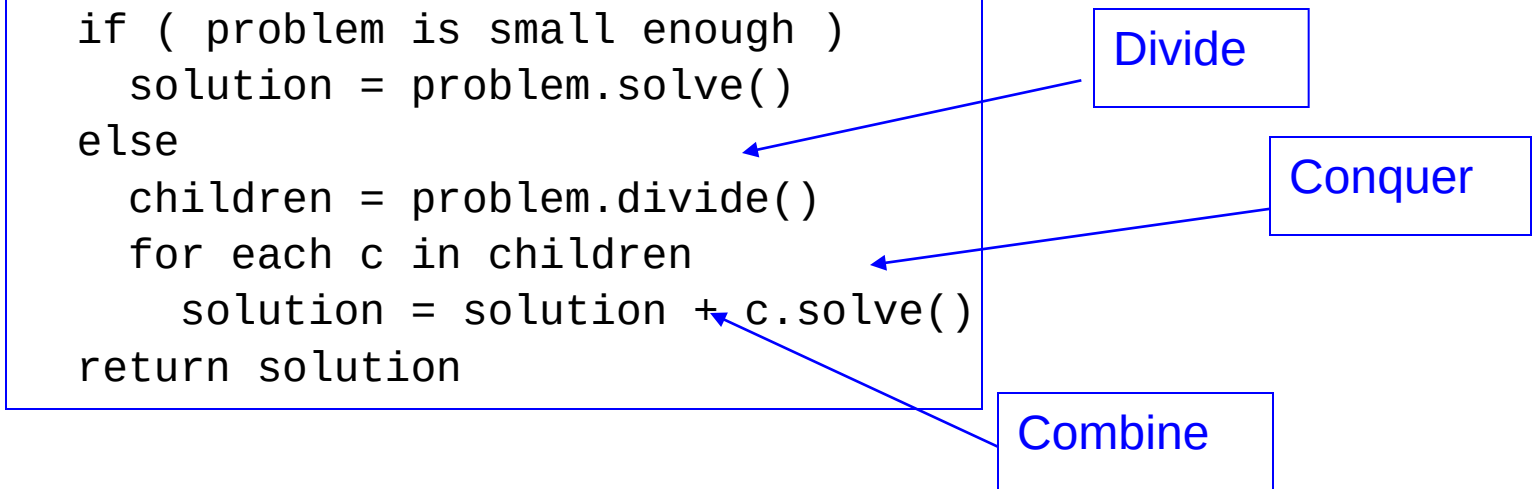
Divide and Conquer

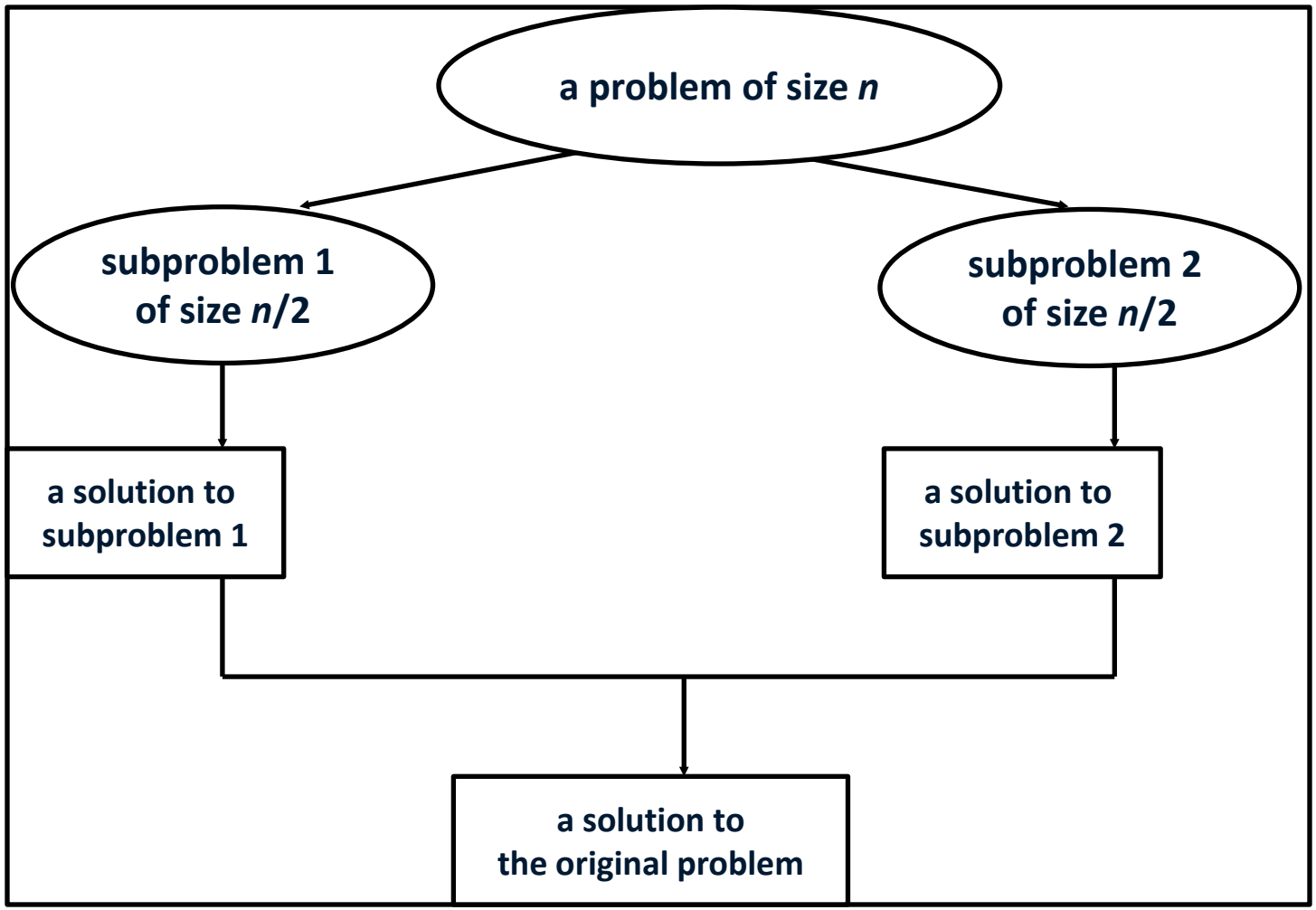
Definition

- Recursion lends itself to a general problem-solving technique (algorithm design) called **divide & conquer**
  - ***Divide*** the problem into 1 or more similar sub-problems
  - ***Conquer*** each sub-problem, usually using a recursive call
  - ***Combine*** the results from each sub-problem to form a solution to the original problem
- Algorithmic Pattern:

```
DC( problem )
  solution = ∅
  if ( problem is small enough )
    solution = problem.solve()
  else
    children = problem.divide()
    for each c in children
      solution = solution + c.solve()
  return solution
```



Basic Idea


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

❖ Applicability

- Use the divide-and-conquer algorithmic pattern when ALL of the following are true:
  - The problem lends itself to division into sub-problems of the same type
  - The sub-problems are relatively independent of one another (ie, no overlap in effort)
  - An acceptable solution to the problem can be constructed from acceptable solutions to sub-problems

❖ <u>Approach:</u>

- Recursion (Top-down approach)

Well-Known Uses

- Searching
  - Binary search
- Sorting
  - Merge Sort
  - Quick Sort
- Mathematics
  - Polynomial and matrix multiplication
  - Exponentiation
  - Large integer manipulation

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
- Merge sorted arrays B and C into array A

