WIA2005: Algorithm Design and Analysis

Lecture 5: Divide & Conquer Algorithm

Asmiza A. Sani

Semester 2, Session 2023/24

Learning objectives

- The student will know and understand the following:
  - Algorithm design paradigm: Divide and Conquer
  - Merge sort
  - Quick sort

Algorithm Design Paradigm

- When we are designing an algorithm, there are several high-level approach that can be taken to solve a certain class of problems.
- Common ones are:
  - Divide and conquer
    - Recursively breaking down a problem into 2 or more sub-problems of the same type.
    - No overlapping sub-problem.
  - Dynamic programming
    - Breaking down a problem into a collection of simpler problem.
    - Sub-problem must overlap.
  - Greedy algorithms
    - Making a locally optimal decision at each stage.
- Others:
  - Brute force
  - Backtracking

The Divide and Conquer Design Paradigm

- The Divide and Conquer algorithm apply the concept of dividing problems into smaller sub-problem.
- The approach:

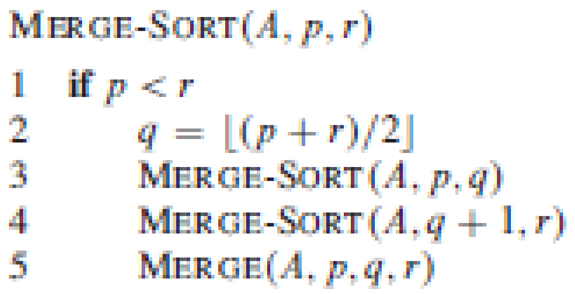
1. *Divide* the problem (instance) into subproblems.
2. *Conquer* the subproblems by solving them recursively.
3. *Combine* subproblem solutions.

Merge Sort Algorithm

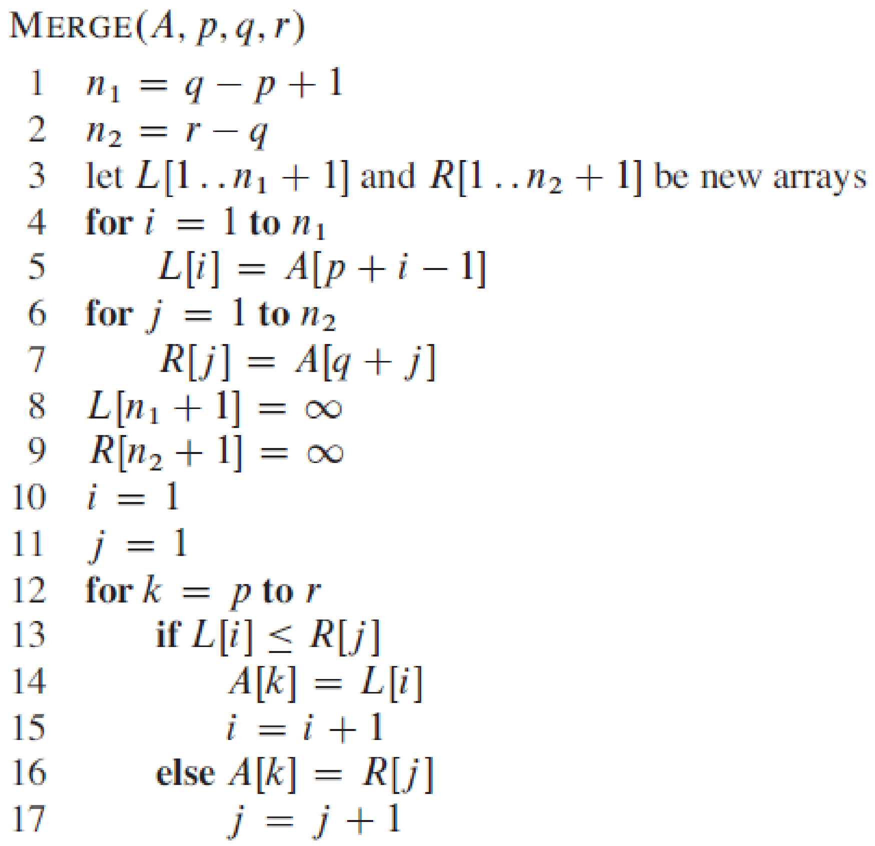
- Merge sort is a sorting algorithm that follows the divide and conquer approach.
- The approach:

1. *Divide:* Trivial.
2. *Conquer:* Recursively sort 2 subarrays.
3. *Combine:* Linear-time merge.

