CSE21 Winter 2017, Day 8 (B00), Day 5 (A00)

January 27, 2017

http://vlsicad.ucsd.edu/courses/cse21-w17
Recursion

Last Time
1. Recursive algorithms and correctness
2. Writing a recurrence for time taken by recursive algorithm
3. Unraveling method to solve recurrences

Today
1. Guess and check method to solve recurrences
2. Counting recursively
3. Important example: Merging two sorted lists

In the textbook: Sections 5.4, 8.3
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.
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:

$$f(n) = 3f(n-1) + 7$$

$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 + countDoubleRec(b₂, . . . , bₙ)
  return countDoubleRec(b₂, . . . , 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:

\[
\begin{align*}
1 & : T(n) = T(n-1) + c \\
2 & : T(n) = [T(n-2) + c] + c \\
3 & : T(n) = \left[ \left[ T(n-3) + c \right] + c \right] + c \\
\vdots & \\
k & : T(n) = T(n-k) + kc \\
\vdots & \\
n-1 & : T(n) = T(n-(n-1)) + (n-1)c = d + (n-1)c = \Theta(n)
\end{align*}
\]
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

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

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

$T(1) = 1$
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? $T(n) = \# \text{ of moves to solve puzzle with } n \text{ disks}$
Towers of Hanoi: WHEN

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. \( T(2) = 2 \)

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.

\( T(n) = \# \text{ of moves to solve puzzle with } n \text{ disks} \)
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>$2T(1) + 1 = 3$</td>
</tr>
<tr>
<td>3</td>
<td>$2T(2) + 1 = 7$</td>
</tr>
<tr>
<td>4</td>
<td>$2T(3) + 1 = 15$</td>
</tr>
<tr>
<td>5</td>
<td>$2T(4) + 1 = 31$</td>
</tr>
<tr>
<td>$n$</td>
<td>$2^n - 1$</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?
Towers of Hanoi: WHEN

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

**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 \( T(1) = 2^1 - 1 = 2 - 1 = 1 \). ☺

IH: Suppose for some \( k \geq 1 \) \( T(k) = 2^k - 1 \).

Show \( T(k+1) = 2^{k+1} - 1 \)

\[
T(k+1) = 2T(k) + 1 = 2(2^k - 1) + 1 = 2^{k+1} - 2 + 1 = 2^{k+1} - 1
\]
Claim: For each positive integer \( n \), \( T(n) = 2^n - 1 \).

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,

\[
T(n) = 2 \cdot T(n-1) + 1 = 2 \cdot (2^{n-1} - 1) + 1 = 2^n - 2 + 1 = 2^n - 1.
\]

by the I.H.
Another method: “UNRAVEL” the recurrence:

\[ T(n) = 2T(n-1) + 1 \]
\[ = 2 \left( 2T(n-2) + 1 \right) + 1 = 4T(n-2) + 2 + 1 \]
\[ = 4 \left( 2T(n-3) + 1 \right) + 2 + 1 = 8T(n-3) + 4 + 2 + 1 \]
\[ \vdots \]
\[ = 2^k T(n-k) + 2^{k-1} + \cdots + 2 + 1 = 2^k T(n-k) + (2^k - 1) \]
\[ \vdots \]
\[ T(n-\frac{n-1}{2}) = 2^{n-1} T(1) + (2^{n-1} - 1) \]
\[ = 2^{n-1} + 2^{n-1} - 1 = 2^n - 1. \]
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.
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 - Pizza

You cut a pizza by making cuts along a diameter.

\[ P(n) = \# \text{ of slices after you have made } n \text{ cuts} \]

Recurrence?

A. \( P(n) = 2P(n-1) \)
B. \( P(n) = P(n-1) + 1 \)
C. \( P(n) = P(n-1) + 2 \)
D. \( P(n) = 2P(n-1) + 2 \)

Base case?

A. \( P(0) = 1 \)
B. \( P(1) = 1 \)
C. \( P(1) = 2 \)
D. \( P(2) = 1 \)
Example – Binary strings avoiding 00

How many binary strings of length \( n \) are there which do not have two consecutive 0s? Let \( B(n) \) be the # of OK strings of length \( n \).

<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>00</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0, 1</td>
<td>01, 10, 11</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>010, 011, 101, 110, 111</td>
<td>00, 000, 001, 100</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>010, 011, 101, 110, 111</td>
<td>00, 000, 001, 100</td>
<td>5</td>
</tr>
</tbody>
</table>

\[ B(n) = B(n-1) + B(n-2) \]
How many binary strings of length \( n \) are there which do not have two consecutive 0s?

