<table>
<thead>
<tr>
<th>Lecture</th>
<th>Instructor</th>
<th>Time</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tiefenbruck</td>
<td>MWF 9-9:50am</td>
<td>Center 212</td>
</tr>
<tr>
<td>B</td>
<td>Jones</td>
<td>MWF 2-2:50pm</td>
<td>Center 214</td>
</tr>
<tr>
<td>C</td>
<td>Tiefenbruck</td>
<td>MWF 11-11:50am</td>
<td>Center 212</td>
</tr>
</tbody>
</table>

http://cseweb.ucsd.edu/classes/wi16/cse21-abc/

January 22, 2016
Recursion

Last Time
1. Recursive algorithms and correctness
2. Coming up with recurrences

Today
1. Recursive algorithms and time analysis
2. Solving recurrences
3. Important example: Merge

In the textbook: Sections 5.4, 8.3
Exponentiation: WHEN

Basic operation: multiplication

M(n) = number of multiplications we need to compute $2^n$ with the algorithm $powerBase2(n)$

What's M(0)?
A. 0
B. 1
C. 2
D. 4
E. None of the above.

```plaintext
procedure powerBase2(n : n a nonnegative integer)
    if n = 0 then return 1
    else return $2 \cdot powerBase2(n - 1)$
```
Exponentiation: WHEN

Basic operation: multiplication

\( M(n) = \) number of multiplications we need to compute \( 2^n \) with the algorithm \( \text{powerBase2}(n) \)

What's \( M(1) \)?
A. 0
B. 1
C. 2
D. 4
E. None of the above.

```plaintext
procedure \text{powerBase2}(n: \text{ } n \text{ a nonnegative integer})
    if \( n = 0 \) then return 1
    else return \( 2 \cdot \text{powerBase2}(n - 1) \)
```
Exponentiation: WHEN

Basic operation: multiplication

M(n) = number of multiplications we need to compute $2^n$ with the algorithm $powerBase2(n)$

What's M(n)?
A. 0
B. n
C. n-1
D. $2(n-1)$
E. None of the above.

procedure $powerBase2(n : n$ a nonnegative integer)

if $n = 0$ then return 1
else return $2 \cdot powerBase2(n - 1)$
Exponentiation: WHEN

Basic operation: multiplication

\[ M(n) = \text{number of multiplications we need to compute } 2^n \text{ with the algorithm } powerBase2(n) \]

What's \( M(n) \)?
A. 0
B. \( n \)
C. \( n - 1 \)
D. \( 2(n-1) \)
E. None of the above.

**procedure** \( powerBase2(n : n \text{ a nonnegative integer}) \)

if \( n = 0 \) then return 1

else return \( 2 \cdot powerBase2(n - 1) \)

Takes \( M(n-1) \) many multiplications
Exponentiation: WHEN

Basic operation: multiplication

\[ M(n) = \text{number of multiplications we need to compute } 2^n \text{ with the algorithm } \text{powerBase2}(n) \]

Recurrence for \( M(n) \):

\[
\begin{align*}
M(n) &= M(n-1) + 1 \\
M(0) &= 0
\end{align*}
\]

\begin{pro}
 procedure \text{powerBase2}(n : n \text{ a nonnegative integer}) \\
 if \( n = 0 \) then return 1 \\
 else return 2 \cdot \text{powerBase2}(n - 1) 
\end{pro}

One new multiplication

Takes \( M(n-1) \) many multiplications
Exponentiation: WHEN

But what's the value of $M(n)$?

\[ M(n) = M(n - 1) + 1 \]
\[ = (M(n - 2) + 1) + 1 = M(n - 2) + 2 \]
\[ = (M(n - 3) + 1) + 2 = M(n - 3) + 3 \]
\[ \vdots \]
\[ = M(n - k) + k \]
\[ \vdots \]
\[ = M(0) + n \]
\[ = n \]

Recurrence for $M(n)$:

- $M(n) = M(n-1) + 1$
- $M(0) = 0$

"Unraveling the recurrence"
Exponentiation: WHEN

But what's the value of $M(n)$?

