Design and Analysis of Algorithms
Part I: Divide and Conquer
Lecture 4: The Polynomial Multiplication Problem and Quicksort Problem

Yongxin Tong (童咏昕)

School of CSE, Beihang University

yxtong@buaa.edu.cn

Outline

- Review to Divide-and-Conquer Paradigm
- Polynomial Multiplication Problem
  - Problem definition
  - A brute force algorithm
  - A first divide-and-conquer algorithm
  - An improved divide-and-conquer algorithm
  - Analysis of the divide-and-conquer algorithm
- Quicksort Problem
  - Basic partition
  - Randomized partition and randomized quicksort
  - Analysis of the randomized quicksort

Outline

- Review to Divide-and-Conquer Paradigm
- Polynomial Multiplication Problem
  - Problem definition
  - A brute force algorithm
  - A first divide-and-conquer algorithm
  - An improved divide-and-conquer algorithm
  - Analysis of the divide-and-conquer algorithm
- Quicksort Problem
  - Basic partition
  - Randomized partition and randomized quicksort
  - Analysis of the randomized quicksort

Review to Divide-and-Conquer Paradigm

- Divide-and-conquer (D&C) is an important algorithm design paradigm.
  - **Divide**
    Dividing a given problem into two or more subproblems (ideally of approximately equal size)
  - **Conquer**
    Solving each subproblem (directly if small enough or recursively)
  - **Combine**
    Combining the solutions of the subproblems into a global solution

Review to Divide-and-Conquer Paradigm

- In Part I, we will illustrate Divide-and-Conquer using several examples:
  - Maximum Contiguous Subarray (最大子数组)
  - Counting Inversions (逆序计数)
  - Polynomial Multiplication (多项式乘法)
  - QuickSort and Partition (快速排序与划分)
  - Lower Bound for Sorting (基于比较的排序下界)

Review to Divide-and-Conquer Paradigm

- In Part I, we will illustrate Divide-and-Conquer using several examples:
  - Maximum Contiguous Subarray (最大子数组)
  - Counting Inversions (逆序计数)
  - Polynomial Multiplication (多项式乘法)
  - QuickSort and Partition (快速排序与划分)
  - Lower Bound for Sorting (基于比较的排序下界)

Outline

- Review to Divide-and-Conquer Paradigm
- Polynomial Multiplication Problem
  - Problem definition
  - A brute force algorithm
  - A first divide-and-conquer algorithm
  - An improved divide-and-conquer algorithm
  - Analysis of the divide-and-conquer algorithm
- Quicksort Problem
  - Basic partition
  - Randomized partition and randomized quicksort
  - Analysis of the randomized quicksort

The Polynomial Multiplication Problem

Definition (Polynomial Multiplication Problem)

Given two polynomials

$$\begin{aligned} A(x) &= a_0 + a_1 x + \cdots + a_n x^n \\ B(x) &= b_0 + b_1 x + \cdots + b_m x^m \end{aligned}$$

Compute the product $A(x)B(x)$

The Polynomial Multiplication Problem

Definition (Polynomial Multiplication Problem)

Given two polynomials

$$
\begin{aligned}
A(x) &= a_0 + a_1 x + \cdots + a_n x^n \\
B(x) &= b_0 + b_1 x + \cdots + b_m x^m
\end{aligned}
$$

Compute the product $A(x)B(x)$

Example

$$
\begin{aligned}
A(x) &= 1 + 2x + 3x^2 \\
B(x) &= 3 + 2x + 2x^2 \\
A(x)B(x) &= 3 + 8x + 15x^2 + 10x^3 + 6x^4
\end{aligned}
$$

The Polynomial Multiplication Problem

Definition (Polynomial Multiplication Problem)

Given two polynomials

$$\begin{aligned} A(x) &= a_0 + a_1 x + \cdots + a_n x^n \\ B(x) &= b_0 + b_1 x + \cdots + b_m x^m \end{aligned}$$

Compute the product $A(x)B(x)$

Example

$$\begin{aligned} A(x) &= 1 + 2x + 3x^2 \\ B(x) &= 3 + 2x + 2x^2 \\ A(x)B(x) &= 3 + 8x + 15x^2 + 10x^3 + 6x^4 \end{aligned}$$

- Assume that the coefficients a_i and b_i are stored in arrays A[0…n] and B[0…m]

The Polynomial Multiplication Problem

Definition (Polynomial Multiplication Problem)

Given two polynomials

$$\begin{aligned} A(x) &= a_0 + a_1 x + \cdots + a_n x^n \\ B(x) &= b_0 + b_1 x + \cdots + b_m x^m \end{aligned}$$

Compute the product $A(x)B(x)$

Example

$$\begin{aligned} A(x) &= 1 + 2x + 3x^2 \\ B(x) &= 3 + 2x + 2x^2 \\ A(x)B(x) &= 3 + 8x + 15x^2 + 10x^3 + 6x^4 \end{aligned}$$

- Assume that the coefficients a_i and b_i are stored in arrays A[0…n] and B[0…m]
- Cost: number of scalar multiplications and additions

What do we need to compute exactly?

Define

- $A(x) = \sum_{i=0}^{n} a_i x^i$
- $B(x) = \sum_{i=0}^{m} b_i x^i$

What do we need to compute exactly?

Define

- $A(x) = \sum_{i=0}^{n} a_i x^i$
- $B(x) = \sum_{i=0}^{m} b_i x^i$
- $C(x) = A(x)B(x) = \sum_{k=0}^{n+m} c_k x^k$

What do we need to compute exactly?

Define

- $A(x) = \sum_{i=0}^{n} a_i x^i$
- $B(x) = \sum_{i=0}^{m} b_i x^i$
- $C(x) = A(x)B(x) = \sum_{k=0}^{n+m} c_k x^k$

Then

- $c_k = \sum_{0 \le i \le n, 0 \le j \le m, i+j=k} a_i b_j$, for all $0 \le k \le m + n$

What do we need to compute exactly?

Define

- $A(x) = \sum_{i=0}^{n} a_i x^i$
- $B(x) = \sum_{i=0}^{m} b_i x^i$
- $C(x) = A(x)B(x) = \sum_{k=0}^{n+m} c_k x^k$

Then

- $c_k = \sum_{0 \le i \le n, 0 \le j \le m, i+j=k} a_i b_j$, for all $0 \le k \le m + n$

Definition

The vector $(c_0, c_1, \ldots, c_{m+n})$ is the convolution of the vectors $(a_0, a_1, \ldots, a_n)$ and $(b_0, b_1, \ldots, b_m)$

What do we need to compute exactly?

Define

- $A(x) = \sum_{i=0}^{n} a_i x^i$
- $B(x) = \sum_{i=0}^{m} b_i x^i$
- $C(x) = A(x)B(x) = \sum_{k=0}^{n+m} c_k x^k$

Then

- $c_k = \sum_{0 \le i \le n, 0 \le j \le m, i+j=k} a_i b_j$, for all $0 \le k \le m + n$

Definition

The vector $(c_0, c_1, \ldots, c_{m+n})$ is the convolution of the vectors $(a_0, a_1, \ldots, a_n)$ and $(b_0, b_1, \ldots, b_m)$

- We need to calculate convolutions. This is a major problem in digital signal processing

Outline

- Review to Divide-and-Conquer Paradigm
- Polynomial Multiplication Problem
  - Problem definition
  - A brute force algorithm
  - A first divide-and-conquer algorithm
  - An improved divide-and-conquer algorithm
  - Analysis of the divide-and-conquer algorithm
- Quicksort Problem
  - Basic partition
  - Randomized partition and randomized quicksort
  - Analysis of the randomized quicksort

Direct (Brute Force) Approach

To ease analysis, assume n = m.

Direct (Brute Force) Approach

To ease analysis, assume n = m.

- $A(x) = \sum_{i=0}^{n} a_i x^i$
- $B(x) = \sum_{i=0}^{m} b_i x^i$

Direct (Brute Force) Approach

To ease analysis, assume n = m.

- $A(x) = \sum_{i=0}^{n} a_i x^i$
- $B(x) = \sum_{i=0}^{m} b_i x^i$
- $C(x) = A(x)B(x) = \sum_{k=0}^{2n} c_k x^k$ with

$$c_k = \sum_{0 \le i, j \le n, i+j=k} a_i b_j \, , for\ all\ 0 \le k \le 2n$$

Direct (Brute Force) Approach

To ease analysis, assume n = m.

- $A(x) = \sum_{i=0}^{n} a_i x^i$
- $B(x) = \sum_{i=0}^{m} b_i x^i$
- $C(x) = A(x)B(x) = \sum_{k=0}^{2n} c_k x^k$ with

$$c_k = \sum_{0 \le i,j \le n, i+j=k} a_i b_j \, , for\ all\ 0 \le k \le 2n$$

Direct approach:

Direct (Brute Force) Approach

To ease analysis, assume n = m.

- $A(x) = \sum_{i=0}^{n} a_i x^i$
- $B(x) = \sum_{i=0}^{m} b_i x^i$
- $C(x) = A(x)B(x) = \sum_{k=0}^{2n} c_k x^k$ with

$$c_k = \sum_{0 \le i, j \le n, i+j=k} a_i b_j \, , for\ all\ 0 \le k \le 2n$$

Direct approach: Compute all c_k's using the formula above.

Direct (Brute Force) Approach

To ease analysis, assume n = m.

- $A(x) = \sum_{i=0}^{n} a_i x^i$
- $B(x) = \sum_{i=0}^{m} b_i x^i$
- $C(x) = A(x)B(x) = \sum_{k=0}^{2n} c_k x^k$ with

$$c_k = \sum_{0 \le i,j \le n, i+j=k} a_i b_j \,, for\ all\ 0 \le k \le 2n$$

Direct approach: Compute all c_k's using the formula above.

- Total number of multiplications: $O(n^2)$

Direct (Brute Force) Approach

To ease analysis, assume n = m.

- $A(x) = \sum_{i=0}^{n} a_i x^i$
- $B(x) = \sum_{i=0}^{m} b_i x^i$
- $C(x) = A(x)B(x) = \sum_{k=0}^{2n} c_k x^k$ with

$$c_k = \sum_{0 \le i,j \le n, i+j=k} a_i b_j \, , for\ all\ 0 \le k \le 2n$$

Direct approach: Compute all c_k's using the formula above.

- Total number of multiplications: $O(n^2)$
- Total number of additions: $O(n^2)$

Direct (Brute Force) Approach

To ease analysis, assume n = m.

- $A(x) = \sum_{i=0}^{n} a_i x^i$
- $B(x) = \sum_{i=0}^{m} b_i x^i$
- $C(x) = A(x)B(x) = \sum_{k=0}^{2n} c_k x^k$ with

$$c_k = \sum_{0 \le i,j \le n, i+j=k} a_i b_j \,, for\ all\ 0 \le k \le 2n$$

Direct approach: Compute all c_k's using the formula above.

- Total number of multiplications: $O(n^2)$
- Total number of additions: $O(n^2)$
- Complexity: $O(n^2)$

Outline

- Review to Divide-and-Conquer Paradigm
- Polynomial Multiplication Problem
  - Problem definition
  - A brute force algorithm
  - A first divide-and-conquer algorithm
  - An improved divide-and-conquer algorithm
  - Analysis of the divide-and-conquer algorithm
- Quicksort Problem
  - Basic partition
  - Randomized partition and randomized quicksort
  - Analysis of the randomized quicksort

The First Divide-and-Conquer: Divide

Assume n is a power of 2

Define

$$A_0(x) = a_0 + a_1 x + \cdots + a_{\frac{n}{2}-1} x^{\frac{n}{2}-1}$$

$$A_1(\mathrm{x}) = \mathrm{a}_{\frac{\mathrm{n}}{2}} + \mathrm{a}_{\frac{\mathrm{n}}{2}+1}\mathrm{x} + \cdots + \mathrm{a}_{\mathrm{n}}\mathrm{x}^{\frac{\mathrm{n}}{2}}$$

$$A(x) =$$

The First Divide-and-Conquer: Divide

Assume n is a power of 2

Define

$$A_0(x) = a_0 + a_1 x + \cdots + a_{\frac{n}{2}-1} x^{\frac{n}{2}-1}$$

$$A_1(\mathrm{x}) = \mathrm{a}_{\frac{\mathrm{n}}{2}} + \mathrm{a}_{\frac{\mathrm{n}}{2}+1}\mathrm{x} + \cdots + \mathrm{a}_{\mathrm{n}}\mathrm{x}^{\frac{\mathrm{n}}{2}}$$

$$A(x) = A_0(x) +$$

The First Divide-and-Conquer: Divide

Assume n is a power of 2

Define

$$A_0(x) = a_0 + a_1 x + \cdots + a_{\frac{n}{2}-1} x^{\frac{n}{2}-1}$$

$$A_1(\mathrm{x}) = \mathrm{a}_{\frac{\mathrm{n}}{2}} + \mathrm{a}_{\frac{\mathrm{n}}{2}+1}\mathrm{x} + \cdots + \mathrm{a}_{\mathrm{n}}\mathrm{x}^{\frac{\mathrm{n}}{2}}$$

$$A(x) = A_0(x) + A_1(x) x^{\frac{n}{2}}$$

The First Divide-and-Conquer: Divide

Assume n is a power of 2

Define

$$A_0(x) = a_0 + a_1 x + \cdots + a_{\frac{n}{2}-1} x^{\frac{n}{2}-1}$$

$$A_1(\mathrm{x}) = \mathrm{a}_{\frac{\mathrm{n}}{2}} + \mathrm{a}_{\frac{\mathrm{n}}{2}+1}\mathrm{x} + \cdots + \mathrm{a}_{\mathrm{n}}\mathrm{x}^{\frac{\mathrm{n}}{2}}$$

$$A(x) = A_0(x) + A_1(x)x^{\frac{n}{2}}$$

Similarly, define B_0(x) and B_1(x) such that

$$B(x) = B_0(x) + B_1(x)x^{\frac{n}{2}}$$

The First Divide-and-Conquer: Divide

Assume n is a power of 2

Define

$$A_0(x) = a_0 + a_1 x + \cdots + a_{\frac{n}{2}-1} x^{\frac{n}{2}-1}$$

$$A_1(\mathrm{x}) = \mathrm{a}_{\frac{\mathrm{n}}{2}} + \mathrm{a}_{\frac{\mathrm{n}}{2}+1}\mathrm{x} + \cdots + \mathrm{a}_{\mathrm{n}}\mathrm{x}^{\frac{\mathrm{n}}{2}}$$

$$A(x) = A_0(x) + A_1(x) x^{\frac{n}{2}}$$

Similarly, define $B_0(x)$ and $B_1(x)$ such that

$$B(x) = B_0(x) + B_1(x) x^{\frac{n}{2}}$$

$$A(x)B(x) =$$

The First Divide-and-Conquer: Divide

Assume n is a power of 2

Define

$$A_0(x) = a_0 + a_1 x + \cdots + a_{\frac{n}{2}-1} x^{\frac{n}{2}-1}$$

$$A_1(\mathrm{x}) = \mathrm{a}_{\frac{\mathrm{n}}{2}} + \mathrm{a}_{\frac{\mathrm{n}}{2}+1}\mathrm{x} + \cdots + \mathrm{a}_{\mathrm{n}}\mathrm{x}^{\frac{\mathrm{n}}{2}}$$

$$A(x) = A_0(x) + A_1(x) x^{\frac{n}{2}}$$

Similarly, define B_0(x) and B_1(x) such that

$$B(x) = B_0(x) + B_1(x) x^{\frac{n}{2}}$$

$$A(x)B(x) = A_0(x)B_0(x) +$$

The First Divide-and-Conquer: Divide

Assume n is a power of 2

Define

$$A_0(x) = a_0 + a_1 x + \cdots + a_{\frac{n}{2}-1} x^{\frac{n}{2}-1}$$

$$A_1(\mathrm{x}) = \mathrm{a}_{\frac{\mathrm{n}}{2}} + \mathrm{a}_{\frac{\mathrm{n}}{2}+1}\mathrm{x} + \cdots + \mathrm{a}_{\mathrm{n}}\mathrm{x}^{\frac{\mathrm{n}}{2}}$$

$$A(x) = A_0(x) + A_1(x) x^{\frac{n}{2}}$$

Similarly, define $B_0(x)$ and $B_1(x)$ such that

$$B(x) = B_0(x) + B_1(x) x^{\frac{n}{2}}$$

$$A(x)B(x) = A_0(x)B_0(x) + A_0(x)B_1(x)x^{\frac{n}{2}}$$
$$+$$

The First Divide-and-Conquer: Divide

Assume n is a power of 2

Define

$$A_0(x) = a_0 + a_1 x + \cdots + a_{\frac{n}{2}-1} x^{\frac{n}{2}-1}$$

$$A_1(\mathrm{x}) = \mathrm{a}_{\frac{\mathrm{n}}{2}} + \mathrm{a}_{\frac{\mathrm{n}}{2}+1}\mathrm{x} + \cdots + \mathrm{a}_{\mathrm{n}}\mathrm{x}^{\frac{\mathrm{n}}{2}}$$

$$A(x) = A_0(x) + A_1(x) x^{\frac{n}{2}}$$

Similarly, define $B_0(x)$ and $B_1(x)$ such that

$$B(x) = B_0(x) + B_1(x) x^{\frac{n}{2}}$$

$$A(x)B(x) = A_0(x)B_0(x) + A_0(x)B_1(x)x^{\frac{n}{2}}$$

$$+A_1(x)B_0(x)x^{\frac{n}{2}} +$$

The First Divide-and-Conquer: Divide

Assume n is a power of 2

Define

$$A_0(x) = a_0 + a_1 x + \cdots + a_{\frac{n}{2}-1} x^{\frac{n}{2}-1}$$

$$A_1(\mathrm{x}) = \mathrm{a}_{\frac{\mathrm{n}}{2}} + \mathrm{a}_{\frac{\mathrm{n}}{2}+1}\mathrm{x} + \cdots + \mathrm{a}_{\mathrm{n}}\mathrm{x}^{\frac{\mathrm{n}}{2}}$$

$$A(x) = A_0(x) + A_1(x)x^{\frac{n}{2}}$$

Similarly, define $B_0(x)$ and $B_1(x)$ such that

$$B(x) = B_0(x) + B_1(x)x^{\frac{n}{2}}$$

$$A(x)B(x) = A_0(x)B_0(x) + A_0(x)B_1(x)x^{\frac{n}{2}} + A_1(x)B_0(x)x^{\frac{n}{2}} + A_1(x)B_1(x)x^n$$

The First Divide-and-Conquer: Divide

Assume n is a power of 2

Define

$$A_0(x) = a_0 + a_1 x + \cdots + a_{\frac{n}{2}-1} x^{\frac{n}{2}-1}$$

$$A_1(\mathrm{x}) = \mathrm{a}_{\frac{\mathrm{n}}{2}} + \mathrm{a}_{\frac{\mathrm{n}}{2}+1}\mathrm{x} + \cdots + \mathrm{a}_{\mathrm{n}}\mathrm{x}^{\frac{\mathrm{n}}{2}}$$

$$A(x) = A_0(x) + A_1(x) x^{\frac{n}{2}}$$

Similarly, define $B_0(x)$ and $B_1(x)$ such that

$$B(x) = B_0(x) + B_1(x) x^{\frac{n}{2}}$$

$$A(x)B(x) = A_0(x)B_0(x) + A_0(x)B_1(x)x^{\frac{n}{2}} + A_1(x)B_0(x)x^{\frac{n}{2}} + A_1(x)B_1(x)x^n$$

The original problem (of size n) is divided into 4 problems of input size n/2

Example

$$A(x) = 2 + 5x + 3x^2 + x^3 - x^4$$
$$B(x) = 1 + 2x + 2x^2 + 3x^3 + 6x^4$$
$$A(x)B(x) = 2 + 9x + 17x^2 + 23x^3 + 34x^4$$
$$+39x^5 + 19x^6 + 3\mathrm{x}^7 - 6\mathrm{x}^8$$
$$A_0(x) = 2 + 5x, A_1(x) = 3 + x - x^2$$
$$A(x) = A_0(x) + A_1(x)x^2$$

Example

$$A(x) = 2 + 5x + 3x^2 + x^3 - x^4$$
$$B(x) = 1 + 2x + 2x^2 + 3x^3 + 6x^4$$
$$A(x)B(x) = 2 + 9x + 17x^2 + 23x^3 + 34x^4$$
$$+39x^5 + 19x^6 + 3\mathrm{x}^7 - 6\mathrm{x}^8$$
$$A_0(x) = 2 + 5x, A_1(x) = 3 + x - x^2$$
$$A(x) = A_0(x) + A_1(x)x^2$$
$$B_0(x) = 1 + 2x, B_1(x) = 2 + 3x + 6x^2$$
$$B(x) = B_0(x) + B_1(x)x^2$$

Example

$$A(x) = 2 + 5x + 3x^2 + x^3 - x^4$$
$$B(x) = 1 + 2x + 2x^2 + 3x^3 + 6x^4$$
$$A(x)B(x) = 2 + 9x + 17x^2 + 23x^3 + 34x^4$$
$$+39x^5 + 19x^6 + 3\mathrm{x}^7 - 6\mathrm{x}^8$$
$$A_0(x) = 2 + 5x, A_1(x) = 3 + x - x^2$$
$$A(x) = A_0(x) + A_1(x)x^2$$
$$B_0(x) = 1 + 2x, B_1(x) = 2 + 3x + 6x^2$$
$$B(x) = B_0(x) + B_1(x)x^2$$
$$A_0(x)B_0(x) = 2 + 9x + 10x^2$$
$$A_1(x)B_1(x) = 6 + 11x + 19x^2 + 3x^3 - 6x^4$$
$$A_0(x)B_1(x) = 4 + 16x + 27x^2 + 30x^3$$
$$A_1(x)B_0(x) = 3 + 7x + x^2 - 2x^3$$

Example

$$A(x) = 2 + 5x + 3x^2 + x^3 - x^4$$
$$B(x) = 1 + 2x + 2x^2 + 3x^3 + 6x^4$$
$$A(x)B(x) = 2 + 9x + 17x^2 + 23x^3 + 34x^4$$
$$+39x^5 + 19x^6 + 3\mathrm{x}^7 - 6\mathrm{x}^8$$
$$A_0(x) = 2 + 5x, A_1(x) = 3 + x - x^2$$
$$A(x) = A_0(x) + A_1(x)x^2$$
$$B_0(x) = 1 + 2x, B_1(x) = 2 + 3x + 6x^2$$
$$B(x) = B_0(x) + B_1(x)x^2$$
$$A_0(x)B_0(x) = 2 + 9x + 10x^2$$
$$A_1(x)B_1(x) = 6 + 11x + 19x^2 + 3x^3 - 6x^4$$
$$A_0(x)B_1(x) = 4 + 16x + 27x^2 + 30x^3$$
$$A_1(x)B_0(x) = 3 + 7x + x^2 - 2x^3$$
$$A_0(x)B_1(x) + A_1(x)B_0(x) = 7 + 23x + 28x^2 + 28x^3$$

$$A_0(x)B_0(x) + \big(A_0(x)B_1(x) + A_1(x)B_0(x)\big)x^2 + A_1(x)B_1(x)x^4$$

Example

$$A(x) = 2 + 5x + 3x^2 + x^3 - x^4$$
$$B(x) = 1 + 2x + 2x^2 + 3x^3 + 6x^4$$
$$A(x)B(x) = 2 + 9x + 17x^2 + 23x^3 + 34x^4$$
$$+39x^5 + 19x^6 + 3\mathrm{x}^7 - 6\mathrm{x}^8$$
$$A_0(x) = 2 + 5x, A_1(x) = 3 + x - x^2$$
$$A(x) = A_0(x) + A_1(x)x^2$$
$$B_0(x) = 1 + 2x, B_1(x) = 2 + 3x + 6x^2$$
$$B(x) = B_0(x) + B_1(x)x^2$$
$$A_0(x)B_0(x) = 2 + 9x + 10x^2$$
$$A_1(x)B_1(x) = 6 + 11x + 19x^2 + 3x^3 - 6x^4$$
$$A_0(x)B_1(x) = 4 + 16x + 27x^2 + 30x^3$$
$$A_1(x)B_0(x) = 3 + 7x + x^2 - 2x^3$$
$$A_0(x)B_1(x) + A_1(x)B_0(x) = 7 + 23x + 28x^2 + 28x^3$$

$$A_0(x)B_0(x) + \big(A_0(x)B_1(x) + A_1(x)B_0(x)\big)x^2 + A_1(x)B_1(x)x^4$$
$$= 2 + 9x + 17x^2 + 23x^3 + 34x^4 + 39x^5 + 19x^6 + 3x^7 - 6x^8$$

The First Divide-and-Conquer: Conquer

Conquer: Solve the four subproblems

- Compute $A_0(x)B_0(x), A_0(x)B_1(x), A_1(x)B_0(x), \mathrm{A}_1(\mathrm{x})\mathrm{B}_1(\mathrm{x})$

The First Divide-and-Conquer: Conquer

Conquer: Solve the four subproblems

- Compute $A_0(x)B_0(x), A_0(x)B_1(x), A_1(x)B_0(x), \mathrm{A}_1(\mathrm{x})\mathrm{B}_1(\mathrm{x})$ by recursively calling the algorithm 4 times

The First Divide-and-Conquer: Conquer

Conquer: Solve the four subproblems

- Compute
  $A_0(x)B_0(x), A_0(x)B_1(x), A_1(x)B_0(x), \mathrm{A}_1(\mathrm{x})\mathrm{B}_1(\mathrm{x})$
  by recursively calling the algorithm 4 times

Combine

The First Divide-and-Conquer: Conquer

Conquer: Solve the four subproblems

- Compute $A_0(x)B_0(x), A_0(x)B_1(x), A_1(x)B_0(x), \mathrm{A}_1(\mathrm{x})\mathrm{B}_1(\mathrm{x})$ by recursively calling the algorithm 4 times

Combine

- Add the following four polynomials

$$A_0(x)B_0(x) + A_0(x)B_1(x)x^{\frac{n}{2}} + A_1(x)B_0(x)x^{\frac{n}{2}} + A_1(x)B_1(x)x^n$$

The First Divide-and-Conquer: Conquer

Conquer: Solve the four subproblems

- Compute $A_0(x)B_0(x), A_0(x)B_1(x), A_1(x)B_0(x), \mathrm{A}_1(\mathrm{x})\mathrm{B}_1(\mathrm{x})$ by recursively calling the algorithm 4 times

Combine

- Add the following four polynomials

$$A_0(x)B_0(x) + A_0(x)B_1(x)x^{\frac{n}{2}} + A_1(x)B_0(x)x^{\frac{n}{2}} + A_1(x)B_1(x)x^n$$

