Sorting Algorithms

Slides and figures have been collected from various publicly available Internet sources for preparing the lecture slides of IT2001 course. I acknowledge and thank all the original authors for their contribution to prepare the content.

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

Why Study Sorting Algorithms?

- There are a variety of situations that we can encounter
  – Do we have randomly ordered keys?
  – Are all keys distinct?
  – How large is the set of keys to be ordered?
  – Need guaranteed performance?

- Various algorithms are better suited to some of these situations

Some Definitions

- Internal Sort
  - The data to be sorted is all stored in the computer's main memory.
- External Sort
  - Some of the data to be sorted might be stored in some external, slower, device.
- In Place Sort
  - The amount of extra space required to sort the data is constant with the input size.

Stability

- A STABLE sort preserves relative order of records with equal keys

Sorted on first key:

Aaron	4	A	664-480-0023	097 Little
Andrews	3	A	874-088-1212	121 Whitman
Battle	4	C	991-878-4944	308 Blair
Chen	2	A	884-232-5341	11 Dickinson
Fox	1	A	243-456-9091	101 Brown
Furia	3	A	766-093-9873	22 Brown
Gazsi	4	B	665-303-0266	113 Walker
Kanaga	3	B	898-122-9643	343 Forbes
Rohde	3	A	232-343-5555	115 Holder
Quilici	1	C	343-987-5642	32 McCosh

Sort file on second key:

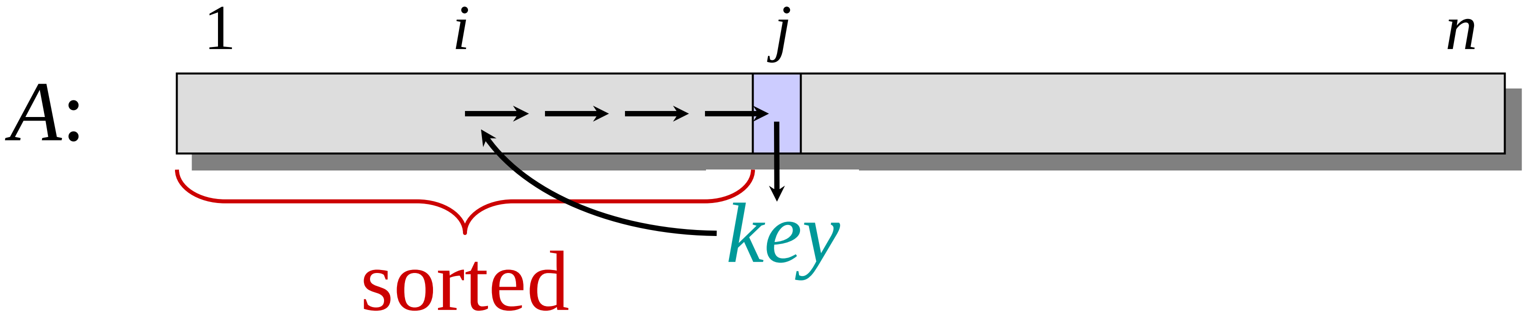
Fox	1	A	243-456-9091	101 Brown
Quilici	1	C	343-987-5642	32 McCosh
Chen	2	A	884-232-5341	11 Dickinson
Kanaga	3	B	898-122-9643	343 Forbes
Andrews	3	A	874-088-1212	121 Whitman
Furia	3	A	766-093-9873	22 Brown
Rohde	3	A	232-343-5555	115 Holder
Battle	4	C	991-878-4944	308 Blair
Gazsi	4	B	665-303-0266	113 Walker
Aaron	4	A	664-480-0023	097 Little

Records with key value 3 are not in order on first key!!

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

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*

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: (NEVER)

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

Analysis of Insertion Sort

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

t_j: # of times the while statement is executed at iteration j

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

Insertion sort analysis

Worst case: Input reverse sorted.

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

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.

Selection Sort

- Idea:
  - Find the smallest element in the array
  - Exchange it with the element in the first position
  - Find the second smallest element and exchange it with the element in the second position
  - Continue until the array is sorted
- Disadvantage:
  - Running time depends only slightly on the amount of order in the file

Example

8	4	6	9	2	3	1

1	4	6	9	2	3	8

1	2	6	9	4	3	8

1	2	3	9	4	6	8

1	2	3	4	9	6	8

1	2	3	4	6	9	8

1	2	3	4	6	8	9

1	2	3	4	6	8	9

Selection Sort

Alg.: SELECTION-SORT(*A*)

```
n ← length[A]
for j ← 1 to n - 1
    do smallest ← j
        for i ← j + 1 to n
            do if A[i] < A[smallest]
                then smallest ← i
        exchange A[j] ↔ A[smallest]
```

8	4	6	9	2	3	1

Analysis of Selection Sort

Alg.: SELECTION-SORT(*A*)

	cost	times
n ← length[A]	c_1	1
for j ← 1 to n - 1	c_2	n
do smallest ← j	c_3	n-1
for i ← j + 1 to n	c_4	$\sum_{j=1}^{n-1}(n-j+1)$
do if A[i] < A[smallest]	c_5	$\sum_{j=1}^{n-1}(n-j)$
then smallest ← i	c_6	$\sum_{j=1}^{n-1}(n-j)$
exchange A[j] ↔ A[smallest]	c_7	n-1

≈$n^2/2$ comparisons

≈n exchanges

$$T(n)=c_1+c_2n+c_3(n-1)+c_4\sum_{j=1}^{n-1}(n-j+1)+c_5\sum_{j=1}^{n-1}(n-j)+c_6\sum_{j=2}^{n-1}(n-j)+c_7(n-1)=\Theta(n^2)$$