Mergesort

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

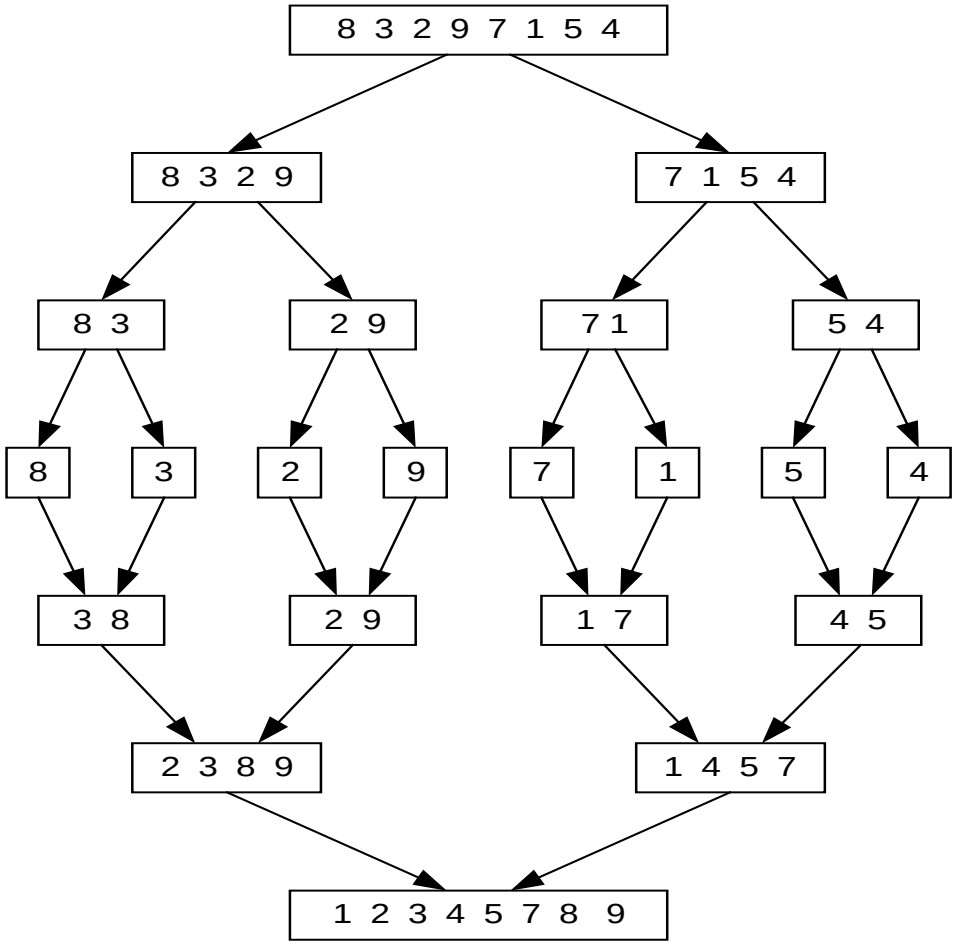
//Sorts array $A[0..n-1]$ by recursive mergesort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
if $n > 1$
    copy $A[0..\lfloor n/2 \rfloor - 1]$ to $B[0..\lfloor n/2 \rfloor - 1]$
    copy $A[\lfloor n/2 \rfloor..n-1]$ to $C[0..\lceil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

Mergesort

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 $A[k..p+q-1]$

Mergesort


8 3 2 9 7 1 5 4
8 3 2 9
7 1 5 4
8 3
2 9
7 1
5 4
8
3
2
9
7
1
5
4
3 8
2 9
1 7
4 5
2 3 8 9
1 4 5 7
1 2 3 4 5 7 8 9

