<table>
<thead>
<tr>
<th>Lecture</th>
<th>Instructor</th>
<th>Day</th>
<th>Time</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lecture A</td>
<td>Tiefenbruck</td>
<td>Tu, Th 11am-12:20pm</td>
<td>Center 119</td>
<td></td>
</tr>
<tr>
<td>Lecture B</td>
<td>Tiefenbruck</td>
<td>Tu, Th 9:30am-10:50am</td>
<td>Center 119</td>
<td></td>
</tr>
<tr>
<td>Lecture C</td>
<td>Minnes</td>
<td>Tu, Th 3:30pm-4:50pm</td>
<td>WLH 2005</td>
<td></td>
</tr>
</tbody>
</table>

http://cseweb.ucsd.edu/classes/fa15/cse21-abc/

Oct 13, 2015
Reminders

Midterm 1: Tuesday October 20.

* In class. Exam will be 1 hour, 15 minutes. (5 minutes for hand in)
* **Practice midterm** available on website/ Piazza.
* Review sessions Saturday & Sunday: see website.
* Extra office hours over the weekend.
* Monday discussion session review.
* **Seating chart on website/ Piazza.**
* 1 handwritten note sheet allowed.
* If you have AFA letter, see me as soon as possible.
Today's plan

1. Merge: iterative and recursive algorithms

2. Proving correctness & doing time analysis for divide & conquer algorithms

*In the textbook: Sections 5.4, 8.3*
Merging sorted arrays WHAT

Given two sorted lists of distinct elements

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

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 arrays WHAT

Given two \textbf{sorted} lists of distinct elements

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

produce a \textbf{sorted} list of length $n=k+l$ which contains all their elements.

\textbf{Design an algorithm to solve this problem}
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 arrays

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.

procedure IMerge\((a_1, \ldots, a_k, b_1, \ldots, b_l : \text{sorted arrays})\)

\[
\begin{align*}
    n &:= k + l \\
    i &:= 1 \\
    j &:= 1 \\
    \text{for } t = 1 \text{ to } n \\
    \quad \begin{cases}
    i > k \text{ then} & c_t := b_j \\
    j := j + 1 \\
    \quad \begin{cases}
    j > l \text{ then} & c_t := a_i \\
    \quad i := i + 1 \\
    \quad \begin{cases}
    a_i \leq b_j \text{ then} & c_t := a_i \\
    \quad i := i + 1 \\
    \quad \text{else} & c_t := b_j \\
    \quad j := j + 1 \\
    \end{cases}
    \end{cases}
    \end{cases}
\end{align*}
\]

return \(c_1, \ldots, c_n\)
Merging sorted arrays

**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 $a_1, \ldots, a_{i-1}, b_1, \ldots, b_{j-1}$

```latex
\begin{align*}
\text{procedure } & \text{Merge}(a_1, \ldots, a_k, b_1, \ldots, b_l : \text{sorted arrays}) \\
& n := k + l \\
& 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{if } j > l \text{ then} \\
& \quad \quad c_t := a_i \\
& \quad \quad i := i + 1 \\
& \quad \text{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_n
\end{align*}
```
Merging sorted arrays

WHEN Time analysis

What's the best-case situation?

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

```
procedure IMerge($c_1$, $c_2$ : sorted arrays)
    $n := k + l$
    $i := 1$
    $j := 1$
    for $t = 1$ to $n$
        if $i > k$ then
            $c_t := b_j$
            $j := j + 1$
        if $j > l$ then
            $c_t := a_i$
            $i := i + 1$
        if $a_i \leq b_j$ then
            $c_t := a_i$
            $i := i + 1$
        else
            $c_t := b_j$
            $j := j + 1$
    return $c_1, \ldots, c_n$
```
Merging sorted arrays

WHEN  Time analysis

Work from the inside out

What's the big-$\theta$ class of the runtime (including all operations)?

