CS2040S
Data Structures and Algorithms

Welcome!

Sorting, Part I

Sorting algorithms

- BubbleSort
- SelectionSort
- InsertionSort
- MergeSort

Properties

- Running time
- Space usage
- Stability

Admin

Tutorials and Recitations start this week

- Find materials on Coursemology
- Find links on Coursemology
- Do work on the questions before class

Chinese New Year

Next week

- Happy new year!
- University holiday: Feb 1/2 (Tuesday, Wednesday)
- Monday: class as usual
- Wednesday: no class
- Tutorials: rescheduled (talk to tutor)

Admin

Covid Issues

- If you test positive and are well, can attend a Zoom session. (And can swap to a Zoom session.)
- If you are unwell, please rest and recover!
- For F2F, must take FET/ART (and have green pass) as per NUS rules. And must use NUS attendance system.
- Please do not attend F2F if positive or *any* symptoms.

Let your tutor know if you cannot make it, if you are missing lecture, etc. They will handle any issues.

Admin

For tutors and TAs:

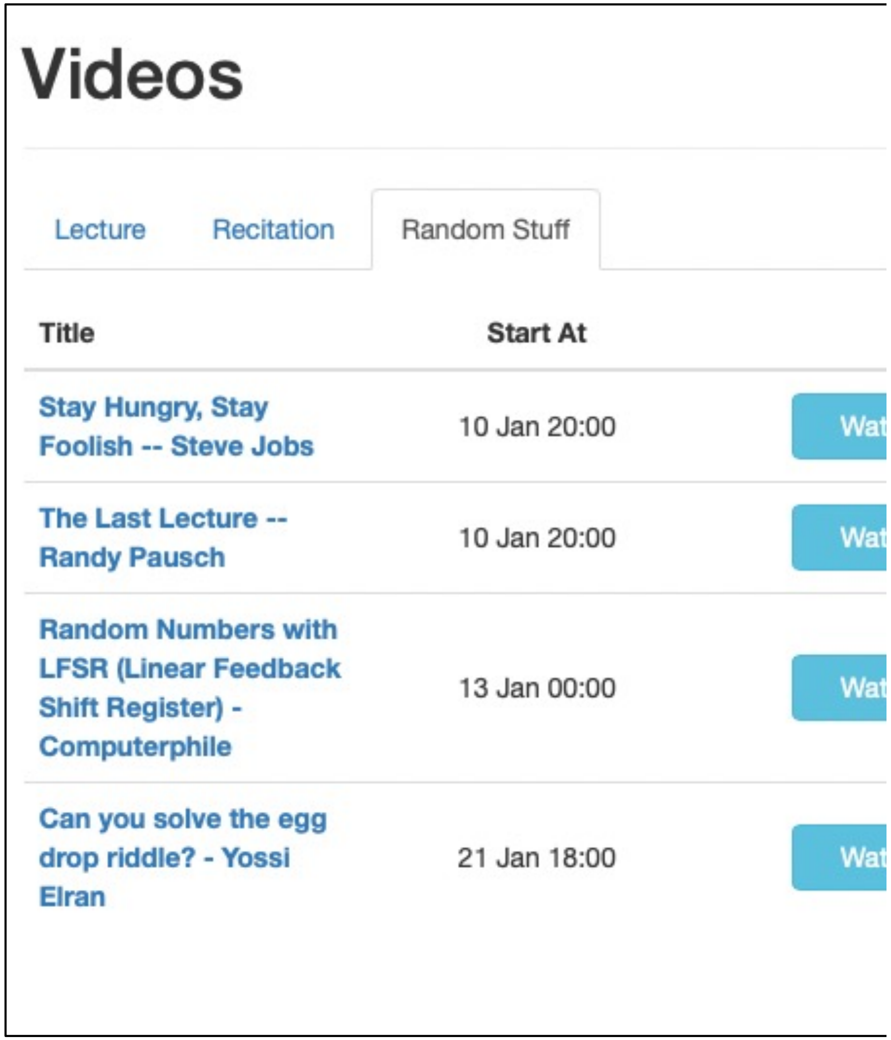
- If they test positive, we will adapt accordingly.
- Already, we will have one replacement TA this week…

Admin

Video of the Week

- Random video posted each week
- Selected by the tutor team as something “fun”
- Sometimes related to class, sometimes a little bit different
- Not just another lecture…

(Nominate videos to your tutor!)



Videos

Lecture | Recitation | Random Stuff

Title	Start At	
Stay Hungry, Stay Foolish -- Steve Jobs	10 Jan 20:00	Wat
The Last Lecture -- Randy Pausch	10 Jan 20:00	Wat
Random Numbers with LFSR (Linear Feedback Shift Register) - Computerphile	13 Jan 00:00	Wat
Can you solve the egg drop riddle? - Yossi Elran	21 Jan 18:00	Wat

Sorting

Problem definition:

Input: array A[1..n] of words / numbers

Output: array B[1..n] that is a permutation of A such that:

$B[1] \leq B[2] \leq \ldots \leq B[n]$

Example:

$A = [9, 3, 6, 6, 6, 4] \rightarrow [3, 4, 6, 6, 6, 9]$

MergeSort

Divide-and-Conquer

1. Divide problem into smaller sub-problems.
2. Recursively solve sub-problems.
3. Combine solutions.

MergeSort

Divide-and-Conquer Sorting

1. Divide: split array into two halves.
2. Recurse: sort the two halves.
3. Combine: merge the two sorted halves.

MergeSort

Divide-and-Conquer Sorting

1. Divide: split array into two halves.

2. Recurse: sort the two halves.

3. Combine: merge the two sorted halves.

Advice:

When thinking about recursion, do not “unroll” the recursion.
Treat the recursive call as a magic black box.
(But don’t forget the base case.)

MergeSort

Step 1:
Divide array into two pieces.

MergeSort(A, n)

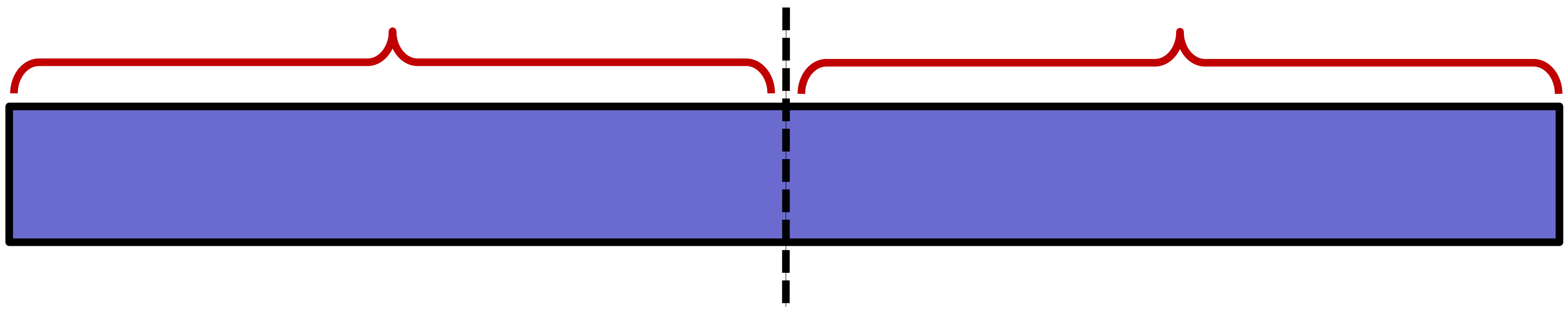
  if (n=1) **then return;**

  else:

    X ←MergeSort(A[1..n/2], n/2);

    Y ←MergeSort(A[n/2+1, n], n/2);

  return Merge (X,Y, n/2);



MergeSort

Step 2:
Recursively sort the two halves.

MergeSort(A, n)

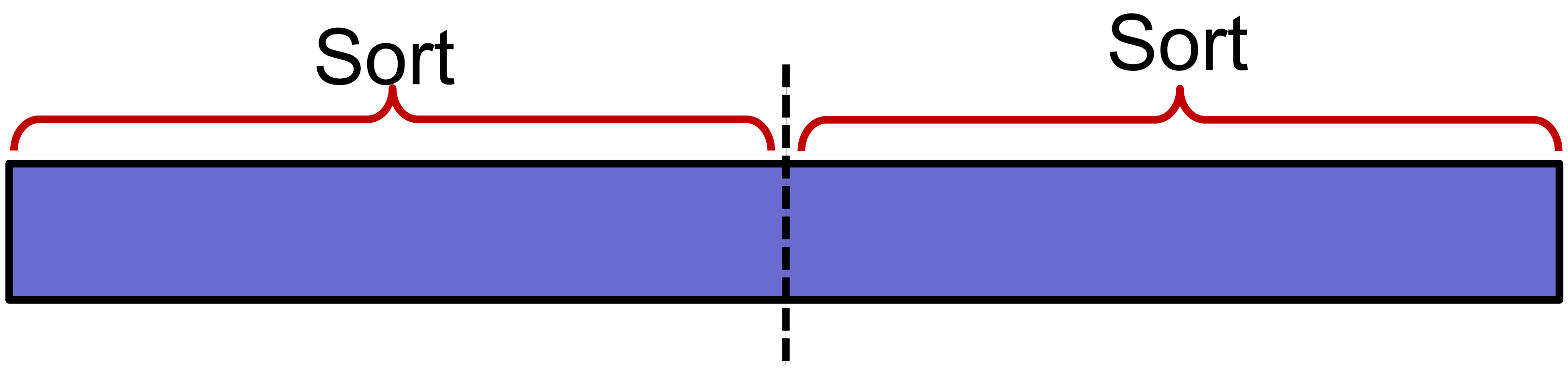
if (n=1) **then return;**

else:

X ←MergeSort**(**A[1..n/2], n/2**);**

Y ←MergeSort**(**A[n/2+1, n], n/2**);**

return Merge **(**X,Y, n/2**);**



MergeSort

Step 3:
Merge the two halves into one sorted array.

MergeSort(A, n)

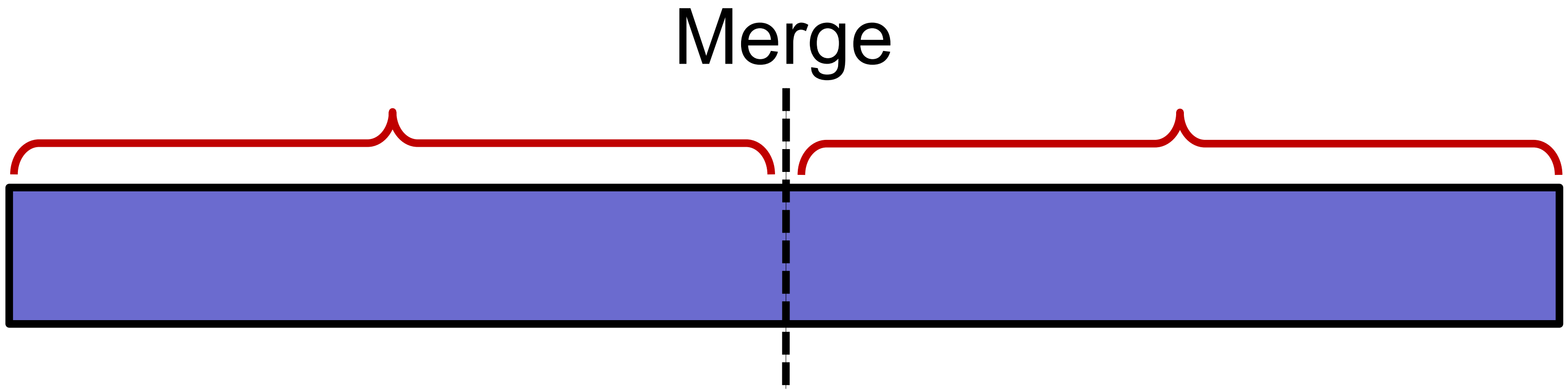
if (n=1) **then return;**

else:

X ←MergeSort**(**A[1..n/2], n/2**);**

Y ←MergeSort**(**A[n/2+1, n], n/2**);**

return Merge **(**X,Y, n/2**);**



MergeSort

Base case

MergeSort(A, n)

if (n=1) **then return;**

else:

X ←MergeSort**(**A[1..n/2], n/2**);**

Y ←MergeSort**(**A[n/2+1, n], n/2**);**

return Merge **(**X,Y, n/2**);**

Recursive “conquer” step

Combine solutions

The only “interesting” part is merging!

