Divide and Conquer

CSC 411

Richard Kelley

Review: Decrease-and-Conquer

1. Reduce problem instance to smaller instance of the same problem
2. Solve smaller instance
3. Extend solution of smaller instance to obtain solution to original instance

- Can be implemented either top-down or bottom-up
- (Richard doesn't love this terminology.)

3 Types of Decrease and Conquer

- *Decrease by a constant* (usually by 1):
  - insertion sort
  - graph traversal algorithms (DFS and BFS)
  - topological sorting
  - algorithms for generating permutations

- *Decrease by a constant factor* (usually by half)
  - binary search and bisection method
  - exponentiation by squaring

- *Variable-size decrease*
  - Euclid's algorithm
  - selection by partition

Divide-and-Conquer

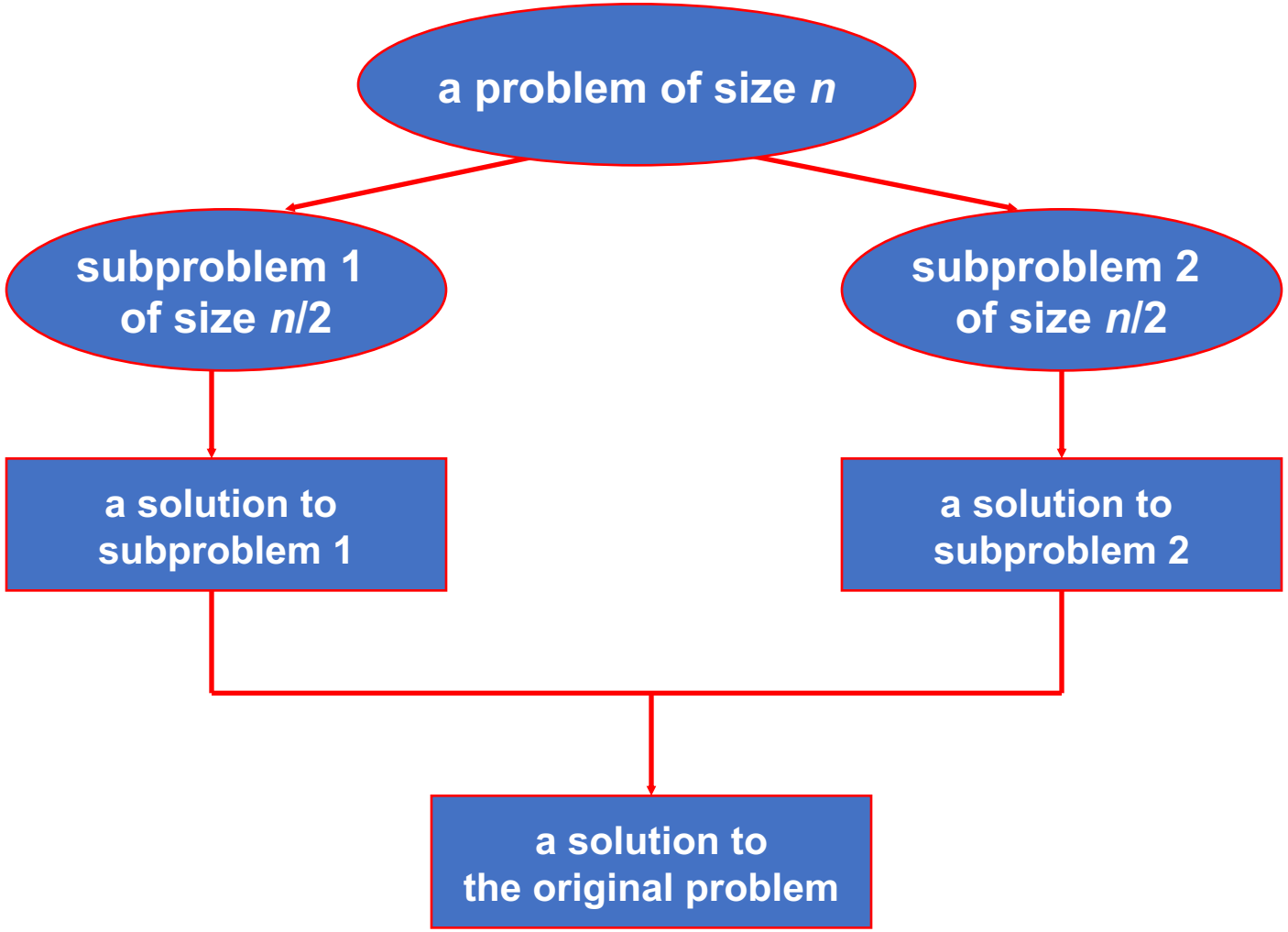
most-well known algorithm design strategy:

1. Divide instance of problem into two or more smaller instances

2. Solve smaller instances recursively

3. Obtain solution to original (larger) instance by combining these solutions

Divide-and-Conquer Technique (cont.)



Divide-and-Conquer Examples

- Sorting: mergesort and quicksort
- Binary tree traversals
- Multiplication of large integers and Matrix multiplication: Strassen's algorithm
- Closest-pair and convex-hull algorithms.

Review: 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 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.

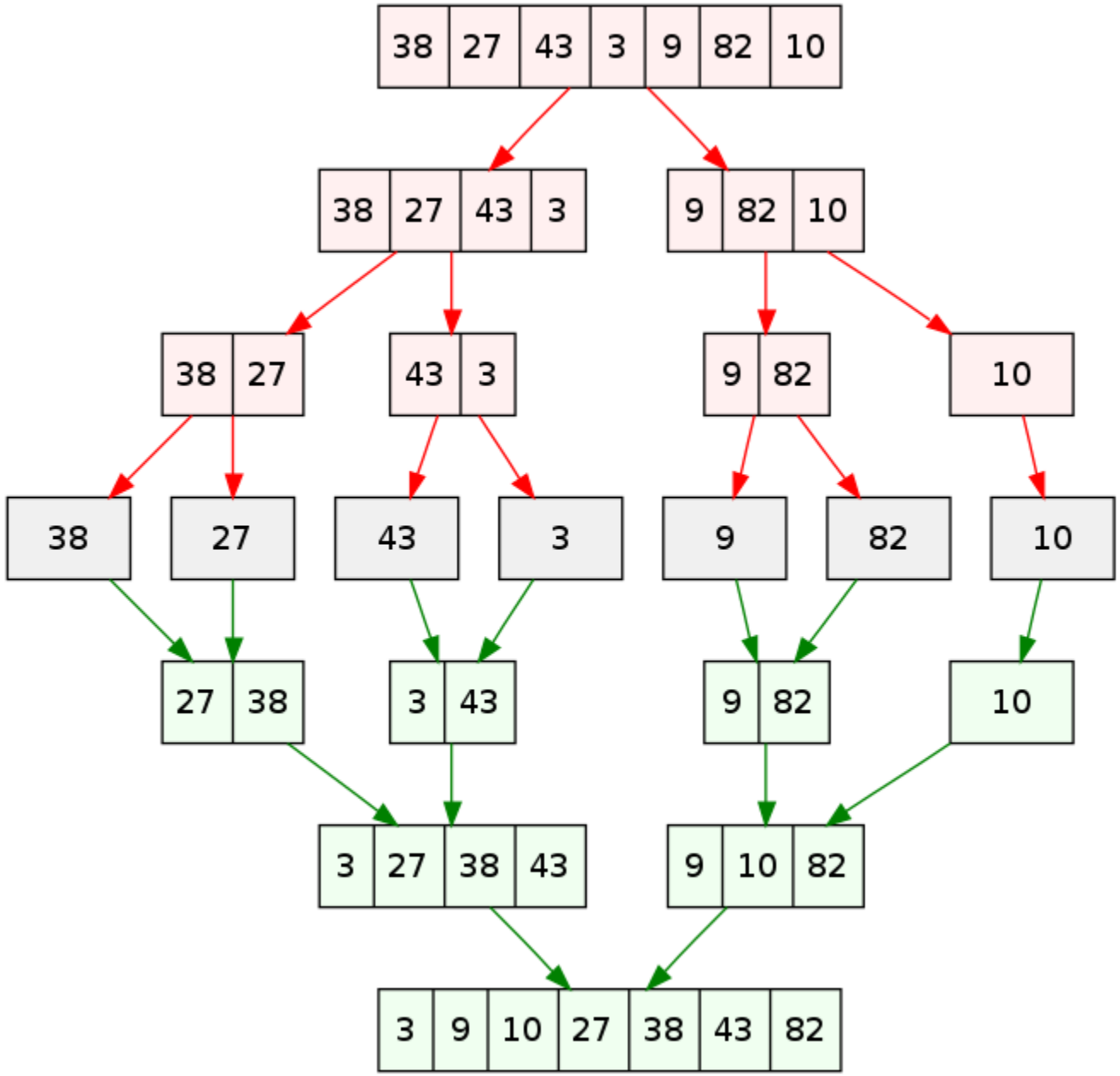
Pseudocode of Mergesort

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)

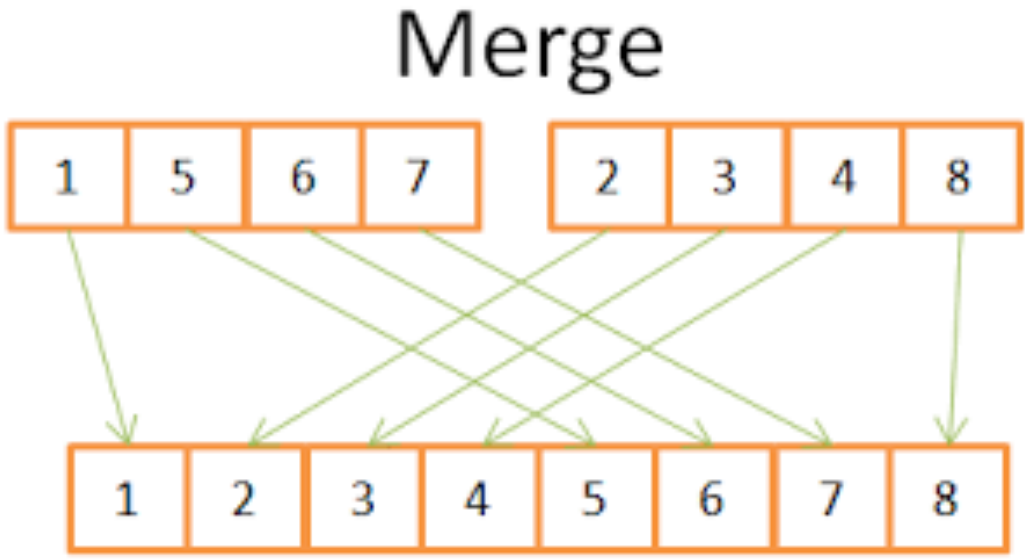
Does anything about this seem wasteful?

Example


38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82

Merging Two Sorted Arrays

- To merge two sorted array into a third (sorted) array, repeatedly compare the two least elements and copy the smaller of the two onto the third array.



