BITS Pilani
Hyderabad Campus

Data Structures and Algorithms Design

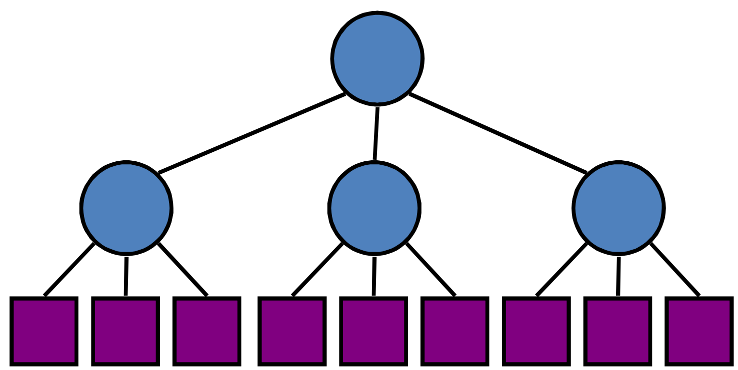
Febin.A.Vahab
2019-20

ONLINE SESSION 12 -PLAN

Sessions(#)	List of Topic Title	Text/Ref Book/external resource
12	**Divide and Conquer -** Design Principles and Strategy, Analysing Divide and Conquer Algorithms, Integer Multiplication Problem **Sorting Problem** - Merge Sort Algorithm	**T1: 5.2, 4.1, 4.3**

Divide-and-Conquer

- **Divide-and conquer** is a general algorithm design paradigm:
  - Divide: divide the input data S in two or more disjoint subsets $S_1, S_2, \ldots$
  - Recur: solve the sub problems recursively
  - Conquer: combine the solutions for $S_1, S_2, \ldots,$ into a solution for S
- The base case for the recursion are sub problems of constant size
- Analysis can be done using **recurrence equations**



Merge Sort

- Merge-sort is a sorting algorithm based on the divide-and-conquer paradigm
- Like heap-sort
  - It uses a comparator
  - It has $\boldsymbol{O}(\boldsymbol{n} \log \boldsymbol{n})$ running time
- Unlike heap-sort
  - It does not use an auxiliary priority queue
  - It accesses data in a sequential manner (suitable to sort data on a disk)

Merge Sort

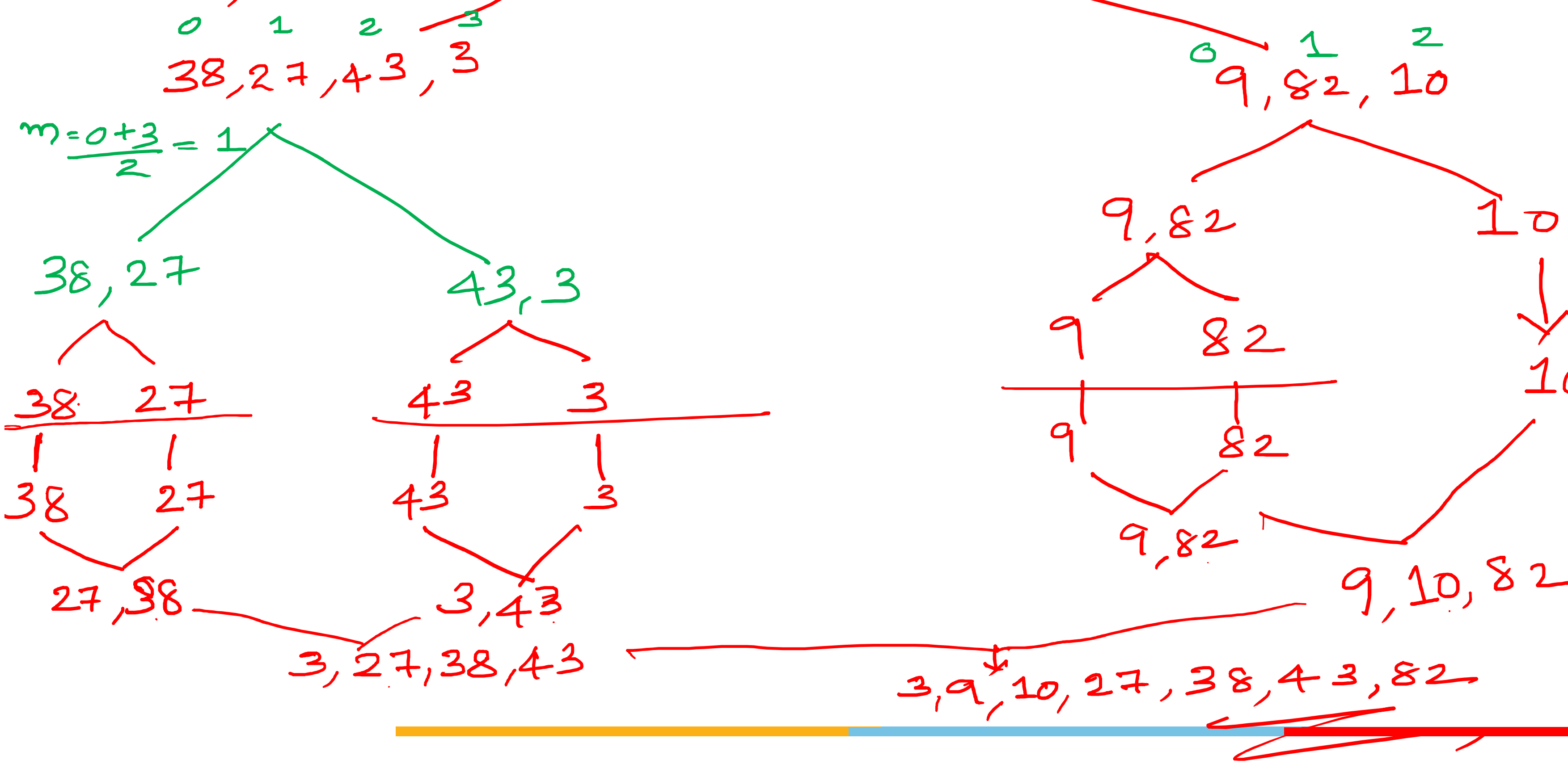
- Merge-sort on an input sequence S with n elements consists of three steps:
  - Divide: partition S into two sequences S_1 and S_2 of about $n/2$ elements each
  - Recur: recursively sort S_1 and S_2
  - Conquer: merge S_1 and S_2 into a unique sorted sequence

Merge Sort

l, r

0	1	2	3	4	5	6
38	27	43	3	9	82	10

$m = \left(\frac{l+r}{2}\right) = \frac{0+6}{2} = 3$



Merge Sort

MergeSort(arr[], l, r)

If r > l

1. ***Find the middle point to divide the array into two halves:***

   middle m = (l+r)/2

2. ***Call MergeSort for first half:***

   Call mergeSort(arr, l, m)

3. ***Call MergeSort for second half:***

   Call mergeSort(arr, m+1, r)

4. ***Merge the two halves sorted in step 2 and 3***

   Call merge(arr, l, m, r)

Merge Sort

- The conquer step of merge-sort consists of merging two sorted sequences ***A*** and ***B*** into a sorted sequence ***S*** containing the union of the elements of ***A*** and ***B***
- Merging two sorted sequences, each with ***n***/2 elements and implemented by means of a doubly linked list, takes ***O***(***n***) time

Merge Sort

Merging two sorted lists, merge(A,B)

X:

~~3~~	~~10~~	~~23~~	54

Y:

~~1~~	~~5~~	~~25~~	75

Result:

1	3	5	10	23	25	54	75

Merge Sort

Algorithm *merge*(*A*, *B*)

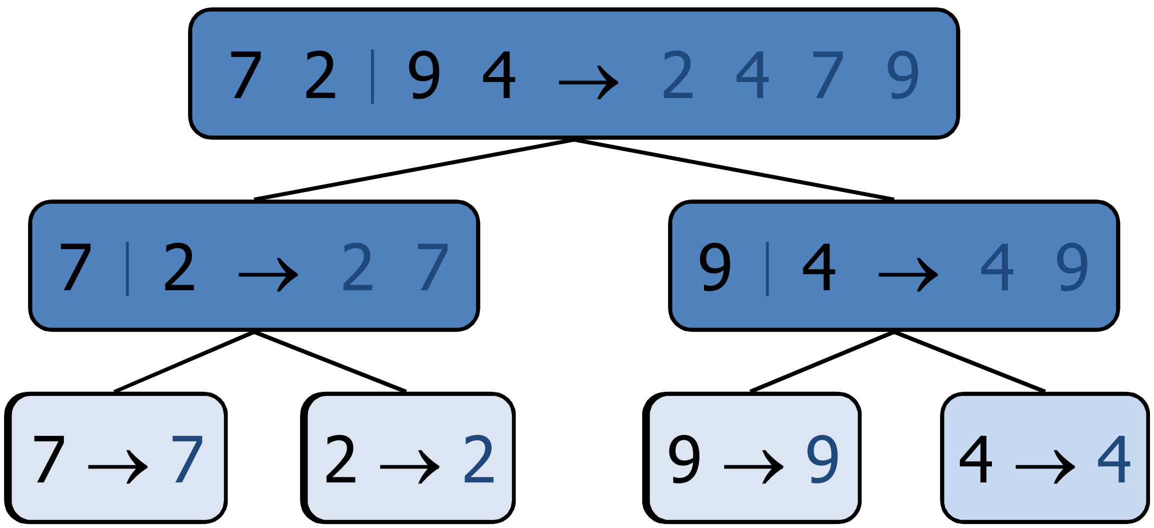
Input sequences *A* and *B* with $n/2$ elements each