MergeSort

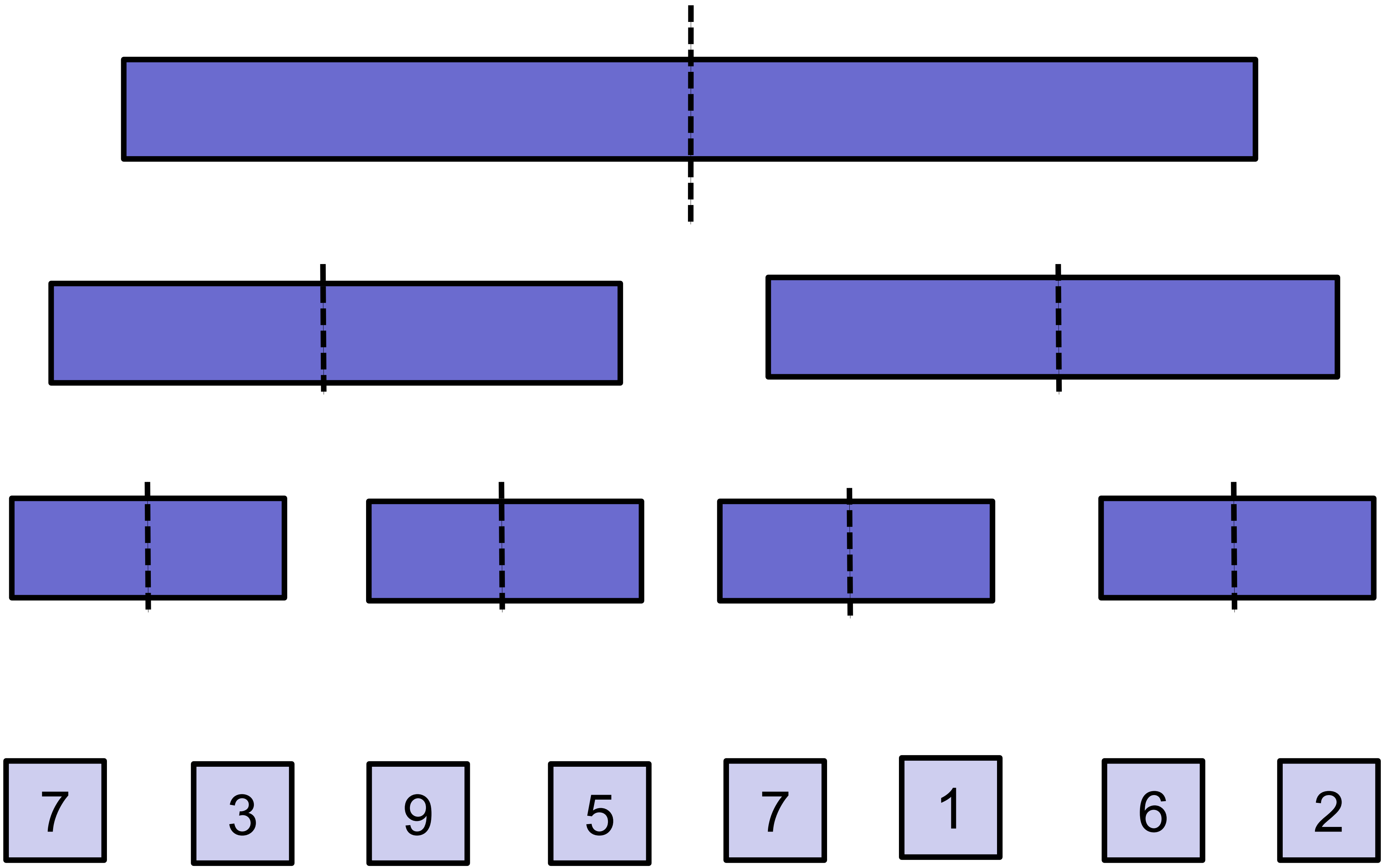
Divide-and-Conquer Sorting

1. Divide: split array into two halves.
2. Recurse: sort the two halves.
3. Combine: merge the two sorted halves.

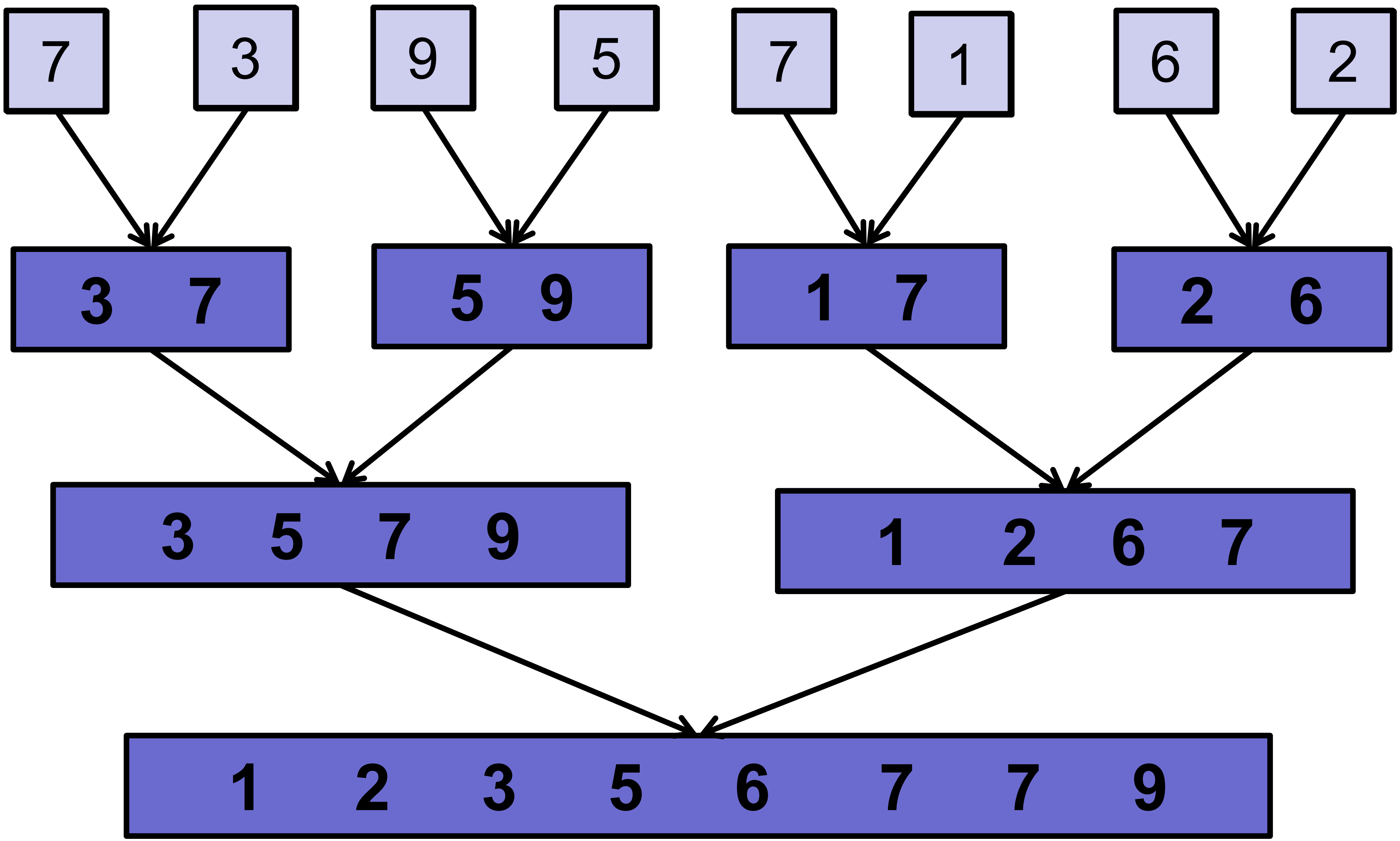
<u>Advice:</u>

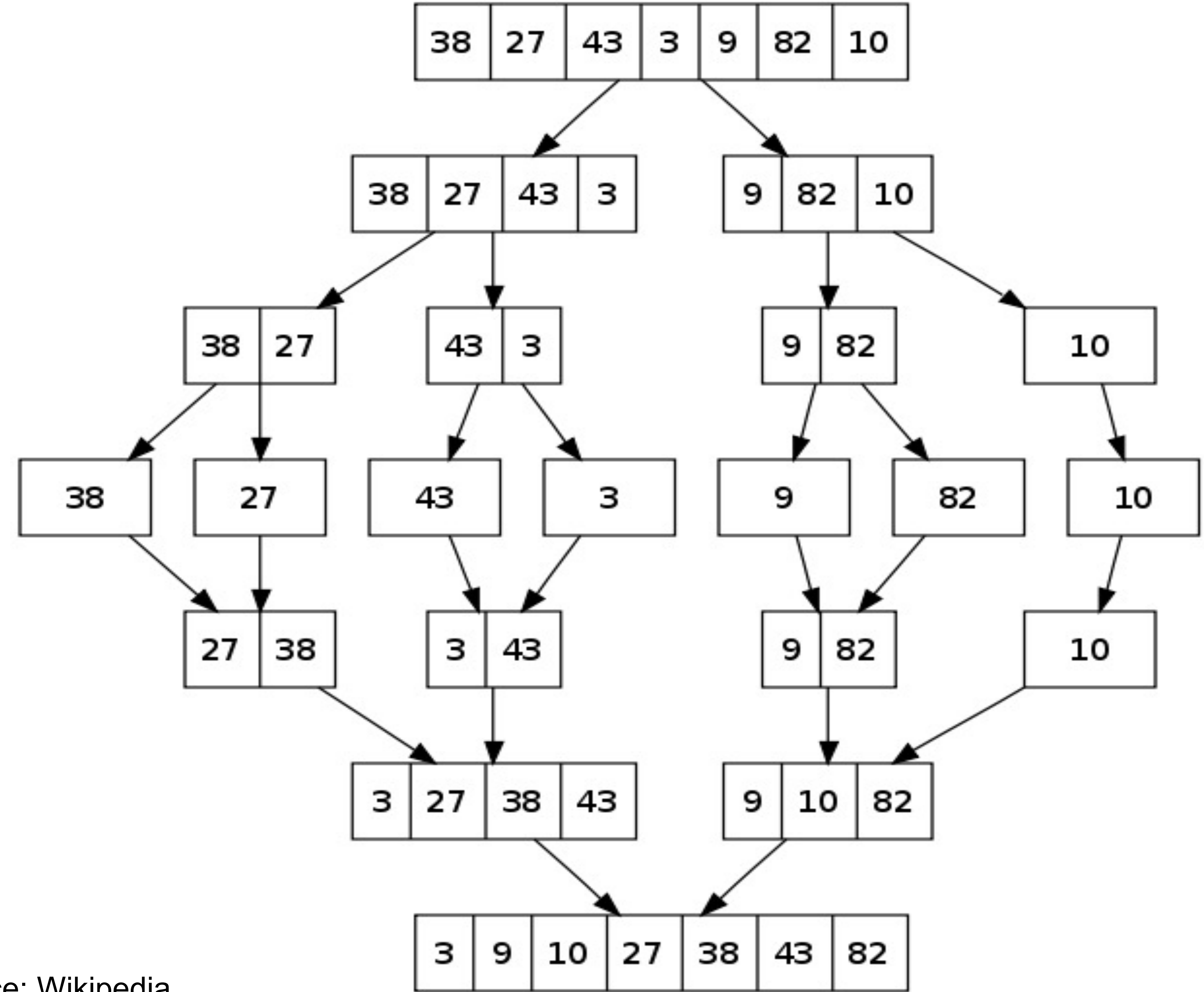
When thinking about recursion, do not “unroll” the recursion.
Treat the recursive call as a magic black box.
(But don’t forget the base case.)

