Dynamic Programming

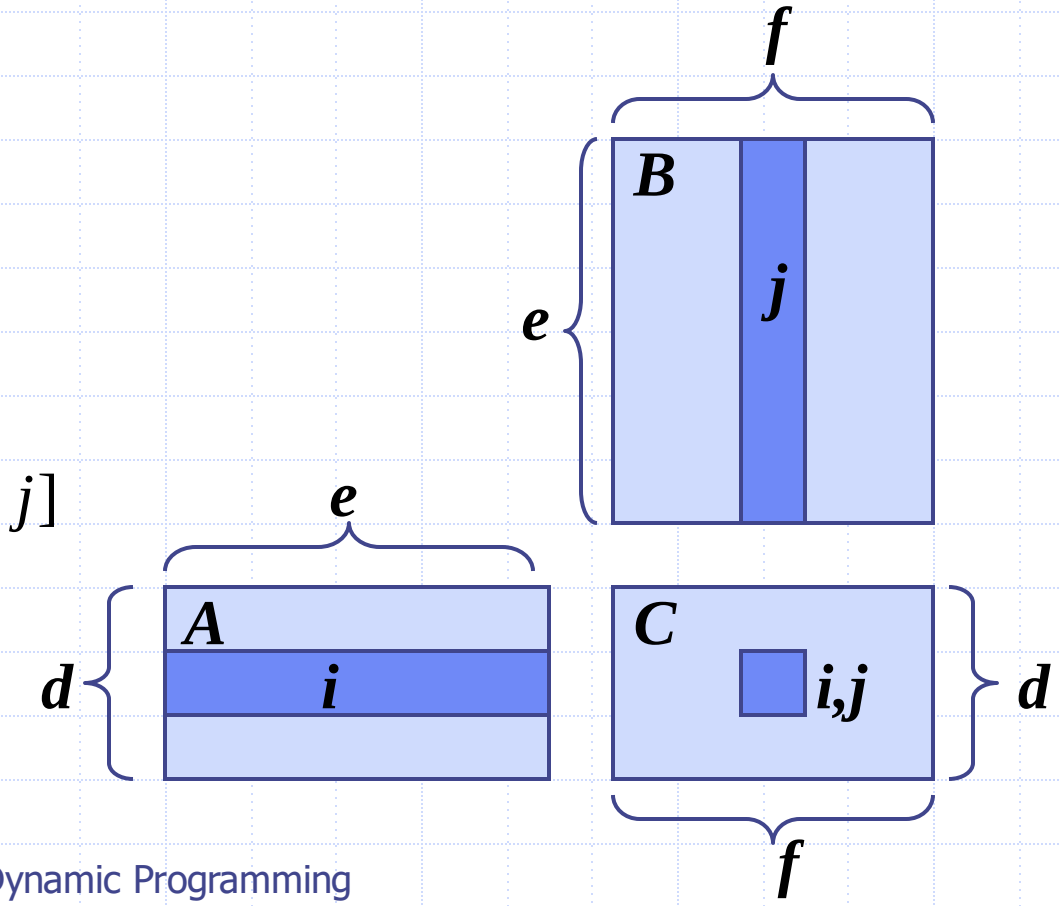
Motivating Example

- Fibonacci number computation
  - Recursive definition
    - fib(n) = fib(n-1)+fib(n-2)
  - Complexity – $O(\Phi^n)$

Matrix Chain-Products

- Dynamic Programming is a general algorithm design paradigm.
  - **Matrix Chain-Products**
- Review: Matrix Multiplication.
  - $C = A*B$
  - A is $d \times e$ and B is $e \times f$
  - $O(def)$ time

$$C[i, j] = \sum_{k=0}^{e-1} A[i,k] * B[k, j]$$



Matrix Chain-Products

- **Matrix Chain-Product:**
  - Compute $A=A_0*A_1*\ldots*A_{n-1}$
  - A_i is $d_i \times d_{i+1}$
  - Problem: How to parenthesize?
- Example
  - B is 3×100
  - C is 100×5
  - D is 5×5
  - (B*C)*D takes 1500 + 75 = 1575 ops
  - B*(C*D) takes 1500 + 2500 = 4000 ops

An Enumeration Approach

- **Matrix Chain-Product Alg.:**
  - Try all possible ways to parenthesize $A=A_0*A_1*\ldots*A_{n-1}$
  - Calculate number of ops for each one
  - Pick the one that is best
- Running time:
  - This is **exponential**!
  - It is called the Catalan number, and it is almost 4^n.
  - This is a terrible algorithm!

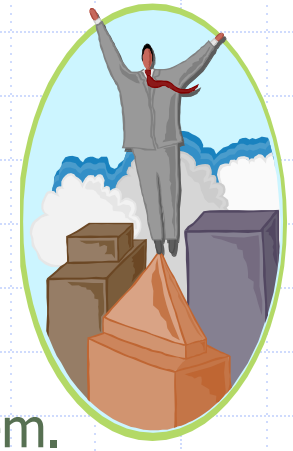
A Greedy Approach

- Idea #1: repeatedly select the product that uses (up) the most operations.
- Counter-example:
  - A is 10×5
  - B is 5×10
  - C is 10×5
  - D is 5×10
  - Greedy idea #1 gives (A*B)*(C*D), which takes 500+1000+500 = 2000 ops
  - A*((B*C)*D) takes 500+250+250 = 1000 ops

Another Greedy Approach

- Idea #2: repeatedly select the product that uses the fewest operations.
- Counter-example:
  - A is 101 × 11
  - B is 11 × 9
  - C is 9 × 100
  - D is 100 × 99
  - Greedy idea #2 gives A*((B*C)*D)), which takes 109989+9900+108900=228789 ops
  - (A*B)*(C*D) takes 9999+89991+89100=189090 ops
- The greedy approach is not giving us the optimal value.

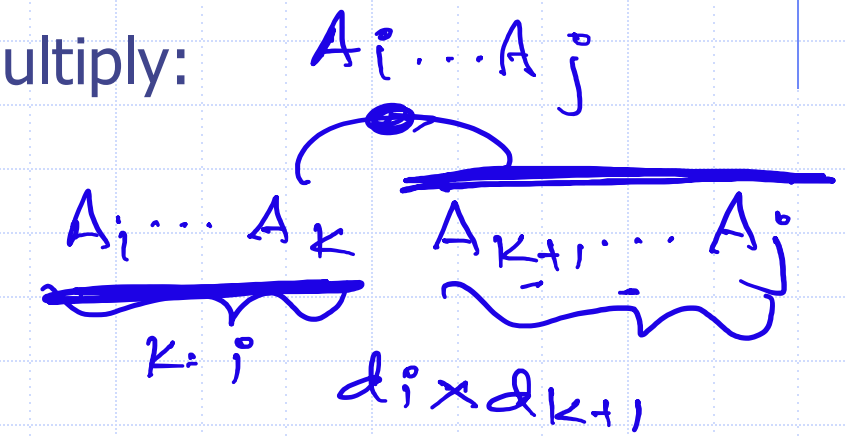
A "Recursive" Approach



- Define **subproblems**:
  - Find the best parenthesization of $A_i*A_{i+1}*...*A_j$.
  - Let $N_{i,j}$ denote the number of operations done by this subproblem.
  - The optimal solution for the whole problem is $N_{0,n-1}$.
- **Subproblem optimality**: The optimal solution can be defined in terms of optimal subproblems
  - There has to be a final multiplication (root of the expression tree) for the optimal solution.
  - Say, the final multiply is at index i: $(A_0*...*A_i)*(A_{i+1}*...*A_{n-1})$.
  - Then the optimal solution $N_{0,n-1}$ is the sum of two optimal subproblems, $N_{0,i}$ and $N_{i+1,n-1}$ plus the time for the last multiply.
  - If the global optimum did not have these optimal subproblems, we could define an even better "optimal" solution.

A Characterizing Equation

- The global optimal has to be defined in terms of optimal subproblems, depending on where the final multiply is at.
- Let us consider all possible places for that final multiply:
  - Recall that A_i is a $d_i \times d_{i+1}$ dimensional matrix.
  - So, a characterizing equation for $N_{i,j}$ is the following:



$$N_{i,j} = \min_{i \le k \le j} \{N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\}$$

- Note that subproblems are not independent--the **subproblems overlap**.

© 2014 Goodrich, Tamassia, Goldwasser Dynamic Programming 9

A Dynamic Programming Algorithm

- Since subproblems overlap, we don't use recursion.
- Instead, we construct optimal subproblems "bottom-up."
- $N_{i,i}$'s are easy, so start with them
- Then do length 2,3,… subproblems, and so on.
- The running time is $O(n^3)$

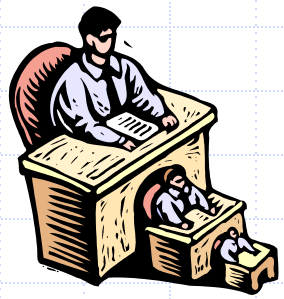
Algorithm *matrixChain*(*S*):

Input: sequence *S* of *n* matrices to be multiplied

Output: number of operations in an optimal paranethization of *S*

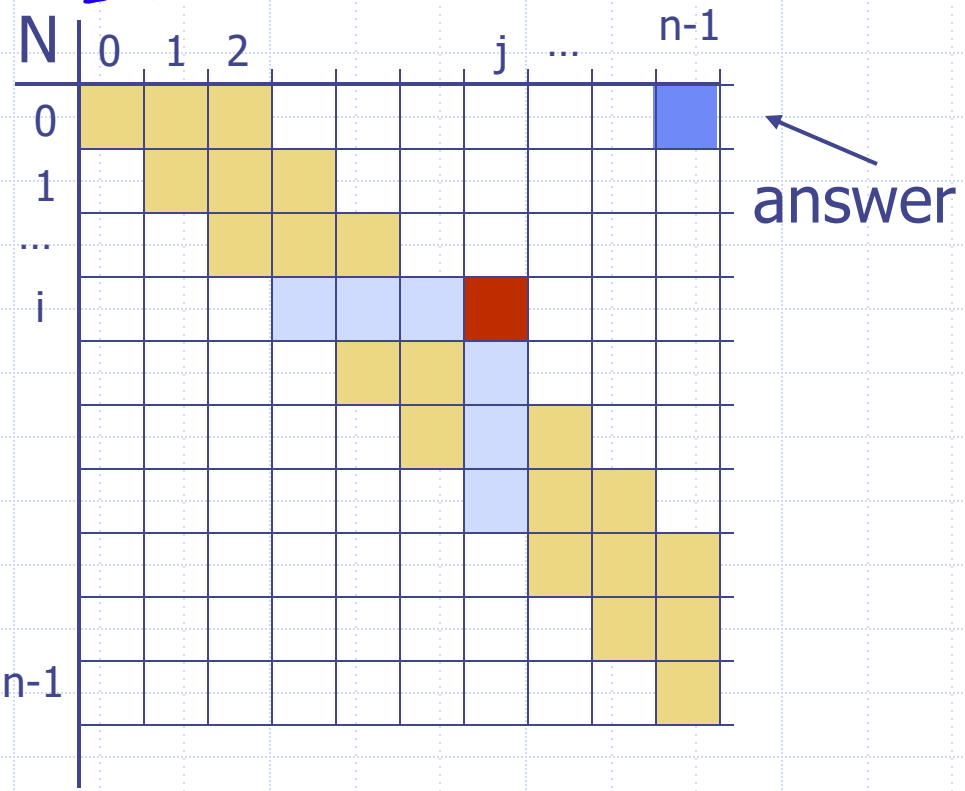
for $i \leftarrow 1$ **to** $n-1$ **do**
  $N_{i,i} \leftarrow 0$
for $b \leftarrow 1$ **to** $n-1$ **do**
  for $i \leftarrow 0$ **to** $n-b-1$ **do**
    $j \leftarrow i+b$
    $N_{i,j} \leftarrow +\infty$
    for $k \leftarrow i$ **to** $j-1$ **do**
      $N_{i,j} \leftarrow \min\{N_{i,j}, N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\}$

A Dynamic Programming Algorithm Visualization



- The bottom-up construction fills in the N array by diagonals
- $N_{i,j}$ gets values from pervious entries in i-th row and j-th column
- Filling in each entry in the N table takes O(n) time.
- Total run time: $O(n^3)$
- Getting actual parenthesization can be done by remembering "k" for each N entry

$$N_{i,j} = \min_{i \le k < j} \{N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\}$$



A_1 A_2 A_3 A_4

10×50 50×100 100×50 50×200

	A_1	A_2	A_3	A_4
A_1	0	50k	•	
A_2		0	•	
A_3			0	
A_4				0

$N_{2,4} =$

N_{ij}

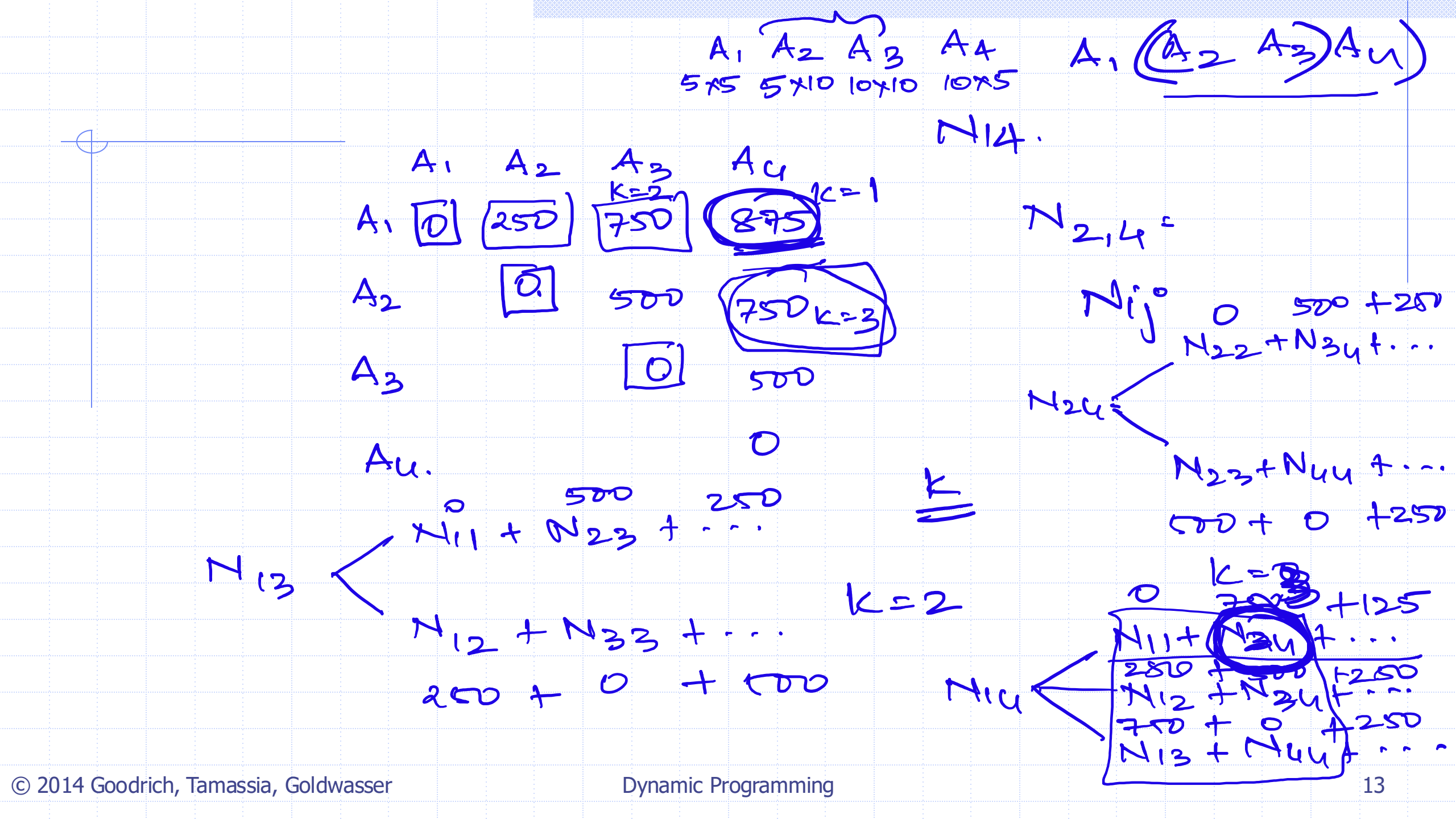
N_{12}

$1 \le k < 2 \{ N_{11}^{0} + N_{22}^{0} + d_1 \cdot d_2 \cdot d_3 \}$

$N_{13} = \min_{1 \le k < 3} \{ N_{1k} + N_{k+1,3} + d_1 d_{k+1} d_4 \}$

$\{ N_{11} + N_{23} + d_1 d_2 d_4 \}$

$N_{12} + N_{33} + d_1 d_3 d_4$


A1 A2 A3 A4
5x5 5x10 10x10 10x5
A1 (A2 A3) A4
N14.
A1 A2 A3 A4
A1 0 250 750 875
k=2
k=1
A2 0 500 750
k=3
A3 0 500
A4. 0
N13
N11 + N23 + ...
0 500 250
N12 + N33 + ...
250 + 0 + 500
k=2
k
N24
N22 + N34 + ...
Nij 0 500 +250
N23 + N44 + ...
500 + 0 +250
N14
k=3
0 +125
N11 + N34 + ...
250 + 500 +250
N12 + N34 + ...
750 + 0 +250
N13 + N44 + ...

$A_1 A_2 \cdots A_N$

for $i = 1 : N$

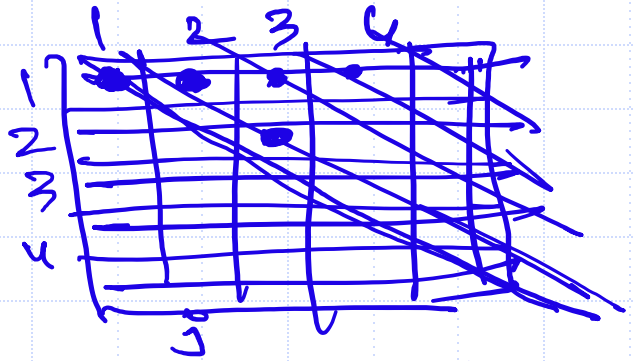
$T[i, i] = 0$

for $i = 1 : N$

for $j = 1 : N - i$

$T[j, j+i] = +\infty$

$N[j, j+i]$



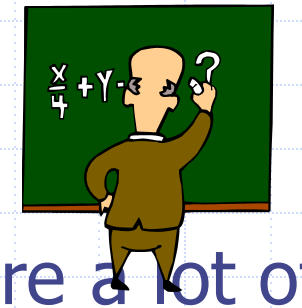
$i = 1, 1, 2, \ldots N-1$

$i = 1$ $j = 2, 3, \ldots N-1$

$i = 1$ $j = 1, 2 \cdots N-1$

$(1,2)$ $(2,3)$ $(N-1, N)$

The General Dynamic Programming Technique



- Applies to a problem that at first seems to require a lot of time (possibly exponential), provided we have:
  - **Simple subproblems:** the subproblems can be defined in terms of a few variables, such as j, k, l, m, and so on.
  - **Subproblem optimality:** the global optimum value can be defined in terms of optimal subproblems
  - **Subproblem overlap:** the subproblems are not independent, but instead they overlap (hence, should be constructed bottom-up).

Subsequences

- A ***subsequence*** of a character string $x_0x_1x_2...x_{n-1}$ is a string of the form $x_{i_1}x_{i_2}...x_{i_k}$, where $i_j < i_{j+1}$.
- Not the same as substring!
- Example String: ABCDEFGHIJK
  - Subsequence: ACEGIJK
  - Subsequence: DFGHK
  - Not subsequence: DAGH

The Longest Common Subsequence (LCS) Problem

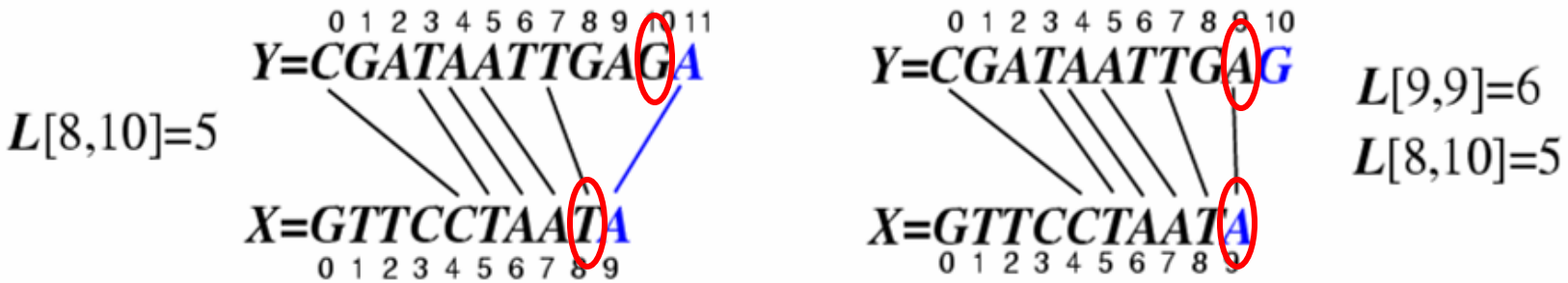
- Given two strings X and Y, the longest common subsequence (LCS) problem is to find a longest subsequence common to both X and Y
- Has applications to DNA similarity testing (alphabet is {A,C,G,T})
- Example: ABCDEFG and XZACKDFWGH have ACDFG as a longest common subsequence

A Poor Approach to the LCS Problem

- A Brute-force solution:
  - Enumerate all subsequences of X
  - Test which ones are also subsequences of Y
  - Pick the longest one.
- Analysis:
  - If X is of length n, then it has 2^n subsequences
  - This is an exponential-time algorithm!