- Takes O( ) operations

The First Divide-and-Conquer: Conquer

Conquer: Solve the four subproblems

- Compute $A_0(x)B_0(x), A_0(x)B_1(x), A_1(x)B_0(x), \mathrm{A}_1(\mathrm{x})\mathrm{B}_1(\mathrm{x})$ by recursively calling the algorithm 4 times

Combine

- Add the following four polynomials

$$A_0(x)B_0(x) + A_0(x)B_1(x)x^{\frac{n}{2}} + A_1(x)B_0(x)x^{\frac{n}{2}} + A_1(x)B_1(x)x^n$$

- Takes O(n) operations

The First Divide-and-Conquer Algorithm

$\mathrm{PolyMulti1}(A(x), B(x))$

Input: $A(x), B(x)$
Output: $A(x) \times B(x)$
$A_0(x) \leftarrow a_0 + a_1 x + \cdots + a_{\frac{n}{2}-1} x^{\frac{n}{2}-1}$;
$A_1(x) \leftarrow a_{\frac{n}{2}} + a_{\frac{n}{2}+1} x + \cdots + a_n x^{\frac{n}{2}}$;
$B_0(x) \leftarrow b_0 + b_1 x + \cdots + b_{\frac{n}{2}-1} x^{\frac{n}{2}-1}$;
$B_1(x) \leftarrow b_{\frac{n}{2}} + b_{\frac{n}{2}+1} x + \cdots + b_n x^{\frac{n}{2}}$;

The First Divide-and-Conquer Algorithm

$\text{PolyMulti1}(A(x), B(x))$

Input: $A(x), B(x)$
Output: $A(x) \times B(x)$
$A_0(x) \leftarrow a_0 + a_1 x + \cdots + a_{\frac{n}{2}-1} x^{\frac{n}{2}-1}$;
$A_1(x) \leftarrow a_{\frac{n}{2}} + a_{\frac{n}{2}+1} x + \cdots + a_n x^{\frac{n}{2}}$;
$B_0(x) \leftarrow b_0 + b_1 x + \cdots + b_{\frac{n}{2}-1} x^{\frac{n}{2}-1}$;
$B_1(x) \leftarrow b_{\frac{n}{2}} + b_{\frac{n}{2}+1} x + \cdots + b_n x^{\frac{n}{2}}$;
$U(x) \leftarrow \text{PolyMulti1}(A_0(x), B_0(x))$; $//T(n/2)$
$V(x) \leftarrow \text{PolyMulti1}(A_0(x), B_1(x))$; $//T(n/2)$
$W(x) \leftarrow \text{PolyMulti1}(A_1(x), B_0(x))$; $//T(n/2)$
$Z(x) \leftarrow \text{PolyMulti1}(A_1(x), B_1(x))$; $//T(n/2)$

The First Divide-and-Conquer Algorithm

$\text{PolyMulti1}(A(x), B(x))$

Input: $A(x), B(x)$
Output: $A(x) \times B(x)$
$A_0(x) \leftarrow a_0 + a_1 x + \cdots + a_{\frac{n}{2}-1} x^{\frac{n}{2}-1}$;
$A_1(x) \leftarrow a_{\frac{n}{2}} + a_{\frac{n}{2}+1} x + \cdots + a_n x^{\frac{n}{2}}$;
$B_0(x) \leftarrow b_0 + b_1 x + \cdots + b_{\frac{n}{2}-1} x^{\frac{n}{2}-1}$;
$B_1(x) \leftarrow b_{\frac{n}{2}} + b_{\frac{n}{2}+1} x + \cdots + b_n x^{\frac{n}{2}}$;
$U(x) \leftarrow \text{PolyMulti1}(A_0(x), B_0(x))$; $//T(n/2)$
$V(x) \leftarrow \text{PolyMulti1}(A_0(x), B_1(x))$; $//T(n/2)$
$W(x) \leftarrow \text{PolyMulti1}(A_1(x), B_0(x))$; $//T(n/2)$
$Z(x) \leftarrow \text{PolyMulti1}(A_1(x), B_1(x))$; $//T(n/2)$
return $(U(x) + [V(x) + W(x)]x^{\frac{n}{2}} + Z(x)x^n)$; $//O(n)$

The First Divide-and-Conquer Algorithm

PolyMulti1$(A(x), B(x))$

Input: $A(x), B(x)$
Output: $A(x) \times B(x)$
$A_0(x) \leftarrow a_0 + a_1 x + \cdots + a_{\frac{n}{2}-1} x^{\frac{n}{2}-1}$;
$A_1(x) \leftarrow a_{\frac{n}{2}} + a_{\frac{n}{2}+1} x + \cdots + a_n x^{\frac{n}{2}}$;
$B_0(x) \leftarrow b_0 + b_1 x + \cdots + b_{\frac{n}{2}-1} x^{\frac{n}{2}-1}$;
$B_1(x) \leftarrow b_{\frac{n}{2}} + b_{\frac{n}{2}+1} x + \cdots + b_n x^{\frac{n}{2}}$;
$U(x) \leftarrow$PolyMulti1$(A_0(x), B_0(x))$;$//T(n/2)$
$V(x) \leftarrow$PolyMulti1$(A_0(x), B_1(x))$;$//T(n/2)$
$W(x) \leftarrow$PolyMulti1$(A_1(x), B_0(x))$;$//T(n/2)$
$Z(x) \leftarrow$PolyMulti1$(A_1(x), B_1(x))$;$//T(n/2)$
return $(U(x) + [V(x) + W(x)]x^{\frac{n}{2}} + Z(x)x^n)$;$//O(n)$

$$T(n) = \begin{cases} 4T(n/2) + n, & \text{if } n > 1, \\ 1, & \text{if } n = 1. \end{cases}$$

Analysis of Running Time

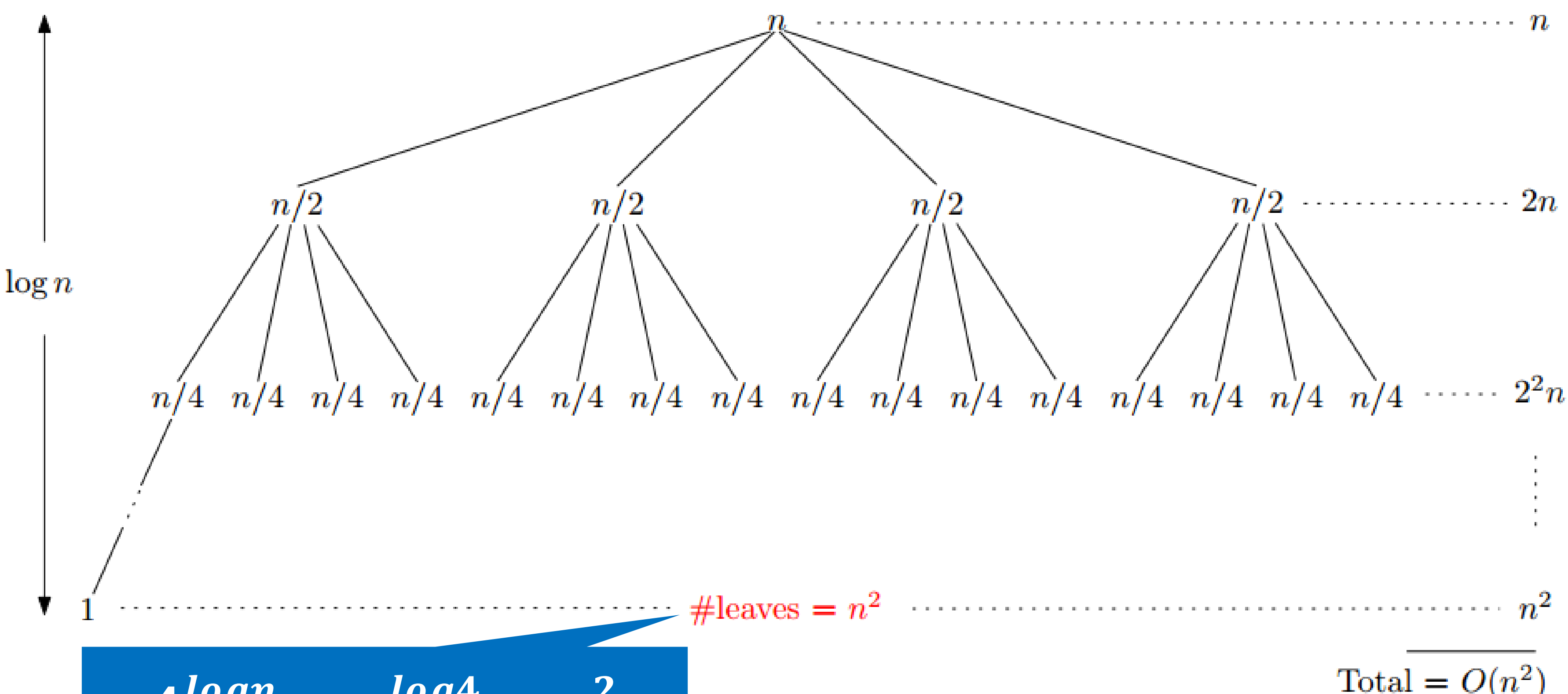
Assume that n is a power of 2

$$T(n) = \begin{cases} 4T(n/2) + n, & \text{if } n > 1, \\ 1, & \text{if } n = 1. \end{cases}$$

Analysis of Running Time

Assume that n is a power of 2

$$T(n) = \begin{cases} 4T(n/2) + n, & \text{if } n > 1, \\ 1, & \text{if } n = 1. \end{cases}$$

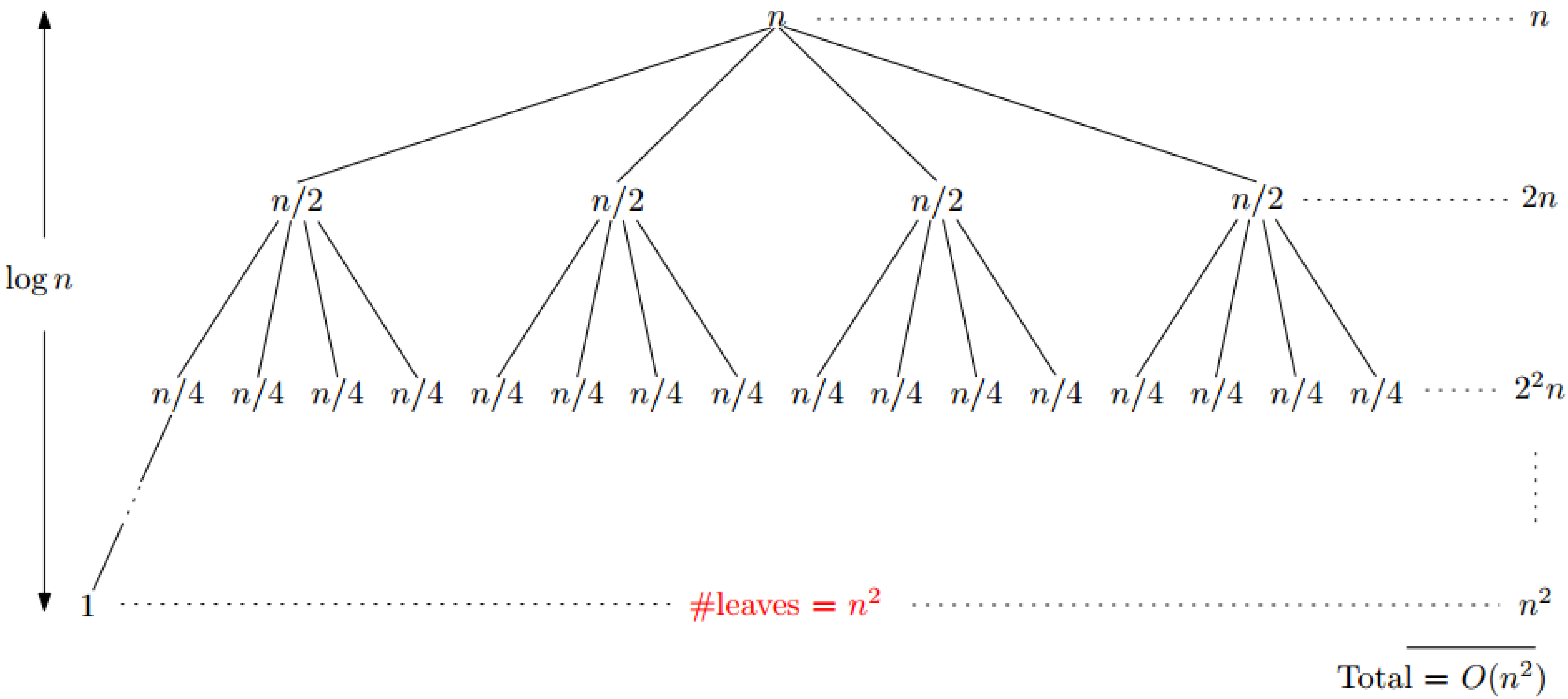


$$4^{\log n} = n^{\log 4} = n^2$$

Analysis of Running Time

Assume that n is a power of 2

$$T(n) = \begin{cases} 4T(n/2) + n, & \text{if } n > 1, \\ 1, & \text{if } n = 1. \end{cases}$$

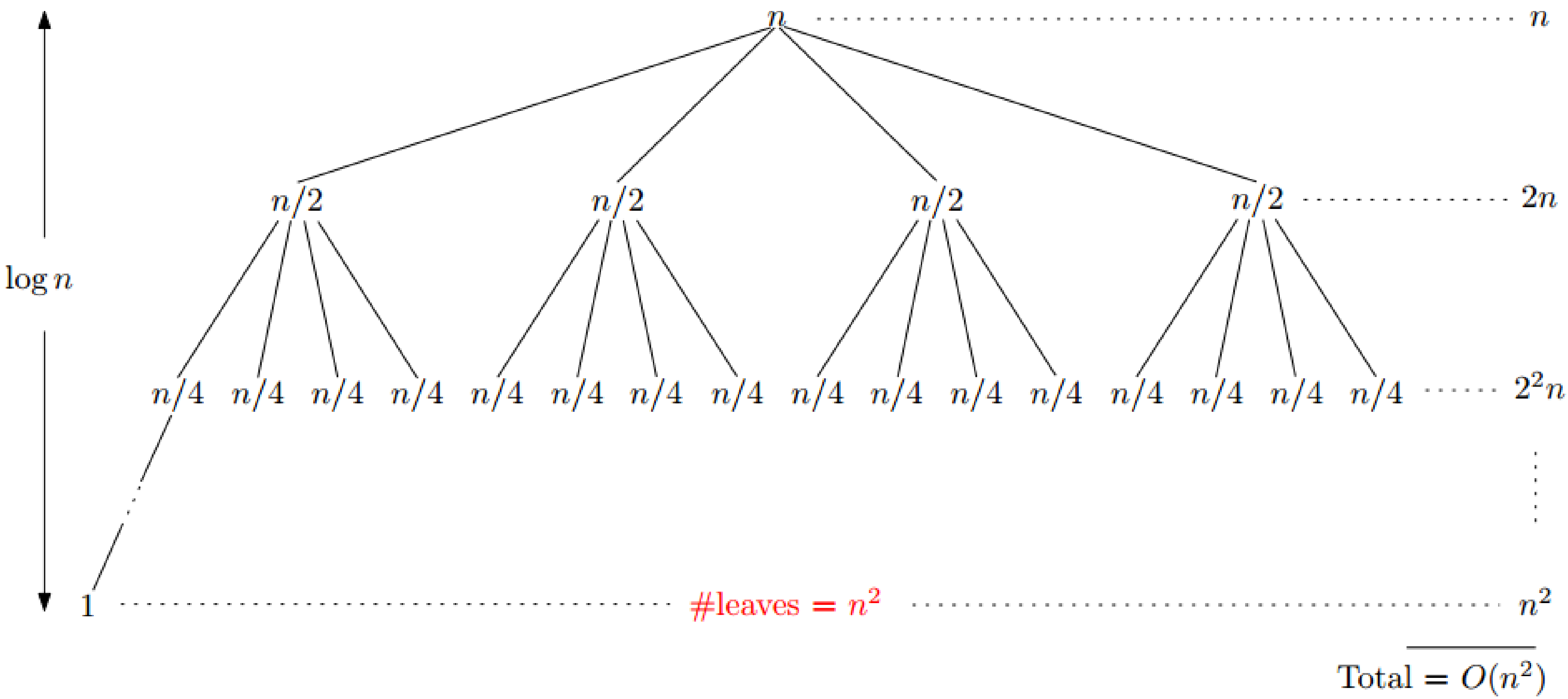


Same order as the brute force approach!

Analysis of Running Time

Assume that n is a power of 2

$$T(n) = \begin{cases} 4T(n/2) + n, & \text{if } n > 1, \\ 1, & \text{if } n = 1. \end{cases}$$



Same order as the brute force approach! No improvement!

Outline

- Review to Divide-and-Conquer Paradigm
- Polynomial Multiplication Problem
  - Problem definition
  - A brute force algorithm
  - A first divide-and-conquer algorithm
  - An improved divide-and-conquer algorithm
  - Analysis of the divide-and-conquer algorithm
- Quicksort Problem
  - Basic partition
  - Randomized partition and randomized quicksort
  - Analysis of the randomized quicksort

Two Observations

Observation 1:

What we really need are the following 3 terms:

$$A_0 B_0,\ A_0 B_1 + A_1 B_0,\ A_1 B_1?$$

Instead of the following 4 terms:

$$A_0 B_0,\ A_0 B_1,\ A_1 B_0,\ A_1 B_1?$$

Two Observations

Observation 1:

What we really need are the following 3 terms:

$$A_0B_0,\ A_0B_1 + A_1B_0,\ A_1B_1?$$

Instead of the following 4 terms:

$$A_0B_0,\ A_0B_1,\ A_1B_0,\ A_1B_1?$$

Observation 2:

The three terms can be obtained using only 3 multiplications:

Two Observations

Observation 1:

What we really need are the following 3 terms:

$$A_0B_0,\ A_0B_1 + A_1B_0,\ A_1B_1?$$

Instead of the following 4 terms:

$$A_0B_0,\ A_0B_1,\ A_1B_0,\ A_1B_1?$$

Observation 2:

The three terms can be obtained using only 3 multiplications:

$$\begin{aligned} Y &= (A_0 + A_1)(B_0 + B_1) \\ U &= A_0B_0 \\ Z &= A_1B_1 \end{aligned}$$

Two Observations

Observation 1:

What we really need are the following 3 terms:

$$A_0B_0, \; A_0B_1 + A_1B_0, \; A_1B_1?$$

Instead of the following 4 terms:

$$A_0B_0, \; A_0B_1, \; A_1B_0, \; A_1B_1?$$

Observation 2:

The three terms can be obtained using only 3 multiplications:

$$\begin{aligned} Y &= (A_0 + A_1)(B_0 + B_1) \\ U &= A_0B_0 \\ Z &= A_1B_1 \end{aligned}$$

- *U and Z are what we originally wanted*

Two Observations

Observation 1:

What we really need are the following 3 terms:

$$A_0 B_0,\ A_0 B_1 + A_1 B_0,\ A_1 B_1?$$

Instead of the following 4 terms:

$$A_0 B_0,\ A_0 B_1,\ A_1 B_0,\ A_1 B_1?$$

Observation 2:

The three terms can be obtained using only 3 multiplications:

$$\begin{aligned} Y &= (A_0 + A_1)(B_0 + B_1) \\ U &= A_0 B_0 \\ Z &= A_1 B_1 \end{aligned}$$

- *U and Z are what we originally wanted*
- $A_0 B_1 + A_1 B_0 =$

Two Observations

Observation 1:

What we really need are the following 3 terms:

$$A_0B_0, \ A_0B_1 + A_1B_0, \ A_1B_1?$$

Instead of the following 4 terms:

$$A_0B_0, \ A_0B_1, \ A_1B_0, \ A_1B_1?$$

Observation 2:

The three terms can be obtained using only 3 multiplications:

$$\begin{aligned} Y &= (A_0 + A_1)(B_0 + B_1) \\ U &= A_0B_0 \\ Z &= A_1B_1 \end{aligned}$$

- *U and Z are what we originally wanted*
- $A_0B_1 + A_1B_0 = Y - U - Z$

The improved Divide-and-Conquer Algorithm

PolyMulti2$(A(x), B(x))$

Input: $A(x), B(x)$
Output: $A(x) \times B(x)$
$A_0(x) \leftarrow a_0 + a_1 x + \cdots + a_{\frac{n}{2}-1} x^{\frac{n}{2}-1};$
$A_1(x) \leftarrow a_{\frac{n}{2}} + a_{\frac{n}{2}+1} x + \cdots + a_n x^{n-\frac{n}{2}};$
$B_0(x) \leftarrow b_0 + b_1 x + \cdots + b_{\frac{n}{2}-1} x^{\frac{n}{2}-1};$
$B_1(x) \leftarrow b_{\frac{n}{2}} + b_{\frac{n}{2}+1} x + \cdots + b_n x^{n-\frac{n}{2}};$

The improved Divide-and-Conquer Algorithm

PolyMulti2$(A(x), B(x))$

Input: $A(x), B(x)$
Output: $A(x) \times B(x)$
$A_0(x) \leftarrow a_0 + a_1 x + \cdots + a_{\frac{n}{2}-1} x^{\frac{n}{2}-1}$;
$A_1(x) \leftarrow a_{\frac{n}{2}} + a_{\frac{n}{2}+1} x + \cdots + a_n x^{n-\frac{n}{2}}$;
$B_0(x) \leftarrow b_0 + b_1 x + \cdots + b_{\frac{n}{2}-1} x^{\frac{n}{2}-1}$;
$B_1(x) \leftarrow b_{\frac{n}{2}} + b_{\frac{n}{2}+1} x + \cdots + b_n x^{n-\frac{n}{2}}$;
$Y(x) \leftarrow$PolyMulti2$(A_0(x) + A_1(x), B_0(x) + B_1(x))$;$//T(n/2)$
$U(x) \leftarrow$PolyMulti2$(A_0(x), B_0(x))$;$//T(n/2)$
$Z(x) \leftarrow$PolyMulti2$(A_1(x), B_1(x))$;$//T(n/2)$

The Second Divide-and-Conquer Algorithm

PolyMulti2$(A(x), B(x))$

Input: $A(x), B(x)$
Output: $A(x) \times B(x)$
$A_0(x) \leftarrow a_0 + a_1 x + \cdots + a_{\frac{n}{2}-1} x^{\frac{n}{2}-1}$;
$A_1(x) \leftarrow a_{\frac{n}{2}} + a_{\frac{n}{2}+1} x + \cdots + a_n x^{n-\frac{n}{2}}$;
$B_0(x) \leftarrow b_0 + b_1 x + \cdots + b_{\frac{n}{2}-1} x^{\frac{n}{2}-1}$;
$B_1(x) \leftarrow b_{\frac{n}{2}} + b_{\frac{n}{2}+1} x + \cdots + b_n x^{n-\frac{n}{2}}$;
$Y(x) \leftarrow$PolyMulti2$(A_0(x) + A_1(x), B_0(x) + B_1(x))$;$//T(n/2)$
$U(x) \leftarrow$PolyMulti2$(A_0(x), B_0(x))$;$//T(n/2)$
$Z(x) \leftarrow$PolyMulti2$(A_1(x), B_1(x))$;$//T(n/2)$
return $(U(x) + [Y(x) - U(x) - Z(x)]x^{\frac{n}{2}} + Z(x)x^{2\frac{n}{2}})$;$//O(n)$

The improved Divide-and-Conquer Algorithm

PolyMulti2($A(x), B(x)$)

Input: $A(x), B(x)$
Output: $A(x) \times B(x)$
$A_0(x) \leftarrow a_0 + a_1 x + \cdots + a_{\frac{n}{2}-1} x^{\frac{n}{2}-1}$;
$A_1(x) \leftarrow a_{\frac{n}{2}} + a_{\frac{n}{2}+1} x + \cdots + a_n x^{n-\frac{n}{2}}$;
$B_0(x) \leftarrow b_0 + b_1 x + \cdots + b_{\frac{n}{2}-1} x^{\frac{n}{2}-1}$;
$B_1(x) \leftarrow b_{\frac{n}{2}} + b_{\frac{n}{2}+1} x + \cdots + b_n x^{n-\frac{n}{2}}$;
$Y(x) \leftarrow$PolyMulti2($A_0(x) + A_1(x), B_0(x) + B_1(x)$);$//T(n/2)$
$U(x) \leftarrow$PolyMulti2($A_0(x), B_0(x)$);$//T(n/2)$
$Z(x) \leftarrow$PolyMulti2($A_1(x), B_1(x)$);$//T(n/2)$
return $(U(x) + [Y(x) - U(x) - Z(x)]x^{\frac{n}{2}} + Z(x)x^{2\frac{n}{2}})$;$//O(n)$

