CMP448: Algorithms

Lecture 01: Algorithm Analysis

Mohamed Alaa El-Dien Aly
Computer Engineering Department
Cairo University
Spring 2013

Agenda

- Analysis of Algorithms
- Insertion Sort
- Asymptotic Analysis
- Merge Sort
- Recurrences

Acknowledgment
A lot of slides adapted from the slides of Erik Demaine and Charles Leiserson

Algorithms

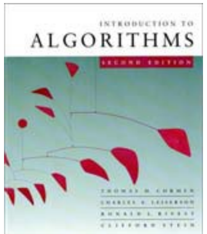
Algorithm

A computational procedure that takes some values as input and produces some values as output

Algorithm Analysis

Determining the *resources* required by the algorithm as a function of the input size.

Resources include *space* and *time*.



Why study algorithms and performance?

- Algorithms help us to understand ***scalability***.
- Performance often draws the line between what is feasible and what is impossible.
- Algorithmic mathematics provides a ***language*** for talking about program behavior.
- Performance is the ***currency*** of computing.
- The lessons of program performance generalize to other computing resources.
- Speed is fun!

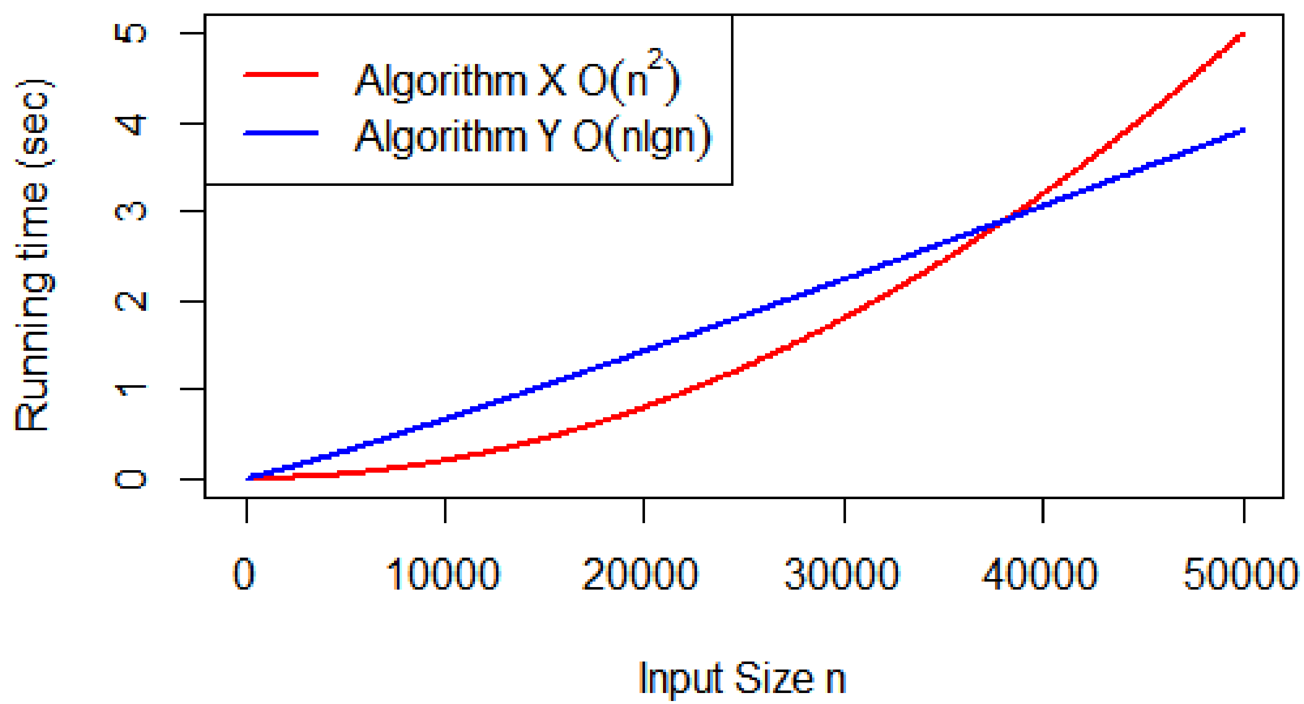
Running time

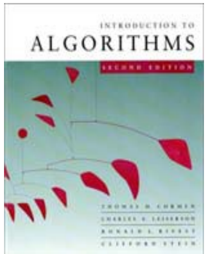
- Assume:
  - Algorithm X takes time $2n^2$ (written by best programmer) running on machine with 1000 MIPS
  - Algorithm Y takes time $50n \lg n$ (written by worst programmer) running on machine with 10 MIPS
- Running time for 10^6 numbers
  - Algorithm X takes 2000 seconds
  - Algorithm Y takes ~100 seconds

Complexity makes a huge difference!

Running time

- Assume:
  - Algorithm X takes time $2n^2$ (written by best programmer) running on machine with 1000 MIPS
  - Algorithm Y takes time $50n \lg n$ (written by worst programmer) running on machine with 10 MIPS





The problem of sorting

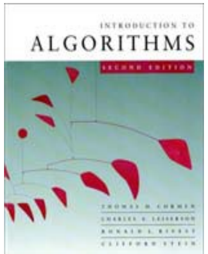
Input: sequence $\langle a_1, a_2, \ldots, a_n \rangle$ of numbers.

