
A dark blue vertical bar is positioned on the left side of the slide, spanning the height of the main content area.

# Divide-and-Conquer

A light blue vertical bar is positioned on the left side of the slide, spanning the height of the footer area.

# Divide-and-Conquer

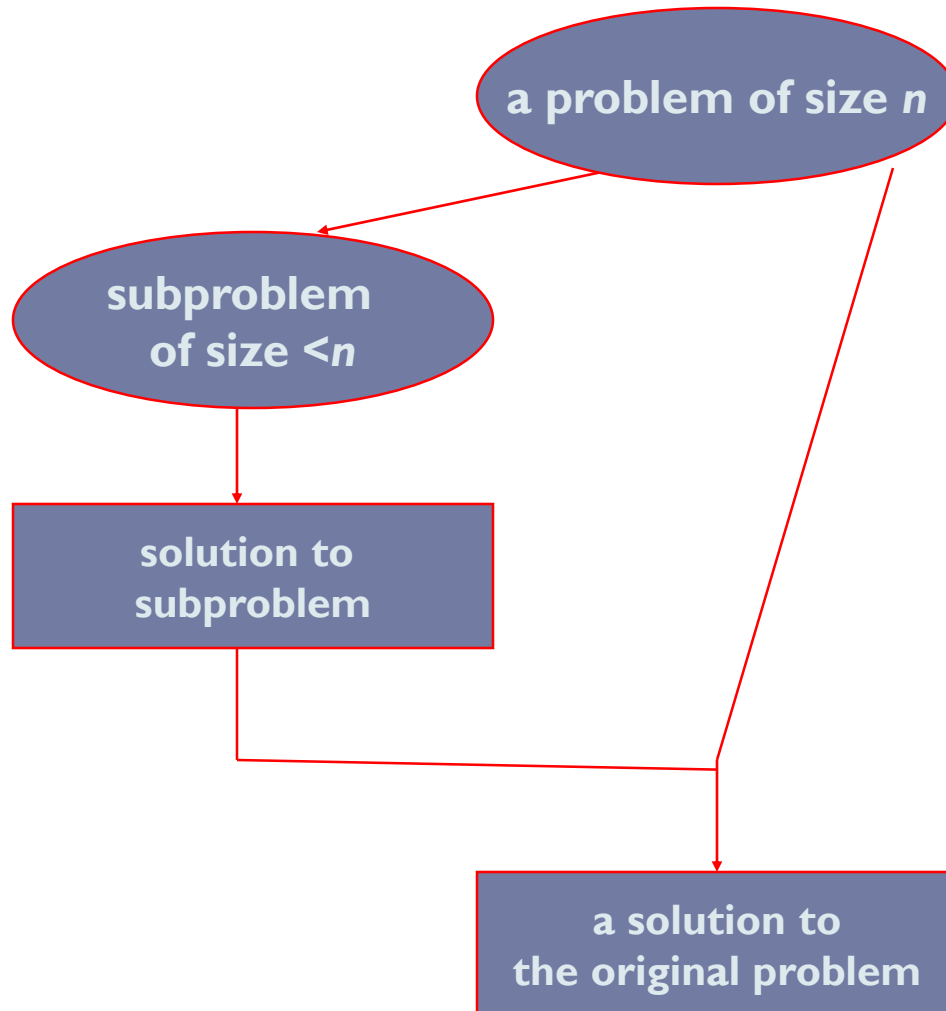
---

The most well-known algorithm design strategy:

- ▶ Divide instance of problem into two or more smaller instances
- ▶ Solve smaller instances recursively
- ▶ Obtain solution to original (larger) instance by combining these solutions

