# CS6161: Design and Analysis of Algorithms (Fall 2020)

# Divide and Conquer (I)

Instructor: Haifeng Xu

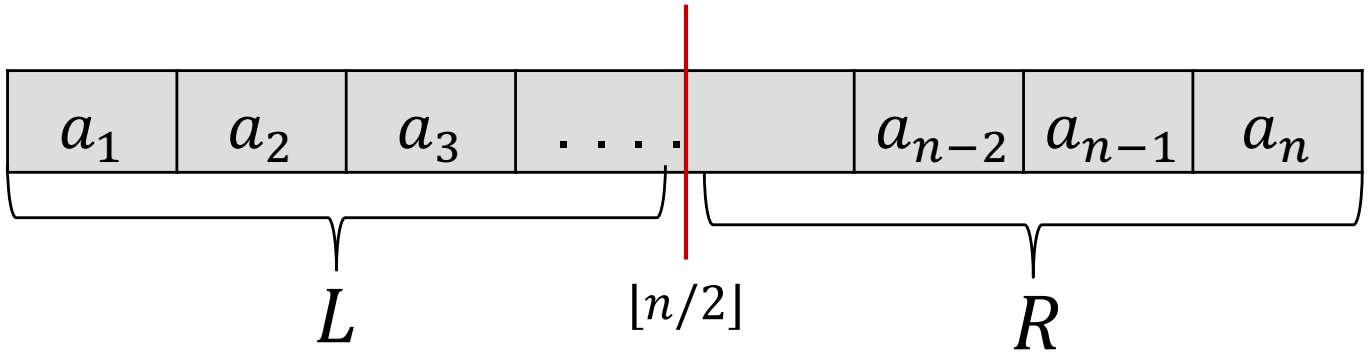
# Divide and Conquer

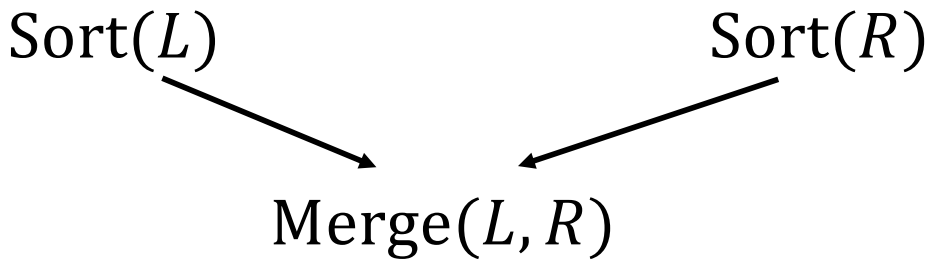
- Divide: break a hard problem into smaller pieces
- Conquer: solve each piece and then "assemble" their solutions to construct a solution for the original problem

# Outline

- Merge Sort
- The Master Theorem
- Matrix Multiplication

# Merge Sort: Divide and Conquer for Sorting



# Pseudo-code

MERGE-SORT$(A, l, r)$ // Sort $A[l : r]$

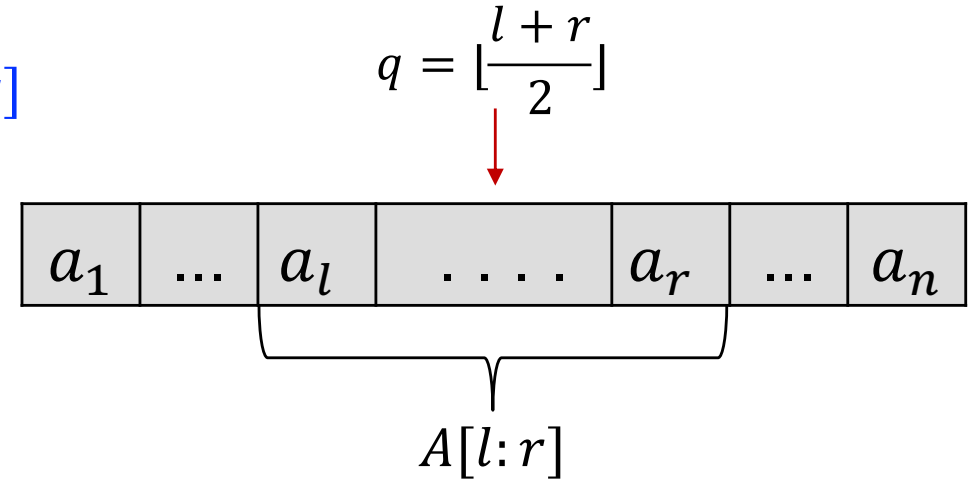
1 **if** $l = r$ **return();**

2 **else** $q = \lfloor \frac{l+r}{2} \rfloor$

3 MERGE-SORT$(A, l, q)$

4 MERGE-SORT$(A, q + 1, r)$

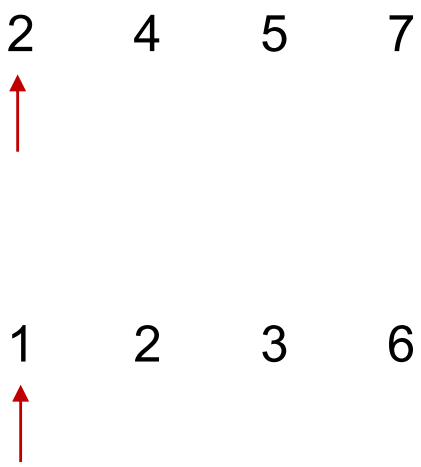
5 MERGE$(A, l, q, r)$




This is a recursive algorithm

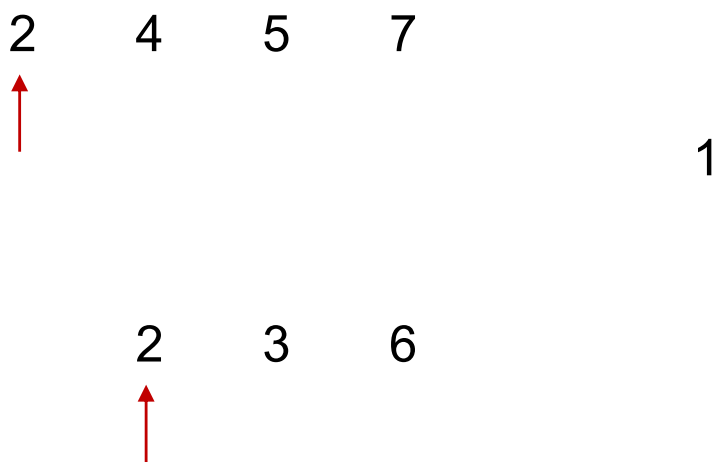
# How to Merge Two Sorted Arrays?

Idea: compare two front numbers, add the smaller one to a to-be-filled array



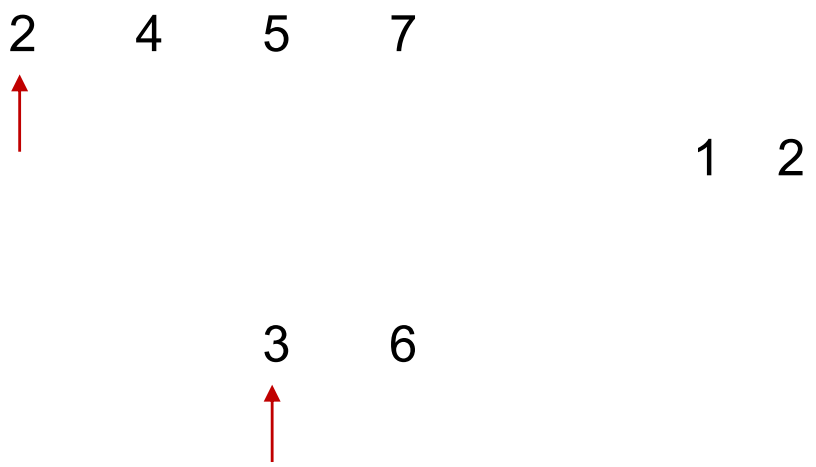
# How to Merge Two Sorted Arrays?

Idea: compare two front numbers, add the smaller one to a to-be-filled array



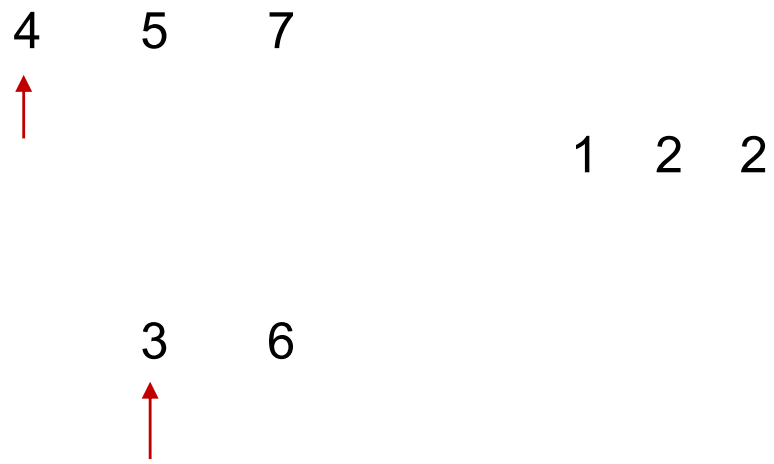
# How to Merge Two Sorted Arrays?

Idea: compare two front numbers, add the smaller one to a to-be-filled array



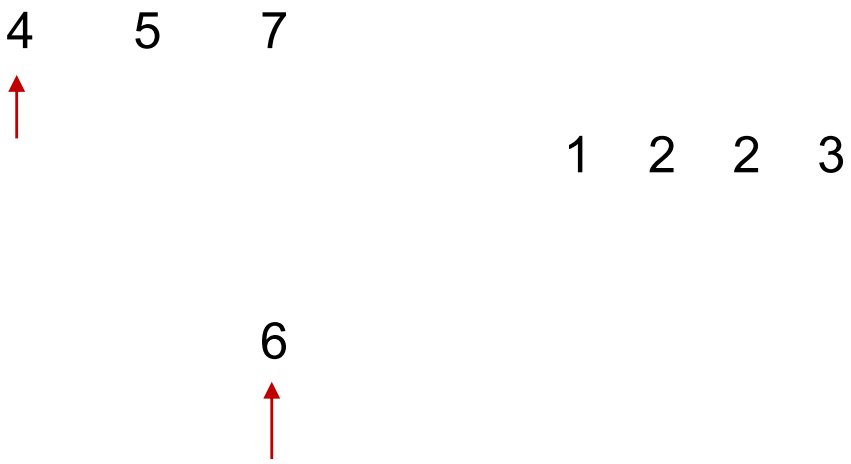
# How to Merge Two Sorted Arrays?

