CE100 Algorithms and Programming II

Week-3 (Matrix Multiplication/ Quick Sort)

Spring Semester, 2021-2022

Download DOC-PDF, DOC-DOCX, SLIDE, PPTX

Matrix Multiplication / Quick Sort

Outline (1)

- Matrix Multiplication
  - Traditional
  - Recursive
  - Strassen

Outline (2)

- Quicksort
  - Hoare Partitioning
  - Lomuto Partitioning
  - Recursive Sorting

Outline (3)

- Quicksort Analysis
  - Randomized Quicksort
  - Randomized Selection
    - Recursive
    - Medians

Matrix Multiplication (1)

- **Input:** $A = [a_{ij}], B = [b_{ij}]$
- **Output:** $C = [c_{ij}] = A \cdot B \Longrightarrow i, j = 1, 2, 3, \ldots, n$

$$\begin{bmatrix} c_{11} & c_{12} & \ldots & c_{1n} \\ c_{21} & c_{22} & \ldots & c_{2n} \\ \vdots & \vdots & \vdots & \ddots \\ c_{n1} & c_{n2} & \ldots & c_{nn} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \vdots & \vdots & \vdots & \ddots \\ a_{n1} & a_{n2} & \ldots & a_{nn} \end{bmatrix} \cdot \begin{bmatrix} b_{11} & b_{12} & \ldots & b_{1n} \\ b_{21} & b_{22} & \ldots & b_{2n} \\ \vdots & \vdots & \vdots & \ddots \\ b_{n1} & a_{n2} & \ldots & b_{nn} \end{bmatrix}$$

Matrix Multiplication (2)

$$\begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{pmatrix}$$

- $c_{ij} = \sum_{1 \le k \le n} a_{ik}.b_{kj}$

Matrix Multiplication: Standard Algorithm

Running Time: $\Theta(n^3)$

```
for i=1 to n do
    for j=1 to n do
        C[i,j] = 0
        for k=1 to n do
            C[i,j] = C[i,j] + A[i,k] + B[k,j]
        endfor
    endfor
endfor
```

Matrix Multiplication: Divide & Conquer (1)

IDEA: Divide the nxn matrix into $2x2$ matrix of $(n/2)x(n/2)$ submatrices.

$$\begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} \qquad \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$

$$c_{11} = a_{11}b_{11} + a_{12}b_{21} \qquad c_{21} = a_{21}b_{11} + a_{22}b_{21}$$

$$\begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} \qquad \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$

$$c_{12} = a_{11}b_{12} + a_{12}b_{22} \qquad c_{22} = a_{21}b_{12} + a_{22}b_{22}$$

Matrix Multiplication: Divide & Conquer (2)

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \cdot \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

$$\text{8 mults and 4 adds of (n/2)*(n/2) submatrices} = \begin{cases} c_{11} = a_{11}b_{11} + a_{12}b_{21} \\ c_{21} = a_{21}b_{11} + a_{22}b_{21} \\ c_{12} = a_{11}b_{12} + a_{12}b_{22} \\ c_{22} = a_{21}b_{12} + a_{22}b_{22} \end{cases}$$

Matrix Multiplication: Divide & Conquer (3)

```
MATRIX-MULTIPLY(A, B)
    // Assuming that both A and B are nxn matrices
    if n == 1 then
        return A * B
    else
        //partition A, B, and C as shown before
        C[1,1] = MATRIX-MULTIPLY (A[1,1], B[1,1]) +
                 MATRIX-MULTIPLY (A[1,2], B[2,1]);

        C[1,2] = MATRIX-MULTIPLY (A[1,1], B[1,2]) +
                 MATRIX-MULTIPLY (A[1,2], B[2,2]);

        C[2,1] = MATRIX-MULTIPLY (A[2,1], B[1,1]) +
        MATRIX-MULTIPLY (A[2,2], B[2,1]);

        C[2,2] = MATRIX-MULTIPLY (A[2,1], B[1,2]) +
        MATRIX-MULTIPLY (A[2,2], B[2,2]);
    endif

    return C
```

Matrix Multiplication: Divide & Conquer Analysis

$$T(n) = 8T(n/2) + \Theta(n^2)$$

- 8 recursive calls $\Longrightarrow 8T(\cdots)$
- each problem has size $n/2 \Longrightarrow \cdots T(n/2)$
- Submatrix addition $\Longrightarrow \Theta(n^2)$

Matrix Multiplication: Solving the Recurrence

- $T(n) = 8T(n/2) + \Theta(n^2)$
  - $a = 8, b = 2$
  - $f(n) = \Theta(n^2)$
  - $n^{log_b^a} = n^3$
- Case 1: $\frac{n^{log_b^a}}{f(n)} = \Omega(n^\varepsilon) \Longrightarrow T(n) = \Theta(n^{log_b^a})$

Similar with ordinary (iterative) algorithm.

Matrix Multiplication: Strassen's Idea (1)

Compute $c_{11}, c_{12}, c_{21}, c_{22}$ using 7 recursive multiplications.

In normal case we need 8 as below.

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \cdot \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

$$8 \text{ mults and } 4 \text{ adds of } (\mathrm{n}/2)^*(\mathrm{n}/2) \text{ submatrices} = \begin{cases} c_{11} = a_{11}b_{11} + a_{12}b_{21} \\ c_{21} = a_{21}b_{11} + a_{22}b_{21} \\ c_{12} = a_{11}b_{12} + a_{12}b_{22} \\ c_{22} = a_{21}b_{12} + a_{22}b_{22} \end{cases}$$

Matrix Multiplication: Strassen's Idea (2)

- **Reminder:**
  - Each submatrix is of size $(n/2) * (n/2)$
  - Each add/sub operation takes $\Theta(n^2)$ time
- Compute $P1 \ldots P7$ using 7 recursive calls to matrix-multiply

$$P_1 = a_{11} * (b_{12} - b_{22})$$
$$P_2 = (a_{11} + a_{12}) * b_{22}$$
$$P_3 = (a_{21} + a_{22}) * b_{11}$$
$$P_4 = a_{22} * (b_{21} - b_{11})$$
$$P_5 = (a_{11} + a_{22}) * (b_{11} + b_{22})$$
$$P_6 = (a_{12} - a_{22}) * (b_{21} + b_{22})$$
$$P_7 = (a_{11} - a_{21}) * (b_{11} + b_{12})$$

Matrix Multiplication: Strassen's Idea (3)

$$P_1 = a_{11} * (b_{12} - b_{22})$$
$$P_2 = (a_{11} + a_{12}) * b_{22}$$
$$P_3 = (a_{21} + a_{22}) * b_{11}$$
$$P_4 = a_{22} * (b_{21} - b_{11})$$
$$P_5 = (a_{11} + a_{22}) * (b_{11} + b_{22})$$
$$P_6 = (a_{12} - a_{22}) * (b_{21} + b_{22})$$
$$P_7 = (a_{11} - a_{21}) * (b_{11} + b_{12})$$

- How to compute c_{ij} using $P1 \dots P7$?

$$c_{11} = P_5 + P_4 - P_2 + P_6$$
$$c_{12} = P_1 + P_2$$
$$c_{21} = P_3 + P_4$$
$$c_{22} = P_5 + P_1 - P_3 - P_7$$

Matrix Multiplication: Strassen's Idea (4)

- 7 recursive multiply calls
- 18 add/sub operations

Matrix Multiplication: Strassen's Idea (5)

e.g. Show that $c_{12} = P_1 + P_2$:

$$
\begin{aligned}
c_{12} &= P_1 + P_2 \\
&= a_{11}(b_{12}-b_{22}) + (a_{11} + a_{12})b_{22} \\
&= a_{11}b_{12} - a_{11}b_{22} + a_{11}b_{22} + a_{12}b_{22} \\
&= a_{11}b_{12} + a_{12}b_{22}
\end{aligned}
$$

Strassen's Algorithm

- **Divide:** Partition A and B into $(n/2) * (n/2)$ submatrices. Form terms to be multiplied using $+$ and $-$.
- **Conquer:** Perform 7 multiplications of $(n/2) * (n/2)$ submatrices recursively.
- **Combine:** Form C using $+$ and $-$ on $(n/2) * (n/2)$submatrices.

Recurrence: $T(n) = 7T(n/2) + \Theta(n^2)$

Strassen's Algorithm: Solving the Recurrence (1)

- $T(n) = 7T(n/2) + \Theta(n^2)$
  - $a = 7, b = 2$
  - $f(n) = \Theta(n^2)$
  - $n^{log_b^a} = n^{lg7}$
- Case 1: $\frac{n^{log_b^a}}{f(n)} = \Omega(n^{\varepsilon}) \Longrightarrow T(n) = \Theta(n^{log_b^a})$

$T(n) = \Theta(n^{log_2^7})$

$2^3 = 8, 2^2 = 4$ so $\Longrightarrow log_2^7 \approx 2.81$

or use https://www.omnicalculator.com/math/log

Strassen's Algorithm: Solving the Recurrence (2)

- The number 2.81 may not seem much smaller than 3
- But, it is significant because the difference is in the exponent.
- Strassen's algorithm beats the ordinary algorithm on today's machines for $n \geq 30$ or so.
- Best to date: $\Theta(n^{2.376\cdots})$ (of theoretical interest only)

Matrix Multiplication Solution Faster Than Strassen's Algorithm

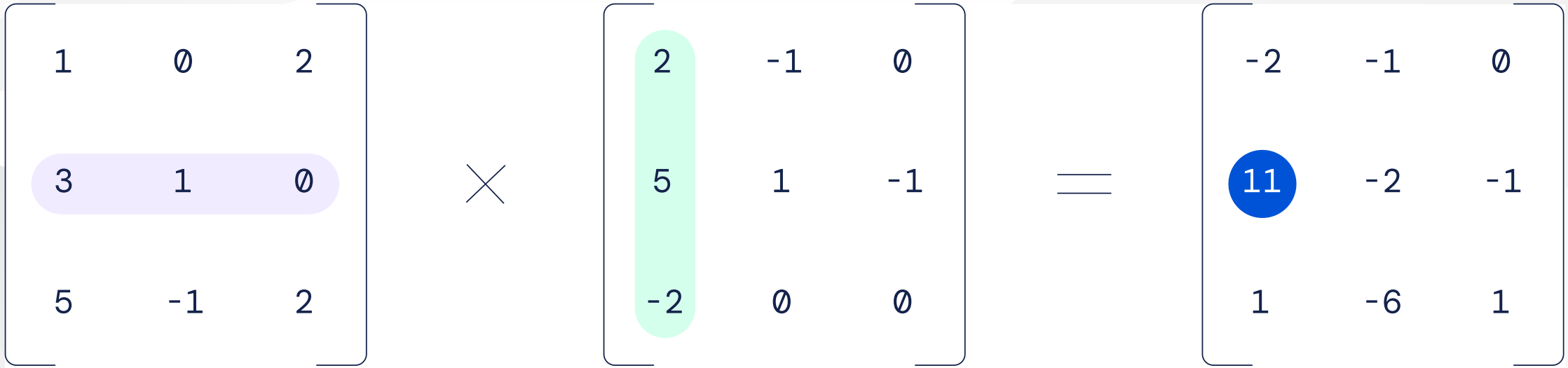
- In 5 Oct. 2022 new paper published
  - Discovering faster matrix multiplication algorithms with reinforcement learning | Nature
  - GitHub - deepmind/alphatensor
  - Article
  - Discovering novel algorithms with AlphaTensor

Matrix Multiplication Solution Faster Than Strassen's Algorithm

For example, if the traditional algorithm taught in school multiplies a 4x5 by 5x5 matrix using 100 multiplications, and this number was reduced to 80 with human ingenuity, AlphaTensor has found algorithms that do the same operation using just 76 multiplications.

Matrix Multiplication Solution Faster Than Strassen's Algorithm

Standard Multiplication



Standard and Strassen Comparison

$$\begin{bmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{bmatrix} \times \begin{bmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \end{bmatrix} = \begin{bmatrix} c_{1,1} & c_{1,2} \\ c_{2,1} & c_{2,2} \end{bmatrix}$$

Standard algorithm	Strassen's algorithm
$h_1 = a_{1,1}\ b_{1,1}$	$h_1 = (a_{1,1} + a_{2,2})\ (b_{1,1} + b_{2,2})$
$h_2 = a_{1,1}\ b_{1,2}$	$h_2 = (a_{2,1} + a_{2,2})\ b_{1,1}$
$h_3 = a_{1,2}\ b_{2,1}$	$h_3 = a_{1,1}\ (b_{1,2} - b_{2,2})$
$h_4 = a_{1,2}\ b_{2,2}$	$h_4 = a_{2,2}\ (-b_{1,1} + b_{2,1})$
$h_5 = a_{2,1}\ b_{1,1}$	$h_5 = (a_{1,1} + a_{1,2})\ b_{2,2}$
$h_6 = a_{2,1}\ b_{1,2}$	$h_6 = (-a_{1,1} + a_{2,1})\ (b_{1,1} + b_{1,2})$
$h_7 = a_{2,2}\ b_{2,1}$	$h_7 = (a_{1,2} - a_{2,2})(b_{2,1} + b_{2,2})$
$h_8 = a_{2,2}\ b_{2,2}$	
$c_{1,1} = h_1 + h_3$	$c_{1,1} = h_1 + h_4 - h_5 + h_7$
$c_{1,2} = h_2 + h_4$	$c_{1,2} = h_3 + h_5$
$c_{2,1} = h_5 + h_7$	$c_{2,1} = h_2 + h_4$
$c_{2,2} = h_6 + h_8$	$c_{2,2} = h_1 - h_2 + h_3 + h_6$