Merge Sort Algorithm

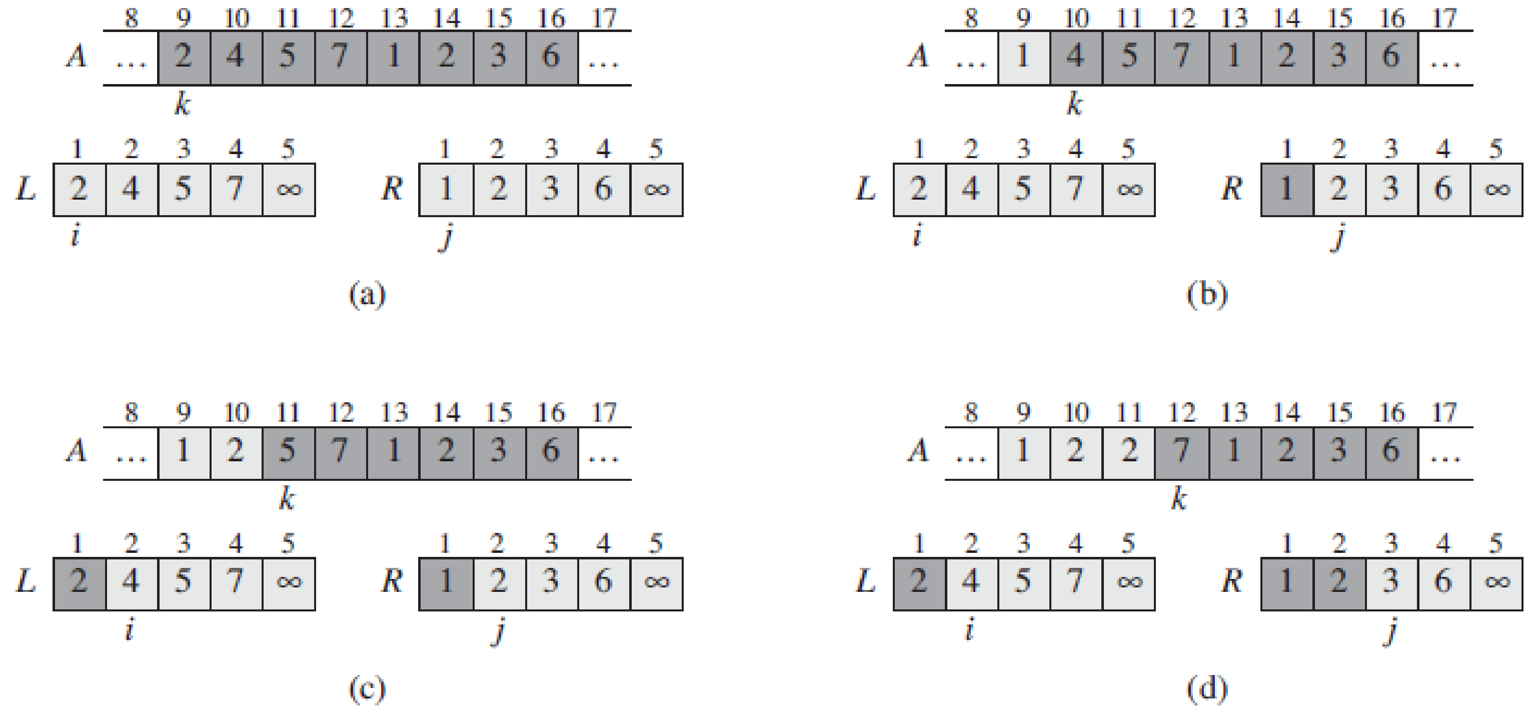


```
MERGE-SORT($A, p, r$)
if $p < r$
    $q = \lfloor (p + r)/2 \rfloor$
    MERGE-SORT($A, p, q$)
    MERGE-SORT($A, q + 1, r$)
    MERGE($A, p, q, r$)
```

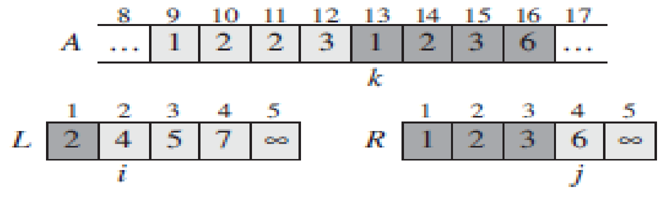


```
MERGE($A, p, q, r$)
$n_1 = q - p + 1$
$n_2 = r - q$
let $L[1 \,..\, n_1 + 1]$ and $R[1 \,..\, n_2 + 1]$ be new arrays
for $i = 1$ to $n_1$
    $L[i] = A[p + i - 1]$
for $j = 1$ to $n_2$
    $R[j] = A[q + j]$
$L[n_1 + 1] = \infty$
$R[n_2 + 1] = \infty$
$i = 1$
$j = 1$
for $k = p$ to $r$
    if $L[i] \leq R[j]$
        $A[k] = L[i]$
        $i = i + 1$
    else $A[k] = R[j]$
        $j = j + 1$
```

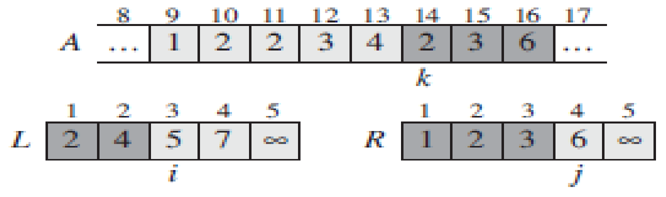
Merge operation



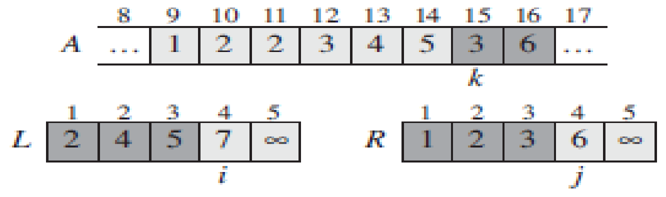
Merge operation Cont..