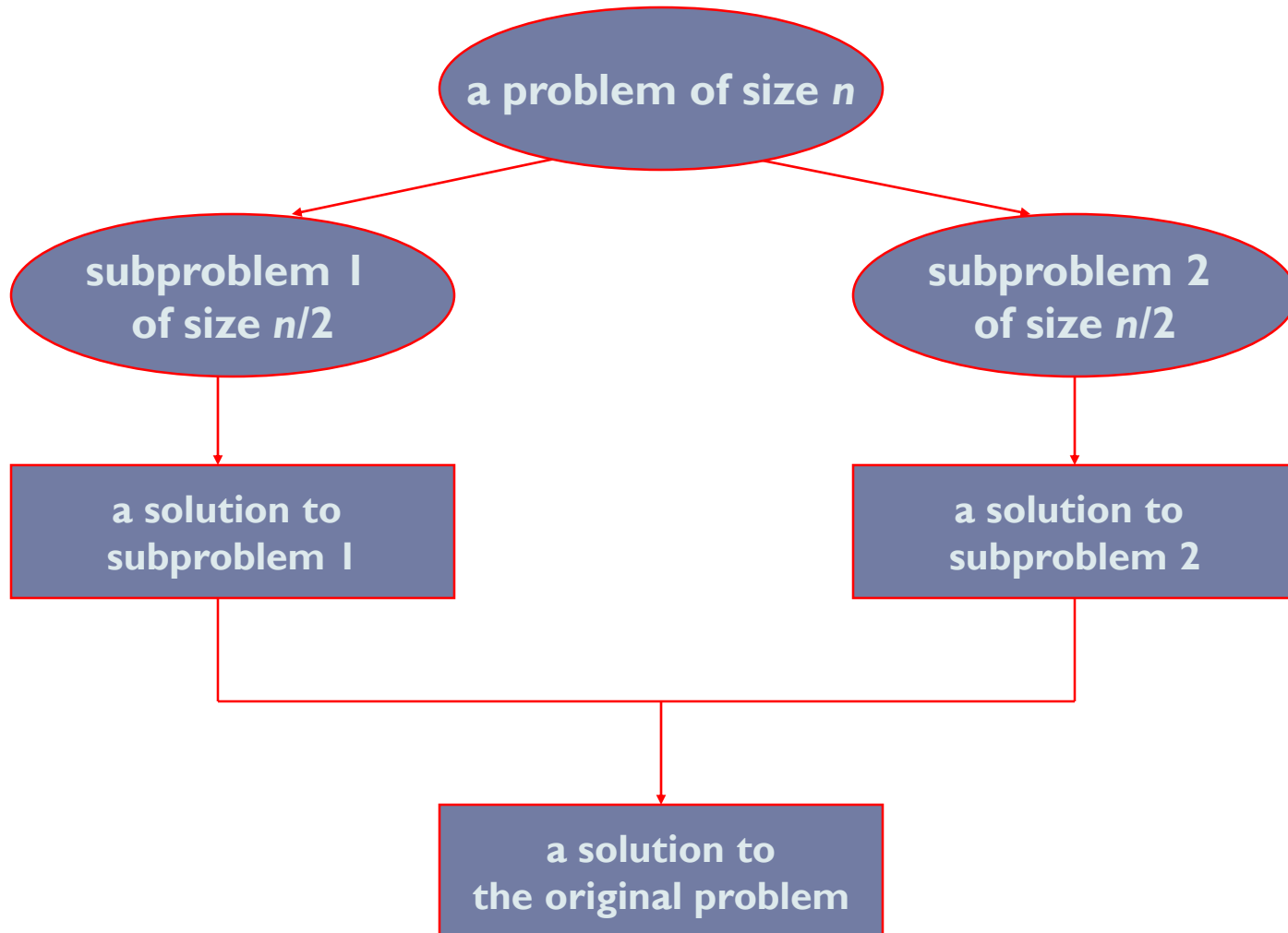
# Decrease-and-Conquer Technique

---



# Divide-and-Conquer Technique

---



# What's the difference?

---

Consider the problem of exponentiation: Compute  $a^n$

- ▶ Brute Force:

$$a^n = a \times a \times \cdots \times a$$

- ▶ Decrease by one:

$$a^n = a^{n-1} \times a$$

- ▶ Decrease by constant factor:

$$a^n = \left(a^{\frac{n}{2}}\right)^2$$

- ▶ Divide and conquer:

$$a^n = a^{\lfloor \frac{n}{2} \rfloor} \times a^{\lceil \frac{n}{2} \rceil}$$

# Divide-and-Conquer

---

- ▶ Sorting: merge-sort and quick-sort
- ▶ Binary tree traversals
- ▶ Binary search
- ▶ Multiplication of large integers
- ▶ Matrix multiplication: Strassen's algorithm
- ▶ Closest-pair and convex-hull algorithms

