“Notes” slides from before lecture

CSE 21, Winter 2017, Section A00

Lecture 5 Notes

Class URL: http://vlsicad.ucsd.edu/courses/cse21-w17/
• HW1: 72-hour window for regrade requests ends tomorrow
• HW3 is published
  – LaTeX source is available, along with a link to “LaTeX Resources”
  – Discussion and Friday evening will mention “Master Theorem”
  – Look for some kind of “Anatomy of the Master Theorem” handout from me
• Discussion Worksheet (#2) late submissions
  – Team’s decision is to not deviate from published policy
  – Even if you have 0 worksheet credits so far, you can collect 7 out of 8 remaining to claim the 5% of your grade
  – Set some kind of alarm or notification for yourself (please….!)
• MT1 will be on Monday of Week 5 = 12 days from today
  – Will have two copies of the review session
  – Expect practice MT posted ~4-5 days before the review session
  – Look for polling in Piazza @177 re review session dates

• Today: Day 7 + Day 8 of posted slides
• Any logistic, other issues?
Intersecting sorted lists: WHEN

More careful analysis ...

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

Intersect \( (a_1, \ldots, a_n, b_1, \ldots, b_n) \)

\[
i := 1
\]

for \( j := 1 \) to \( n \)

\[
\text{while } (b_j > a_i \text{ and } i \leq n) \\
i := i + 1
\]

if \( i > n \) then return false

if \( b_j = a_i \) then return true

return false

\( \star \) Note: \( i, j \) can each advance only \( O(n) \) times!
Intersecting sorted lists: WHEN

More careful analysis ...

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

\[
i := 1
\]

\[
\text{for } j := 1 \text{ to } n \quad \text{(total times)}
\]

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

\[
i := i + 1
\]

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

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

\[
\text{return } false
\]

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

More careful analysis …

Intersect\((a_1, \ldots, a_n, b_1, \ldots, b_n)\)

\[
\begin{align*}
i & := 1 \\
\text{for } j & := 1 \text{ to } n \\
\quad \text{while } (b_j > a_i \text{ and } i \leq n) \\
\quad & \quad i := i + 1 \\
\quad \text{if } i > n \text{ then return } false \\
\quad \text{if } b_j = a_i \text{ then return } true \\
\text{return } false
\end{align*}
\]

This executes \(O(n)\) times total (across all iterations of for loop)

Key takeaways from end of last time:

\(O(n^2)\) was not tight

Total: \(\Theta(n)\)

Be careful: product rule isn't always tight!
A Preview: **MERGING** Two Sorted Lists

**Sorted List X:** \(x[1], x[2], \ldots, x[m]\)

**Sorted List Y:** \(y[1], y[2], \ldots, y[n]\)

**Goal:** Merge two sorted lists into one sorted list

“pointer walking” \(\ldots\): if \(y[i] < x[j]\) then compare \(y[i+1]\) and \(x[j]\), else compare \(y[i]\) and \(x[j+1]\)

| Comparisons | 1-4, | 2-4, | 3-4, | 7-4, | 7-5, etc. | (same idea as before !) |
|--------------|------|------|------|------|---------|

After comparing \(x[1]\) and \(y[1]\), we advance in \(x[.]\) and \(y[.]\) a total of \((m-1) + (n-1)\) times

\[\Rightarrow m + n - 1\] comparisons (edges between red and blue) = **linear time**
MERGESORT Tree of Recursive Subproblems

DIVIDE

log n

5 1 8 3 7 4 6 2

5 1 8 3
5 1
5

7 4 6 2
7 4
7

1 2 3 4 5 6 7 8
1 3 5 8
1 5
1

2 4 6 7
2 6
2

CONQUER, COMBINE

log n

(total of list sizes at any level) = \( n \)

\( n \) comparisons per level) X (log n levels) = (n log n runtime)
Today’s Plan

From last time:
- intersecting sorted lists and tight bounds

New topic: **Recursion**
- recursive algorithms
- correctness of recursive algorithms
- solving recurrence relations

> e.g., “divide and conquer” algorithms
  - matrix multiplication
  - FFT
  - sorting
  - long-number multiplication
  - closest pair
  - median-finding (in CSE 21)
What is recursion?