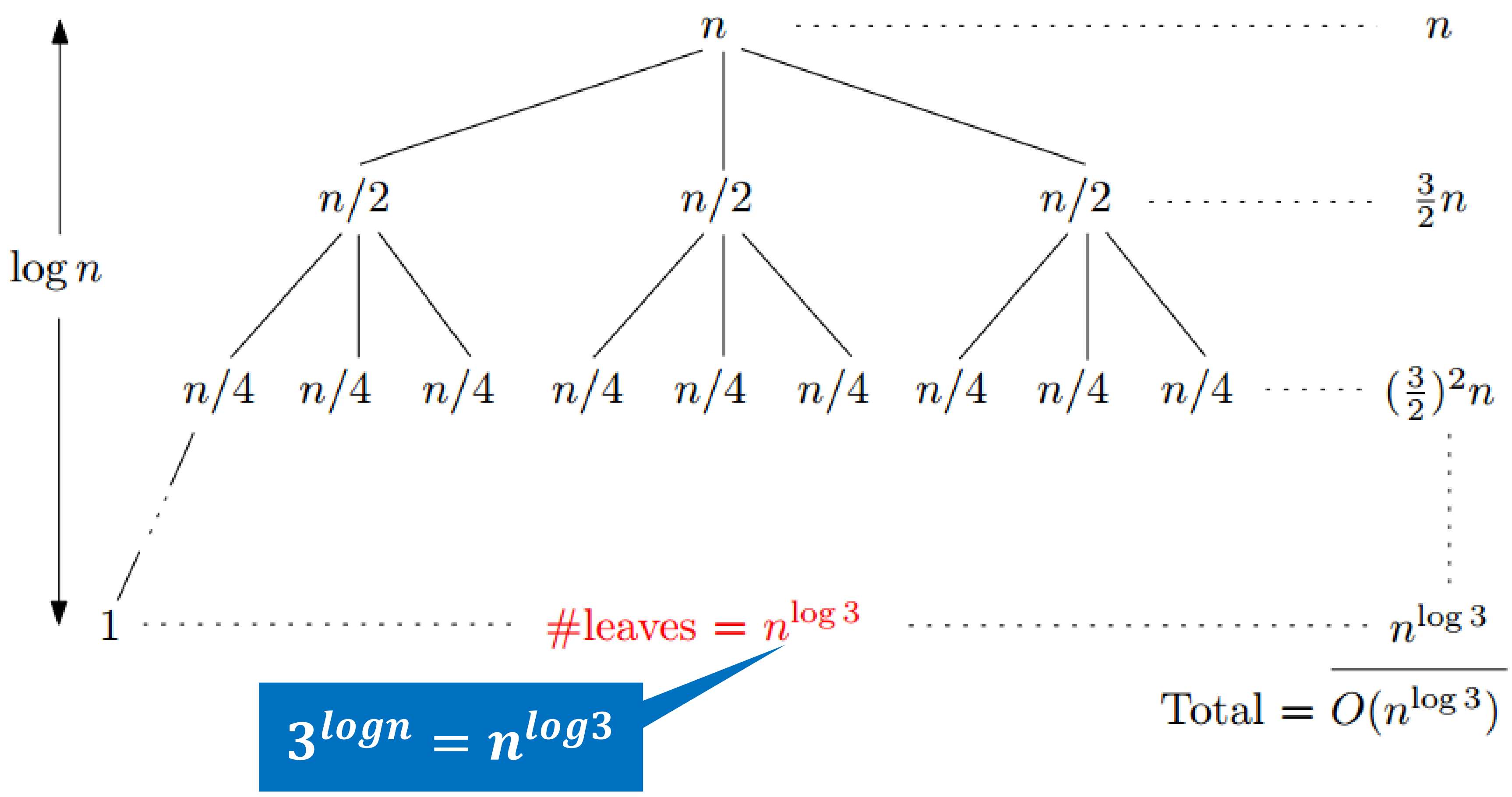
$$T(n) = \begin{cases} 3T(n/2) + n, & \text{if } n > 1, \\ 1, & \text{if } n = 1. \end{cases}$$

Running Time of the Improved Algorithm

$$T(n) = \begin{cases} 3T(n/2) + n, & \text{if } n > 1, \\ 1, & \text{if } n = 1. \end{cases}$$

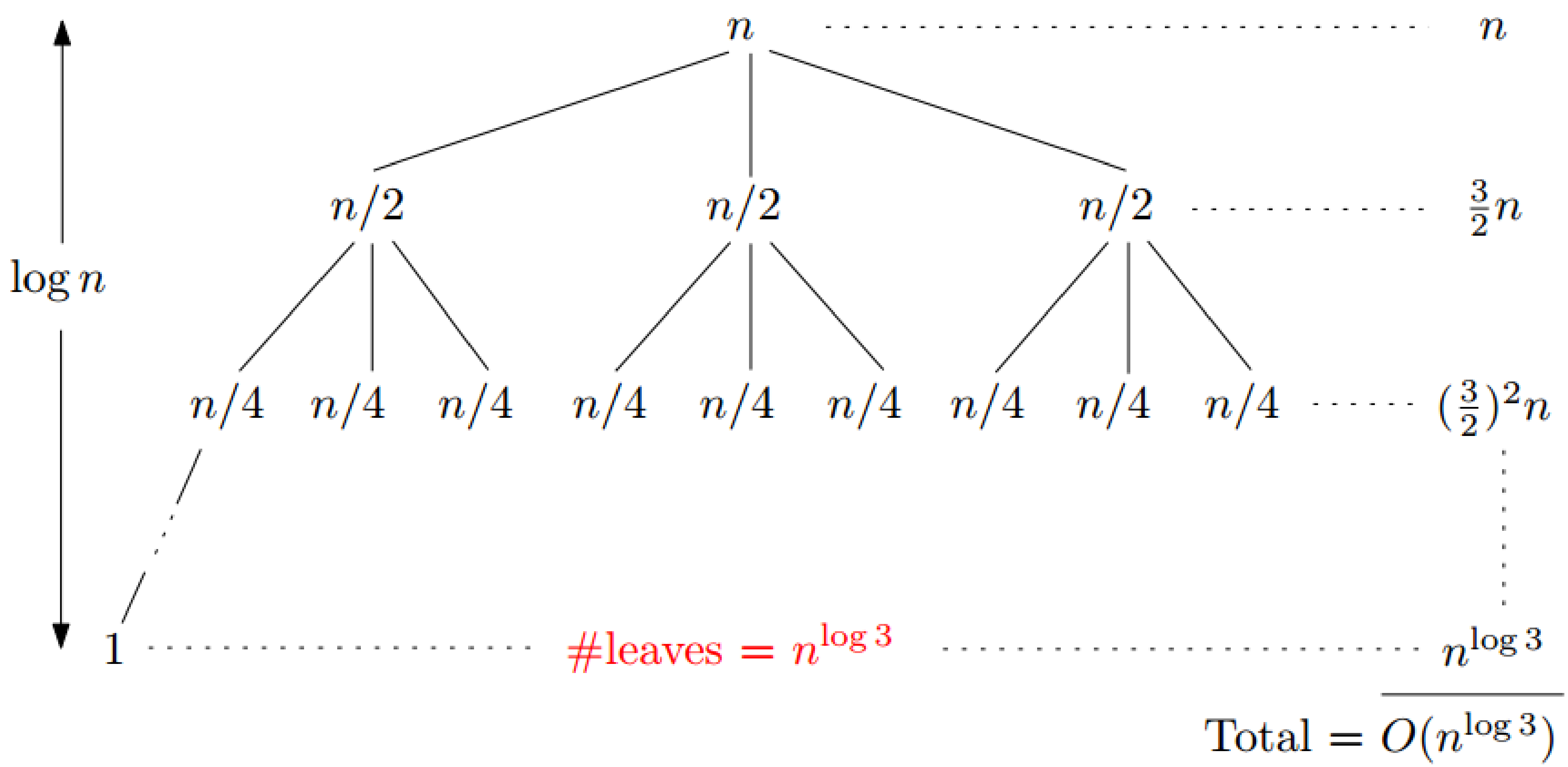
Running Time of the Improved Algorithm

$$T(n) = \begin{cases} 3T(n/2) + n, & \text{if } n > 1, \\ 1, & \text{if } n = 1. \end{cases}$$



Running Time of the Improved Algorithm

$$T(n) = \begin{cases} 3T(n/2) + n, & \text{if } n > 1, \\ 1, & \text{if } n = 1. \end{cases}$$



The second method is much better!

Outline

- Review to Divide-and-Conquer Paradigm
- Polynomial Multiplication Problem
  - Problem definition
  - A brute force algorithm
  - A first divide-and-conquer algorithm
  - An improved divide-and-conquer algorithm
  - Analysis of the divide-and-conquer algorithm
- Quicksort Problem
  - Basic partition
  - Randomized partition and randomized quicksort
  - Analysis of the randomized quicksort

Analysis of the D&C algorithm

- The divide-and-conquer approach does not always give you the best solution
  - Our original algorithm was just as bad as brute force

Analysis of the D&C algorithm

- The divide-and-conquer approach does not always give you the best solution
  - Our original algorithm was just as bad as brute force
- There is actually an O(n log n) solution to the polynomial multiplication problem

Analysis of the D&C algorithm

- The divide-and-conquer approach does not always give you the best solution
  - Our original algorithm was just as bad as brute force
- There is actually an $O(n \log n)$ solution to the polynomial multiplication problem
  - It involves using the Fast Fourier Transform algorithm as a subroutine
  - The FFT is another classic divide-and-conquer algorithm(check Chapt 30 in CLRS if interested)

Analysis of the D&C algorithm

- The divide-and-conquer approach does not always give you the best solution
  - Our original algorithm was just as bad as brute force
- There is actually an O(n log n) solution to the polynomial multiplication problem
  - It involves using the Fast Fourier Transform algorithm as a subroutine
  - The FFT is another classic divide-and-conquer algorithm(check Chapt 30 in CLRS if interested)
- The idea of using 3 multiplications instead of 4 is used in large-integer multiplications
  - A similar idea is the basis of the classic Strassen matrix multiplication algorithm (CLRS 4.2)

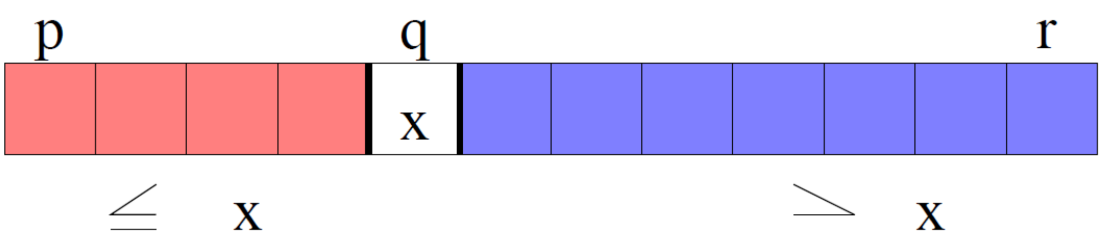
Outline

- Review to Divide-and-Conquer Paradigm
- Polynomial Multiplication Problem
  - Problem definition
  - A brute force algorithm
  - A first divide-and-conquer algorithm
  - An improved divide-and-conquer algorithm
  - Analysis of the divide-and-conquer algorithm
- Quicksort Problem
  - Basic partition
  - Randomized partition and randomized quicksort
  - Analysis of the randomized quicksort

Partition

- Partition
  - Given: An array of numbers
  - Partition: Rearrange the array A[p..r] in place into two (possibly empty) subarrays A[p..q-1] and A[q+1..r] such that

$$A[u] < A[q] < A[v] \quad for\ any\ p \le u \le q-1\ and\ q+1 \le v \le r$$

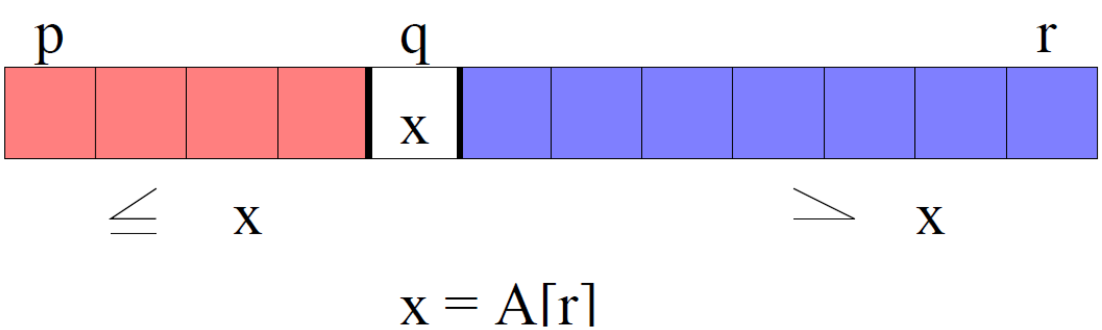


x = A[r]

Partition

- Partition
  - Given: An array of numbers
  - Partition: Rearrange the array A[p..r] in place into two (possibly empty) subarrays A[p..q-1] and A[q+1..r] such that

$$A[u] < \ A[q] < A[v] \ \ for\ any\ p \leq u \leq q-1\ and\ q+1 \leq v \leq r$$

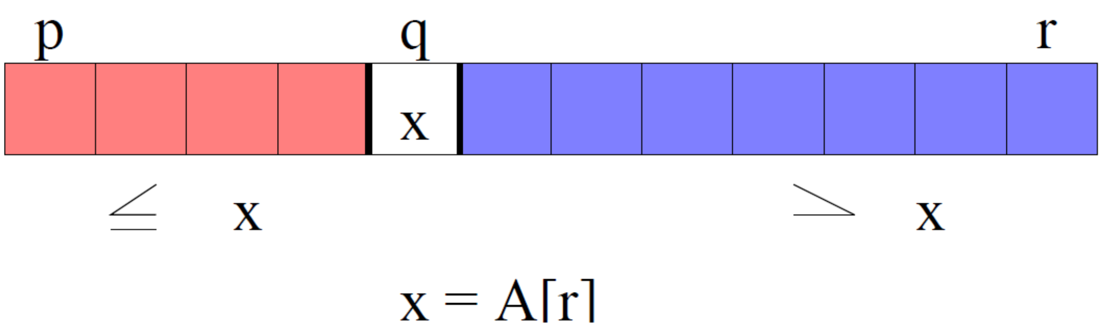


  - x is called the pivot. Assume x = A[r]; if not, swap first

Partition

- Partition
  - Given: An array of numbers
  - Partition: Rearrange the array A[p..r] in place into two (possibly empty) subarrays A[p..q-1] and A[q+1..r] such that

$$A[u] < A[q] < A[v] \ \ for\ any\ p \le u \le q-1\ and\ q+1 \le v \le r$$

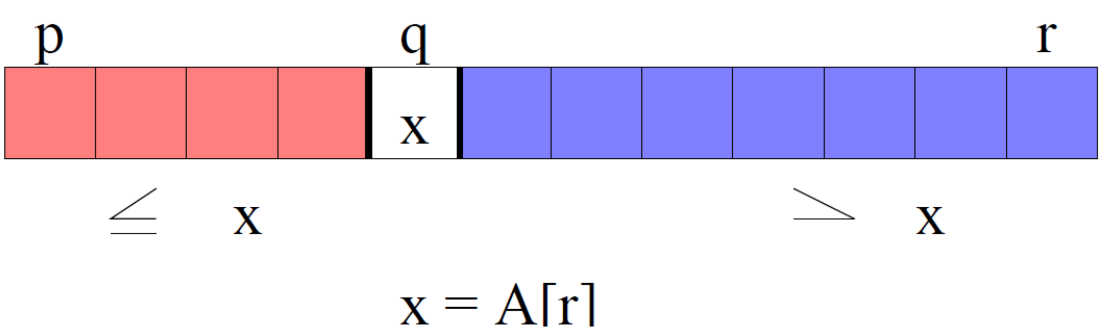


  - x is called the pivot. Assume x = A[r]; if not, swap first
  - Quicksort works by:

Partition

- Partition
  - Given: An array of numbers
  - Partition: Rearrange the array A[p..r] in place into two (possibly empty) subarrays A[p..q-1] and A[q+1..r] such that

$$\color{red}{A[u]} < A[q] < \color{blue}{A[v]} \;\; for\ any\ p \le u \le q-1\ and\ q+1 \le v \le r$$

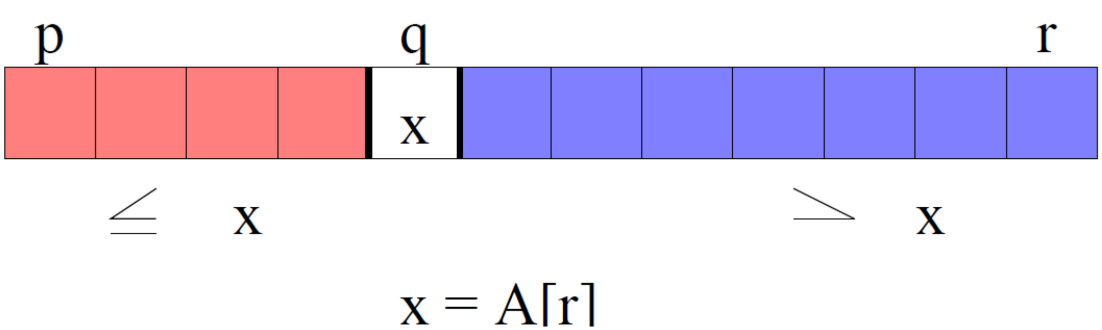


  - x is called the pivot. Assume x = A[r]; if not, swap first
  - Quicksort works by:
    - calling partition first

Partition

- Partition
  - Given: An array of numbers
  - Partition: Rearrange the array A[p..r] in place into two (possibly empty) subarrays A[p..q-1] and A[q+1..r] such that

$$\color{red}{A[u]} < A[q] < \color{blue}{A[v]} \;\; for\ any\ p \leq u \leq q-1\ and\ q+1 \leq v \leq r$$

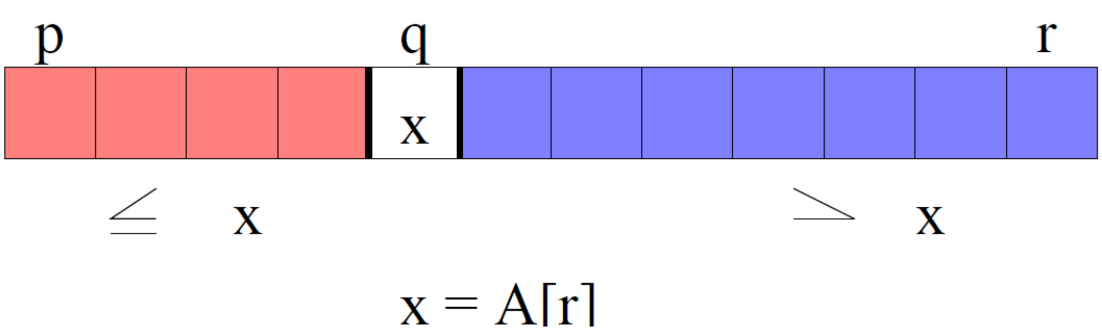


  - x is called the pivot. Assume x = A[r]; if not, swap first
  - Quicksort works by:
    - calling partition first
    - recursively sorting A[ ] and A[ ]

Partition

- Partition
  - Given: An array of numbers
  - Partition: Rearrange the array A[p..r] in place into two (possibly empty) subarrays A[p..q-1] and A[q+1..r] such that

$$A[u] < A[q] < A[v] \;\; for\ any\ p \leq u \leq q-1\ and\ q+1 \leq v \leq r$$

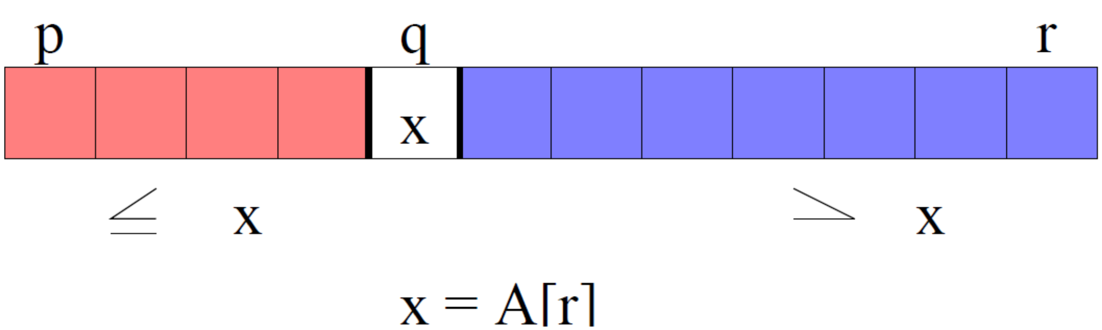


  - x is called the pivot. Assume x = A[r]; if not, swap first
  - Quicksort works by:
    - calling partition first
    - recursively sorting A[p..q-1] and A[ ]

Partition

- Partition
  - Given: An array of numbers
  - Partition: Rearrange the array A[p..r] in place into two (possibly empty) subarrays A[p..q-1] and A[q+1..r] such that

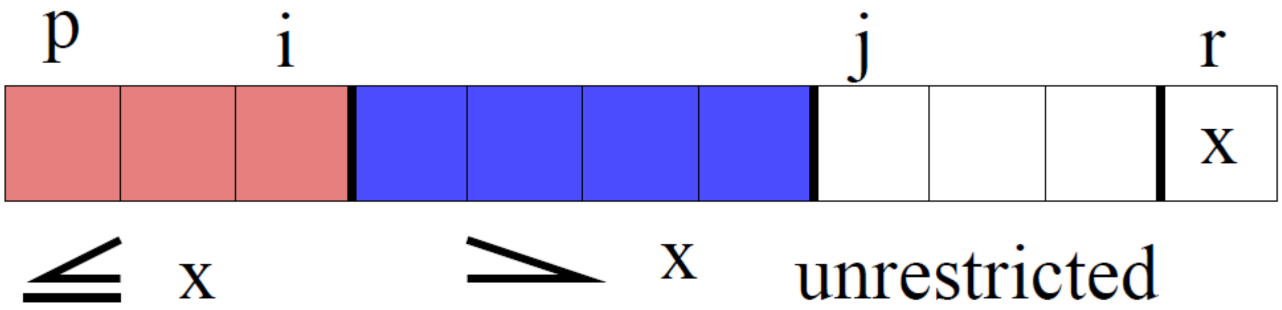
$$\color{red}{A[u]} < A[q] < \color{blue}{A[v]} \;\; for\, any\, p \le u \le q-1 \,and\, q+1 \le v \le r$$



  - x is called the pivot. Assume x = A[r]; if not, swap first
  - Quicksort works by:
    - calling partition first
    - recursively sorting A[p..q-1] and A[q+1..r]

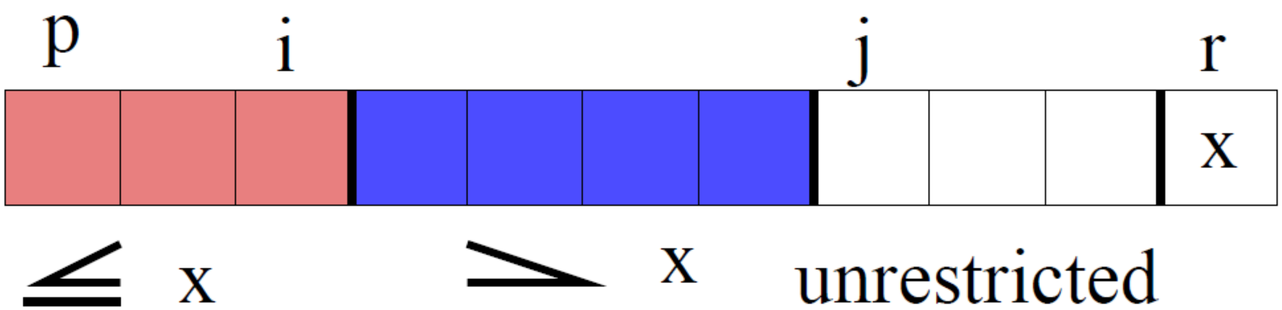
Partition

- The idea of Partition(A, p, r)
  - Use A[r] as the pivot, and grow partition from left to right



Partition

- The idea of Partition(A, p, r)
  - Use A[r] as the pivot, and grow partition from left to right



  - Initially (i, j) = (p–1, p)
  - Increase j by 1 each time to find a place for A[j]
    At the same time increase i when necessary
  - Stops when j = r

Partition

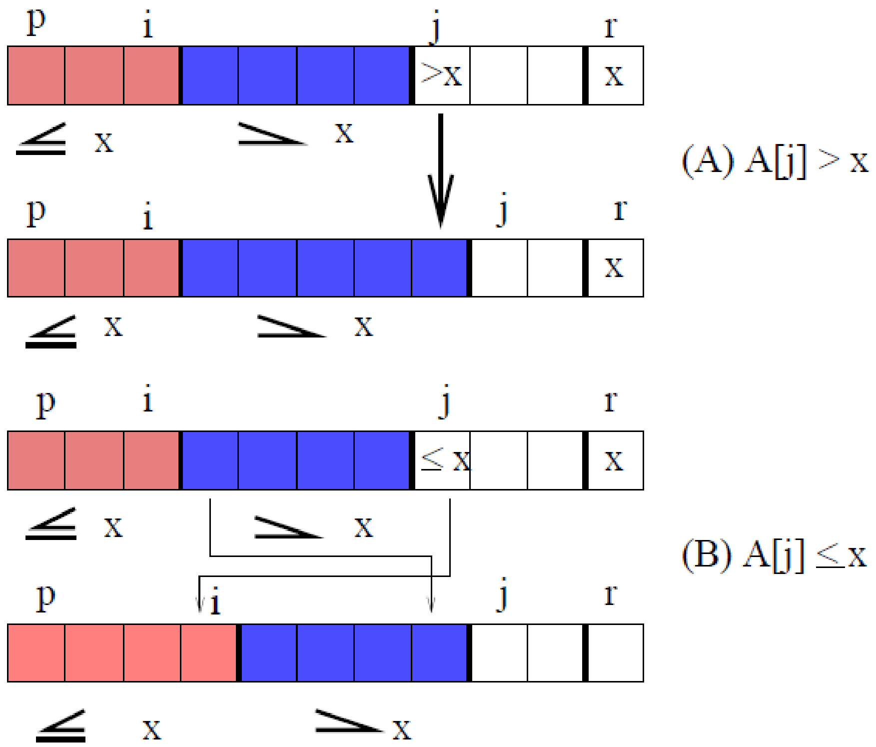
- One Iteration of the Procedure Partition
  - Increase j by 1 each time to find a place for A[j]

    At the same time increase i when necessary

Partition

- One Iteration of the Procedure Partition
  - Increase j by 1 each time to find a place for A[j]

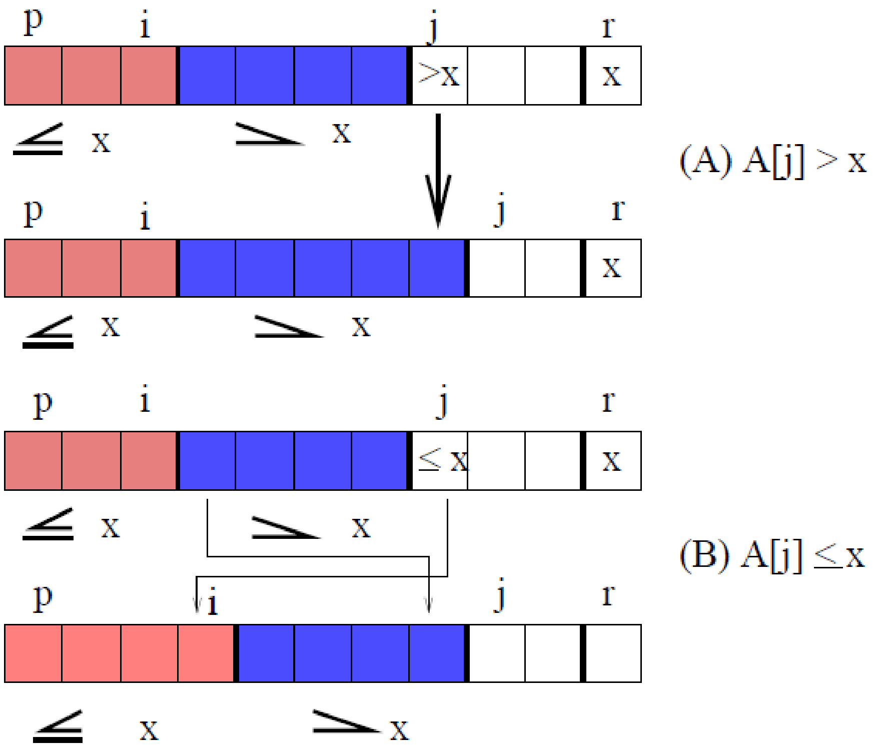
    At the same time increase i when necessary



Partition

- One Iteration of the Procedure Partition
  - Increase j by 1 each time to find a place for A[j]

    At the same time increase i when necessary

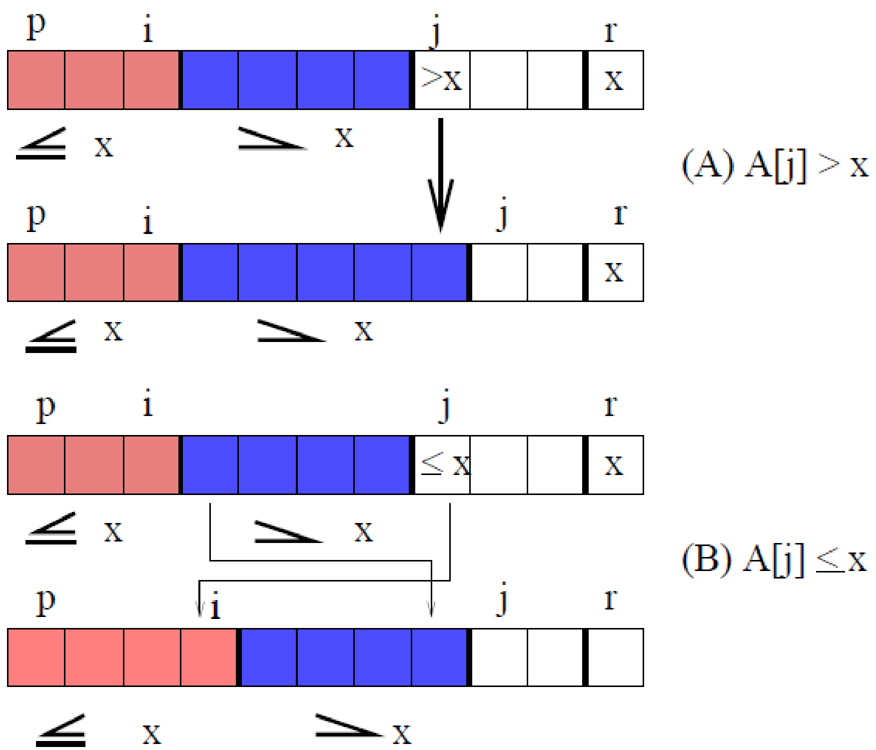


  - Case (A): Only increase j by 1

Partition

- One Iteration of the Procedure Partition
  - Increase j by 1 each time to find a place for A[j]

    At the same time increase i when necessary



  - Case (A): Only increase j by 1
  - Case (B): i = i + 1; A[i] ↔ A[j]; j = j + 1.