Idea: compare two front numbers, add the smaller one to a to-be-filled array



# How to Merge Two Sorted Arrays?

Idea: compare two front numbers, add the smaller one to a to-be-filled array



# How to Merge Two Sorted Arrays?

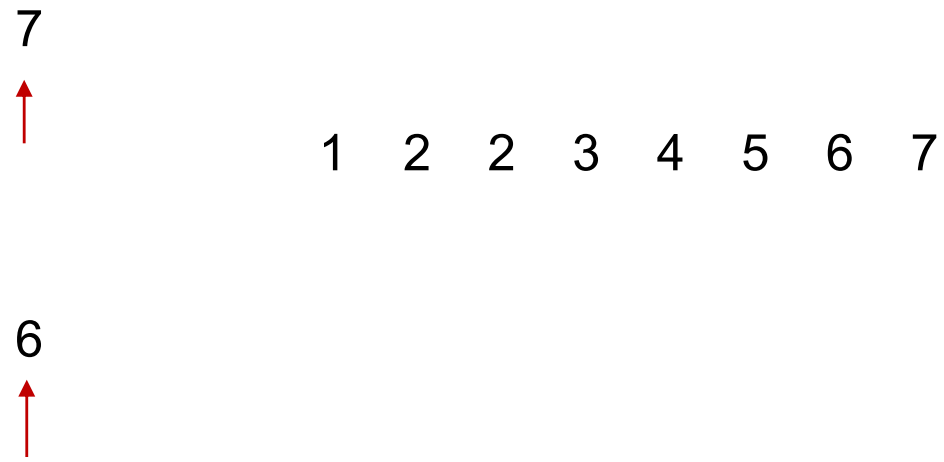
Idea: compare two front numbers, add the smaller one to a to-be-filled array

# How to Merge Two Sorted Arrays?

Idea: compare two front numbers, add the smaller one to a to-be-filled array

# How to Merge Two Sorted Arrays?

Idea: compare two front numbers, add the smaller one to a to-be-filled array




Running time: $O(n)$

# Pseudo-code for Merging

MERGE $(A, l, q, r)$
// Merge sorted $A[l : q]$ and $A[q + 1 : r]$

1

2

3

4

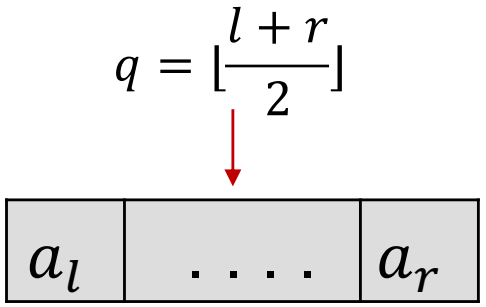
5

6

7

8

9



# Pseudo-code for Merging

MERGE $(A, l, q, r)$
// Merge sorted $A[l : q]$ and $A[q + 1 : r]$

1 Build array $L = A[l : q]$ and $R = A[q + 1 : r]$

2 $L.\text{append}(\infty)$, $R.\text{append}(\infty)$,

3

4

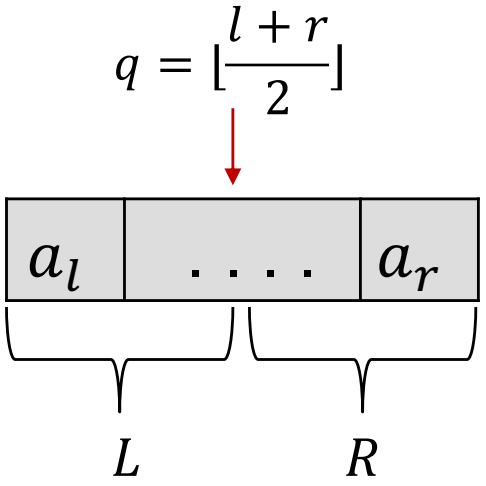
5

6

7

8

9



# Pseudo-code for Merging

MERGE $(A, l, q, r)$
// Merge sorted $A[l : q]$ and $A[q + 1 : r]$

1 Build array $L = A[l : q]$ and $R = A[q + 1 : r]$

2 $L.\text{append}(\infty)$, $R.\text{append}(\infty)$,

3 $i = 1, j = 1$

4

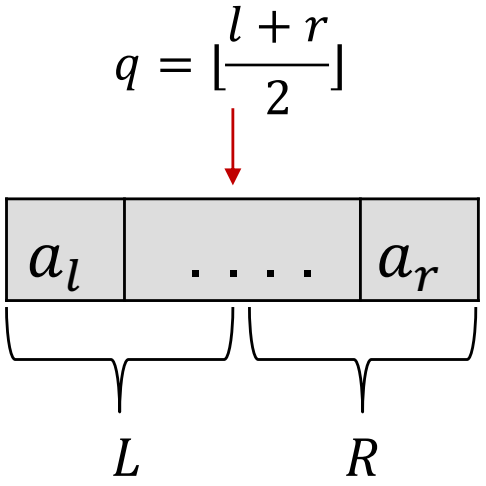
5

6

7

8

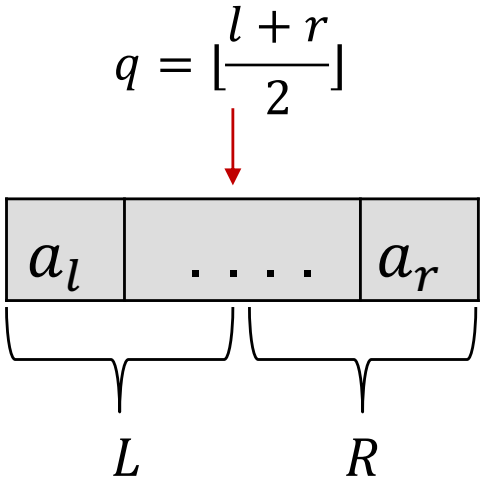
9



# Pseudo-code for Merging

MERGE $(A, l, q, r)$
// Merge sorted $A[l : q]$ and $A[q + 1 : r]$

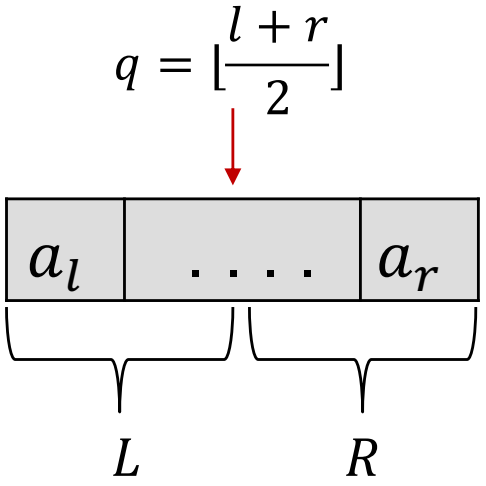
1 Build array $L = A[l : q]$ and $R = A[q + 1 : r]$
2 $L.\text{append}(\infty)$, $R.\text{append}(\infty)$,
3 $i = 1, j = 1$
4 for $k = l$ to $r$
5 if $L[i] \leq R(j)$
6 $A[k] = L(i)$
7 $i = i + 1$
8
9



