Recursion: Divide and Conquer (mergesort.)

	Miles Jones	MTThF 8:30-9:50am	CSE 4140

August 9, 2016

Subsequences

Given a string (finite sequence) of symbols

$$b_1 \; b_2 \; b_3 \; \ldots \; b_n$$

A subsequence of length k of that string is a string of the form

$$b_{i_1}, b_{i_2}, \ldots, b_{i_k}$$

where $1 \leq i_1 < i_2 < \cdots < i_k \leq n$. The subsequence 010 can be found in a whole bunch of places in 0100101000.

0100101000
0100101000
0100101000

Example – Longest Common Subsequence: WHAT

Given two strings (finite sequences) of characters*

$$a_1 \ a_2 \ a_3 \ \ldots \ a_n$$
$$b_1 \ b_2 \ b_3 \ \ldots \ b_m$$

what's the length of the longest string which is a subsequence in both strings?

What should be the output for the strings
AGGACAT and ATTACGAT?

A. 1
B. 2
C. 3
D. 4
E. 5

* Could be 0s and 1s, or ACTG in DNA

Example – Longest Common Subsequence: HOW

Given two strings (finite sequences) of characters*

$$a_1 \; a_2 \; a_3 \; \ldots \; a_n$$
$$b_1 \; b_2 \; b_3 \; \ldots \; b_m$$

what's the length of the longest string which is a subsequence in both strings?

Design a recursive algorithm to solve this problem

* Could be 0s and 1s, or ACTG in DNA

Example – Longest Common Subsequence: HOW

A Recursive Algorithm

Do the strings agree at the head? Then solve for the rest.

procedure $lcsRec(a_1, \ldots, a_m; b_1, \ldots, b_n)$

  if $(m = 0$ **or** $n = 0)$ **then return** 0

  if $a_1 = b_1$ **then return** $1 + lcsRec(a_2, \ldots, a_m; b_2, \ldots, b_n)$

  return $max(lcsRec(a_1, \ldots, a_m; b_2, \ldots, b_n), lcsRec(a_2, \ldots, a_m; b_1, \ldots, b_n))$

Example – Longest Common Subsequence: HOW

A Recursive Algorithm

Do the strings agree at the head? Then solve for the rest.

procedure $lcsRec(a_1, \ldots, a_m; b_1, \ldots, b_n)$

if $(m = 0$ **or** $n = 0)$ **then return** 0

if $a_1 = b_1$ **then return** $1 + lcsRec(a_2, \ldots, a_m; b_2, \ldots, b_n)$

return $max(lcsRec(a_1, \ldots, a_m; b_2, \ldots, b_n), lcsRec(a_2, \ldots, a_m; b_1, \ldots, b_n))$

What would an iterative algorithm look like?

Example – Longest Common Subsequence: Why

procedure $lcsRec(a_1, \ldots, a_m; b_1, \ldots, b_n)$

  if $(m = 0$ **or** $n = 0)$ **then return** 0

  if $a_1 = b_1$ **then return** $1 + lcsRec(a_2, \ldots, a_m; b_2, \ldots, b_n)$

  return $max(lcsRec(a_1, \ldots, a_m; b_2, \ldots, b_n), lcsRec(a_2, \ldots, a_m; b_1, \ldots, b_n))$

Proof by induction:

Base Case: if n=0 or m=0 then one of the lists is empty and you can't have a common subsequence with an empty list so the algorithm should return 0

Base Case: (of induction:) when $m+n=0$

$\Rightarrow$ $m=0$ and $n=0$

the algorithm returns 0. ✓

Example – Longest Common Subsequence: Why

procedure $lcsRec(a_1, \ldots, a_m; b_1, \ldots, b_n)$
  if $(m = 0$ **or** $n = 0)$ **then return** 0
  if $a_1 = b_1$ **then return** $1 + lcsRec(a_2, \ldots, a_m; b_2, \ldots, b_n)$
  return $max(lcsRec(a_1, \ldots, a_m; b_2, \ldots, b_n), lcsRec(a_2, \ldots, a_m; b_1, \ldots, b_n))$

Proof by induction:

Inductive Hypothesis: (strong induction.)

