Design and Analysis of Algorithms

Sorting

Si Wu

School of CSE, SCUT

cswusi@scut.edu.cn

TA: 1684350406@qq.com

The problem of sorting

Input: sequence $\langle a_1, a_2, \ldots, a_n \rangle$ of numbers.

Output: permutation $\langle a'_1, a'_2, \ldots, a'_n \rangle$ such that $a'_1 \le a'_2 \le \cdots \le a'_n$.

Example:

Input: 8 2 4 9 3 6

Output: 2 3 4 6 8 9

Overview

- ***Goals:***

  Start using frameworks for describing and analyzing algorithms.

  – See how to describe algorithms in pseudocode.

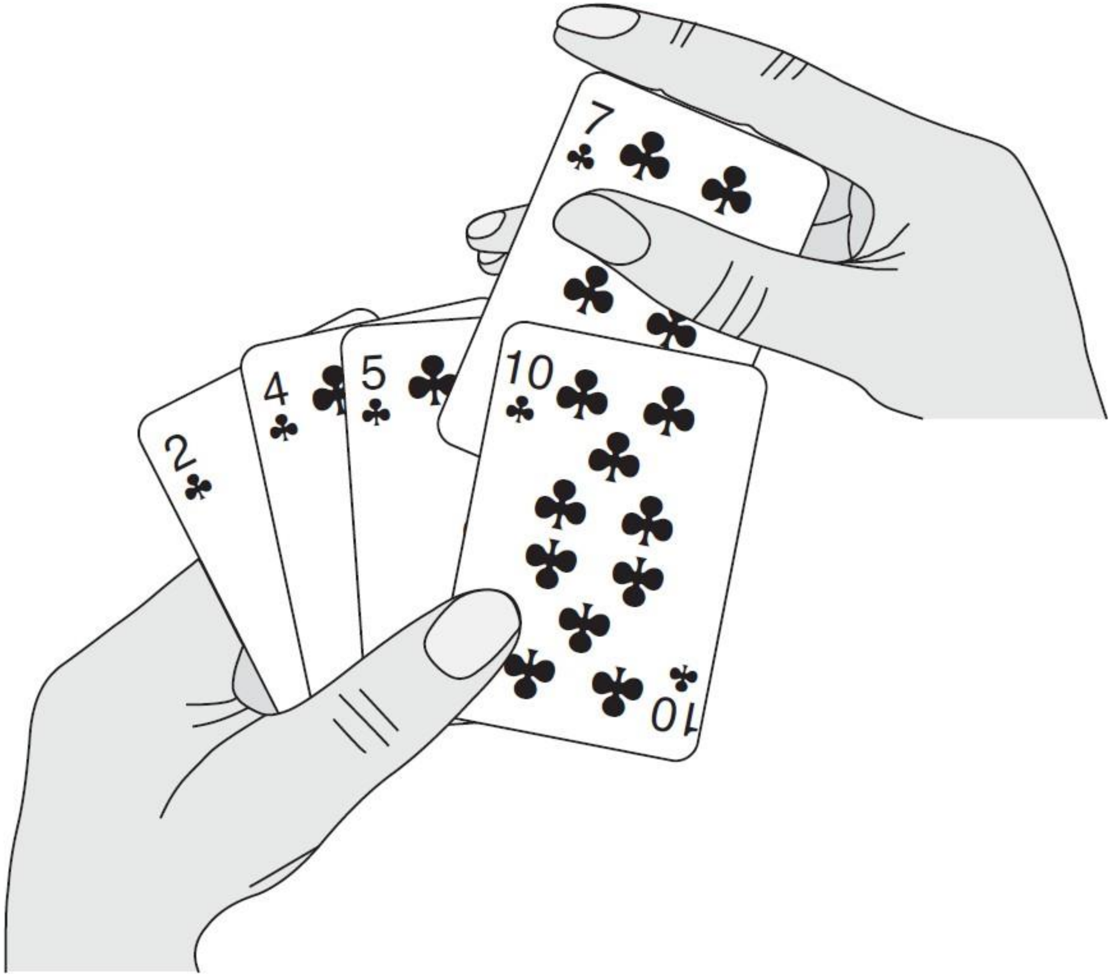
  – Begin using asymptotic notation to express running-time analysis.

  – Learn the technique of "divide and conquer" in the context of merge-sort.

  – Examine two algorithms for sorting: insertion-sort and merge-sort.

Insertion Sort

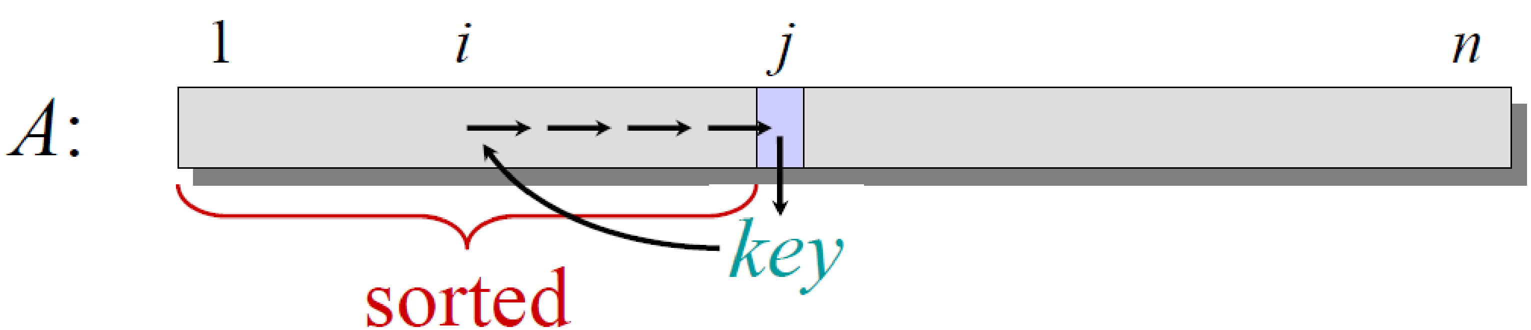
- Sorting a hand of cards using insertion sort.



Insertion sort

"pseudocode"

```
INSERTION-SORT (A, n)      ▷ A[1 . . n]
   for j ← 2 to n
        do key ← A[ j]
           i ← j – 1
           while i > 0 and A[i] > key
                do A[i+1] ← A[i]
                   i ← i – 1
           A[i+1] = key
```



Example of insertion sort

8 **2** 4 9 3 6

```
INSERTION-SORT (A, n)        ▷ A[1 . . n]
  for j ← 2 to n
       do key ← A[j]
          i ← j – 1
          while i > 0 and A[i] > key
               do A[i+1] ← A[i]
                  i ← i – 1
          A[i+1] = key
```

Example of insertion sort

8 2 4 9 3 6

Example of insertion sort

8	2	4	9	3	6
2	8	4	9	3	6

Example of insertion sort

8 2 4 9 3 6

2 8 4 9 3 6

Example of insertion sort

8	2	4	9	3	6
2	8	4	9	3	6
2	4	8	9	3	6

Example of insertion sort

8 2 4 9 3 6

2 8 4 9 3 6

2 4 8 9 3 6

Example of insertion sort

8	2	4	9	3	6
2	8	4	9	3	6
2	4	8	9	3	6
2	4	8	9	3	6

Example of insertion sort

8	**2**	4	9	3	6
2	8	**4**	9	3	6
2	4	8	**9**	3	6
2	4	8	9	**3**	6

Example of insertion sort

8	2	4	9	3	6
2	8	4	9	3	6
2	4	8	9	3	6
2	4	8	9	3	6
2	3	4	8	9	6

Example of insertion sort

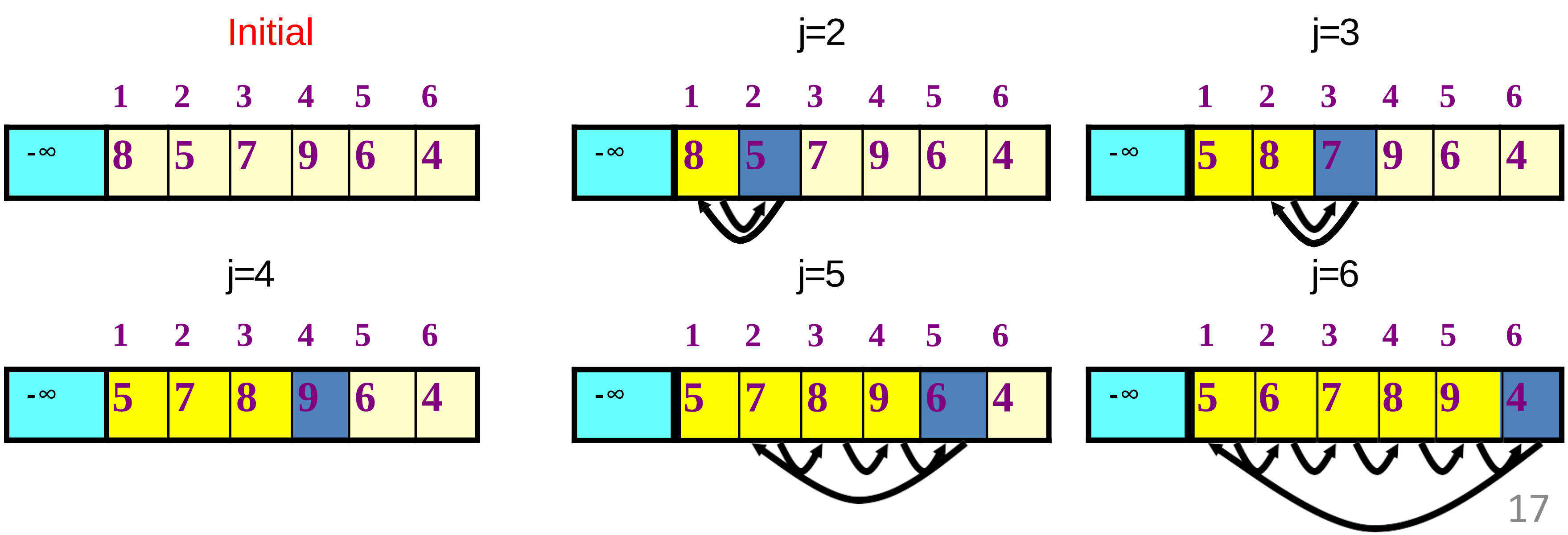
8	2	4	9	3	6
2	8	4	9	3	6
2	4	8	9	3	6
2	4	8	9	3	6
2	3	4	8	9	6

Example of insertion sort

8	2	4	9	3	6	
2	8	4	9	3	6	
2	4	8	9	3	6	
2	4	8	9	3	6	
2	3	4	8	9	6	
2	3	4	6	8	9	*done*

Insertion Sort (another example)

```
INSERTION-SORT (A, n) ⊳        A[1 . . n]
for j ← 2 to n
        do key ← A[j ]
           i ← j − 1
           while i > 0 and A[i] > key
                do A[i + 1] ← A[i]
                   i ← i − 1
           A[i + 1] = key
```



Running time

- The running time depends on the input: an already sorted sequence is easier to sort.