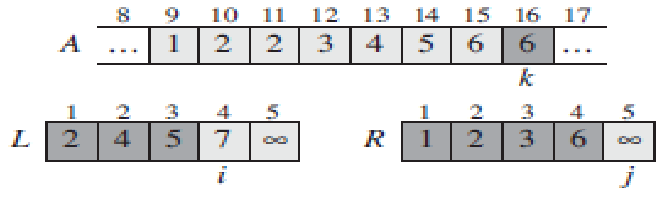
(e)



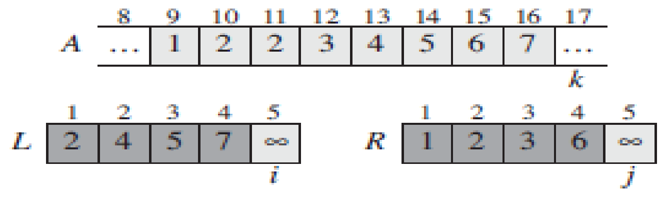
(f)



(g)

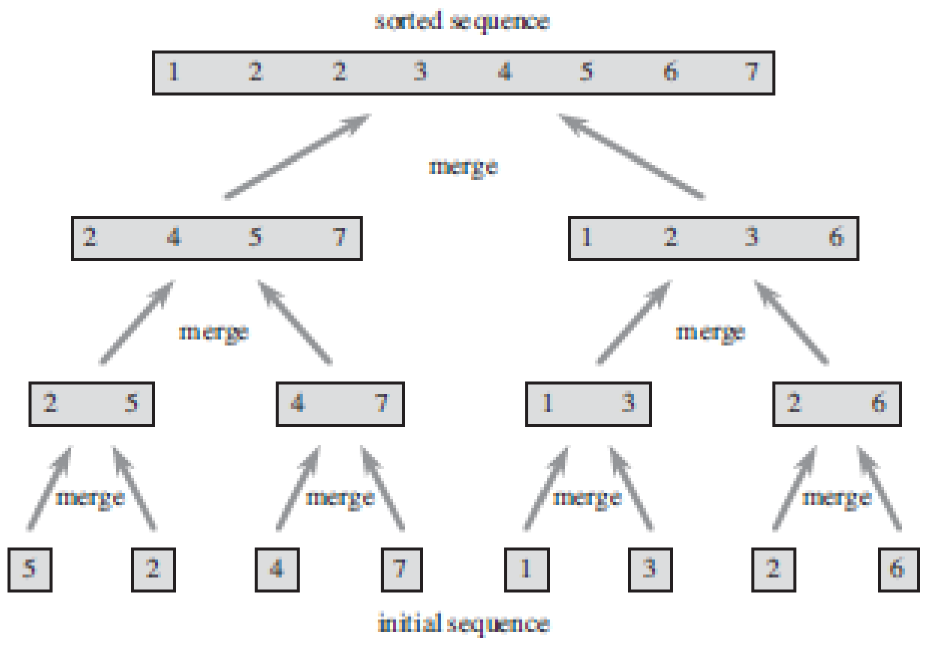


(h)



(i)

Merge Sort operation



Running Time Complexity – Merge Sort

1. *Divide:* Trivial.
2. *Conquer:* Recursively sort 2 subarrays.
3. *Combine:* Linear-time merge.

What is running time complexity of Merge Sort?

- Recurrence relation:

$$T(n) = 2\,T(n/2) + \Theta(n)$$

# subproblems

subproblem size

work dividing and combining

Running Time Complexity

1. *Divide:* Trivial.
2. *Conquer:* Recursively sort 2 subarrays.
3. *Combine:* Linear-time merge.

$$T(n) = \underbrace{2}_{\textit{\# subproblems}}\, T(\underbrace{n/2}_{\textit{subproblem size}}) + \underbrace{\Theta(n)}_{\textit{work dividing and combining}}$$

Using Master Theorem

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

Quicksort Algorithm

- Approach (Quicksort an *n*-element array):
  1. *Divide:* Partition the array into two subarrays around a *pivot x* such that *elements in lower subarray* $\leq x \leq$ *elements in upper subarray*.
  2. *Conquer:* Recursively sort the two subarrays.
  3. *Combine:* Trivial.

  - Key: *Linear-time partitioning subroutine.*

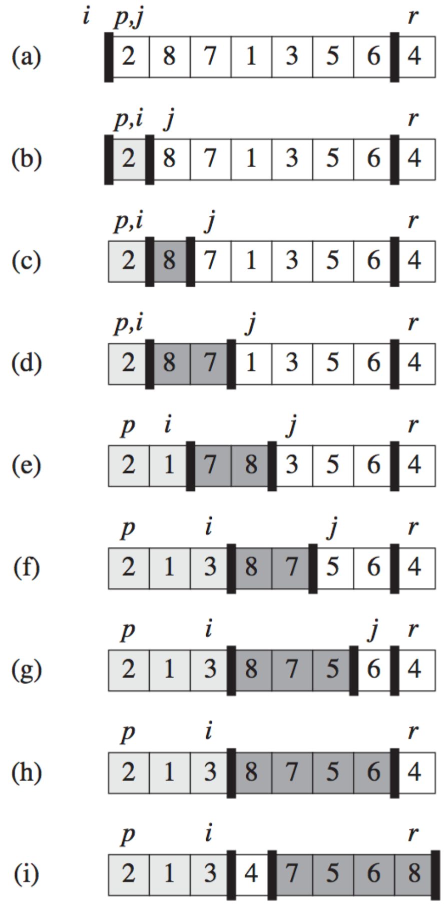
Quicksort Algorithm

```
QUICKSORT($A, p, r$)
if $p < r$
    $q$ = PARTITION($A, p, r$)
    QUICKSORT($A, p, q - 1$)
    QUICKSORT($A, q + 1, r$)
```

```
PARTITION($A, p, r$)
$x = A[r]$
$i = p - 1$
for $j = p$ to $r - 1$
    if $A[j] \leq x$
        $i = i + 1$
        exchange $A[i]$ with $A[j]$
exchange $A[i + 1]$ with $A[r]$
return $i + 1$
```

Quicksort Algorithm

- Array entry A[r] becomes the pivot element x.
- Lightly shaded array elements are all in the first partition with values no greater than x.
- Heavily shaded elements are in the second partition with values greater than x.
- The unshaded elements have not yet been put in one of the first two partitions, and the final white element is the pivot x.



