COMP 446 / 546 ALGORITHM DESIGN AND ANALYSIS

LECTURE 2 SORTING

ALPTEKİN KÜPÇÜ

Based on slides of Serdar Taşıran, Shafi Goldwasser, and Erik Demaine

SORTING

- **Sorting**
  - **Input:** a sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$
  - **Output:** a re-ordering $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the sequence such that $a'_1 \le a'_2 \le \ldots \le a'_n$
- **Example:**
  - Input: 8 2 4 9 3 6
  - Output: 2 3 4 6 8 9
  - Called an instance of the problem
- **Check these demos:**
- http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/sortcontest/sortcontest.htm
- http://www.cs.oswego.edu/~mohammad/classes/csc241/samples/sort/Sort2-E.html
- http://www.cs.ubc.ca/~harrison/Java/sorting-demo.html
- http://www.cs.pitt.edu/~kirk/cs1501/animations/Sort3.html

INSERTION SORT

- **Takes array A[1..n] containing a sequence of length n to be sorted**
- **Sorts the array in place**
  - Numbers re-arranged inside A with at most a constant number of them stored outside.
  - O(1) extra space

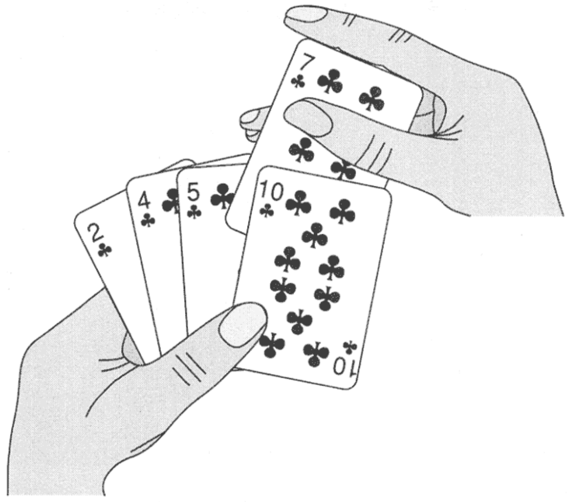
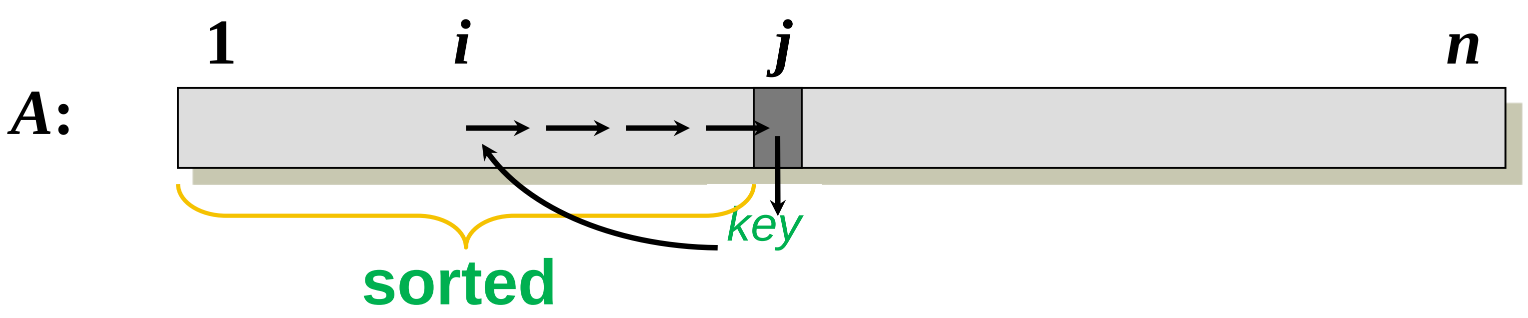


Figure 2.1 Sorting a hand of cards using insertion sort.

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

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*

CORRECTNESS OF INSERTION SORT

```
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
```

- **Loop invariant:**
  - At the start of each iteration, the subarray A[1..j-1] contains the elements originally in A[1..j-1] but in sorted (increasing) order
- **Prove initialization, maintenance, termination**
- **Invariant must imply interesting property about algorithm**

RUNNING TIME

- The running time of insertion sort depends on the input: e.g., an already sorted sequence is easier to sort.
- Major Simplifying Convention: Parameterize the running time by the size of the input, since short sequences are easier to sort than long ones.
  - $T_A(n)$ = time of A when run with inputs of length n
    - n: Number of bits required to encode input
  - Ignore machine-dependent constants
  - Look at *growth* of $T(n)$ as $n \to \infty$
    - For small inputs, the algorithm will run fast anyway. Important thing is for how big inputs we can still run the algorithm given a reasonable amount of time.