Partition-Example

- The Operation of Partition(A, p, r)

i	p,j						r
	2	**8**	**7**	**1**	**3**	**5**	**6**	**4**

Partition-Example

- The Operation of Partition(A, p, r)

i	p,j							r
	2	**8**	**7**	**1**	**3**	**5**	**6**	**4**

$$A[j] < A[r]$$

Partition-Example

- The Operation of Partition(A, p, r)

p,i	j						r
2	8	7	1	3	5	6	4

$$i = i + 1, A[i] \leftrightarrow A[j], j = j + 1$$

Partition-Example

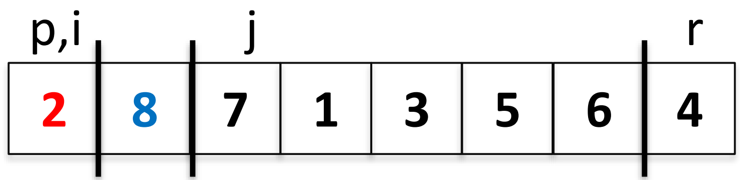
- The Operation of Partition(A, p, r)

p,i	j						r
2	8	7	1	3	5	6	4

$$A[j] > A[r]$$

Partition-Example

- The Operation of Partition(A, p, r)



$$Increase\ j\ by\ 1$$

Partition-Example

- The Operation of Partition(A, p, r)

p,i		j					r
2	8	7	1	3	5	6	4

$$A[j] > A[r]$$

Partition-Example

- The Operation of Partition(A, p, r)

p,i			j				r
2	8	7	1	3	5	6	4

$$Increase\ j\ by\ 1$$

Partition-Example

- The Operation of Partition(A, p, r)

p,i			j				r
2	8	7	1	3	5	6	4

$$A[j] < A[r]$$

Partition-Example

- The Operation of Partition(A, p, r)

p	i			j			r
2	1	7	8	3	5	6	4

$$i = i + 1, A[i] \leftrightarrow A[j], j = j + 1$$

Partition-Example

- The Operation of Partition(A, p, r)

p	i			j			r
2	1	7	8	3	5	6	4

$$A[j] < A[r]$$

Partition-Example

- The Operation of Partition(A, p, r)

p		i			j		r
2	1	3	8	7	5	6	4

$$i = i + 1, A[i] \leftrightarrow A[j], j = j + 1$$

Partition-Example

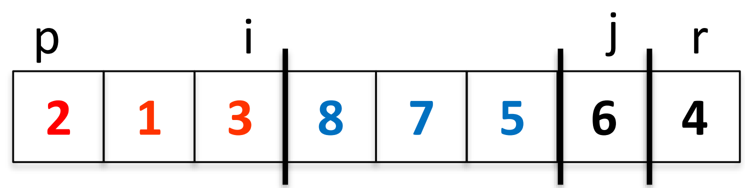
- The Operation of Partition(A, p, r)

p		i			j		r
2	1	3	8	7	5	6	4

$$A[j] > A[r]$$

Partition-Example

- The Operation of Partition(A, p, r)



$$Increase\ j\ by\ 1$$

Partition-Example

- The Operation of Partition(A, p, r)

p		i				j	r
2	1	3	8	7	5	6	4

$$A[j] > A[r]$$

Partition-Example

- The Operation of Partition(A, p, r)

p		i					j,r
2	1	3	8	7	5	6	4

$$Increase\ j\ by\ 1$$

Partition-Example

- The Operation of Partition(A, p, r)

p		i					j,r
2	1	3	8	7	5	6	4

Partition-Example

- The Operation of Partition(A, p, r)

p		i					j,r
2	1	3	4	7	5	6	8

$$A[i+1] \leftrightarrow A[r]$$

Partition - Pseudocode

Partition(A,p,r)

Input: An array $\boldsymbol{A}$ waiting to be sorted,the range of index $\boldsymbol{p}$,$\boldsymbol{r}$
Output: Index of the pivot after partition
$x \leftarrow A[r]$;$//A[r]$ is the pivot element

Partition - Pseudocode

Partition(A,p,r)

Input: An array $\boldsymbol{A}$ waiting to be sorted,the range of index $\boldsymbol{p}$,$\boldsymbol{r}$
Output: Index of the pivot after partition
$x \leftarrow A[r]$;//$A[r]$ is the pivot element
$i \leftarrow p - 1$;

Partition - Pseudocode

Partition(A,p,r)

Input: An array $\boldsymbol{A}$ waiting to be sorted,the range of index $\boldsymbol{p}$,$\boldsymbol{r}$
Output: Index of the pivot after partition
$x \leftarrow A[r]$;//$A[r]$ is the pivot element
$i \leftarrow p - 1$;
for $j \leftarrow p$ *to* $r - 1$ **do**

end

Partition - Pseudocode

Partition(A,p,r)

Input: An array $\boldsymbol{A}$ waiting to be sorted,the range of index $\boldsymbol{p}$,$\boldsymbol{r}$
Output: Index of the pivot after partition
$x \leftarrow A[r]$;$//A[r]$ is the pivot element
$i \leftarrow p - 1$;
for $j \leftarrow p$ *to* $r - 1$ **do**
    if $A[j] \leq x$ **then**

    end
end

Partition - Pseudocode

Partition(A,p,r)

Input: An array $\boldsymbol{A}$ waiting to be sorted,the range of index $\boldsymbol{p}$,$\boldsymbol{r}$
Output: Index of the pivot after partition

```
x ← A[r]; //A[r] is the pivot element
i ← p − 1;
for j ← p to r − 1 do
    if A[j] ≤ x then
        i ← i + 1;
        exchange A[i] and A[j];
    end
end
```

Partition - Pseudocode

Partition(A,p,r)

```
Input: An array A waiting to be sorted,the range of index p,r
Output: Index of the pivot after partition
x ← A[r];//A[r] is the pivot element
i ← p − 1;
for j ← p to r − 1 do
    if A[j] ≤ x then
        i ← i + 1;
        exchange A[i] and A[j];
    end
end
exchangeA[i + 1] and A[r];//Put pivot in position
```

Partition - Pseudocode

Partition(A,p,r)

Input: An array $\boldsymbol{A}$ waiting to be sorted,the range of index $\boldsymbol{p}$,$\boldsymbol{r}$
Output: Index of the pivot after partition
$x \leftarrow A[r]$;//$A[r]$ is the pivot element
$i \leftarrow p - 1$;
for $j \leftarrow p$ *to* $r - 1$ **do**
    if $A[j] \leq x$ **then**
        $i \leftarrow i + 1$;
        exchange $A[i]$ and $A[j]$;
    end
end
exchange$A[i + 1]$ and $A[r]$;//Put pivot in position
return

Partition - Pseudocode

Partition(A,p,r)

```
Input: An array A waiting to be sorted,the range of index p,r
Output: Index of the pivot after partition
x ← A[r];//A[r] is the pivot element
i ← p − 1;
for j ← p to r − 1 do
    if A[j] ≤ x then
        i ← i + 1;
        exchange A[i] and A[j];
    end
end
exchangeA[i + 1] and A[r];//Put pivot in position
return i + 1;//q ← i + 1
```

Partition - Pseudocode

Partition(A,p,r)

Input: An array $\boldsymbol{A}$ waiting to be sorted,the range of index $\boldsymbol{p}$,$\boldsymbol{r}$
Output: Index of the pivot after partition

```
x ← A[r]; //A[r] is the pivot element
i ← p − 1;
for j ← p to r − 1 do
    if A[j] ≤ x then
        i ← i + 1;
        exchange A[i] and A[j];
    end
end
exchange A[i + 1] and A[r]; //Put pivot in position
return i + 1; //q ← i + 1
```

- Running time is O(   )

Partition - Pseudocode

Partition(A,p,r)

Input: An array $\boldsymbol{A}$ waiting to be sorted,the range of index $\boldsymbol{p}$,$\boldsymbol{r}$
Output: Index of the pivot after partition

```
x ← A[r];//A[r] is the pivot element
i ← p − 1;
for j ← p to r − 1 do
    if A[j] ≤ x then
        i ← i + 1;
        exchange A[i] and A[j];
    end
end
exchangeA[i + 1] and A[r];//Put pivot in position
return i + 1;//q ← i + 1
```

- Running time is O(r – p)

Partition - Pseudocode

Partition(A,p,r)

Input: An array $\boldsymbol{A}$ waiting to be sorted,the range of index $\boldsymbol{p}$,$\boldsymbol{r}$
Output: Index of the pivot after partition
$x \leftarrow A[r]$;//$A[r]$ is the pivot element
$i \leftarrow p - 1$;
for $j \leftarrow p$ *to* $r - 1$ **do**
    if $A[j] \leq x$ **then**
        $i \leftarrow i + 1$;
        exchange $A[i]$ and $A[j]$;
    end
end
exchange$A[i + 1]$ and $A[r]$;//Put pivot in position
return $i + 1$;//$q \leftarrow i + 1$

- Running time is O(r – p)
  - linear in the length of the array A[p..r]

Quicksort

Quicksort(A,p,r)

Input: An array $\boldsymbol{A}$ waiting to be sorted,the range of index $\boldsymbol{p},\boldsymbol{r}$
Output: Sorted array $\boldsymbol{A}$
if $p < r$ **then**
  $q \leftarrow$Partition(A,p,r);
  Quicksort$(A,$ $)$;
  Quicksort$(A,$ $)$;
end
return A;

Quicksort

Quicksort(A,p,r)

Input: An array $\boldsymbol{A}$ waiting to be sorted,the range of index $\boldsymbol{p}$,$\boldsymbol{r}$
Output: Sorted array $\boldsymbol{A}$
if $p < r$ **then**
  $q \leftarrow$Partition(A, p, r);
  Quicksort$(A, p, q - 1)$;
  Quicksort$(A,$ $)$;
end
return A;

Quicksort

Quicksort(A,p,r)

Input: An array $\boldsymbol{A}$ waiting to be sorted,the range of index $\boldsymbol{p}$,$\boldsymbol{r}$
Output: Sorted array $\boldsymbol{A}$
if $p < r$ **then**
  $q \leftarrow$Partition(A, p, r);
  Quicksort$(A, p, q - 1)$;
  Quicksort$(A, q + 1, r)$;
end
return A;

A Divide-and-Conquer Framework

Quicksort

Quicksort(A,p,r)

Input: An array $\boldsymbol{A}$ waiting to be sorted,the range of index $\boldsymbol{p}$,$\boldsymbol{r}$
Output: Sorted array $\boldsymbol{A}$
if $p < r$ **then**
  $q \leftarrow$Partition(A, p, r);
  Quicksort($A, p, q - 1$);
  Quicksort($A, q + 1, r$);
end
return A;

A Divide-and-Conquer Framework

- If we could always partition the array into halves, then we have the recurrence $T(n) \leq 2T(n/2) + O(n)$, hence $T(n) = O(n \log n)$.

Quicksort

Quicksort(A,p,r)

> **Input:** An array $\boldsymbol{A}$ waiting to be sorted,the range of index $\boldsymbol{p},\boldsymbol{r}$
> **Output:** Sorted array $\boldsymbol{A}$
> **if** $p < r$ **then**
> $q \leftarrow$Partition(A, p, r);
>   Quicksort$(A, p, q-1)$;
>   Quicksort$(A, q+1, r)$;
> **end**
> **return** A;

A Divide-and-Conquer Framework

- If we could always partition the array into halves, then we have the recurrence $T(n) \leq 2T(n/2) + O(n)$, hence $T(n) = O(n \log n)$.
- However, if we always get unlucky with very unbalanced partitions, then $T(n) \leq T(n-1) + O(n)$, hence $T(n) = O(n^2)$.

Outline

- Review to Divide-and-Conquer Paradigm
- Polynomial Multiplication Problem
  - Problem definition
  - A brute force algorithm
  - A first divide-and-conquer algorithm
  - An improved divide-and-conquer algorithm
  - Analysis of the divide-and-conquer algorithm
- Quicksort Problem
  - Basic partition
  - Randomized partition and randomized quicksort
  - Analysis of the randomized quicksort

Randomized-Partition(A, p, r)

- Idea
  - In the algorithm Partition(A, p, r), A[r] is always used as the pivot *x* to partition the array A[p..r].

Randomized-Partition(A, p, r)

- Idea
  - In the algorithm Partition(A, p, r), A[r] is always used as the pivot *x* to partition the array A[p..r].
  - In the algorithm Randomized-Partition(A, p, r), we randomly choose an j , p ≤ j ≤ r , and use A[j] as pivot.

Randomized-Partition(A, p, r)

- Idea
  - In the algorithm Partition(A, p, r), A[r] is always used as the pivot x to partition the array A[p..r].
  - In the algorithm Randomized-Partition(A, p, r), we randomly choose an j , p ≤ j ≤ r , and use A[j] as pivot.
  - Idea is that if we choose randomly, then the chance that we get unlucky every time is extremely low.

Randomized-Partition(A, p, r)

- Pseudocode of Randomized-Partition
  - Let random(p, r) be a pseudorandom-number generator that returns a random number between p and r.

Randomized-Partition(A, p, r)

- Pseudocode of Randomized-Partition
  - Let random(p, r) be a pseudorandom-number generator that returns a random number between p and r.

Randomized-Partition(A,p,r)

Input: An array $\boldsymbol{A}$ waiting to be sorted,the range of index $\boldsymbol{p}$,$\boldsymbol{r}$
Output: A random index in $[p..j]$

Partition(A, p, r);
return j;

Randomized-Partition(A, p, r)

- Pseudocode of Randomized-Partition
  - Let random(p, r) be a pseudorandom-number generator that returns a random number between p and r.

Randomized-Partition(A,p,r)

```
Input: An array A waiting to be sorted,the range of index p,r
Output: A random index in [p..j]
j ←random(p, r);

Partition(A, p, r);
return j;
```

Randomized-Partition(A, p, r)

- Pseudocode of Randomized-Partition
  - Let random(p, r) be a pseudorandom-number generator that returns a random number between p and r.

Randomized-Partition(A,p,r)

Input: An array $\boldsymbol{A}$ waiting to be sorted,the range of index $\boldsymbol{p}$,$\boldsymbol{r}$
Output: A random index in $[p..j]$
$j \leftarrow$random(p, r);
exchange $A[r]$ and $A[j]$;
Partition(A, p, r);
return j;

Randomized-Partition(A, p, r)

- Pseudocode of Randomized-Quicksort
  - We make use of the Randomized-Partition idea to develop a new version of quicksort.

Randomized-Partition(A, p, r)

- Pseudocode of Randomized-Quicksort
  - We make use of the Randomized-Partition idea to develop a new version of quicksort.

Randomized-Quicksort(A,p,r)

```
Input: An array A waiting to be sorted,the range of index p,r
Output: Sorted array A
if p < r then
    q ← Randomized-Partition(A, p, r);
    Randomized-Quicksort(A,          );
    Randomized-Quicksort(A,          );
end
return A;
```

Randomized-Partition(A, p, r)

- Pseudocode of Randomized-Quicksort
  - We make use of the Randomized-Partition idea to develop a new version of quicksort.

Randomized-Quicksort(A,p,r)

```
Input: An array A waiting to be sorted,the range of index p,r
Output: Sorted array A
if p < r then
    q ← Randomized-Partition(A, p, r);
    Randomized-Quicksort(A, p, q − 1);
    Randomized-Quicksort(A,            );
end
return A;
```

Randomized-Partition(A, p, r)

- Pseudocode of Randomized-Quicksort
  - We make use of the Randomized-Partition idea to develop a new version of quicksort.

Randomized-Quicksort(A,p,r)

```
Input: An array A waiting to be sorted,the range of index p,r
Output: Sorted array A
if p < r then
    q ← Randomized-Partition(A, p, r);
    Randomized-Quicksort(A, p, q − 1);
    Randomized-Quicksort(A, q + 1, r);
end
return A;
```

