CME 2001
Data Structures and Algorithms

Zerrin Işık
zerrin@cs.deu.edu.tr

Sorting Algorithms and Their Analysis

Sorting Problem?

- **Input:** A sequence of n numbers $(a_1, a_2, \ldots, a_n)$
- **Output:** A permutation (reordering) $(a_1', a_2', \ldots, a_n')$ of the input sequence such that $a_1' \leq a_2' \leq \ldots \leq a_n'$.
- **Example:**

  Input : 3 7 9 1 2

  Output : 1 2 3 7 9

Insertion Sort

- A good algorithm for sorting a small number of elements.
- Lets assume you will sort a hand of playing cards:
  - Start with an empty left hand and the cards face down on the table.
  - Each time remove one card from the table, and insert it into the correct position in the left hand.
  - To find the correct position for a card, compare it with each of the cards already in the hand, from right to left.
  - At all times, the cards held in the left hand are sorted, and these cards were originally the top cards of the pile on the table.

```
INSERTION-SORT(A, n)
  for j = 2 to n
      key = A[j]
      // Insert A[j] into the sorted sequence A[1 . . j − 1].
      i = j − 1
      while i > 0 and A[i] > key
          A[i + 1] = A[i]
          i = i − 1
      A[i + 1] = key
```

Insertion Sort Example

Sorted **Unsorted**

12	67	34	3	25	45

Original Array

Insertion Sort Example

Sorted **Unsorted**

12	67	34	3	25	45	Original Array
12	67	34	3	25	45	After 1. pass

Insertion Sort Example

Sorted **Unsorted**

12	67	34	3	25	45	Original Array
12	67	34	3	25	45	After 1. pass
12	34	67	3	25	45	After 2. pass

Insertion Sort Example

Sorted **Unsorted**

12	67	34	3	25	45	Original Array
12	67	34	3	25	45	After 1. pass
12	34	67	3	25	45	After 2. pass
3	12	34	67	25	45	After 3. pass

Insertion Sort Example

Sorted **Unsorted**

12	67	34	3	25	45	Original Array
12	67	34	3	25	45	After 1. pass
12	34	67	3	25	45	After 2. pass
3	12	34	67	25	45	After 3. pass
3	12	25	34	67	45	After 4. pass

Insertion Sort Example

Sorted **Unsorted**

12	67	34	3	25	45	Original Array
12	67	34	3	25	45	After 1. pass
12	34	67	3	25	45	After 2. pass
3	12	34	67	25	45	After 3. pass
3	12	25	34	67	45	After 4. pass
3	12	25	34	45	67	After 5. pass

Analysis of Algorithms

- How is the running time of an algorithm analyzed?
  - Based on the *input itself* and *input size*
- Input:
  - Sorting 100 numbers takes longer than sorting 5 numbers.
  - A sorting algorithm might takes different amounts of time on two inputs of the same size (e.g., assume one input is already sorted).
- Input Size:
  - Usually, the number of items in the input : n
  - For integer multiplication, it is the total number of bits in the two integers.

Types of Analysis

- **Best-Case**
  - Lower bound (i.e., minimum) on the running time for any input
- **Worst-Case** *(often guarantee)*
  - Upper bound (i.e., maximum) on the running time for any input
- **Average-Case**
  - Expected running time for any input, generally as bad as worst-case time

Running time

It is the number of primitive operations (steps) executed.

- Each line of pseudocode takes a constant amount of time.
- Execution of line i always takes the same time c_i.
- Assume that each line consists only of primitive operations.