KINDS OF ANALYSES

- **Worst-case: (mostly-used)**
  - **$T(n)$ = maximum time of algorithm on any input of size n.**
  - **Generally, we seek upper bounds on the running time, to have a guarantee of performance.**
- **Average-case: (sometimes used)**
  - **$T(n)$ = expected time of algorithm over all inputs of size n.**
  - **Need assumption of statistical distribution of real inputs.**
- **Best-case: (almost never used)**
  - **$T(n)$ = best possible time of algorithm over any input of size n.**
  - **Cheat with a slow algorithm that works fast on *some* input.**
  - **May be useful when proving negative results**
    - **e.g., Even the best case of algorithm X is slower than the worst case of algorithm Y.**

RUNNING-TIME OF INSERTION SORT

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

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

$$T(n) = c_1 n + c_2 (n-1) + c_3 (n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_8 (n-1)$$

The running time depends on the values of t_j . This vary according to the input.

INSERTION SORT ANALYSIS

Worst case: **Input reverse sorted.**

$$T(n) = \sum_{j=2}^{n} \Theta(j) = \Theta(n^2)$$ **[arithmetic series]**

Average case: **All permutations equally likely.**

$$T(n) = \sum_{j=2}^{n} \Theta(j/2) = \Theta(n^2)$$

***Best case*:** **Input already sorted. O(n)**

Is insertion sort a fast sorting algorithm?

- **Moderately so, for small *n*.**
- **Not at all, for large *n*.**

HOW TO DESIGN ALGORITHMS?

- **We'll see many example styles throughout the semester**
- **Insertion sort was an example of an "incremental algorithm"**
- **Another common paradigm: Divide and Conquer**
  - Divide into simpler/smaller sub-problems
  - Solve (conquer) sub-problems recursively
  - Combine results of sub-problems

