) = 1
factorial(n) = n*factorial(n-1) *for n >= 1*

What if n < 0?
function is **undefined**

- In programming
  - when a method calls itself

```
public int factorial( int n )
{
 if ( n == 0 )
     return 1;
   else
     return n * factorial( n-1 );
}
```

```
n<0 should throw exception!
```

Iteration vs. Recursion

Consider an iterative method to print a line that counts down from n to 0

```
public void countDown( int num )
{
  while ( num >= 0 )
  {
    out.print( num + " " );
    num = num - 1;
  }
  out.println();
}
```

Note: the loop *continuation* condition becomes a recursion *stopping* condition

```
public void countDown( int num )
{
  if ( num < 0 )
    out.println();
  else
  {
    out.print( num + " " );
    countDown( num - 1 );
  }
}
```

- Let's delete the loop
- After printing *num*, we can call *countDown*(*num-1*) to print *num-1*
- But, we need to stop! We stopped the loop when num < 0. We stop recursing then also!

Why use recursion?

- Mathematics
  - it's a concise, yet rigorous, approach for defining <u>some</u> functions
- Programming
  - it's a concise and elegant way to implement <u>some</u> algorithms

Why avoid recursion?

- It can be less efficient than iteration in both cpu cycles and memory
  - but, how often is it significant?
- Recursive code can turn into an infinite loop.
  - but, so can an iterative solution
- A recursive algorithm might be difficult to understand
  - but, it's often simpler
- Key: if you understand it, you can use it effectively!

Self-Similarity

- Recursion is effective when applied to a problem that is <u>*self similar*</u>
  - i.e., a version of the problem can be described in terms of a simpler version of the same problem!
  - 3! can be described as 3 * 2!
    2! can be described as 2 * 1!
    1! is trivial
- Whether a "problem" is self-similar depends on whether it is easy to <u>describe</u> it as self-similar

Recursion Basics

- Using recursion requires:
  - Self-similar problem description
  - A **base case** that can be solved by itself
  - A way of simplifying the problem at each step so that it gets closer to the base case.
- It also requires support from the language and execution environment
  - Need to have multiple "copies" of a single method
  - Each copy needs to have its own data
  - This is handled at runtime by activation records

Recursion Example 1

- What does the following method return?

```
public static int f( int n )
{
    if ( n == 0 )    // base case
        return 0;
    else             // self-similar case
        return n % 10 + f( n / 10 );
}
```

- Try some examples: f(32), f(169)

Recursion Example 2

- What does the following method return?

```
public static float g( float num1, int num2 )
{
   if ( num2 < 0 )          // base case #1
      return 0.0f;
   else if ( num2 == 0 )    // base case #2
      return 1.0f  ;
   else                     // self-similar case
      return num1 * g( num1, num2 - 1 );
}
```

- Try some examples: g(3, 2), g(2, 5)

Recursive String *reverse*

- Recursive function that reverses its string parameter
  - What's the base case? What string is a reverse of itself?
    - when the string has 1 character — or none!
  - Recursive case: remove 1st char, reverse rest, add 1st to end

```
public static String reverse( String word )
{
  if ( word.length() <= 1 )      // base case
     return word;                //   it is reverse of itself
  else                           // self-similar case
  {
     char   first = word.charAt( 0 );        // first char
     String rest  = word.substring( 1 );     // all but first
     return reverse( rest ) + first;
  }
}
```

```
// else clause is a verbose version of
return reverse( word.substring( 1 )) + word.charAt( 0 );
```