The running time of an algorithm is:

$$\sum_{\text{all statements}} (\text{cost of statement}) \cdot (\text{number of times statement is executed})$$

Analysis of Insertion Sort

INSERTION-SORT(A, n)	*cost*	*times*
for $j = 2$ **to** n	c_1	n
$\quad$ $key = A[j]$	c_2	$n-1$
$\quad$ // Insert $A[j]$ into the sorted sequence $A[1\,..\,j-1]$.	0	$n-1$
$\quad$ $i = j - 1$	c_4	$n-1$
$\quad$ **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
$\qquad$ $A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n}(t_j - 1)$
$\qquad$ $i = i - 1$	c_7	$\sum_{j=2}^{n}(t_j - 1)$
$\quad$ $A[i+1] = key$	c_8	$n-1$

Analysis of Insertion Sort

INSERTION-SORT(A, n)	*cost*	*times*
for $j = 2$ **to** n	c_1	n
$\quad key = A[j]$	c_2	$n-1$
$\quad$// Insert $A[j]$ into the sorted sequence $A[1\,..\,j-1]$.	0	$n-1$
$\quad i = j - 1$	c_4	$n-1$
$\quad$**while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
$\qquad A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
$\qquad i = i - 1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
$\quad A[i+1] = key$	c_8	$n-1$

$$
\begin{aligned}
T(n) \;=\; & c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) \\
& + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)\ .
\end{aligned}
$$

Best-Case Running Time

Assume the input is already sorted:

- Always find that *A[i]* ≤ *key* upon the first time **while** loop is run
- All t_j are 1.
- The running time is:

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) \ . \end{aligned}$$

- Can express *T(n)* as *an + b* for constants *a* and *b* :

  => *T(n)* is a *linear function* of *n*

Worst-Case Running Time

Assume the input is in reverse sorted order:

- Always find that *A[i]* > *key* in the **while** loop test.
- Compare *key* with all elements to the left of the j^{th} position.
- The **while** loop reaches to 0, one more test after the j-1 test => $t_j = j$.
- The running time is:

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\
&\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\
&= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n \\
&\quad - (c_2 + c_4 + c_5 + c_8)\ .
\end{aligned}
$$

- Can express *T(n)* as $an^2 + bn + c$ for constants *a*, *b*, *c* :

  => *T(n)* is a *quadratic function* of *n*

Average-Case Running Time

Assume we randomly choose ***n*** number as the input for insertion sort:

- On average, the key in A[j] is less than half the elements in A[1…j-1] and it's greater than the other half => $t_j \approx (j/2)$.
- The average-case running time is approximately half of the worst-case running time, it's still a *quadratic function* of *n*.

Order of Growth

- Only consider the leading term of the formula for running time.
- Drop lower-order terms
- Ignore constant coefficient in the leading term
- For insertion sort, we already abstracted away the actual statement costs to conclude that the worst-case running time is $an^2 + bn + c$.
  - Drop lower-order terms => an^2.
  - Ignore constant coefficient => n^2.
- We cannot say that the worst-case running time $T(n)=n^2$. It only *grows like* n^2.
- So, the running time is $\Theta(n^2)$ to capture the notion that the *order of growth* is n^2.
- One algorithm is assumed to be more efficient if its worst-case running time has a smaller order of growth.

Designing Algorithms

- Many ways to design algorithms.
- Insertion sort is ***incremental*** : having sorted *A[1…j-1]*, place *A[j]* correctly, so that *A[1…j]* is sorted.
- Another common approach is **Divide and Conquer**.

Divide and Conquer Algorithms

- **Divide** problem into sub-problems.
- **Conquer** by solving sub-problems recursively. If the sub-problems are small enough, solve them in brute force fashion.
- **Combine** the solutions of sub-problems into a solution of the original problem.