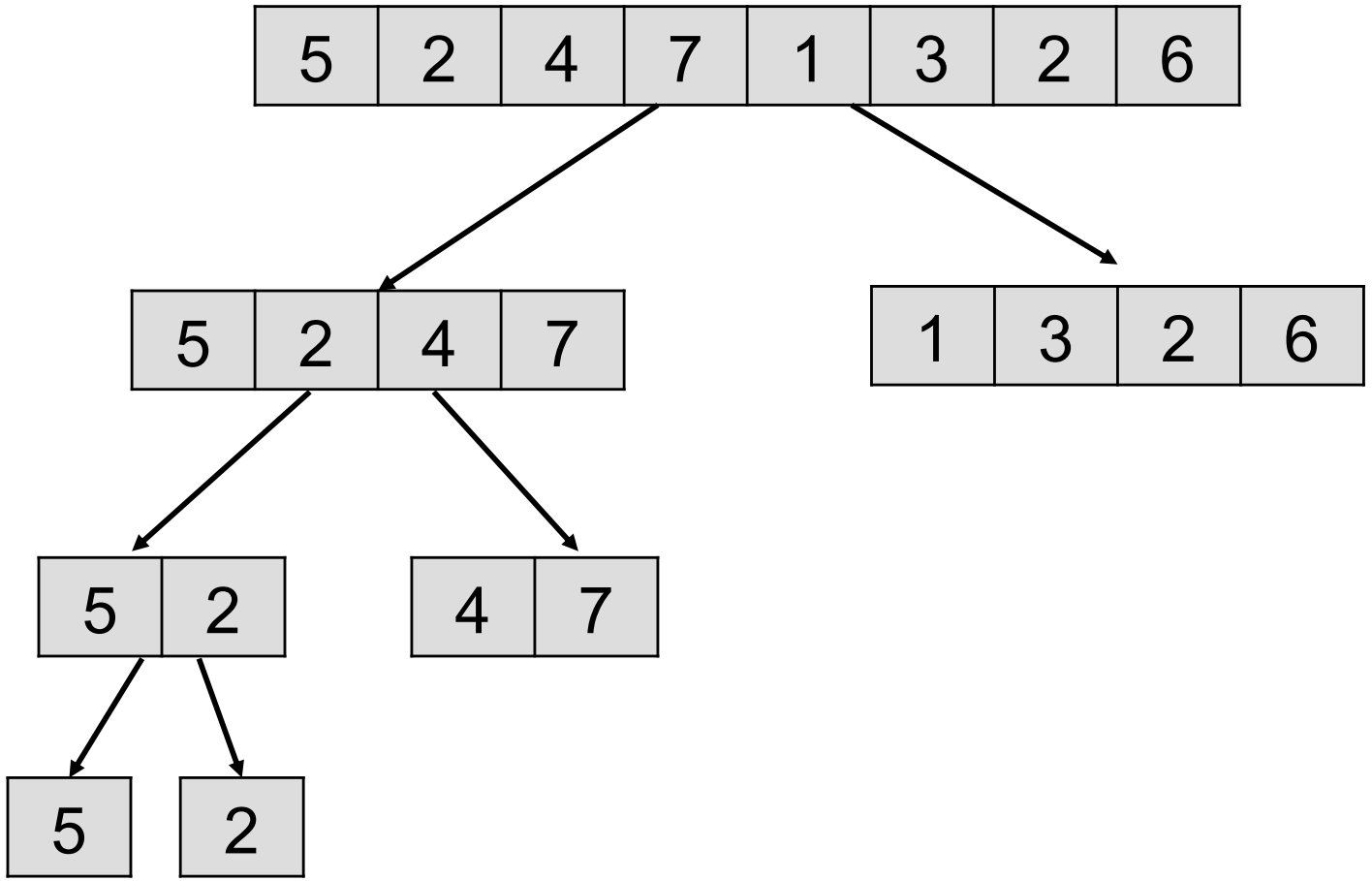
# Pseudo-code for Merging

MERGE $(A, l, q, r)$
// Merge sorted $A[l : q]$ and $A[q + 1 : r]$

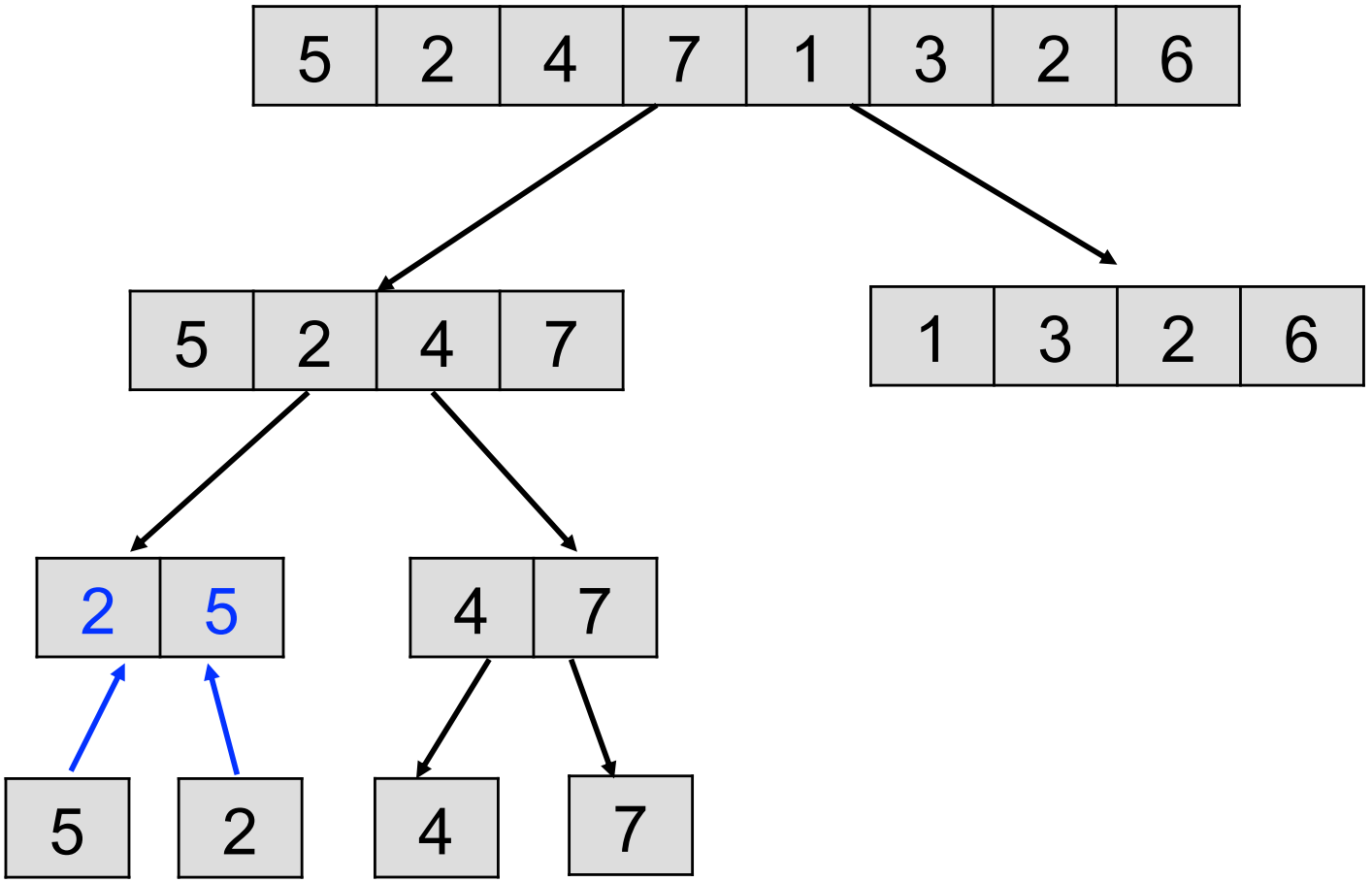
1 Build array $L = A[l : q]$ and $R = A[q + 1 : r]$
2 $L.\text{append}(\infty)$, $R.\text{append}(\infty)$,
3 $i = 1, j = 1$
4 for $k = l$ to $r$
5 if $L[i] \leq R(j)$
6 $A[k] = L(i)$
7 $i = i + 1$
8 else $A[k] = R(j)$
9 $j = j + 1$



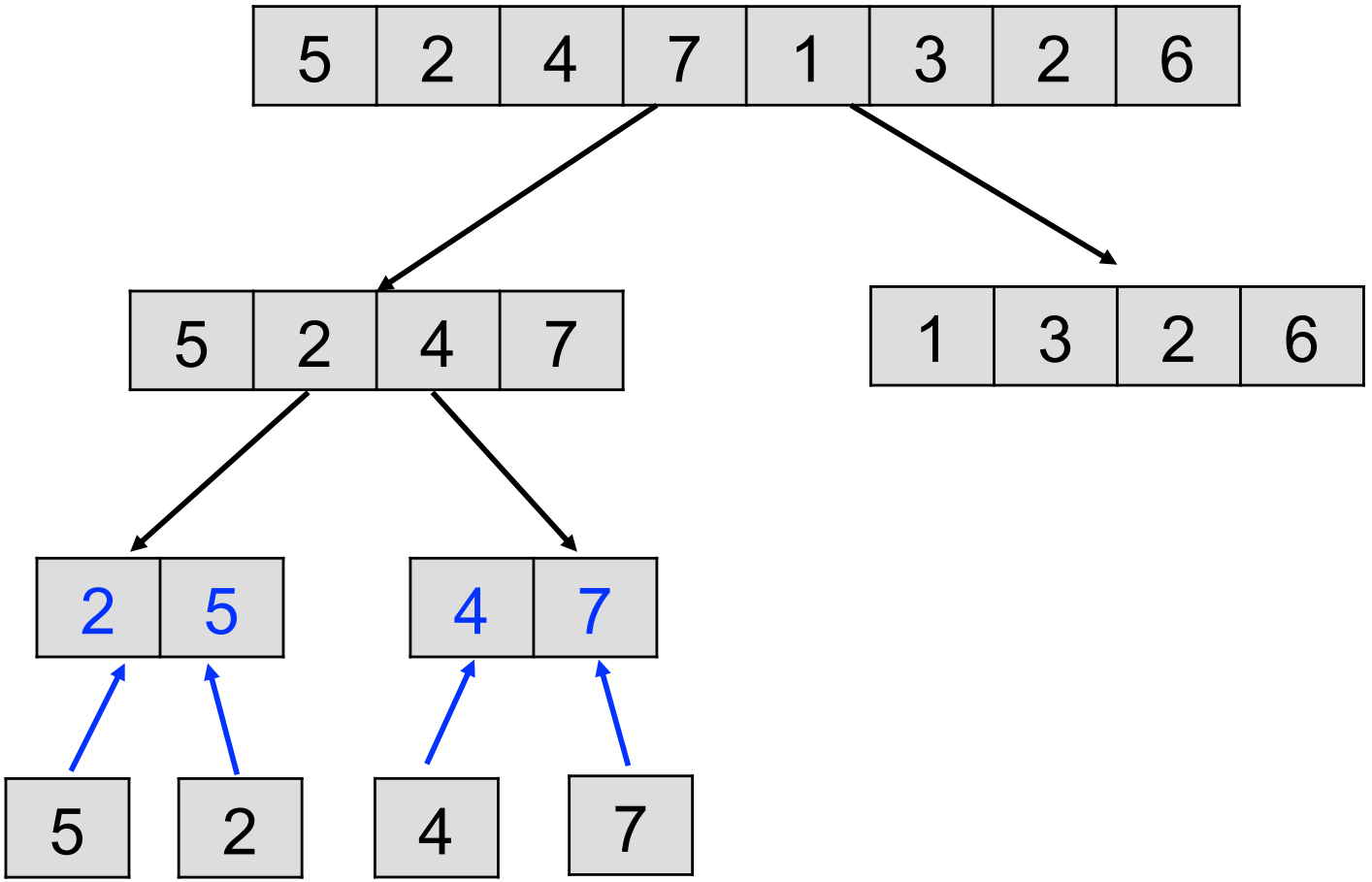
# How Does This Recursive Algo Run?



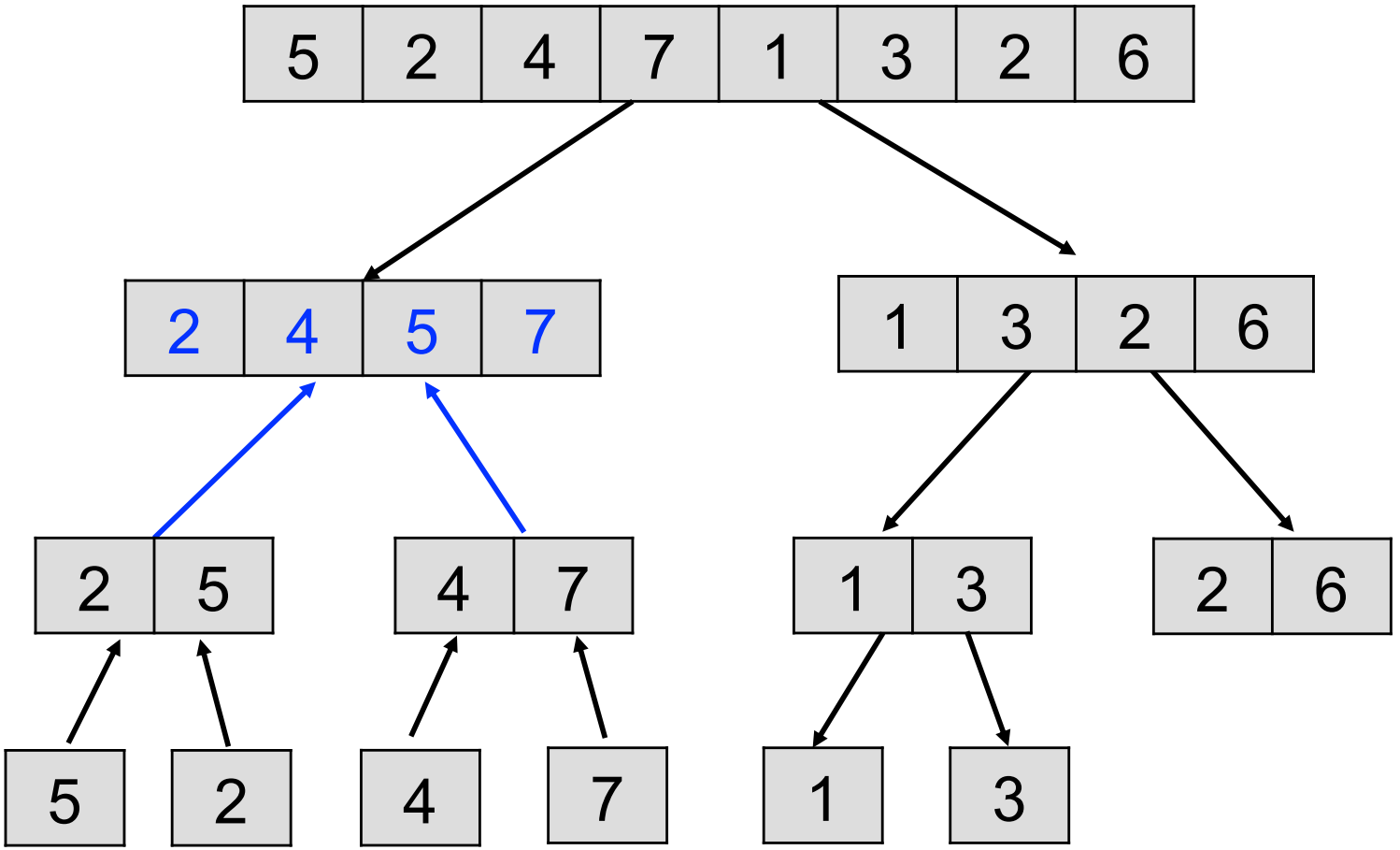
# How Does This Recursive Algo Run?