Divide-and-Conquer



Merging





Source: Wikipedia

Merging Two Sorted Lists

Key subroutine: Merge

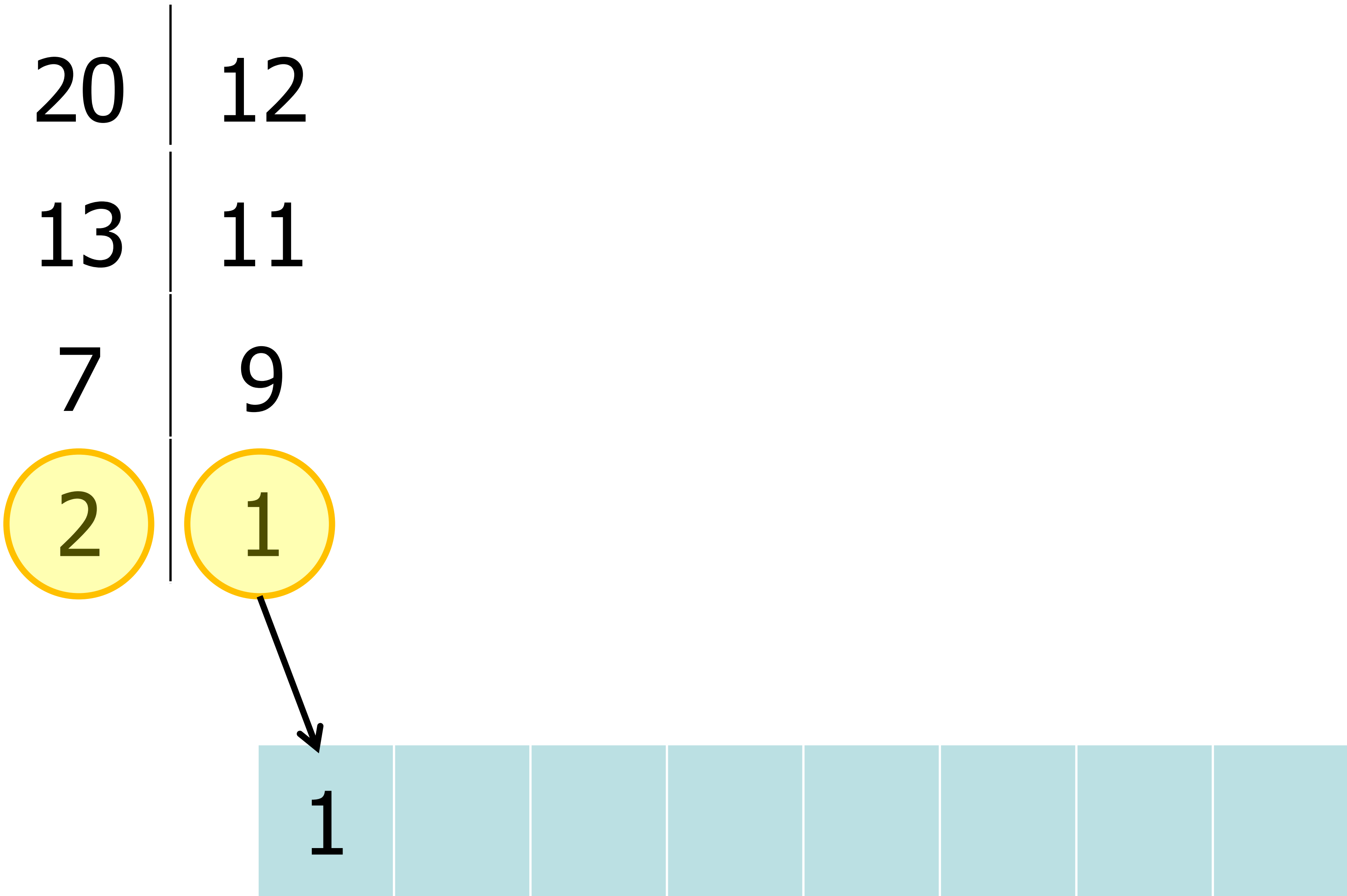
- How to merge?
- How fast can we merge?

