Recursive Sorts

Recursive Sorts

- Recursive sorts divide the data roughly in half and are called again on the smaller data sets.
- This is called the Divide-and-Conquer paradigm.
- We will see 2 recursive sorts:
  - Merge-Sort
  - Quick-Sort

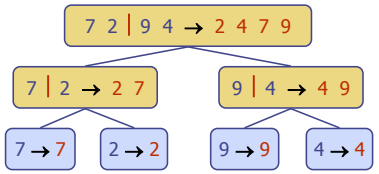
Divide-and-Conquer

- Divide-and-conquer paradigm:
  - Divide: divide one large problem into 2 smaller problems of the same type.
  - Recur: solve the 2 sub-problems.
  - Conquer: combine the 2 solutions into a solution to the larger problem.

Divide-and-Conquer

- Merge-sort is a sorting algorithm based on the divide-and-conquer paradigm
- Like heap-sort
  - It uses a comparator
  - 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



Outline and Reading

- Divide-and-conquer paradigm (§10.1.1)
- Merge-sort (§10.1)
  - Algorithm
  - Merging two sorted sequences
  - Merge-sort tree
  - Execution example
  - Analysis
- Generic merging and set operations (§10.1.2)

Merge-Sort

Merge-sort on an input sequence S with n elements consists of three steps:

- Divide: partition into 2 groups of about n/2 each
- Recur: recursively sort S_1 and S_2
- Conquer: merge S_1 and S_2 into a unique sorted sequence

Merging Two Sorted Sequences

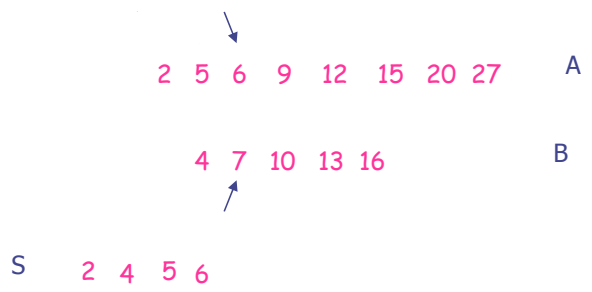
- The conquer step merges the 2 sorted sequences A and B into one sorted sequence S
- ***How: Compare the lowest element of each of A and B and insert whichever is smaller.***
- Merging two sorted sequences, each with $n/2$ elements and implemented by means of a doubly linked list, takes $O(n)$ time

Merge-Sort

```
Algorithm mergeSort(S)
    Input sequence S with n elements,
    Output sequence S sorted
    if S.size() > 1
        (S1, S2) ← partition(S, n/2)
        mergeSort(S1)
        mergeSort(S2)
        S ← merge(S1, S2)
```


Merging Two Sorted Sequences

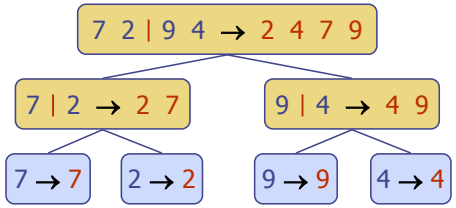
```
Algorithm merge(A, B)
    Input sorted sequences A and B
    Output sorted sequence of A ∪ B

    S ← empty sequence
    while !A.isEmpty() ∧ !B.isEmpty()
        if isLessThan(A.first().element(), B.first().element())
            S.insertLast(A.remove(A.first()))
        else
            S.insertLast(B.remove(B.first()))
    while !A.isEmpty()
            S.insertLast(A.remove(A.first()))
    while !B.isEmpty()
            S.insertLast(B.remove(B.first()))
    return S
```

Merge-Sort Tree

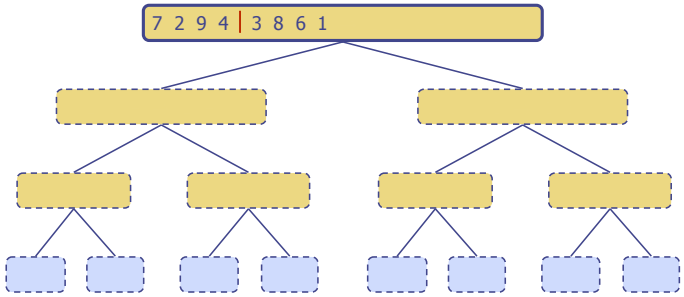
An execution of merge-sort is depicted by a binary tree

- 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 children are calls on subsequences
- the leaves are calls on sequences of size 0 or 1



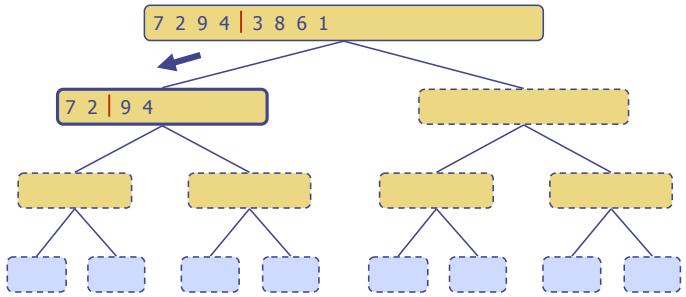
Execution Example

- Partition



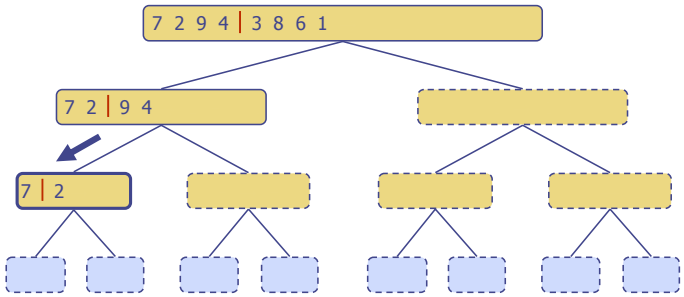
Execution Example (cont.)

- Recursive call, partition



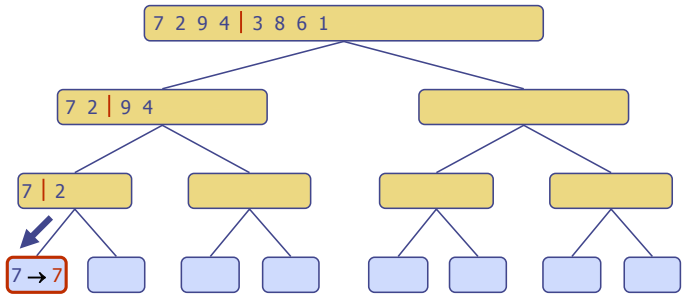
Execution Example (cont.)

- Recursive call, partition



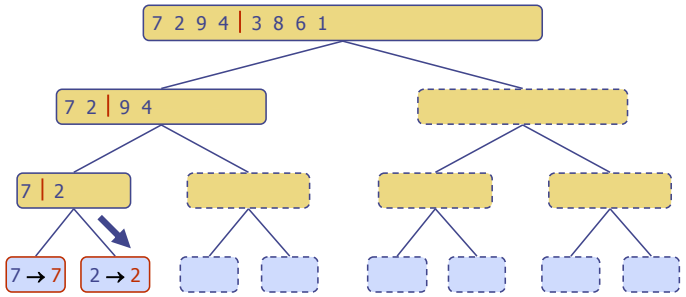
Execution Example (cont.)

- Recursive call, base case



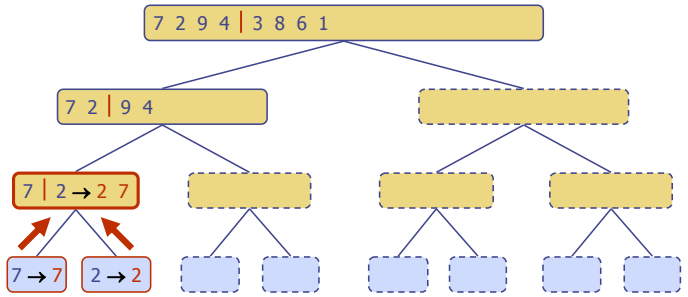
Execution Example (cont.)

- Recursive call, base case



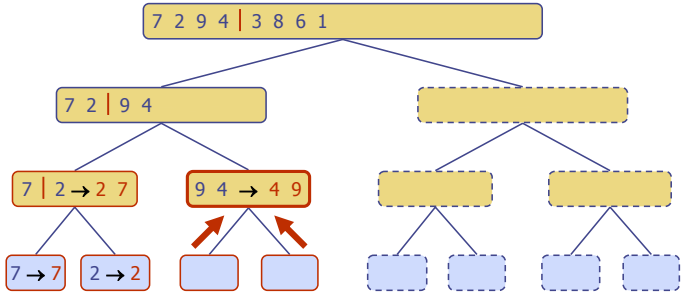
Execution Example (cont.)

- Merge



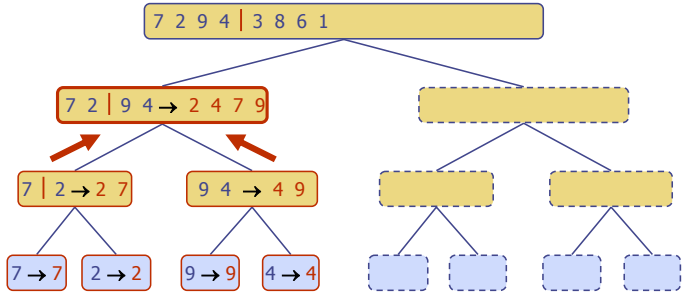
Execution Example (cont.)