Quicksort - Example

2	**8**	**7**	**1**	**3**	**5**	**6**	**4**

Quicksort - Example

Divide

2	8	7	1	3	5	6	4

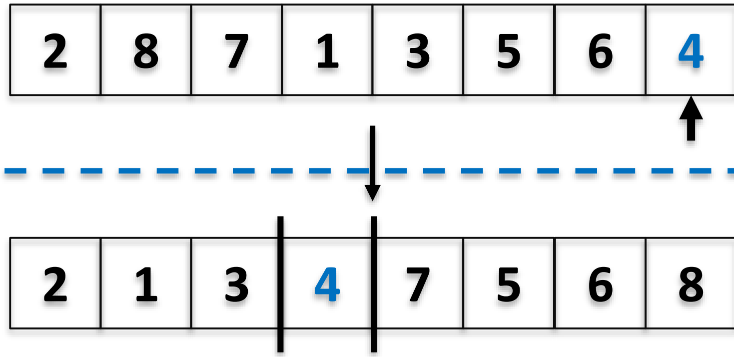
Quicksort - Example

Divide

2	8	7	1	3	5	6	4

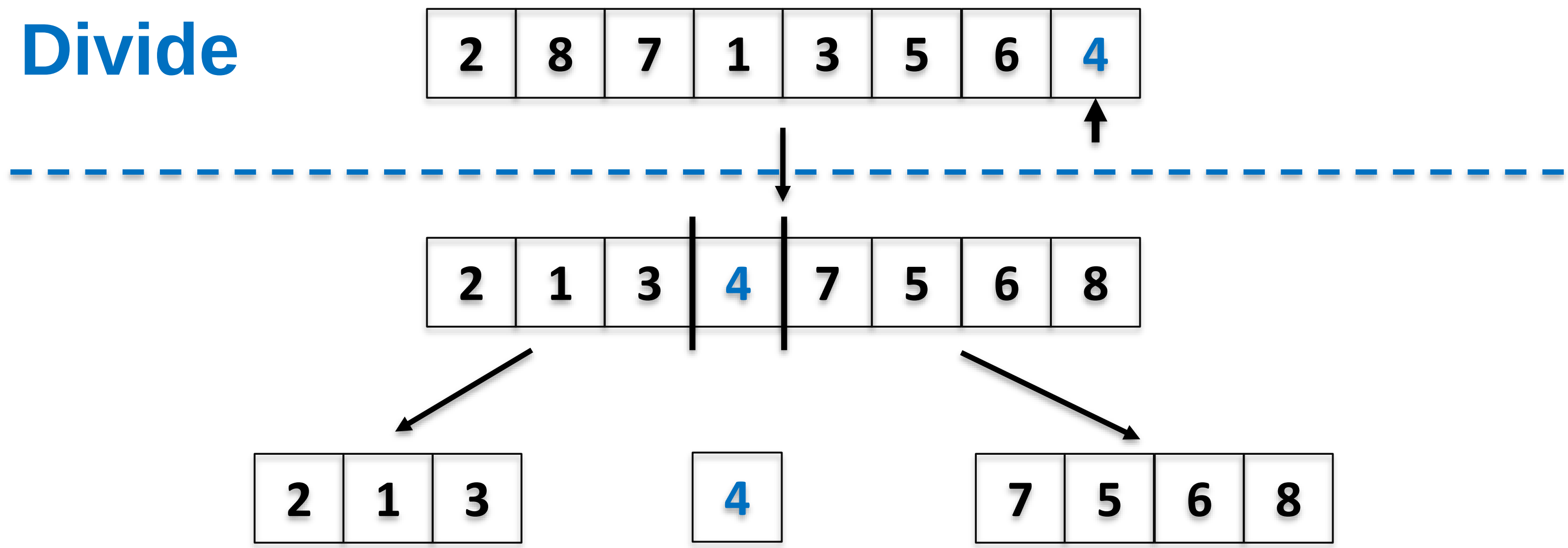
Quicksort - Example

Divide



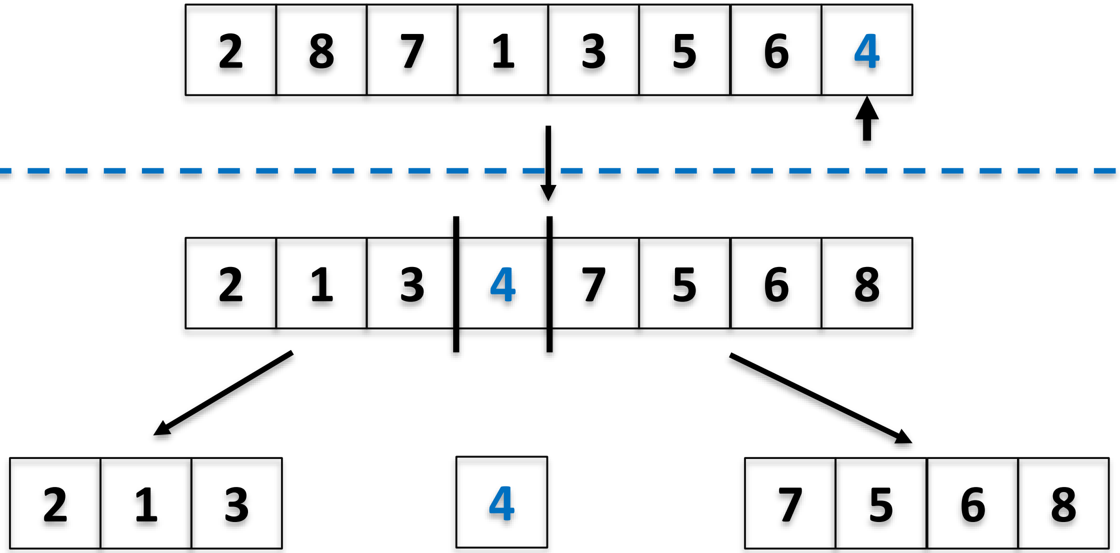
Quicksort - Example

Divide



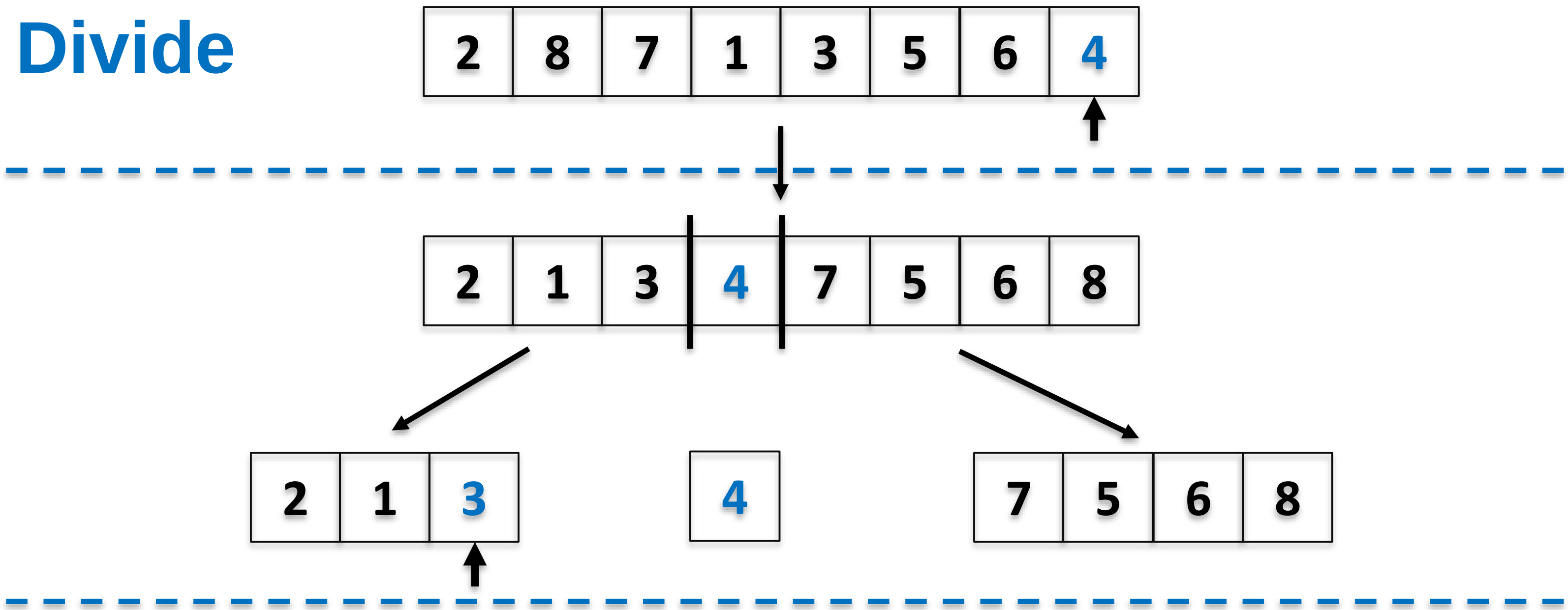
Quicksort - Example

Divide

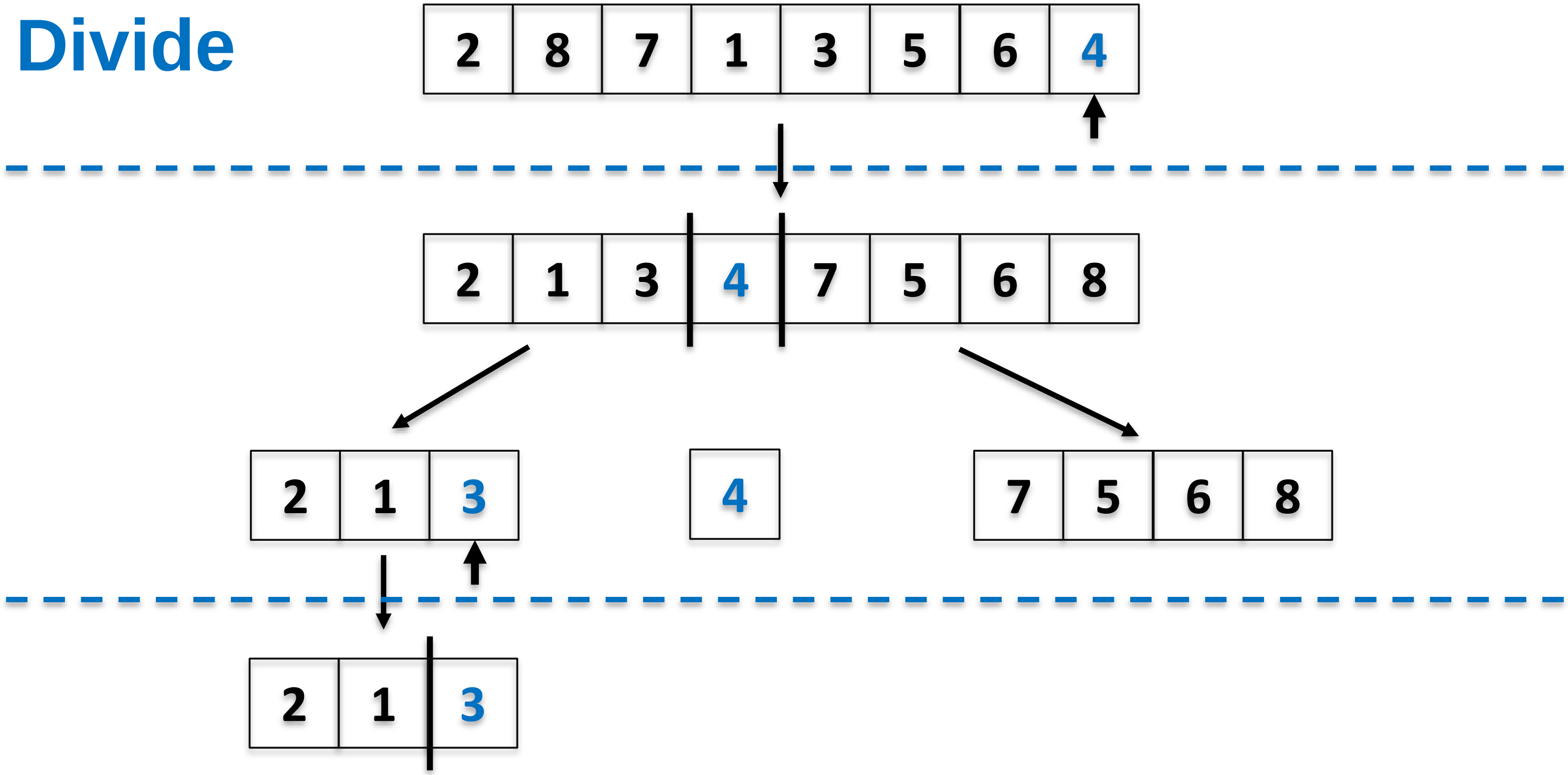


Quicksort - Example

Divide

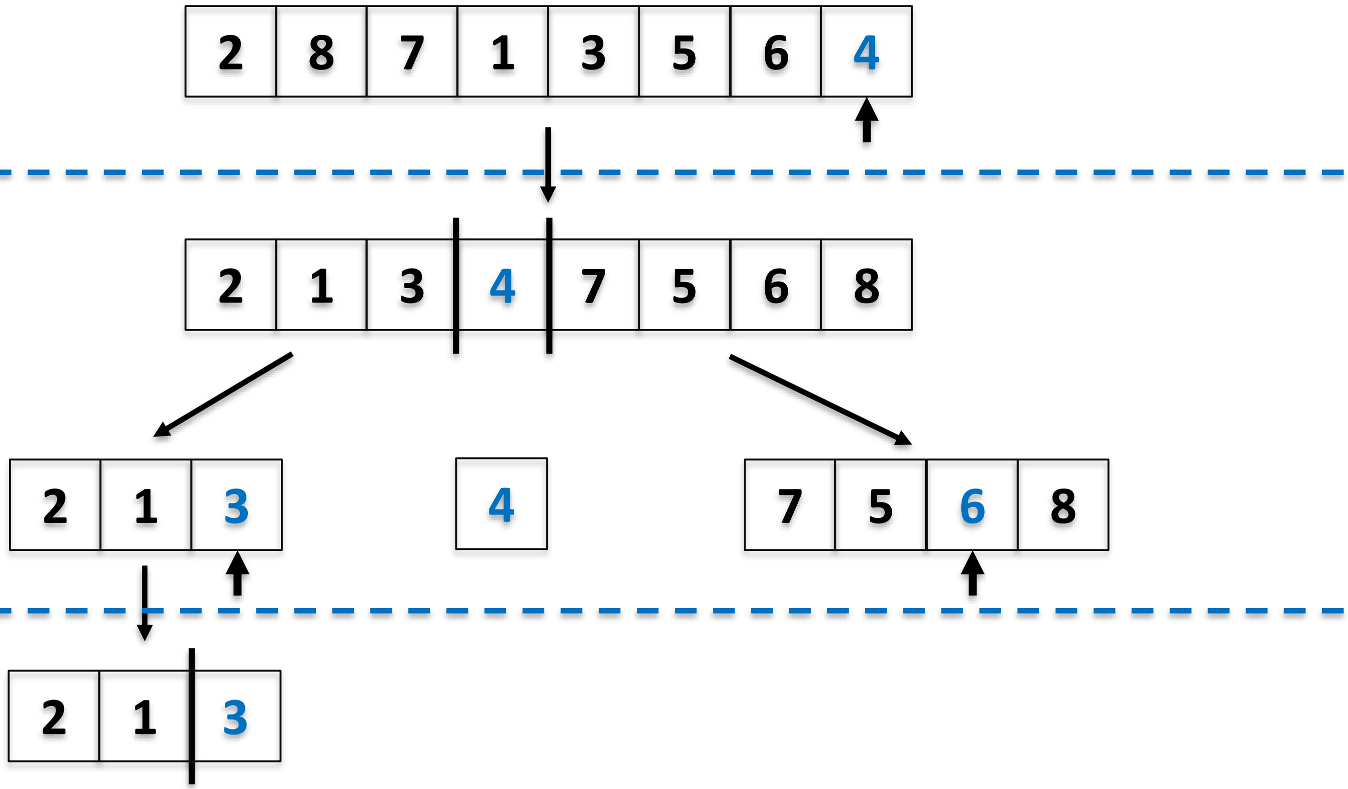


Quicksort - Example



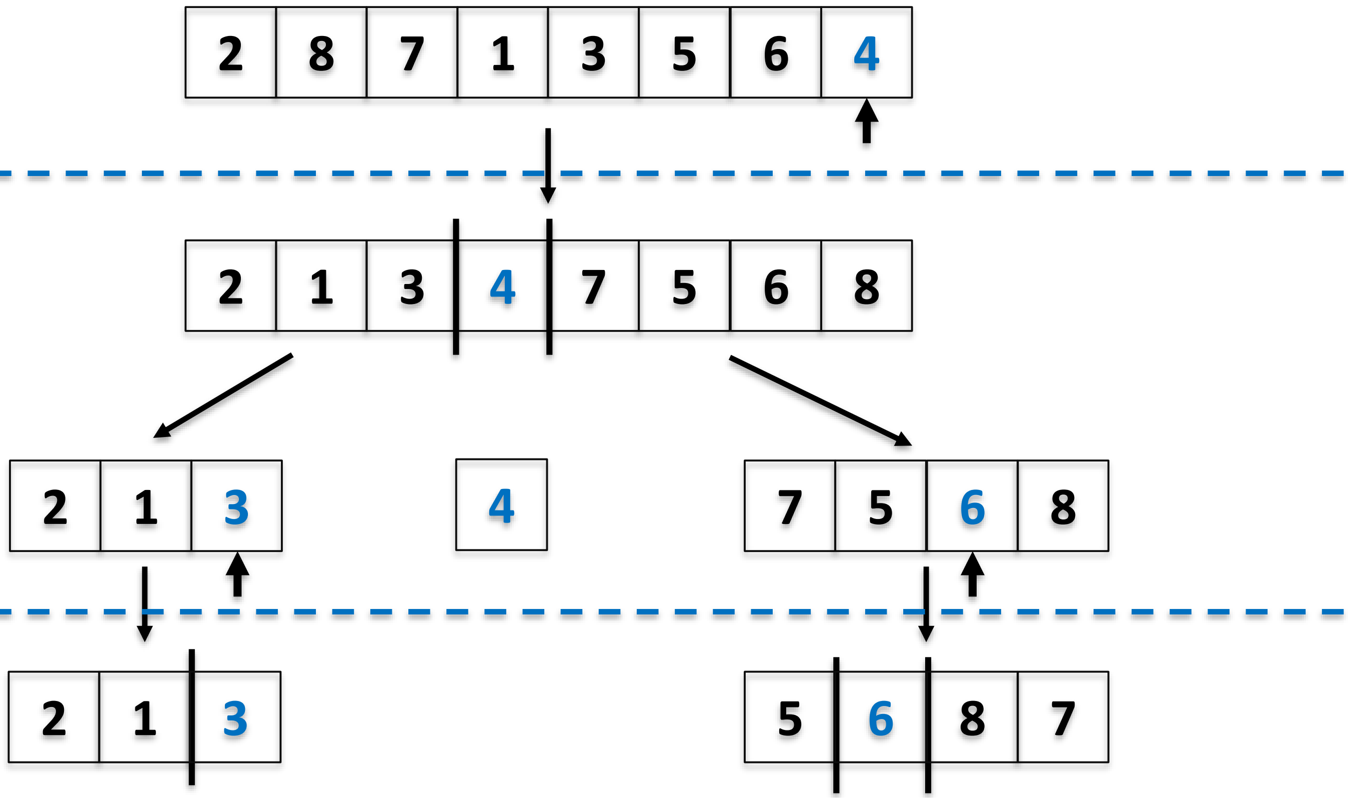
Quicksort - Example

Divide



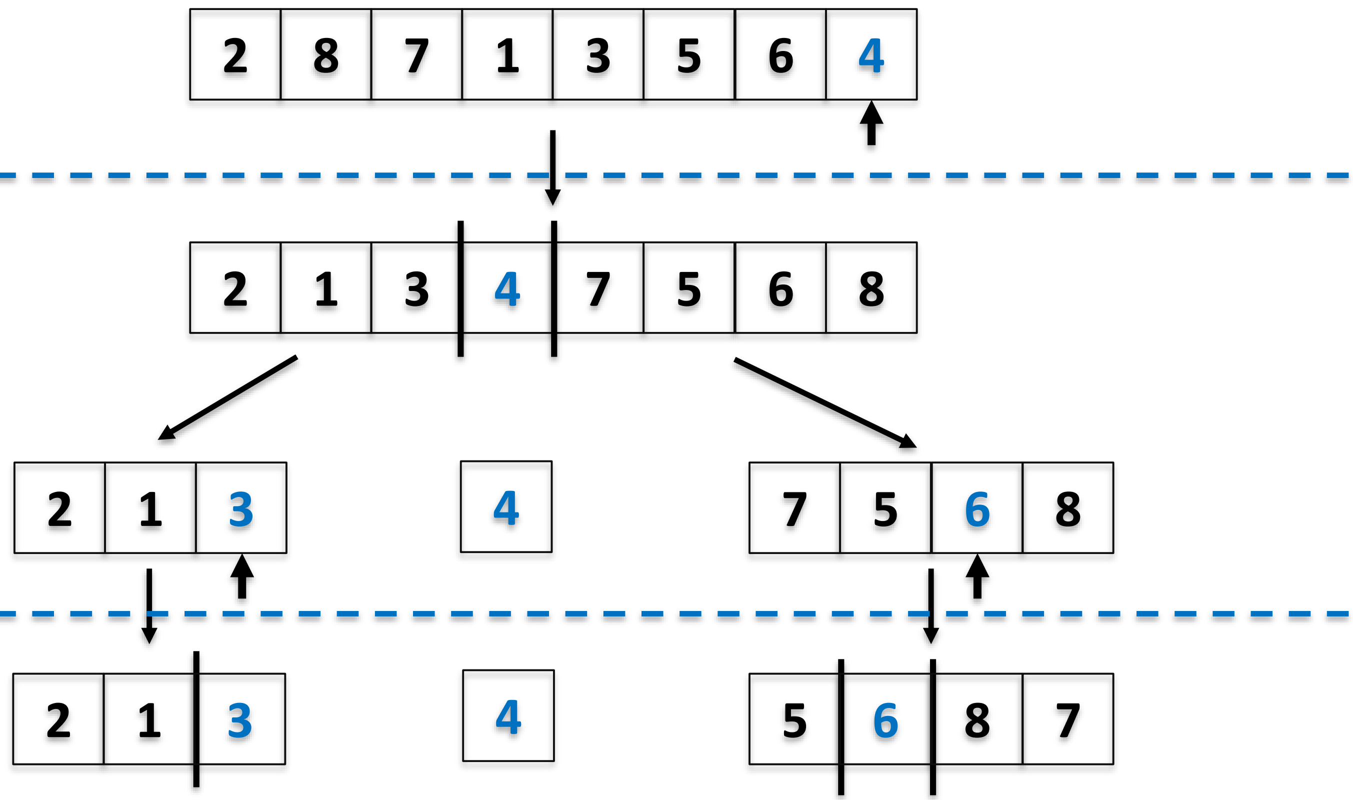
Quicksort - Example

Divide

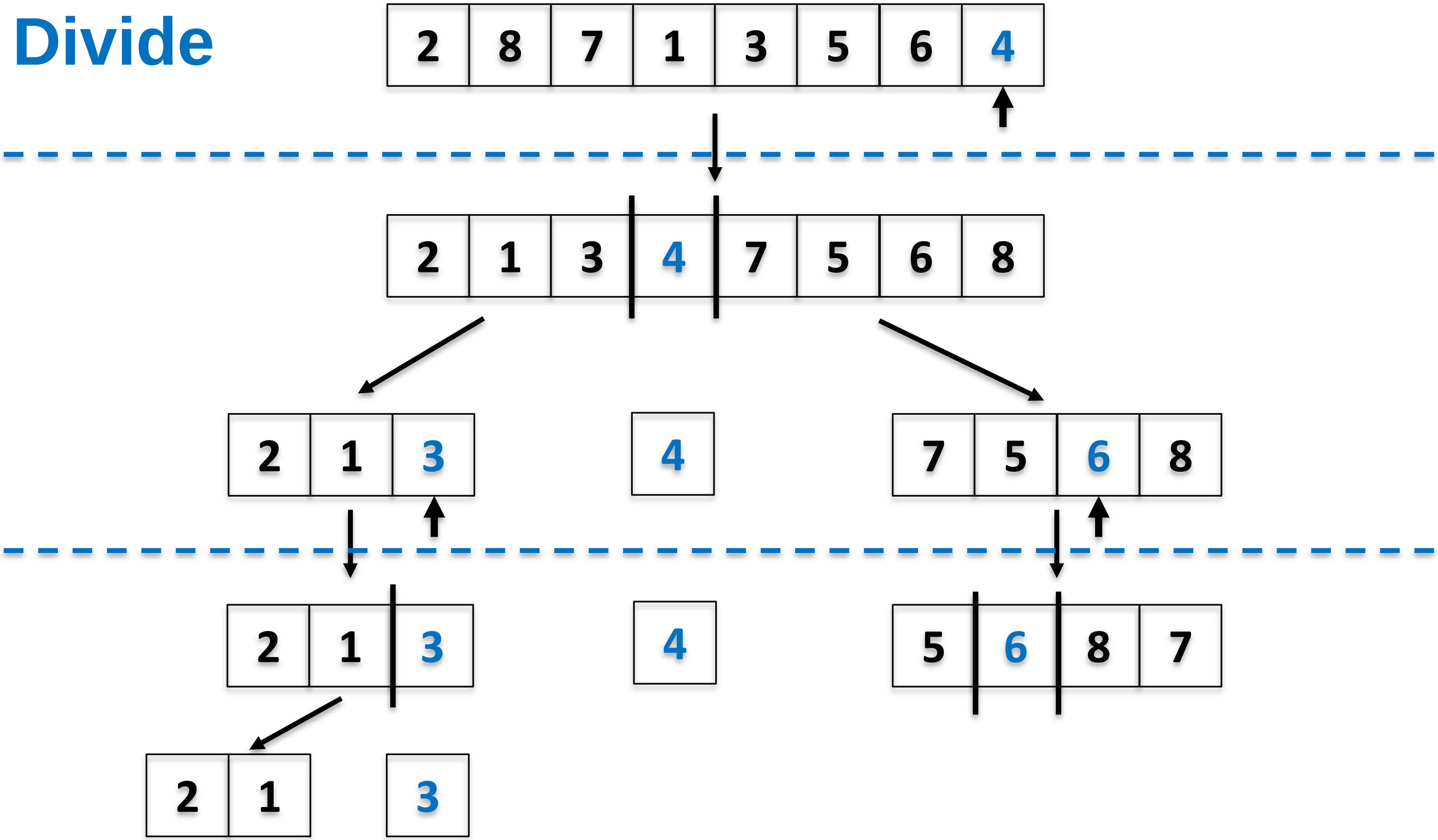


Quicksort - Example

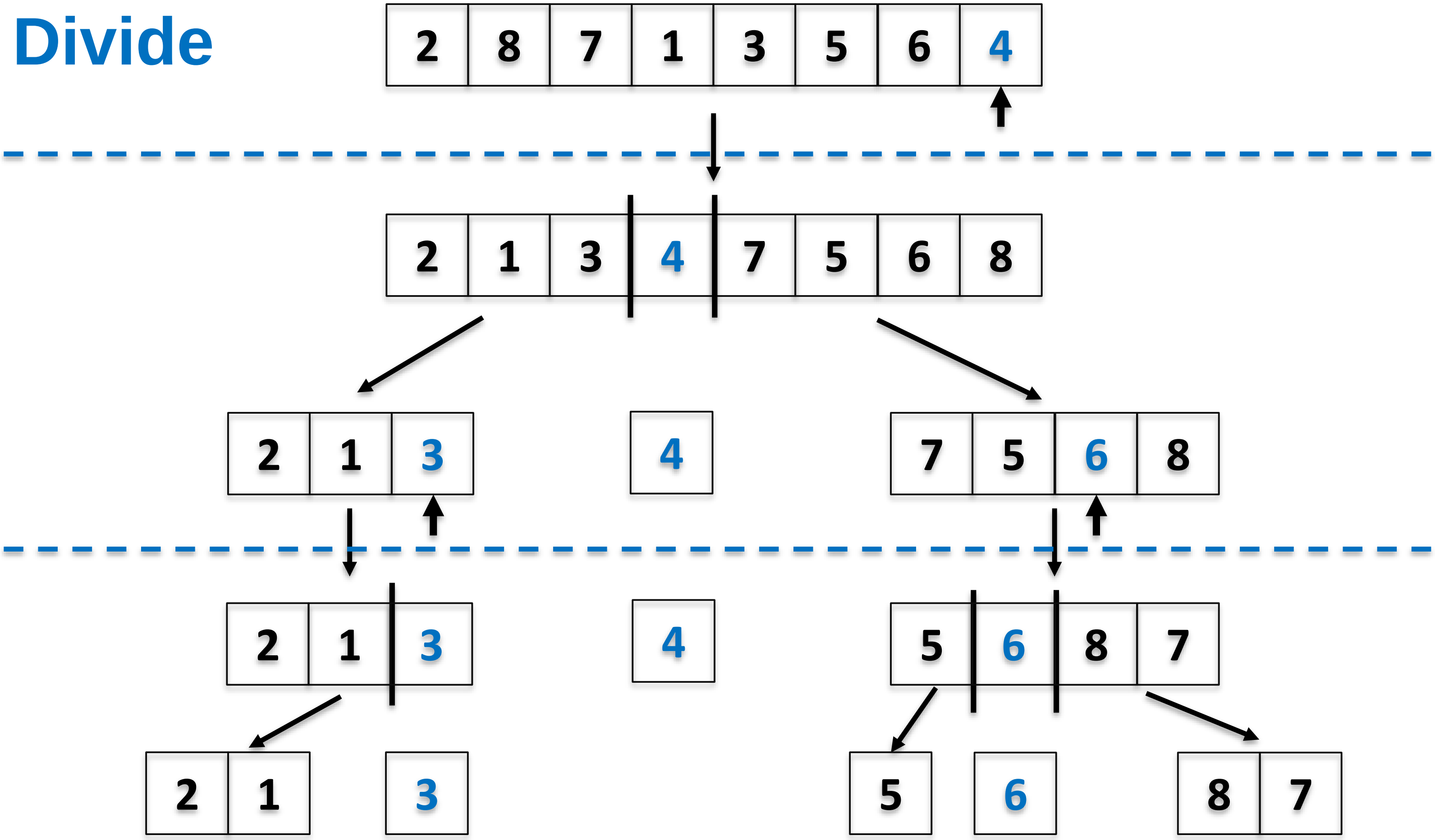
Divide



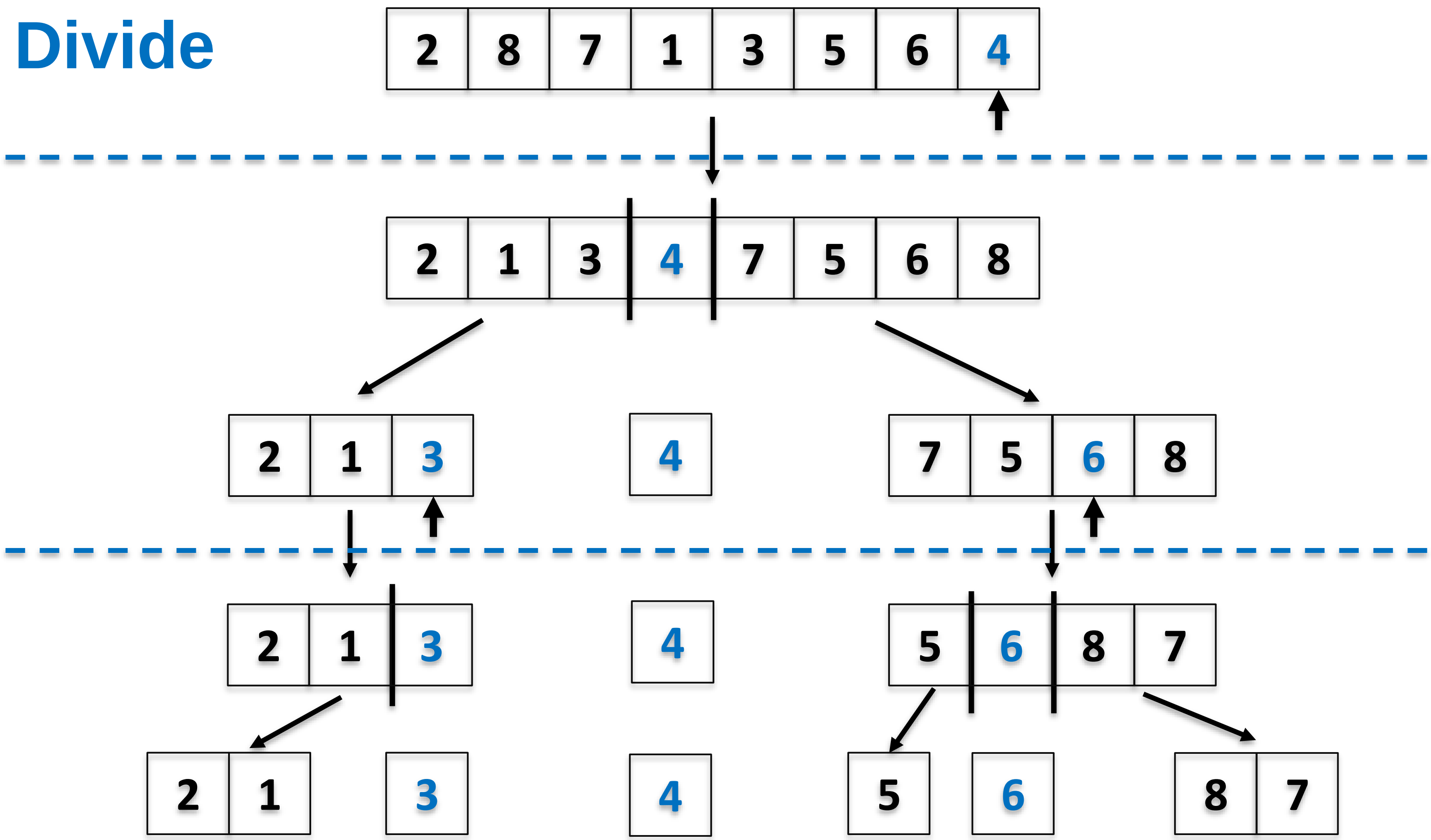
Quicksort - Example



