Sorting

Chapter 11

Sorting

- We have seen the advantage of sorted data representations for a number of applications
  - Sparse vectors
  - Maps
  - Dictionaries
- Here we consider the problem of how to efficiently transform an unsorted representation into a sorted representation.
- We will focus on sorted array representations.

Sorting Algorithms

- Comparison Sorting
  - Selection Sort
  - Bubble Sort
  - Insertion Sort
  - Merge Sort
  - Heap Sort
  - Quick Sort
- Linear Sorting
  - Counting Sort
  - Radix Sort
  - Bucket Sort

Comparison Sorts

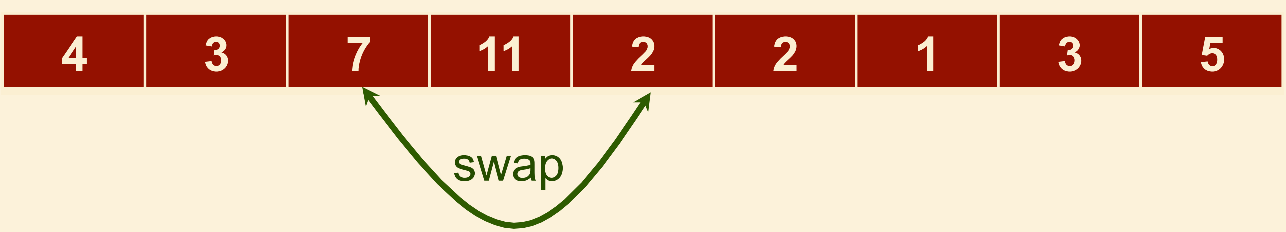
- Comparison Sort algorithms sort the input by successive comparison of pairs of input elements.
- Comparison Sort algorithms are very general: they make no assumptions about the values of the input elements.

4	3	7	11	2	2	1	3	5

e.g., $3 \le 11$?

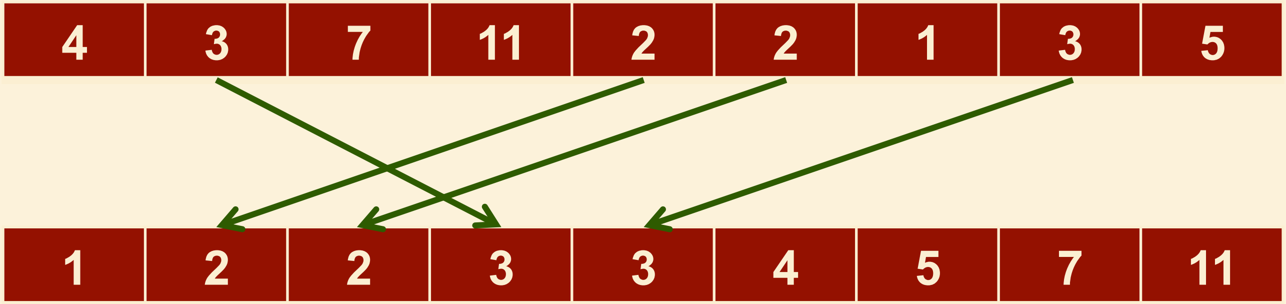
Sorting Algorithms and Memory

- Some algorithms sort by swapping elements within the input array
- Such algorithms are said to **sort in place**, and require only O(1) additional memory.
- Other algorithms require allocation of an output array into which values are copied.
- These algorithms do not sort in place, and require O(n) additional memory.



Stable Sort

- A sorting algorithm is said to be **stable** if the ordering of identical keys in the input is preserved in the output.
- The stable sort property is important, for example, when entries with identical keys are already ordered by another criterion.
- (Remember that stored with each key is a record containing some useful information.)



Selection Sort

- Selection Sort operates by first finding the smallest element in the input list, and moving it to the output list.
- It then finds the next smallest value and does the same.
- It continues in this way until all the input elements have been selected and placed in the output list in the correct order.
- Note that every selection requires a search through the input list.
- Thus the algorithm has a nested loop structure
- Selection Sort Example

Selection Sort

for i = 0 to n-1

LI: A[0…i-1] contains the i smallest keys in sorted order.
A[i…n-1] contains the remaining keys

j_{min} = i

for j = i+1 to n-1

if A[j] < A[j_{min}]

j_{min} = j

swap A[i] with A[j_{min}]

Running time?
$O(n-i-1)$

$$T(n) = \sum_{i=0}^{n-1} \left(n-i-1\right) = \sum_{i=0}^{n-1} i \quad = O(n^2)$$

Bubble Sort

- Bubble Sort operates by successively comparing adjacent elements, swapping them if they are out of order.
- At the end of the first pass, the largest element is in the correct position.
- A total of n passes are required to sort the entire array.
- Thus bubble sort also has a nested loop structure
- Bubble Sort Example

Expert Opinion on Bubble Sort

Bubble Sort

for i = n-1 downto 1

LI: A[i+1…n-1] contains the n-i-1 largest keys in sorted order.
A[0…i] contains the remaining keys

```
for j = 0 to i-1
    if A[ j ] > A[ j + 1 ]
        swap A[ j ] and A[ j + 1 ]
```

Running time?
$O(i)$

$$T(n) = \sum_{i=1}^{n-1} i \quad = O(n^2)$$

Comparison

- Thus both Selection Sort and Bubble Sort have $O(n^2)$ running time.
- However, both can also easily be designed to
  - Sort in place
  - Stable sort

Insertion Sort

- Like Selection Sort, Insertion Sort maintains two sublists:
  - A left sublist containing sorted keys
  - A right sublist containing the remaining unsorted keys
- Unlike Selection Sort, the keys in the left sublist are not the smallest keys in the input list, but the first keys in the input list.
- On each iteration, the next key in the right sublist is considered, and inserted at the correct location in the left sublist.
- This continues until the right sublist is empty.
- Note that for each insertion, some elements in the left sublist will in general need to be shifted right.
- Thus the algorithm has a nested loop structure
- Insertion Sort Example

Insertion Sort

for i = 1 to n-1

LI: A[0…i-1] contains the first i keys of the input in sorted order.
A[i…n-1] contains the remaining keys

key = A[i]

j = i

while j > 0 & A[j-1] > key

A[j] ← A[j-1]

j = j-1

A[j] = key

Running time?
$O(i)$

$$T(n) = \sum_{i=1}^{n-1} i \; = O(n^2)$$

Comparison

- Selection Sort
- Bubble Sort
- Insertion Sort
  - Sort in place
  - Stable sort
  - But $O(n^2)$ running time.
- Can we do better?

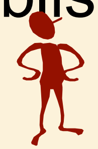
Divide-and-Conquer

- 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 is a subproblem of size 0 or 1

Recursive Sorts

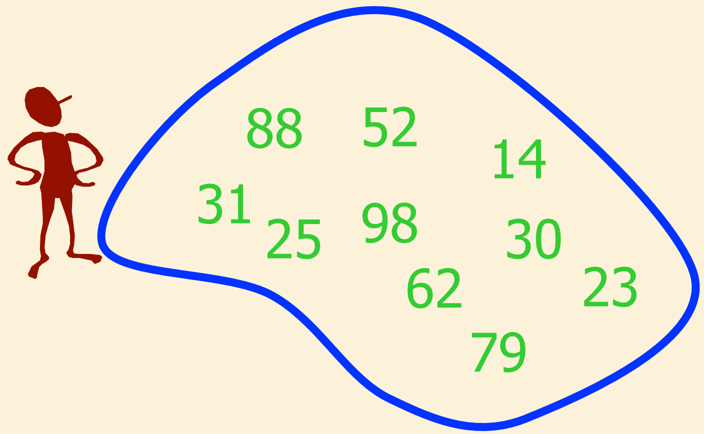
- Given list of objects to be sorted
- Split the list into two sublists.

- Recursively have two friends sort the two sublists.



- Combine the two sorted sublists into one entirely sorted list.

Merge Sort

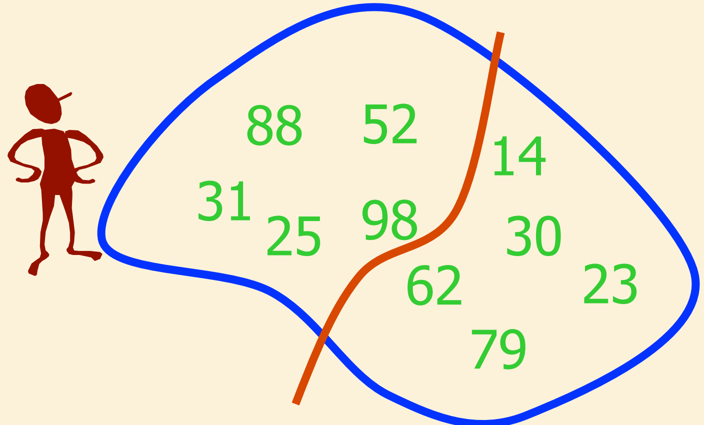


Divide and Conquer

Merge Sort

- Merge-sort is a sorting algorithm based on the divide-and-conquer paradigm
- It was invented by John von Neumann, one of the pioneers of computing, in 1945

Merge Sort



Split Set into Two
(no real work)

Get one friend to
sort the first half.

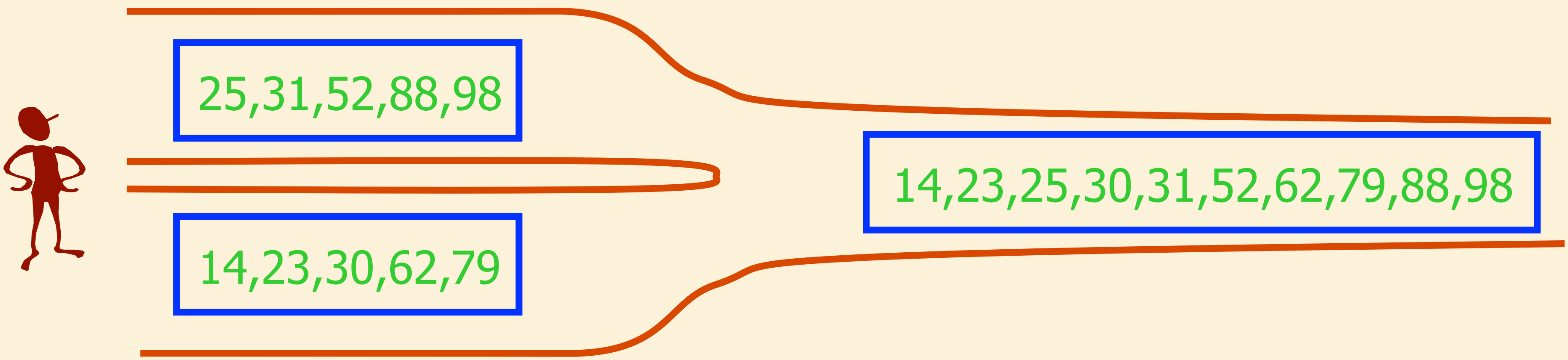
25,31,52,88,98

Get one friend to
sort the second half.

14,23,30,62,79

Merge Sort

Merge two sorted lists into one



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
  if $S.size() > 1$
    $(S_1, S_2) \leftarrow split(S, n/2)$
    $mergeSort(S_1)$
    $mergeSort(S_2)$
    $merge(S_1, S_2, S)$

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 takes $O(n)$ time
- Straightforward to make the sort stable.
- Normally, merging is not in-place: new memory must be allocated to hold S.
- It **is** possible to do in-place merging using linked lists.
  - Code is more complicated
  - Only changes memory usage by a constant factor

Merging Two Sorted Sequences (As Arrays)

Algorithm merge(S_1, S_2, S):

Input: Sorted sequences S_1 and S_2 and an empty sequence S, implemented as arrays

Output: Sorted sequence S containing the elements from S_1 and S_2

$i \leftarrow j \leftarrow 0$

while $i < S_1$.size() **and** $j < S_2$.size() **do**

  if S_1.get(i) $\leq S_2$.get(j) **then**

    S.addLast(S_1.get(i))

    $i \leftarrow i + 1$

  else

    S.addLast(S_2.get(j))

    $j \leftarrow j + 1$

while $i < S_1$.size() **do**

  S.addLast(S_1.get(i))

  $i \leftarrow i + 1$

while $j < S_2$.size() **do**

  S.addLast(S_2.get(j))

  $j \leftarrow j + 1$

Merging Two Sorted Sequences (As Linked Lists)

Algorithm merge(S_1, S_2, S):

Input: Sorted sequences S_1 and S_2 and an empty sequence S, implemented as linked lists

Output: Sorted sequence S containing the elements from S_1 and S_2

while $S_1 \neq \varnothing$ **and** $S_2 \neq \varnothing$ **do**

  if S_1.first().element() $\leq$ S_2.first().element() **then**

    S.addLast(S_1.remove(S_1.first()))

  else

    S.addLast(S_2.remove(S_2.first()))

while $S_1 \neq \varnothing$ **do**

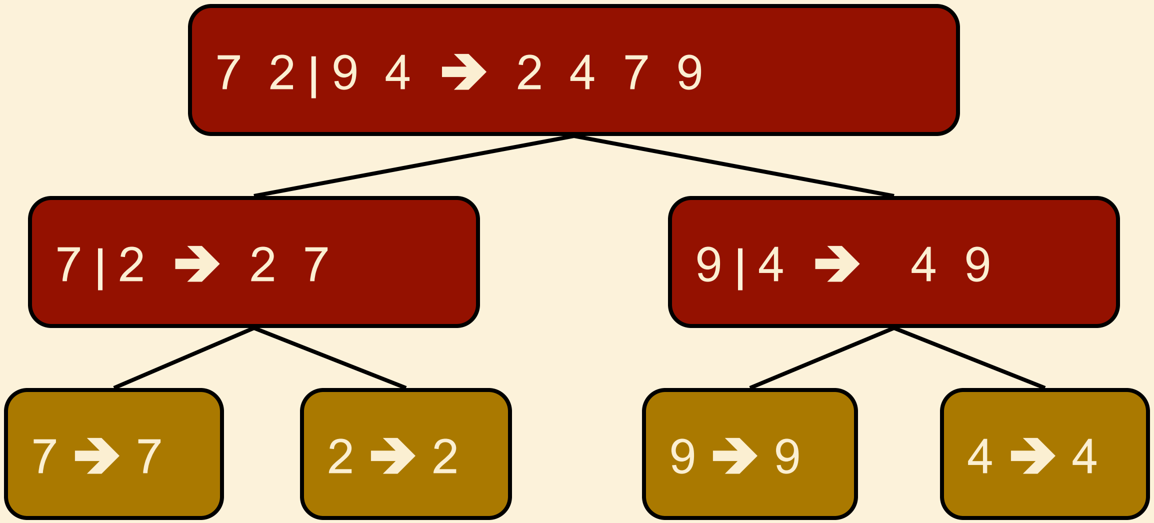
  S.addLast(S_1.remove(S_1.first()))

while $S_2 \neq \varnothing$ **do**

  S.addLast(S_2.remove(S_2.first()))

