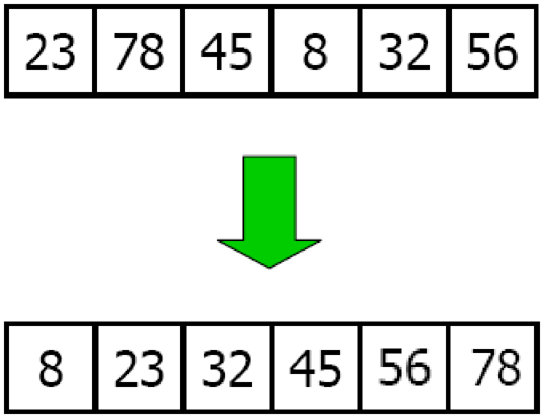
Chapter 10 - Sorting

One of the most important concepts and common applications in computing.



Sorting

- Internal sort: all data are held in primary memory during the sorting process.

- External sort: primary memory for data currently being sorted and secondary storage for data that do not fit in primary memory.

Sorting

Sort stability: data with equal keys maintain their relative input order in the output.

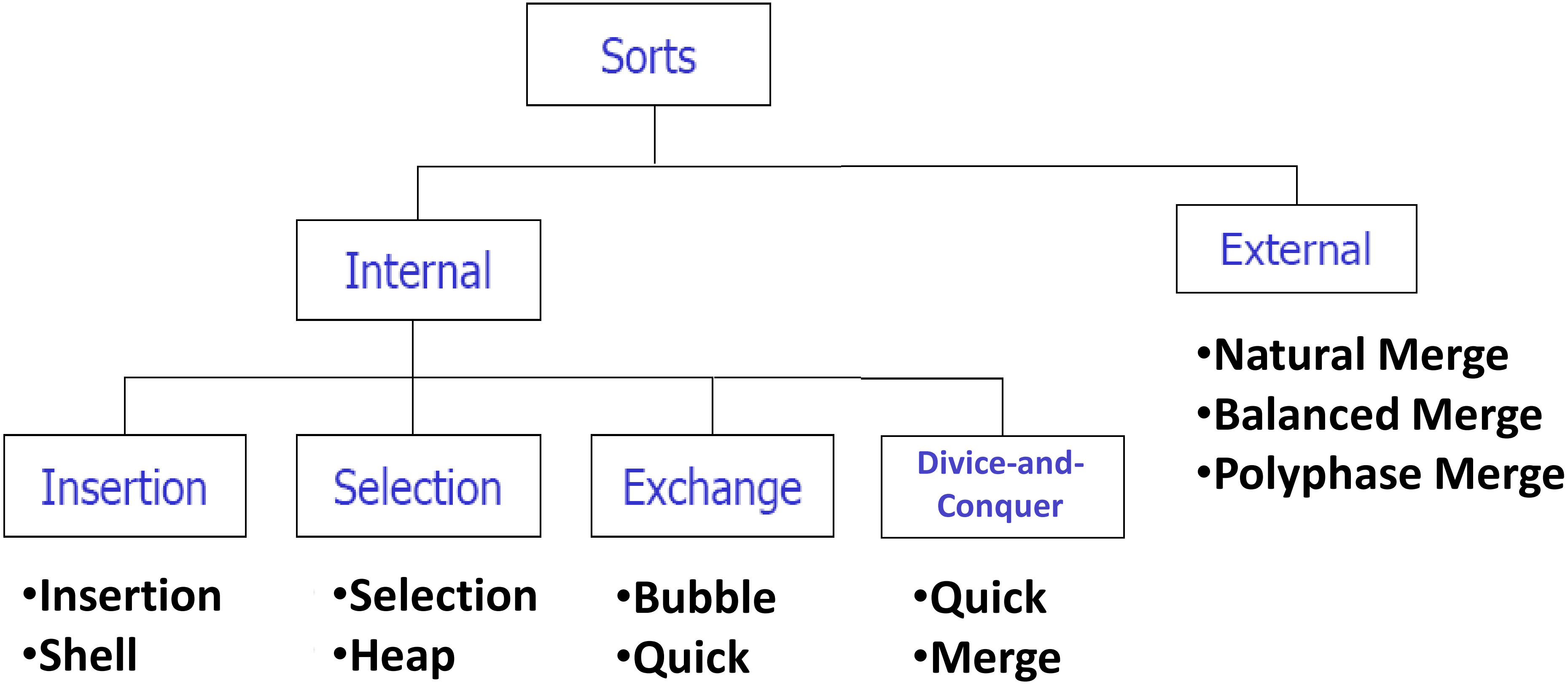
78	8	45	8	32	56

8	8	32	45	56	78

Sorting

- Sort efficiency: a measure of the relative efficiency of a sort = number of comparisons + number of moves