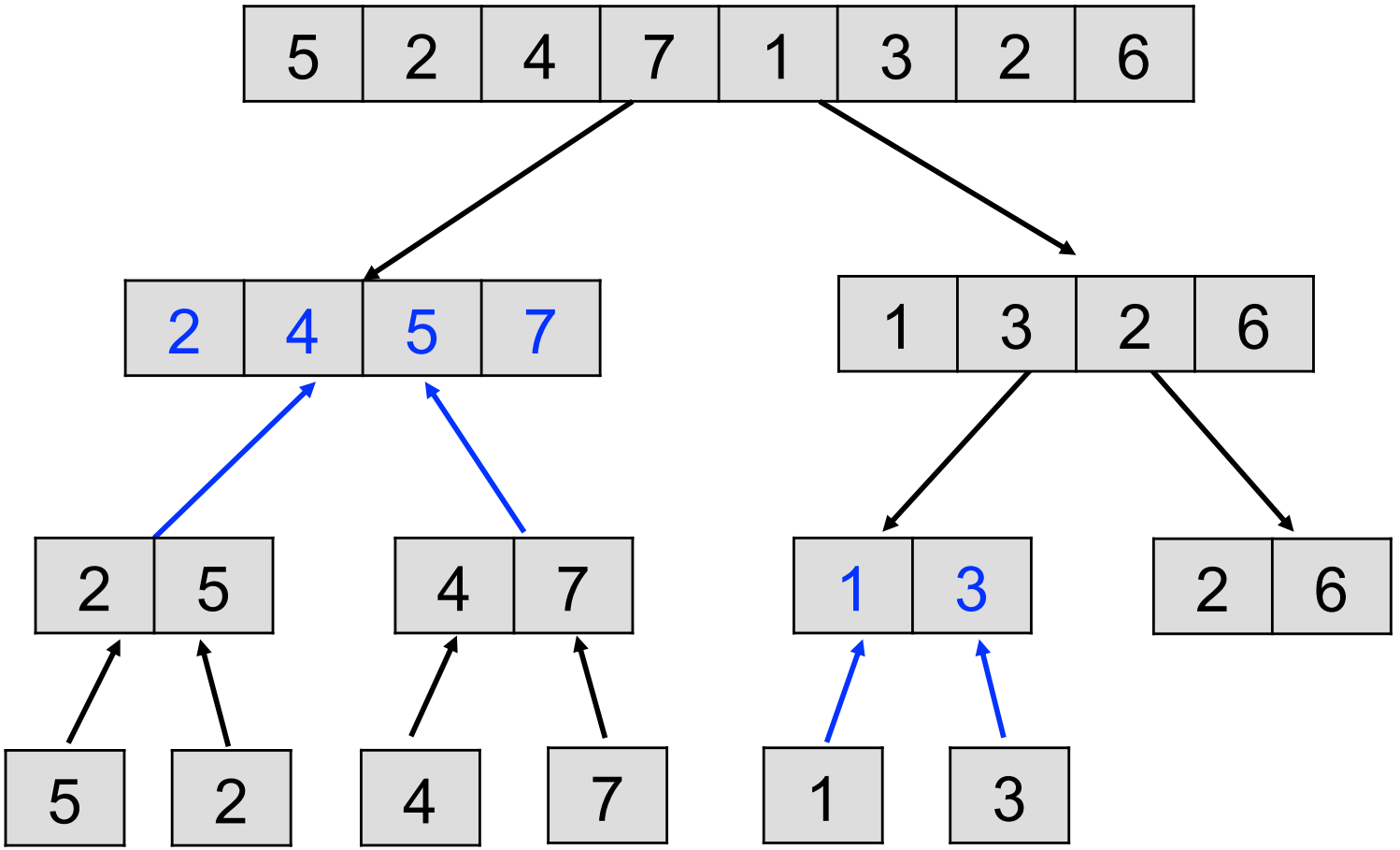
# How Does This Recurse Algo Run?



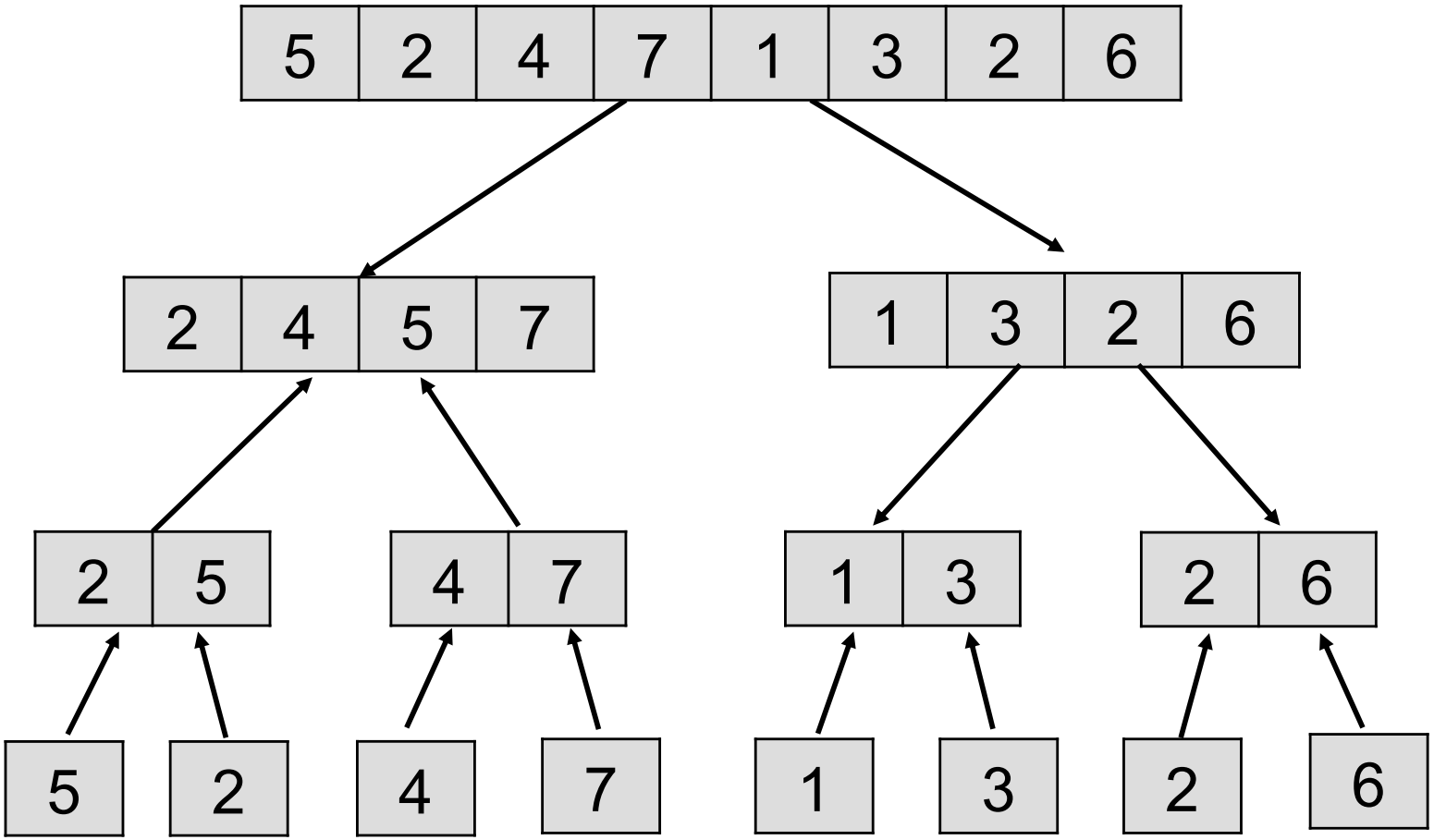
# How Does This Recurse Algo Run?



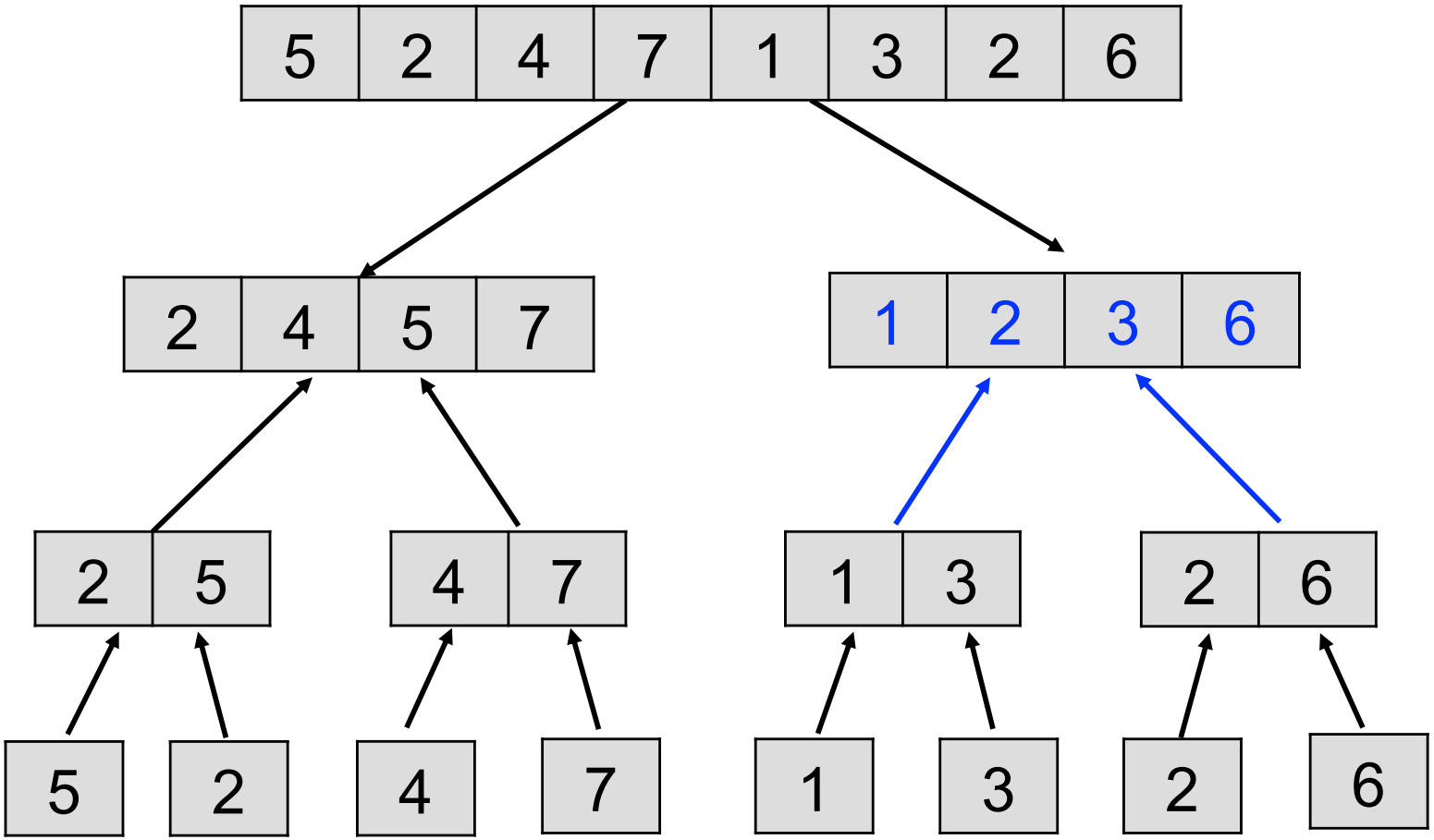
# How Does This Recurse Algo Run?



