Thinking Recursively

CS 106B

Programming Abstractions
Fall 2016
Stanford University
Computer Science Department
Announcements

- Assignment 2 due Saturday
- Remember no LaIR on Friday night
- Book Readings
Today is an exciting day
Learn a new way to think
Course Syllabus

- Intro to Abstractions
  - ADTs

- Recursion

- Under the Hood
  - Vectors
  - Linked Lists
  - Hash Maps
  - Trees

- Graphs

You are here
Towers of Hanoi

Move this tower...

...to this spindle.
Towers of Hanoi

A

B

C
We’ve Gotten Ahead of Ourselves

Source: The Hobbit
Start at the Beginning

Source: The Hobbit
Recursion:

A problem solving technique in which problems are solved by reducing them into smaller problems of the same form.
Recursion: A problem solving technique in which problems are solved by reducing them into smaller problems of the same form.

What does that mean?
Pedagogy: Many simple examples
Operation: Understand Word
How Many Students Behind You?

How many students total are directly behind you in your "column" of the classroom?

1. You can see only the people right next to you. So you can't just look back and count.

2. But you are allowed to ask questions of the person next to you.

3. How can we solve this problem (recursively)
int numStudentsBehind(Student curr) {
    if (lastInRow(curr)) {
        return 0;
    } else {
        Student personBehind = curr.getBehind();
        return numStudentsBehind(personBehind) + 1;
    }
}

How Many Students Behind You?
The structure of recursive functions is typically like the following:

```plaintext
recursiveFunction:
    if (test for simple case) {
        Compute the solution without recursion
    } else {
        Break the problem into subproblems of the same form
        Call recursiveFunction on each subproblem
        Reassemble the results of the subproblems
    }
```
Every recursive algorithm involves at least 2 cases:

- **base case:** The simple case; an occurrence that can be answered directly; the case that recursive calls reduce to.

- **recursive case:** A more complex occurrence of the problem that cannot be directly answered, but can instead be described in terms of smaller occurrences of the same problem.
int numStudentsBehind(Student curr) {
    if (lastInRow(curr)) {
        return 0;
    } else {
        Student personBehind = curr.getBehind();
        return numStudentsBehind(personBehind) + 1;
    }
}
int numStudentsBehind(Student curr) {
    if (lastInRow(curr)) {
        return 0;
    } else {
        Student personBehind = curr.getBehind();
        return numStudentsBehind(personBehind) + 1;
    }
}

Base Case
Recursive Case

```java
int numStudentsBehind(Student curr) {
    if (lastInRow(curr)) {
        return 0;
    } else {
        Student personBehind = curr.getBehind();
        return numStudentsBehind(personBehind) + 1;
    }
}
```
int numStudentsBehind(Student curr) {
    if (lastInRow(curr)) {
        return 0;
    } else {
        Student personBehind = curr.getBehind();
        return numStudentsBehind(personBehind) + 1;
    }
}
Three Musts of Recursion

1. Your code must have a case for all valid inputs.

2. You must have a base case (makes no recursive calls).

3. When you make a recursive call it should be to a simpler instance (forward progress towards base case)
Write a recursive function that takes in a number \( (x) \) and an exponent and returns the result of \( x^{\text{exp}} \)
Powers

\[ x^0 = 1 \]
\[ x^n = x \cdot x^{n-1} \]
Each previous call waits for the next call to finish.

```cpp
// first call: power(5,3)
int power(int x, int exp) {
    // second call: power(5,2)
    int power(int x, int exp) {
        // second call: power(5,1)
        int power(int x, int exp) {
            // second call: power(5,0)
            int power(int x, int exp) {
                if (exp == 0) {
                    return 1;
                } else {
                    return x * power(x, exp - 1);
                }
            }
            return 1;
        }
        else {
            return x * power(x, exp - 1);
        }
    }
    return 1;
}
```

Powers
def power(x, exp):
    if exp == 0:
        # base case
        return 1
    else:
        if exp % 2 == 1:
            # if exp is odd
            return x * power(x, exp - 1)
        else:
            # else, if exp is even
            int y = power(x, exp / 2);
            return y * y

Exponentiation by squaring
How do you eat an entire bowl of chips?
Using a For Loop

How do you eat an entire bowl of chips?