MERGE SORT

MERGE-SORT $A[1 \,..\, n]$

If $n = 1$, return

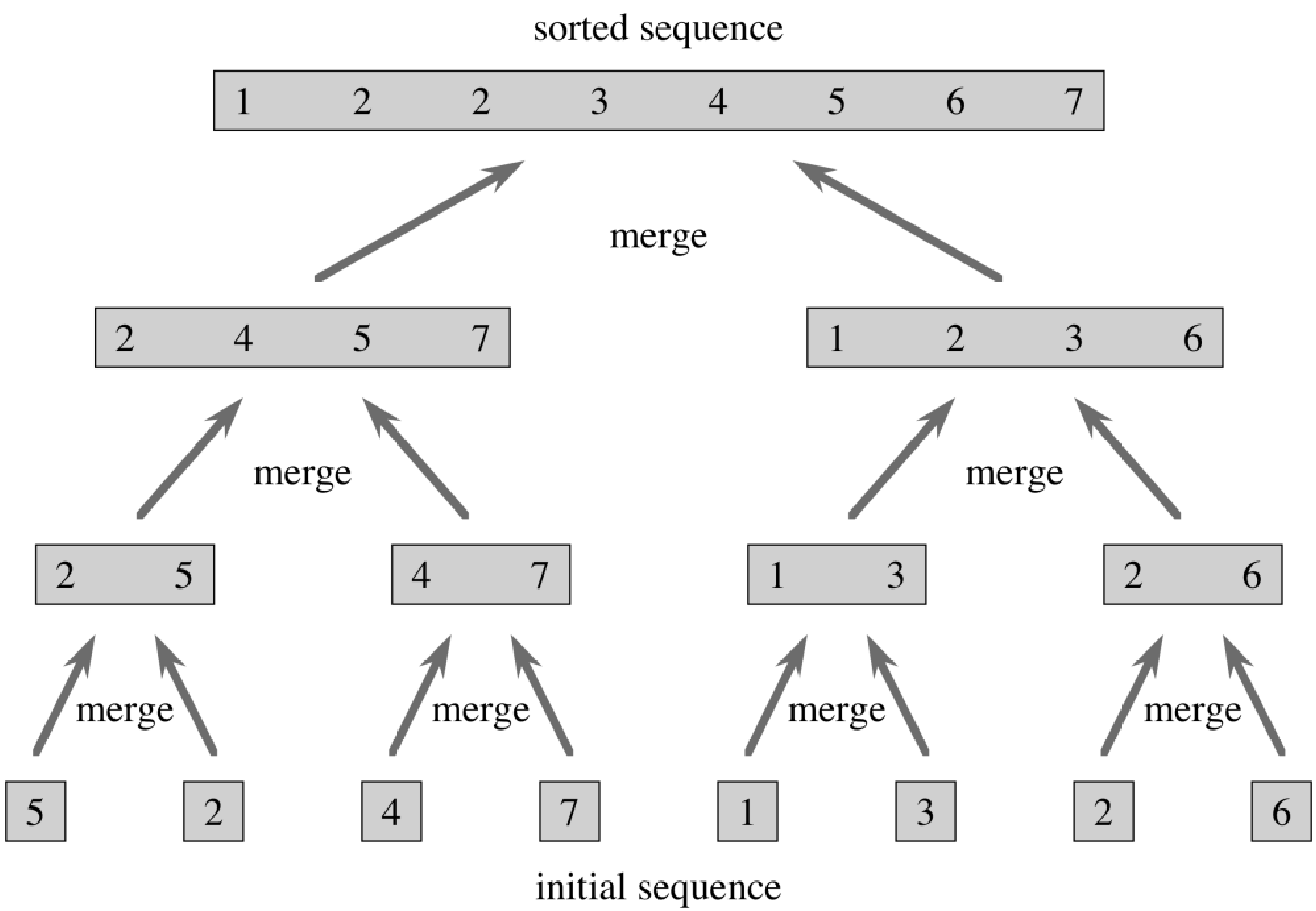
Recursively sort

$A[\,1 \,..\, \lceil n/2 \rceil\,]$ and $A[\,\lceil n/2 \rceil + 1 \,..\, n\,]$

Merge the two sorted lists

Key subroutine: MERGE

MERGE SORT EXAMPLE



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