Analysis of Mergesort

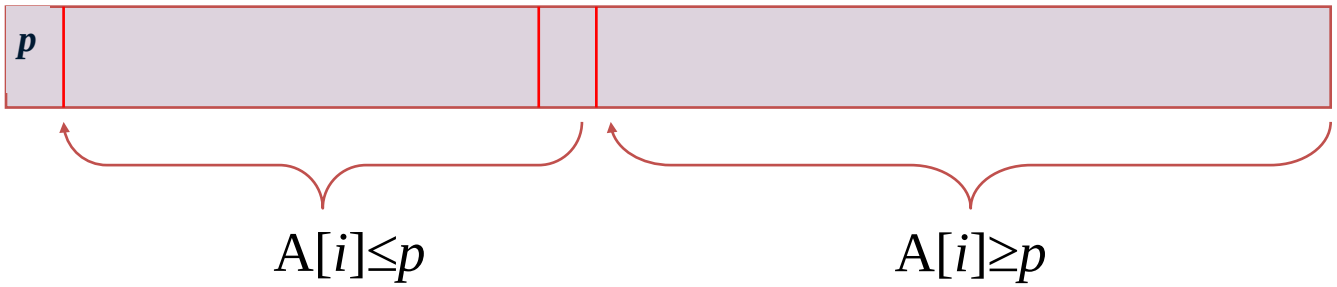
- All cases have same efficiency: $\Theta(n \log n)$
- Number of comparisons in the worst case is close to theoretical minimum for comparison-based sorting:

$$\lceil \log_2 n! \rceil \approx n \log_2 n - 1.44n$$

- Space requirement: $\Theta(n)$ (not in-place)
- Can be implemented without recursion (bottom-up)

Quicksort

- Select a *pivot* (partitioning element) – here, the first element
- 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 or equal to the pivot (see next slide for an algorithm)



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

Analysis of Quicksort

- Best case: split in the middle — $\Theta(n \log n)$
- Worst case: sorted array — $\Theta(n^2)$
- Average case: random arrays — $\Theta(n \log n)$

- **Improvements:**
  - better pivot selection: median of three partitioning
  - switch to insertion sort on small subfiles
  - elimination of recursion

- Quicksort is considered the method of choice for internal sorting of large files ($n \geq 10000$)

Binary Search

- Very efficient algorithm for searching in **sorted array:**

K

vs

A[0] . . . A[m] . . . A[n-1]

- If K = A[m], stop (successful search);
- otherwise, continue searching by the same method

in A[0..m-1] if K < A[m],

and in A[m+1..n-1] if K > A[m]

Analysis of Binary Search

- Time efficiency

  Worst-case recurrence:

  $C_w(n) = 1 + C_w(\lfloor n/2 \rfloor), \quad C_w(1) = 1$

  solution: $C_w(n) = \lceil \log_2(n+1) \rceil$

- Optimal for searching a sorted array
- Limitations: must be a sorted array (not linked list)

Binary Tree Algorithms

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

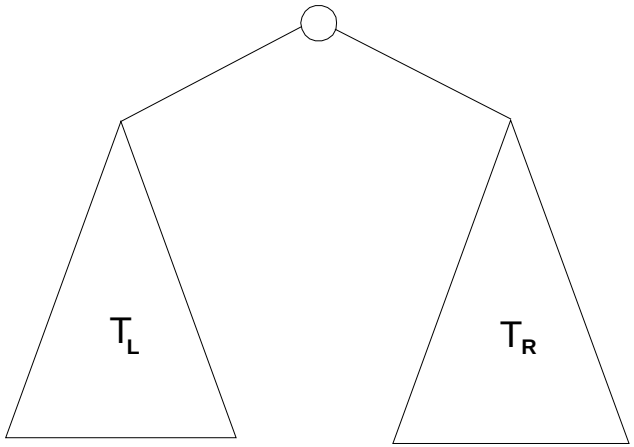
Ex. 1: Classic traversals (preorder, inorder, postorder)

- **Algorithm *Inorder*(*T*)**
  if *T* ≠ ∅
  ***Inorder*(*Tleft*)**
  print(root of *T*)
  ***Inorder*(*Tright*)**

- **Efficiency: Θ(*n*)**

Binary Tree Algorithms

Ex. 2: Computing the height of a binary tree



$h(T) = \max\{ h(T_L), h(T_R) \} + 1$ if $T \neq \emptyset$ and $h(\emptyset) = -1$

Efficiency: $\Theta(n)$

Combine the D&C algorithm with other simple algorithm

Switching point

- Recursive method may not show the advantage in the case of small n as compared to the alternative algorithms
- Recursive algorithm requires a fair amount of overhead.