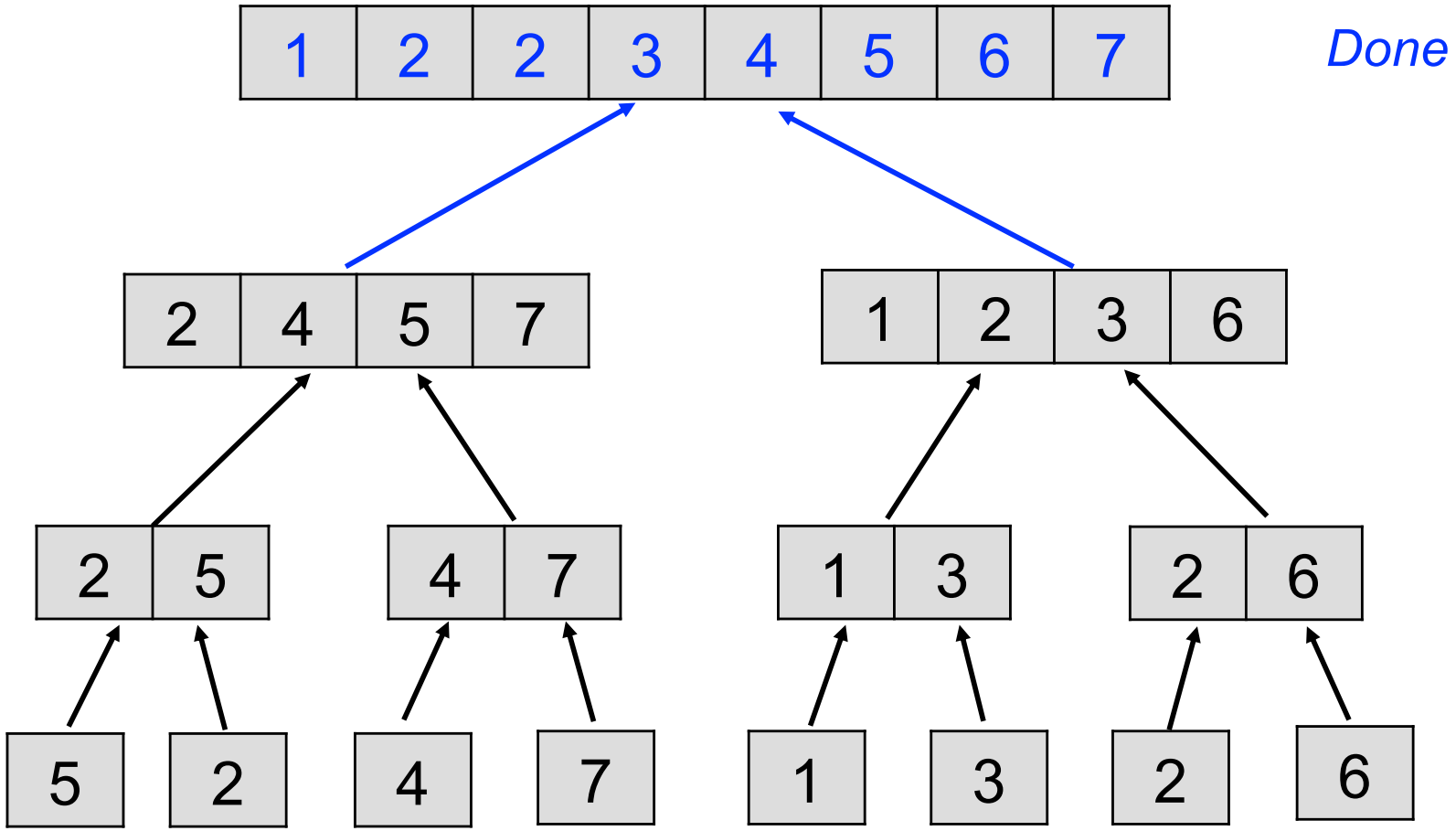
# How Does This Recurse Algo Run?



# How Does This Recurse Algo Run?



# How Does This Recurse Algo Run?



Correctness is almost obvious, next we analyze its running time.

# Running Time of Merge Sort

MERGE-SORT$(A, l, r)$ // Sort $A[l : r]$

1 **if** $l = r$ **return();**

2 **else** $q = \lfloor \frac{l+r}{2} \rfloor$

3 MERGE-SORT$(A, l, q)$ → $T(n/2)$

4 MERGE-SORT$(A, q+1, r)$ → $T(n/2)$

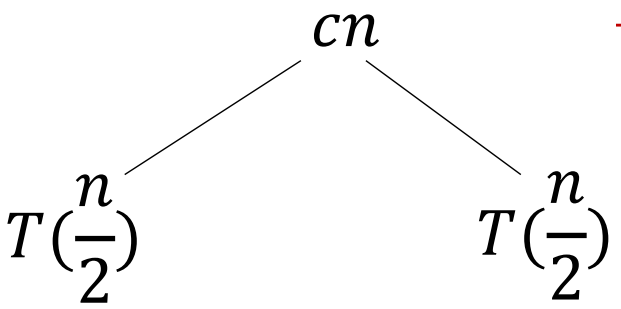
5 MERGE$(A, l, q, r)$ → $cn$

We want MERGE-SORT$(A, 1, n)$

$$T(n) = \begin{cases} c & if \ \ n = 1 \\ 2T\left(\frac{n}{2}\right) + cn & if \ \ n > 1 \end{cases}$$

# How to Calculate $T(n)$?

$T(n) =$

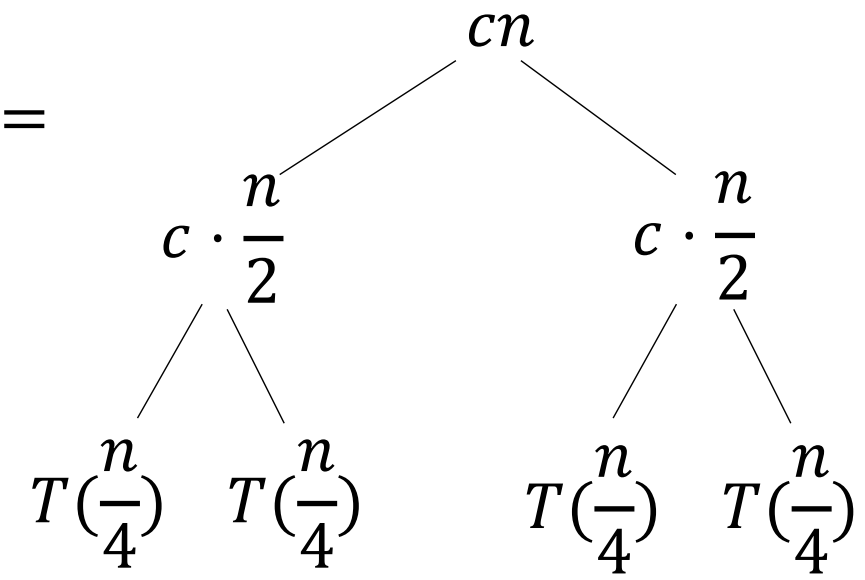



→ From MERGE

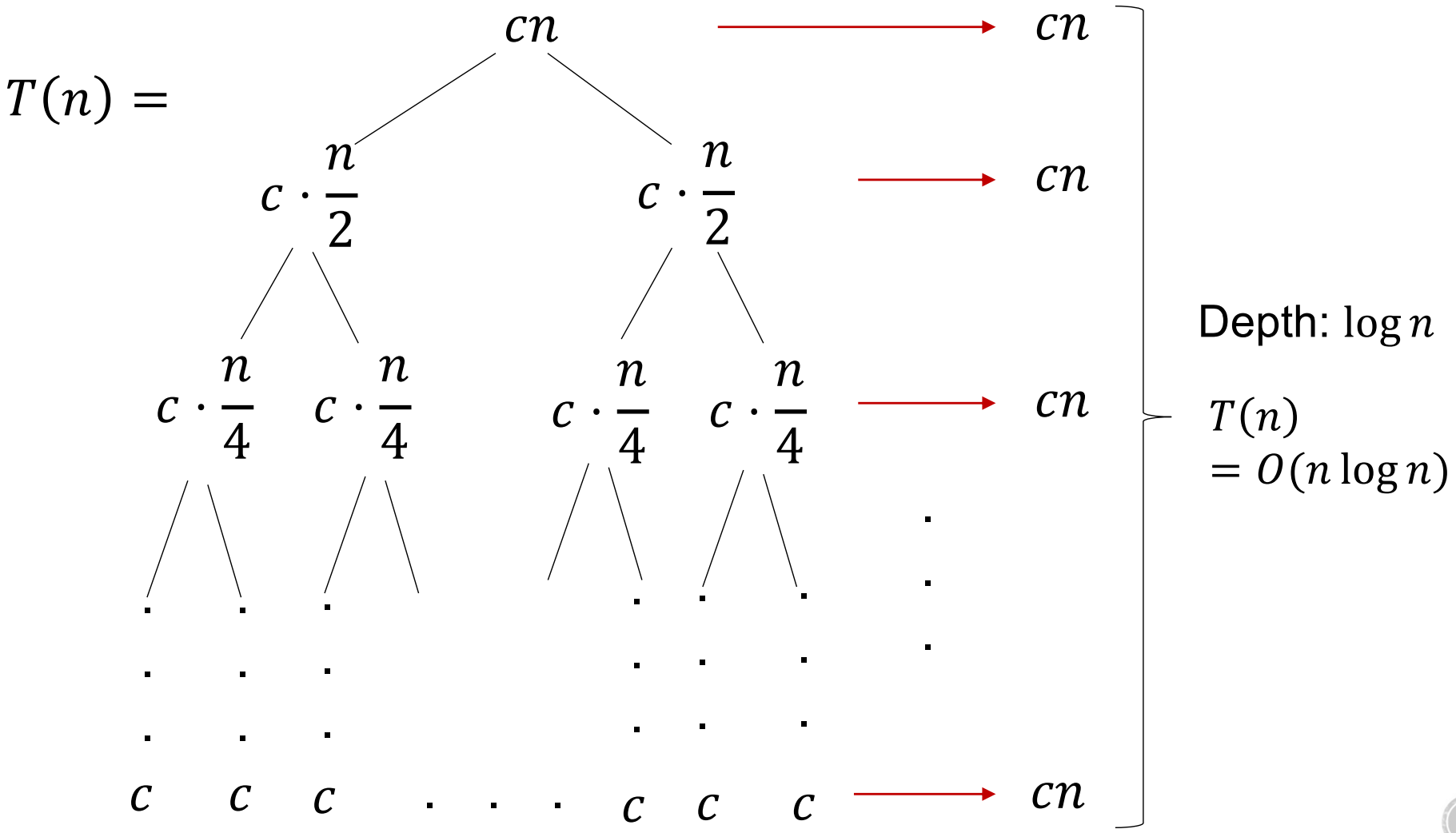
→ Solving sub-problem

# How to Calculate $T(n)$?

$T(n) =$



# How to Calculate $T(n)$?



# To Summarize…

- MERGE-SORT is correct, and takes $O(n \log n)$ time

MERGE-SORT$(A, l, r)$ // Sort $A[l : r]$

1 **if** $l = r$ **return();**

2 **else** $q = \lfloor \frac{l+r}{2} \rfloor$

3 MERGE-SORT$(A, l, q)$

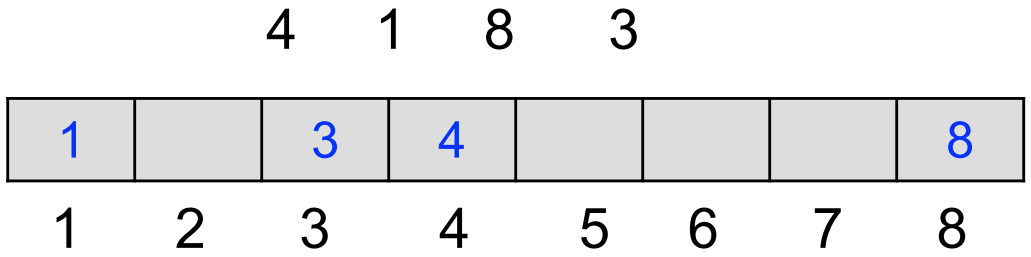
4 MERGE-SORT$(A, q + 1, r)$

5 MERGE$(A, l, q, r)$

# Is $O(n \log n)$ the Best Possible for Sorting?

- Yes, if the algorithm only makes comparisons between elements
  - I.e., the algorithm makes decisions based on comparing two elements

What is an algorithm that does not use comparisons?



