Recursion: Introduction and Correctness

	Miles Jones	MTThF 8:30-9:50am	CSE 4140

August 8, 2016

Today's plan

1. What's recursion?
2. Correctness of recursive algorithms
3. Recurrence relations

In the textbook: Chapter 5 on Induction and Recursion and Sections 8.1-8.3 on Recurrence Equations

What's recursion?

What's recursion?

Solving a problem by

successively

reducing it to the

same problem

with

smaller

inputs.

Rosen p. 360

Strings and substrings

A string is a finite sequence of symbols such as 0s and 1s. Which we write as

$$b_1 \ b_2 \ b_3 \ \ldots \ b_n$$

A substring of length k of that string is a string of the form

$$b_i \ b_{i+1} \ b_{i+2} \ \ldots \ b_{i+k-1}$$

The substring 010 can be found in several places 0100101000.

Strings and substrings

A string is a finite sequence of symbols such as 0s and 1s. Which we write as

$$b_1 \; b_2 \; b_3 \; \ldots \; b_n$$

A substring of length k of that string is a string of the form

$$b_i \; b_{i+1} \; b_{i+2} \; \ldots \; b_{i+k-1}$$

The substring 010 can be found in several places 0100101000.

Strings and substrings

A string is a finite sequence of symbols such as 0s and 1s. Which we write as

$$b_1 \; b_2 \; b_3 \; \dots \; b_n$$

A substring of length k of that string is a string of the form

$$b_i \; b_{i+1} \; b_{i+2} \; \dots \; b_{i+k-1}$$

The substring 010 can be found in several places 0100101000.

Strings and substrings

A string is a finite sequence of symbols such as 0s and 1s. Which we write as

$$b_1 \ b_2 \ b_3 \ \ldots \ b_n$$

A substring of length k of that string is a string of the form

$$b_i \ b_{i+1} \ b_{i+2} \ \ldots \ b_{i+k-1}$$

The substring 010 can be found in several places 0100101000.

Example – Counting a pattern: WHAT

Count how many times the substring `00` occurs in the string 0100101000.

A. 0
B. 1
C. 2
D. 3
E. 4

Example – Counting a pattern: WHAT

Problem: Given a string (finite sequence) of 0s and 1s

$$b_1 \ b_2 \ b_3 \ \ldots \ b_n$$

count how many times the substring `00` occurs in the string.

HOW

Design an algorithm to solve this problem

Example – Counting a pattern: HOW

An Iterative Algorithm

Step through each position and see if pattern starts there.

procedure $countDoubleIter(b_1, \ldots, b_n : \text{each 0 or 1})$

$count := 0$

if $n < 2$ **then return** 0

for $i := 1$ **to** $n - 1$

  if $(b_i = 0$ **and** $b_{i+1} = 0)$ **then**

    $count := count + 1$

return $count$

Example – Counting a pattern: HOW

A Recursive Algorithm

Does pattern occur at the head? Then solve for the rest.

procedure $countDoubleRec(b_1, \ldots, b_n : \text{each 0 or 1})$

  if $n < 2$ **then return** 0

  if $(b_1 = 0$ **and** $b_2 = 0)$ **then return** $1 + countDoubleRec(b_2, \ldots, b_n)$

  return $countDoubleRec(b_2, \ldots, b_n)$

Recursive vs. Iterative

This example shows that essentially the same algorithm can be described as iterative or recursive.

But describing an algorithm recursively can give us new insights and sometimes lead to more efficient algorithms.

It also makes correctness proofs more intuitive.

Template for proving correctness of recursive alg.

Overall Structure: Prove that algorithm is correct on inputs of size n by induction on n.

Base Case: The base cases of recursion will be the base cases of induction. For each one, say what the algorithm does and say why it is the correct answer.

Template for proving correctness of recursive alg.

(Strong) Inductive Hypothesis: The algorithm is correct on all inputs of size (up to) k

Goal (Inductive Step): Show that the algorithm is correct on any input of size $k + 1$.

Note: The induction hypothesis allows us to conclude that the algorithm is correct on all recursive calls for such an input.