Improved Solution

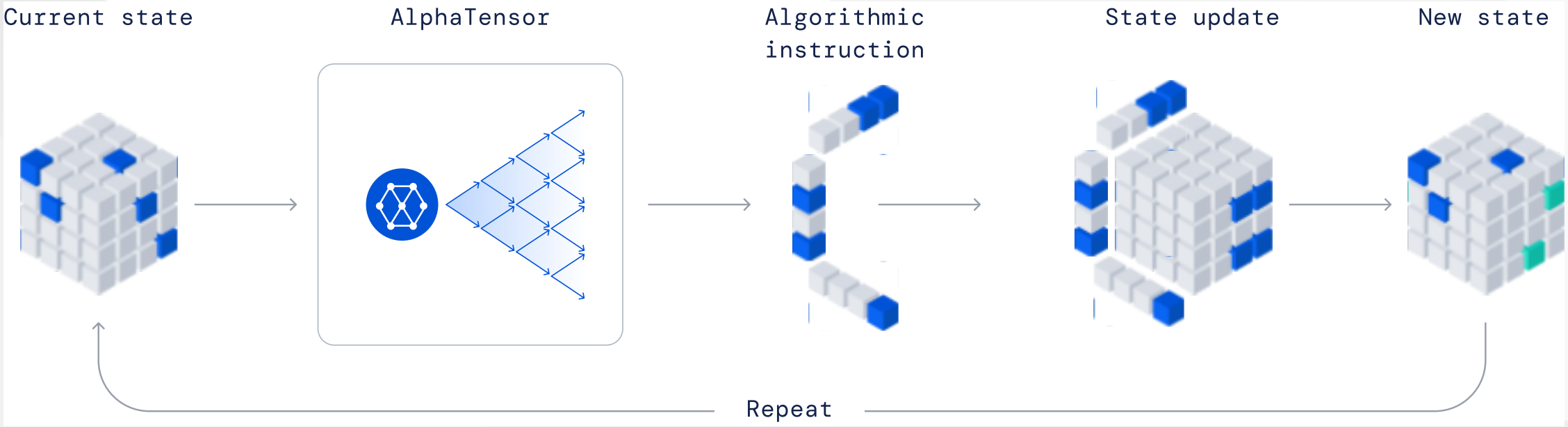
$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} & a_{1,5} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} & a_{2,5} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} & a_{3,5} \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} & a_{4,5} \end{bmatrix} \times \begin{bmatrix} b_{1,1} & b_{1,2} & b_{1,3} & b_{1,4} & b_{1,5} \\ b_{2,1} & b_{2,2} & b_{2,3} & b_{2,4} & b_{2,5} \\ b_{3,1} & b_{3,2} & b_{3,3} & b_{3,4} & b_{3,5} \\ b_{4,1} & b_{4,2} & b_{4,3} & b_{4,4} & b_{4,5} \\ b_{5,1} & b_{5,2} & b_{5,3} & b_{5,4} & b_{5,5} \end{bmatrix} = \begin{bmatrix} c_{1,1} & c_{1,2} & c_{1,3} & c_{1,4} & c_{1,5} \\ c_{2,1} & c_{2,2} & c_{2,3} & c_{2,4} & c_{2,5} \\ c_{3,1} & c_{3,2} & c_{3,3} & c_{3,4} & c_{3,5} \\ c_{4,1} & c_{4,2} & c_{4,3} & c_{4,4} & c_{4,5} \end{bmatrix}$$

$$\begin{aligned}
h_1 &= a_{3,2}(-b_{2,1} - b_{2,5} - b_{3,1}) \\
h_2 &= (a_{2,2} + a_{2,5} - a_{3,5})(-b_{2,5} - b_{5,1}) \\
h_3 &= (-a_{3,1} - a_{4,1} + a_{4,2})(-b_{1,1} + b_{2,5}) \\
h_4 &= (a_{1,2} + a_{1,4} + a_{3,4})(-b_{2,5} - b_{4,1}) \\
h_5 &= (a_{1,5} + a_{2,2} + a_{2,5})(-b_{2,4} + b_{5,1}) \\
h_6 &= (-a_{2,2} - a_{2,5} - a_{4,5})(b_{2,3} + b_{5,1}) \\
h_7 &= (-a_{1,1} + a_{4,1} - a_{4,2})(b_{1,1} + b_{2,4}) \\
h_8 &= (a_{3,2} - a_{3,3} - a_{4,3})(-b_{2,3} + b_{5,1}) \\
h_9 &= (-a_{1,2} - a_{1,4} + a_{4,4})(b_{2,3} + b_{4,1}) \\
h_{10} &= (a_{2,2} + a_{2,5})b_{5,1} \\
h_{11} &= (-a_{2,1} - a_{4,1} + a_{4,2})(-b_{1,1} + b_{2,2}) \\
h_{12} &= (a_{4,1} - a_{4,2})b_{1,1} \\
h_{13} &= (a_{1,2} + a_{1,4} + a_{2,4})(b_{2,2} + b_{4,1}) \\
h_{14} &= (a_{1,3} - a_{3,2} + a_{3,3})(b_{2,4} + b_{3,1}) \\
h_{15} &= (-a_{1,2} - a_{1,4})b_{4,1} \\
h_{16} &= (-a_{3,2} + a_{3,3})b_{3,1} \\
h_{17} &= (a_{1,2} + a_{1,4} - a_{2,1} + a_{2,2} - a_{2,3} + a_{2,4} - a_{3,2} + a_{3,3} - a_{4,1} + a_{4,2})b_{2,2} \\
h_{18} &= a_{2,1}(b_{1,1} + b_{1,2} + b_{5,2}) \\
h_{19} &= -a_{2,3}(b_{3,1} + b_{3,2} + b_{5,2}) \\
h_{20} &= (-a_{1,5} + a_{2,1} + a_{2,3} - a_{2,5})(-b_{1,1} - b_{1,2} + b_{1,4} - b_{5,2}) \\
h_{21} &= (a_{2,1} + a_{2,3} - a_{2,5})b_{5,2} \\
h_{22} &= (a_{1,3} - a_{1,4} - a_{2,4})(b_{1,1} + b_{1,2} - b_{1,4} - b_{3,1} - b_{3,2} + b_{3,4} + b_{4,4}) \\
h_{23} &= a_{1,3}(-b_{3,1} + b_{3,4} + b_{4,4}) \\
h_{24} &= a_{1,5}(-b_{4,4} - b_{5,1} + b_{5,4}) \\
h_{25} &= -a_{1,1}(b_{1,1} - b_{1,4}) \\
h_{26} &= (-a_{1,3} + a_{1,4} + a_{1,5})b_{4,4} \\
h_{27} &= (a_{1,3} - a_{3,1} + a_{3,2})(b_{1,1} - b_{1,4} + b_{1,5} + b_{3,5}) \\
h_{28} &= -a_{3,4}(-b_{1,5} - b_{4,1} - b_{4,5}) \\
h_{29} &= a_{3,1}(b_{1,3} + b_{1,5} + b_{3,5}) \\
h_{30} &= (a_{3,1} - a_{3,3} + a_{3,4})b_{3,5} \\
h_{31} &= (-a_{1,4} - a_{1,5} - a_{3,4})(-b_{4,4} - b_{5,1} + b_{5,4} - b_{5,5}) \\
h_{32} &= (a_{2,1} + a_{4,1} + a_{4,4})(b_{1,3} - b_{4,1} - b_{4,2} - b_{4,3}) \\
h_{33} &= a_{4,3}(-b_{3,1} - b_{3,3}) \\
h_{34} &= a_{4,4}(-b_{1,3} + b_{4,1} + b_{4,3}) \\
h_{35} &= -a_{4,5}(b_{1,3} + b_{5,1} + b_{5,3}) \\
h_{36} &= (a_{2,3} - a_{2,5} - a_{4,5})(b_{3,1} + b_{3,2} + b_{3,3} + b_{5,2}) \\
h_{37} &= (-a_{4,1} - a_{4,4} + a_{4,5})b_{1,3} \\
h_{38} &= (-a_{2,3} - a_{3,1} + a_{3,3} - a_{3,4})(b_{3,5} + b_{4,1} + b_{4,2} + b_{4,5}) \\
h_{39} &= (-a_{3,1} - a_{4,1} - a_{4,4} + a_{4,5})(b_{1,3} + b_{5,1} + b_{5,3} + b_{5,5}) \\
h_{40} &= (-a_{1,3} + a_{1,4} + a_{1,5} - a_{4,4})(-b_{3,1} - b_{3,3} + b_{3,4} + b_{4,4}) \\
h_{41} &= (-a_{1,1} + a_{4,1} - a_{4,5})(b_{1,3} + b_{3,1} + b_{3,3} - b_{5,4} + b_{5,1} + b_{5,3} - b_{5,4}) \\
h_{42} &= (-a_{2,1} + a_{2,5} - a_{3,5})(-b_{1,1} - b_{1,2} - b_{1,5} + b_{4,1} + b_{4,2} + b_{4,5} - b_{5,2}) \\
h_{43} &= a_{2,4}(b_{4,1} + b_{4,2}) \\
h_{44} &= (a_{2,3} + a_{3,2} - a_{3,3})(b_{2,2} - b_{3,1}) \\
h_{45} &= (-a_{3,3} + a_{3,4} - a_{4,3})(b_{3,5} + b_{4,1} + b_{4,3} + b_{4,5} + b_{5,1} + b_{5,3} + b_{5,5}) \\
h_{46} &= -a_{3,5}(-b_{5,1} - b_{5,3}) \\
h_{47} &= (a_{2,1} - a_{2,5} - a_{3,1} + a_{3,5})(b_{1,1} + b_{1,2} + b_{1,5} - b_{4,1} - b_{4,2} - b_{4,5}) \\
h_{48} &= (-a_{2,3} + a_{3,3})(b_{2,2} + b_{3,2} + b_{3,5} + b_{4,1} + b_{4,2} + b_{4,5}) \\
h_{49} &= (-a_{1,1} - a_{1,3} + a_{1,4} + a_{1,5} - a_{2,1} - a_{2,3} + a_{2,4} + a_{2,5})(-b_{1,1} - b_{1,2} + b_{1,4}) \\
h_{50} &= (-a_{1,4} - a_{2,4})(b_{2,2} - b_{5,1} - b_{5,2} + b_{5,4} - b_{4,2} + b_{4,4})
\end{aligned}$$

$$\begin{aligned}
h_{51} &= a_{2,2}(b_{2,1} + b_{2,2} - b_{5,1}) \\
h_{52} &= a_{4,2}(b_{1,1} + b_{2,1} + b_{2,3}) \\
h_{53} &= -a_{1,2}(-b_{2,1} + b_{2,4} + b_{4,1}) \\
h_{54} &= (a_{1,2} + a_{1,4} - a_{2,2} - a_{2,5} - a_{3,2} + a_{3,3} - a_{4,2} + a_{4,3} - a_{4,4} - a_{4,5})b_{2,3} \\
h_{55} &= (a_{1,4} - a_{4,4})(-b_{2,3} + b_{5,1} + b_{1,3} - b_{1,4} + b_{4,3} - b_{4,4}) \\
h_{56} &= (a_{1,1} - a_{1,5} - a_{4,1} + a_{4,5})(b_{3,1} + b_{3,3} - b_{3,4} + b_{5,1} + b_{5,3} - b_{5,4}) \\
h_{57} &= (-a_{3,1} - a_{4,1})(-b_{1,3} - b_{1,5} - b_{2,5} - b_{5,1} - b_{5,3} - b_{5,5}) \\
h_{58} &= (-a_{1,4} - a_{1,5} - a_{3,4} - a_{3,5})(-b_{5,1} + b_{5,4} - b_{5,5}) \\
h_{59} &= (-a_{3,3} + a_{3,4} - a_{4,3} + a_{4,4})(b_{4,1} + b_{4,3} + b_{4,5} + b_{5,1} + b_{5,3} + b_{5,5}) \\
h_{60} &= (a_{2,5} + a_{4,5})(b_{2,3} - b_{3,1} - b_{3,2} - b_{3,3} - b_{5,2} - b_{5,3}) \\
h_{61} &= (a_{1,4} + a_{3,4})(b_{1,1} - b_{1,4} + b_{1,5} - b_{2,5} - b_{4,4} + b_{4,5} - b_{5,1} + b_{5,4} - b_{5,5}) \\
h_{62} &= (a_{2,1} + a_{4,1})(b_{1,2} + b_{1,3} + b_{2,2} - b_{4,1} - b_{4,2} - b_{4,3}) \\
h_{63} &= (-a_{3,3} - a_{4,3})(-b_{2,3} - b_{3,3} - b_{3,5} - b_{4,1} - b_{4,3} - b_{4,5}) \\
h_{64} &= (a_{1,1} - a_{1,3} - a_{1,4} + a_{3,1} - a_{3,3} - a_{3,4})(b_{1,1} - b_{1,4} + b_{1,5}) \\
h_{65} &= (-a_{3,1} + a_{4,1})(-b_{1,3} + b_{1,4} + b_{2,4} - b_{5,1} - b_{5,3} + b_{5,4}) \\
h_{66} &= (a_{1,1} - a_{1,2} + a_{1,3} - a_{1,5} - a_{2,2} - a_{2,5} - a_{3,2} + a_{3,3} - a_{4,1} + a_{4,2})b_{2,4} \\
h_{67} &= (a_{2,5} - a_{3,5})(b_{1,1} + b_{1,2} + b_{1,5} - b_{2,5} - b_{4,1} - b_{4,2} - b_{4,5} + b_{5,2} + b_{5,5}) \\
h_{68} &= (a_{1,1} + a_{1,3} - a_{1,4} - a_{1,5} - a_{4,1} - a_{4,3} + a_{4,4} + a_{4,5})(-b_{3,1} - b_{3,3} + b_{3,4}) \\
h_{69} &= (-a_{1,3} + a_{1,4} - a_{2,3} + a_{2,4})(-b_{2,4} - b_{3,1} - b_{3,2} + b_{3,4} - b_{5,2} + b_{5,4}) \\
h_{70} &= (a_{2,3} - a_{2,5} + a_{4,3} - a_{4,5})(-b_{3,1} - b_{3,2} - b_{3,3}) \\
h_{71} &= (-a_{3,1} + a_{3,3} - a_{3,4} + a_{3,5} - a_{4,1} + a_{4,3} - a_{4,4} + a_{4,5})(-b_{5,1} - b_{5,3} - b_{5,5}) \\
h_{72} &= (-a_{2,1} - a_{2,4} - a_{4,1} - a_{4,4})(b_{4,1} + b_{4,2} + b_{4,3}) \\
h_{73} &= (a_{1,3} - a_{1,4} - a_{1,5} + a_{2,3} - a_{2,4} - a_{2,5})(b_{1,1} + b_{1,2} - b_{1,4} + b_{2,4} + b_{5,2} - b_{5,4}) \\
h_{74} &= (a_{2,1} - a_{2,3} + a_{2,4} - a_{3,1} + a_{3,3} - a_{3,4})(b_{4,1} + b_{4,2} + b_{4,5}) \\
h_{75} &= -(a_{1,2} + a_{1,4} - a_{2,2} - a_{2,5} - a_{3,1} + a_{3,2} + a_{3,4} + a_{3,5} - a_{4,1} + a_{4,2})b_{2,5} \\
h_{76} &= (a_{1,3} + a_{3,3})(-b_{1,1} + b_{1,4} - b_{1,5} + b_{2,4} + b_{3,4} - b_{3,5}) \\
c_{1,1} &= -h_{10} + h_{12} + h_{14} - h_{15} - h_{16} + h_{63} + h_5 - h_{66} - h_7 \\
c_{2,1} &= h_{10} + h_{11} - h_{12} + h_{13} + h_{15} + h_{16} - h_{17} - h_{44} + h_{51} \\
c_{3,1} &= h_{10} - h_{12} + h_{15} + h_{16} - h_1 + h_2 + h_5 - h_4 + h_{75} \\
c_{4,1} &= -h_{10} + h_{12} - h_{15} - h_{16} + h_{52} + h_{64} - h_6 - h_8 + h_9 \\
c_{1,2} &= h_{13} + h_{15} + h_{20} + h_{21} - h_{22} + h_{23} + h_{25} - h_{43} + h_{49} + h_{50} \\
c_{2,2} &= -h_{11} + h_{12} - h_{13} - h_{15} - h_{16} + h_{17} + h_{18} - h_{19} - h_{21} + h_{43} + h_{44} \\
c_{3,2} &= -h_{16} - h_{19} - h_{21} - h_{28} - h_{29} - h_{38} + h_{42} + h_{44} - h_{47} + h_{48} \\
c_{4,2} &= h_{11} - h_{12} - h_{18} + h_{21} - h_{32} + h_{33} - h_{34} - h_{36} + h_{62} - h_{70} \\
c_{1,3} &= h_{15} + h_{23} + h_{24} + h_{34} - h_{37} + h_{40} - h_{41} + h_{55} - h_{56} - h_9 \\
c_{2,3} &= -h_{10} + h_{19} + h_{32} + h_{35} + h_{36} + h_{37} - h_{43} - h_{60} - h_6 - h_{72} \\
c_{3,3} &= -h_{16} - h_{28} + h_{33} + h_{37} - h_{39} + h_{45} - h_{46} + h_{63} - h_{71} - h_8 \\
c_{4,3} &= h_{10} + h_{15} + h_{16} - h_{33} + h_{34} - h_{35} - h_{37} - h_{54} + h_6 + h_8 - h_9 \\
c_{1,4} &= -h_{10} + h_{12} + h_{14} - h_{16} + h_{23} + h_{24} + h_{25} + h_{26} + h_5 - h_{66} - h_7 \\
c_{2,4} &= h_{10} + h_{18} - h_{19} + h_{26} - h_{22} - h_{24} - h_{26} - h_5 - h_{69} + h_{73} \\
c_{3,4} &= -h_{14} + h_{16} - h_{23} - h_{26} + h_{27} + h_{29} + h_{31} + h_{46} - h_{58} + h_{76} \\
c_{4,4} &= h_{12} + h_{25} + h_{26} - h_{33} - h_{35} - h_{40} + h_{41} + h_{65} - h_{68} - h_7 \\
c_{1,5} &= h_{15} + h_{24} + h_{25} + h_{37} - h_{28} + h_{30} + h_{31} - h_4 + h_{61} + h_{64} \\
c_{2,5} &= -h_{10} - h_{18} - h_2 - h_{30} - h_{38} + h_{42} - h_{43} + h_{46} + h_{67} + h_{74} \\
c_{3,5} &= -h_{10} + h_{13} - h_{15} + h_{28} + h_{29} - h_2 - h_{30} - h_3 + h_{46} + h_4 - h_{75} \\
c_{4,5} &= -h_{12} - h_{29} + h_{30} - h_{34} + h_{35} + h_{39} + h_3 - h_{45} + h_{57} + h_{59}
\end{aligned}$$

How it's done

> "Single-player game played by AlphaTensor, where the goal is to find a correct matrix multiplication algorithm. The state of the game is a cubic array of numbers (shown as grey for 0, blue for 1, and green for -1), representing the remaining work to be done."