# General Divide-and-Conquer Recurrence

---

Master Theorem:

$$T(n) = a T\left(\frac{n}{b}\right) + f(n) \quad \text{where } f(n) \in \Theta(n^d), d \geq 0$$

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

The same results hold with  $O$  and  $\Omega$  too.

# General Divide-and-Conquer Recurrence

---

$$T(n) = a T\left(\frac{n}{b}\right) + f(n) \quad \text{where } f(n) \in \Theta(n^d), d \geq 0$$

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

Examples:

$$T(n) = 4 T\left(\frac{n}{2}\right) + n \implies T(n) \in \Theta(?)$$



# General Divide-and-Conquer Recurrence

---

$$T(n) = a T\left(\frac{n}{b}\right) + f(n) \quad \text{where } f(n) \in \Theta(n^d), d \geq 0$$

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

Examples:

$$T(n) = 4 T\left(\frac{n}{2}\right) + n \Rightarrow T(n) \in \Theta(?)$$

$$a = 4, b = 2, d = 1 \rightarrow 4 > 2^1 \rightarrow T(n) \in \Theta(n^{\log_2 4}) = \Theta(n^2)$$

# General Divide-and-Conquer Recurrence

---

$$T(n) = a T\left(\frac{n}{b}\right) + f(n) \quad \text{where } f(n) \in \Theta(n^d), d \geq 0$$

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

Examples:

$$T(n) = 4 T\left(\frac{n}{2}\right) + n^2 \Rightarrow T(n) \in \Theta(?)$$

# General Divide-and-Conquer Recurrence

---

$$T(n) = a T\left(\frac{n}{b}\right) + f(n) \quad \text{where } f(n) \in \Theta(n^d), d \geq 0$$

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

Examples:

$$T(n) = 4 T\left(\frac{n}{2}\right) + n^2 \Rightarrow T(n) \in \Theta(?)$$

$$a = 4, b = 2, d = 2 \rightarrow 4 = 2^2 \rightarrow T(n) \in \Theta(n^2 \log n)$$

# General Divide-and-Conquer Recurrence

---

$$T(n) = a T\left(\frac{n}{b}\right) + f(n) \quad \text{where } f(n) \in \Theta(n^d), d \geq 0$$

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

Examples:

$$T(n) = 4 T\left(\frac{n}{2}\right) + n^3 \Rightarrow T(n) \in \Theta(?)$$

# General Divide-and-Conquer Recurrence

---

$$T(n) = a T\left(\frac{n}{b}\right) + f(n) \quad \text{where } f(n) \in \Theta(n^d), d \geq 0$$

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

Examples:

$$T(n) = 4 T\left(\frac{n}{2}\right) + n^3 \Rightarrow T(n) \in \Theta(?)$$

$$a = 4, b = 2, d = 3 \rightarrow 4 < 2^3 \rightarrow T(n) \in \Theta(n^3)$$

# Mergesort

---

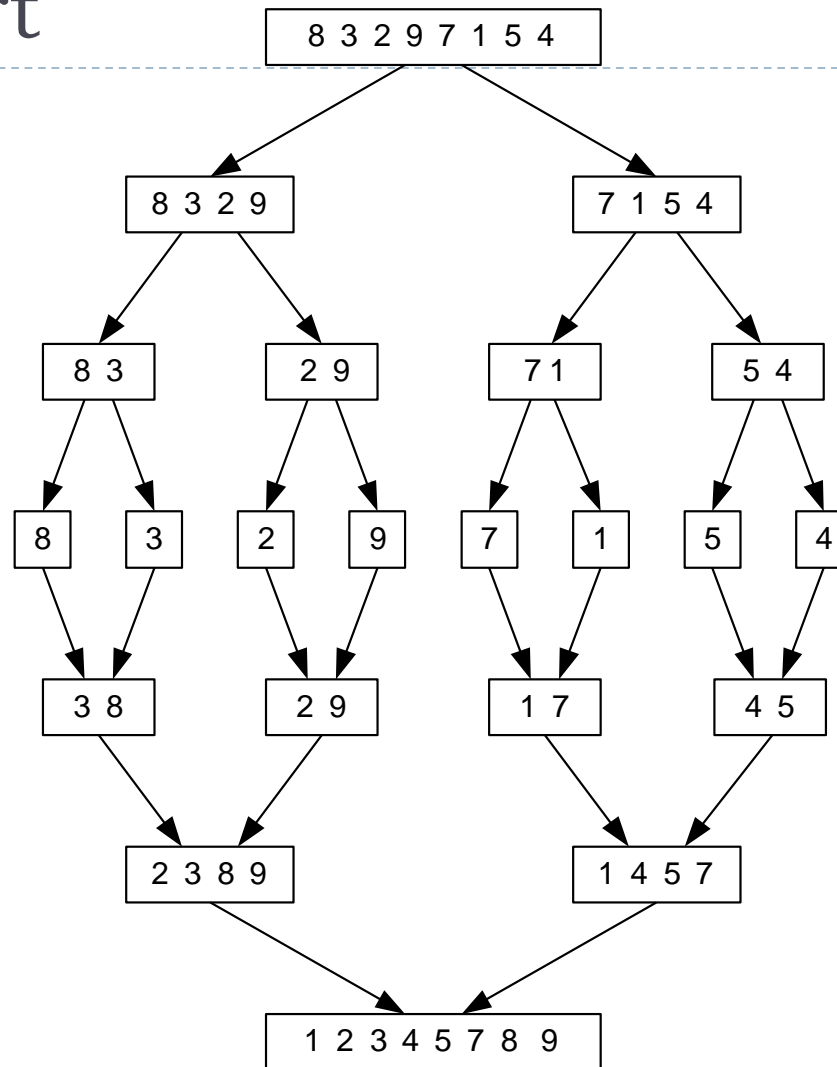
- ▶ Split array  $A[0..n-1]$  in two about equal halves and make copies of each half in arrays  $B$  and  $C$
- ▶ Sort arrays  $B$  and  $C$  recursively
- ▶ Merge sorted arrays  $B$  and  $C$  into array  $A$  as follows:
  - ▶ Repeat the following until no elements remain in one of the arrays:
    - ▶ compare the first elements in  $B$  and  $C$
    - ▶ 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  $A$ .

# Mergesort

---

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

# Mergesort





# 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

**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*( $B, C, A$ )

**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

**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  $C[0..\lfloor n/2 \rfloor - 1]$

*Mergesort*( $B[0..\lfloor n/2 \rfloor - 1]$ )

*Mergesort*( $C[0..\lfloor n/2 \rfloor - 1]$ )

*Merge*( $B, C, A$ )