- Parameterize the running time by the size of the input, since short sequences are easier to sort than long ones.

- Generally, we seek upper bounds on the running time, because everybody likes a guarantee.

Θ-notation

Math:

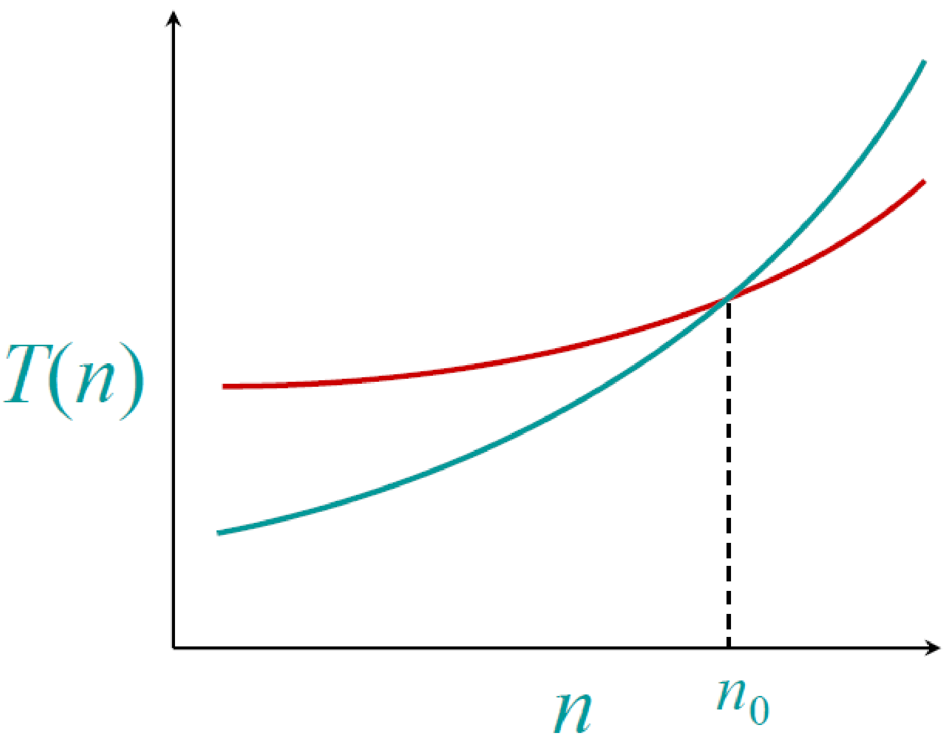
$\Theta(g(n)) = \{ f(n)$: there exist positive constants c_1, c_2, and n_0 such that $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0 \}$

Engineering:

- Drop low-order terms; ignore leading constants.
- Example: $3n^3 + 90n^2 - 5n + 6046 = \Theta(n^3)$

Asymptotic performance

When n gets large enough, a $\Theta(n^2)$ algorithm *always* beats a $\Theta(n^3)$ algorithm.



- We shouldn't ignore asymptotically slower algorithms, however.
- Real-world design situations often call for a careful balancing of engineering objectives.
- Asymptotic analysis is a useful tool to help to structure our thinking.

Analysis of INSERTION-SORT

	cost	times
INSERTION-SORT (A, n) ⊳ A[1 . . n]	cost	times
for j ← 2 to n	c_1	n - 1
do key ← A[j]		
i ← j – 1		
while i > 0 and A[i] > key		
do A[i + 1] ← A[i]		
i ← i – 1		
A[i + 1] = key		

Analysis of INSERTION-SORT

INSERTION-SORT (A, n) ⊳ A[1 . . n]	cost	times
for j ← 2 to n	c_1	n - 1
do key ← A[j]	c_2	n - 1
i ← j − 1		
while i > 0 and A[i] > key		
do A[i + 1] ← A[i]		
i ← i − 1		
A[i + 1] = key		

Analysis of INSERTION-SORT

INSERTION-SORT (A, n) ⊳ A[1 . . n]	cost	times
for j ← 2 to n	c_1	n - 1
do key ← A[j]	c_2	n - 1
i ← j − 1	c_3	n - 1
while i > 0 and A[i] > key		
do A[i + 1] ← A[i]		
i ← i − 1		
A[i + 1] = key		

Analysis of INSERTION-SORT

INSERTION-SORT (A, n) ⊳ A[1 . . n]	cost	times
for j ← 2 to n	c_1	n - 1
do key ← A[j]	c_2	n - 1
i ← j − 1	c_3	n −1
while i > 0 and A[i] > key	c_4	$\sum_{j=2}^{n} t_j$
do A[i + 1] ← A[i]		
i ← i − 1		
A[i + 1] = key		

Analysis of INSERTION-SORT

INSERTION-SORT (A, n) ⊳ A[1 . . n]	cost	times
for j ← 2 to n	c_1	n - 1
do key ← A[j]	c_2	n - 1
i ← j − 1	c_3	n -1
while i > 0 and A[i] > key	c_4	$\sum_{j=2}^{n} t_j$
do A[i + 1] ← A[i]	c_5	$\sum_{j=2}^{n} (t_j - 1)$
i ← i − 1		
A[i + 1] = key		

Analysis of INSERTION-SORT

INSERTION-SORT (A, n) ⊳ A[1 . . n]	cost	times
for j ← 2 to n	c_1	$n - 1$
do key ← A[j]	c_2	$n - 1$
i ← j − 1	c_3	$n - 1$
while i > 0 and A[i] > key	c_4	$\sum_{j=2}^{n} t_j$
do A[i + 1] ← A[i]	c_5	$\sum_{j=2}^{n} (t_j - 1)$
i ← i − 1	c_6	$\sum_{j=2}^{n} (t_j - 1)$
A[i + 1] = key		

Analysis of INSERTION-SORT

INSERTION-SORT (A, n) ⊳ A[1 . . n]	cost	times
for j ← 2 to n	c_1	n - 1
do key ← A[j]	c_2	n - 1
i ← j − 1	c_3	n - 1
while i > 0 and A[i] > key	c_4	$\sum_{j=2}^{n} t_j$
do A[i + 1] ← A[i]	c_5	$\sum_{j=2}^{n}(t_j - 1)$
i ← i − 1	c_6	$\sum_{j=2}^{n}(t_j - 1)$
A[i + 1] = key	c_7	n - 1

Analysis of INSERTION-SORT

INSERTION-SORT (A, n) ⊳ A[1 . . n]	cost	times
for j ← 2 to n	c_1	$n - 1$
do key ← A[j]	c_2	$n-1$
i ← j − 1	c_3	n-1
while i > 0 and A[i] > key	c_4	$\sum_{j=2}^{n} t_j$
do A[i + 1] ← A[i]	c_5	$\sum_{j=2}^{n} (t_j - 1)$
i ← i − 1	c_6	$\sum_{j=2}^{n} (t_j - 1)$
A[i + 1] = key	c_7	$n - 1$

Let $T(n)$ = running time of **INSERTION-SORT**.

$$T(n) = c_1 (n-1) + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7(n-1)$$

INSERTION-SORT (A, n) ⊳ A[1 . . n]	cost	times
for j ← 2 to n	c_1	n-1
do key ← A[j]	c_2	$n-1$
i ← j − 1	c_3	$n-1$
while i > 0 and A[i] > key	c_4	$\sum_{j=2}^{n} t_j$
do A[i + 1] ← A[i]	c_5	$\sum_{j=2}^{n} (t_j - 1)$
i ← i − 1	c_6	$\sum_{j=2}^{n} (t_j - 1)$
A[i + 1] = key	c_7	$n-1$

$$T(n) = c_1 (n-1) + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7(n-1)$$

Best-case: The array is already sorted.

- Always find that **A[i] ≤ key** upon the first time the while loop test is run (when i = j −1).
- All t_j are 1.
- Running time is

$$T(n) = c_1 (n-1) + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_7(n-1)$$
$$=(c_1+c_2+c_3+c_4+c_7)n - (c_1+c_2+c_3+c_4+c_7)$$

- Can express $T(n)$ as $an+b$ for constants a and b (that depend on the statement costs c_i) $\Rightarrow T(n)$ is a linear function of n. $\Rightarrow T(n)=\Theta(n)$

INSERTION-SORT (A, n) ▷ A[1 . . n]	cost	times
for j ← 2 to n	c_1	n-1
do key ← A[j]	c_2	n-1
i ← j − 1	c_3	n-1
while i > 0 and A[i] > key	c_4	$\sum_{j=2}^{n} t_j$
do A[i + 1] ← A[i]	c_5	$\sum_{j=2}^{n} (t_j - 1)$
i ← i − 1	c_6	$\sum_{j=2}^{n} (t_j - 1)$
A[i + 1] = key	c_7	n −1

$$T(n) = c_1(n\text{-}1) + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7(n-1)$$

Worst-case: The array is in reverse sorted order.

- Always find that ***A*[*i*] > *key*** in while loop test.