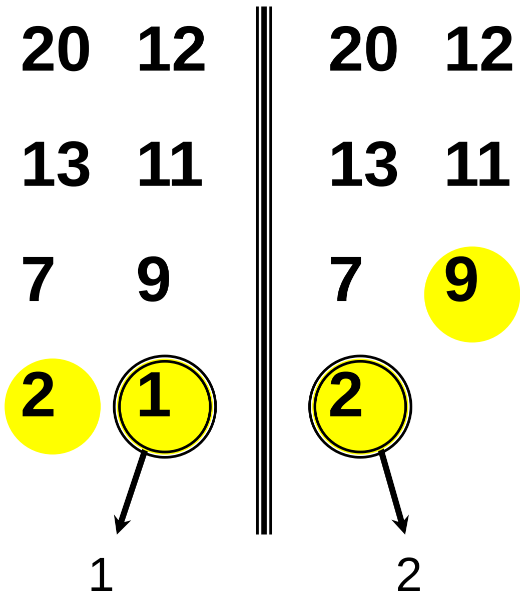
MERGING TWO SORTED ARRAYS

20	12
13	11
7	9
2	1

1

20	12
13	11
7	9
2	

MERGING TWO SORTED ARRAYS



MERGING TWO SORTED ARRAYS

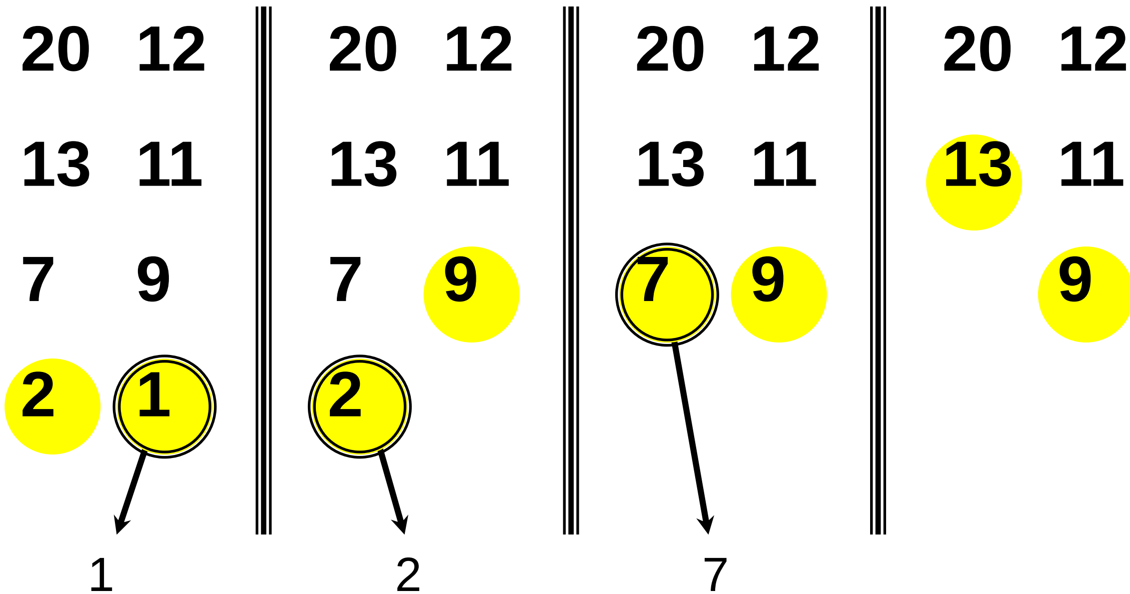
20	12		20	12		20	12
13	11		13	11		13	11
7	9		7	9		7	9
2	1		2				
	1			2			