- Self-referential
- Method that calls itself

```python
def factorial(n):
    if n == 1:
        return 1
    result = factorial(n-1) * n
    return result
```
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
Problem: Given a string of 0s and 1s

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

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

\[
\begin{align*}
O(n) \quad \text{for } i & = 1 \ \text{to} \ n-1 \\
O(1) \quad \text{if } ((b_i = 0) \ \text{and} \ (b_{i+1} = 0)) \\
\text{count} & = \text{count} + 1
\end{align*}
\]
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

\( O(n) \), iterative
An **Iterative Algorithm:**
Step through each position and see if pattern starts there.

```plaintext
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 \( (b_i = 0 \text{ and } b_{i+1} = 0) \) 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 $n$.

*weak/ordinary induction*

*strong induction*
When should I use strong induction?

- Ordinary
  - Problem size "drops by 1" in recursive call

- Strong
  - Problem size "drops by more than 1"
  - "Knowing about n is enough to know about n+1"
  - "Knowing about n is NOT enough to know about n+1"
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.
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(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)

Goal: Prove that for any string b₁, b₂, b₃, ... bₙ,
countDoubleRec(b₁, b₂, b₃, ... bₙ) = 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, countDoubleRec(b₁, b₂, b₃, . . . bₖ) = the number of places the substring
00 occurs.

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

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)

Case 1: b₁ = 0 and b₂ = 0: countDoubleRec(b₁, b₂, b₃, ... bₖ₊₁) = 1 + countDoubleRec(b₂, b₃, ... bₖ₊₁) = 1 + the number of occurrences of 00 in b₂, b₃, ... bₖ₊₁ = one occurrence of 00 in first two positions + number of occurrences in later appearances.

Case 2: otherwise: countDoubleRec(b₁, b₂, b₃, ... bₖ₊₁) = countDoubleRec(b₂, b₃, ... bₖ₊₁) = the number of occurrences of 00 in b₂, b₃, ... bₖ₊₁ = the number of occurrences starting at the second position = the total number of occurrences since the first two are not an occurrence.
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 \textit{countDoubleRec}(b_1, \ldots, b_n: each 0 or 1)
  if \( n < 2 \) then return 0 = C''
  if \((b_1 = 0 \text{ and } b_2 = 0)\) then return 1 + \textit{countDoubleRec}(b_2, \ldots, b_n)
  return \textit{countDoubleRec}(b_2, \ldots, b_n)

\[ T(n) = T(n-1) + C \quad (n \geq 2) \]

How long does this algorithm take?

\( T(n) \) = time taken on input of size \( n \)

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 $f(n-1)$, $f(n-2)$, $f(n-3)$....

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(b_1, \ldots, b_n : 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)

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:

\[ T(n) = c(n-1) + d \quad n \geq 1 \]
factorial?
Fibonacci?
Mergesort?
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? \( T(n) = \# \ of \ moves \ to \ solve \ puzzle \ with \ n \ disks \)
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. $T(0) = 0$
B. $T(2) = 2$

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

$T(n) = \# \text{ of moves to solve puzzle with } n \text{ disks}$
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>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>
<tr>
<td>$n$</td>
<td>(?)</td>
</tr>
</tbody>
</table>

Recurrence for $T(n)$:

\[
T(n) = 2T(n-1) + 1 \\
T(1) = 1 \quad \text{(b.c.)}
\]

\[
2^n - 1 = T(n) \quad \text{"guess"}
\]
Towers of Hanoi: WHEN

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 $T(n)$:

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

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

Is there a pattern we can guess?
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 \( T(1) = 2^1 - 1 = 2 - 1 = 1 \). ☺
Towers of Hanoi: WHEN

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,

\[
T(n) = 2T(n-1) + 1 = 2(2^{n-1} - 1) + 1 = 2^n - 2 + 1 = 2^n - 1.
\]
Towers of Hanoi: WHEN

Another method: “UNRAVEL” the recurrence:

Recurrence for $T(n)$:

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

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

$= 2\left(2T(n - 2) + 1\right) + 1 = 4T(n - 2) + 2 + 1$

$= 4\left(2T(n - 3) + 1\right) + 2 + 1 = 8T(n - 3) + 4 + 2 + 1$

$\vdots$

$= 2^kT(n - k) + 2^{k-1} + \cdots + 2 + 1 = 2^kT(n - k) + (2^k - 1)$

$\vdots$

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

$= 2^{n-1} + 2^{n-1} - 1 = 2^n - 1.$
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></td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>01, 10, 11</td>
<td>00, 000, 001, 100</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>010, 011, 101, 110, 111</td>
<td>8</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??

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

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

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

"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??**

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

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

\[ B(n-1) \quad 1 \text{__________} \quad \text{or} \quad 01 \text{__________} \quad B(n-2) \]

"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 Monday of 5\(^{th}\) Week!

Keep an eye out for practice MT, review session, etc.
End of Day 7 (MWF schedule)
Start of Day 8 (not repeated from Day 7)  
(MWF schedule)
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
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.

$T(n) = 2^n - 1$  
*Tower of Hanoi*

2. **Unravel**

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

$T(n) = c(n-1) + d$  
*CountDoubleRec*
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)!

Example Recurrence

\[ f(n) = 2f(n-1) + 1 \]
\[ f(n) = f(n-1) + f(n-2) + 17 \]
\[ f(n) = f(n-3) + 1 \]

# Base Cases needed

- One base case
- Two base cases
- Three base cases

How many are needed?
Another Example - Pizza

You cut a pizza by making cuts. **Each cut is 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. \( P(2) = 1 \)

No, leads to wrong values \( P(1) = 3, P(2) = 5, \ldots \)

This works
Merging sorted lists: WHAT

Given two \textbf{sorted} lists

\begin{align*}
a_1 & \quad a_2 & \quad a_3 & \quad \ldots & \quad a_k \\
b_1 & \quad b_2 & \quad b_3 & \quad \ldots & \quad b_l
\end{align*}

produce a \textbf{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.

*Design a recursive algorithm to solve this problem*
Merging sorted lists: HOW

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

\[
R\text{Merge} \left( \frac{1,3,4,5,7}{4}, \frac{0,10,13}{3} \right) = 0 \circ R\text{Merge} \left( \frac{1,3,4,5,7}{3}, \frac{10,13}{3} \right)
\]