Divide and Conquer
(Merge Sort)

Divide and Conquer

- Recursive in structure
  - ***Divide*** the problem into sub-problems that are similar to the original but smaller in size
  - ***Conquer*** the sub-problems by solving them recursively. If they are small enough, just solve them in a straight forward manner.
  - ***Combine*** the solutions to create a solution to the original problem

An Example: Merge Sort

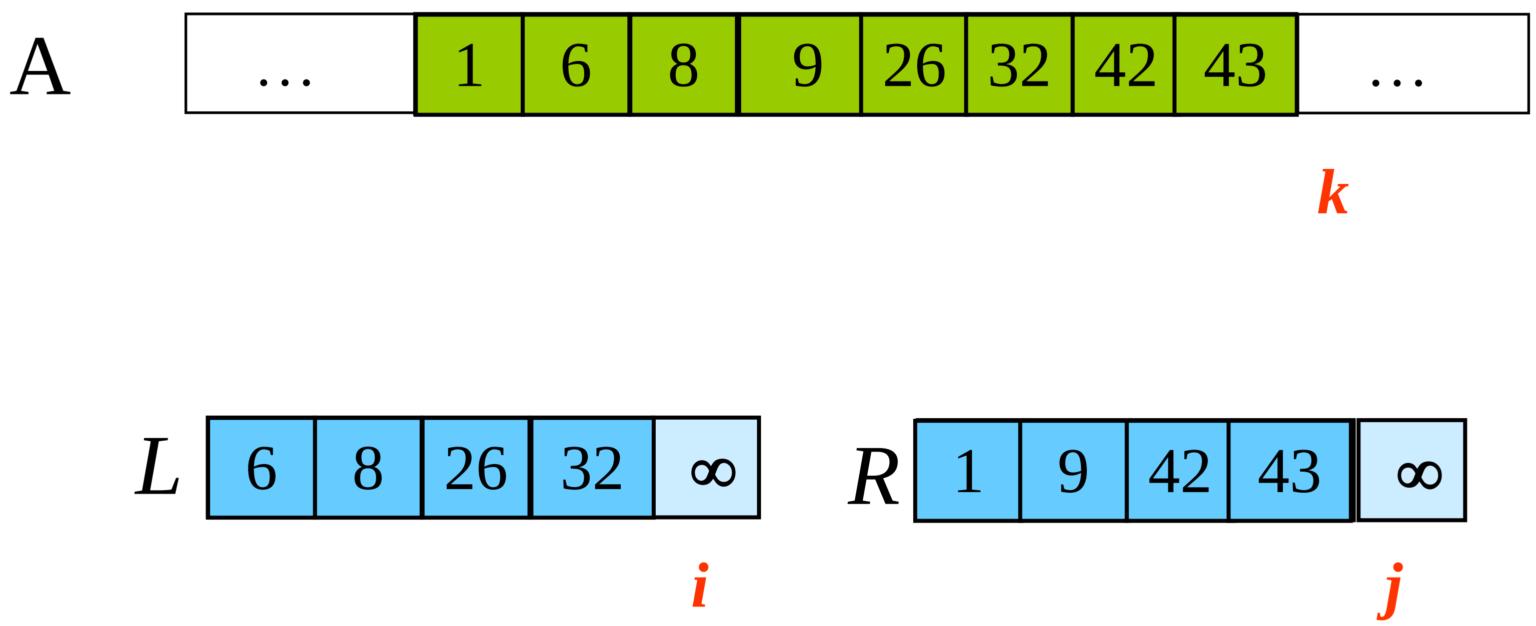
***<u>Sorting Problem</u>*:** Sort a sequence of *n* elements into non-decreasing order.

- ***Divide*:** Divide the *n*-element sequence to be sorted into two subsequences of *n/2* elements each

- ***Conquer:*** Sort the two subsequences recursively using merge sort.

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

Merge – Example



Procedure Merge

```
Merge(A, p, q, r)
n1 ← q – p + 1
n2 ← r – q
    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] ← ∞
    R[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
```

Input: Array containing sorted subarrays $A[p..q]$ and $A[q+1..r]$.

Output: Merged sorted subarray in $A[p..r]$.

Sentinels, to avoid having to check if either subarray is fully copied at each step.

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

Merge-Sort (A, p, r)