Output sorted sequence of $A \cup B$

```
S ← empty sequence
while ¬A.isEmpty() ∧ ¬B.isEmpty()
    if A.first().element() < B.first().element()
        S.insertLast(A.remove(A.first()))
    else
        S.insertLast(B.remove(B.first()))
while ¬A.isEmpty()          S.insertLast(A.remove(A.first()))
while ¬B.isEmpty()          S.insertLast(B.remove(B.first()))
return S
```

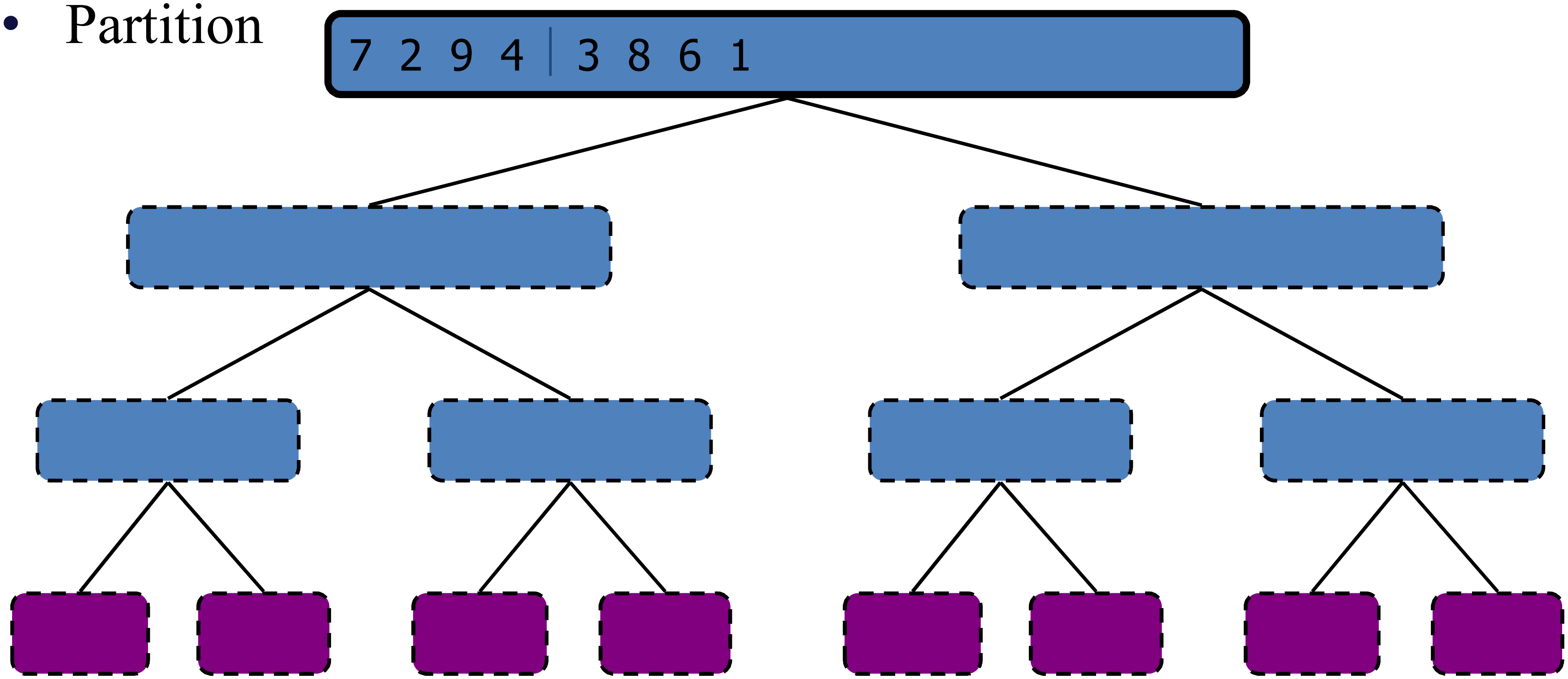
Merge Sort

- An execution of merge-sort is depicted by a binary tree
  - each node represents a recursive call of merge-sort and stores
    - unsorted sequence before the execution and its partition
    - sorted sequence at the end of the execution
  - the root is the initial call
  - the leaves are calls on subsequences of size 0 or 1



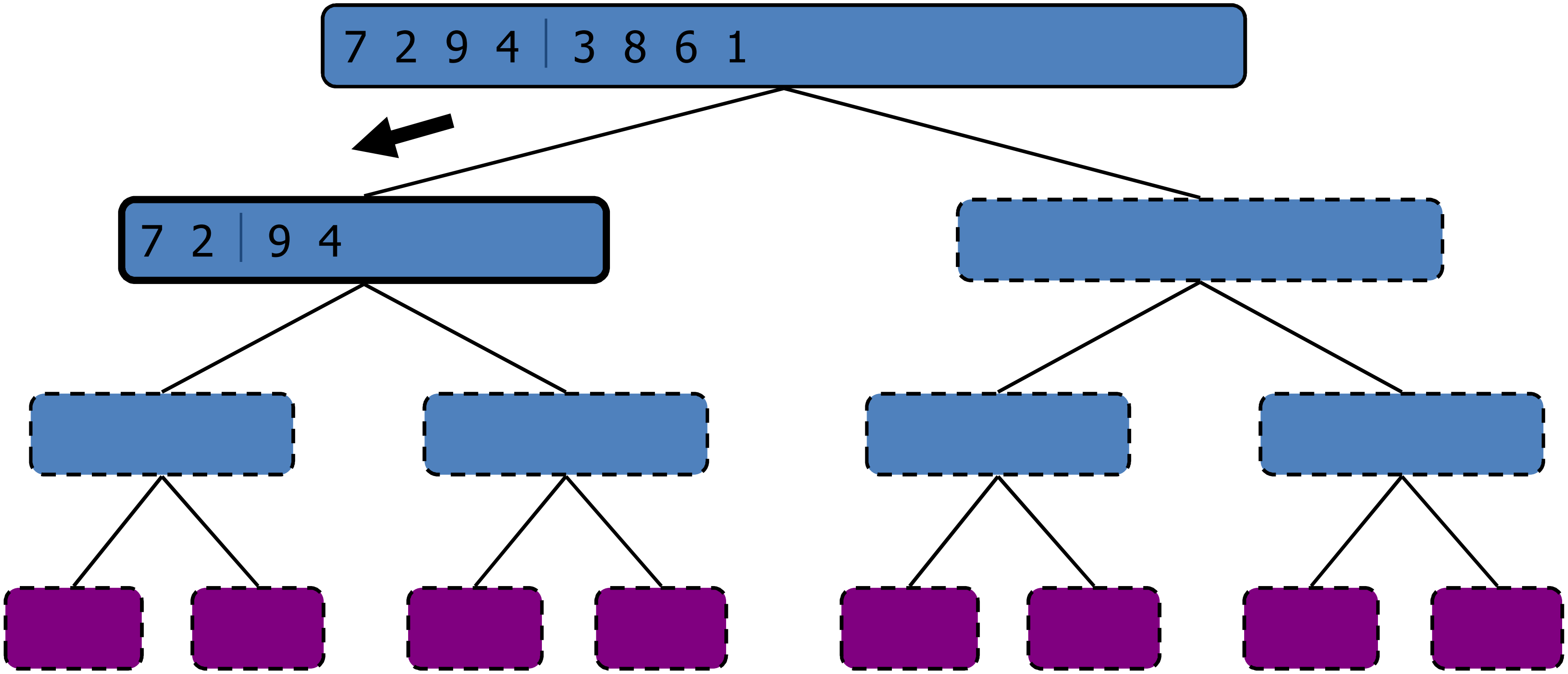
Execution Example

- Partition



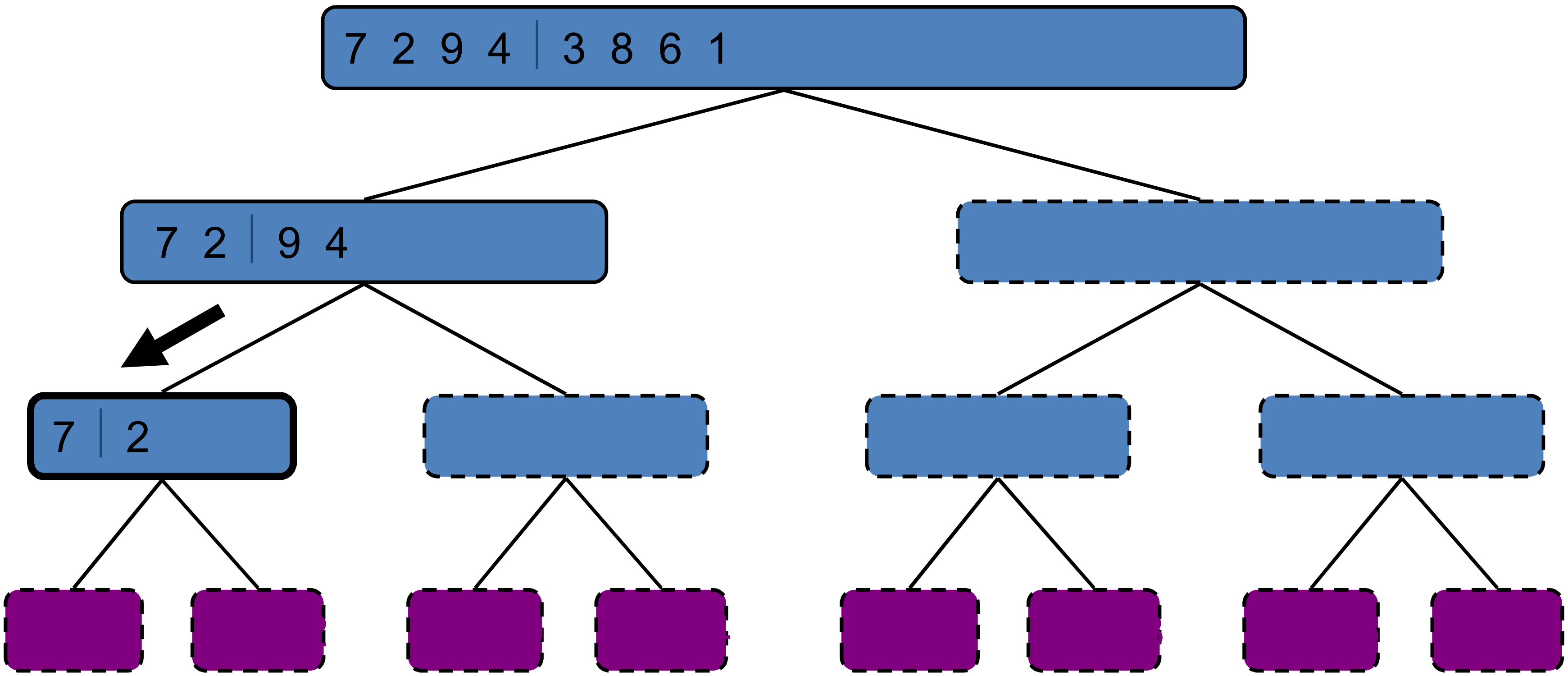
Execution Example (cont.)