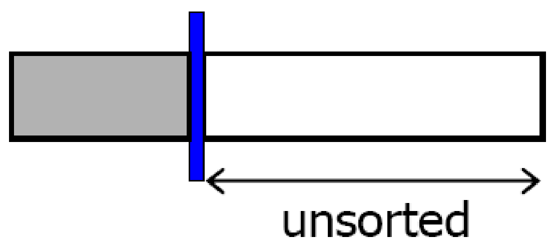
Sorting



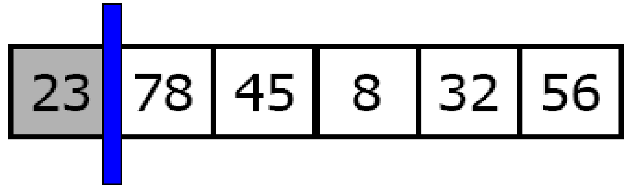
Sorts
Internal
External
•Natural Merge
•Balanced Merge
•Polyphase Merge
Insertion
Selection
Exchange
Divice-and-Conquer
•Insertion
•Shell
•Selection
•Heap
•Bubble
•Quick
•Quick
•Merge

Straight Insertion Sort

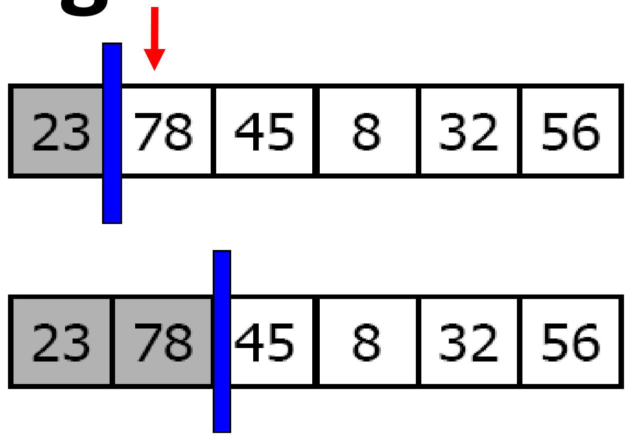
- The list is divided into two parts: sorted and unsorted.
- In each pass, the first element of the unsorted sublist is inserted into the sorted sublist.