A Dynamic-Programming Approach to the LCS Problem

- Define L[i,j] to be the length of the longest common subsequence of X[0..i] and Y[0..j].
- Allow for -1 as an index, so L[-1,k] = 0 and L[k,-1]=0, to indicate that the null part of X or Y has no match with the other.
- Then we can define L[i,j] in the general case as follows:
  1. If $x_i = y_j$, then L[i,j] = L[i-1,j-1] + 1 (we can add this match)
  2. If $x_i \neq y_j$, then L[i,j] = max{L[i-1,j], L[i,j-1]} (we have no match here)

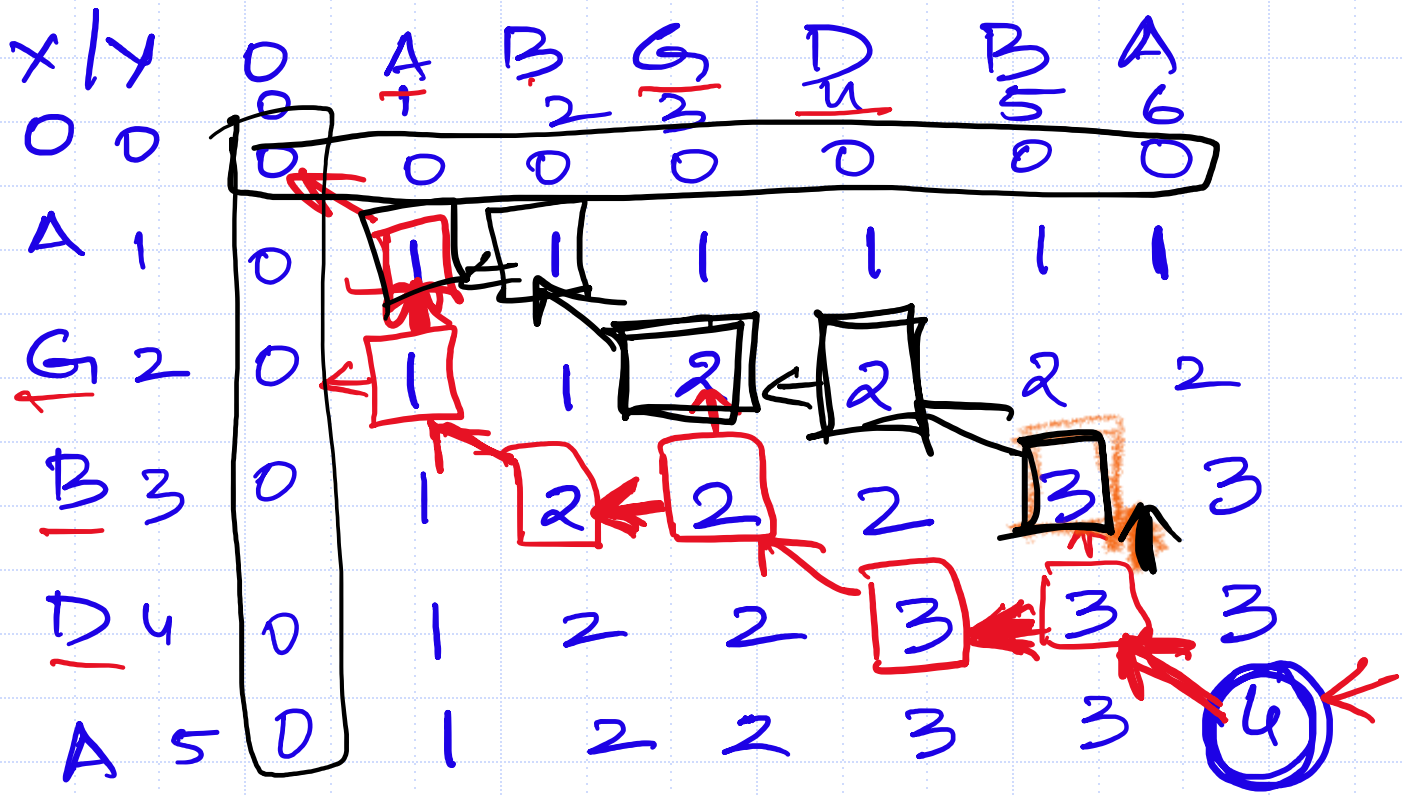


An LCS Algorithm

Algorithm LCS(X,Y):
Input: Strings X and Y with n and m elements, respectively
Output: For i = 0,…,n-1, j = 0,…,m-1, the length L[i, j] of a longest string that is a subsequence of both the string X[0..i] = $x_0x_1x_2…x_i$ and the string Y [0.. j] = $y_0y_1y_2…y_j$

```
for i =1 to n-1 do
    L[i,-1] = 0
for j =0 to m-1 do
    L[-1,j] = 0
for i =0 to n-1 do
    for j =0 to m-1 do
        if x_i = y_j then
            L[i, j] = L[i-1, j-1] + 1
        else
            L[i, j] = max{L[i-1, j] , L[i, j-1]}
return array L
```

$X = A\ G\ B\ D\ A$ (1 2 3 4 5) $Y = A\ B\ G\ D\ B\ A$ (1 2 3 4 5 6)



A G B A

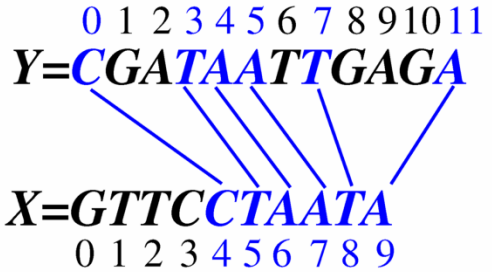
A B D A

```
for i: 0 : N
    L[0,i] = 0
    L[i,0] = 0
for i = 1 : M
    for j = 1 : N
        if x[i] = x[j]
            L[i,j] = L[i-1,j-1] + 1
        Else
            L[i,j] = max
                { L[i-1,j],
                  L[i,j-1] }
```

$L(M,N)$

Visualizing the LCS Algorithm

L	-1	0	1	2	3	4	5	6	7	8	9	10	11
-1	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	1	1	1	1	1	1	1	1	1	1
1	0	0	1	1	2	2	2	2	2	2	2	2	2
2	0	0	1	1	2	2	2	3	3	3	3	3	3
3	0	1	1	1	2	2	2	3	3	3	3	3	3
4	0	**1**	1	1	2	2	2	3	3	3	3	3	3
5	0	1	1	1	**2**	2	2	3	4	4	4	4	4
6	0	1	1	2	2	**3**	3	3	4	4	5	5	5
7	0	1	1	2	2	3	**4**	4	4	4	5	5	6
8	0	1	1	2	3	3	4	5	**5**	5	5	5	6
9	0	1	1	2	3	4	4	5	5	5	6	6	**6**

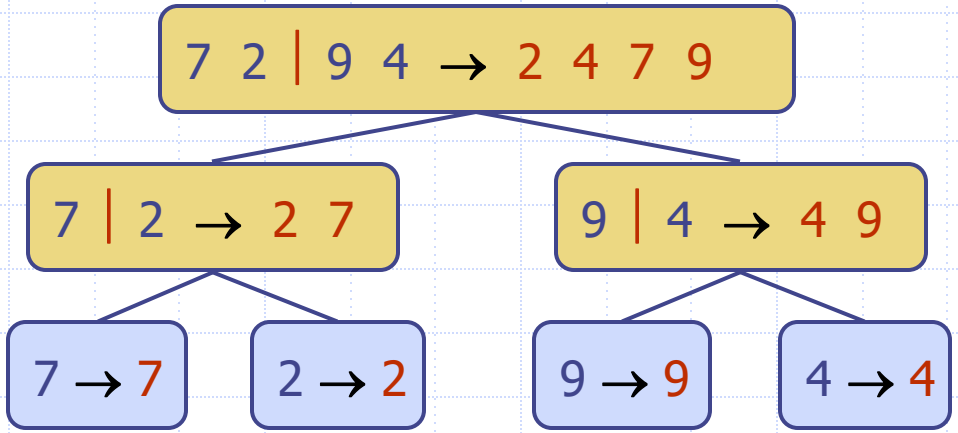


```
i = M  j = N        S = []
while i ≠ 0 or j ≠ 0
    if X[i] == Y[j]
        i = i - 1
        j = j - 1
        S.push(X[i])
    else
        if L[i-1, j] > L[i, j-1]
            i = i - 1
        else
            j = j - 1
end
```

Analysis of LCS Algorithm

- We have two nested loops
  - The outer one iterates n times
  - The inner one iterates m times
  - A constant amount of work is done inside each iteration of the inner loop
  - Thus, the total running time is $O(nm)$
- Answer is contained in L[n,m] (and the subsequence can be recovered from the L table).

Sorting and More Sorting



Divide-and-Conquer

- Divide-and conquer is a general algorithm design paradigm:
  - Divide: divide the input data S in two disjoint subsets S_1 and S_2
  - Recur: solve the subproblems associated with S_1 and S_2
  - Conquer: combine the solutions for S_1 and S_2 into a solution for S
- The base case for the recursion are subproblems of size 0 or 1

- Merge-sort is a sorting algorithm based on the divide-and-conquer paradigm
- Like heap-sort
  - It has $O(n \log n)$ running time
- Unlike heap-sort
  - It does not use an auxiliary priority queue
  - It accesses data in a sequential manner (suitable to sort data on a disk)

Merge-Sort

- Merge-sort on an input sequence S with n elements consists of three steps:
  - Divide: partition S into two sequences S_1 and S_2 of about $n/2$ elements each
  - Recur: recursively sort S_1 and S_2
  - Conquer: merge S_1 and S_2 into a unique sorted sequence

Algorithm *mergeSort*(S)
  Input sequence S with n elements
  Output sequence S sorted according to C
  if $S.size() > 1$
    $(S_1, S_2) \leftarrow partition(S, n/2)$
    $mergeSort(S_1)$
    $mergeSort(S_2)$
    $S \leftarrow merge(S_1, S_2)$

Merging Two Sorted Sequences

- The conquer step of merge-sort consists of merging two sorted sequences A and B into a sorted sequence S containing the union of the elements of A and B
- Merging two sorted sequences, each with $n/2$ elements and implemented by means of a doubly linked list, takes $O(n)$ time

Algorithm *merge*(A, B)

Input sequences A and B with $n/2$ elements each

Output sorted sequence of $A \cup B$

$S \leftarrow$ empty sequence

while $\neg A.isEmpty() \wedge \neg B.isEmpty()$

if $A.first().element() < B.first().element()$

$S.addLast(A.remove(A.first()))$

else

$S.addLast(B.remove(B.first()))$

while $\neg A.isEmpty()$

$S.addLast(A.remove(A.first()))$

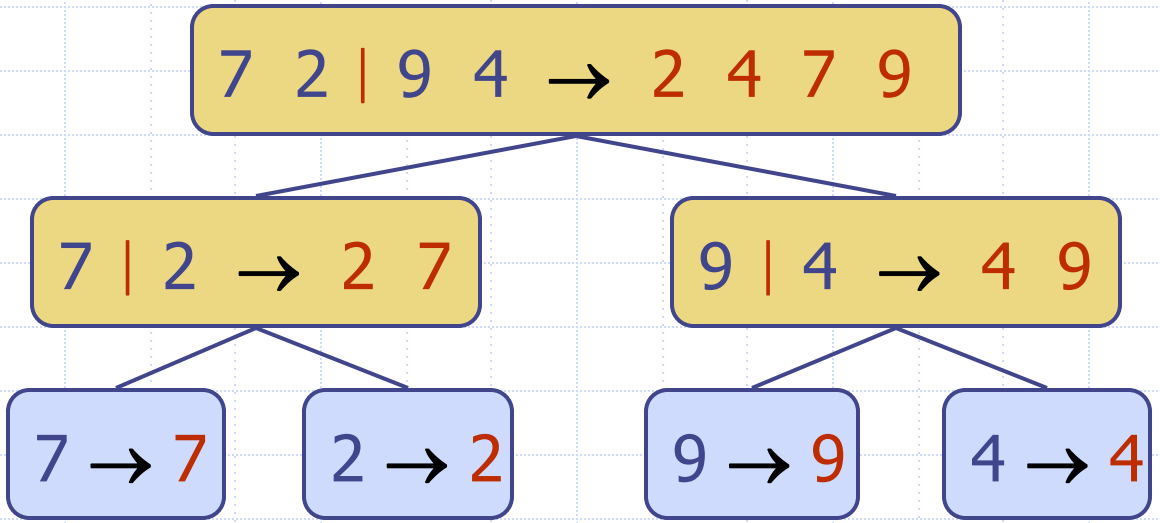
while $\neg B.isEmpty()$

$S.addLast(B.remove(B.first()))$

return S

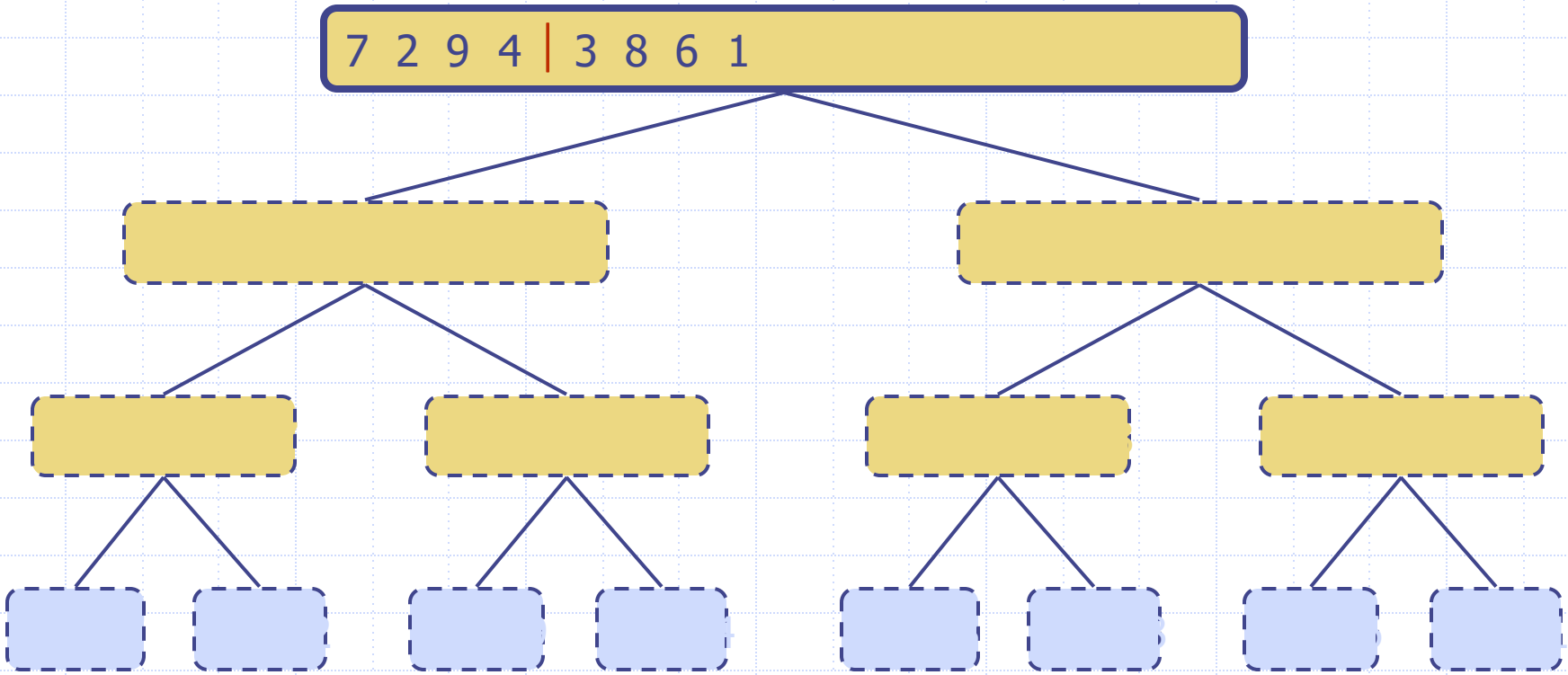
Merge-Sort Tree

- An execution of merge-sort is depicted by a binary tree
  - each node represents a recursive call of merge-sort and stores
    - unsorted sequence before the execution and its partition
    - sorted sequence at the end of the execution
  - the root is the initial call
  - the leaves are calls on subsequences of size 0 or 1



