CPSC 5031 :
Data Structures & Algorithms

Lecture 5: Divide-and-Conquer*

(Sedgewick, Chapter 2, pp 402-)

*Better name:

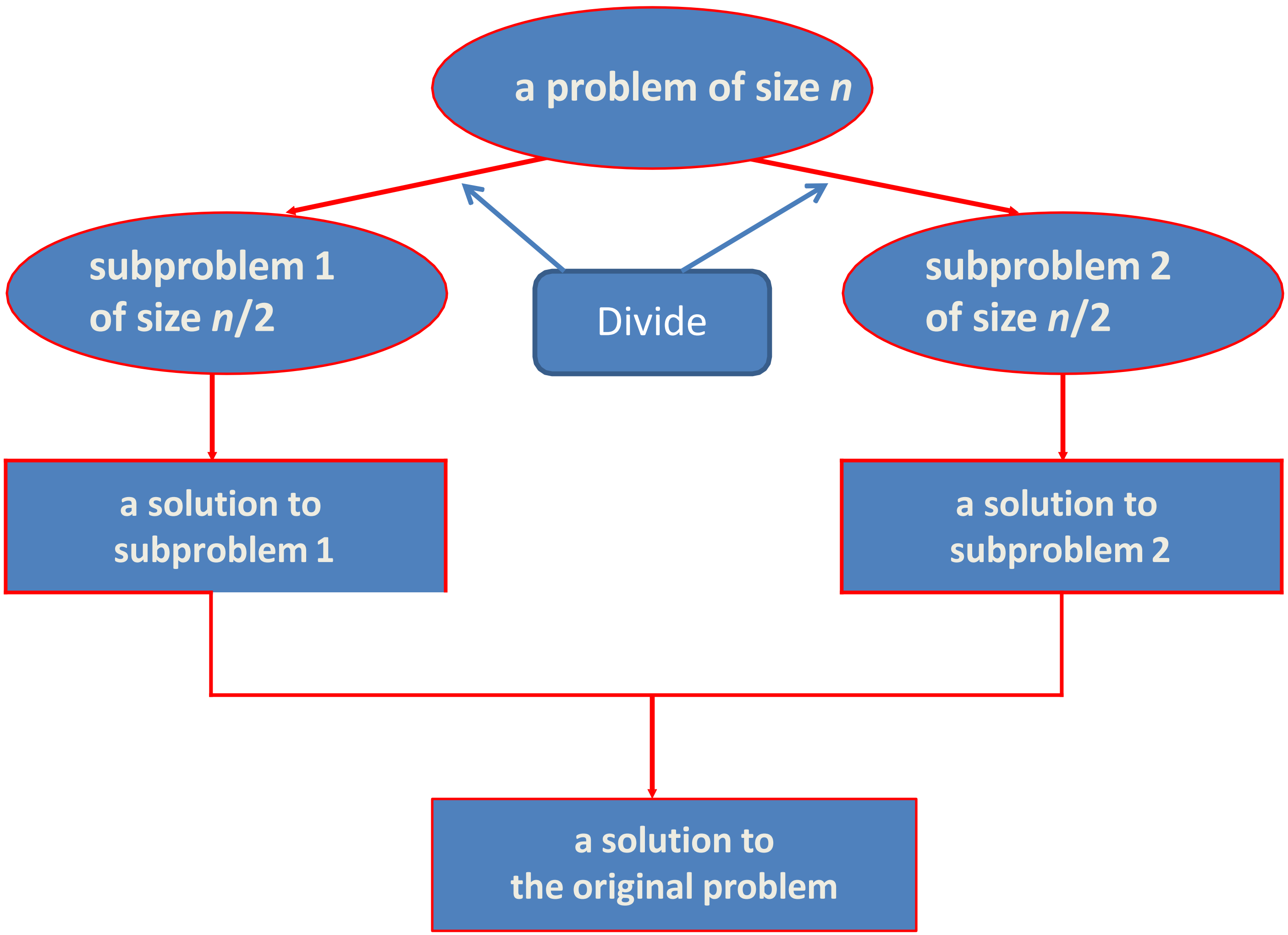
Divide-by-a-constant-amount and conquer

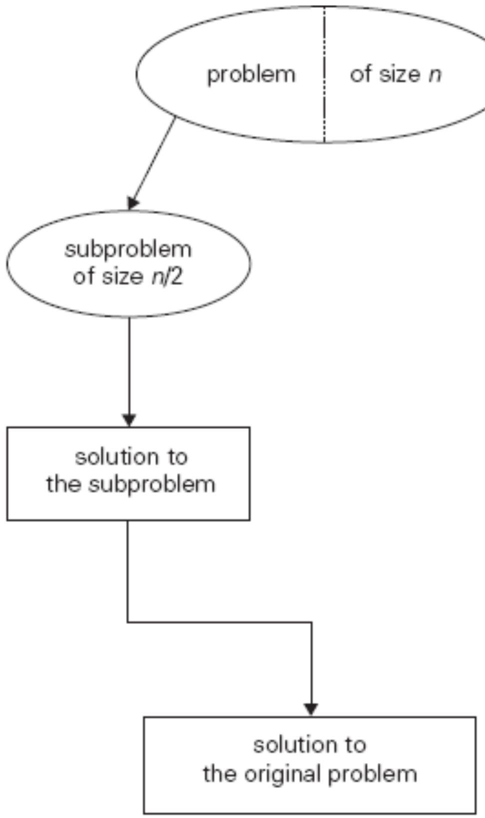
Divide-and-Conquer

The most-well known algorithm design technique:

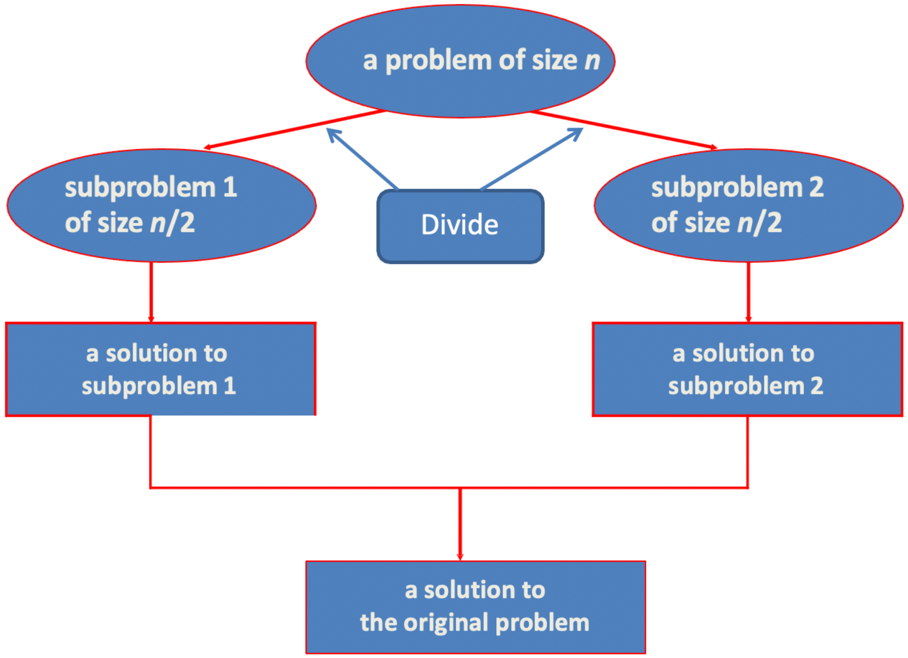
1. Divide instance of problem into two or more smaller instances
2. Solve smaller instances *independently* and recursively
   - When to stop?
3. Obtain solution to original (larger) instance by combining these solutions

Divide-and-Conquer Technique (cont.)





problem
of size n
subproblem
of size $n/2$
solution to
the subproblem
solution to
the original problem



a problem of size n
subproblem 1
of size $n/2$
Divide
subproblem 2
of size $n/2$
a solution to
subproblem 1
a solution to
subproblem 2
a solution to
the original problem

A General Template

// *S* is a large problem with input size of *n*

```
Algorithm divide_and_conquer(S)
   if  (S is small enough to handle)
      solve it  //conquer
  else
      split  S  into  two (equally-sized) subproblems   S1 and S2
      divide_and_conquer(S1)
      divide_and_conquer(S2)
      combine solutions to S1 and S2
   endif
End
```

General Divide-and-Conquer Recurrence

- Recursive algorithms are a natural fit for divide-and-conquer
  - Distinguish from Dynamic Programming
- Recall algorithm efficiency analysis for recursive algorithms
  - Key: Recurrence Relation
  - Solve: backward substitution, often cumbersome!