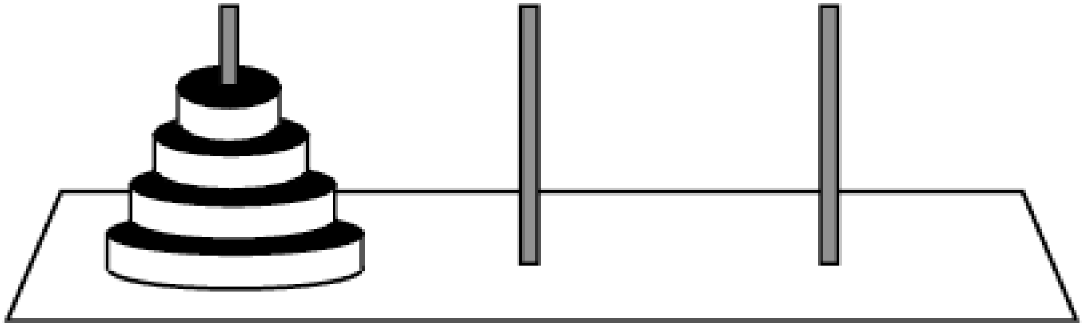
Recursive *modulus*

- Write a recursive function that will calculate modulus (%)

```
public static int modulus( int val, int divisor )
 {
   if ( val < divisor  )    // base case
      return val;
   else                     // self-similar case
      return modulus( val - divisor, divisor );
 }
```

Towers of Hanoi

- The towers of Hanoi is a simple game in which you try to move a stack of disks from one peg to another
- The disks are moved one at a time
- A larger disk cannot be placed on a smaller disk
- Solve for 1, use that for 2, use that for 3, etc.



http://www.pbs.org/teachers/mathline/

It is a Hindu legend that at the beginning of time monks at a temple were given 64 disks to move using “tower” rules. When they were finished the temple would be dust.

That is 18,446,744,073,709,551,615 moves

At 1 second per move

= 3.07E17 minutes

= 5.12E15 hours

= 2.13E14 days

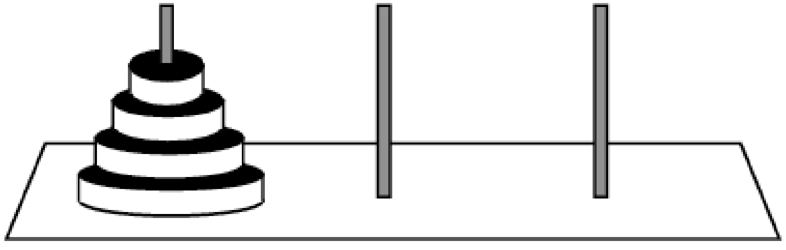
= 5.85E11 years = 584,942,417,355 ≈ 585 billion years

That is about 42 times as long as the universe has existed.

See, the answer really is 42!

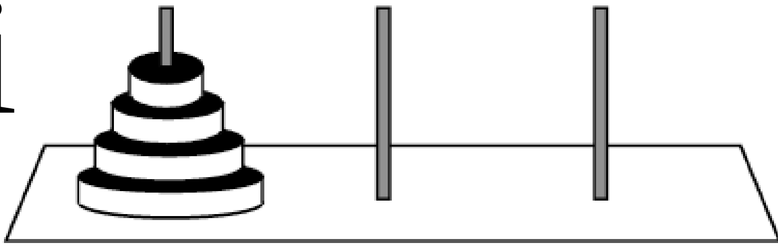
Towers of Hanoi

http://wipos.p.lodz.pl/zylla/games/hanoi5e.html



- How do we describe the solution (as code)
- It is difficult to come up with a non-recursive algorithm to solve the problem
- If you think recursively, there is a simple solution

Towers of Hanoi



Recursive solution

- Base case: it is easy to move one disk to an empty peg (or one that has a larger disk on top).
- Recursive case: To move n > 0 disks from the first peg to the last (using the middle one):
  - move n-1 disks from first to middle (using last)
  - move 1 disk from first to last (last is empty)
  - move n-1 disks from middle to last (using the first)
- It works because once you get the largest to the target peg, it's like an "empty" peg for all the rest

Towers of Hanoi Code

- Let's convert this recursive algorithm to Java code?
- "move" is the key operation and there are 2 versions:
  - recursion case: move n-1 disks from a to b using c; this is the key one: what are its parameters?
    - number of disks to move
    - source peg (src)
    - destination peg (dst)
    - work peg
  - *move(int n, Peg src, Peg dst, Peg work);*
  - base case: "move 1 disk from a to b"
  - *move(Peg src, Peg dst);*

Towers of Hanoi

```
public void move( int n, Peg src, Peg dst, Peg work )
{
  if ( n == 1 )
    move( src, dst );
  else
  {
    move( n-1, src, work, dst );
    move( src, dst );
    move( n-1, work, dst, src );
  }
}
```

Move n-1 smaller from src to work

Move largest (of the n) from src to dst

Move n-1 smaller from work to dst

- Review this problem in the text; it has 2 recursions
- This is a minor variation that I find simpler to read

Anagrams

- Problem:
  - Given a string of *n* characters
  - For all possible permutations of the characters
    - Check if the permutation is a valid English word
- Key *algorithmic* problem is generating the permutations
- Also need a dictionary of valid words!