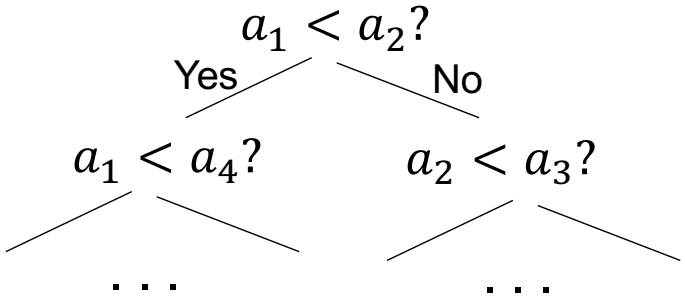

- Has $O(n)$ time if input are integers
- But problematic…
  - Large space – what if having number $10^{1000}$ or 3.1415926?
  - Not used much in practice

# Is $O(n \log n)$ the Best Possible for Sorting?

- Yes, if the algorithm only makes comparisons between elements
  - I.e., the algorithm makes decisions based on comparing two elements

Proof:

- Sorting is to find one permutation of $A$ from $n!$ candidates
- Each comparisons rules out some permutations
  - E.g., $a_1 < a_2$ rules out permutations with $a_2$ ahead $a_1$
- Comparison-based sorting is a search in a tree – we stop when we reach a leaf, which is a permutation
- There are $n!$ leaves, so worst running time = height $\geq \log(n!) = O(n \log n)$



# Is $O(n \log n)$ the Best Possible for Sorting?

- Yes, if the algorithm only makes comparisons between elements
  - I.e., the algorithm makes decisions based on comparing two elements

Proof:

- Sorting is to find one permutation of $A$ from $n!$ candidates
- Each comparisons rules out some permutations
  - E.g., $a_1 < a_2$ rules out permutations with $a_2$ ahead $a_1$
- Comparison-based sorting is a search in a tree – we stop when we reach a leaf, which is a permutation
- There are $n!$ leaves, so worst running time = height $\geq \log(n!) = O(n \log n)$

This is called lower bound analysis

- That is, prove that your problem cannot have a better algorithm
- Another example: NP-completeness proof

# Outline

- Merge Sort
- The Master Theorem
- Matrix Multiplication

In recursions, we see the following running time analysis very often

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

For example, in Merge-Sort, we have

$$T(n) = 2T\left(\frac{n}{2}\right) + cn$$

How to solve $T(n)$ generally?

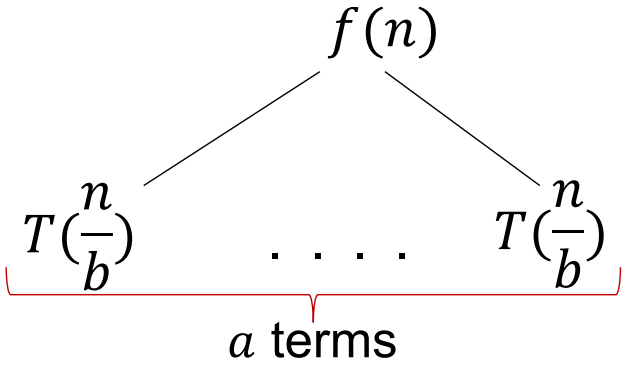
Theorem (Master Theorem). Suppose $a > 1, b > 1$ and $T(n) = aT\left(\frac{n}{b}\right) + f(n)$ for some function $f(n)$. Then $T(n)$ has following asymptotic bound:

1. If $f(n) = \mathrm{O}(\mathrm{n}^{\log_b(a)\,-\epsilon})$ for small constant $\epsilon$, then $T(n) = \Theta(\mathrm{n}^{\log_b a})$
2. If $f(n) = \Theta(\mathrm{n}^{\log_b a})$, then $T(n) = \Theta(n^{log_b a} \log n)$
3. If $f(n) = \Omega(\mathrm{n}^{\log_b(a)\,+\epsilon})$ for small constant $\epsilon$, and if $af\left(\frac{n}{b}\right) \leq cf(n)$ for some constant $c \leq 1$ for all large $n$, then $T(n) = \Theta(f(n))$.

- Recall: $\Theta$ means the same order: $f(n) = \Theta(\mathrm{n}^2) \Leftrightarrow c_1 n^2 \leq f(n) \leq c_2 n^2$
- $\log_b(a)$ is called the critical exponent
- E.g., Merge-Sort has $T(n) = 2T\left(\frac{n}{2}\right) + cn$ (case 2), so $T(n) = \Theta(n \log n)$