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



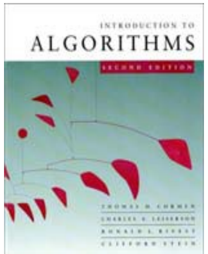
Insertion sort

"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
```

Insertion Sort

```
void insertion_sort(vector<int>& A) {
  for (int j = 1; j < A.size(); ++j) {
    int key = A[j];
    int i = j - 1;
    for (; i >= 0 && A[i] > key;) {
      A[i+1] = A[i--];
    }
    A[i+1] = key;
  }
}
```



Insertion sort

"pseudocode"

INSERTION-SORT (A, n) $\triangleleft A[1 \,..\, n]$

    for $j \leftarrow 2$ **to** n

        do $key \leftarrow A[j]$

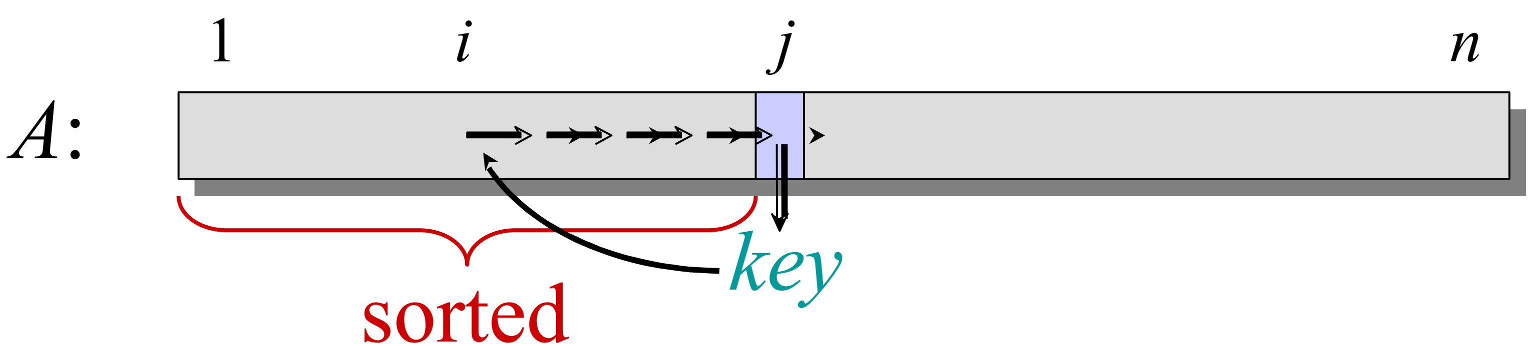
            $i \leftarrow j - 1$

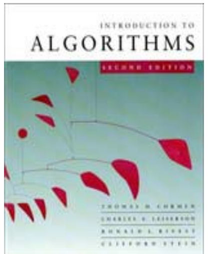
            while $i > 0$ and $A[i] > key$

                do $A[i+1] \leftarrow A[i]$

                    $i \leftarrow i - 1$

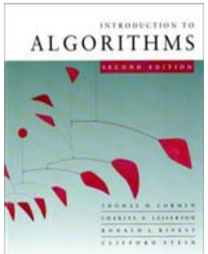
            $A[i+1] = key$





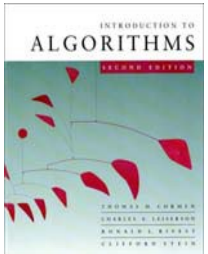
Example of insertion sort

8 2 4 9 3 6



Example of insertion sort

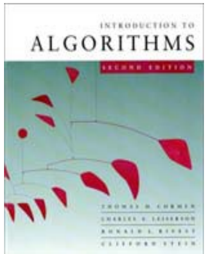
8 2 4 9 3 6



Example of insertion sort

8 2 4 9 3 6

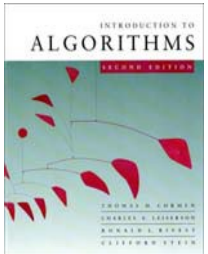
2 8 4 9 3 6



Example of insertion sort

8 **2** 4 9 3 6

2 8 **4** 9 3 6

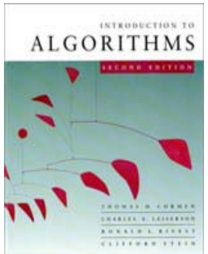


Example of insertion sort

8 2 4 9 3 6

2 8 4 9 3 6

2 4 8 9 3 6

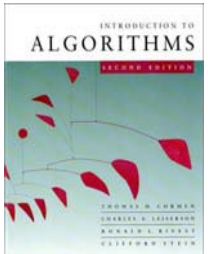


Example of insertion sort

8 2 4 9 3 6

2 8 4 9 3 6

2 4 8 9 3 6



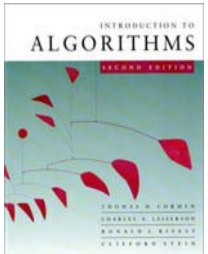
Example of insertion sort

8 **2** 4 9 3 6

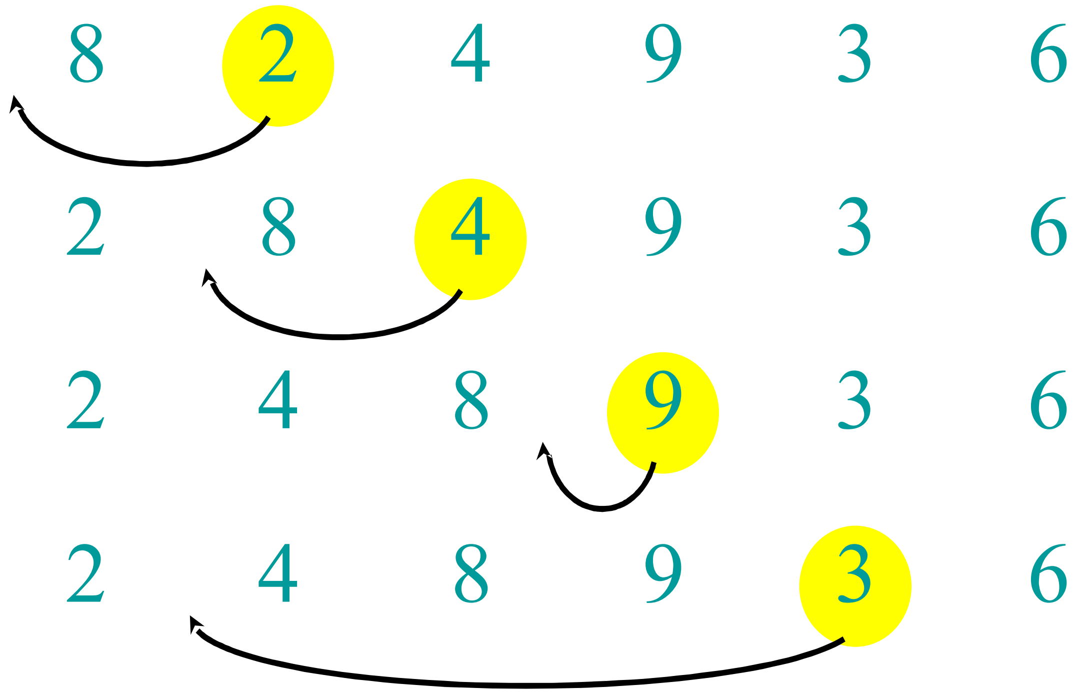
2 8 **4** 9 3 6

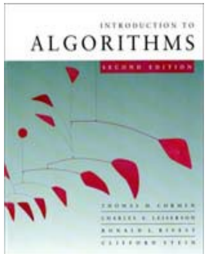
2 4 8 **9** 3 6

2 4 8 9 3 6



Example of insertion sort





Example of insertion sort

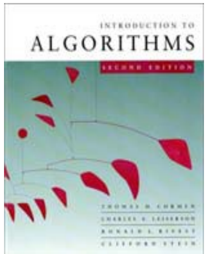
8 2 4 9 3 6

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

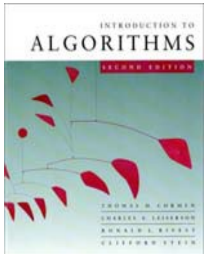
8 **2** 4 9 3 6

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

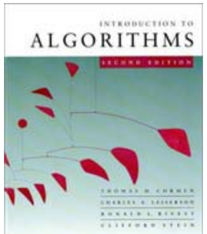
2 8 4 9 3 6

2 4 8 9 3 6

2 4 8 9 3 6

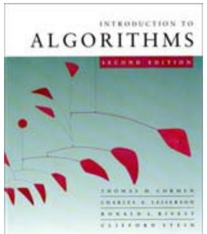
2 3 4 8 9 6

2 3 4 6 8 9 *done*



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.



Kinds of analyses

Worst-case: (usually)

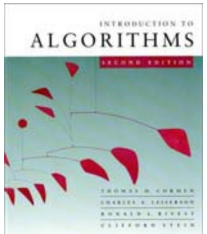
- $T(n)$ = maximum time of algorithm on any input of size n.

Average-case: (sometimes)

- $T(n)$ = expected time of algorithm over all inputs of size n.
- Need assumption of statistical distribution of inputs.

Best-case: (bogus)

- Cheat with a slow algorithm that works fast on *some* input.



Machine-independent time

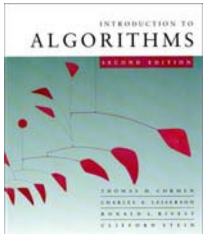
What is insertion sort's worst-case time?

- It depends on the speed of our computer:
    - relative speed (on the same machine),
    - absolute speed (on different machines).

BIG IDEA:

- Ignore machine-dependent constants.
- Look at ***growth*** of $T(n)$ as $n \rightarrow \infty$.

"Asymptotic Analysis"



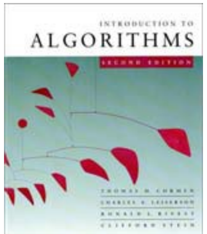
Θ-notation

Math:

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

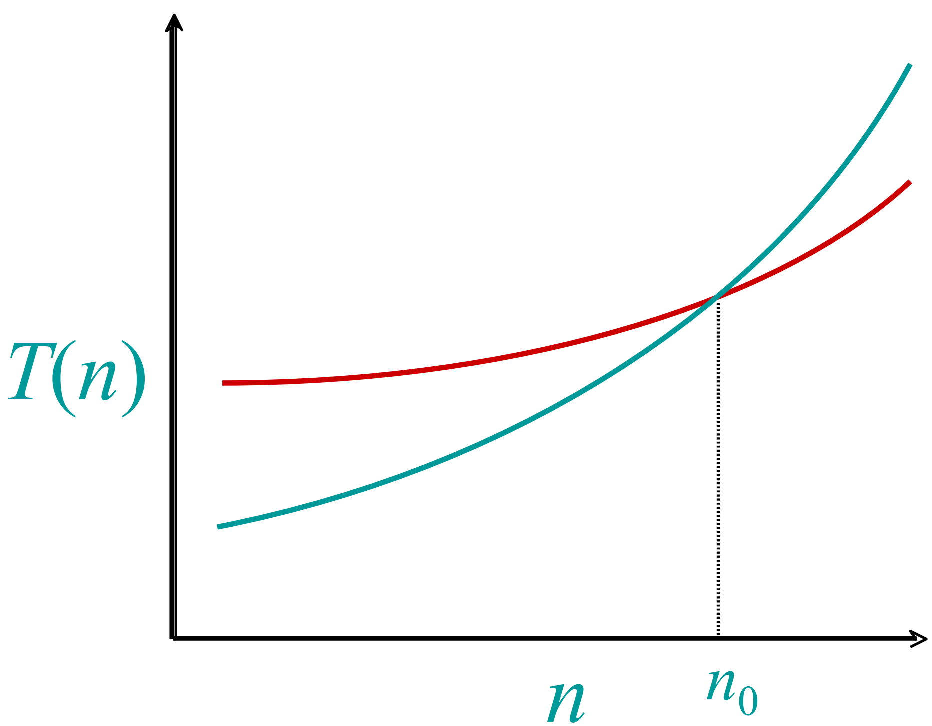
Engineering:

- 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.

Insertion Sort Analysis

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

Sum of all terms

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

Insertion Sort Analysis

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

Best case: sorted array

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

$$T(n) = an + b = \Theta(n)$$

Insertion Sort Analysis

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

Worst case: reversed array

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

Why? $$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$ Why?

Insertion Sort Analysis

$$\sum_{i=1}^{n} i = 1 + \quad 2 + \quad \cdots + \quad n-1 + \quad n$$

$$\sum_{i=1}^{n} i = n + \quad n-1 + \quad \cdots + \quad 2 + \quad 1$$

$$2\sum_{i=1}^{n} i = (n+1) + \quad (n+1) + \quad \cdots + \quad (n+1) + \quad (n+1)$$

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Review in Appendix A

Insertion Sort Analysis

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

Worst case: reversed array

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

$$T(n) = a\,n^2 + bn + c = \Theta(n^2)$$

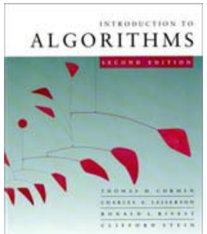
Insertion Sort Analysis

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

Average case: all permutations equally likely

$$\sum_{j=2}^{n} t_j = \sum_{j=2}^{n} \frac{j}{2}$$

$$T(n) = a\,n^2 + bn + c = \Theta(n^2)$$



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.

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

Is insertion sort a fast sorting algorithm?

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

Insertion Sort Analysis

- What about space?
- Insertion sorts "in place" as it does not copy the array anywhere
- It only takes a constant amount of extra storage, independent of n
- Therefore $S(n) = \Theta(1)$

Merge Sort

- Divide-and-Conquer
  - *Divide* the problem into a number of sub-problems
  - *Conquer* the smaller problems
  - *Combine* the results of the sub-problems into a solution for the big problem

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

Merge Sort

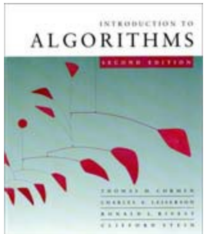
- Divide-and-Conquer
  - *Divide* the problem into a number of sub-problems
  - *Conquer* the smaller problems
  - *Combine* the results of the sub-problems into a solution for the big problem