Master Theorem

Let $T(n)$ be a monotonically increasing (positive) function that satisfies,

$$T(n) = aT(n/b) + f(n)$$
$$T(1) = c$$

where $a \geq 1$, $b \geq 2$, $c > 0$. If $f(n) \in \Theta(n^d)$, where $d \geq 0$, then,

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log(n)) & \text{if } a = b^d \\ \Theta(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

n	=	size of problem
n/b	=	size of subproblem
a	=	# of subproblems in recursion
$f(n)$	=	cost of work done outside recursive calls

Divide-and-Conquer Examples

- Exponentiation
- Sorting:
  - Mergesort
  - Selection
  - Quickselect
  - Quicksort
- Counting Inversions
- Multiplication of large integers
- Matrix multiplication: Strassen's algorithm
- Closest-pair algorithm
- Binary Tree algorithm

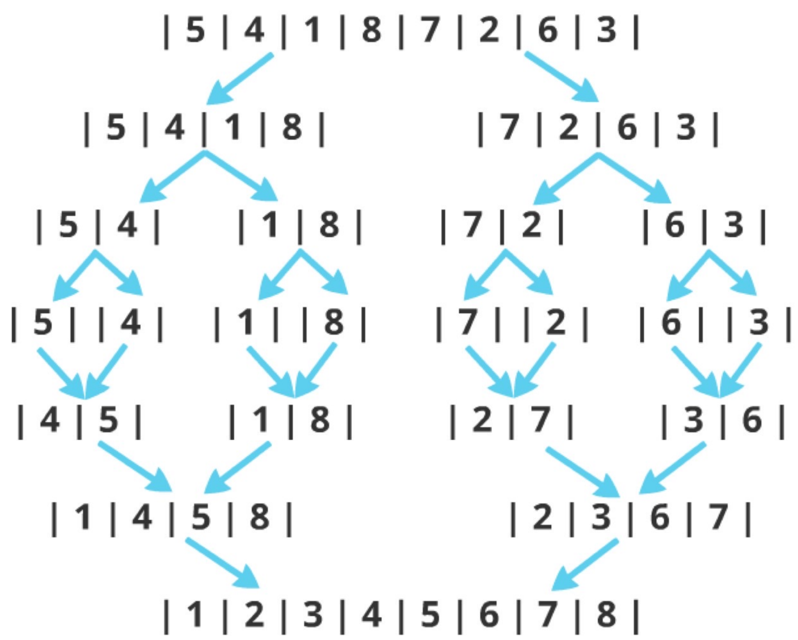
Divide-and-Conquer Examples

- Exponentiation
- Sorting:
  - Mergesort
  - Selection
  - Quickselect
  - Quicksort
- Counting Inversions
- Multiplication of large integers
- Matrix multiplication: Strassen's algorithm
- Closest-pair algorithm
- Binary Tree algorithm

Mergesort

- Split array A[0..n-1] in two about equal halves and make copies of each half in arrays B and C
- Sort arrays B and C recursively
  - **Q: when to stop?**
- Merge sorted arrays B and C into array A as follows:
  - Repeat the following until no elements remain in one of the arrays:
    - compare the first elements in the remaining unprocessed portions of the arrays
    - copy the smaller of the two into A, while incrementing the index indicating the unprocessed portion of that array
  - Once all elements in one of the arrays are processed, copy the remaining unprocessed elements from the other array into A.