- Recursive call, partition



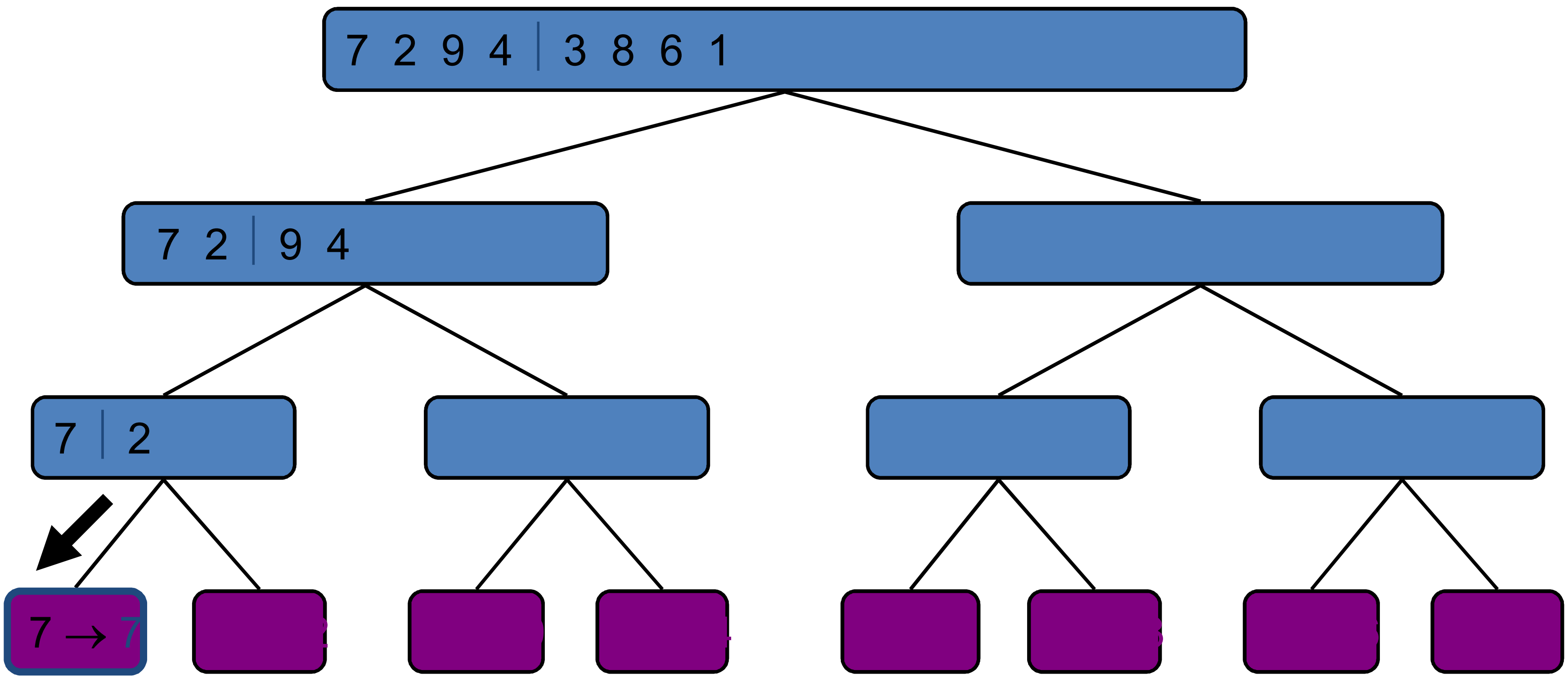
Execution Example (cont.)

- Recursive call, partition



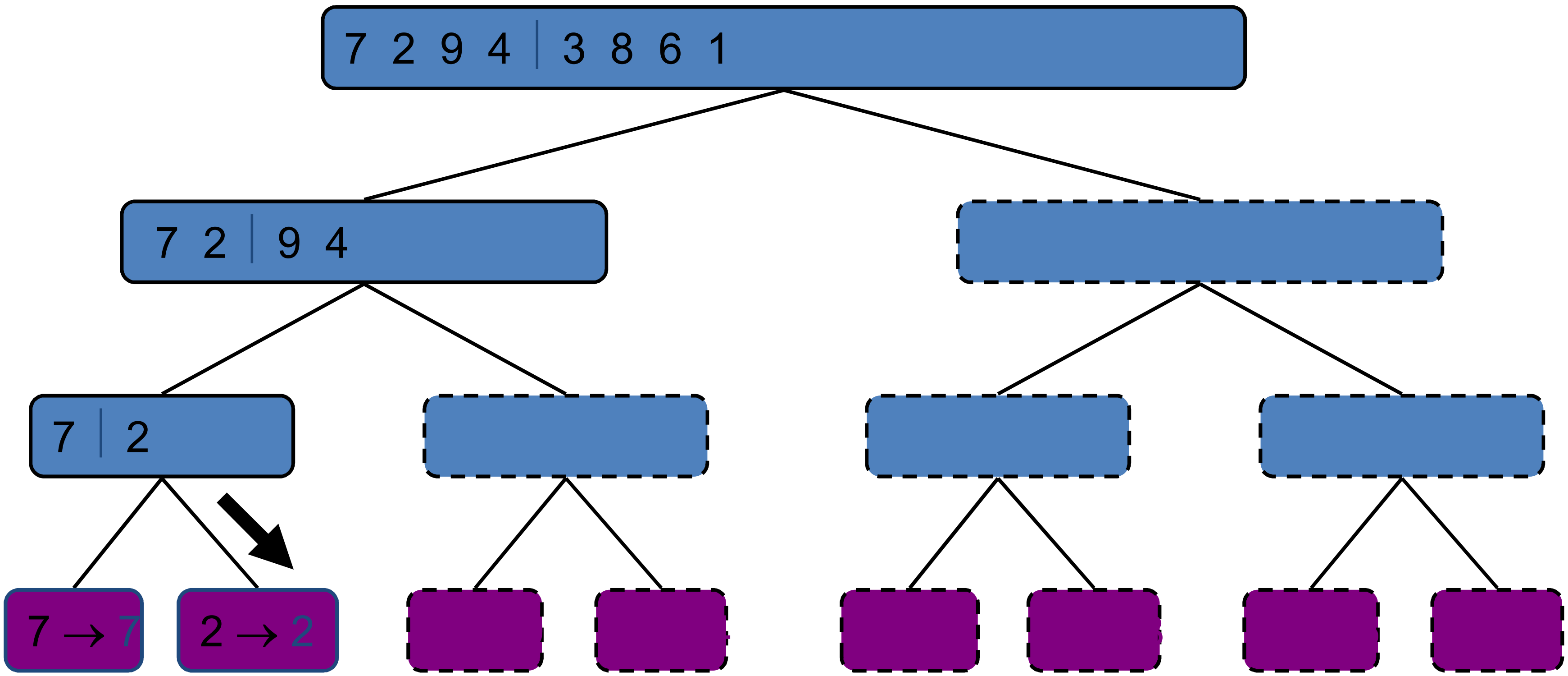
Execution Example (cont.)

