Recursion: Introduction and Correctness

CSE21 Winter 2017, Day 7 (B00), Day 4-5 (A00)

January 25, 2017

http://vlsicad.ucsd.edu/courses/cse21-w17
Today’s Plan

From last time:
● intersecting sorted lists and tight bounds

New topic: **Recursion**
● recursive algorithms
● correctness of recursive algorithms
● solving recurrence relations
Intersecting sorted lists: WHAT

Given two sorted lists

\[ a_1, a_2, \ldots, a_n \text{ and } b_1, b_2, \ldots, b_n \]

determine if there are indices \( i, j \) such that

\[ a_i = b_j \]

Design an algorithm to look for indices of intersection
Intersecting sorted lists: HOW

Given two sorted lists

\[ a_1, a_2, \ldots, a_n \text{ and } b_1, b_2, \ldots, b_n \]

determine if there are indices i,j such that

\[ a_i = b_j \]

High-level description:

- Use linear search to see if \( b_1 \) is anywhere in first list, using early abort
- Since \( b_2 > b_1 \), start the search for \( b_2 \) where the search for \( b_1 \) left off
- And in general, start the search for \( b_j \) where the search for \( b_{j-1} \) left off
Intersecting sorted lists: HOW

\[ \text{Intersect}(a_1, \ldots, a_n, b_1, \ldots, b_n) \]
\[ i := 1 \]
\[ \text{for } j := 1 \text{ to } n \]
\[ \text{while } (b_j > a_i \text{ and } i \leq n) \]
\[ i := i + 1 \]
\[ \text{if } i > n \text{ then return } \text{false} \]
\[ \text{if } b_j = a_i \text{ then return } \text{true} \]
\[ \text{return } \text{false} \]
Intersecting sorted lists: WHY

\[ \text{Intersect}(a_1, \ldots, a_n, b_1, \ldots, b_n) \]

\[ i := 1 \]

\[ \text{for } j := 1 \text{ to } n \]

\[ \quad \text{while } (b_j > a_i \text{ and } i \leq n) \]

\[ \quad i := i + 1 \]

\[ \quad \text{if } i > n \text{ then return } false \]

\[ \quad \text{if } b_j = a_i \text{ then return } true \]

\[ \text{return } false \]

\textit{To practice: trace examples & generalize argument for correctness}
Intersecting sorted lists: WHEN

Using product rule

\[ \text{Intersect}(a_1, \ldots, a_n, b_1, \ldots, b_n) \]

\[ i := 1 \]

\[ \text{for } j := 1 \text{ to } n \]

\[ \text{while } (b_j > a_i \text{ and } i \leq n) \]

\[ i := i + 1 \]

\[ \text{if } i > n \text{ then return false} \]

\[ \text{if } b_j = a_i \text{ then return true} \]

return false
Intersecting sorted lists: WHEN

Using product rule

\[ \text{Intersect}(a_1, \ldots, a_n, b_1, \ldots, b_n) \]

\[ i := 1 \]

\[ \text{for } j := 1 \text{ to } n \]

\[ \text{O}(n) \]

\[ \text{return } false \]

Total: \( \text{O}(n^2) \)
Intersecting sorted lists: WHEN

More careful analysis ...

\[ \text{Intersect}(a_1, \ldots, a_n, b_1, \ldots, b_n) \]
\[ i := 1 \]
\[ \text{for } j := 1 \text{ to } n \]
\[ \text{while } (b_j > a_i \text{ and } i \leq n) \]
\[ i := i + 1 \]
\[ \text{if } i > n \text{ then return } false \]
\[ \text{if } b_j = a_i \text{ then return } true \]
\[ \text{return } false \]

Every time the while loop condition is true, i is incremented. If i ever reaches n+1, the program terminates (returns)
Intersecting sorted lists: WHEN

More careful analysis ...

\[
\text{Intersect}(a_1, \ldots, a_n, b_1, \ldots, b_n)
\]

\[i := 1\]

\[\text{for } j := 1 \text{ to } n\]

\[\text{while } (b_j > a_i \text{ and } i \leq n)\]

\[i := i + 1\]

\[\text{if } i > n \text{ then return } false\]

\[\text{if } b_j = a_i \text{ then return } true\]

\[\text{return } false\]

This executes $O(n)$ times total (across all iterations of for loop)
Intersecting sorted lists: WHEN

More careful analysis ...

\[ \text{Intersect}(a_1, \ldots, a_n, b_1, \ldots, b_n) \]
\[ i := 1 \]
\[ \text{for } j := 1 \text{ to } n \]
\[ \text{while } (b_j > a_i \text{ and } i \leq n) \]
\[ i := i + 1 \]
\[ \text{if } i > n \text{ then return } false \]
\[ \text{if } b_j = a_i \text{ then return } true \]
\[ \text{return } false \]

Total: \(O(n)\)

Be careful: product rule isn't always tight!