Maximum Subarray Problem

Input: An array of values

Output: The contiguous subarray that has the largest sum of elements

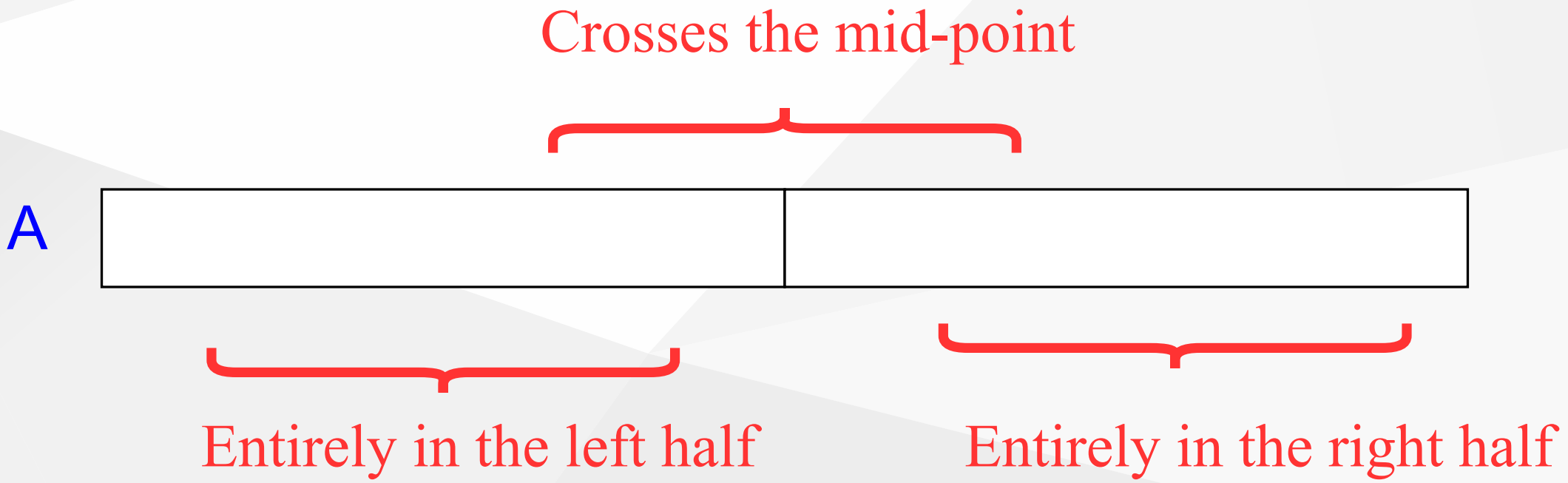
- Input array:

$$[13][-3][-25][20][-3][-16][-23] \overbrace{[18][20][-7][12]}^{\text{max. contiguous subarray}} [-22][-4][7]$$

Maximum Subarray Problem: Divide & Conquer (1)

- **Basic idea:**
  - **Divide** the input array into 2 from the middle
  - Pick the **best** solution among the following:
    - The max subarray of the **left half**
    - The max subarray of the **right half**
    - The max subarray **crossing the mid-point**

Maximum Subarray Problem: Divide & Conquer (2)



Maximum Subarray Problem: Divide & Conquer (3)

- **Divide:** Trivial (divide the array from the middle)
- **Conquer:** Recursively compute the max subarrays of the left and right halves
- **Combine:** Compute the max-subarray crossing the $mid - point$
  - (can be done in $\Theta(n)$ time).
  - Return the max among the following:
    - the max subarray of the left-subarray
    - the max subarray of the rightsubarray
    - the max subarray crossing the mid-point

TODO : detailed solution in textbook...

Conclusion : Divide & Conquer

- Divide and conquer is just one of several powerful techniques for algorithm design.
- Divide-and-conquer algorithms can be analyzed using recurrences and the master method (so practice this math).
- Can lead to more efficient algorithms

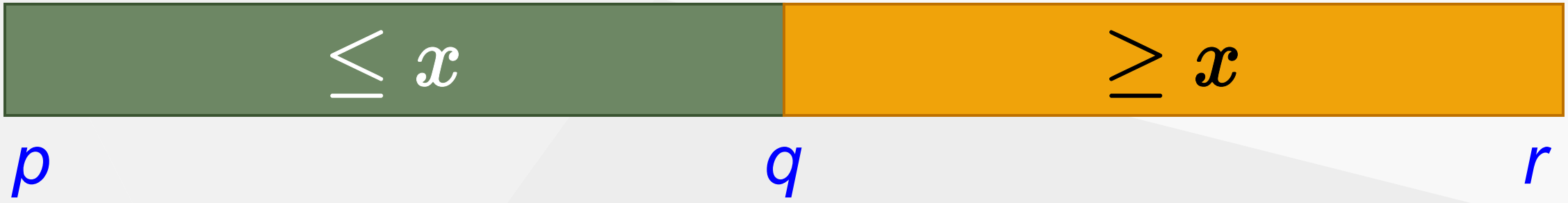
Quicksort (1)

- One of the most-used algorithms in practice
- Proposed by **C.A.R.** *Hoare* in 1962.
- Divide-and-conquer algorithm
- In-place algorithm
  - The additional space needed is O(1)
  - The sorted array is returned in the input array
  - *Reminder: Insertion-sort is also an in-place algorithm, but Merge-Sort is not in-place.*
- Very practical

Quicksort (2)

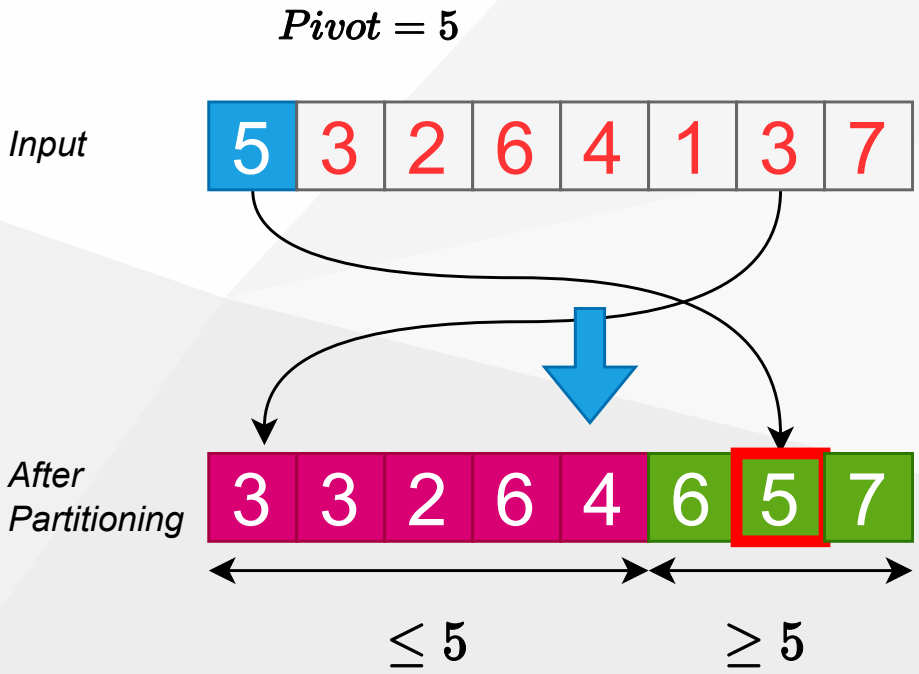
- **Divide:** Partition the array into 2 subarrays such that elements in the lower part $\leq$ elements in the higher part
- **Conquer:** Recursively sort 2 subarrays
- **Combine:** Trivial (because in-place)

Key: Linear-time $(\Theta(n))$ partitioning algorithm



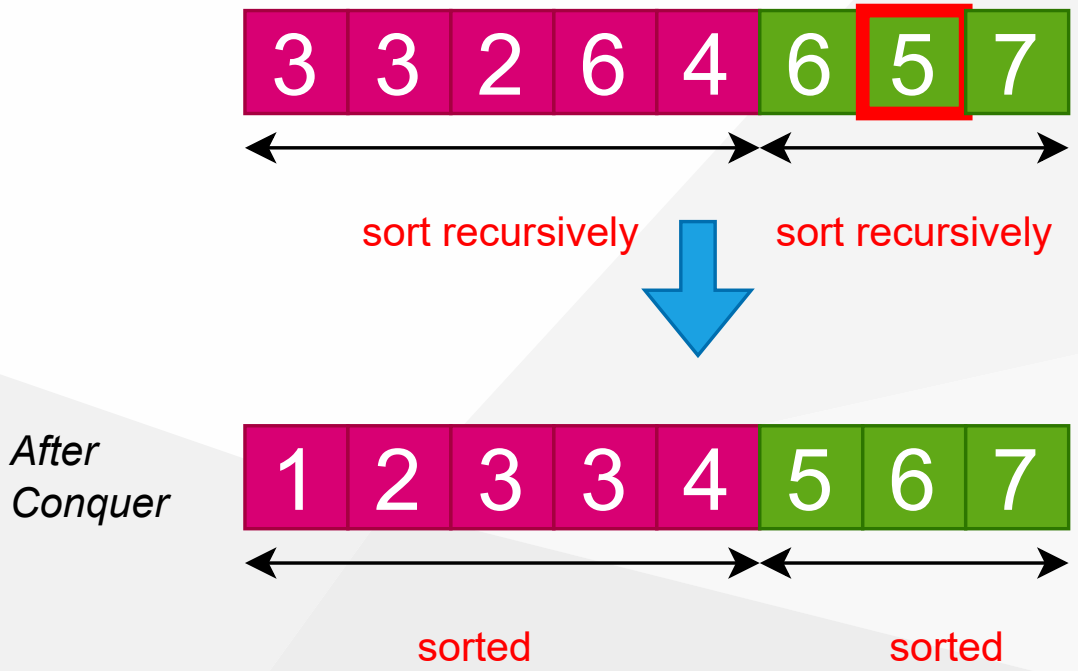
Divide: Partition the array around a pivot element

- Choose a pivot element x
- Rearrange the array such that:
  - Left subarray: All elements $\leq x$
  - Right subarray: All elements $\geq x$



Conquer: Recursively Sort the Subarrays

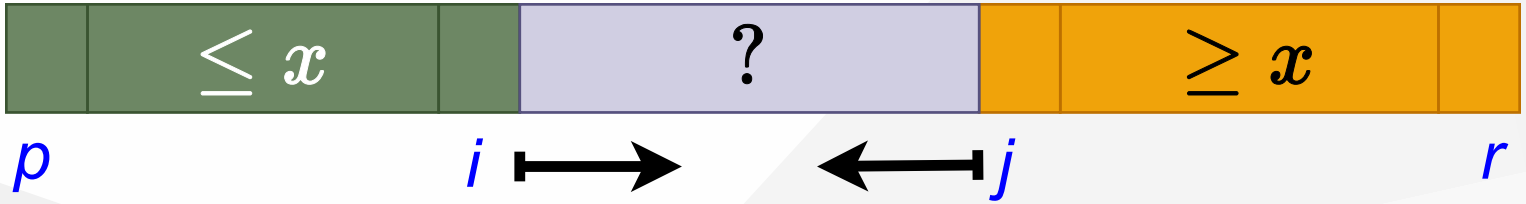
Note: Everything in the left subarray ≤ everything in the right subarray



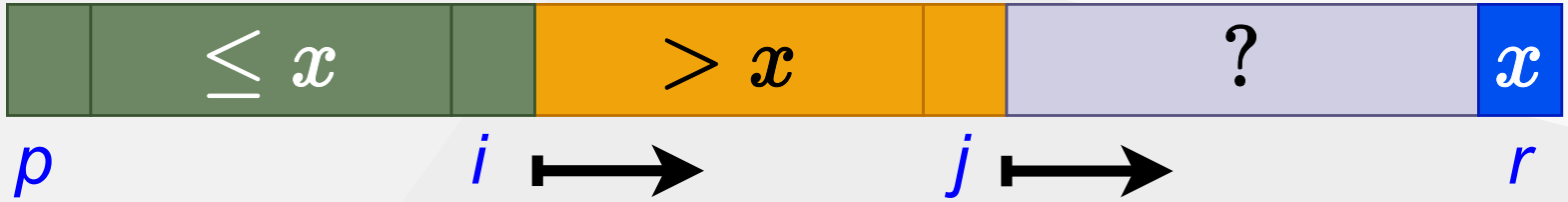
Note: Combine is trivial after conquer. Array already sorted.

Two partitioning algorithms

- **Hoare's algorithm:**
  Partitions around the first element of subarray
  - $(pivot = x = A[p])$

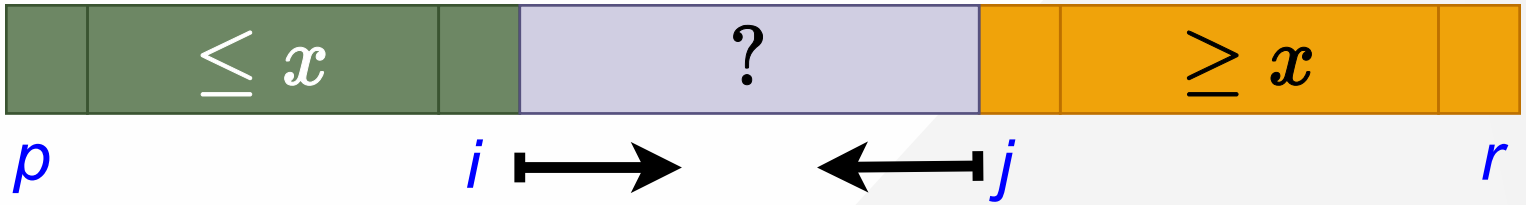


- **Lomuto's algorithm:**
  Partitions around the last element of subarray
  - $(pivot = x = A[r])$