Suppose that for some k>0 the algorithm lcsRec(a[1],…,a[m];b[1],…,b[n]) returns the length of the longest common subsequence of a[1..m] and b[1…n] for all pairs of lists such that $0 \leq$ m+n ≤ k.

Example – Longest Common Subsequence: Why

procedure $lcsRec(a_1, \ldots, a_m; b_1, \ldots, b_n)$

    if $(m = 0$ **or** $n = 0)$ **then return** 0

    if $a_1 = b_1$ **then return** $1 + lcsRec(a_2, \ldots, a_m; b_2, \ldots, b_n)$

    return $max(lcsRec(a_1, \ldots, a_m; b_2, \ldots, b_n), lcsRec(a_2, \ldots, a_m; b_1, \ldots, b_n))$

Proof by induction:

Inductive Step: (Want to show.)

We want to show that lcsRec(a[1..m],b[1..n]) returns the length of the longest common subsequence of a[1..m] and b[1..n] when m+n=k+1.

Example – Longest Common Subsequence: Why

procedure $lcsRec(a_1, \ldots, a_m; b_1, \ldots, b_n)$
  if $(m = 0$ **or** $n = 0)$ **then return** 0
  if $a_1 = b_1$ **then return** $\boxed{1 + lcsRec(a_2, \ldots, a_m; b_2, \ldots, b_n)}$
  return $max(lcsRec(a_1, \ldots, a_m; b_2, \ldots, b_n), lcsRec(a_2, \ldots, a_m; b_1, \ldots, b_n))$

Proof by induction:

Inductive Step:

Suppose that m+n = k. Then

input of size m+n-2

By the inductive hypothesis.

Case 1: a[1]=b[1]

the algorithm will return 1 + lcs(a[2..m],b[2..n]) which is 1 + the length of the lcs of a[2..m],b[2..n]. The subsequence that starts with a[1]=b[1] and the rest is the lcs of a[2..m],b[2..n] is the lcs of a[1..m],b[1..n]. Therefore 1 + lcs(a[2..m],b[2..n]) is the length of the lcs of a[1…m],b[1…n].

Example – Longest Common Subsequence: Why

procedure $lcsRec(a_1, \ldots, a_m; b_1, \ldots, b_n)$

  if $(m = 0$ **or** $n = 0)$ **then return** 0

  if $a_1 = b_1$ **then return** $1 + lcsRec(a_2, \ldots, a_m; b_2, \ldots, b_n)$

  return $max(lcsRec(a_1, \ldots, a_m; b_2, \ldots, b_n), lcsRec(a_2, \ldots, a_m; b_1, \ldots, b_n))$

Proof by induction:

Inductive Step:

Suppose that m+n = k. Then

Case 2: a[1]≠b[1]

We know that either a[1] or b[1] does not contribute to the lcs of a[1…n],b[1…m] so we just take the maximum of the lcs of a[2…n],b[1…m] and the lcs of a[1…n],b[2…m], i.e. the lcs of the two inputs that exclude either a[1] or b[1]. This is what the algorithm returns. ✓

input of size m+n-1

Example – Longest Common Subsequence: Why

procedure $lcsRec(a_1, \ldots, a_m; b_1, \ldots, b_n)$

  if $(m = 0$ **or** $n = 0)$ **then return** 0

  if $a_1 = b_1$ **then return** $1 + lcsRec(a_2, \ldots, a_m; b_2, \ldots, b_n)$

  return $max(lcsRec(a_1, \ldots, a_m; b_2, \ldots, b_n), lcsRec(a_2, \ldots, a_m; b_1, \ldots, b_n))$

Proof by induction:

Conclusion:

Therefore the algorithm works on all inputs m,n such that m+n≥ 0.

