CSCI 135
Recursion

Semester Roadmap

1. Introduction
2. Control: Selection, Iteration, Function Composition
3. Data Structures: Primitives, strings, vectors, arrays, pointers, dynamic data structures
4. Organization: Functions and Modular Programming, Object Oriented Programming

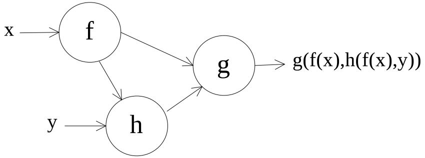
Completing the [pre-OO] picture: What if a function can call itself (called a recursive function)?

Why Recursion?

- Sometimes the most natural and elegant way of thinking about a problem
- Much easier way to convince yourself that your program is correct (when problem is naturally recursive)
- Forms basis of several major classes of algorithms
- Leads to another way of thinking about problems (as opposed to iteration)
- Some programming paradigms (*e.g.*, functional, logic) are primarily recursive, not iterative.

Breaking a Problem Down

- The 'traditional' way: Step 1, step 2, ...
  Use iteration for any step, when the number of substeps depends on some other problem parameter (say, n)
- The 'functional' way:
  - Decompose problem into functions that are composed:



  - If the number of compositions of function f depends on some parameter n, use recursion to have f call itself as many times as needed.

Thinking Recursively to Solve f(n)

WRONG way - trace through call: f(n) calls f(n-1) which calls f(n-2) which calls . . .

Better way

1. **Assume** you know the answer to f(n-1).
2. Write code to determine f(n) given the answer to f(n-1).
   DO NOT think about how f(n-1) is computed.
3. Add base case (*e.g.*, n==0) to avoid infinite chain of assumptions.

This corresponds to [finite mathematical] induction!

Thinking Recursively to Solve f(n)

WRONG way - trace through call: f(n) calls f(n-1) which calls f(n-2) which calls . . .

Better way *(to construct a building):*

1. **Assume** you know the answer to f(n-1).
   Assume you can construct a building with n-1 floors
2. Write code to determine f(n) given the answer to f(n-1).
   DO NOT think about how f(n-1) is computed.
   Figure out how to add a floor to a building
3. Add base case (*e.g.*, n==0) to avoid infinite chain of assumptions.
   Figure out how to build a foundation

This corresponds to [finite mathematical] induction!

Simple Recursion Example

Recall: n! = n * (n-1)! (base case: 0!=1)

```
int fact(int n) {
  if (n==0)                       base case
    return(1);
  else                            recursive case
    return(n * fact(n-1));        assuming fact(n-1) is known
}
```

- Allows us to directly code the problem without mapping it onto an iteration (and an extra iterator variable, i).

Another Recursion Example

Sometimes, you need more than one base case.
Fibonacci Numbers are given by the recurrence relation:

$$F_n = F_{n-1} + F_{n-2}$$

where $F_0 = 0, F_1 = F_2 = 1$
? How many base cases?

Another Recursion Example

Sometimes, you need more than one base case.
Fibonacci Numbers are given by the recurrence relation:

$$F_n = F_{n-1} + F_{n-2}$$

where $F_0 = 0, F_1 = F_2 = 1$

? How many base cases? ! 2

Recursive Version

```
int fib(int n) {
  if (n==0)
    return(0);
  else if (n==1)
    return(1);
  else
    return
      (fib(n-1)+fib(n-2));
}
```

Another Recursion Example

Sometimes, you need more than one base case.
Fibonacci Numbers are given by the recurrence relation:

$$F_n = F_{n-1} + F_{n-2}$$

where $F_0 = 0, F_1 = F_2 = 1$

? How many base cases? ! 2

Recursive Version

```
int fib(int n) {
  if (n==0)
    return(0);
  else if (n==1)
    return(1);
  else
    return
      (fib(n-1)+fib(n-2));
}
```

Iterative Version

```
int fib(int n) {
  if (n==0) return 0;
  int cur=1, prev=1, prev2=0;
  for (int i=2; i<n; i++) {
    prev2 = prev;
    prev  = cur;
    cur   = prev+prev2;
  };
  return(prev);
};
```

! Compare the elegances of the two ways!

Yet Another Example: Print Digits

Problem: Print the digits of a [decimal] number, n, one per line.
Let n have k digits.
Key Assumption: you can do it for a k-1 digit number.

1. Handle base case (k=1)
2. Output all but least significant digit
3. Output least significant digit

Yet Another Example: Print Digits

Problem: Print the digits of a [decimal] number, n, one per line.
Let n have k digits.
Key Assumption: you can do it for a k-1 digit number.