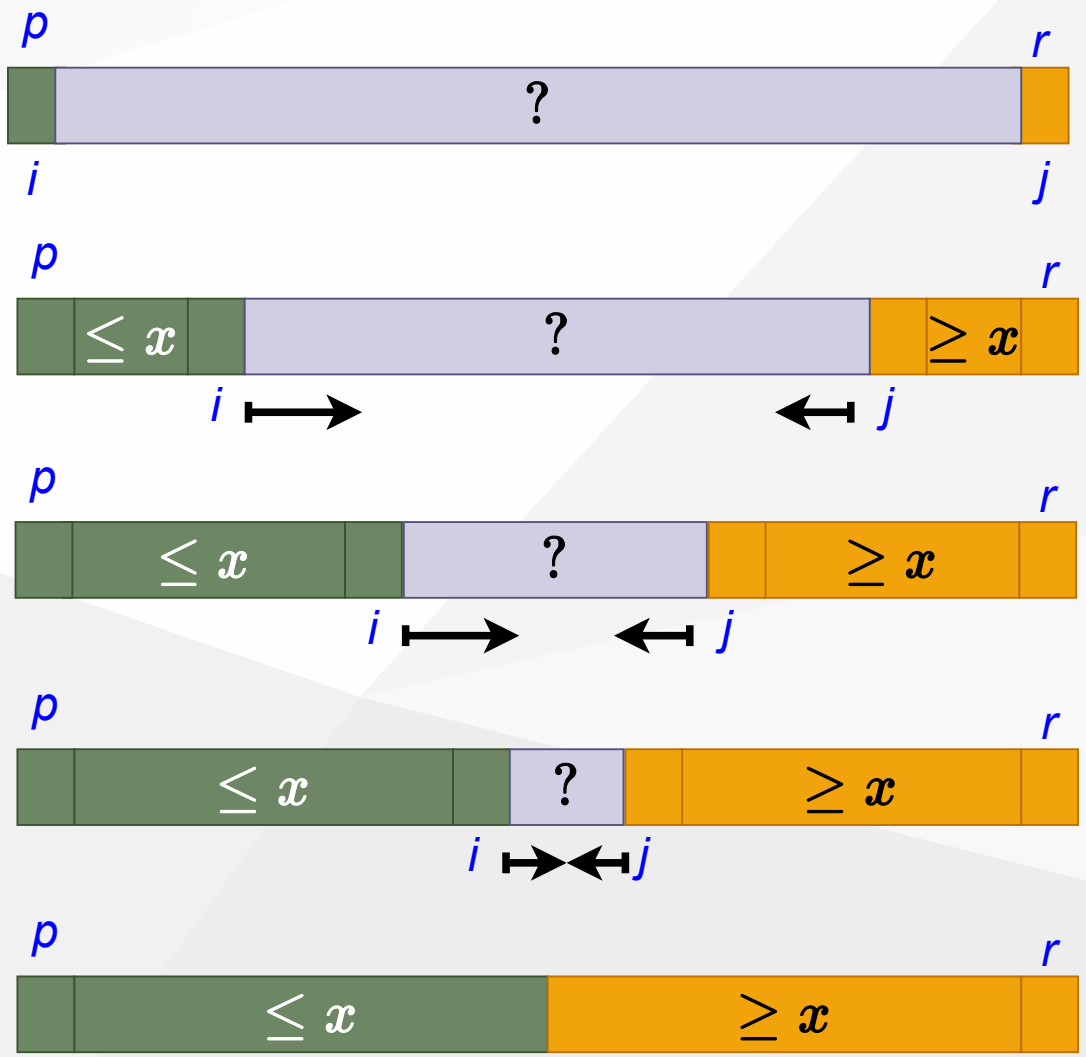
Hoare's Partitioning Algorithm (1)

- Choose a pivot element: $pivot = x = A[p]$



- Grow two regions:
  - from left to right: $A[p \ldots i]$
  - from right to left: $A[j \ldots r]$
    - such that:
  - every element in $A[p \ldots i] \leq$ pivot
  - every element in $A[p \ldots i] \geq$ pivot

Hoare's Partitioning Algorithm (2)

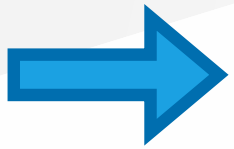


Hoare's Partitioning Algorithm (3)

- Elements are exchanged when
  - $A[i]$ is **too large** to belong to the **left** region
  - $A[j]$ is **too small** to belong to the **right** region
    - assuming that the inequality is strict
- The two regions $A[p \ldots i]$ and $A[j \ldots r]$ grow until $A[i] \geq pivot \geq A[j]$

```
H-PARTITION(A, p, r)
      pivot = A[p]
        i = p - 1
        j = r - 1
        while true do
          repeat j = j - 1 until A[j] <= pivot
          repeat i = i - 1 until A[i] <= pivot
        if i < j then
          exchange A[i] with A[j]
        else
          return j
```

Hoare's Partitioning Algorithm Example (Step-1)



H-PARTITION (A, *p*, *r*)

```
pivot ← A[p]
i ← p − 1
j ← r + 1
while true do
  repeat j ← j − 1 until A[j] ≤ pivot
  repeat i ← i + 1 until A[i] ≥ pivot
  if i < j then
        exchange A[i] ↔ A[j]
  else
        return j
```

$$Pivot = 5$$

Input

5	3	2	6	4	1	3	7

$$STEP - 1$$

Hoare's Partitioning Algorithm Example (Step-2)

H-PARTITION (A, *p*, *r*)

```
pivot ← A[p]
i ← p − 1
j ← r + 1
while true do
  repeat j ← j − 1 until A[j] ≤ pivot
  repeat i ← i + 1 until A[i] ≥ pivot
  if i < j then
        exchange A[i] ↔ A[j]
  else
        return j
```

$Pivot = 5$

Input	5	3	2	6	4	1	3	7

i j

$STEP - 2$

Hoare's Partitioning Algorithm Example (Step-3)

H-PARTITION (A, *p*, *r*)

```
pivot ← A[p]
i ← p − 1
j ← r + 1
while true do
  repeat j ← j − 1 until A[j] ≤ pivot
  repeat i ← i + 1 until A[i] ≥ pivot
  if i < j then
        exchange A[i] ↔ A[j]
  else
        return j
```

$Pivot = 5$

Input

5	3	2	6	4	1	3	7

i j

$STEP - 3$

Hoare's Partitioning Algorithm Example (Step-4)

H-PARTITION (A, *p*, *r*)

```
pivot ← A[p]
i ← p − 1
j ← r + 1
while true do
  repeat j ← j − 1 until A[j] ≤ pivot
  repeat i ← i + 1 until A[i] ≥ pivot
  if i < j then
        exchange A[i] ↔ A[j]
  else
        return j
```

$Pivot = 5$

$STEP - 4$

Hoare's Partitioning Algorithm Example (Step-5)

H-PARTITION (A, *p*, *r*)

```
pivot ← A[p]
i ← p − 1
j ← r + 1
while true do
  repeat j ← j − 1 until A[j] ≤ pivot
  repeat i ← i + 1 until A[i] ≥ pivot
  if i < j then
        exchange A[i] ↔ A[j]
  else
        return j
```

$Pivot = 5$

Input

5	3	2	6	4	1	3	7

i j

$STEP - 5$

Hoare's Partitioning Algorithm Example (Step-6)

H-PARTITION (A, *p*, *r*)

```
pivot ← A[p]
i ← p − 1
j ← r + 1
while true do
  repeat j ← j − 1 until A[j] ≤ pivot
  repeat i ← i + 1 until A[i] ≥ pivot
  if i < j then
        exchange A[i] ↔ A[j]
  else
        return j
```

$Pivot = 5$

$STEP - 6$

Hoare's Partitioning Algorithm Example (Step-7)

H-PARTITION (A, *p*, *r*)

```
pivot ← A[p]
i ← p – 1
j ← r + 1
while true do
  repeat j ← j – 1 until A[j] ≤ pivot
  repeat i ← i + 1 until A[i] ≥ pivot
  if i < j then
        exchange A[i] ↔ A[j]
  else
        return j
```

$Pivot = 5$

$STEP - 7$

Hoare's Partitioning Algorithm Example (Step-8)

H-PARTITION (A, *p*, *r*)

```
pivot ← A[p]
i ← p − 1
j ← r + 1
while true do
  repeat j ← j − 1 until A[j] ≤ pivot
  repeat i ← i + 1 until A[i] ≥ pivot
  if i < j then
        exchange A[i] ↔ A[j]
  else
        return j
```

$Pivot = 5$

Input

3	3	2	6	4	1	5	7

i j

$STEP - 8$

Hoare's Partitioning Algorithm Example (Step-9)

H-PARTITION (A, *p*, *r*)

```
pivot ← A[p]
i ← p − 1
j ← r + 1
while true do
  repeat j ← j − 1 until A[j] ≤ pivot
  repeat i ← i + 1 until A[i] ≥ pivot
  if i < j then
        exchange A[i] ↔ A[j]
  else
        return j
```

$Pivot = 5$

$STEP - 9$

Hoare's Partitioning Algorithm Example (Step-10)

H-PARTITION (A, *p*, *r*)

pivot ← A[*p*]
i ← *p* – 1
j ← *r* + 1
while true do
  repeat *j* ← *j* – 1 **until** A[*j*] ≤ *pivot*
  repeat *i* ← *i* + 1 **until** A[*i*] ≥ *pivot*
  if *i* < *j* then
    exchange A[*i*] ↔ A[*j*]
  else
    return *j*

$Pivot = 5$

Input

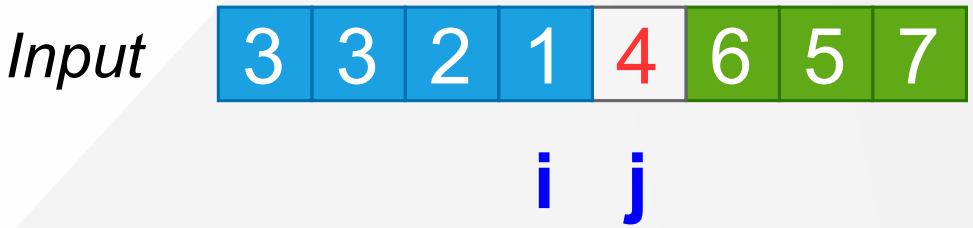
$STEP - 10$

Hoare's Partitioning Algorithm Example (Step-11)

H-PARTITION (A, *p*, *r*)

```
pivot ← A[p]
i ← p − 1
j ← r + 1
while true do
  repeat j ← j − 1 until A[j] ≤ pivot
  repeat i ← i + 1 until A[i] ≥ pivot
  if i < j then
        exchange A[i] ↔ A[j]
  else
        return j
```

$Pivot = 5$



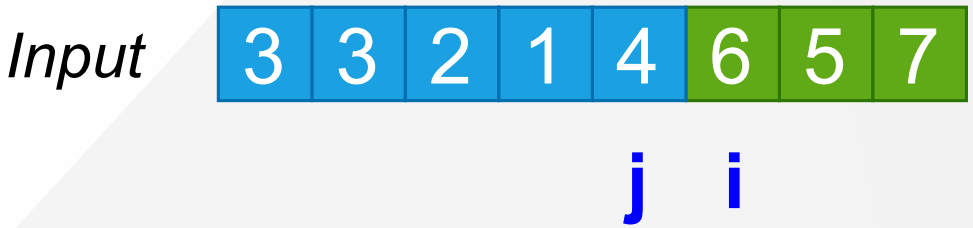
$STEP - 11$

Hoare's Partitioning Algorithm Example (Step-12)

H-PARTITION (A, *p*, *r*)

```
pivot ← A[p]
i ← p − 1
j ← r + 1
while true do
  repeat j ← j − 1 until A[j] ≤ pivot
  repeat i ← i + 1 until A[i] ≥ pivot
  if i < j then
        exchange A[i] ↔ A[j]
  else
        return j
```

$Pivot = 5$



$STEP - 12$

Hoare's Partitioning Algorithm - Notes

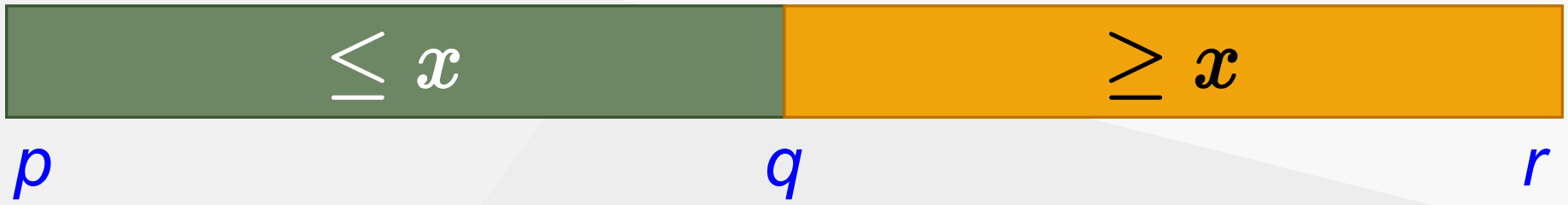
- Elements are exchanged when
  - $A[i]$ is **too large** to belong to the **left** region
  - $A[j]$ is **too small** to belong to the **right** region
    - assuming that the inequality is strict
- The two regions $A[p \ldots i]$ and $A[j \ldots r]$ grow until $A[i] \geq pivot \geq A[j]$
- The asymptotic runtime of Hoare's partitioning algorithm $\Theta(n)$

```
H-PARTITION(A, p, r)
      pivot = A[p]
        i = p - 1
        j = r - 1
        while true do
          repeat j = j - 1 until A[j] <= pivot
          repeat i = i - 1 until A[i] <= pivot
        if i < j then exchange A[i] with A[j]
        else return j
```

Quicksort with Hoare's Partitioning Algorithm

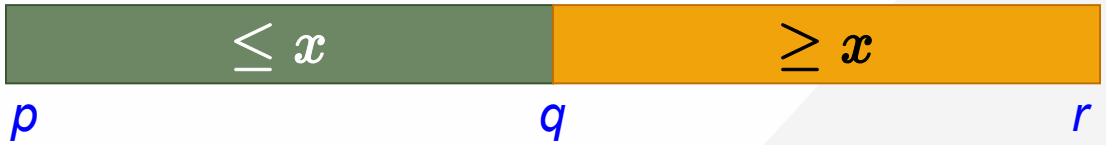
```
QUICKSORT (A, p, r)
  if p < r then
    q = H-PARTITION(A, p, r)
    QUICKSORT(A, p, q)
    QUICKSORT(A, q + 1, r)
  endif
```

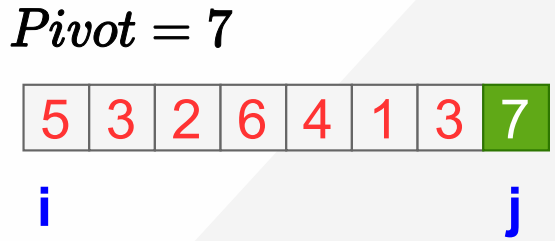
Initial invocation: `QUICKSORT(A,1,n)`



Hoare's Partitioning Algorithm: Pivot Selection

- if we select pivot to be $A[r]$ instead of $A[p]$ in **H-PARTITION**



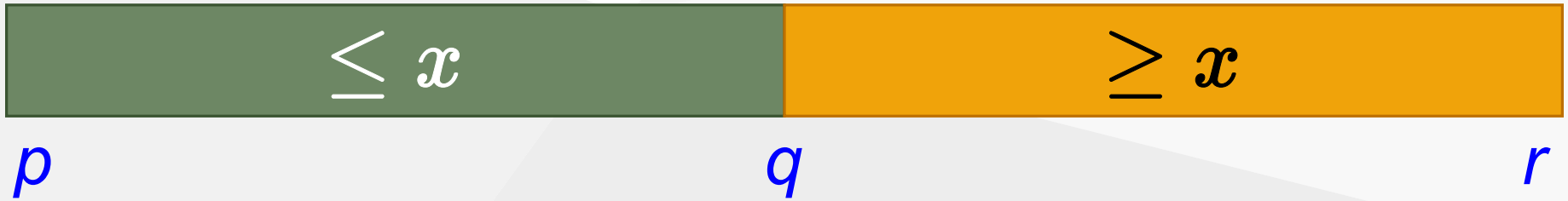


- Consider the example where $A[r]$ is the largest element in the array:
  - End of H-PARTITION: $i = j = r$
  - In QUICKSORT: $q = r$
    - So, recursive call to:
      - `QUICKSORT(A, p, q=r)`
        - **infinite loop**

Correctness of Hoare's Algorithm (1)

We need to prove 3 claims to show correctness:

- Indices i and j never reference A outside the interval $A[p \ldots r]$
- Split is always non-trivial; i.e., $j \neq r$ at termination
- Every element in $A[p \ldots j] \leq$ every element in $A[j+1 \ldots r]$ at termination



Correctness of Hoare's Algorithm (2)

- Notations:
  - k: # of times the while-loop iterates until termination
  - i_m: the value of index i at the end of iteration m
  - j_m: the value of index j at the end of iteration m
  - x: the value of the pivot element
- **Note**: We always have $i_1 = p$ and $p \leq j_1 \leq r$ because $x = A[p]$

Correctness of Hoare's Algorithm (3)

Lemma 1: Either $i_k = j_k$ or $i_k = j_k + 1$ at termination

Proof of Lemma 1:

- The algorithm terminates when $i \geq j$ (the else condition).
- So, it is sufficient to prove that $i_k - j_k \leq 1$
- There are 2 cases to consider:
  - Case 1: $k = 1$, i.e. the algorithm terminates in a single iteration
  - Case 2: $k > 1$, i.e. the alg. does not terminate in a single iter.