**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**  $B[i] \leq C[j]$

$A[k] \leftarrow B[i]; i \leftarrow i + 1$

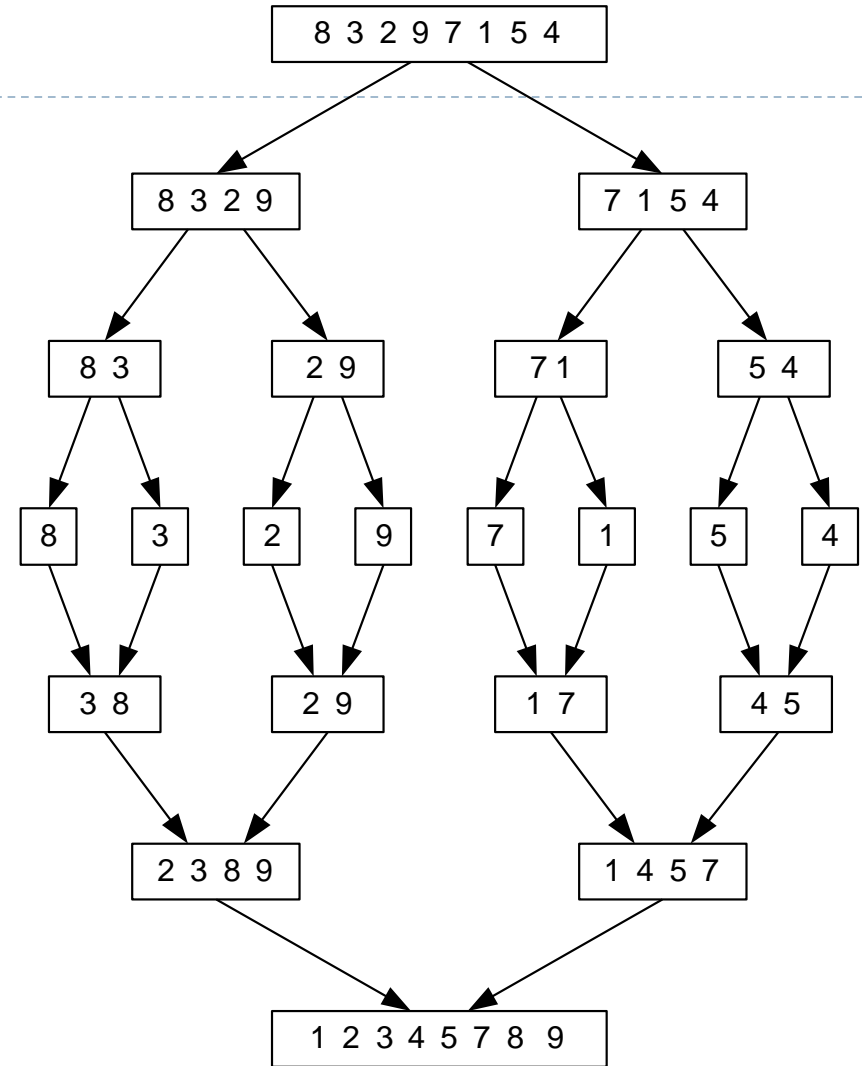
**else**  $A[k] \leftarrow C[j]; j \leftarrow j + 1$

$k \leftarrow k + 1$

**if**  $i = p$

    copy  $C[j..q-1]$  to  $A[k..p+q-1]$

**else** copy  $B[i..p-1]$  to  $A[k..p+q-1]$



$$C_{\text{worse}}(n) = 2C_{\text{worst}}(n/2) + n - 1 \in \theta(n \log n)$$

# Mergesort

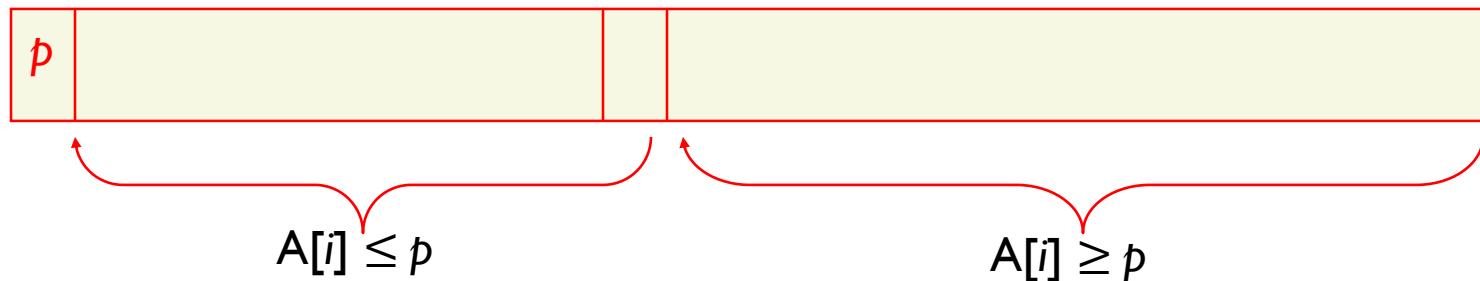
---

- ▶ 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) – e.g. 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