(a) i p,j r 2 8 7 1 3 5 6 4
(b) p,i j r 2 8 7 1 3 5 6 4
(c) p,i j r 2 8 7 1 3 5 6 4
(d) p,i j r 2 8 7 1 3 5 6 4
(e) p i j r 2 1 7 8 3 5 6 4
(f) p i j r 2 1 3 8 7 5 6 4
(g) p i j r 2 1 3 8 7 5 6 4
(h) p i r 2 1 3 8 7 5 6 4
(i) p i r 2 1 3 4 7 5 6 8

The initial array and variable settings. None of the elements have been placed in either of the first two partitions.

The value 2 is “swapped with itself” and put in the partition of smaller values

The values 8 and 7 are added to the partition of larger values.

The values 1 and 8 are swapped, and the smaller partition grows.

The values 3 and 7 are swapped, and the smaller partition grows.

The larger partition grows to include 5 and 6, and the loop terminates.

In lines 7–8, the pivot element is swapped so that it lies between the two partitions.

Running Time Complexity

- Assume all input elements are distinct (else use the 3-way quicksort).
- In practice, there are better partitioning algorithms for when duplicate input elements may exist.

Running time: $T(n) = T(k) + T(n-k-1) + \Theta(n)$

Running time depends on the input array and the partition strategy.

When will the worst-case behaviour happen in Quicksort?

Worst Case of Quicksort

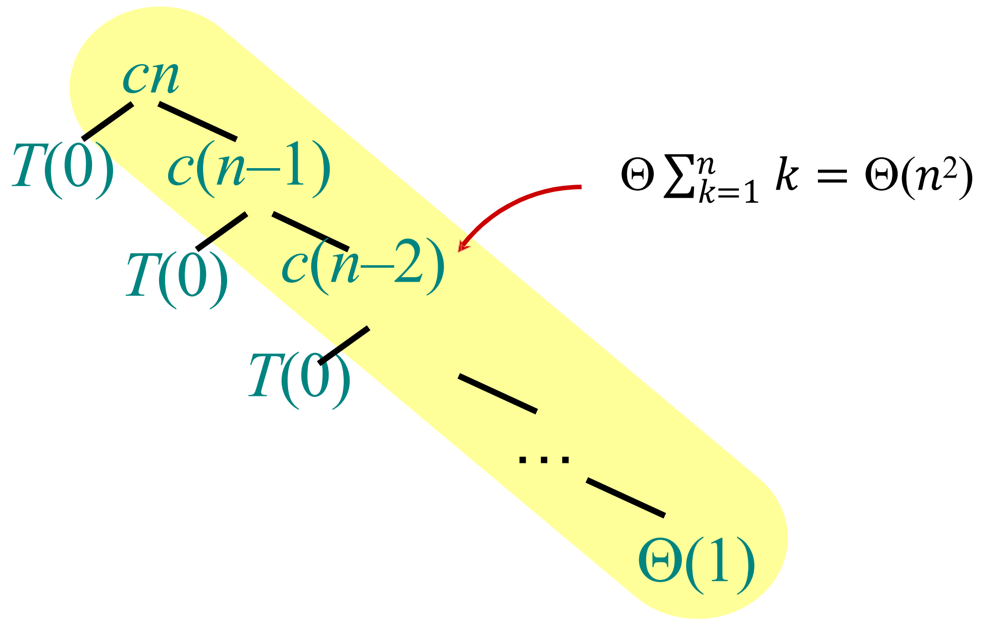
- Input sorted or reverse sorted.
- Partition around min or max element.
- One side of partition always has no elements.
- Using back-substitution method:

$$\begin{aligned} T(n) &= T(0) + T(n-1) + \Theta(n) \\ &= \Theta(1) + T(n-1) + \Theta(n) \\ &= T(n-1) + \Theta(n) \\ &= \Theta(n^2) \end{aligned}$$

(arithmetic series)

How will the Recursion Tree of Quicksort look like?