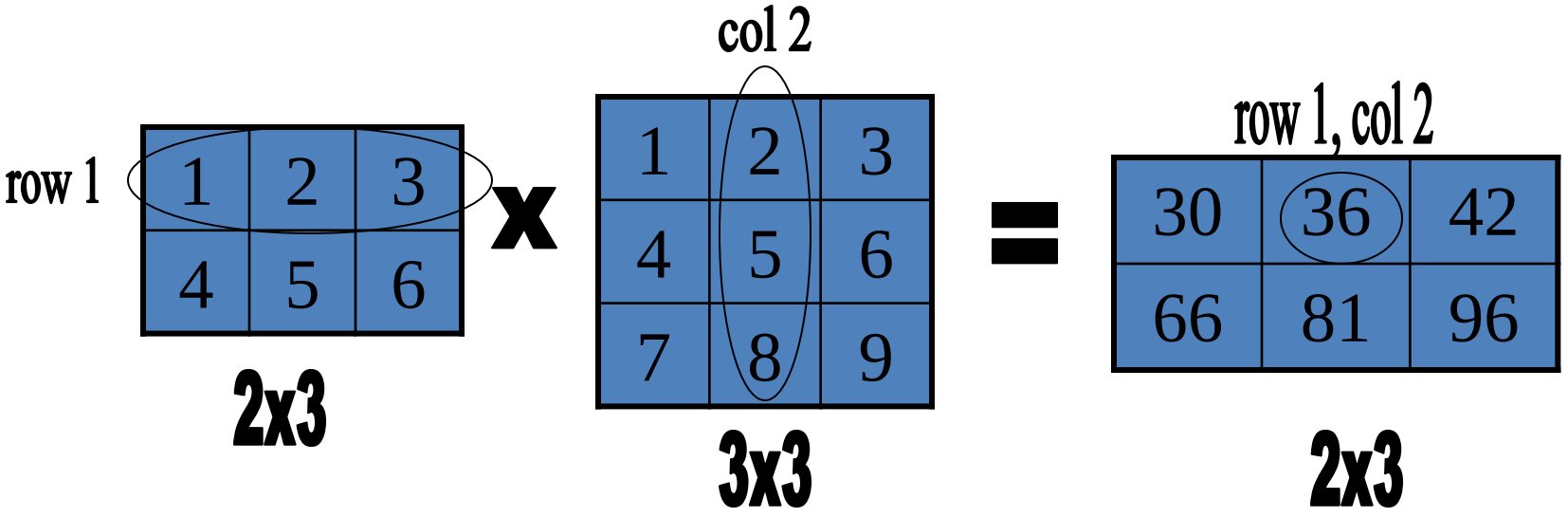
- We need to determine for what value of n it is at least as fast to call an alternative algorithm as it is to divide the instance further.

- The dividing process is stopped in a certain switching point (or threshold) for recursive algorithm, and then is switched to the alternative algorithm

Example

- Use the standard matrix multiplication to multiply two 2 *2 matrices

Matrix Multiplication



1x2 + 2x5 + 3x8 = 2 + 10 + 24 = 36

- rows from the first matrix
- columns from the second matrix

Matrix Multiplication

- "inner" dimensions must match
- result is "outer" dimension
- Examples:
  - 2×3 X 3×3 **=** 2x3
  - 3×4 X 4×5 **=** 3x5
  - 2×3 X 4×3 **=** **cannot multiply**

Matrix Multiplication

Divide & Conquer:

- Easy to break into subproblems
  - Multiplication can be performed *blockwise*

$$X \times Y = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \times \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & DF + DH \end{bmatrix}$$

- Divide & Conquer steps…
  - Divide each matrix into 4 blocks
  - Recursively compute each multiplication
  - Combine multiplications with a few additions and assemble final matrix

When not to use D&C algorithm

- Two cases:

  - An instance of size n is divided into two or more instances each almost of size n, e.g., Recursive Fibonacci term is $O(2^n)$, while the iterative one is O(n)

  - An instance of size n is divided into almost n instances of size n/c (c is constant)