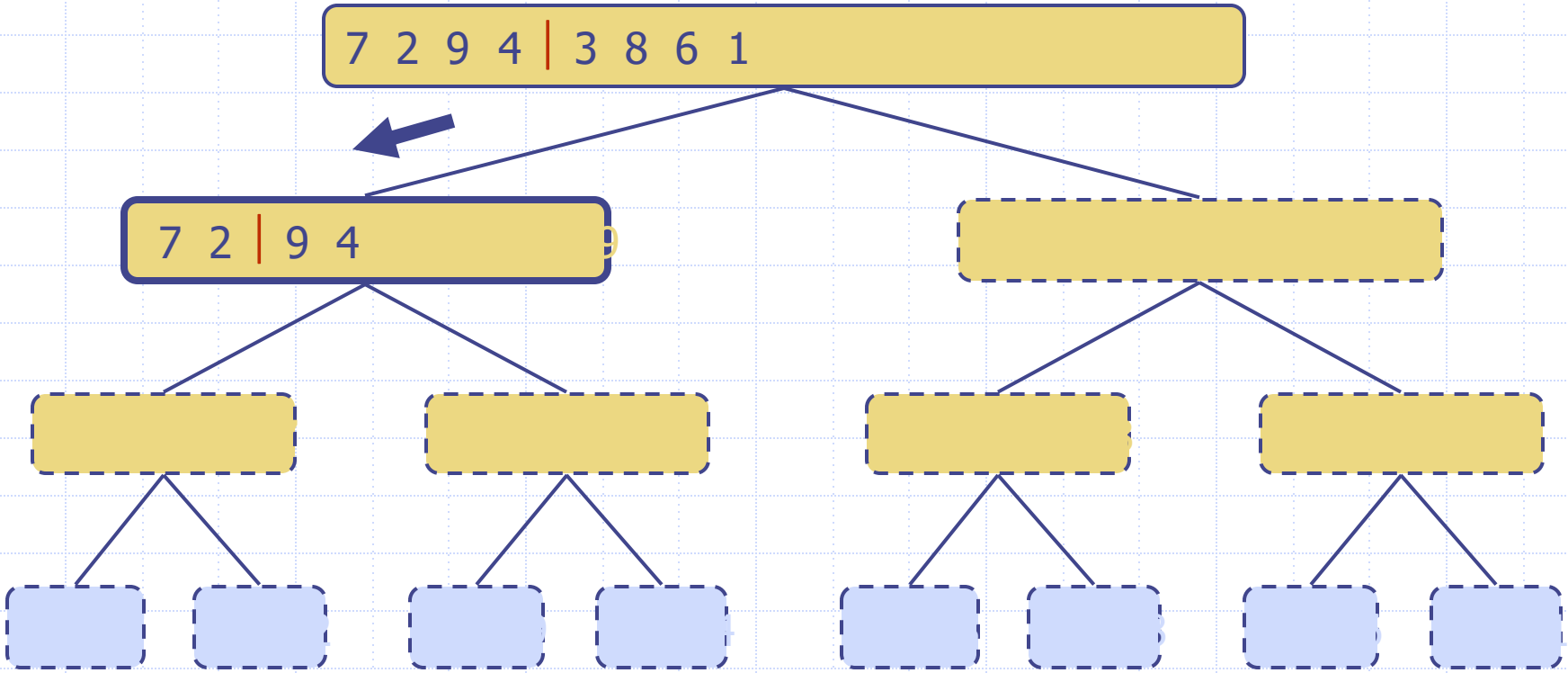
Execution Example

- Partition



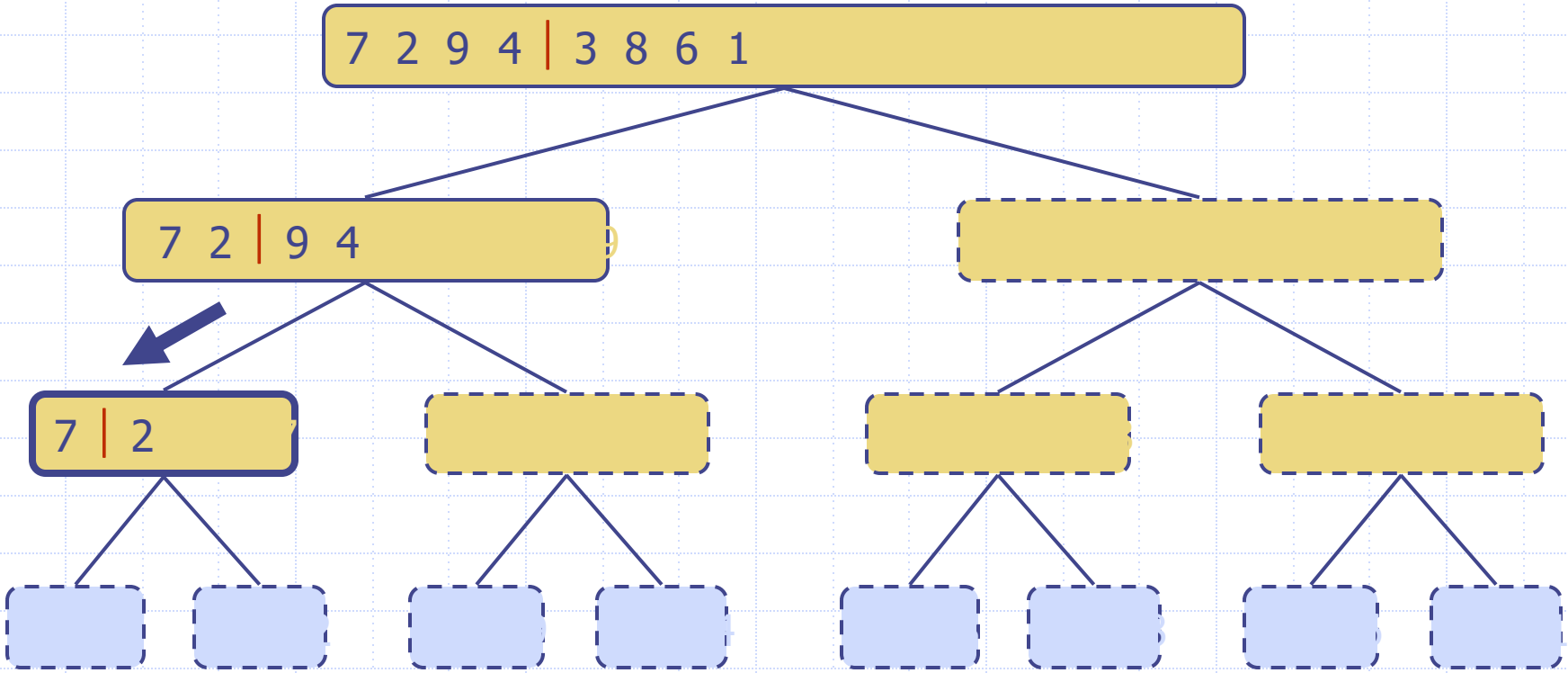
Execution Example (cont.)

- Recursive call, partition



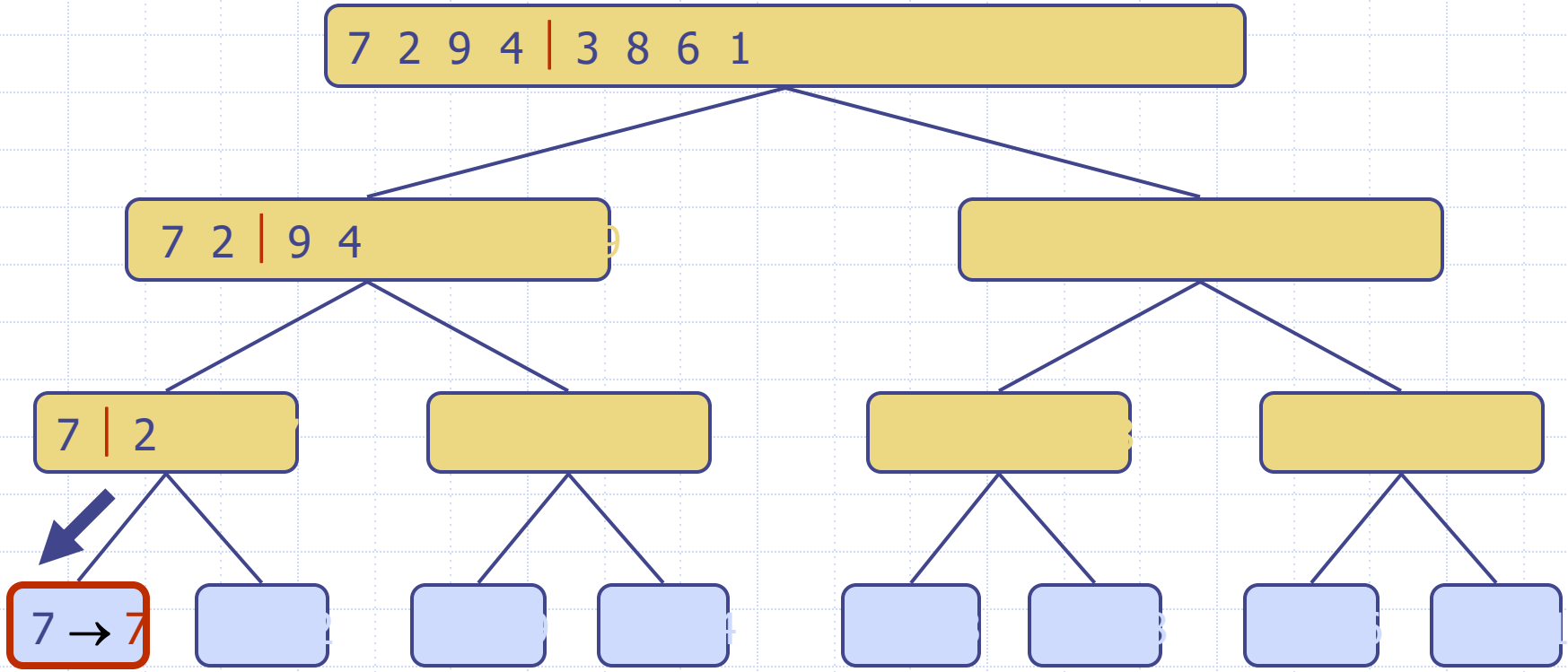
Execution Example (cont.)

- Recursive call, partition



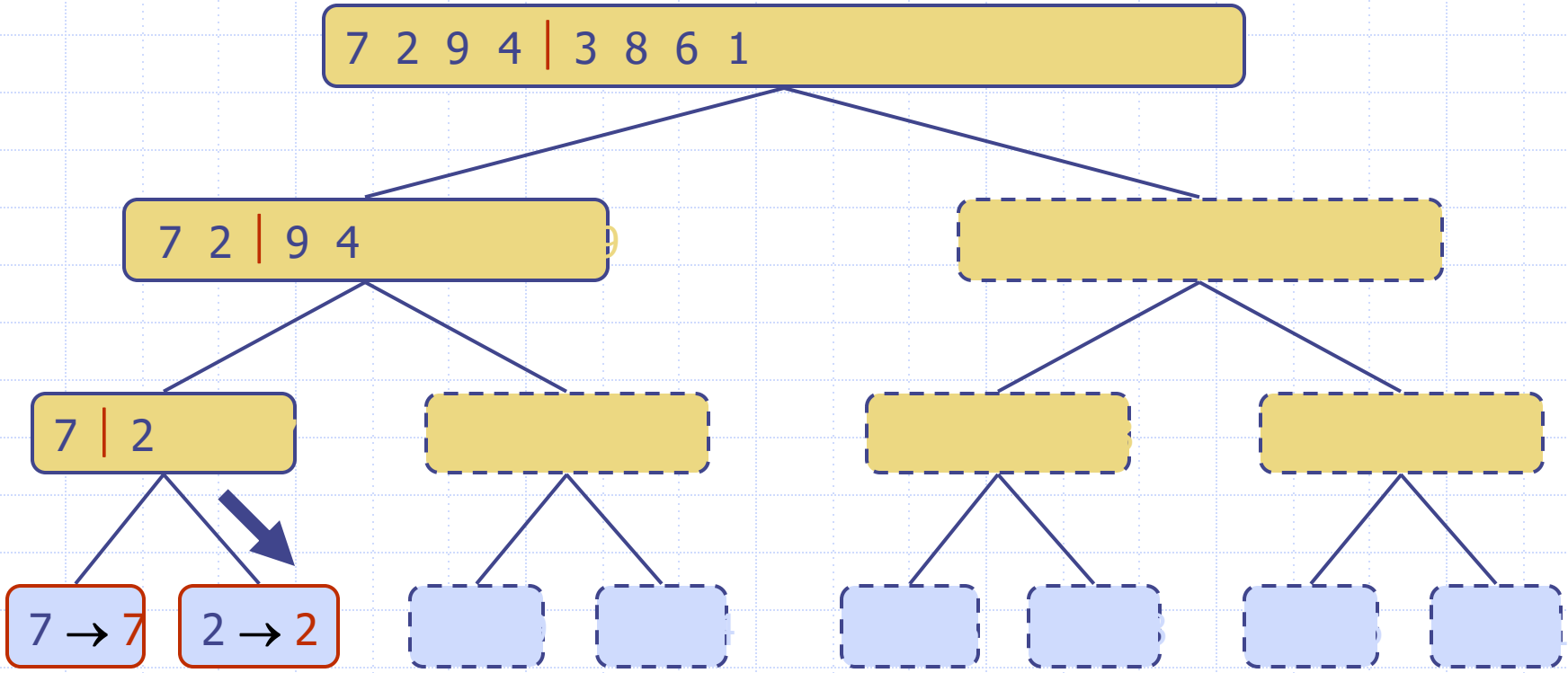
Execution Example (cont.)

- Recursive call, base case



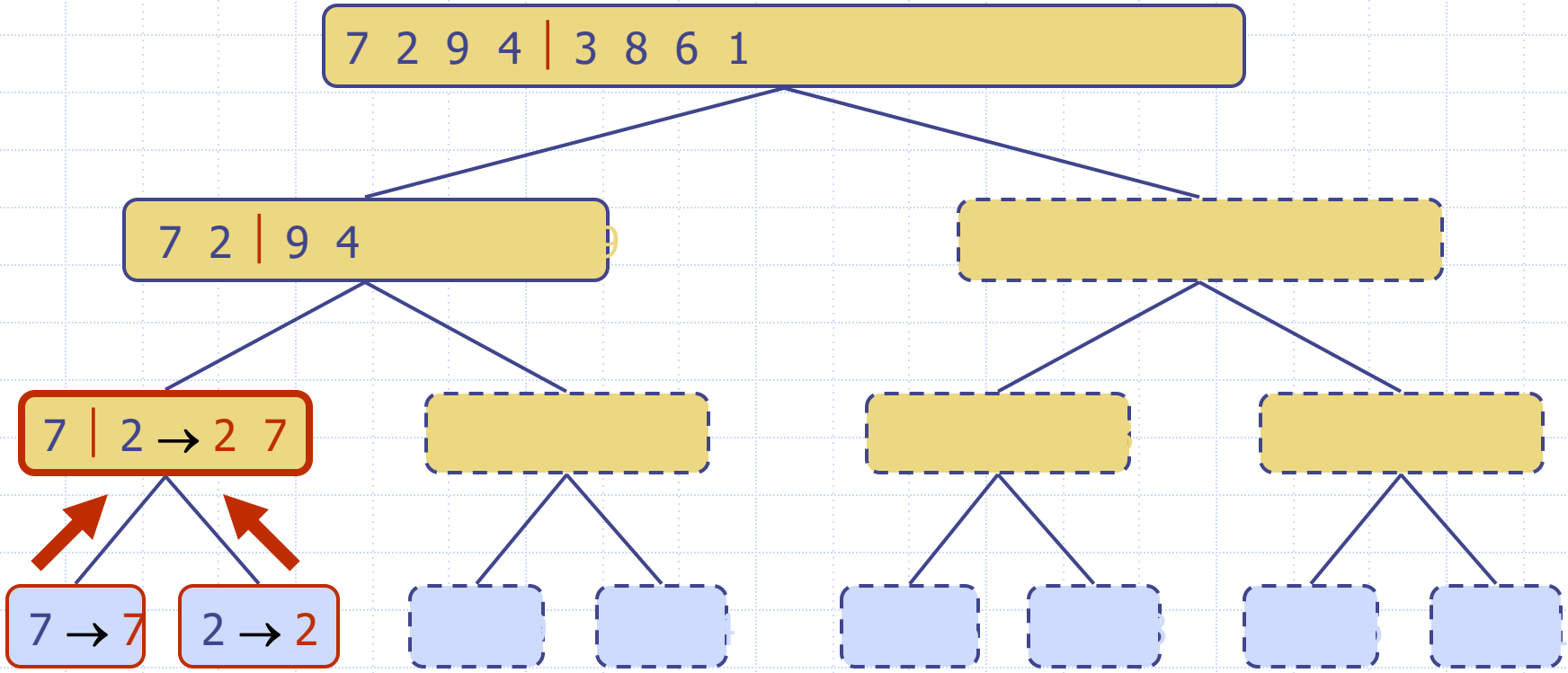
Execution Example (cont.)

- Recursive call, base case



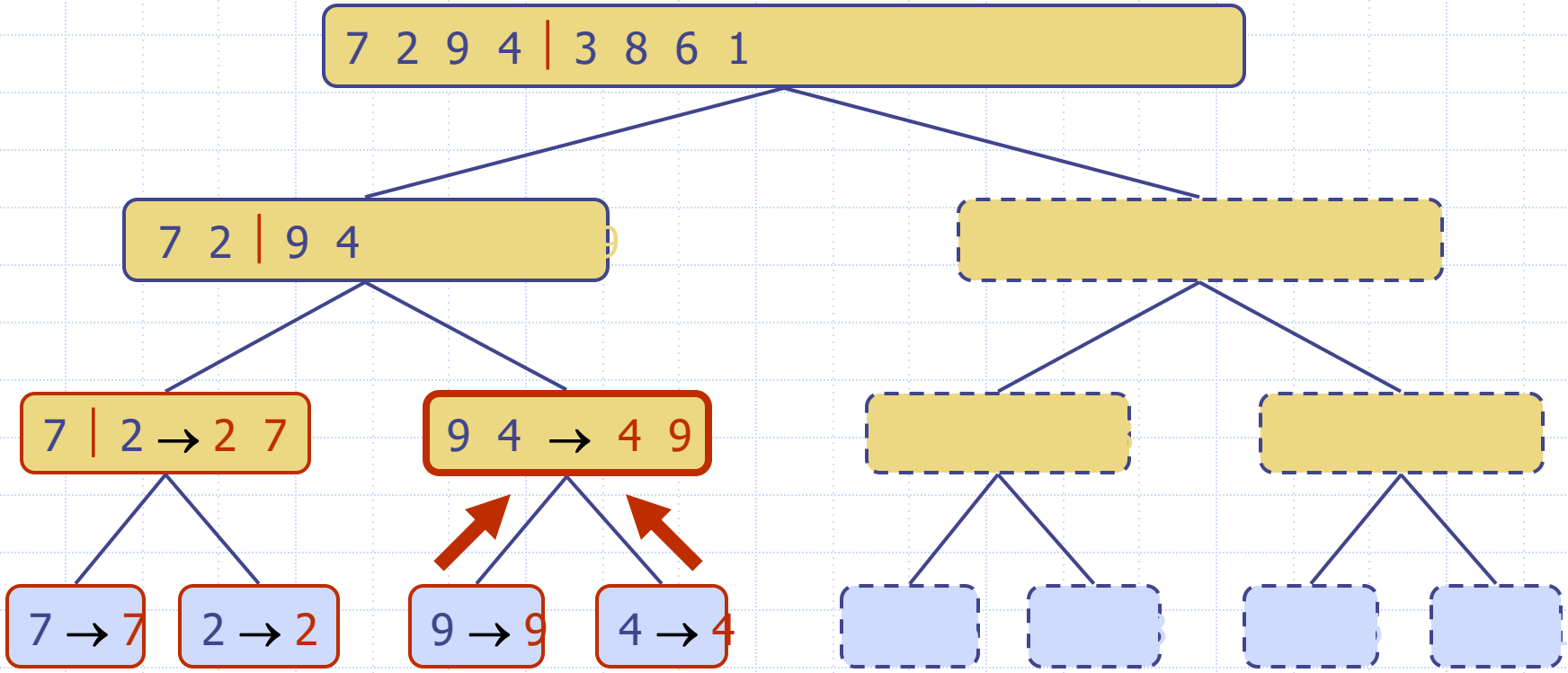
Execution Example (cont.)

- Merge



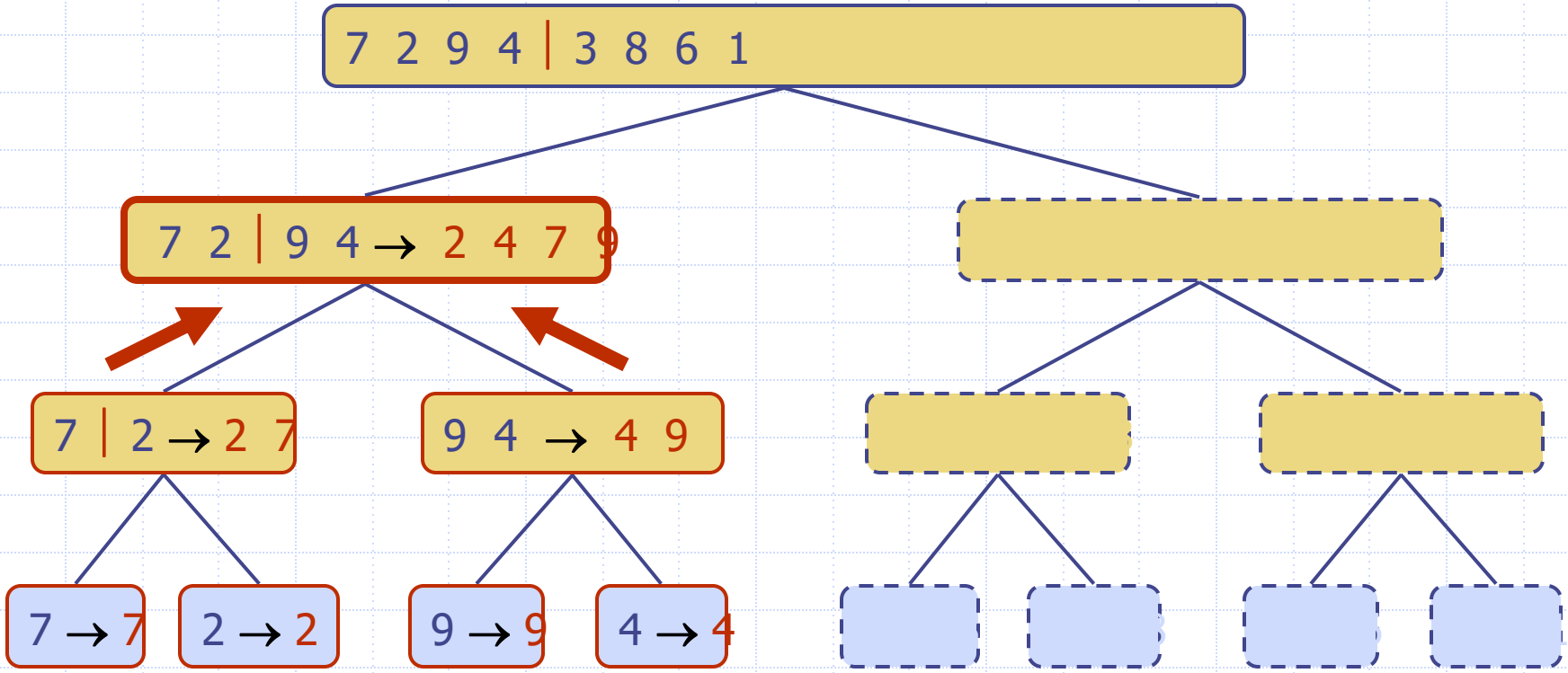
Execution Example (cont.)