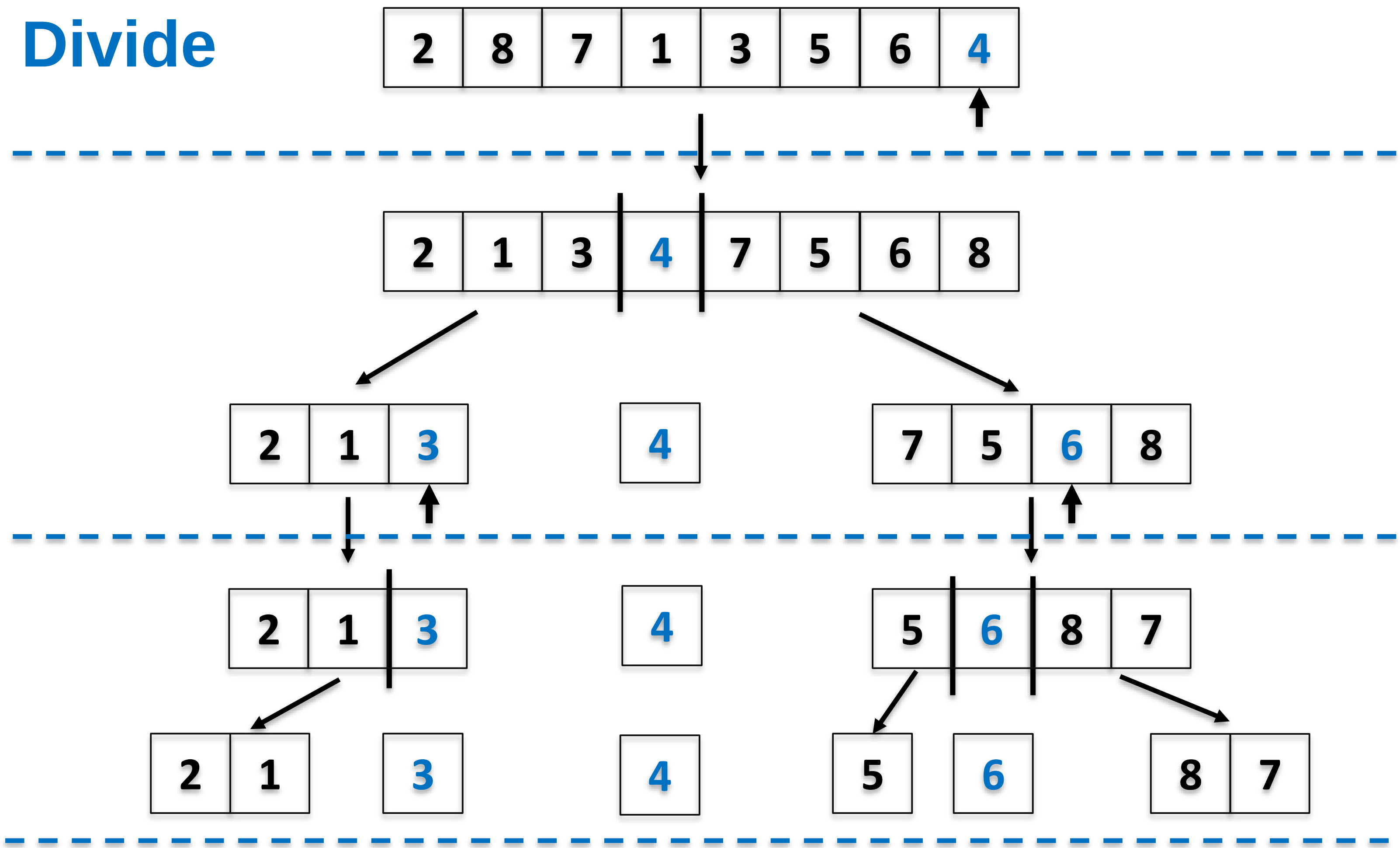
Quicksort - Example



Quicksort - Example

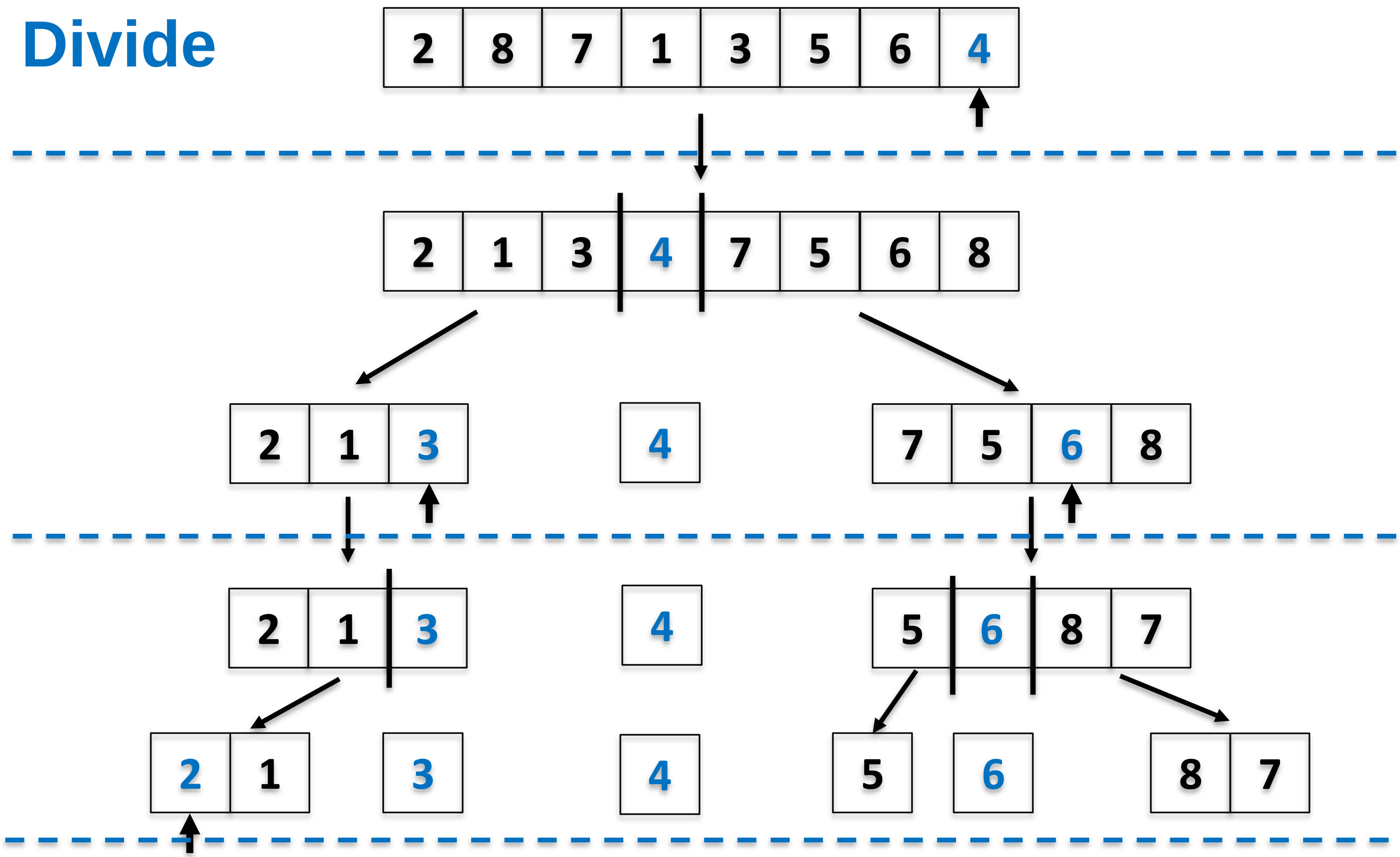


Quicksort - Example



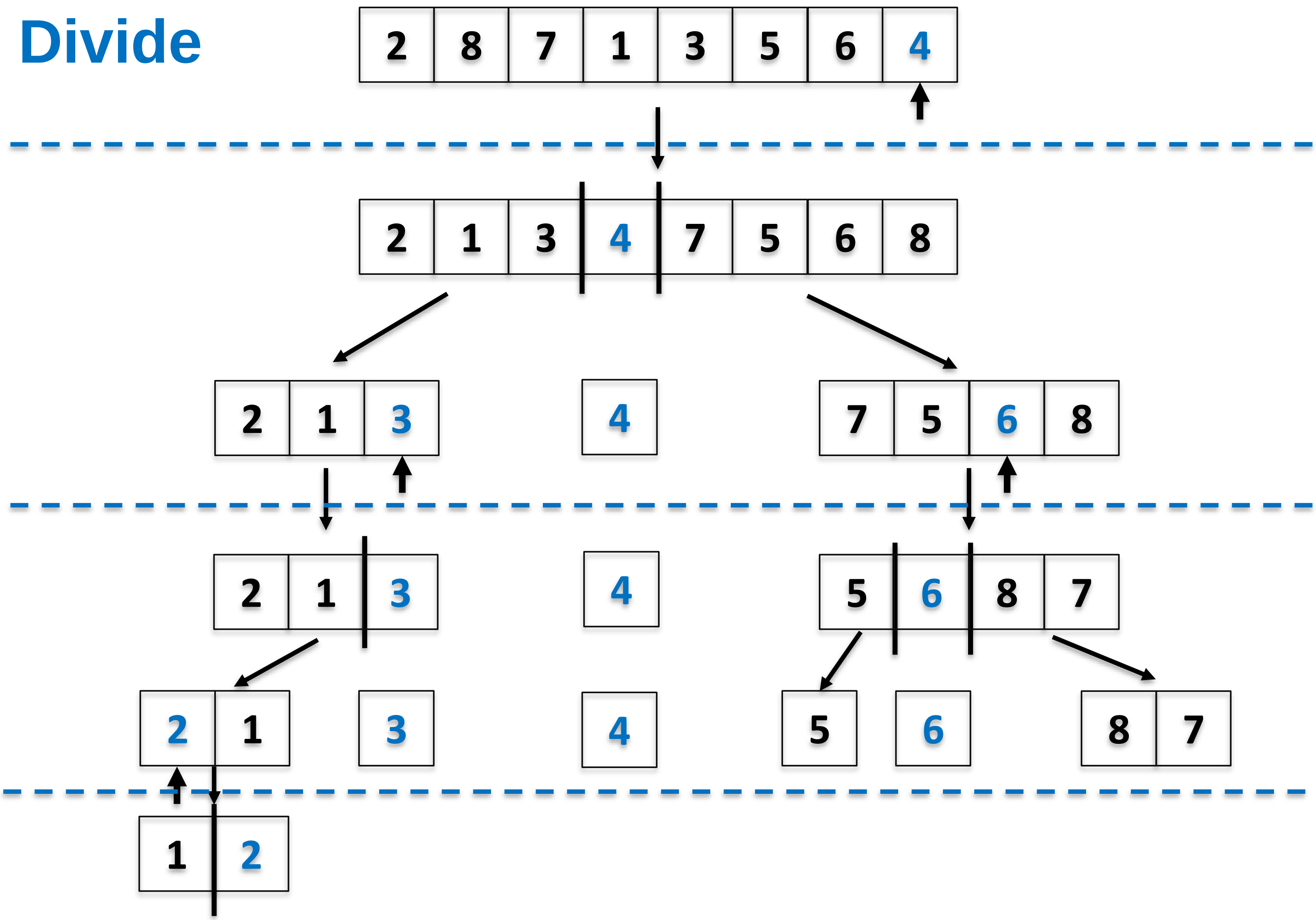
Quicksort - Example

Divide



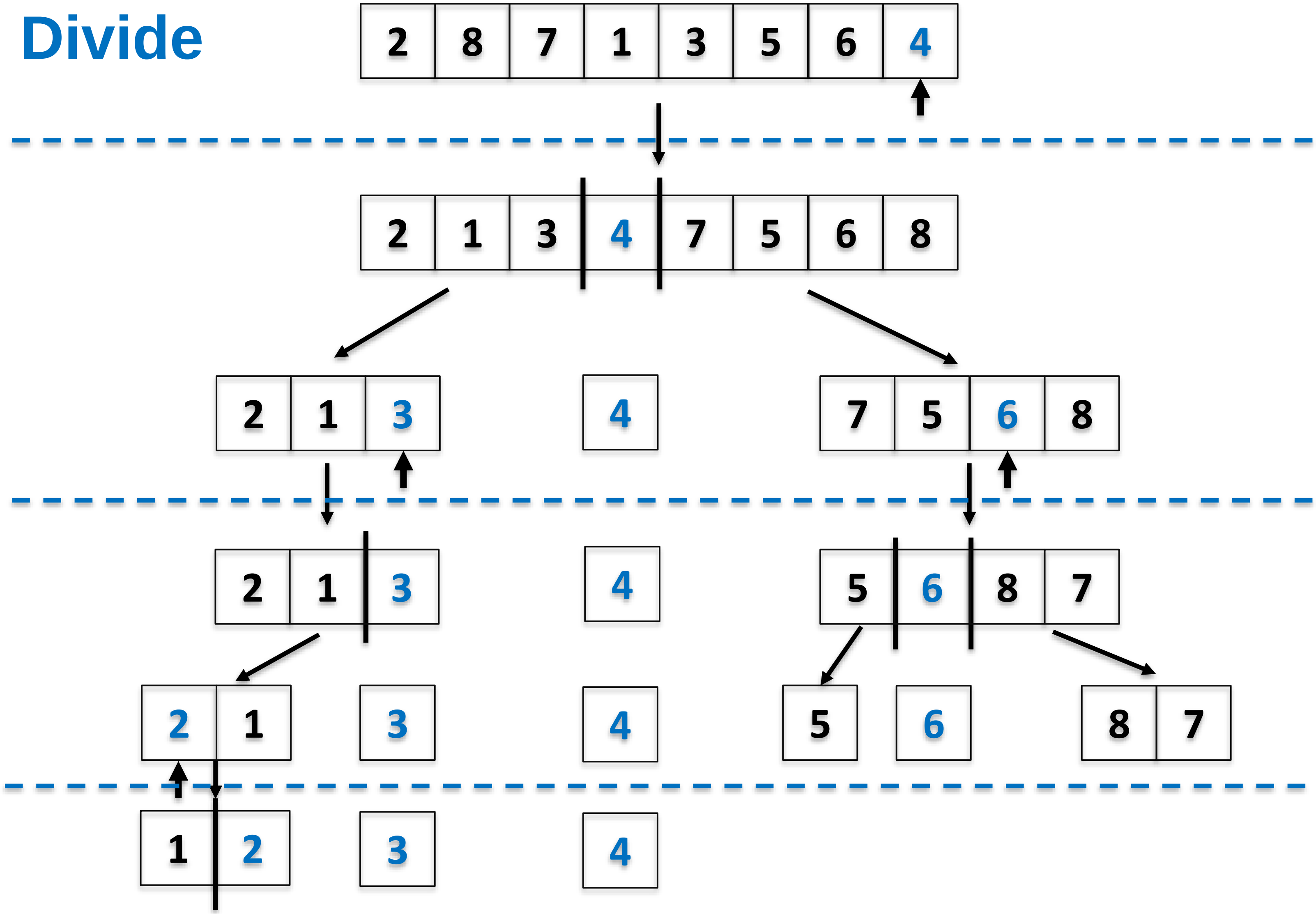
Quicksort - Example

Divide



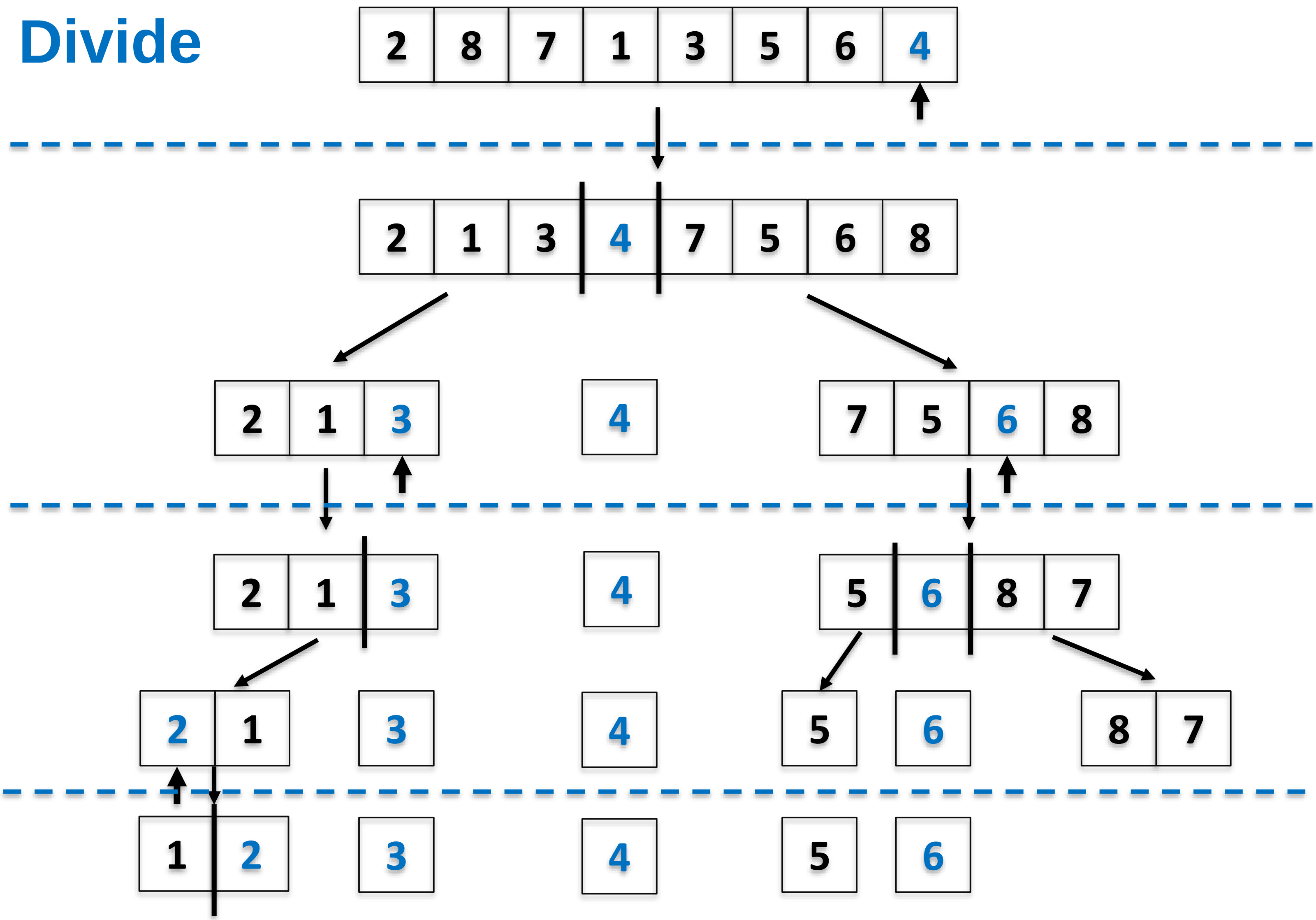
Quicksort - Example

Divide

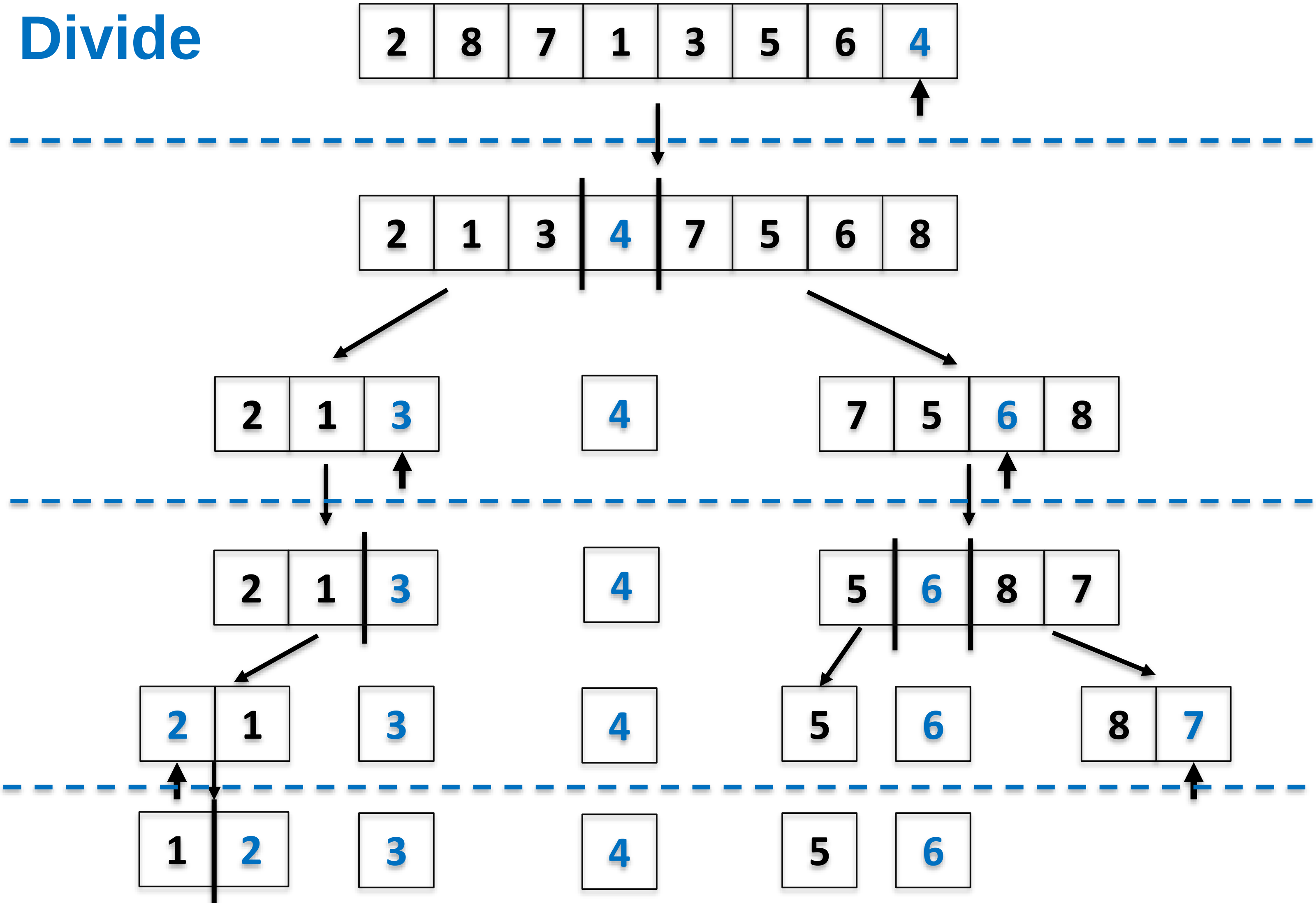


Quicksort - Example

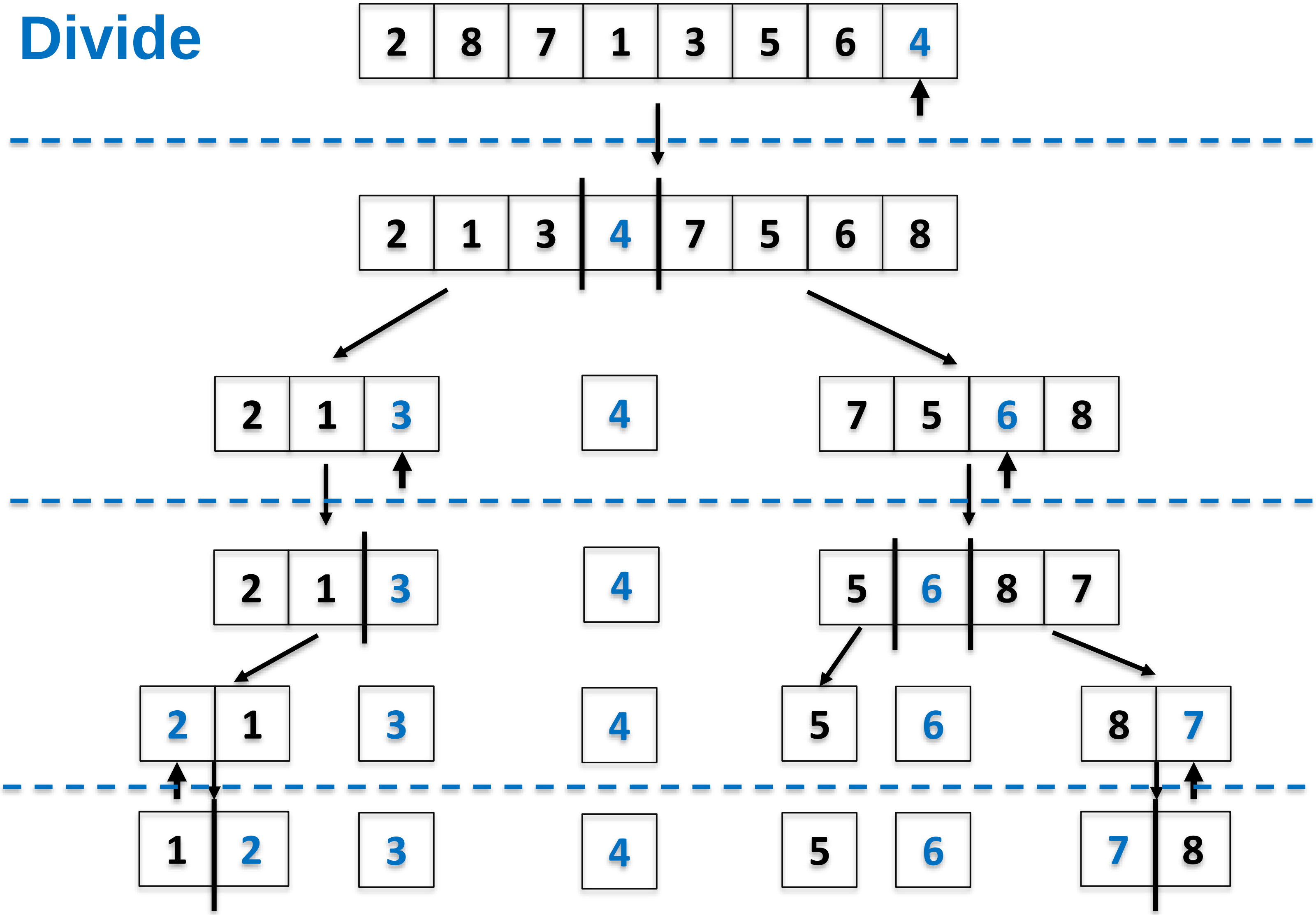
Divide



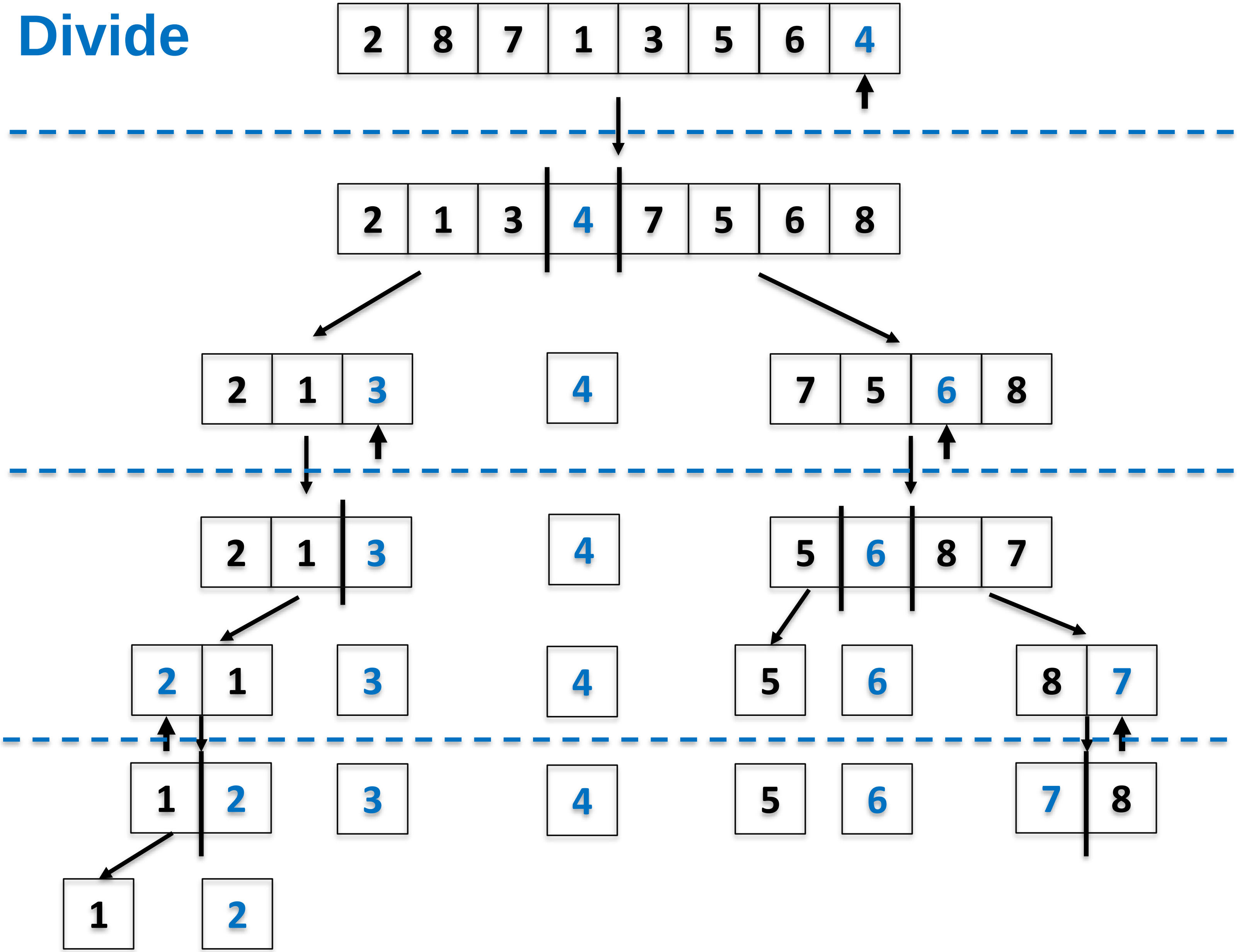
Quicksort - Example



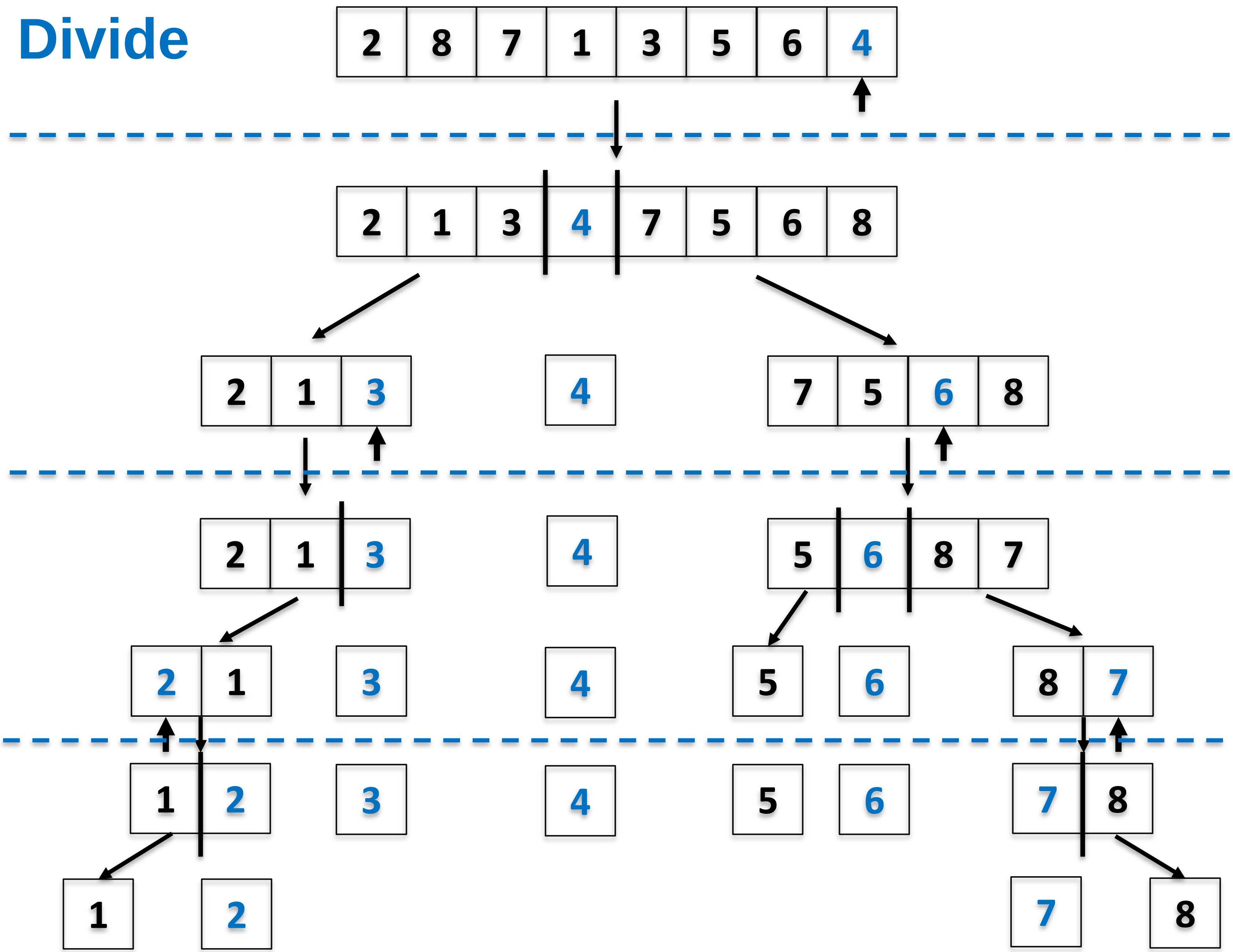
Quicksort - Example



Quicksort - Example

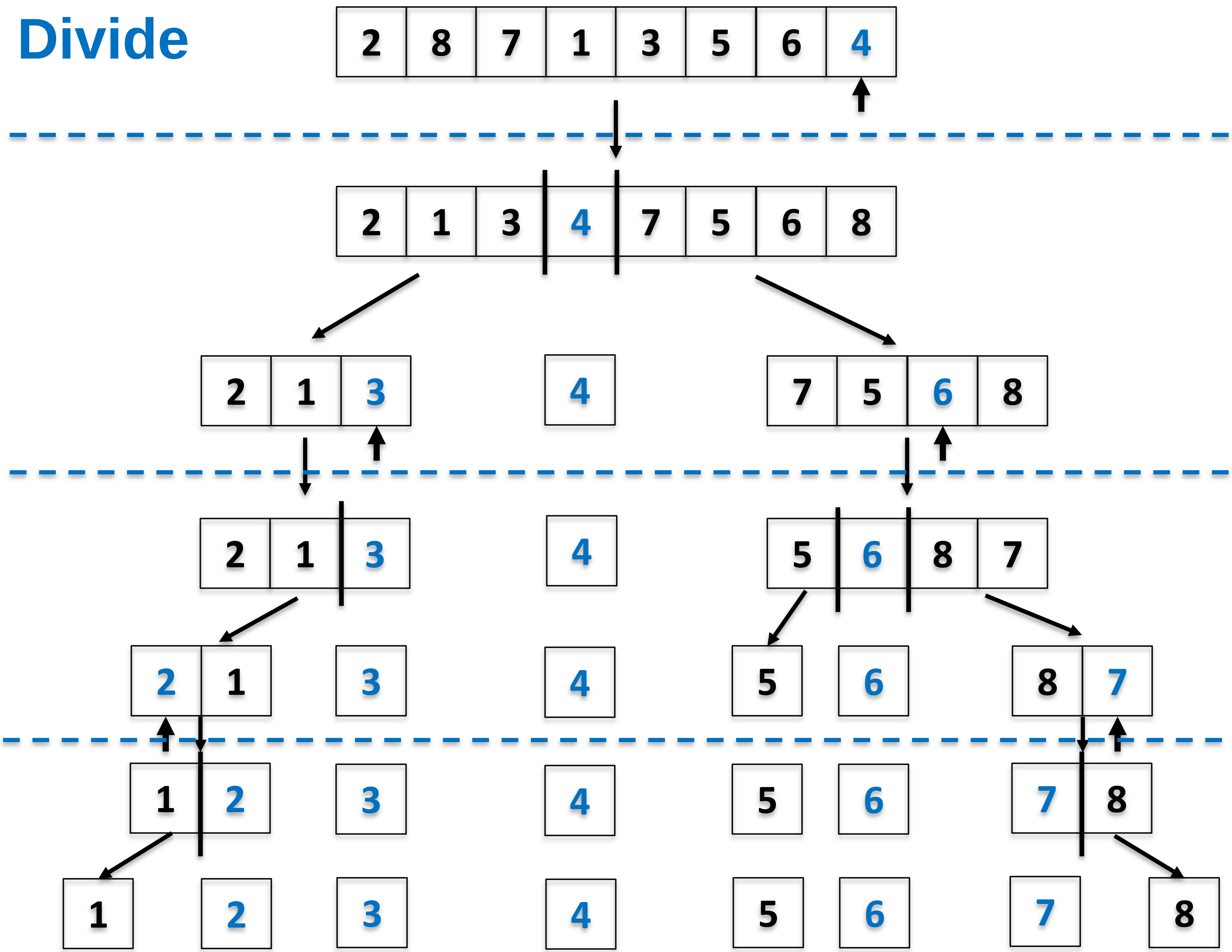


Quicksort - Example



Quicksort - Example

Divide



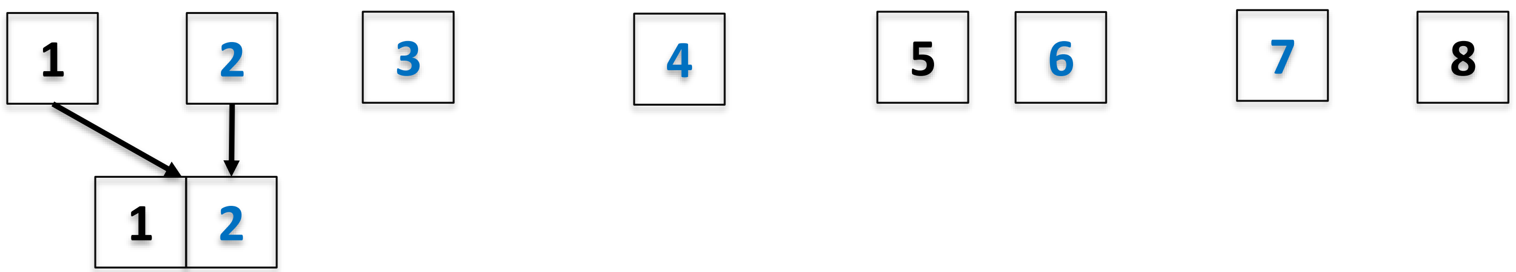
Quicksort - Example

Conquer

1	2	3	4	5	6	7	8

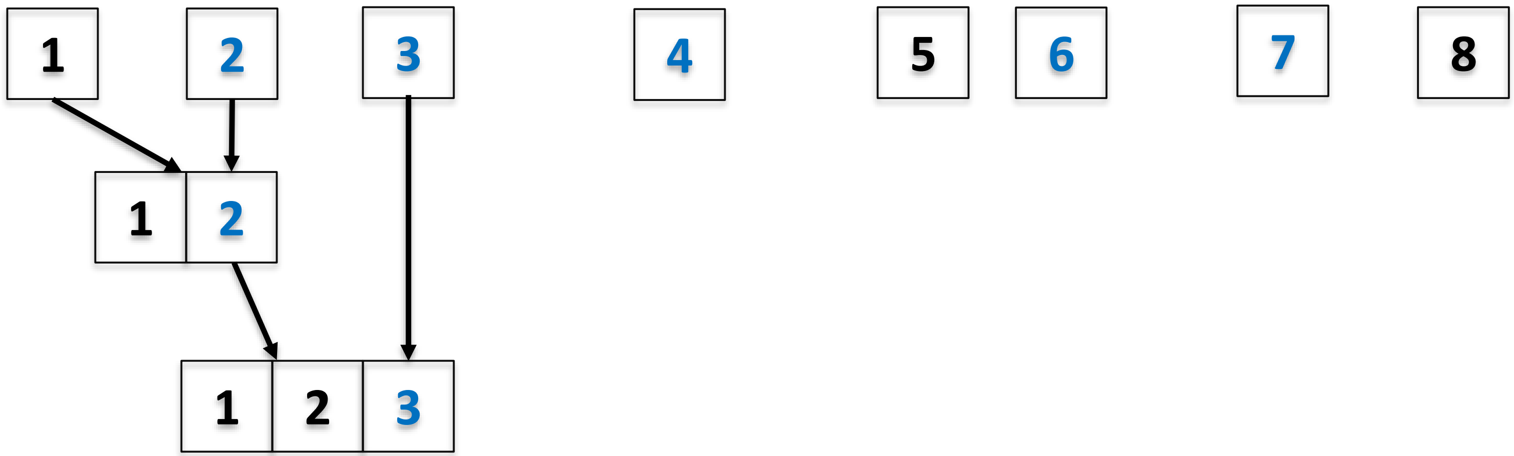
Quicksort - Example

Conquer



