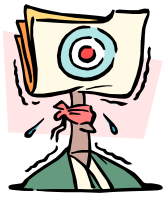
Recursion and Induction



Overview

- Recursion
  - a strategy for writing programs that compute in a "divide-and-conquer" fashion
  - solve a large problem by breaking it up into smaller problems of same kind
- Induction
  - a mathematical strategy for proving statements about integers (more generally, about sets that can be ordered in some fairly general ways)
- Understanding induction is useful for figuring out how to write recursive code.

Defining Functions

- It is often useful to write a given function in different ways.
  - (eg) Let S:int →int be a function where S(n) is the sum of the natural numbers from 0 to n.
    S(0) = 0, S(3) = 0+1+2+3 = 6
  - One definition: iterative form
    - S(n) = 0+1+…+n
  - Another definition: closed-form
    - S(n) = n(n+1)/2

Equality of function definitions

- How would you prove the two definitions of S(n) are equal?
  - In this case, we can use fact that terms of series form an arithmetic progression.
- Unfortunately, this is not a very general proof strategy, and it fails for more complex (and more interesting) functions.

Sum of Squares Functions

- Here is a more complex example.
  - (eg) Let SQ:int →int be a function where SQ(n) is the sum of the squares of natural numbers from 0 to n.

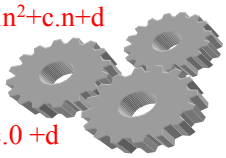
    $SQ(0) = 0$, $SQ(3) = 0^2+1^2+2^2+3^2 = 14$
- One definition:
  - $SQ(n) = 0^2+1^2+\ldots+n^2$
- Is there a closed-form expression for SQ(n)?

Closed-form expression for SQ(n)

- Sum of natural numbers up to n was n(n+1)/2 which is a quadratic in n.
- Inspired guess: perhaps sum of squares on natural numbers up to n is a cubic in n.
- So conjecture: $SQ(n) = a.n^3+b.n^2+c.n+d$ where a,b,c,d are unknown coefficients.
- How can we find the values of the four unknowns?
  - Use any 4 values of n to generate 4 linear equations, and solve.

Finding coefficients

$SQ(n) = 0^2+1^2+\ldots+n^2 = a.n^3+b.n^2+c.n+d$



- Let us use n=0,1,2,3.
- SQ(0) = $0 = a.0 + b.0 + c.0 + d$
- SQ(1) = $1 = a.1 + b.1 + c.1 + d$
- SQ(2) = $5 = a.8 + b.4 + c.2 + d$
- SQ(3) = $14 = a.27 + b.9 + c.3 + d$
- Solve these 4 equations to get

  $a = 1/3$, $b = ½$, $c = 1/6$, $d = 0$

- This suggests

  $SQ(n) \equiv 0^2+1^2+\ldots+n^2$

  $= n^3/3 + n^2/2 + n/6$

  $= n(n+1)(2n+1)/6$
- Question: How do we know this closed-form solution is true for all values of n?
  - Remember, we only used n = 0..3 to determine these co-efficients. We do not know that the closed-form expression is valid for other values of n.

- One approach:
  - Try a few values of n to see if they work.
  - Try n = 5. SQ(n) = 0+1+4+9+16+25 = 55.
  - Closed-form expression: 5*6*11/6 = 55.
  - Works!
  - Try some more values….
- Problem: we can never prove validity of closed-form solution for all values of n this way since there are an infinite number of values of n.

To solve this problem, let us express SQ(n) in another way.

$$SQ(n) = \underbrace{0^2 + 1^2 + \ldots + (n-1)^2}_{SQ(n-1)} + n^2$$

This leads to the following recursive definition of SQ:

$$SQ(0) = 0$$
$$SQ(n) = SQ(n-1) + n^2 \mid n > 0$$

To get a feel for this definition, let us look at

$$SQ(4) = SQ(3) + 4^2 = SQ(2) + 3^2 + 4^2 = SQ(1) + 2^2 + 3^2 + 4^2$$
$$= SQ(0) + 1^2 + 2^2 + 3^2 + 4^2 = 0 + 1^2 + 2^2 + 3^2 + 4^2$$

Notation for recursive functions

Base case

$SQ(0) = 0$
$SQ(n) = SQ(n-1) + n^2 \mid n > 0$

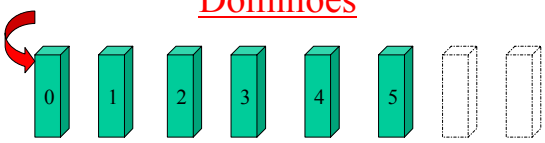
Recursive case

Can we show that these two definitions of SQ(n) are equal?