```
void merge_sort(vector<int>& A, int p, int r) {
  if (p >= r) return;

  int q = (p + r) / 2;

  merge_sort(A, p, q);
  merge_sort(A, q+1, r);

  merge(A, p, q, r);
}
```

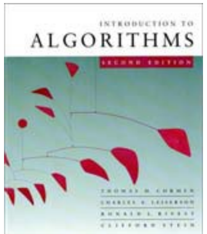


Merge sort

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

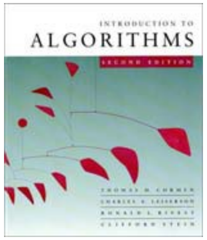
1. If $n = 1$, done.
2. Recursively sort $A[\,1 \,..\, \lceil n/2 \rceil\,]$ and $A[\,\lceil n/2 \rceil + 1 \,..\, n\,]$.
3. "***Merge***" the 2 sorted lists.

Key subroutine: **MERGE**



Merging two sorted arrays

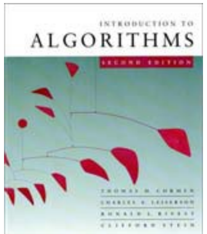
20	12
13	11
7	9
2	1



Merging two sorted arrays

20	12
13	11
7	9
2	1

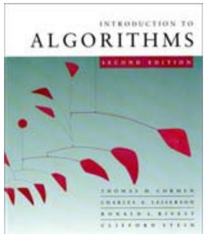
1



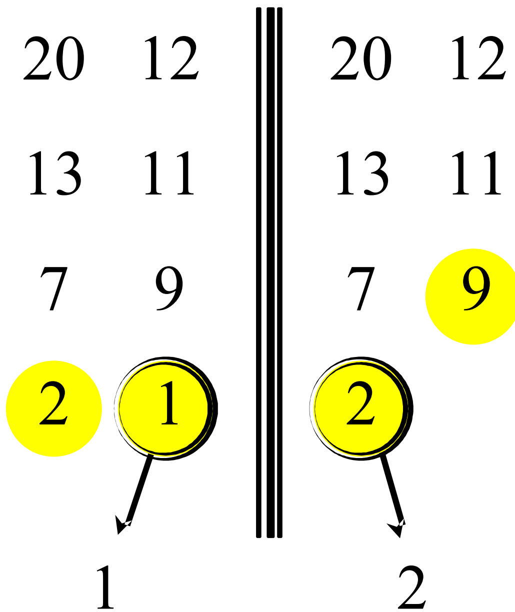
Merging two sorted arrays

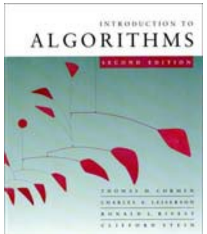
20	12	20	12
13	11	13	11
7	9	7	9
2	1	2	

1

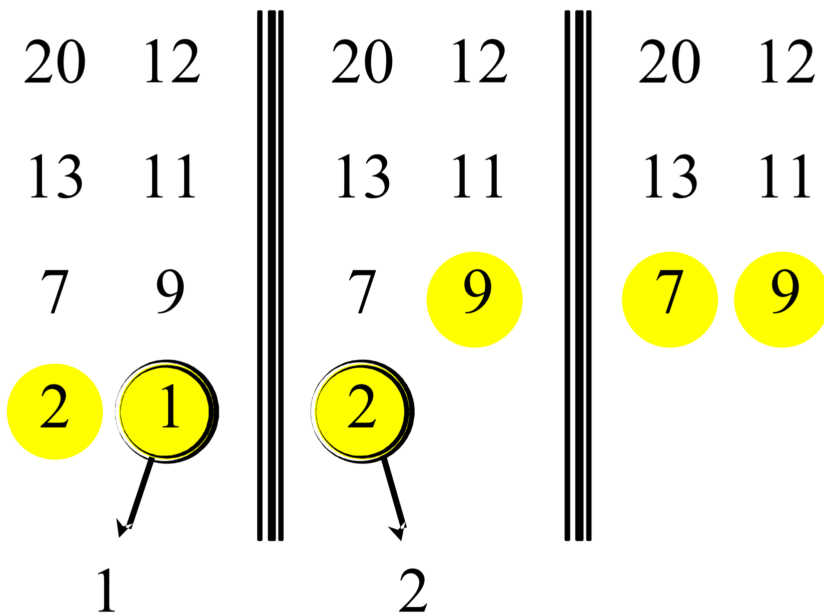


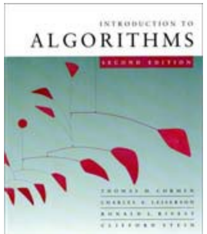
Merging two sorted arrays



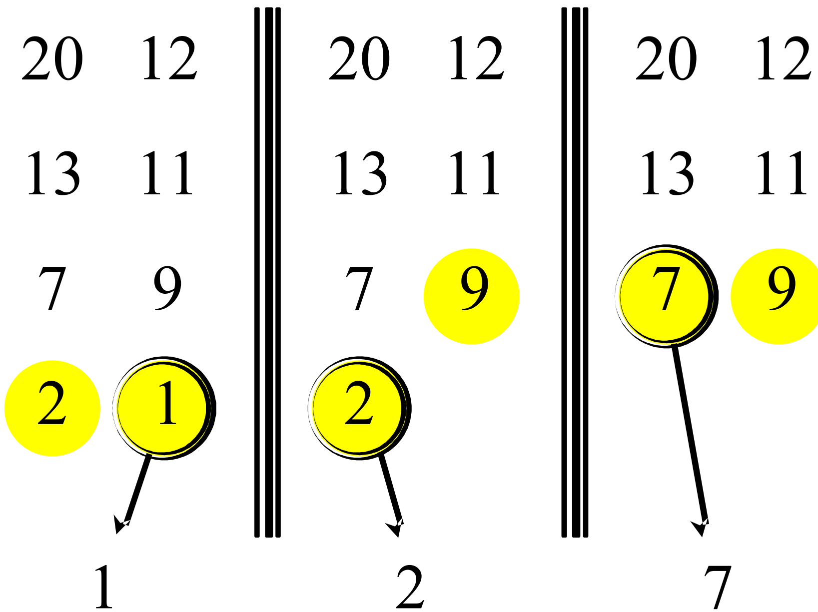


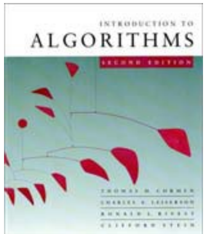
Merging two sorted arrays



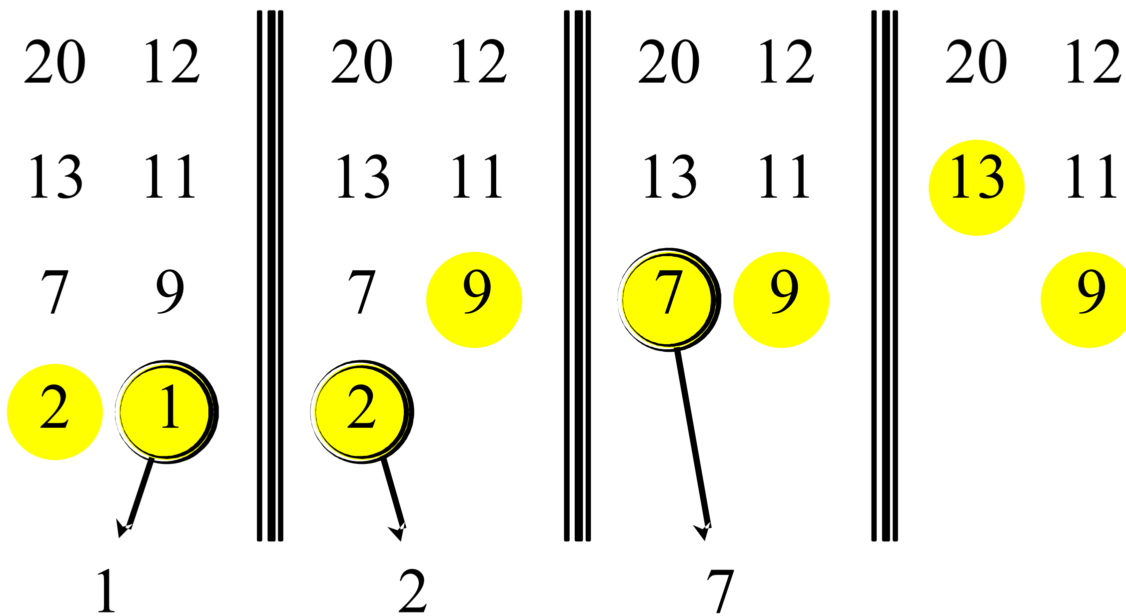


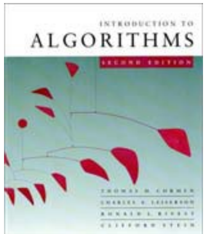