**Iterative solution, for-loop:**

- You know exactly how many chips there are
- You start with Chip 0 and you keep eating until you eat Chip 238
How do you eat an entire bowl of chips?

**Iterative solution, while-loop:**

- You're not really sure how many chips there are
- But you're going to eat one chip at a time until the bowl is empty
How do you eat an entire bowl of chips?

A recursive solution:
How do you eat an entire bowl of chips?

**A recursive solution:**

- A bowl of N chips... is really: 1 chip + a bowl of (N - 1) chips
- So I'm going to eat 1 chip, then eat a bowl of (N - 1) chips
How do you eat an entire bowl of chips?

**A recursive solution:**

A bowl of \( N \) chips... is really: 1 chip + a bowl of \( (N - 1) \) chips

So I'm going to eat 1 chip, then eat a bowl of \( (N - 1) \) chips

I will keep doing this until I find that my bowl has 0 chips
Recursive tracing

Consider the following recursive function:

```c
int mystery(int n) {
    if (n < 10) {
        return n;
    } else {
        int a = n / 10;
        int b = n % 10;
        return mystery(a + b);
    }
}
```

Q: What is the result of mystery(648)?
A. 8
B. 9
C. 54
D. 72
E. 648
Write a recursive function `isPalindrome` accepts a string and returns true if it reads the same forwards as backwards.

- `isPalindrome("madam") → true`
- `isPalindrome("racecar") → true`
- `isPalindrome("step on no pets") → true`
- `isPalindrome("Java") → false`
- `isPalindrome("byebye") → false`
Three Musts of Recursion

1. Your code must have a case for all valid inputs.

2. You must have a base case (makes no recursive calls).

3. When you make a recursive call it should be to a simpler instance (forward progress towards base case)
"The code already works!"

- This is the most important strategy for recursion!
- When you are writing a recursive function, *pretend that it already works* and use it whenever possible in the body of the function.

"Look for the subproblem of the same form"

- Before writing your recursive function, write down what it is supposed to do.
- Then see how you can express the result of the function in terms of a smaller version of the original problem.
// Returns true if the given string reads the same
// forwards as backwards.
// Trivially true for empty or 1-letter strings.
bool isPalindrome(const string& s) {
    if (s.length() < 2) {  // base case
        return true;
    } else {  // recursive case
        if (s[0] != s[s.length() - 1]) {
            return false;
        }
        string middle = s.substr(1, s.length() - 2);
        return isPalindrome(middle);
    }
}
Print the sequences of numbers that you take to get from N until 1, using the Hailstone (Collatz) production rules:

If \( n == 1 \), you are done.

If \( n \) is odd your next number is \( 3 \times n + 1 \).

If \( n \) is even your next number is \( n / 2 \).
// Couts the sequence of numbers from n to one
// produced by the Hailstone (aka Collatz)
void hailstone(int n) {
    cout << n << endl;
    if(n == 1) {
        return;
    } else {
        if(n % 2 == 0) {
            // n is even so we repeat with n/2
            hailstone(n / 2);
        } else {
            // n is odd so we repeat with 3 * n + 1
            hailstone(3 * n + 1);
        }
    }
}
// Couts the sequence of numbers from n to one // produced by the Hailstone (aka Collatz) // procedure

```cpp
void hailstone(int n) {
    cout << n << endl;
    if (n == 1) {
        return;
    } else {
        if (n % 2 == 0) {
            // n is even so we repeat with n/2
            hailstone(n / 2);
        } else {
            // n is odd so we repeat with 3 * n + 1
            hailstone(3 * n + 1);
        }
    }
}
```

3. When you make a recursive call it should be to a simpler instance (forward progress towards base case)

Well that seems to be true…
Works for numbers up to $5 \times 10^{18}$
Reward for proof
$1,400
Here we are

Source: The Hobbit
This disk...