Inside the inductive step

1. Express what the algorithm does in terms of the answers to the recursive calls to smaller inputs.

2. Replace the answers for recursive calls with the correct answers according to the problem (inductive hypothesis.)

3. Show that the result is the correct answer for the actual input.

Example – Counting a pattern

procedure $countDoubleRec(b_1, \ldots, b_n : \text{each } 0 \text{ or } 1)$

    if $n < 2$ **then return** 0

    if $(b_1 = 0$ **and** $b_2 = 0)$ **then return** $1 + countDoubleRec(b_2, \ldots, b_n)$

    return $countDoubleRec(b_2, \ldots, b_n)$

Goal: Prove that for any string $b_1, b_2, b_3, \ldots b_n$, *countDoubleRec*$(b_1, b_2, b_3, \ldots b_n)$ = the number of places the substring 00 occurs.

Overall Structure: We are proving this claim by induction on n.

Proof of Base Case

procedure $countDoubleRec(b_1, \ldots, b_n : \text{each } 0 \text{ or } 1)$

  $\boxed{\textbf{if } n < 2}$ **then return** 0 Base Case

  if $(b_1 = 0$ **and** $b_2 = 0)$ **then return** $1 + countDoubleRec(b_2, \ldots, b_n)$

  return $countDoubleRec(b_2, \ldots, b_n)$

Base Case: $n < 2$ i.e. $n = 0$, $n = 1$.

$n = 0$: The only input is the empty string which has no substrings. The algorithm returns 0 which is correct.

$n = 1$: The input is a single bit and so has no 2-bit substrings. The algorithm returns 0 which is correct.

Proof: Inductive hypothesis

procedure $countDoubleRec(b_1, \ldots, b_n : \text{each } 0 \text{ or } 1)$

  if $n < 2$ **then return** 0

  if $(b_1 = 0$ **and** $b_2 = 0)$ **then return** $1 + countDoubleRec(b_2, \ldots, b_n)$

  return $countDoubleRec(b_2, \ldots, b_n)$

Inductive hypothesis: Assume that for any input string of length k, *countDoubleRec*$(b_1, b_2, b_3, \ldots b_k)$ = the number of places the substring 00 occurs.

Inductive Step: We want to show that *countDoubleRec*$(b_1, b_2, b_3, \ldots b_{k+1})$ = the number of places the substring 00 occurs for any input of length $k + 1$.

Proof: Inductive step

procedure $countDoubleRec(b_1, \ldots, b_n : \text{each 0 or 1})$
  if $n < 2$ **then return** 0
  if $(b_1 = 0$ **and** $b_2 = 0)$ **then return** $1 + countDoubleRec(b_2, \ldots, b_n)$
  return $countDoubleRec(b_2, \ldots, b_n)$

Case 1: $b_1 = 0$ and $b_2 = 0$: *countDoubleRec*$(b_1, b_2, b_3, \ldots b_{k+1})$ = 1 + *countDoubleRec*$(b_2, b_3, \ldots b_{k+1})$ = 1 + the number of occurrences of 00 in $b_2, b_3, \ldots b_{k+1}$ = one occurrence of 00 in first two positions + number of occurrences in later appearances.

Case 2: otherwise:
countDoubleRec$(b_1, b_2, b_3, \ldots b_{k+1})$ = *countDoubleRec*$(b_2, b_3, \ldots b_{k+1})$ = the number of occurrences of 00 in $b_2, b_3, \ldots b_{k+1}$ = the number of occurrences starting at the second position = the total number of occurrences since the first two are not an occurrence.

Proof: Conclusion

procedure $countDoubleRec(b_1, \ldots, b_n : \text{each } 0 \text{ or } 1)$

  if $n < 2$ **then return** 0

  if $(b_1 = 0$ **and** $b_2 = 0)$ **then return** $1 + countDoubleRec(b_2, \ldots, b_n)$

  return $countDoubleRec(b_2, \ldots, b_n)$

We showed the algorithm was correct for inputs of length 0 and 1. And we showed that if it is correct for inputs of length k > 0, then it is correct for inputs of length k + 1.

Therefore, by induction on the input length, the algorithm is correct for all inputs of any length. ☺

Time analysis for counting patterns.

procedure *countDoubleRec*($b_1, \ldots, b_n$: each 0 or 1)

Time(Suppose countDoubleRec($b_1, \ldots b_n$)) $= T(n)$

$\boxed{\textbf{if } n < 2 \textbf{ then return } 0}$ c_1

c_2 $\boxed{\textbf{if } (b_1 = 0 \textbf{ and } b_2 = 0)}$ **then** $\boxed{\textbf{return } 1 + countDoubleRec(b_2, \ldots, b_n)}$ $T(n-1)$

$\boxed{\textbf{return } countDoubleRec(b_2, \ldots, b_n)}$ $T(n-1)$

How long does this algorithm take?