1. Handle base case (k=1)
2. Output all but least significant digit
3. Output least significant digit

```
void outDigits(n) {
  if (n<10) cout << n << endl;
  else {
    outDigits (n/10);
    cout << (n % 10) << endl;
  };
};
```

(!) We did not think about how this program executes. We only solved the general case, assuming the smaller case was solved (as long as we eventually get to the base case).

Advise Writing Recursive Function

- Assume smaller case/s
- Write function assuming solution/s for smaller case/s are available
- Write base case. Ensure that you are making progress towards base case (to avoid infinite chain of calls) – *e.g.*, base case is n$\leq$0 and you are reducing n on each call.

(Above is not in order – you can write base case first if you want)

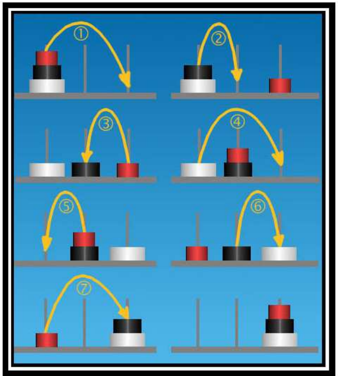
Again, DONT trace through calls!

Example: Towers of Hanoi

Move the column of n disks from peg 0 to peg 2, subject to the following constraints:

1. You may only move one disk in each step.
2. The moved disk must be from the top of a stack.
3. A disk may never rest on top of a smaller disk.

Example (n=3):



Towers of Hanoi Algorithm

How do we solve it (for general n)?

Towers of Hanoi Algorithm

How do we solve it (for general n)?
Iterative solution: need to come up with a general algorithm ☹
Recursive solution:

1. **Assume** you can solve the problem of moving n-1 disks. So, move n-1 disks from peg 0 to peg 2
   ☹What now? This doesn't help us

Towers of Hanoi Algorithm

How do we solve it (for general n)?
Iterative solution: need to come up with a general algorithm ☹
Recursive solution:

1. **Assume** you can solve the problem of moving n-1 disks. So, move n-1 disks from peg 0 to peg 2
   ☹What now? This doesn't help us

Recursive solution 2:

1. **Assume** you can solve the problem of moving n-1 disks from peg u to peg v ($u, v \in \{0, 1, 2\}$). So, move n-1 disks from peg u to the empty peg (denoted auxiliary peg).
2. Move the disk from peg u to peg v. Note this is the largest disk (so any disk may be placed on top of it).
3. Move n-1 disks from the auxiliary peg to peg v.

Base case (n=1) is trivial.

Towers of Hanoi Code

```
void move(int numdisks,
          int pegFrom, int pegTo, int pegAux)
{
  if (numdisks == 1)
    cout << "Move␣disk␣from␣peg␣" << pegFrom
         << "␣to␣" << pegTo;
  else {
    move (numdisks-1, pegFrom, pegAux, pegTo);
    cout << "Move␣disk␣from␣peg␣" << pegFrom
         << "␣to␣" << pegTo;
    move (numdisks-1, pegAux, pegTo, pegFrom);
  };
  return;
};
```

Recurrence Relations

- Many computer science problems can be expressed as **recurrence relations** – *i.e.*, something whose solution is some function of smaller instances of the same problem.
- The natural way of coding recurrence relations is as a recursion.
- But recursion is not always the most efficient implementation (using imperative languages such as C/C++).

Recurrence relation examples:

1. $F_n = F_{n-4} * F_{n-1}$
   $\Rightarrow$ Code is straightforward, but needs 4 base cases
2. $p(n+1) = \frac{(1+g-h)p(n)-g(p(n))^2}{M}$
   $\Rightarrow$ Code is straightforward, given base case p(0)
3. $C(n,k) = C(n-1,k-1) + C(n-1,k)$
   $\Rightarrow$ Code needs to recurse over two variables

Divide and Conquer

A general algorithmic method based on recursion:

- Divide the problem into smaller subproblems
- Write code to solve the problem *assuming* solutions to the smaller subproblems are available
- Make sure you have all necessary base cases (sometimes more than 1)

D-Q Example: Binary Search

Problem: Given a sorted array v and a target element te, determine the index of te in v.

? What do we recurse on?

D-Q Example: Binary Search

Problem: Given a sorted array v and a target element te, determine the index of te in v.

? What do we recurse on?

! The part of the array we are searching, represented as the range of indices.

Pseudocode for `int binSearch(v,te,indLo,indHi)`:

1. Base case: size of array subpart is ≤ 2. In this case, return whichever of indHi and indLo is correct.
2. Initialize index of median element:
3. If target is in left half, recursively search the left half:
4. Otherwise, search the right half:

The above would initially be called with `binSearch(v,te,0,size-1)`.