$$T(n) = T(0) + T(n-1) + cn$$



Best-case analysis

- To see how Quicksort can ensure $\Theta(n \log n)$ running time on any input, we need to understand what is the partition condition that guarantee this.
- If we're lucky, PARTITION splits the array evenly:

$$T(n) = 2T(n/2) + \Theta(n)$$

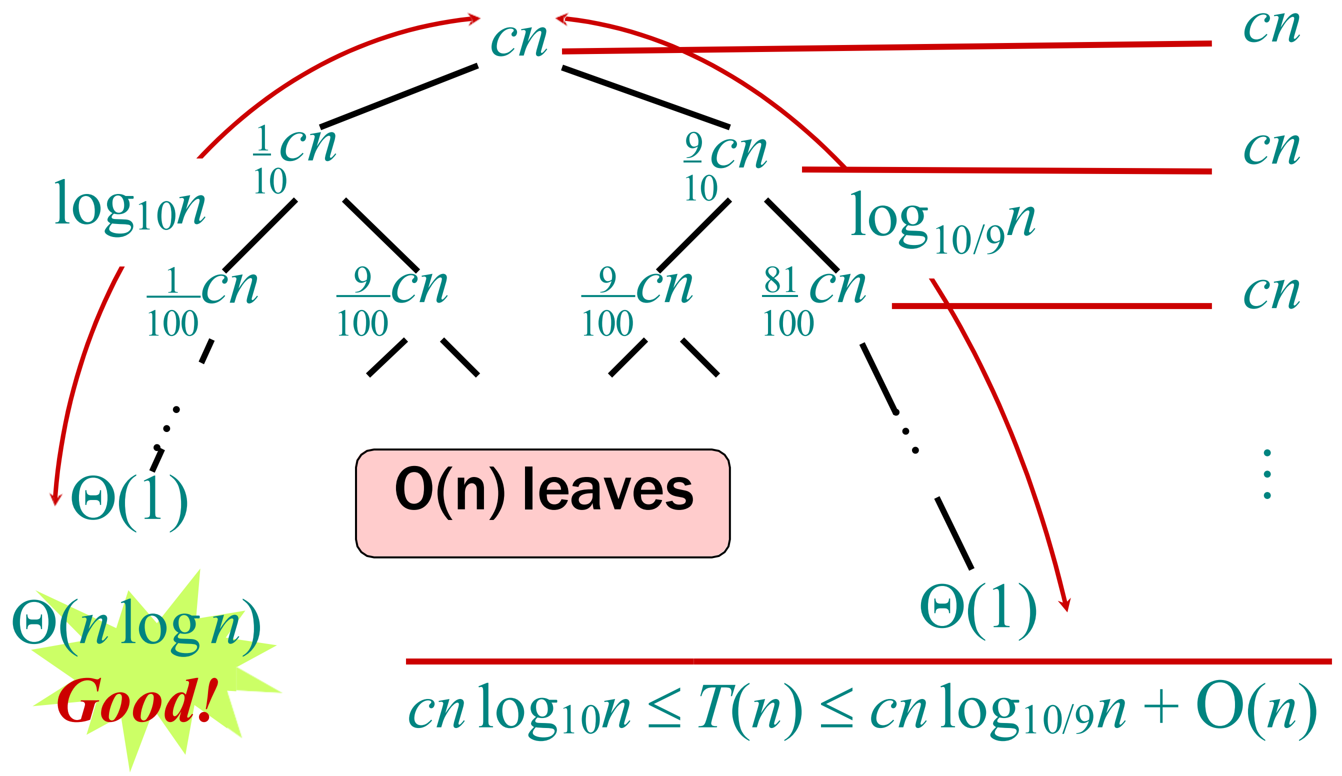
$$= \Theta(n \log n)$$ **(same as merge sort)**

- But what if the split is always $\frac{1}{10} : \frac{9}{10}$ Are we still going to get $\Theta(n \log n)$ running time? Or we are reaching $\Theta(n^2)$?

$$T(n) = T(\frac{1}{10}n) + T(\frac{9}{10}n) + \Theta(n)$$

What is the solution to this recurrence?

How will the Recursion Tree of Quicksort with $\frac{1}{10} : \frac{9}{10}$ partition look like?



$$cn \log_{10} n \le T(n) \le cn \log_{10/9} n + O(n)$$

More Intuition

- Here, we can further see, how Quicksort can still perform in $\Theta(n \log n)$.
- Suppose we have alternate Good, Not Good,.... partition each time:

$$G(n) = 2N(n/2) + \Theta(n)$$ **Good**

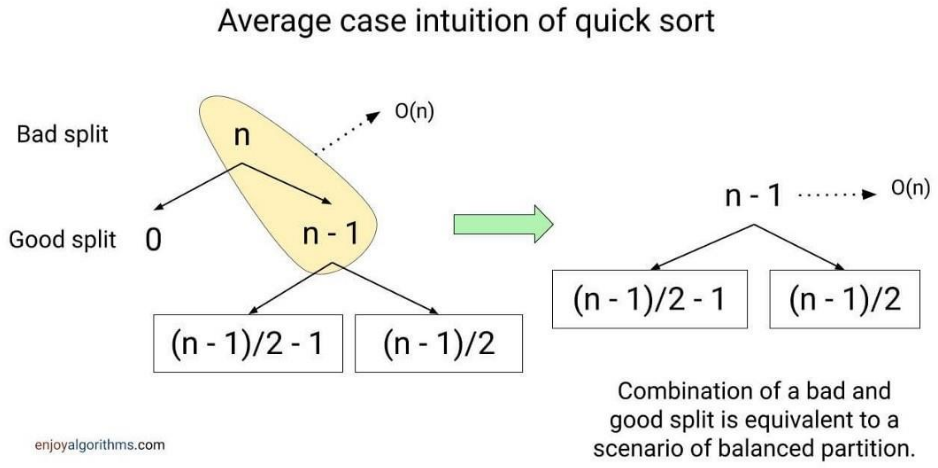
$$N(n) = G(n-1) + \Theta(n)$$ **Not Good**

- **Solving:**

$$\begin{aligned} G(n) &= 2(G(n/2 - 1) + \Theta(n/2)) + \Theta(n) \\ &= 2G(n/2 - 1) + \Theta(n) \\ &= \Theta(n \log n) \end{aligned}$$

Good!