$T(n) = c_1 + c_2 + \begin{cases} T(n-1) \\ T(n-1) \end{cases}$

It's hard to give a direct answer because it seems we need to know how long the algorithm takes to know how long the algorithm takes.

$$\boxed{T(n) = c + T(n-1)}$$

Solution: We really need to know how long the algorithm takes on smaller instances to know how long it takes for larger lengths.

$f(1) = 2$

$f(2) = 3 \cdot f(1) + 7 = 13$

$f(3) = 3 \cdot 13 + 7 = 46$

$f(4) = 3 \cdot 46 + 7 = 145$

Recurrences

formula

base case.

A **recurrence relation**
(also called a recurrence or recursive formula)
expresses **f(n)**
in terms of **previous** values, such as
f(n-1), f(n-2), f(n-3)….

Example:

f(n) = 3*f(n-1) + 7 → tells us how to find f(n) from f(n-1)

f(1) = 2 → also need a base case to tell us where to start

Recurrence relation for time analysis

$n=1$

$T(1) = c_0$
$T(0) = c_0$

procedure *countDoubleRec*$(b_1, \ldots, b_n$: each 0 or 1)

  if $n < 2$ **then** $\boxed{\textbf{return } 0}$ c_0

  if $(b_1 = 0$ **and** $b_2 = 0)$ **then return** $1 + \boxed{countDoubleRec(b_2, \ldots, b_n)}$

  return $\boxed{countDoubleRec(b_2, \ldots, b_n)}$

Base case

Let T(n) represent the time it takes for this algorithm on an input of length n.

recurrence.

Then $\boxed{T(n) = \boxed{T(n-1)} + c}$ for some constant c.
(The recursive call is of length n-1 and so it takes time T(n – 1). The rest of the algorithm is constant time.)

Solving the Recurrence

$T(n) = T(n-k) + kc$

$T(n) = T(n-(n-1)) + (n-1)c$

T(0) = T(1) = c0
T(n) = T(n-1) + c   for all $n > 1$

$= T(1) + (n-1)c$

$\boxed{T(n) = c_0 + (n-1)c}$

To find a closed form of T(n), we can unravel this recurrence.

$= \Theta(n)$

$T(n-1) = T(n-2) + c$

$$\boxed{T(n) = \boxed{T(n-1)} + c} = (T(n-2) + c) + c$$
$$= ((T(n-3) + c) + c) + c = \cdots = T(1) + c + c + \cdots + c$$
$$= T(1) + (n-1)c = cn + c0 - c \in \theta(n)$$

Two ways to solve recurrences

What does it mean to "solve"?

find the closed form from Recurrence.

1. Guess and Check

Start with small values of n and look for a pattern. Confirm your guess with a proof by induction.

2. Unravel

Start with the general recurrence and keep replacing n with smaller input values. Keep unraveling until you reach the base case.

Want to prove that $\sum_{k=0}^{n-1} r^k = \frac{r^n - 1}{r - 1}$

$S(n) = \sum_{k=0}^{n-1} r^k$, $rS(n) = \sum_{k=1}^{n} r^k$

$rS(n) - S(n) = r^n - 1$

$S(n)(r-1) = r^n - 1$

$$\boxed{S(n) = \frac{r^n - 1}{r - 1}}$$

$$f(1) = 2 \qquad f(n) = 3^{n-1} f(1) + 7 \cdot \sum_{k=0}^{n-2} 3^k$$

$$f(n) = 3 \cdot f(n-1) + 7 \quad \text{for all } n > 1$$

n	$f(n)$
1	2
2	13
3	46
4	145
5	
6	
7	

$$f(n) = 3\left[3f(n-2) + 7\right] + 7$$

$$\boxed{f(n) = 3^{n-1} \cdot 2 + 7 \cdot \frac{3^{n-1} - 1}{2}}$$

$$\boxed{f(n) = 3^{n-1}\left[\frac{11}{2}\right] - \frac{7}{2}}$$

$$= 3^4 f(n-4) + 7 \sum_{k=0}^{3} 3^k$$

$$f(n) = 3^i f(n-i) + 7 \sum_{k=0}^{i-1} 3^k$$

Subsequences

Given a string (finite sequence) of symbols

$$b_1 \ b_2 \ b_3 \ \dots \ b_n$$

A subsequence of length k of that string is a string of the form

$$b_{i_1}, b_{i_2}, \dots, b_{i_k}$$

where $1 \le i_1 < i_2 < \cdots < i_k \le n$. The subsequence 010 can be found in a whole bunch of places in 0100101000.