- Recursive call, …, base case, merge



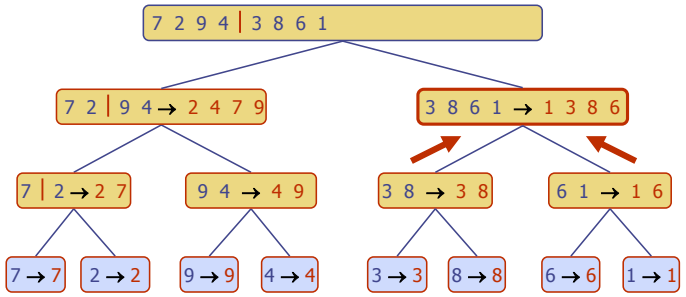
Execution Example (cont.)

- Merge



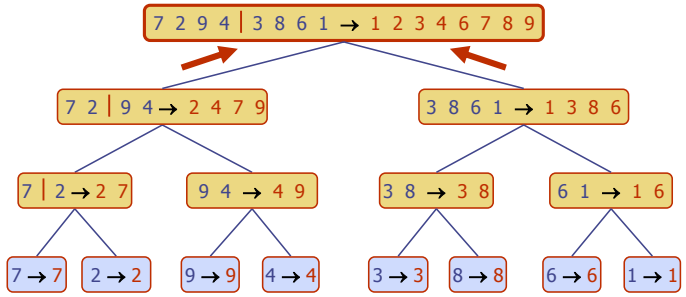
Execution Example (cont.)

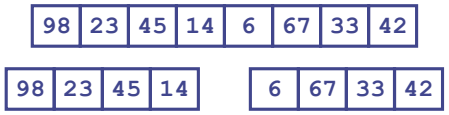
- Recursive call, …, merge, merge

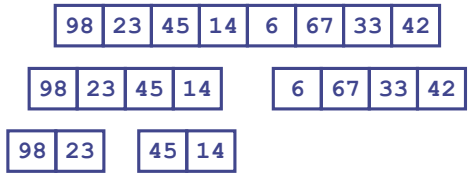


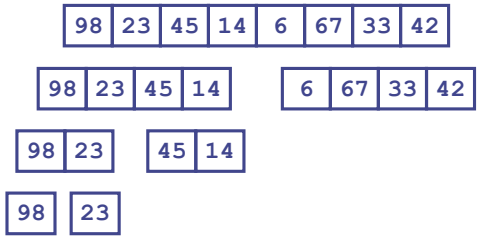
Execution Example (cont.)

