## Divide & Conquer

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

[http://cseweb.ucsd.edu/classes/fa15/cse21-abc/](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
Given two **sorted** lists of distinct elements

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

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 arrays WHAT

Given two **sorted** lists of distinct elements

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

produce a **sorted** list of length \( n=k+1 \) which contains all their elements.

*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.
procedure Merge($a_1, \ldots, a_k, b_1, \ldots, b_l$ : 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

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

Correctness follows from $t = n$.

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

\begin{align*}
  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 \\
\end{align*}

return $c_1, \ldots, c_n$
Merging sorted arrays

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. $b_l < a_k$
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

$c_t := a_i$

else

$c_t := b_j$

return $c_1, \ldots, c_n$
Merging sorted arrays

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.

```
procedurer 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 \text{if } i > k \text{ then} \\
  &\quad c_t := b_j \\
  &\quad j := j + 1 \\
  \quad \text{if } j > l \text{ then} \\
  &\quad c_t := a_i \\
  &\quad i := i + 1 \\
  \quad \text{if } a_i \leq b_j \text{ then} \\
  &\quad c_t := a_i \\
  &\quad i := i + 1 \\
  \quad \text{else} \\
  &\quad c_t := b_j \\
  &\quad j := j + 1 \\
\end{align*}

\text{return } c_1, \ldots, c_n
```
Merging sorted arrays

WHEN \textit{Time analysis}

Work from the inside out

\begin{tikzpicture}
\begin{scope}[local bounding box=alg]
\node{\textbf{procedure} \textsc{IMerge}(a_1, \ldots, a_k, b_1, \ldots, b_l : \text{sorted arrays})}
\node[below=of alg.north]{\begin{align*}
n &:= k + l \\
i &:= 1 \\
j &:= 1 \\
\text{for } t = 1 \text{ to } n \quad \begin{align*}
\text{if } i > k \text{ then} \\
&\quad c_t := b_j \\
&\quad j := j + 1 \\
\text{if } j > l \text{ then} \\
&\quad c_t := a_i \\
&\quad i := i + 1 \\
\text{if } a_i \leq b_j \text{ then} \\
&\quad c_t := a_i \\
&\quad i := i + 1 \\
\text{else} \\
&\quad c_t := b_j \\
&\quad j := j + 1 \\
\end{align*}\end{align*}}
\node[right=of alg.north,anchor=west,inner sep=0pt]{\(\Theta(1)\)}
\node[right=of alg.east,inner sep=0pt]{
\begin{align*}
\text{return } c_1, \ldots, c_n 
\end{align*}}
\end{scope}
\end{tikzpicture}
Merging sorted arrays HOW

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

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

concatenate
Merging sorted arrays WHY

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

procedure \texttt{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

\text{return} \( a_1 \circ \text{RMerge}(a_2, \ldots, a_k, b_1, \ldots, b_l) \)

else

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

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

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

else

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

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

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

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

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

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:

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 arrays WHY

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

\[
\text{procedure } RMerge(a_1, \ldots, a_k, b_1, \ldots, b_l : \text{ sorted arrays}) \\
\text{if first list is empty then return } b_1, \ldots, b_l \\
\text{if second list is empty then return } a_1, \ldots, a_k \\
\text{if } a_1 \leq b_1 \text{ then} \\
\quad \text{return } a_1 \circ RMerge(a_2, \ldots, a_k, b_1, \ldots, b_l) \\
\text{else} \\
\quad \text{return } b_1 \circ RMerge(a_1, \ldots, a_k, b_2, \ldots, b_l)
\]

**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:

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

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

- 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: 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

procedure $R$Merge($a_1, \ldots, a_k, b_1, \ldots, b_l$ : 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 $a_1 \circ R$Merge($a_2, \ldots, a_k, b_1, \ldots, b_l$)
  else
    return $b_1 \circ R$Merge($a_1, \ldots, a_k, b_2, \ldots, b_l$)

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 $R$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.
Prepending $a_1$ to the start maintains the order and gives a sorted list with all elements. 😊
Merging sorted arrays WHY

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

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

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

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

else

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

Are we done with the proof?
A. Yes
B. No \( \text{almost...} \)
C. ??
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 $b_1 \circ RMerge(a_1, \ldots, a_k, b_2, \ldots, b_l)$

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

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

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

if $a_1 \leq b_1$ then

\hspace{1cm} return $a_1 \circ RMerge(a_2, \ldots, a_k, b_1, \ldots, b_l)$

else

\hspace{1cm} return $b_1 \circ RMerge(a_1, \ldots, a_k, b_2, \ldots, b_l)$

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
Merging sorted arrays WHEN

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

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

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

What's a solution to this recurrence equation?

A. $T(n) \in O(T(n-1))$ (not a sol'n)
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.
Back to sorting …

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."
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) 

\textbf{if } n > 1 \textbf{ then} 

\begin{align*}
m &:= \lfloor n/2 \rfloor \\
L_1 &:= a_1, \ldots, a_m \\
L_2 &:= a_{m+1}, \ldots, a_n \\
\text{return } R\text{Merge}(& \text{ MergeSort}(L_1), \text{ MergeSort}(L_2) ) \\
\text{else return } &a_1, \ldots, a_n
\end{align*}
procedure MergeSort\(a_1, \ldots, a_n\)

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

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 \(n-1\).
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

procedure MergeSort(a₁, . . . , aₙ)
  if n > 1 then
    m := \lfloor n/2 \rfloor
    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.

Suppose n=1.
Merge Sort WHY

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

Claim that result is a sorted list containing all elements.