A. $\theta(1)$
B. $\theta(\log n)$
C. $\theta(n)$
D. $\theta(n \log n)$
E. None of the above.

```
procedure IMerge(a_1, \ldots, a_k, b_1, \ldots, b_l : sorted arrays)
    n := k + l
    i := 1
    j := 1
    for t = 1 to n
        if i > k then
            c_t := b_j
            j := j + 1
        if j > l then
            c_t := a_i
            i := i + 1
        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_n
```
Merging sorted arrays

WHEN  \textit{Time analysis}

Work from the inside out

procedure \textsc{Merge}(a_1, \ldots, a_k, b_1, \ldots, b_l : \text{sorted arrays})

\begin{align*}
  n & := k + l \\
  i & := 1 \\
  j & := 1 \\
  \textbf{for } t = 1 \textbf{ to } n & \\
\end{align*}

if \( i > k \) then
\begin{align*}
  c_t & := b_j \\
  j & := j + 1
\end{align*}

if \( j > l \) then
\begin{align*}
  c_t & := a_i \\
  i & := i + 1
\end{align*}

if \( a_i \leq b_j \) then
\begin{align*}
  c_t & := a_i \\
  i & := i + 1
\end{align*}

else
\begin{align*}
  c_t & := b_j \\
  j & := j + 1
\end{align*}

\textbf{return} \( c_1, \ldots, c_n \)
A recursive algorithm
Focus on merging head elements, then rest.

```plaintext
procedure Merge(a₁, ..., aₖ, b₁, ..., bₙ : sorted arrays)
    if first list is empty then return b₁, ..., bₙ
    if second list is empty then return a₁, ..., aₖ
    if a₁ ≤ b₁ then
        return a₁ ∪ Merge(a₂, ..., aₖ, b₁, ..., bₙ)
    else
        return b₁ ∪ Merge(a₁, ..., aₖ, b₂, ..., bₙ)
```

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

```
procedure RMerge(a_1, \ldots, a_k, b_1, \ldots, b_l : sorted arrays)
  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 RMerge(a_2, \ldots, a_k, b_1, \ldots, b_l)
  else
    return b_1 \circ RMerge(a_1, \ldots, a_k, b_2, \ldots, b_l)
```

Claim that result is a sorted list containing all elements from either list

Proof by induction on n, the total input size
Merging sorted arrays WHY

Claim that result is a sorted list containing all elements from either list

Proof by induction on n, the total input size

procedure \( RMerge(a_1, \ldots, a_k, b_1, \ldots, b_l : \text{sorted arrays}) \)

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 \( a_1 \circ RMerge(a_2, \ldots, a_k, b_1, \ldots, b_l) \)

else

return \( b_1 \circ RMerge(a_1, \ldots, a_k, b_2, \ldots, b_l) \)

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.
Merging sorted arrays WHY

Claim that result is a sorted list containing all elements from either list

**procedure** $RMerge(a_1, \ldots, a_k, b_1, \ldots, b_l : \text{sorted arrays})$

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 $a_1 \circ RMerge(a_2, \ldots, a_k, b_1, \ldots, b_l)$

else

    return $b_1 \circ RMerge(a_1, \ldots, a_k, b_2, \ldots, b_l)$

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.
Merging sorted arrays WHY

procedure \text{RMerge}(a_1, \ldots, a_k, b_1, \ldots, b_l : \text{sorted arrays})

\begin{itemize}
  \item if first list is empty then return $b_1, \ldots, b_l$
  \item if second list is empty then return $a_1, \ldots, a_k$
  \item if $a_1 \leq b_1$ then
    \begin{itemize}
      \item return $a_1 \circ \text{RMerge}(a_2, \ldots, a_k, b_1, \ldots, b_l)$
    \end{itemize}
  \item else
    \begin{itemize}
      \item return $b_1 \circ \text{RMerge}(a_1, \ldots, a_k, b_2, \ldots, b_l)$
    \end{itemize}
\end{itemize}

\textbf{Induction Step}: Suppose $n>1$ 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:

\begin{enumerate}
  \item $\text{RMerge}(a_1, \ldots, a_k, a_{k+1}, b_1, \ldots, b_l)$ returns a sorted list containing all elements from either list.
  \item $\text{RMerge}(a_1, \ldots, a_k, b_1, \ldots, b_l, b_{l+1})$ returns a sorted list containing all elements from either list.
  \item $\text{RMerge}(a_1, \ldots, a_k, b_1, \ldots, b_l)$ returns a sorted list containing all elements from either list whenever $k+l = n$.
\end{enumerate}
Merging sorted arrays WHY