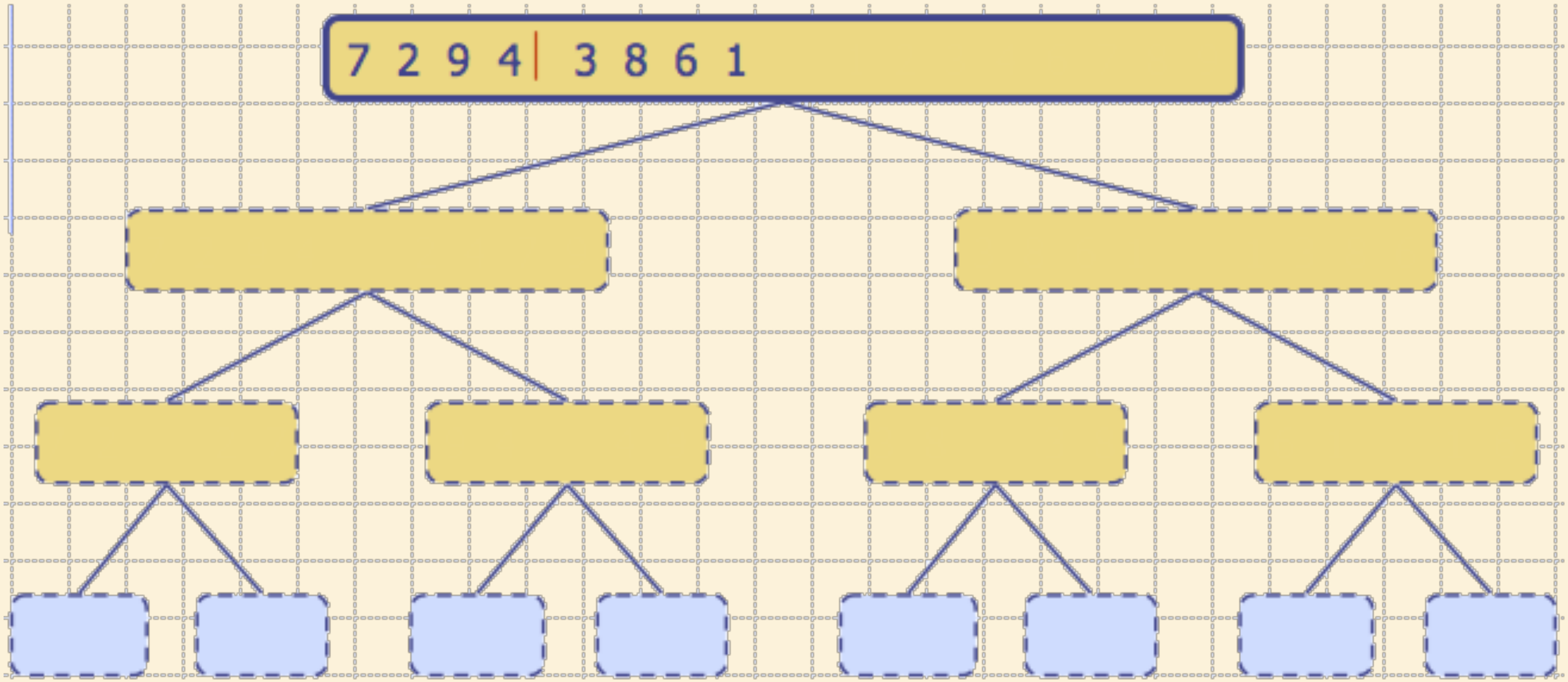
Merge-Sort Tree

- 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



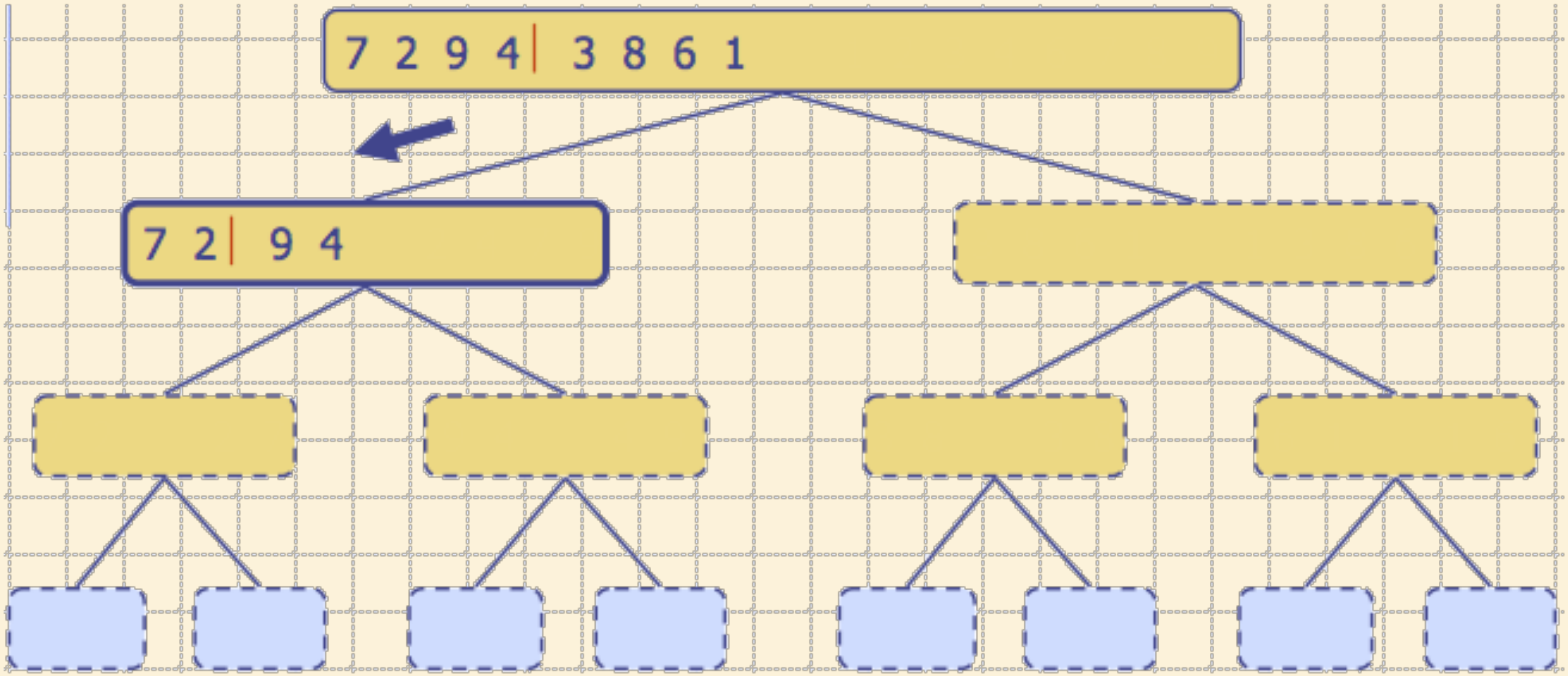
Execution Example

- Partition



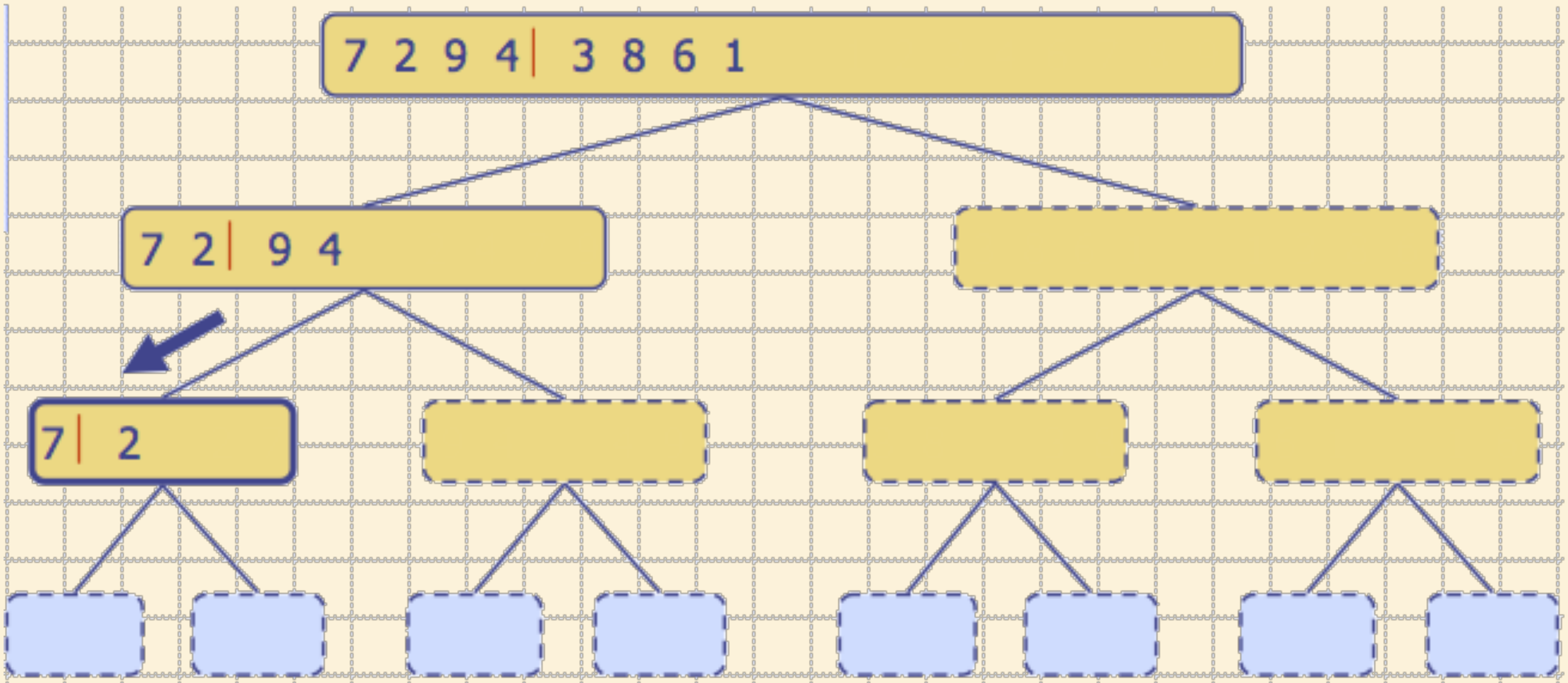
Execution Example (cont.)

- Recursive call, partition



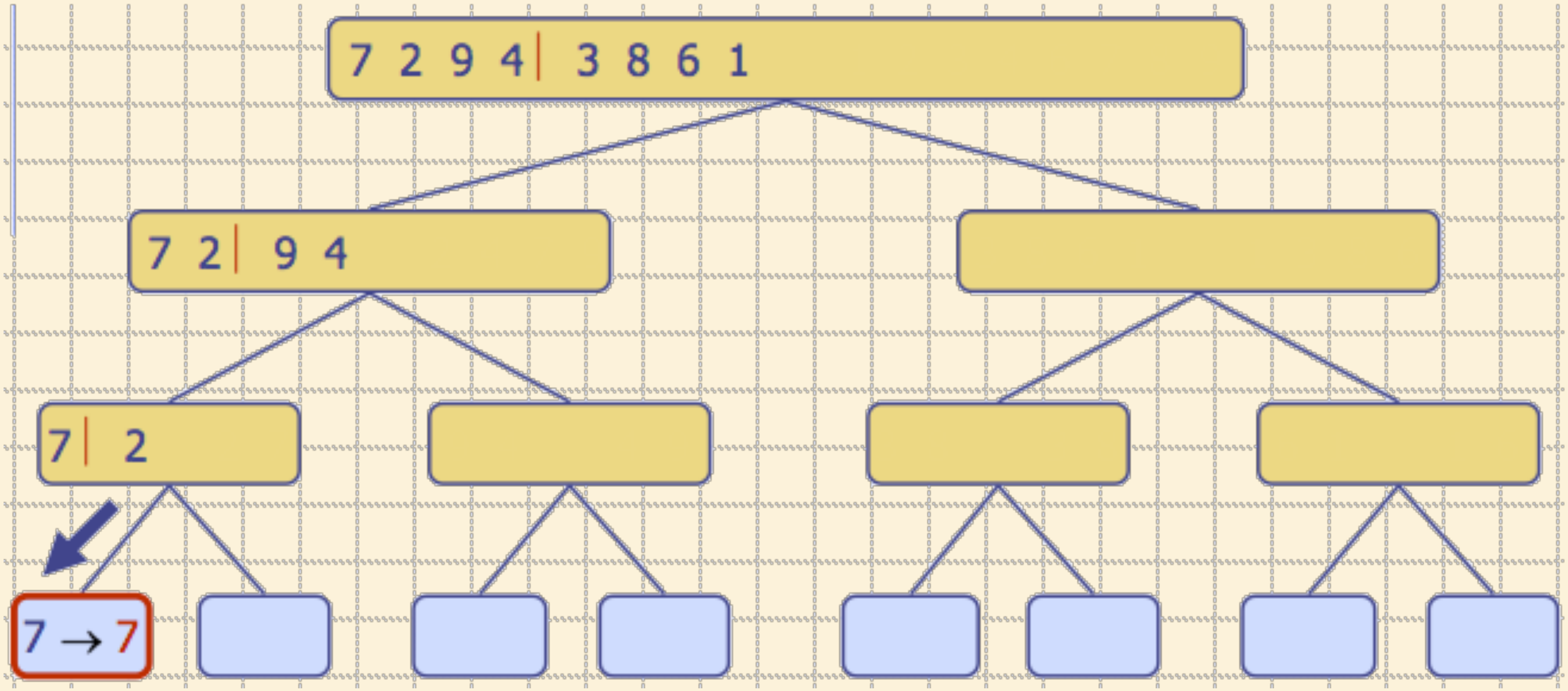
Execution Example (cont.)

- Recursive call, partition



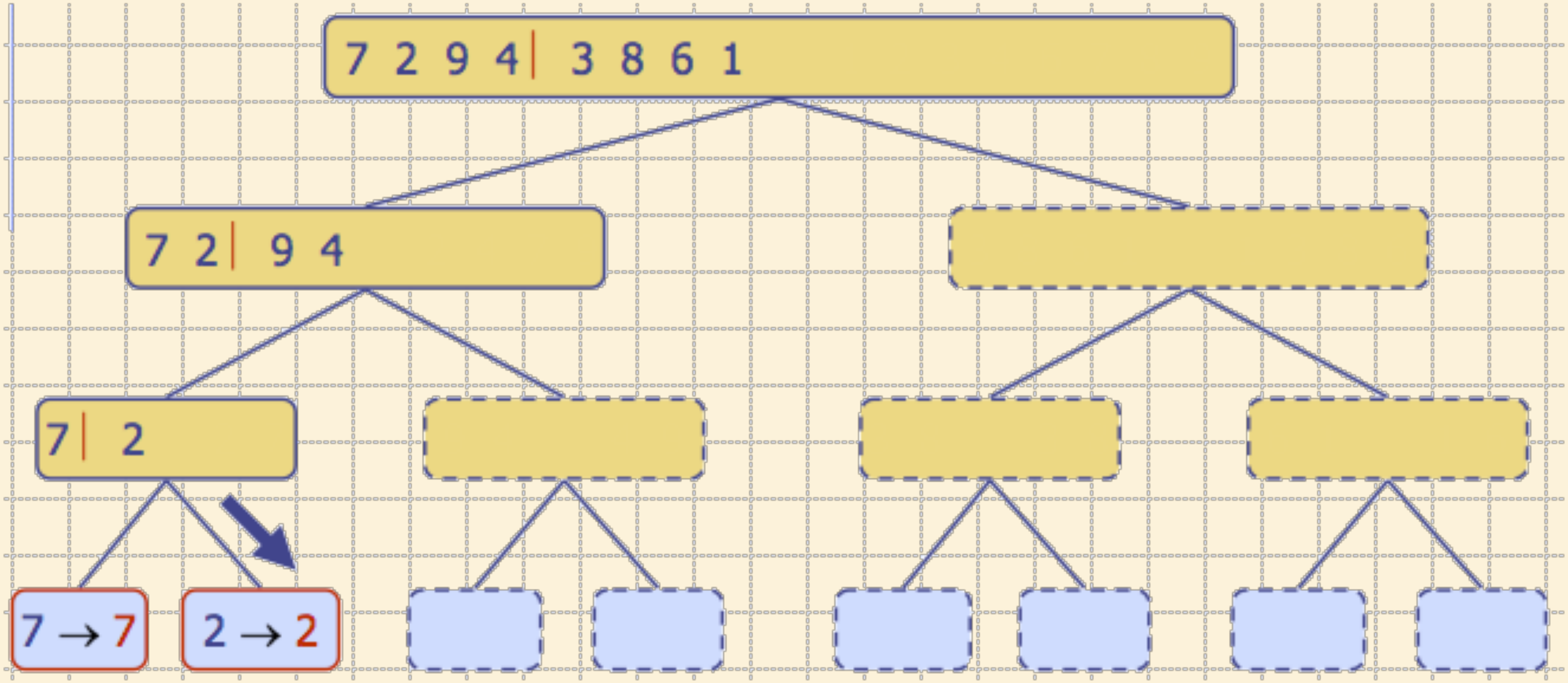
Execution Example (cont.)

- Recursive call, base case



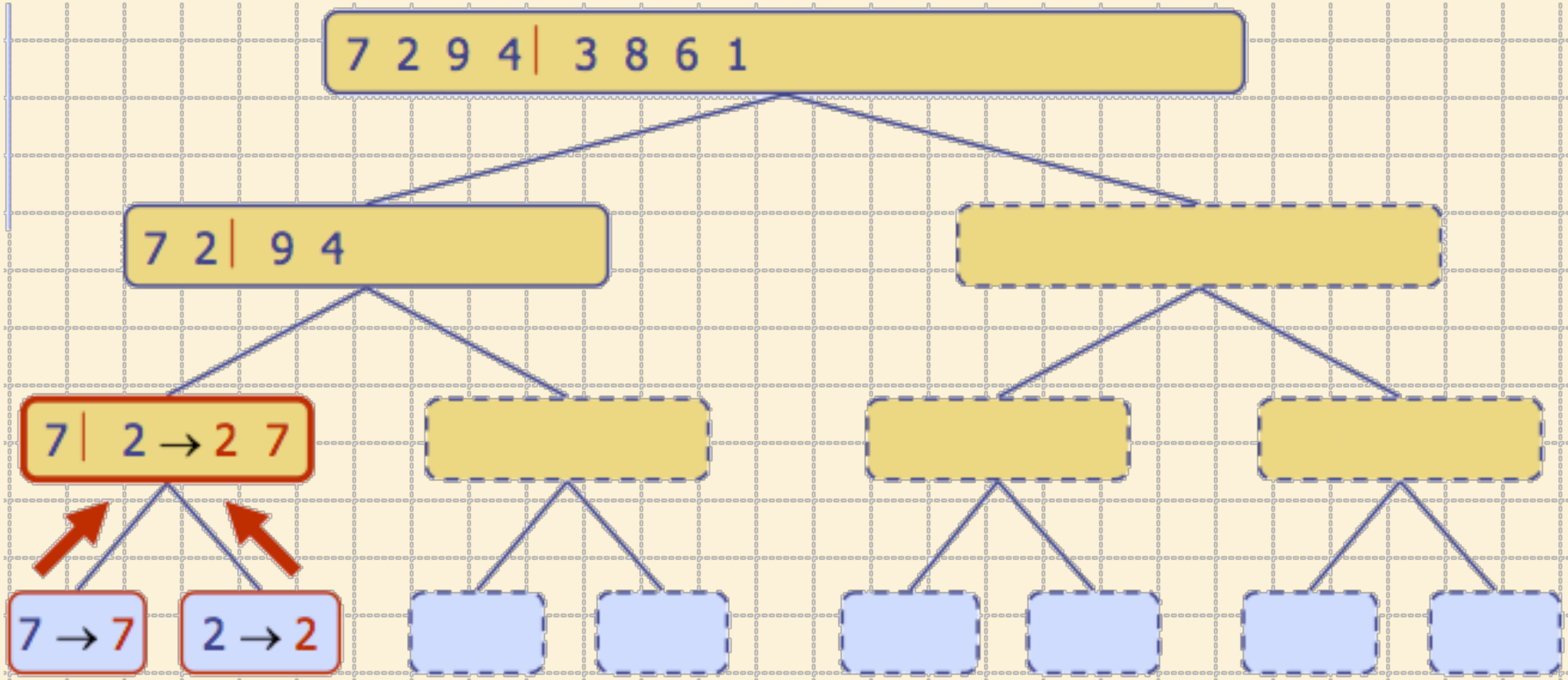
Execution Example (cont.)

- Recursive call, base case



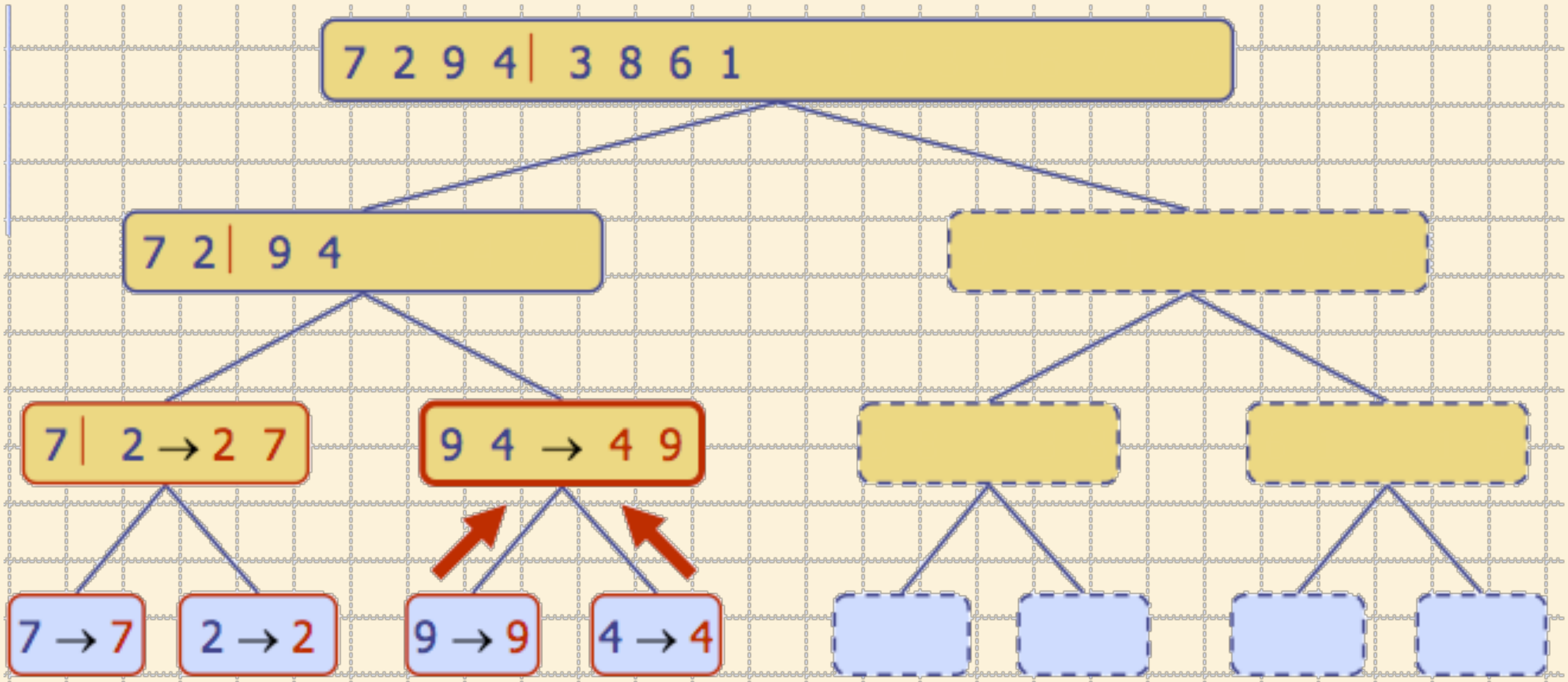
Execution Example (cont.)