- Recursive call, …, base case, merge



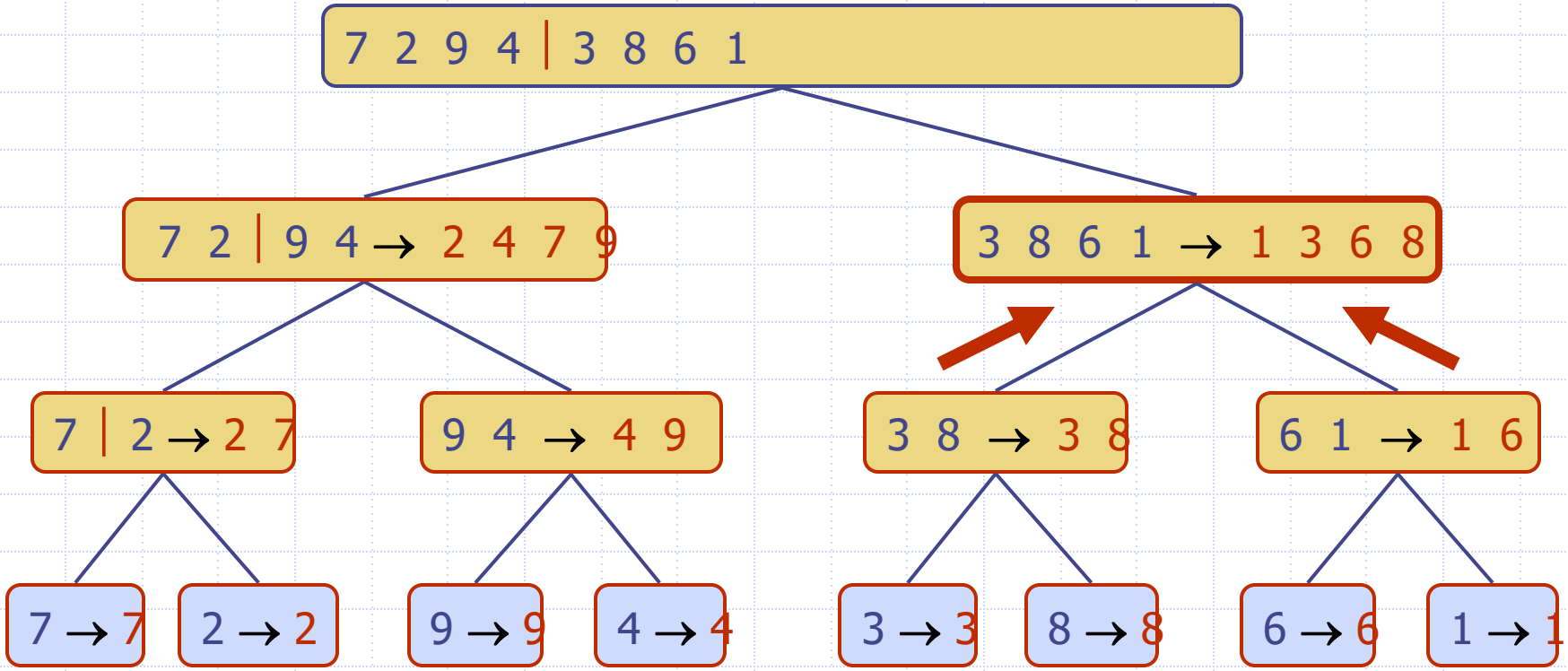
Execution Example (cont.)

- Merge



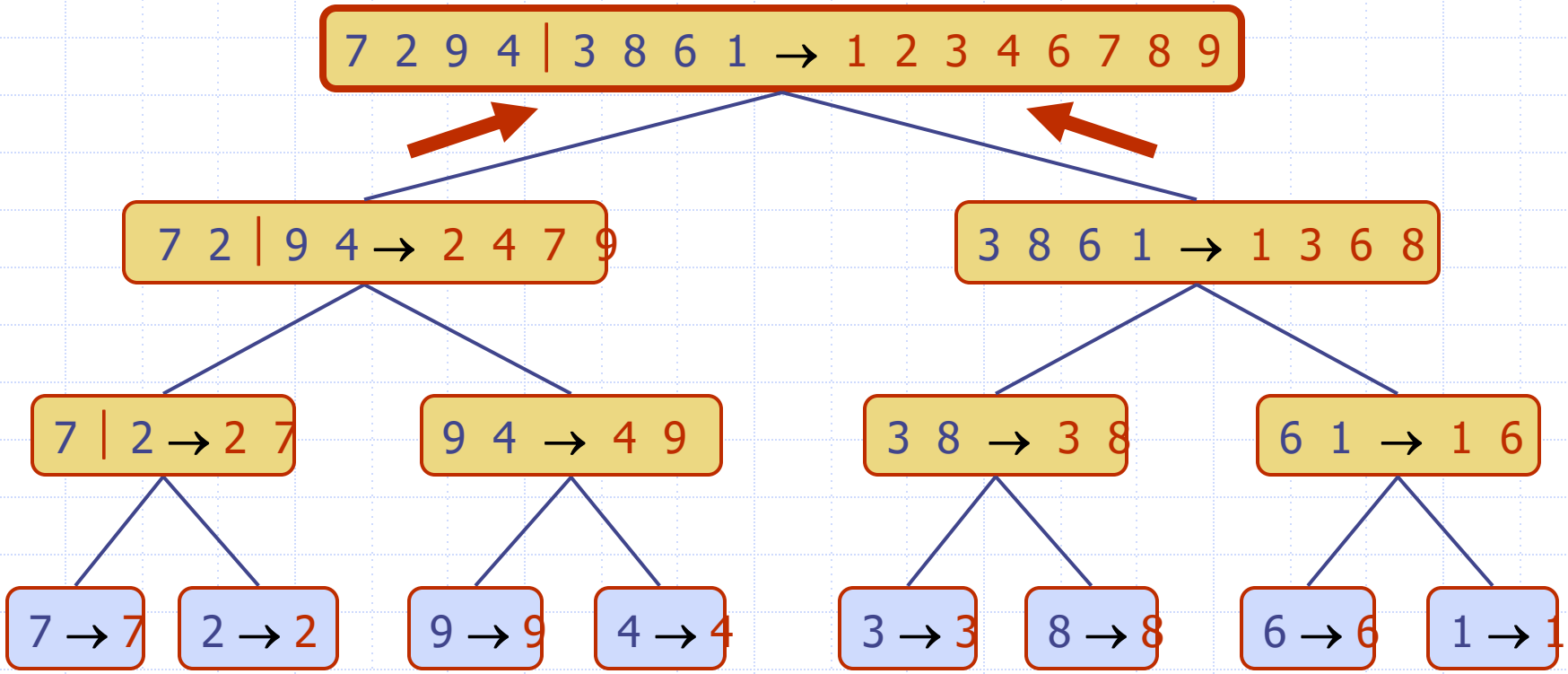
Execution Example (cont.)

- Recursive call, …, merge, merge

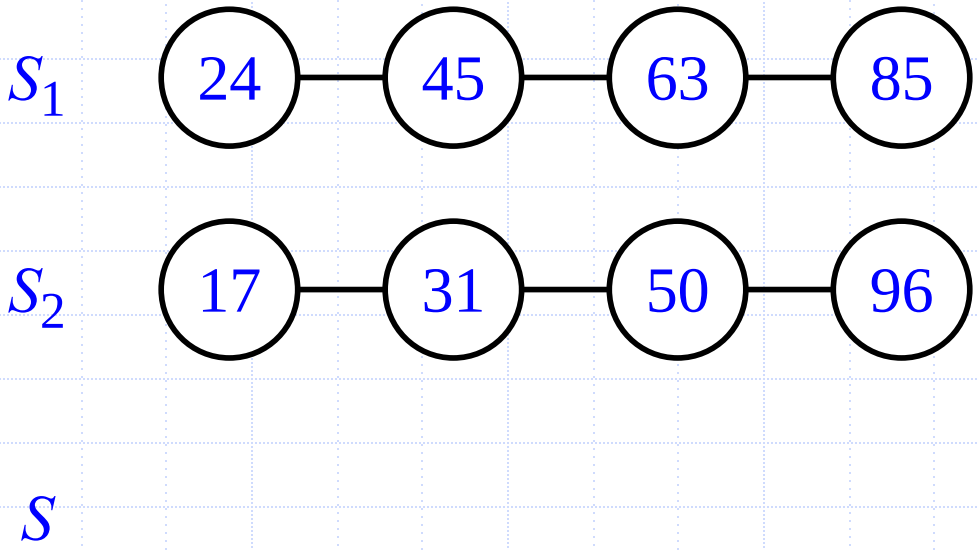


Execution Example (cont.)

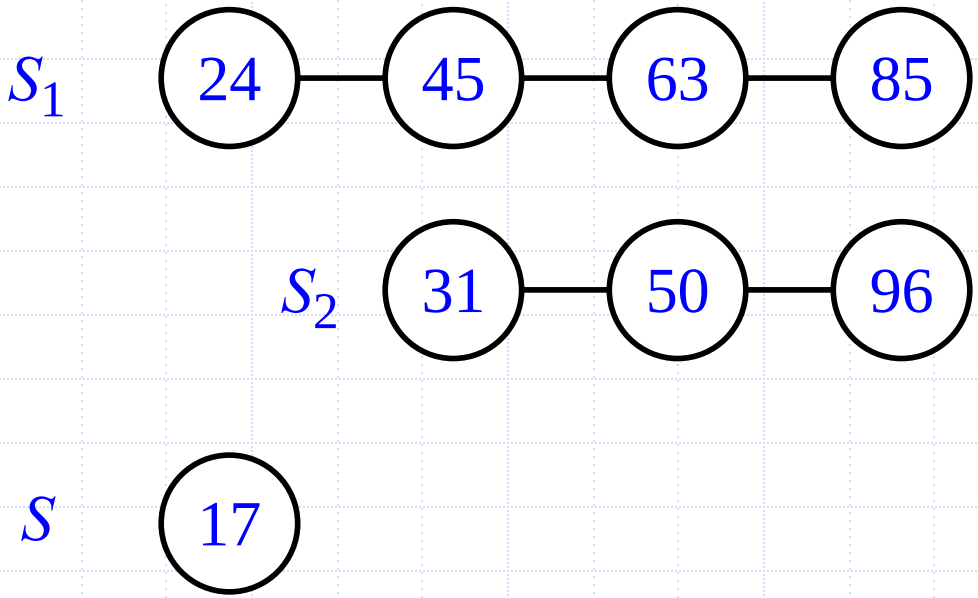
- Merge



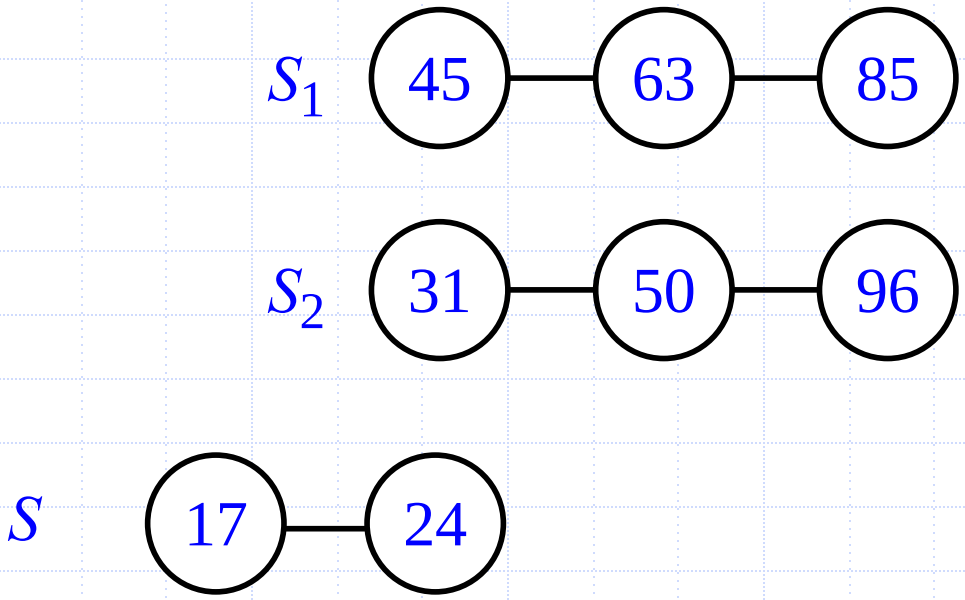
Merge - Example



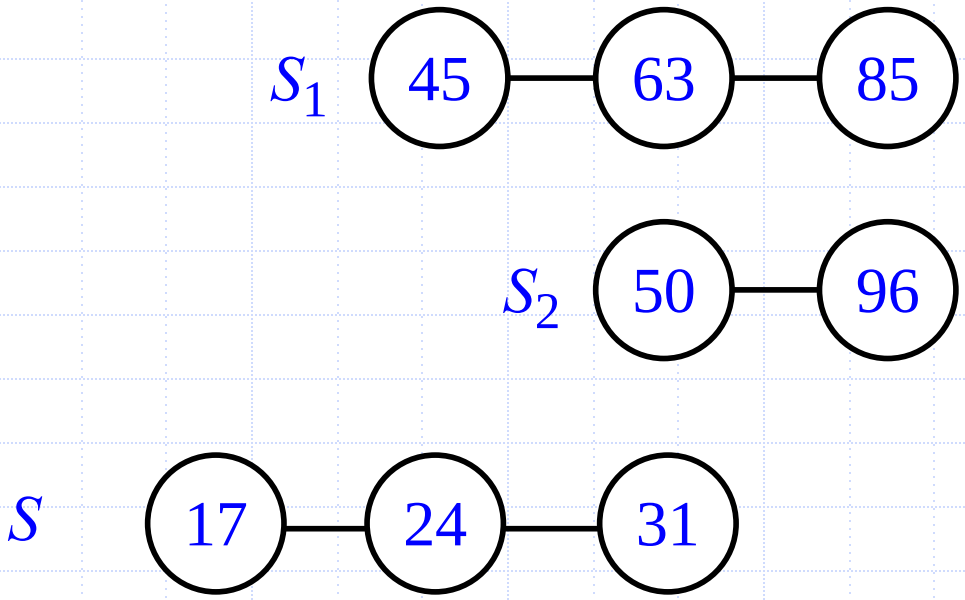
Merge - Example



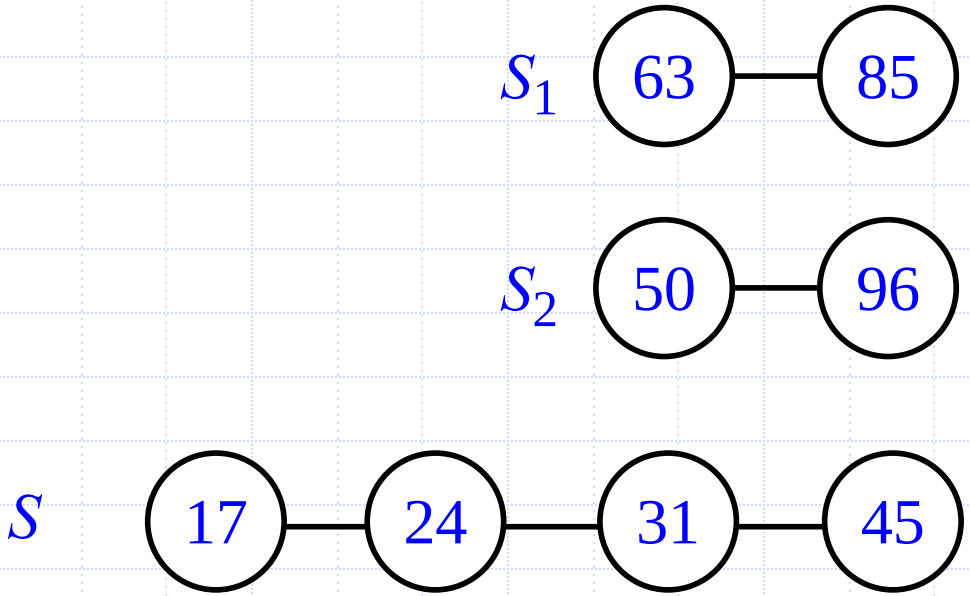
Merge - Example



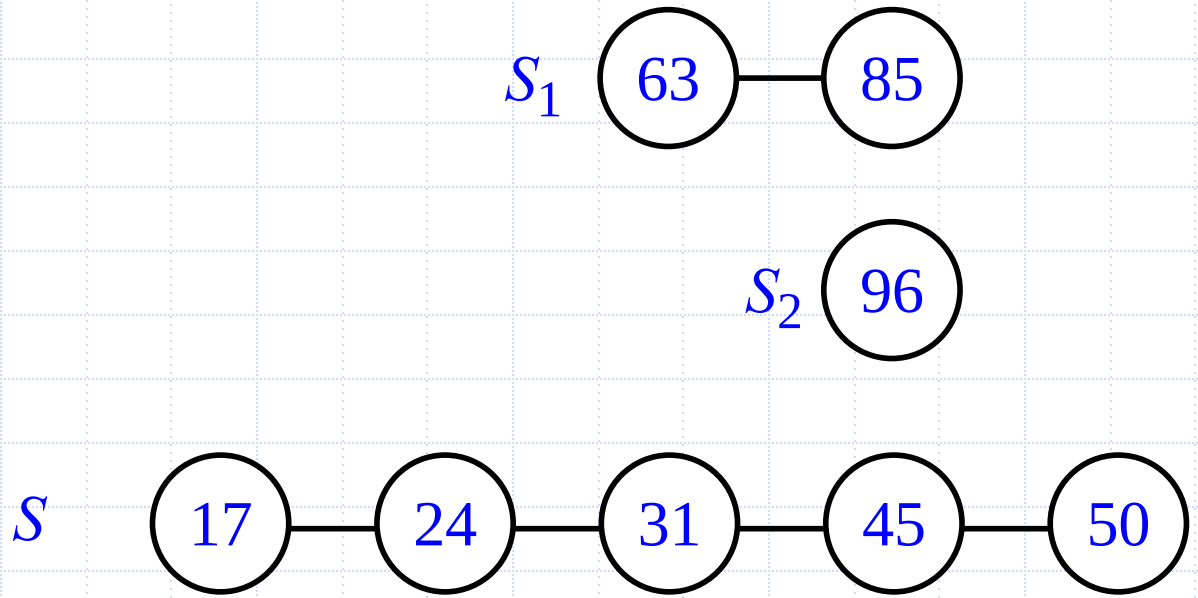
Merge - Example



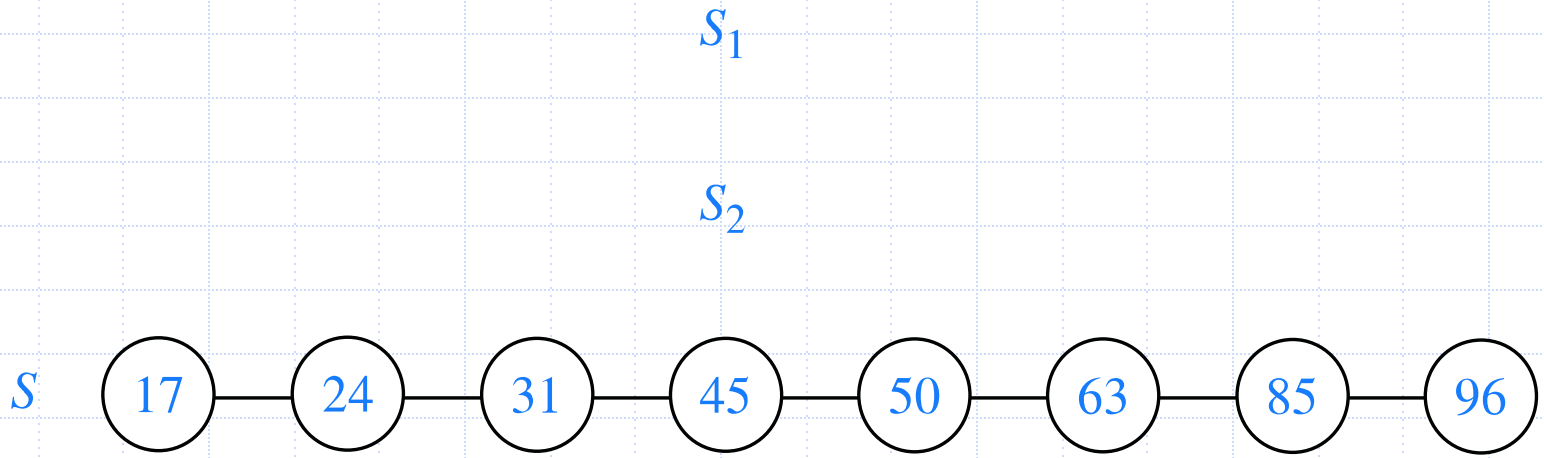
Merge - Example



Merge - Example



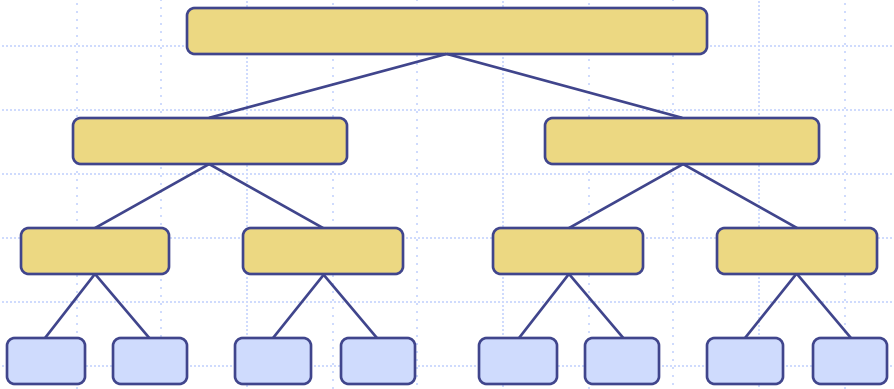
Merge - Example



Analysis of Merge-Sort

- The height h of the merge-sort tree is $O(\log n)$
  - at each recursive call we divide in half the sequence,
- The overall amount of work done at the nodes of depth i is $O(n)$
  - we partition and merge 2^i sequences of size $n/2^i$
  - we make 2^{i+1} recursive calls
- Thus, the total running time of merge-sort is $O(n \log n)$

depth	#seqs	size
0	1	n
1	2	$n/2$
i	2^i	$n/2^i$
…	…	…



Recurrences

- Running times of algorithms with Recursive calls can be described using recurrences.
- A recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs.
- T(n)
  - time for solving the trivial problem (base case)
  - number of problems * T(n/sub-problem size factor)+dividing+combining

Recurrence Equation Analysis

- The conquer step of merge-sort consists of merging two sorted sequences, each with $n/2$ elements and implemented by means of a doubly linked list, takes at most bn steps, for some constant b.
- Likewise, the basis case ($n < 2$) will take at b most steps.
- Therefore, if we let $T(n)$ denote the running time of merge-sort:

$$T(n) = \begin{cases} b & \text{if } n < 2 \\ 2T(n/2) + bn & \text{if } n \geq 2 \end{cases}$$

- We can therefore analyze the running time of merge-sort by finding a closed form solution to the above equation.
  - That is, a solution that has $T(n)$ only on the left-hand side.