INPUT: **a sequence of *n* numbers stored in array A**

OUTPUT: **an ordered sequence of *n* numbers**

MergeSort **(A, p, r)** // sort $A[p..r]$ by divide & conquer

if $p < r$ // If $n = 1$, done

  then $q \leftarrow \lfloor (p+r)/2 \rfloor$

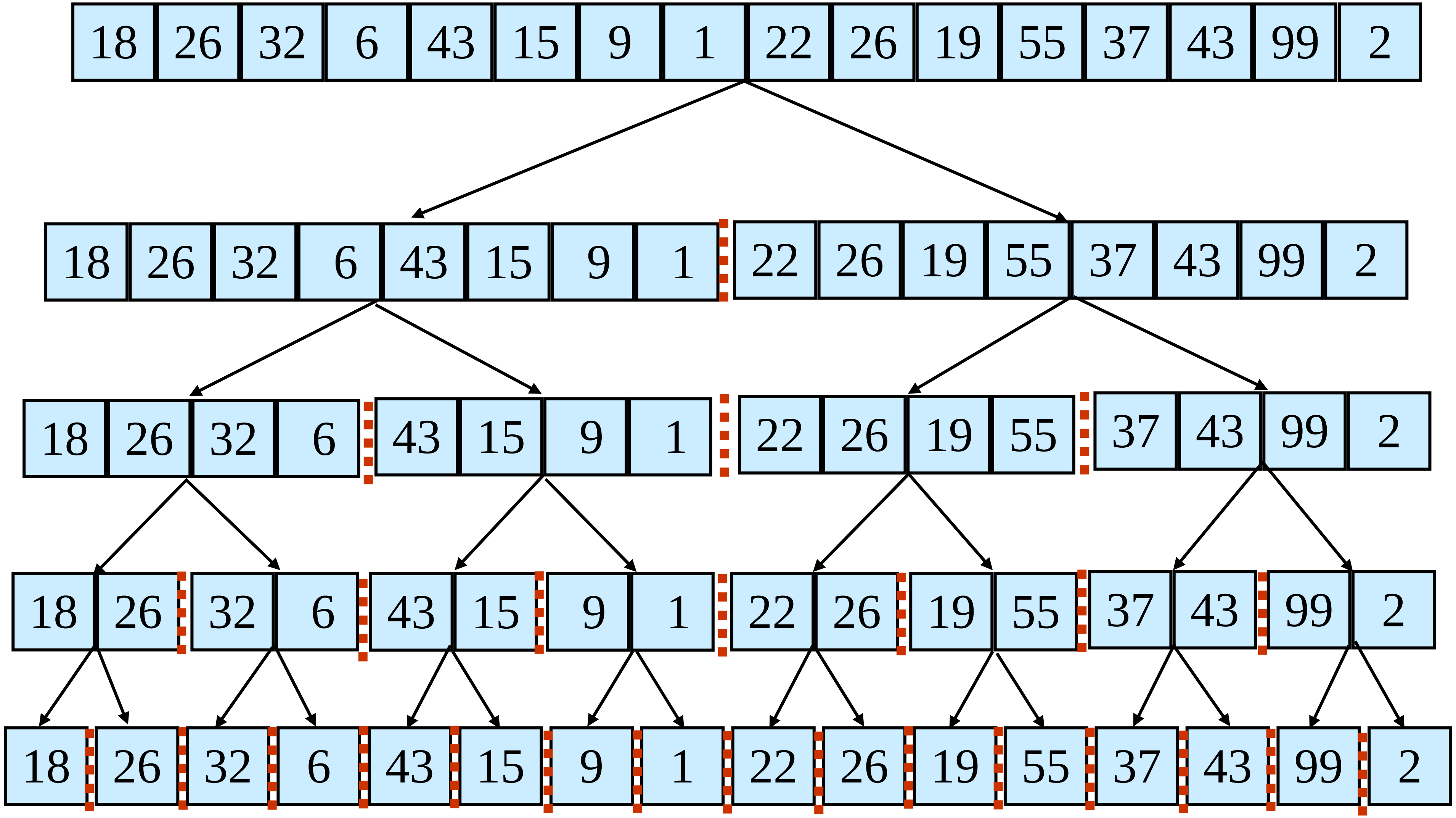
    MergeSort (A, p, q)

    MergeSort (A, q+1, r)

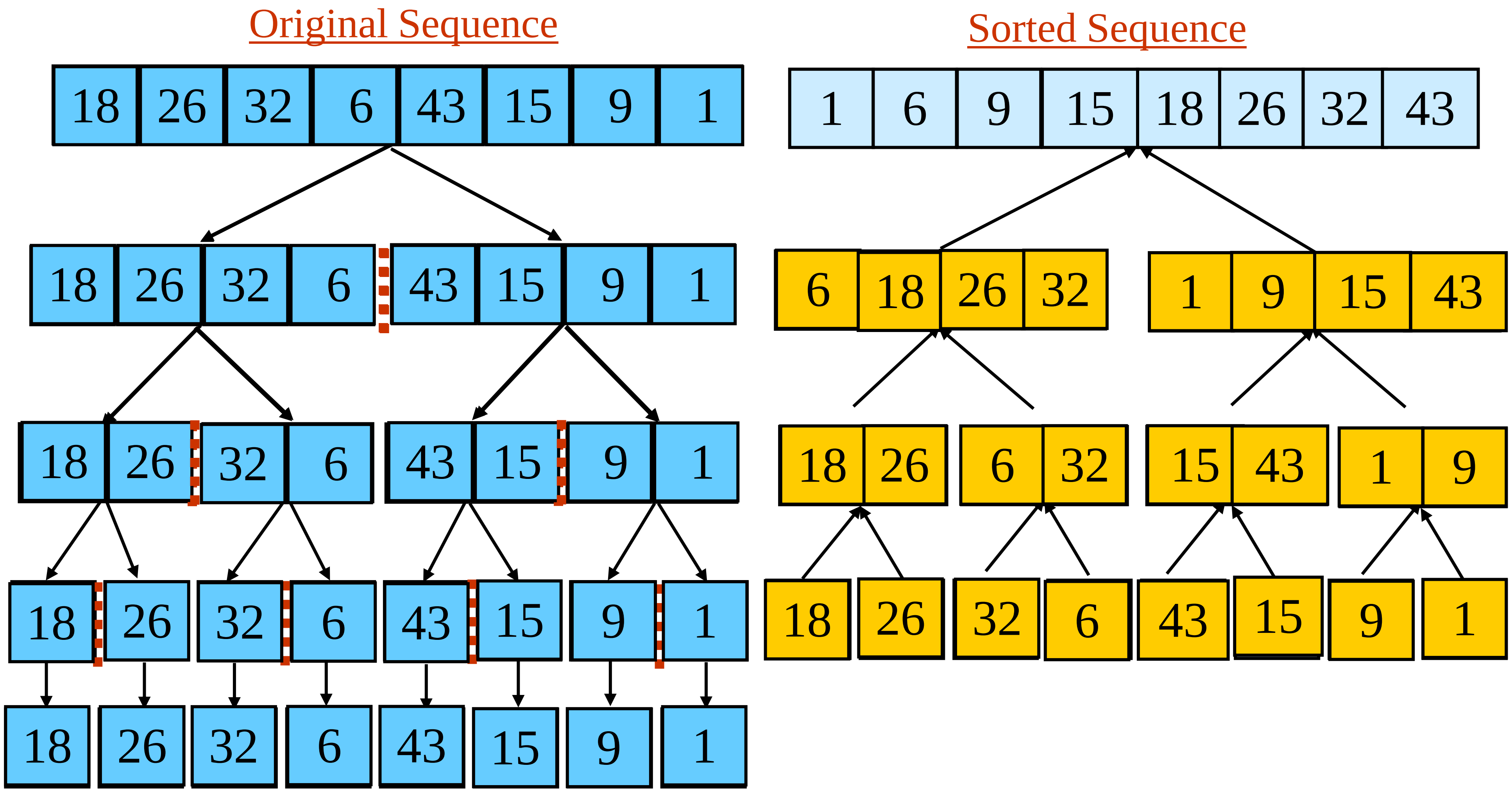
    Merge (A, p, q, r) // merges $A[p..q]$ with $A[q+1..r]$