- Merge



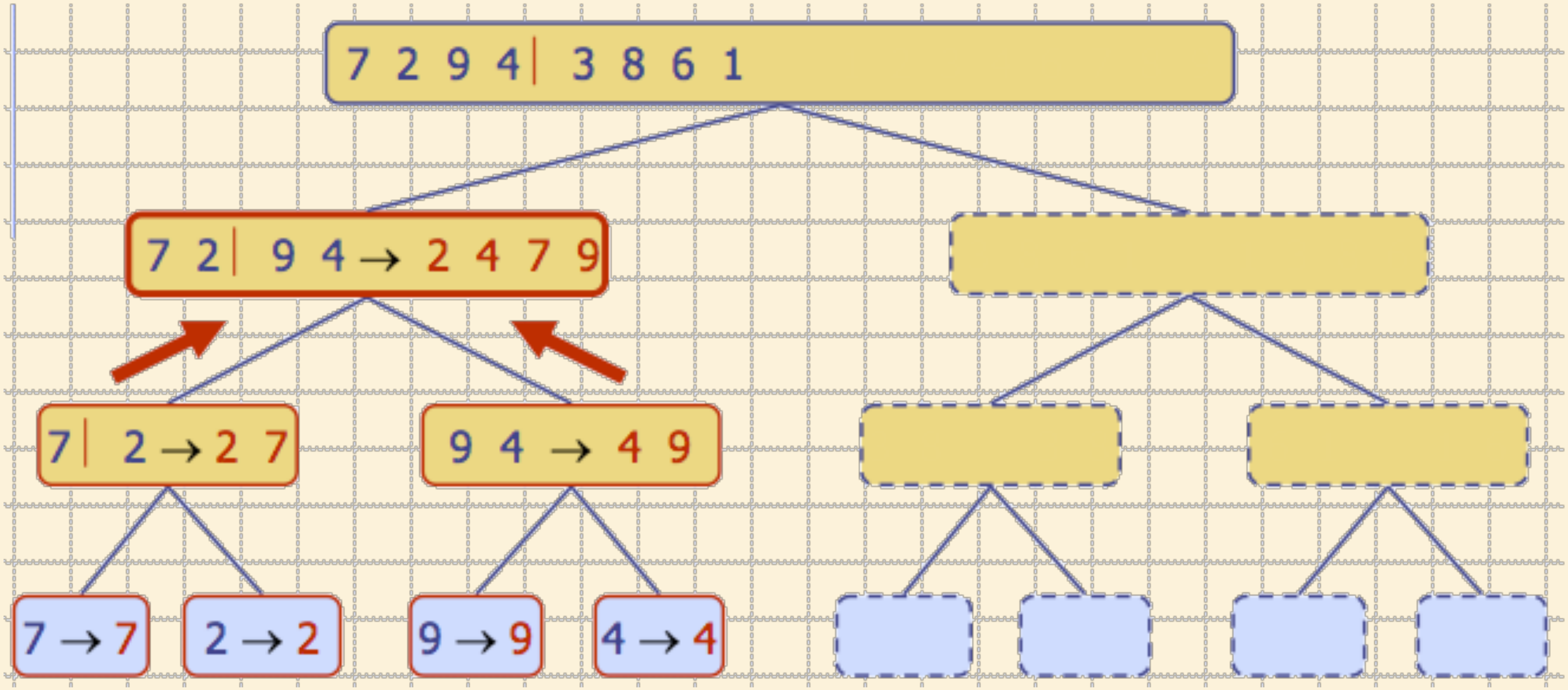
Execution Example (cont.)

- Recursive call, …, base case, merge



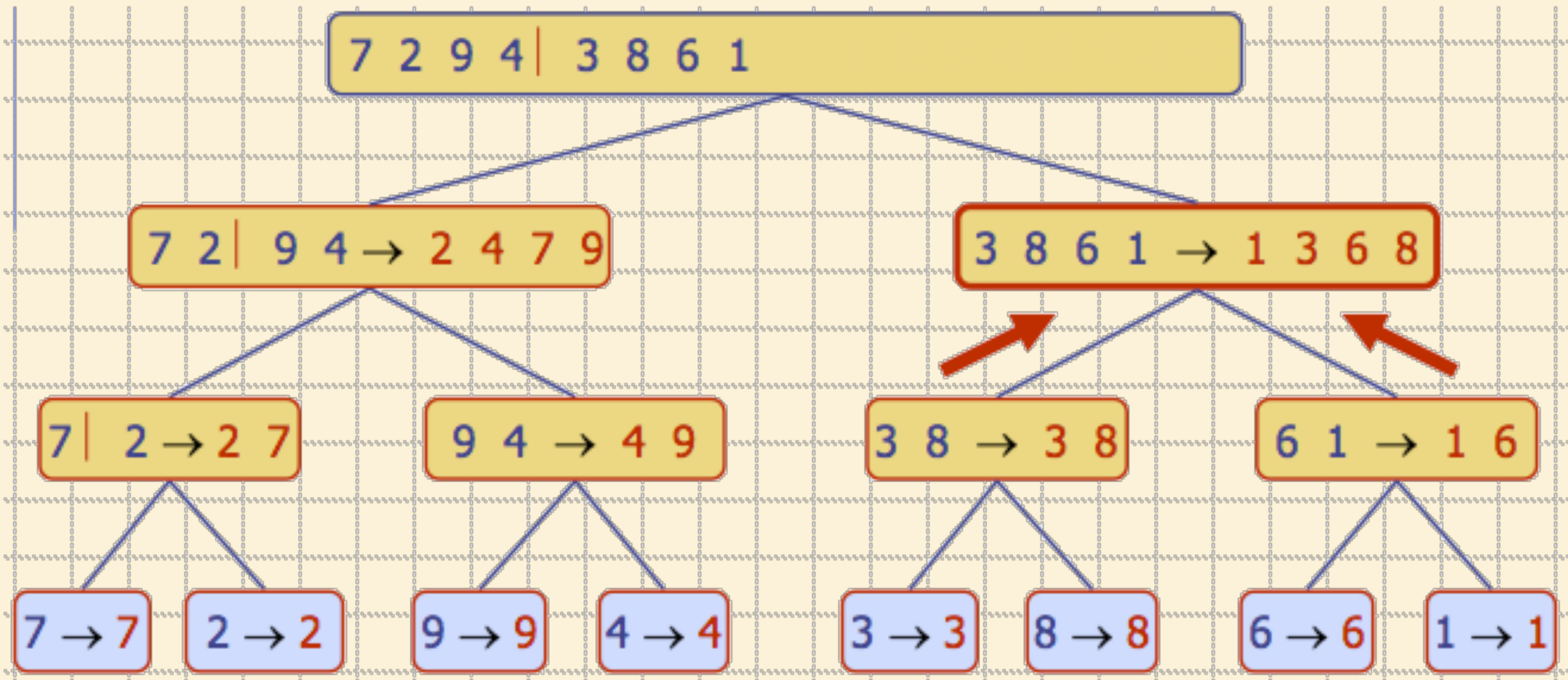
Execution Example (cont.)

- Merge



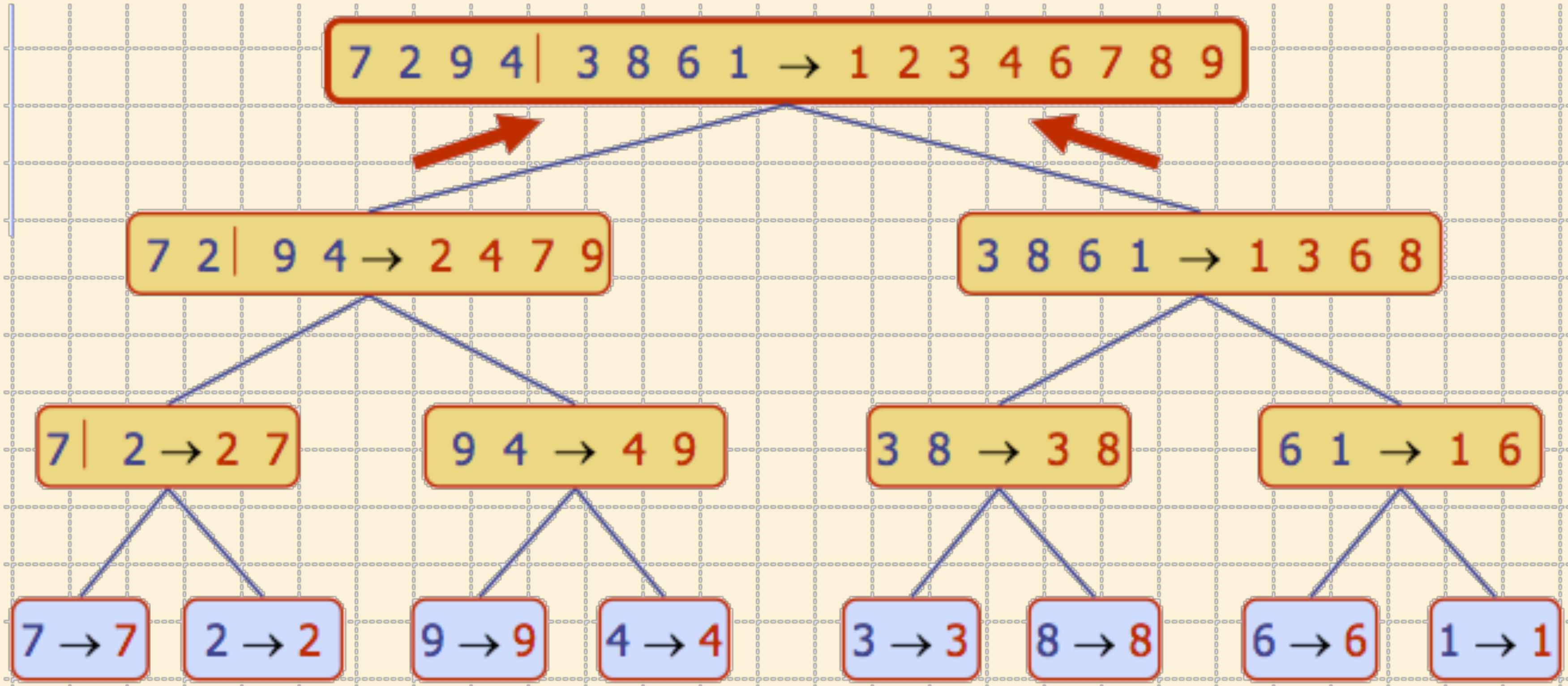
Execution Example (cont.)

- Recursive call, …, merge, merge



Execution Example (cont.)

- Merge



End of Lecture

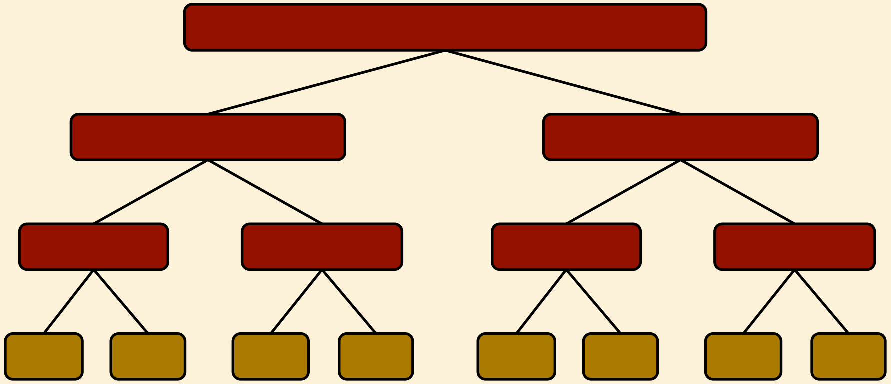
March 11, 2012

Analysis of Merge-Sort

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

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

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



Running Time of Comparison Sorts

- Thus MergeSort is much more efficient than SelectionSort, BubbleSort and InsertionSort. Why?
- You might think that to sort n keys, each key would have to at some point be compared to every other key:

$$\rightarrow O\left(n^2\right)$$

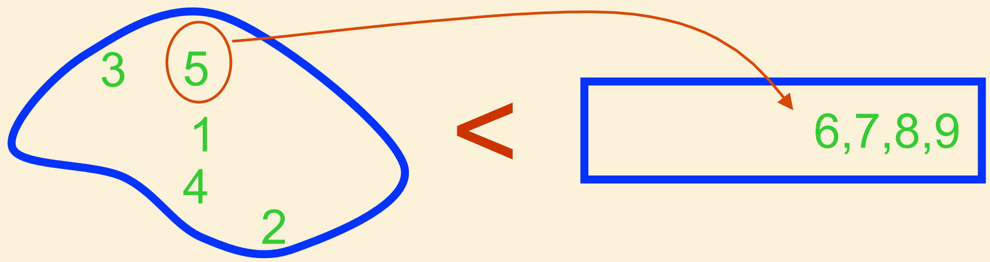
- However, this is not the case.
  - Transitivity: If A < B and B < C, then you know that A < C, even though you have never directly compared A and C.
  - MergeSort takes advantage of this transitivity property in the merge stage.

Heapsort

- Invented by Williams & Floyd in 1964
- O(*nlogn*) worst case – like merge sort
- Sorts in place – like selection sort
- Combines the best of both algorithms

Selection Sort

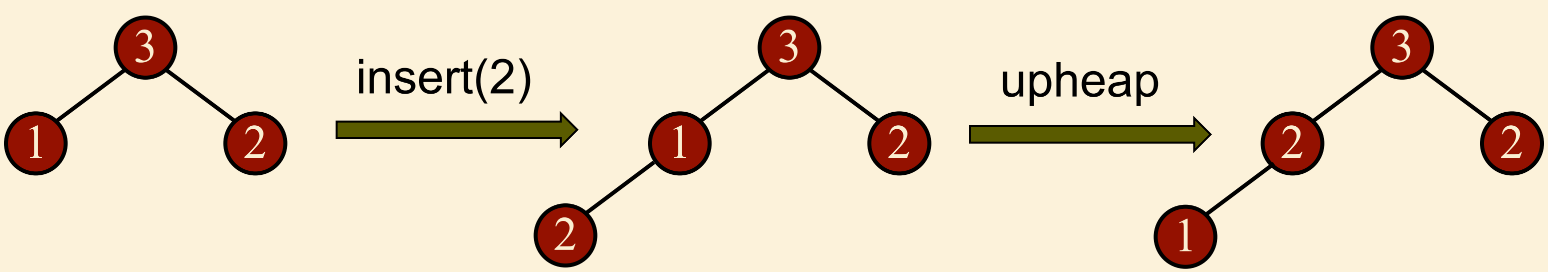
Largest i values are sorted on the right.
Remaining values are off to the left.



Max is easier to find if the unsorted subarray is a heap.

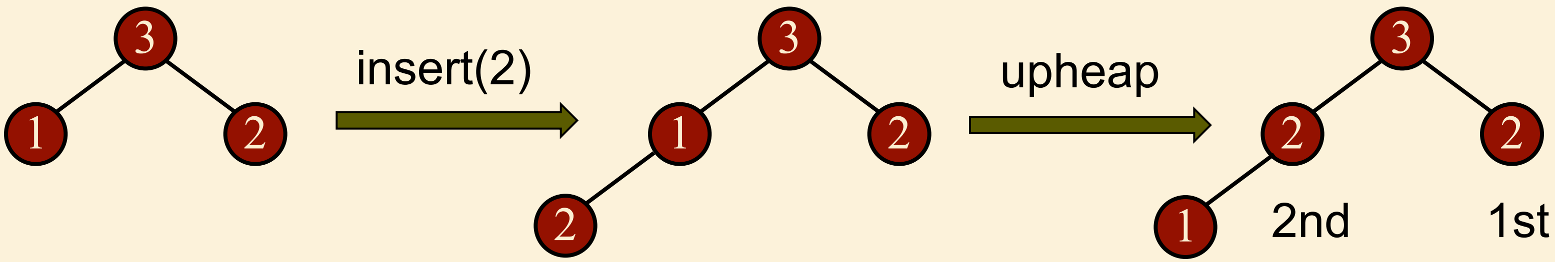
Heap-Sort Algorithm

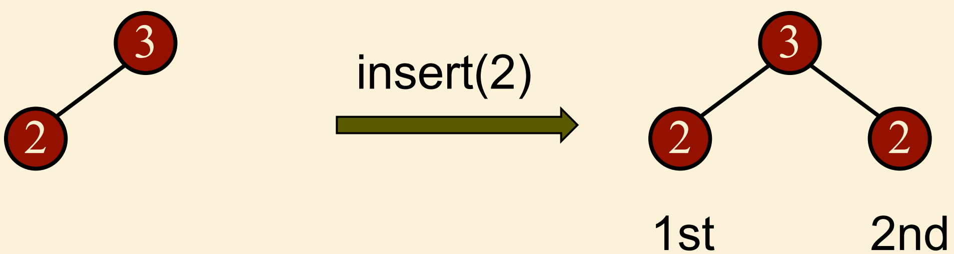
- Build an array-based (max) heap
- Iteratively call removeMax() to extract the keys in descending order
- Store the keys as they are extracted in the unused tail portion of the array
- Thus HeapSort is in-place!
- But is it stable?
  - No – heap operations may disorder ties



Heapsort is Not Stable

- Example (MaxHeap)





Heap-Sort Algorithm

Algorithm HeapSort(S)

Input: S, an unsorted array of comparable elements

Output: S, a sorted array of comparable elements

```
T = MakeMaxHeap (S)
for i = n-1 downto 0
    S[i] = T.removeMax()
```

Heap Sort Example

(Using Min Heap)

Heap-Sort Running Time

- The heap can be built bottom-up in O(n) time
- Extraction of the ith element takes O(log(n - i+1)) time (for downheaping)
- Thus total run time is

$$
\begin{aligned}
T(n) &= O(n) + \sum_{i=1}^{n} \log(n-i+1) \\
&= O(n) + \sum_{i=1}^{n} \log i \\
&\le O(n) + \sum_{i=1}^{n} \log n \\
&= O(n \log n)
\end{aligned}
$$

Heap-Sort Running Time

- It turns out that HeapSort is also Ω(nlogn). Why?

$$T(n) = O(n) + \sum_{i=1}^{n} \log i, \text{ where}$$

$$\sum_{i=1}^{n} \log i \geq (n/2)\log(n/2)$$

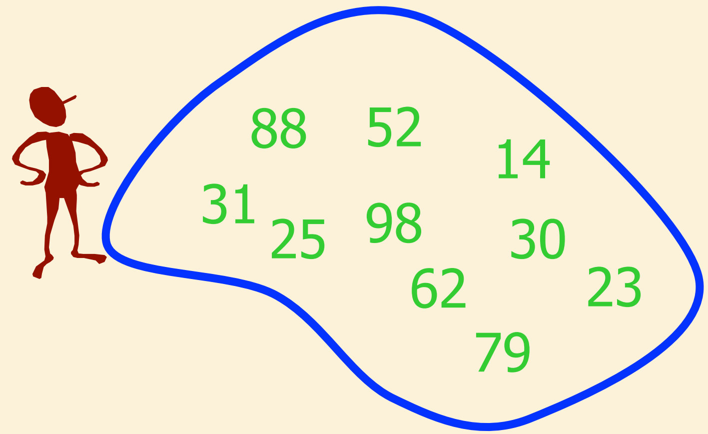
$$= (n/2)(\log n - 1)$$

$$= (n/4)(\log n + \log n - 2)$$

$$\geq (n/4)\log n \quad \forall n \geq 4.$$

- Thus HeapSort is θ(nlogn).