Breakdown of Divide & Conquer

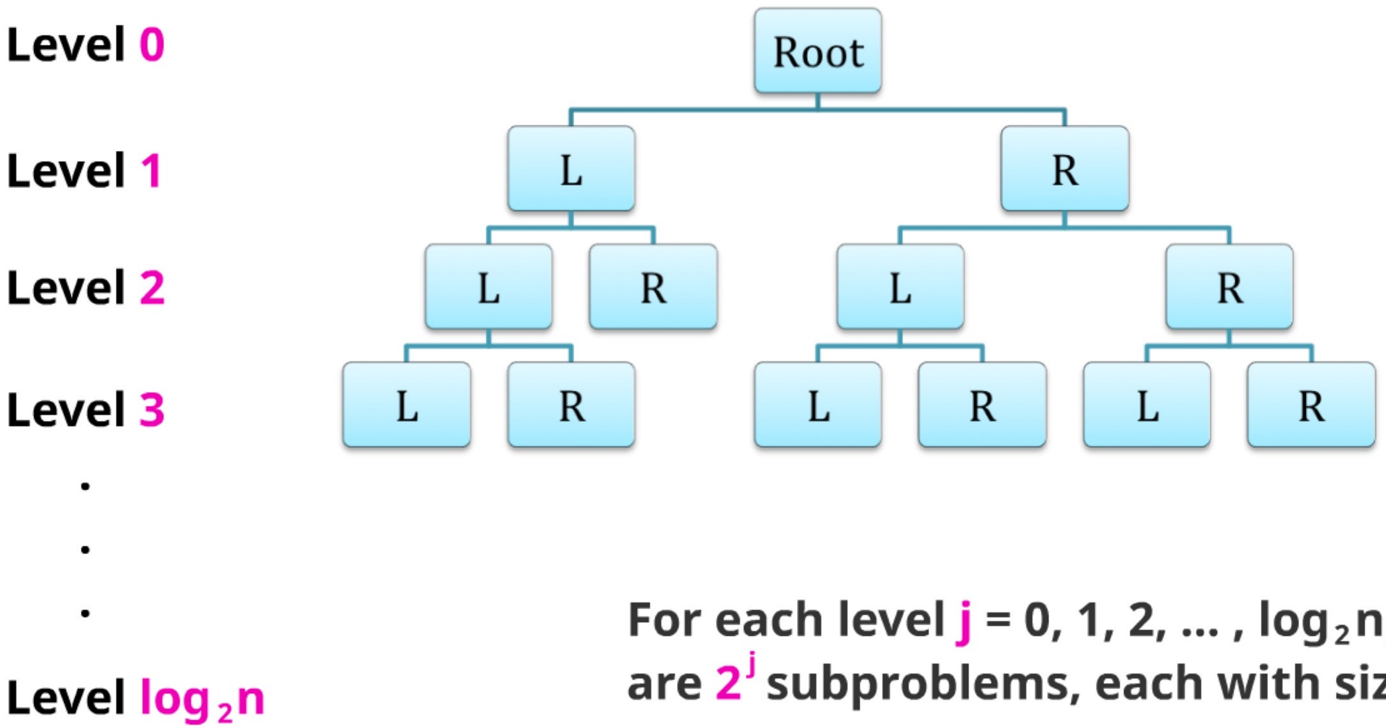


***DIVIDE* into subproblems**

***RECURSIVE* calls**

***MERGE* subproblems**

Recursion Tree



For each level $j = 0, 1, 2, \ldots, \log_2 n$, there are 2^j subproblems, each with size $n/2^j$.

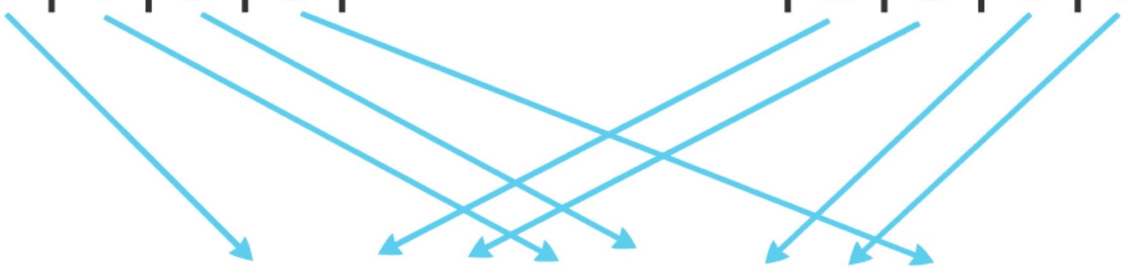
Merge step

A: 1st sorted array (n/2)

| 1 | 4 | 5 | 8 |

B: 2nd sorted array (n/2)

| 2 | 3 | 6 | 7 |



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

C: output array (length n)