Merging two sorted arrays



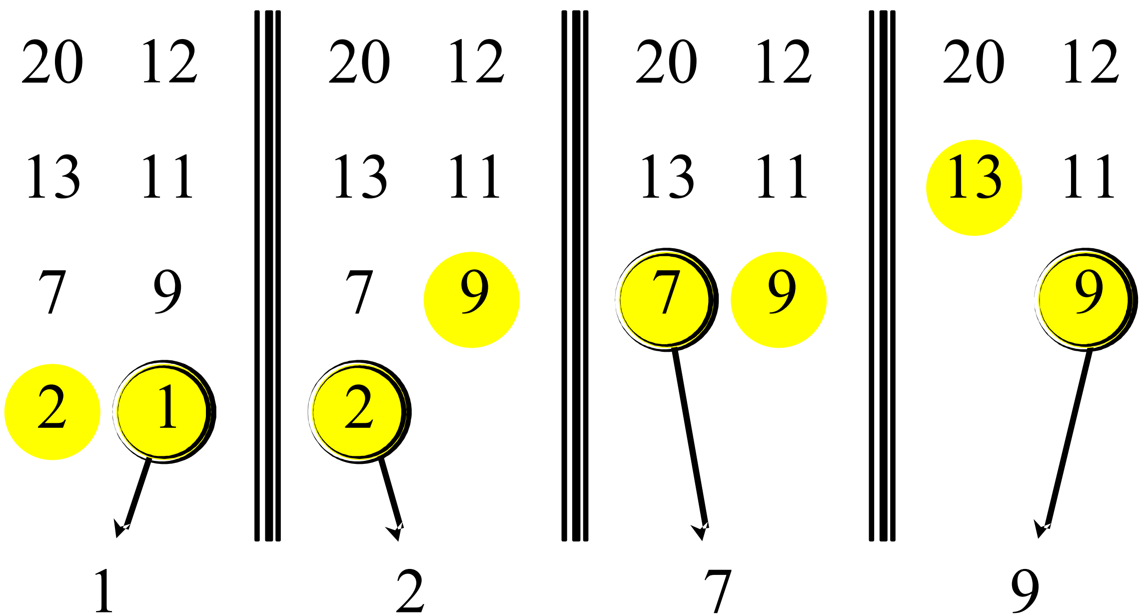


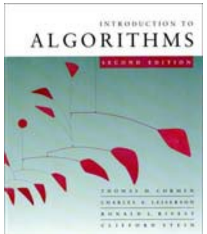
Merging two sorted arrays



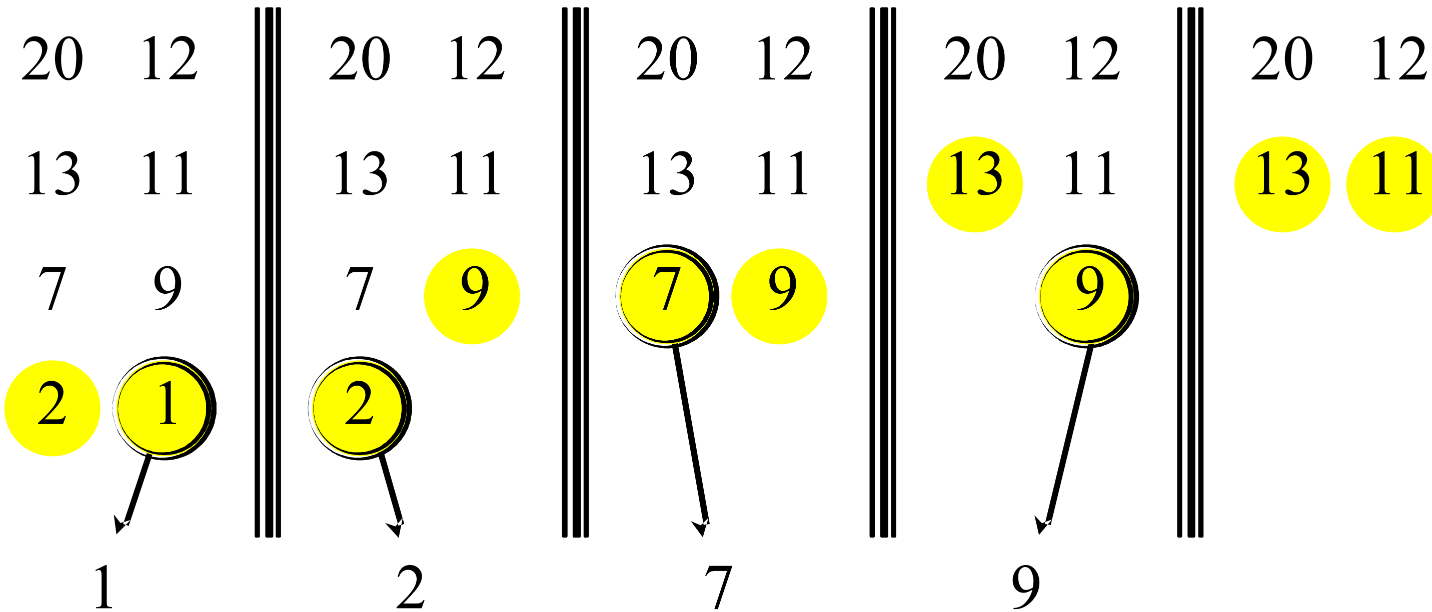


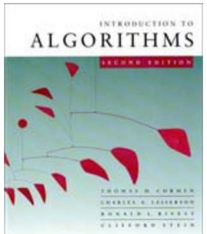
Merging two sorted arrays



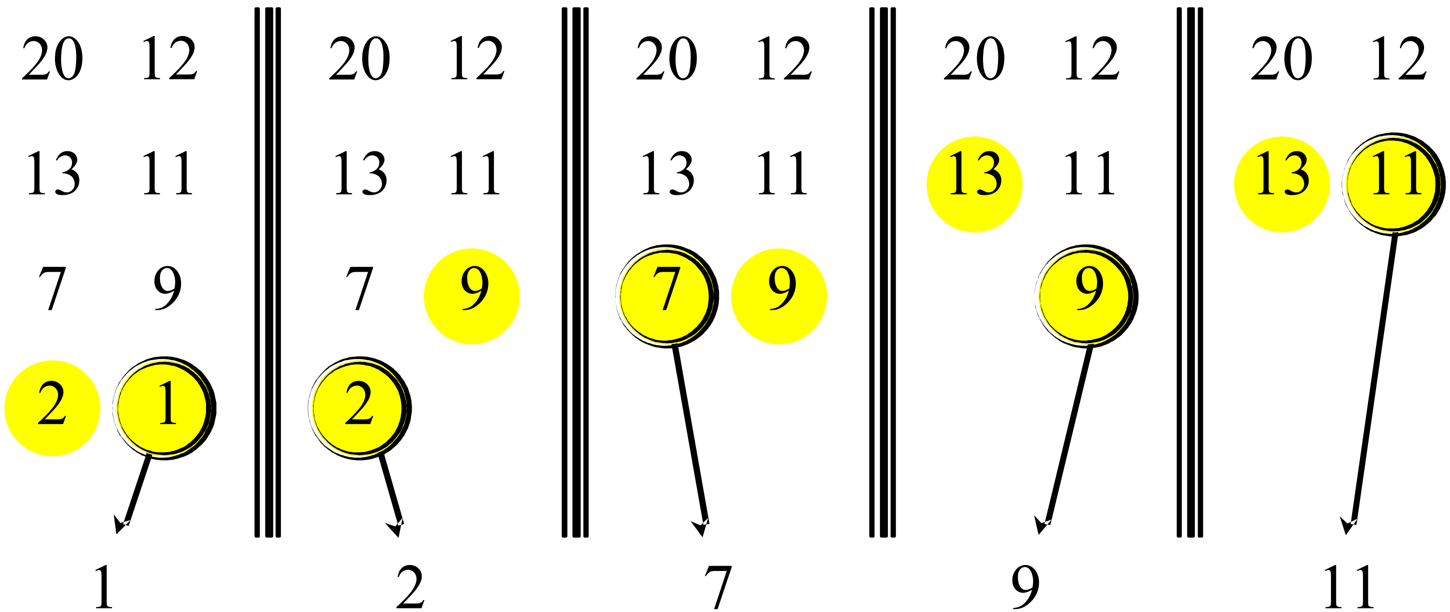


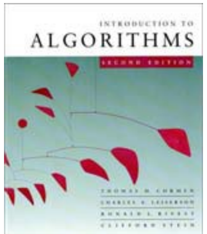
Merging two sorted arrays



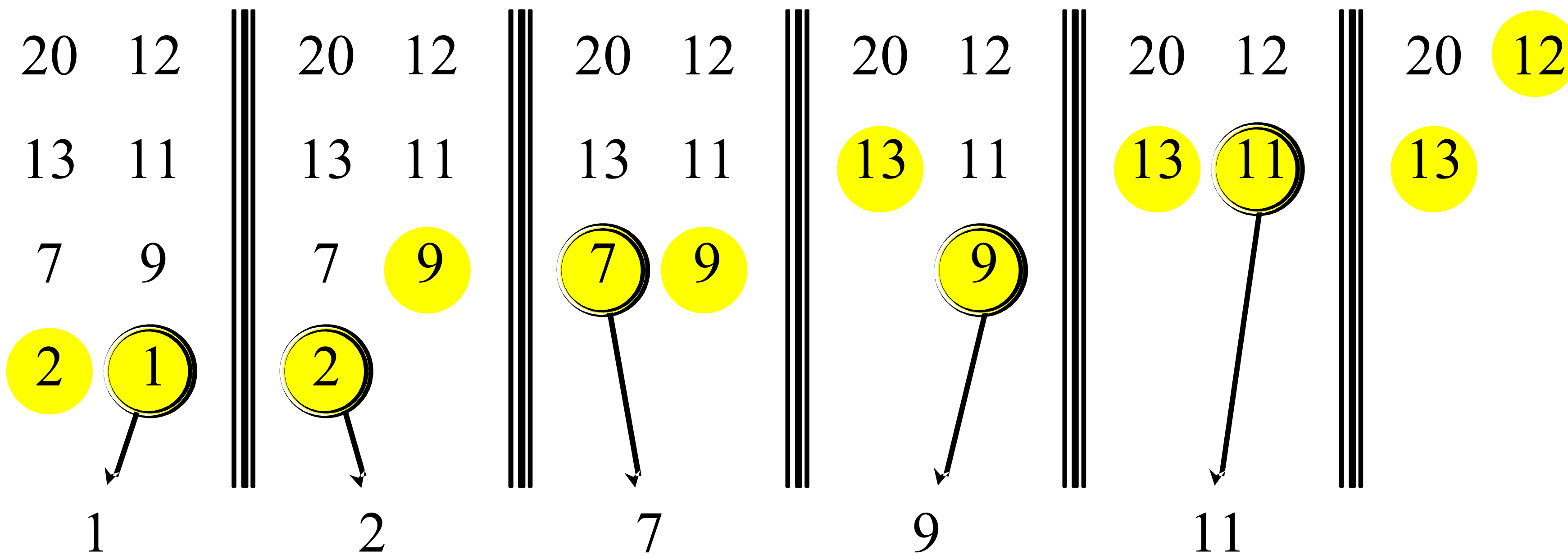


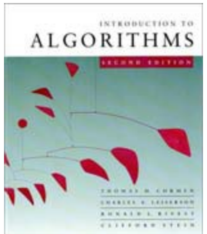
Merging two sorted arrays



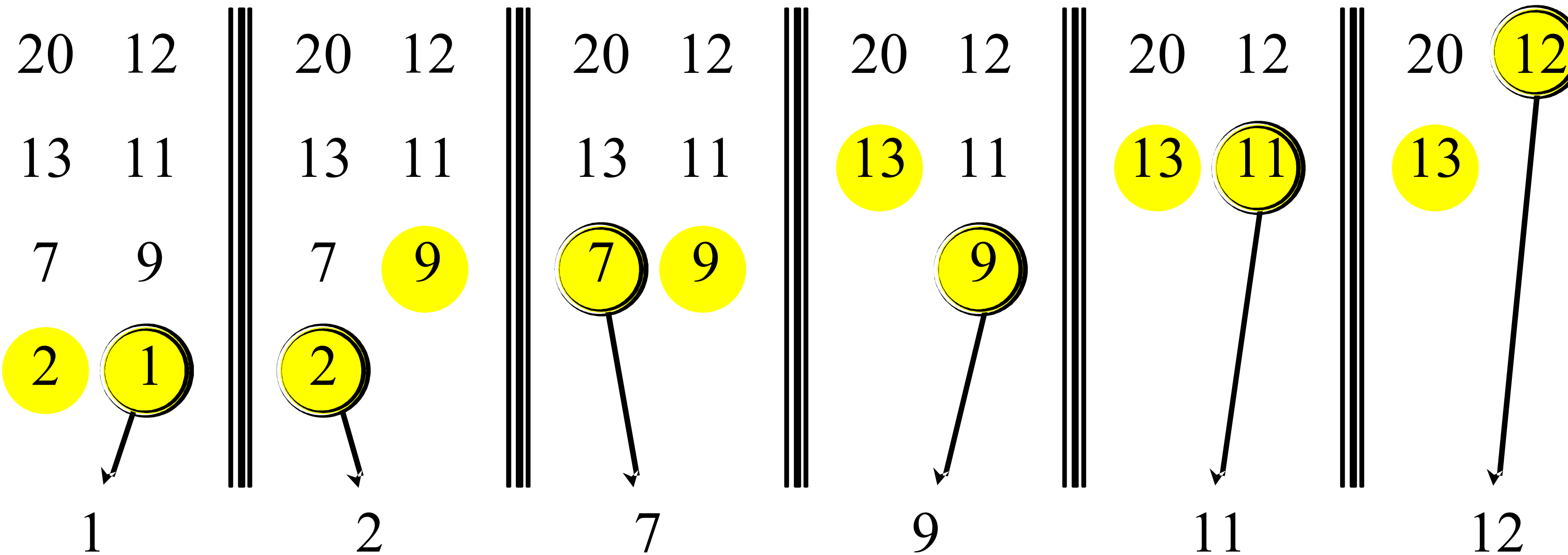


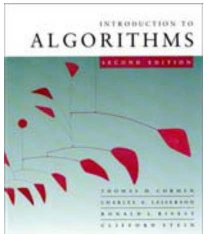
Merging two sorted arrays



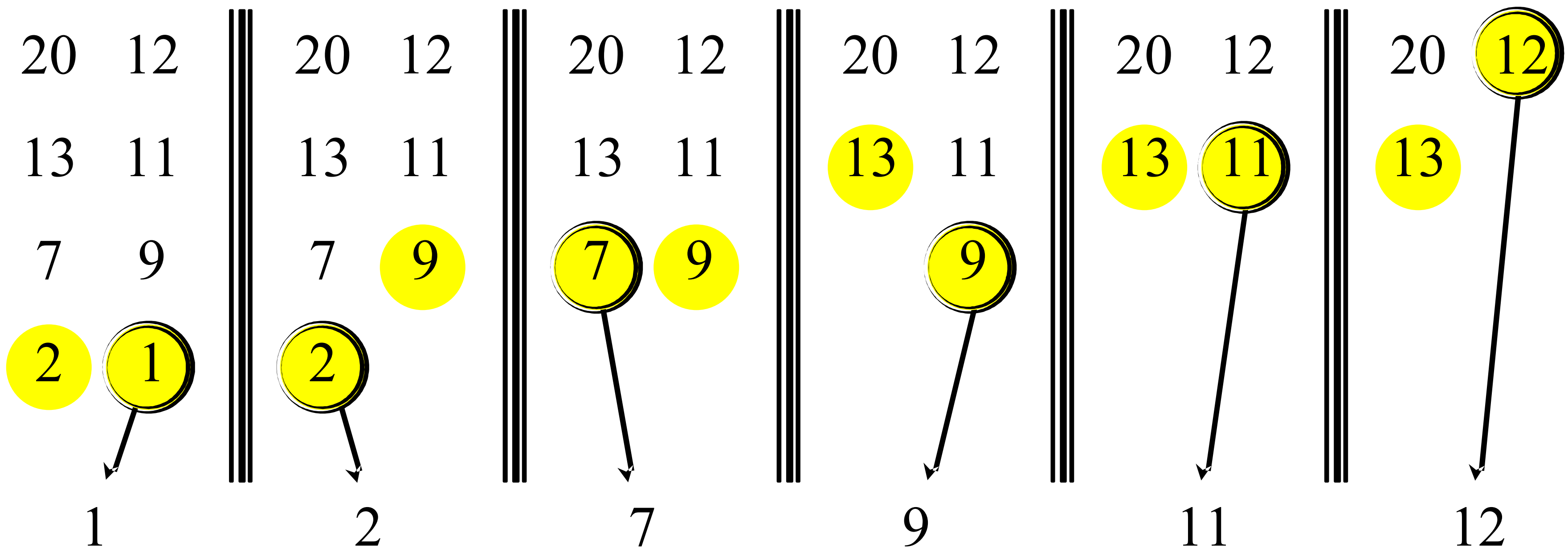


Merging two sorted arrays





Merging two sorted arrays



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

Merge

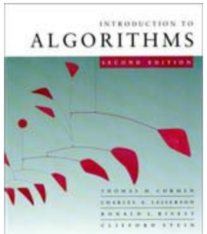
```
MERGE(A, p, q, r)
n1 = q − p + 1
n2 = r − q
let L[1 .. n1 + 1] and R[1 .. n2 + 1] be new arrays
for i = 1 to n1
        L[i] = A[p + i − 1]
for j = 1 to n2
        R[j] = A[q + j]
L[n1 + 1] = ∞
R[n2 + 1] = ∞
i = 1
j = 1
for k = p to r
        if L[i] ≤ R[j]
                A[k] = L[i]
          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