INSERTION-SORT (A, n) ⊳ A[1 . . n]	cost	times
for j ← 2 to n	c_1	n-1
do key ← A[j]	c_2	n-1
i ← j − 1	c_3	n-1
while i > 0 and A[i] > key	c_4	$\sum_{j=2}^{n} t_j$
do A[i + 1] ← A[i]	c_5	$\sum_{j=2}^{n} (t_j - 1)$
i ← i − 1	c_6	$\sum_{j=2}^{n} (t_j - 1)$
A[i + 1] = key	c_7	n −1

$$T(n) = c_1(n-1) + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7(n-1)$$

Worst-case: The array is in reverse sorted order.

- Have to compare ***key*** with all elements to the left of the *j* th position ⇒ compare with *j* − *1* elements.

INSERTION-SORT (A, n) ⊳ A[1 . . n]	cost	times
for j ← 2 to n	c_1	n-1
do key ← A[j]	c_2	n-1
i ← j − 1	c_3	n-1
while i > 0 and A[i] > key	c_4	$\sum_{j=2}^{n} t_j$
do A[i + 1] ← A[i]	c_5	$\sum_{j=2}^{n} (t_j - 1)$
i ← i − 1	c_6	$\sum_{j=2}^{n} (t_j - 1)$
A[i + 1] = key	c_7	$n - 1$

$$T(n) = c_1(n-1) + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7(n-1)$$

Worst-case: The array is in reverse sorted order.

- Since the while loop exits because i reaches 0, there's one additional test after the *j* − *1* tests ⇒ $t_j = j$.

INSERTION-SORT (A, n) ⊳ A[1 . . n]	cost	times
for j ← 2 to n	c_1	n-1
do key ← A[j]	c_2	n-1
i ← j − 1	c_3	n-1
while i > 0 and A[i] > key	c_4	$\sum_{j=2}^{n} t_j$
do A[i + 1] ← A[i]	c_5	$\sum_{j=2}^{n} (t_j - 1)$
i ← i − 1	c_6	$\sum_{j=2}^{n} (t_j - 1)$
A[i + 1] = key	c_7	n −1

$T(n)$ = c_1(n-1) + $c_2(n-1)$ + $c_3(n-1)$ + $c_4 \sum_{j=2}^{n} t_j$ + $c_5 \sum_{j=2}^{n} (t_j - 1)$ +
$c_6 \sum_{j=2}^{n} (t_j - 1)$ + $c_7(n-1)$

Worst-case: The array is in reverse sorted order.

- $\sum_{j=2}^{n}(t_j - 1) = \sum_{j=2}^{n}(j - 1)$

INSERTION-SORT (A, n) ⊳ A[1 . . n]	cost	times
for j ← 2 to n	c_1	n-1
do key ← A[j]	c_2	n-1
i ← j − 1	c_3	n-1
while i > 0 and A[i] > key	c_4	$\sum_{j=2}^{n} t_j$
do A[i + 1] ← A[i]	c_5	$\sum_{j=2}^{n} (t_j - 1)$
i ← i − 1	c_6	$\sum_{j=2}^{n} (t_j - 1)$
A[i + 1] = key	c_7	$n - 1$

$$T(n) = c_1(n-1) + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7(n-1)$$

Worst-case: The array is in reverse sorted order. Running time:

- $T(n) = c_1(n-1) + c_2(n-1) + c_3(n-1) + c_4\left(\frac{n(n+1)}{2} - 1\right) + c_5\left(\frac{n(n-1)}{2} - 1\right) + c_6\left(\frac{n(n-1)}{2} - 1\right) + c_7(n-1)$

$$= \underbrace{\boxed{\left(\frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2}\right)}}_{a} n^2 + \underbrace{\boxed{\left(c_1 + c_2 + c_3 - \left(\frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2}\right) + c_7\right)}}_{b} n \underbrace{\boxed{-(c_1 + c_3 + c_4 + c_7)}}_{c}$$

INSERTION-SORT (A, n) ⊳ A[1 . . n]	cost	times
for j ← 2 to n	c_1	n-1
do key ← A[j]	c_2	n-1
i ← j − 1	c_3	n-1
while i > 0 and A[i] > key	c_4	$\sum_{j=2}^{n} t_j$
do A[i + 1] ← A[i]	c_5	$\sum_{j=2}^{n} (t_j - 1)$
i ← i − 1	c_6	$\sum_{j=2}^{n} (t_j - 1)$
A[i + 1] = key	c_7	n −1

$$T(n) = c_1(n-1) + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7(n-1)$$

Worst-case: The array is in reverse sorted order.

- *Can express* $T(n)$ *as* $an^2 + bn + c$ *for constants* a,b, c
- $T(n)$ *is a quadratic function of* n. $\Rightarrow T(n) = \Theta(n^2)$

Order of Growth

We will only consider order of growth of running time:

- We can ignore the lower-order terms, since they are relatively insignificant for very large n.
- We can also ignore leading term's constant coefficients, since they are not as important for the rate of growth in computational efficiency for very large n.
- For the insertion-sort algorithm, we just said that best case was linear to *n* and worst/average case quadratic to *n*.

Designing Algorithms

- **We discussed insertion sort**

  - Can we design better than n^2 sorting algorithms?

  - We will do so using one of the most powerful algorithm design techniques.

Divide-and-Conquer

- **To solve problem P:**
  - Divide P into smaller problems $P_1, P_2, \ldots, P_k$.
  - Conquer by solving the (smaller) subproblems recursively.
  - Combine the solutions to $P_1, P_2, \ldots, P_k$ into the solution for P.

Merge-Sort Algorithm

- Using divide-and-conquer, we can obtain the ***Merge-Sort*** algorithm

  - Divide: Divide the n elements into two subsequences of n/2 elements each.

  - Conquer: Sort the two subsequences recursively.

  - Combine: Merge the two sorted subsequences to produce the sorted answer.

Merge-Sort (A, p, r)

- **INPUT: a sequence of n numbers stored in array A**
- **OUTPUT: an ordered sequence of n numbers**

```
MERGE-SORT (A, p, r)
1   if p < r
2       then q ← ⌊(p + r)/2⌋
3           MERGE-SORT(A, p, q)
4           MERGE-SORT(A, q + 1, r)
5           MERGE(A, p, q, r)
```

Merge (A, p, q, r)

```
MERGE (A, p, q, r)
    n1 ← q – p + 1
    n2 ← r – q
    create arrays L[1..n1 + 1] and R[1..n2 + 1]
    for i ← 1 to n1
          do L[i] ← A[p + i - 1]
    for j ← 1 to n2
          do R[j] ← A[q + j]
    L[n1 + 1] ← ∞
    L[n2 + 1] ← ∞
    i ← 1
    j ← 1
    for k ← p to r
          do if L[i] ≤ R[j]
                then A[k] ← L[i]
                     i ← i + 1
                else A[k] ← R[j]
                     j ← j + 1
```