Claim that result is a sorted list containing all elements from either list

**procedure** $RMerge(a_1, \ldots, a_k, b_1, \ldots, b_l : \text{sorted arrays})$

- 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 $a_1 \circ RMerge(a_2, \ldots, a_k, b_1, \ldots, b_l)$
- else
  return $b_1 \circ RMerge(a_1, \ldots, a_k, b_2, \ldots, b_l)$

**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:

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

**Case 1:** one of the lists is empty.

**Case 2:** both lists are nonempty.
Merging sorted arrays WHY

Claim that result is a sorted list containing all elements from either list

procedure \( RMerge(a_1, \ldots, a_k, b_1, \ldots, b_l) \) : sorted arrays

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 \( a_1 \circ RMerge(a_2, \ldots, a_k, b_1, \ldots, b_l) \)
else
    return \( b_1 \circ RMerge(a_1, \ldots, a_k, b_2, \ldots, b_l) \)

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:

\[ RMerge(a_1, \ldots, a_k, b_1, \ldots, b_l) \] 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.
Merging sorted arrays WHY

Claim that result is a sorted list containing all elements from either list

**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. 😊

**procedure** \( RMerge(a_1, \ldots, a_k, b_1, \ldots, b_l : \text{ sorted arrays}) \)

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** \( a_1 \circ RMerge(a_2, \ldots, a_k, b_1, \ldots, b_l) \)

else

**return** \( b_1 \circ RMerge(a_1, \ldots, a_k, b_2, \ldots, b_l) \)
Merging sorted arrays WHY

Claim that result is a sorted list containing all elements from either list

procedure \textit{RMerge}(a_1, \ldots, a_k, b_1, \ldots, b_l : \text{sorted arrays})

\begin{align*}
\text{if first list is empty then return } b_1, \ldots, b_l \\
\text{if second list is empty then return } a_1, \ldots, a_k \\
\text{if } a_1 \leq b_1 \text{ then} \\
\quad \text{return } a_1 \circ \textit{RMerge}(a_2, \ldots, a_k, b_1, \ldots, b_l) \\
\text{else} \\
\quad \text{return } b_1 \circ \textit{RMerge}(a_1, \ldots, a_k, b_2, \ldots, b_l)
\end{align*}

Are we done with the proof?  
A. Yes  
B. No  
C. ??
Claim that result is a sorted list containing all elements from either list

procedure $RMerge(a_1, \ldots, a_k, b_1, \ldots, b_l : \text{sorted arrays})$

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 $a_1 \circ RMerge(a_2, \ldots, a_k, b_1, \ldots, b_l)$

else

return $b_1 \circ RMerge(a_1, \ldots, a_k, b_2, \ldots, b_l)$

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

procedure $R\text{Merge}(a_1, \ldots, a_k, b_1, \ldots, b_l : \text{sorted arrays})$

θ(1) if first list is empty then return $b_1, \ldots, b_l$

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

if $a_1 \leq b_1$ then

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

else

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

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

$T(1) = c$

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

where $c$, $c'$ are some constants
Merging sorted arrays WHEN

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

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

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.
Iterative strategies

* Selection sort (MinSort)
* Insertion sort
* Bubble sort
"We split into two groups and organized each of the groups, then got back together and figured out how to interleave the groups in order."
Merge Sort HOW

A divide & conquer (recursive) strategy:

**Divide** list into two sub-lists

**Recursively** sort each sublist

**Conquer** by merging the two sorted sublists into a single sorted list
procedure MergeSort(a_1, \ldots, a_n)
    if \ n > 1 \ then
        m := \lfloor n/2 \rfloor
        L_1 := a_1, \ldots, a_m
        L_2 := a_{m+1}, \ldots, a_n
        return \text{RMerge(} \text{MergeSort}(L_1), \text{MergeSort}(L_2) \text{)}
    else return \ a_1, \ldots, a_n
Merge Sort WHY

procedure MergeSort(a_1, ..., a_n)
    if \( n > 1 \) then
        \( m := \lfloor n/2 \rfloor \)
        \( L_1 := a_1, \ldots, a_m \)
        \( L_2 := a_{m+1}, \ldots, a_n \)
        return RMerge( MergeSort(L_1), MergeSort(L_2) )
    else return \( a_1, \ldots, a_n \)