compute sizes

copy arrays to be merged

adds “sentinel”

merge



Analyzing merge sort

$T(n)$	**MERGE-SORT** $A[1 \,.\,.\, n]$
$\Theta(1)$	1. If $n = 1$, done.
$2T(n/2)$	2. Recursively sort $A[\,1 \,.\,.\, \lceil n/2 \rceil\,]$ and $A[\,\lceil n/2 \rceil + 1 \,.\,.\, n\,]$.
$\Theta(n)$	3. ***"Merge"*** the 2 sorted lists

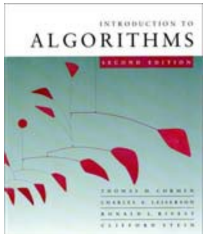
Abuse

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

Recurrence for Merge Sort

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

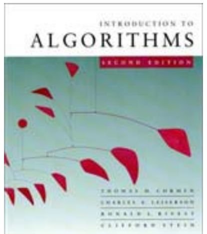
- Next we need to solve this recurrence relation i.e. find *T(n)* as a function of *n* without *T(n/2)*



Recurrence for merge sort

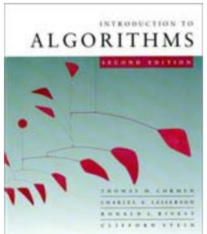
$$T(n) = \begin{cases} \Theta(1) \text{ if } n = 1; \\ 2T(n/2) + \Theta(n) \text{ if } n > 1. \end{cases}$$

- We shall usually omit stating the base case when $T(n) = \Theta(1)$ for sufficiently small n, but only when it has no effect on the asymptotic solution to the recurrence.
- CLRS and Lecture 2 provide several ways to find a good upper bound on $T(n)$.



Recursion tree

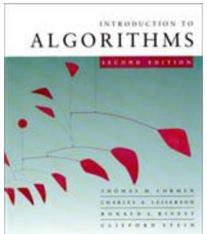
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



Recursion tree

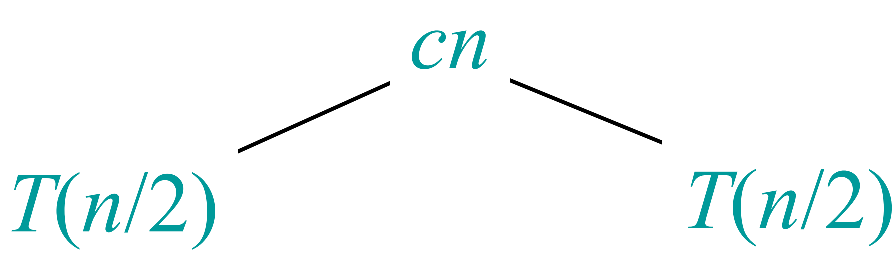
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