By contradiction, assume there is a run with $i_k - j_k > 1$

Correctness of Hoare's Algorithm (4)

Original correctness claims:

- Indices i and j never reference A outside the interval $A[p \ldots r]$
- Split is always non-trivial; i.e., $j \neq r$ at termination

Proof:

- **For $k = 1$:**
  - Trivial because $i_1 = j_1 = p$ (*see Case 1 in proof of Lemma 2*)
- **For $k > 1$:**
  - $i_k > p$ and $j_k < r$ (*due to the repeat-until loops moving indices*)
  - $i_k \leq r$ and $j_k \geq p$ (*due to Lemma 1 and the statement above*)

The proof of claims (a) and (b) complete

Correctness of Hoare's Algorithm (5)

Lemma 2: At the end of iteration m, where $m < k$ (*i.e. m is not the last iteration*), we must have: $A[p \dots i_m] \leq x$ and $A[j_m \dots r] \geq x$

Proof of Lemma 2:

- **Base case:** $m = 1$ and $k > 1$ (*i.e. the alg. does not terminate in the first iter.*)

Ind. Hyp.: At the end of iteration $m - 1$, where $m < k$ (*i.e. m is not the last iteration*), we must have: $A[p \dots i_m - 1] \leq x$ and $A[j_m - 1 \dots r] \geq x$

General case: The lemma holds for m, where $m < k$

Proof of base case complete!

Correctness of Hoare's Algorithm (6)

Original correctness claim:

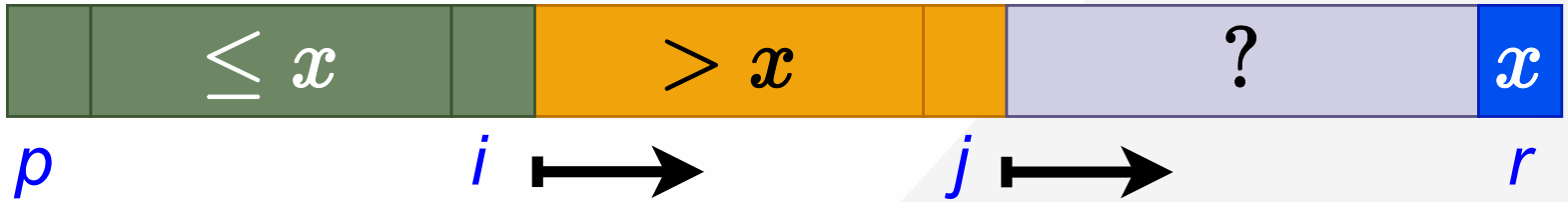
- (c) Every element in $A[\ldots j] \leq$ every element in $A[j + \ldots r]$ at termination

Proof of claim (c)

- There are 3 cases to consider:
  - **Case 1:** $k = 1$, i.e. the algorithm terminates in a single iteration
  - **Case 2:** $k > 1$ and $i_k = j_k$
  - **Case 3:** $k > 1$ and $i_k = j_k + 1$

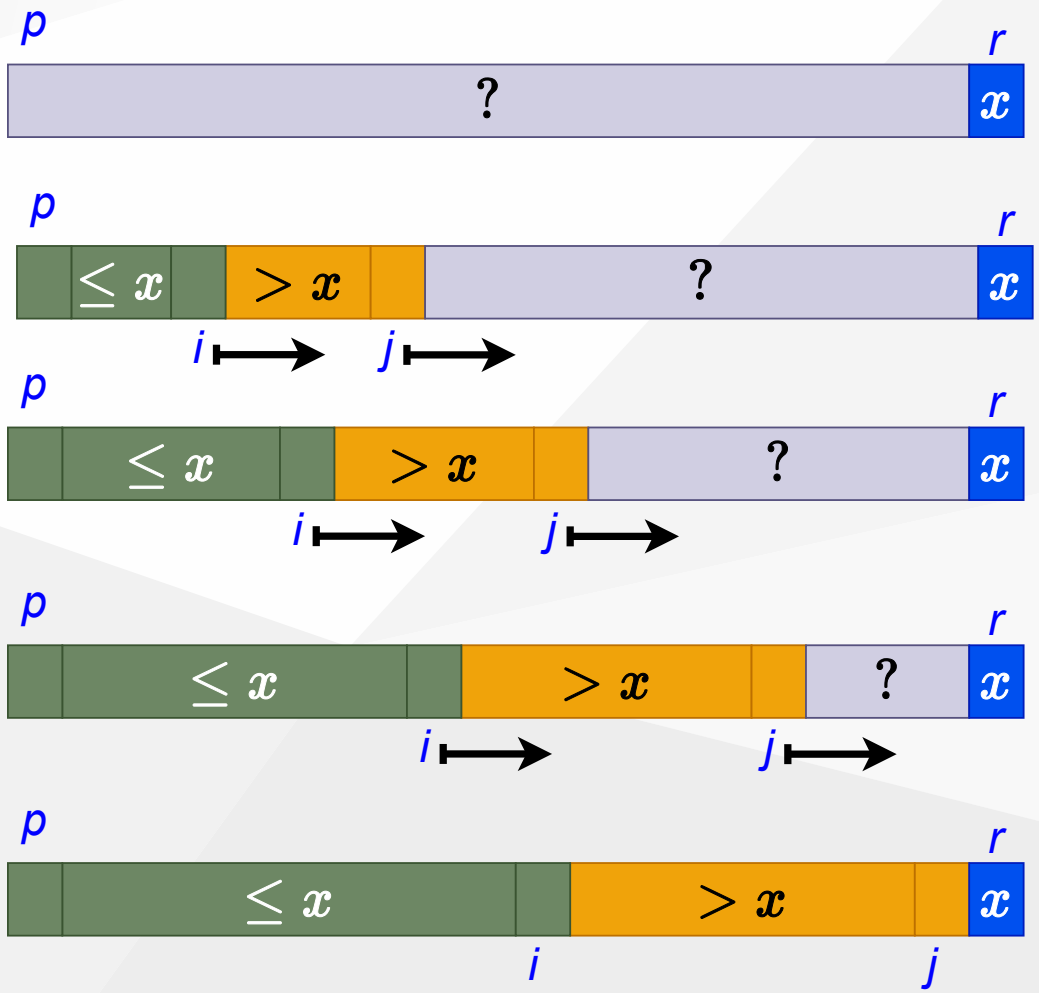
Lomuto's Partitioning Algorithm (1)

- Choose a pivot element: $pivot = x = A[r]$

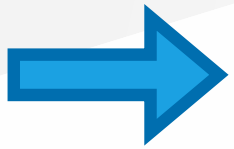


- Grow two regions:
  - from left to right: $A[p \ldots i]$
  - from left to right: $A[i+1 \ldots j]$
    - such that:
      - every element in $A[p \ldots i] \leq pivot$
      - every element in $A[i+1 \ldots j] > pivot$

Lomuto's Partitioning Algorithm (2)



Lomuto's Partitioning Algorithm Ex. (Step-1)



L-PARTITION (A, *p*, *r*)

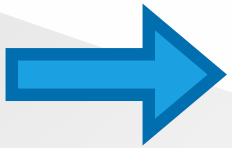
pivot ← A[*r*]
i ← *p* – 1
for j ← *p* to *r* – 1 do
  if A[*j*] ≤ *pivot* then
    i ← *i* + 1
    exchange A[*i*] ↔ A[*j*]
exchange A[*i* + 1] ↔ A[*r*]
return *i* + 1

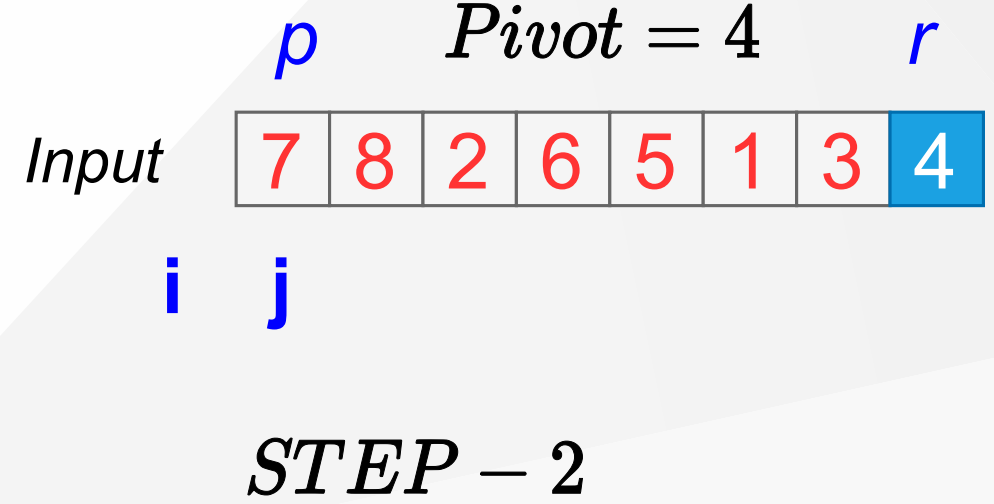
$STEP - 1$

Lomuto's Partitioning Algorithm Ex. (Step-2)

L-PARTITION (A, *p*, *r*)

pivot ← A[*r*]
i ← *p* – 1
for j ← *p* to *r* – 1 do
  if A[*j*] ≤ *pivot* then
    i ← *i* + 1
    exchange A[*i*] ↔ A[*j*]
exchange A[*i* + 1] ↔ A[*r*]
return *i* + 1

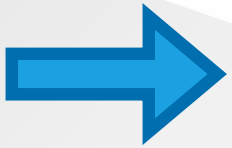


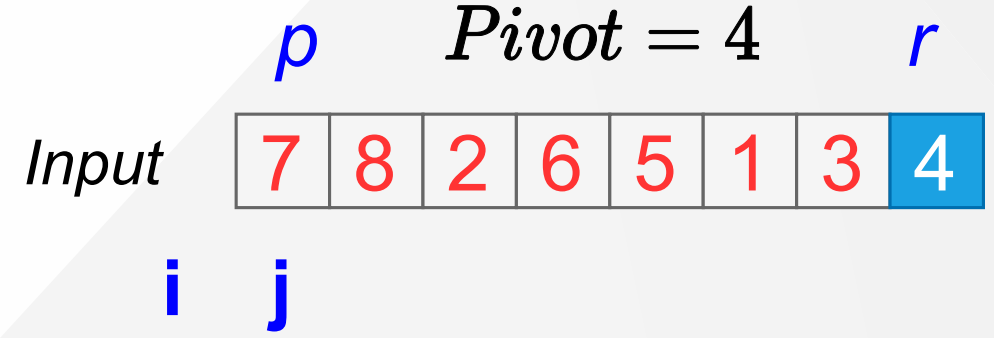


Lomuto's Partitioning Algorithm Ex. (Step-3)

L-PARTITION (A, *p*, *r*)

pivot ← A[*r*]
i ← *p* – 1
for j ← *p* to *r* – 1 do
  if A[*j*] ≤ *pivot* then
    i ← *i* + 1
    exchange A[*i*] ↔ A[*j*]
exchange A[*i* + 1] ↔ A[*r*]
return *i* + 1

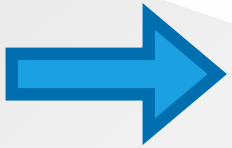


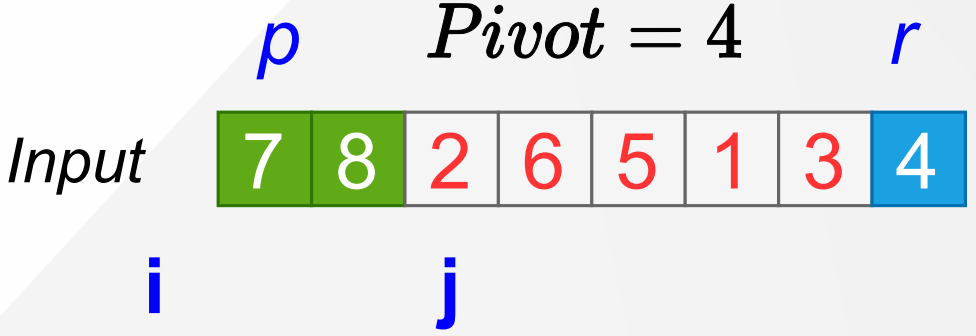


Lomuto's Partitioning Algorithm Ex. (Step-4)

L-PARTITION (A, *p*, *r*)

pivot ← A[*r*]
i ← *p* – 1
for j ← *p* to *r* – 1 do
  if A[*j*] ≤ *pivot* then
    i ← *i* + 1
    exchange A[*i*] ↔ A[*j*]
exchange A[*i* + 1] ↔ A[*r*]
return *i* + 1



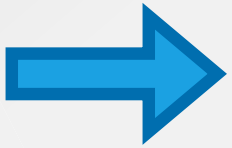


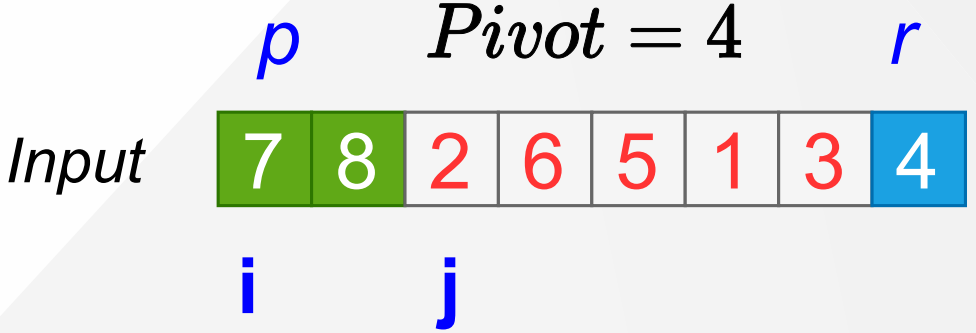
$STEP-4$

Lomuto's Partitioning Algorithm Ex. (Step-5)

L-PARTITION (A, *p*, *r*)

pivot ← A[*r*]
i ← *p* – 1
for j ← *p* to *r* – 1 do
  if A[*j*] ≤ *pivot* then
    i ← *i* + 1
    exchange A[*i*] ↔ A[*j*]
exchange A[*i* + 1] ↔ A[*r*]
return *i* + 1

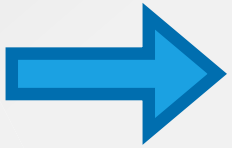


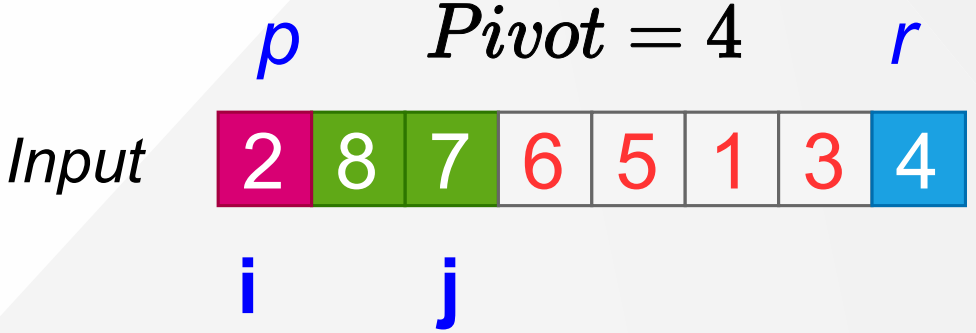


Lomuto's Partitioning Algorithm Ex. (Step-6)