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

# Quicksort

---

**ALGORITHM** *Quicksort*( $A[l..r]$ )

//Sorts a subarray by quicksort

//Input: Subarray of array  $A[0..n - 1]$ , defined by its left and right

// indices  $l$  and  $r$

//Output: Subarray  $A[l..r]$  sorted in nondecreasing order

**if**  $l < r$

$s \leftarrow \text{Partition}(A[l..r])$  //  $s$  is a split position

*Quicksort*( $A[l..s - 1]$ )

*Quicksort*( $A[s + 1..r]$ )

---

**ALGORITHM** *HoarePartition*( $A[l..r]$ )

//Partitions a subarray by Hoare's algorithm, using the first element  
// as a pivot  
//Input: Subarray of array  $A[0..n - 1]$ , defined by its left and right  
// indices  $l$  and  $r$  ( $l < r$ )  
//Output: Partition of  $A[l..r]$ , with the split position returned as  
// this function's value  
 $p \leftarrow A[l]$   
 $i \leftarrow l; j \leftarrow r + 1$   
**repeat**  
    **repeat**  $i \leftarrow i + 1$  **until**  $A[i] \geq p$   
    **repeat**  $j \leftarrow j - 1$  **until**  $A[j] \leq p$   
    swap( $A[i], A[j]$ )  
**until**  $i \geq j$   
swap( $A[i], A[j]$ ) //undo last swap when  $i \geq j$   
swap( $A[l], A[j]$ )  
**return**  $j$



---

**ALGORITHM** *Quicksort*( $A[l..r]$ )

//Sorts a subarray by quicksort  
//Input: Subarray of array  $A[0..n-1]$ , defined by its left and right  
// indices  $l$  and  $r$   
//Output: Subarray  $A[l..r]$  sorted in nondecreasing order  
**if**  $l < r$   
     $s \leftarrow \text{Partition}(A[l..r])$  //  $s$  is a split position  
    *Quicksort*( $A[l..s-1]$ )  
    *Quicksort*( $A[s+1..r]$ )

**ALGORITHM** *HoarePartition*( $A[l..r]$ )

//Partitions a subarray by Hoare's algorithm, using the first element  
// as a pivot  
//Input: Subarray of array  $A[0..n-1]$ , defined by its left and right  
// indices  $l$  and  $r$  ( $l < r$ )  
//Output: Partition of  $A[l..r]$ , with the split position returned as  
// this function's value  
 $p \leftarrow A[l]$   
 $i \leftarrow l; j \leftarrow r + 1$   
**repeat**  
    **repeat**  $i \leftarrow i + 1$  **until**  $A[i] \geq p$   
    **repeat**  $j \leftarrow j - 1$  **until**  $A[j] \leq p$   
    swap( $A[i], A[j]$ )  
**until**  $i \geq j$   
swap( $A[i], A[j]$ ) //undo last swap when  $i \geq j$   
swap( $A[l], A[j]$ )  
**return**  $j$