Start:

i=1, j=1

k=1

Pseudocode of Mergesort

ALGORITHM *Mergesort*$(A[0..n-1])$

```
//Sorts array A[0..n − 1] by recursive mergesort
//Input: An array A[0..n − 1] of orderable elements
//Output: Array A[0..n − 1] sorted in nondecreasing order
if n > 1
    copy A[0..⌊n/2⌋ − 1] to B[0..⌊n/2⌋ − 1]
    copy A[⌊n/2⌋..n − 1] to C[0..⌈n/2⌉ − 1]
    Mergesort(B[0..⌊n/2⌋ − 1])
    Mergesort(C[0..⌈n/2⌉ − 1])
    Merge(B, C, A)
```

Pseudocode of Mergesort

ALGORITHM *Mergesort*$(A[0..n-1])$

//Sorts array $A[0..n-1]$ by recursive mergesort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
if $n > 1$
    copy $A[0..\lfloor n/2 \rfloor - 1]$ to $B[0..\lfloor n/2 \rfloor - 1]$
    copy $A[\lfloor n/2 \rfloor..n-1]$ to $C[0..\lceil n/2 \rceil - 1]$
    Mergesort$(B[0..\lfloor n/2 \rfloor - 1])$
    Mergesort$(C[0..\lceil n/2 \rceil - 1])$
    Merge(B, C, A)

"Floor" (round down) $\lfloor n/2 \rfloor$

"Floor" (round down) $\lceil n/2 \rceil$

Pseudocode of Merge

ALGORITHM $Merge(B[0..p-1], C[0..q-1], A[0..p+q-1])$

//Merges two sorted arrays into one sorted array
//Input: Arrays $B[0..p-1]$ and $C[0..q-1]$ both sorted
//Output: Sorted array $A[0..p+q-1]$ of the elements of B and C
$i \leftarrow 0$; $j \leftarrow 0$; $k \leftarrow 0$
while $i < p$ **and** $j < q$ **do**
    if $B[i] \leq C[j]$
        $A[k] \leftarrow B[i]$; $i \leftarrow i+1$
    else $A[k] \leftarrow C[j]$; $j \leftarrow j+1$
    $k \leftarrow k+1$
if $i = p$
    copy $C[j..q-1]$ to $A[k..p+q-1]$
else copy $B[i..p-1]$ to $A[k..p+q-1]$

Analysis of Mergesort

- Time efficiency by recurrence relation:

  $T(n) = 2T(n/2) + f(n)$

  n-1 comparisons in merge operation for worst case!

  $T(n) = \Theta(n \log n)$

- Number of comparisons in the worst case is close to theoretical minimum for comparison-based sorting:

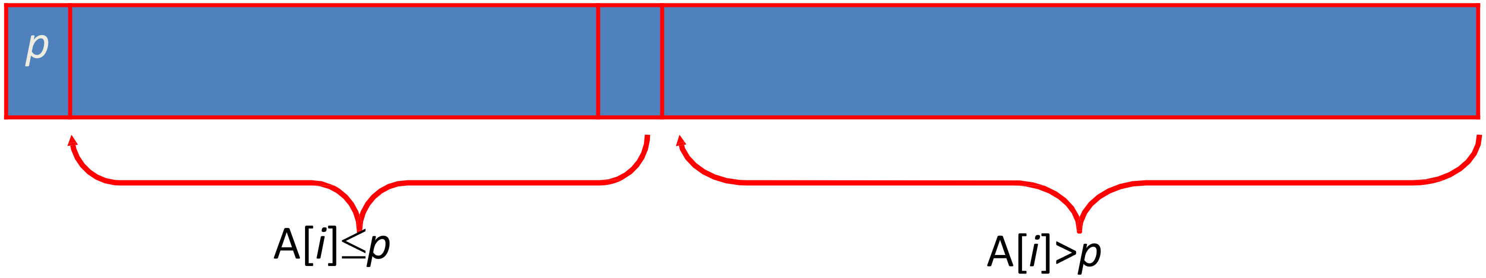
  $\lceil \log_2 n! \rceil \approx n \log_2 n - 1.44n$ (Section 11.2)

- Space requirement: $\Theta(n)$ (<u>not</u> in-place)

- Can be implemented without recursion (bottom-up)