Claim that result is a sorted list containing all elements.
Proof by strong induction on \( n \):

Why do we need strong induction?
A. Because we're breaking the list into two parts.
B. Because the input to the recursive function call is less than \( n \).
C. Because we're calling the function recursively twice.
D. Because we're using a subroutine, \( RMerge \).
E. None of the above.
Merge Sort WHY

Claim that result is a sorted list containing all elements.

Proof by strong induction on n:

**Base case**: Suppose \( n = 0 \).

Suppose \( n = 1 \).

procedure \( \text{MergeSort}(a_1, \ldots, a_n) \)

\[
\begin{align*}
\text{if} & \quad n > 1 \text{ then} \\
& \quad m := \lfloor n/2 \rfloor \\
& \quad L_1 := a_1, \ldots, a_m \\
& \quad L_2 := a_{m+1}, \ldots, a_n \\
& \quad \text{return } \text{RMerge}( \text{MergeSort}(L_1), \text{MergeSort}(L_2) ) \\
\text{else} & \quad \text{return } a_1, \ldots, a_n
\end{align*}
\]
**Merge Sort WHY**

```plaintext
procedure MergeSort(a₁, ..., aₙ)
    if n > 1 then
        m := ⌈n/2⌉
        L₁ := a₁, ..., aₘ
        L₂ := aₘ₊₁, ..., aₙ
        return RMerge( MergeSort(L₁), MergeSort(L₂) )
    else return a₁, ..., aₙ
```

**Claim that result is a sorted list containing all elements.**

Proof by **strong** induction on n:

**Base case**: Suppose n=0. Then, in the else branch, we return the empty list, (trivially) sorted.

Suppose n=1. Then, in the else branch, we return a₁, a (trivally) sorted list containing all elements. 😊
Merge Sort WHY

\begin{algorithm}
\begin{algorithmic}
\Procedure{MergeSort}{a_1, \ldots, a_n}
\If{$n > 1$}
    \State $m := \lfloor n/2 \rfloor$
    \State $L_1 := a_1, \ldots, a_m$
    \State $L_2 := a_{m+1}, \ldots, a_n$
    \State \textbf{return} $R\text{Merge}(\text{MergeSort}(L_1), \text{MergeSort}(L_2))$
\Else \textbf{return} $a_1, \ldots, a_n$
\EndIf
\EndProcedure
\end{algorithmic}
\end{algorithm}

**Claim** that result is a sorted list containing all elements.

**Induction step**: Suppose $n>1$. Assume, as the strong induction hypothesis, that $\text{MergeSort}$ correctly sorts all lists with $k$ elements, for $1 \leq k < n$.

Goal: prove that $\text{MergeSort}(a_1, \ldots, a_n)$ returns a sorted list containing all $n$ elements.
**Merge Sort WHY**

procedure $\text{MergeSort}(a_1, \ldots, a_n)$

if $n > 1$ then

$m := \lfloor n/2 \rfloor$

$L_1 := a_1, \ldots, a_m$

$L_2 := a_{m+1}, \ldots, a_n$

return $\text{RMerge}(\text{MergeSort}(L_1), \text{MergeSort}(L_2))$

else return $a_1, \ldots, a_n$

**IH:** $\text{MergeSort}$ correctly sorts all lists with $k$ elements, for $1 \leq k < n$

**Goal:** prove that $\text{MergeSort}(a_1, \ldots, a_n)$ returns a sorted list containing all $n$ elements.

Since $n > 1$, in the if branch we return $\text{RMerge}(\text{MergeSort}(L_1), \text{MergeSort}(L_2))$, where $L_1$ and $L_2$ each have no more than $(n/2) + 1$ elements and together they contain all elements.