MERGING TWO SORTED ARRAYS

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

1	2	7

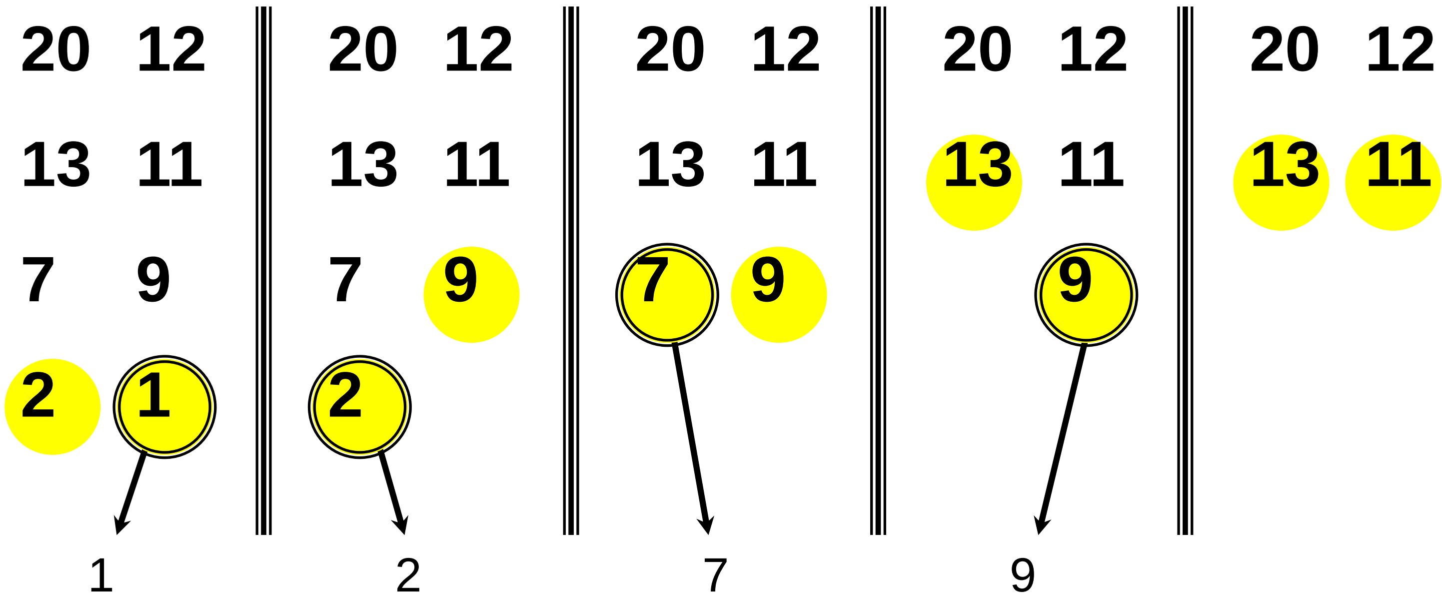
MERGING TWO SORTED ARRAYS



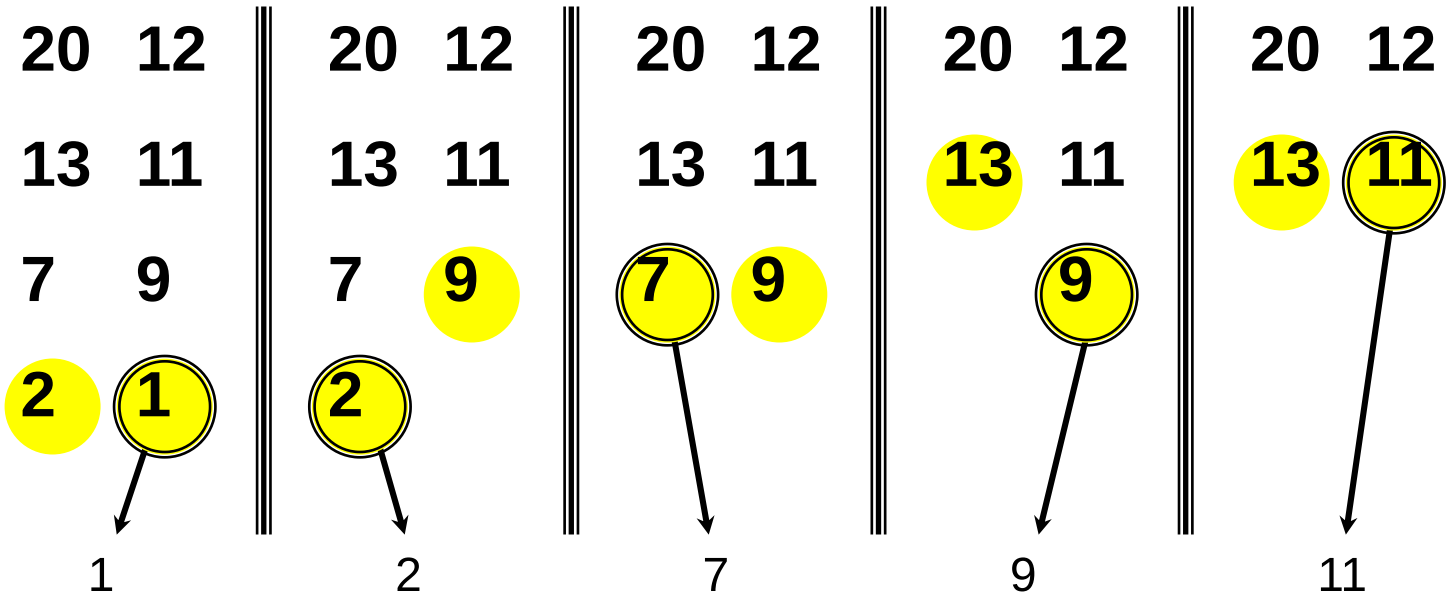
MERGING TWO SORTED ARRAYS

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

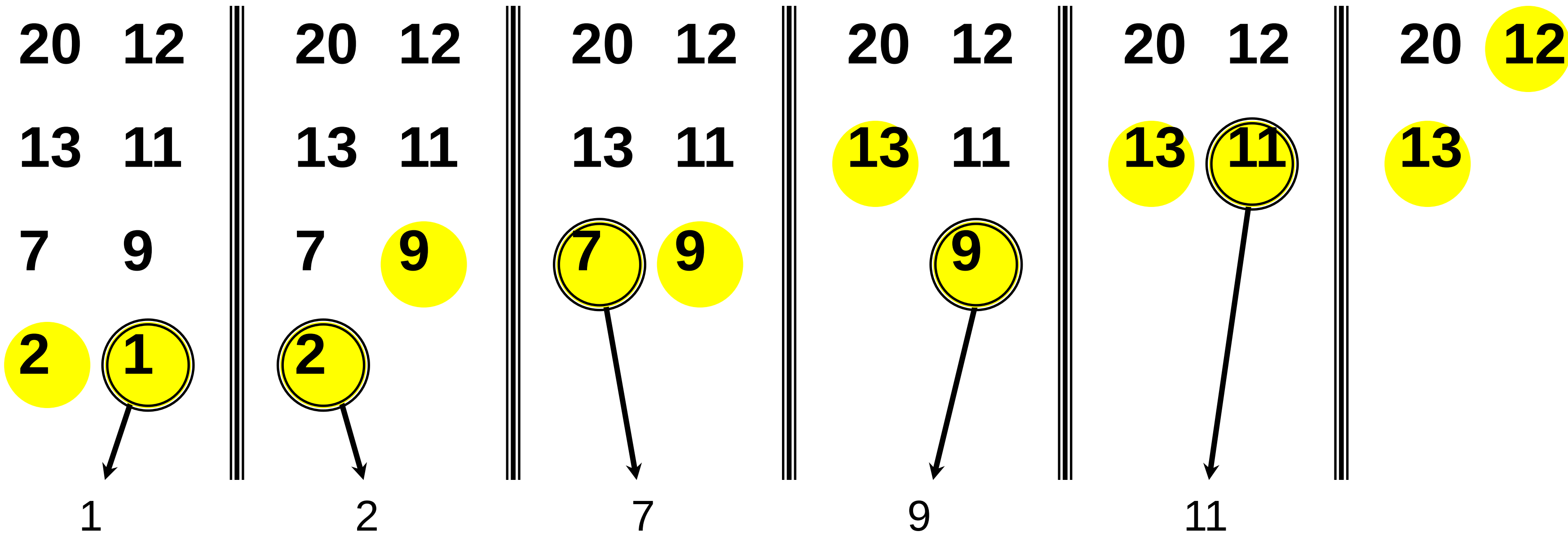
MERGING TWO SORTED ARRAYS



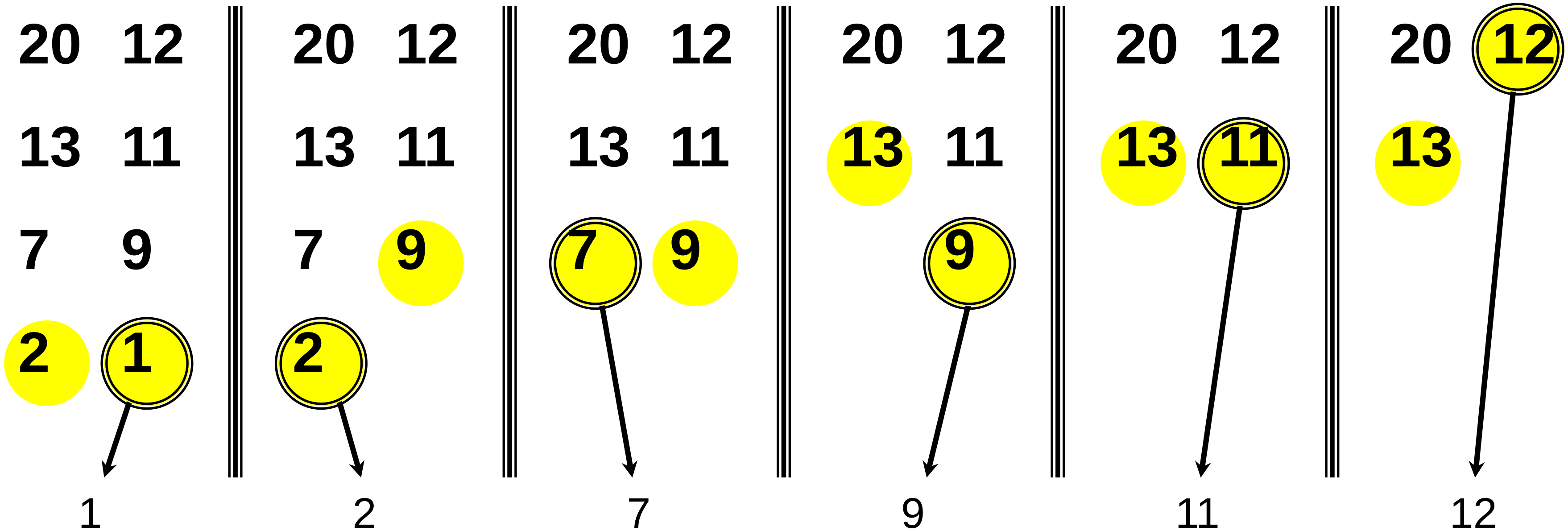