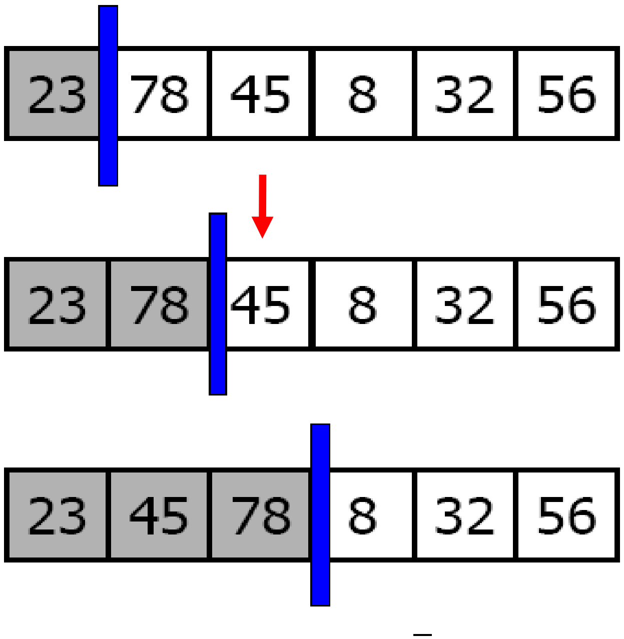
Straight Insertion Sort



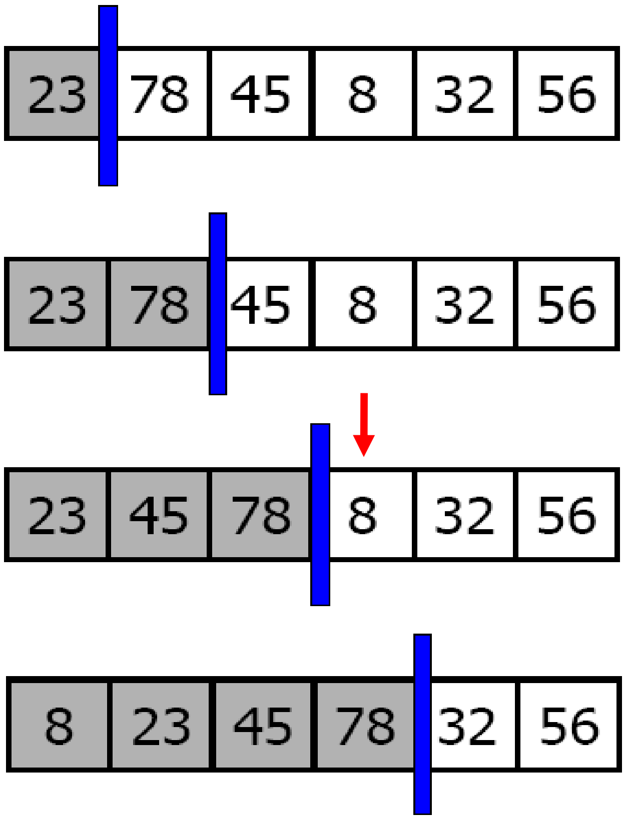
Straight Insertion Sort



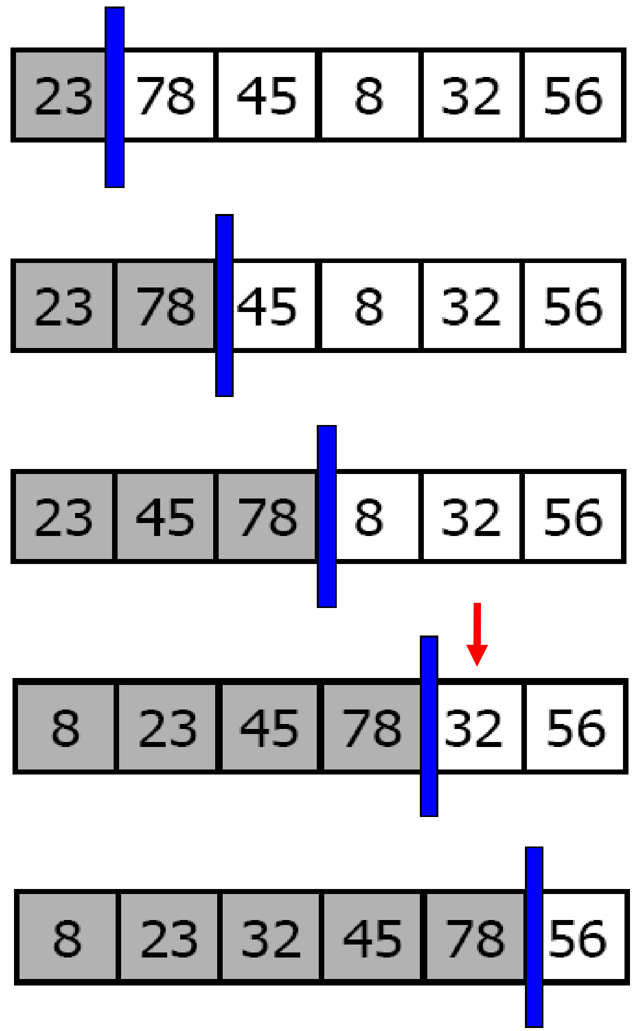
Straight Insertion Sort



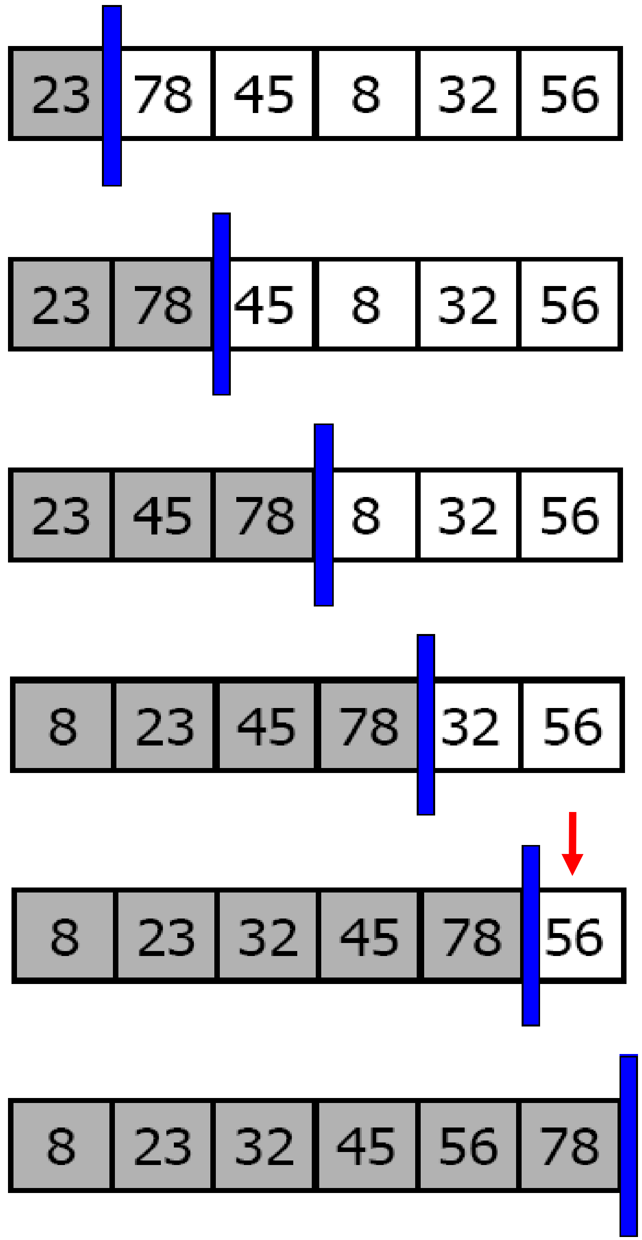
Straight Insertion Sort



Straight Insertion Sort



Straight Insertion Sort



Straight Insertion Sort

Algorithm **InsertionSort** ()

Sorts the contiguous list using straight insertion sort

Post sorted list.