By IH, each of $\text{MergeSort}(L_1)$ and $\text{MergeSort}(L_2)$ are sorted and by the correctness of $\text{Merge}$, the returned list is a sorted list containing all the elements. 😊
procedure $\text{MergeSort}(a_1, \ldots, a_n)$

if $n > 1$ then

$\theta(1) \quad m := \lfloor n/2 \rfloor$

$L_1 := a_1, \ldots, a_m$

$L_2 := a_{m+1}, \ldots, a_n$

$T_{\text{Merge}}(n/2 + n/2) \quad \text{return} \quad R\text{Merge}(\text{MergeSort}(L_1), \text{MergeSort}(L_2))$

else return $a_1, \ldots, a_n$

$T_{\text{MS}}(n/2) \quad T_{\text{MS}}(n/2)$

If $T_{\text{MS}}(n)$ is number of operations taken by $\text{MergeSort}$ on list of size $n$,

$T_{\text{MS}}(1) = c'$

$T_{\text{MS}}(n) = 2T_{\text{MS}}(n/2) + T_{\text{Merge}}(n) + c'' n$

where $c'$, $c''$ are some constants
Merge Sort WHEN

procedure MergeSort(a_1, \ldots, a_n)

if \( n > 1 \) then

\( \theta(1) \) \( m := \lfloor n/2 \rfloor \)

? \( L_1 := a_1, \ldots, a_m \)

? \( L_2 := a_{m+1}, \ldots, a_n \)

\[ T_{\text{Merge}}(n) \text{ is in } O(n) \]

\[ T_{\text{Merge}}(n/2 + n/2) \text{ return } R\text{Merge}(\text{MergeSort}(L_1), \text{MergeSort}(L_2)) \]

else return \( a_1, \ldots, a_n \)

\[ T_{\text{MS}}(n/2) T_{\text{MS}}(n/2) \]

If \( T_{\text{MS}}(n) \) is number of operations taken by \( \text{MergeSort} \) on list of size \( n \),

\[ T_{\text{MS}}(1) = c' \]

\[ T_{\text{MS}}(n) = 2T_{\text{MS}}(n/2) + cn \]

where \( c', c \) are some constants
Merging sorted arrays WHEN

If $T_{MS}(n)$ is number of operations taken by MergeSort on list of size $n$,

$$T_{MS}(1) = c'$$
$$T_{MS}(n) = 2T_{MS}(n/2) + cn$$

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

Solving the recurrence by **unravelling**:

$$T_{MS}(n) = 2T_{MS}(n/2) + cn$$
$$= 2 \left( 2T_{MS}(n/4) + c(n/2) \right) = 4T_{MS}(n/4) + 2c(n/2) + cn = 4T_{MS}(n/4) + 2cn$$
$$= 4 \left( 2T_{MS}(n/8) + c(n/4)\right) + 2cn = 8T_{MS}(n/8) + 3cn$$
$$\vdots$$
$$= 2^kT_{MS}(n/2^k) + k(cn)$$
Solving the recurrence by unravelling:

\[ T_{MS}(n) = 2T_{MS}(n/2) + cn \]

\[ = 2 \left( 2T_{MS}(n/4) + c(n/2) \right) = 4T_{MS}(n/4) + 2c(n/2) + cn = 4T_{MS}(n/4) + 2cn \]

\[ = 4 \left( 2T_{MS}(n/8) + c(n/4) \right) + 2cn = 8T_{MS}(n/8) + 3cn \]

\[ \vdots \]

\[ = 2^k T_{MS}(n/2^k) + k(cn) \]

What value of \( k \) should we substitute to finish unravelling (i.e. to get to the base case)?