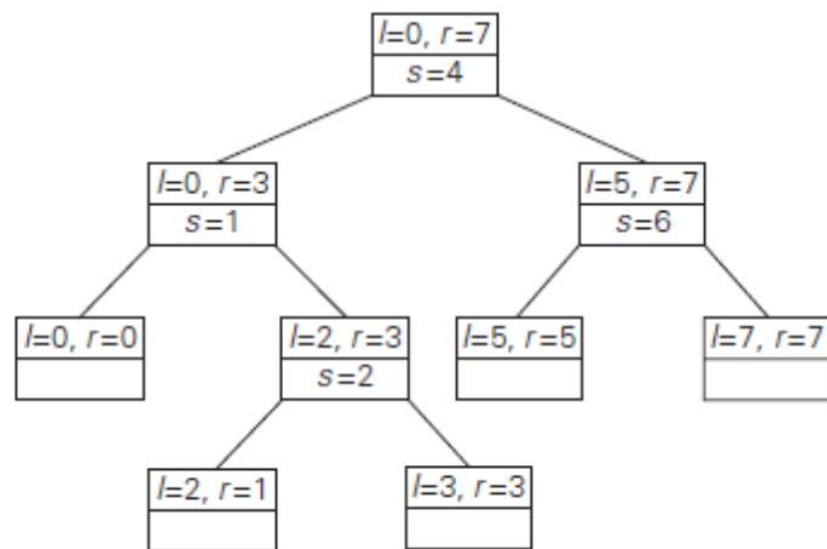
---

0	1	2	3	4	5	6	7
5	<i>i</i> 3	1	9	8	2	4	<i>j</i> 7

$l=0, r=7$
$s=4$

0	1	2	3	4	5	6	7
	<i>i</i>						<i>j</i>
5	3	1	9	8	2	4	7
5	3	1	9	8	2	4	7
5	3	1	4	8	2	9	7
5	3	1	4	8	2	9	7
5	3	1	4	2	8	9	7
5	3	1	4	2	8	9	7
2	3	1	4	5	8	9	7
2	<i>i</i>		<i>j</i>				
2	3	1	4				
2	<i>i</i>	<i>j</i>					
2	3	1	4				
2	<i>i</i>	<i>j</i>					
2	1	3	4				
2	<i>j</i>	<i>i</i>	4				
1	2	3	4				
1							
		3	<i>ij</i>				
		<i>j</i>	4				
		3	<i>i</i>				
			4				
					8	<i>i</i>	<i>j</i>
					8	9	7
					8	7	<i>j</i>
					8	7	<i>i</i>
					7	8	9
					7		
							9

(a)



(b)

# Quicksort

---

- ▶ Best case:

- ▶ split in the middle —  $C_{best}(n) = 2C_{best}(n/2) + n \in \theta(n \log n)$

- ▶ Worst case:

- ▶ sorted array! —  $C_{worst}(n) = (n+1) + n + \dots + 3 \in \theta(n^2)$

- ▶ Average case:

- ▶ random arrays —  $C_{avg}(n) = \frac{1}{n} \sum_{s=0}^{n-1} [(n+1) + C_{avg}(s) + C_{avg}(n-1-s)] \in \theta(n \log n)$

# Quicksort

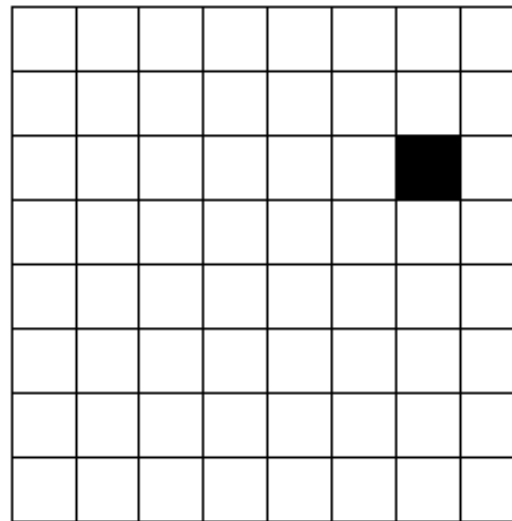
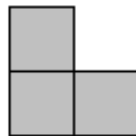
---

- ▶ Improvements:
  - ▶ better pivot selection
  - ▶ median of three partitioning (uses the median of the leftmost, rightmost, and the middle element of the array)
  - ▶ switch to insertion sort on small sub-files
  - ▶ elimination of recursion
- ▶ These combine to 20 – 25% improvement
- ▶ Considered the method of choice for internal sorting of large files ( $n \geq 10000$ )

# Discussion

---

*Tromino puzzle* A tromino is an L-shaped tile formed by adjacent 1-by-1 squares. The problem is to cover any  $2^n$ -by- $2^n$  chessboard with one missing square (anywhere on the board) with trominoes. Trominoes should cover all the squares of the board except the missing one with no overlaps.