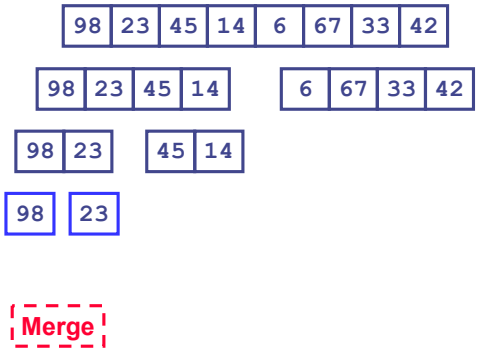
- Merge

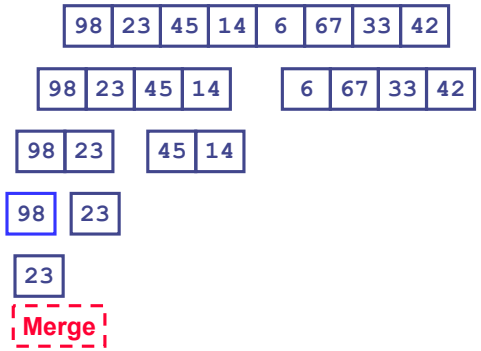


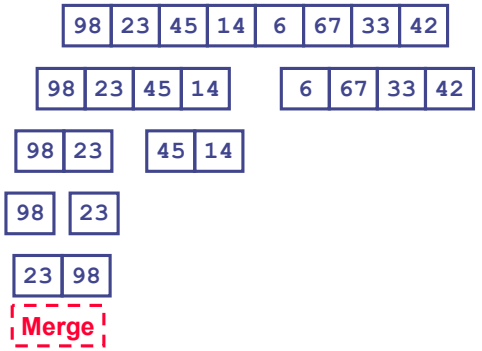


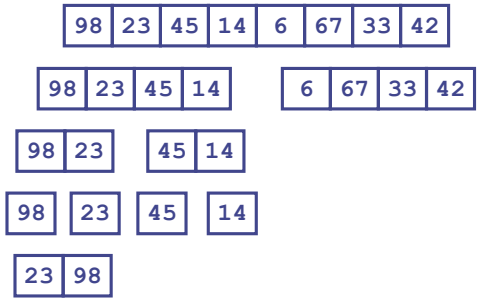

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14

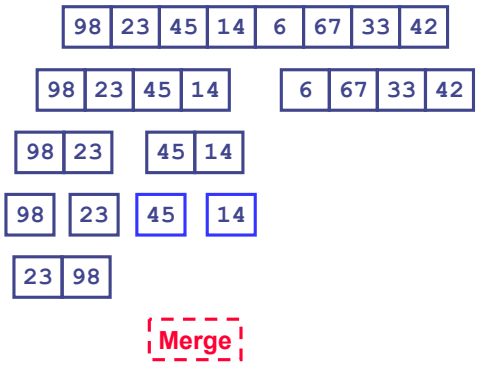

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23

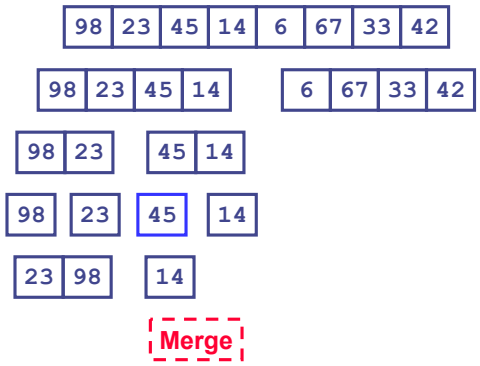

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
Merge

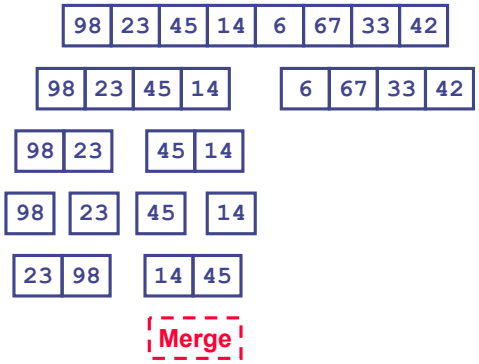

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
23
Merge

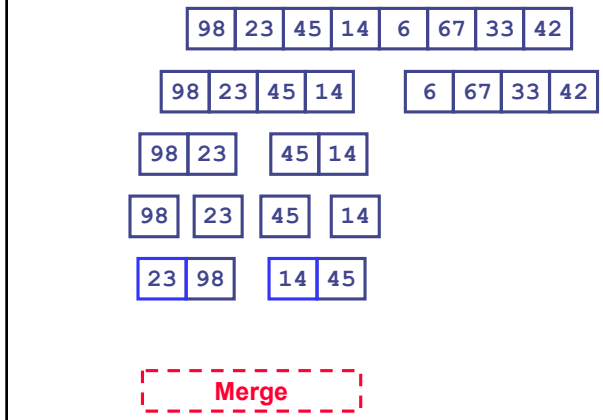

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
23 98
Merge

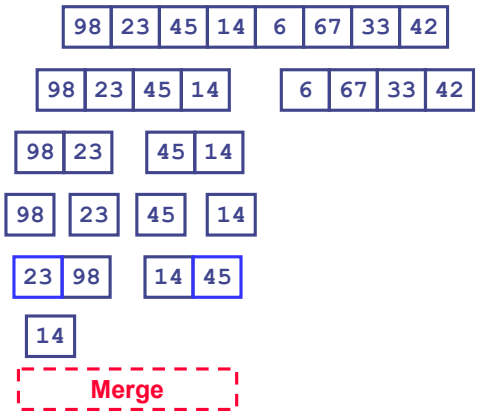

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98

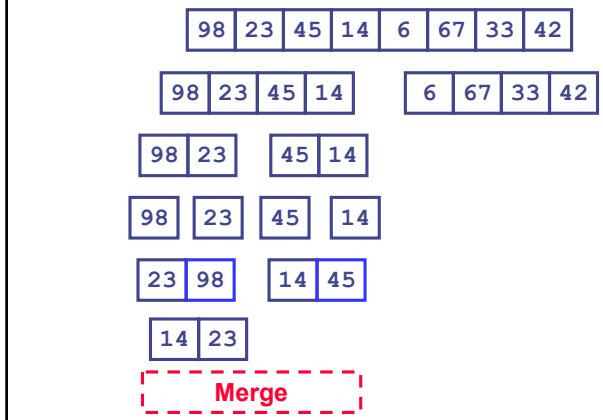

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
Merge

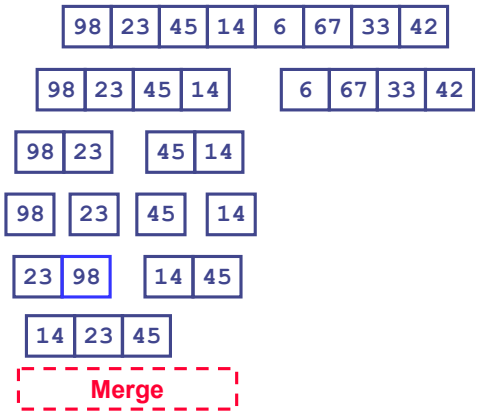

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14
Merge

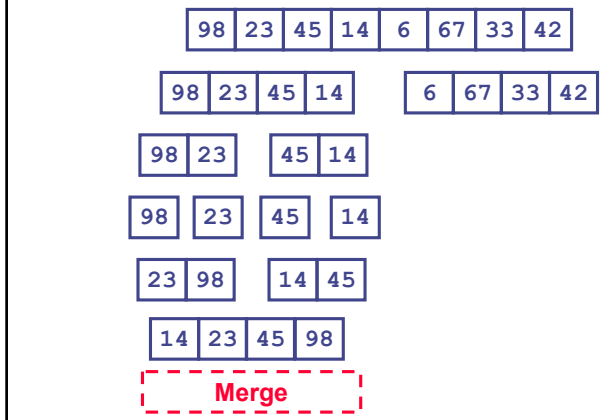

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98 23
45 14
23 98
14 45
Merge

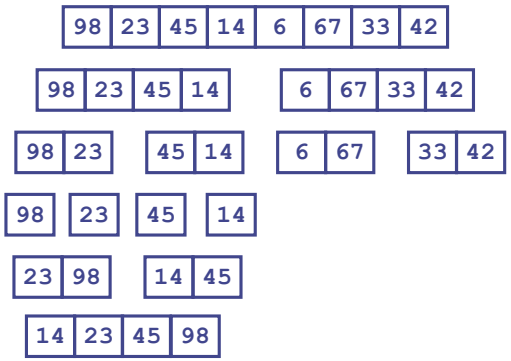

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98 23
45 14
23 98
14 45
Merge

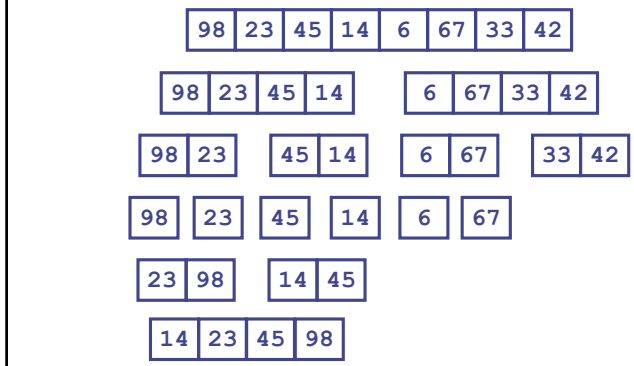

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98 23
45 14
23 98
14 45
14
Merge

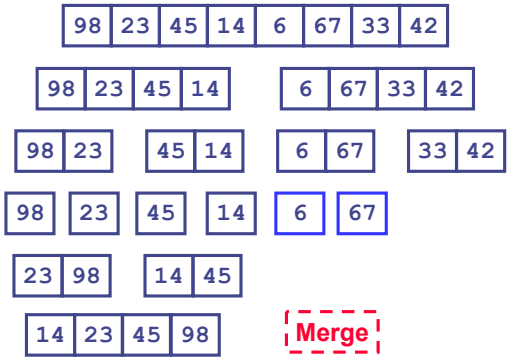

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98 23
45 14
23 98
14 45
14 23
Merge

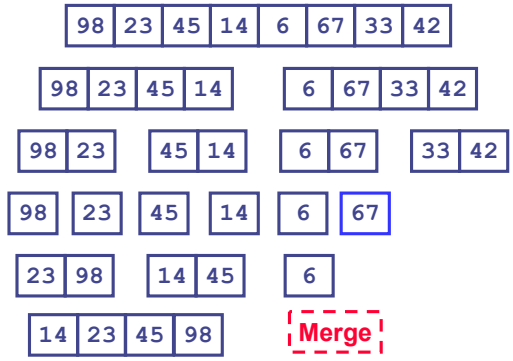

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14 45
14 23 45
Merge

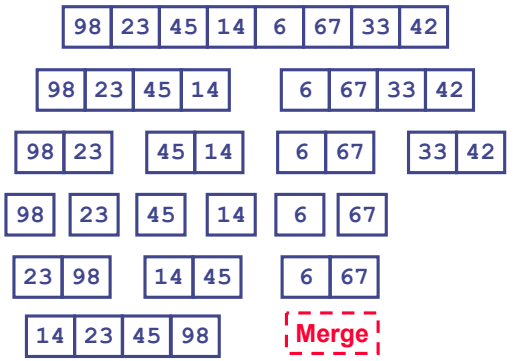

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
98
23
45
14
23 98
14 45
14 23 45 98
Merge

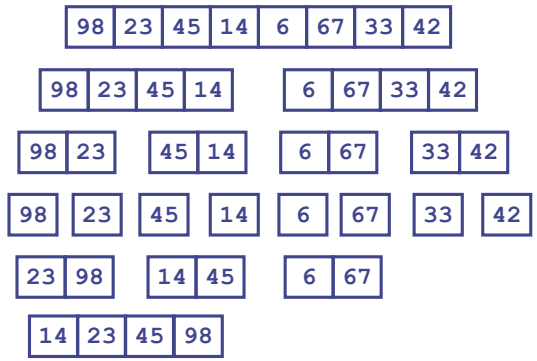

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
23 98
14 45
14 23 45 98

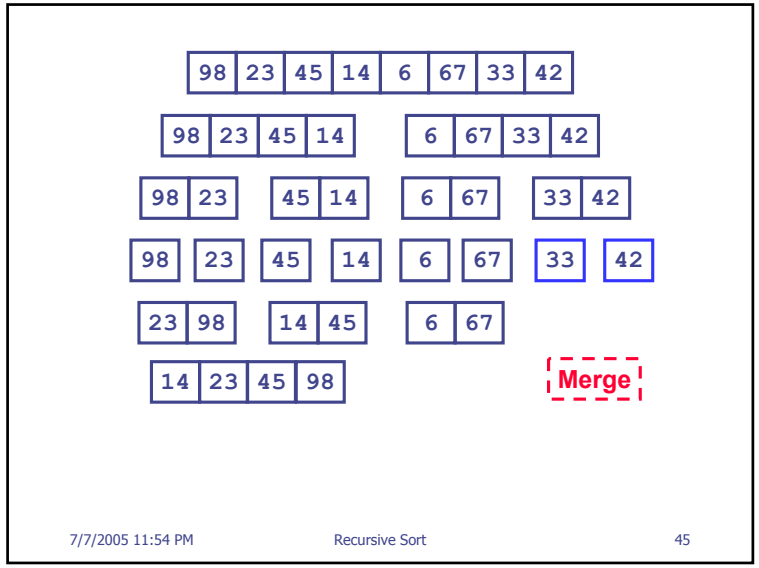

98 23 45 14 6 67 33 42
98 23 45 14
6 67 33 42
98 23
45 14
6 67
33 42
98
23
45
14
6
67
23 98
14 45
14 23 45 98

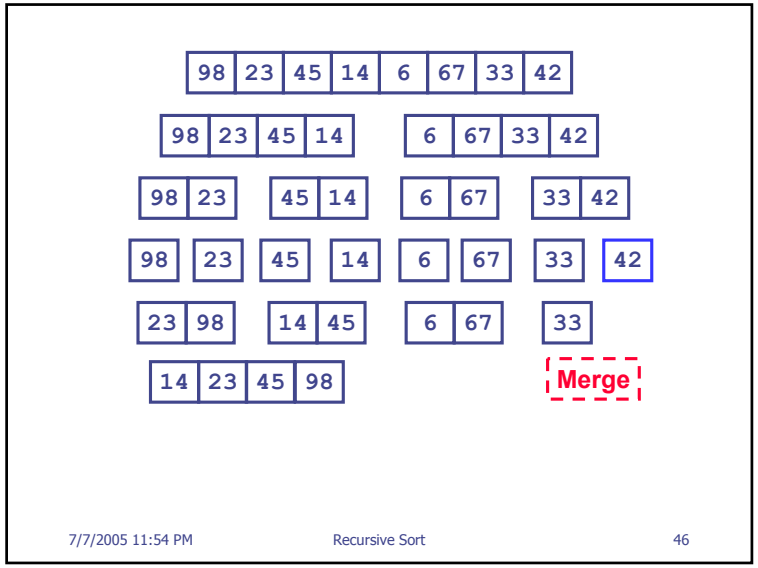

98 23 45 14 6 67 33 42
98 23 45 14 6 67 33 42
98 23 45 14 6 67 33 42
98 23 45 14 6 67
23 98 14 45
14 23 45 98
Merge