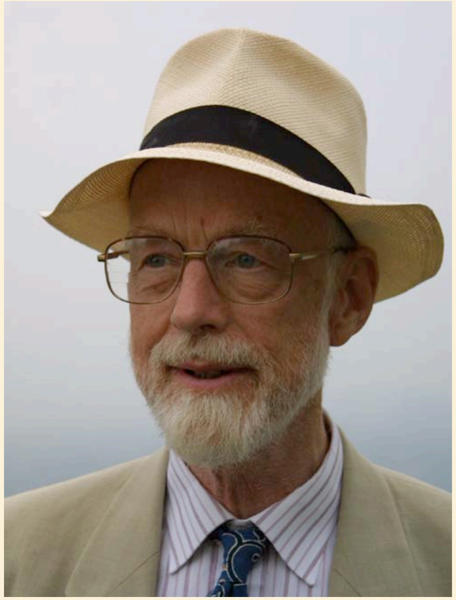
Quick-Sort



Divide and Conquer

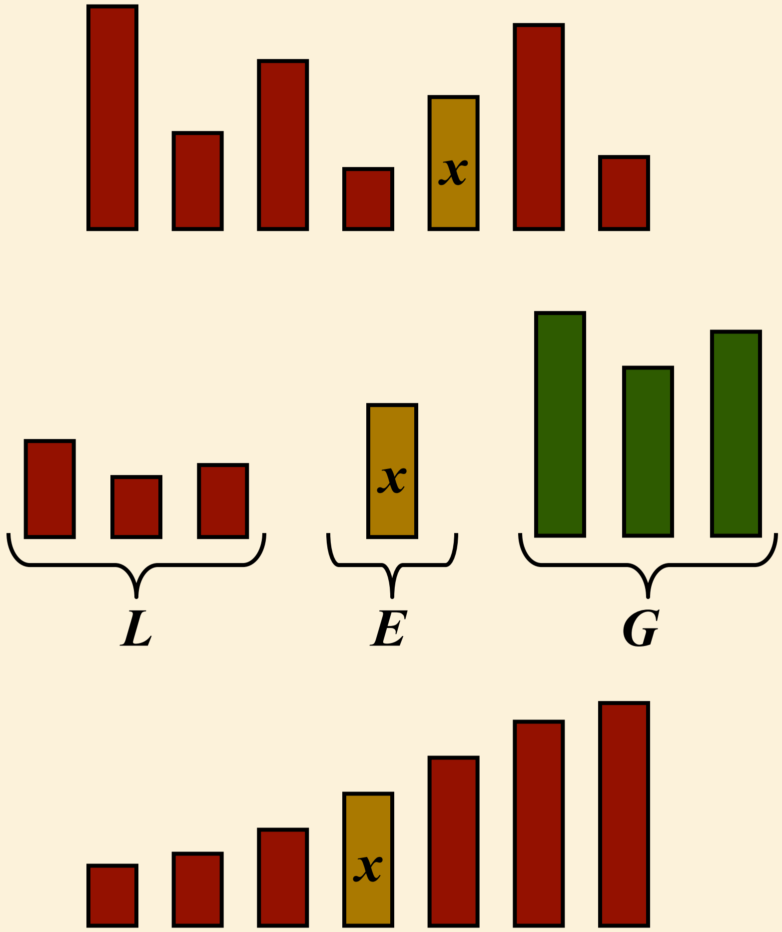
QuickSort

- Invented by C.A.R. Hoare in 1960
- "There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult."



Quick-Sort

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



The Quick-Sort Algorithm

```
Algorithm QuickSort(S)
    if S.size() > 1
        (L, E, G) = Partition(S)
        QuickSort(L) //Small elements are sorted
        QuickSort(G) //Large elements are sorted
        S = (L, E, G) //Thus input is sorted
```

Partition

- Remove, in turn, each element y from S and
- Insert y into list L, E or G, depending on the result of the comparison with the pivot x (e.g., last element in S)
- Each insertion and removal is at the beginning or at the end of a list, and hence takes $O(1)$ time
- Thus, partitioning takes $O(n)$ time

Algorithm ***Partition***(S)

    Input list S

    Output sublists L, E, G of the elements of S less than, equal to, or greater than the pivot, resp.

    $L, E, G \leftarrow$ empty lists

    $x \leftarrow S.getLast().element$

    while $\neg S.isEmpty()$

        $y \leftarrow S.removeFirst(S)$

        if $y < x$

            $L.addLast(y)$

        else if $y = x$

            $E.addLast(y)$

        else $\{ y > x \}$

            $G.addLast(y)$

    return L, E, G

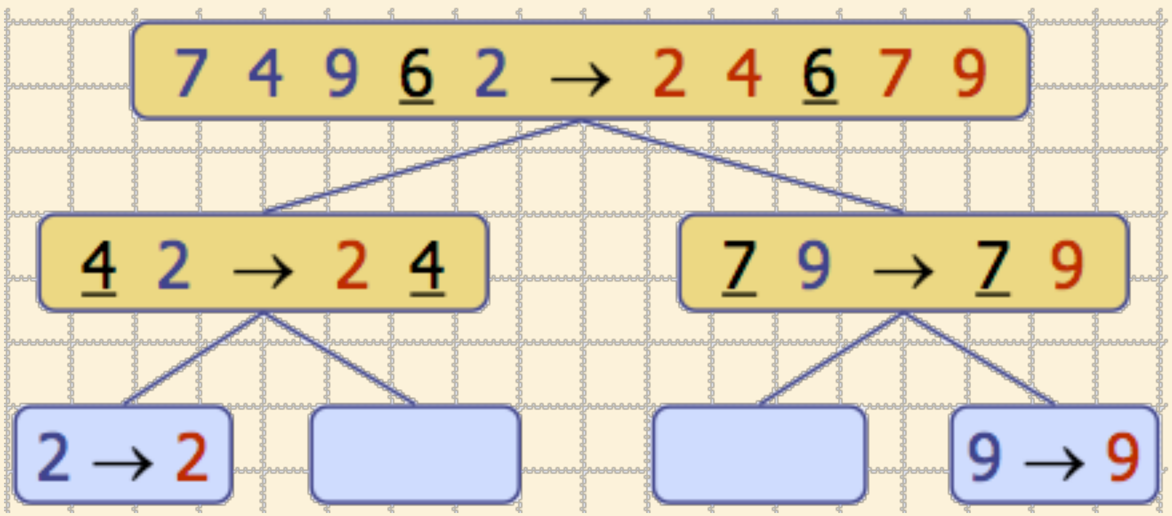
Partition

- Since elements are removed at the beginning and added at the end, this partition algorithm is **stable**.

Algorithm *Partition*(S)
  Input sequence S
  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
  $x \leftarrow S.getLast().element$
  while $\neg S.isEmpty()$
    $y \leftarrow S.removeFirst(S)$
    if $y < x$
      $L.addLast(y)$
    else if $y = x$
      $E.addLast(y)$
    else $\{ y > x \}$
      $G.addLast(y)$
  return L, E, G

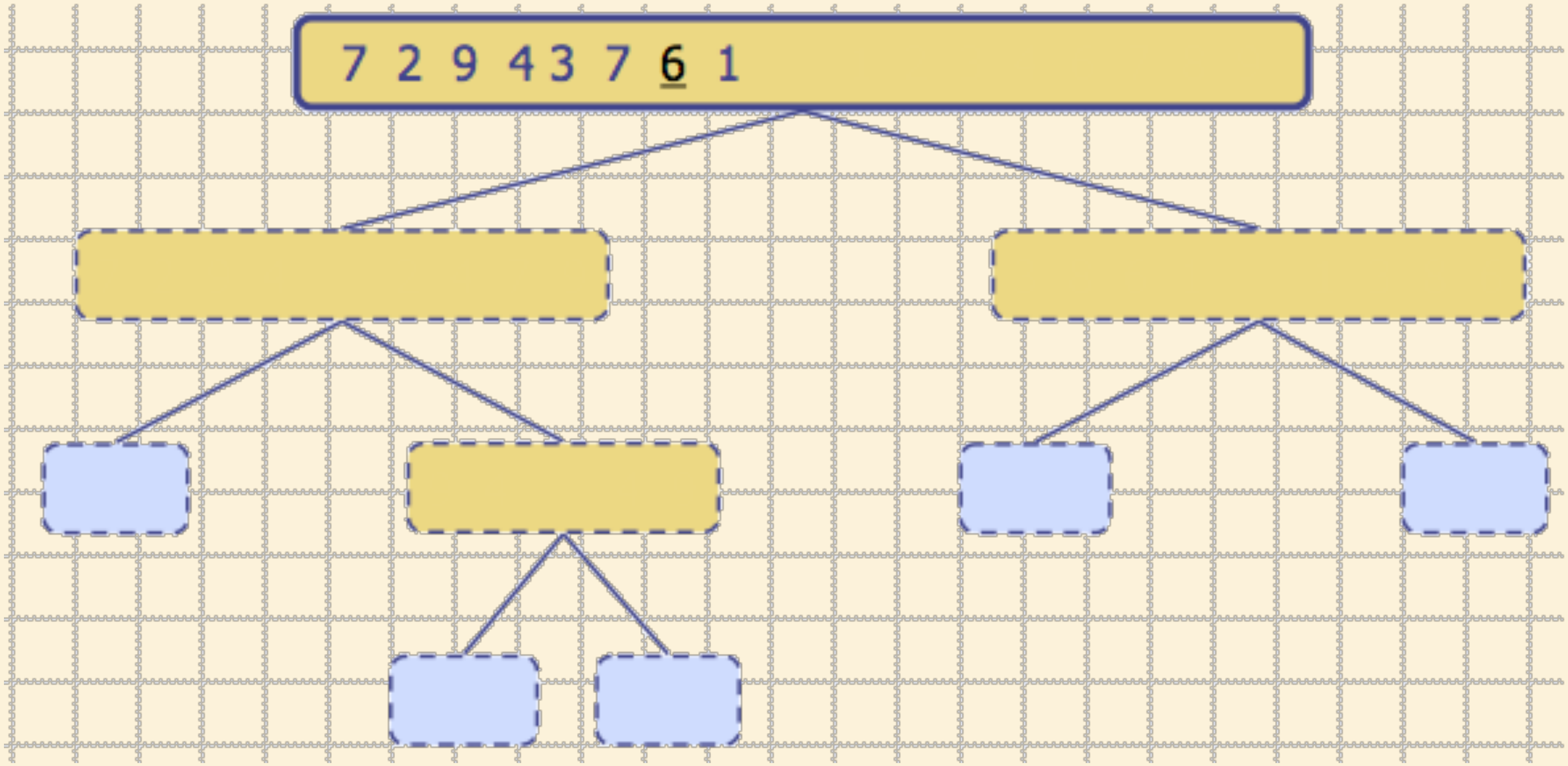
Quick-Sort Tree

- 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



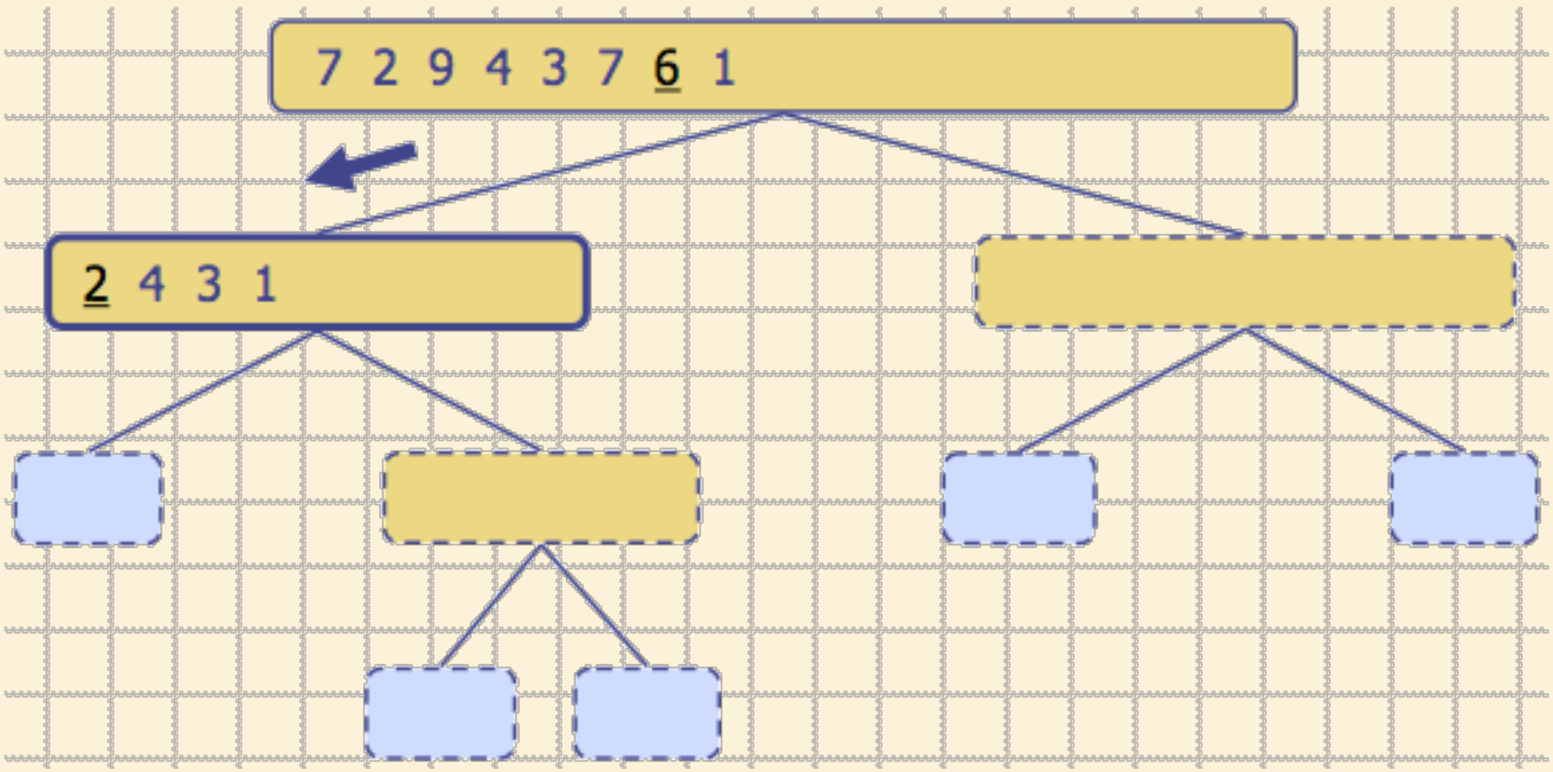
Execution Example

- Pivot selection



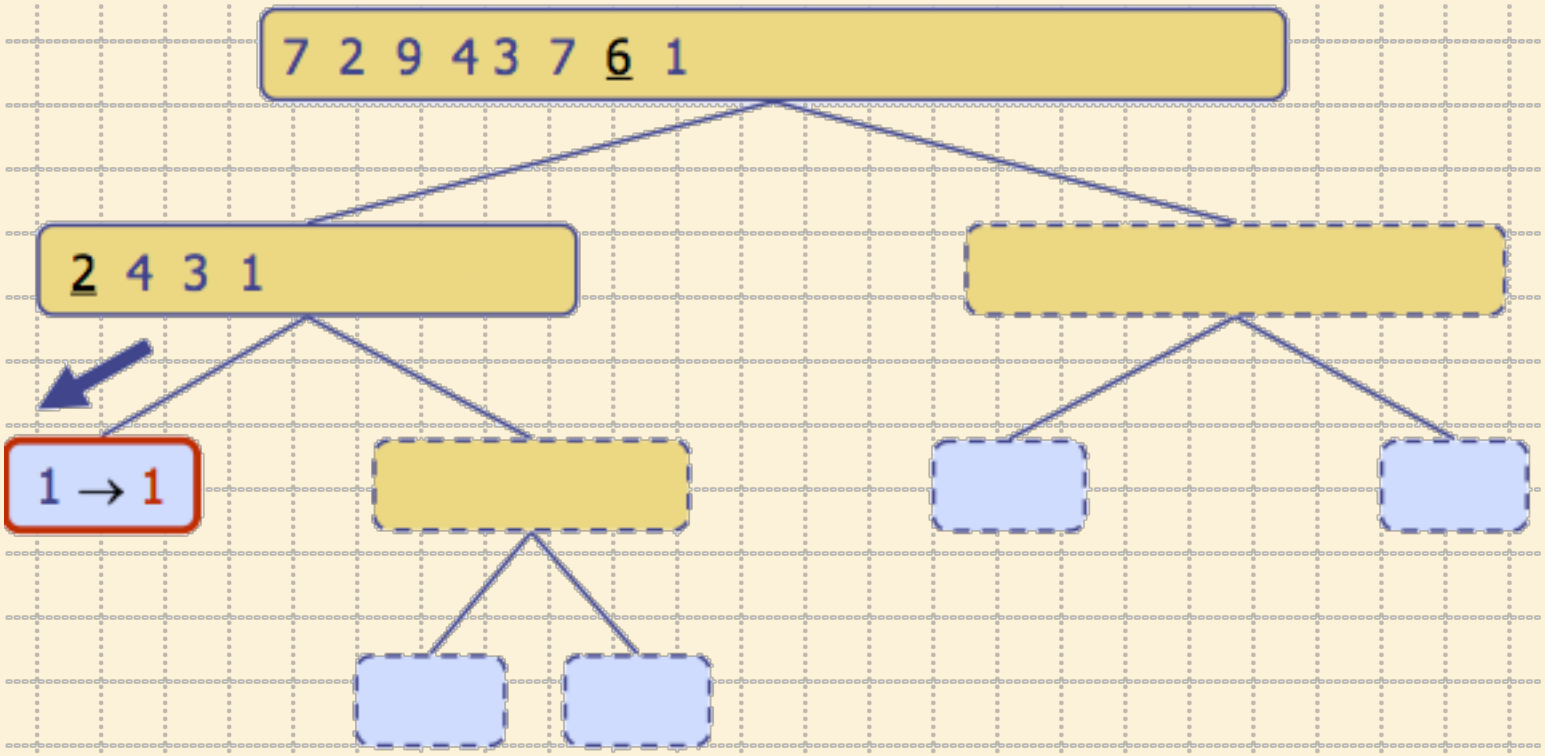
Execution Example (cont.)

- Partition, recursive call, pivot selection



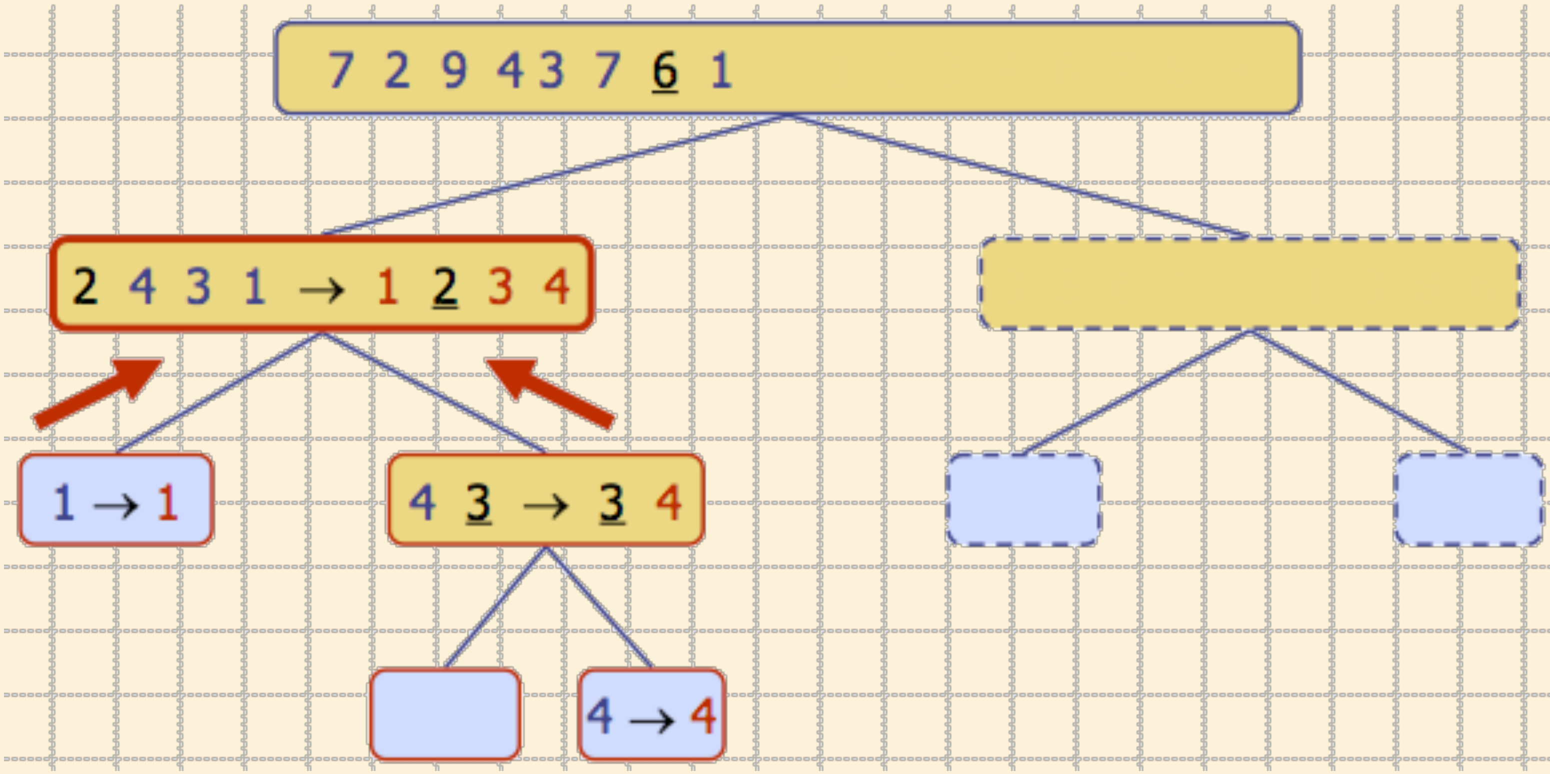
Execution Example (cont.)