# of comparison:
7 in this example
(1, 2)
(5, 2)
(5, 3)
(5, 4)
(5, 8)
(6, 8)
(7, 8)

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

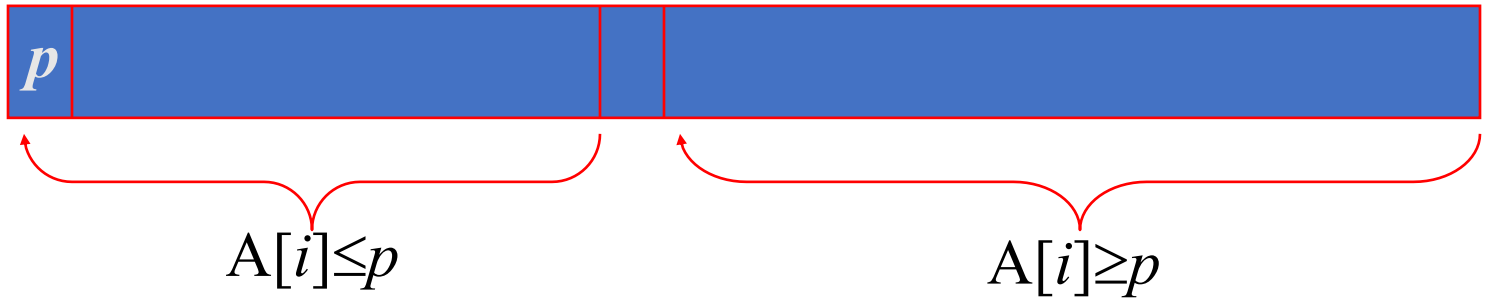
Analysis of Mergesort

- All cases have same efficiency: $\Theta(n \log n)$

- Why? See below…

Quicksort

- Select a *pivot* (partitioning element) – here, the first element



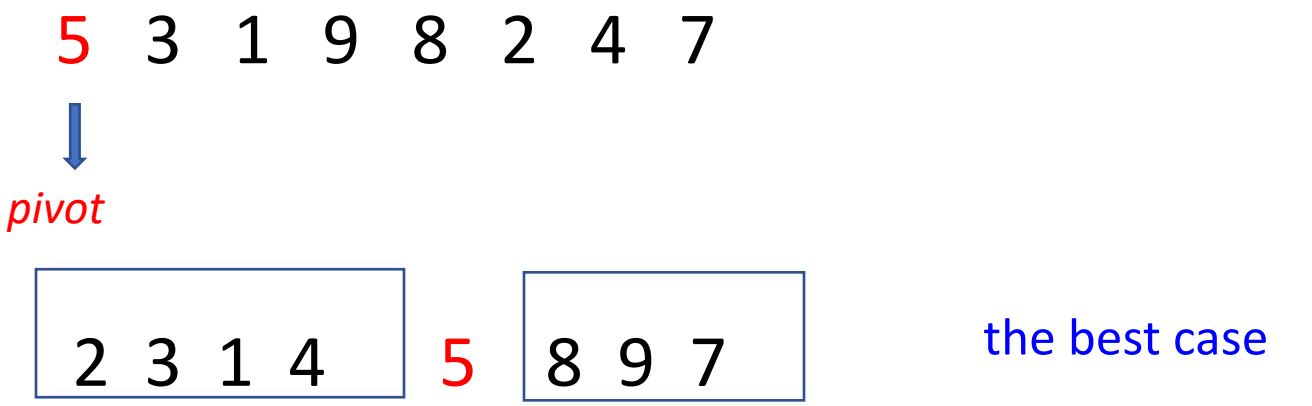
- Rearrange the list so that all the elements in the first *s* positions are smaller than or equal to the pivot and all the elements in the remaining *n-s* positions are larger than or equal to the pivot (see next slide for an algorithm)
- Exchange the pivot with the last element in the first (i.e., ≤) subarray — the pivot is now in its final position
- Sort the two subarrays recursively