20	12
13	11
7	9
2	1

Merging two sorted arrays

20	12
13	11
7	9
2	1

Merging two sorted arrays

20	12
13	11
7	9
2	1

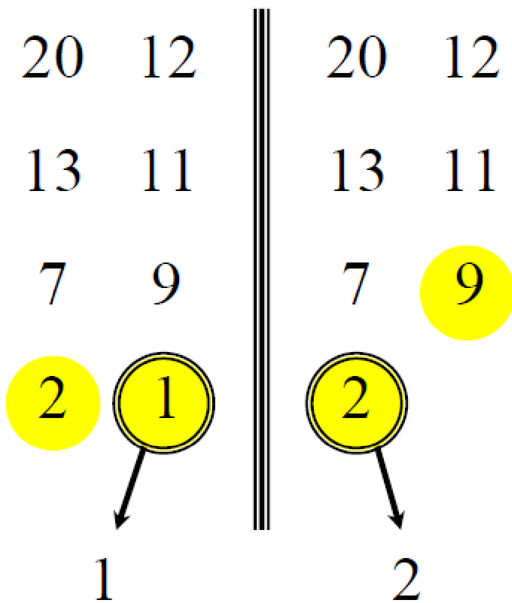
1

Merging two sorted arrays

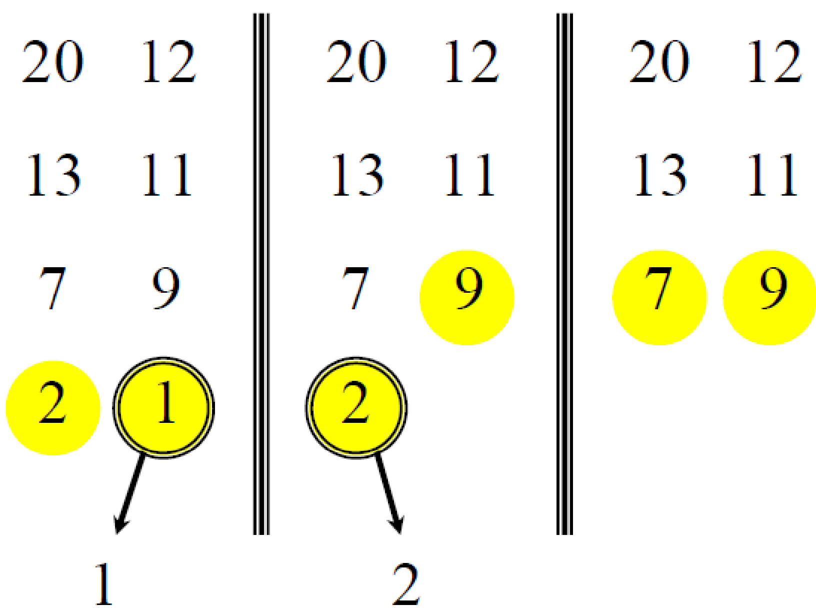
20	12	20	12
13	11	13	11
7	9	7	9
2	1	2	

1

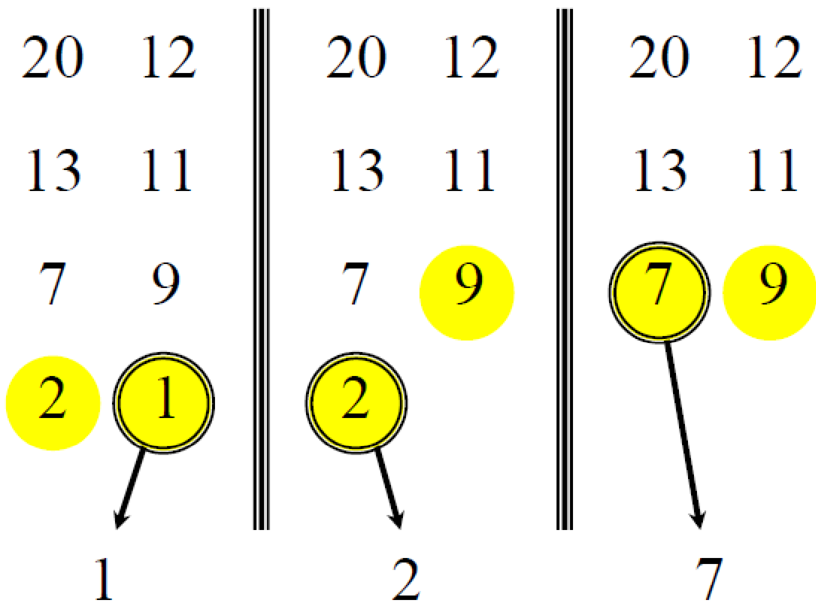
Merging two sorted arrays



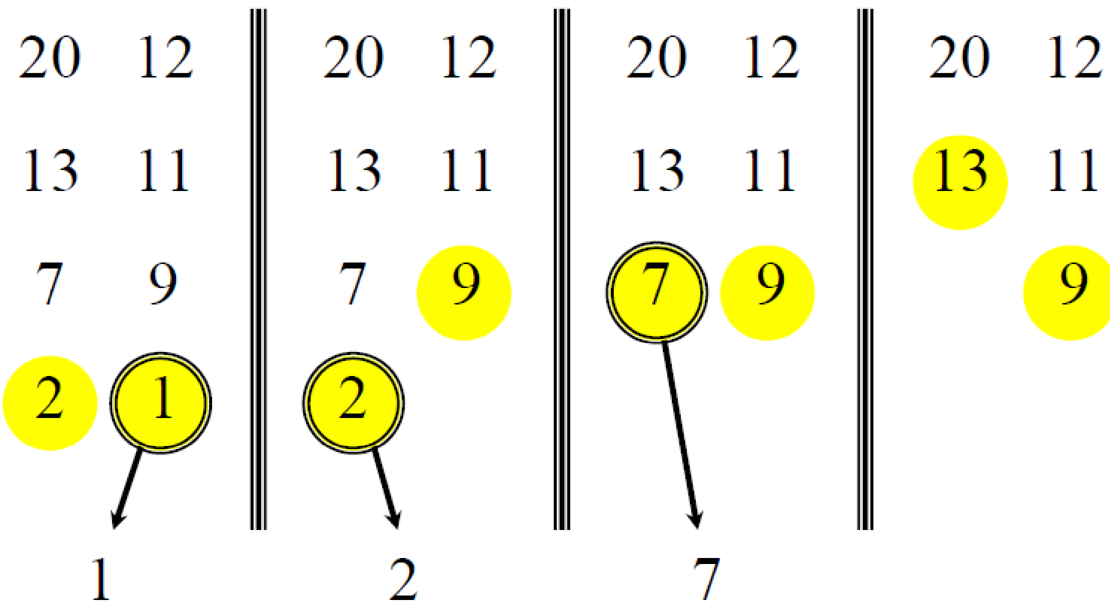
Merging two sorted arrays



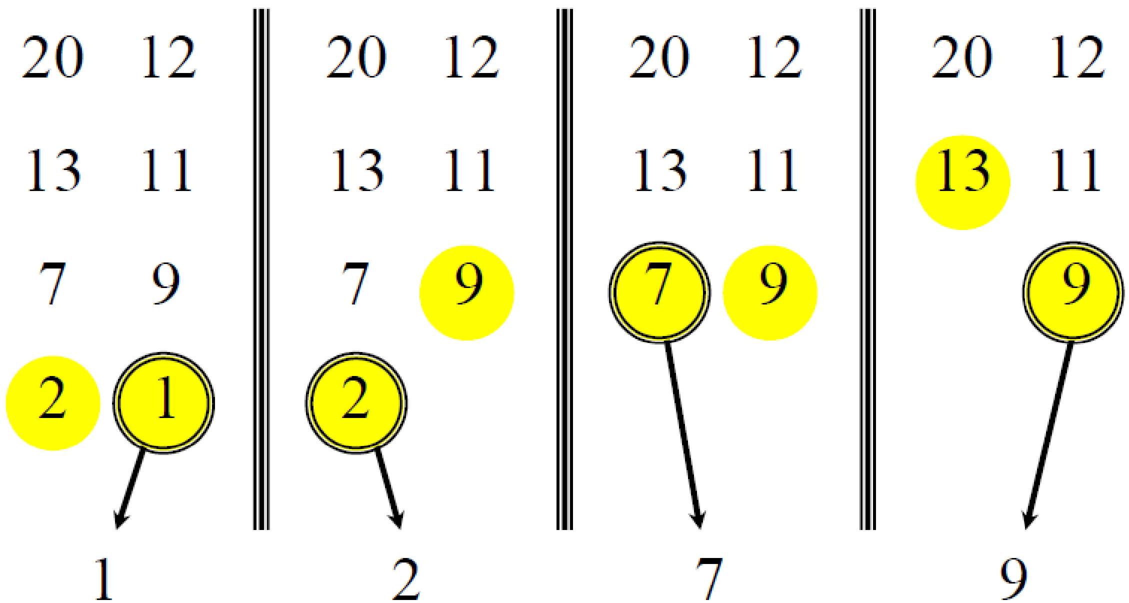
Merging two sorted arrays



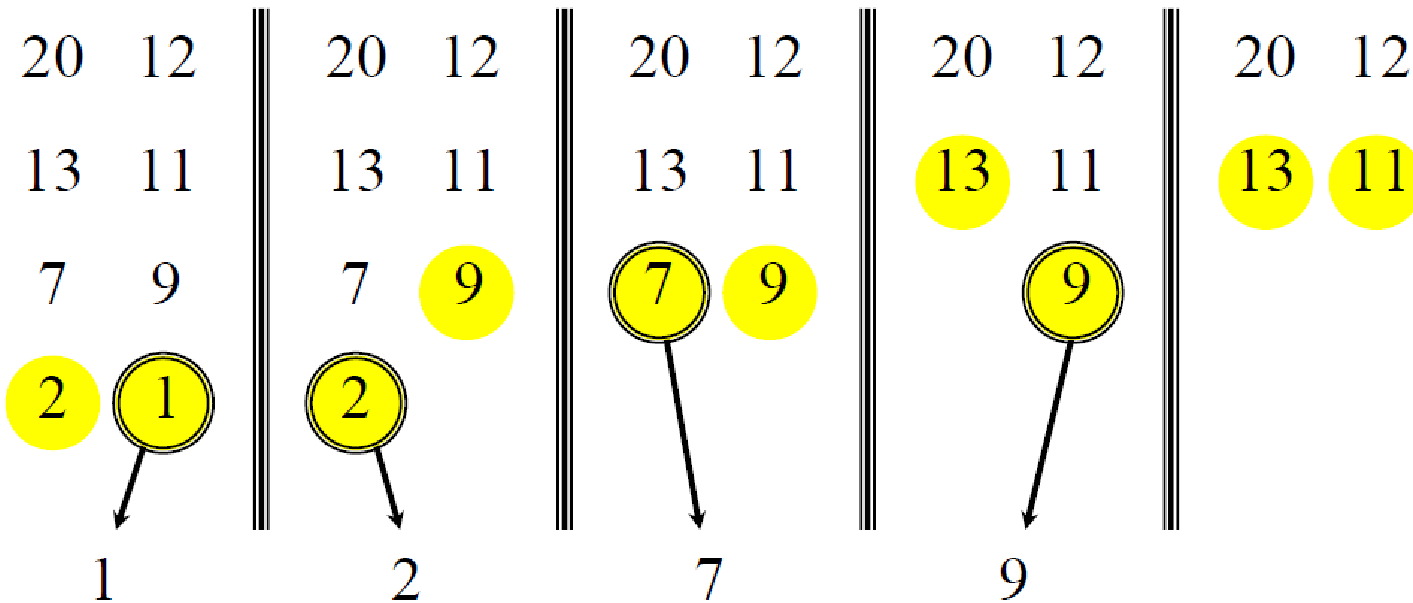
Merging two sorted arrays



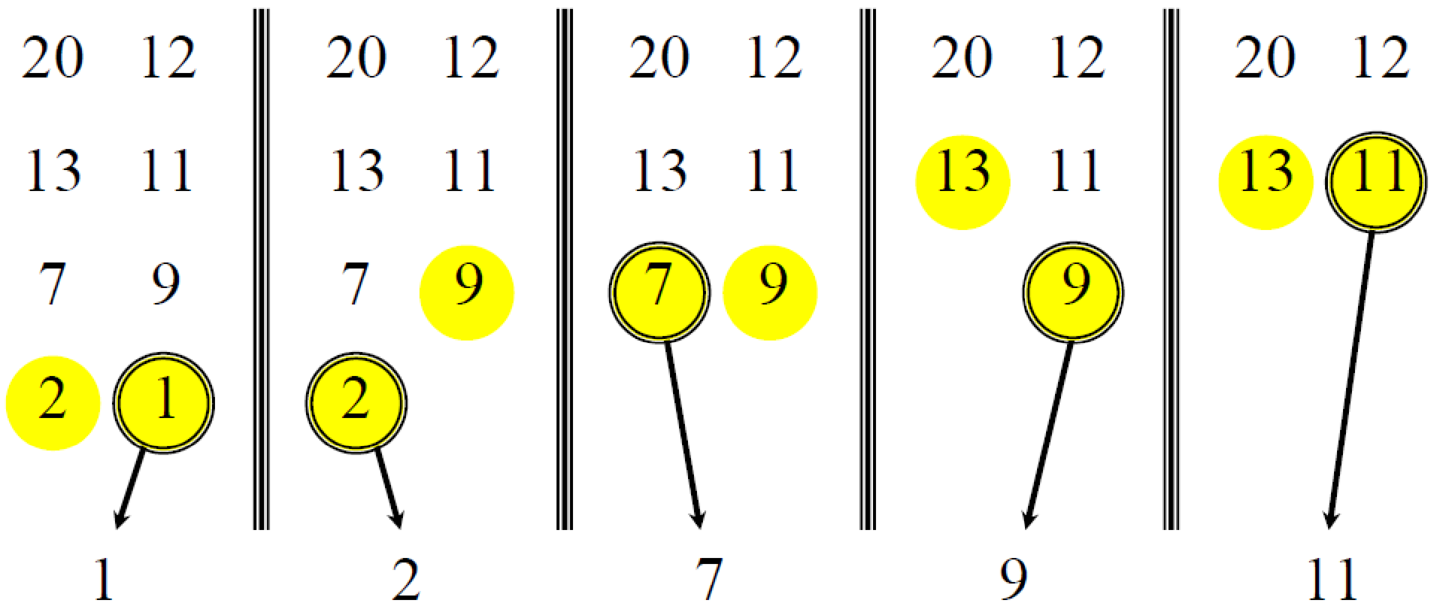
Merging two sorted arrays



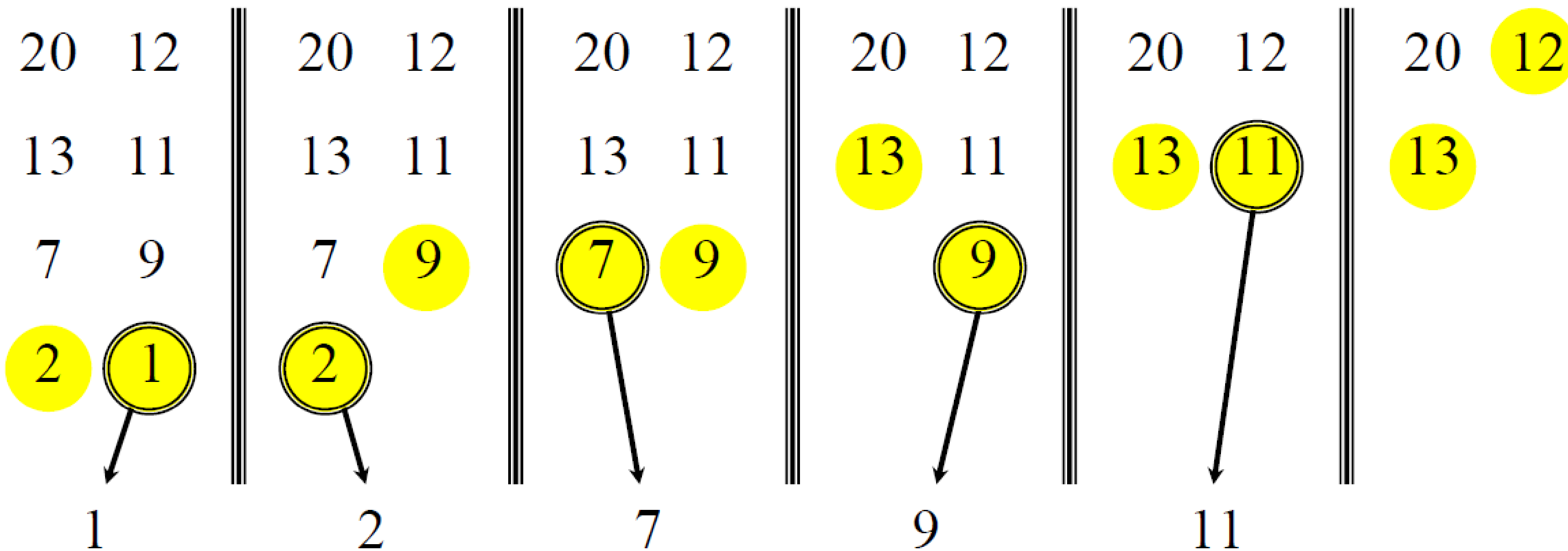