Recursion

Last Time

1. Recursive algorithms and correctness
2. Coming up with recurrences
3. Using recurrences for time analysis

Today: Using recursion to design faster algorithms
Important example: Mergesort
Important sub-procedure: Merge
Example of ``divide-and-conquer'' algorithm design

In the textbook: Sections 5.4, 8.3

Merging sorted lists: WHAT

Given two **sorted** lists

$$a_1 \quad a_2 \quad a_3 \quad \ldots \quad a_k$$
$$b_1 \quad b_2 \quad b_3 \quad \ldots \quad b_\ell$$

produce a **sorted** list of length $n=k+\ell$ 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: HOW

Given two **sorted** lists

$$a_1 \;\; a_2 \;\; a_3 \;\; \ldots \;\; a_k$$
$$b_1 \;\; b_2 \;\; b_3 \;\; \ldots \;\; b_\ell$$

produce a **sorted** list of length $n=k+\ell$ which contains all their elements.

Design an algorithm to solve this problem

recursive.

Merging sorted lists: HOW

Similar to Rosen p. 369

A recursive algorithm

Idea: Find the smallest element.

Put it first in the sorted list

``Delete'' it from the list it came from

Merge the remaining parts of the lists recursively

If the input lists $a_1 .. a_k$

and $b_1 .. b_\ell$

Are sorted, which elements could be the smallest in the merged list?

Merging sorted lists: HOW

Similar to Rosen p. 369

A recursive algorithm

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

Find the smallest element

Merge the remaining parts

"o"= concatenate

Merging sorted lists: WHY

Similar to Rosen p. 369

<u>A recursive algorithm</u>

Focus on merging head elements, then rest.

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 $a_1 \le b_1$ **then**
        return $a_1 \circ RMerge(a_2, \ldots, a_k, b_1, \ldots, b_\ell)$
    else
        return $b_1 \circ RMerge(a_1, \ldots, a_k, b_2, \ldots, b_\ell)$

Base Case: $k+\ell=0$
Inductive Hypothesis
Inductive Step (what you want to show)
Inductive step
Conclusion.

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

Proof by induction on n=k+ℓ,
the total input size

Merging sorted lists: WHY

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

Proof by induction on n, $=k+\ell$ the total input size

$$
\begin{array}{l}
\textbf{procedure } RMerge(a_1,\ldots,a_k,b_1,\ldots,b_\ell\text{: sorted lists}) \\
\quad \textbf{if first list is empty then return } b_1,\ldots,b_\ell \\
\quad \textbf{if second list is empty then return } a_1,\ldots,a_k \\
\quad \textbf{if } a_1 \le b_1 \textbf{ then} \\
\qquad \textbf{return } a_1 \circ RMerge(a_2,\ldots,a_k,b_1,\ldots,b_\ell) \\
\quad \textbf{else} \\
\qquad \textbf{return } b_1 \circ RMerge(a_1,\ldots,a_k,b_2,\ldots,b_\ell)
\end{array}
$$

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

Merging sorted lists: WHY

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

Proof by induction on n, the total input size

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 $a_1 \le b_1$ **then**
    return $a_1 \circ RMerge(a_2, \ldots, a_k, b_1, \ldots, b_\ell)$
  else
    return $b_1 \circ RMerge(a_1, \ldots, a_k, b_2, \ldots, b_\ell)$

Hypothesis

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**. What do we want to prove?

want to show

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

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

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.**

pretend I did this for LCS.

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

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

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

Case 2a: both lists nonempty and $a_1 <= b_1$
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 $b_1, \ldots, b_l$

a_1 is smaller than or equal to everything else.

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

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

a_1 and b_1

Merging sorted lists: WHEN

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

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

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

if $a_1 \le b_1$ **then** $\Theta(1)$

One recursive call

  return $a_1 \circ RMerge(a_2, \ldots, a_k, b_1, \ldots, b_\ell)$ $T(n-1) + c'$

else

  return $b_1 \circ RMerge(a_1, \ldots, a_k, b_2, \ldots, b_\ell)$ $T(n-1) + c'$

Suppose $T(n)$

If T(n) is the time taken by *RMerge* on input of total size n, $= k + \ell$

T(0) = c

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

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

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.