$$M(n) = M(n - 1) + 1$$

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

$$= \left( M(n - 3) + 1 \right) + 2 = M(n - 3) + 3$$

\vdots

$$= M(n - k) + k$$

\vdots

$$= M(0) + n$$

$$= n$$

Recurrence for $M(n)$:

$$M(n) = M(n-1) + 1$$

$$M(0) = 0$$

Exponentiation is $O(n)$
Towers of Hanoi: WHEN

* Move the stack of the smallest n-1 disks to an empty pole.
* Move the largest disk to the remaining empty pole.
* Move the stack of the smallest n-1 disks to the pole with the largest disk.

What is the recurrence for $T(n)$?

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

What is the base case?

A. $T(1) = 1$
B. $T(1) = 2$
C. $T(0) = 0$
D. $T(2) = 2$
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></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 $T(n)$:

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

$T(1) = 1$
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>
<tr>
<td>n</td>
<td>??</td>
</tr>
</tbody>
</table>

Recurrence for $T(n)$:

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

$T(1) = 1$

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 $n=1$ to the formula gives $2^1 - 1 = 2 - 1 = 1$. 😊
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,

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

IH
Towers of Hanoi: WHEN

Instead … let's unravel the recurrence:

Recurrence for $T(n)$:

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

What's the result of merging the lists 1,4,8 and 2, 3, 10, 20

A. 1,4,8,2,3,10,20
B. 1,2,4,3,8,10,20
C. 1,2,3,4,8,10,20
D. 20,10,8,4,3,2,1
E. None 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 an algorithm to solve this problem*
Merging sorted lists: HOW

An iterative algorithm

Go through each position between 1 and n and decide which element goes in by looking at next possible elements in each list.
Merging sorted lists

An iterative algorithm

Go through each position between 1 and \(n\) and decide which element goes in by looking at next possible elements in each list.

\[
\begin{align*}
\text{procedure } IMerge(a_1, \ldots, a_k, b_1, \ldots, b_\ell: \text{ sorted lists}) \\
n &:= k + \ell \\
i &:= 1 \\
j &:= 1 \\
\text{for } t := 1 \text{ to } n \\
&\quad \text{if } i > k \text{ then} \\
&\quad \quad c_t := b_j \\
&\quad \quad j := j + 1 \\
&\quad \text{else if } j > \ell \text{ then} \\
&\quad \quad c_t := a_i \\
&\quad \quad i := i + 1 \\
&\quad \text{else if } a_i \leq b_j \text{ then} \\
&\quad \quad c_t := a_i \\
&\quad \quad i := i + 1 \\
&\quad \text{else} \\
&\quad \quad c_t := b_j \\
&\quad \quad j := j + 1 \\
\text{return } c_1, \ldots, c_t
\end{align*}
\]
Merging sorted lists

WHY Correctness

Loop invariant: After \( t \) iterations, \( c_1, \ldots, c_t \) are the \( t \) smallest elements of the union, they are sorted, and they contain all elements in \( a_1, \ldots, a_{i-1}, b_1, \ldots, b_{j-1} \).

```
procedure IMerge(a_1, \ldots, a_k, b_1, \ldots, b_\ell: sorted lists)
    n := k + \ell
    i := 1
    j := 1
    for t := 1 to n
        if i > k then
            c_t := b_j
            j := j + 1
        else if j > \ell then
            c_t := a_i
            i := i + 1
        else if a_i \leq b_j then
            c_t := a_i
            i := i + 1
        else
            c_t := b_j
            j := j + 1
    return c_1, \ldots, c_t
```
Merging sorted lists

WHEN Time analysis

What’s the best-case situation?