A. \( k \)
B. \( n \)
C. \( 2^n \)
D. \( \log_2 n \)
E. None of the above.
Merging sorted arrays WHEN

Solving the recurrence by **unravelling**:

\[ T_{MS}(n) = 2T_{MS}(n/2) + cn \]

\[ = 2 \left( 2T_{MS}(n/4) + c(n/2) \right) = 4T_{MS}(n/4) + 2c(n/2) + cn = 4T_{MS}(n/4) + 2cn \]

\[ = 4 \left( 2T_{MS}(n/8) + c(n/4) \right) + 2cn = 8T_{MS}(n/8) + 3cn \]

\[ \vdots \]

\[ = 2^kT_{MS}(n/2^k) + k(cn) \]

With \( k = \log_2 n \), \( T_{MS}(n/2^k) = T_{MS}(n/n) = T_{MS}(1) = c' \):

\[ T_{MS}(n) = 2^{\log n} T_{MS}(1) + (\log_2 n)(cn) = c'n + c \cdot n \log_2 n \]
Merge Sort

In terms of worst-case performance, Merge Sort outperforms all other sorting algorithms we've seen.

\[ n \quad n \log n \quad n^{1.001} \quad n^2 \]

<table>
<thead>
<tr>
<th>n</th>
<th>n^2</th>
<th>n log n</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 000</td>
<td>1 000 000</td>
<td>~10 000</td>
</tr>
<tr>
<td>1 000 000</td>
<td>1 000 000 000 000</td>
<td>~20 000 000</td>
</tr>
</tbody>
</table>

*Divide and conquer wins big!*
Divide & Conquer

What we saw:

Dividing into two subproblems each with half the size was a big win

Will this work in other contexts?
Given two \( n \)-digit (or bit) integers

\[ a = a_{n-1} \ldots a_1 a_0 \]

and

\[ b = b_{n-1} \ldots b_1 b_0 \]

return the decimal (or binary) representation of their product.

\[
\begin{array}{c}
25 \\
\times 17 \\
\hline
175 \\
+ 250 \\
\hline
425
\end{array}
\]
Given two \( n \)-digit (or bit) integers

\[
a = a_{n-1} \ldots a_1 a_0
\]

and

\[
b = b_{n-1} \ldots b_1 b_0
\]

return the decimal (or binary) representation of their product.

\[
\begin{array}{c c c c c}
& & & 2 & 5 \\
\times & 1 & 7 & & \\
\hline
& & 1 & 7 & 5 \\
+ & 2 & 5 & 0 & \\
\hline
& & 4 & 2 & 5 \\
\end{array}
\]

Compute partial products (using single digit multiplications), shift, then add.

How many operations? \( \mathcal{O}(n^2) \)
Divide and conquer? Divide $n$ bit numbers into two $n/2$ bit numbers.

If $a = 12345678$ and $b = 24681357$, we can write

$$a = (1234) \times 10^4 + (5678)$$
$$b = (2468) \times 10^4 + (1357)$$

To multiply:

$$\left( (1234) \times 10^4 + (5678) \right) \left( (2468) \times 10^4 + (1357) \right) =$$

$$1234(2468) \times 10^8 + (1234)(1357) \times 10^4 + (2468)(5678) \times 10^4 + (1357)(5678)$$
Multiplication WHEN

One 8-digit multiplication

\[
\begin{pmatrix}
12345678 \\
24681357
\end{pmatrix}
= 
\begin{pmatrix}
1234 \\
2468
\end{pmatrix} \times 10^4 + \begin{pmatrix}
5678 \\
1357
\end{pmatrix}
\]

\[
\begin{pmatrix}
1234 \\
2468
\end{pmatrix} \times 10^8 + \begin{pmatrix}
1234 \\
2468
\end{pmatrix} \times \begin{pmatrix}
1357 \\
5678
\end{pmatrix} \times 10^4 + \begin{pmatrix}
2468 \\
5678
\end{pmatrix} \times 10^4 + \begin{pmatrix}
1357 \\
5678
\end{pmatrix}
\]

Four 4-digit multiplications (plus some shifts, sums)
Multiplication WHEN

One 8-digit multiplication

\[
\begin{pmatrix}
12345678 \\
24681357
\end{pmatrix} = \begin{pmatrix}
(1234) \times 10^4 + (5678) \\
(2468) \times 10^4 + (1357)
\end{pmatrix} = \\
(1234)(2468) \times 10^8 + (1234)(1357) \times 10^4 + (2468)(5678) \times 10^4 + (1357)(5678)
\]