**Recurrence??**

\[ B(n) = \text{the number of OK strings of length } n \]

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

1________ or 01________

"OK" binary string of length \( n-1 \)

"OK" binary string of length \( n-2 \)

There are \( B(n-1) \) \( \text{OK} \) strings of length \( n \) beginning with 0

There are \( B(n-2) \) \( \text{OK} \) strings of length \( n \) beginning with 1
How many binary strings of length $n$ are there which do not have two consecutive 0s?

**Recurrence??**

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

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

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

<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>
</tbody>
</table>

Fibonacci numbers
Given two sorted lists

\[ a_1, a_2, a_3, \ldots, a_k \]
\[ b_1, b_2, b_3, \ldots, b_l \]

produce a sorted list of length \( n = k + l \) which contains all their elements.

Which of the following could be the first element of the output?

A. \( a_1 \)
B. \( a_2 \)
C. \( b_1 \)
D. \( b_2 \)
E. More than one of the above.
Merging sorted lists: WHAT

Given two sorted lists

\[ a_1, a_2, a_3, \ldots, a_k \]
\[ b_1, b_2, b_3, \ldots, b_l \]

produce a sorted list of length \( n = k + l \) which contains all their elements.

\textit{Design a recursive algorithm to solve this problem}
A recursive algorithm
Focus on merging head elements, then rest.

```
procedure RMerge(a₁, ..., aₖ, b₁, ..., bₗ: sorted lists)
  if first list is empty then return b₁, ..., bₗ
  if second list is empty then return a₁, ..., aₖ
  if a₁ ≤ b₁ then
    return a₁ ∪ RMerge(a₂, ..., aₖ, b₁, ..., bₗ)
  else
    return b₁ ∪ RMerge(a₁, ..., aₖ, b₂, ..., bₗ)
```

concatenate
Merging sorted lists: WHY

A recursive algorithm
Focus on merging head elements, then rest.

Similar to Rosen p. 369

Claim: returns a sorted list containing all elements from either list

Proof by induction on n, the total input size

Meaning: “all elements from both lists”

procedure $R\text{Merge}(a_1, \ldots, a_k, b_1, \ldots, b_\ell$: sorted lists)
if first list is empty then return $b_1, \ldots, b_\ell$
if second list is empty then return $a_1, \ldots, a_k$
if $a_1 \leq b_1$ then
  return $a_1 \circ R\text{Merge}(a_2, \ldots, a_k, b_1, \ldots, b_\ell)$
else
  return $b_1 \circ R\text{Merge}(a_1, \ldots, a_k, b_2, \ldots, b_\ell)$
Claim: returns a sorted list containing all elements from either list

Proof by induction on $n$, the total input size

What is the base case?
A. Both input lists are empty ($n=0$).
B. The first list is empty.
C. The second list is empty.
D. One of the lists is empty and the other has exactly one element ($n=1$).
E. None of the above.

**procedure** $R$Merge$(a_1, \ldots, a_k, b_1, \ldots, b_\ell$: sorted lists)
if first list is empty then return $b_1, \ldots, b_\ell$
if second list is empty then return $a_1, \ldots, a_k$
if $a_1 \leq b_1$ then
  return $a_1 \circ R$Merge$(a_2, \ldots, a_k, b_1, \ldots, b_\ell)$
else
  return $b_1 \circ R$Merge$(a_1, \ldots, a_k, b_2, \ldots, b_\ell)$
Merging sorted lists: WHY

Claim: returns a sorted list containing all elements from either list

Proof by induction on n, the total input size

procedure RMerge(a₁, ..., aₖ, b₁, ..., bₗ: sorted lists)
  if first list is empty then return b₁, ..., bₗ
  if second list is empty then return a₁, ..., aₖ
  if a₁ ≤ b₁ then
    return a₁ \circ RMerge(a₂, ..., aₖ, b₁, ..., bₗ)
  else
    return b₁ \circ RMerge(a₁, ..., aₖ, b₂, ..., bₗ)

Base case: Suppose n=0. Then both lists are empty. So, in the first line we return the (trivially sorted) empty list containing all elements from the second list. But this list contains all (zero) elements from either list, because both lists are empty.
Merging sorted lists: WHY

Claim: returns a sorted list containing all elements from either list

Proof by induction on \( n \), the total input size

**Induction Step**: Suppose \( n \geq 1 \) and \( R\text{Merge}(a_1, \ldots, a_k, b_1, \ldots, b_l) \) returns a sorted list containing all elements from either list whenever \( k+l = n-1 \). We want to prove:

A. \( R\text{Merge}(a_1, \ldots, a_k, a_{k+1}, b_1, \ldots, b_l) \) returns a sorted list containing all elements from either list.
B. \( R\text{Merge}(a_1, \ldots, a_k, b_1, \ldots, b_l, b_{l+1}) \) returns a sorted list containing all elements from either list.
C. \( R\text{Merge}(a_1, \ldots, a_k, b_1, \ldots, b_l) \) returns a sorted list containing all elements from either list whenever \( k+l = n \).