- Recursive call, base case



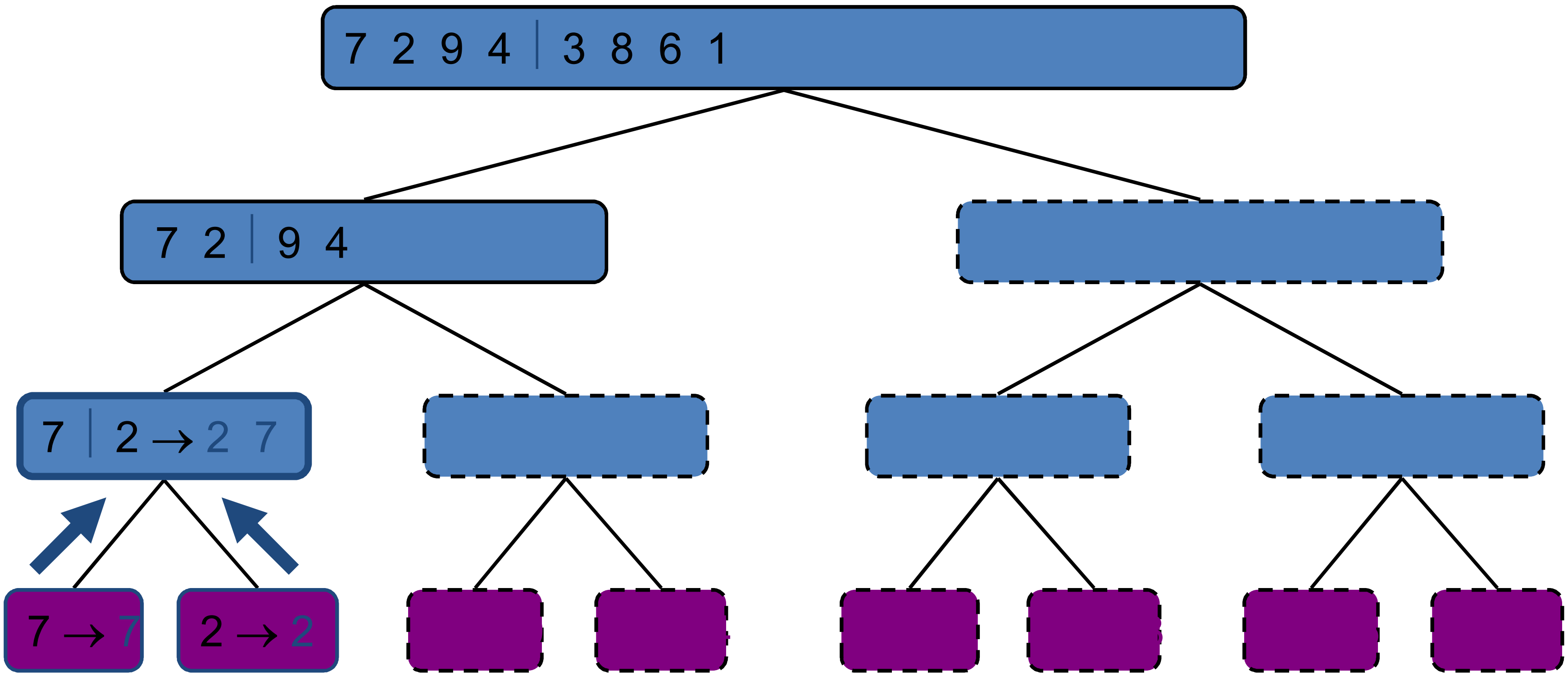
Execution Example (cont.)

- Recursive call, base case



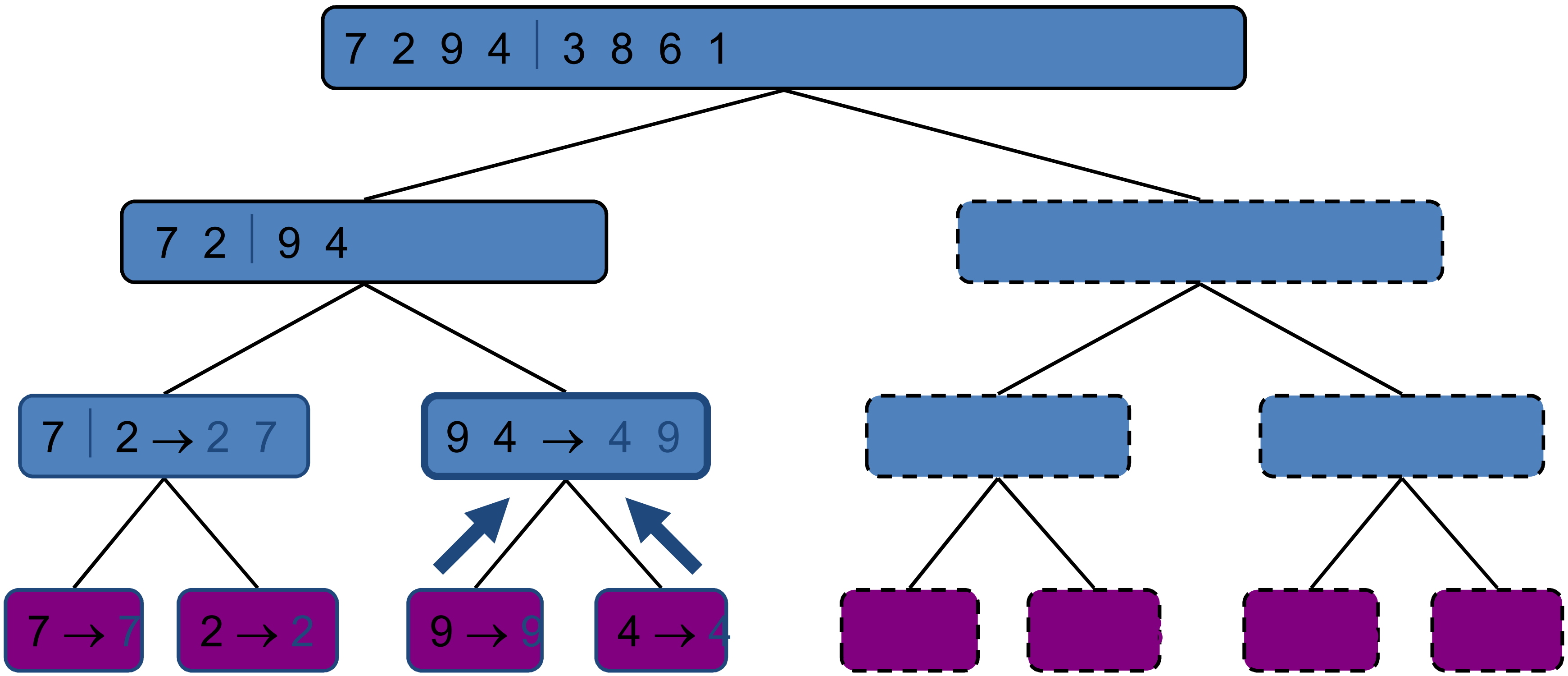
Execution Example (cont.)

- Merge



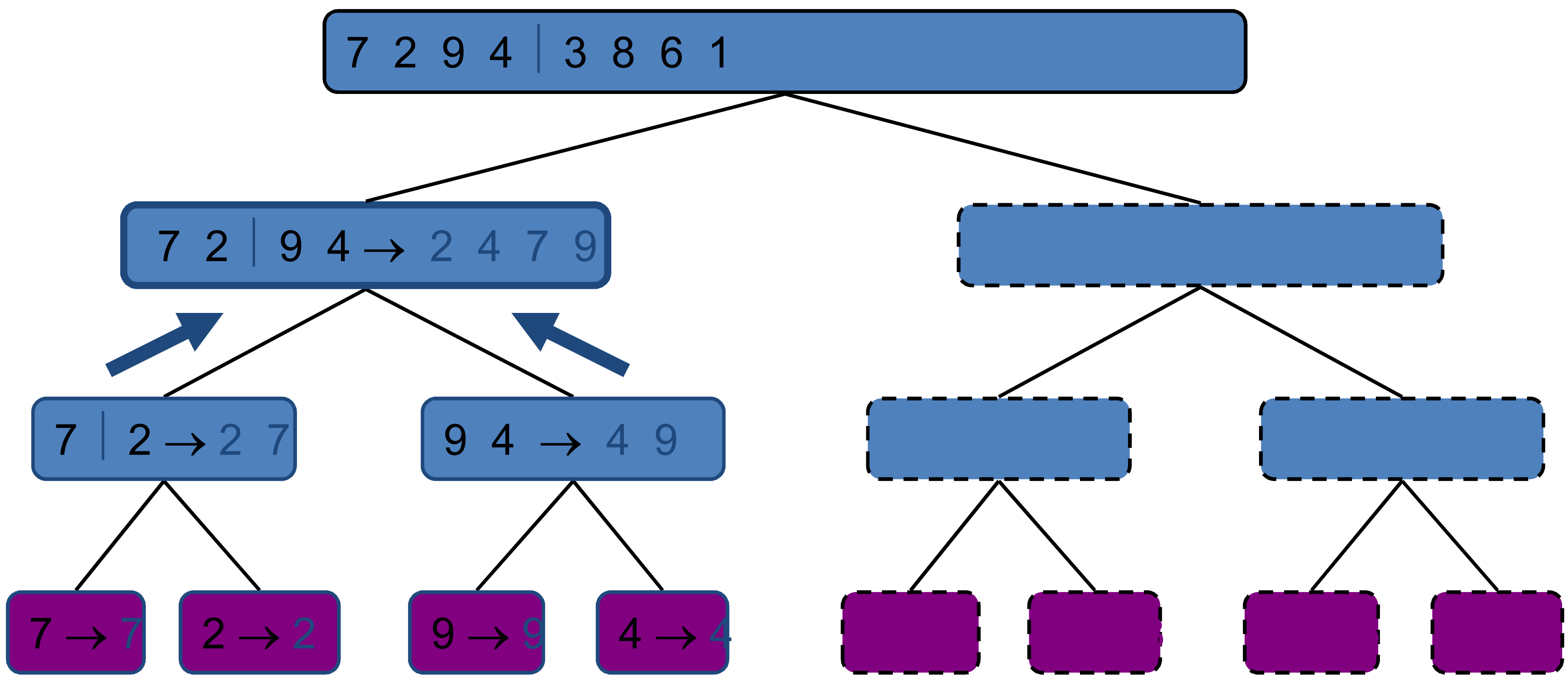
Execution Example (cont.)