Merging two sorted arrays



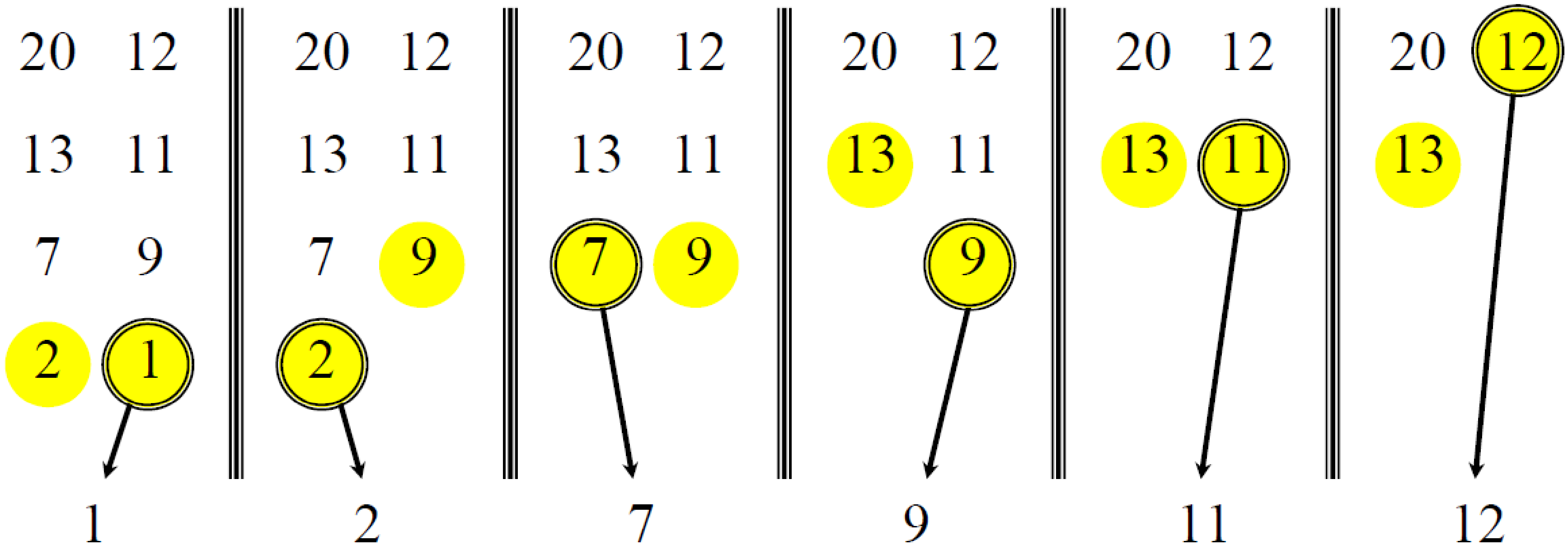
Merging two sorted arrays



Merging two sorted arrays

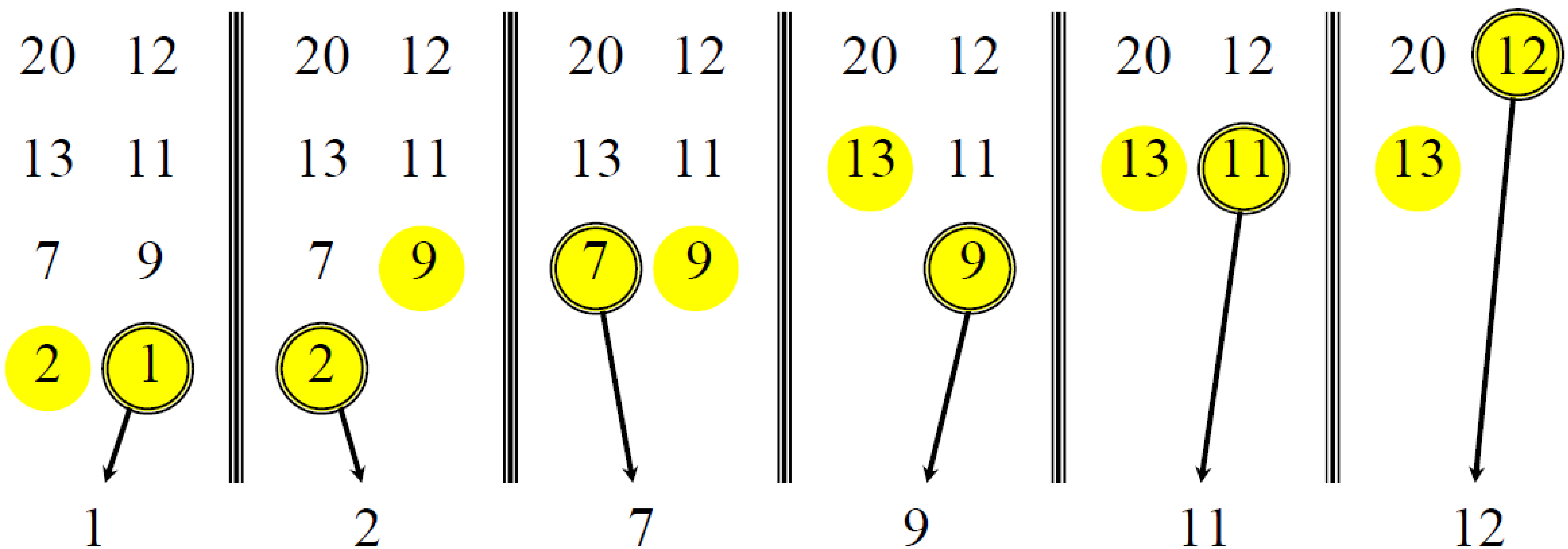


Merging two sorted arrays



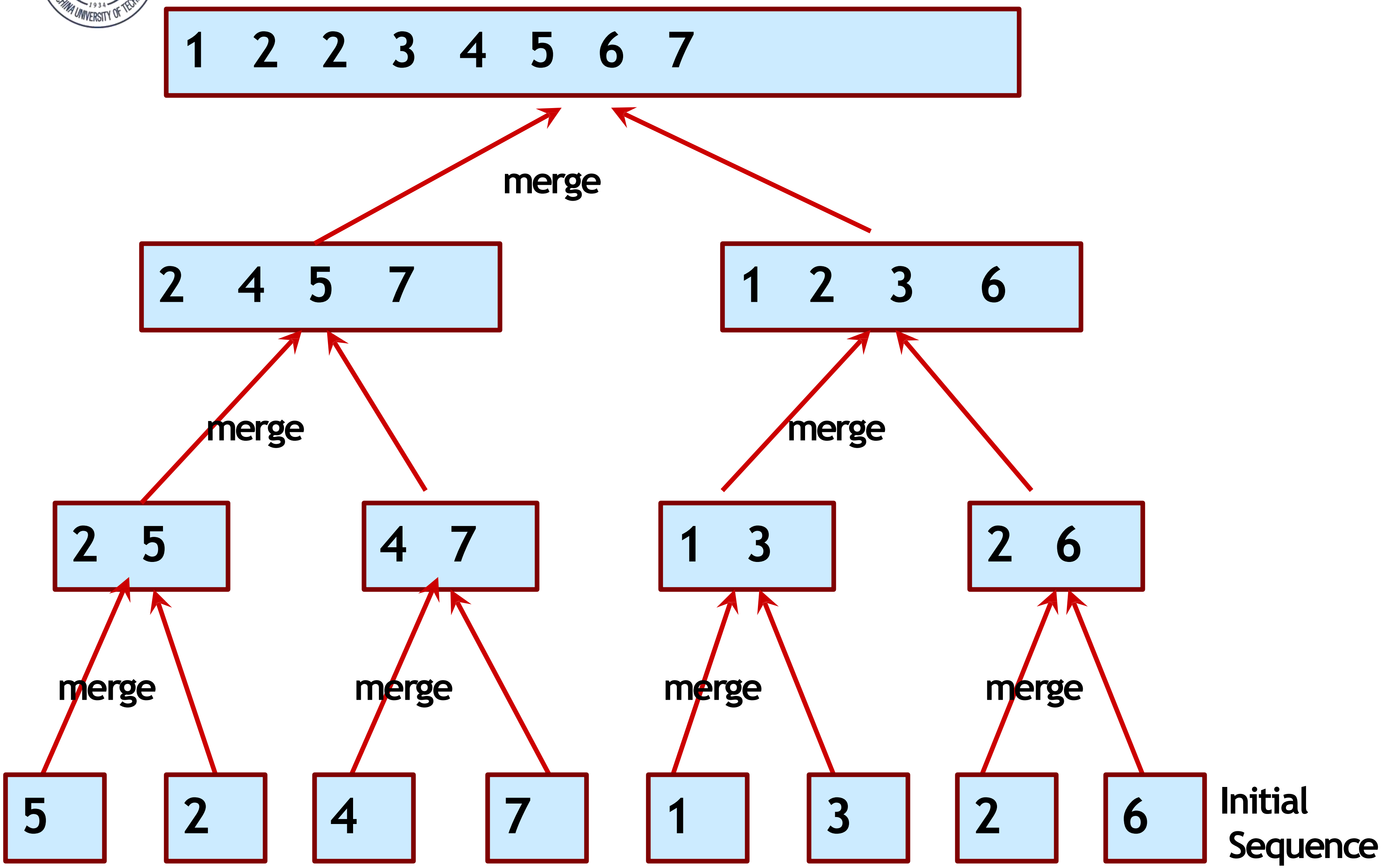
Time?

Merging two sorted arrays



Time = $\Theta(n)$ to merge a total of n elements (linear time).

Action of Merge Sort



Analyzing Merge-Sort

- **How long does merge-sort take?**

  -- Bottleneck = merging (and copying).

  >> merging two files of size *n/2* requires *n* comparisons

  -- *T(n)* = comparisons to merge sort *n* elements.

  >>to make analysis cleaner, assume n is a power of 2