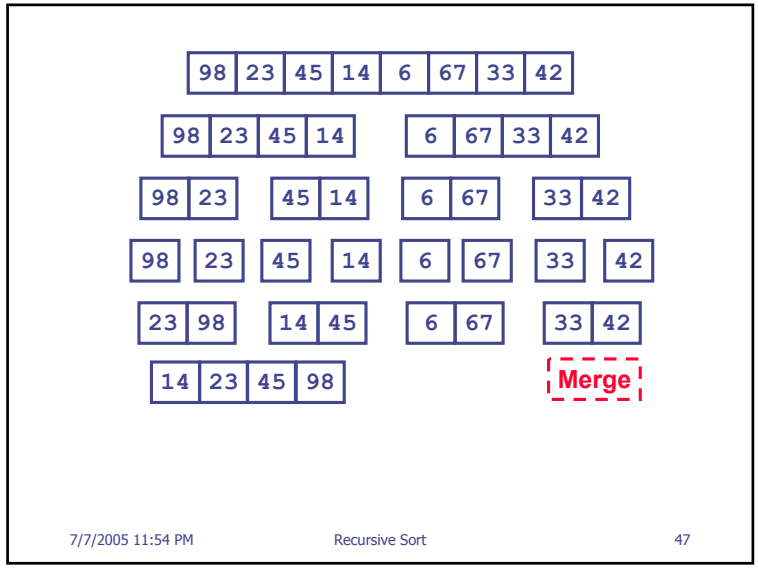

98 23 45 14 6 67 33 42
98 23 45 14 6 67 33 42
98 23 45 14 6 67 33 42
98 23 45 14 6 67
23 98 14 45 6
14 23 45 98
Merge

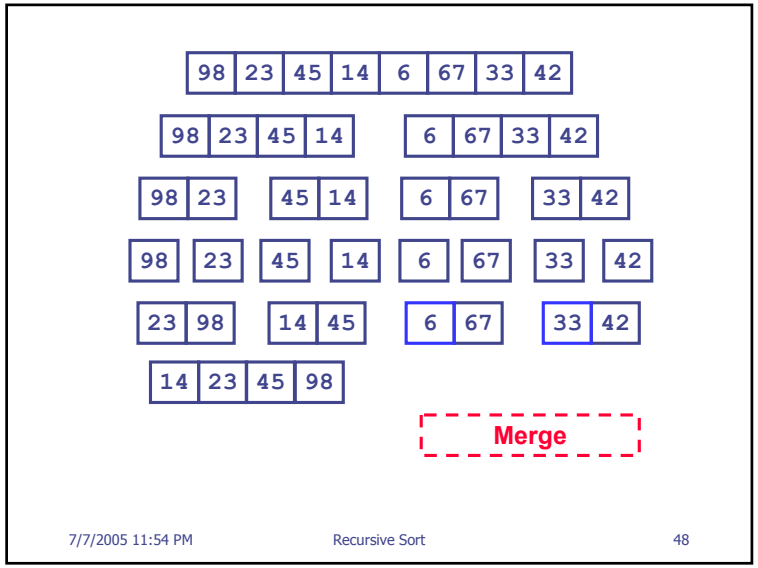

98 23 45 14 6 67 33 42
98 23 45 14 6 67 33 42
98 23 45 14 6 67 33 42
98 23 45 14 6 67
23 98 14 45 6 67
14 23 45 98
Merge

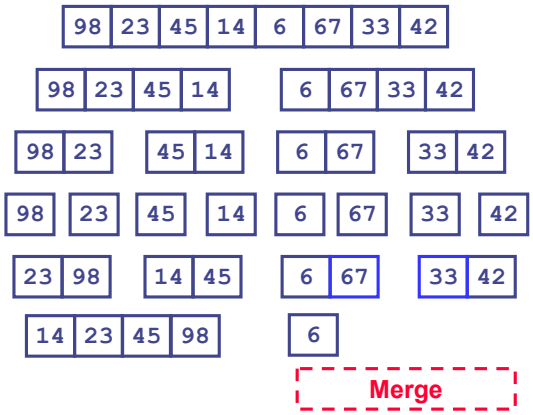

98 23 45 14 6 67 33 42
98 23 45 14 6 67 33 42
98 23 45 14 6 67 33 42
98 23 45 14 6 67 33 42
23 98 14 45 6 67
14 23 45 98

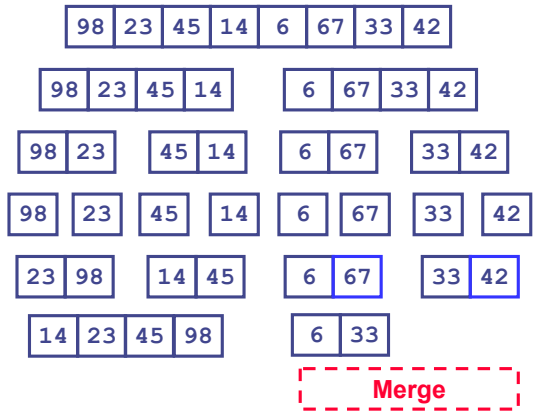

98 23 45 14 6 67 33 42
98 23 45 14 6 67 33 42
98 23 45 14 6 67 33 42
98 23 45 14 6 67 33 42
98 23 45 14 6 67 33 42
23 98 14 45 6 67
14 23 45 98
Merge

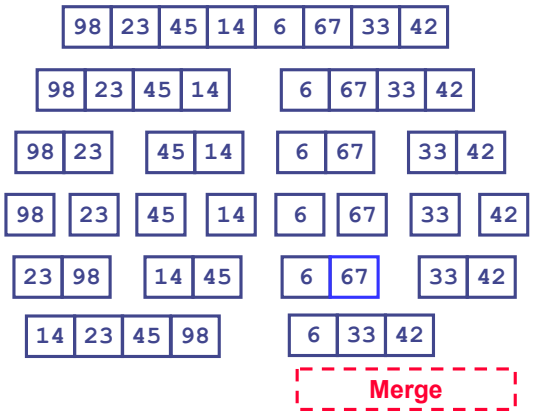

98 23 45 14 6 67 33 42
98 23 45 14 6 67 33 42
98 23 45 14 6 67 33 42
98 23 45 14 6 67 33 42
23 98 14 45 6 67 33
14 23 45 98
Merge

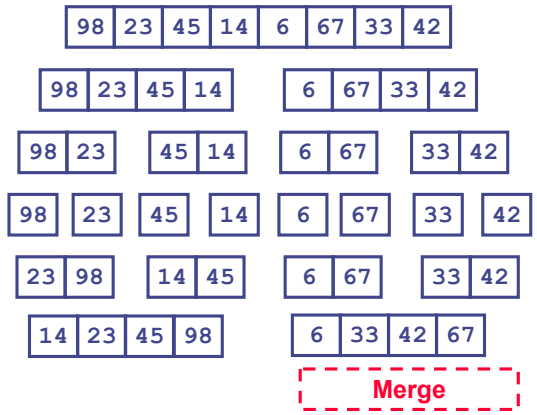

98 23 45 14 6 67 33 42
98 23 45 14 6 67 33 42
98 23 45 14 6 67 33 42
98 23 45 14 6 67 33 42
23 98 14 45 6 67 33 42
14 23 45 98
Merge

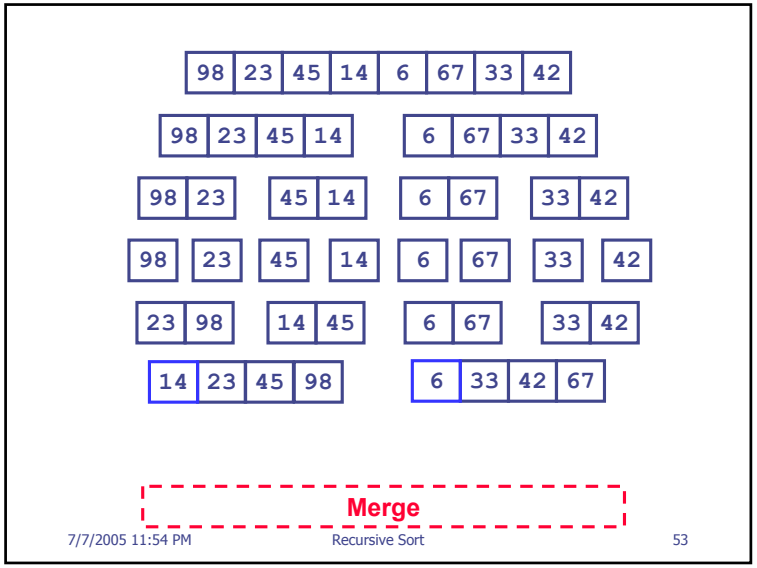

98 23 45 14 6 67 33 42
98 23 45 14 6 67 33 42
98 23 45 14 6 67 33 42
98 23 45 14 6 67 33 42
23 98 14 45 6 67 33 42
14 23 45 98
Merge