Merge Sort

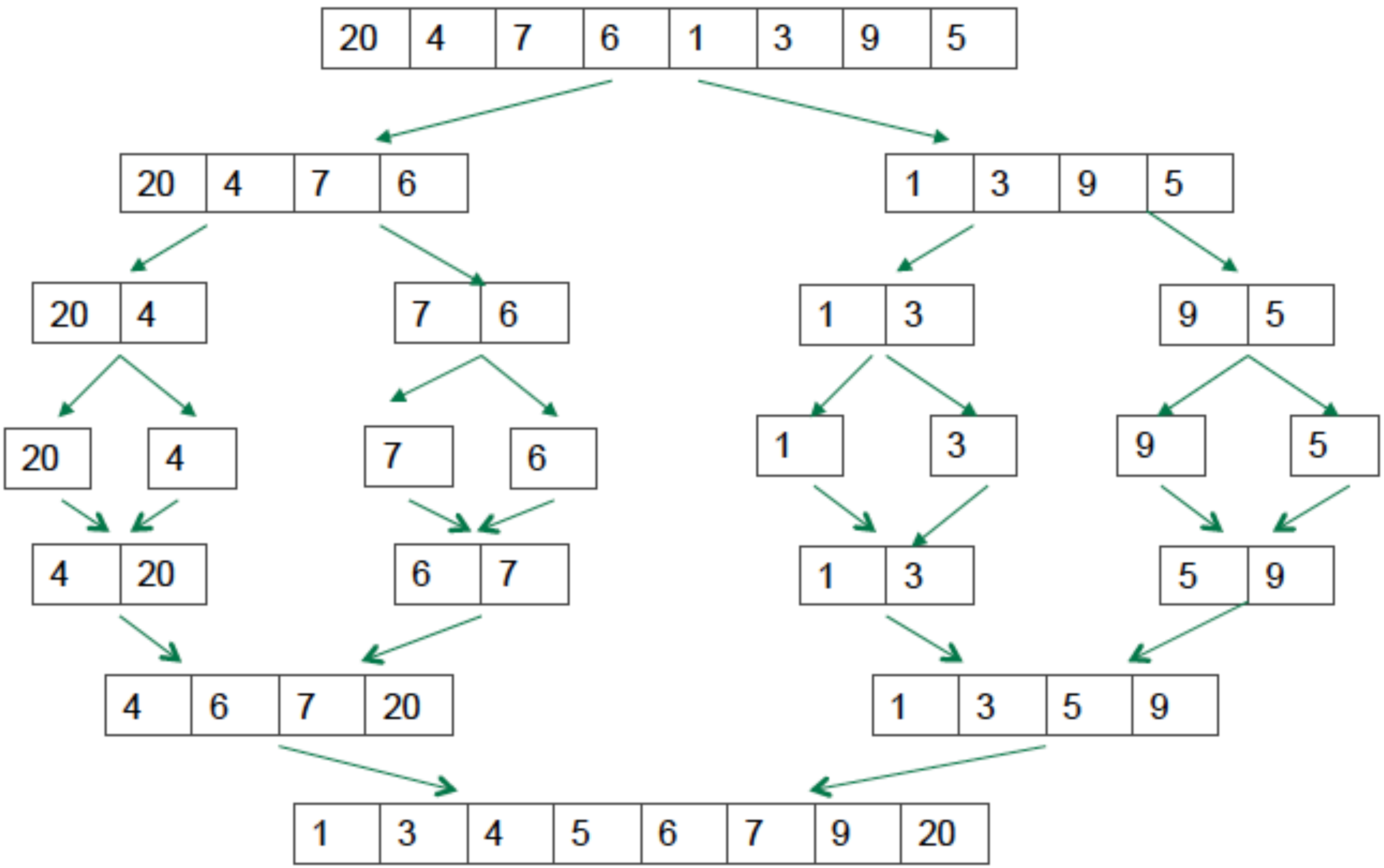
Define each sub-problem as sorting a sub-array *A[p…r]*.

Initially: $p=1$, $r=n$ (these values change as we recurse through sub-problems)

To sort *A[p…r]*:

- **Divide** by splitting into two sub-arrays *A[p…q]* and *A[q+1…r]*, where *q* is the halfway point of *A[p…q]*.
- **Conquer** by recursively sorting two sub-arrays *A[p…q]* and *A[q+1…r]*.
- **Combine** by merging two sorted sub-arrays *A[p…q]* and *A[q+1…r]* to create a single sorted sub-array *A[p…r]*. To perform this task define a *MERGE(A,p,q,r)* subroutine.