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ \underbrace{2T(n/2)}_{\text{Sorting both halves}} + \underbrace{\Theta(n)}_{\text{merging}} & \text{otherwise} \end{cases}$$

- **Claim. $T(n) = n \log_2 n$**

  -- Note: same number of comparisons for ANY file.

  >> even already sorted

Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

$$T(n)$$

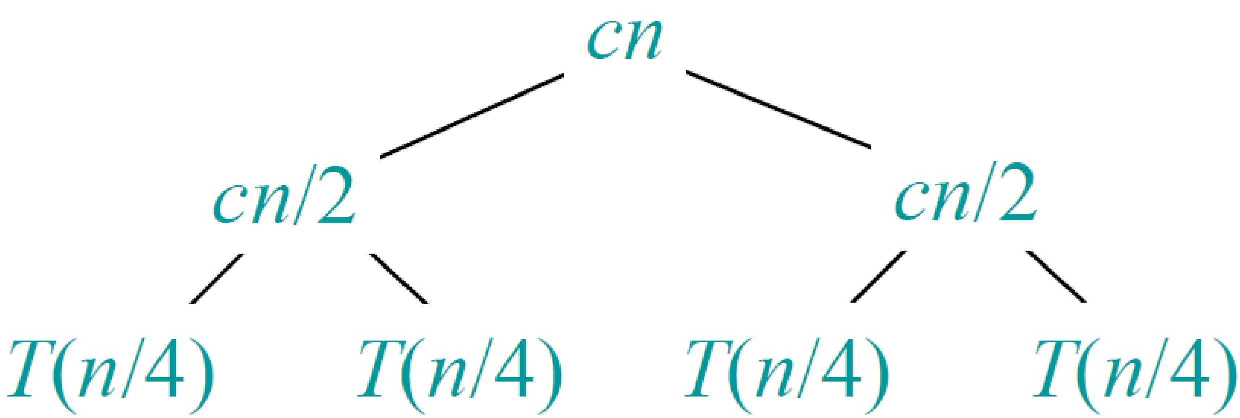
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

$$
\begin{array}{ccc}
 & cn & \\
\swarrow & & \searrow \\
T(n/2) & & T(n/2)
\end{array}
$$

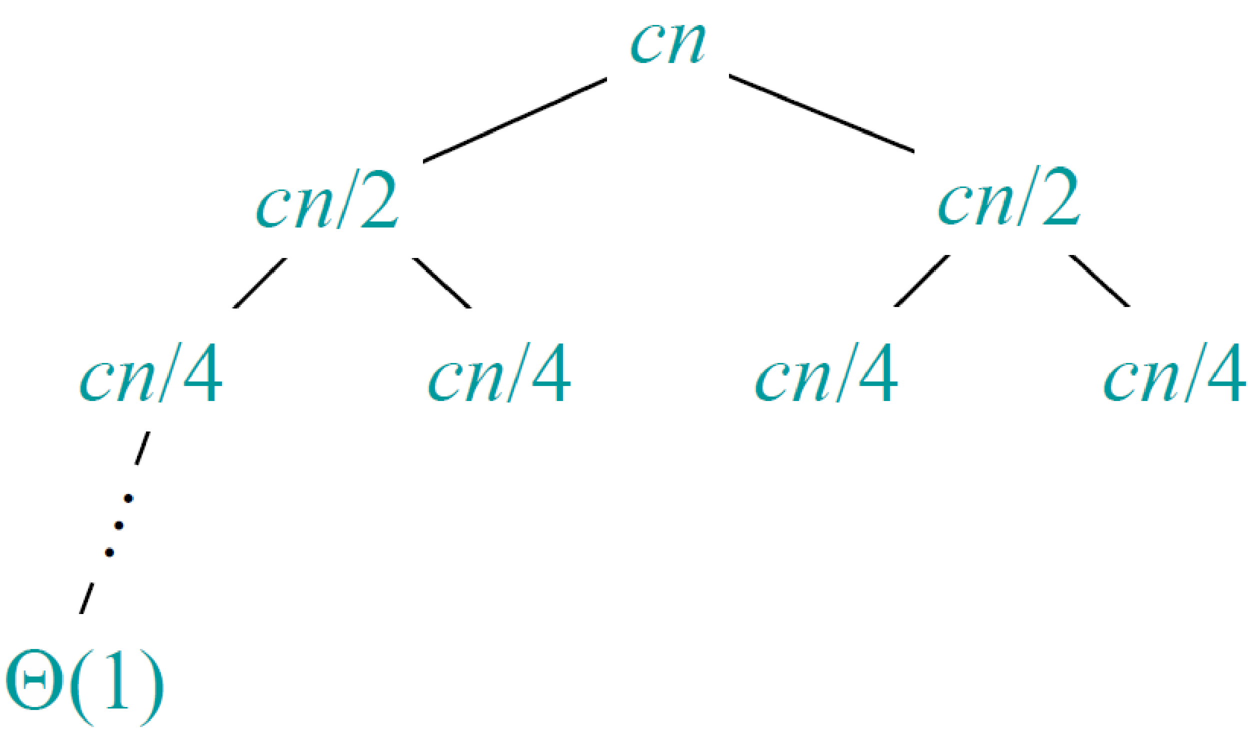
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