L-PARTITION (A, *p*, *r*)

pivot ← A[*r*]
i ← *p* – 1
for j ← *p* to *r* – 1 do
  if A[*j*] ≤ *pivot* then
    i ← *i* + 1
    exchange A[*i*] ↔ A[*j*]
exchange A[*i* + 1] ↔ A[*r*]
return *i* + 1

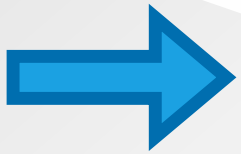


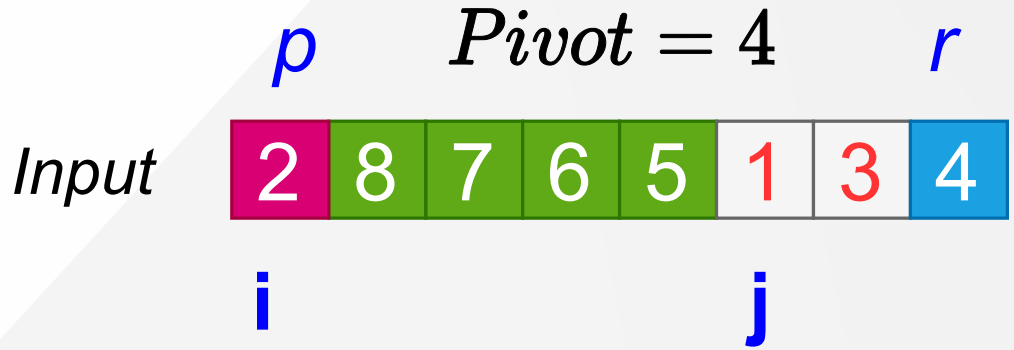


Lomuto's Partitioning Algorithm Ex. (Step-7)

L-PARTITION (A, *p*, *r*)

pivot ← A[*r*]
i ← *p* – 1
for j ← *p* to *r* – 1 do
  if A[*j*] ≤ *pivot* then
    i ← *i* + 1
    exchange A[*i*] ↔ A[*j*]
exchange A[*i* + 1] ↔ A[*r*]
return *i* + 1

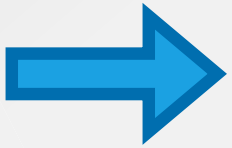


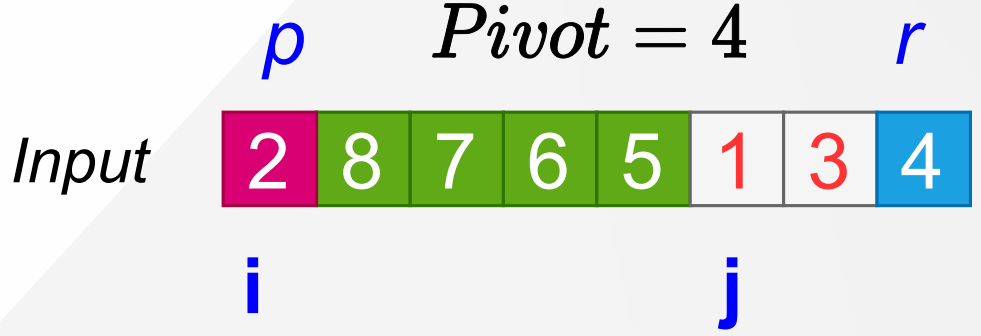


Lomuto's Partitioning Algorithm Ex. (Step-8)

L-PARTITION (A, *p*, *r*)

pivot ← A[*r*]
i ← *p* – 1
for *j* ← *p* **to** *r* – 1 **do**
  if A[*j*] ≤ ***pivot*** **then**
    i ← *i* + 1
    exchange A[*i*] ↔ A[*j*]
exchange A[*i* + 1] ↔ A[*r*]
return *i* + 1



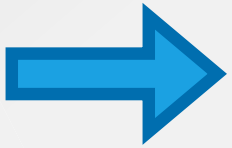


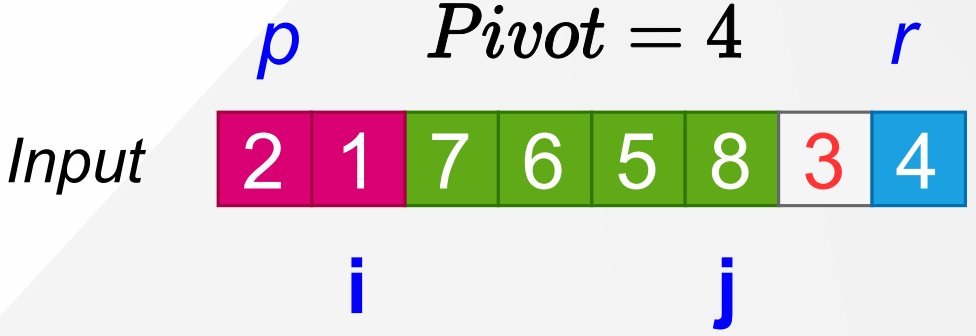
$STEP - 8$

Lomuto's Partitioning Algorithm Ex. (Step-9)

L-PARTITION (A, *p*, *r*)

pivot ← A[*r*]
i ← *p* – 1
for j ← *p* to *r* – 1 do
  if A[*j*] ≤ *pivot* then
    i ← *i* + 1
    exchange A[*i*] ↔ A[*j*]
exchange A[*i* + 1] ↔ A[*r*]
return *i* + 1



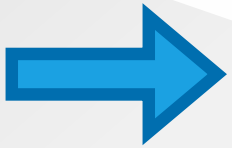


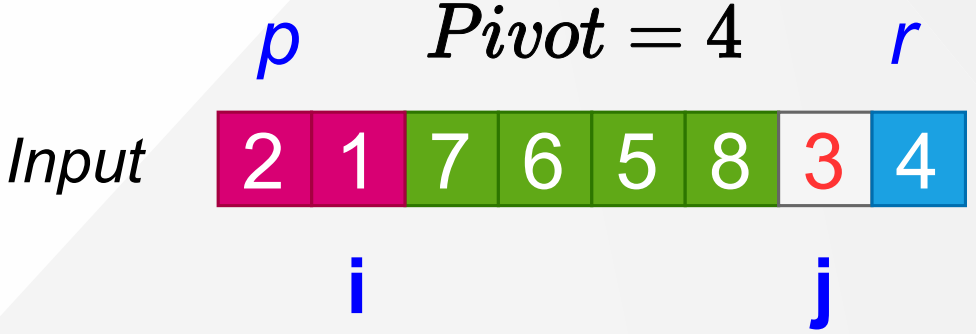
$STEP - 9$

Lomuto's Partitioning Algorithm Ex. (Step-10)

L-PARTITION (A, *p*, *r*)

pivot ← A[*r*]
i ← *p* – 1
for *j* ← *p* **to** *r* – 1 **do**
  if A[*j*] ≤ ***pivot*** **then**
    i ← *i* + 1
    exchange A[*i*] ↔ A[*j*]
exchange A[*i* + 1] ↔ A[*r*]
return *i* + 1



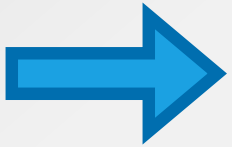


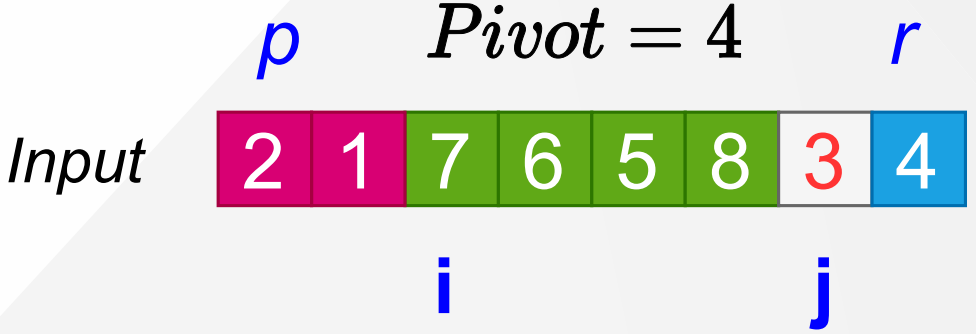
$STEP - 10$

Lomuto's Partitioning Algorithm Ex. (Step-11)

L-PARTITION (A, *p*, *r*)

pivot ← A[*r*]
i ← *p* – 1
for j ← *p* to *r* – 1 do
  if A[*j*] ≤ *pivot* then
    i ← *i* + 1
    exchange A[*i*] ↔ A[*j*]
exchange A[*i* + 1] ↔ A[*r*]
return *i* + 1



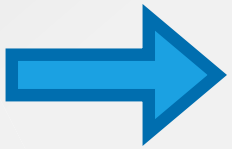


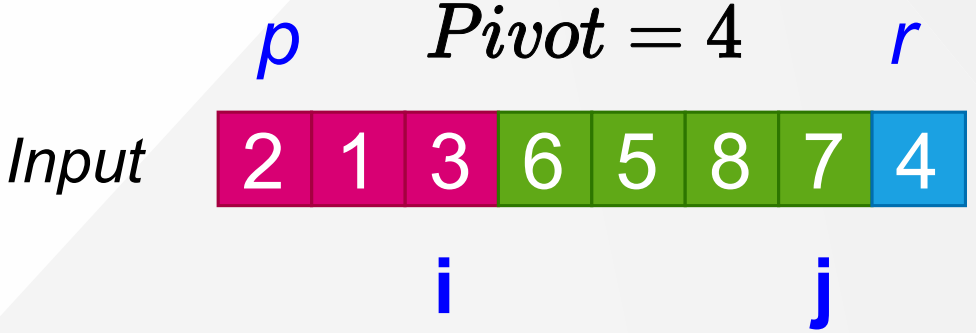
$STEP - 11$

Lomuto's Partitioning Algorithm Ex. (Step-12)

L-PARTITION (A, *p*, *r*)

pivot ← A[*r*]
i ← *p* – 1
for j ← *p* to *r* – 1 do
  if A[*j*] ≤ *pivot* then
    i ← *i* + 1
    exchange A[*i*] ↔ A[*j*]
exchange A[*i* + 1] ↔ A[*r*]
return *i* + 1



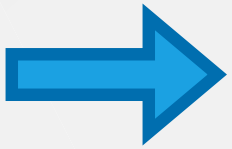


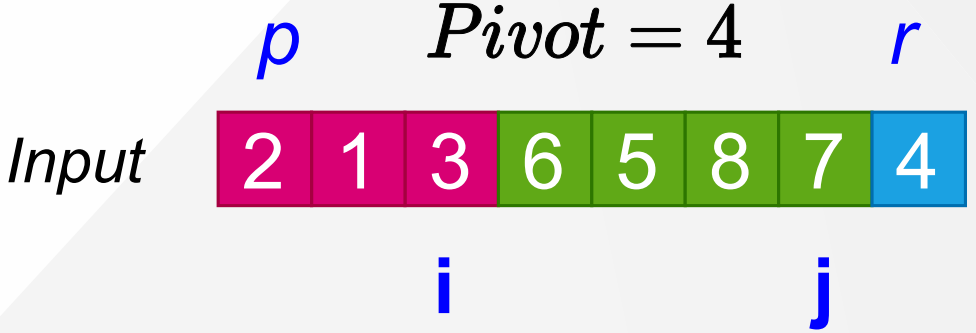
$STEP - 12$

Lomuto's Partitioning Algorithm Ex. (Step-13)

L-PARTITION (A, *p*, *r*)

pivot ← A[*r*]
i ← *p* – 1
for j ← *p* to *r* – 1 do
  if A[*j*] ≤ *pivot* then
    i ← *i* + 1
    exchange A[*i*] ↔ A[*j*]
exchange A[*i* + 1] ↔ A[*r*]
return *i* + 1



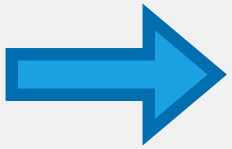


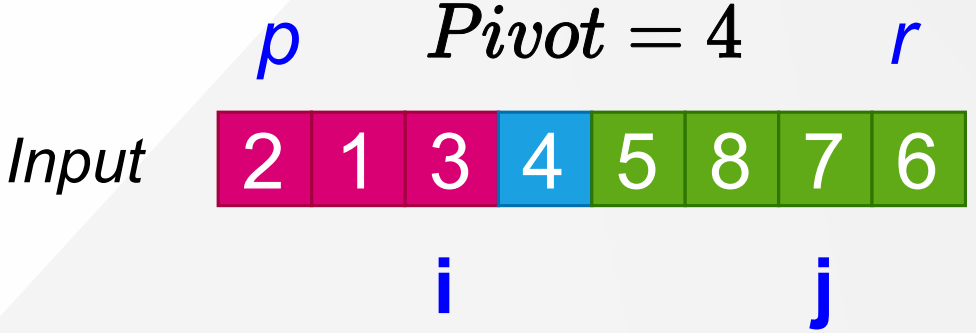
$STEP - 13$

Lomuto's Partitioning Algorithm Ex. (Step-14)

L-PARTITION (A, *p*, *r*)

pivot ← A[*r*]
i ← *p* – 1
for j ← *p* to *r* – 1 do
  if A[*j*] ≤ *pivot* then
    i ← *i* + 1
    exchange A[*i*] ↔ A[*j*]
exchange A[*i* + 1] ↔ A[*r*]
return *i* + 1



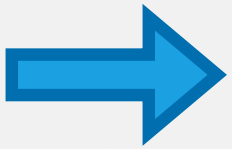


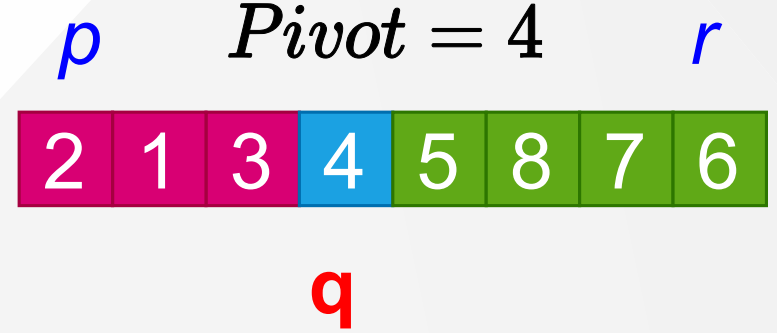
$STEP - 14$

Lomuto's Partitioning Algorithm Ex. (Step-15)

L-PARTITION (A, *p*, *r*)

pivot ← A[*r*]
i ← *p* – 1
for j ← *p* to *r* – 1 do
  if A[*j*] ≤ *pivot* then
    i ← *i* + 1
    exchange A[*i*] ↔ A[*j*]
exchange A[*i* + 1] ↔ A[*r*]
return *i* + 1



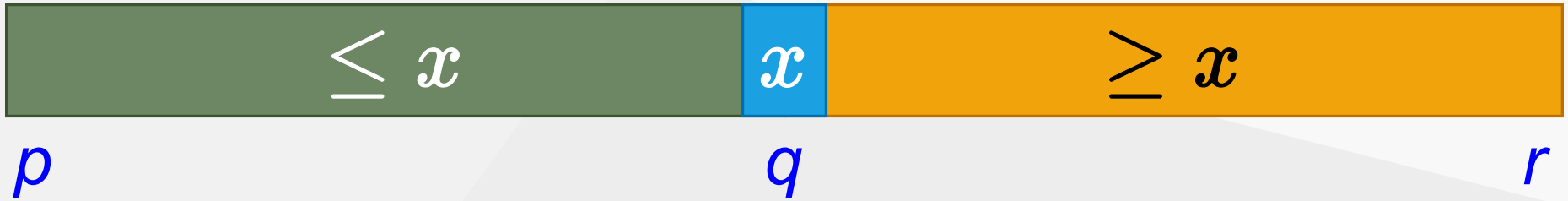


$STEP - 15$

Quicksort with Lomuto's Partitioning Algorithm

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

Initial invocation: `QUICKSORT(A,1,n)`



Comparison of Hoare's & Lomuto's Algorithms (1)