- Partition, recursive call, base case



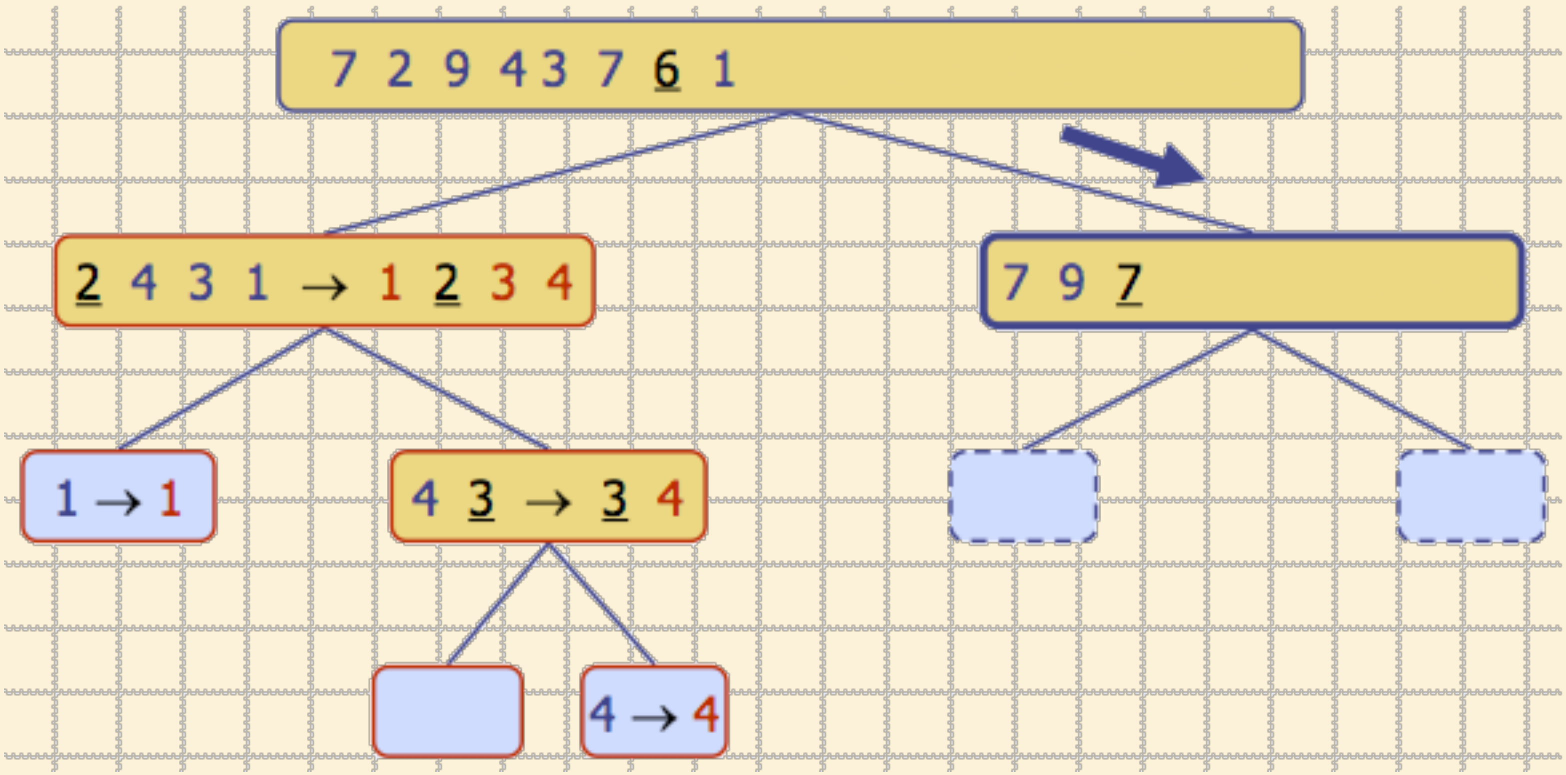
Execution Example (cont.)

- Recursive call, …, base case, join



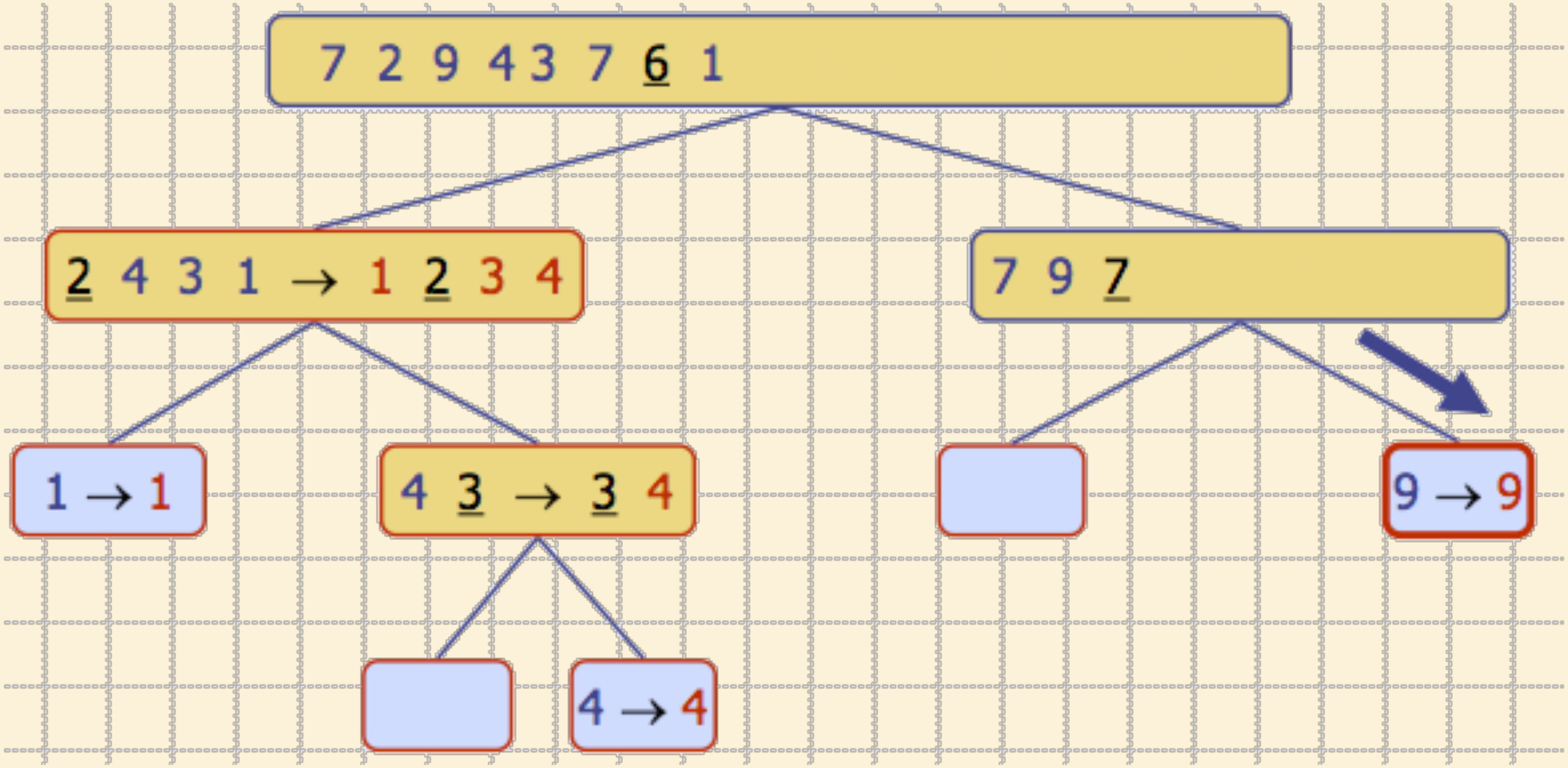
Execution Example (cont.)

- Recursive call, pivot selection



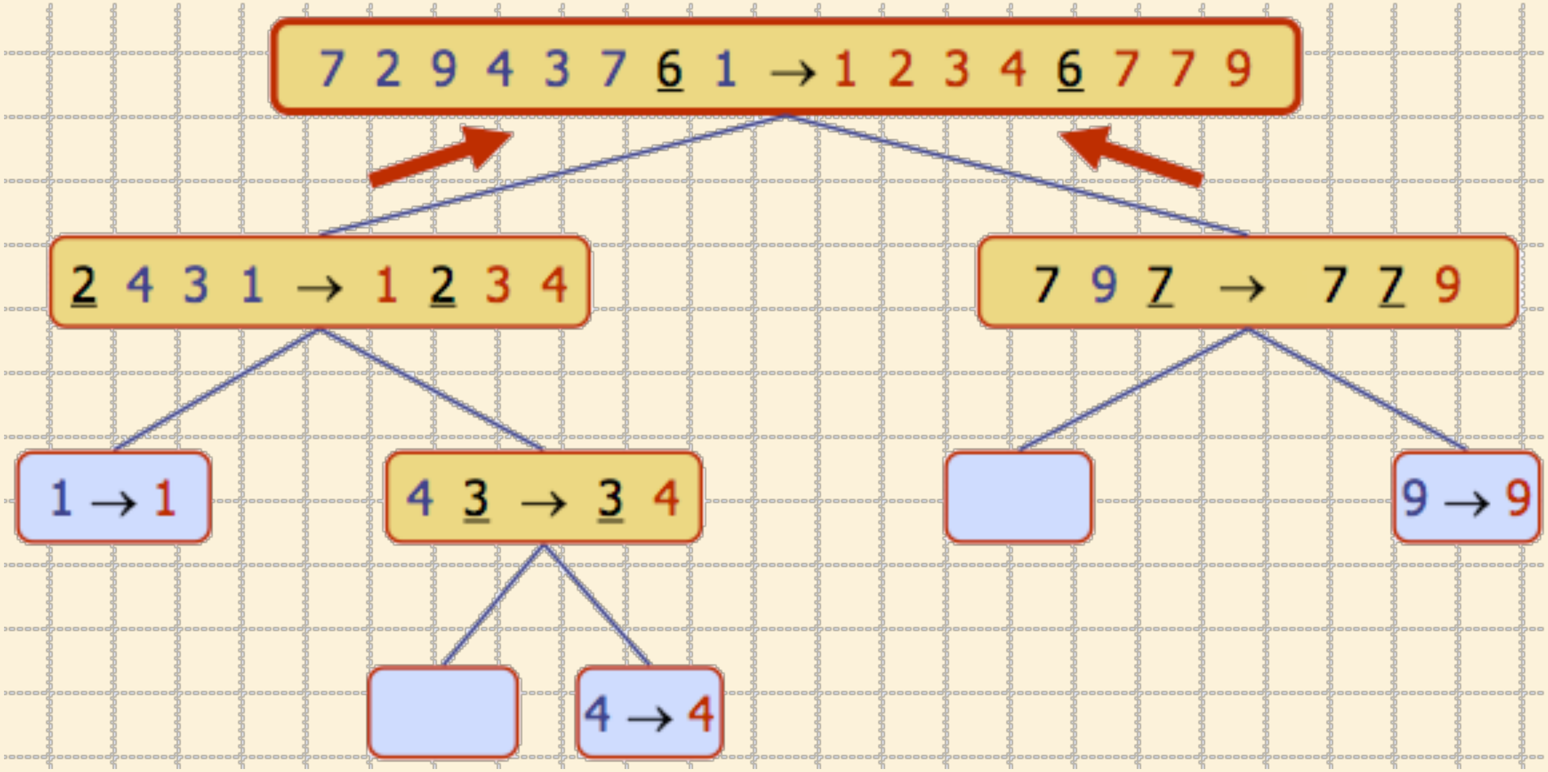
Execution Example (cont.)

- Partition, …, recursive call, base case



Execution Example (cont.)

- Join, join



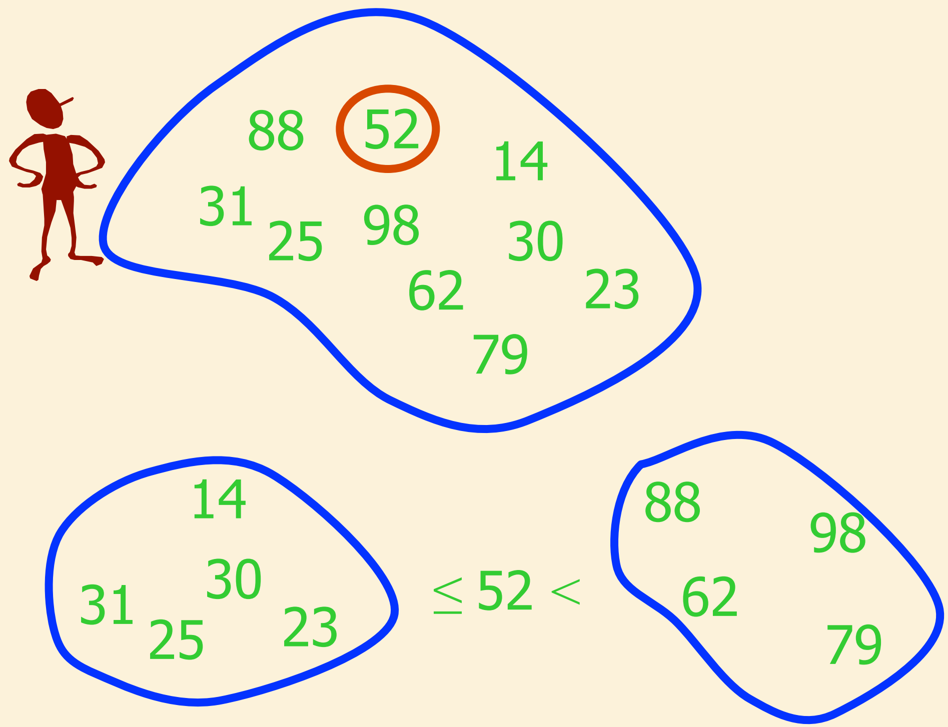
Quick-Sort Properties

- The algorithm just described is **stable**, since elements are removed from the beginning of the input sequence and placed on the end of the output sequences (L,E, G).
- However it **does not sort in place**: O(n) new memory is allocated for L, E and G
- Is there an in-place quick-sort?

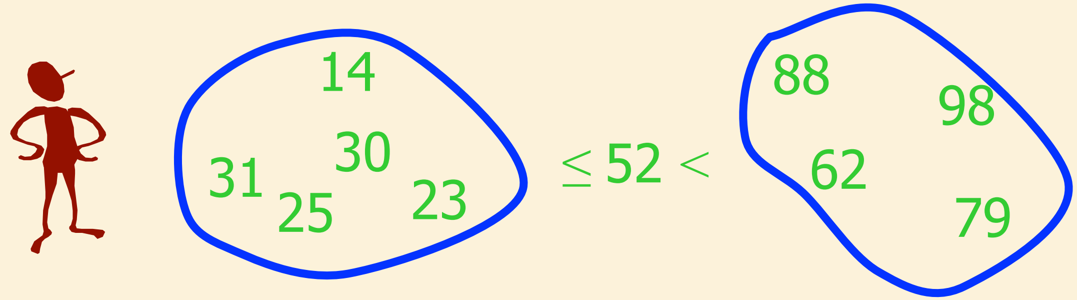
In-Place Quick-Sort

- **Note**: Use the lecture slides here instead of the textbook implementation (Section 11.2.2)

Partition set into **two** using randomly chosen pivot



In-Place Quick-Sort



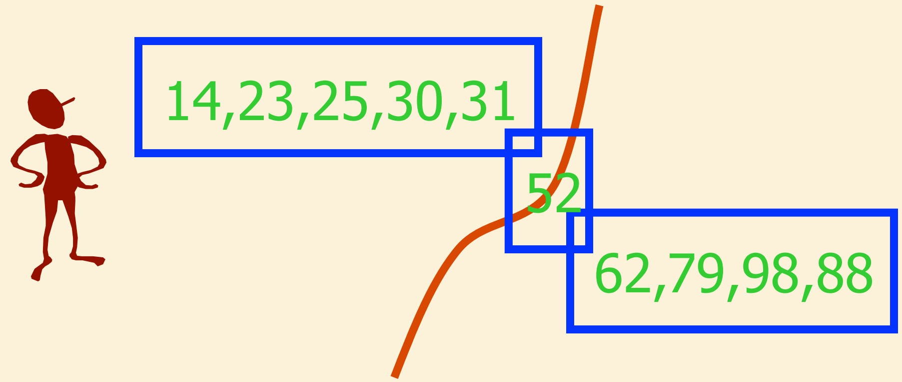
Get one friend to sort the first half.

14,23,25,30,31

Get one friend to sort the second half.

62,79,98,88

In-Place Quick-Sort



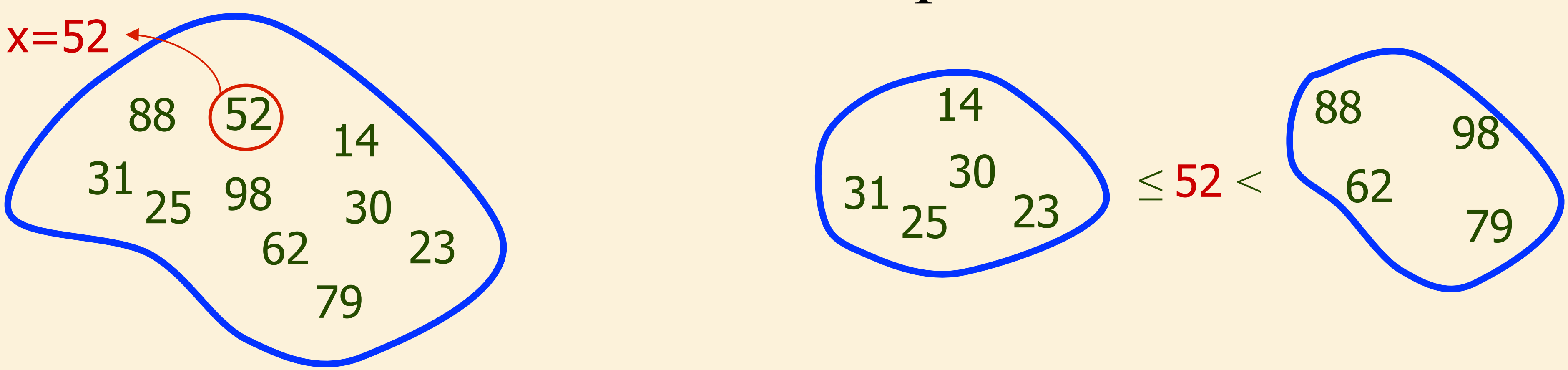
Glue pieces together.
(No real work)

14,23,25,30,31,52,62,79,88,98

The In-Place Partitioning Problem

Input:

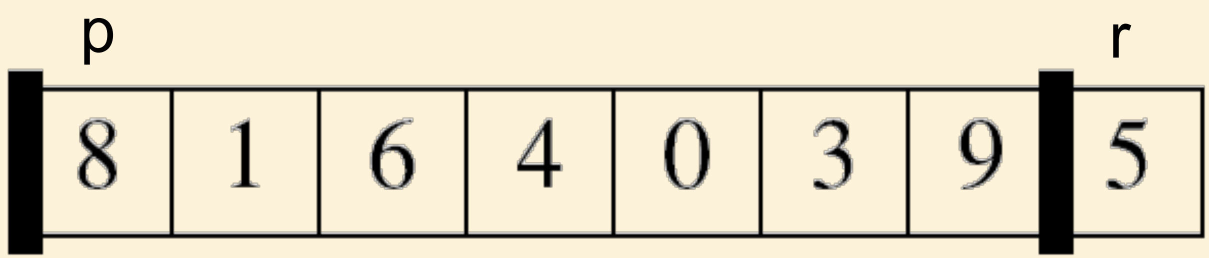
Output:



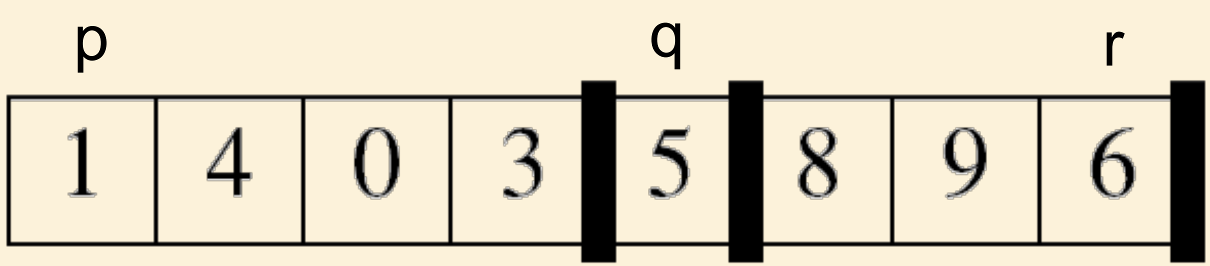
Problem: Partition a list into a set of small values and a set of large values.