D-Q Example: Binary Search Code

Call with `binSearch(v,te,0,size-1)`

```
// Precond: te exists in v
int binSearch(SomeType[] v, SomeType te,
                  int indLo, int indHi) {
  int indMed;
  if (indHi-indLo <= 1) {
    if (v[indHi] == te) return indHi;
    else return indLo;
    };
  else {
    indMed = indLo + (indHi-indLo)/2;
    if (v[indMed] == te) return indMed;
    else if (v[indMed] > te)
      return binSearch(v,te,indLo,indMed);
    else return binSearch(v,te,indMed,indHi);
    };
}
```

D-Q Example: Merge Sort

Problem: Given an array (or vector) arr, sort it in nondecreasing order.
Algorithm:

1. Handle base case (*e.g.*, 1 element in array)
2. Sort the left half of array
3. Sort the right half of array
4. Merge the two halves (easy to do, since we only need to compare the least (leftmost) element of the two arrays in each iteration).

This ends up being one of the most efficient sorting algorithms!

Example: Reverse String

Problem: given a C++ string, return its reverse.

? What to assume???

Example: Reverse String

Problem: given a C++ string, return its reverse.

? What to assume???

! You have the reverse of all but the first character of the string (say, in variable revRest)

? How do we compute the reverse of the whole string?

Example: Reverse String

Problem: given a C++ string, return its reverse.

? What to assume???

! You have the reverse of all but the first character of the string (say, in variable revRest)

? How do we compute the reverse of the whole string?

! Append the first character of the string to revRest

We will use the substr function from the string library – substr(n) returns a string with all but the first n characters.

```
string reverse(string s) {
  if (s.size() == 0) return s;
  else {
    string temp = s.substr(1);
    return (reverse(temp) + s[0]);
  }
}
```

Example: Interleave String

Problem: Given two C++ strings s1 and s2, return a string that interleaves alternate elements.
Ex: interleave("abc","def") = "adbecf"

Example: Interleave String

Problem: Given two C++ strings s1 and s2, return a string that interleaves alternate elements.
Ex: interleave("abc","def") = "adbecf"
Spec Bug! What if s1 and s2 have different length?
$\Rightarrow$ Add precondition that both are of same length
? What to assume???

Example: Interleave String

Problem: Given two C++ strings s1 and s2, return a string that interleaves alternate elements.
Ex: interleave("abc","def") = "adbecf"
Spec Bug! What if s1 and s2 have different length?
⇒ Add precondition that both are of same length
? What to assume???

! You have the interleaving of tail(s1) and tail(s2) where tail returns a string with all but the first character of its argument.

```
// Precond: s1.length() == s2.length()
string interleave(string s1, string s2) {
  return (s1[0] + s2[0] +
          interleave(tail(s1),tail(s2));
```

Need base case (exercise for reader)

Caveats in Writing Recursive Code

Again, DONT trace through calls! But check following properties:

- Base case/s return correct values
- Recursive case correctly solves the big problem, assuming smaller problems have correctly been solved
- There are an apppriate number of base cases
- Recursive case actually makes progress towards a base case

Caveats in Writing Recursive Code

Again, DONT trace through calls! But check following properties:

- Base case/s return correct values
  Hanoi: 1-disk case is trivial ✓
- Recursive case correctly solves the big problem, assuming smaller problems have correctly been solved
  Hanoi: ✓
- There are an apppriate number of base cases
  Hanoi: just 1 needed since we always reduce the n disk problem to n-1 disk problems ✓
- Recursive case actually makes progress towards a base case
  Hanoi: we always [strictly] reduce the number of disks in the recursive calls ✓

⚠Often easier to solve more general problem (we did this in Towers of Hanoi)

When To Use / Avoid Recursion

Use Recursion:

- Searching through a space by searching through subspaces, and combining the results of the searches
- A problem can be formulated as an assembling of solutions to smaller problems
- Traversing inherently recursive dynamic data structures (*e.g.*, a family tree)
- Parsing grammars, both programming and natural languages

Avoid Recursion:

- Side effects (*e.g.*, printing something)
- Iterating through a fixed size data structure
- *Sometimes* not efficient (in imperative languages such as C/C++)

Exercises

1. Consider the recurrence relation, $f_n = kf_{n-3} + f_{n-3}$. How many base cases do you need (to avoid overspecifying or underspecifying the problem)? Let $f_n = n$ for these base cases. Write a recursive function `f(int n, int k)`.
2. Repeat the above iteratively.
3. Write a function to compute $C(n, k)$ recursively.
4. The number of bacteria in a Petri dish is twice what it was the previous week less 10% of the population in week 0 (truncated since you can't have half a bacterium). Write a function to return the number of bacteria in week n (n is the argument).
5. Pick any loop this semester, and reformulate it as a recursion.
6. Given an array of Color where Color is an enum, return a histogram of the elements in the array as another array.
7. Given a string, return the set of all permutations of the string.