This executes \(O(n)\) times total (across all iterations of for loop)
What is recursion?
What is 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, written as

\[ b_1 \ b_2 \ b_3 \ \ldots \ b_n \]

A **substring of length** \( k \) **contains** \( k \) consecutive symbols of the string,

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

In how many places can we find 010 as a substring of 0100101000?

- A. 1
- B. 2
- C. 3
- D. 4
Counting a pattern: WHAT

Problem: Given a string of 0s and 1s

\[ b_1 \; b_2 \; b_3 \; \ldots \; b_n \]

count how many times the substring 00 occurs in the string.
Counting a pattern: HOW

Problem: Given a string of 0s and 1s

\[ b_1 \ b_2 \ b_3 \ \ldots \ b_n \]

count how many times the substring \(00\) occurs in the string.

*Design an algorithm to solve this problem*
An Iterative Algorithm:
Step through each position and see if pattern starts there.

```
procedure countDoubleIter(b_1, \ldots, b_n : 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
```
Counting a pattern: HOW

A Recursive Algorithm:
Does pattern occur at the head? Then solve for the rest.

procedure countDoubleRec(b₁, . . . , bₙ : each 0 or 1)
    if n < 2 then return 0
    if (b₁ = 0 and b₂ = 0) then return 1 + countDoubleRec(b₂, . . . , bₙ)
    return countDoubleRec(b₂, . . . , bₙ)
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.
Induction and recursion

**Induction**

A *proof strategy* where we prove

- The base case
- How to prove the statement is true about \( n+1 \) if we get to assume that it is true for \( n \).

**Recursion**

A way of *solving a problem* where we must give

- The base case
- How to solve a problem of size \( n+1 \), assuming we can solve a problem of size \( n \).
When should I use strong induction?
Template for proving a recursive algorithm correct

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.
procedure countDoubleRec(b_1, \ldots, b_n : 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)

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(b₁, ..., bₙ : each 0 or 1)
  if n < 2 then return 0
  if (b₁ = 0 and b₂ = 0) then return 1 + countDoubleRec(b₂, ..., bₙ)
  return countDoubleRec(b₂, ..., bₙ)

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(b₁, ..., bₙ : each 0 or 1)
    if n < 2 then return 0
    if (b₁ = 0 and b₂ = 0) then return 1 + countDoubleRec(b₂, ..., bₙ)
    return countDoubleRec(b₂, ..., bₙ)

Inductive hypothesis: Assume that for any input string of length $k$, $\text{countDoubleRec}(b₁, b₂, b₃, ..., bₘ) =$ the number of places the substring 00 occurs.

Inductive Step: We want to show that $\text{countDoubleRec}(b₁, b₂, b₃, ..., bₘ₊₁) =$ the number of places the substring 00 occurs for any input of length $k + 1$. 
Proof: Inductive step

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

- if \( n < 2 \) then return 0
- if \( (b_1 = 0 \text{ and } b_2 = 0) \) then return \( 1 + \text{countDoubleRec}(b_2, \ldots, b_n) \)
- return \( \text{countDoubleRec}(b_2, \ldots, b_n) \)

**Case 1:** \( b_1 = 0 \text{ and } b_2 = 0 \): \( \text{countDoubleRec}(b_1, b_2, b_3, \ldots b_{k+1}) = 1 + \text{countDoubleRec}(b_2, b_3, \ldots b_{k+1}) \) = \( 1 \) + 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:
\( \text{countDoubleRec}(b_1, b_2, b_3, \ldots b_{k+1}) = \text{countDoubleRec}(b_2, b_3, \ldots b_{k+1}) \) = 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(b₁, ..., bₙ : each 0 or 1)
    if n < 2 then return 0
    if (b₁ = 0 and b₂ = 0) then return 1 + countDoubleRec(b₂, ..., bₙ)
    return countDoubleRec(b₂, ..., bₙ)

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.
Counting a pattern: WHEN

procedure countDoubleRec(b₁, ..., bₙ : each 0 or 1)
    if n < 2 then return 0
    if (b₁ = 0 and b₂ = 0) then return 1 + countDoubleRec(b₂, ..., bₙ)
    return countDoubleRec(b₂, ..., bₙ)

How long does this algorithm take?

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.

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

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

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
Counting a pattern: WHEN

procedure countDoubleRec(b₁, . . . , bₙ : each 0 or 1)
    if \( n < 2 \) then return 0
    if \( b₁ = 0 \) and \( b₂ = 0 \) then return \( 1 + \text{countDoubleRec}(b₂, \ldots, bₙ) \)
    return \text{countDoubleRec}(b₂, \ldots, bₙ)

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

Then \( T(n) = T(n-1) + c \) for some constant \( c \).

Base cases: \( T(0) = T(1) = d \) for some constant \( d \).

Why not use a specific number?
Counting a pattern: WHEN

\[ T(n) = T(n-1) + c \]

\[ T(0) = T(1) = d \]

We can solve this recurrence by unraveling to get an explicit closed form solution:
Two ways to solve recurrences

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.
The Tower of Hanoi

Recursive solution:
1) Move the stack of the smallest \(n-1\) disks to an empty pole.
2) Move the largest disk to the remaining empty pole.
3) Move the stack of the smallest \(n-1\) disks to the pole with the largest disk.