Merge Sort Example



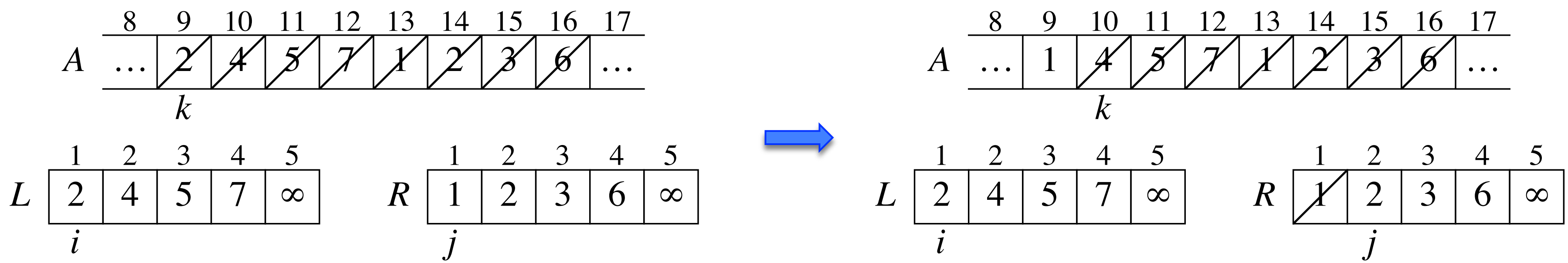
```
MERGE-SORT(A, p, r)
  if p < r
      q = ⌊(p + r)/2⌋
      MERGE-SORT(A, p, q)
      MERGE-SORT(A, q + 1, r)
      MERGE(A, p, q, r)
```

```
MERGE(A, p, q, r)
  n1 = q − p + 1
  n2 = r − q
  let L[1 .. n1 + 1] and R[1 .. n2 + 1] be new arrays
  for i = 1 to n1
      L[i] = A[p + i − 1]
  for j = 1 to n2
      R[j] = A[q + j]
  L[n1 + 1] = ∞
  R[n2 + 1] = ∞
  i = 1
  j = 1
  for k = p to r
      if L[i] ≤ R[j]
          A[k] = L[i]
          i = i + 1
      else A[k] = R[j]
          j = j + 1
```

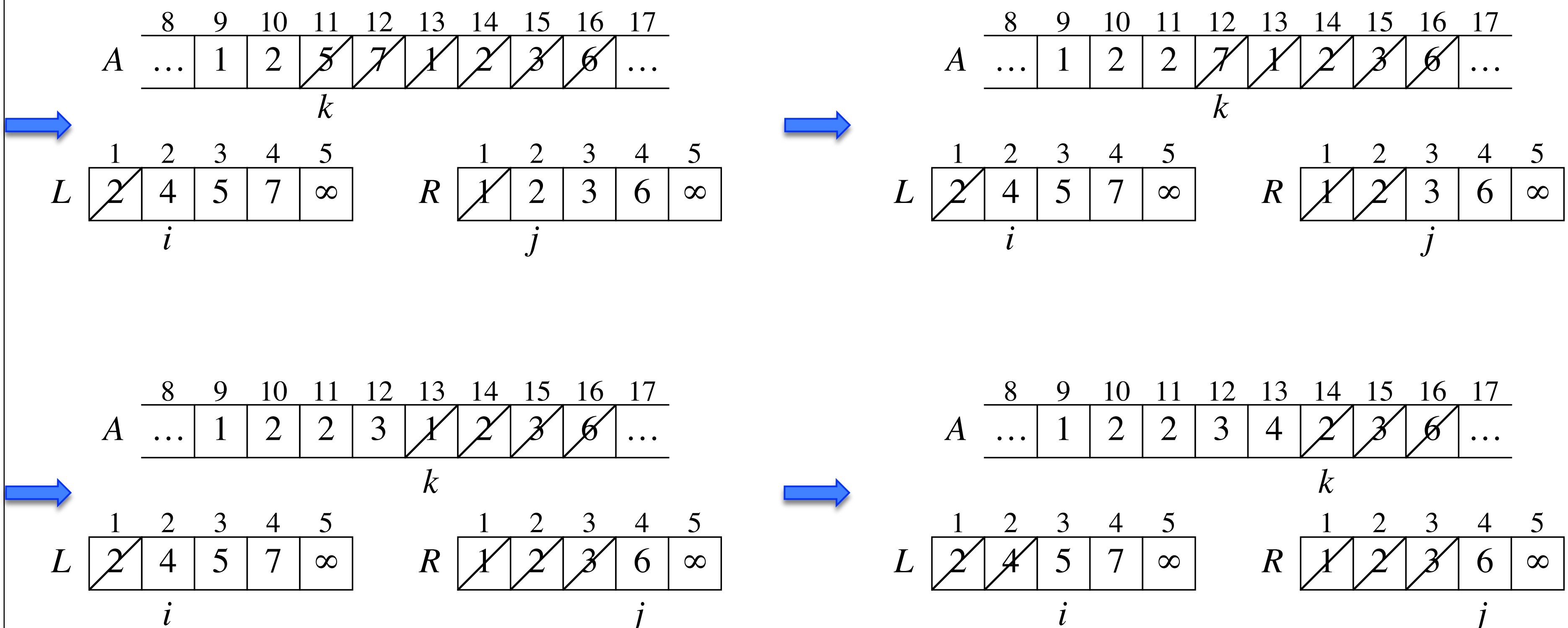
Note: The recursion (MERGE-SORT call) will end when the sub-array has just 1 element, it's already sorted.