0100101000
0100101000
0100101000

Example – Longest Common Subsequence: WHAT

Given two strings (finite sequences) of characters*

$$a_1 \ a_2 \ a_3 \ \ldots \ a_n$$
$$b_1 \ b_2 \ b_3 \ \ldots \ b_n$$

what's the length of the longest string which is a subsequence in both strings?

What should be the output for the strings
AGGACAT and ATTACGAT?

A. 1
B. 2
C. 3
D. 4
E. 5

AACAT

* Could be 0s and 1s, or ACTG in DNA

Example – Longest Common Subsequence: WHAT

Given two strings (finite sequences) of characters*

$$a_1 \;\; a_2 \;\; a_3 \;\; \ldots \;\; a_n$$

$$b_1 \;\; b_2 \;\; b_3 \;\; \ldots \;\; b_n \; m$$

what's the length of the longest string which is a subsequence in both strings?

What should be the output for the strings

AGGACAT and ATTACGAT?

A. 1
B. 2
C. 3
D. 4
E. 5

* Could be 0s and 1s, or ACTG in DNA

$LCS(\overbrace{a_1 \dots a_n, b_1 \dots b_m}^{\text{input size } n+m})$: return length of longest common subseq.

if $n = 0$ return 0

IF $m = 0$ return 0

if $a_1 = b_1$:

$\qquad$ return $1 + \overbrace{LCS(a_2 \dots a_n, b_2 \dots b_m)}^{n+m-2}$

else: // ($a_1 \neq b_1$)

$$\text{return } \max \overbrace{\begin{bmatrix} LCS(a_1 \dots a_n, b_2 \dots b_m), \\ LCS(a_2 \dots a_n, b_1 \dots b_m) \end{bmatrix}}^{n+m-1}$$

Example – Longest Common Subsequence: HOW

Given two strings (finite sequences) of characters*

$$a_1 \; a_2 \; a_3 \; \ldots \; a_n$$
$$b_1 \; b_2 \; b_3 \; \ldots \; b_n$$

what's the length of the longest string which is a subsequence in both strings?

Design a recursive algorithm to solve this problem

* Could be 0s and 1s, or ACTG in DNA

Example – Longest Common Subsequence: HOW

A Recursive Algorithm

Do the strings agree at the head? Then solve for the rest.

$$
\begin{array}{l}
\textbf{procedure } lcsRec(a_1, \ldots, a_m; b_1, \ldots, b_n) \\
\quad \textbf{if } (m = 0 \textbf{ or } n = 0) \textbf{ then return } 0 \\
\quad \textbf{if } a_1 = b_1 \textbf{ then return } 1 + lcsRec(a_2, \ldots, a_m; b_2, \ldots, b_n) \\
\quad \textbf{return } max(lcsRec(a_1, \ldots, a_m; b_2, \ldots, b_n), lcsRec(a_2, \ldots, a_m; b_1, \ldots, b_n))
\end{array}
$$

Example – Longest Common Subsequence: HOW

A Recursive Algorithm

Do the strings agree at the head? Then solve for the rest.

$$\begin{array}{l}
\textbf{procedure } lcsRec(a_1, \ldots, a_m; b_1, \ldots, b_n) \\
\quad \textbf{if } (m = 0 \textbf{ or } n = 0) \textbf{ then return } 0 \\
\quad \textbf{if } a_1 = b_1 \textbf{ then return } 1 + lcsRec(a_2, \ldots, a_m; b_2, \ldots, b_n) \\
\quad \textbf{return } max(lcsRec(a_1, \ldots, a_m; b_2, \ldots, b_n), lcsRec(a_2, \ldots, a_m; b_1, \ldots, b_n))
\end{array}$$

What would an iterative algorithm look like?