Iterative Substitution

- In the iterative substitution, or "plug-and-chug," technique, we iteratively apply the recurrence equation to itself and see if we can find a pattern:

$$\begin{aligned} T(n) &= 2T(n/2) + bn \\ &= 2(2T(n/2^2)) + b(n/2)) + bn \\ &= 2^2 T(n/2^2) + 2bn \\ &= 2^3 T(n/2^3) + 3bn \\ &= 2^4 T(n/2^4) + 4bn \\ &= \ldots \\ &= 2^i T(n/2^i) + ibn \end{aligned}$$

- Note that base, T(n)=b, case occurs when $2^i = n$. That is, i = log n.
- Thus, T(n) is O(n log n).

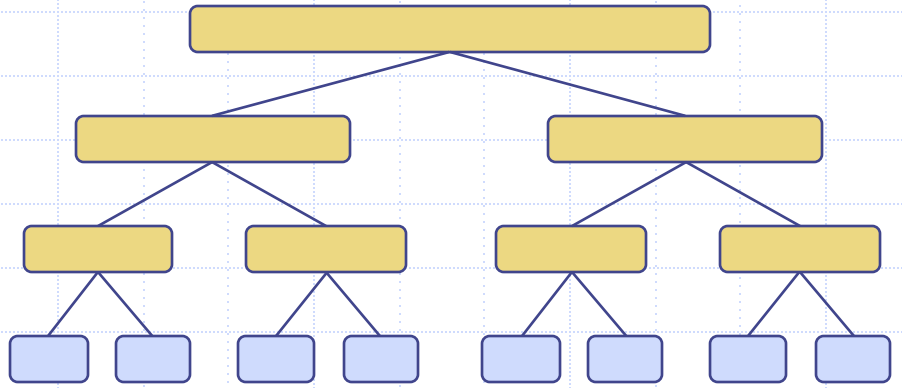
$$T(n) = bn + bn \log n$$

The Recursion Tree

- Draw the recursion tree for the recurrence relation and look for a pattern:

$$T(n) = \begin{cases} b & \text{if } n < 2 \\ 2T(n/2) + bn & \text{if } n \geq 2 \end{cases}$$

depth	T's	size	time
0	1	n	bn
1	2	$n/2$	bn
i	2^i	$n/2^i$	bn
…	…	…	…



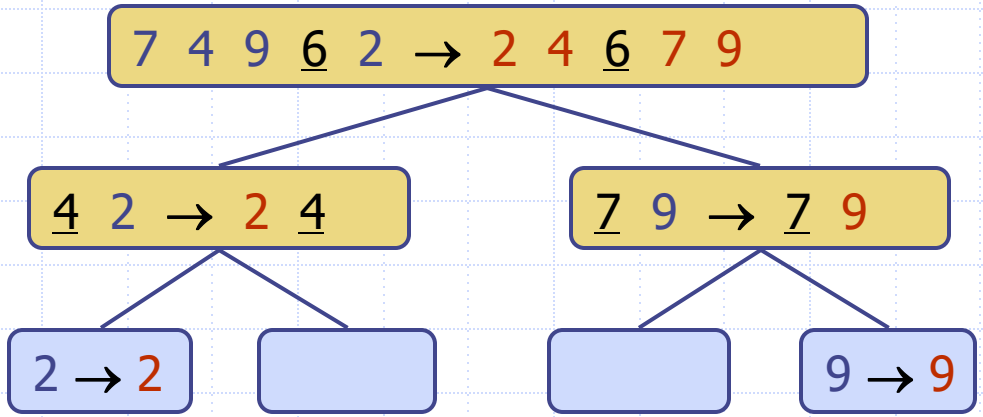
Total time = $bn + bn \log n$

(last level plus all previous levels)

Summary of Sorting Algorithms

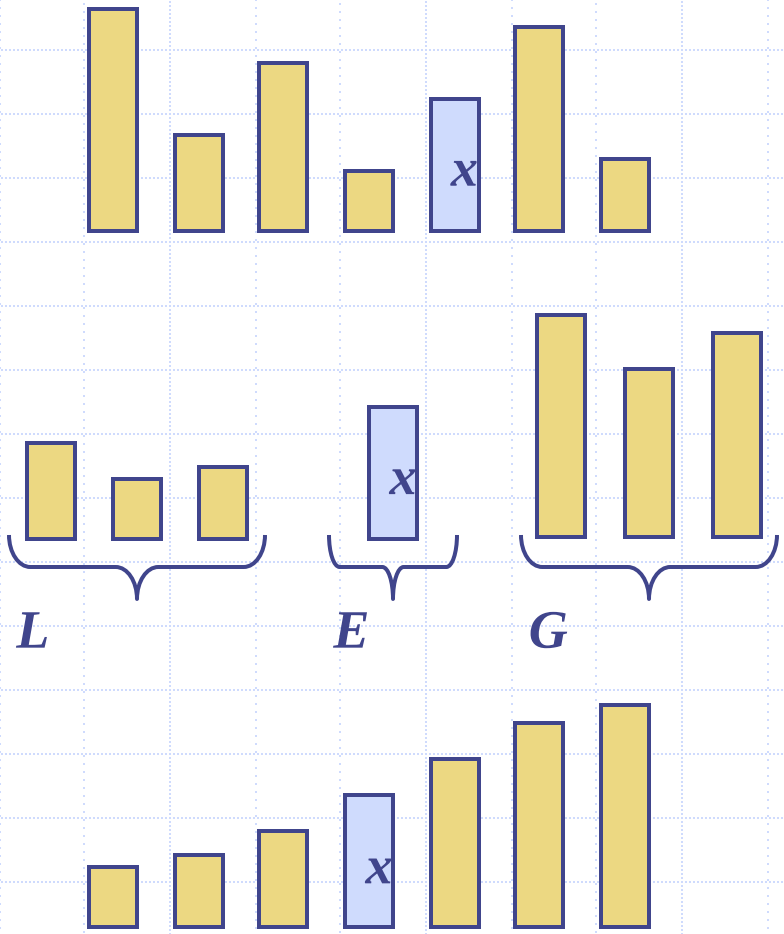
Algorithm	Time	Notes
selection-sort	$O(n^2)$	▪ slow ▪ in-place ▪ for small data sets (< 1K)
insertion-sort	$O(n^2)$	▪ slow ▪ in-place ▪ for small data sets (< 1K)
heap-sort	$O(n \log n)$	▪ fast ▪ in-place ▪ for large data sets (1K — 1M)
merge-sort	$O(n \log n)$	▪ fast ▪ sequential data access ▪ for huge data sets (> 1M)

Quick-Sort



Quick-Sort

- Quick-sort is a randomized sorting algorithm based on the divide-and-conquer paradigm:
  - Divide: pick a random element x (called pivot) and partition S into
    - L elements less than x
    - E elements equal x
    - G elements greater than x
  - Recur: sort L and G
  - Conquer: join L, E and G



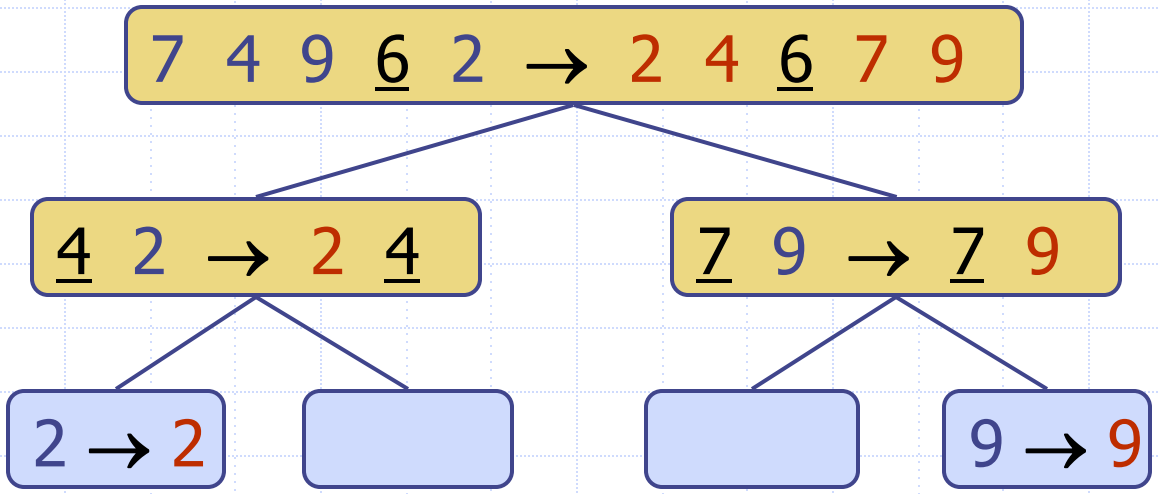
Partition

- We partition an input sequence as follows:
  - We remove, in turn, each element y from S and
  - We insert y into L, E or G, depending on the result of the comparison with the pivot x
- Each insertion and removal is at the beginning or at the end of a sequence, and hence takes $O(1)$ time
- Thus, the partition step of quick-sort takes $O(n)$ time

Algorithm *partition*(S, p)
  Input sequence S, position p of pivot
  Output subsequences L, E, G of the elements of S less than, equal to, or greater than the pivot, resp.
  $L, E, G \leftarrow$ empty sequences
  $x \leftarrow S.remove(p)$
  while $\neg S.isEmpty()$
    $y \leftarrow S.remove(S.first())$
    if $y < x$
      $L.addLast(y)$
    else if $y = x$
      $E.addLast(y)$
    else { $y > x$ }
      $G.addLast(y)$
  return L, E, G

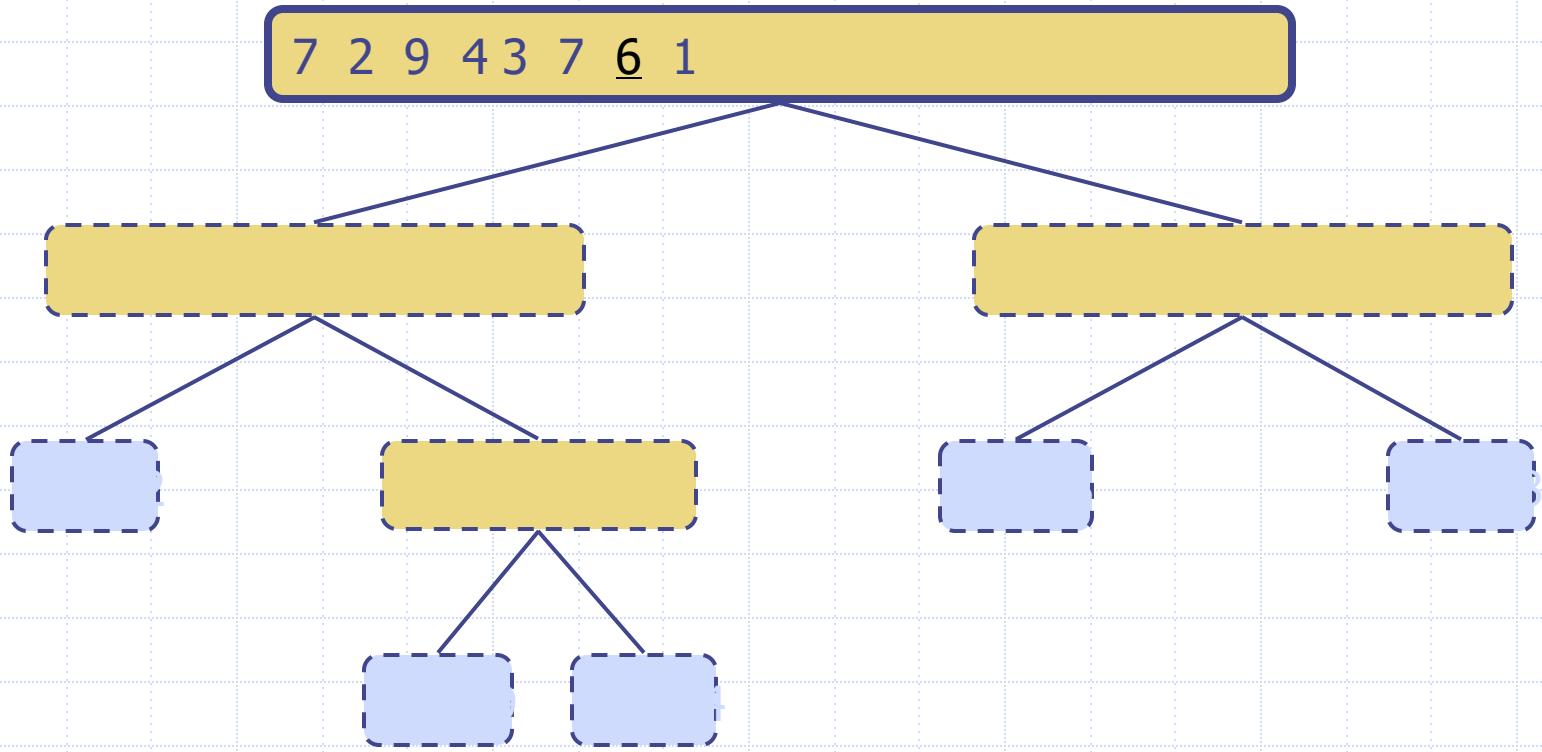
Quick-Sort Tree

- An execution of quick-sort is depicted by a binary tree
  - Each node represents a recursive call of quick-sort and stores
    - Unsorted sequence before the execution and its pivot
    - Sorted sequence at the end of the execution
  - The root is the initial call
  - The leaves are calls on subsequences of size 0 or 1



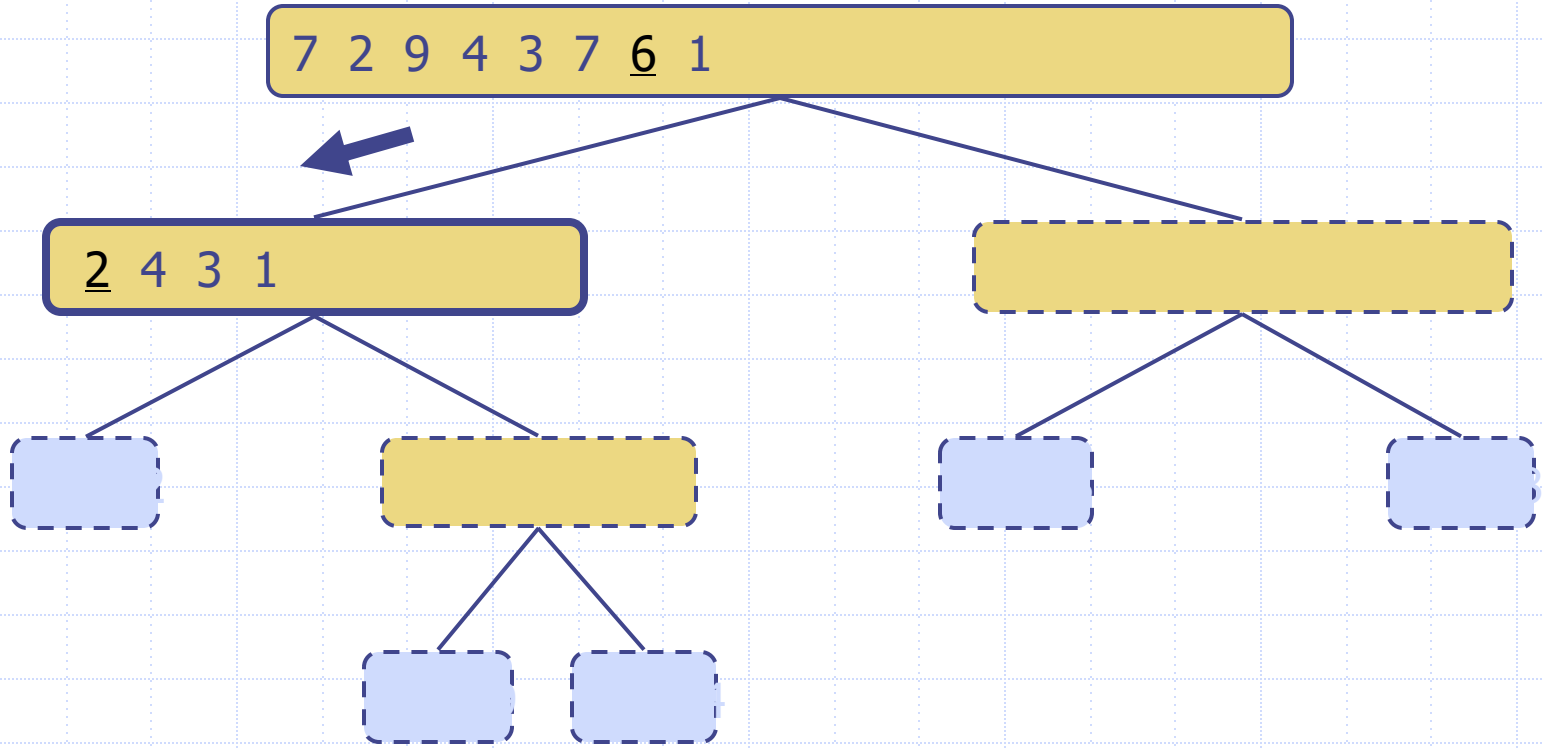
Execution Example

- Pivot selection



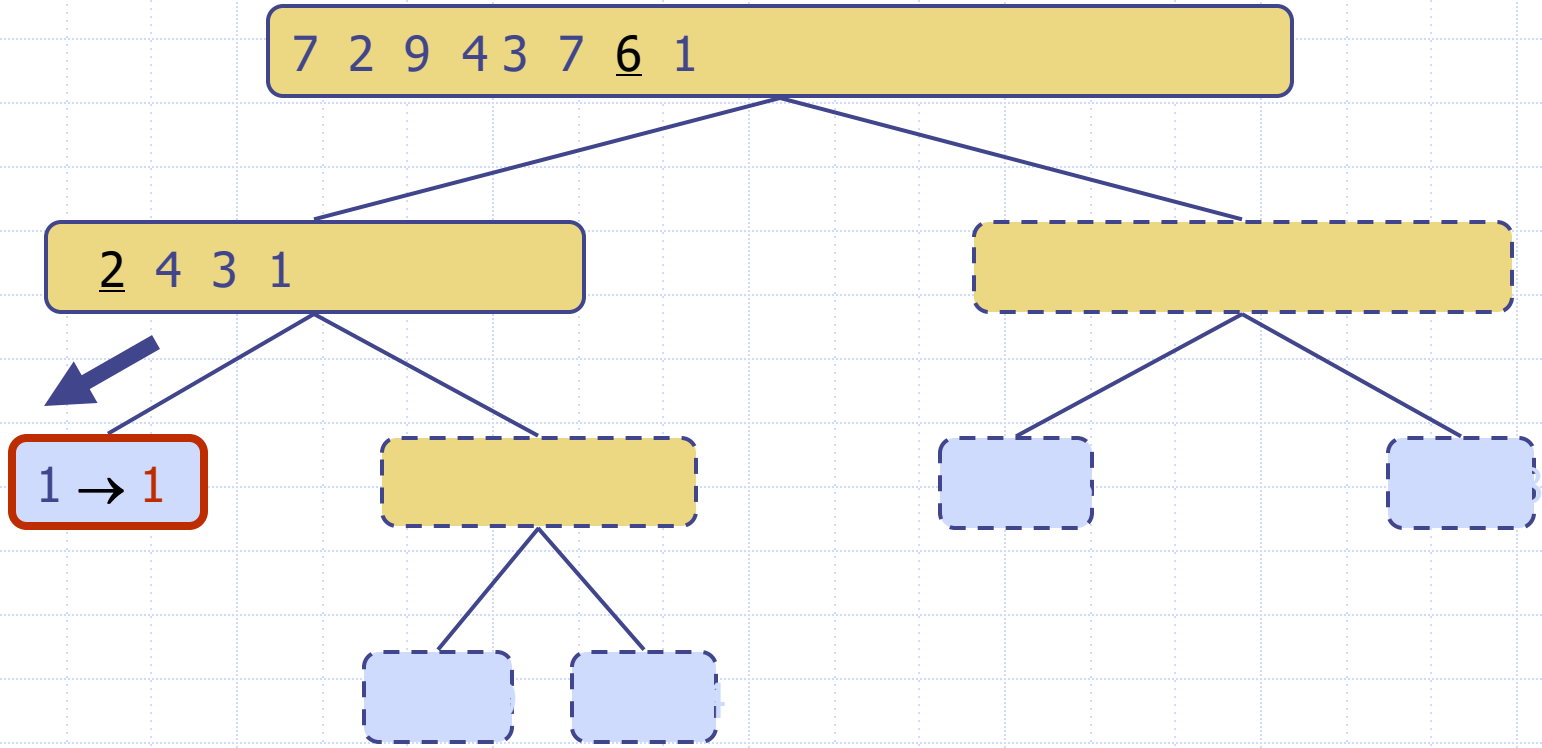
Execution Example (cont.)