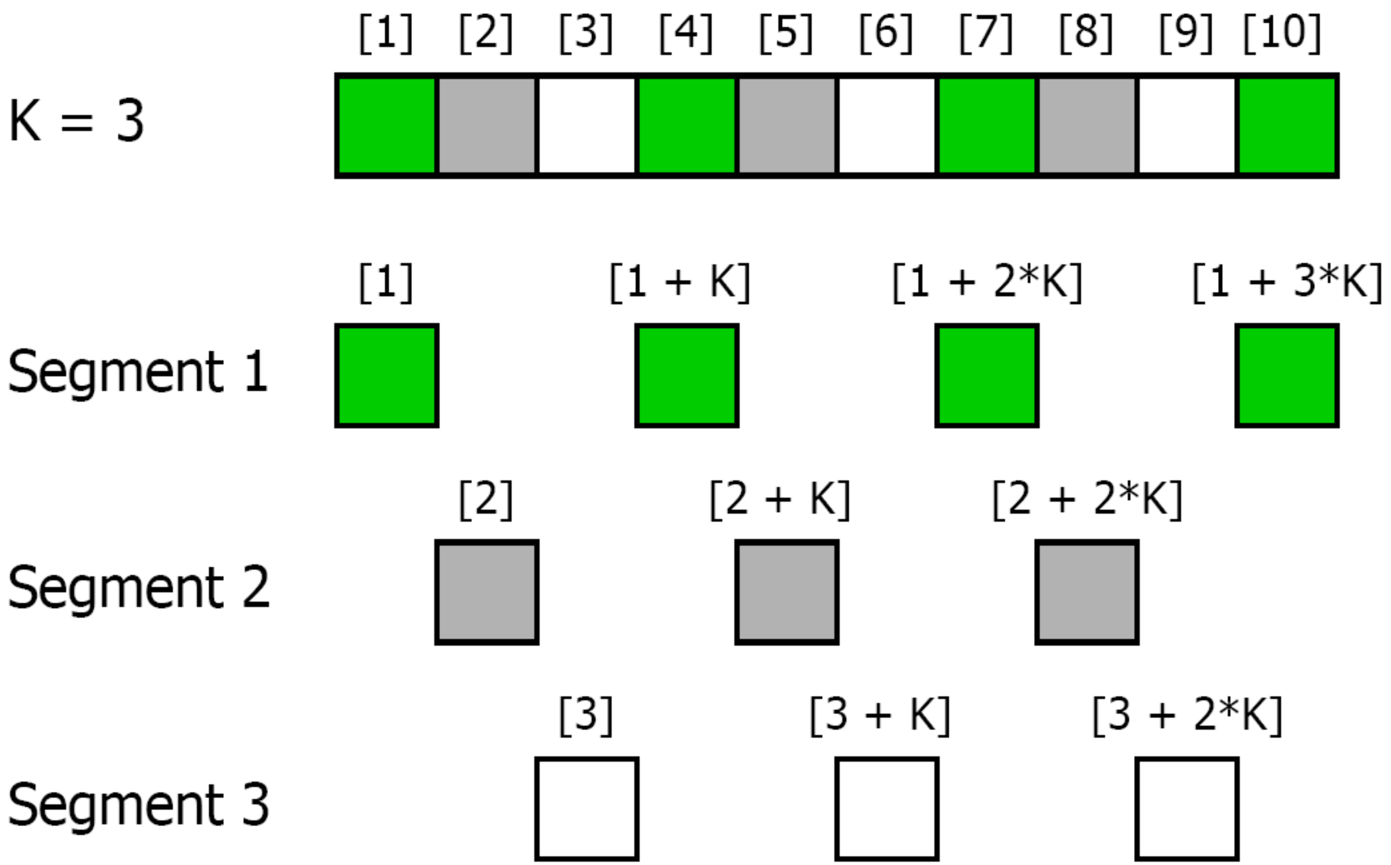
1. **if** (count > 1)
    1. current = 1
    2. **loop** (current < count)
        1. temp = $data_{current}$
        2. walker = current-1
        3. **loop** (walker >=0) AND (temp.key < $data_{walker}$.key)
            1. $data_{walker+1}$ = $data_{walker}$
            2. walker = walker -1
        4. $data_{walker+1}$ = temp
        5. current = current + 1

End InsertionSort

Shell Sort

- Named after its creator Donald L. Shell (1959).
- Given a list of N elements, the list is divided into K segments (K is called the increment).
- Each segment contains N/K or more elements.
- Segments are dispersed throughout the list.
- Also is called diminishing-increment sort

Shell Sort



K = 3
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
[9]
[10]
Segment 1
[1]
[1 + K]
[1 + 2*K]
[1 + 3*K]
Segment 2
[2]
[2 + K]
[2 + 2*K]
Segment 3
[3]
[3 + K]
[3 + 2*K]