Quicksort Pseudocode

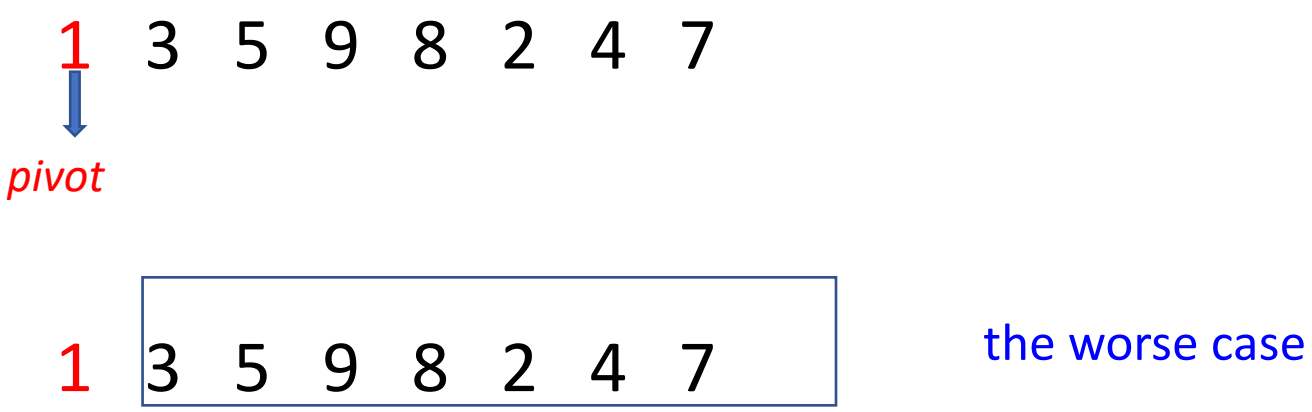
ALGORITHM *Quicksort*$(A[l..r])$

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

Quicksort Example



the best case



the worse case

The slickest Quicksort Implementation: Haskell

```
quicksort [] = []
quicksort (x:xs) =
  quicksort [y | y <- xs, y <= x]
  ++ [x] ++
  quicksort [y | y <- xs, y > x]
```

But probably not one you should ever use…

Partitioning Algorithm

Algorithm *Partition*$(A[l..r])$
//Partitions a subarray by using its first element as a pivot
//Input: A subarray $A[l..r]$ of $A[0..n-1]$, defined by its left and right
// indices l and r $(l < r)$
//Output: A 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** $A[j] \cdot p$
  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

Analysis of Quicksort

- Best case: split in the middle $\Theta(n \log n)$

- Why?

Analysis of Quicksort

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

- What is the worst-case for quicksort?

- How do we analyze mergesort and quicksort?

Master Theorem

Let T be an increasing function that satisfies the recurrence relation:

$$T(n) = a\,T(n/b) + cn^d$$

whenever $n = b^k$, where k is a positive integer, a >= 1, b is an integer greater than 1, c is a positive real number, and d is a non-negative real number. Then:

$$T(n) = \begin{cases} O(n^d) & \text{if } a < b^d \quad \text{case 1} \\ O(n^d \log_b n) & \text{if } a = b^d \quad \text{case 2} \\ O(n^{\log_b a}) & \text{if } a > b^d \quad \text{case 3} \end{cases}$$

a >= 1: The number of subproblems

b > 1: Amount by which problems shrink

cn^d: Amount of nonrecursive work at each level of recursion.

The cases are (effectively) comparing cn^d with n^(log_b(a))

Analysis of Mergesort

- All cases have same efficiency: $\Theta(n \log n)$

- Why? What is its growth function?
  - $T(n) = 2\,T(n/2) + f(n)$ =? The time needed for merging

  - $T_{worst}(n) = 2\,T_{worst}(n/2) + n-1$ *for n>1, T(1)=0*

  - *Using* **Master Theorem**

    1. If $a < b^d$, $T(n) \in \Theta(n^d)$
    2. If $a = b^d$, $T(n) \in \Theta(n^d \log n)$
    3. If $a > b^d$, $T(n) \in \Theta(n^{\log_b a})$

    a =2, b=2, d=1

    => Case 2, $T(n) \in \Theta(n \log n)$

Analysis of Quicksort

- Best case: split in the middle $\Theta(n \log n)$

- Why?
  - $T_{best}(n) = 2\,T_{best}(n/2) + f(n)$ =? The time needed for partition
  - $T_{best}(n) = 2\,T_{best}(n/2) + n-1$ *for n>1, T(1)=0*
  - *Using* **Master Theorem**

1. If $a < b^d$, $T(n) \in \Theta(n^d)$
2. If $a = b^d$, $T(n) \in \Theta(n^d \log n)$
3. If $a > b^d$, $T(n) \in \Theta(n^{\log_b a})$

a =2, b=2, d=1

=> Case 2, $T(n) \in \Theta(n \log n)$

Analysis of Quicksort

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