Merging Two Sorted Lists

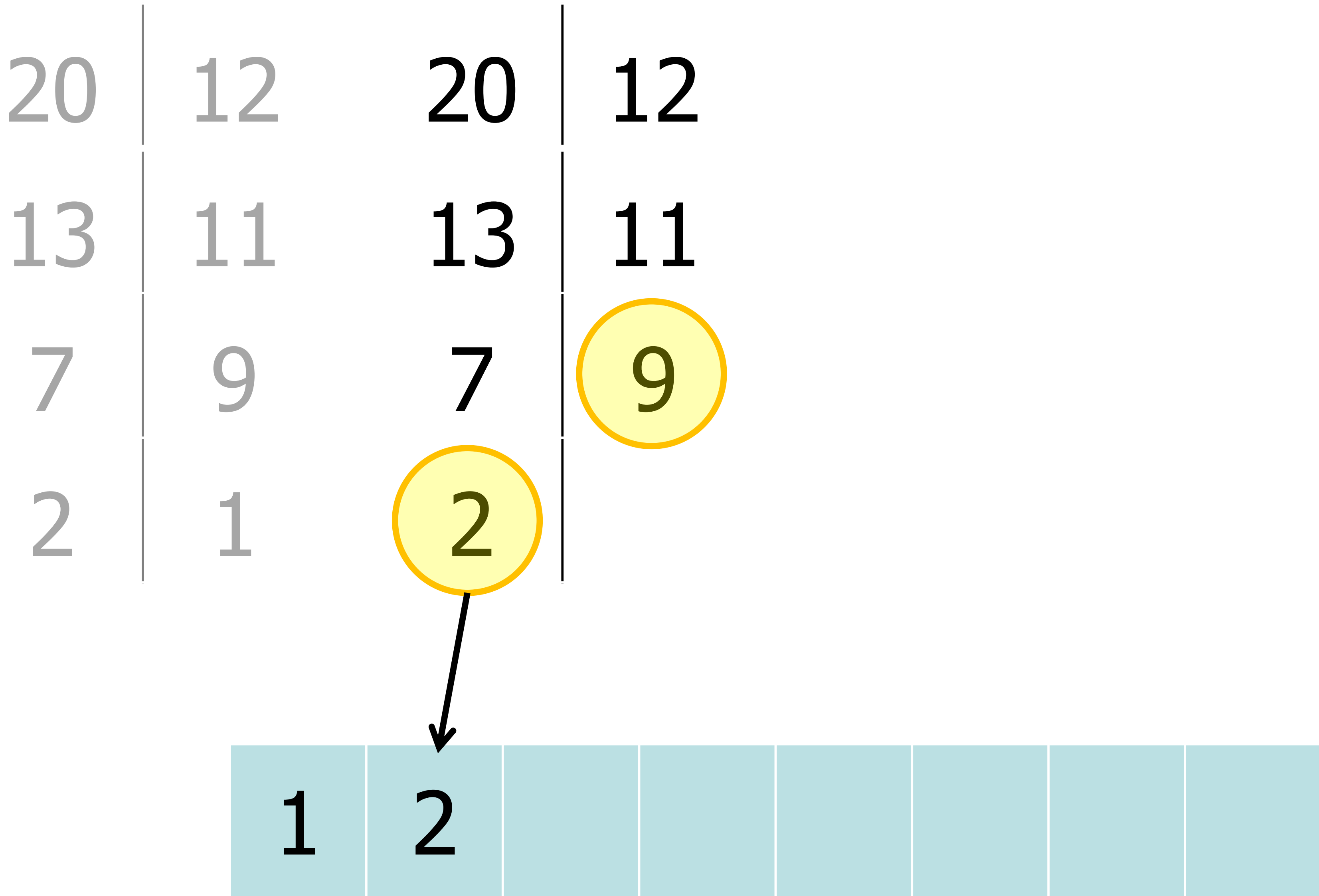
20	12
13	11
7	9
2	1

sorted
from
smallest
to
biggest

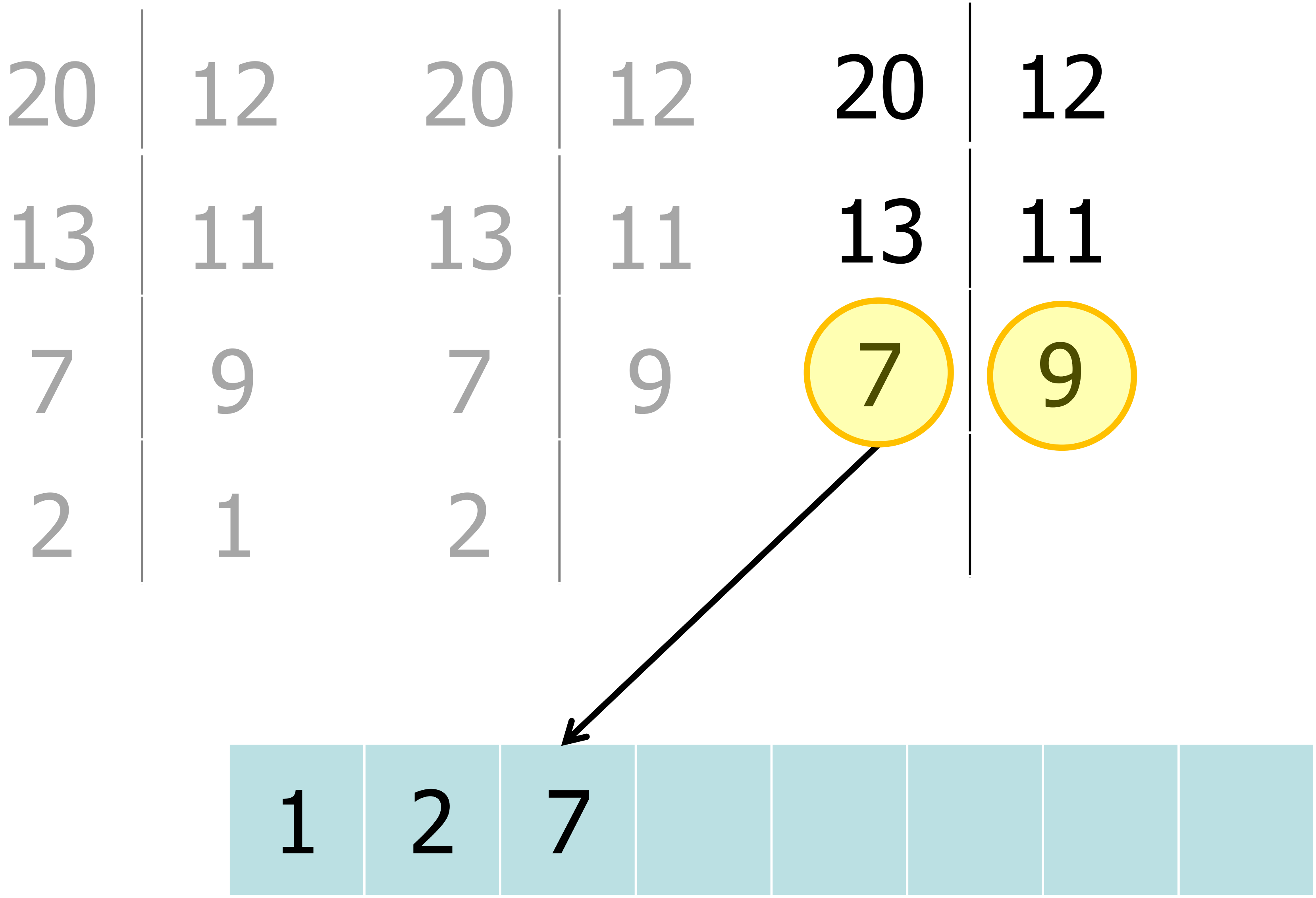
Merging Two Sorted Lists



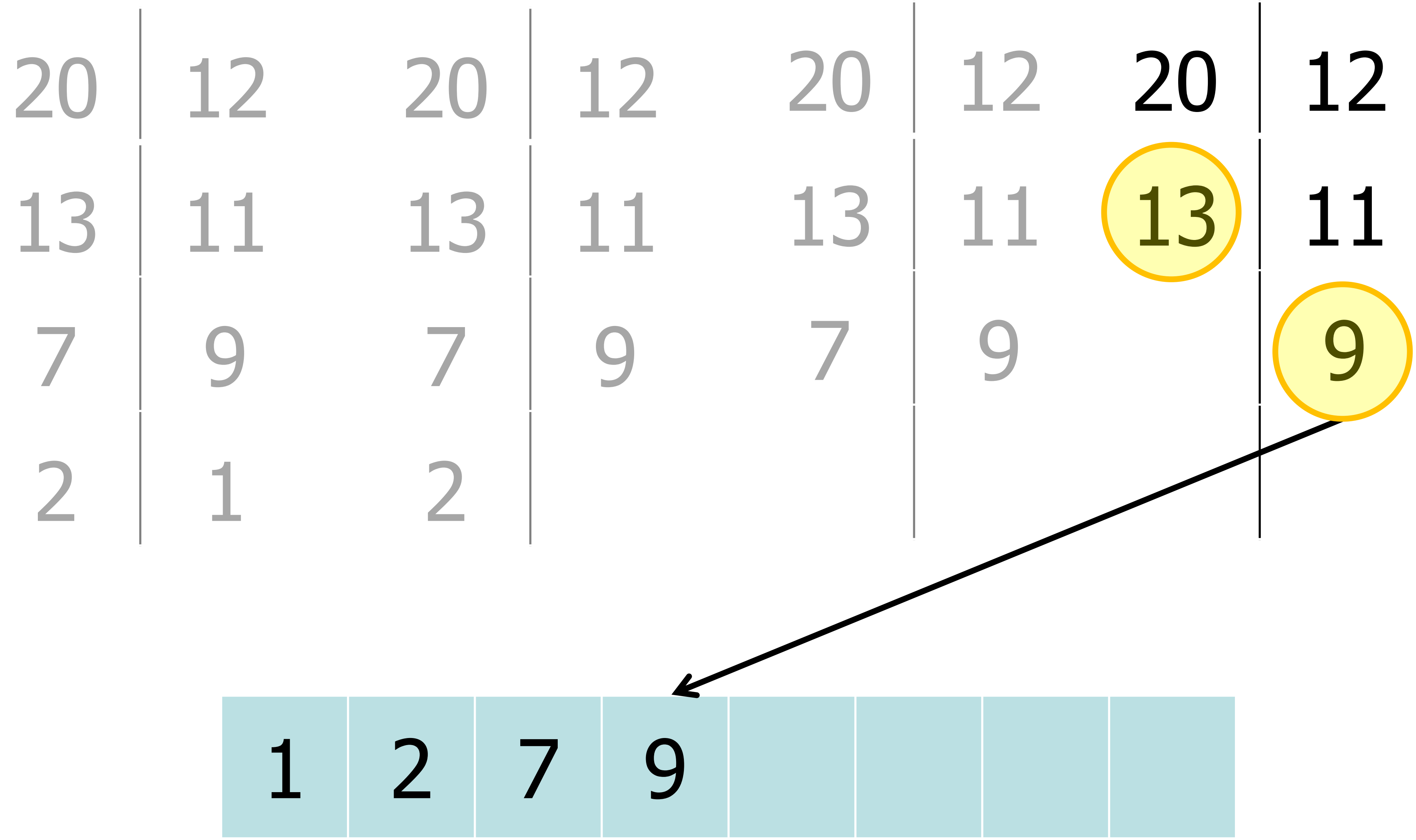
Merging Two Sorted Lists



Merging Two Sorted Lists



Merging Two Sorted Lists



Merging Two Sorted Lists

20	12
13	11
7	9
2	1

20	12
13	11
7	9
2	

20	12
13	11
7	9

20	12
13	11

1	2	7	9	11	12	13	20

Merge: Running Time

Given two lists:

- A of size n/2
- B of size n/2

Total running time: ??

ARCHIPELAGO

is open

Merge: Running Time

Given two lists:

- A of size n/2
- B of size n/2

Total running time: $O(n) = cn$

- In each iteration, move *one* element to final list.
- After n iterations, all the items are in the final list.
- Each iteration takes $O(1)$ time to compare two elements and copy one.

Merge-Sort Analysis

Let $T(n)$ be the worst-case running time for an array of n elements.

MergeSort(A, n)

if $(n=1)$ **then return;** <-------------------- $\theta(1)$

else:

$X \leftarrow$ Merge-Sort**(**...**);** <------------- $T(n/2)$

$Y \leftarrow$ Merge-Sort**(**...**);** <------------- $T(n/2)$

return Merge **(**X,Y, n/2**);** <------------- $\theta(n)$

MergeSort Analysis

Let $T(n)$ be the worst-case running time for an array of n elements.

$$T(n) = \theta(1) \quad \textbf{if } (n=1)$$

$$ = 2T(n/2) + cn \quad \textbf{if } (n>1)$$

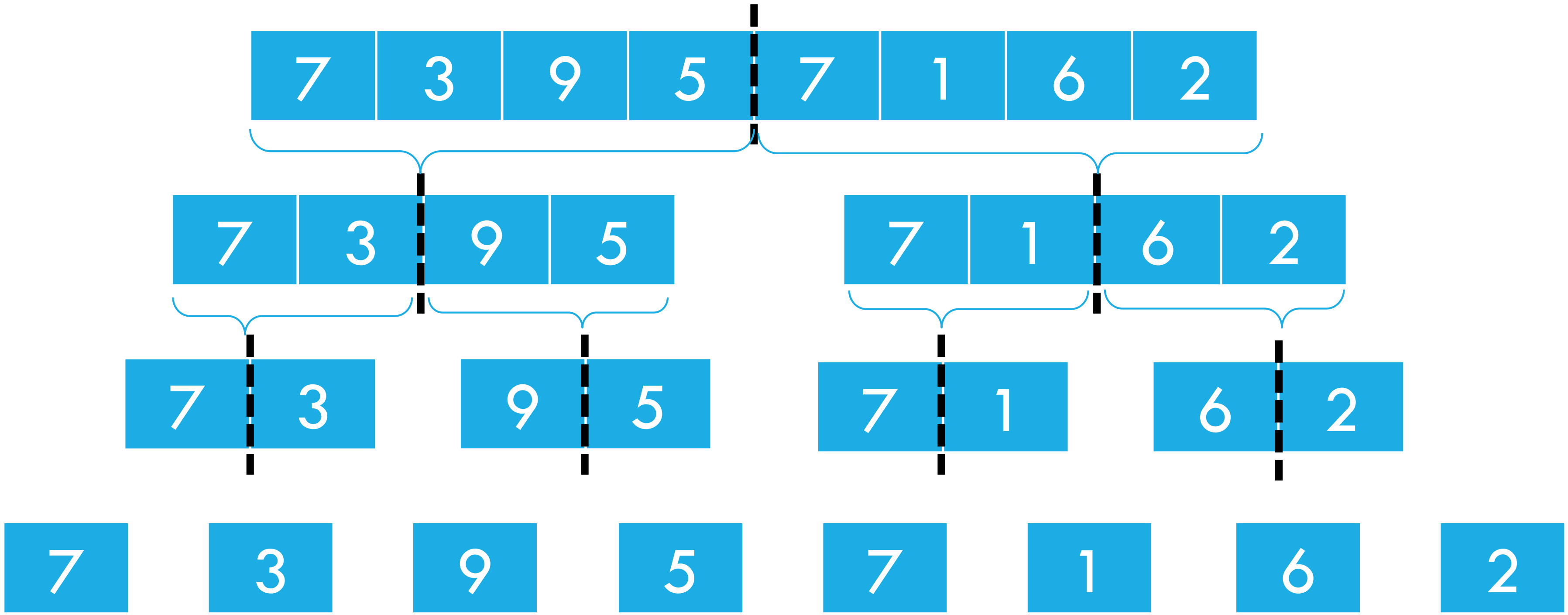
Techniques for Solving Recurrences

1. Guess and verify (via induction).

2. Draw the recursion tree.

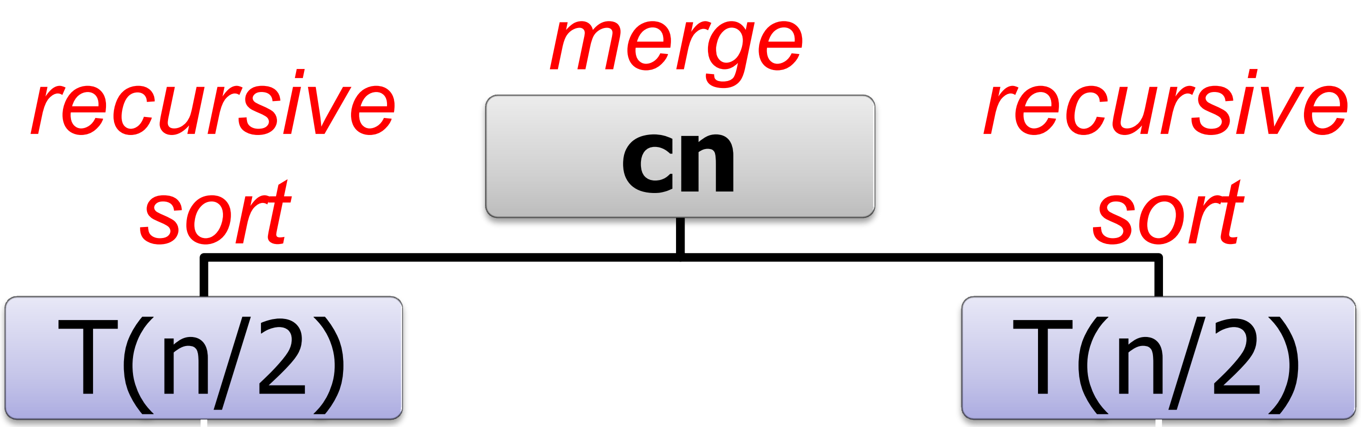
3. Use the Master Theorem (see CS3230) or the Akra–Bazzi Method, or other advanced techniques.

MergeSort: Recurse “downwards”