Shell Sort

23	78	45	8	32	56

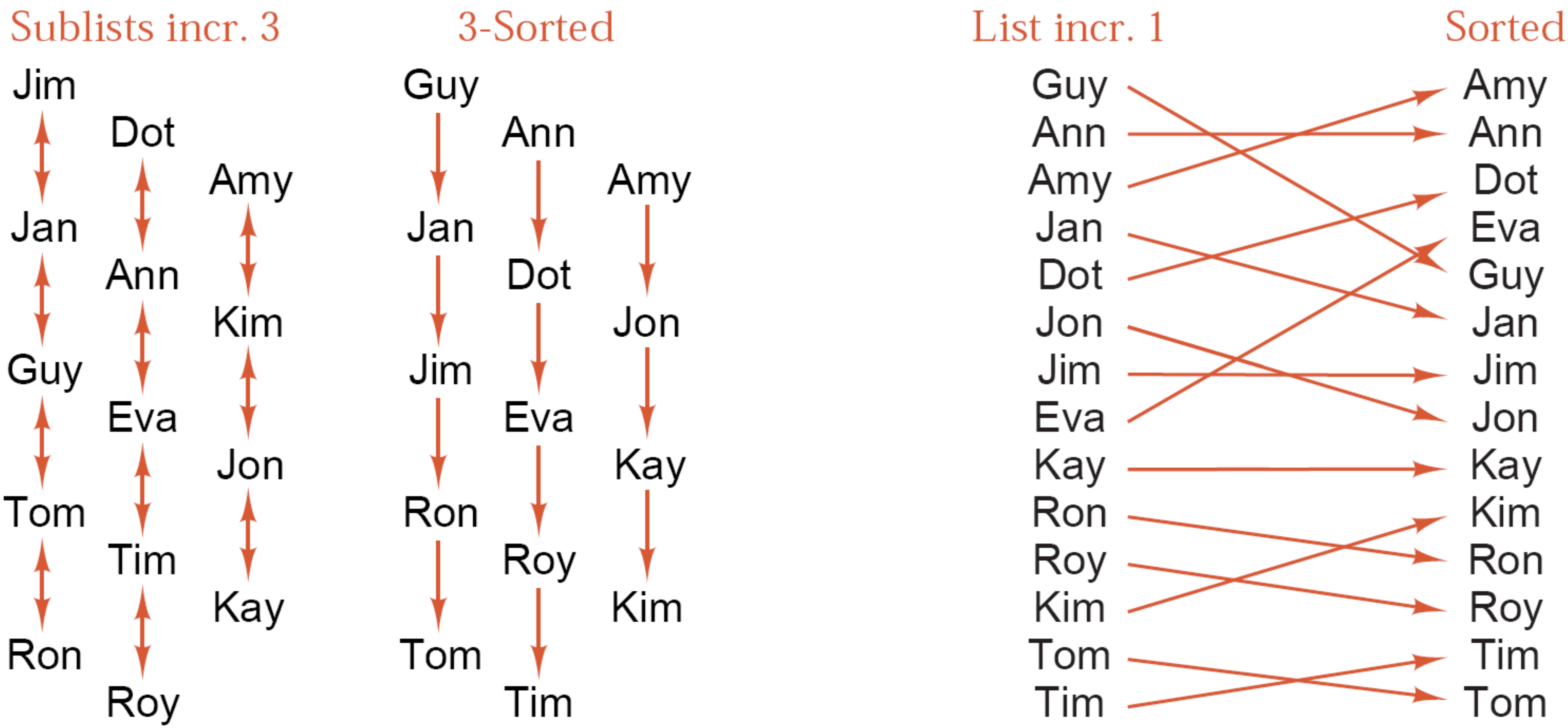
- For the value of K in each iteration, sort the K segments.

- After each iteration, K is reduced until it is 1 in the final iteration.

Example of Shell Sort

Unsorted	Sublists incr. 5					5-Sorted					Recombined
Tim	Tim					Jim					Jim
Dot		Dot					Dot				Dot
Eva			Eva					Amy			Amy
Roy				Roy					Jan		Jan
Tom					Tom					Ann	Ann
Kim	Kim					Kim					Kim
Guy		Guy					Guy				Guy
Amy			Amy					Eva			Eva
Jon				Jon					Jon		Jon
Ann					Ann					Tom	Tom
Jim	Jim					Tim					Tim
Kay		Kay					Kay				Kay
Ron			Ron					Ron			Ron
Jan				Jan					Roy		Roy

Example of Shell Sort



Choosing incremental values

- From more of the comparisons, it is better when we can receive more new information.

- Incremental values should not be multiples of each other, other wise, the same keys compared on one pass would be compared again at the next.

- The final incremental value must be 1.

Choosing incremental values

Incremental values may be:

1, 4, 13, 40, 121, ...

$k_t = 1$

$k_{i-1} = 3 * k_i + 1$

$t = |\log_3(n)| - 1$

or :

1, 3, 7, 15, 31,...

$k_t = 1$

$k_{i-1} = 2 * k_i + 1$

$t = |\log_2(n)| - 1$

Shell Sort

Algorithm **ShellSort** ()

Sorts the contiguous list using Shell sort

Post sorted list.

1. k = first_incremental_value
2. **loop** (k >= 1)
    1. segment = 1
    2. **loop** (segment <= k)
        1. SortSegment(segment)
        2. segment = segment + 1
    3. k = next_incremental_value

End ShellSort

Shell Sort

Algorithm **SortSegment**(val segment <int>, val k <int>)

Sorts the segment beginning at segment using insertion sort, step between elements in the segment is k.

Post sorted segment.