Precise Specification

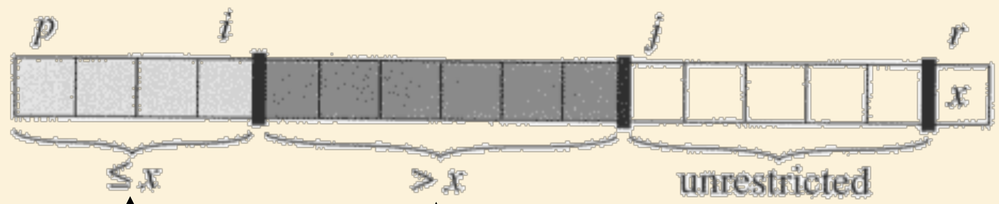
Precondition: $A[p...r]$ is an arbitrary list of values. $x = A[r]$ is the pivot.



Postcondition: A is rearranged such that $A[p...q-1] \le A[q] = x < A[q+1...r]$ for some q.



Loop Invariant



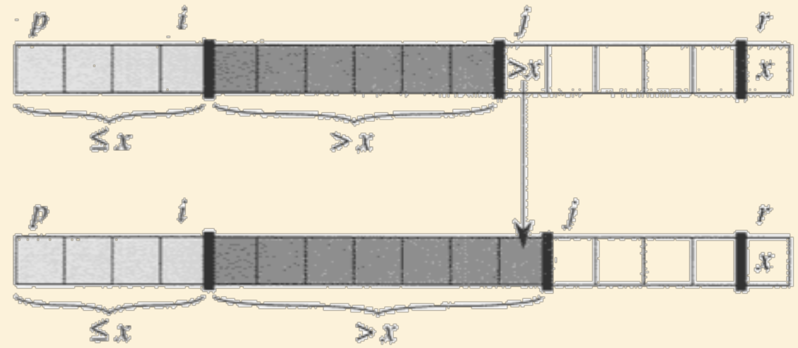
- 3 subsets are maintained
  - One containing values less than or equal to the pivot
  - One containing values greater than the pivot
  - One containing values yet to be processed

Loop invariant:

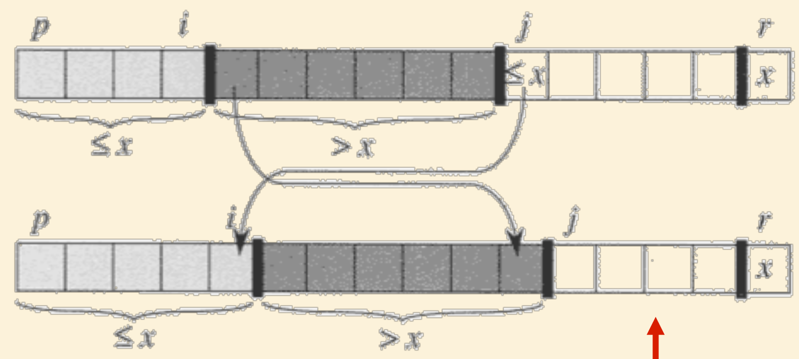
1. All entries in $A[p \mathinner{.\,.} i]$ are $\leq$ pivot.
2. All entries in $A[i+1 \mathinner{.\,.} j-1]$ are $>$ pivot.
3. $A[r] =$ pivot.

Maintaining Loop Invariant

- Consider element at location j
  - If greater than pivot, incorporate into '> set' by incrementing j.



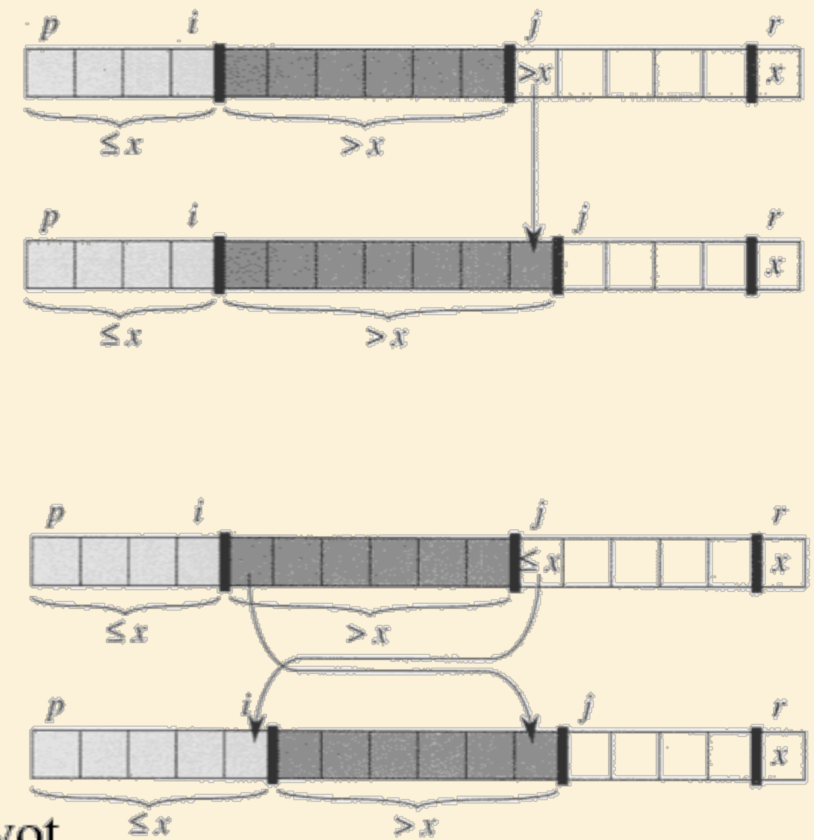
  - If less than or equal to pivot, incorporate into '≤ set' by swapping with element at location i+1 and incrementing both i and j.



  - Measure of progress: size of unprocessed set.

Maintaining Loop Invariant

```
PARTITION(A, p, r)
x ← A[r]
i ← p − 1
for j ← p to r − 1
        do if A[j] ≤ x
                then i ← i + 1
                        exchange A[i] ↔ A[j]
exchange A[i + 1] ↔ A[r]
return i + 1
```



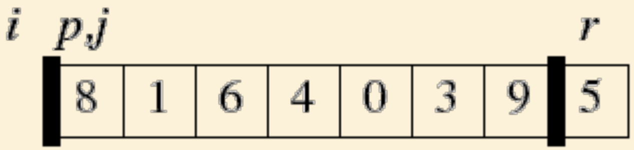
Loop invariant:

1. All entries in $A[p \,..\, i]$ are $\leq$ pivot.
2. All entries in $A[i+1 \,..\, j-1]$ are $>$ pivot.
3. $A[r] =$ pivot.

Establishing Loop Invariant

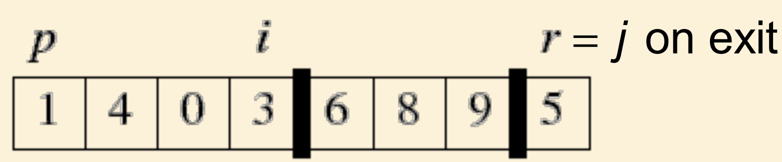
Loop invariant:

1. All entries in $A[p\,..\,i]$ are $\leq$ pivot.
2. All entries in $A[i+1\,..\,j-1]$ are $>$ pivot.
3. $A[r] =$ pivot.



Establishing Postcondition

```
PARTITION(A, p, r)
    x ← A[r]
    i ← p − 1
    for j ← p to r − 1
            do if A[j] ≤ x
                    then i ← i + 1
                            exchange A[i] ↔ A[j]
    exchange A[i + 1] ↔ A[r]
    return i + 1
```



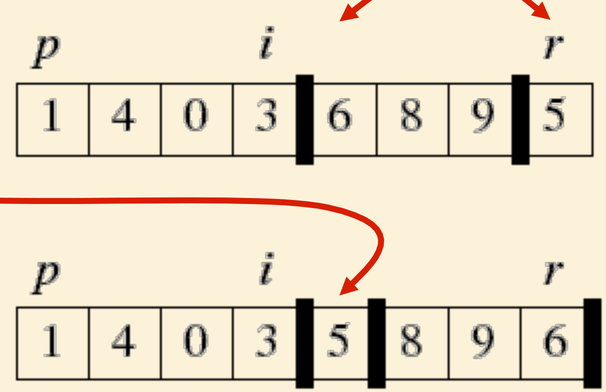
Exhaustive on exit

Loop invariant:

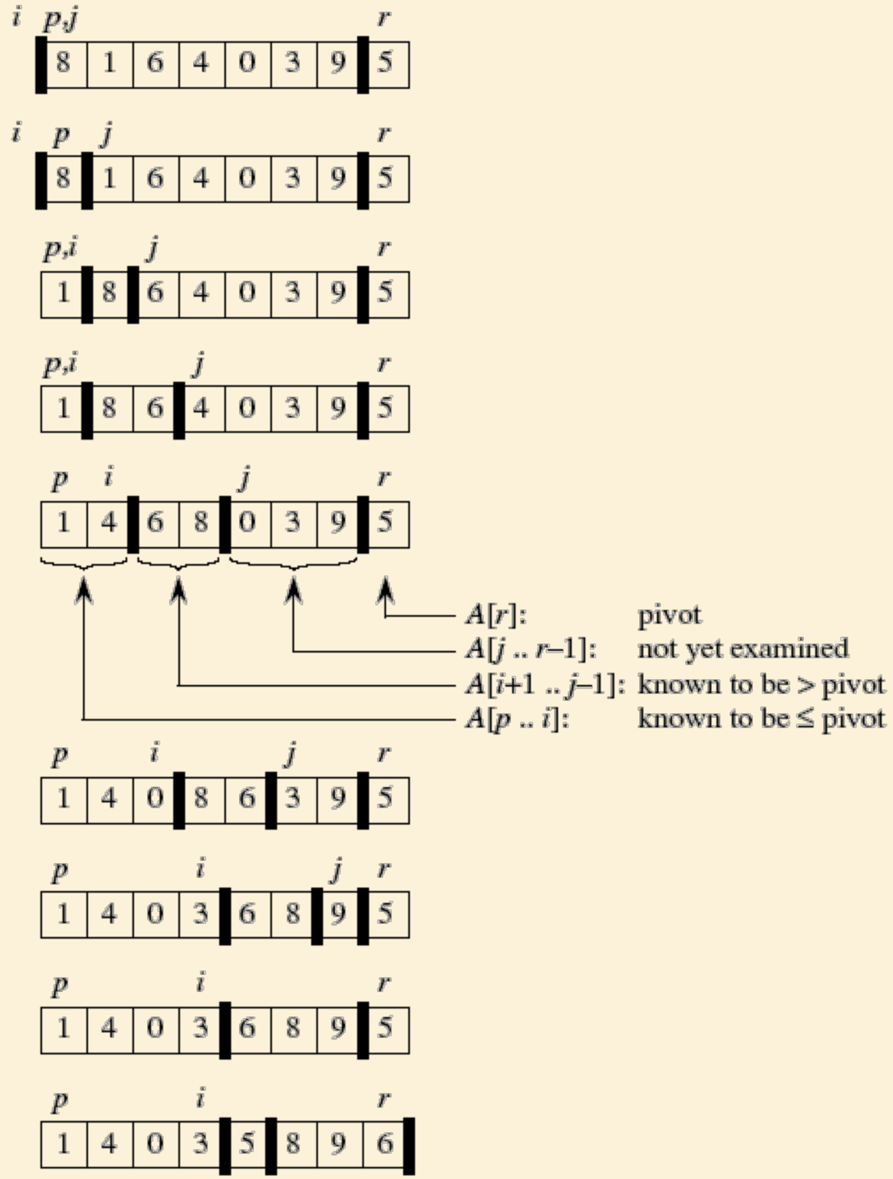
1. All entries in $A[p \,..\, i]$ are $\leq$ pivot.
2. All entries in $A[i+1 \,..\, j-1]$ are $>$ pivot.
3. $A[r] =$ pivot.

Establishing Postcondition

```
PARTITION(A, p, r)
  x ← A[r]
  i ← p − 1
  for j ← p to r − 1
        do if A[j] ≤ x
              then i ← i + 1
                   exchange A[i] ↔ A[j]
  exchange A[i + 1] ↔ A[r]
  return i + 1
```

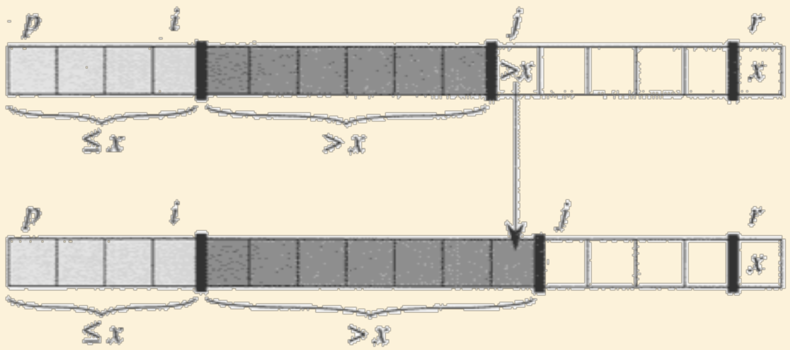


An Example

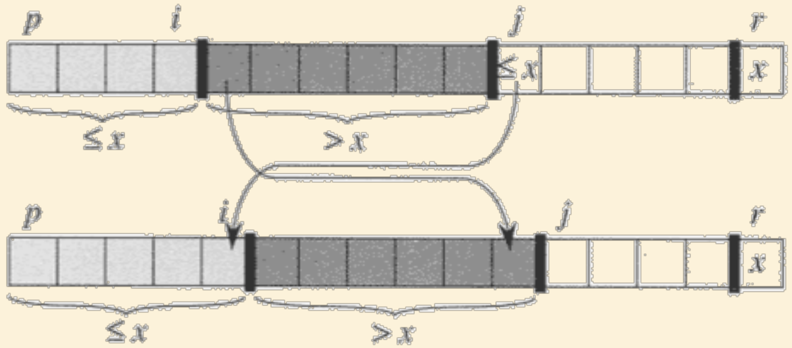

i p,j r
8 1 6 4 0 3 9 5
i p j r
8 1 6 4 0 3 9 5
p,i j r
1 8 6 4 0 3 9 5
p,i j r
1 8 6 4 0 3 9 5
p i j r
1 4 6 8 0 3 9 5
A[r]: pivot
A[j .. r–1]: not yet examined
A[i+1 .. j–1]: known to be > pivot
A[p .. i]: known to be ≤ pivot
p i j r
1 4 0 8 6 3 9 5
p i j r
1 4 0 3 6 8 9 5
p i r
1 4 0 3 6 8 9 5
p i r
1 4 0 3 5 8 9 6

In-Place Partitioning: Running Time

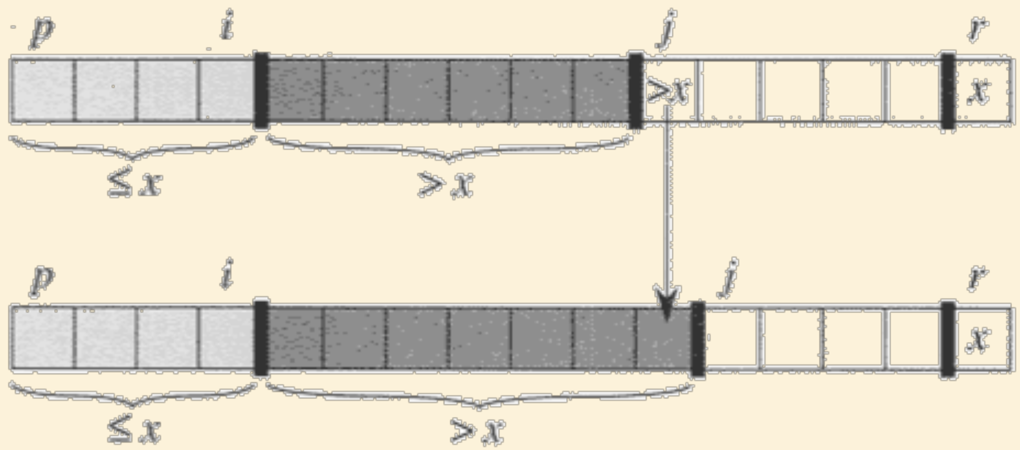
Each iteration takes O(1) time → Total = O(n)



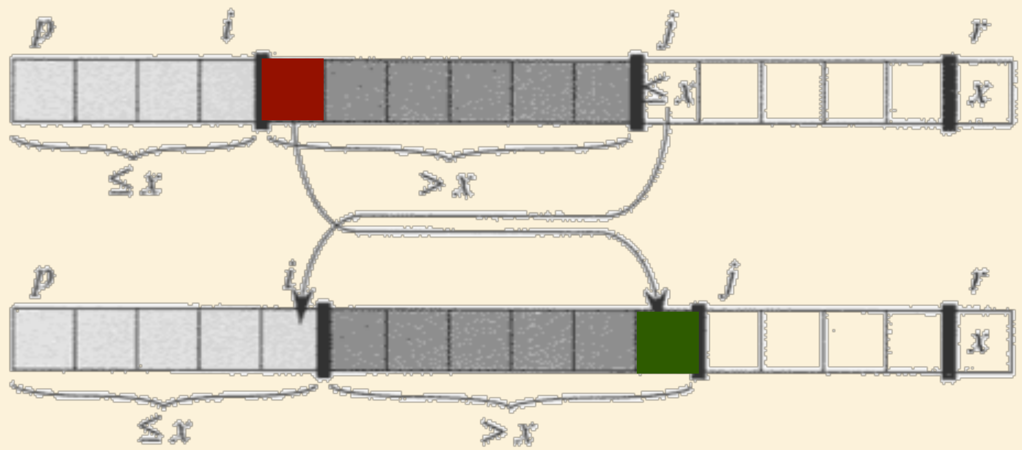
or



In-Place Partitioning is NOT Stable



or



The In-Place Quick-Sort Algorithm

Algorithm QuickSort(A, p, r)

if p < r

q = Partition(A, p, r)