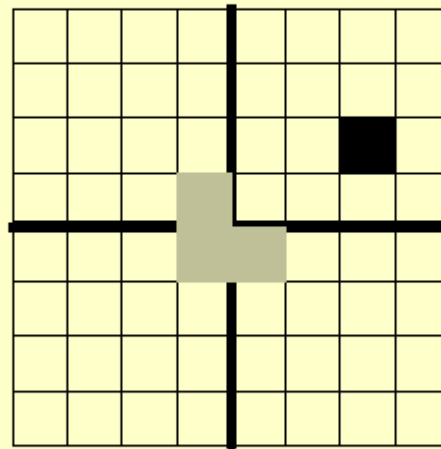


Design a divide-and-conquer algorithm for this problem.

# Discussion

---

For  $n > 1$ , we can always place one L-tromino at the center of the  $2^n \times 2^n$  chessboard with one missing square to reduce the problem to four subproblems of tiling  $2^{n-1} \times 2^{n-1}$  boards, each with one missing square too. The orientation of this centrally placed piece is determined by the board's quarter with the missing square as shown by the example below.



Then each of the four smaller problems can be solved recursively until a trivial case of a  $2 \times 2$  board with a missing square is reached.

# Discussion

▷ *Nuts and bolts* You are given a collection of  $n$  bolts of different widths and  $n$  corresponding nuts. You are allowed to try a nut and bolt together, from which you can determine whether the nut is larger than the bolt, smaller than the bolt, or matches the bolt exactly. However, there is no way to compare two nuts together or two bolts together. The problem is to match each bolt to its nut. Design an algorithm for this problem with average-case efficiency in  $\Theta(n \log n)$ . [Raw91]



# Discussion

---

Randomly select a nut and try each of the bolts for it to find the matching bolt and separate the bolts that are smaller and larger than the selected nut into two disjoint sets. Then try each of the unmatched nuts against the matched bolt to separate those that are larger from those that are smaller than the bolt. As a result, we've identified a matching pair and partitioned the remaining nuts and bolts into two smaller independent instances of the same problem. The average number of nut-bolt comparisons  $C(n)$  is defined by the recurrence very similar to the one for quicksort in Section 4.2:

$$C(n) = \frac{1}{n} \sum_{s=0}^{n-1} [(2n-1) + C(s) + C(n-1-s)], \quad C(1) = 0, \quad C(0) = 0.$$

The solution to this recurrence can be shown to be in  $\Theta(n \log n)$  by repeating the steps outlined in the solution to Problem 7.

# Programming Exercise

---

- ▶ Apply quicksort to sort the list E, X, A, M, P, L, E in alphabetical order. Draw the tree of the recursive calls made. Implement the algorithm to verify your answer.

# Programming Exercise

- Apply quicksort to sort the list E, X, A, M, P, L, E in alphabetical order. Draw the tree of the recursive calls made.

0	1	2	3	4	5	6
E	<sup>i</sup> X	A	M	P	L	<sup>j</sup> E
E	E	<sup>j</sup> A	<sup>i</sup> M	P	L	X
A	E	E	M	P	L	X
A	<sup>i,j</sup> E					
<sup>j</sup> A	<sup>i</sup> E					
A	E					
	E					

M	<sup>i</sup> P	L	<sup>j</sup> X
M	<sup>i</sup> P	<sup>j</sup> L	X
M	<sup>i</sup> L	<sup>j</sup> P	X
M	<sup>j</sup> L	<sup>i</sup> P	X
L	M	P	X
L			
	<sup>i,j</sup> P	<sup>i</sup> X	
	<sup>j</sup> P	<sup>i</sup> X	
	P	X	
		X	