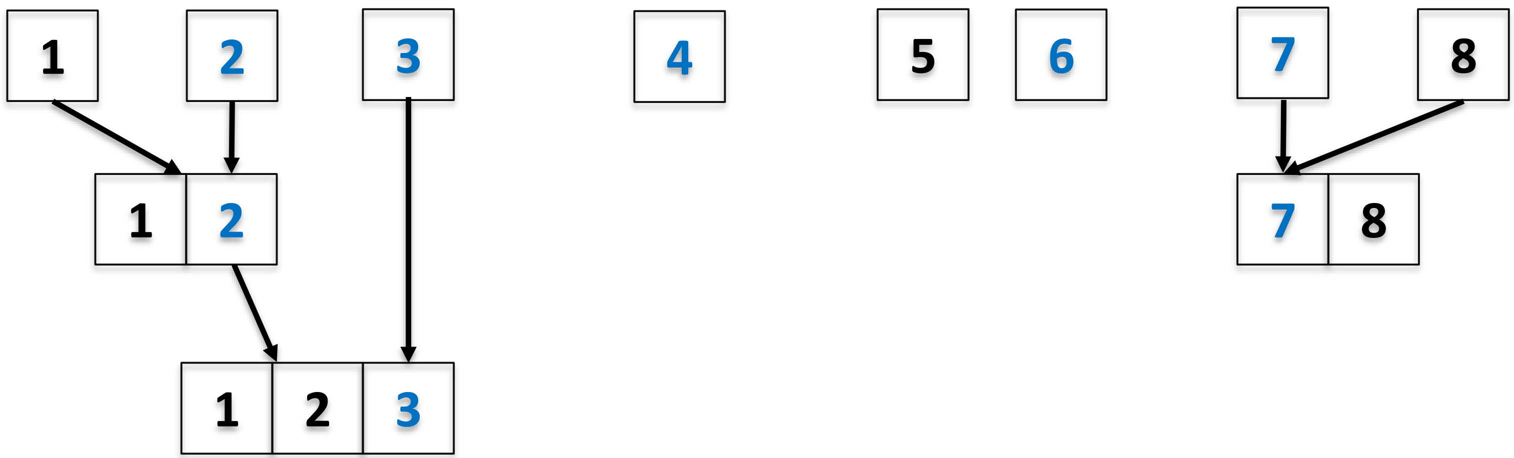
Quicksort - Example

Conquer



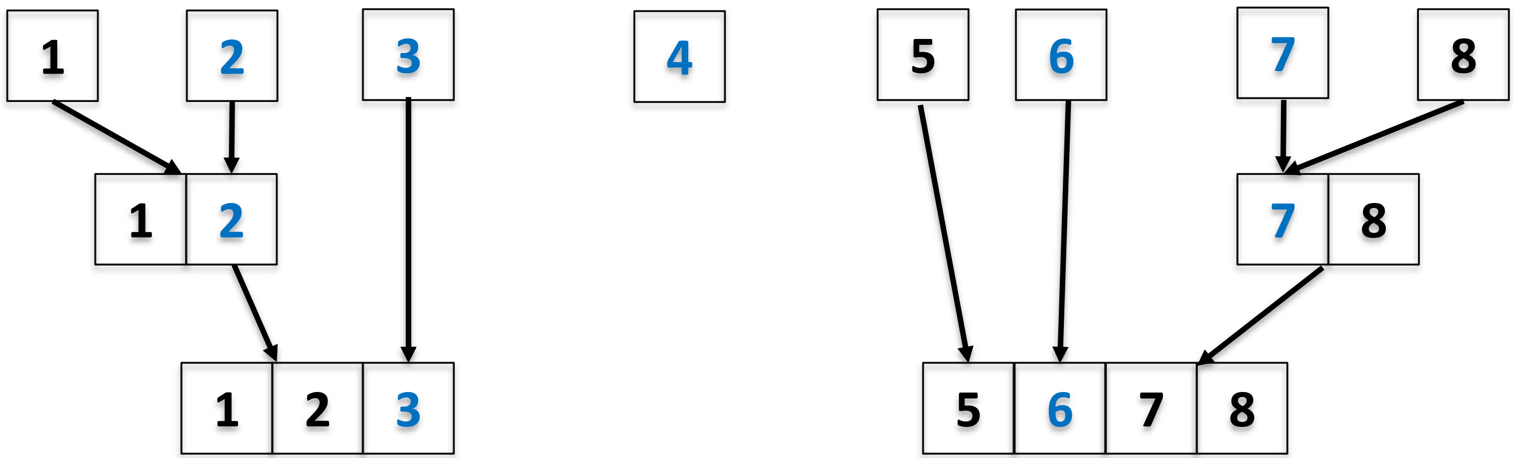
Quicksort - Example

Conquer



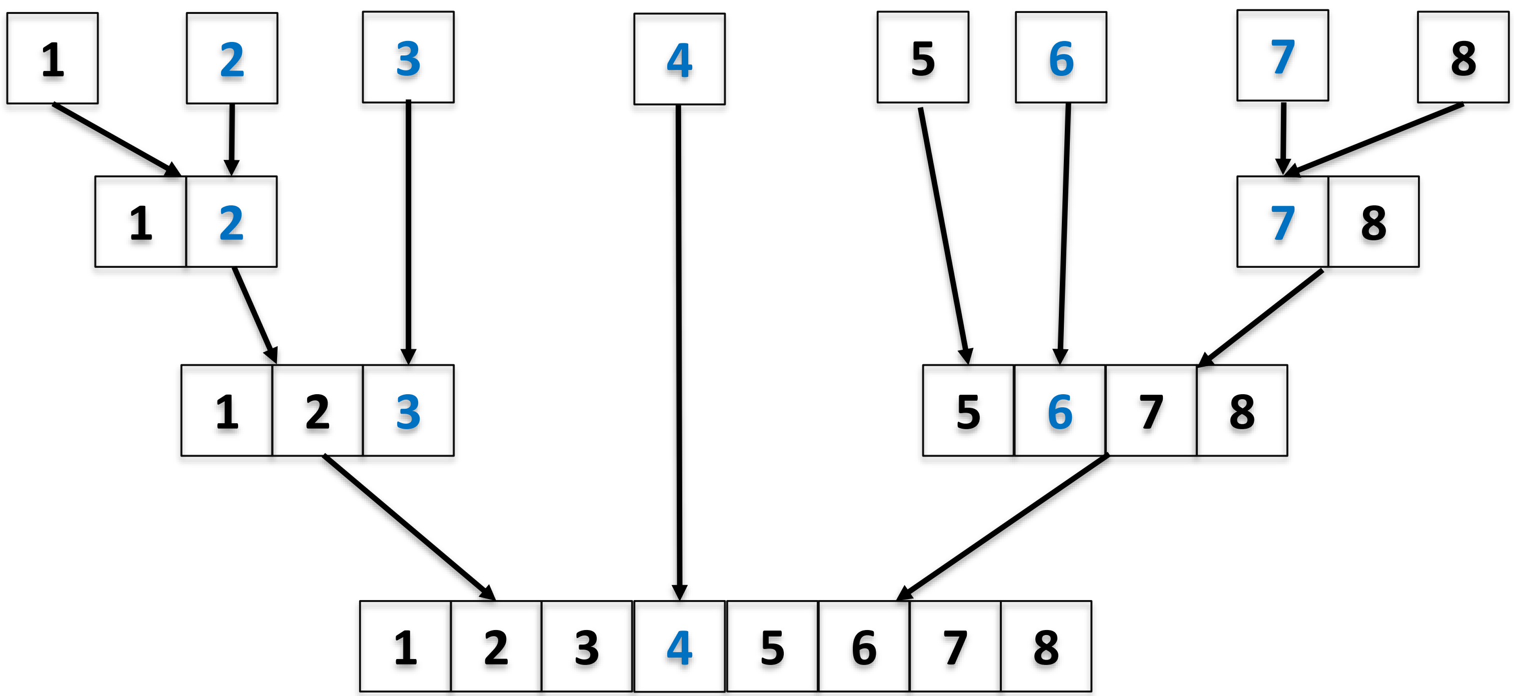
Quicksort - Example

Conquer



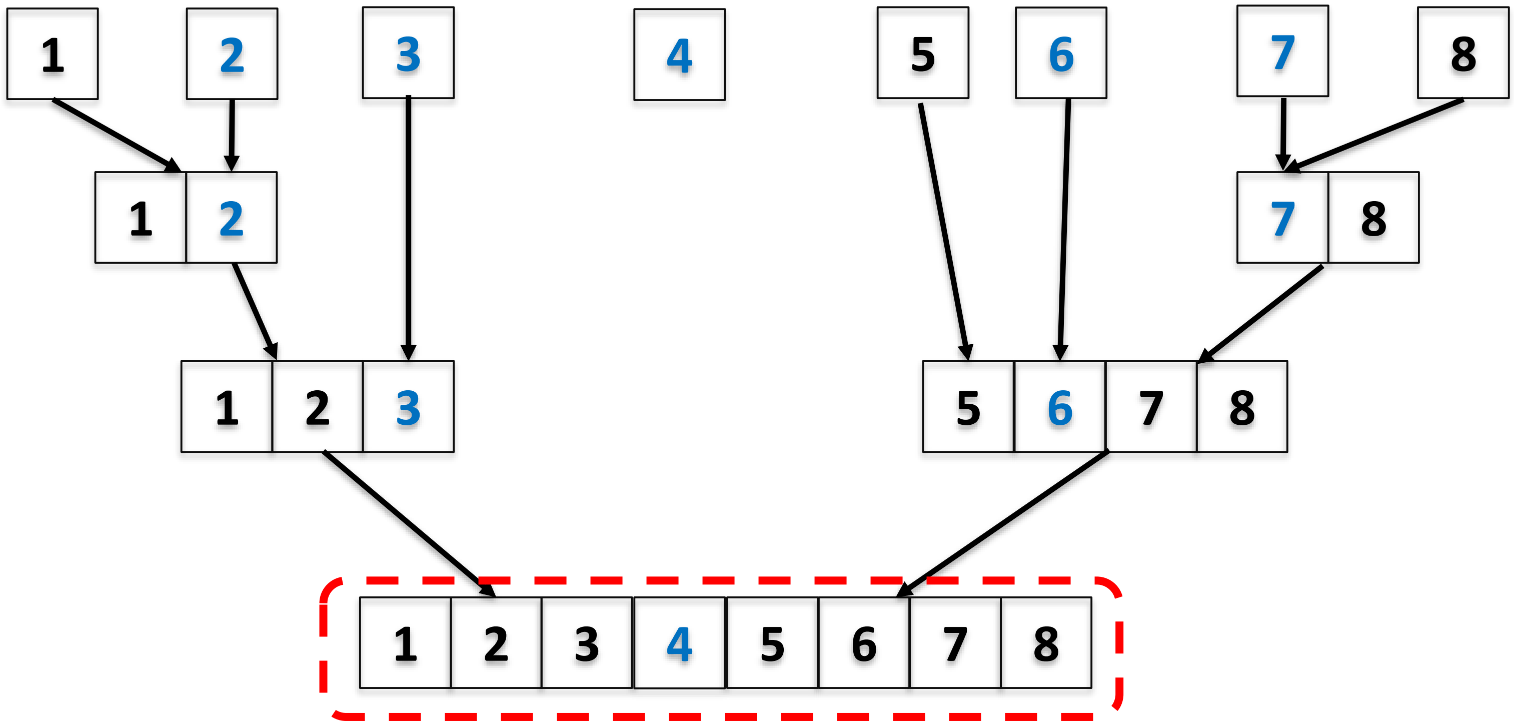
Quicksort - Example

Conquer



Quicksort - Example

Conquer



Outline

- Review to Divide-and-Conquer Paradigm
- Polynomial Multiplication Problem
  - Problem definition
  - A brute force algorithm
  - A first divide-and-conquer algorithm
  - An improved divide-and-conquer algorithm
  - Analysis of the divide-and-conquer algorithm
- Quicksort Problem
  - Basic partition
  - Randomized partition and randomized quicksort
  - Analysis of the randomized quicksort

Running Time of the Quicksort

- Measuring running time:

Running Time of the Quicksort

- Measuring running time:
  - The running time is dominated by the time spent in partition.