- Recursive call, …, base case, merge



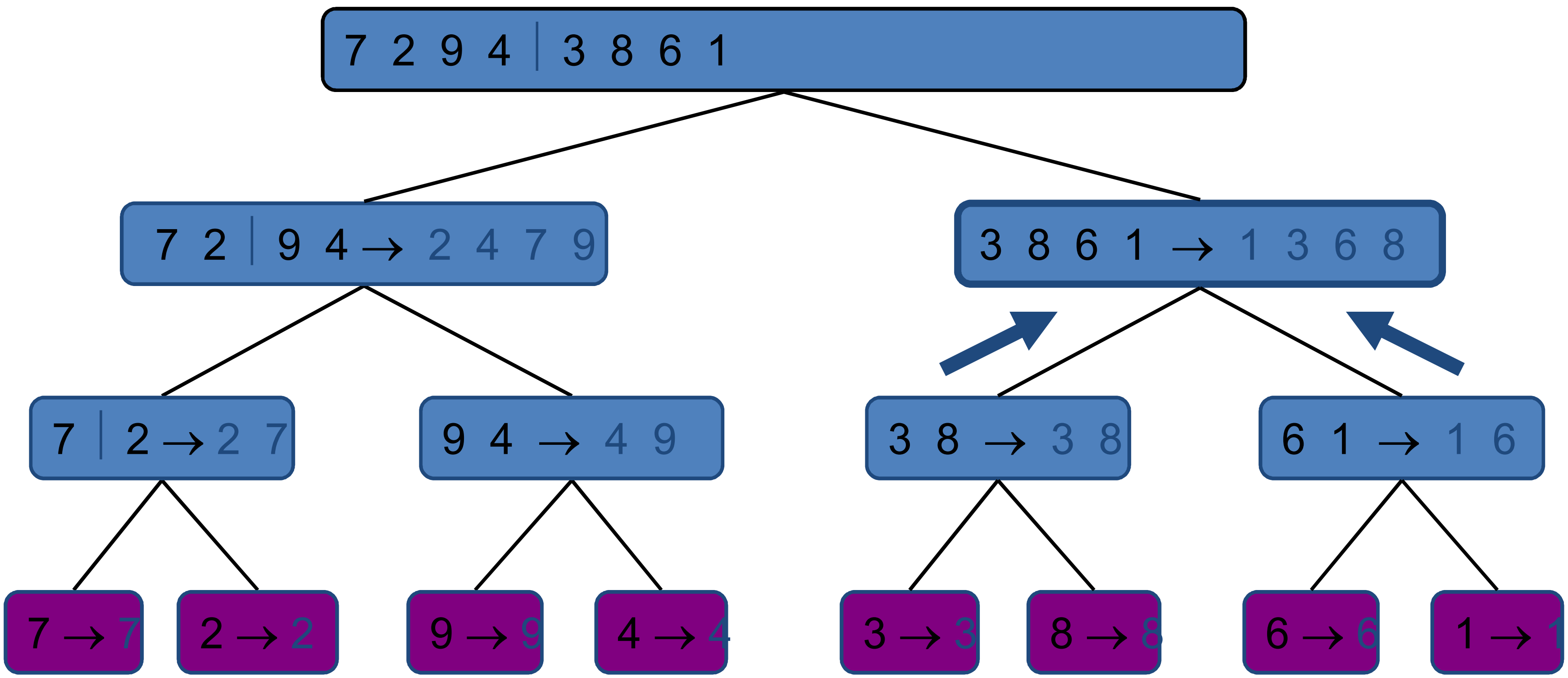
Execution Example (cont.)

- Merge



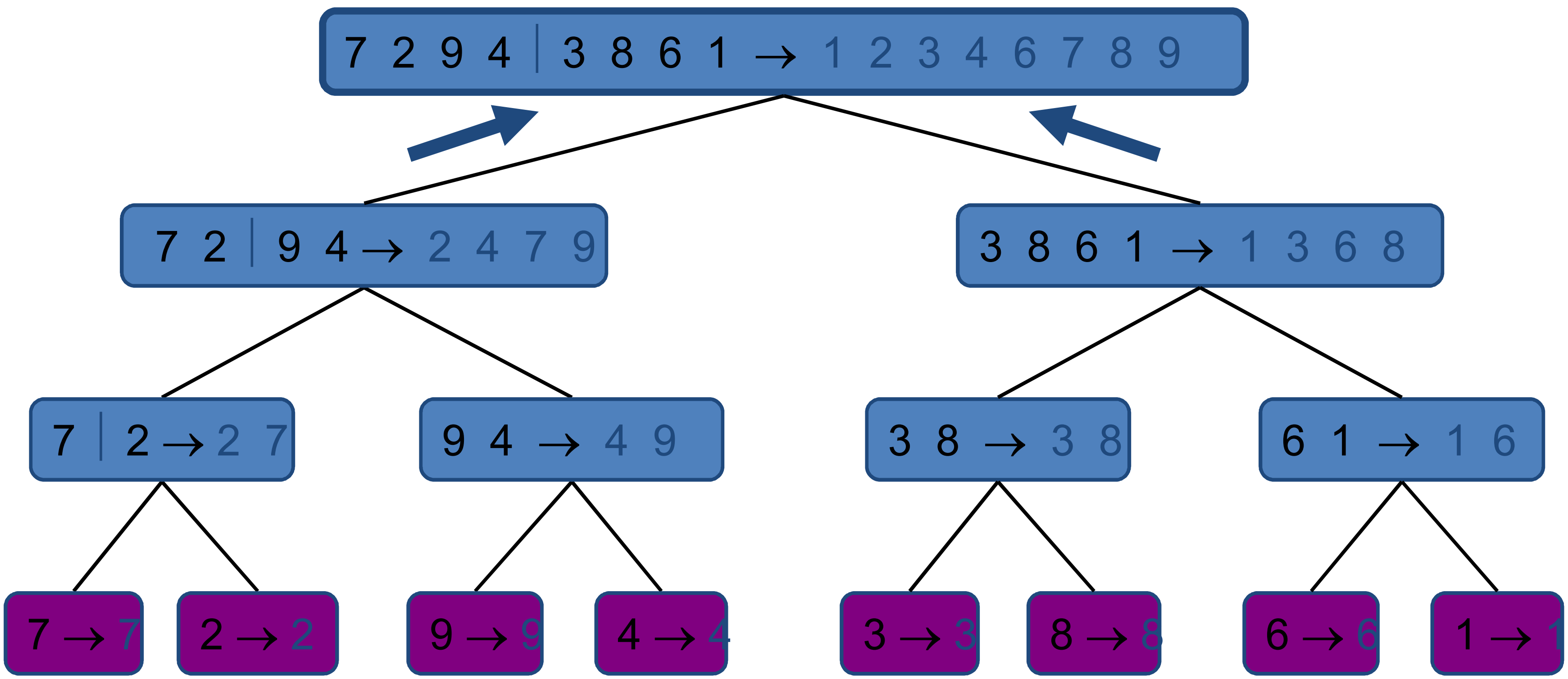
Execution Example (cont.)

- Recursive call, …, merge, merge



Execution Example (cont.)