Initial Call: MergeSort(A, 1, n)

Merge Sort – Example



Merge Sort – Example



Analysis of Merge Sort

- Running time $\boldsymbol{T(n)}$ of Merge Sort:
- Divide: computing the middle takes $\Theta(1)$
- Conquer: solving 2 subproblems takes $2T(n/2)$
- Combine: merging n elements takes $\Theta(n)$
- Total:

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

$\Rightarrow T(n) = \Theta(n \lg n)$ (CLRS book, Chapter 4)

Conclusions

- $\Theta(n \lg n)$ 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 $n > 30$ or so.

Comparison Sorting

Sort	Worst Case	Average Case	Best Case	Comments
Selection Sort	$\Theta(N^2)$	$\Theta(N^2)$	$\Theta(N^2)$	Fast for small N
Insertion Sort	$\Theta(N^2)$	$\Theta(N^2)$	$\Theta(N)$	Fast for small N
Merge Sort	$\Theta(N \lg N)$	$\Theta(N \lg N)$	$\Theta(N \lg N)$	Requires memory
Heap Sort	$\Theta(N \lg N)$	$\Theta(N \lg N)$	$\Theta(N \lg N)$	Large constants
Quick Sort	$\Theta(N^2)$	$\Theta(N \lg N)$	$\Theta(N \lg N)$	Small constants

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.

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.