Proof by \textbf{strong} induction on $n$:

\textbf{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 $a_1$, a (trivially) sorted list containing all elements. 😊
Merge Sort WHY

procedure MergeSort(a₁, …, aₙ)
    if n > 1 then
        m := \lfloor n/2 \rfloor
        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.

Induction step: Suppose n>1. Assume, as the strong induction hypothesis, that

MergeSort correctly sorts all lists with k elements, for 1≤k<n

Goal: prove that MergeSort(a₁, …, aₙ) returns a sorted list containing all n elements.
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

IH: MergeSort correctly sorts all lists with k elements, for 1≤k≤n

Goal: prove that MergeSort(a₁, ..., aₙ) returns a sorted list containing all n elements.

Since n>1, in the if branch we return RMerge( MergeSort(L₁), MergeSort(L₂) ), where L₁ and L₂ each have no more than (n/2) + 1 elements and together they contain all elements.

By IH, each of MergeSort(L₁) and MergeSort(L₂) are sorted and by the correctness of Merge, the returned list is a sorted list containing all the elements. 😊
Merge Sort WHEN

procedure MergeSort(a_1, \ldots, a_n)
\[\text{if } n > 1 \text{ then}\]
\[\text{\qquad } m := \lceil n/2 \rceil\]
\[\text{\qquad } L_1 := a_1, \ldots, a_m\]
\[\text{\qquad } L_2 := a_{m+1}, \ldots, a_n\]
\[\text{\qquad } T_{\text{Merge}}(n/2 + n/2) \text{ return } R\text{Merge}(\text{MergeSort}(L_1), \text{MergeSort}(L_2))\]
\[\text{else return } a_1, \ldots, a_n \text{ } T_{\text{MS}}(n/2) \text{ } T_{\text{MS}}(n/2)\]

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

\[
T_{\text{MS}}(1) = c' \quad \text{\textless base case}\]
\[
T_{\text{MS}}(n) = 2T_{\text{MS}}(n/2) + T_{\text{Merge}}(n) + c'' n
\]

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

procedure MergeSort(a₁, ..., aₙ)
    if n > 1 then
        θ(1) m := ⌊n/2⌋
        L₁ := a₁, ..., aₘ
        L₂ := aₘ₊₁, ..., aₙ
    else return a₁, ..., aₙ
    Tₘₑₑₙ(n) is in O(n)

Tₘₑₑₙ(n) = 2Tₘₑₑₙ(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) \leq general\ formula
Merging sorted arrays WHEN

Solving the recurrence by **unravelling**:

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

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

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

\[ \vdots \]

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

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.

\[ \text{When is } \frac{n}{2^k} = 1? \quad \text{solve for } k \]

\[ n = 2^k \]

\[ k = \log_2 n \quad T(1) = c \]
Merging sorted arrays WHEN

Solving the recurrence by **unravelling**:

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

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

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

\[ \vdots \]

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

\[ = 2^{\log_2 n} T_{MS}\left( \frac{n}{2^{\log_2 n}} \right) + (\log_2 n)(cn) \]

With \( k = \log_2 n \):

\[ T_{MS}(n/2^k) = T_{MS}(n/n) = T_{MS}(1) = c' : \]

\[ T_{MS}(n) = 2^{\log n} T_{MS}(1) + (\log_2 n)(cn) = c'n + c n \log_2 n \]
In terms of worst-case performance, Merge Sort outperforms all other sorting algorithms we've seen.

<table>
<thead>
<tr>
<th>n</th>
<th>n log n</th>
<th>n^{1.001}</th>
<th>n^2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>1 000 000</td>
<td>~10 000</td>
<td></td>
</tr>
<tr>
<td>1 000 000</td>
<td>1 000 000 000 000</td>
<td>~20 000 000</td>
<td></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.
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.

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:

\[
(1234 \times 10^4 + 5678)(2468 \times 10^4 + 1357) = \\
(1234 \times 2468) \times 10^8 + (1234)(1357) \times 10^4 + (2468)(5678) \times 10^4 + (1357)(5678)
\]

4 products of 4-digit #
Multiplication WHEN

One 8-digit multiplication

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

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

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

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

One 8-digit multiplication

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

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

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

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

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

Unravelling
Multiplication WHEN

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

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

Unravelling

Substitute \( k = \log_2 n \)

\[ T(n) = 4^{\log_2 n} \cdot T(1) + (2^{\log_2 n} - 1) cn \]

What's \( 2^{\log_2 n} \)?

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

\[
\begin{align*}
T(n) &= 4T(n/2) + cn \\
&= 4\left( 4T(n/4) + c(n/2) \right) + cn = 16T(n/4) + 3cn \\
&= 16\left( 4T(n/8) + c(n/4) \right) + 3cn = 64T(n/8) + 7cn \\
&\vdots \\
&= 4^kT(n/2^k) + (2^k - 1)cn
\end{align*}
\]

Unravelling

Substitute \( k = \log_2 n \)

\[ T(n) = 4^{\log_2 n}T(1) + (2^{\log_2 n} - 1) cn \]

\[
\begin{align*}
4^{\log n} &= (2^2)^{\log n} = 2^{2 \log n} \\
&= 2^{\log n^2} = n^2
\end{align*}
\]

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

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

Substitute \( k = \log_2 n \)

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

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

Andrey Kolmogorov 1903 - 1987
Anatoly Karatsuba 1937 - 2008

Rosen p. 528
Multiplication HOW

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

\[
(1234)(2468) \times 10^8 + (1234)(1357) \times 10^4 + (2468)(5678) \times 10^4 + (1357)(5678)
\]

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

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

Instead of

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

get

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

Unravelling is similar but with 3s instead of 4s

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

\[ 3^{\log n} = n^{\log 3} = n^{1.58...} \]

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.