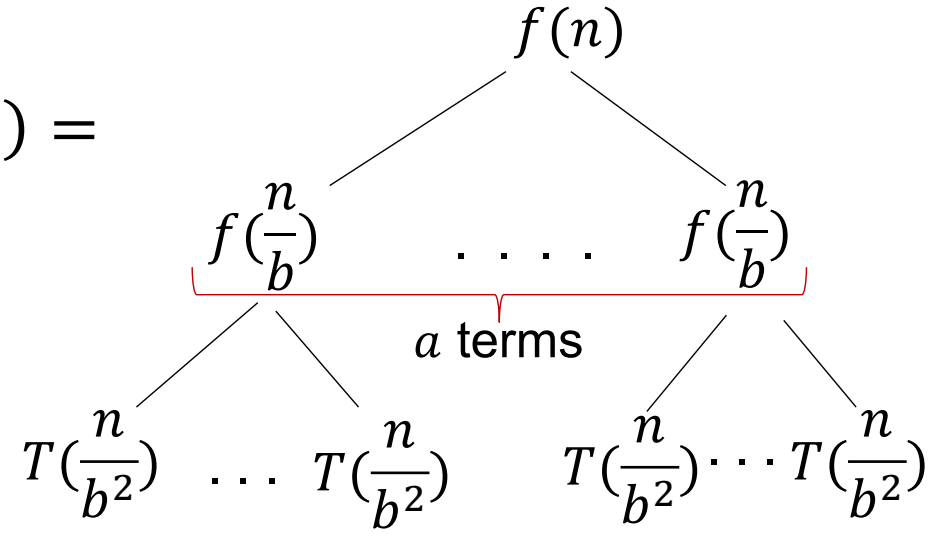
# Proof Sketch

$T(n) =$

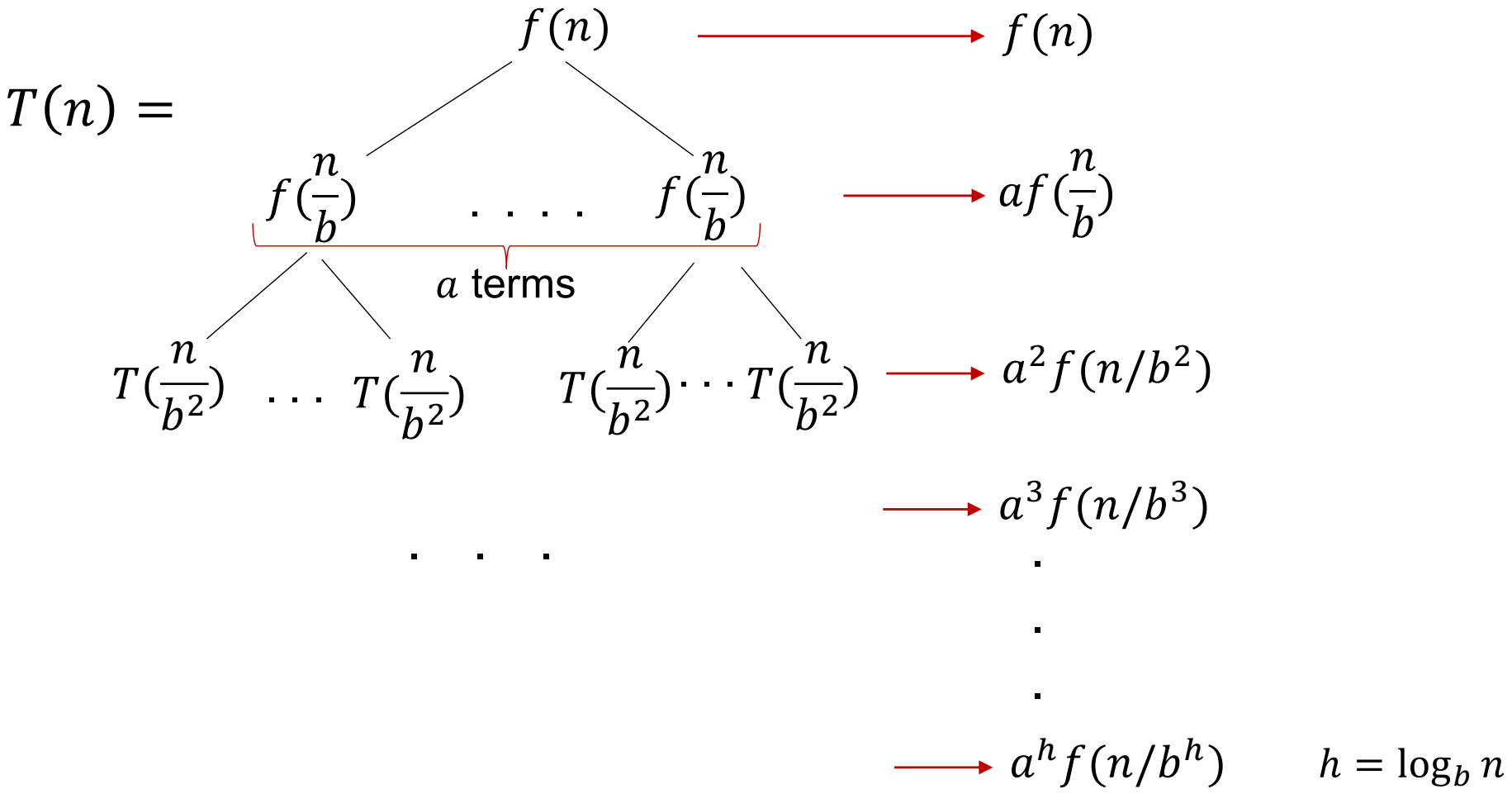


# Proof Sketch

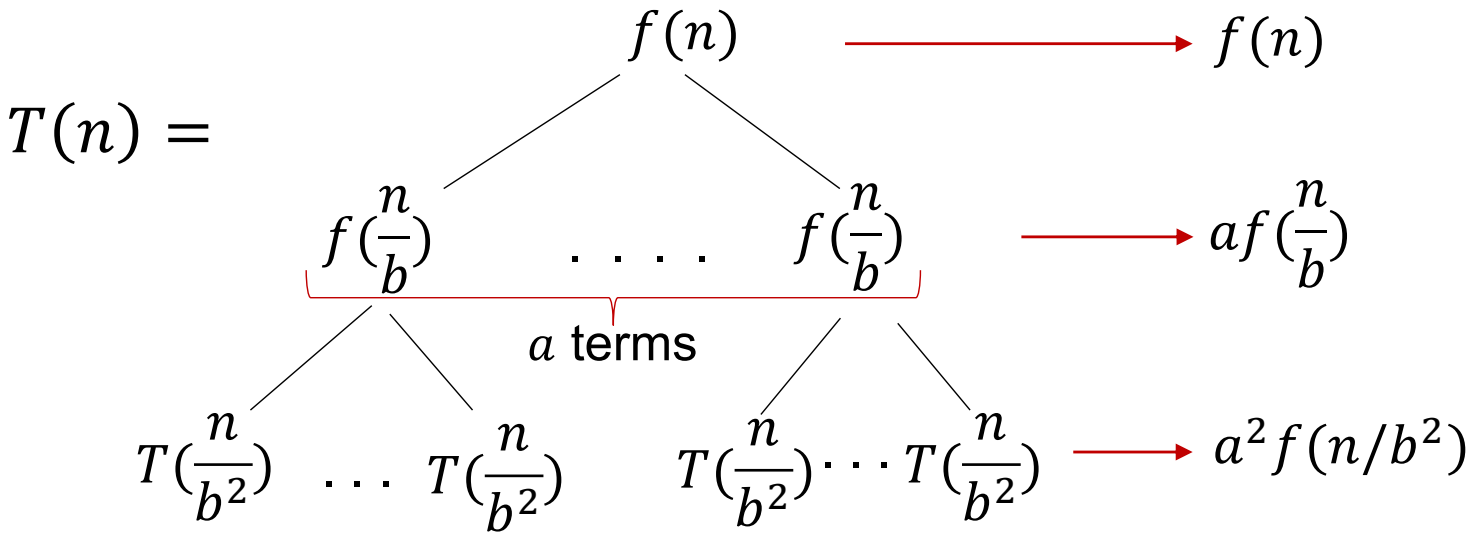
$T(n) =$



# Proof Sketch



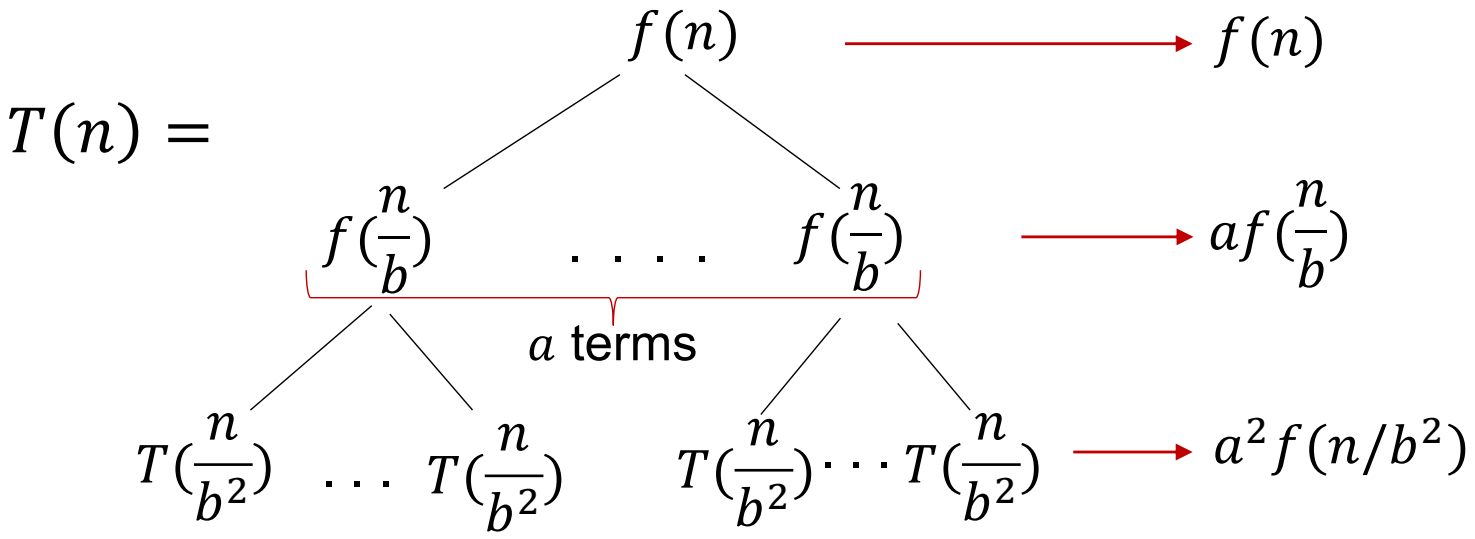
# Proof Sketch




- Suppose $f(n) = n^{\log_b a}$ (Case 2), for any $k \geq 1$ we have

$$a^k f\left(\frac{n}{b^k}\right) = a^k \left(\frac{n}{b^k}\right)^{\log_b a}$$

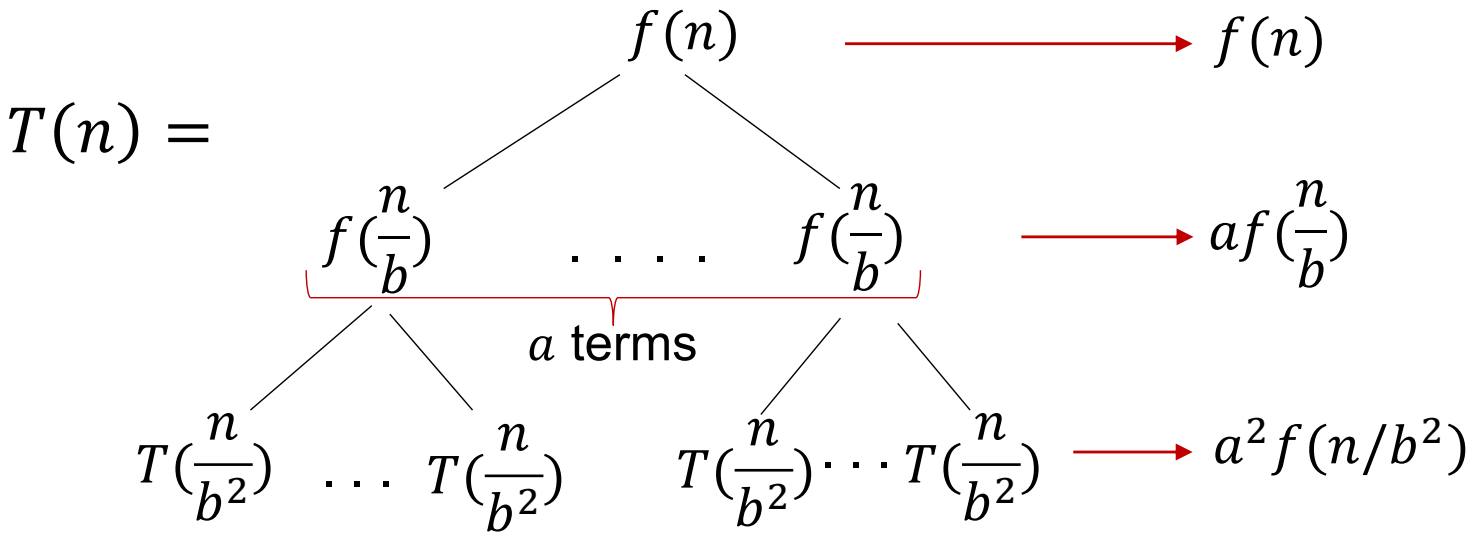
# Proof Sketch




- Suppose $f(n) = n^{log_b a}$ (Case 2), for any $k \geq 1$ we have

$$a^k f\left(\frac{n}{b^k}\right) = a^k \left(\frac{n}{b^k}\right)^{log_b a} = a^k \cdot \frac{n^{\log_b a}}{b^{k \log_b a}}$$

# Proof Sketch

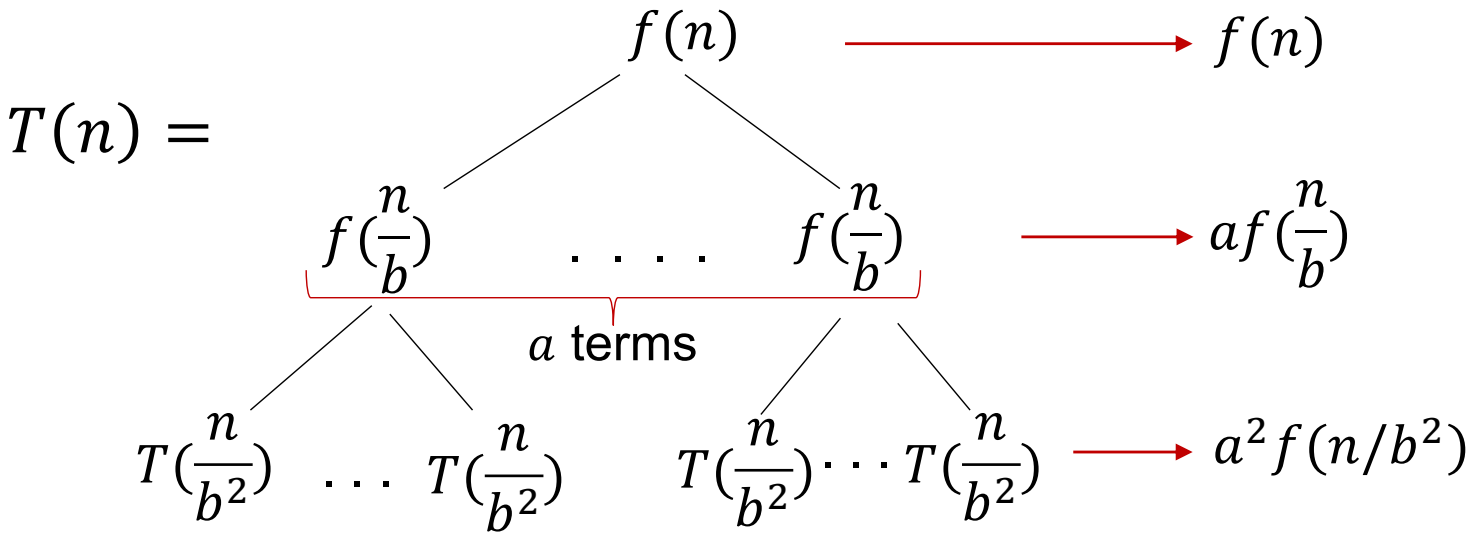



- Suppose $f(n) = n^{log_b\, a}$ (Case 2), for any $k \geq 1$ we have

$$a^k f\left(\frac{n}{b^k}\right) = a^k \left(\frac{n}{b^k}\right)^{log_b\, a} = a^k \cdot \frac{n^{\log_b a}}{b^{k \log_b a}} = a^k \cdot \frac{n^{\log_b a}}{a^k}$$