A. $a_k < b_1$
B. $a_1 < b_1$
C. $b_1 < a_k$
D. $b_1 < a_k$
E. None of the above.

procedure IMerge($a_1, \ldots, a_k, b_1, \ldots, b_\ell$: sorted lists)

$n := k + \ell$
$i := 1$
$j := 1$

for $t := 1$ to $n$

\[ \text{if } i > k \text{ then} \]
\[ c_t := b_j \]
\[ j := j + 1 \]
\[ \text{else if } j > \ell \text{ then} \]
\[ c_t := a_i \]
\[ i := i + 1 \]
\[ \text{else if } a_i \leq b_j \text{ then} \]
\[ c_t := a_i \]
\[ i := i + 1 \]
\[ \text{else} \]
\[ c_t := b_j \]
\[ j := j + 1 \]

return $c_1, \ldots, c_t$
WHEN Time analysis

Work from the inside out

What's the big-θ class of the runtime (including all operations)?

A. θ(1)
B. θ(log n)
C. θ(n)
D. θ(n log n)
E. None of the above.
Merging sorted lists

WHEN Time analysis

Work from the inside out

procedure IMerge(a₁, ..., aₖ, b₁, ..., bₗ: sorted lists)
    n := k + ℓ
    i := 1
    j := 1
    for t := 1 to n
        if i > k then
            cₜ := bⱼ
            j := j + 1
        else if j > ℓ then
            cₜ := aᵢ
            i := i + 1
        else if aᵢ ≤ bⱼ then
            cₜ := aᵢ
            i := i + 1
        else
            cₜ := bⱼ
            j := j + 1
    return c₁, ..., cₜ
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 \circ RMerge(a₂, ..., aₖ, b₁, ..., bₗ)
  else
    return b₁ \circ RMerge(a₁, ..., aₖ, b₂, ..., bₗ)
```
Merging sorted lists: WHY

Similar to Rosen p. 369

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

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

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

```
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: WHY

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

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

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

What is the \textbf{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.
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 \( RMerge(a_1, \ldots, a_k, b_1, \ldots, b_\ell): \) 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

**Induction Step**: Suppose n\(\geq 1\) 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:

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

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

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

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

**Case 1**: one of the lists is empty: 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 not bigger than
* any of the elements in the list $a_2, \ldots, a_k$
* any of the elements in the list $b_1, \ldots, b_l$

The total size of the input of $RMerge(a_2, \ldots, a_k, b_1, \ldots, b_l)$ is $(k-1) + l = n-1$ so by the IH, it returns a sorted list containing all elements from either list.
Prepending $a_1$ 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

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 $b_1 \circ RMerge(a_1, \ldots, a_k, b_2, \ldots, b_\ell)$

Are we done with the proof?
A. Yes
B. No
C. ??
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 $R\text{Merge}(a_1, \ldots, a_k, b_1, \ldots, b_\ell$: 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 $b_1 \circ R\text{Merge}(a_1, \ldots, a_k, b_2, \ldots, b_\ell)$

Case 2b: both lists nonempty and $a_1 > b_1$
Same as before but reverse the roles of the lists. 😊
Merging sorted lists: WHEN

\[
\begin{align*}
\text{procedure } \text{RMerge}(a_1, \ldots, a_k, b_1, \ldots, b_\ell : \text{sorted lists}) \\
\theta(1) \quad \text{if first list is empty then return } b_1, \ldots, b_\ell \\
\theta(1) \quad \text{if second list is empty then return } a_1, \ldots, a_k \\
\quad \text{if } a_1 \leq b_1 \text{ then} \\
\quad \quad \text{return } a_1 \circ \text{RMerge}(a_2, \ldots, a_k, b_1, \ldots, b_\ell) \\
\quad \text{else} \\
\quad \quad \text{return } b_1 \circ \text{RMerge}(a_1, \ldots, a_k, b_2, \ldots, b_\ell)
\end{align*}
\]

One recursive call

If \( T(n) \) is the time taken by \text{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.
Merging sorted lists: WHEN

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.
Reminders

HW 4 due **Wednesday 11:59pm** via Gradescope.

**Midterm 1**: one week from today, Friday, Jan 29 in class

* Practice midterm on website/Piazza.
* Review sessions Tuesday & Wednesday: see website/Piazza.
* **Seating chart coming soon on website/Piazza.**
* One double-sided handwritten note sheet allowed.
* If you have AFA letter, see me as soon as possible.