Merge Sort Example

A call of MERGE(A, 9, 12, 16)



Merge Sort Example



Merge Sort Example

	8	9	10	11	12	13	14	15	16	17
A	…	1	2	2	3	4	5	~~3~~	~~6~~	…
								k		

	1	2	3	4	5
L	~~2~~	~~4~~	~~5~~	7	∞
				i	

	1	2	3	4	5
R	~~1~~	~~2~~	~~3~~	6	∞
				j	

	8	9	10	11	12	13	14	15	16	17
A	…	1	2	2	3	4	5	6	~~6~~	…
									k	

	1	2	3	4	5
L	~~2~~	~~4~~	~~5~~	7	∞
				i	

	1	2	3	4	5
R	~~1~~	~~2~~	~~3~~	~~6~~	∞
					j

	8	9	10	11	12	13	14	15	16	17
A	…	1	2	2	3	4	5	6	7	…
										k

	1	2	3	4	5
L	~~2~~	~~4~~	~~5~~	~~7~~	∞
					i

	1	2	3	4	5
R	~~1~~	~~2~~	~~3~~	~~6~~	∞
					j

MERGE-SORT(A, p, r)

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

MERGE$(A, p, q, r) \Rightarrow \Theta(n)$

$n_1 = q - p + 1$
$n_2 = r - q$
let $L[1 \,..\, n_1 + 1]$ and $R[1 \,..\, n_2 + 1]$ be new arrays
for $i = 1$ **to** n_1
  $L[i] = A[p + i - 1]$ $\quad \Big\} \; \Theta(n_1)$
for $j = 1$ **to** n_2
  $R[j] = A[q + j]$ $\quad \Big\} \; \Theta(n_2)$
$L[n_1 + 1] = \infty$
$R[n_2 + 1] = \infty$
$i = 1$
$j = 1$
for $k = p$ **to** r
  if $L[i] \leq R[j]$
    $A[k] = L[i]$
    $i = i + 1$ $\quad \Bigg\} \; \Theta(n)$
  else $A[k] = R[j]$
    $j = j + 1$

MERGE-SORT(A, p, r)

if $p < r$

$q = \lfloor (p + r)/2 \rfloor$ $\Theta(1)$

MERGE-SORT(A, p, q)

MERGE-SORT$(A, q + 1, r)$ $\Big\}$ $2T(n/2)$

MERGE(A, p, q, r) $\Theta(n)$

MERGE$(A, p, q, r) \Rightarrow \Theta(n)$

$n_1 = q - p + 1$

$n_2 = r - q$

let $L[1 \, . . \, n_1 + 1]$ and $R[1 \, . . \, n_2 + 1]$ be new arrays

for $i = 1$ **to** n_1

$L[i] = A[p + i - 1]$ $\Big\}$ $\Theta(n_1)$

for $j = 1$ **to** n_2

$R[j] = A[q + j]$ $\Big\}$ $\Theta(n_2)$

$L[n_1 + 1] = \infty$

$R[n_2 + 1] = \infty$

$i = 1$

$j = 1$

for $k = p$ **to** r

if $L[i] \le R[j]$

$A[k] = L[i]$

$i = i + 1$

else $A[k] = R[j]$

$j = j + 1$ $\Big\}$ $\Theta(n)$

MERGE-SORT(A, p, r)

```
if p < r
    q = ⌊(p + r)/2⌋                 Θ(1)
    MERGE-SORT(A, p, q)           ⎫
    MERGE-SORT(A, q + 1, r)       ⎭ 2T(n/2)
    MERGE(A, p, q, r)             Θ(n)
```