- Partition, recursive call, pivot selection



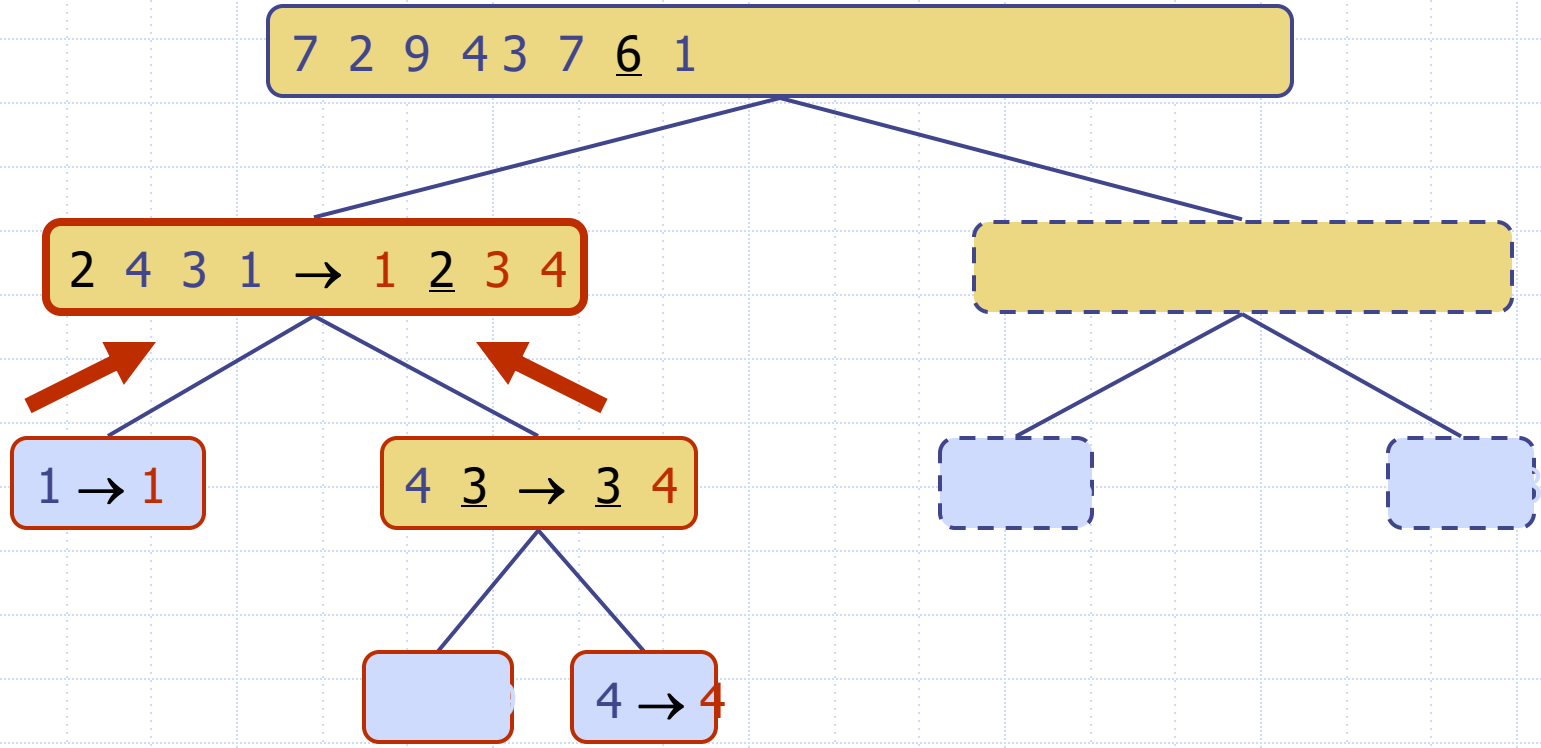
Execution Example (cont.)

- Partition, recursive call, base case



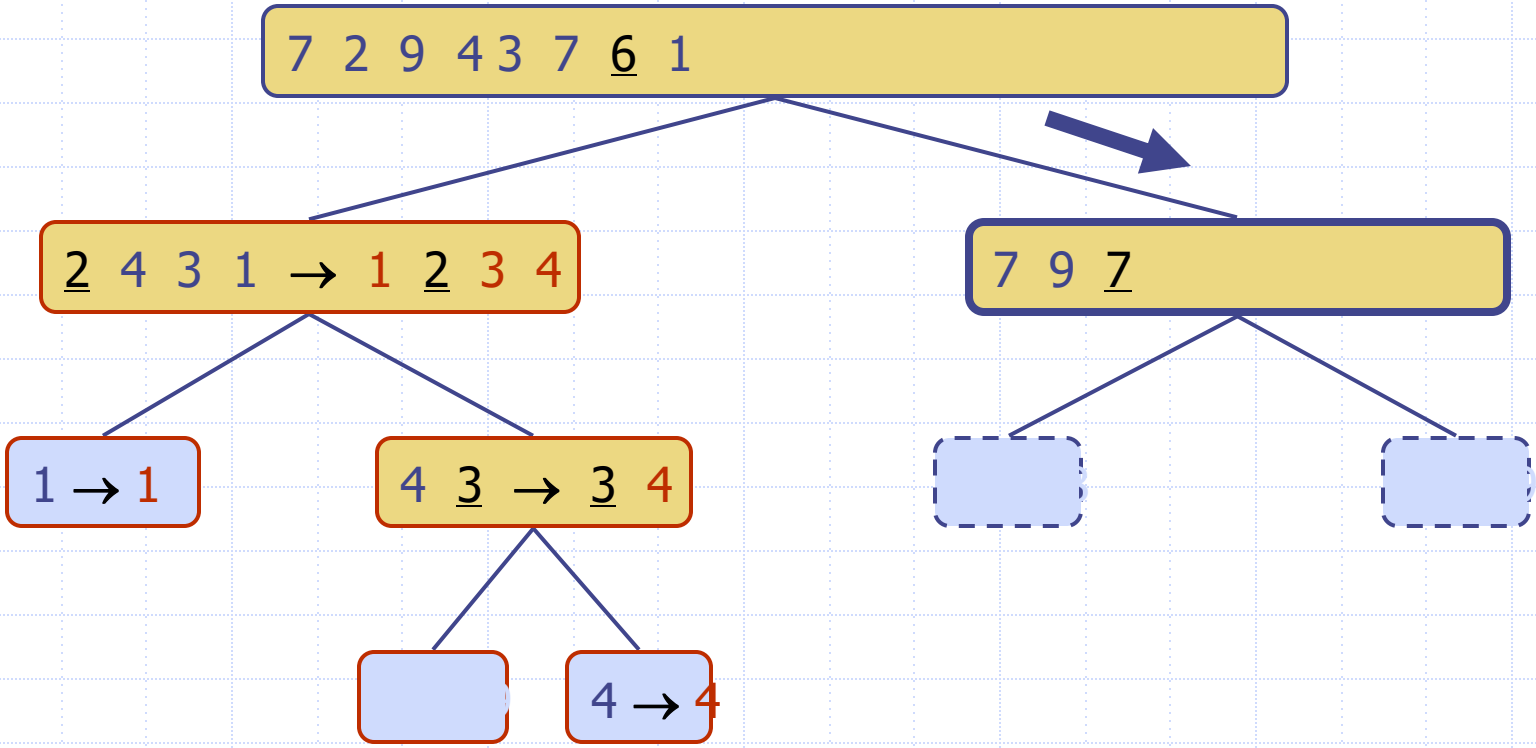
Execution Example (cont.)

- Recursive call, …, base case, join



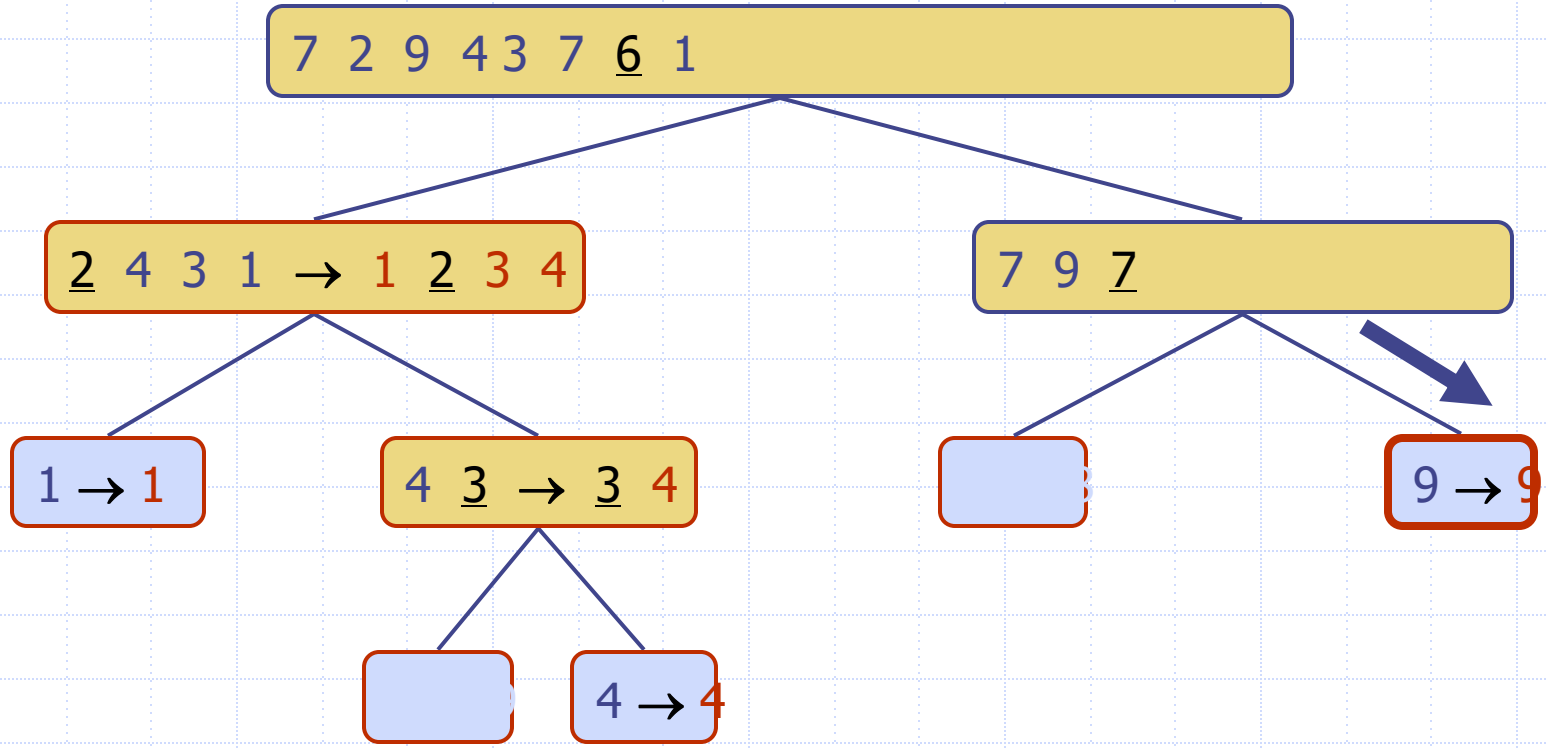
Execution Example (cont.)

- Recursive call, pivot selection



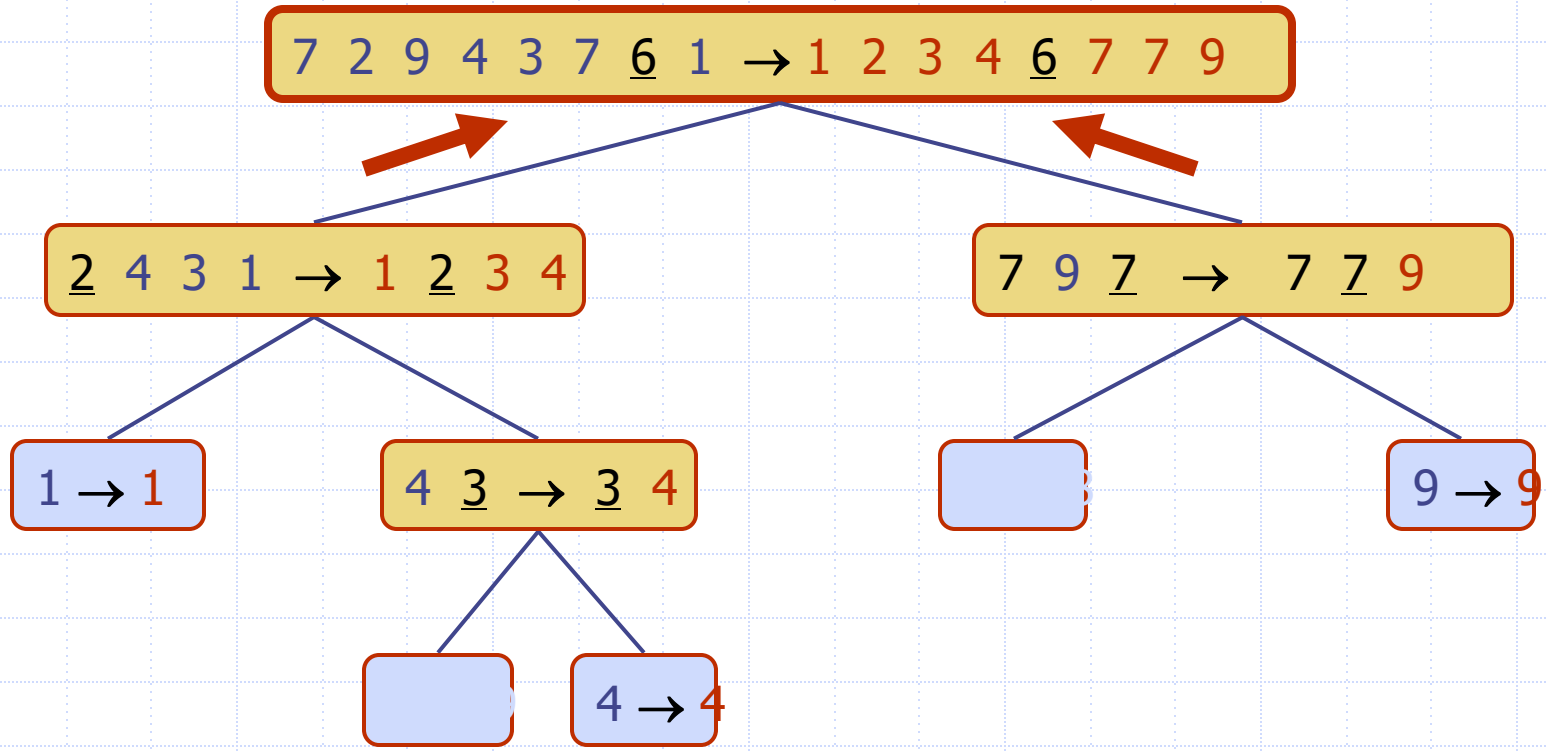
Execution Example (cont.)

- Partition, ..., recursive call, base case



Execution Example (cont.)