1. current = segment + k
2. **loop** (current < count)
    1. temp = data[current]
    2. walker = current - k
    3. **loop** (walker >=0) AND (temp.key < data[walker].key)
        1. data[walker + k] = data[walker]
        2. walker = walker – k
    4. data[walker + k] = temp
    5. current = current + k

End SortSegment

Insertion Sort Efficiency

- Straight insertion sort:

$$f(n) = n(n + 1)/2 = O(n^2)$$

- Shell sort:

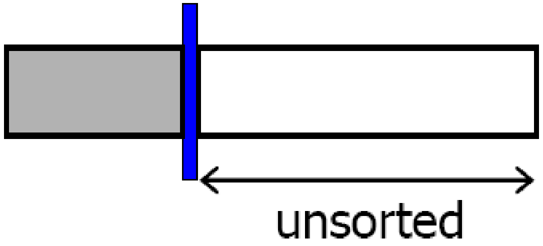
$O(n^{1.25})$ Empirical study

Selection Sort

- In each pass, the smallest/largest item is selected and placed in a sorted list.

Straight Selection Sort

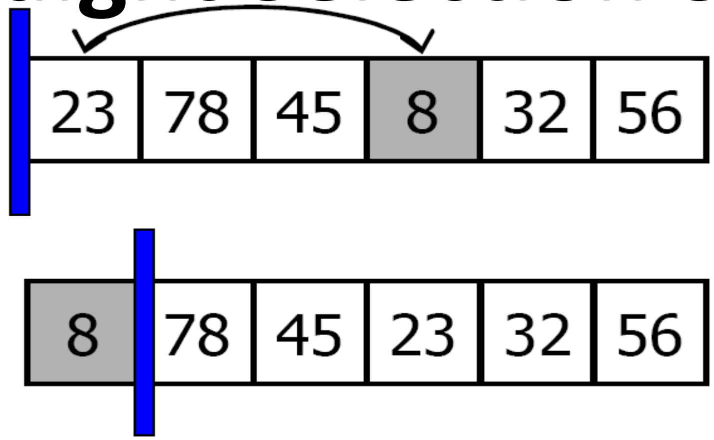
- The list is divided into two parts: sorted and unsorted.
- In each pass, in the unsorted sublist, the smallest element is selected and exchanged with the first element.