```
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 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)
```

concatenate
Merging sorted lists: WHY

Similar to Rosen p. 369

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

procedure \( R\text{Merge}(a_1, \ldots, a_k, b_1, \ldots, b_\ell: \text{sorted lists}) \)
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 \( a_1 \circ R\text{Merge}(a_2, \ldots, a_k, b_1, \ldots, b_\ell) \)
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

Meaning: “all elements from both lists”

\( \text{Note again: } n = k + \ell \)
Merging sorted lists: WHY

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

Proof by induction on n, the total input size

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 \(RMerge(a_1, \ldots, a_k, b_1, \ldots, b_\ell):\) 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 \(a_1 \circ RMerge(a_2, \ldots, a_k, b_1, \ldots, b_\ell)\)
else
return \(b_1 \circ RMerge(a_1, \ldots, a_k, b_2, \ldots, b_\ell)\)
Merging sorted lists: WHY

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

Proof by induction on n, the total input size

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.

```
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 o RMerge(a₂, ..., aₖ, b₁, ..., bₗ)
    else
        return b₁ o RMerge(a₁, ..., aₖ, b₂, ..., bₗ)
```
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 \( \text{RMerge}(a_1, \ldots, a_k, b_1, \ldots, b_l) \) returns a sorted list containing all elements from either list whenever \( k+l = n-1 \). We want to prove:

A. \( \text{RMerge}(a_1, \ldots, a_k, a_{k+1}, b_1, \ldots, b_l) \) returns a sorted list containing all elements from either list.
B. \( \text{RMerge}(a_1, \ldots, a_k, b_1, \ldots, b_l, b_{l+1}) \) returns a sorted list containing all elements from either list.
C. \( \text{RMerge}(a_1, \ldots, a_k, b_1, \ldots, b_l) \) returns a sorted list containing all elements from either list whenever \( k+l = n \).
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 \( RMerge(a_1, \ldots, a_k, b_1, \ldots, b_\ell) \) 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 \( \text{RMerge}(a_1,\ldots,a_k,b_1,\ldots,b_\ell) \) returns a sorted list containing all elements from either list whenever \( k+\ell = n-1 \). We want to prove:

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

Case 1: one of the lists is empty: similar to base case. In first or second line return rest of list.
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 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** \( R\text{Merge}(a_1, \ldots, a_k, b_1, \ldots, b_\ell : \text{sorted lists}) \)

- 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

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

else

\[ \text{return } b_1 \circ R\text{Merge}(a_1, \ldots, a_k, b_2, \ldots, b_\ell) \]
Merging sorted lists: WHEN

procedure \( RMerge(a_1, \ldots, a_k, b_1, \ldots, b_\ell: \text{sorted lists}) \)
\( \theta(1) \)
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
\[ \text{return } a_1 \circ RMerge(a_2, \ldots, a_k, b_1, \ldots, b_\ell) \]
else
\[ \text{return } b_1 \circ RMerge(a_1, \ldots, a_k, b_2, \ldots, b_\ell) \]

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
If $T(n)$ is the time taken by $RMerge$ on input of total size $n$,

$$
\begin{align*}
T(0) &= c \\
T(n) &= T(n-1) + c'
\end{align*}
$$

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.
Announcements

Midterm #1
Monday Feb. 6 in class.
Bring Student ID.
Note sheet. No calculators.
Covers through recursion (HWs 1, 2, 3).

Review Sessions
covering the Practice Midterm
Current plan: ???
Times, Locations TBD
(Piazza @177 poll!)

HW3
Due next Tuesday!

FUN! 😊
Let $P(n)$ = the maximum possible number of pieces into which a circular pizza can be cut by $n$ straight-line cuts (not necessarily diameters of the circle). Base cases: $P(0) = 1$, $P(1) = 2$. State a recurrence for $P(n)$ and prove a closed-form solution.

In 1979, the NBA introduced the 3-point shot. Before that time, points could be scored only on free throws (1 point) and on field goals (2 points). Let $W(n)$ = the number of distinct ways in which it is possible to reach a total of $n$ points scored, before 1979. Base cases: $W(0) = 1$, $W(1) = 1$. State a recurrence for $W(n)$ and prove a closed-form solution.