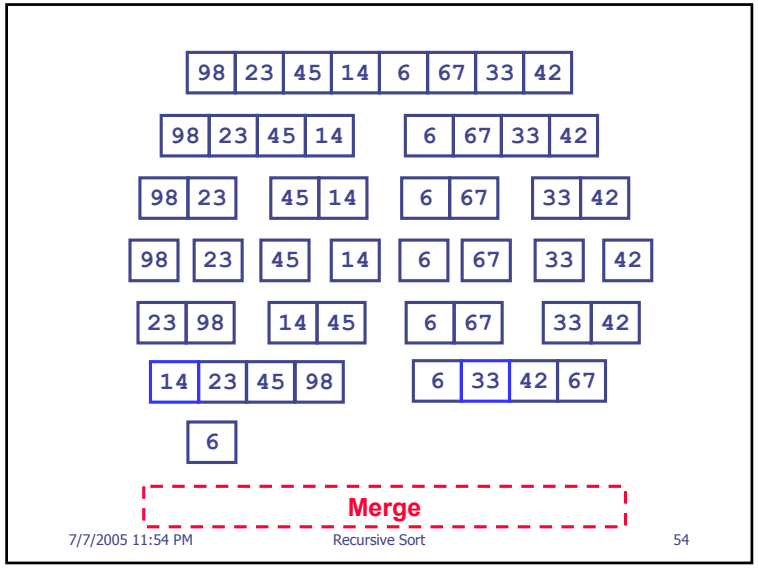

98 23 45 14 6 67 33 42
98 23 45 14 6 67 33 42
98 23 45 14 6 67 33 42
98 23 45 14 6 67 33 42
23 98 14 45 6 67 33 42
14 23 45 98 6
Merge

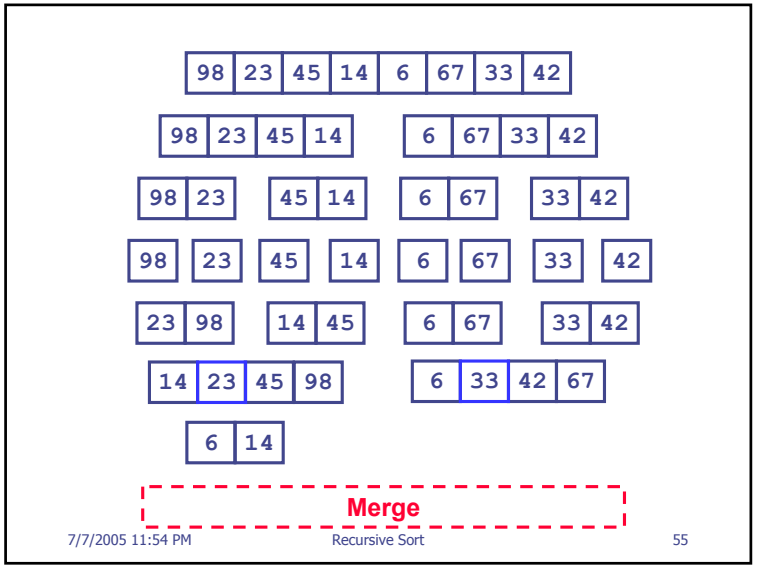

98 23 45 14 6 67 33 42
98 23 45 14 6 67 33 42
98 23 45 14 6 67 33 42
98 23 45 14 6 67 33 42
23 98 14 45 6 67 33 42
14 23 45 98 6 33
Merge

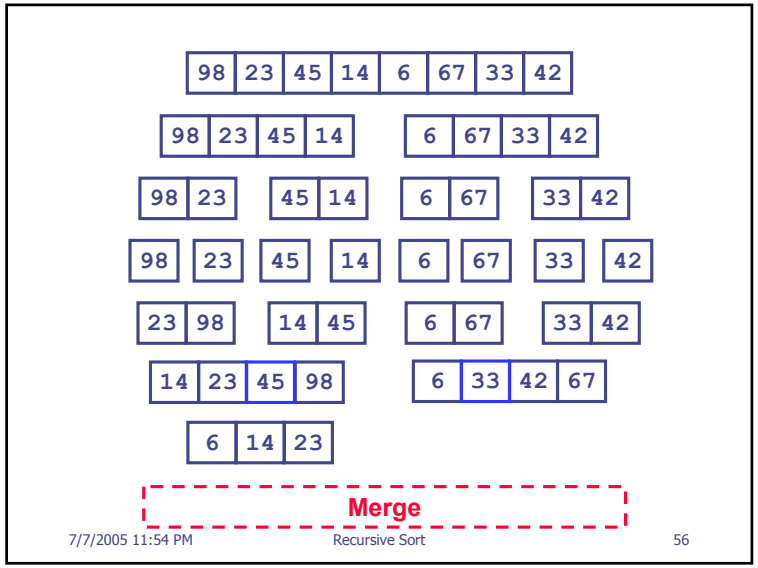

98 23 45 14 6 67 33 42
98 23 45 14 6 67 33 42
98 23 45 14 6 67 33 42
98 23 45 14 6 67 33 42
23 98 14 45 6 67 33 42
14 23 45 98 6 33 42
Merge

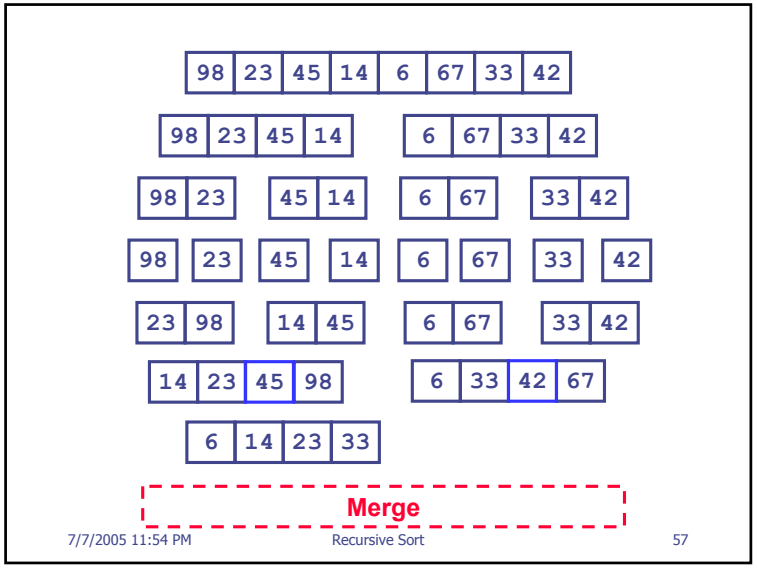

98 23 45 14 6 67 33 42
98 23 45 14 6 67 33 42
98 23 45 14 6 67 33 42
98 23 45 14 6 67 33 42
23 98 14 45 6 67 33 42
14 23 45 98 6 33 42 67
Merge

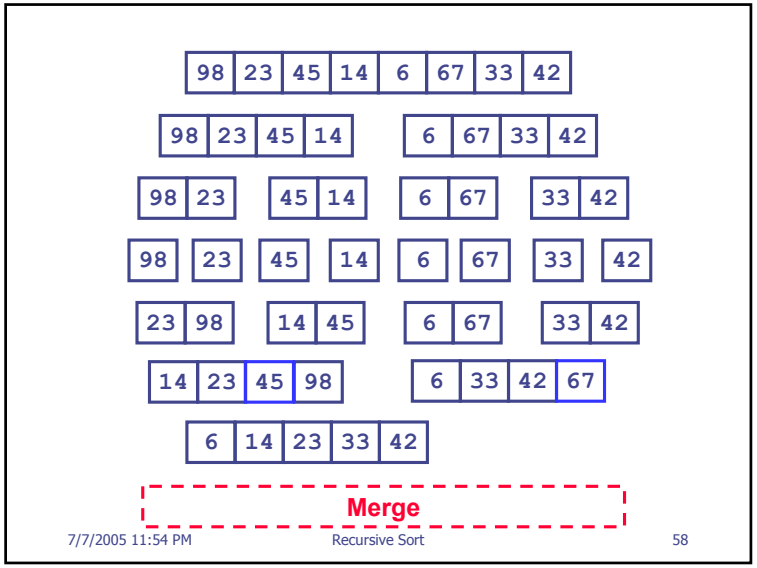

98 23 45 14 6 67 33 42
98 23 45 14 6 67 33 42
98 23 45 14 6 67 33 42
98 23 45 14 6 67 33 42
23 98 14 45 6 67 33 42
14 23 45 98 6 33 42 67
Merge