$$SQ_1(0) = 0$$
$$SQ_1(n) = SQ_1(n-1) + n^2 \mid n > 0$$

$$SQ_2(n) = n(n+1)(2n+1)/6$$

Dominoes



- Assume equally spaced dominoes, and assume that spacing between dominoes is less than domino length.
- How would you argue that all dominoes would fall?
- Dumb argument:
  - Domino 0 falls because we push it over.
  - Domino 1 falls because domino 0 falls, domino 0 is longer than inter-domino spacing, so it knocks over domino 1.
  - Domino 2 falls because domino 1 falls, domino 1 is longer than inter-domino spacing, so it knocks over domino 2.
  - ……
- Is there a more compact argument we can make?

Better argument

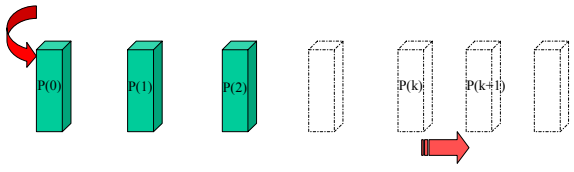
- Argument:
  - Domino 0 falls because we push it over.
  - Suppose domino k falls over. Because its length is larger than inter-domino spacing, it will knock over domino k+1.
  - Therefore, all dominoes will fall over.
- This is an inductive argument.
- Not only is it more compact, but it works even for an infinite number of dominoes!

Induction over integers

- We want to prove that some property P holds for all integers.
- Inductive argument:
  - P(0): show that property P is true for integer 0
  - P(k) => P(k+1): if property P is true for integer k, it is true for integer k+1
  - This means P(n) holds for all integers n.

Inductive proof that $SQ_1(n) = SQ_2(n)$ for all n

Let P(j): $SQ_1(j) = SQ_2(j)$



Prove P(0).

Prove P(k+1) assuming P(k).

$SQ_1(0) = 0$

$SQ_1(n) = SQ_1(n-1) + n^2$

$SQ_2(n) = n(n+1)(2n+1)/6$

Let P(j) be the proposition that $SQ_1(j) = SQ_2(j)$.

Proof by induction:

P(0): show $SQ_1(0) = SQ_2(0)$

(easy) $SQ_1(0) = 0 = SQ_2(0)$

P(k) => P(k+1):

Assume $SQ_1(k) = SQ_2(k)$

$SQ_1(k+1) = SQ_1(k) + (k+1)^2$	(definition of SQ_1)
$= SQ_2(k) + (k+1)^2$	(inductive assumption)
$= k(k+1)(2k+1)/6 + (k+1)^2$	(definition of SQ_2)
$=(k+1)(k+2)(2k+3)/6$	(algebra)
$= SQ_2(k+1)$	(definition of SQ_2)

Another example of induction

Prove that the sum of the first n integers is n(n+1)/2.

Let $S(i) = 0+1+2+...+i$

Show that $S(n) = n(n+1)/2$.

- Base case: ($n=0$)
  - $S(0) = 0$
  - So required result is proved for $n = 0$
- Inductive step:
  - Assume result is true for $0,1,2,...,k-1$
  - $S(k) = 0 + 1 + ... + (k-1) + k = S(k-1) + k$
    $= k(k-1)/2 + k$
    $= (k+1)(k)/2$
  - Therefore, if result is true for $k-1$, it is true for k.
- Conclusion: result follows for all integers.
- Note: we did not use arithmetic progressions theory.

Essence of proof is the following recursive description of $S(k)$

$S(0) = 0$

$S(k) = S(k-1) + k \mid k>0$

Editorial comments

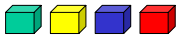
- Induction is a powerful technique for proving propositions.
- We used recursive definition of functions as a step towards formulating inductive proofs.
- However, recursion is useful in its own right.
- There are closed-form expressions for sum of cubes of natural numbers, sum of fourth powers etc. (see any book on number theory).