How many moves? \(T(n) = \# \text{ of moves to solve puzzle with } n \text{ disks}\)
Tower of Hanoi: WHEN

Recursive solution:
1) Move the stack of the smallest n-1 disks to an empty pole.
2) Move the largest disk to the remaining empty pole.
3) Move the stack of the smallest n-1 disks to the pole with the largest disk.

$T(n) = \# \text{ of moves to solve puzzle with } n \text{ disks}$

Recurrence?

A. $T(n) = 2T(n-1)$
B. $T(n) = T(n-1) + 1$
C. $T(n) = n-1 + T(n)$
D. $T(n) = 2T(n-1) + 1$

Base case?

A. $T(1) = 1$
B. $T(1) = 2$
C. $T(0) = 0$
D. $T(2) = 2$
But what's the value of $T(n)$?

<table>
<thead>
<tr>
<th>$n$</th>
<th>$T(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>$n$</td>
<td></td>
</tr>
</tbody>
</table>

Recurrence for $T(n)$:

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

$T(1) = 1$
But what's the value of $T(n)$?

<table>
<thead>
<tr>
<th>$n$</th>
<th>$T(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>15</td>
</tr>
<tr>
<td>5</td>
<td>31</td>
</tr>
</tbody>
</table>

Recurrence for $T(n)$:

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

$T(1) = 1$

Is there a pattern we can guess?
Tower of Hanoi: WHEN

**Claim:** For each positive int \( n \), \( T(n) = 2^n - 1 \).

**Recurrence for \( T(n) \):**
\[
T(n) = 2T(n-1) + 1 \\
T(1) = 1
\]

**Proof by induction on \( n \) ...**

(Base case) If \( n = 1 \), then \( T(n) = 1 \) (according to the recurrence). Plugging \( n = 1 \) into the formula gives \( T(1) = 2^1 - 1 = 2 - 1 = 1 \). ☺
Claim: For each positive int n, $T(n) = 2^n - 1$.

Proof by induction on $n$ ...

(Induction step) Suppose $n$ is a positive integer greater than 1 and, as the induction hypothesis, assume that $T(n-1) = 2^{n-1} - 1$. We need to show that $T(n) = 2^n - 1$. From the recurrence,

$$T(n) = 2T(n-1) + 1 = 2(2^{n-1} - 1) + 1 = 2^n - 2 + 1 = 2^n - 1.$$ 

by the I.H.
Another method: "UNRAVEL" the recurrence:

\[ T(n) = 2T(n-1) + 1 \]
\[ = 2 \left( 2T(n-2) + 1 \right) + 1 = 4T(n-2) + 2 + 1 \]
\[ = 4 \left( 2T(n-3) + 1 \right) + 2 + 1 = 8T(n-3) + 4 + 2 + 1 \]
\[ : \]
\[ = 2^k T(n-k) + 2^{k-1} + \cdots + 2 + 1 = 2^k T(n-k) + (2^k - 1) \]
\[ : \]
\[ = 2^{n-1} T(1) + (2^{n-1} - 1) \]
\[ = 2^{n-1} + 2^{n-1} - 1 = 2^n - 1. \]
Counting recursively

We can write recurrence relations to describe the number of ways to do something, which is sometimes easier than counting the number of ways directly.

Don’t forget the base case(s)!

How many are needed?
Example – Binary strings avoiding 00

How many binary strings of length \( n \) are there which do not have two consecutive 0s?

<table>
<thead>
<tr>
<th>( n )</th>
<th>OK</th>
<th>NOT OK</th>
<th>How many OK?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0, 1</td>
<td>00</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>01, 10, 11</td>
<td>00</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>010, 011, 101, 110, 111</td>
<td>000, 001, 100</td>
<td>5</td>
</tr>
</tbody>
</table>
Example – Binary strings avoiding 00

How many binary strings of length n are there which do not have two consecutive 0s?

Recurrence?? $B(n) =$ the number of OK strings of length n

Any (long) "OK" binary string must look like

1_________ or 01_________

"OK" binary string of length n-1  "OK" binary string of length n-2
Example – Binary strings avoiding 00

How many binary strings of length $n$ are there which do not have two consecutive 0s?

**Recurrence??**

$$B(n) = B(n-1) + B(n-2) \quad B(0) = 1, \quad B(1)=2$$

Any (long) "OK" binary string must look like

```
B(n-1) 1_________ or 01_________
```

"OK" binary string of length $n-1$  "OK" binary string of length $n-2$
Example – Binary strings avoiding 00

$B(n) = B(n-1) + B(n-2)$ \quad B(0) = 1, \ B(1)=2$

<table>
<thead>
<tr>
<th>n</th>
<th>B(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>13</td>
</tr>
<tr>
<td>n</td>
<td>??</td>
</tr>
</tbody>
</table>
Announcements

HW3
Has been assigned
Due 1/31

MT1 is two weeks from today!

Keep an eye out for practice MT, review session, etc.