QuickSort(A, p, q - 1) //Small elements are sorted

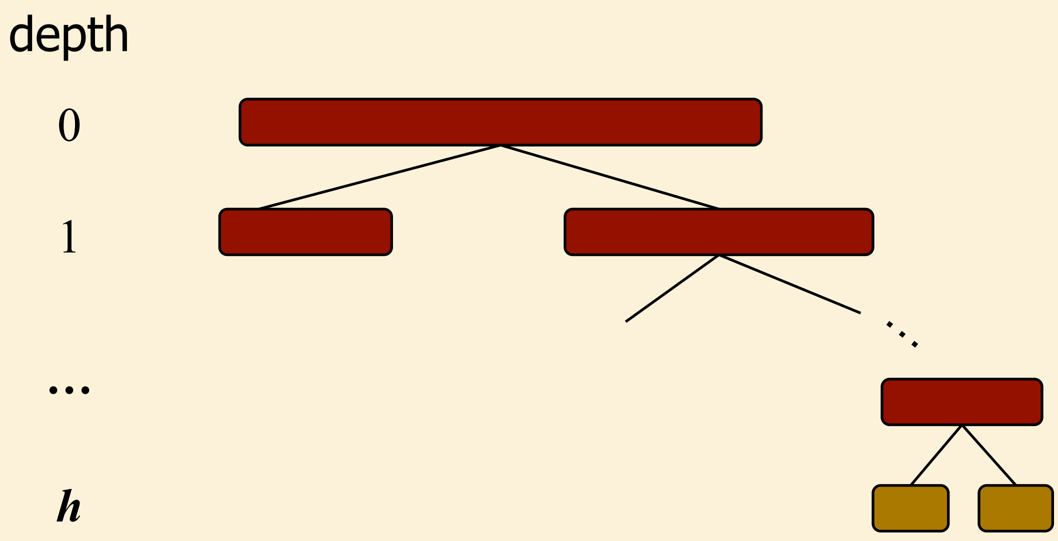
QuickSort(A, q + 1, r) //Large elements are sorted

//Thus input is sorted

Running Time of Quick-Sort

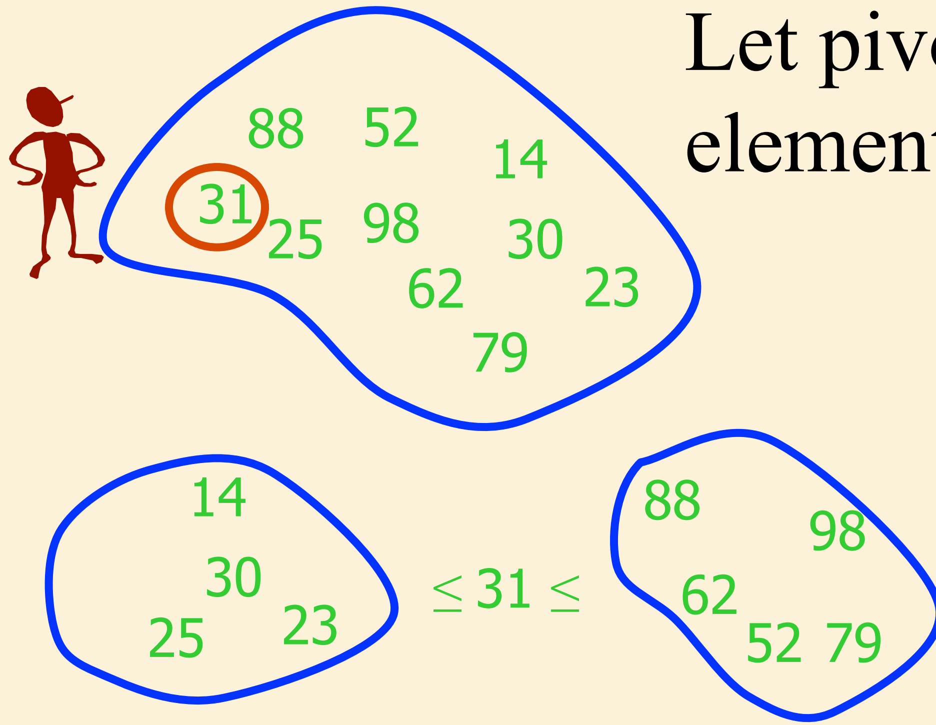
Quick-Sort Running Time

- We can analyze the running time of Quick-Sort using a recursion tree.
- At depth i of the tree, the problem is partitioned into 2^i sub-problems.
- The running time will be determined by how balanced these partitions are.

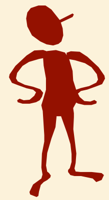


Quick Sort

Let pivot be the first element in the list?



Quick Sort



$\boxed{14,23,25,30,31,52,62,79,88,98}$

$\leq 14 <$ $\boxed{23,25,30,31,52,62,79,88,98}$

If the list is already sorted,
then the list is worst case unbalanced.

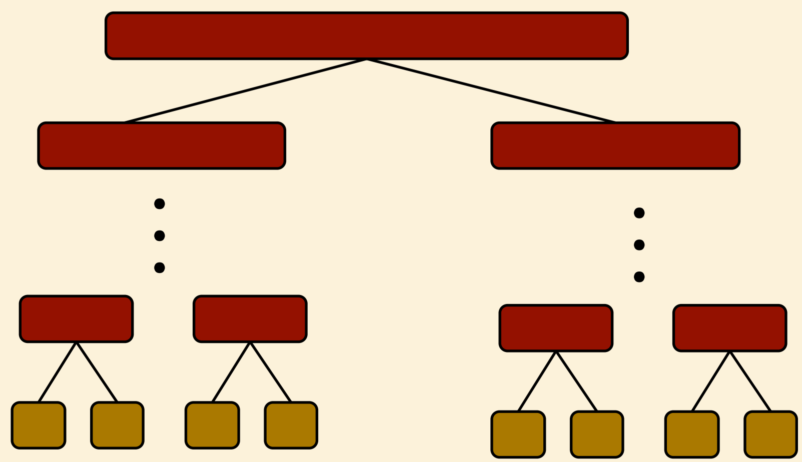
QuickSort: Choosing the Pivot

- Common choices are:
  - random element
  - middle element
  - median of first, middle and last element

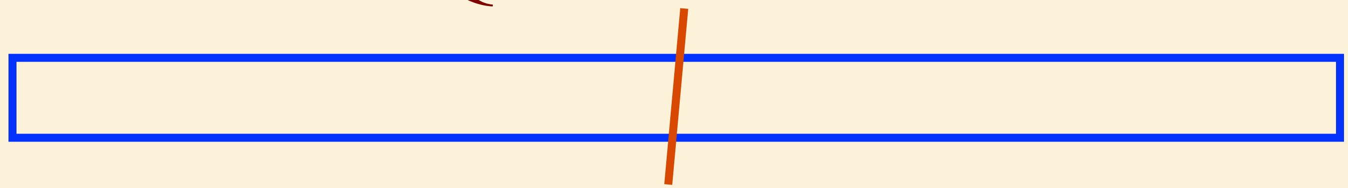
Best-Case Running Time

- The best case for quick-sort occurs when each pivot partitions the array in half.
- Then there are O(log n) levels
- There is O(n) work at each level
- Thus total running time is O(n log n)

depth	time
0	n
1	n
…	…
i	n
…	…
$\log n$	n



Quick Sort



Best Time: $T(n) = 2T(n/2) + \Theta(n)$

$= \Theta(n \log n)$

Worst Time:

Expected Time:

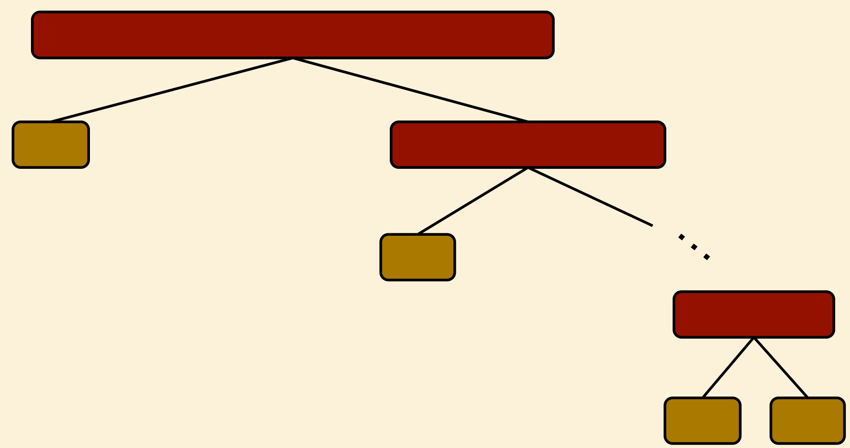
Worst-case Running Time

- The worst case for quick-sort occurs when the pivot is the unique minimum or maximum element
- One of L and G has size $n-1$ and the other has size 0
- The running time is proportional to the sum

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

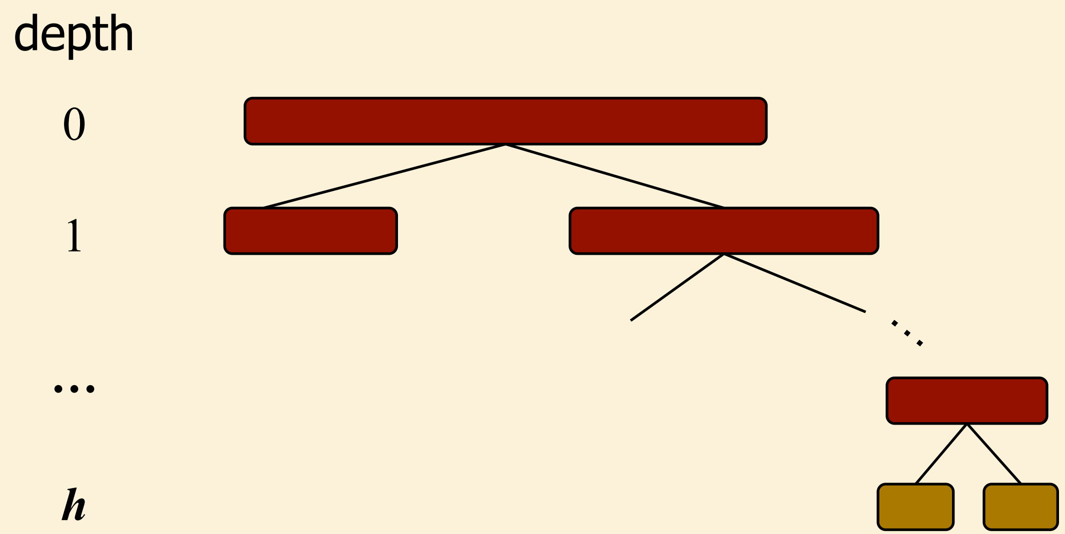
- Thus, the worst-case running time of quick-sort is $O(n^2)$

depth	time
0	n
1	$n-1$
…	…
$n-1$	1



Average-Case Running Time

- If the pivot is selected randomly, the average-case running time for Quick Sort is O(n log n).
- Proving this requires a probabilistic analysis.
- We will simply provide an intution for why average-case O(n log n) is reasonable.

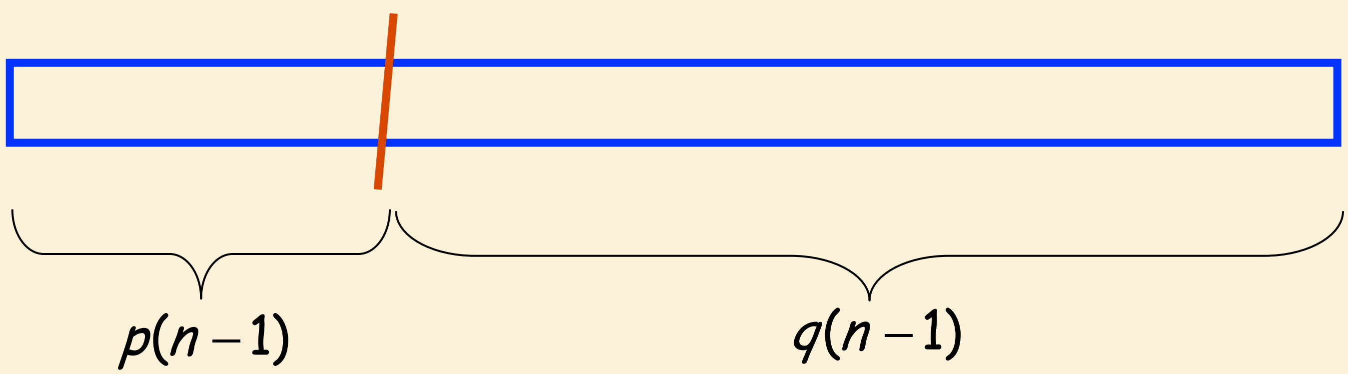


Expected Time Complexity for Quick Sort

Q: Why is it reasonable to expect $O(n \log n)$ time complexity?

A: Because on average, the partition is not too unbalanced.

Example: Imagine a deterministic partition,
in which the 2 subsets are always in fixed proportion, i.e.,
$p(n-1)$ & $q(n-1)$, where p,q are constants, $p,q \in [0...1], p+q=1$.



Expected Time Complexity for Quick Sort

Then $T(n) = T(p(n-1)) + T(q(n-1)) + O(n)$

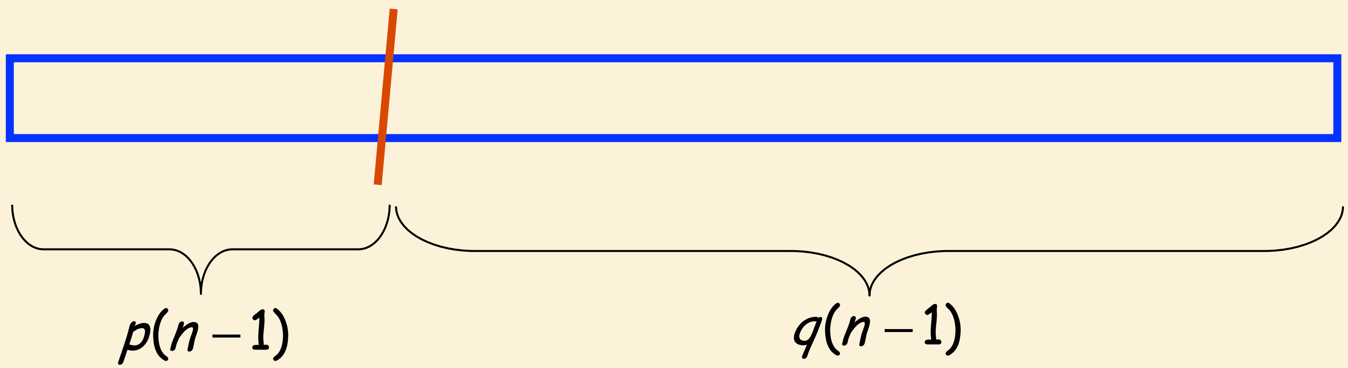
wlog, suppose that $q > p$.

Let k be the depth of the recursion tree

Then $q^k n = 1 \rightarrow k = \log n / \log(1/q)$

Thus $k \in O(\log n)$:

$O(n)$ work done per level $\rightarrow T(n) = O(n \log n)$.



Properties of QuickSort

- In-place? ✓
- Stable? ✓

But not both!

- Fast?
  - Depends.
  - Worst Case: $\Theta(n^2)$
  - Expected Case: $\Theta(n \log n)$, with small constants

Summary of Comparison Sorts

Algorithm	Best Case	Worst Case	Average Case	In Place	Stable	Comments
Selection	n^2	n^2		Yes	Yes	
Bubble	n	n^2		Yes	Yes	Must count swaps for linear best case running time.
Insertion	n	n^2		Yes	Yes	Good if often almost sorted
Merge	n log n	n log n		No	Yes	Good for very large datasets that require swapping to disk
Heap	n log n	n log n		Yes	No	Best if guaranteed n log n required
Quick	n log n	n^2	n log n	Yes	Yes	Usually fastest in practice

But not both!

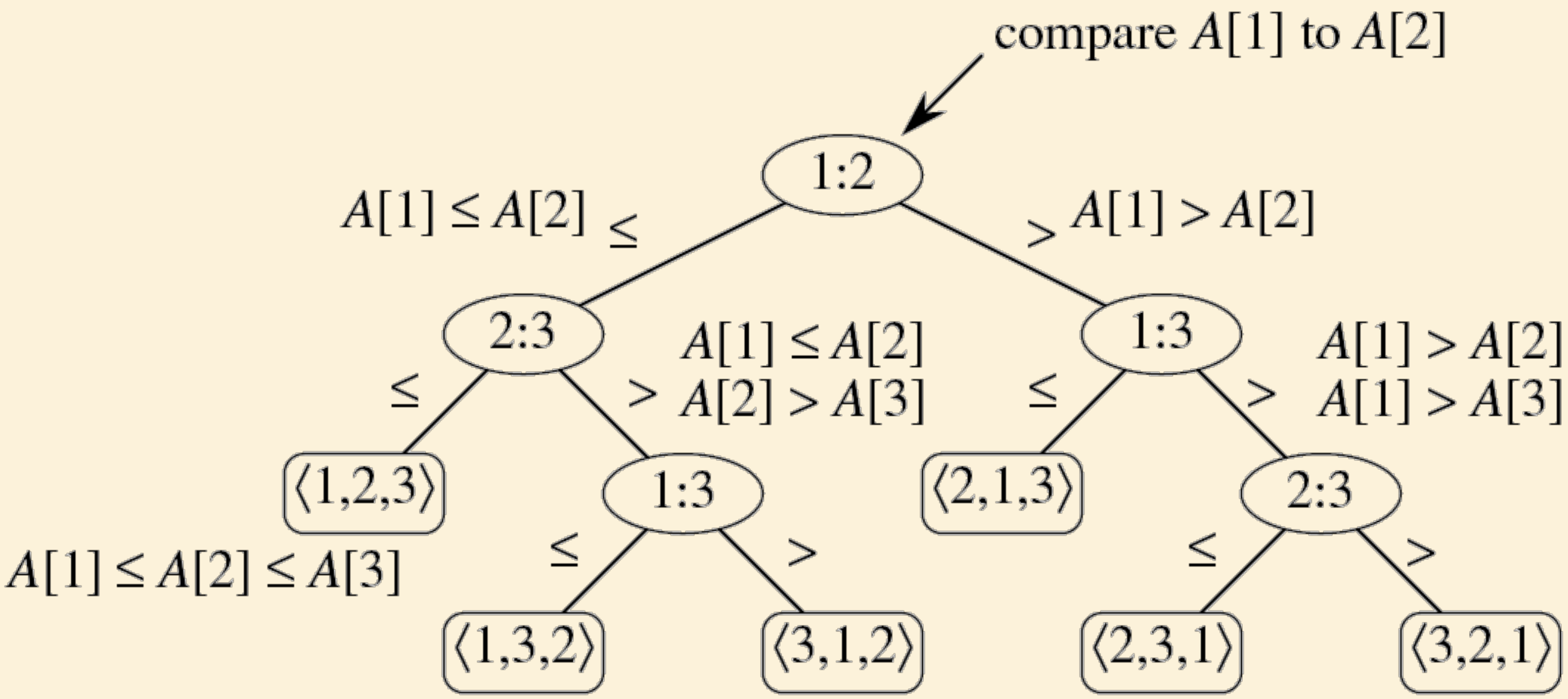
Comparison Sort: Lower Bound

MergeSort and HeapSort are both $\theta(n \log n)$ (worst case).

Can we do better?