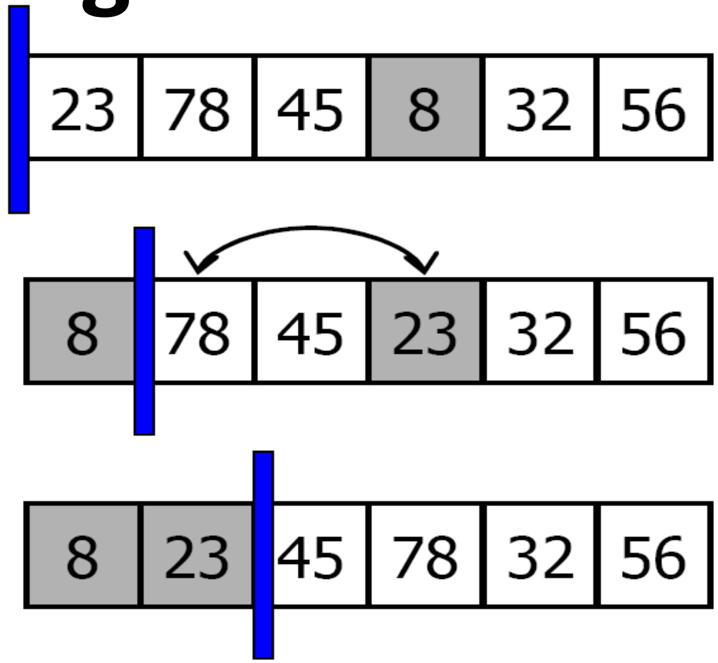
Straight Selection Sort

23	78	45	8	32	56

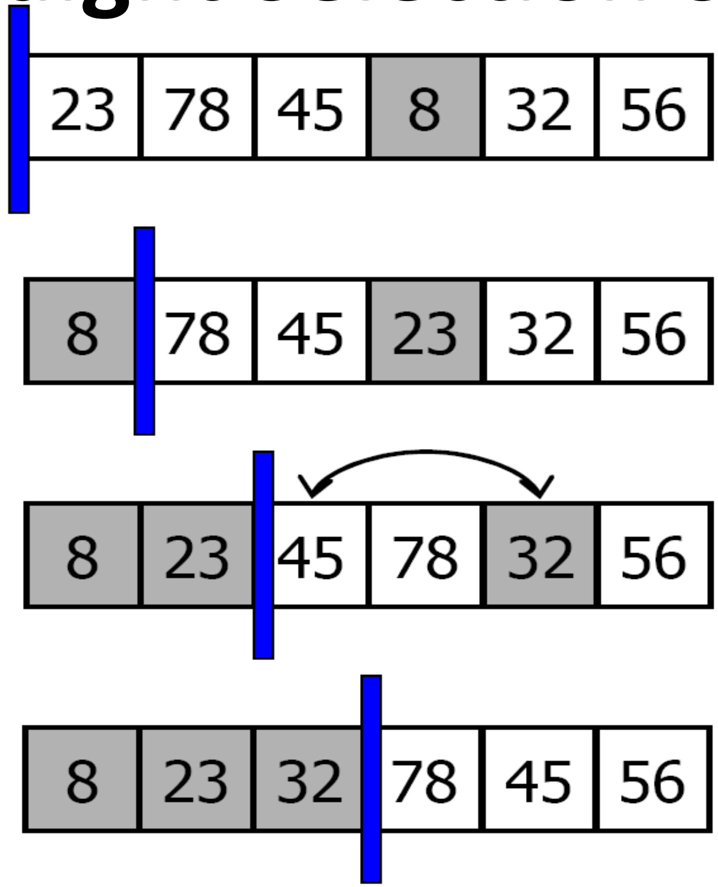
Straight Selection Sort



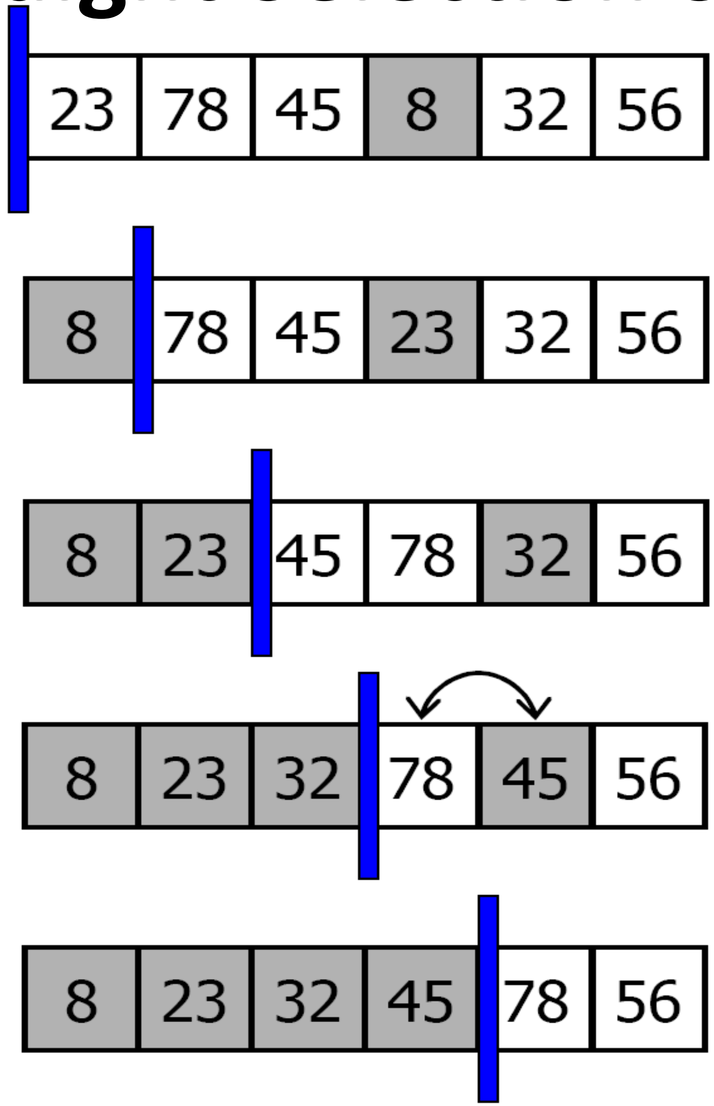
Straight Selection Sort



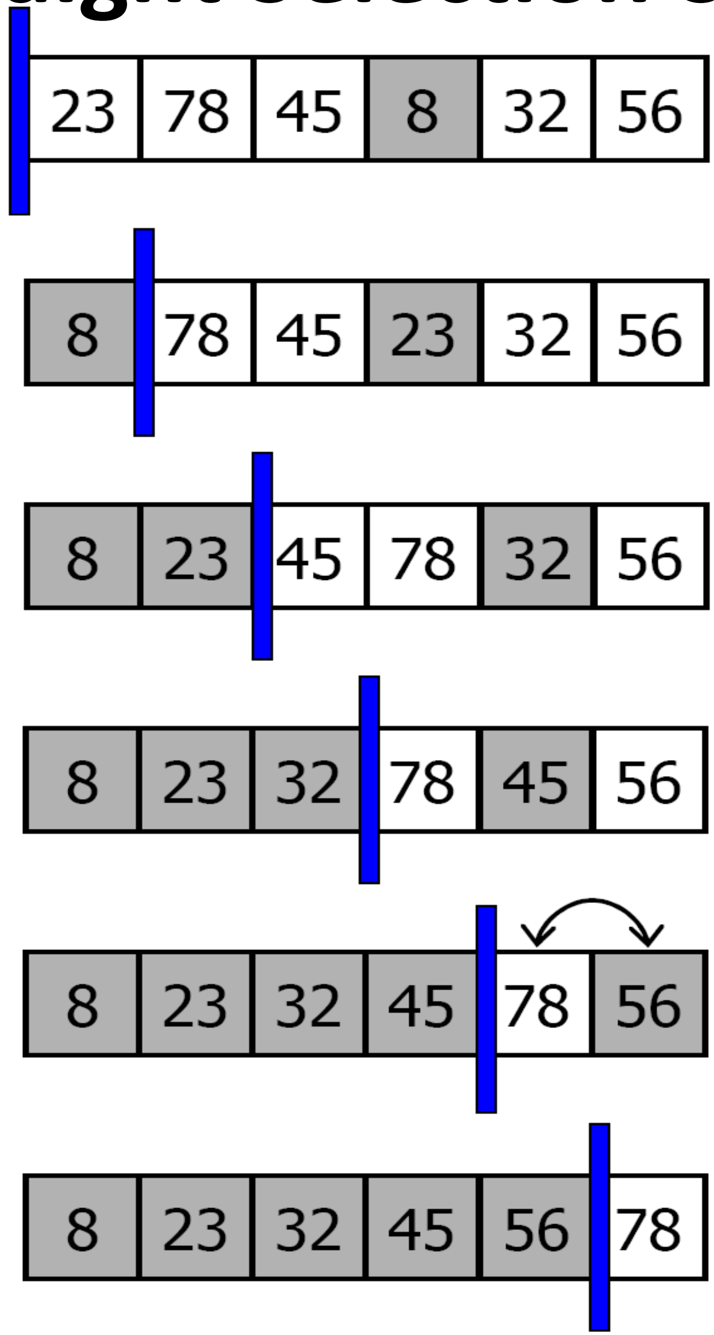
Straight Selection Sort



Straight Selection Sort



Straight Selection Sort



Selection Sort

Algorithm **SelectionSort** ()

Sorts the contiguous list using straight selection sort

Post   sorted list.