Recall $b^{\log_b a} = a$ by definition

# Proof Sketch

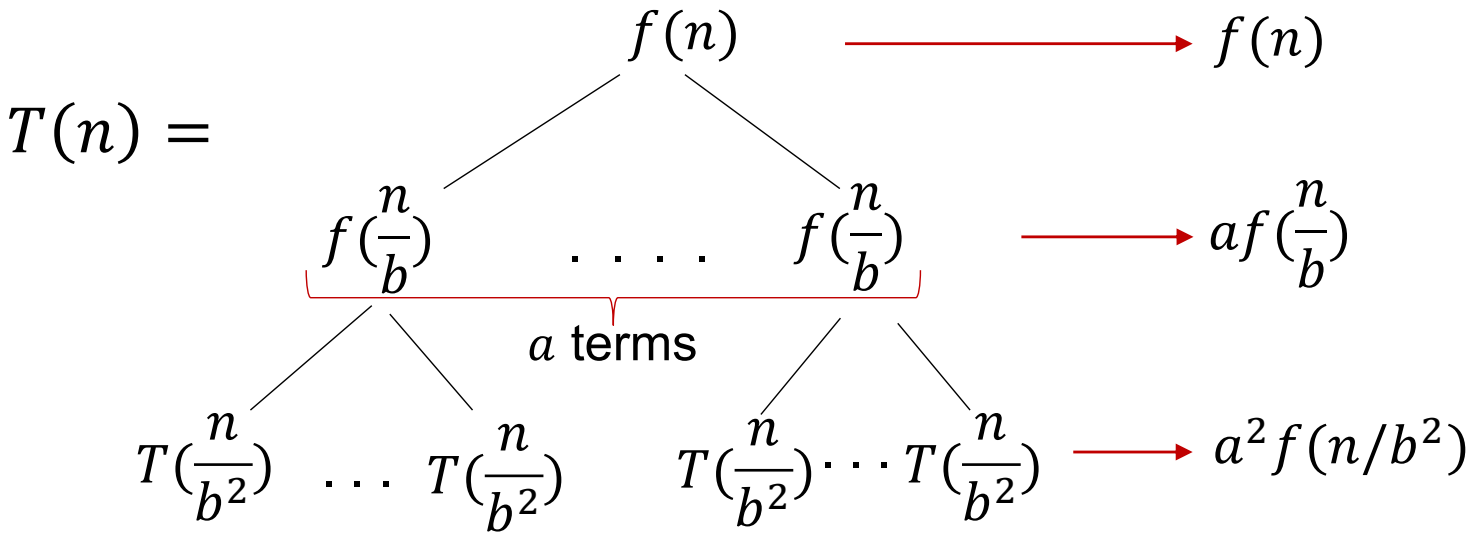



- Suppose $f(n) = n^{log_b\, a}$ (Case 2), for any $k \geq 1$ we have

$$a^k f\left(\frac{n}{b^k}\right) = a^k \left(\frac{n}{b^k}\right)^{log_b\, a} = a^k \cdot \frac{n^{\log_b a}}{b^{k \log_b a}} = a^k \cdot \frac{n^{\log_b a}}{a^k} = n^{\log_b a}$$

- Therefor, when $f(n) = \Theta(n^{log_b\, a})$ (Case 2), $\mathrm{T(n)} = \mathrm{O}(n^{log_b\, a} \log n)$

# Proof Sketch




- Suppose $f(n) = n^{log_b\, a}$ (Case 2), for any $k \geq 1$ we have

$$a^k f\left(\frac{n}{b^k}\right) = a^k \left(\frac{n}{b^k}\right)^{log_b\, a} = a^k \cdot \frac{n^{\log_b a}}{b^{k \log_b a}} = a^k \cdot \frac{n^{\log_b a}}{a^k} = n^{\log_b a}$$

Crucial note: when $f(n) = \Theta(n^{log_b\, a})$, cost at each iteration is always $\approx f(n)$.

# Examples 1

$$T(n) = \begin{cases} c & if\ \ n = 1 \\ 4T\left(\frac{n}{2}\right) + n^3 & if\ \ n > 1 \end{cases}$$

- $a = 4,\ b = 2$
- $n^{\log_b a} \approx n^2$, $\mathrm{f(n)} = \mathrm{n}^3$, so $\mathrm{f(n)} = \Theta(n^{\log_b a + \epsilon})$ → Case 3
- Since $4\left(\frac{n}{2}\right)^3 \leq cn^3$ for $c = 1/2$, Master theorem implies $T(n) = \Theta(n^3)$

# Examples 2

$$T(n) = \begin{cases} c & if\ \ n = 1 \\ 7T\left(\frac{n}{2}\right) + cn^2 & if\ \ n > 1 \end{cases}$$

- $a = 7,\ b = 2$
- $n^{\log_b a} \approx n^{2.8}$, $\mathrm{f(n)} = \mathrm{cn}^2$, so $\mathrm{f(n)} = \Theta(n^{\log_b a - \epsilon})$ → Case 1
- Master theorem implies $T(n) = \Theta\left(n^{\log_b a}\right) \approx \Theta(n^{2.8})$

# Outline

- Merge Sort
- The Master Theorem
- Matrix Multiplication

# What is Matrix Multiplication?

Well….

$$\underset{k\times n}{\begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{k1} & \cdots & a_{kn} \end{pmatrix}} \times \underset{n\times m}{\begin{pmatrix} b_{11} & \cdots & b_{1m} \\ \vdots & \ddots & \vdots \\ b_{n1} & \cdots & a_{nm} \end{pmatrix}} = \underset{k\times m}{\begin{pmatrix} c_{11} & \cdots & c_{1m} \\ \vdots & \ddots & \vdots \\ c_{k1} & \cdots & c_{km} \end{pmatrix}}$$

$$A \quad \times \quad B \quad = \quad C$$

Straightforward algorithm:

1 **for** $i = 1, \cdots k$

2 **for** $j = 1, \cdots m$

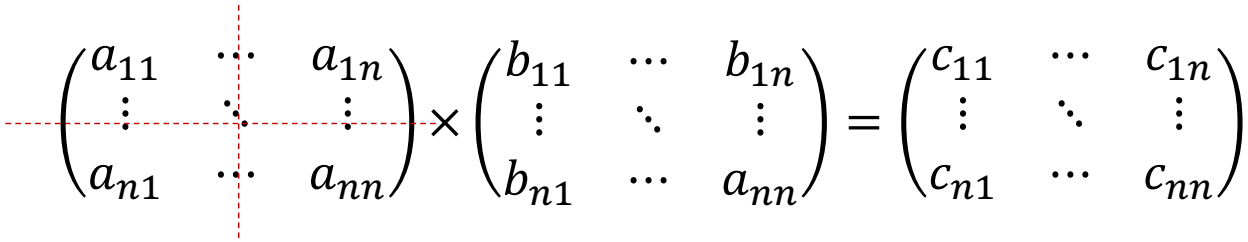
3 $C[i,j] = \sum_{l=1}^{n} a_{il}\, b_{lj}$

Running time? $knm$

Q: can we do better?

# Let's Try Divide and Conquer!

For convenience, consider $n \times n$ square matrices here:




Q: how to divide and how to "conquer"?

# Let's Try Divide and Conquer!

For convenience, consider $n \times n$ square matrices here:

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \times \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

$$C_{11} = A_{11}B_{11} + A_{12}B_{21}$$

$$C_{12} = A_{11}B_{12} + A_{12}B_{22}$$

$$C_{21} = A_{21}B_{11} + A_{22}B_{21}$$

$$C_{22} = A_{21}B_{12} + A_{22}B_{22}$$

→ 8 multiplications
4 additions

# Pseudo-Code

```
SQUARE-MATRIX-MULTIPLY-RECURSIVE(A, B)
 1  n = A.rows
 2  let C be a new n × n matrix
 3  if n == 1
 4      c11 = a11 · b11
 5  else partition A, B, and C as in equations (4.9)
 6      C11 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B11)
              + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B21)
 7      C12 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B12)
              + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B22)
 8      C21 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B11)
              + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B21)
 9      C22 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B12)
              + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B22)
10  return C
```

# Running Time via Master Theorem

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

8 multiplications

4 additions

- $a = 8,\ b = 2$
- $n^{\log_b a} = n^3$, $\mathrm{f(n)} = \Theta(n^2)$, so $\mathrm{f(n)} = \Theta(n^{\log_b a - \epsilon})$ → Case 1
- Master theorem implies $T(n) = \Theta(n^3)$

Same as direct multiplication!

# A Clever Idea That Uses 7 Multiplications!

This idea, by Volker Strassen, is somewhat like a magic!

$$
\begin{aligned}
S_1 &= B_{12} - B_{22} \\
S_2 &= A_{11} + A_{12} \\
S_3 &= A_{21} + A_{22} \\
S_4 &= B_{21} - B_{11} \\
S_5 &= A_{11} + A_{22} \\
S_6 &= B_{11} + B_{22} \\
S_7 &= A_{12} - A_{22} \\
S_8 &= B_{21} + B_{22} \\
S_9 &= A_{11} - A_{21} \\
S_{10} &= B_{11} + B_{12}
\end{aligned}
$$

# A Clever Idea That Uses 7 Multiplications!

This idea, by Volker Strassen, is somewhat like a magic!

$$
\begin{aligned}
S_1 &= B_{12} - B_{22} \\
S_2 &= A_{11} + A_{12} \\
S_3 &= A_{21} + A_{22} \\
S_4 &= B_{21} - B_{11} \\
S_5 &= A_{11} + A_{22}
\end{aligned}
$$

- Key insight: use (constantly) many more additions to trade for the saving of one multiplication
  - Since addition takes $O(n^2)$, much faster than multiplication
- How did he come up with such magic?
  - I don't know…a lot of trial and failure, until succeeded I guess
  - Algorithm design requires creativity!

$$
\begin{aligned}
P_4 &= A_{22} \cdot S_4 = A_{22} \cdot B_{21} - A_{22} \cdot B_{11}, \\
P_5 &= S_5 \cdot S_6 = A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22} \\
P_6 &= S_7 \cdot S_8 = A_{12} \cdot B_{21} + A_{12} \cdot B_{22} - A_{22} \cdot B_{21} - A_{22} \cdot B_{22} \\
P_7 &= S_9 \cdot S_{10} = A_{11} \cdot B_{11} + A_{11} \cdot B_{12} - A_{21} \cdot B_{11} - A_{21} \cdot B_{12}
\end{aligned}
$$

# Are We Indeed Faster Now?

$$T(n) = \begin{cases} c & if \;\; n = 1 \\ 7T\left(\frac{n}{2}\right) + cn^2 & if \;\; n > 1 \end{cases}$$

7 multiplications

About 18 additions

# Are We Indeed Faster Now?

$$T(n) = \begin{cases} c & if \ \ n = 1 \\ 7T\left(\frac{n}{2}\right) + cn^2 & if \ \ n > 1 \end{cases}$$

- This is precisely the Example 2 after introducing Master theorem
- Have shown $T(n) = \Theta\left(n^{\log_2 7}\right) \approx \Theta(n^{2.8})$ → Yes, indeed faster!

Is this asymptotically the best possible?

- No, there are $O(n^{2.38})$ time algorithm
  - Super complicated, and the omitted constant is extremely large
- Strassen's algorithm is the most widely used one in practice

# Thank You

Haifeng Xu

University of Virginia

hx4ad@virginia.edu