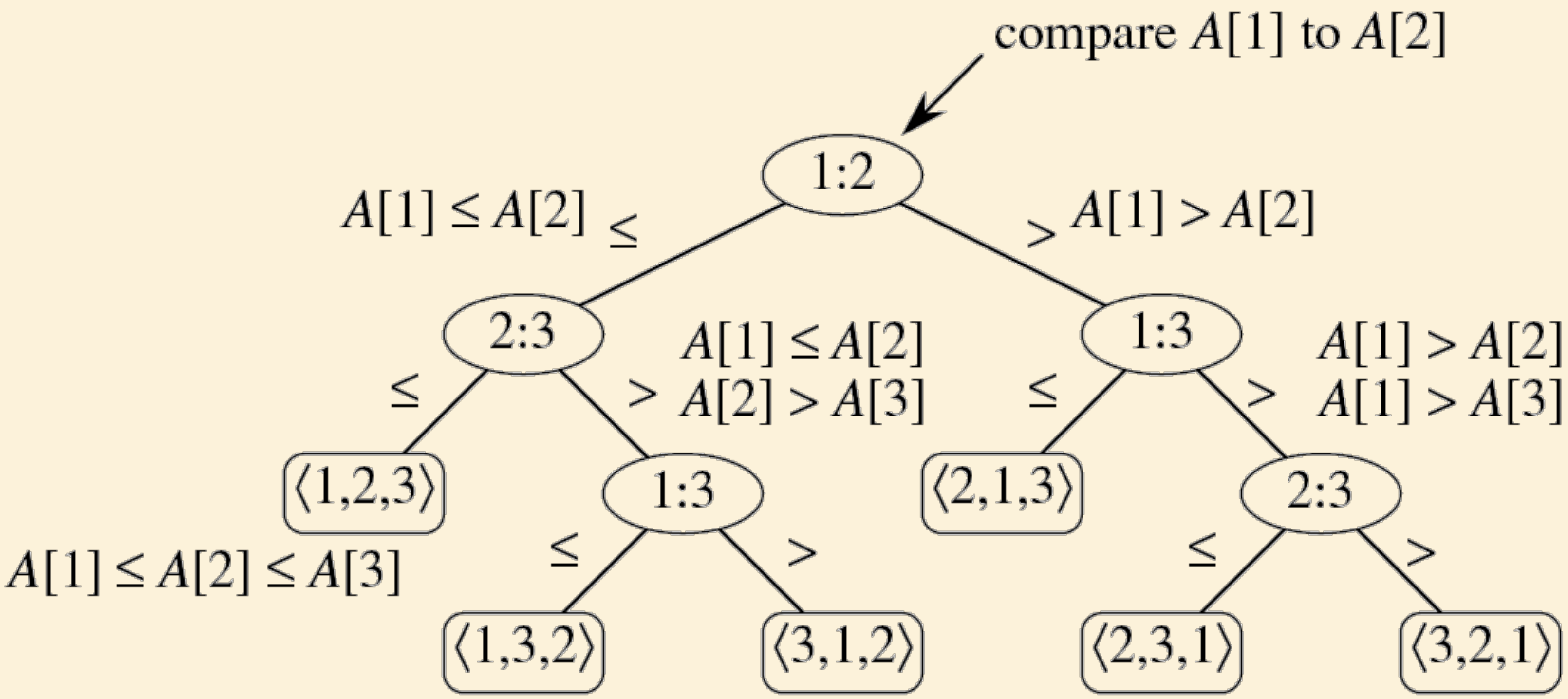
Comparison Sort: Decision Trees

- Example: Sorting a 3-element array A[1..3]



Comparison Sort: Decision Trees

- For a 3-element array, there are 6 external nodes.
- For an n-element array, there are *n*! external nodes.



Comparison Sort

- To store n! external nodes, a decision tree must have a height of at least $\lceil \log n! \rceil$
- Worst-case time is equal to the height of the binary decision tree.

$$\text{Thus } T(n) \in \Omega\left(\log n!\right)$$

$$\text{where} \log n! = \sum_{i=1}^{n} \log i \geq \sum_{i=1}^{\lfloor n/2 \rfloor} \log \lfloor n/2 \rfloor \in \Omega(n \log n)$$

$$\text{Thus } T(n) \in \Omega(n \log n)$$

Thus MergeSort & HeapSort are asymptotically optimal.

Linear Sorts?

Comparison sorts are very general, but are $\Omega(n \log n)$

Faster sorting may be possible if we can constrain the nature of the input.

Example 1. Counting Sort

- Invented by Harold Seward in 1954.
- Counting Sort applies when the elements to be sorted come from a finite (and preferably small) set.
- For example, the elements to be sorted are integers in the range [0…k-1], for some fixed integer k.
- We can then create an array V[0…k-1] and use it to count the number of elements with each value [0…k-1].
- Then each input element can be placed in exactly the right place in the output array in constant time.

Counting Sort

Input:	1	0	0	1	3	1	1	3	1	0	2	1	0	1	1	2	2	1	0
Output:	0	0	0	0	0	1	1	1	1	1	1	1	1	1	2	2	3	3	3

- Input: N records with integer keys between [0…3].
- Output: Stable sorted keys.
- Algorithm:
  - Count frequency of each key value to determine transition locations
  - Go through the records in order putting them where they go.

CountingSort

Input:	1	0	0	1	3	1	1	3	1	0	2	1	0	1	1	2	2	1	0
Output:	0	0	0	0	0	1	1	1	1	1	1	1	1	1	2	2	2	3	3
Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Stable sort: If two keys are the same, their order does not change.

Thus the 4th record in input with digit 1 must be
the 4th record in output with digit 1.

It belongs at output index 8, because 8 records go before it
ie, 5 records with a smaller digit & 3 records with the same digit

Count These!

CountingSort

Input:	1	0	0	1	3	1	1	3	1	0	2	1	0	1	1	2	2	1	0
Output:																			
Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Value v:	0	1	2	3
# of records with digit v:	5	9	3	2

N records. Time to count? $\theta(N)$

CountingSort

Input:	1	0	0	1	3	1	1	3	1	0	2	1	0	1	1	2	2	1	0
Output:																			
Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Value v:	0	1	2	3
# of records with digit v:	5	9	3	3
# of records with digit < v:	0	5	14	17

N records, k different values. Time to count? $\theta(k)$

CountingSort

Input:	1	0	0	1	3	1	1	3	1	0	2	1	0	1	1	2	2	1	0
Output:	0	0	0	0	0	1	1	1	1	1	1	1	1	1	2	2	2	3	3
Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Value v:	0	1	2	3
# of records with digit < v:	0	5	14	17

= location of first record with digit v.

CountingSort

Input:	1	0	0	1	3	1	1	3	1	0	2	1	0	1	1	2	2	1	0
Output:	0	?				1													
Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Value v:	0	1	2	3
Location of first record with digit v.	0	5	14	17

Algorithm: Go through the records in order putting them where they go.

Loop Invariant

- The first *i-1* keys have been placed in the correct locations in the output array
- The auxiliary data structure *v* indicates the location at which to place the i^{th} key for each possible key value from *[0..k-1]*.

CountingSort

Input:	~~1~~	0	0	1	3	1	1	3	1	0	2	1	0	1	1	2	2	1	0
Output:						1													
Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Value v:	0	1	2	3
Location of **next** record with digit v.	0	5	14	17

Algorithm: Go through the records in order putting them where they go.

CountingSort

Input:	~~1~~	~~0~~	0	1	3	1	1	3	1	0	2	1	0	1	1	2	2	1	0
Output:	0					1													
Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Value v:	0	1	2	3
Location of next record with digit v.	0	6	14	17

Algorithm: Go through the records in order putting them where they go.

CountingSort

Input:	~~1~~	~~0~~	~~0~~	1	3	1	1	3	1	0	2	1	0	1	1	2	2	1	0
Output:	0	0				1													
Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Value v:	0	1	2	3
Location of next record with digit v.	1	6	14	17

Algorithm: Go through the records in order
putting them where they go.

CountingSort

Input:	~~1~~	~~0~~	~~0~~	~~1~~	3	1	1	3	1	0	2	1	0	1	1	2	2	1	0
Output:	0	0				1	1												
Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Value v:	0	1	2	3
Location of next record with digit v.	2	6	14	17

Algorithm: Go through the records in order putting them where they go.

CountingSort

Input:	~~1~~	~~0~~	~~0~~	~~1~~	~~3~~	1	1	3	1	0	2	1	0	1	1	2	2	1	0
Output:	0	0				1	1											3	
Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Value v:	0	1	2	3
Location of next record with digit v.	2	7	14	17

Algorithm: Go through the records in order putting them where they go.

CountingSort

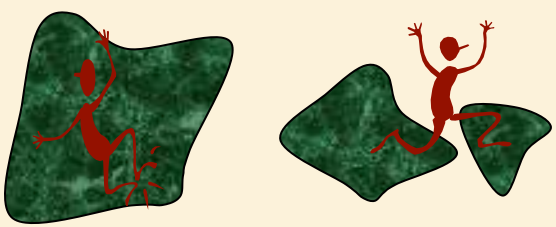
Input:	~~1~~	~~0~~	~~0~~	~~1~~	~~3~~	~~1~~	1	3	1	0	2	1	0	1	1	2	2	1	0
Output:	0	0				1	1	1										3	
Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Value v:	0	1	2	3
Location of next record with digit v.	2	7	14	18

Algorithm: Go through the records in order putting them where they go.

CountingSort

Input:	~~1~~	~~0~~	~~0~~	~~1~~	~~3~~	~~1~~	~~1~~	3	1	0	2	1	0	1	1	2	2	1	0
Output:	0	0				1	1	1	1									3	
Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18



Value v:	0	1	2	3
	2	8	14	18

Location of next record with digit v.

Algorithm: Go through the records in order putting them where they go.

CountingSort

Input:	~~1~~	~~0~~	~~0~~	~~1~~	~~3~~	~~1~~	~~1~~	~~3~~	1	0	2	1	0	1	1	2	2	1	0
Output:	0	0				1	1	1	1									3	3
Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Value v:	0	1	2	3
Location of **next** record with digit v.	2	9	14	18

Algorithm: Go through the records in order putting them where they go.

CountingSort

Input:	~~1~~	~~0~~	~~0~~	~~1~~	~~3~~	~~1~~	~~1~~	~~3~~	~~1~~	0	2	1	0	1	1	2	2	1	0
Output:	0	0				1	1	1	1	1								3	3
Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Value v:	0	1	2	3
Location of next record with digit v.	2	9	14	19

Algorithm: Go through the records in order putting them where they go.

CountingSort

Input:	~~1~~	~~0~~	~~0~~	~~1~~	~~3~~	~~1~~	~~1~~	~~3~~	~~1~~	~~0~~	2	1	0	1	1	2	2	1	0
Output:	0	0	0			1	1	1	1	1								3	3
Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Value v:	0	1	2	3
Location of next record with digit v.	2	10	14	19

Algorithm: Go through the records in order putting them where they go.

CountingSort

Input:	~~1~~	~~0~~	~~0~~	~~1~~	~~3~~	~~1~~	~~1~~	~~3~~	~~1~~	~~0~~	~~2~~	1	0	1	1	2	2	1	0
Output:	0	0	0			1	1	1	1	1					2			3	3
Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Value v:	0	1	2	3
Location of next record with digit v.	3	10	14	19

Algorithm: Go through the records in order putting them where they go.

CountingSort

Input:	~~1~~	~~0~~	~~0~~	~~1~~	~~3~~	~~1~~	~~1~~	~~3~~	~~1~~	~~0~~	~~2~~	~~1~~	0	1	1	2	2	1	0
Output:	0	0	0			1	1	1	1	1	1				2			3	3
Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Value v:	0	1	2	3
Location of next record with digit v.	3	10	15	19

Algorithm: Go through the records in order putting them where they go.

CountingSort

Input:	~~1~~	~~0~~	~~0~~	~~1~~	~~3~~	~~1~~	~~1~~	~~3~~	~~1~~	~~0~~	~~2~~	~~1~~	0	1	1	2	2	1	0
Output:	0	0	0			1	1	1	1	1	1				2			3	3
Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Value v:	0	1	2	3
Location of next record with digit v.	3	10	15	19

Algorithm: Go through the records in order putting them where they go.

CountingSort

Input:	~~1~~	~~0~~	~~0~~	~~1~~	~~3~~	~~1~~	~~1~~	~~3~~	~~1~~	~~0~~	~~2~~	~~1~~	~~0~~	~~1~~	~~1~~	~~2~~	~~2~~	~~1~~	~~0~~
Output:	0	0	0	0	0	1	1	1	1	1	1	1	1	1	2	2	2	3	3
Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Value v:	0	1	2	3
	5	14	17	19

Location of next record with digit v.

Time = $\theta(N)$

Total = $\theta(N+k)$

Example 2. RadixSort

344	125
125	224
333	225
134	325
224	333
334	134
143	334
225	344
325	143
243	243

Input:

- An array of *N* numbers.
- Each number contains *d* digits.
- Each digit between *[0…k-1]*

Output:

- Sorted numbers.

Digit Sort:

- Select one digit
- Separate numbers into k piles based on selected digit (e.g., Counting Sort).

Stable sort: If two cards are the same for that digit, their order does not change.

RadixSort

344		125		125
125		134		224
333	Sort wrt which digit first?	143	Sort wrt which digit Second?	225
134		224		325
224		225		134
334	The most significant.	243	The next most significant.	333
143		344		334
225		333		143
325		334		243
243		325		344

All meaning in first sort lost.

RadixSort

344
125
333
134
224
334
143
225
325
243

Sort wrt which digit first?

The least significant.

333
143
243
344
134
224
334
125
225
325

Sort wrt which digit Second?

The next least significant.

224
125
225
325
333
134
334
143
243
344

RadixSort

344
125
333
134
224
334
143
225
325
243

Sort wrt which digit first?

The least significant.

333
143
243
344
134
224
334
125
225
325

Sort wrt which digit Second?

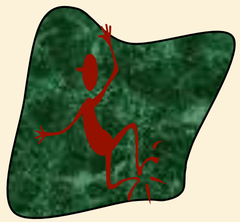
The next least significant.

2 24
1 25
2 25
3 25
3 33
1 34
3 34
1 43
2 43
3 44

Is sorted wrt least sig. 2 digits.

RadixSort

i+1	
2	24
1	25
2	25
3	25
3	33
1	34
3	34
1	43
2	43
3	44



Is sorted wrt first i digits.

Sort wrt i+1st digit.

1	25
1	34
1	43
2	24
2	25
2	43
3	25
3	33
3	34
3	44

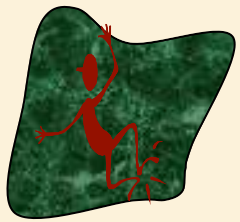
Is sorted wrt first i+1 digits.

These are in the correct order because sorted wrt high order digit

RadixSort

2	24
1	25
2	25
3	25
3	33
1	34
3	34
1	43
2	43
3	44

i+1



Is sorted wrt first i digits.

Sort wrt i+1st digit.

1	25
1	34
1	43
2	24
2	25
2	43
3	25
3	33
3	34
3	44

Is sorted wrt first i+1 digits.

These are in the correct order because was sorted & stable sort left sorted

Loop Invariant

- The keys have been correctly stable-sorted with respect to the *i-1* least-significant digits.

Running Time

RADIX-SORT(A, d)

for $i \leftarrow 1$ **to** d

  do use a stable sort to sort array A on digit i

Running time is $\Theta(d(n+k))$

Where

$d =$ # of digits in each number

$n =$ # of elements to be sorted

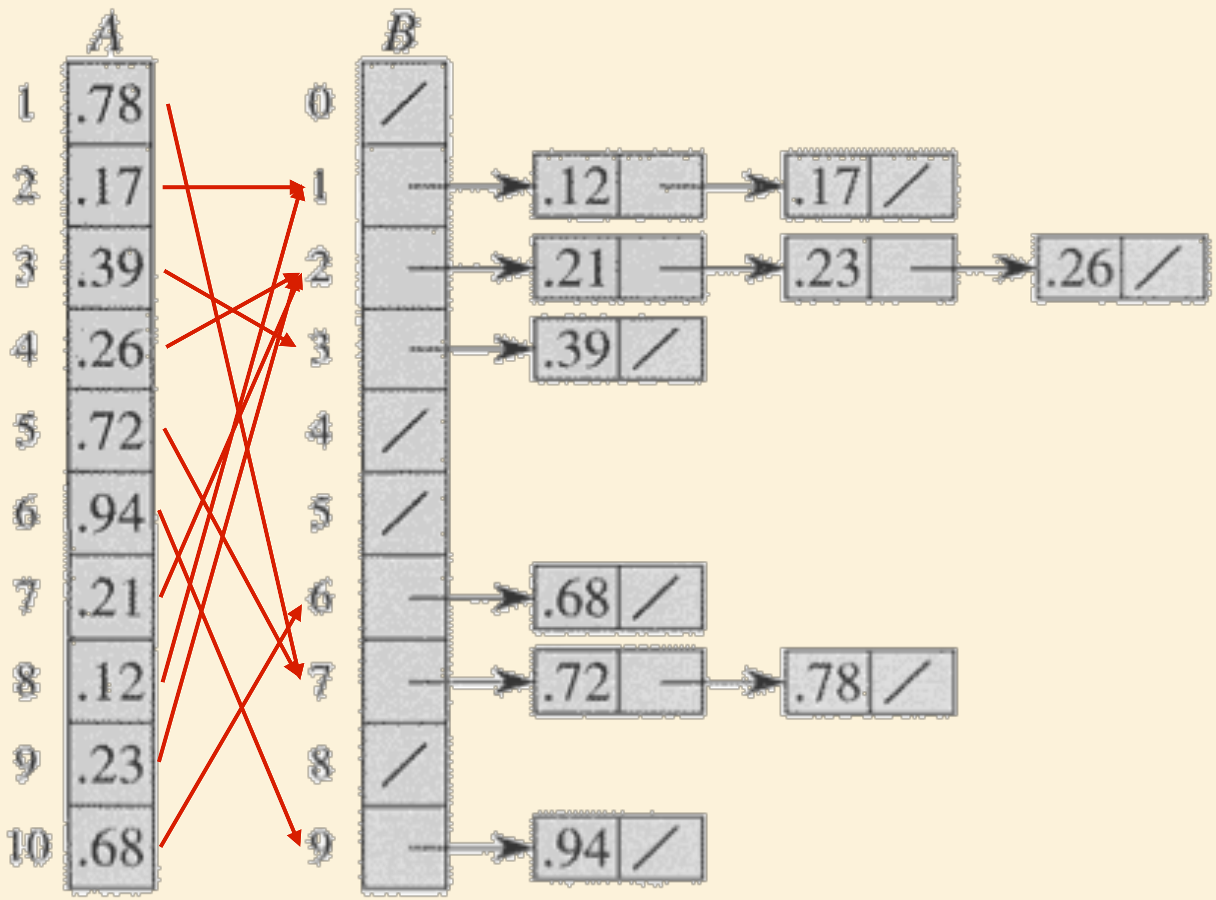
$k =$ # of possible values for each digit

Example 3. Bucket Sort

- Applicable if input is constrained to finite interval, e.g., real numbers in the range [0…1).
- If input is random and uniformly distributed, **expected** run time is $\Theta(n)$.

Bucket Sort

insert $A[i]$ into list $B[\lfloor n \cdot A[i] \rfloor]$



Loop Invariants

- Loop 1
  - The first *i-1* keys have been correctly placed into buckets of width *1/n*.
- Loop 2
  - The keys within each of the first *i-1* buckets have been correctly stable-sorted.

PseudoCode

BUCKET-SORT(A, n)

Expected Running Time

for $i \leftarrow 1$ **to** n

do insert $A[i]$ into list $B[\lfloor n \cdot A[i] \rfloor]$ $\longleftarrow \Theta(1) \times n$

for $i \leftarrow 0$ **to** $n - 1$

do sort list $B[i]$ with insertion sort $\longleftarrow \Theta(1) \times n$

concatenate lists $B[0], B[1], \ldots, B[n-1]$ $\longleftarrow \Theta(n)$

return the concatenated lists

$\Theta(n)$

Sorting Algorithms

- Comparison Sorting
  - Selection Sort
  - Bubble Sort
  - Insertion Sort
  - Merge Sort
  - Heap Sort
  - Quick Sort
- Linear Sorting
  - Counting Sort
  - Radix Sort
  - Bucket Sort

Sorting: Learning Outcomes

- You should be able to:
  - Explain the difference between comparison sorts and linear sorting methods
  - Identify situations when linear sorting methods can be applied and know why
  - Select a sorting method that is well-suited for a specific application.
  - Explain what is meant by sorting in place and stable sorting
  - State a tight bound on the problem of comparison sorting, and explain why no algorithm can do better.
  - Explain and/or code any of the sorting algorithms we have covered, and state their asymptotic run times.