Intro to Programming: Java - Lecture 16

**Announcements**

- Program 5 up, due 11/4
- Ch 8 embedded due tonight (trapdoor req)
- Ch 8 OWL assignment due next Monday
- Ch 9 embedded problems due next Wed
- Extra credit programs coming
- Second midterm ~11/18
- Final 12/17
Recursion

A somewhat unusual and subtle but very general problem-solving technique

This is statement-level Java - really: a different way to solve subproblems using methods
What do methods do? They solve (sub)problems:
(from WordFrequency program...)

w is some word, its handed to updateWords, which solves the problem of accounting for this word occ

```java
public void updateWords(String w){
    int where = -1;
    where = findWord(w);
    if (where >= 0)
        words[where].incCount();
    else addWord(w);
}
```
public void updateWords(String w) {
    int where = -1;
    where = findWord(w);
    if (where >= 0)
        words[where].incCount();
    else addWord(w);
}

public int findWord(String w) {
    int ans = -1;
    for (int j = 0; j < lastEmpty; j++) {
        ...
        return ans;
    }
Can methods call themselves?

Answer: yes -> called recursive methods
Subproblem transformation

We’ll think of recursion (recursive method conception) as

A transformation from a problem to a smaller problem instance
If s is a String, then s.substring(1) is the substring you get when first char removed:

\[
\text{s} = "\text{abcde}" , \quad \text{s}.\text{substring}(1) \rightarrow "\text{bcde}" \\
\]

We’ll often call s.substring(1) the TAIL of s (and sometimes we’ll call first character the HEAD of s)

What’s the head of:  rrbweoriger, vwebe, blogger

What’s the tail of:  know, clone, zipper
If \( s \) is a String, then \( s \text{.substring}(1) \) is the substring you get when first char removed:

\[
s = "\text{abcde}" , \quad s \text{.substring}(1) \rightarrow "\text{bcde}"
\]

We’ll often call \( s \text{.substring}(1) \) the TAIL of \( s \) (and sometimes we’ll call first character the HEAD of \( s \)

What’s the head of:  \text{rrbweoriger, vwebe, blogger}

What’s the tail of:  \text{know, clone, zipper}
columnPrint(s) - print letters of s in a column

Subproblem - print letters of tail of s in a column

Final assembly - print the head, THEN do subproblem
columnPrint(s) - print letters of s in a column

Subproblem - print letters of tail of s in a column

Final assembly - print the head, THEN do subproblem

void columnPrint(String s){
    System.out.println(s.charAt(0)); // print head
    columnPrint(s.substring(1)); // do subproblem
}
What does this mean?

If we take on faith that `columnPrint “works”` then `columnPrint(“abcde”)` does

```
  a  (is printed)
  b
  c
  d
  e
```

Printed by subproblem
Problem - can’t do head/tail decomposition if string is empty -
we need to account for this!
This - below - really says: if (s.length() == 0) do nothing, just exit the method.

```java
void columnPrint(String s){
    if(s.length() > 0){
        System.out.println(s.charAt(0));
        columnPrint(s.substring(1));
    }
}
```
revColumnPrint(s) - print letters of s in a column, in reverse order - revColumnPrint("cat") prints:

\texttt{t}

\texttt{a}

\texttt{c}

Subproblem - print letters of tail of s in a column in reverse order

Final assembly - do subproblem first, then print head
void revColumnPrint(String s) {
    if (s.length() > 0) {
        revColumnPrint(s.substring(1));
        System.out.println(s.charAt(0));
    }
}
digitSum(s) - s a string of digits, add these digits - so digitSum("48556") should return 28

Subproblem - add the string of digits in the tail of s.

Final assembly - add the lead digit to the answer to the subproblem

[recall: doing arithmetic with chars really does arithmetic with their positions in type. Since 0-9 run consecutively

        int k = ('9' - '5');

        Puts the int 4 into k
int digitSum(String s) {  // s is all digits
    return((head(s)-'0') + digitSum(tail(s)));
}

int digitSum(String s) {  // s is all digits
    return((s.charAt(0)-'0') +
            digitSum(s.substring(1));
}

But what if s is empty (string of length 0???)
int digitSum(String s) { // s is all digits
    if(s.length() == 0) return 0;
    else
        return((s.charAt(0) - '0') +
                digitSum(s.substring(1));
}

What happens here if the string input is "6"
What's the head? 6
What's the tail? "" --- the empty string, for which
digitSum returns 0
So this returns (6 + 0)
dashCheck(s) - return true if s contains a ' - '

Subproblem - return true if tail of s contains a ' - '

Final assembly - either first char is a ' - ',
OR
result of subproblem (that is, tail has ' - ')
sum(a,b) - sums ints from a to b

Subproblem - sums ints from a to (b-1)

Final assembly - add b to subproblem answer
factorial(n) = 1 \cdot 2 \cdot \ldots \cdot (n-1) \cdot n

Subproblem factorial (n-1)

Final assembly - multiply ans to subproblem by n
How would you compute factorial recursively?
This is n! -- 4! = 4*3*2*1, also 0! = 1 (by definition)

What’s a subproblem of (calculating) 4!?
How would you compute factorial recursively?
This is $n!$ -- $4! = 4 \times 3 \times 2 \times 1$, also $0! = 1$ (by definition)

What’s a subproblem of (calculating) $4!$?

**Answer:** (calculating) $3!$

of course then do something to that result: multiply it by 4
public static int fac(int n) {
    if (n <= 0)
        return 1;
    else return n*(fac(n-1));
}

fac(9) transformed into 9 * fac(8), which becomes 9*8*fac(7), which becomes ...
arraySum - add up the elements in an int array.