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

This the same recurrence as we solved Monday for counting 00's inm a string. So we can just remember that this works out to T(n) ∈ $\theta(n)$

Merge Sort: HOW

"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

Merge Sort: HOW

Similar to Rosen p. 368

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 $RMerge(\ MergeSort(L_1), MergeSort(L_2)\ )$
    else return $a_1, \ldots, a_n$

Use *RMerge* as subroutine

Merge Sort: WHY

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 $RMerge(\ MergeSort(L_1), MergeSort(L_2)\ )$

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 size of 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. Because the input size of the recursive function call is less than n-1.

Merge Sort: WHY

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 $RMerge(\ MergeSort(L_1), MergeSort(L_2)\ )$

else return $a_1, \ldots, a_n$

Claim that result is a sorted list containing all elements.

Proof by **strong** induction on n:

Base case : Suppose n=0.

Suppose n=1.

Merge Sort: WHY

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 $RMerge(\ MergeSort(L_1), MergeSort(L_2)\ )$

  else return $a_1, \ldots, a_n$

Claim that result is a sorted list containing all elements.

Proof by **strong** induction on n:

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 (trivally) sorted list containing all elements. ☺

Merge Sort: WHY

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 $RMerge(\ MergeSort(L_1), MergeSort(L_2)\ )$

  else return $a_1, \ldots, a_n$

Claim that result is a sorted list containing all elements.

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

MergeSort correctly sorts all lists with k elements, for any 0<=k<n.

Goal: prove that *MergeSort*(a_1, …, a_n) returns a sorted list containing all n elements.

Merge Sort: WHY

conclusion:
therefore Mergesort sorts all lists of size $n \geq 0$

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 $RMerge(\ MergeSort(L_1), MergeSort(L_2)\ )$
  else return $a_1, \ldots, a_n$

L_1 and L_2 both have $n/2$ elem. if n is even

L_1 and L_2 have no more than $n/2+1$ elem if n is odd

IH: *MergeSort* correctly sorts all lists with k elements, for any 0<=k<n

Goal: prove that *MergeSort*(a_1, …, a_n) returns a sorted list containing all n elements.

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

$0 \leq |L_1| < n$ and $0 \leq |L_2| < n$

By IH, each of *MergeSort(L_1)* and *MergeSort(L_2)* are sorted and by the correctness of *Merge*, the returned list is a sorted list containing all the elements. ☺

Merge Sort: WHEN

procedure $MergeSort(a_1, \ldots, a_n)$

if $n > 1$ **then**

θ(1) $\boxed{m := \lfloor n/2 \rfloor}$

say θ(n) $\boxed{L_1 := a_1, \ldots, a_m}$ $= n/2 = \Theta(n)$

say θ(n) $L_2 := a_{m+1}, \ldots, a_n$ $= n/2 = \Theta(n)$

$\boxed{T_{Merge}}(n/2 + n/2)$ **return** $RMerge(\ MergeSort(L_1), MergeSort(L_2)\ )$