MERGE(A, p, q, r) ⟹ $\Theta(n)$

```
n1 = q - p + 1
n2 = r - q
let L[1 .. n1 + 1] and R[1 .. n2 + 1] be new arrays
for i = 1 to n1                   ⎫
    L[i] = A[p + i - 1]           ⎭ Θ(n1)
for j = 1 to n2                   ⎫
    R[j] = A[q + j]               ⎭ Θ(n2)
L[n1 + 1] = ∞
R[n2 + 1] = ∞
i = 1
j = 1
for k = p to r                    ⎫
    if L[i] ≤ R[j]                ⎪
        A[k] = L[i]               ⎪
        i = i + 1                 ⎬ Θ(n)
    else A[k] = R[j]              ⎪
        j = j + 1                 ⎭
```

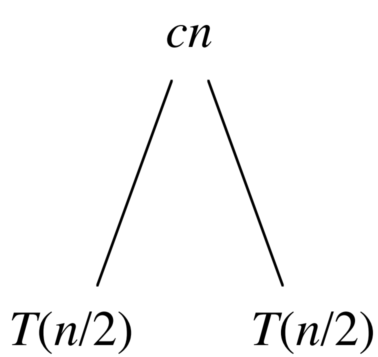
MERGE-SORT running time :

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

Recursion Tree for Recurrence

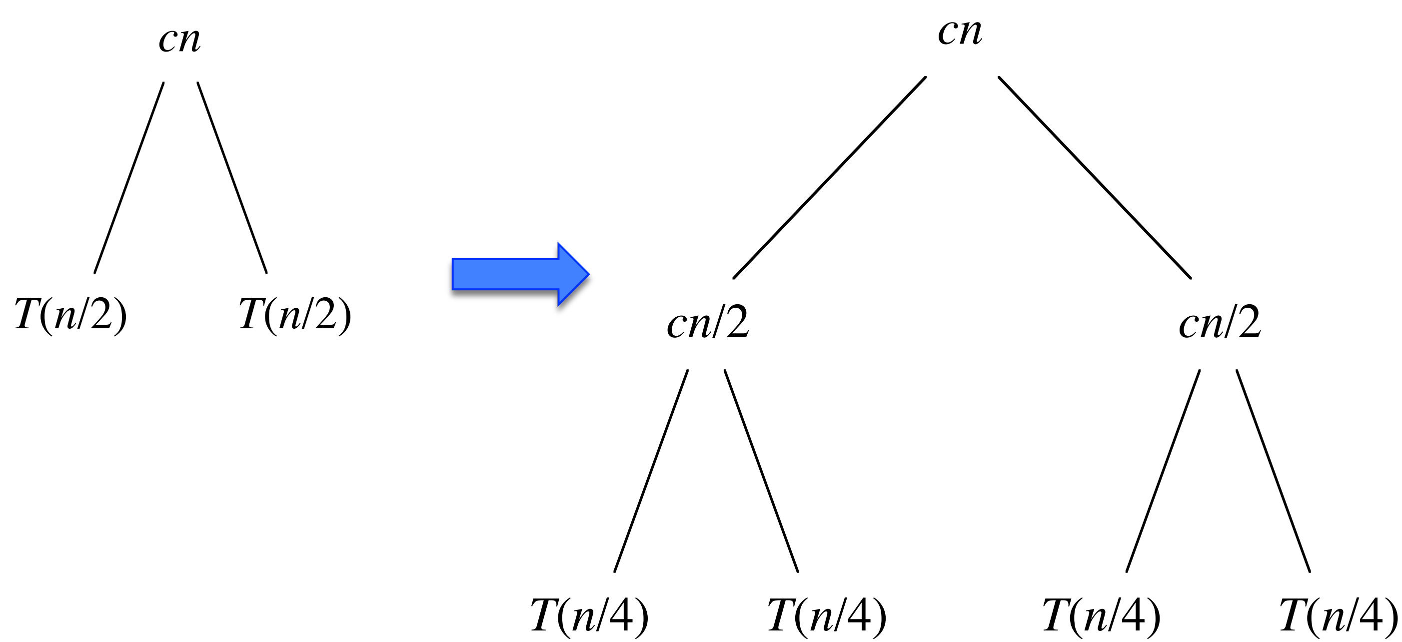
$$T(n) = \begin{cases} c & \text{if } n = 1 \,, \\ 2T(n/2) + cn & \text{if } n > 1 \,. \end{cases}$$