Running Time of the Quicksort

- Measuring running time:
  - The running time is dominated by the time spent in partition.
  - The running time of the partition procedure can be measured by the number of key comparisons.

Running Time of the Quicksort

- Measuring running time:
  - The running time is dominated by the time spent in partition.
  - The running time of the partition procedure can be measured by the number of key comparisons.
- T(n): running time on array of size n.
- Recurrence: $T(n) =$

Running Time of the Quicksort

- Measuring running time:
  - The running time is dominated by the time spent in partition.
  - The running time of the partition procedure can be measured by the number of key comparisons.
- T(n): running time on array of size n.
- Recurrence: $T(n) = T(m) +$

Running Time of the Quicksort

- Measuring running time:
  - The running time is dominated by the time spent in partition.
  - The running time of the partition procedure can be measured by the number of key comparisons.
- T(n): running time on array of size n.
- Recurrence: $T(n) = T(m) + T(n - m - 1) +$

Running Time of the Quicksort

- Measuring running time:
  - The running time is dominated by the time spent in partition.
  - The running time of the partition procedure can be measured by the number of key comparisons.
- T(n): running time on array of size n.
- Recurrence: $T(n) = T(m) + T(n - m - 1) + O(n)$

Running Time of the Quicksort

- Measuring running time:
  - The running time is dominated by the time spent in partition.
  - The running time of the partition procedure can be measured by the number of key comparisons.
- T(n): running time on array of size n.
- Recurrence: $T(n) = T(m) + T(n - m - 1) + O(n)$
- Worst Case:

Running Time of the Quicksort

- Measuring running time:
  - The running time is dominated by the time spent in partition.
  - The running time of the partition procedure can be measured by the number of key comparisons.
- T(n): running time on array of size n.
- Recurrence: $T(n) = T(m) + T(n - m - 1) + O(n)$
- Worst Case:

$$T(n) \;=\; T(0) \;+\; T(n-1) \;+\; O(n)$$

Running Time of the Quicksort

- Measuring running time:
  - The running time is dominated by the time spent in partition.
  - The running time of the partition procedure can be measured by the number of key comparisons.
- T(n): running time on array of size n.
- Recurrence: $T(n) = T(m) + T(n-m-1) + O(n)$
- Worst Case:

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

$$T(n) = O(\quad)$$

Running Time of the Quicksort

- Measuring running time:
  - The running time is dominated by the time spent in partition.
  - The running time of the partition procedure can be measured by the number of key comparisons.
- T(n): running time on array of size n.
- Recurrence: $T(n) = T(m) + T(n - m - 1) + O(n)$
- Worst Case:

$$T(n) = T(0) + T(n-1) + O(n)$$
$$T(n) = O(n^2)$$

Running Time of the Quicksort

- Measuring running time:
  - The running time is dominated by the time spent in partition.
  - The running time of the partition procedure can be measured by the number of key comparisons.
- T(n): running time on array of size n.
- Recurrence: $T(n) = T(m) + T(n - m - 1) + O(n)$
- Worst Case:

$$T(n) = T(0) + T(n-1) + O(n)$$
$$T(n) = O(n^2)$$

- What inputs give worst case performance?

Running Time of the Quicksort

- Measuring running time:
  - The running time is dominated by the time spent in partition.
  - The running time of the partition procedure can be measured by the number of key comparisons.
- T(n): running time on array of size n.
- Recurrence: $T(n) = T(m) + T(n-m-1) + O(n)$
- Worst Case:

$$T(n) = T(0) + T(n-1) + O(n)$$
$$T(n) = O(n^2)$$

- What inputs give worst case performance?
  - Whether performance is the worst is not determined by input.

Running Time of the Quicksort

- Measuring running time:
  - The running time is dominated by the time spent in partition.
  - The running time of the partition procedure can be measured by the number of key comparisons.
- T(n): running time on array of size n.
- Recurrence: $T(n) = T(m) + T(n-m-1) + O(n)$
- Worst Case:

$$T(n) = T(0) + T(n-1) + O(n)$$
$$T(n) = O(n^2)$$

- What inputs give worst case performance?
  - Whether performance is the worst is not determined by input.
  - An important property of randomized algorithms.

Running Time of the Quicksort

- Measuring running time:
  - The running time is dominated by the time spent in partition.
  - The running time of the partition procedure can be measured by the number of key comparisons.
- T(n): running time on array of size n.
- Recurrence: $T(n) = T(m) + T(n-m-1) + O(n)$
- Worst Case:

$$T(n) = T(0) + T(n-1) + O(n)$$
$$T(n) = O(n^2)$$

- What inputs give worst case performance?
  - Whether performance is the worst is not determined by input.
  - An important property of randomized algorithms.
  - Worst case performance results only if the random number generator always produces the worst choice.

Randomized Algorithms

- Analysis for Randomized Algorithms:
  - Worst-case doesn't make sense: for any given input, the worst case is very unlikely to happen.

Randomized Algorithms

- Analysis for Randomized Algorithms:
  - Worst-case doesn't make sense: for any given input, the worst case is very unlikely to happen.
  - Use expected running time analysis for randomized algorithms!

Randomized Algorithms

- Analysis for Randomized Algorithms:
  - Worst-case doesn't make sense: for any given input, the worst case is very unlikely to happen.
  - Use expected running time analysis for randomized algorithms!

Average case analysis	Expected case analysis
• Used for deterministic algorithms	• Used for randomized algorithms

Randomized Algorithms

- Analysis for Randomized Algorithms:
  - Worst-case doesn't make sense: for any given input, the worst case is very unlikely to happen.
  - Use expected running time analysis for randomized algorithms!

Average case analysis

- Used for deterministic algorithms
- Assume the input is chosen randomly from some distribution

Expected case analysis

- Used for randomized algorithms
- Need to work for any given input

Randomized Algorithms

- Analysis for Randomized Algorithms:
  - Worst-case doesn't make sense: for any given input, the worst case is very unlikely to happen.
  - Use expected running time analysis for randomized algorithms!

Average case analysis

- Used for deterministic algorithms
- Assume the input is chosen randomly from some distribution
- Depends on assumptions on the input, weaker

Expected case analysis

- Used for randomized algorithms
- Need to work for any given input
- Randomization is inherent within the algorithm, stronger

Randomized Algorithms

- Analysis for Randomized Algorithms:
  - Worst-case doesn't make sense: for any given input, the worst case is very unlikely to happen.
  - Use expected running time analysis for randomized algorithms!

Average case analysis

- Used for deterministic algorithms
- Assume the input is chosen randomly from some distribution
- Depends on assumptions on the input, weaker
- Not required in this course

Expected case analysis

- Used for randomized algorithms
- Need to work for any given input
- Randomization is inherent within the algorithm, stronger
- Required in this course

Expected Case

- Two methods to analyze the expected running time of a divide-and-conquer randomized algorithm:
  - Old fashioned: Write our a recurrence on T(n), where T(n) is the expected running time of the algorithm on an input of size n, and solve it.
    —— (Almost) always works but needs complicated maths.

Expected Case

- Two methods to analyze the expected running time of a divide-and-conquer randomized algorithm:
  - Old fashioned: Write our a recurrence on T(n), where T(n) is the expected running time of the algorithm on an input of size n, and solve it.
    —— (Almost) always works but needs complicated maths.
  - New: Indicator variables.
    —— Simple and elegant, but needs practice to master.

Expected Case

- Two facts about key comparisons:

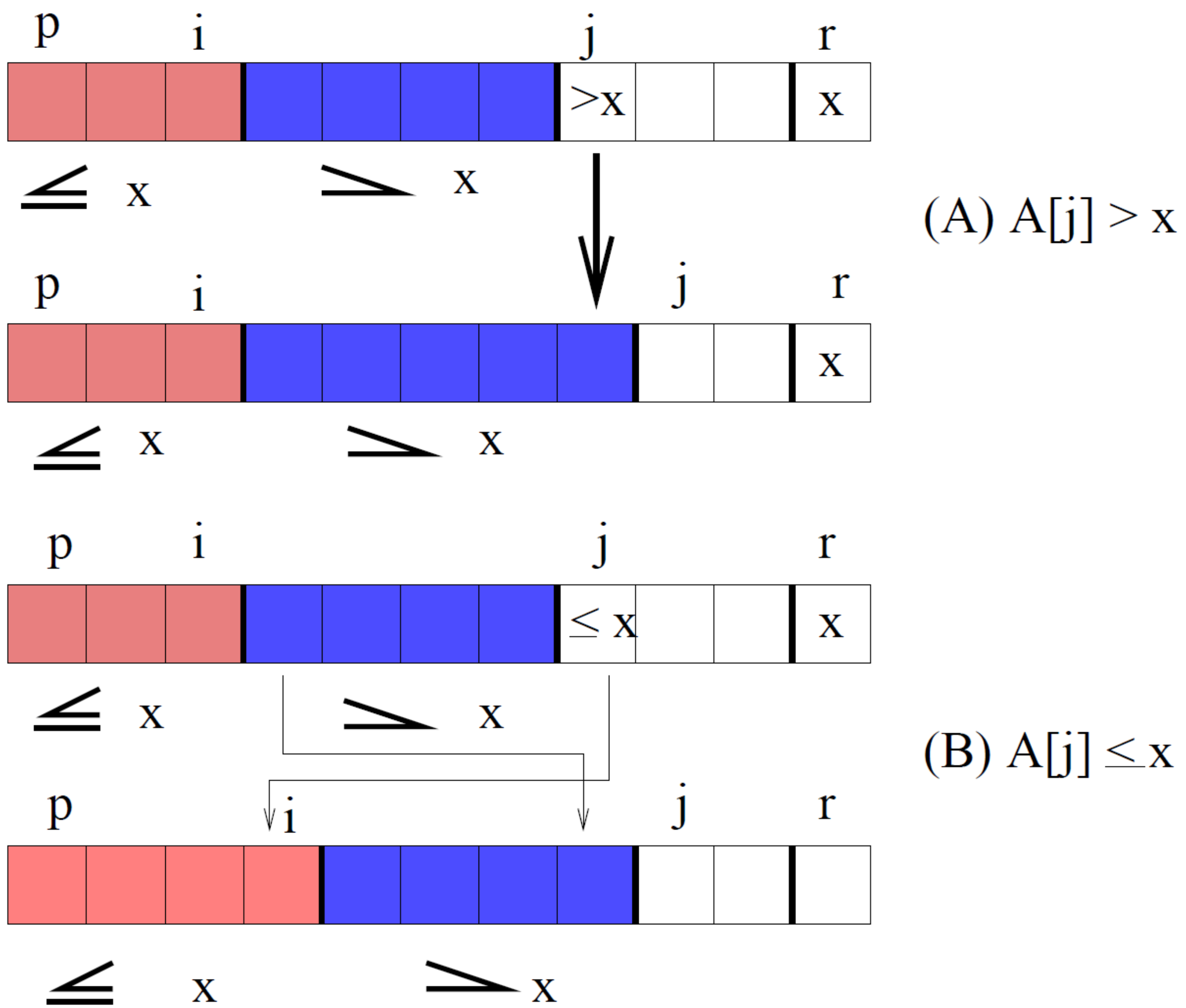
Expected Case

- Two facts about key comparisons:
  - When a pivot is selected, the pivot is compared with every other elements, then the elements are partitioned into two parts accordingly

Expected Case

- Two facts about key comparisons:
  - When a pivot is selected, the pivot is compared with every other elements, then the elements are partitioned into two parts accordingly
  - Elements in different partitions are never compared with each other in all operations

Expected Case



Expected Running Time

- Expected Running Time of Randomized-Quicksort:
  - Let $z_1 < z_2 < \cdots < z_n$ be the n elements in sorted order

Expected Running Time

- Expected Running Time of Randomized-Quicksort:
  - Let $z_1 < z_2 < \cdots < z_n$ be the n elements in sorted order
  - X: total number of comparisons performed in all calls to randomized-partition

Expected Running Time

- Expected Running Time of Randomized-Quicksort:
  - Let $z_1 < z_2 < \cdots < z_n$ be the n elements in sorted order
  - X: total number of comparisons performed in all calls to randomized-partition
  - X_{ij} : number of comparisons between z_i and z_j

Expected Running Time

- Expected Running Time of Randomized-Quicksort:
  - Let $z_1 < z_2 < \cdots < z_n$ be the n elements in sorted order
  - X: total number of comparisons performed in all calls to randomized-partition
  - X_{ij} : number of comparisons between z_i and z_j
    - can only be 0 or 1

Expected Running Time

- Expected Running Time of Randomized-Quicksort:
  - Let $z_1 < z_2 < \cdots < z_n$ be the n elements in sorted order
  - X: total number of comparisons performed in all calls to randomized-partition
  - X_{ij} : number of comparisons between z_i and z_j
    - can only be 0 or 1

$$E[X] =$$

Expected Running Time

- Expected Running Time of Randomized-Quicksort:
  - Let $z_1 \ < z_2 < \cdots < z_n$ be the n elements in sorted order
  - X: total number of comparisons performed in all calls to randomized-partition
  - X_{ij} : number of comparisons between z_i and z_j
    - can only be 0 or 1

$$E[X] = E\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] =$$

Expected Running Time

- Expected Running Time of Randomized-Quicksort:
  - Let $z_1 < z_2 < \cdots < z_n$ be the n elements in sorted order
  - X: total number of comparisons performed in all calls to randomized-partition
  - X_{ij} : number of comparisons between z_i and z_j
    - can only be 0 or 1

$$E[X] = E\left[\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} X_{ij}\right] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \mathrm{E}[X_{ij}]$$

$$=$$

Expected Running Time

- Expected Running Time of Randomized-Quicksort:
  - Let $z_1 < z_2 < \cdots < z_n$ be the n elements in sorted order
  - X: total number of comparisons performed in all calls to randomized-partition
  - X_{ij} : number of comparisons between z_i and z_j
    - can only be 0 or 1

$$E[X] = E\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathrm{E}[X_{ij}]$$

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} [\Pr\{$$

Expected Running Time

- Expected Running Time of Randomized-Quicksort:
  - Let $z_1 < z_2 < \cdots < z_n$ be the n elements in sorted order
  - X: total number of comparisons performed in all calls to randomized-partition
  - X_{ij} : number of comparisons between z_i and z_j
    - can only be 0 or 1

$$E[X] = E\left[\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} X_{ij}\right] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \mathrm{E}[X_{ij}]$$

$$= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} [\Pr\{z_i \ is \ compared \ with \ z_j\} \times 1$$

$$+ \Pr\{z_i \ is \ not \ compared \ with \ z_j\} \times 0]$$

Expected Running Time

- Expected Running Time of Randomized-Quicksort:
  - Let $z_1 < z_2 < \cdots < z_n$ be the n elements in sorted order
  - X: total number of comparisons performed in all calls to randomized-partition
  - X_{ij} : number of comparisons between z_i and z_j
    - can only be 0 or 1

$$E[X] = E\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathrm{E}[X_{ij}]$$

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} [\Pr\{z_i \textit{ is compared with } z_j\} \times 1$$

$$+ \Pr\{z_i \textit{ is not compared with } z_j\} \times 0]$$

$$=$$

Expected Running Time

- Expected Running Time of Randomized-Quicksort:
  - Let $z_1 < z_2 < \cdots < z_n$ be the n elements in sorted order
  - X: total number of comparisons performed in all calls to randomized-partition
  - X_{ij} : number of comparisons between z_i and z_j
    - can only be 0 or 1

$$E[X] = E\left[\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} X_{ij}\right] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \mathrm{E}[X_{ij}]$$

$$= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} [\Pr\{z_i \textit{ is compared with } z_j\} \times 1$$

$$+ \Pr\{z_i \textit{ is not compared with } z_j\} \times 0]$$

$$= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \Pr\{z_i \textit{ is compared with } z_j\}$$

Observations

For $1 \leq i \leq j \leq n$, let $Z_{ij} = \{z_i, z_{i+1}, \ldots, z_j\}$

- remember $z_i < z_{i+1} < \cdots < z_j$

Observations

For $1 \le i \le j \le n$, let $Z_{ij} = \{z_i, z_{i+1}, \ldots, z_j\}$

- remember $z_i < z_{i+1} < \cdots < z_j$

Observations:

- Partition divides an array into three segments, left, pivot, and Right.
- When z_i and z_j are first placed in DIFFERENT segments of the array by partitioning, the pivot is one of the elements in Z_{ij}

Observations

For $1 \le i \le j \le n$, let $Z_{ij} = \{z_i, z_{i+1}, \dots, z_j\}$

- remember $z_i < z_{i+1} < \cdots < z_j$

Observations:

- Partition divides an array into three segments, left, pivot, and Right.
- When z_i and z_j are first placed in DIFFERENT segments of the array by partitioning, the pivot is one of the elements in Z_{ij}
- If the pivot is either z_i or z_j

Observations

For $1 \le i \le j \le n$, let $Z_{ij} = \{z_i, z_{i+1}, \ldots, z_j\}$

- remember $z_i < z_{i+1} < \cdots < z_j$

Observations:

- Partition divides an array into three segments, left, pivot, and Right.
- When z_i and z_j are first placed in DIFFERENT segments of the array by partitioning, the pivot is one of the elements in Z_{ij}
- If the pivot is either z_i or z_j
  - z_i and z_j will be compared

Observations

For $1 \leq i \leq j \leq n$, let $Z_{ij} = \{z_i, z_{i+1}, \dots, z_j\}$

- remember $z_i < z_{i+1} < \cdots < z_j$

Observations:

- Partition divides an array into three segments, left, pivot, and Right.
- When z_i and z_j are first placed in DIFFERENT segments of the array by partitioning, the pivot is one of the elements in Z_{ij}
- If the pivot is either z_i or z_j
  - z_i and z_j will be compared
- If the pivot is any element in Z_{ij} other than z_i or z_j
  - z_i and z_j are not compared with each other in all randomized-partition calls

How to Find Pr{z_i is compared with z_j}?

$$\Pr\left\{z_i \ is \ compared \ with \ z_j\right\}$$

$$=$$

How to Find $\Pr\{z_i$ is compared with $z_j\}$?

$$\Pr\{z_i \text{ is compared with } z_j\}$$

$$= \Pr\{z_i \text{ or } z_j \text{ is the first pivot chosen from } Z_{ij}\}$$

$$=$$

How to Find Pr{z_i is compared with z_j}?

$$\Pr\{z_i \textit{ is compared with } z_j\}$$

$$= \Pr\{z_i \text{ or } z_j \text{ is the first pivot chosen from } Z_{ij}\}$$

$$= \Pr\{z_i \text{ is the first pivot chosen from } Z_{ij}\}$$

$$+$$

How to Find Pr{z_i is compared with z_j}?

$$\Pr\{z_i \; is \; compared \; with \; z_j\}$$

$$= \Pr\{z_i \text{ or } z_j \text{ is the first pivot chosen from } Z_{ij}\}$$

$$= \Pr\{z_i \text{ is the first pivot chosen from } Z_{ij}\}$$

$$+ \Pr\{z_j \text{ is the first pivot chosen from } Z_{ij}\}$$

$$= \frac{1}{j-i+1} +$$

How to Find Pr{z_i is compared with z_j}?

$$
\begin{aligned}
&\Pr\{z_i \textit{ is compared with } z_j\} \\
&\quad = \Pr\{z_i \text{ or } z_j \text{ is the first pivot chosen from } Z_{ij}\} \\
&\quad = \Pr\{z_i \text{ is the first pivot chosen from } Z_{ij}\} \\
&\qquad + \Pr\{z_j \text{ is the first pivot chosen from } Z_{ij}\} \\
&\quad = \frac{1}{j-i+1} + \frac{1}{j-i+1} =
\end{aligned}
$$

How to Find Pr{z_i is compared with z_j}?

$$
\begin{aligned}
&\Pr\{z_i \textit{ is compared with } z_j\} \\
&\quad = \Pr\{z_i \text{ or } z_j \text{ is the first pivot chosen from } Z_{ij}\} \\
&\quad = \Pr\{z_i \text{ is the first pivot chosen from } Z_{ij}\} \\
&\qquad + \Pr\{z_j \text{ is the first pivot chosen from } Z_{ij}\} \\
&\quad = \frac{1}{j-i+1} + \frac{1}{j-i+1} = \frac{2}{j-i+1}
\end{aligned}
$$

How to Find Pr{z_i is compared with z_j}?

$$\Pr\{z_i \text{ is compared with } z_j\}$$

$$= \Pr\{z_i \text{ or } z_j \text{ is the first pivot chosen from } Z_{ij}\}$$

$$= \Pr\{z_i \text{ is the first pivot chosen from } Z_{ij}\}$$

$$+ \Pr\{z_j \text{ is the first pivot chosen from } Z_{ij}\}$$

$$= \frac{1}{j-i+1} + \frac{1}{j-i+1} = \frac{2}{j-i+1}$$

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr\{z_i \text{ is compared with } z_j\} =$$

How to Find Pr{z_i is compared with z_j}?

$$\Pr\{z_i \textit{ is compared with } z_j\}$$
$$= \Pr\{z_i \text{ or } z_j \text{ is the first pivot chosen from } Z_{ij}\}$$
$$= \Pr\{z_i \text{ is the first pivot chosen from } Z_{ij}\}$$
$$+ \Pr\{z_j \text{ is the first pivot chosen from } Z_{ij}\}$$
$$= \frac{1}{j-i+1} + \frac{1}{j-i+1} = \frac{2}{j-i+1}$$

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr\{z_i \textit{ is compared with } z_j\} = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$$

$$=$$

How to Find Pr{z_i is compared with z_j}?

$$
\begin{aligned}
&\Pr\{z_i \textit{ is compared with } z_j\} \\
&\quad = \Pr\{z_i \text{ or } z_j \text{ is the first pivot chosen from } Z_{ij}\} \\
&\quad = \Pr\{z_i \text{ is the first pivot chosen from } Z_{ij}\} \\
&\qquad + \Pr\{z_j \text{ is the first pivot chosen from } Z_{ij}\} \\
&\quad = \frac{1}{j-i+1} + \frac{1}{j-i+1} = \frac{2}{j-i+1}
\end{aligned}
$$

$$
E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr\{z_i \textit{ is compared with } z_j\} = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}
$$

$$
= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1}
$$

How to Find Pr{z_i is compared with z_j}?

$$\Pr\{z_i \; is \; compared \; with \; z_j\}$$

$$= \Pr\{z_i \text{ or } z_j \text{ is the first pivot chosen from } Z_{ij}\}$$

$$= \Pr\{z_i \text{ is the first pivot chosen from } Z_{ij}\}$$

$$+ \Pr\{z_j \text{ is the first pivot chosen from } Z_{ij}\}$$

$$= \frac{1}{j-i+1} + \frac{1}{j-i+1} = \frac{2}{j-i+1}$$

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr\{z_i \; is \; compared \; with \; z_j\} = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$$

$$= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} < \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k}$$

How to Find Pr{z_i is compared with z_j}?

$$\Pr\{z_i \text{ is compared with } z_j\}$$
$$= \Pr\{z_i \text{ or } z_j \text{ is the first pivot chosen from } Z_{ij}\}$$
$$= \Pr\{z_i \text{ is the first pivot chosen from } Z_{ij}\}$$
$$+ \Pr\{z_j \text{ is the first pivot chosen from } Z_{ij}\}$$
$$= \frac{1}{j-i+1} + \frac{1}{j-i+1} = \frac{2}{j-i+1}$$

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr\{z_i \text{ is compared with } z_j\} = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$$
$$= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} < \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} = \sum_{i=1}^{n-1} O(\log n)$$

Note: $\sum_{k=1}^{n} \frac{1}{k} \le \log(n)$

How to Find Pr{z_i is compared with z_j}?

$$\Pr\{z_i \textit{ is compared with } z_j\}$$
$$= \Pr\{z_i \text{ or } z_j \text{ is the first pivot chosen from } Z_{ij}\}$$
$$= \Pr\{z_i \text{ is the first pivot chosen from } Z_{ij}\}$$
$$+ \Pr\{z_j \text{ is the first pivot chosen from } Z_{ij}\}$$
$$= \frac{1}{j-i+1} + \frac{1}{j-i+1} = \frac{2}{j-i+1}$$

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr\{z_i \textit{ is compared with } z_j\} = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$$
$$= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} < \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} = \sum_{i=1}^{n-1} O(\log n) = O(n \log n)$$

Note: $\sum_{k=1}^{n} \frac{1}{k} \leq \log(n)$

How to Find Pr{z_i is compared with z_j}?

$$
\begin{aligned}
&\Pr\{z_i \textit{ is compared with } z_j\} \\
&\quad = \Pr\{z_i \text{ or } z_j \text{ is the first pivot chosen from } Z_{ij}\} \\
&\quad = \Pr\{z_i \text{ is the first pivot chosen from } Z_{ij}\} \\
&\qquad + \Pr\{z_j \text{ is the first pivot chosen from } Z_{ij}\} \\
&\quad = \frac{1}{j-i+1} + \frac{1}{j-i+1} = \frac{2}{j-i+1}
\end{aligned}
$$

$$
E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr\{z_i \textit{ is compared with } z_j\} = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}
$$

$$
= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} < \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} = \sum_{i=1}^{n-1} O(\log n) = O(n \log n)
$$

Note: $\sum_{k=1}^{n} \frac{1}{k} \le \log(n)$

Hence, the expected number of comparisons is $O(n \log n)$, which is the expected running time of Randomized-Quicksort


dank u
ju faleminderit
Tack
谢谢
Tak
mulţumesc
Asante
kiitos
Gracias
Salamat!
Terima kasih
Aliquam
Merci
Dankie
Obrigado
ありがとう
köszönöm
grazie
Aliquam
Go raibh maith agat
děkuii
Thank you
gam