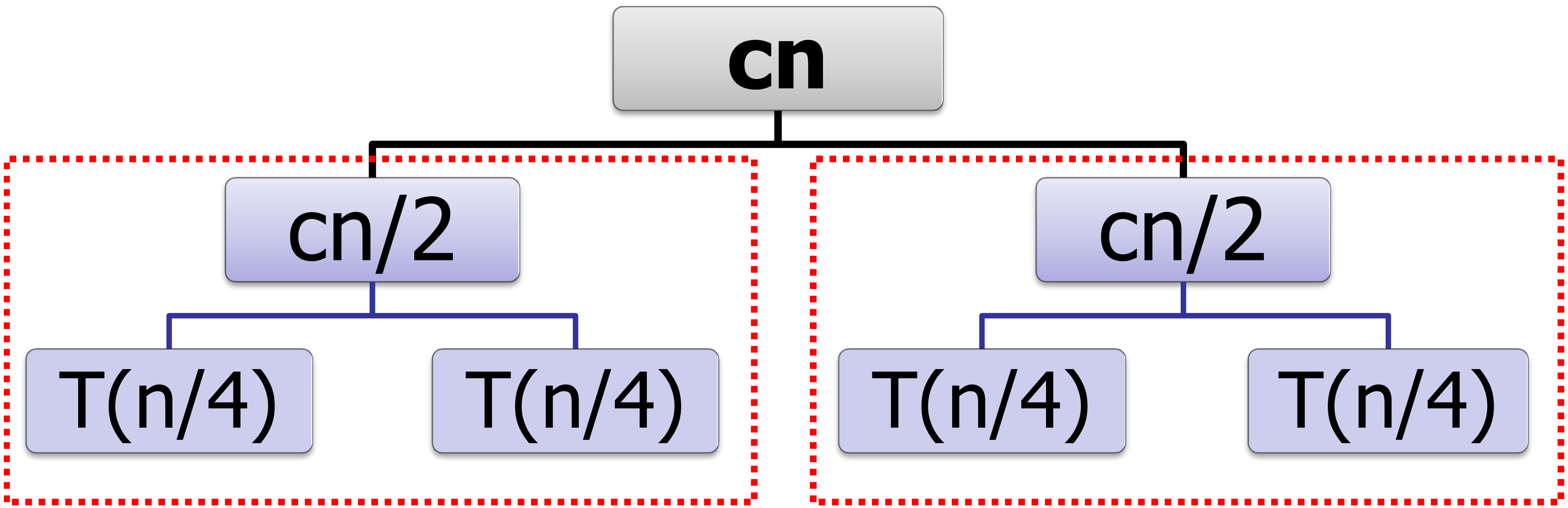
MergeSortAnalysis

$T(n) = 2T(n/2) + cn$



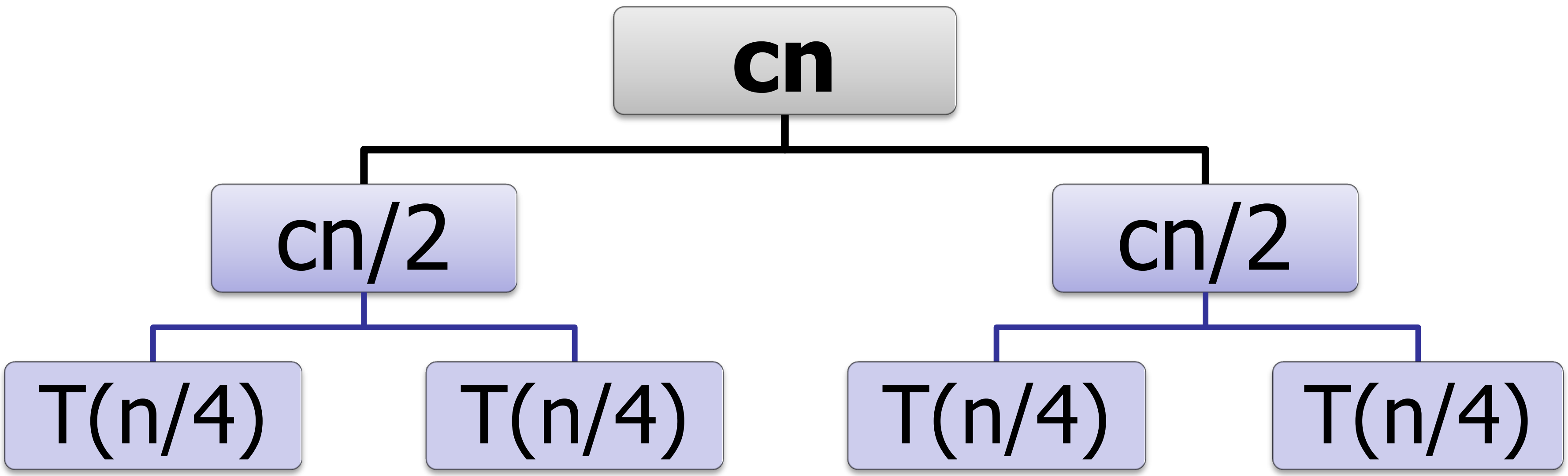
MergeSortAnalysis

$$T(n) = 2T(n/2) + cn$$



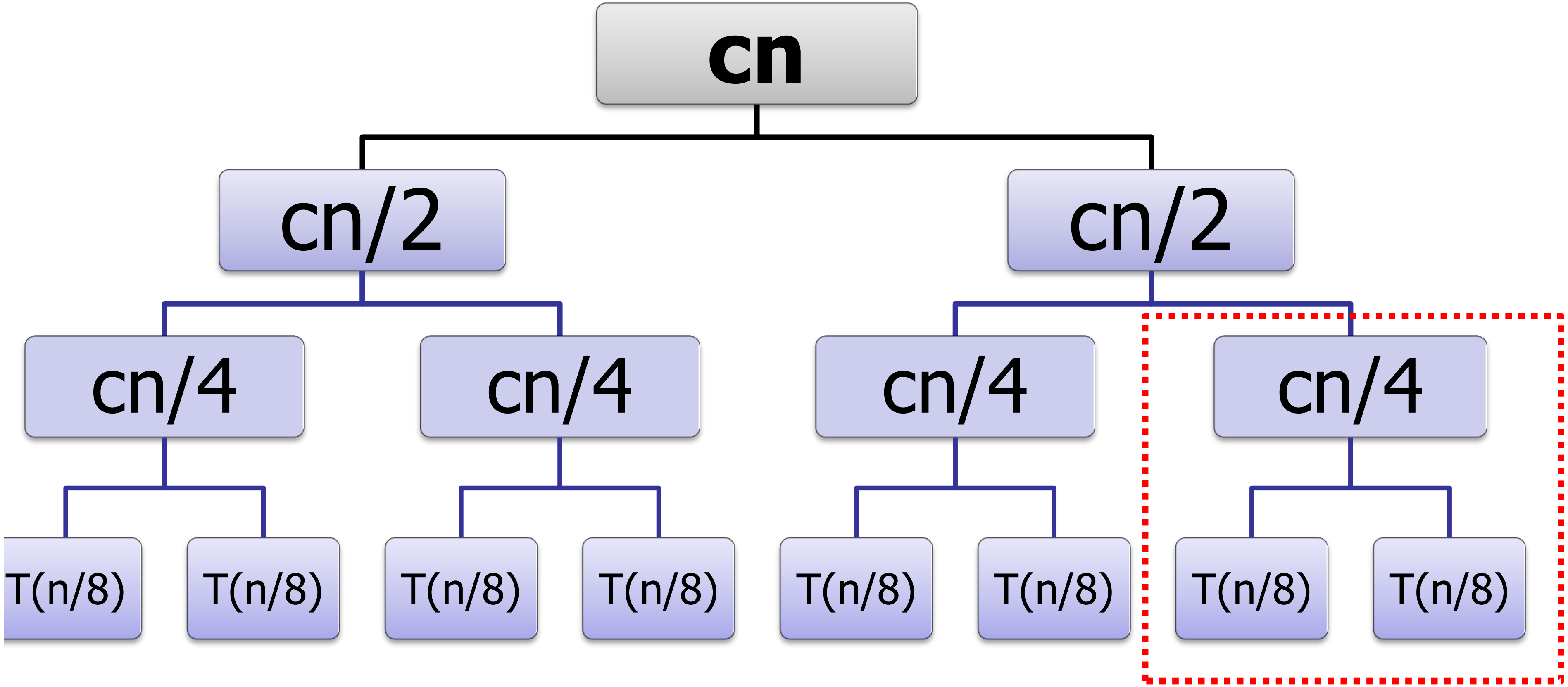
MergeSortAnalysis

$$T(n) = 2T(n/2) + cn$$



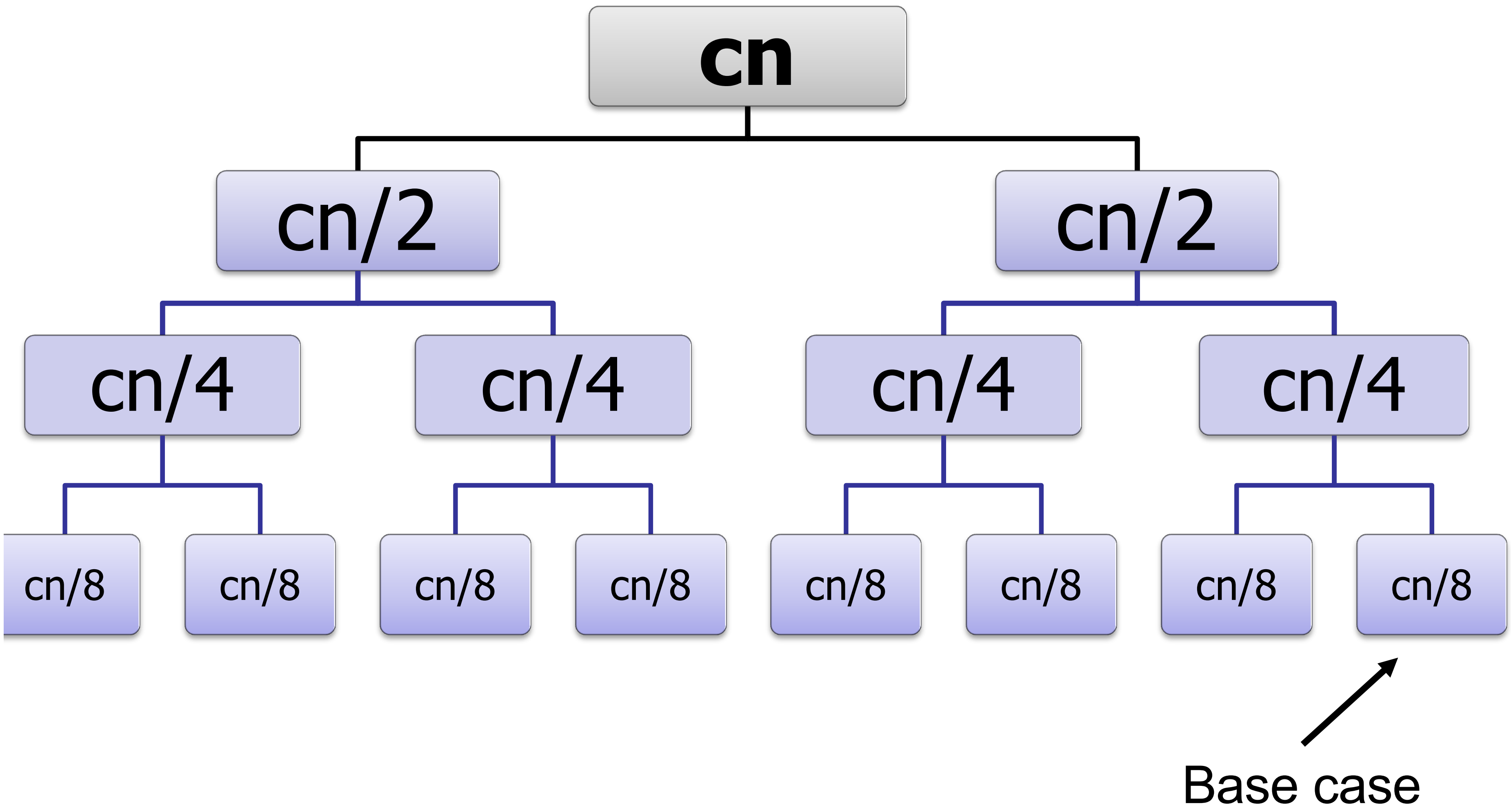
MergeSort Analysis

$T(n) = 2T(n/2) + cn$



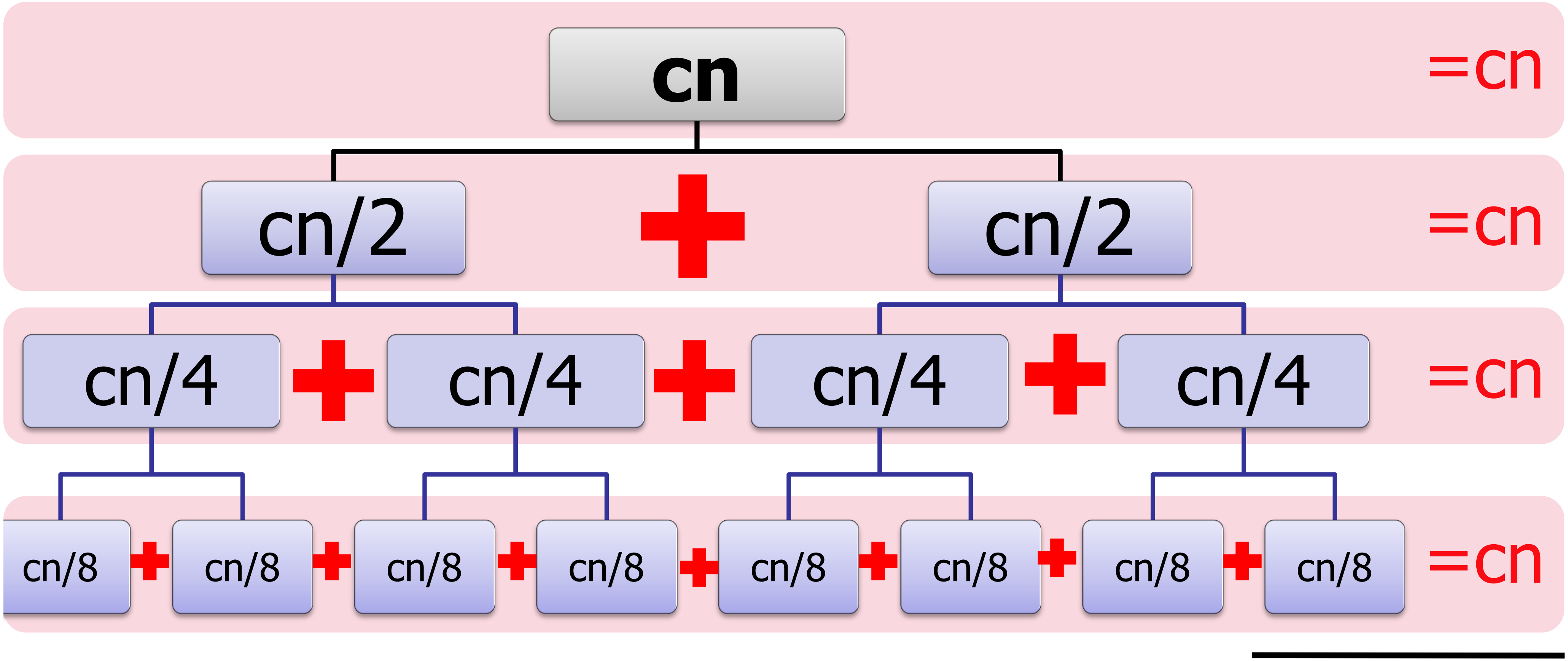
MergeSort Analysis

$$T(n) = 2T(n/2) + cn$$



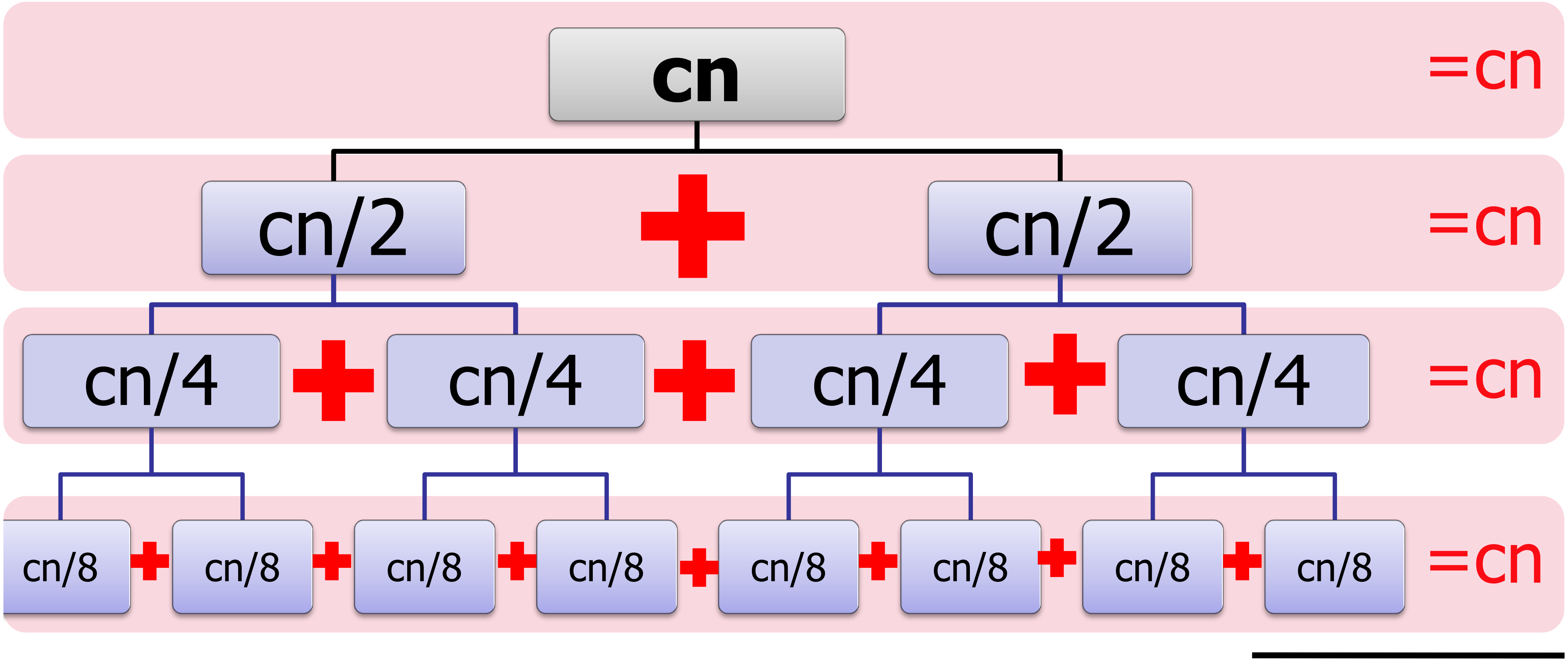
MergeSort Analysis

$$T(n) = 2T(n/2) + cn$$



MergeSort Analysis

$$T(n) = 2T(n/2) + cn$$



Key question: how many levels?

MergeSort Analysis

$T(n) = 2T(n/2) + cn$

level	number
0	1
1	2
2	4
3	8
4	16
…	…
h	??

$$\text{number} = 2^{\text{level}}$$

MergeSort Analysis

$T(n) = 2T(n/2) + cn$

level	number
0	1
1	2
2	4
3	8
4	16
…	…
h	*n*

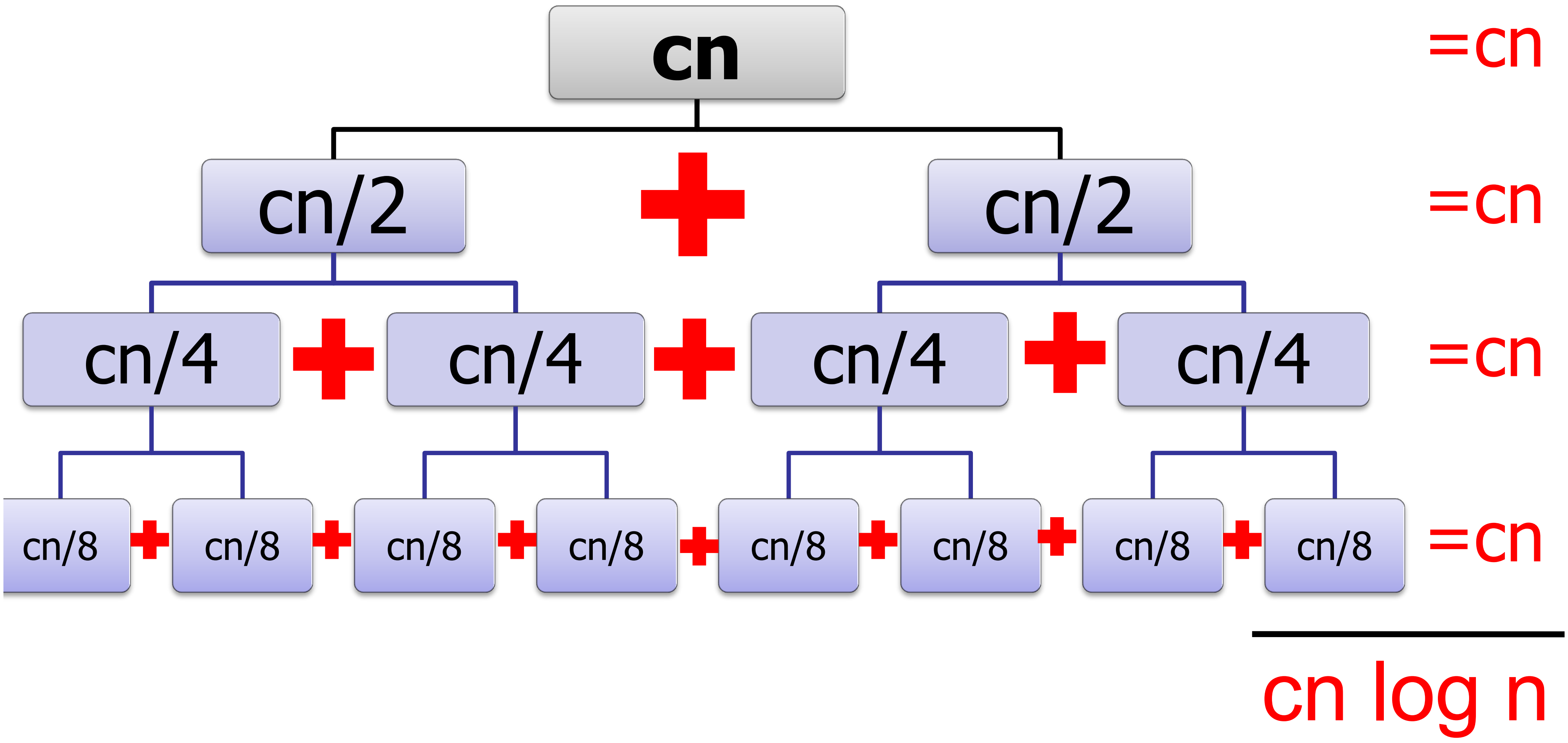
$$\text{number} = 2^{\text{level}}$$

$$n = 2^h$$

$$\log n = h$$

MergeSort Analysis

$$T(n) = 2T(n/2) + cn$$



MergeSortAnalysis

$T(n) = O(n \log n)$

MergeSort(A, n)

if (n=1) **then return;**

else:

X ← MergeSort**(**...**);**

Y ← MergeSort**(**...**);**

return Merge **(**X,Y, n/2**);**

Techniques for Solving Recurrences

1. Guess and verify (via induction).

2. Draw the recursion tree.

3. Use the Master Theorem (see CS3230) or the Akra–Bazzi Method, or other advanced techniques.