How can we make sure we are usually having a good partition?


Average case intuition of quick sort
Bad split
Good split
n
O(n)
0
n - 1
(n - 1)/2 - 1
(n - 1)/2
n - 1
O(n)
(n - 1)/2 - 1
(n - 1)/2
Combination of a bad and good split is equivalent to a scenario of balanced partition.
enjoyalgorithms.com

Randomized Quicksort

- To make sure that Quicksort will always have a lucky O(n log n) running time:
  - IDEA: Partition around a random element.
    - Running time is independent of the input order.
    - No assumptions need to be made about the input distribution.
    - No specific input elicits the worst-case behaviour.
    - The worst case is determined only by the output of a random-number generator.

What is the classification of Quicksort?

Additional common problem solve using divide and conquer approach

Binary Search Algorithm

- Find an element in a sorted array:

  1. *Divide:* Check middle element.
  2. *Conquer:* Recursively search 1 subarray.
  3. *Combine:* Trivial.

Binary search

- Find an element in a sorted array:

1. ***Divide:*** **Check middle element.**
2. ***Conquer:*** **Recursively search 1 subarray.**
3. ***Combine:*** **Trivial.**

- ***Example:*** Find 9 in the following array A

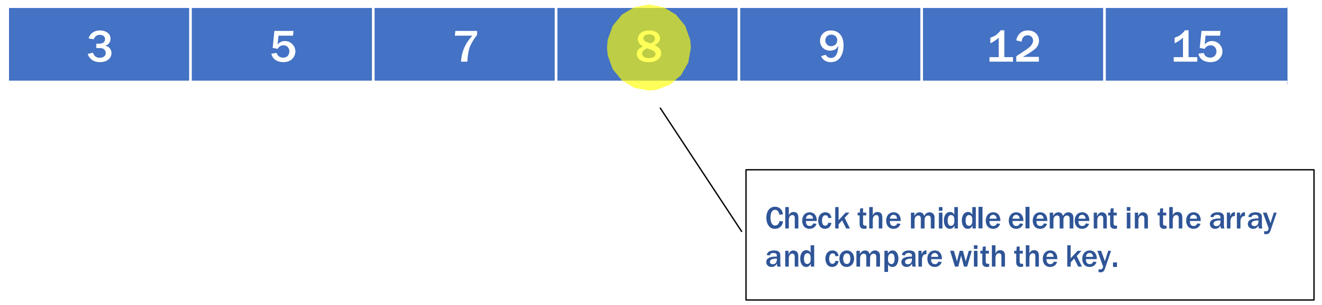
3	5	7	8	9	12	15

Binary search

- Find an element in a sorted array:

  1. *Divide:* **Check middle element.**
  2. *Conquer:* **Recursively search 1 subarray.**
  3. *Combine:* **Trivial.**

- ***Example:*** Find 9 in the following array A

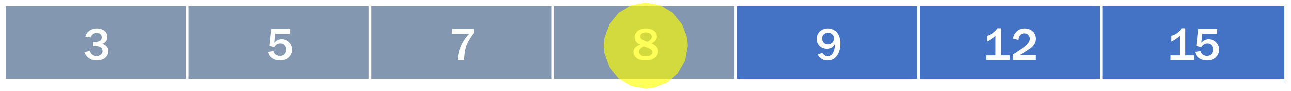

3
5
7
8
9
12
15
Check the middle element in the array and compare with the key.

Binary search

- Find an element in a sorted array:

  1. ***Divide:*** Check middle element.
  2. ***Conquer:*** Recursively search 1 subarray.
  3. ***Combine:*** Trivial.

- ***Example:*** Find 9 in the following array A



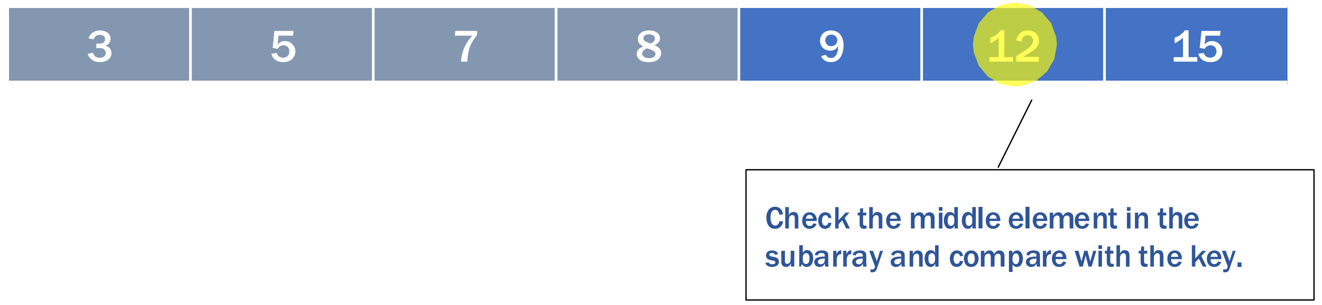
In this case, key is bigger than the middle element. Therefore, we can eliminate the lower subarray and repeat Step 1, which is check the middle element.