Quicksort

- Select a *pivot* (partitioning element) – here, the first element for simplicity!
- Rearrange the list so that all the elements in the first *s* positions are smaller than or equal to the pivot and all the elements in the remaining *n-s* positions are larger than the pivot (see next slide for an algorithm)



- Exchange the pivot with the last element in the first (i.e., ≤) subarray — *the pivot is now in its final position*
- Sort the two subarrays recursively

Divide-and-Conquer Examples

- Exponentiation
- Sorting:
  - Mergesort
  - Selection
  - Quickselect
  - Quicksort
- Counting Inversions
- Multiplication of large integers
- Matrix multiplication: Strassen's algorithm
- Closest-pair algorithm
- Binary Tree algorithm

Divide-and-Conquer Examples

- Exponentiation
- Sorting:
  - Mergesort
  - Selection
  - Quickselect
  - Quicksort
- Counting Inversions
- Multiplication of large integers
- Matrix multiplication: Strassen's algorithm
- Closest-pair algorithm
- Binary Tree algorithm

Binary Tree Algorithms

Binary tree is a divide-and-conquer ready structure!

Ex. 1: Classic traversals (preorder, inorder, postorder)

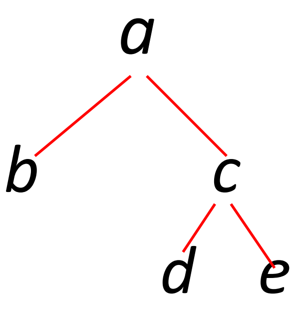
Algorithm *Inorder*(T)

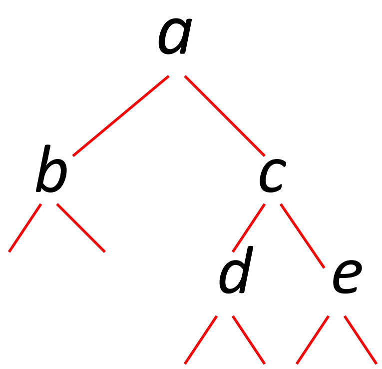
if $T \neq \varnothing$

    Inorder(T_{left})

    print(root of T)

    Inorder(T_{right})



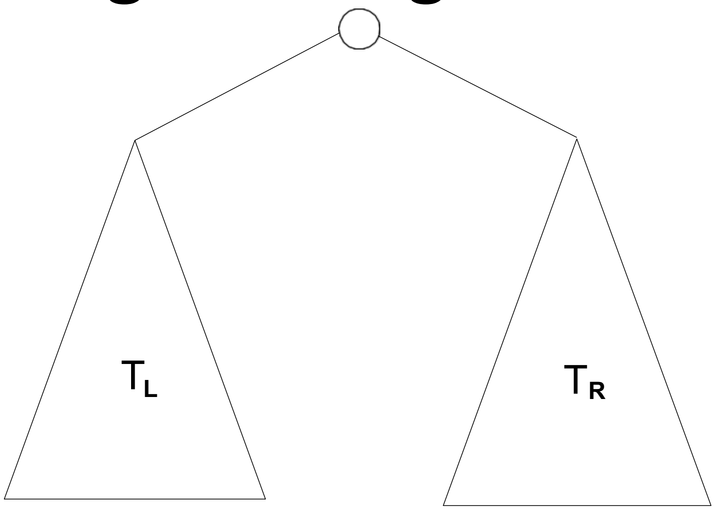


Efficiency: O(n)

Why not O(n2)? Why not O(n log n)?

Binary Tree Algorithms (cont.)

Ex. 2: Computing the height of a binary tree



$\boldsymbol{h(T) = \max\{h(T_L), h(T_R)\} + 1}$ **if** $\boldsymbol{T \neq \varnothing}$

Efficiency: O(n)

Why not O(n log n)?

Map-Reduce

- Multi-machine divide-and-conquer

- This is what built Google!

- *Map* the problem into *n* sub-problems
- Run the sub-problems in **parallel**
- *Reduce* the results into a single result set

- aka MergeSort writ large!

Map-Reduce case study

- PlayNetwork

- 70K music players, offline,
  but need playback info for royalties

- How to calculate music royalties based on number of plays *when offline*?

Map-Reduce case study

- Simulate!
- Would have taken a week to simulate 7x24 hours…
- Broke it up over 100 AWS virtual machines
- Took 2 hours ☺