- Draw a **recursion tree** that shows successive expansions of the recurrence.
- We have a cost of ***cn*** and the two sub-problems, each one has a cost of ***T(n/2)***



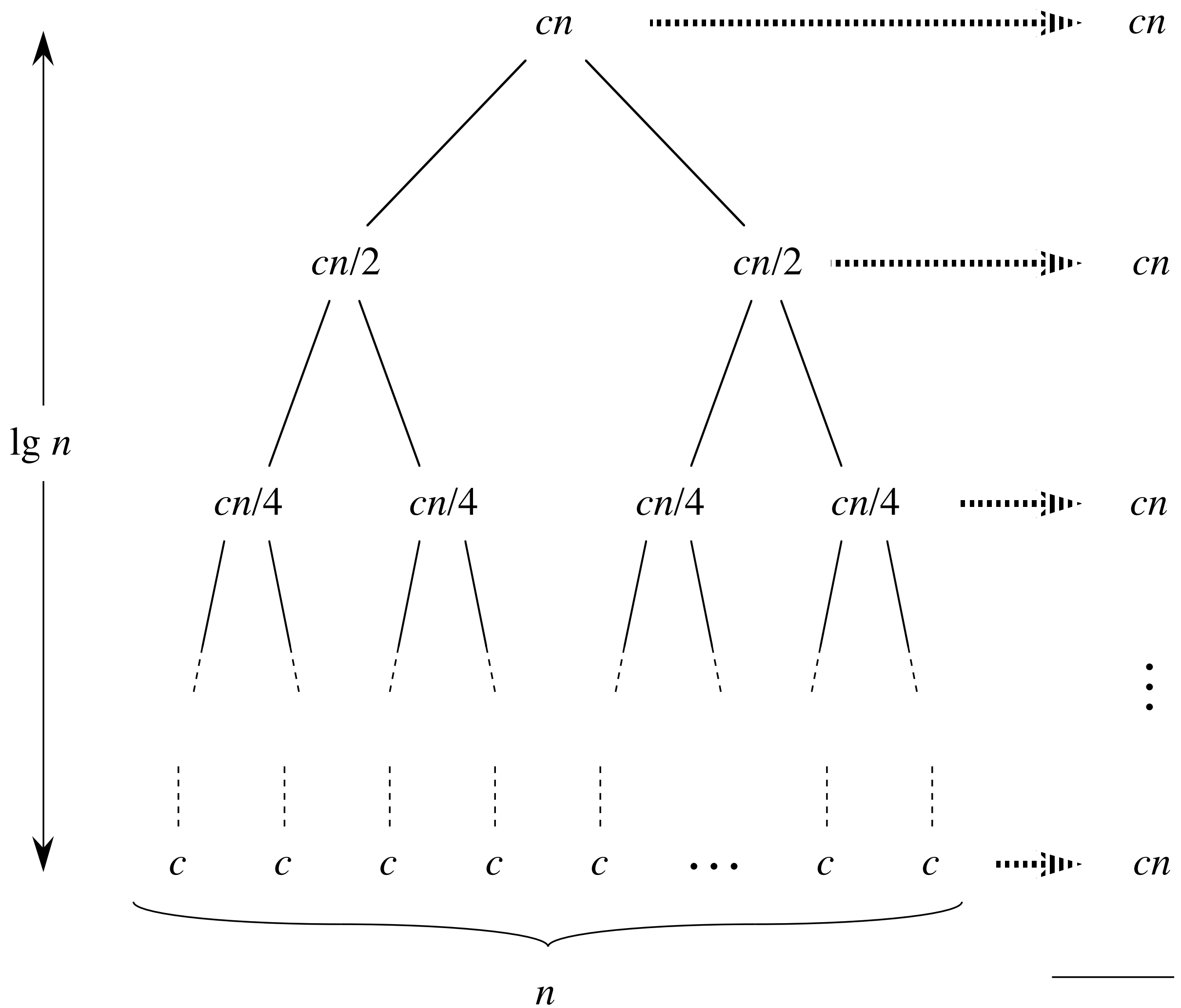
Recursion Tree for Recurrence

- For each of the ***size-n/2*** sub-problems, we have a cost of ***cn/2*** and the two sub-problems, each one has a cost of ***T(n/4)***