$$T(n)$$

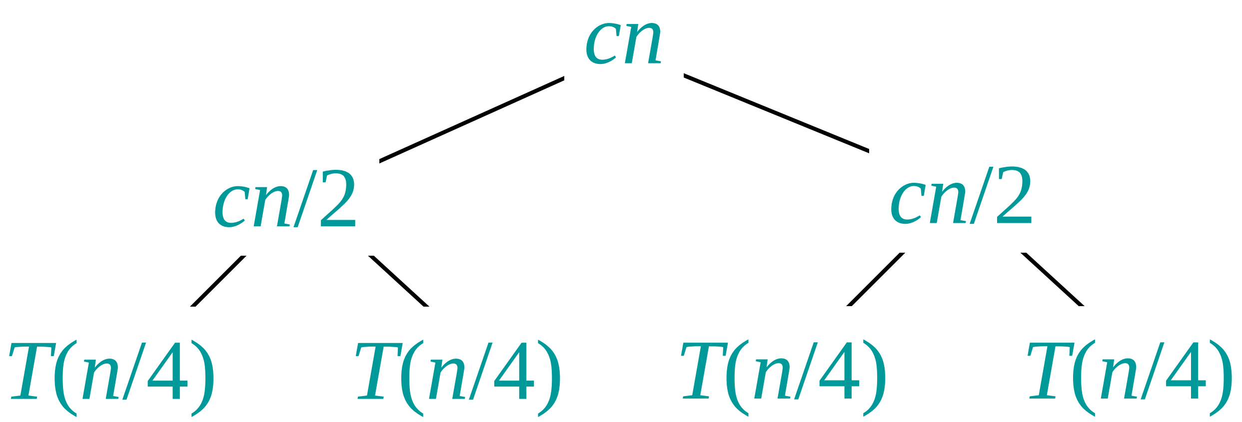
Recursion tree

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

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

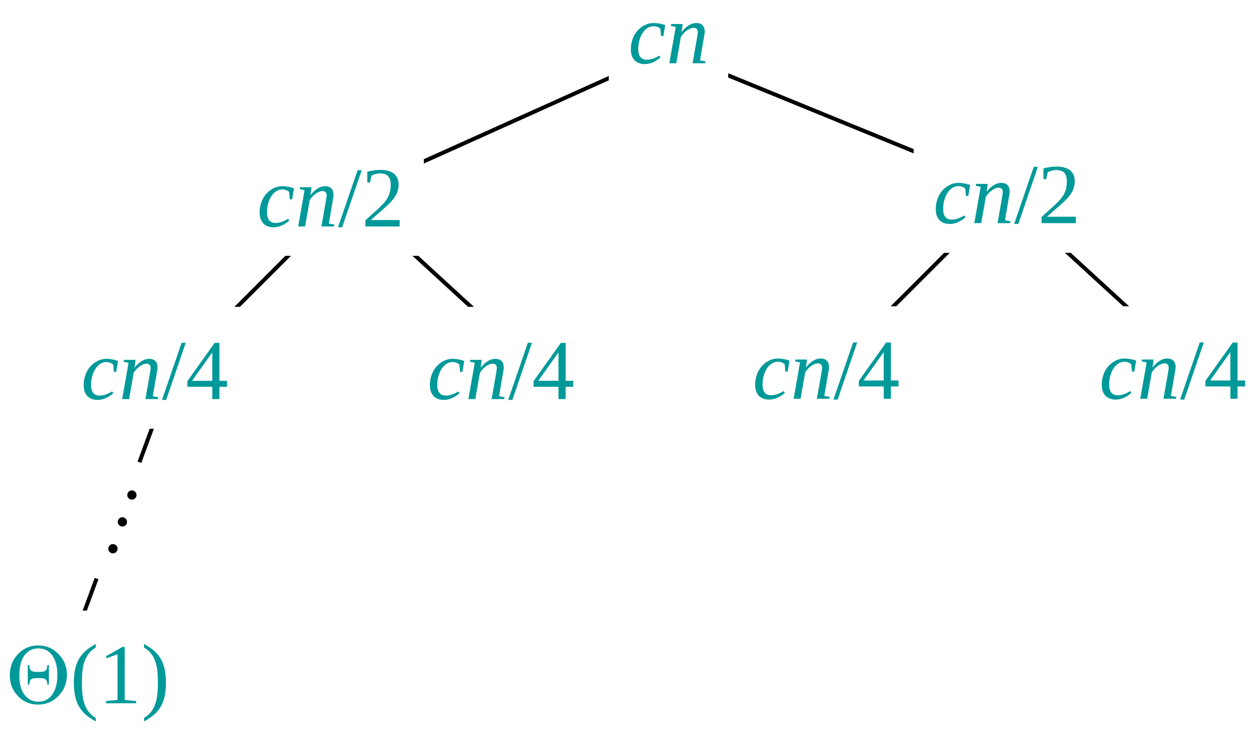
Recursion tree

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