**Procedure** \( R\text{Merge}(a_1, \ldots, a_k, b_1, \ldots, b_l) \): sorted lists
- If first list is empty then return \( b_1, \ldots, b_l \)
- If second list is empty then return \( a_1, \ldots, a_k \)
- If \( a_1 \leq b_1 \) then
  - return \( a_1 \circ R\text{Merge}(a_2, \ldots, a_k, b_1, \ldots, b_l) \)
- Else
  - return \( b_1 \circ R\text{Merge}(a_1, \ldots, a_k, b_2, \ldots, b_l) \)
Merging sorted lists: WHY

Claim: returns a sorted list containing all elements from either list

Proof by induction on n, the total input size

Induction Step: Suppose n>=1 and \( RMerge(a_1, \ldots, a_k, b_1, \ldots, b_{\ell}) \) returns a sorted list containing all elements from either list whenever \( k+\ell = n-1 \). We want to prove:

\[
RMerge(a_1, \ldots, a_k, b_1, \ldots, b_{\ell}) \text{ returns a sorted list containing all elements from either list whenever } k+\ell = n.
\]

Case 1: one of the lists is empty.  
Case 2: both lists are nonempty.
Merging sorted lists: WHY

Claim: returns a sorted list containing all elements from either list

Proof by induction on n, the total input size

Induction Step: Suppose n>=1 and \( RMerge(a_1, \ldots, a_k, b_1, \ldots, b_l) \) returns a sorted list containing all elements from either list whenever \( k+l = n-1 \). We want to prove:

\[
RMerge(a_1, \ldots, a_k, b_1, \ldots, b_l) \text{ returns a sorted list containing all elements from either list whenever } k+l = n.
\]

Case 1: one of the lists is empty: similar to base case. In first or second line return rest of list.
Claim: returns a sorted list containing all elements from either list

Proof by induction on $n$, the total input size

Case 2a: both lists nonempty and $a_1 \leq b_1$
Since both lists are sorted, this means $a_1$ is the smallest overall.

The total size of the input of $R\text{Merge}(a_2, \ldots, a_k, b_1, \ldots, b_l)$ is $(k-1) + l = n-1$ so by the IH, it returns a sorted list containing all elements from either list.

Adding $a_1$ to the start maintains the order and gives a sorted list with all elements. 😊
Merging sorted lists: WHY

Claim: returns a sorted list containing all elements from either list

Proof by induction on \( n \), the total input size

Case 2b: both lists nonempty and \( a_1 > b_1 \)
Same as before but reverse the roles of the lists. 😊

procedure \( RMerge(a_1, \ldots, a_k, b_1, \ldots, b_\ell: \text{sorted lists}) \)
if first list is empty then return \( b_1, \ldots, b_\ell \)
if second list is empty then return \( a_1, \ldots, a_k \)
if \( a_1 \leq b_1 \) then
    return \( a_1 \circ RMerge(a_2, \ldots, a_k, b_1, \ldots, b_\ell) \)
else
    return \( b_1 \circ RMerge(a_1, \ldots, a_k, b_2, \ldots, b_\ell) \)
Merging sorted lists: WHEN

procedure $R\text{Merge}(a_1, \ldots, a_k, b_1, \ldots, b_\ell$: sorted lists)

$\theta(1)$ if first list is empty then return $b_1, \ldots, b_\ell$

$\theta(1)$ if second list is empty then return $a_1, \ldots, a_k$

if $a_1 \leq b_1$ then

return $a_1 \circ R\text{Merge}(a_2, \ldots, a_k, b_1, \ldots, b_\ell)$

else

return $b_1 \circ R\text{Merge}(a_1, \ldots, a_k, b_2, \ldots, b_\ell)$

If $T(n)$ is the time taken by $R\text{Merge}$ on input of total size $n$,

$T(0) = c$

$T(n) = T(n-1) + c'$

where $c, c'$ are some constants
If $T(n)$ is the time taken by $RMerge$ on input of total size $n$,

$$T(0) = c$$
$$T(n) = T(n-1) + c'$$

where $c$, $c'$ are some constants

What's a solution to this recurrence equation?

A. $T(n) \in O(T(n - 1))$
B. $T(n) \in O(n)$
C. $T(n) \in O(n^2)$
D. $T(n) \in O(2^n)$
E. None of the above.