- Merge

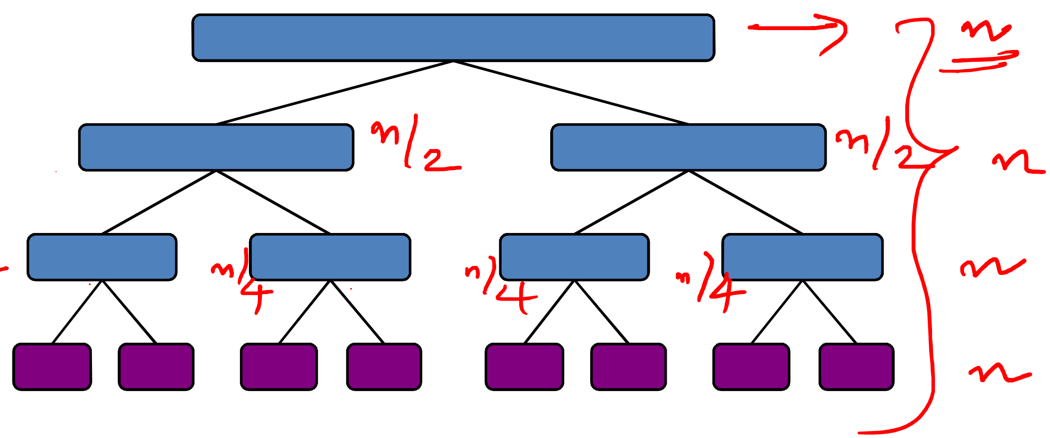


Analysis of Merge-Sort

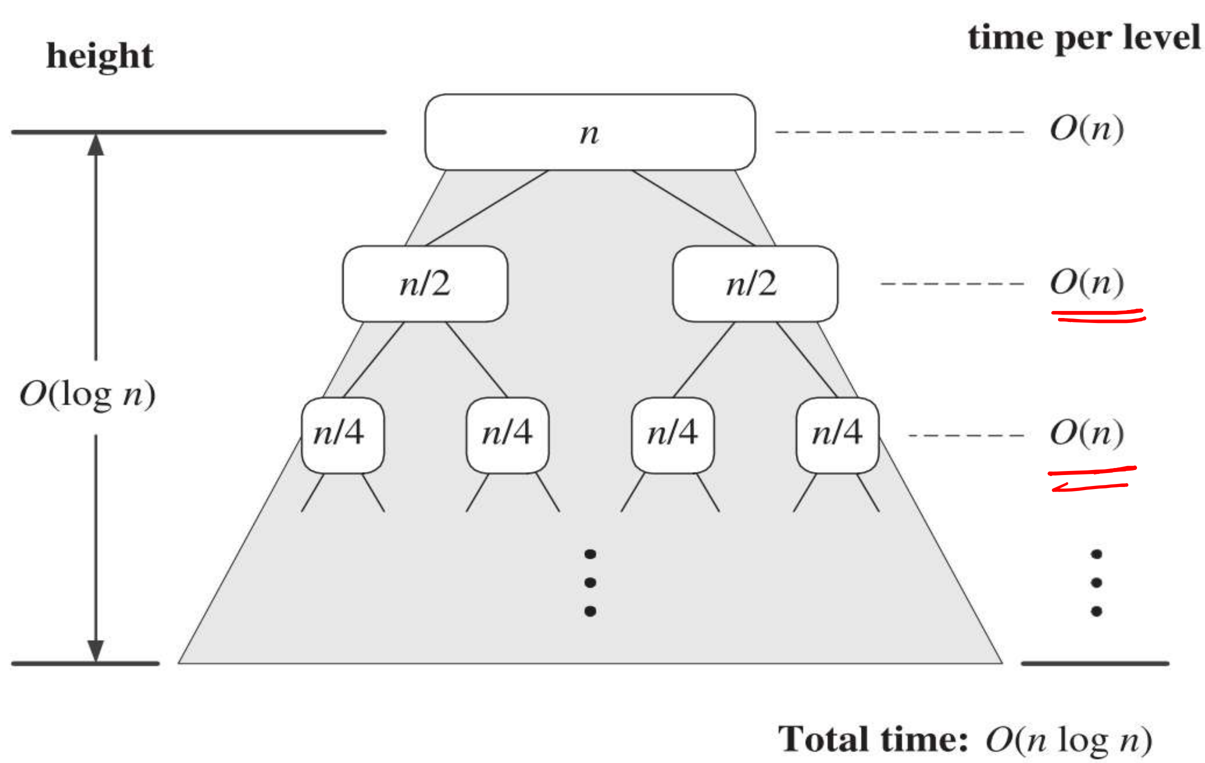
- An execution of merge-sort is depicted by a binary tree
- The height h of the merge-sort tree is $O(\log n)$
  - at each recursive call we divide in half the sequence,
- The overall amount or work done at the nodes of depth i is $O(n)$
  - we partition and merge 2^i sequences of size $n/2^i$
  - we make 2^{i+1} recursive calls
- Thus, the total running time of merge-sort is $O(n \log n)$

depth	#seqs	size
0	1	n
1	2	$n/2$
i	2^i	$n/2^i$
…	…	…

log n



Analysis of Merge-Sort



Recurrence Equation Analysis

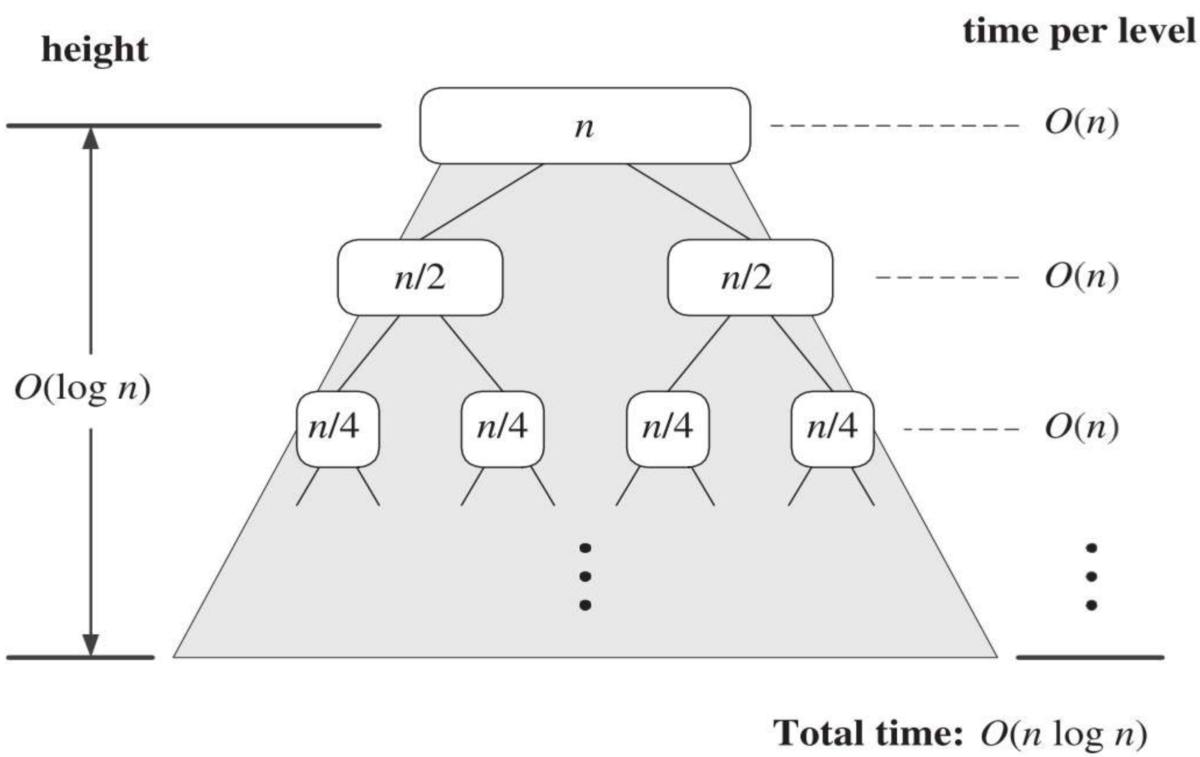
- The conquer step of merge-sort consists of merging two sorted sequences, each with ***n***/2 elements and implemented by means of a doubly linked list, takes at most ***bn*** steps, for some constant ***b***.
- Likewise, the basis case (***n*** < 2) will take at ***b*** most steps.
- Therefore, if we let ***T*(*n*)** denote the running time of merge-sort:

$$T(n) = \begin{cases} b & \text{if } n < 2 \\ 2T(n/2) + bn & \text{if } n \geq 2 \end{cases}$$

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

$$= 2\left[2T\left(\frac{n}{2^2}\right) + b\left(\frac{n}{2}\right)\right] + bn$$

Merge sort - Analysis



$$t(n) = \begin{cases} b & \text{if } n = 1 \text{ or } n = 0 \\ t(\lceil n/2 \rceil) + t(\lfloor n/2 \rfloor) + cn & \text{otherwise} \end{cases}$$

Assume, n is a power of 2 :

$$t(n) = \begin{cases} b & \text{if } n = 1 \\ 2t(n/2) + cn & \text{otherwise} \end{cases}$$

Merge sort - Analysis

$$t(n) = \begin{cases} b & \text{if } n = 1 \\ 2t(n/2) + cn & \text{otherwise} \end{cases}$$

By repeated substitution,

$$\begin{aligned} t(n) &= 2\left(2t\left(n/2^2\right) + (cn/2)\right) + cn \\ &= 2^2 t\left(n/2^2\right) + 2cn. \end{aligned}$$

Merge sort - Analysis

$$t(n) = \begin{cases} b & \text{if } n = 1 \\ 2t(n/2) + cn & \text{otherwise} \end{cases}$$

By repeated substitution,

$$\begin{aligned} t(n) &= 2\left(2t\left(n/2^2\right) + (cn/2)\right) + cn \\ &= 2^2 t\left(n/2^2\right) + 2cn. \end{aligned}$$

Substituting Again…

$$t(n) = 2^3 t\left(n/2^3\right) + 3cn$$

Substituting Again…

$$t(n) = 2^4 t\left(n/2^4\right) + 4cn$$

Merge sort - Analysis

$$t(n) = \begin{cases} b & \text{if } n = 1 \\ 2t(n/2) + cn & \text{otherwise} \end{cases}$$

By repeated substitution,

$$\begin{aligned} t(n) &= 2\left(2t\left(n/2^2\right) + (cn/2)\right) + cn \\ &= 2^2 t\left(n/2^2\right) + 2cn. \end{aligned}$$

Substituting Again…

$$t(n) = 2^3 t\left(n/2^3\right) + 3cn$$

Substituting Again…

$$t(n) = 2^4 t\left(n/2^4\right) + 4cn$$

Generalizing it …

$$t(n) = 2^i t\left(n/2^i\right) + icn$$

Merge sort - Analysis

$$t(n) = \begin{cases} b & \text{if } n = 1 \\ 2t(n/2) + cn & \text{otherwise} \end{cases}$$

By repeated substitution,

$$\begin{aligned} t(n) &= 2\left(2t\left(n/2^2\right) + (cn/2)\right) + cn \\ &= 2^2 t\left(n/2^2\right) + 2cn. \end{aligned}$$

Substituting Again…

$$t(n) = 2^3 t\left(n/2^3\right) + 3cn$$

Substituting Again…

$$t(n) = 2^4 t\left(n/2^4\right) + 4cn$$

Generalizing it …

$$t(n) = 2^i t\left(n/2^i\right) + icn$$

when i=log n, 2^i= n

$$\begin{aligned} t(n) &= 2^{\log n} t\left(n/2^{\log n}\right) + (\log n)cn \\ &= nt(1) + cn\log n \\ &= nb + cn\log n. \end{aligned}$$

Merge sort - Analysis

$$t(n) = \begin{cases} b & \text{if } n = 1 \\ 2t(n/2) + cn & \text{otherwise} \end{cases}$$

By repeated substitution,

$$\begin{aligned} t(n) &= 2\left(2t\left(n/2^2\right) + (cn/2)\right) + cn \\ &= 2^2 t\left(n/2^2\right) + 2cn. \end{aligned}$$

Substituting Again…

$$t(n) = 2^3 t\left(n/2^3\right) + 3cn$$

Substituting Again…

$$t(n) = 2^4 t\left(n/2^4\right) + 4cn$$

Generalizing it …

$$t(n) = 2^i t\left(n/2^i\right) + icn$$

when i=log n, 2^i= n

$$\begin{aligned} t(n) &= 2^{\log n} t\left(n/2^{\log n}\right) + (\log n)cn \\ &= nt(1) + cn \log n \\ &= nb + cn \log n. \end{aligned}$$

This implies,

$t(n)$ is $O(n \log n)$

Master Method

- Many divide-and-conquer recurrence equations have the form:

$$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$$

- The Master Theorem:

1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
2. if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$, provided $af(n/b) \leq \delta f(n)$ for some $\delta < 1$.