Binary search

- Find an element in a sorted array:

  1. *Divide:* **Check middle element.**
  2. *Conquer:* **Recursively search 1 subarray.**
  3. *Combine:* **Trivial.**

- ***Example:*** Find 9 in the following array A



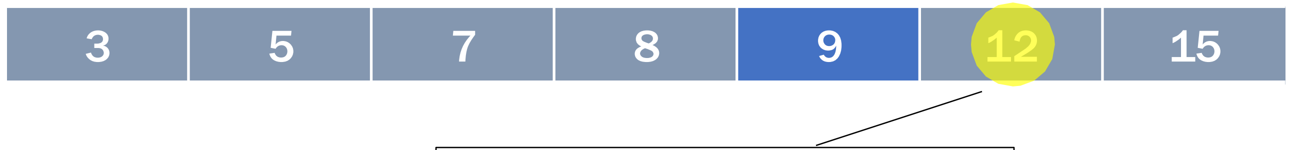
3
5
7
8
9
12
15
Check the middle element in the subarray and compare with the key.

Binary search

- Find an element in a sorted array:

  1. ***Divide:*** **Check middle element.**
  2. ***Conquer:*** **Recursively search 1 subarray.**
  3. ***Combine:*** **Trivial.**

- ***Example:*** Find 9 in the following array A



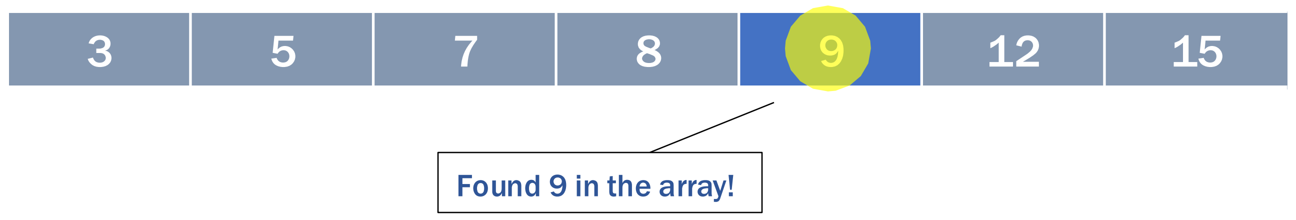
In this case, key is smaller than the middle element. Therefore, we can eliminate the higher subarray and repeat Step 1, which is check the middle element.

Binary search

- Find an element in a sorted array:

  1. ***Divide:*** **Check middle element.**
  2. ***Conquer:*** **Recursively search 1 subarray.**
  3. ***Combine:*** **Trivial.**

- ***Example:*** Find 9 in the following array A



Running time complexity

$$T(n) = \underbrace{1}_{\textit{\# subproblems}}\, T(\underbrace{n/2}_{\textit{subproblem size}}) + \underbrace{\Theta(1)}_{\textit{work dividing and combining}}$$

Using Masters Theorem

T(n) = Θ(lg n)

Powering a number

Problem: Compute a^n, where $n \in N$.

Naive algorithm: $\Theta(n)$.

Powering a number

Problem: Compute a^n, where $n \in N$.

Naive algorithm: $\Theta(n)$.

Divide-and-conquer algorithm:

$$a^n = \begin{cases} a^{n/2} \cdot a^{n/2} & \text{if } n \text{ is even;} \\ a^{(n-1)/2} \cdot a^{(n-1)/2} \cdot a & \text{if } n \text{ is odd.} \end{cases}$$

$$T(n) = T(n/2) + \Theta(1) \;\Rightarrow\; T(n) = \Theta(\lg n).$$

Fibonacci numbers

Recursive definition:

$$F_n = \begin{cases} 1 & \text{if } n = 0; \\ 2 & \text{if } n = 1; \\ F_{n-1} + F_{n-2} & \text{if } n \geq 2. \end{cases}$$

0 1 1 2 3 5 8 13 21 34 L

Computing Fibonacci numbers

Bottom-up:

- Compute $F_0, F_1, F_2, \ldots, F_n$ in order, forming each number by summing the two previous.
- Running time: $\Theta(n)$.

Naive recursive squaring:

$F_n = \phi^n / \sqrt{5}$ rounded to the nearest integer.

- Recursive squaring: $\Theta(\lg n)$ time.
- This method is unreliable, since floating-point arithmetic is prone to round-off errors.

Recursive squaring

Theorem: $$\begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n .$$

Algorithm: Recursive squaring.
Time = $\Theta(\lg n)$.

Proof of theorem. (Induction on n.)

Base ($n = 1$): $$\begin{bmatrix} F_2 & F_1 \\ F_1 & F_0 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^1 .$$

Recursive squaring

Inductive step ($n \geq 2$):

$$\begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} = \begin{bmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} \cdot \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n} \qquad \blacksquare$$