- Join, join

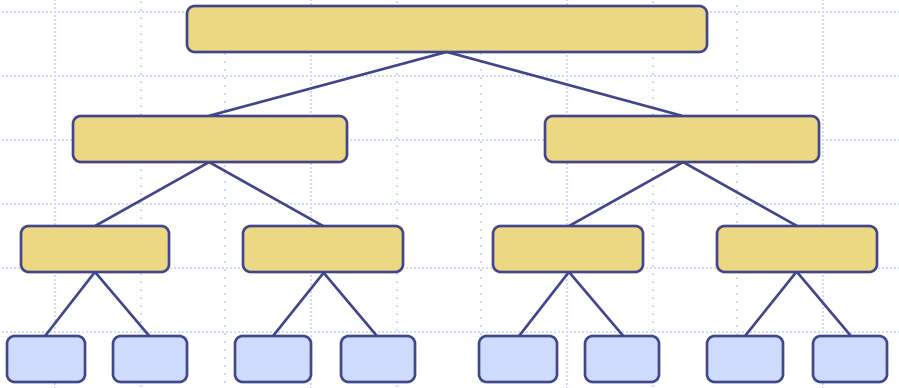


Best-case Running Time

- If we are lucky, Partition splits the array evenly

$$T(n) = \begin{cases} b & \text{if } n < 2 \\ 2T(n/2) + bn & \text{if } n \geq 2 \end{cases}$$

depth	T's	size	time
0	1	n	bn
1	2	$n/2$	bn
i	2^i	$n/2^i$	bn
…	…	…	…



Total time = $bn + bn \log n$

(last level plus all previous levels)

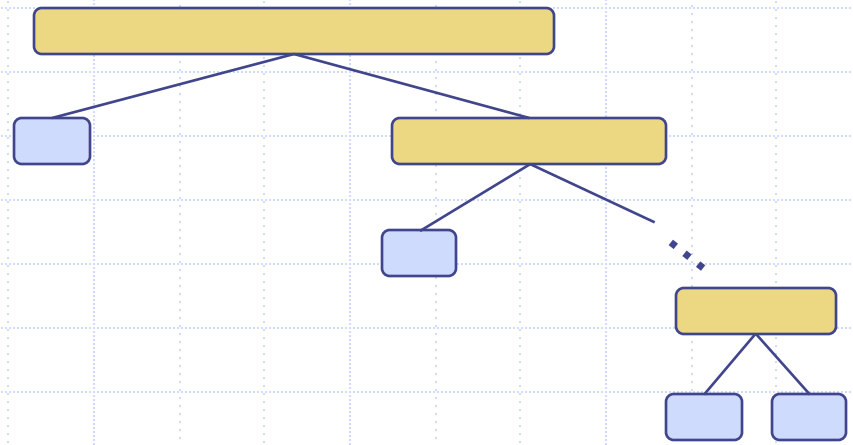
Worst-case Running Time

- The worst case for quick-sort occurs when the pivot is the unique minimum or maximum element and the input sequence is in ascending or descending order
- One of L and G has size $n-1$ and the other has size 1
- The running time is proportional to the sum

$$n + (n-1) + \ldots + 2 + 1$$

- Thus, the worst-case running time of quick-sort is $O(n^2)$

depth	time
0	n
1	$n-1$
…	…
$n-1$	1



Expected Time Analysis

- Fix the input
  - expectation is over different randomly selected pivots
- Let T(n) be the expected number of comparisons needed to quicksort n numbers
- Probability of each split – 1/n
  - T(n) = T(j-1)+T(n-j)+n-1 with probability 1/n

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

$$= \frac{2}{n}\sum_{j=0}^{n-1}T(j)+n-1$$

Expected Time Analysis (2)

- Since

$$T(n-1) = \frac{2}{n-1}\sum_{j=0}^{n-2} T(j) + n - 2$$

- we have

$$\frac{2}{n}\sum_{j=0}^{n-2} T(j) = \frac{n-1}{n}(T(n-1) - n + 2)$$

- substituting in the expression for T(n)

$$T(n) = \frac{n-1}{n}(T(n-1) - n + 2) + \frac{2}{n}T(n-1) + n - 1$$

$$= \frac{n+1}{n}T(n-1) + \frac{2(n-1)}{n}$$

Expected Time Analysis (3)

$$
\begin{aligned}
T(n) &= \frac{n+1}{n}T(n-1) + \frac{2(n-1)}{n} \\
&< \frac{n+1}{n}T(n-1) + 2 \\
&< \frac{n+1}{n}\left(\frac{n}{n-1}T(n-2) + 2\right) + 2 \\
&= \frac{n+1}{n-1}T(n-2) + \frac{2(n+1)}{n} + 2 \\
&< \frac{n+1}{n-2}T(n-3) + 2(n+1)\left(\frac{1}{n} + \frac{1}{n-1}\right) + 2 \\
&< \frac{n+1}{n-3}T(n-4) + 2(n+1)\left(\frac{1}{n} + \frac{1}{n-1} + \frac{1}{n-2}\right) + 2 \\
&< (n+1)T(0) + 2(n+1)\left(\frac{1}{n} + \frac{1}{n-1} + \ldots + \frac{1}{2}\right) + 2 \\
&= 2(n+1)\left(\frac{1}{n} + \frac{1}{n-1} + \ldots + \frac{1}{2}\right) + 2
\end{aligned}
$$

Expected Time Analysis

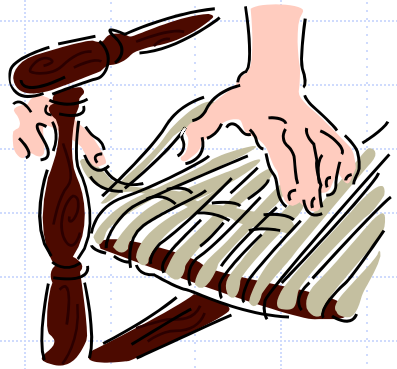
$$T(n) < 2(n+1)\left(\frac{1}{n} + \frac{1}{n-1} + \ldots + \frac{1}{2}\right) + 2$$

$$= 2(n+1)\int_1^n \frac{dx}{x} + 2$$

$$= 2(n+1)\log n + 2$$

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

In-Place Quick-Sort



- Quick-sort can be implemented to run in-place
- In the partition step, we use replace operations to rearrange the elements of the input sequence such that
  - the elements less than the pivot have rank less than h
  - the elements equal to the pivot have rank between h and k
  - the elements greater than the pivot have rank greater than k
- The recursive calls consider
  - elements with rank less than h
  - elements with rank greater than k

Algorithm *inPlaceQuickSort*(S, l, r)

Input sequence S, ranks l and r

Output sequence S with the elements of rank between l and r rearranged in increasing order

if $l \geq r$

return

$i \leftarrow$ a random integer between l and r

$x \leftarrow S.elemAtRank(i)$

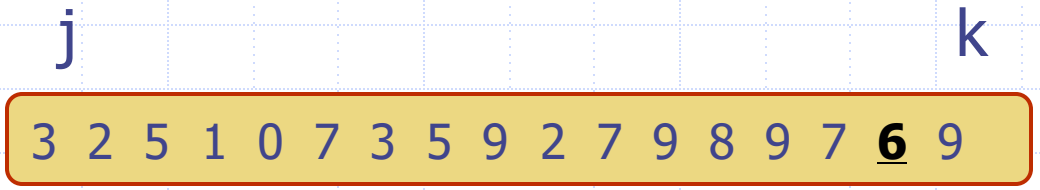
$(h, k) \leftarrow inPlacePartition(x)$

$inPlaceQuickSort(S, l, h - 1)$

$inPlaceQuickSort(S, k + 1, r)$

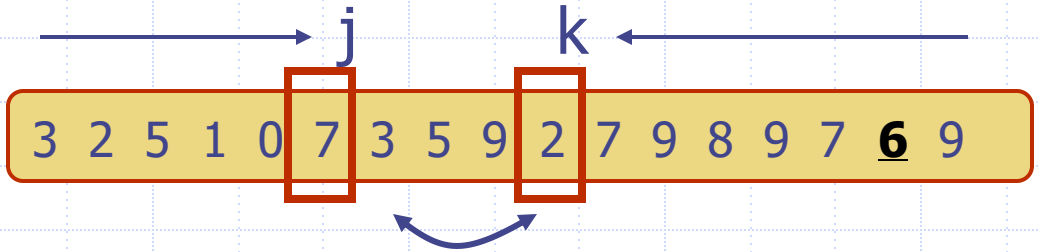
In-Place Partitioning

- Perform the partition using two indices to split S into L and E U G (a similar method can split E U G into E and G).



(pivot = 6)

- Repeat until j and k cross:
  - Scan j to the right until finding an element $\geq$ x.
  - Scan k to the left until finding an element < x.
  - Swap elements at indices j and k



Summary of Sorting Algorithms

Algorithm	Time	Notes
selection-sort	$O(n^2)$	▪ in-place ▪ slow (good for small inputs)
insertion-sort	$O(n^2)$	▪ in-place ▪ slow (good for small inputs)
quick-sort	$O(n \log n)$ expected	▪ in-place, randomized ▪ fastest (good for large inputs)
heap-sort	$O(n \log n)$	▪ in-place ▪ fast (good for large inputs)
merge-sort	$O(n \log n)$	▪ sequential data access ▪ fast (good for huge inputs)

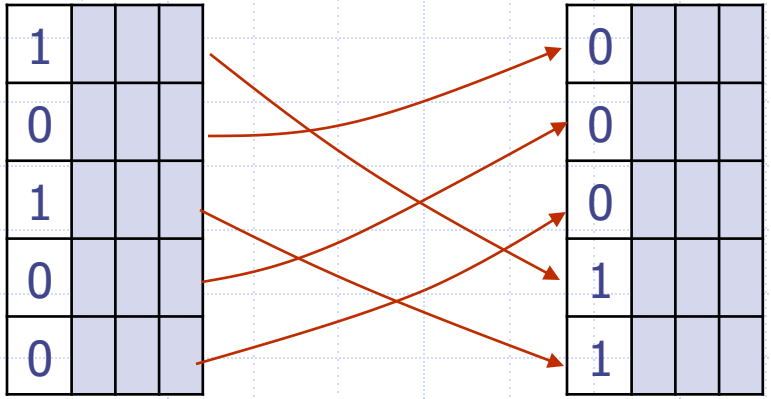
Radix Sort

- Considers structure of the keys
- Assume keys are represented in base M number system (M=radix) i.e., if M=1, the keys are represented in binary format.

- Sorting is performed by comparing bits in the same position
- Extension to keys that are alphanumeric strings

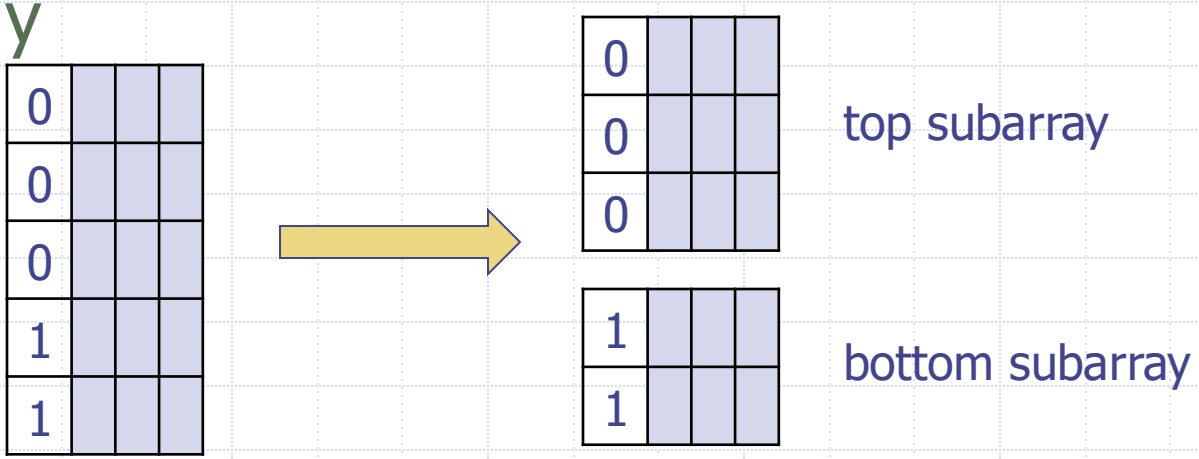
Radix Exchange Sort

- All the keys are represented with a fixed number of bits
- Examine bits from left to right
  - sort array with respect to leftmost bit



Radix Exchange Sort

- All the keys are represented with a fixed number of bits
- Examine bits from left to right
  - sort array with respect to leftmost bit
  - partition array

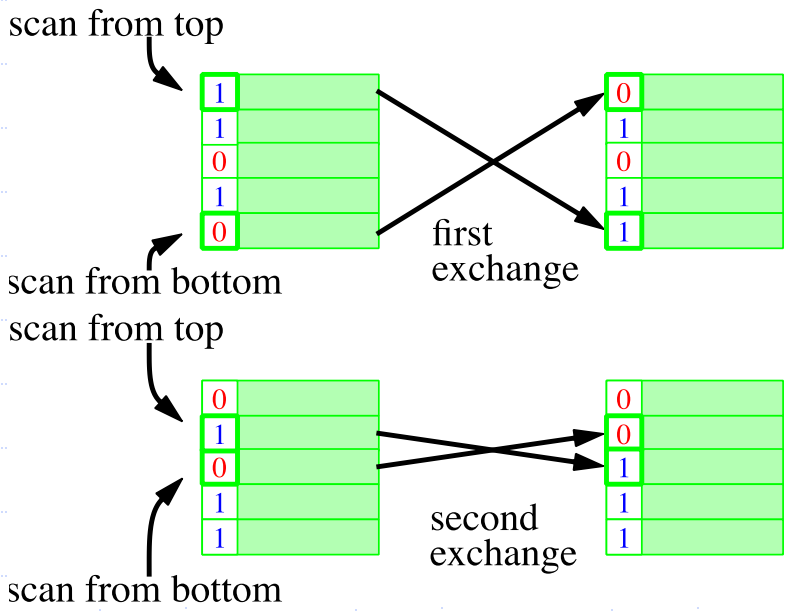


Radix Exchange Sort

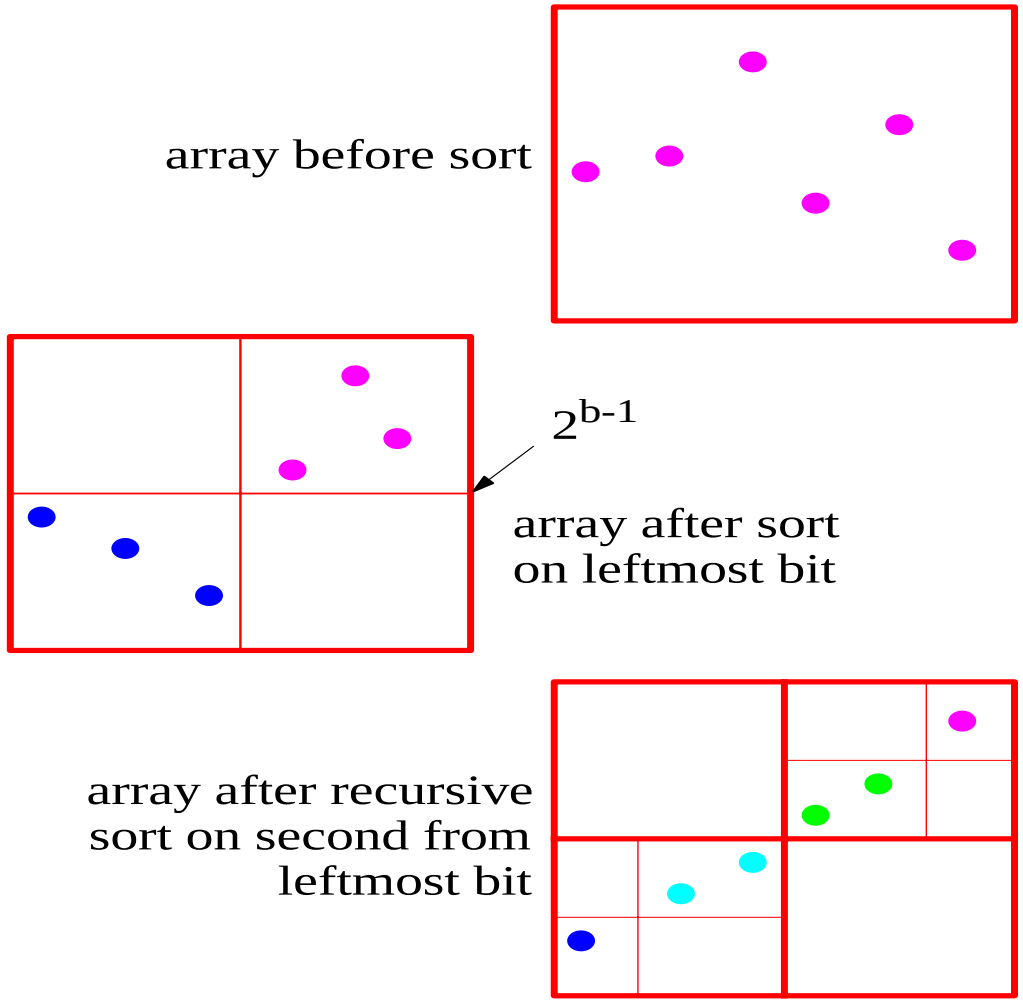
- All the keys are represented with a fixed number of bits
- Examine bits from left to right
  - sort array with respect to leftmost bit
  - partition array
  - recursion
    - recursively sort the top subarray, ignoring the leftmost bit
    - recursively sort the bottom subarray, ignoring the leftmost bit
  - Complexity – n numbers of b bits – O(b n)

Radix Exchange Sort

- Partition
  - repeat
    - scan top-down to find key starting with 1
    - scan bottom-up to find key starting with 0
    - swap the keys
  - scan till indices cross.
- Complexity is O(n)



Radix Exchange Sort



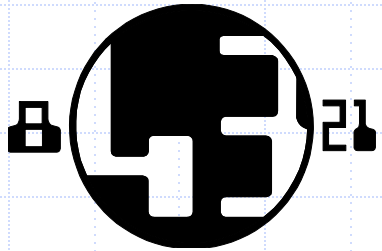
Radix Exchange Sort vs. Quicksort

- Similarities
  - partition arrays
  - recursively sort sub-arrays
- Differences
  - method of partitioning
    - radix exchange divides array based on greater than or less than 2^{b-1}
    - quicksort partitions based on greater than or less than some element of the array
  - Time complexity
    - radix exchange O(bn)
    - Quicksort
      - average case O(n log n)
      - worst case $O(n^2)$

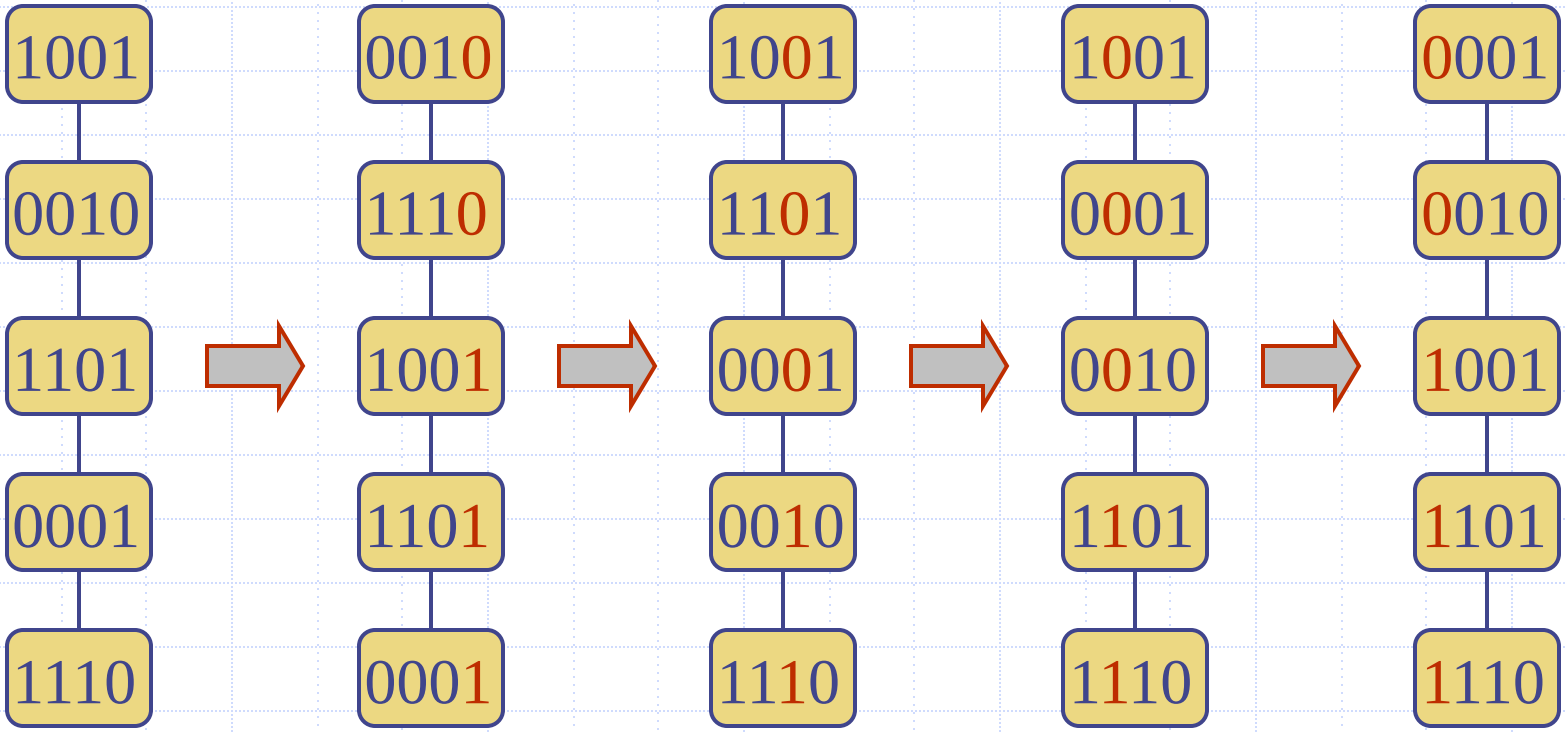
Straight Radix Sort

- Examines bit from right to left
  - for k:=0 to b-1
    - sort the array in a stable way
    - looking only at bit k

Example

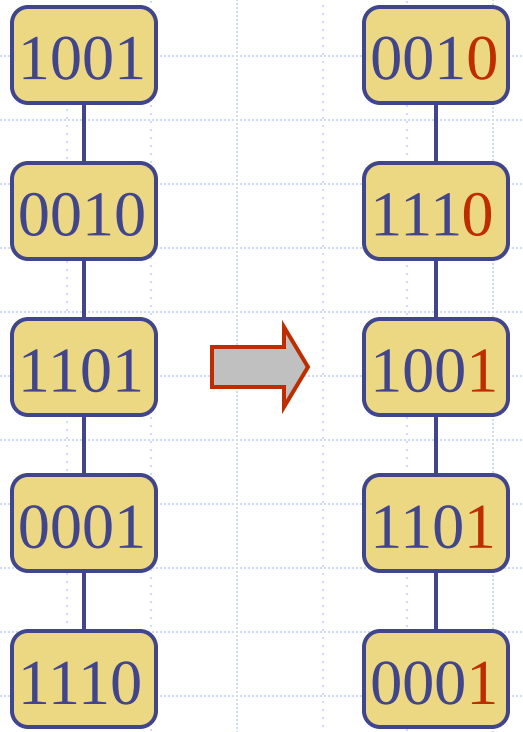


- Sorting a sequence of 4-bit integers



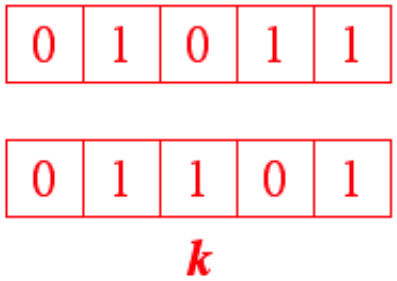
Sort in a Stable Way

- In a stable sort, the initial relative order of equal keys is unchanged
- For example, observe the first step of the sort.
- Note that the relative order of those keys ending with 0 is unchanged, and the same is true for elements ending in 1.