Recursion

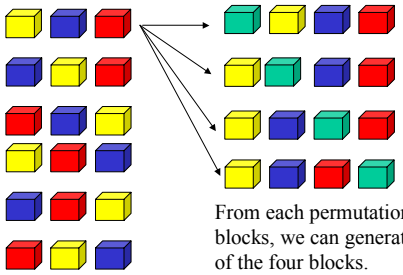
- Let us now study recursion in its own right.
- Recursion is a powerful technique for specifying functions, sets, and programs.
- Recursively-defined functions
  - factorial
  - counting combinations
  - differentiation of polynomials
- Recursively-defined sets
  - language of expressions

Factorial function

- How many ways can you arrange *n* distinct objects? This function is called *fact(n)*.
  - If *n = 1*, then there is just one way.
  - If *n > 1*, number of ways =
    *n** number of ways to arrange *(n-1)* objects
    (see next slide for example)
  - *fact(1) = 1*
    *fact(n) = n*fact(n-1) | (n > 1)*
- Another description of *fact(n):*
  *fact(n) = 1*2*...*n = n!*
- Convention: *fact(0) = 1*

Permutations of 

Permutations of non-green blocks



From each permutation of non-green blocks, we can generate 4 permutations of the four blocks.

Total number = 4*6 = 24 = 4!

Recursive program: factorial

fact(0) = 1

*fact(n) = n*fact(n-1) | (n > 0)*

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

Recursively-defined functions: Counting Combinations

$$^{n}C_{r} = {}^{n-1}C_{r} + {}^{n-1}C_{r-1} \mid n > r > 0$$
$$^{n}C_{n} = 1$$
$$^{n}C_{0} = 1$$

- How many ways can you choose r items from a set S of n elements?
  - Consider some element A.
  - Any subset of r items from set S either contains A or it does not.
  - Number of subsets of r items that do not contain A = $^{n-1}C_{r}$.
  - Number of subsets of r items that contain A = $^{n-1}C_{r-1}$.
  - Required result follows.
- You can show that

$$^{n}C_{r} = n!/r!(n-r)!$$

Example:

S = {A,B,C,D,E}

Consider subsets of 2 elements.

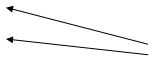
Subsets containing A:
{A,B}, {A,C}, {A,D},{A,E}

Subsets not containing A:
{B,C},{B,D},{C,D},{B,E},{C,E},{D,E}

Therefore, $^{5}C_{2} = {}^{4}C_{1} + {}^{4}C_{2}$

Counting combinations has two base cases

$$^{n}C_{r} = {}^{n-1}C_{r} + {}^{n-1}C_{r-1} \mid n > r > 0$$
$$^{n}C_{n} = 1$$
$$^{n}C_{0} = 1$$



Two base cases

- Coming up with right base cases can be tricky!
- General idea:
  - Figure out argument values for which recursive case cannot be applied.
  - Introduce a base case for each one of these.
- Rule of thumb: (not always valid) if you have *r* recursive calls on right hand side of function definition, you may need *r* base cases.

Recursive program: counting combinations

$$^{n}C_{r} = {}^{n-1}C_{r} + {}^{n-1}C_{r-1} \mid n > r > 1$$
$$^{n}C_{n} = 1$$
$$^{n}C_{0} = 1$$

```
static int combs(int n, int r){//assume n>r>1
    if ((r == 0) return 1;//base case
    else if (n == r) return 1;//base case
    else return combs(n-1,r) + combs(n-1,r-1);
}
```

Polynomial differentiation

Recursive cases:
d(uv)/dx = udv/dx + v du/dx
d(u+v)/dx = du/dx + dv/dx
Base cases:
dx/dx = 1
dc/dx = 0

Example:

d(3x)/dx = 3dx/dx + x d(3)/dx = 3*1 + x*0 = 3

Positive integer powers

$a^n = a * a * .... * a$ (n times)

Alternative description:

$a^0 = 1$

$a^n = a * a^{n-1}$

- Let us write this using standard function notation:
  power(a,n) = a*power(a,n-1) | n > 0
  power(a,0) = 1

Recursive program for power

power(a,n) = a*power(a,n-1) | n > 0
power(a,0) = 1

```
static int power(int a, int n) {
    if (n == 0) return 1;
    else return a*power(a,n-1);
}
```

Smarter power program

- If *n* is non-zero and even, $a^n = (a^{n/2})^2$
- If *n* is odd, $a^n = (a^{n/2})^2 * a$

```
static int coolPower(int a, int n){
    if (n == 0) return 1;
    else {int halfPower = coolPower(a,n/2);
        if ((n/2)*2 == n) return halfPower*halfPower;
        else return halfPower*halfPower*a;
}
```

As we will see later, this version is much faster than dumb version.