- Notation: $n = r - p + 1$
  - $pivot = A[p]$ (*Hoare*)
  - $pivot = A[r]$ (*Lomuto*)
- **# of element exchanges:** $e(n)$
  - Hoare: $0 \geq e(n) \geq \lfloor \frac{n}{2} \rfloor$
    - **Best**: $k = 1$ with $i_1 = j_1 = p$ (i.e., $A[p+1 \ldots r] > pivot$)
    - **Worst**: $A[p+1 \ldots p + \lfloor \frac{n}{2} \rfloor - 1] \geq pivot \geq A[p + \lceil \frac{n}{2} \rceil \ldots r]$
  - Lomuto : $1 \leq e(n) \leq n$
    - **Best**: $A[p \ldots r-1] > pivot$
    - **Worst**: $A[p \ldots r-1] \leq pivot$

Comparison of Hoare's & Lomuto's Algorithms (2)

- # of element comparisons: $c_e(n)$
  - Hoare: $n + 1 \leq c_e(n) \leq n + 2$
    - Best: $i_k = j_k$
    - Worst: $i_k = j_k + 1$
  - Lomuto: $c_e(n) = n - 1$
- # of index comparisons: $c_i(n)$
  - Hoare: $1 \leq c_i(n) \leq \lfloor \frac{n}{2} \rfloor + 1 | (c_i(n) = e(n) + 1)$
  - Lomuto: $c_i(n) = n - 1$

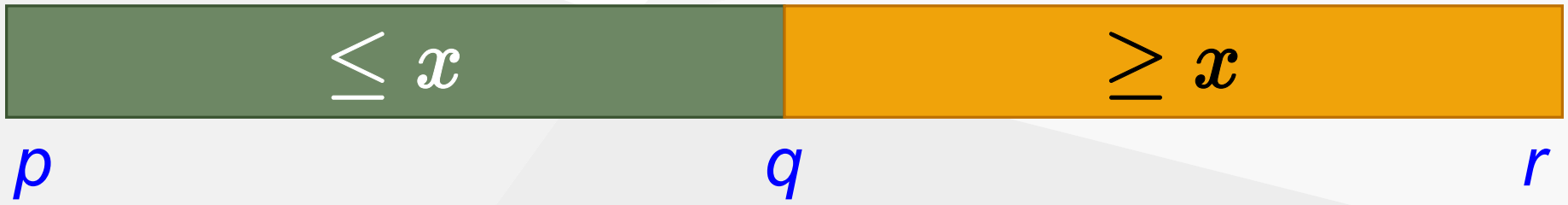
Comparison of Hoare's & Lomuto's Algorithms (3)

- **# of index increment/decrement operations:** $a(n)$
  - **Hoare**: $n+1 \leq a(n) \leq n+2 | (a(n) = c_e(n))$
  - **Lomuto**: $n \leq a(n) \leq 2n-1 | (a(n) = e(n) + (n-1))$
- Hoare's algorithm is in general faster
- Hoare behaves better when pivot is repeated in $A[p \ldots r]$
  - **Hoare**: Evenly distributes them between left & right regions
  - **Lomuto**: Puts all of them to the left region

Analysis of Quicksort (1)

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

Initial invocation: `QUICKSORT(A,1,n)`



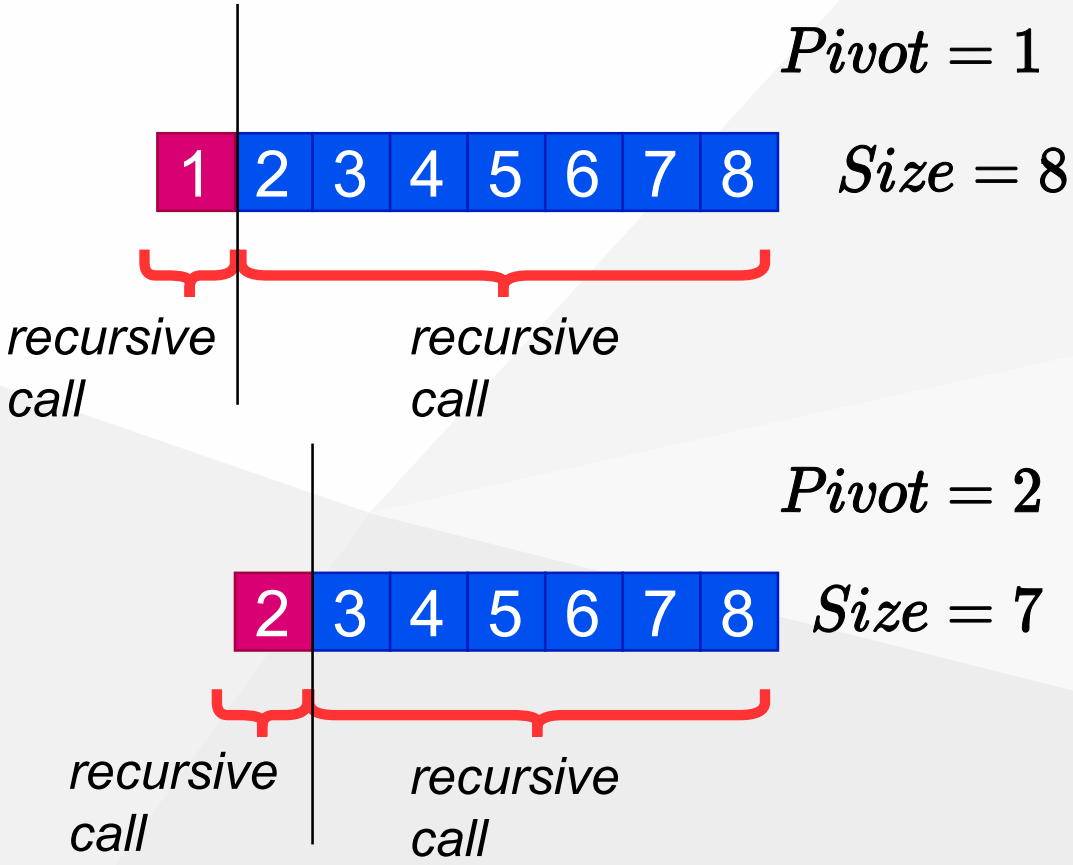
Assume **all elements are distinct** in the following analysis

Analysis of Quicksort (2)

- **H-PARTITION** always chooses $A[p]$ (the first element) as the pivot.
- The runtime of **QUICKSORT** on an already-sorted array is $\Theta(n^2)$

Example: An Already Sorted Array

Partitioning always leads to 2 parts of size 1 and $n - 1$



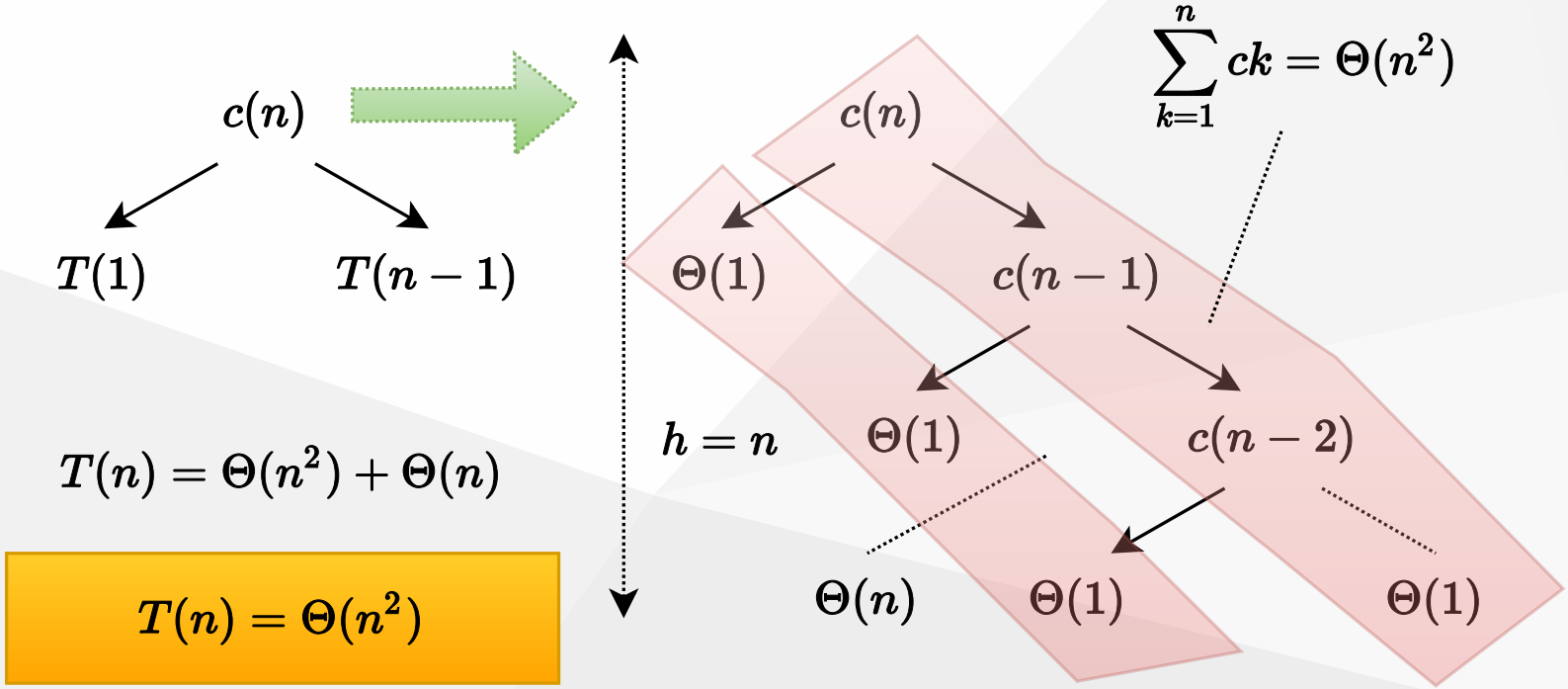
Worst Case Analysis of Quicksort

- **Worst case** is when the **PARTITION** algorithm always returns **imbalanced partitions** (of size 1 and $n-1$) in every recursive call.
  - This happens when the pivot is selected to be either the min or **max** element.
  - This happens for **H-PARTITION** when the input array is already sorted or reverse sorted

$$\begin{aligned} T(n) &= T(1) + T(n-1) + \Theta(n) \\ &= T(n-1) + \Theta(n) \\ &= \Theta(n2) \end{aligned}$$

Worst Case Recursion Tree

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



Best Case Analysis (for intuition only)

- If we're extremely lucky, **H-PARTITION** splits the array evenly at every recursive call

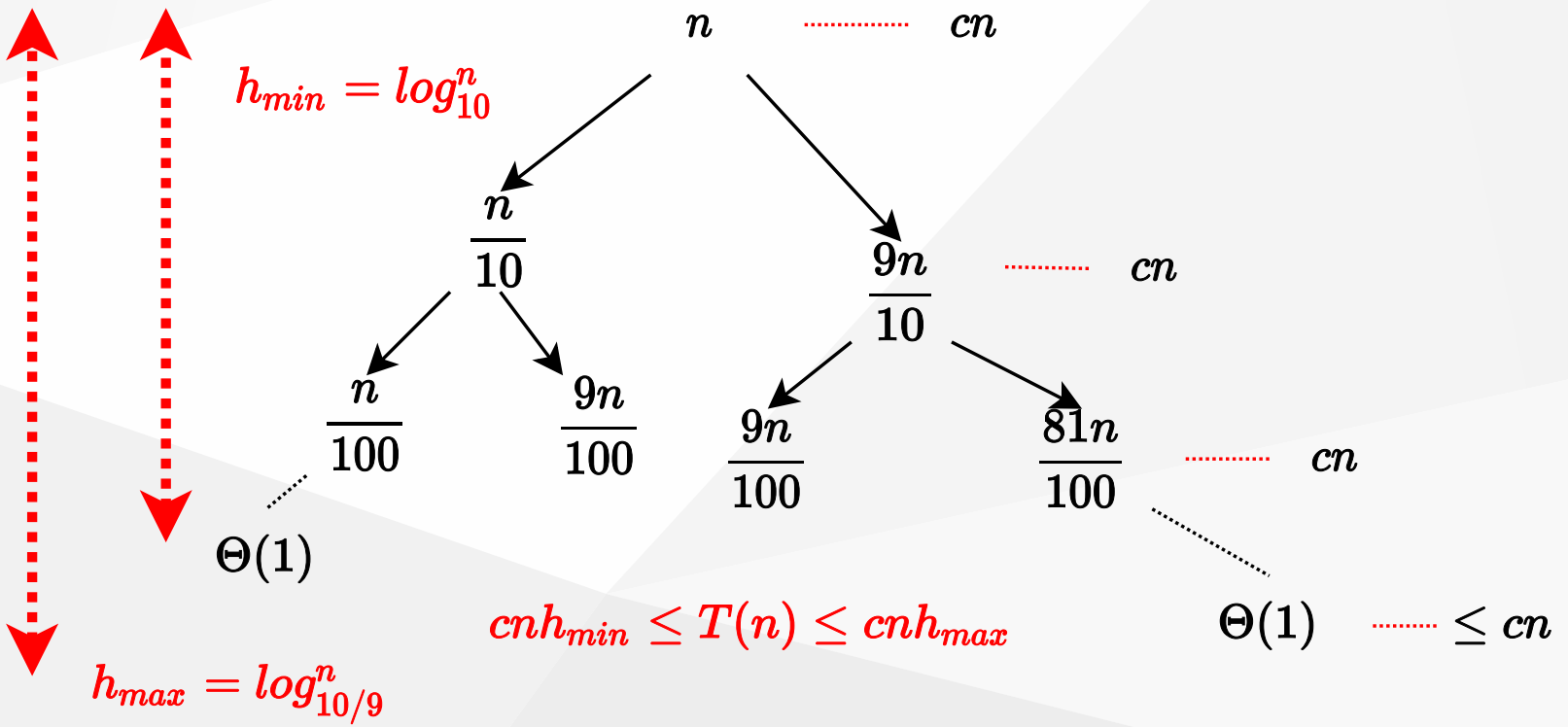
$$T(n) = 2T(n/2) + \Theta(n)$$
$$= \Theta(nlgn)$$

(same as merge sort)

- Instead of splitting $0.5 : 0.5$, if we split $0.1 : 0.9$ then we need solve following equation.

$$T(n) = T(n/10) + T(9n/10) + \Theta(n)$$
$$= \Theta(nlgn)$$

"Almost-Best" Case Analysis



$$cnh_{min} \leq T(n) \leq cnh_{max}$$

$$cnlog_{10}^{n} \leq T(n) \leq cnlog_{10/9}^{n}$$

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

Balanced Partitioning (1)

- We have seen that if **H-PARTITION** always splits the array with $0.1 - to - 0.9$ ratio, the runtime will be $\Theta(nlgn)$.
- Same is true with a split ratio of $0.01 - to - 0.99$, etc.
- Possible to show that if the split has always constant $(\Theta(1))$ proportionality, then the runtime will be $\Theta(nlgn)$.
- In other words, for a **constant** $\alpha|(0 < \alpha \leq 0.5)$:
  - α–to–$(1 - \alpha)$ proportional split yields $\Theta(nlgn)$ total runtime

Balanced Partitioning (2)

- In the rest of the analysis, assume that all input permutations are equally likely.
  - This is only to gain some intuition
  - We cannot make this assumption for average case analysis
  - We will revisit this assumption later
- Also, assume that all input elements are distinct.