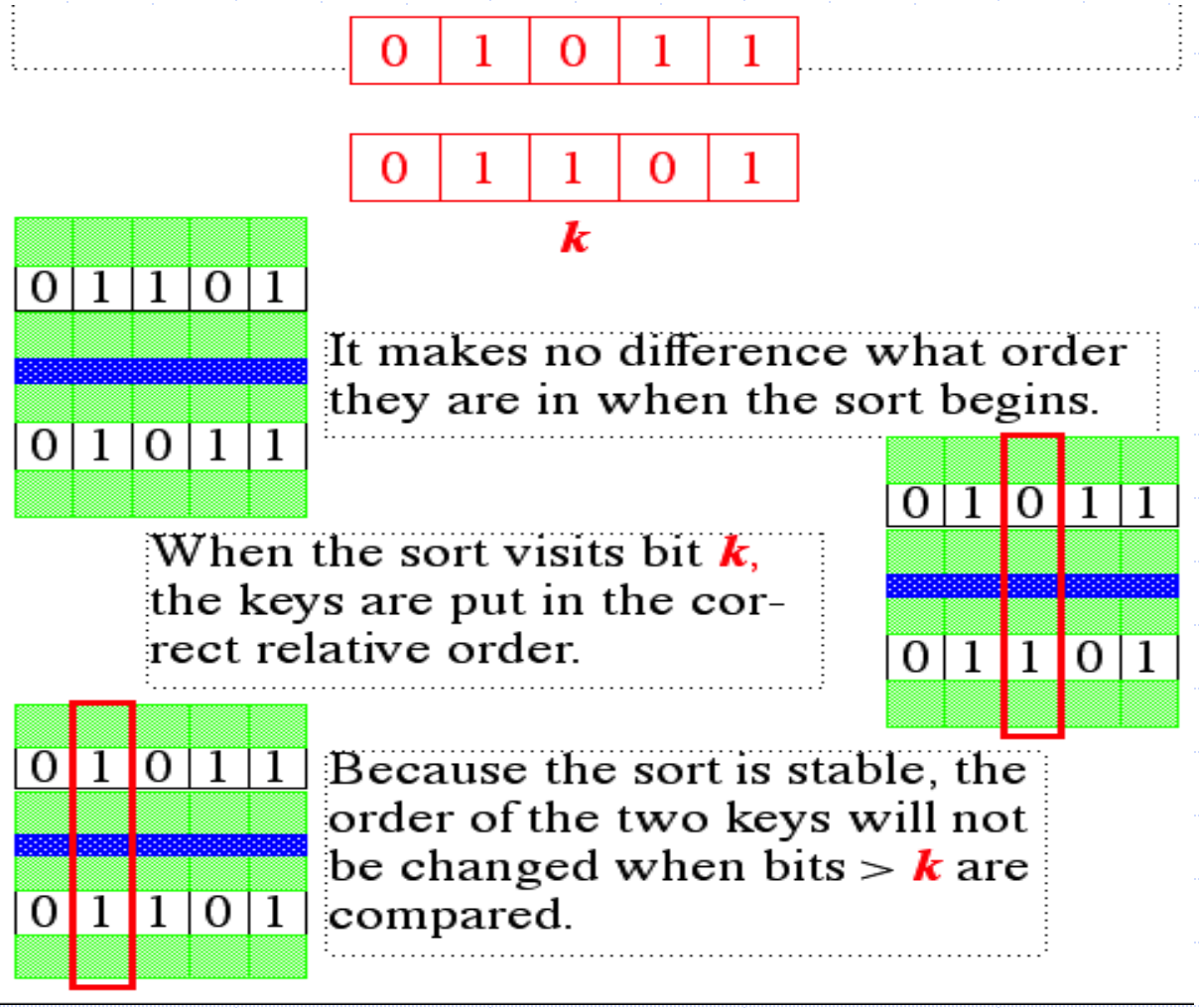
Correctness

- We show that any two keys are in the correct relative order at the end of the algorithm
- Given two keys, let k be the leftmost bit-position where they differ

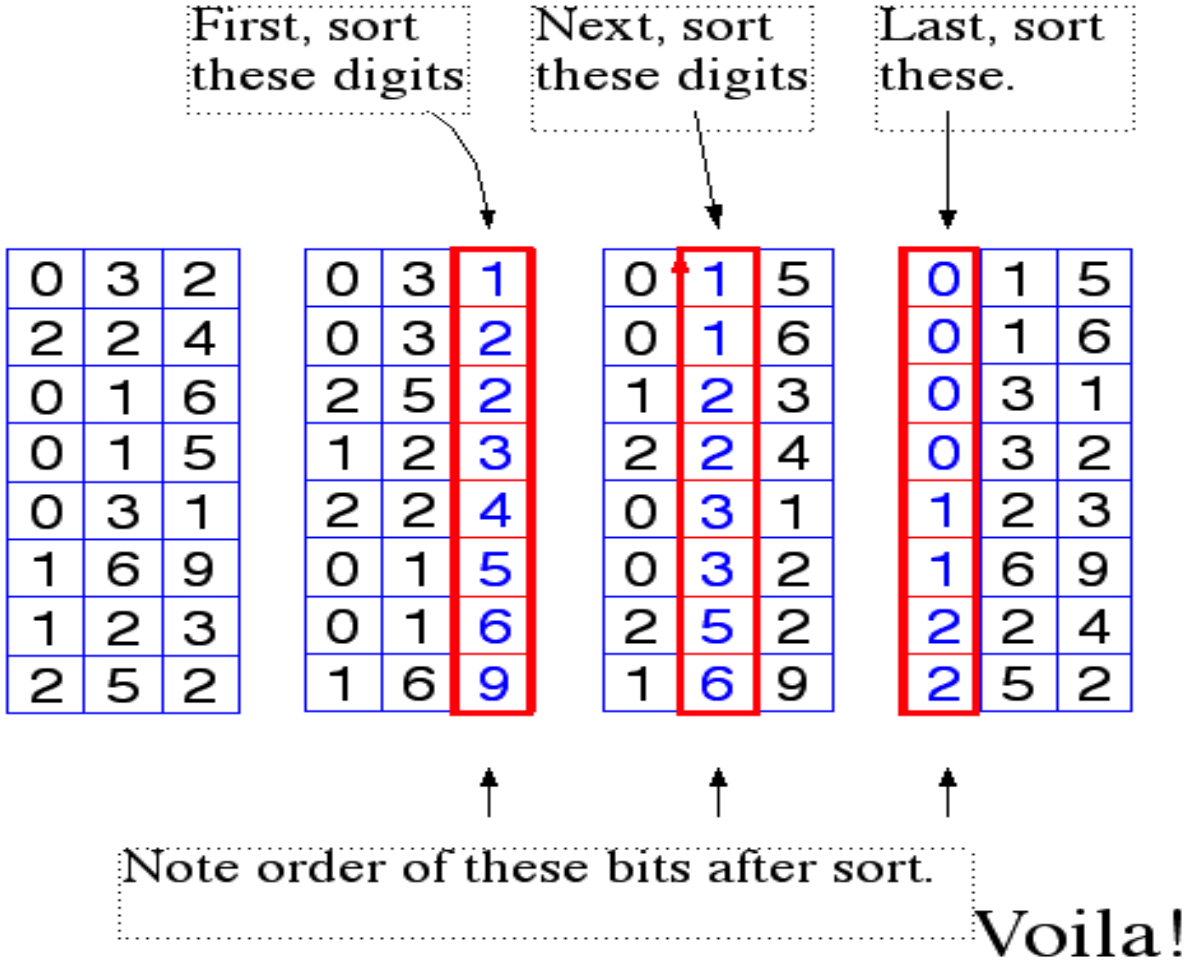


- At step k the two keys are put in the correct relative order
- Because of *stability*, the successive steps do not change the relative order of the two keys

Example


0 1 0 1 1
0 1 1 0 1
k
0 1 1 0 1
0 1 0 1 1
It makes no difference what order they are in when the sort begins.
0 1 0 1 1
0 1 1 0 1
When the sort visits bit k, the keys are put in the correct relative order.
0 1 0 1 1
0 1 1 0 1
Because the sort is stable, the order of the two keys will not be changed when bits > k are compared.

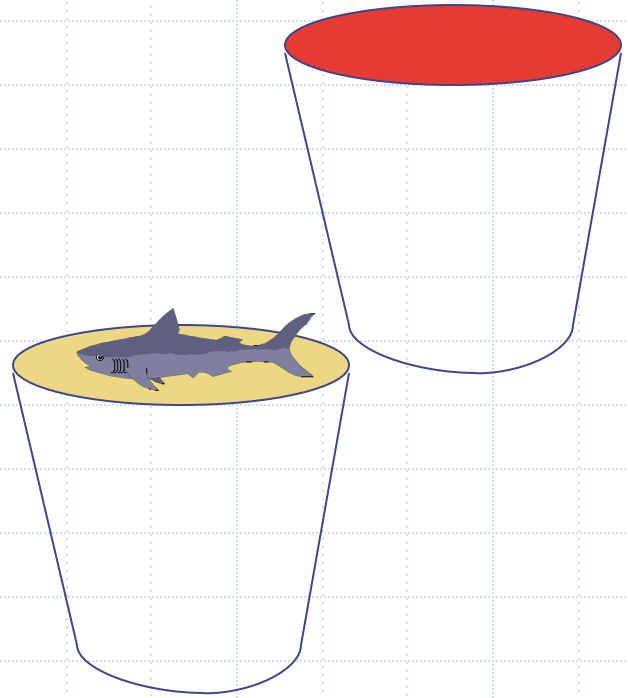
Example – Decimal Numbers



Straight Radix Sort Time Complexity

- for k = 0 to b - 1
  - sort the array in a stable way, looking only at bit k
- Suppose we can perform the stable sort above in O(n) time. The total time complexity would be O(bn)
- We can perform a stable sort based on the keys' k^{th} digit in O(n) time.
  - how?

Bucket Sort



BASICS:

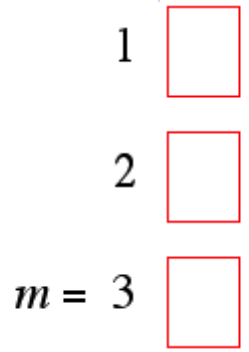
- n numbers
- Each number $\in \{1, 2, 3, \ldots m\}$
- Stable
- Time: O (n + m)

For example, m = 3 and our array is:

2	1	3	1	2

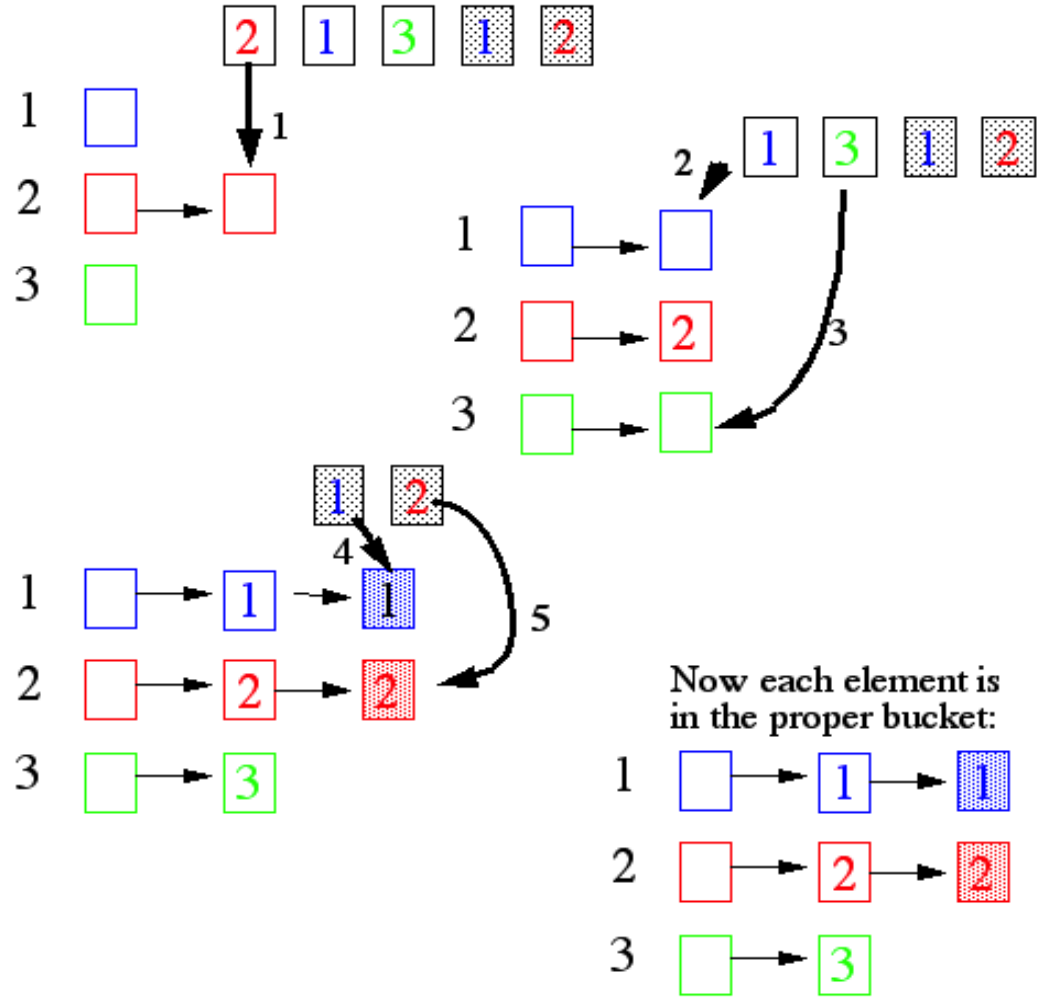
(note that there are two "2"s and two "1"s)

First, we create M "buckets"



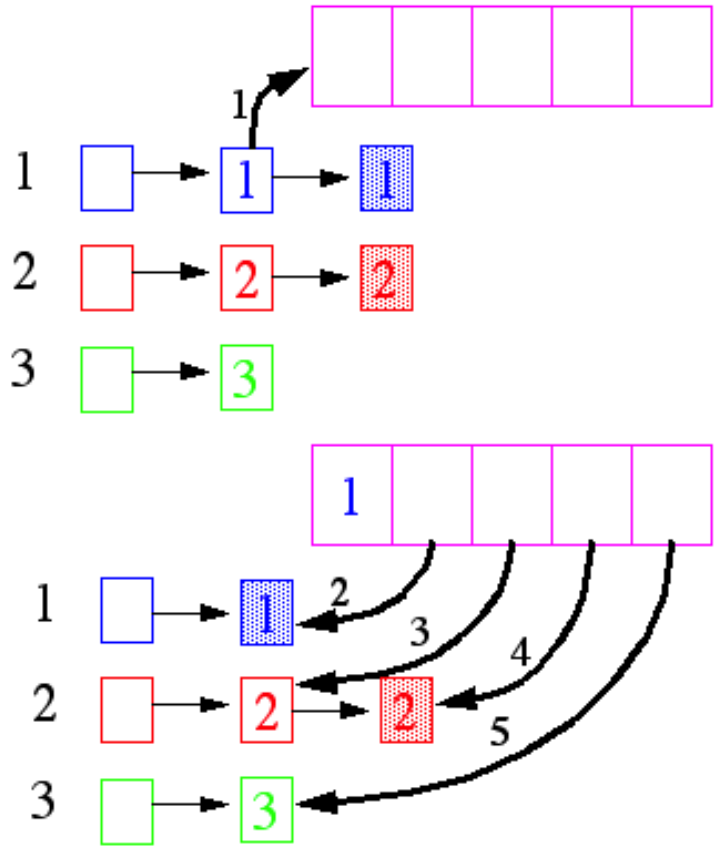
Bucket Sort

Each element of the array is put in one of the m "buckets"



Bucket Sort

Now, pull the elements from the buckets into the array



At last, the sorted array (sorted in a stable way):

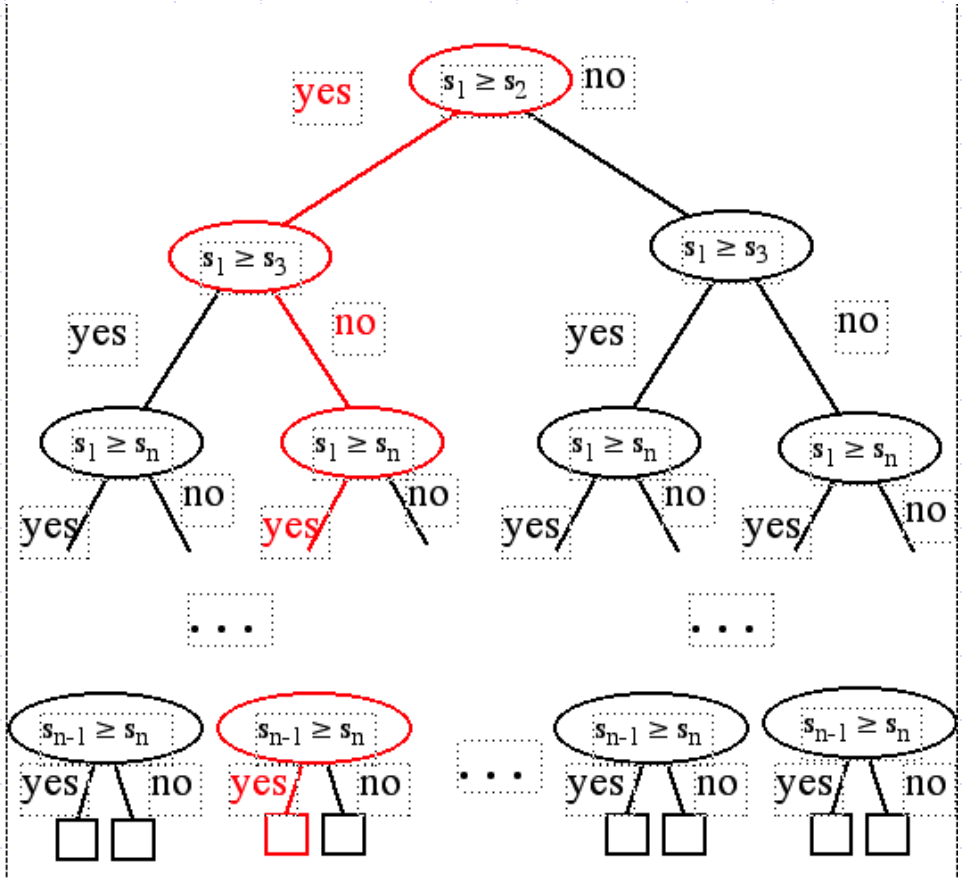
In-Place Sorting

- A sorting algorithm is said to be ***in-place*** if
  - it uses no auxiliary data structures (however, O(1) auxiliary variables are allowed)
  - it updates the input sequence only by means of operations replaceElement and swapElements
- Which sorting algorithms seen so far can be made to work in place?

selection-sort	
insertion-sort	
heap-sort	
merge-sort	
quick-sort	
radix-sort	
bucket-sort	

Lower Bound for Comparison Based Sorting

- internal node: comparison
- external node: permutation
- algorithm execution: root-to-leaf path



How Fast Can We Sort?

- Proposition: The running time of any comparison-based algorithm for sorting an n-element sequence S is $\Omega(n \log n)$.
- **Justification:** The running time of a comparison-based sorting algorithm must be equal to or greater than the depth of the decision tree T associated with this algorithm.
- Each internal node of T is associated with a comparison that establishes the ordering of two elements of S.
- Each external node of T represents a distinct permutation of the elements of S.
- Hence T must have at least n! external nodes which again implies T has a height of at least log(n!)
- Since n! has at least n/2 terms that are greater than or equal to n/2, we have:

$$\log(n!) \geq (n/2) \log(n/2)$$

- Total Time Complexity: $\Omega(n \log n)$.