Recursively-defined sets: Grammars

E → integer
E → (E + E)

- Grammar: set of rules for generating sentences in a computer language.
- This is a grammar for simple expressions:
  - every integer is an expression.
  - if E_1 and E_2 are expressions, so is $(E_1 + E_2)$.
- Set of legal sentences in this grammar is a recursively-defined set.

E → integer
E → (E + E)

Here are some legal expressions:

2

(3 + 34)

((4+23) + 89)

((89 + 23) + (23 + (34+12)))

Here are some illegal expressions:

(3

3 + 4

- Parsing: given a grammar and some text, how do we determine if that text is a legal sentence in the language defined by that grammar?
- For many grammars such the simple expression grammar, we can write efficient programs to answer this question.
- Next slides: parser for our small expression language
  - Caveat: code uses CS211In object for doing input from a file, so it is not an Ur-Java program.
  - However, you should understand the structure of the code to see the parallel between the language definition (recursive set) and the parser (recursive function)

Helper class: CS211In

- Read the on-line code for the CS211In class
- Code lets you
  - open file for input:
    - CS211In f = new CS211In(String-for-file-name)
  - examine what the next thing in file is: f.peekAtKind()
    - Integer?: such as 3, -34, 46
    - Word?: such as x, r45, y78z (variable name in Java)
    - Operator?: such as +, -, *, (,) , etc.
  - read next thing from file:
    - integer: f.getInt()
    - Word: f.getWord()
    - Operator: f.getOp()

- Useful methods in CS211In class:
  - f.check(char c):
    - Example: f.check('*'); //true if next thing in input is *
    - Check if next thing in input is c
      - If so, eat it up and return true
      - Otherwise, return false
  - f.check(String s):
    - Example of its use: f.check("if");
      - Return true if next thing in input is word if

Parser for expression language

```
static boolean expParser(String fileName) {//returns true if file has single expression
        CS211In f  = new CS211In(fileName);
        boolean gotIt = getExp(f);
        if (f.peekAtKind() == f.EOF)//no junk in file after expression
           return  gotIt;
        else //file contains some junk after expression, so return false
           return false;
}
static boolean getExp(CS211In f) {//reads one expression from file
        switch (f.peekAtKind()) {
          case f.INTEGER: //E → integer
                          {f.getInt();
                           return true;
                          }
          case f.OPERATOR: //E→(E+E)
                           return f.check('(') && getExp(f) && f.check('+') &&
                                                    getExp(f ) && f.check(')'));
           default:   return false;
        }
}
```

Note on boolean operators

- Java supports two kinds of boolean operators:
  - E1 & E2:
    - Evaluate both E1 and E2 and compute their conjunction (i.e.,"and")
  - E1 && E2:
    - Evaluate E1. If E1 is false, E2 is not evaluated, and value of expression is false. If E1 is true, E2 is evaluated, and value of expression is the conjunction of the values of E1 and E2.
- In our parser code, we use &&
  - if "f.check('(') returns false, we simply return false without trying to read anything more from input file. This gives a graceful way to handling errors.
  - don't worry about this detail if it seems too abstruse…

Modifying parser to do SaM code generation

- Let us modify the parser so that it generates SaM code to evaluate arithmetic expressions: (eg)

```
2          : PUSHIMM 2
             STOP

(2 + 3)    : PUSHIMM 2
             PUSHIMM 3
             ADD
             STOP
```

Idea

- Recursive method getExp should return a string containing SaM code for expression it has parsed.
- To-level method expParser should tack on a STOP command after code it receives from getExp.
- Method getExp generates code in a recursive way:
  - For integer i, it returns string "PUSHIMM" + i + "\n"
  - For (E1 + E2),
    - recursive calls return code for E1 and E2
      - say these are strings S1 and S2
    - method returns S1 + S2 + "ADD\n"

CodeGen for expression language

```
static String expCodeGen(String fileName) {//returns SaM code for expression in file
        CS211In f = new CS211In(fileName);
        String pgm = getExp(f);
        return pgm + "STOP\n"; //not doing error checking to keep it simple
}
static String getExp(CS211In f) {//no error checking to keep it simple
        switch (f.peekAtKind()) {
          case f.INTEGER: //E → integer
                                return "PUSHIMM" + f.getInt() + "\n";
          case f.OPERATOR: //E→(E+E)
                                        {   f.check('(');
                                            String s1 = getExp(f);
                                            f.check('+');
                                            String s2 = getExp(f);
                                            f.check(')');
                                            return s1 + s2 + "ADD\n";
                                        }
          default:  return "ERROR\n";
        }
}
```

Exercises

- Think about recursive calls made to parse and generate code for simple expressions
  - 2
  - (2 + 3)
  - ((2 + 45) + (34 + -9))
- Can you derive an expression for the total number of calls made to get getExp for parsing an expression?
  - Hint: think inductively
- Can you derive an expression for the maximum number of recursive calls that are active at any time during the parsing of an expression?