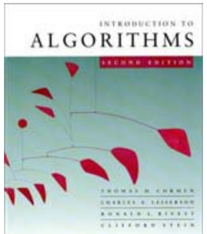
$$T(n)$$



Recursion tree

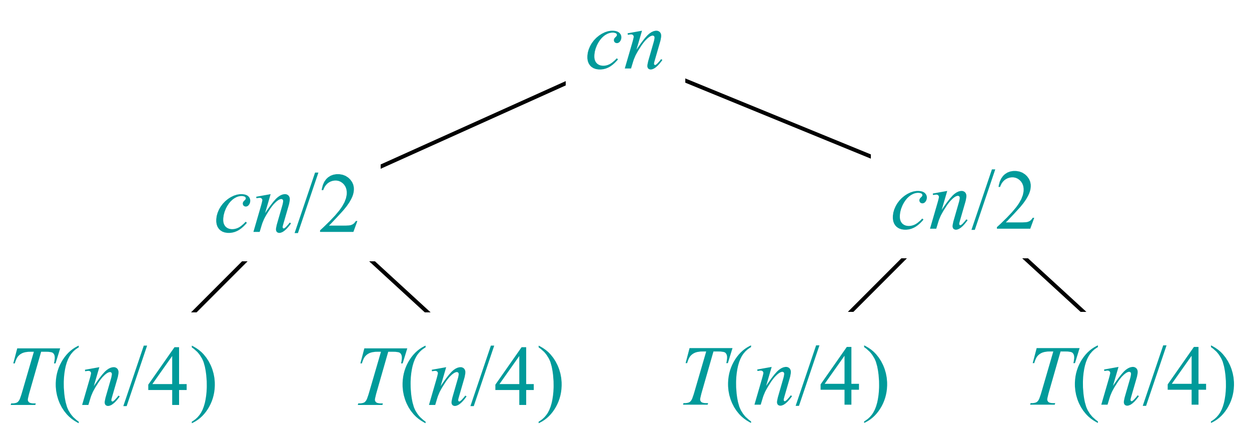
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

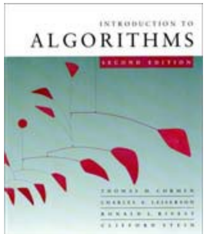




Recursion tree

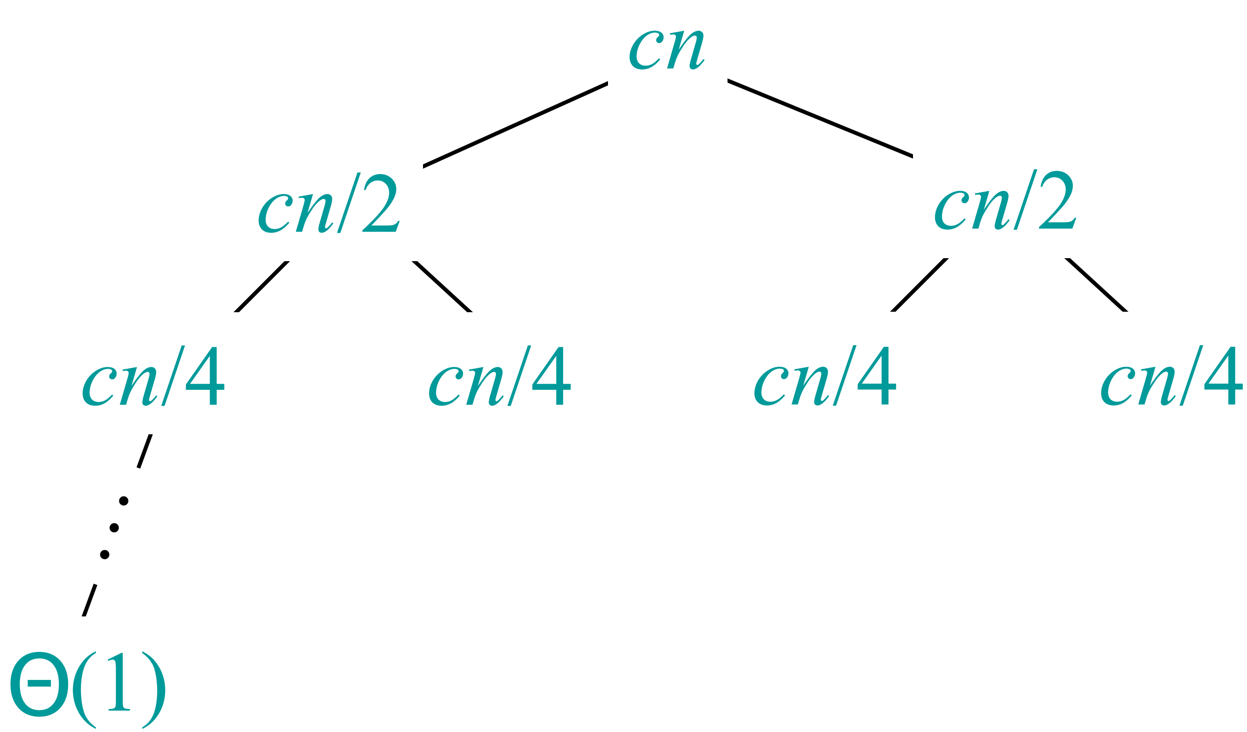
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

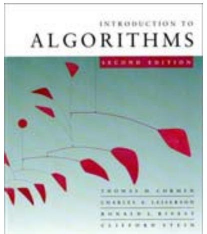




Recursion tree

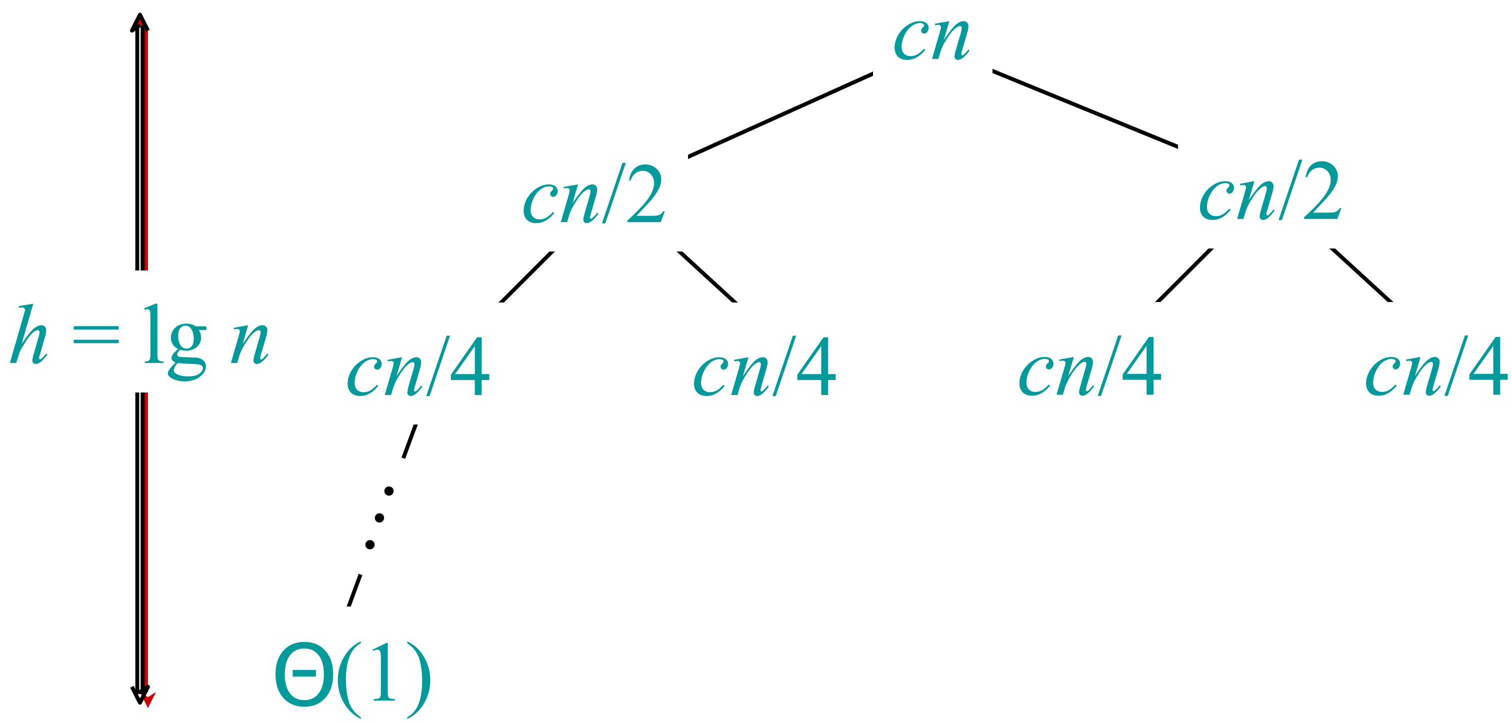
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