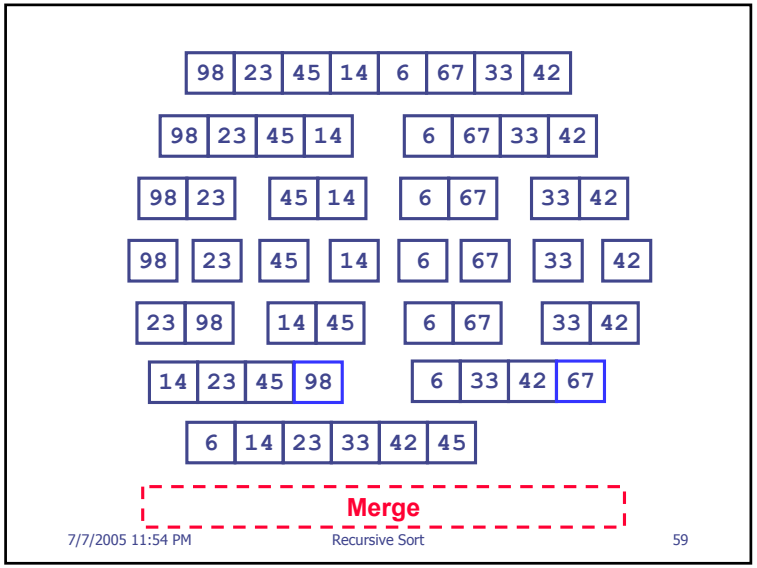

98 23 45 14 6 67 33 42
98 23 45 14 6 67 33 42
98 23 45 14 6 67 33 42
98 23 45 14 6 67 33 42
23 98 14 45 6 67 33 42
14 23 45 98 6 33 42 67
6
Merge

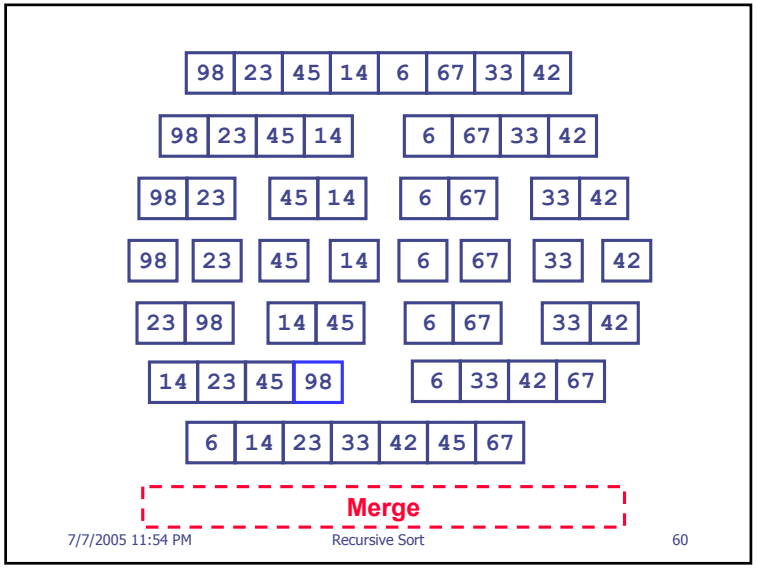

98 23 45 14 6 67 33 42
98 23 45 14 6 67 33 42
98 23 45 14 6 67 33 42
98 23 45 14 6 67 33 42
23 98 14 45 6 67 33 42
14 23 45 98 6 33 42 67
6 14
Merge

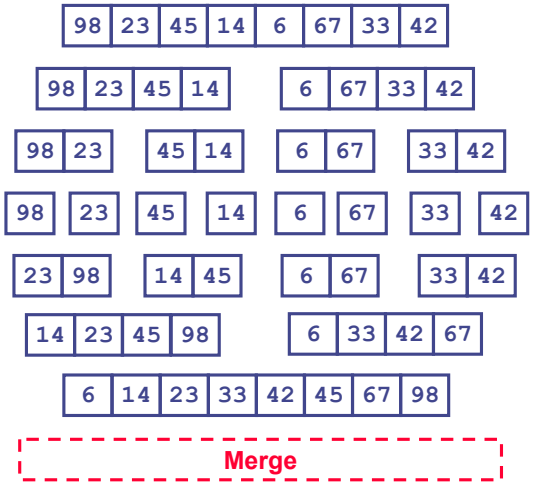

98 23 45 14 6 67 33 42
98 23 45 14 6 67 33 42
98 23 45 14 6 67 33 42
98 23 45 14 6 67 33 42
23 98 14 45 6 67 33 42
14 23 45 98 6 33 42 67
6 14 23
Merge

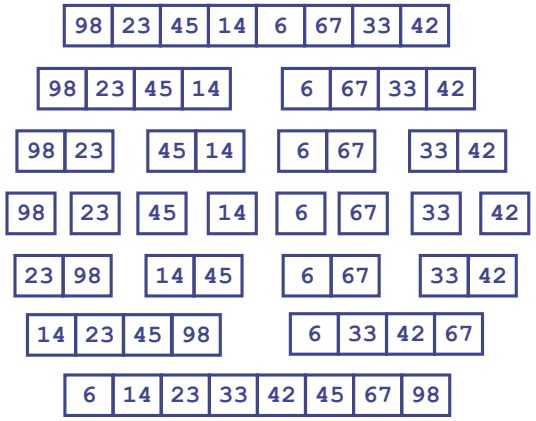

98 23 45 14 6 67 33 42
98 23 45 14 6 67 33 42
98 23 45 14 6 67 33 42
98 23 45 14 6 67 33 42
23 98 14 45 6 67 33 42
14 23 45 98 6 33 42 67
6 14 23 33
Merge

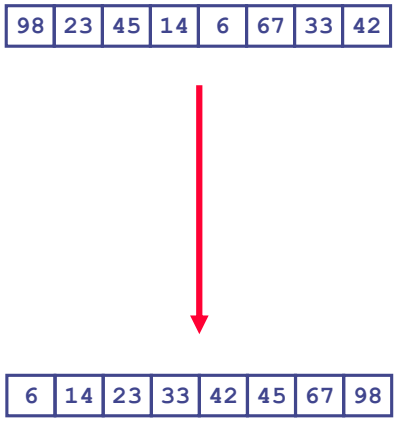

98 23 45 14 6 67 33 42
98 23 45 14 6 67 33 42
98 23 45 14 6 67 33 42
98 23 45 14 6 67 33 42
23 98 14 45 6 67 33 42
14 23 45 98 6 33 42 67
6 14 23 33 42
Merge


98 23 45 14 6 67 33 42
98 23 45 14 6 67 33 42
98 23 45 14 6 67 33 42
98 23 45 14 6 67 33 42
23 98 14 45 6 67 33 42
14 23 45 98 6 33 42 67
6 14 23 33 42 45
Merge


98 23 45 14 6 67 33 42
98 23 45 14 6 67 33 42
98 23 45 14 6 67 33 42
98 23 45 14 6 67 33 42
23 98 14 45 6 67 33 42
14 23 45 98 6 33 42 67
6 14 23 33 42 45 67
Merge



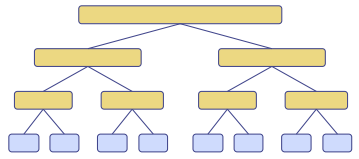




Analysis of Merge-Sort

- 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 or 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	$n/2^i$
…	…	…



Generic Merging

- Generalized merge of two sorted sequences A and B
- Template method genericMerge
- Auxiliary methods
  - aIsLess
  - bIsLess
  - bothAreEqual
- Runs in $O(n_A + n_B)$ time provided the auxiliary methods run in $O(1)$ time

Generic Merging

```
Algorithm genericMerge(A, B)
    S ← empty sequence
    while !A.isEmpty() ∧ !B.isEmpty()
        a ← A.first().element(); b ← B.first().element()
        if a < b
            aIsLess(a, S); A.remove(A.first())
        else if b < a
            bIsLess(b, S); B.remove(B.first())
        else { b = a }
            bothAreEqual(a, b, S)
            A.remove(A.first()); B.remove(B.first())
    while !A.isEmpty()
        aIsLess(a, S); A.remove(A.first())
    while !B.isEmpty()
        bIsLess(b, S); B.remove(B.first())
    return S
```

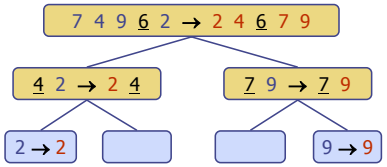

Set Operations

- We represent a set by the sorted sequence of its elements
- By specializing the auxliliary methods, generic merge algorithm can do:
  - union
  - intersection
  - subtraction
- The running time of an operation on sets A and B is $O(n_A + n_B)$

Set Operations Examples

- Set union:
  - *aIsLess(a, S)*
    S.insertLast(a)
  - *bIsLess(b, S)*
    S.insertLast(b)
  - *bothAreEqual(a, b, S)*
    S. insertLast(a)
- Set intersection:
  - *aIsLess(a, S)*
    { do nothing }
  - *bIsLess(b, S)*
    { do nothing }
  - *bothAreEqual(a, b, S)*
    S. insertLast(a)

Quick-Sort

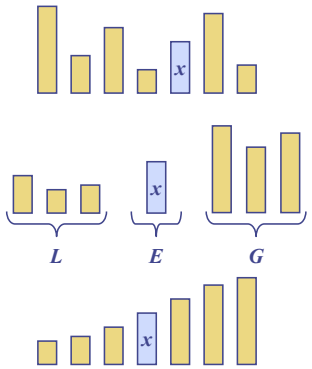


Outline and Reading

- Quick-sort (§10.2)
  - Algorithm
  - Partition step
  - Quick-sort tree
  - Execution example
- Analysis of quick-sort
- In-place quick-sort
- Summary of sorting algorithms