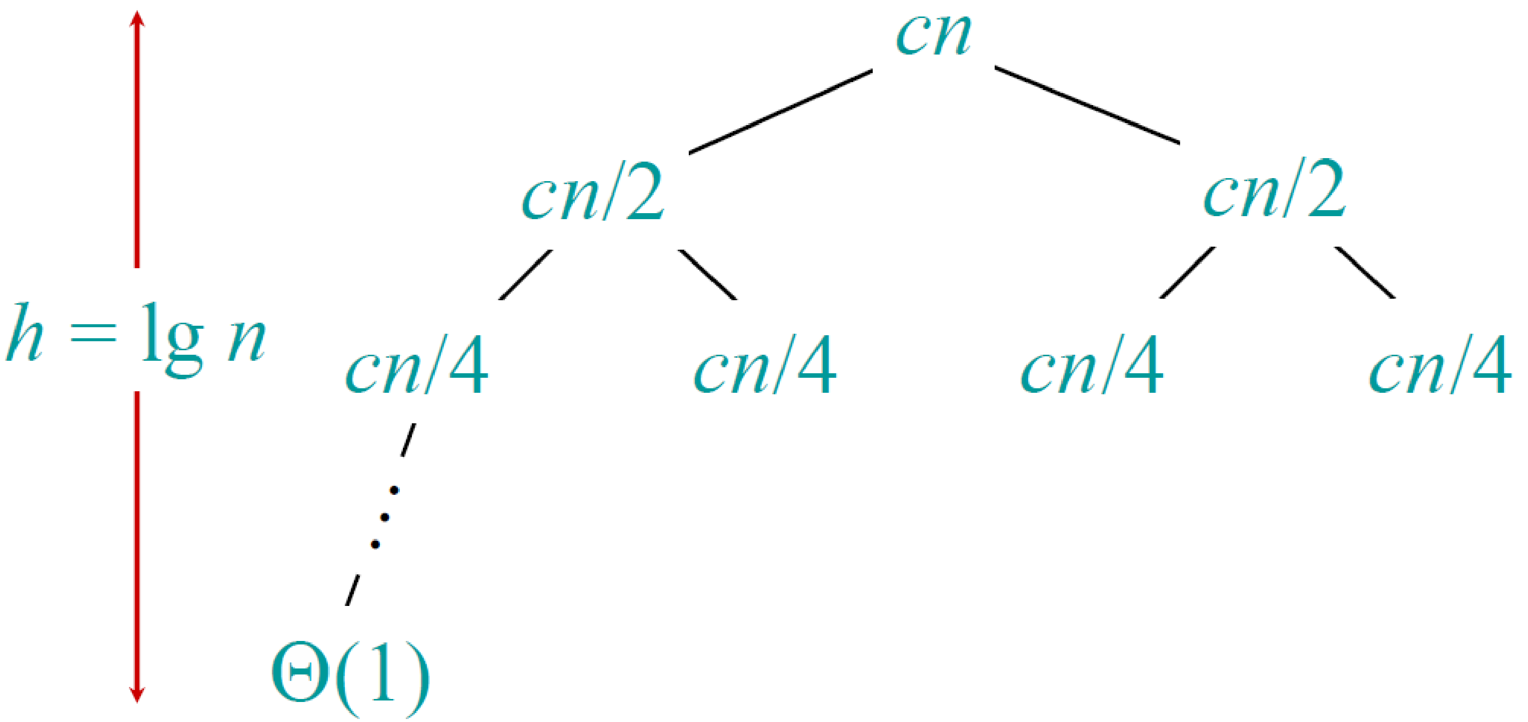
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



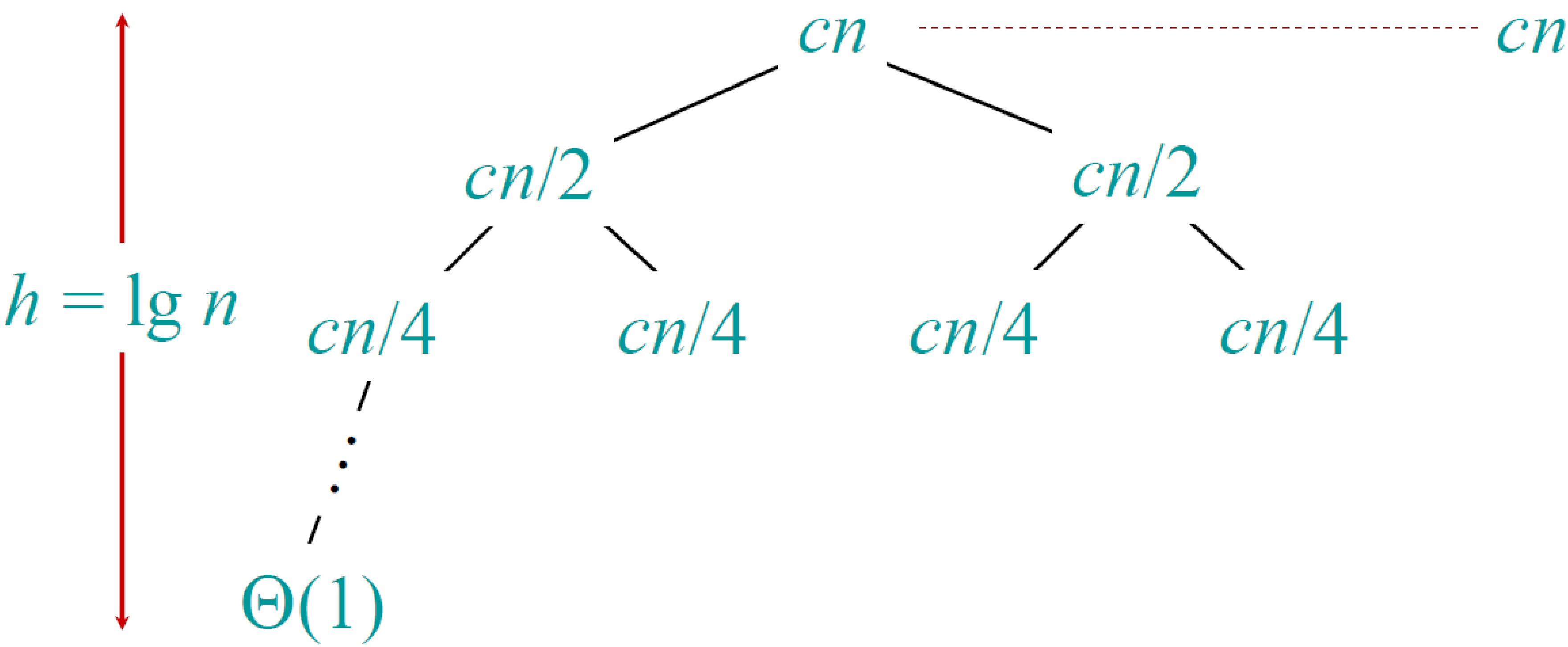
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



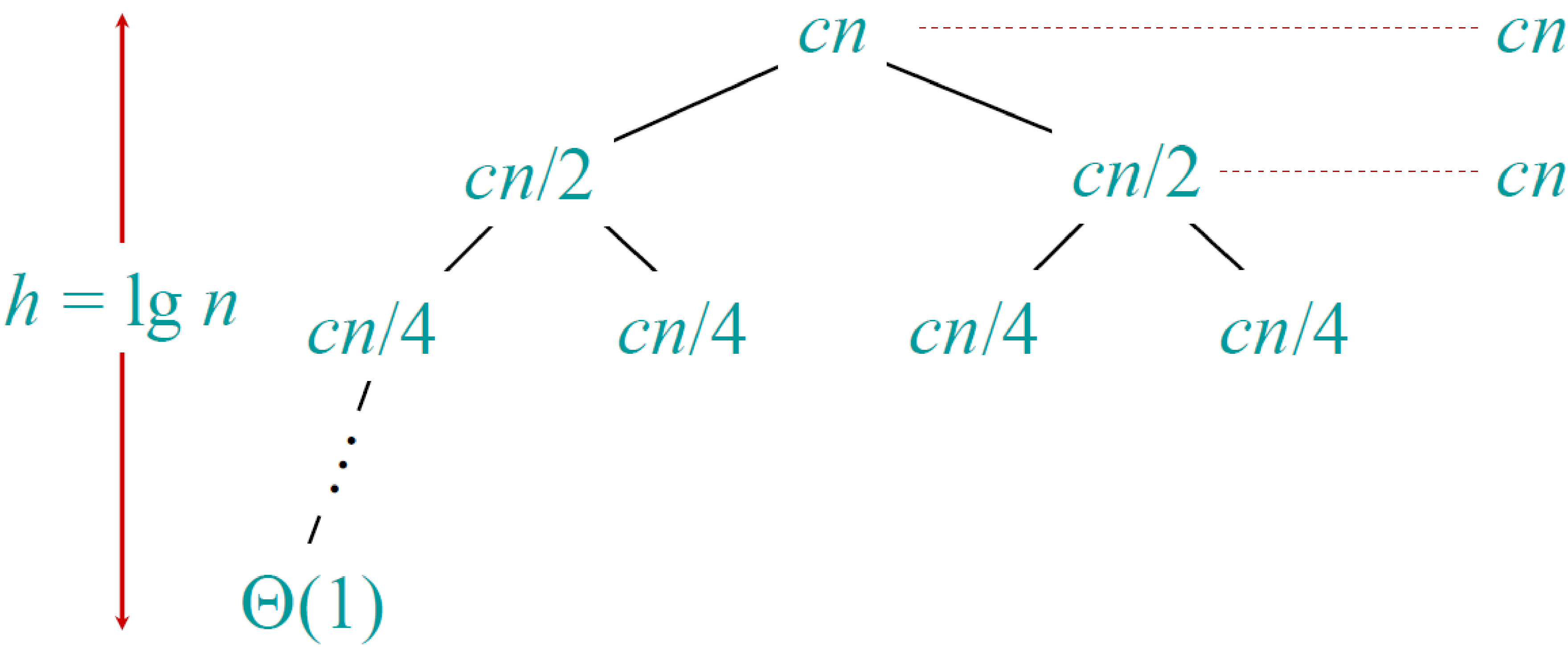
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