Master Method

- $$T(n) = \begin{cases} 1 & when\ n = 1 \\ T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + T(n) & Otherwise \end{cases}$$

Time required to sort the left array

Time required to sort the right array

Time to merge once instance of the solution

– The general divide and conquer recurrence is of the form

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

Master Method

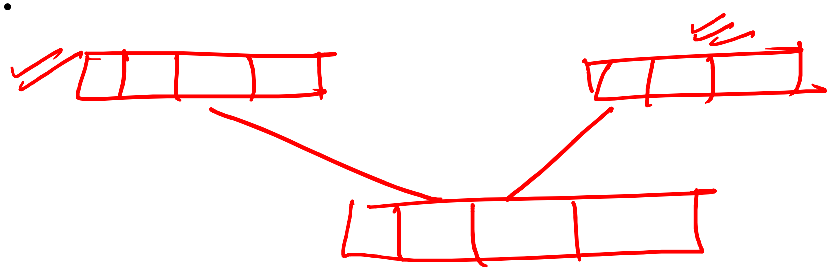
- $$T(n) = \begin{cases} 1 & when\, n = 1 \\ T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + T(n) & Otherwise \end{cases}$$

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

- Using Master method to solve
  - Case 2 applies
  - **$T(n)=\theta(n \log n)$**

Merge-Sort Properties

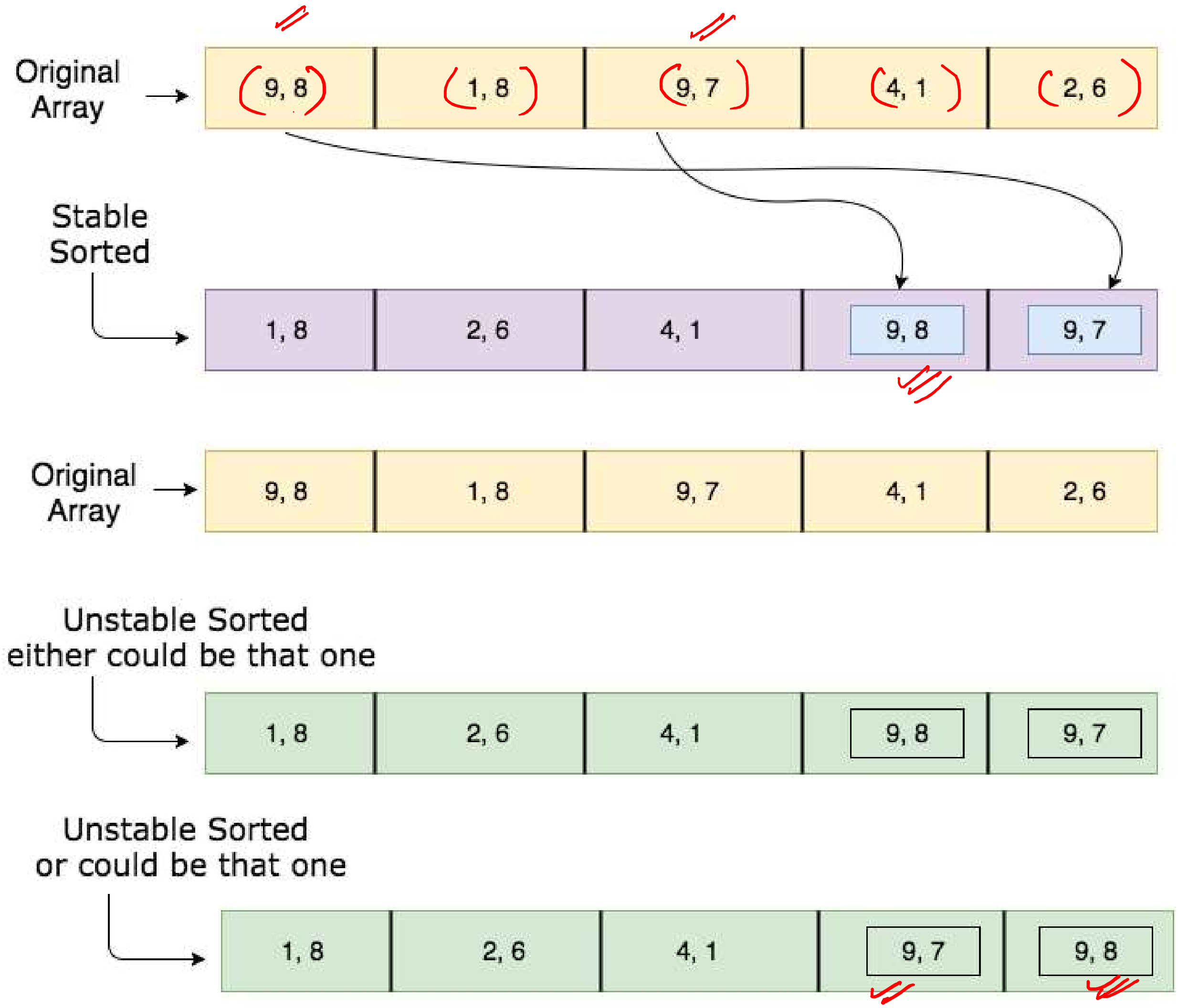
- **Not Adaptive** : Running time doesn't change with data pattern
  - algorithm will re-order every single item in the list even if it is already sorted.
- **Stable/ Unstable** : Both implementations are possible .
- **Not Incremental** : Does not sort one by one element in each pass.
- **Not online** : Need all data to be in memory at the time of sorting.
- **Not in place** : It need O(n) extra space to sort two sub list of size(n/2).



Merge-Sort Properties

- **Merge sort is a stable sort**
- A sorting algorithm is said to be **stable** if two objects with equal keys appear in the same order in sorted output as they appear in the input array to be sorted

Merge-Sort Properties



Merge-Sort Applications

- **Merge sort is often the best choice for sorting a linked list.**
  - Linked list nodes may not be adjacent in memory. In linked list, we can insert items in the end in O(1) extra space and O(1) time.
  - In linked list to access i'th index, we have to travel each and every node from the head to i'th node as we don't have continuous block of memory. Merge sort accesses data sequentially and the need of random access is low.

Merge-Sort Applications

- **External Sorting:**
  - External sorting is a class of sorting algorithms that can handle massive amounts of data. External sorting is required when the data being sorted do not fit into the main memory of a computing device (usually RAM) and instead they must reside in the slower external memory, usually a hard disk drive.
  - External merge sort uses a hybrid sort-merge strategy. In the sorting phase, chunks of data small enough to fit in main memory are read, sorted, and written out to a temporary file. In the merge phase, the sorted subfiles are combined into a single larger file.

Merge-Sort Applications

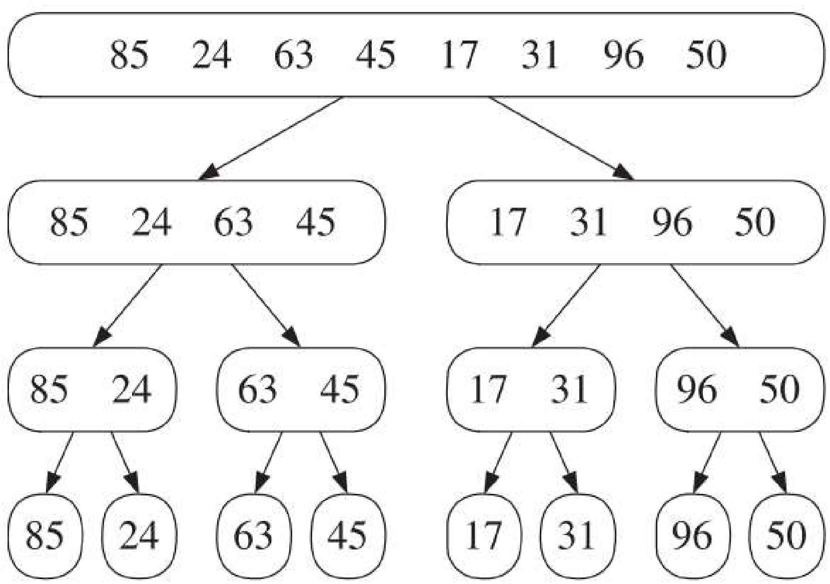
- In Java, the Arrays.sort() methods use merge sort
- The Linux kernel uses merge sort for its linked lists
- Python uses Timsort, another tuned hybrid of merge sort and insertion sort, that has become the standard sort algorithm in Java SE 7

Merge-Sort Applications

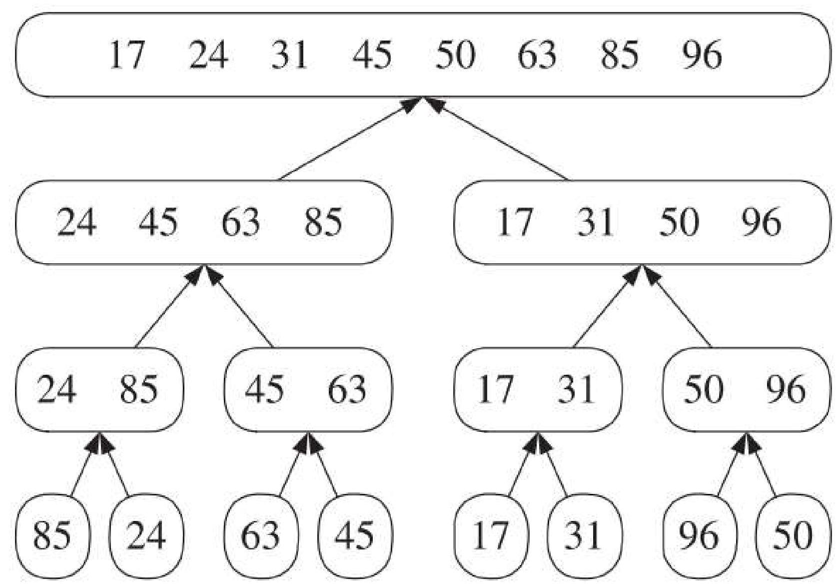
- The e-commerce application
- Have you ever noticed on any e-commerce website, they have this section of "You might like"
- They have maintained an array for all the user accounts and then whichever has the least number of inversion with your array of choices, they start recommending what they have bought or they like
- Array inversion count. [2, 4, 1, 3, 5]

(2, 1) (4, 1) (4, 3)

Merge sort - Example -2



Input sequence processed at each node - Divide



Output sequence generated at each node - Conquer

Integer Multiplication

- The problem of multiplying big integers, that is, integers represented by a large number of bits that cannot be handled directly by the arithmetic unit of a single processor.