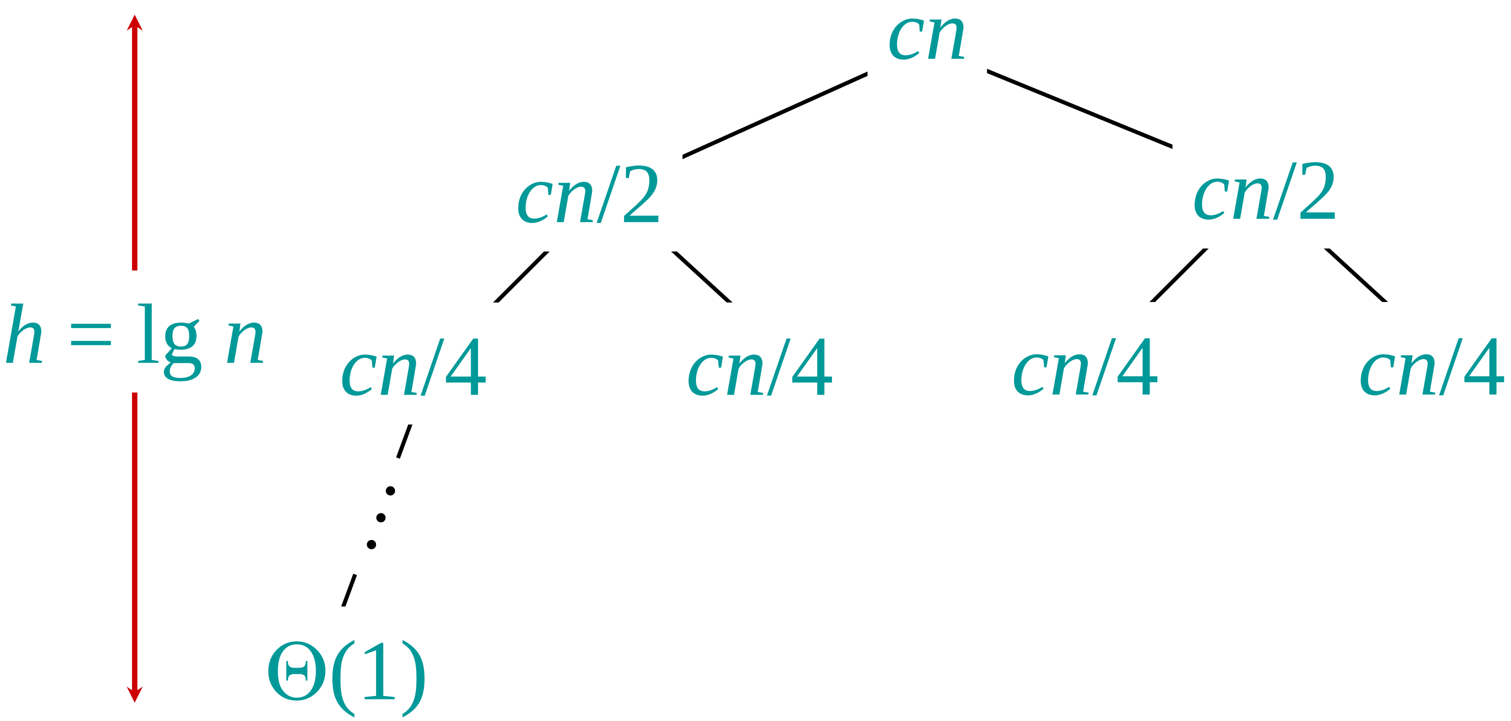
Recursion tree

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



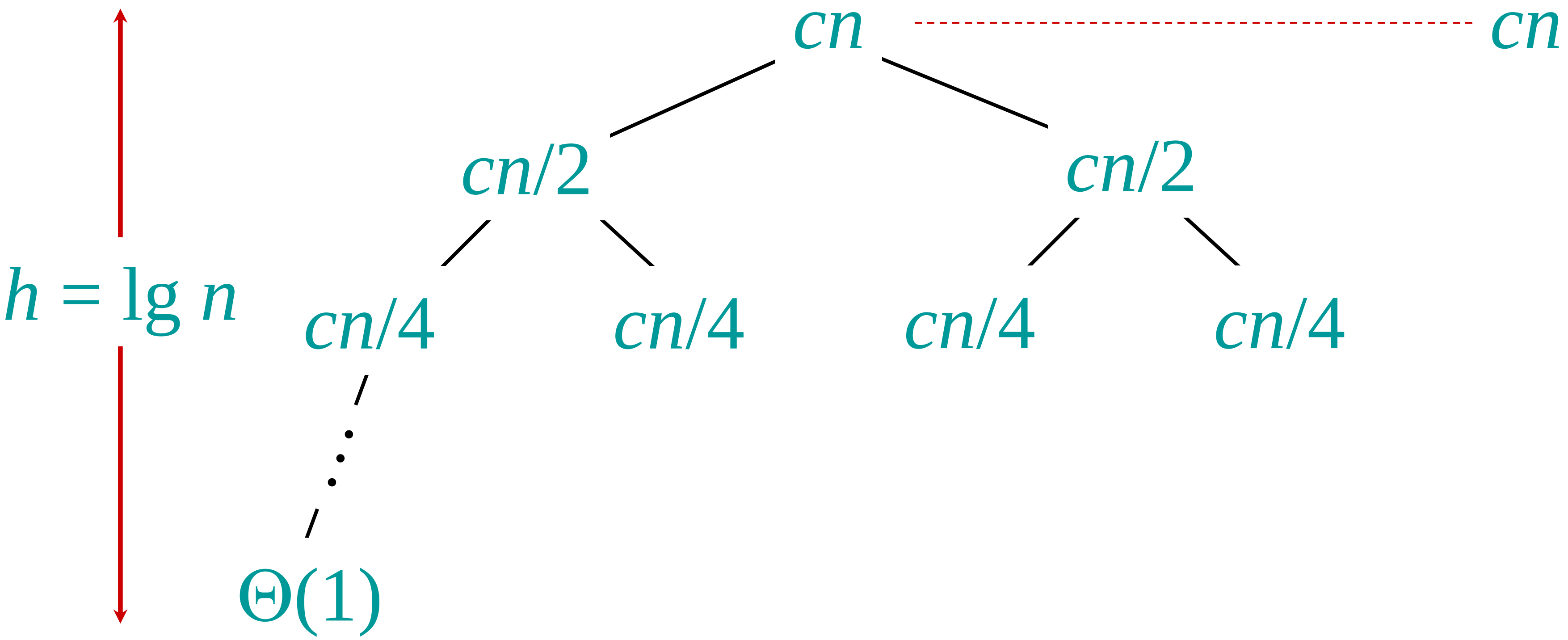
Recursion tree

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