...needs to get over here.
This disk... needs to get over here.
This disk... needs to get over here.
This disk... needs to get over here.
Towers of Hanoi Insight
This disk... needs to get over here.
This disk... needs to get over here.

Towers of Hanoi Insight
This disk... needs to get over here.
This disk... needs to get over here.
We need to find a very simple case that we can solve directly in order for the recursion to work.

If the tower has size one, we can just move that single disk from the source to the destination.
int main() {
    moveTower(3, 'a', 'c', 'b');
}

A
C
A
B
C
int main() {
    moveTower(3, 'a', 'b', 'c');
}

void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}

void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}

n 2 from b to c temp a
The Solution

```c
void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}
```
This problem is very hard to solve “iteratively”
Welcome to the world of recursion
Your Brain is Recursive
There is a pathway in your brain for imagination
Step 1: Imagine your life at the end of CS106B
Step 1: Imagine your life at the end of CS106B
Step 2: Imagine one thing you could do next
Step 2: Imagine one thing you could do next
Step 3: Imagine your life after that choice

Executes the same “function” with different inputs
Step 3: Imagine your life after that choice


Executes the same “function” with different inputs
The End
1. Great style
2. Some things are naturally recursive
3. Master of control flow
More Practice
int mystery(int n) {
    if (n < 10) {
        return (10 * n) + n;
    } else {
        int a = mystery(n / 10);
        int b = mystery(n % 10);
        return (100 * a) + b;
    }
}

Q: What is the result of mystery(348)?
A. 3828
B. 348348
C. 334488
D. 3408
E. none of the above
int mystery(int n) {
    if (n < 10) {
        return (10 * n) + n;
    } else {
        int a = mystery(n / 10);
        int b = mystery(n % 10);
        return (100 * a) + b;
    }
}

// call 1: 348
// call 2a: 34
// call 2b: 8
// call 3a: 3
// call 3b: 4
• Write a recursive function evenDigits that accepts an integer and returns a new number containing only the even digits, in the same order. If there are no even digits, return 0.

  – Example: evenDigits(8342116) returns 8426
  – Example: evenDigits(40109) returns 400
  – Example: evenDigits(8) returns 8
  – Example: evenDigits(-163505) returns -60
  – Example: evenDigits(35179) returns 0
// Returns a new integer containing only the even-valued digits from the given integer, in the same order. 
// Returns 0 if there are no even digits.

int evenDigits(int n) {
    if (n < 0) {
        return -evenDigits(-n);
    } else if (n == 0) {
        return 0;
    } else if (n % 2 == 0) {
        return 10 * evenDigits(n / 10) + n % 10;
    } else {
        return evenDigits(n / 10);
    }
}
- What is a very easy power to compute without a loop?
- How is the task of computing exponents *self-similar*?

```c
int power(int base, int exp) {
    if (????) {
        // base case; no recursive calls needed
        ...
    } else {
        // recursive case
        ...
    }
}
```
• Each previous call waits for the next call to finish.

```cpp
- cout << power(5, 3) << endl;

// first call: 5 3
int power(int base, int exp) {
    if (exp == 1) {
        return base;
    } else {
        return base * power(base, exp - 1);
    }
}

// second call: 5 2
int power(int base, int exp) {
    if (exp == 1) {
        return base;  // 5
    } else {
        return base * power(base, exp - 1);
    }
}

// third call: 5 1
int power(int base, int exp) {
    if (exp == 1) {
        return base;  // 5
    } else {
        return base * power(base, exp - 1);
    }
}
```
• Recursion is about solving a small piece of a large problem.
  – What is 69743 in binary?
    • Do we know *anything* about its representation in binary?
  – Case analysis:
    • What is/are easy numbers to print in binary?
    • Can we express a larger number in terms of a smaller number(s)?
• Suppose we are examining some arbitrary integer N.
  – if N's binary representation is 10010101011
  – (N / 2)'s binary representation is 1001010101
  – (N % 2)'s binary representation is 1