Implementing recursive methods

- Ur-Java implementation model already supports recursive methods.
- Key idea:
  - each method invocation gets its own frame
  - frame for method invocation *I* has storage for
    - method parameters
    - method variables
    - returned value (methods invoked by method invocation *I* deposit their return value here)
  - caveat: frame actually contains a few other things as well but we will ignore them for now

Let us look at how stack frames are pushed and popped for execution of the invocation pow(5,3).

```
static int pow(int b, int p){
        if (p = = 0) return 1;
        else return b*pow(b,p-1);
}
```

At conceptual level, here is the sequence of method invocations:

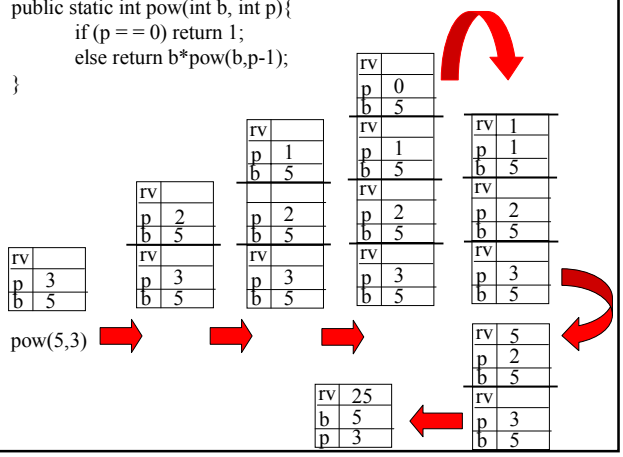
pow(5,3) → pow(5,2) → pow(5,1) → pow(5,0)

Initial Stack frame

rv	
p	3
b	5

Methods invoked by pow(5,3) will deposit result here.

```
public static int pow(int b, int p){
        if (p = = 0) return 1;
        else return b*pow(b,p-1);
}
```



Caveat

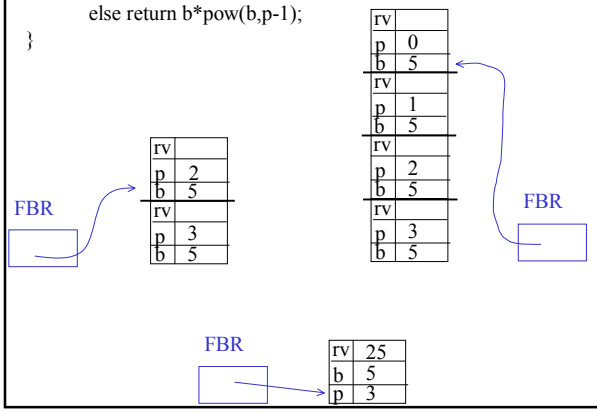
- In reality, frames for method invocations are a little more complex because they contain additional information.
- However, understanding the idealized picture of frames presented here is useful as a step to mastering nuances which will come later.

Things to think about

- At any point in execution, many invocations of pow may be in existence, so many stack frames for pow invocations may be in stack area.
- This means that variables p and b in text of program may correspond to several memory locations at any time.
- How does SaM know which location is relevant at any point in computation?
  - another example of association between name and "thing" (in this case, stack location)

- Answer:
  - Computational activity takes place only in the topmost (most recently pushed) frame.
  - Special SaM register called Frame Base Register (FBR) keeps track of where the topmost frame is.
- What would happen if method parameters and variables were bound to fixed locations like class variables were?
  - Recursive procedures would be disallowed because recursive calls would step on each other.
  - FORTRAN-77: early programming language adopted this policy, so recursion is not allowed in FORTRAN-77.
  - First languages to allow recursion: ALGOL-60, LISP

```
public static int pow(int b, int p){
        if (p = = 0) return 1;
        else return b*pow(b,p-1);
}
```



- Recursion is a very powerful technique for writing programs.
- Common mistakes:
  - Unwinding the recursion mentally (where do you stop??)
  - Incorrect or missing base cases
- Try to write "mathematical" description of the recursive algorithm like we have been doing, and reason about base cases etc. before writing program.
  - Why? Syntactic junk such as type declarations etc. may create mental fog which obscures the underlying recursive algorithm.