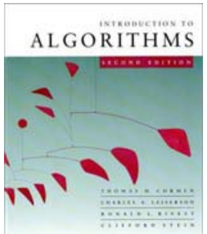




Recursion tree

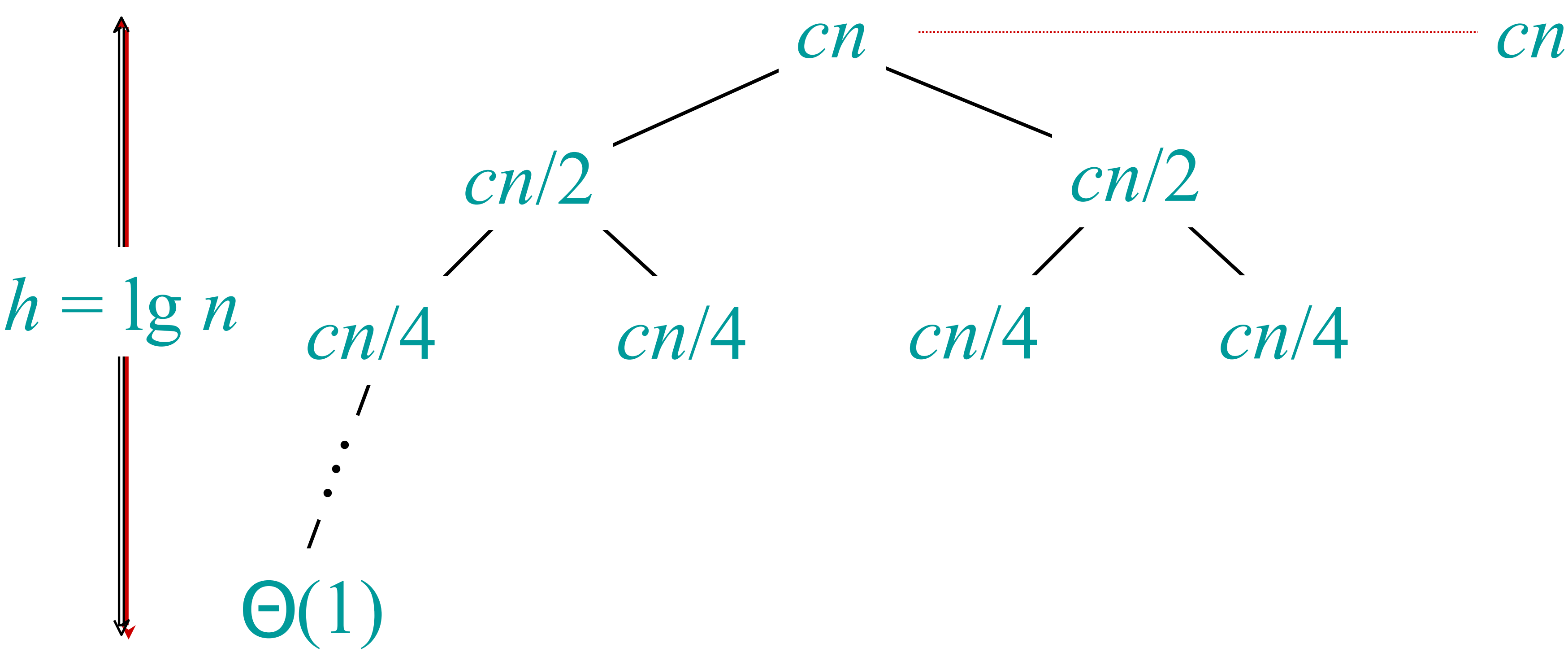
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

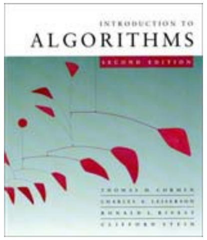




Recursion tree

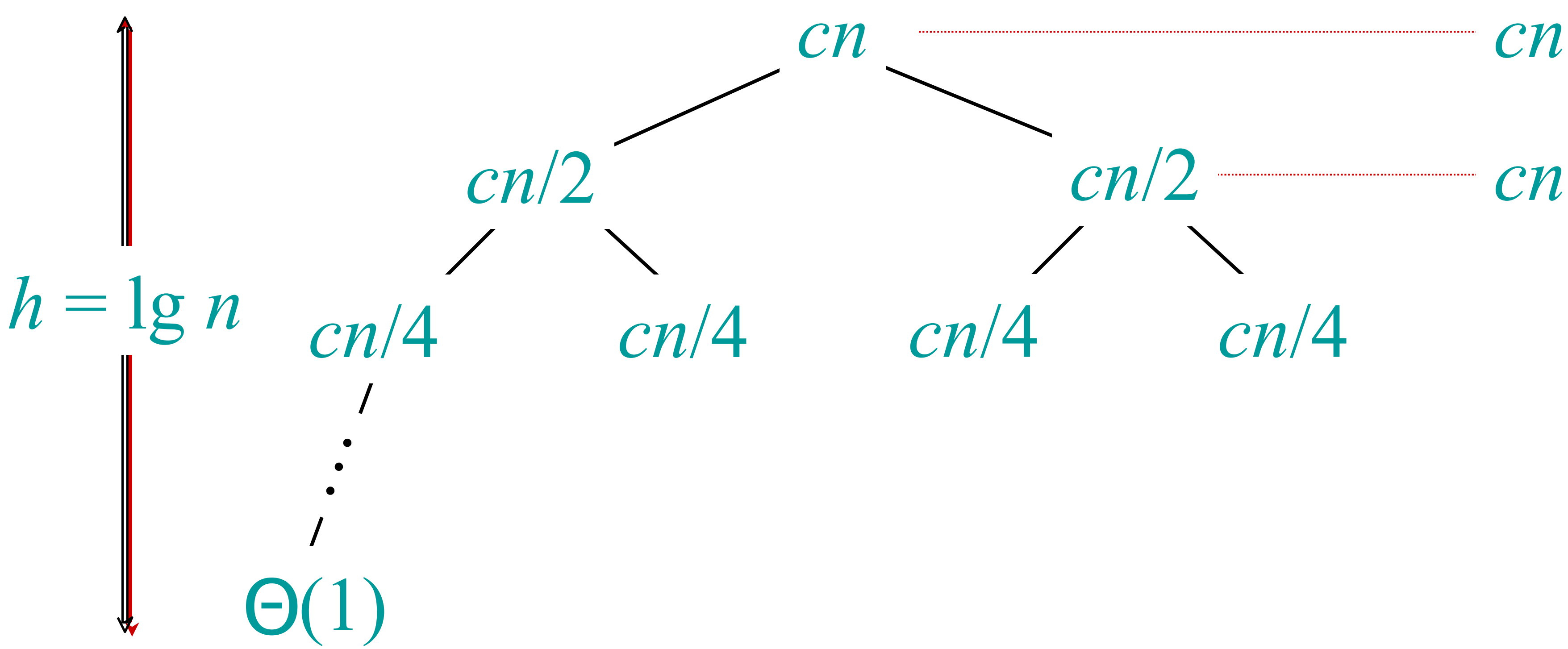
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

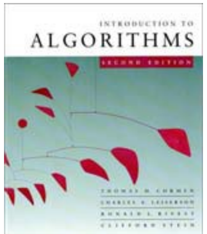




Recursion tree

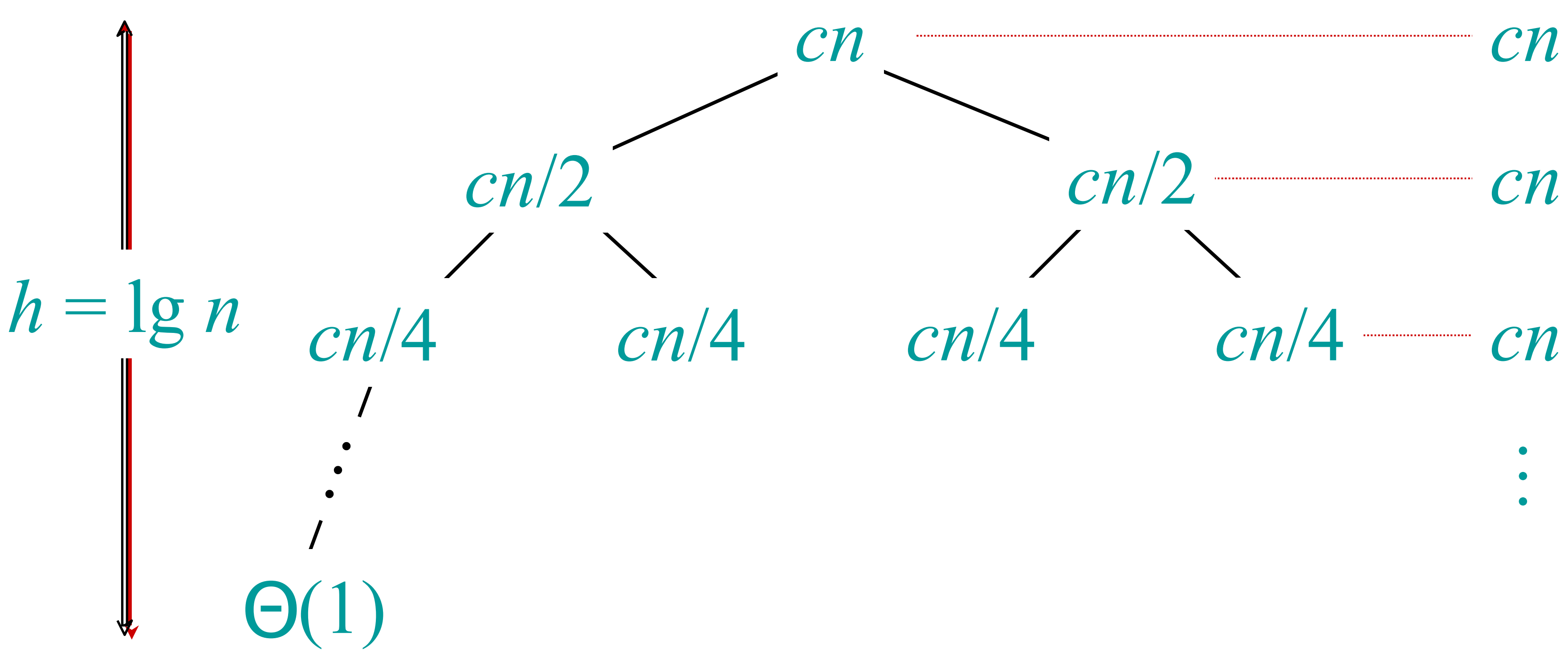
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