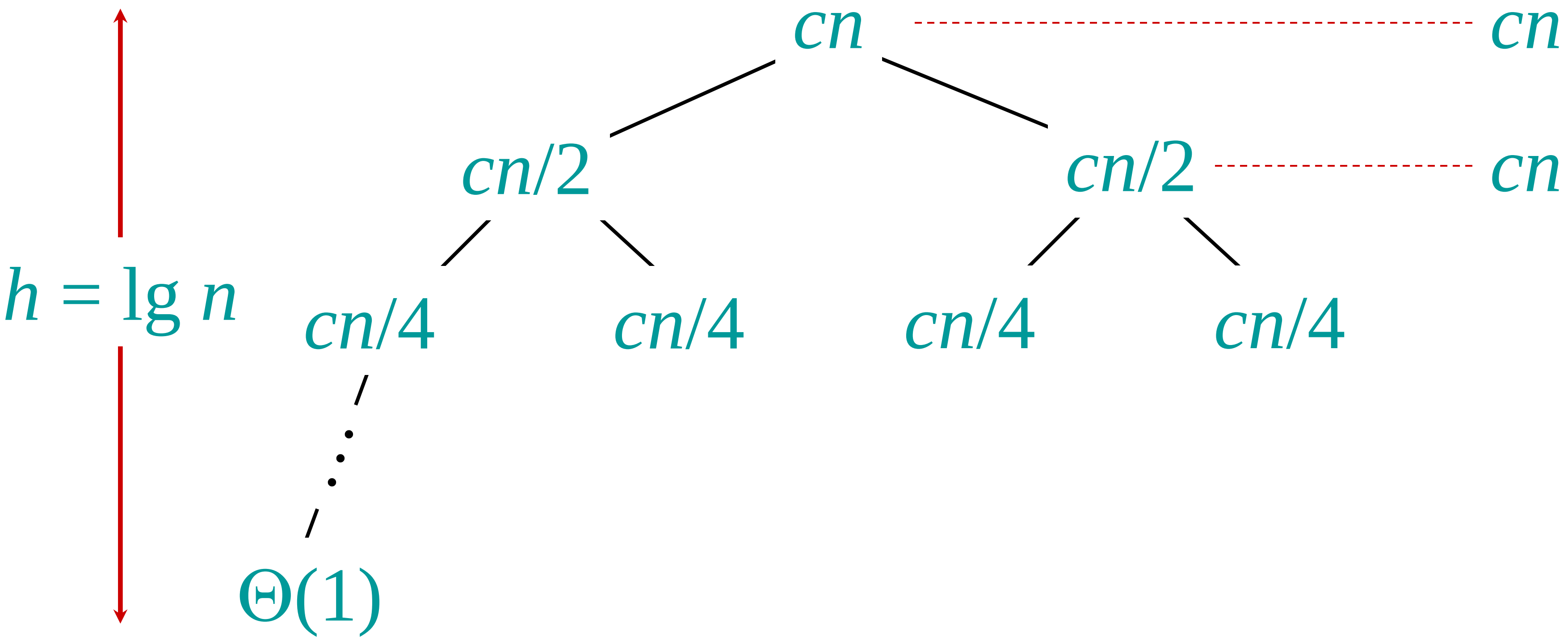
Recursion tree

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



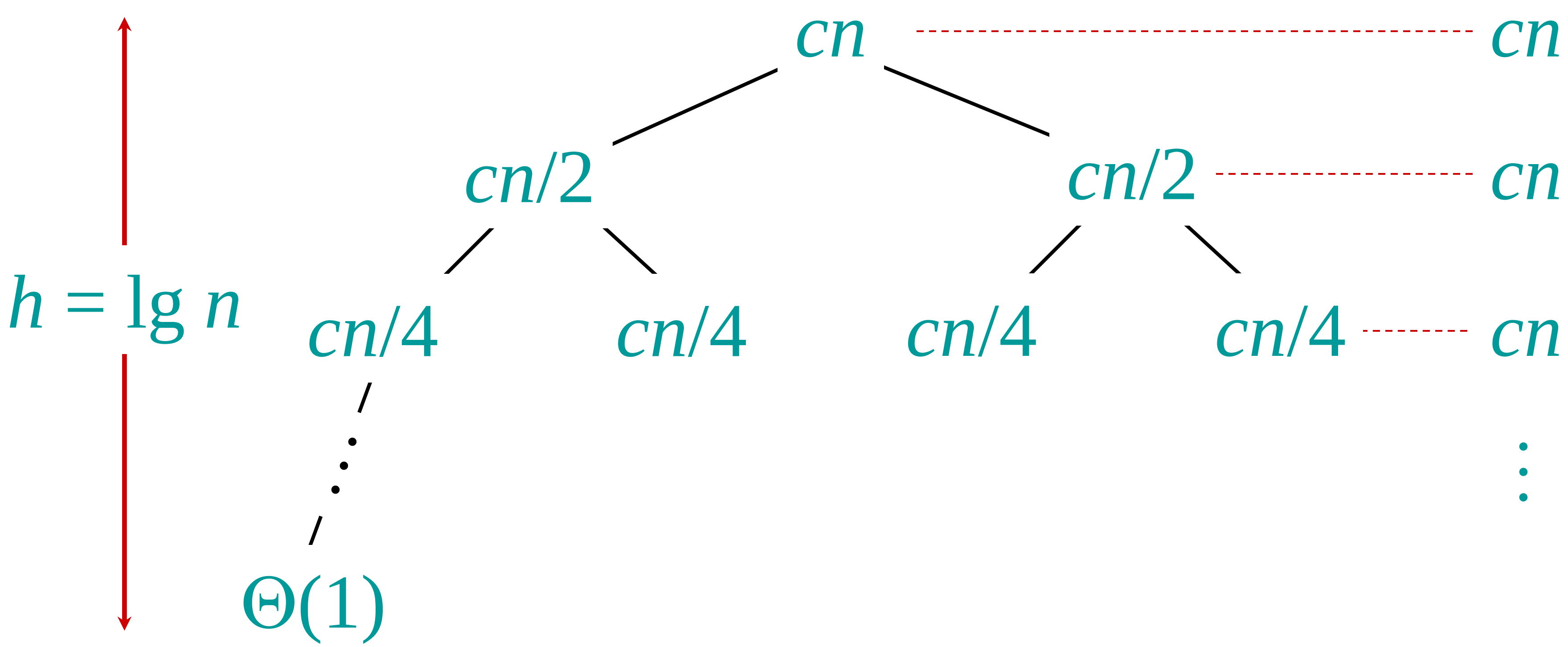
Recursion tree

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



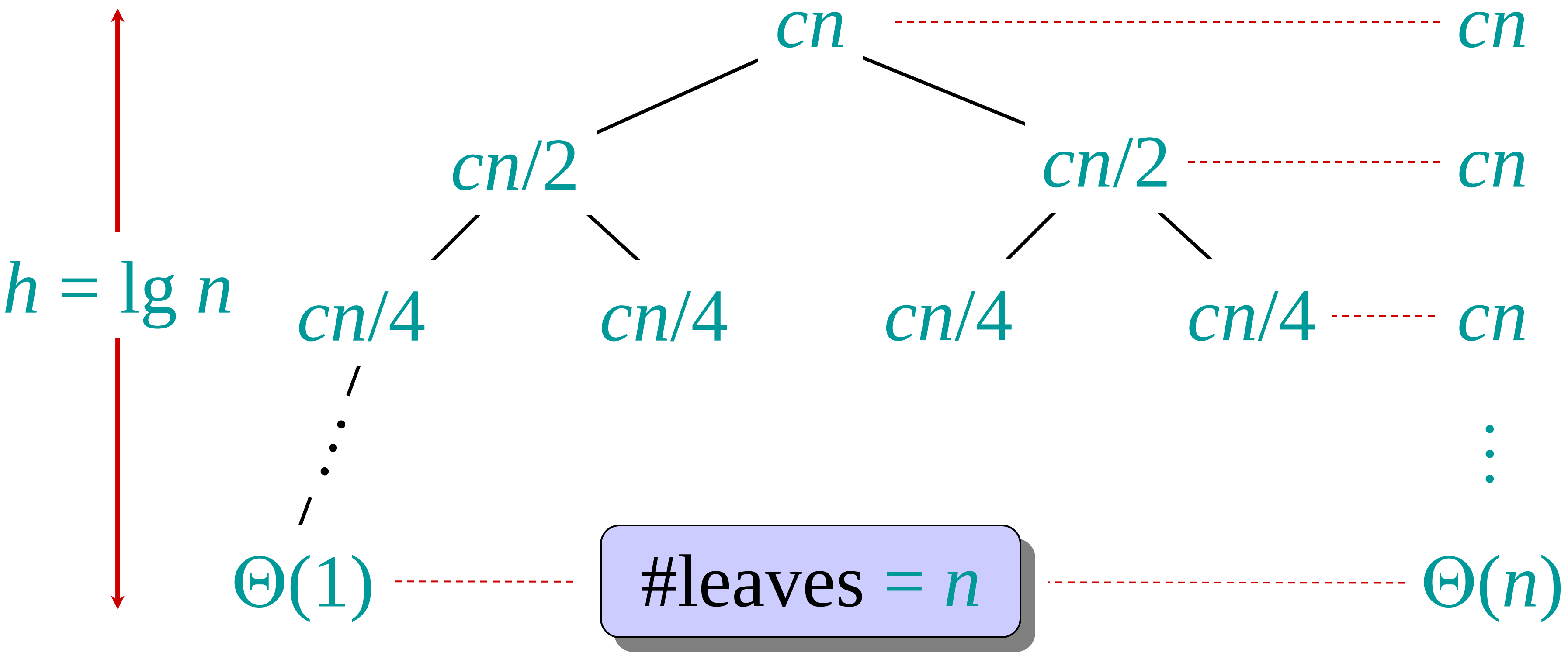
Recursion tree