Guess: $T(n) = O(n \log n)$

<u>Recurrence being analyzed:</u>

$T(n) = 2T(n/2) + c \cdot n$

$T(1) = c$

Guess: $T(n) = c \cdot n \log n$

More precise guess:
Fix constant c.

Recurrence being analyzed:

$T(n) = 2T(n/2) + c \cdot n$

$T(1) = c$

Guess: $T(n) = c \cdot n \log n$

Induction:
Base case

$T(1) = c$

Recurrence being analyzed:

$T(n) = 2T(n/2) + c \cdot n$

$T(1) = c$

Guess: $T(n) = c \cdot n \log n$

Induction:
Assume true for all smaller values.

$T(1) = c$

$T(x) = c \cdot x \log x$ for all $x < n$.

Recurrence being analyzed:

$T(n) = 2T(n/2) + c \cdot n$

$T(1) = c$

Guess: T(n) = c·n log n

Induction:
Prove for n.

T(1) = c

T(x) = c·x log x for all x < n.

$$
\begin{aligned}
T(n) &= 2T(n/2) + cn \\
&= 2(c(n/2)\log(n/2)) + cn \\
&= cn\log(n/2) + cn \\
&= cn\log(n) - cn\log(2) + cn \\
&= cn\log(n)
\end{aligned}
$$

Recurrence being analyzed:

T(n) = 2T(n/2) + c·n

T(1) = c

Guess: $T(n) = c \cdot n \log n$

$T(1) = c$

$T(x) = c \cdot x \log x$ for all $x < n$.

$$
\begin{aligned}
T(n) &= 2T(n/2) + cn \\
&= 2(c(n/2)\log(n/2)) + cn \\
&= cn\log(n/2) + cn \\
&= cn\log(n) - cn\log(2) + cn \\
&= cn\log(n)
\end{aligned}
$$

Induction:
It works!

<u>Recurrence being analyzed:</u>

$T(n) = 2T(n/2) + c \cdot n$

$T(1) = c$

Top-Down vs. …

Step 1:
Divide array into two pieces.