Integer Multiplication

- Given two big integers I and J represented with n bits each, we can easily compute I + J and I - J in O(n) time.

13+6

$$\begin{array}{r} 1101 + \\ 0110 \\ \hline 10011 \end{array}$$

$$\begin{array}{r} 1101 - \\ 0110 \\ \hline 0111 \end{array}$$

13 − 6

Integer Multiplication

- Efficiently computing the product I · J using the common grade-school algorithm requires, however, O (n^2) time.

```
       101010
       010101
  ------------
       101010
      000000x
     101010
    000000x
   101010
  000000x
  ------------
  01101110010
```

Integer Multiplication-Divide and Conquer

- Divide and Conquer to Multiply - Attempt- #1
  - Let us represent I and J as below.
    - Attempt to rewrite the multiplication of I and J in terms of their components.
      - That is, this gives raise to recursion

211×89

$$I \rightarrow \overset{I_h}{\underbrace{1101}}\ \overset{I_L}{\underbrace{0011}}$$

$$J \rightarrow \underbrace{0101}_{J_h}\ \underbrace{1001}_{J_L}$$

Integer Multiplication-Divide and Conquer

- Divide and Conquer to Multiply - Attempt- #1

211 × 89

$I \rightarrow \underbrace{1101}_{I_h}\ \underbrace{0011}_{I_L}$

$J \rightarrow \underbrace{0101}_{J_h}\ \underbrace{1001}_{J_L}$

Now,

$$I = I_h 2^{n/2} + I_l$$

$$J = J_h 2^{n/2} + J_l$$

Example

$\underbrace{12}_{a}\underbrace{34}_{b}$

$= 12 \times 10^2 + 34$

$= 1200 + 34$

$= 1234$

$x = ab$

$y = cd$

ie $x = a \times 10^{n/2} + b$

$y = c \times 10^{n/2} + d$

Integer Multiplication-Divide and Conquer

– We can then define I*J by multiplying the parts and adding:

$$I = I_h 2^{n/2} + I_l$$

$$J = J_h 2^{n/2} + J_l$$

$$I * J = (I_h 2^{n/2} + I_l) * (J_h 2^{n/2} + J_l)$$

$$= I_h J_h 2^n + I_h J_l 2^{n/2} + I_l J_h 2^{n/2} + I_l J_l$$

$x = ab \quad y = cd$

$x = (a \times 10^{n/2} + b)$

$y = (c \times 10^{n/2} + d)$

$\therefore xy = (a \times 10^{n/2} + b) \times (c \times 10^{n/2} + d)$

ie $xy = (ac \times 10^{n}) + (ad + bc) 10^{n/2} + bd$

Integer Multiplication-Divide and Conquer

n

$$I * J = (I_h 2^{n/2} + I_l) * (J_h 2^{n/2} + J_l)$$

$$= \underbrace{I_h J_h 2^n}_{1} + \underbrace{I_h J_l 2^{n/2}}_{2} + \underbrace{I_l J_h 2^{n/2}}_{3} + \underbrace{I_l J_l}_{4}$$

$\left(\frac{n}{2}\right)$

$5 \times 2^1 = 10$

$5 = 101$

$101 \leftarrow 0$

$10 = 1010$

Multiplication of 2 n bit numbers is now broken down into

- 4 Multiplications of n/2 bit numbers
- Plus 3 Additions (n)

Multiplying any binary numbers by any arbitrary power of 2 is just a shift operation of bits

T(n) = 4T(n/2) + n,
implies T(n) is O(n^2)

Integer Multiplication-Divide and Conquer

- $O(n^2)$ is not an improvement indeed.
  - But we are able to multiply large integers in terms of smaller ones, now !
- We need to compute the following with lesser # of multiplication

$$I \cdot J = I_h J_h 2^n + I_l J_h 2^{n/2} + I_h J_l 2^{n/2} + I_l J_l$$

Integer Multiplication-Divide and Conquer

- Let us try to rewrite the following with 3 multiplications

$$I \cdot J = I_h J_h 2^n + I_l J_h 2^{n/2} + I_h J_l 2^{n/2} + I_l J_l \rightarrow 1$$

- Let $P1 = (I_h + I_l) * (J_h + J_l)$

  $= I_h J_h + I_h J_l + I_l J_h + I_l J_l$

  $P2 = I_h J_h$

  $P3 = I_l J_l$

  $P1 - P2 - P3 = I_h J_l + I_l J_h$

Now

$I * J = P2 * 2^n + [P1 - P2 - P3] * 2^{n/2} + P3 \rightarrow 2$

Integer Multiplication-Divide and Conquer

- Let us try to rewrite the following with 3 multiplications

$$I \cdot J \;=\; I_h J_h 2^n + I_l J_h 2^{n/2} + I_h J_l 2^{n/2} + I_l J_l$$

- Let P1=$(I_h+I_l) * (J_h+J_l)$ 2, $\frac{n}{2}$ bit Addtn
  1, $\frac{n}{2}$ x-tion

  $=I_hJ_h + I_hJ_l + I_lJ_h + I_lJ_l$

  P2= $I_hJ_h \rightarrow \frac{n}{2}$ x-tion

  P3= $I_lJ_l \rightarrow \frac{n}{2}$ x-tion

  P1-P2-P3= $I_hJ_l + I_lJ_h$

Now

$$I*J = \underset{\uparrow 1}{P2} * 2^n + \underset{\uparrow 1}{[P1-P2-P3]} * 2^{n/2} + \underset{\uparrow 1}{P3}$$

T(n) = 3T(n/2) + n,
By Master Theorem.
T(n) is $O(n^{\log_2 3})$,
Thus, T(n) is $O(n^{1.585})$.

Algorithm

Input: Positive integers x and y, in binary
Output: Their product

$n =$ max(size of x, size of y)
if $n = 1$: return xy

x_L, $x_R =$ leftmost $\lceil n/2 \rceil$, rightmost $\lfloor n/2 \rfloor$ bits of x
y_L, $y_R =$ leftmost $\lceil n/2 \rceil$, rightmost $\lfloor n/2 \rfloor$ bits of y

$P_1 = \text{multiply}(x_L, y_L)$
$P_2 = \text{multiply}(x_R, y_R)$
$P_3 = \text{multiply}(x_L + x_R, y_L + y_R)$
return $P_1 \times 2^n + (P_3 - P_1 - P_2) \times 2^{n/2} + P_2$

An Improved Integer Multiplication Algorithm

- Algorithm: Multiply two n-bit integers I and J.
- **O(n log n)** , by using a more complex divide-and-conquer algorithm called the Fast Fourier transform

Example-Decimal

$\Rightarrow \overset{a}{1980} \times \overset{b}{2315}$

$a_L = 19 \quad b_L = 23$

$a_R = 80 \quad b_R = 15$

$x_1 = (19+80)(23+15) = 3762$

$= 99 \times 38$

$x_2 = 19 \times 23$

$x_3 = 80 \times 15$

Let $x_1 = (a_L + a_R)(b_L + b_R)$

$x_2 = a_L b_L$

$x_3 = a_R b_R$

$a \times b = x_2 \times 10^n + (x_1 - x_2 - x_3) 10^{n/2} + x_3$

99×38

$x_{11} = (9+9)(3+8) = 18 \times 11 = 198$

$x_{12} = 9 \times 3 = 27$

$x_{13} = 9 \times 8 = 72$

$a \times b = 27 \times 10^2 + (198 - 27 - 72) 10 + 72$

$= 2700 + 990 + 72$

$= 3762$

18×11

$x_{111} = (\overset{9}{1+8})(\overset{2}{1+1}) = 18$

$x_{112} = 1 \times 1 = 1$

$x_{113} = 8 \times 1 = 8$

$a \times b = x_2 \times 10^n + (x_1 - x_2 - x_3) \times 10^{n/2} + x_3$

$= 1 \times 10^2 + (18 - 1 - 8) 10 + 8 = 100 + 90 + 8 = 198$

Application

- Multiplying big integers has applications to data security, where big integers are used in encryption schemes.
  - More specifically, some important cryptographic algorithms such as RSA critically depend on the fact that prime factorization of large numbers takes a long time. Basically you have a "public key" consisting of a product of two large primes used to encrypt a message, and a "secret key" consisting of those two primes used to decrypt the message. You can make the public key public, and everyone can use it to encrypt messages to you, but only you know the prime factors and can decrypt the messages. Everyone else would have to factor the number, which takes too long to be practical, given the current state of the art of number theory.

BITS Pilani
Hyderabad Campus

THANK YOU!