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



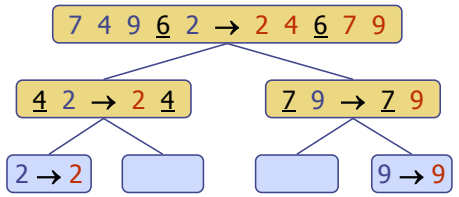
Partition

- 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 $!S.isEmpty()$
    $y \leftarrow S.remove(S.first())$
    if $y < x$
      $L.insertLast(y)$
    else if $y = x$
      $E.insertLast(y)$
    else { $y > x$ }
      $G.insertLast(y)$
  return L, E, G

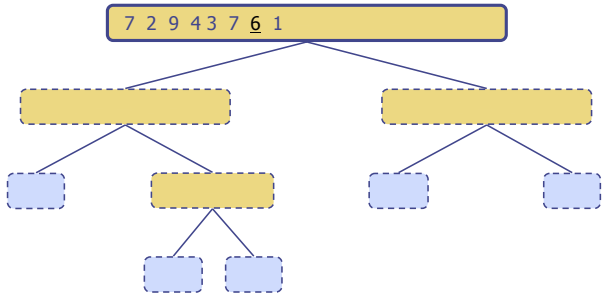
Quick-Sort Tree

- 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



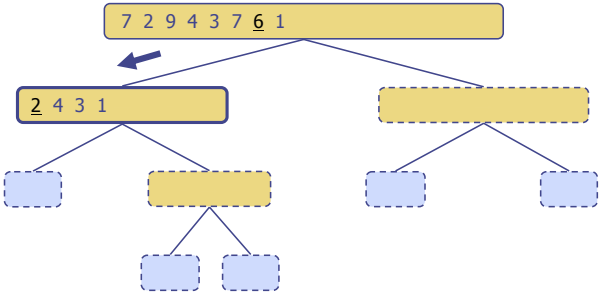
Execution Example

- Pivot selection



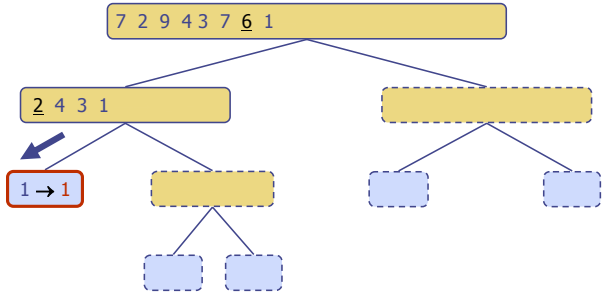
Execution Example (cont.)

- Partition, recursive call, pivot selection



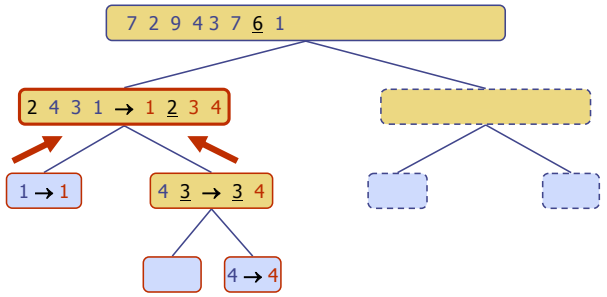
Execution Example (cont.)

- Partition, recursive call, base case



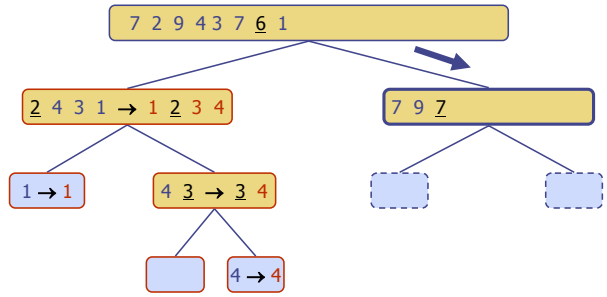
Execution Example (cont.)

- Recursive call, …, base case, join



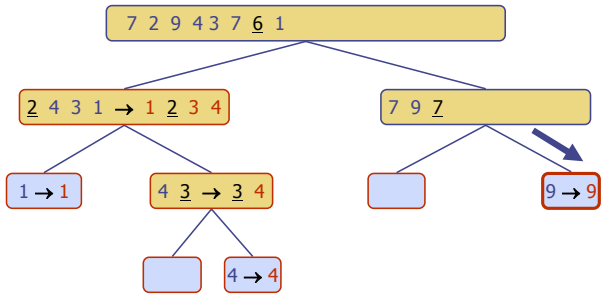
Execution Example (cont.)

- Recursive call, pivot selection



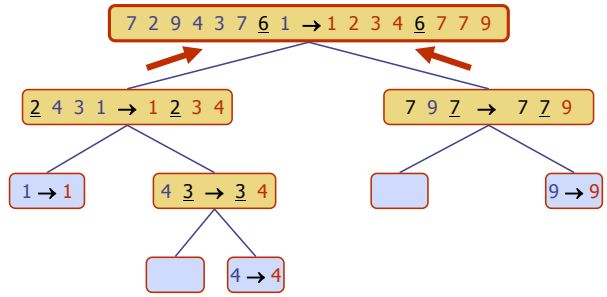
Execution Example (cont.)

- Partition, …, recursive call, base case

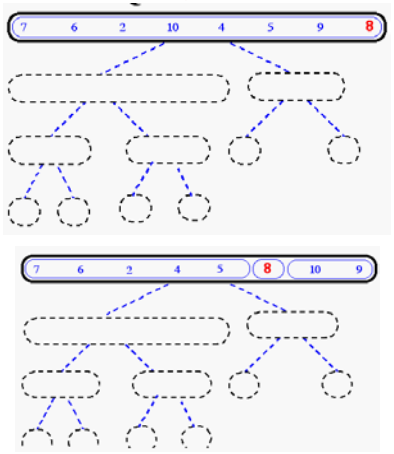


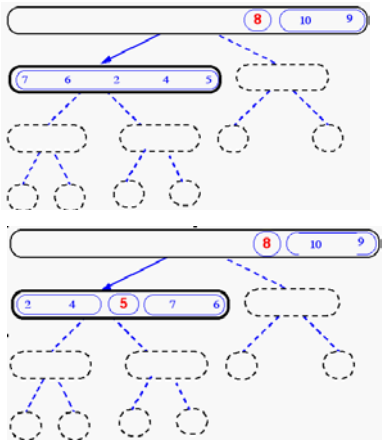
Execution Example (cont.)