MergeSort(A, n)

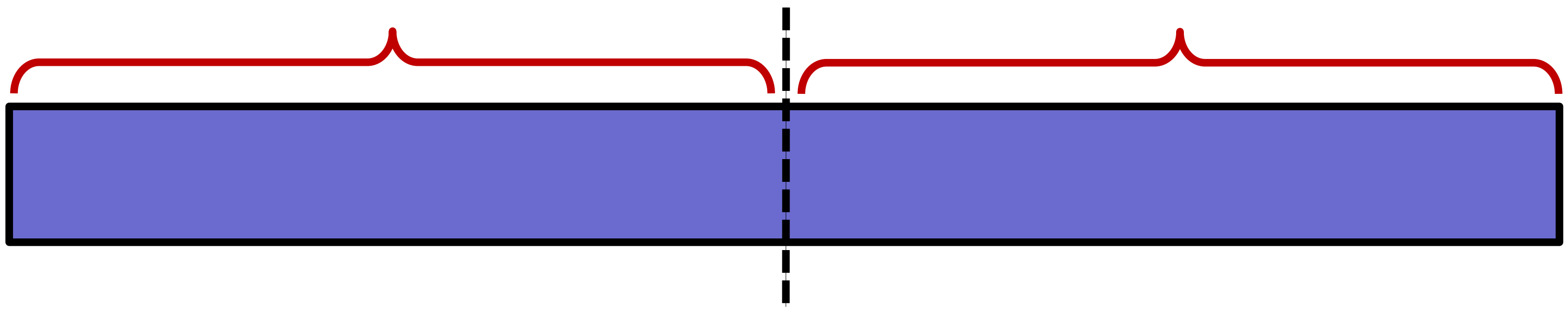
if (n=1) **then return;**

else:

X ←MergeSort**(**A[1..n/2], n/2**);**

Y ←MergeSort**(**A[n/2+1, n], n/2**);**

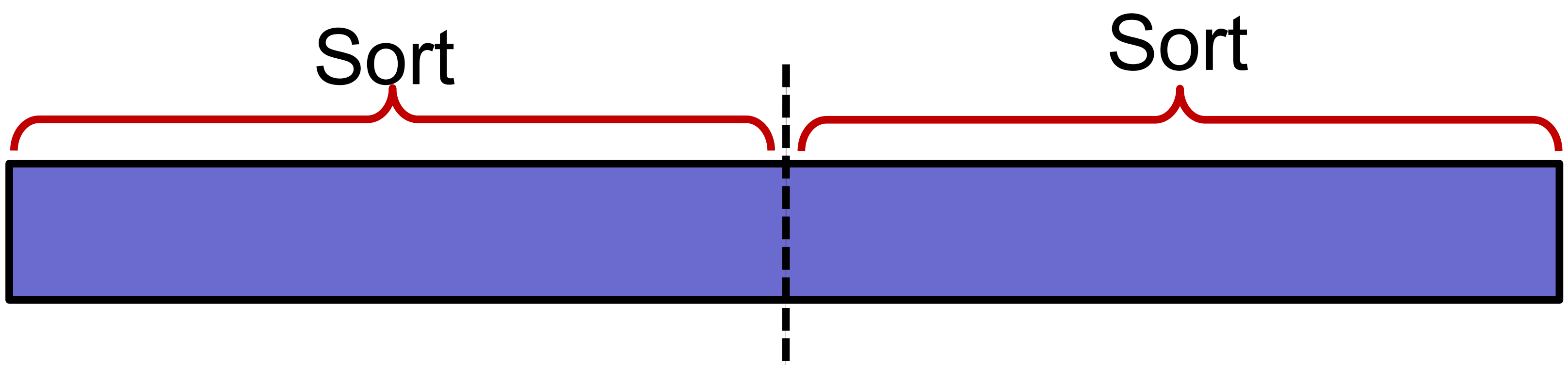
return Merge **(**X,Y, n/2**);**



Top-Down vs. ...

Step 2:
Recursively sort the two halves.

```
MergeSort(A, n)
    if (n=1) then return;
    else:
        X ←MergeSort(A[1..n/2],  n/2);
        Y ←MergeSort(A[n/2+1, n],  n/2);
    return Merge (X,Y, n/2);
```



Top-Down vs. ...

Step 3:
Merge the two halves into one sorted array.

MergeSort(A, n)

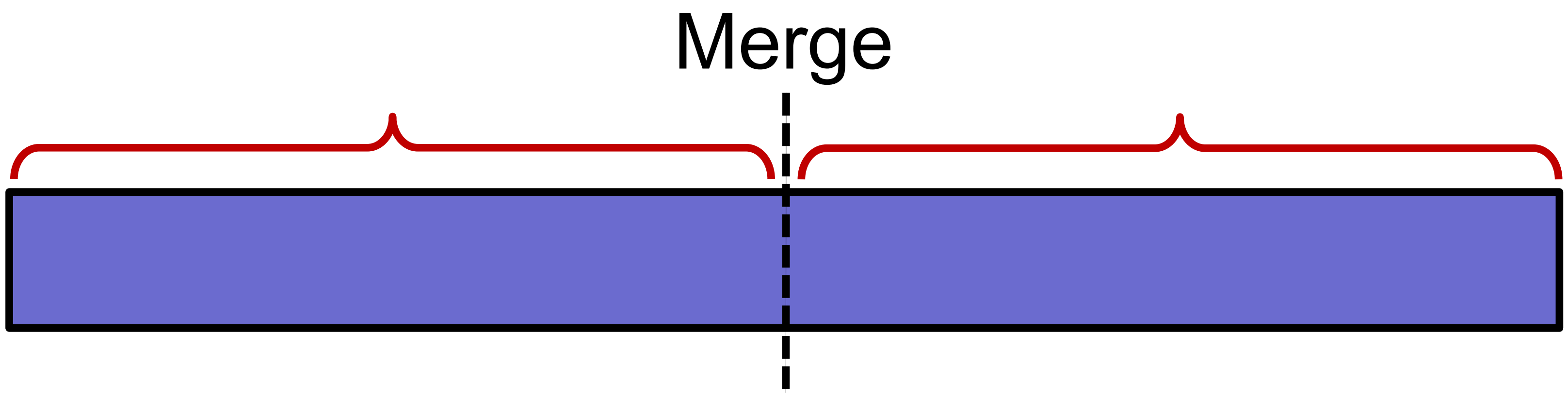
if (n=1) **then return;**

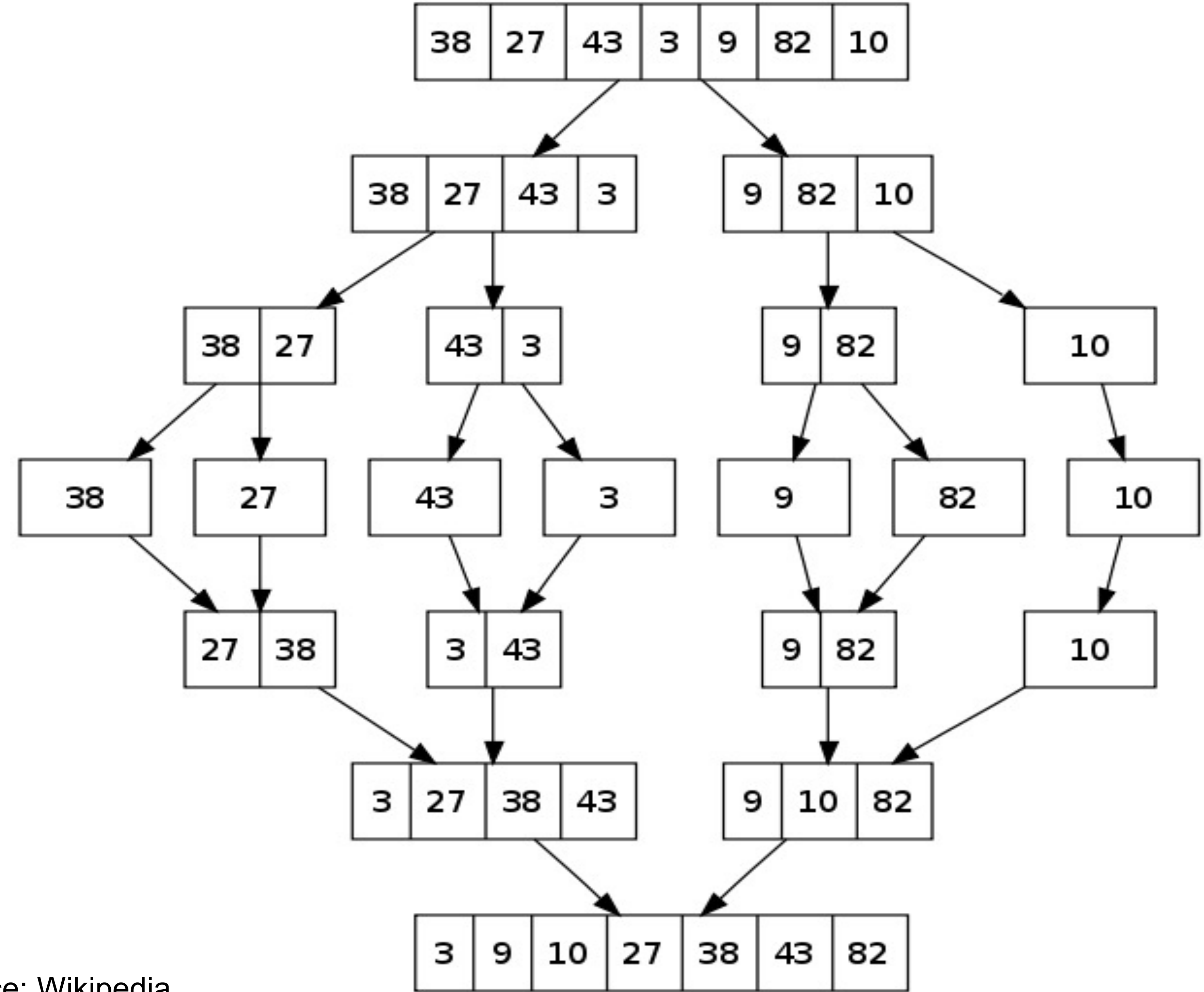
else:

X ← MergeSort**(**A[1..n/2], n/2**);**

Y ← MergeSort**(**A[n/2+1, n], n/2**);**

return Merge **(**X,Y, n/2**);**





Source: Wikipedia

MergeSort, Bottom Up

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

How much does it matter?

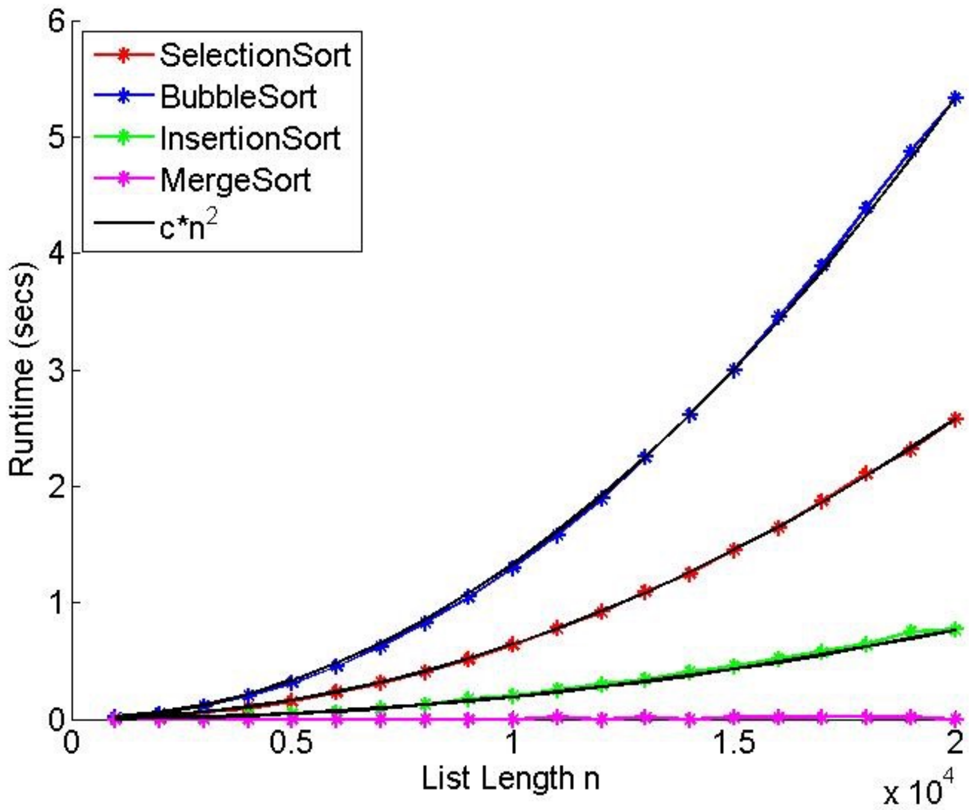
Comparing words in two files:

Version	Change	Running Time
Version 1		4,311.00s
Version 2	Better file handling	676.50s
Version 3	Mergesort replaces SelectionSort	6.59s
Version 4	Hashing replaces sorting	2.35s

Algorithm:

1. Read all text in both files.
2. Sort words.
3. Count how many times each word appears in each file.

real world performance



http://www.cs.toronto.edu/~jepson/csc148/2007F/notes/sorting.html]

When is it better to use InsertionSort instead of MergeSort?