- Continue the expansion until the problem size becomes 1

Recursion Tree for Recurrence

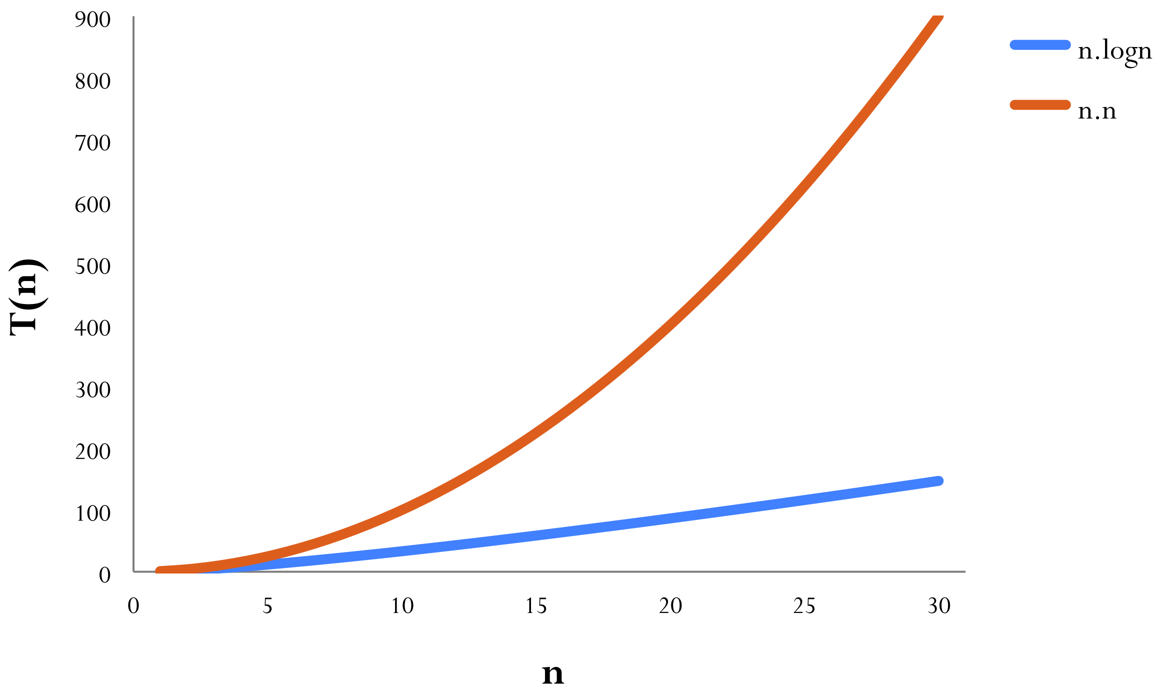


Height = logn
Levels = logn +1

Total: $cn \lg n + cn$

Comparison of Two Algorithms

- Merge Sort asymptotically beats Insertion Sort in the worst-case
- Because $\Theta(n.\log n)$ grows slowly than $\Theta(n^2)$



Next Week Topics

- Growth of Functions (Chapter 3-4)