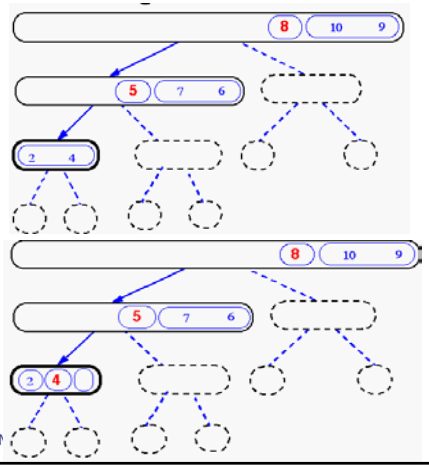
- Join, join

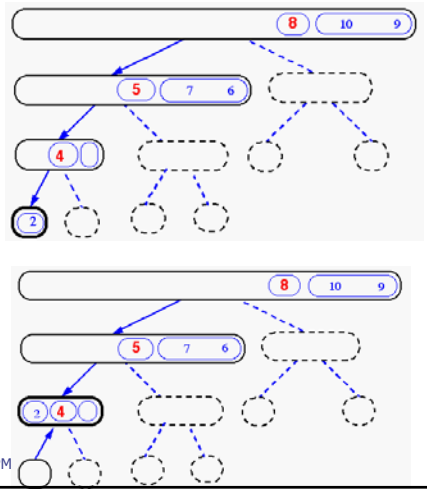


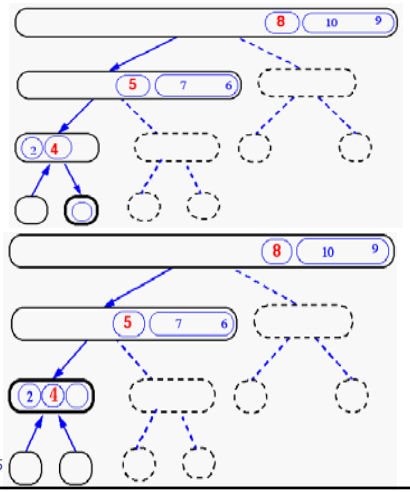
Another example

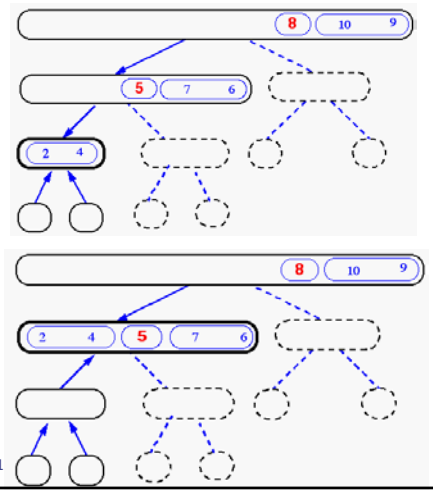


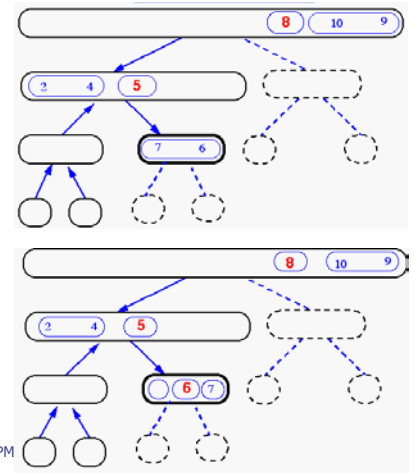


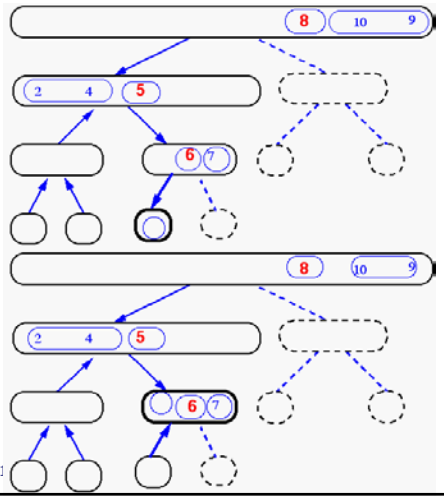


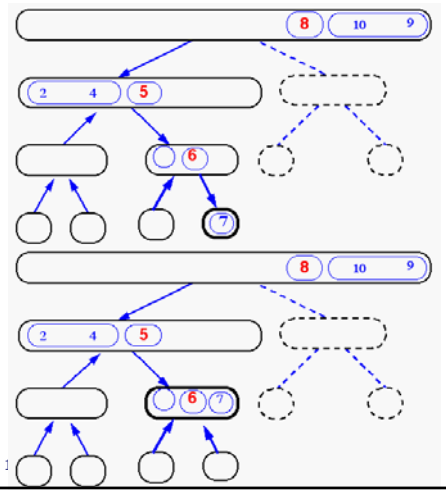


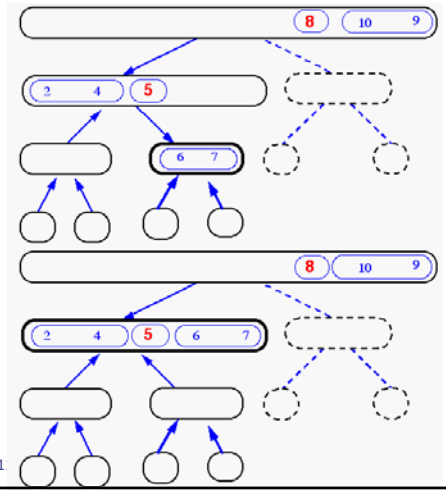

8 10 9
5 7 6
2 4
8 10 9
5 7 6
2 4

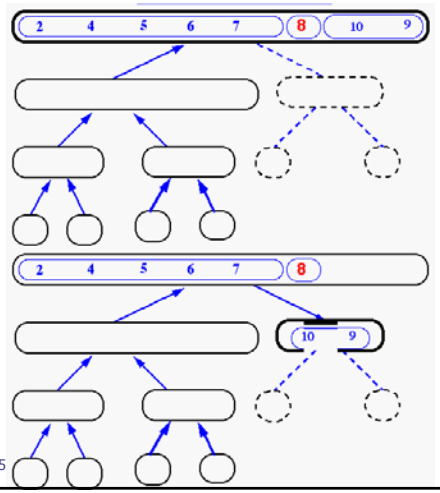

8 10 9
5 7 6
2 4
8 10 9
2 4 5 7 6

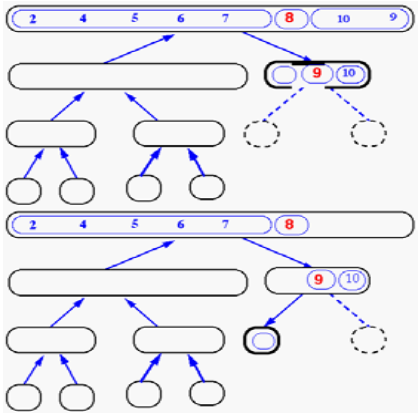

8 10 9
2 4 5
7 6
8 10 9
2 4 5
6 7


8 10 9
2 4 5
6 7
8 10 9
2 4 5
6 7


8 10 9
2 4 5
6
7
8 10 9
2 4 5
6 7

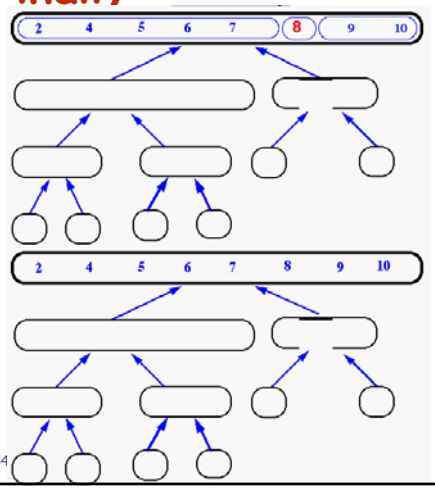

8 10 9
2 4 5
6 7
8 10 9
2 4 5 6 7


2 4 5 6 7 8 10 9
2 4 5 6 7 8
10 9


2 4 5 6 7 8 10 9
9 10
2 4 5 6 7 8
9 10

Skipping ...

… Finally

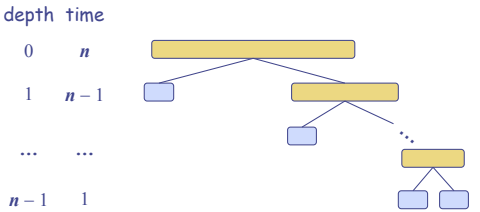


Worst-case Running Time

- 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 $O(n^2)$

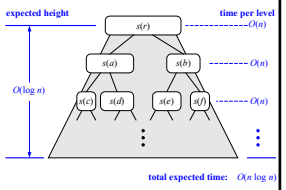


Expected Running Time

Consider a recursive call of quick-sort on a sequence of size s

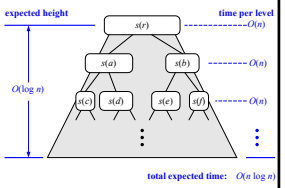
- Good call: the sizes of L and G are each less than $3s/4$
- Bad call: one of L and G has size greater than $3s/4$

- A call is good with probability $1/2$
- Hence, for a node of depth i, we expect that
  - $i/2$ parent nodes are associated with good calls
  - the size of the input sequence for the current call is at most $(3/4)^{i/2}n$



Expected Running Time

- Thus, we have
  - For a node of depth $2\log_{4/3}n$, the expected size of the input sequence is one
  - The expected height of the quick-sort tree is $O(\log n)$
- The overall amount or work done at the nodes of the same depth of the quick-sort tree is $O(n)$
- Thus, the expected running time of quick-sort is $O(n \log n)$



In-Place Quick-Sort

In the partition step, we use replace operations to rearrange the elements of the input sequence such that

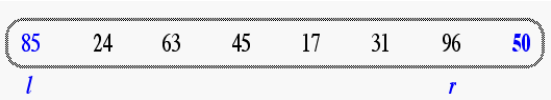
- the elements less than the pivot have rank less than h
- the elements equal to the pivot have rank between h and k
- the elements greater than the pivot have rank greater than k

The recursive calls consider

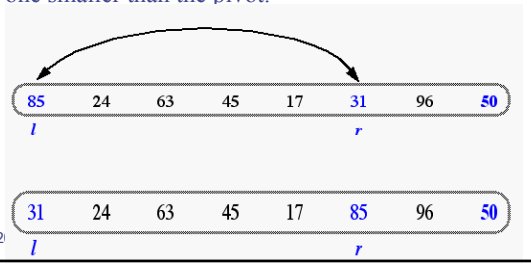
- elements with rank less than h
- elements with rank greater than k

In-Place Quick-Sort

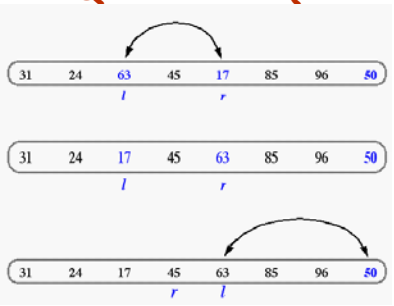
Divide step: l scans the sequence from the left, and r from the right.



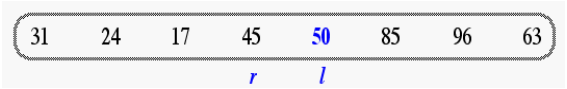
A swap is performed when l is at an element larger than the pivot and r is at one smaller than the pivot.



In Place Quick Sort (contd.)



A final swap with the pivot completes the divide step



In-Place Quick-Sort

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$(S, k + 1, r)$

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)