  – What can we infer from this relationship?
// Prints the given integer's binary representation.
// Precondition: n >= 0
void printBinary(int n) {
    if (n < 2) {
        // base case; same as base 10
        cout << n;
    } else {
        // recursive case; break number apart
        printBinary(n / 2);
        printBinary(n % 2);
    }
}

– Can we eliminate the precondition and deal with negatives?
// Prints the given integer's binary representation.
void printBinary(int n) {
    if (n < 0) {
        // recursive case for negative numbers
        cout << "-";
        printBinary(-n);
    } else if (n < 2) {
        // base case; same as base 10
        cout << n << endl;
    } else {
        // recursive case; break number apart
        printBinary(n / 2);
        printBinary(n % 2);
    }
}
((1+3)*(2*(4+1)))
A tricky problem

```
((1+3)*(2*(4+1)))
```

95
Challenge

Implement a function which evaluates an expression string:

“(((1+3)*(2*(4+1))))”

or

“(7+6)”

or

“(((4*(1+2))+6)*7)”

Only * or +

Fully parenthesized
Motivating Problem

Implement a function which evaluates an expression:

\[
((1+3)*(2*(4+1)))
\]

or

\[
(7+6)
\]

or

\[
(((4*(1+2))+6)*7)
\]

Only * or +

Fully parenthesized
Implement `int evaluate(string & expression)`: 
- Takes a parameter of format "((1+3)*(2*(4+1)))"
- Fully parenthesized, no whitespace
- Only multiplication and addition
- Returns the integer result of the expression, e.g. 40

Assume you have a helper function:
- `indexOfOperator(string& exp);`
  which returns the index of the next operator to be evaluated
Anatomy of an Expression

Left Expression

((1*3) + (4+2))

Joining Operator

Right Expression
It is Recursive

\(((1*3) + (4+2))\)

The big instance of this problem is:
\(((1*3) + (4+2))\)

The smaller instances are:
\((1*3)\) and \((4+2)\)

What's the algorithm for solving expressions?
int evaluate(expression):
    • If expression is a number, return expression
    • Otherwise, break up expression by its joining operator:
        – leftResult = evaluate(leftExpression)
        – rightResult = evaluate(rightExpression)
        – Return leftResult operator rightResult

((1*3) + (4+2))
How do we evaluate \(((1 \times 17) + (2 \times (3 + (4 \times 9))))\)?
How do we evaluate \(((1\times17)+(2\times(3+(4\times9))))\)?
How do we evaluate \(((1\times17)+(2\times(3+(4\times9))))\)?
How do we evaluate \(((1\times17)+(2\times(3+(4\times9))))\)?

\[
(1 \times 17) + (2 \times (3 + (4 \times 9)))
\]

\[
(1 \times 17) + 39
\]

\[
17 + 3
\]

\[
4 \times 9
\]

\[
36
\]

\[
4 \times 9 = 36
\]

\[
3 + 36 = 39
\]

\[
2 \times 39 = 78
\]

\[
17 + 78 = 95
\]
How do we evaluate \(((1 \times 17) + (2 \times (3 + (4 \times 9))))\)?
How do we evaluate \(((1\times17)+(2\times(3+(4\times9))))\)?

\[
\begin{align*}
(1 \times 17) + (2 \times (3 + (4 \times 9)))
\end{align*}
\]
```cpp
int evaluate(const string& expression) {
    if (stringIsInteger(expression)) {
        return stringToInteger(expression);
    }

    string unwrappedExpression = expression.substr(1, expression.length() - 2);
    int opIndex = indexOfNextOperator(unwrappedExpression);

    string left = unwrappedExpression.substr(0, opIndex);
    string right = unwrappedExpression.substr(opIndex + 1);
    int leftResult = evaluate(left);
    int rightResult = evaluate(right);

    char op = unwrappedExpression[opIndex];
    if (op == '+') {
        return leftResult + rightResult;
    } else if (op == '*') {
        return leftResult * rightResult;
    }
    throw "invalid expression string";
}
```