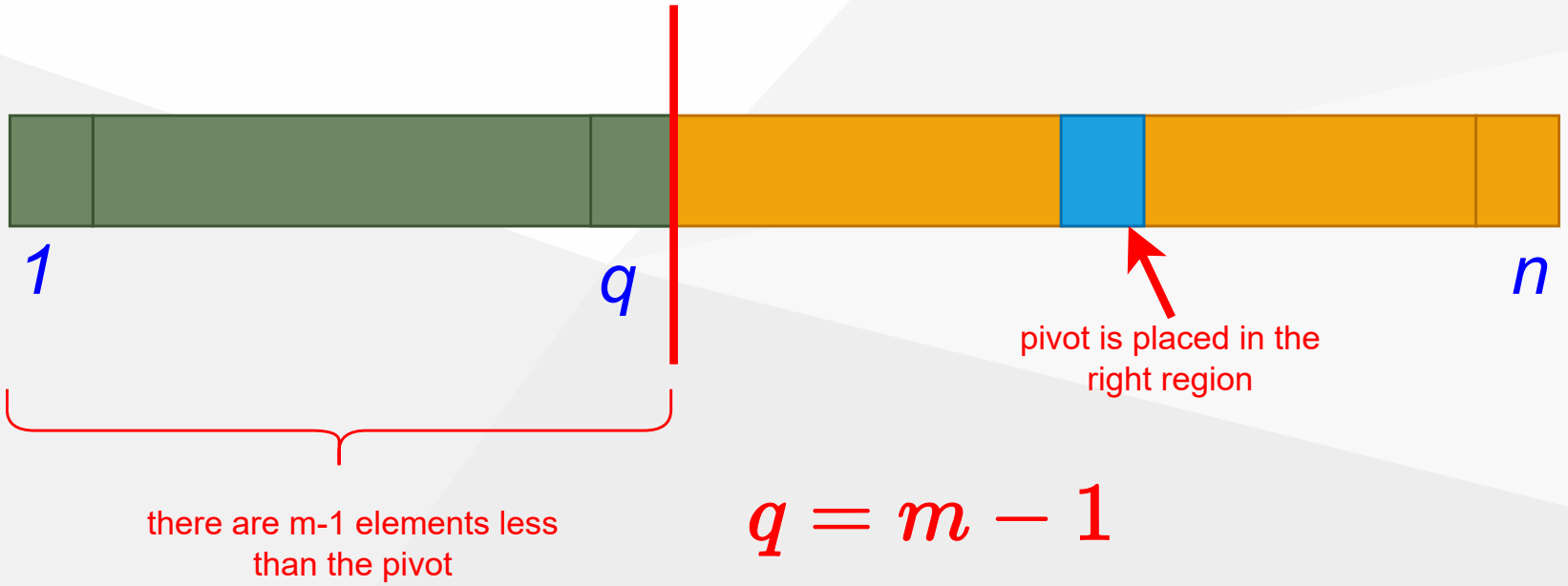
Balanced Partitioning (3)

- **Question:** What is the probability that H-PARTITION returns a split that is more balanced than $0.1 - to - 0.9$?

Balanced Partitioning (4)

Reminder: *H-PARTITION* will place the pivot in the right partition unless the pivot is the smallest element in the arrays.

Question: If the pivot selected is the mth smallest value $(1 < m \leq n)$ in the input array, what is the size of the left region after partitioning?



$$q = m - 1$$

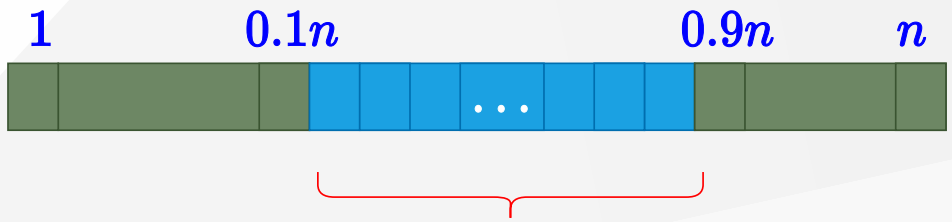
Balanced Partitioning (5)

- **Question:** What is the probability that the **pivot** selected is the m^{th} smallest value in the array of size n?
  - $1/n$ (*since all input permutations are equally likely*)
- **Question:** What is the probability that the left partition returned by **H-PARTITION** has size m, where $1 < m < n$?
  - $1/n$ (*due to the answers to the previous 2 questions*)

Balanced Partitioning (6)

- **Question:** What is the probability that H-PARTITION returns a split that is more balanced than $0.1 - to - 0.9$?

$$
\begin{aligned}
Probability &= \sum_{q=0.1n+1}^{0.9n-1} \frac{1}{n} \\
&= \frac{1}{n}(0.9n - 1 - 0.1n - 1 + 1) \\
&= 0.8 - \frac{1}{n} \\
&\approx 0.8 \text{ for large n}
\end{aligned}
$$



The partition boundary will be in this region for a more balanced split than $0.1 - to - 0.9$

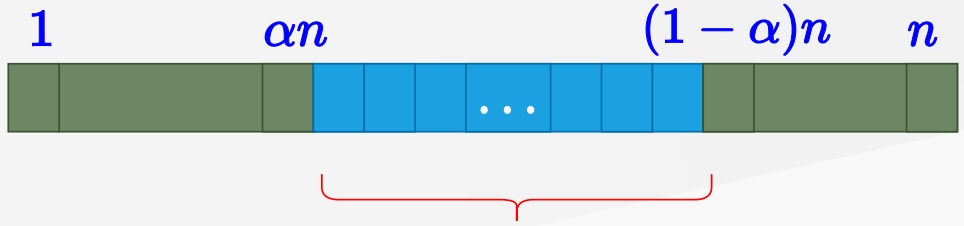
Balanced Partitioning (7)

- The probability that **H-PARTITION** yields a split that is more balanced than $0.1 - to - 0.9$ is 80% on a random array.
- Let $P_{\alpha>}$ be the probability that **H-PARTITION** yields a split more balanced than $\alpha - to - (1 - \alpha)$, where $0 < \alpha \leq 0.5$
- Repeat the analysis to generalize the previous result

Balanced Partitioning (8)

- **Question:** What is the probability that H-PARTITION returns a split that is more balanced than $\alpha - to - (1-\alpha)$?

$$
\begin{aligned}
Probability &= \sum_{q=\alpha n+1}^{(1-\alpha)n-1} \frac{1}{n} \\
&= \frac{1}{n}((1-\alpha)n - 1 - \alpha n - 1 + 1) \\
&= (1-2\alpha) - \frac{1}{n} \\
&\approx (1-2\alpha) \text{ for large n}
\end{aligned}
$$



The partition boundary will be in this region for a more balanced split than

$\alpha n - to - (1-\alpha)n$

Balanced Partitioning (9)

- We found $P_{\alpha>} = 1 - 2\alpha$
  - Ex: $P_{0.1>} = 0.8$ and $P_{0.01>} = 0.98$
- Hence, **H-PARTITION** produces a split
  - **more balanced than a**
    - $0.1 - to - 0.9$ split 80% of the time
    - $0.01 - to - 0.99$ split 98% of the time
  - **less balanced than a**
    - $0.1 - to - 0.9$ split 20% of the time
    - $0.01 - to - 0.99$ split 2% of the time

Intuition for the Average Case (1)

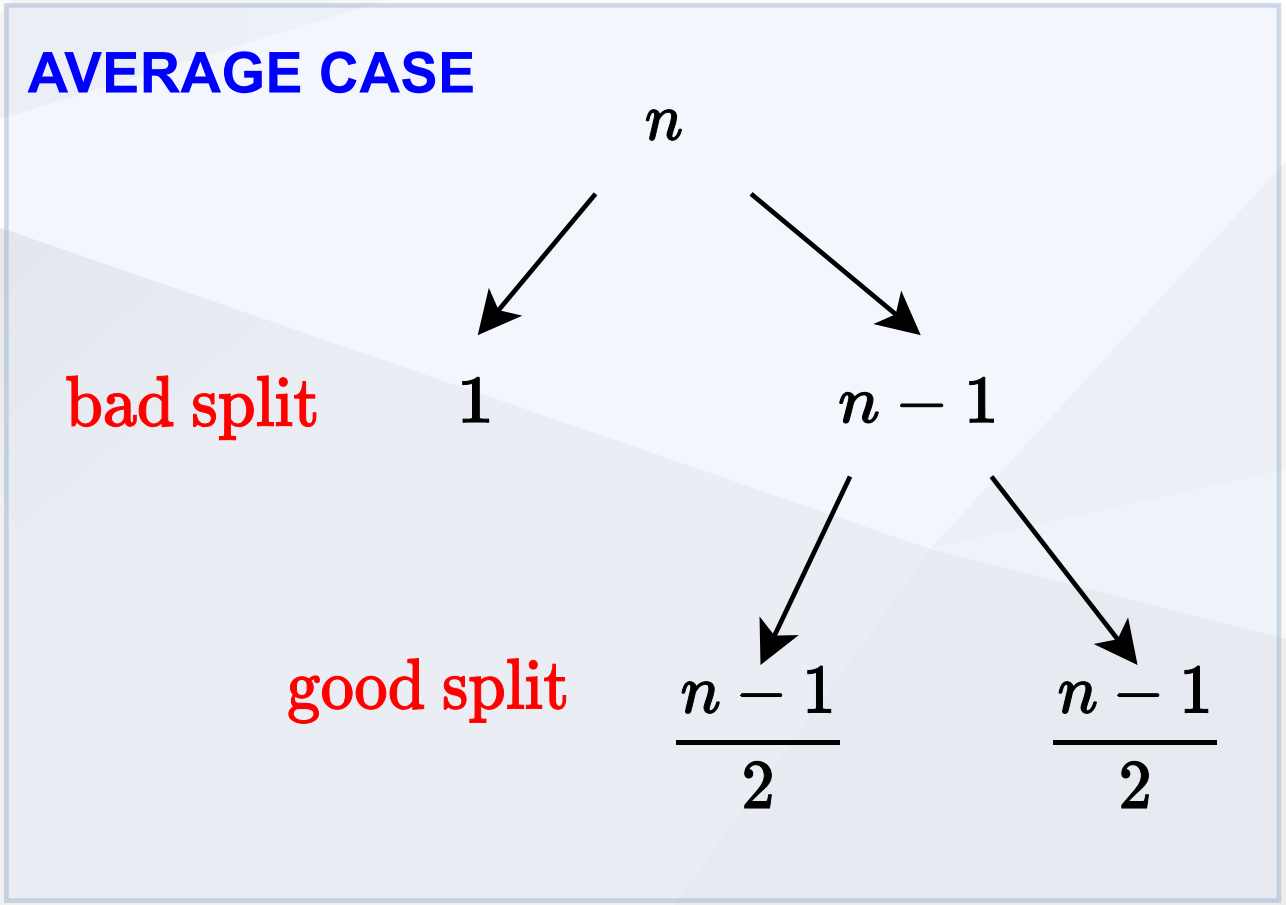
- **Assumption:** All permutations are equally likely
  - Only for intuition; we'll revisit this assumption later
- **Unlikely:** Splits always the same way at every level
- **Expectation:**
  - Some splits will be reasonably balanced
  - Some splits will be fairly unbalanced
- **Average case:** A mix of good and bad splits
  - **Good** and **bad** splits distributed randomly thru the tree

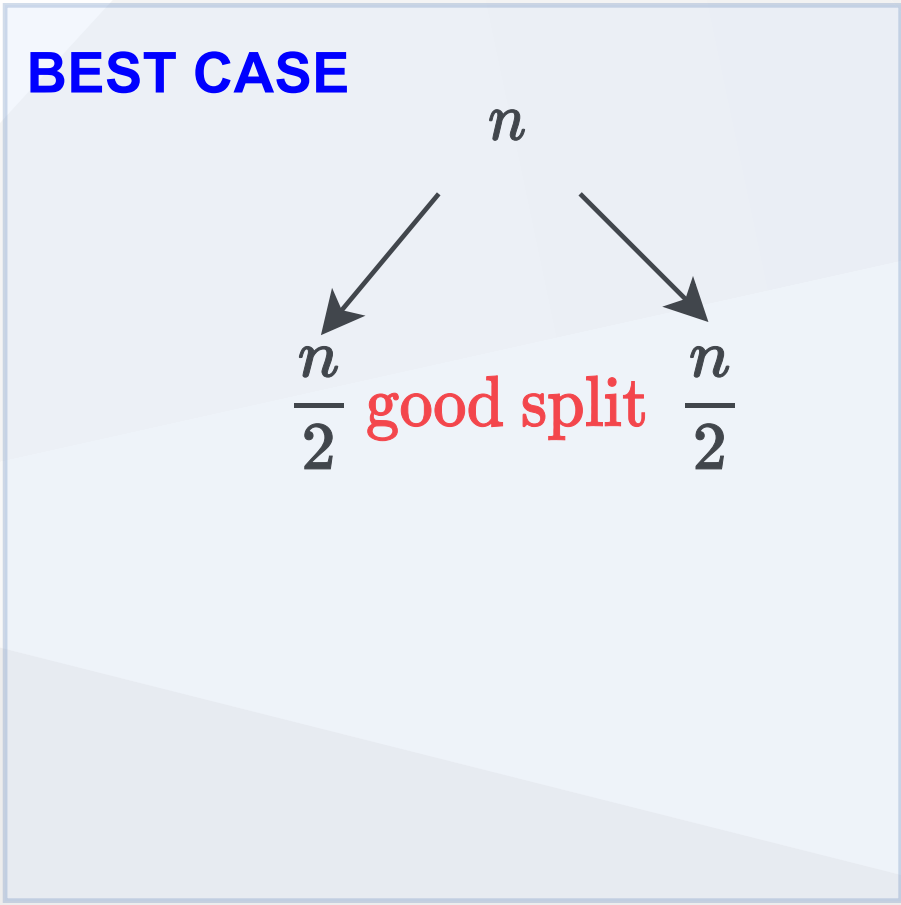
Intuition for the Average Case (2)

- **Assume for intuition:** Good and bad splits occur in the alternate levels of the tree
  - **Good split:** Best case split
  - **Bad split:** Worst case split

Intuition for the Average Case (3)

Compare 2-successive levels of avg case vs. 1 level of best case





Intuition for the Average Case (4)

- In terms of the remaining subproblems, **two levels of avg case** is slightly better than the **single level of the best case**
- The avg case has **extra divide cost of** $\Theta(n)$ at alternate levels
- The extra divide cost $\Theta(n)$ of bad splits absorbed into the $\Theta(n)$ of good splits.
- Running time is still $\Theta(nlgn)$
    - But, slightly larger hidden constants, because the height of the recursion tree is about twice of that of best case.

\end{align*}

- All elements in $L \leq$ all elements in R - L contains: - $|L| = q–p+1$ $=$ k smallest elements of $A[p...r]$ - if $i \leq |L| = k$ then - **search** L recursively for its i^{th} smallest element - else - **search** R recursively for its $(i-k)^{th}$ smallest element --- ## Runtime Analysis (1) - **Worst case:** - Imbalanced partitioning at every level and the recursive call always to the larger partition

\begin{align*}
& = {1,\underbrace{2,3,4,5,6,7,8}{*text{recursive call}} } &
*i & = 8 *
& = {2,\underbrace{3,4,5,6,7,8}{\text{recursive call}} } &
i & = 7
\end{align*}

undefined

\begin{align*}
T(n) &= T(n-1) + \Theta(n) \
T(n) &= \Theta(n^2)
\end{align*}

$$-**Bestcase:**Balancedpartitioningateveryrecursivelevel$$

\begin{align*}