Matrix Multiplication

Suppose that we partition each of A, B, and C into four n/2 x n/2 matrices

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}, \tag{4.9}$$

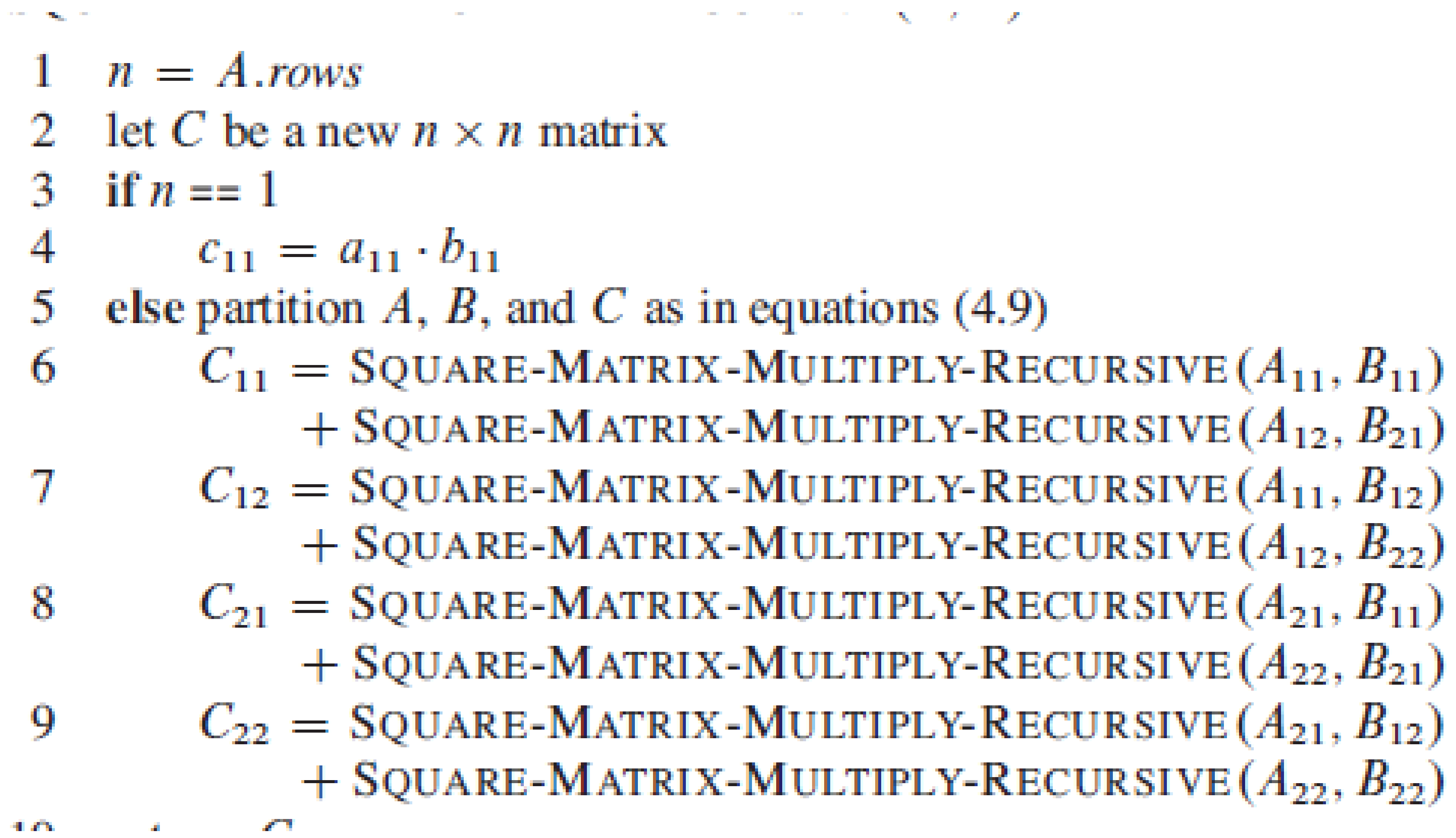
so that we rewrite the equation C = A.B as

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}. \tag{4.10}$$

Equation (4.10) corresponds to the four equations

$$C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}, \tag{4.11}$$
$$C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22}, \tag{4.12}$$
$$C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21}, \tag{4.13}$$
$$C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22}. \tag{4.14}$$

Matrices simple algorithm



```
$n = A.rows$
let $C$ be a new $n \times n$ matrix
if $n == 1$
    $c_{11} = a_{11} \cdot b_{11}$
else partition $A$, $B$, and $C$ as in equations (4.9)
    $C_{11} =$ SQUARE-MATRIX-MULTIPLY-RECURSIVE$(A_{11}, B_{11})$
            $+$ SQUARE-MATRIX-MULTIPLY-RECURSIVE$(A_{12}, B_{21})$
    $C_{12} =$ SQUARE-MATRIX-MULTIPLY-RECURSIVE$(A_{11}, B_{12})$
            $+$ SQUARE-MATRIX-MULTIPLY-RECURSIVE$(A_{12}, B_{22})$
    $C_{21} =$ SQUARE-MATRIX-MULTIPLY-RECURSIVE$(A_{21}, B_{11})$
            $+$ SQUARE-MATRIX-MULTIPLY-RECURSIVE$(A_{22}, B_{21})$
    $C_{22} =$ SQUARE-MATRIX-MULTIPLY-RECURSIVE$(A_{21}, B_{12})$
            $+$ SQUARE-MATRIX-MULTIPLY-RECURSIVE$(A_{22}, B_{22})$
```

Running time

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1\,, \\ 8T(n/2) + \Theta(n^2) & \text{if } n > 1\,. \end{cases} \qquad (4.17)$$

From master methods:

$$T(n) = \Theta(n^3).$$

Reference

- MIT open courseware, Introduction to Algorithms, 2005.
- Cormen, Lieserson and Rivest, Introduction to Algorithms, Third Edition, MIT Press, 2009.

We are also going to look at Heapsort today.