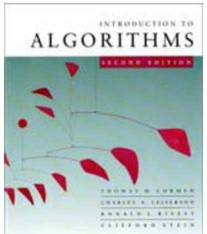




Recursion tree

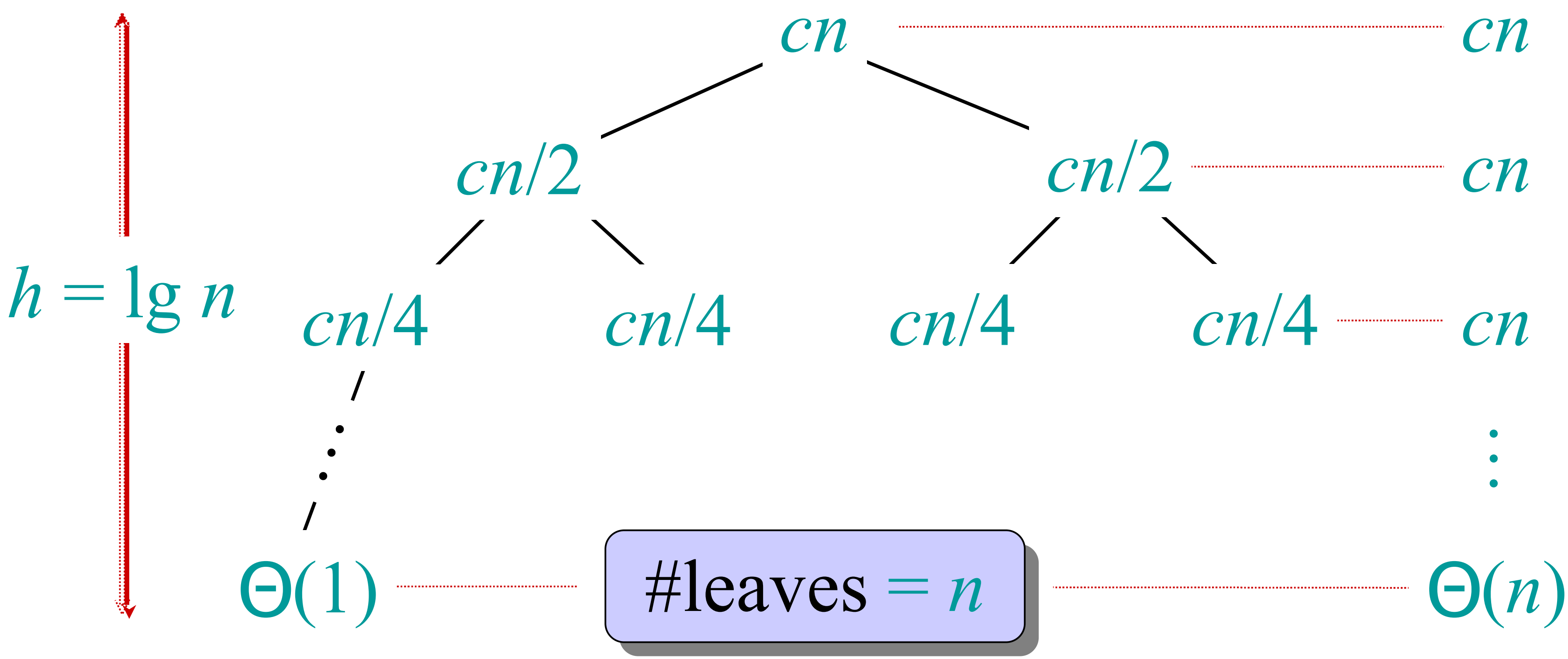
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

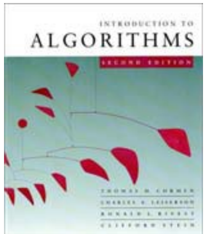




Recursion tree

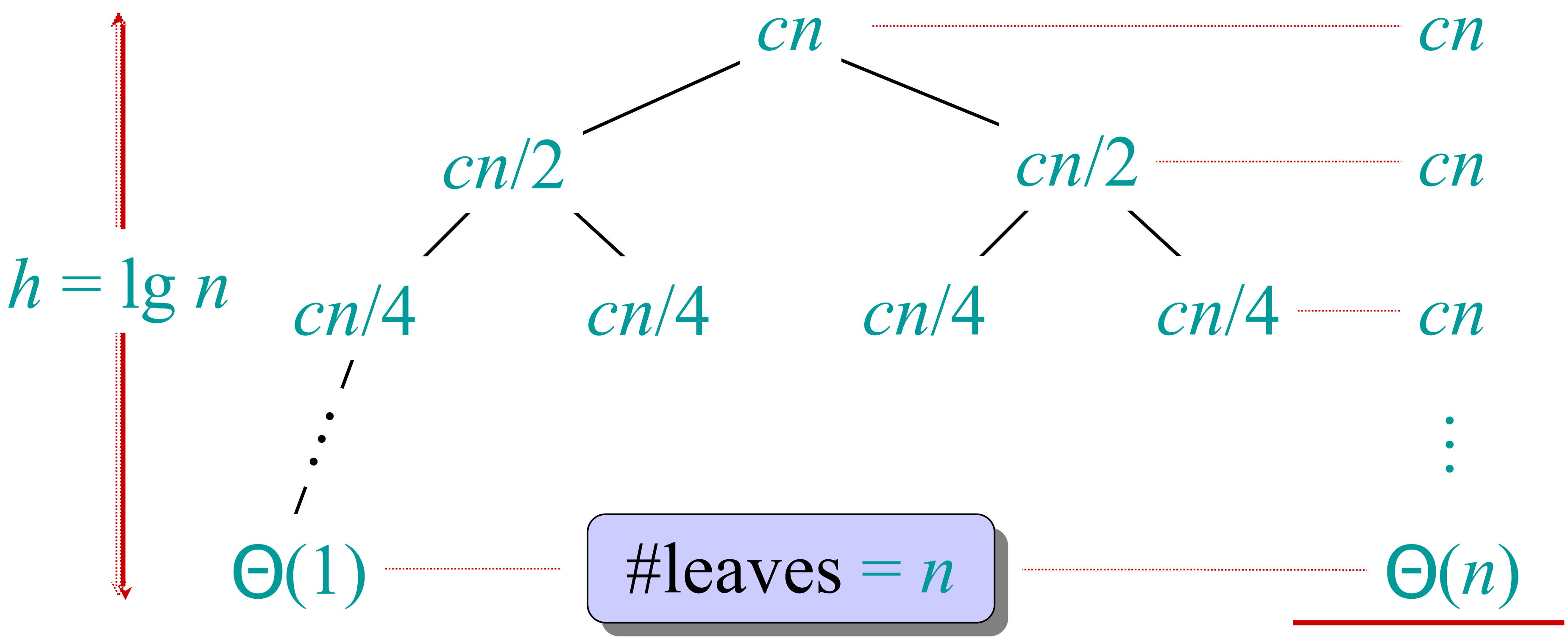
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.





Recursion tree

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



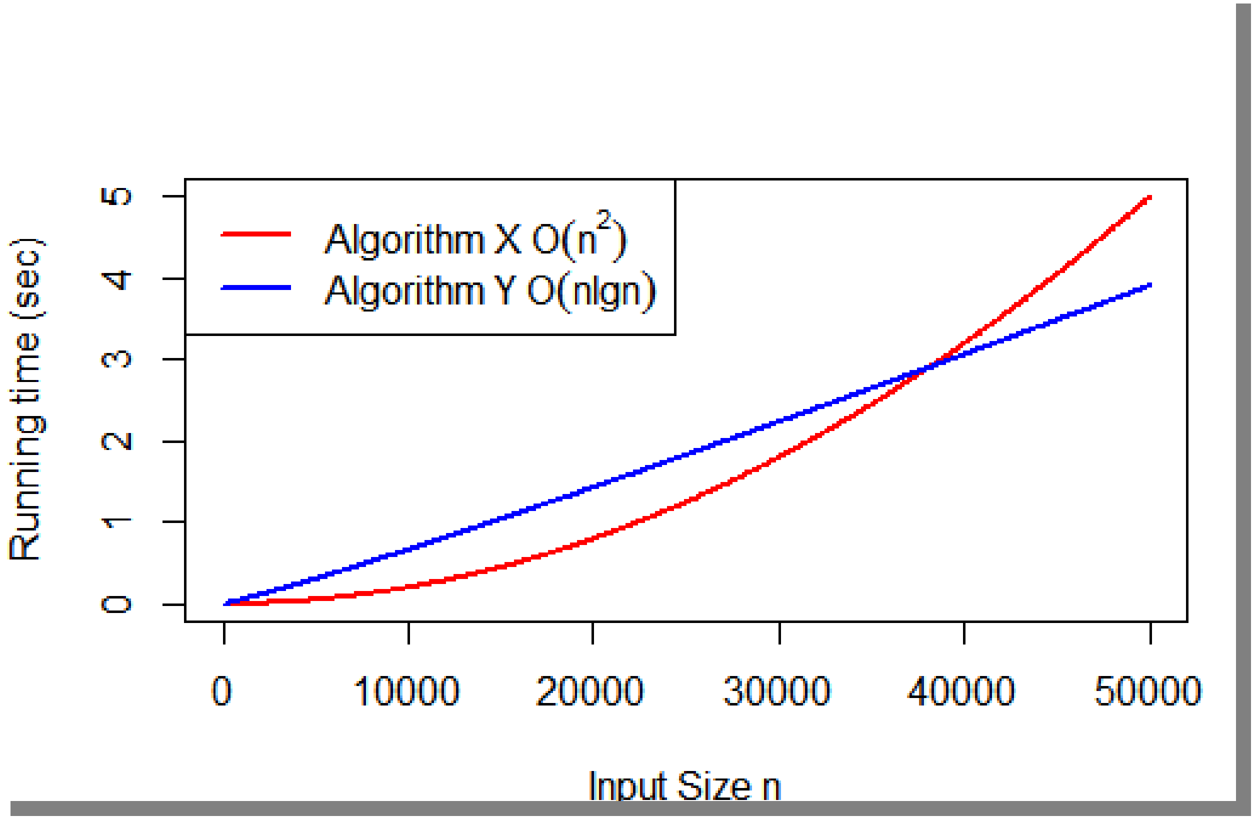
Total $= \Theta(n \lg n)$

Merge Sort Analysis

- What about space?
- Merge Sort does not sort "in place" as it needs temporary space for the "Merge" subroutine
- It needs space: $S(n) = \Theta(n)$

Conclusion

- Insertion Sort:
  - $T(n) = \Theta(n^2)$
  - $S(n) = \Theta(1)$
- Merge Sort:
  - $T(n) = \Theta(n \lg n)$
  - $S(n) = \Theta(n)$



- Insertion sort better for smaller n, merge sort for larger n

Recap

- Insertion Sort
- Merge Sort
- Asymptotic Analysis
- Recurrences
- Next: More on recurrences and asymptotic analysis