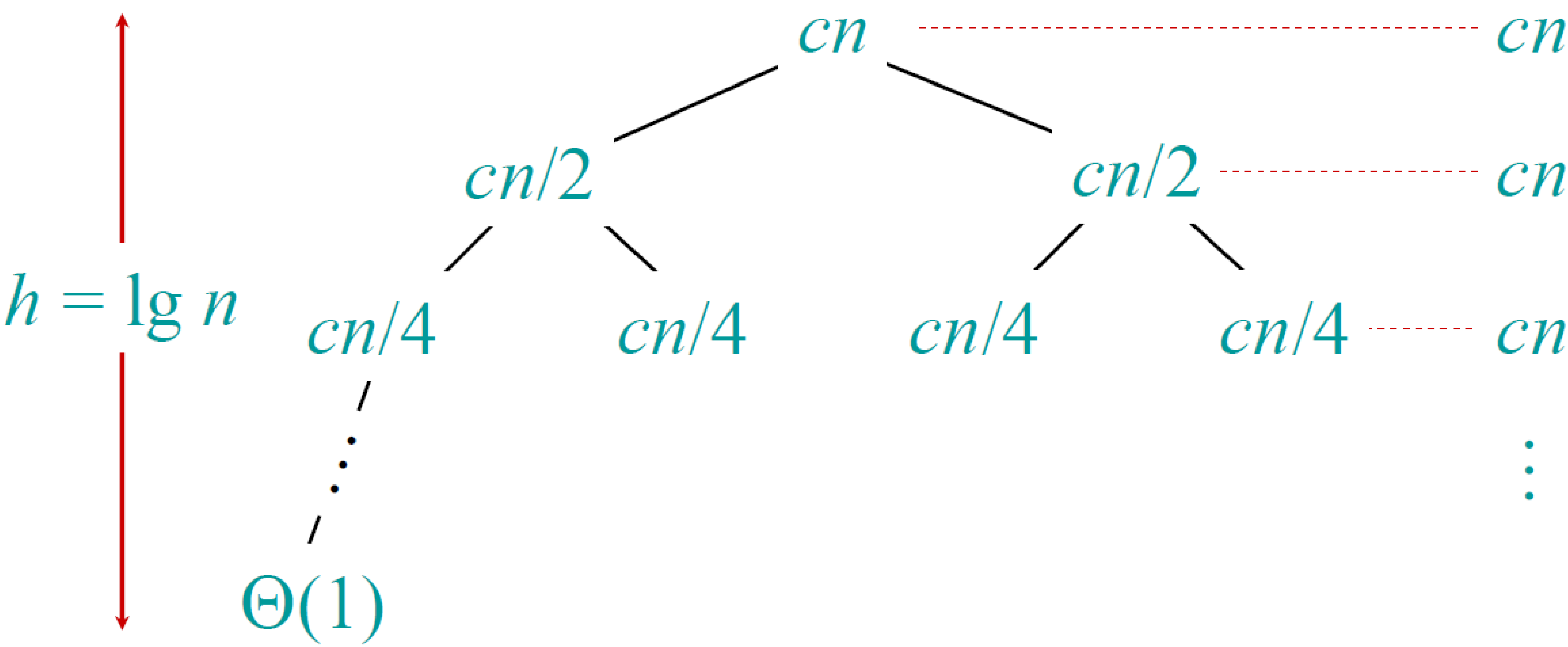
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



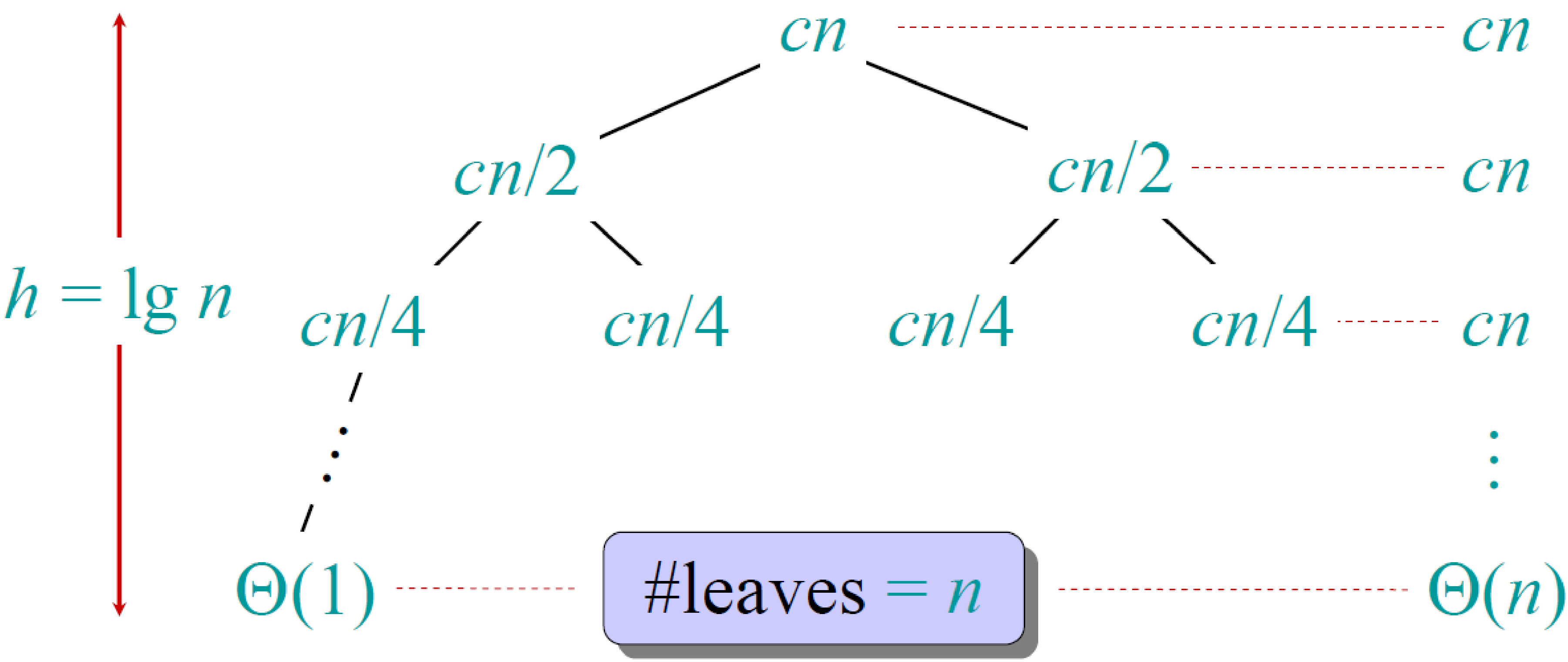
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



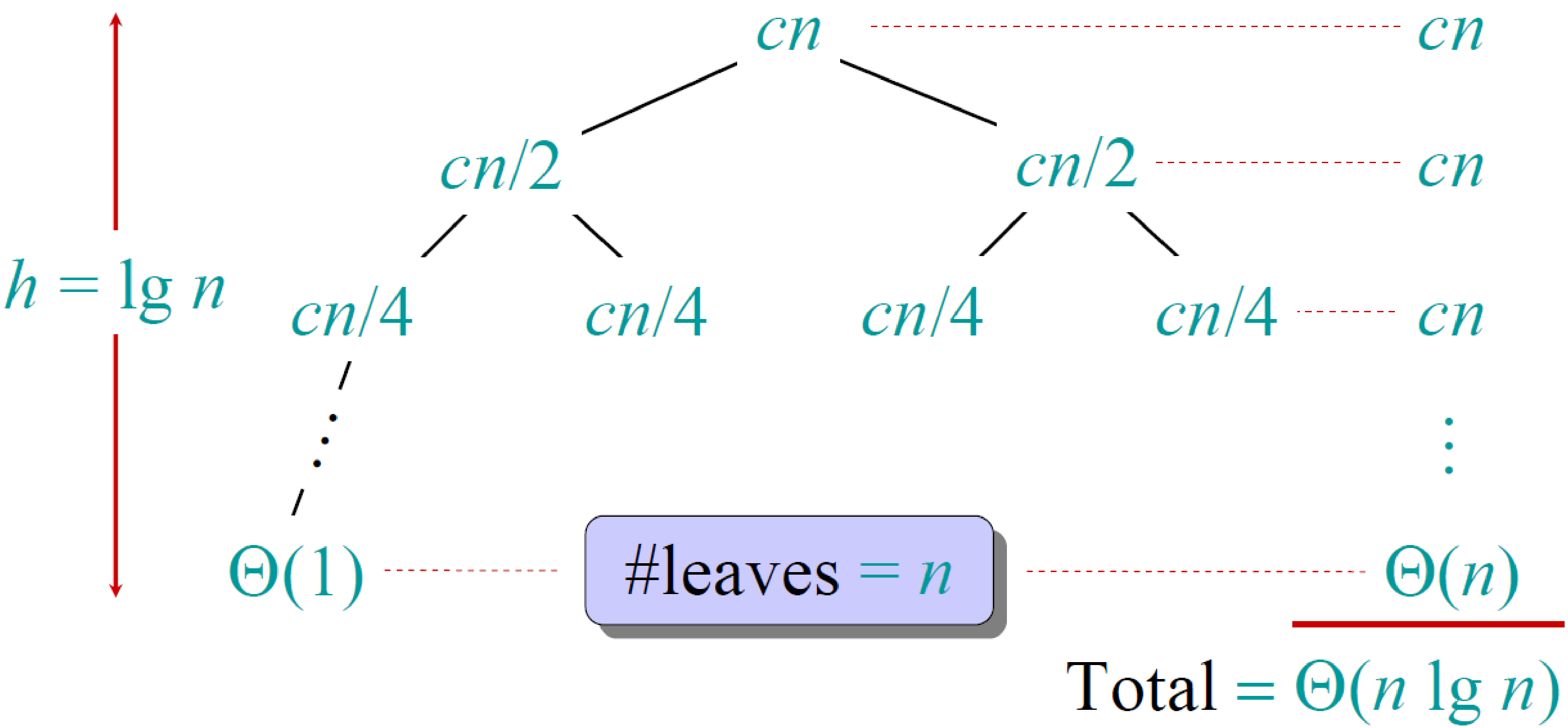
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



Conclusions

- $\Theta(nlgn)$ grows more slowly than $\Theta(n^2)$.
- Therefore, merge-sort asymptotically beats insertion-sort in the worst case.
- In practice, merge-sort beats insertion-sort for ***n > 30*** .