MERGING TWO SORTED ARRAYS



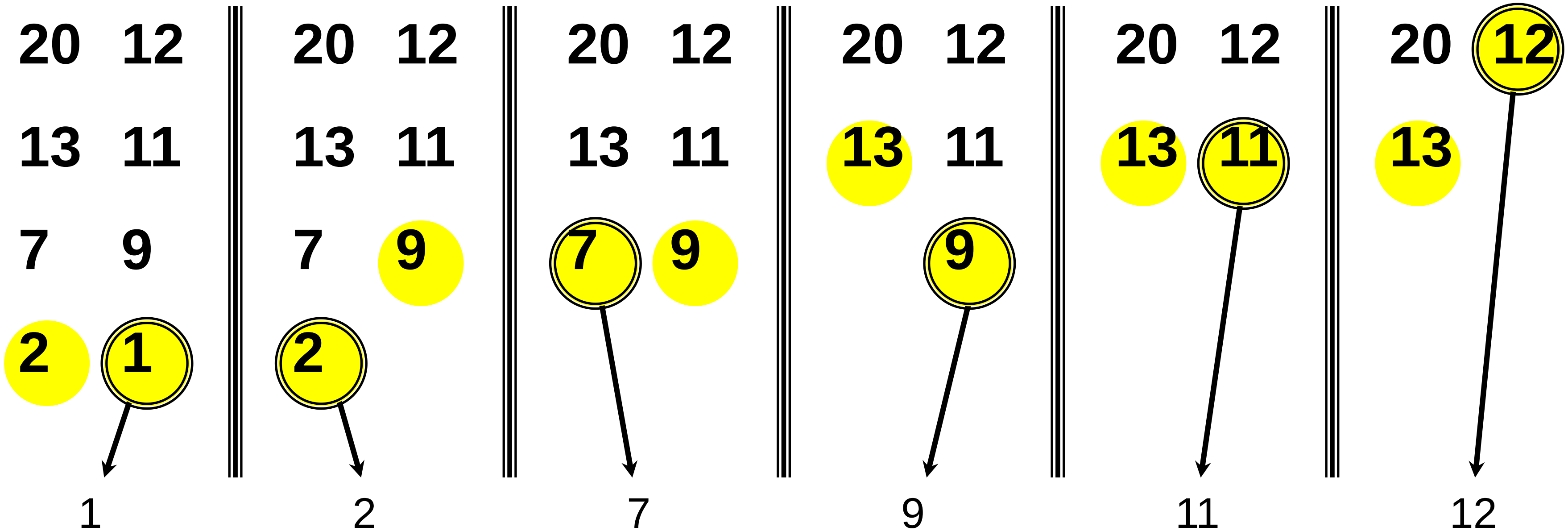
MERGING TWO SORTED ARRAYS



MERGING TWO SORTED ARRAYS



MERGING TWO SORTED ARRAYS



Time = O(*n*) to merge a total of *n* elements (linear time).

ANALYZING MERGE SORT

$T(n)$	MERGE-SORT $A[1 \,.\,.\, n]$
$O(1)$	If $n = 1$, return
$2T(n/2)$	Recursively sort $A[\,1 \,.\,.\, \lceil n/2 \rceil\,]$ and $A[\,\lceil n/2 \rceil + 1 \,.\,.\, n\,]$
$O(n)$	*Merge* the two sorted lists

Sloppiness: **Should be $T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor)$, but it turns out this does not matter asymptotically.**

RECURRENCE FOR MERGE SORT

$$T(n) = \begin{cases} O(1) \text{ if } n = 1 \\ 2T(n/2) + O(n) \text{ if } n > 1 \end{cases}$$

Note: Usually the base case $T(1) = O(1)$

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.