Head is last element / tail is the rest of the elements

A little tricky

```java
static int arraySum(int[] a, int top) {
    // sum up to top
    if(top < 0) return 0;
    else return(a[top] + arraySum(a, top-1));
}
```

This is the head    tail
The Stamp problem

Suppose you have a sheet of 10 37¢ stamps and 10 4¢ stamps, and you want to put exactly $2.21 on an envelope, using the stamps available.

Can you do it?

(call this a (10,10,221) problem)
The subproblem: there are two

(10,10,221) -> (9,10,217)
(10,10,221) -> (10,9,184)

first one: simply solve the problem after you've used one 4¢ stamp;
second one: after you've used a 37¢ stamp
**Sudoku - how to solve recursively**

```
8 5 0 0 0 0 4 0 1
0 0 0 0 0 0 6 7 0
0 0 2 1 0 0 0 0 3
0 0 8 5 0 0 0 0 7
0 0 0 9 8 2 0 0 0
3 0 0 0 0 1 5 0 0
5 0 0 0 0 4 3 0 0
0 3 7 0 0 0 0 0 0
2 0 9 0 0 0 0 5 8
```
<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>8</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>9</td>
<td>8</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>8</td>
</tr>
</tbody>
</table>
A seat-of-the-pants way to proceed:

Place a 37¢ stamp on the envelope, and look at remaining problem: with remaining 9 37¢ stamps, 10 4¢ stamps

Can you solve the new problem:

Make up $1.84 with the remaining stamps

This is a problem transformation:

$$(10,10,221) \rightarrow (9,10,184)$$
We'll actually be a little more systematic, and allow two possible transformations:

$(10,10,221) \rightarrow (9,10,184) \text{ or } (10,9,217)$

What does this mean??
Informal aside - this is going to work only if there’s a way for the problem transformation process to “bottom out”

That is, we can’t keep transforming the problem infinitely...
We’ll write program with this header:

```java
static boolean stampCheck(int s37, int s4, int t) {
```

```java
10
```

```java
10
```

```java
221
```

```java
37¢
```

```java
4¢
```

```java
221¢
```

}``
static boolean stampCheck(int s37, int s4, int t) {
    if ((s37 < 0) || (s4 < 0) || (t < 0))
        return false;
    if (t == 0) return true;
    else
        return ((stampCheck(s37 - 1, s4, t - 37))
                ||
                (stampCheck(s37, s4 - 1, t - 4)));
}
public static boolean stampCheck(int s37, int s4, int t) {
    if ((s37 < 0) || (s4 < 0) || (t < 0))
        return false;
    if (t == 0) return true;
    else
        return ((stampCheck(s37 - 1, s4, t - 37)) ||
                (stampCheck(s37, s4 - 1, t - 4)));
}
Three significant aspects of a recursive method:

• It calls itself (this could be indirect)
• It has base cases - non-recursive ways out
• All recursive paths lead to a base case
• Conceptually it’s about subproblems: to solve a problem instance, 1) check if it’s a base case; and 2) if it isn’t, solve by combining a little work with a (smaller) subproblem.
When is one string a prefix of another?

donkey
dog
donkey
dog
donkey
dog
public static boolean isPrefix(String s, String t){
    if (isEmpty(s)) return true;
    else if (isEmpty(t)) return false;
    else if (s.charAt(0) != t.charAt(0)) return false;
    else return isPrefix(s.substring(1), t.substring(1));
}

Note: 3 base cases. Order they're checked matters. The subproblem is: knock off lead character of both strings, and continue..
public static boolean isPrefix2(String s, String t){
    if (isEmpty(s)) return true;
    else if (isEmpty(t)) return false;
    else if (head(s) != head(t)) return false;
    else return isPrefix(tail(s), tail(t));
}
public static void triString(String s, int level){
    if(s.length() > 0)
    {
        for(int i = 0; i<level;i++)
            System.out.print(s.charAt(0));
        System.out.println();
        triString(s.substring(1),level+1);
    }
}

What does Methods.triString("abcd",3) do??
What is the base case??
Why do you know that you'll reach base??
Methods.palString("abcd");

abcdddcba
What’s the base case? What subproblem does the recursive call address?
Efficiency - The stamp problem revisited

(3,10,39)
(2,10,2)(3,9,35)
(1,10,-35)(2,9,-2)(3,9,35)
(2,9,-2)(3,8,31)
(2,8,-6)(3,7,27)
...

(2,5,-18)
(3,4,15)
(2,4,-22)
(3,3,11)
(2,3,-26)
(3,2,7)
(2,2,-30)
(3,1,3)
(2,1,-34)
(3,0,-1)
no way
call count: 23
Fibonacci numbers - a catastrophe

1,1,2,3,5,8,13

```java
public static int fib1(int n){
    if (n==0) return 1;
    if (n==1) return 1;
    return(fib1(n-1) + fib1(n-2));
}
```
fib(8) = fib(7) + fib(6)

fib(7) = fib(6) + fib(5)

fib(6) = fib(5) + fib(4)

fib(6) = fib(5) + fib(4)

fib(5) = fib(4) + fib(3)

fib(5) = fib(4) + fib(3)
public static int fib2(int b2, int b1, int term, int n) {
    if (n == 0) return 1;
    if (n == 1) return 1;
    if (term == n) return b1;
    else return fib2(b1, b1 + b2, term + 1, n);
}

To get the kth Fibonacci number (starting from the 0th)

fib2(1, 1, 1, k)