Four 4-digit multiplications (plus some shifts, sums)

\[T(n) = 4 \ T(n/2) + cn \quad \text{with} \quad T(1) = c' \quad \text{and} \quad c, \ c' \text{ constants}\]
Multiplication WHEN

\[ T(n) = 4 T(n/2) + cn \]

with \( T(1) = c' \) and \( c, c' \) constants

Unravelling

\[
T(n) = 4T(n/2) + cn \\
= 4 \left( 4T(n/4) + c(n/2) \right) + cn = 16T(n/4) + 3cn \\
= 16 \left( 4T(n/8) + c(n/4) \right) + 3cn = 64T(n/8) + 7cn \\
\vdots \\
= 4^kT(n/2^k) + (2^k - 1)cn
\]
Multiplication WHEN

\[ T(n) = 4T(n/2) + cn \]

with \( T(1) = c' \) and \( c, c' \) constants

Unravelling

Substitute \( k = \log_2 n \)

\[ T(n) = 4^k T(n/2^k) + (2^k - 1)cn \]

What's \( 2^{\log n} \)?
A. \( n \)
B. \( n^2 \)
C. \( 2^n \)
D. 1
E. None of the above
Multiplication WHEN

\[ T(n) = 4T(n/2) + cn \]

with \( T(1) = c' \) and \( c, c' \) constants

Unravelling

Substitute \( k = \log_2 n \)

\[ T(n) = 4^k T(n/2^k) + (2^k - 1) cn \]

What's \( 4^{\log n} \)?

A. \( n \)
B. \( n^2 \)
C. \( 2^n \)
D. \( 2n \)
E. None of the above
Multiplication WHEN

\[ T(n) = 4T(n/2) + cn \]
with \( T(1) = c' \) and \( c, c' \) constants

\[ T(n) = 4T(n/2) + cn \]
\[ = 4\left( 4T(n/4) + c(n/2) \right) + cn = 16T(n/4) + 3cn \]
\[ = 16\left( 4T(n/8) + c(n/4) \right) + 3cn = 64T(n/8) + 7cn \]
\[ \vdots \]
\[ = 4^kT(n/2^k) + (2^k - 1)cn \]

Substitute \( k = \log_2 n \)

\[ T(n) = c' n^2 + (n-1) cn \in \Theta(n^2) \]

Oh no!!!
Multiplication HOW

Insight: replace one (of the 4) multiplications by (linear time) subtraction

Andrey Kolmogorov 1903 - 1987

Anatoly Karatsuba 1937 - 2008

Rosen p. 528
(12345678)(24681357) = (1234) * 10^4 + (5678)(2468) * 10^4 + (1357)

(1234)(2468) * 10^8 + (1234)(1357) * 10^4 + (2468)(5678) * 10^4 + (1357)(5678)

(1234)(2468) * (10^8+10^4) + [(1234) - (5678)][(1357)-(2468)] * 10^4 + (1357)(5678) * (10^4+1)

Insight: replace one (of the 4) multiplications by (linear time) subtraction
Instead of

\[ T(n) = 4 \, T(n/2) + cn \quad \text{with} \quad T(1) = c' \quad \text{and} \quad c, \, c' \text{ constants} \]

get

\[ T_K(n) = 3 \, T_K(n/2) + d \, n \quad \text{with} \quad T_K(1) = d' \quad \text{and} \quad d, \, d' \text{ constants} \]

Unravelling is similar but with 3s instead of 4s

\[ T_K(n) \in \Theta(3^{\log_2 n}) \]
Karatsuba Multiplication WHEN

$3^\log n \approx n^{\log 3} = n^{1.58...}$

so definitely better than $n^2$

Progress since then …

1963: Toom and Cook develop series of algorithms that are time $O(n^{1+...})$

2007: Furer uses number theory to achieve the best known time for multiplication.

2015: Still open whether there is a linear time algorithm for multiplication.
Next Time…

• Graphs: definitions and examples
• Puzzles and algorithms.
Reminders

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

Midterm 1: Tuesday October 20.
* Practice midterm available on website / Piazza.
* Review sessions Saturday & Sunday: see website.
* Extra office hours over the weekend.
* Monday discussion session review.
* **Seating chart on website / Piazza.**
* 1 handwritten note sheet allowed.