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



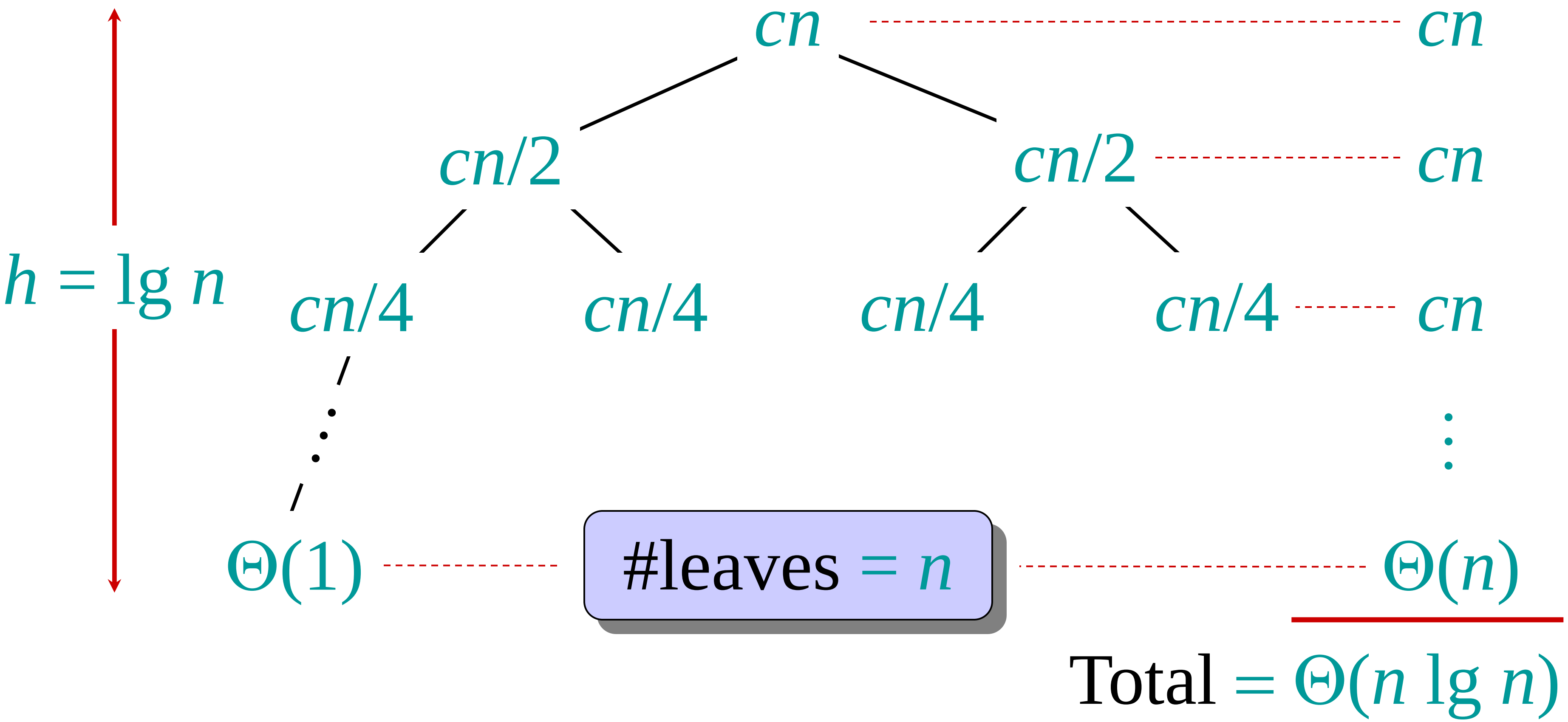
Recursion tree

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.



References

- http://www.cse.unr.edu/~bebis/
- http://courses.csail.mit.edu/6.046/spring04/lectures