A. When there is limited space?
B. When there are a lot of items to sort?
C. When there is a large memory cache?
D. When there are a small number of items?
E. When the list is mostly sorted?
F. Always
G. Never

MergeSort

When the list is mostly sorted:

- InsertionSort is fast!
- MergeSort is O(n log n)

How "close to sorted" should a list be for InsertionSort to be faster?

How would you check?

MergeSort

Small number of items to sort:

- MergeSort is slow!
- Caching performance, branch prediction, etc.
- User InsertionSort for $n < 1024$, say.

Base case of recursion:

- Use slower sort.

Run an experiment
and post on the forum
what the best switch-over
point is for your machine.

MergeSort

Space usage…

- Need extra space to do merge.
- Merge copies data to new array.

Space Complexity

Question:

How much space is allocated during a call to MergeSort?

Note:
Measure total allocated space.
We will not model *garbage collection* or other Java details.

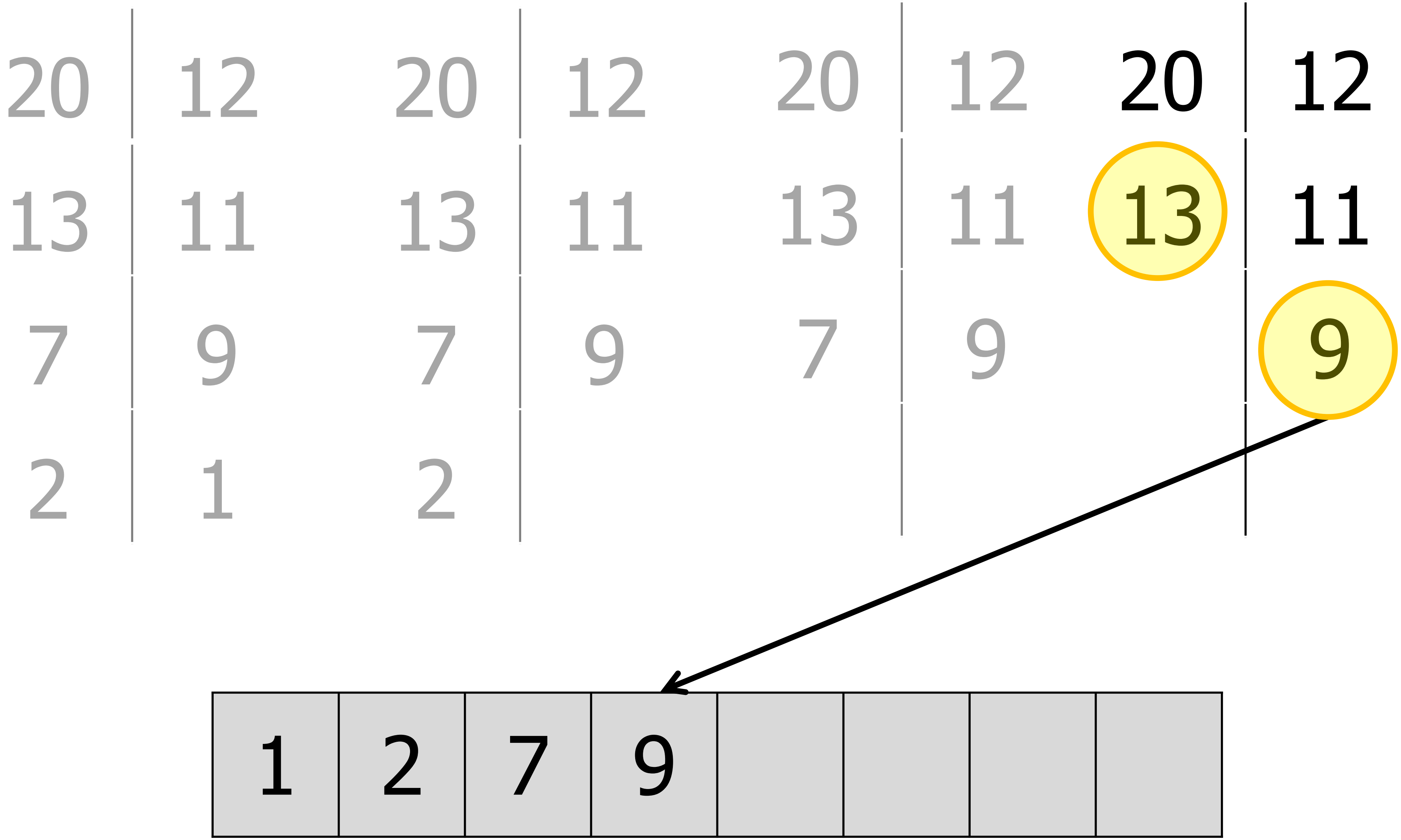
Space Complexity

Question:

How much space is allocated during a call to MergeSort?

Key subroutine: Merge

Merging Two Sorted Lists



Need temporary array of size n.

Space Analysis

Let $S(n)$ be the worst-case space allocated for an array of n elements.

MergeSort(A, n)

    if $(n=1)$ **then return;** ←------------------ $\theta(1)$

    else:

        $X \leftarrow$ Merge-Sort**(**...**);** ←------------ $S(n/2)$

        $Y \leftarrow$ Merge-Sort**(**...**);** ←------------ $S(n/2)$

    return Merge **(**$X, Y, n/2$**);** ←-------------- n

$$S(n) = 2S(n/2) + n$$

$$S(n) = ?$$

A. $O(\log n)$
B. $O(n)$
C. $O(n \log n)$
D. $O(n^2)$
E. $O(n^2 \log n)$
F. $O(2^n)$

ARCHIPELAGO

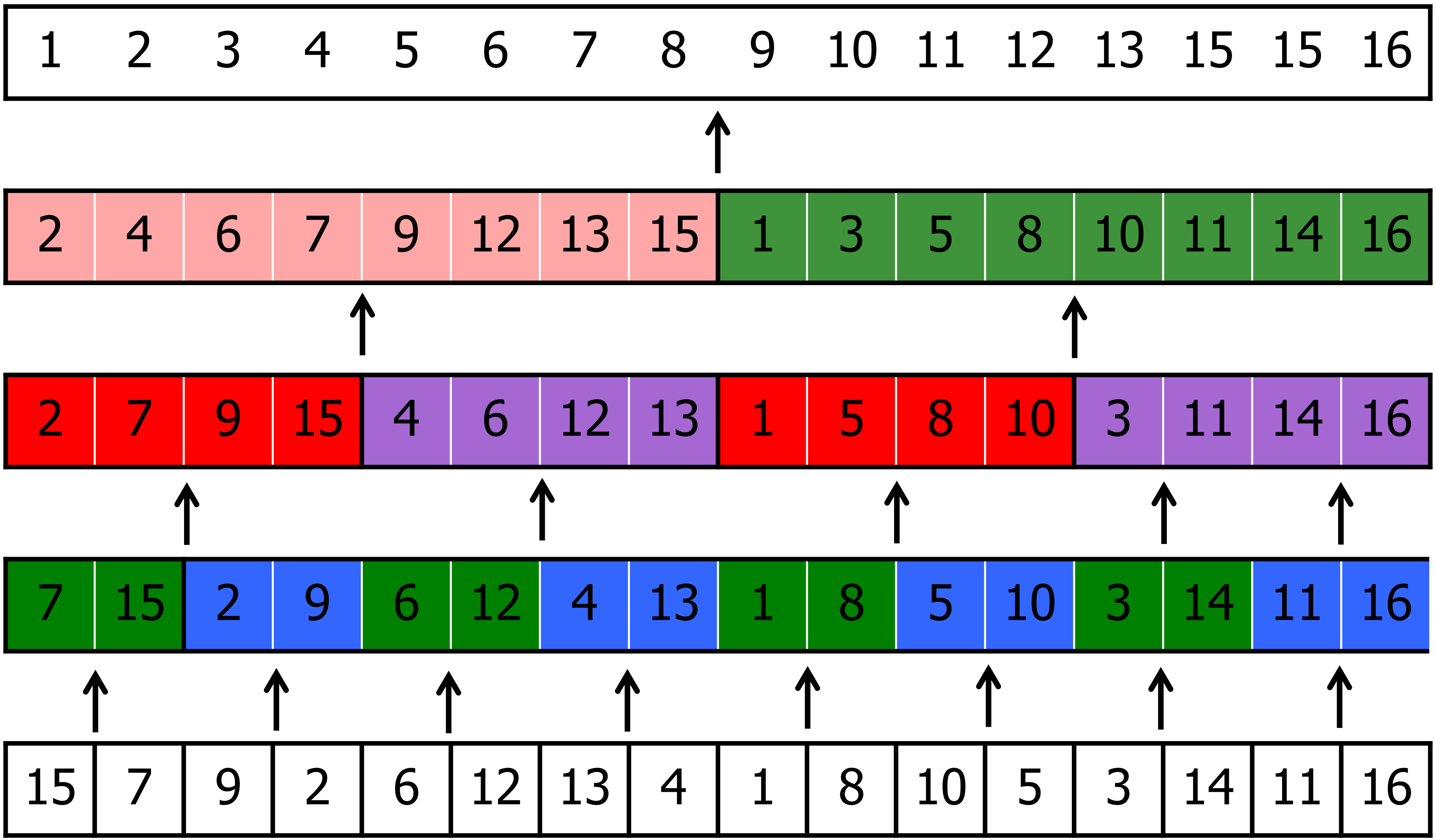
is open

Space Analysis

Let $S(n)$ be the worst-case space for an array of n elements.

$$
\begin{aligned}
S(n) &= \theta(1) && \textbf{if } (n=1) \\
&= 2S(n/2) + n && \textbf{if } (n>1) \\
&= O(n \log n)
\end{aligned}
$$

MergeSort



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

Challenge of the Day:

Design a version of MergeSort that minimizes the amount of extra space needed.

Hint: Do not allocate any new space during the recursive calls!

Stability

Is MergeSort stable?

ARCHIPELAGO

is open

MergeSort

Stability:

- MergeSort is stable if “merge” is stable.
- Merge is stable if carefully implemented.

Sorting Analysis

Summary:

- BubbleSort: $O(n^2)$
- SelectionSort: $O(n^2)$
- InsertionSort: $O(n^2)$
- MergeSort: $O(n \log n)$

Also:

The power of divide-and-conquer!

How to solve recurrences…

Properties: time, space, stability

Slowest Sorting Algorithm?

Step 1:

- Generate all the permutations of the input.

Step 2:

- Sort the permutations (by number of inversions).

Step 3:

- Return the first element in the sorted list of permutations.

Slowest Sorting Algorithm?

Step 1:

- Generate all the permutations of the input.

Step 2:

- Sort the permutations (by number of inversions).

Use BogoSort!

Roughly: $O((n!)!))$

Step 3:

- Return the first element in the sorted list of permutations.

Slowest Sorting Algorithm?

Step 1:

- Generate all the permutations of the input.

Step 2:

- Sort the permutations (by number of inversions).

Recurse!
Recursive instance is larger than original!

Step 3:

- Return the first element in the sorted list of permutations.

Slowest Sorting Algorithm?

Step 1:

- Generate all the permutations of the input.

Step 2:

- Sort the permutations (by number of inversions).

Recurse!
After n! recursions, use QuickSort for the “base case”.

Step 3:

- Return the first element in the sorted list of permutations.

Ingrassia-Kurtz Sort

Step 1:

- Generate all the permutations of the input.

Step 2:

- Sort the permutations (by number of inversions).

Recurse!
After n! recursions, use QuickSort for the "base case".

Step 3:

- Return the first element in the sorted list of permutations.

Sorting, Part II

QuickSort

- Divide-and-Conquer
- Paranoid QuickSort
- Randomized Analysis

(Warning: PS3 opens today and depends on QuickSort, but you can get started without that.)

Summary

Name	Best Case	Average Case	Worst Case	Extra Memory	Stable?
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	No
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	Yes