Permutation Perusal

- How are we going to generate permutations?
- Examples often help
  - ab → ab, ba
  - abc → ?
    - "a" concatenated with all permutations of "bc" → abc, acb
    - "b" concatenated with all permutations of "ac" → bac, bca
    - "c" concatenated with all permutations of "ab" → cab, cba
  - abc → abc, acb, bac, bca, cab, cba
- Hints at a self-similar description

Self-Similar Permutation

```
permutations( string ):
    if string.length == 1
        return string in a list
    else
        results = null
        for each char in string
            rest = string without the char
            list = permutations( rest )
            for each perm in list
                results.add( char + perm )
        return results
```

Base case

Recursion step: each recursion is done on a shorter string

This solution generates and saves all permutations in a list. Often want to process each permutation as it is found and never store all of them.

Self-Similar Permutation II

- Want a self-similar description that does not save the permutations
- Instead of concatenating the “chosen” character to generated permutations on the way up the recursion ladder, pass the concatenation of all previous “chosen” characters as you go down the recursion ladder

permute(head, tail)

where *head* is the string of selected characters and
tail is the remaining characters that still need
all permutations generated for this *head*

Permutation by Recursion

- High-level algorithm:

```
main:
  Let input be string to permute
  permute( “”, input );
```

```
permute( head, tail ):
  if ( tail is empty )
    print head;
  else
    for i = 0 to length( tail ) - 1
      newHead = head + tail[i]
      newTail = tail w/o tail[i]
      permute( newHead, newTail )
```

- Hand-simulation

```
input = “abc”
permute(“”, “abc” )
    permute(“a”, “bc” )
        permute(“ab”, “c” )
            permute(“abc”, “”):print
        permute(“ac”, “b” )
            permute(“acb”, “”):print
    permute(“b”, “ac” )
        permute(“ba”, “c” )
            permute(“bac”, “”):print
        permute(“bc”, “a” )
            permute(“bca”, “”):print
    permute(“c”, “ab” )
    ...
```

Anagram Algorithm

- Use the permutation algorithm to build an anagram algorithm:
  - Given a string of letters and a word dictionary, find all permutations that are valid words

```
main:
  read dictionary
  read input
  foundWords = empty
  while input != null
    lookup( foundWords, "", input )
    if foundWords is empty
       print "no valid words"
    else
       print all words in foundWords
```

```
lookup( foundWords, head, tail ):
  if ( tail is empty )
    if isWord( head )
       foundWords.add( head )
  else
    for i = 0 to length( tail ) - 1
       newHead = head + tail[i ]
       newTail = tail w/o tail[i ]
       lookup( foundWords, newHead, newTail )
```

What if we want all words made of 3 or more characters in the permutation?

What if we want to avoid duplicate words?

Anagram Program

- ~cs416/public/demos/anagram/
  - Anagram.java
  - opted3to8.txt: the dictionary
- Run it for 3, 4, 5, 6, 7, 8 character words
  - What happens?

Anagram Performance

- Anagram works pretty well for 3-5 character strings, then slows dramatically
- Where's the problem?
  - Recursive generation of the permutations?
    - A string of length 8 has 8! = 40,320 permutations
  - Test for a valid word?
    - Dictionary has about 60K words
- It is easy to comment out the code in *ifWord*
  - Clearly, that was the bottleneck
  - Replacing *Vector* with *HashSet* solves the problem

Anagram Analysis

- What does *Vector.contains(String)* do?
  - It searches through the *Vector* starting at 0 comparing each entry to the argument.
  - On average it tests half of the entries (30K) before finding a match; and it must make 60K comparisons to find that the argument is <u>not</u> in the array (normal for this program)
  - This a *linear* algorithm, as an order of magnitude, it must make *n* comparisons (*n* is # words in the dictionary).
  - We say it is an *order n* algorithm or *O(n)*.
- What does *HashSet.contains(String)* do?
  - This is a *constant* time algorithm: it takes about the same amount of time regardless of *n*. We say it is *O(1)*.
  - Later this semester we'll find out how it performs this magic!