$$T(n)$$

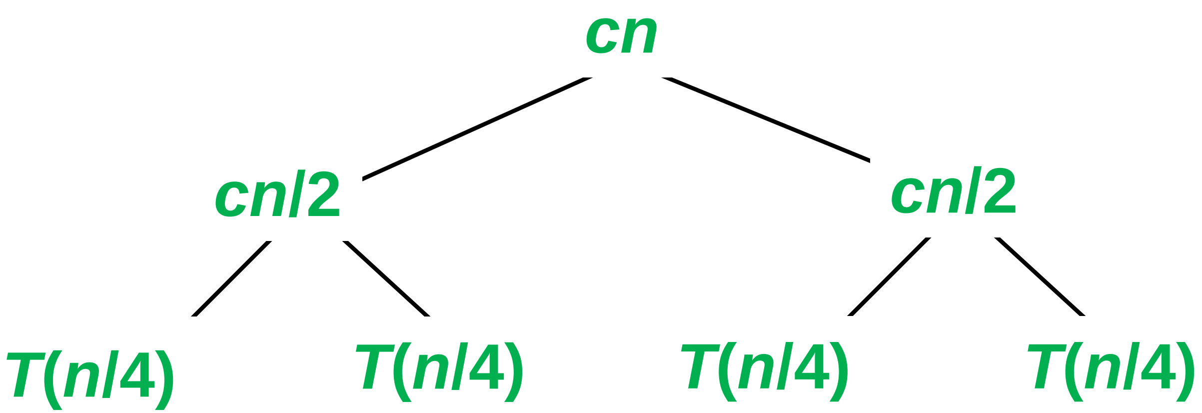
RECURSION TREE

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

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

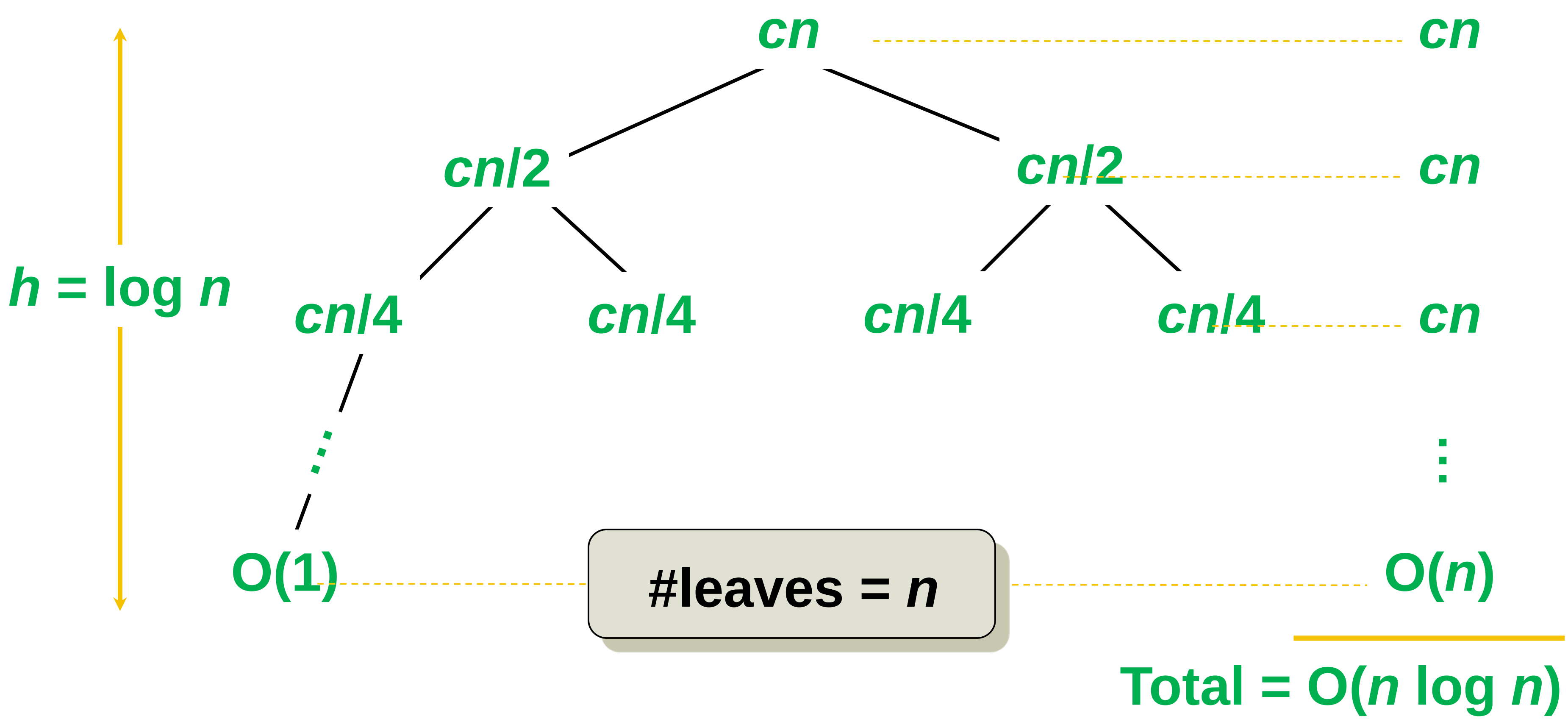
RECURSION TREE

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.



CONCLUSIONS

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

MASTER THEOREM

- **Let T(n) = a T(n/b) + f(n) where**
  - a ≥ 1, b > 1 are constants
  - f(n) is an asymptotically positive function

- **Then, T(n) can be bounded asymptotically as follows**
  - If $f(n)=O(n^{\log_b a-\varepsilon})$ for some ε > 0, then $T(n)=\Theta(n^{\log_b a})$
  - If $f(n)=\Theta(n^{\log_b a})$ then $T(n)=\Theta(n^{\log_b a} \log n)$
  - If $f(n)=\Omega(n^{\log_b a+\varepsilon})$ for some ε > 0, and if, for all sufficiently large n and a constant c < 1 we have *af(n/b)* ≤ *cf(n)* , then $T(n)=\Theta(f(n))$