else return $a_1, \ldots, a_n$ $T_{MS}(n/2)$ $T_{MS}(n/2)$

If $\boxed{T_{MS}(n)}$ is runtime of *MergeSort* on list of size n,

$\boxed{T_{MS}(0) = c_0}$ $\boxed{T_{MS}(1) = c'}$

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

where c_0, c', c'' are some constants

$\Theta(n)$

Merge Sort: WHEN

$T_{Merge}(n)$ is in $O(n)$

procedure $MergeSort(a_1, \ldots, a_n)$

if $n > 1$ **then**

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

? $L_1 := a_1, \ldots, a_m$

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

$T_{Merge}(n/2 + n/2)$ **return** $RMerge(\ MergeSort(L_1), MergeSort(L_2)\ )$

else return $a_1, \ldots, a_n$ $T_{MS}(n/2)$ $T_{MS}(n/2)$

If $T_{MS}(n)$ is runtime of *MergeSort* on list of size n,

$T_{MS}(0) = c_0$ $T_{MS}(1) = c'$

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

where c_0, c, c' are some constants

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

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

$$= 2^2 T_{MS}(n/2^2) + 2cn + cn$$

$$= 2^2\left[2T_{MS}(n/2^3) + cn\right] + 2cn + cn$$

$$= 2^3 T_{MS}(n/2^3) + 2^2 cn + 2cn + cn$$

$$\vdots$$

$$= 2^k T_{MS}(n/2^k) + cn\sum_{j=0}^{k-1} 2^j$$

$$T_{MS}(n) = 2^k T_{MS}\left(\frac{n}{2^k}\right) + \left(\sum_{j=0}^{k-1} 2^j\right) cn$$

$$= 2^k T_{MS}\left(\frac{n}{2^k}\right) + \left(\frac{2^k - 1}{2 - 1}\right) cn$$

$$\boxed{T_{MS}(n) = 2^k T_{MS}\left(\frac{n}{2^k}\right) + 2^k cn - cn}$$

after $k = \log_2(n)$ many unravellings

$n/2^k = 1$

Merging sorted lists: WHEN

If $T_{MS}(n)$ is runtime of *MergeSort* on list of size n,

$T_{MS}(0) = c_0$ $\qquad T_{MS}(1) = c'$

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

where c_0, c, c' are some constants

Solving the recurrence by **unravelling**:

$$
\begin{aligned}
T_{MS}(n) &= 2T_{MS}(n/2) + cn \\
&= 2(\ 2T_{MS}(n/4) + c(n/2)\ ) = 4T_{MS}(n/4) + 2c(n/2) + cn = 4T_{MS}(n/4) + 2cn \\
&= 4(\ 2T_{MS}(n/8) + c(n/4)\ ) + 2cn = 8T_{MS}(n/8) + 3cn \\
&\ \vdots \\
&= 2^k T_{MS}(n/2^k) + k(cn)
\end{aligned}
$$

Merging sorted lists: WHEN

Solving the recurrence by **unravelling**:

$$T_{MS}(n) = 2T_{MS}(n/2) + cn$$
$$= 2(\ 2T_{MS}(n/4) + c(n/2)\ ) = 4T_{MS}(n/4) + 2c(n/2) + cn = 4T_{MS}(n/4) + 2cn$$
$$= 4(\ 2T_{MS}(n/8) + c(n/4)\ ) + 2cn = 8T_{MS}(n/8) + 3cn$$
$$\vdots$$
$$= 2^k T_{MS}(n/2^k) + k(cn)$$

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 lists: WHEN

Solving the recurrence by **unravelling**:

$$
\begin{aligned}
T_{MS}(n) &= 2T_{MS}(n/2) + cn \\
&= 2(\ 2T_{MS}(n/4) + c(n/2)\ ) = 4T_{MS}(n/4) + 2c(n/2) + cn = 4T_{MS}(n/4) + 2cn \\
&= 4(\ 2T_{MS}(n/8) + c(n/4)\ ) + 2cn = 8T_{MS}(n/8) + 3cn \\
&\ \vdots \\
&= 2^k T_{MS}(n/2^k) + k(cn)
\end{aligned}
$$

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

$\boxed{T_{MS}(n) = 2^{\log n}\ T_{MS}(1) + (\log_2 n)(cn) = c'n + c\ n \log_2 n}$

$$\boxed{T_{MS}(n) \in \Theta(n \log n)}$$

Merge Sort

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

n	n^2	n log n
1 000	1 000 000	~10 000
1 000 000	1 000 000 000 000	~20 000 000

Divide and conquer wins big!