1. current = 0
2. **loop** (current < count - 1)
    1. smallest = current
    2. walker = current + 1
    3. **loop** (walker < count)
        1. if (data [walker].key < data [smallest].key)
            1. smallest = walker
        2. walker = walker+1
    4. swap(current, smallest)
    5. current = current + 1

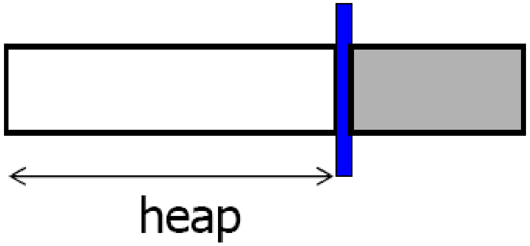
End SelectionSort

Heap Sort

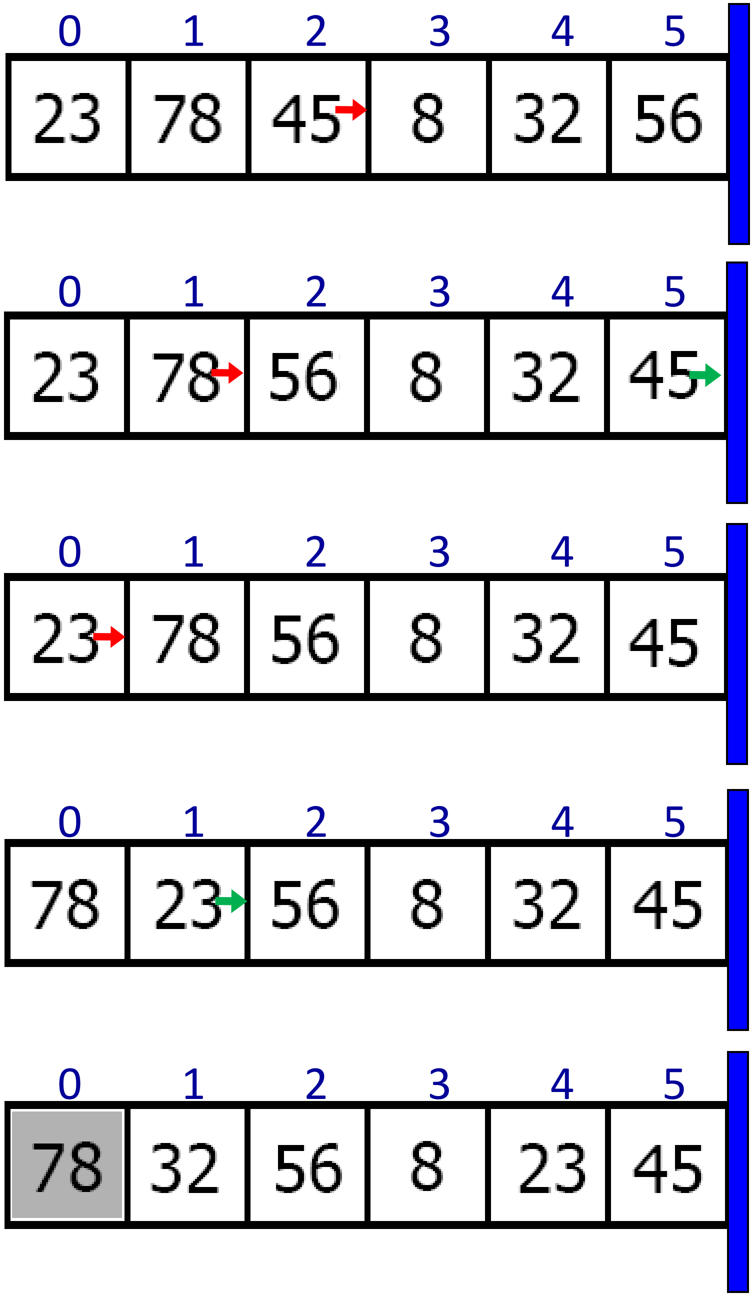
- The unsorted sublist is organized into a heap.

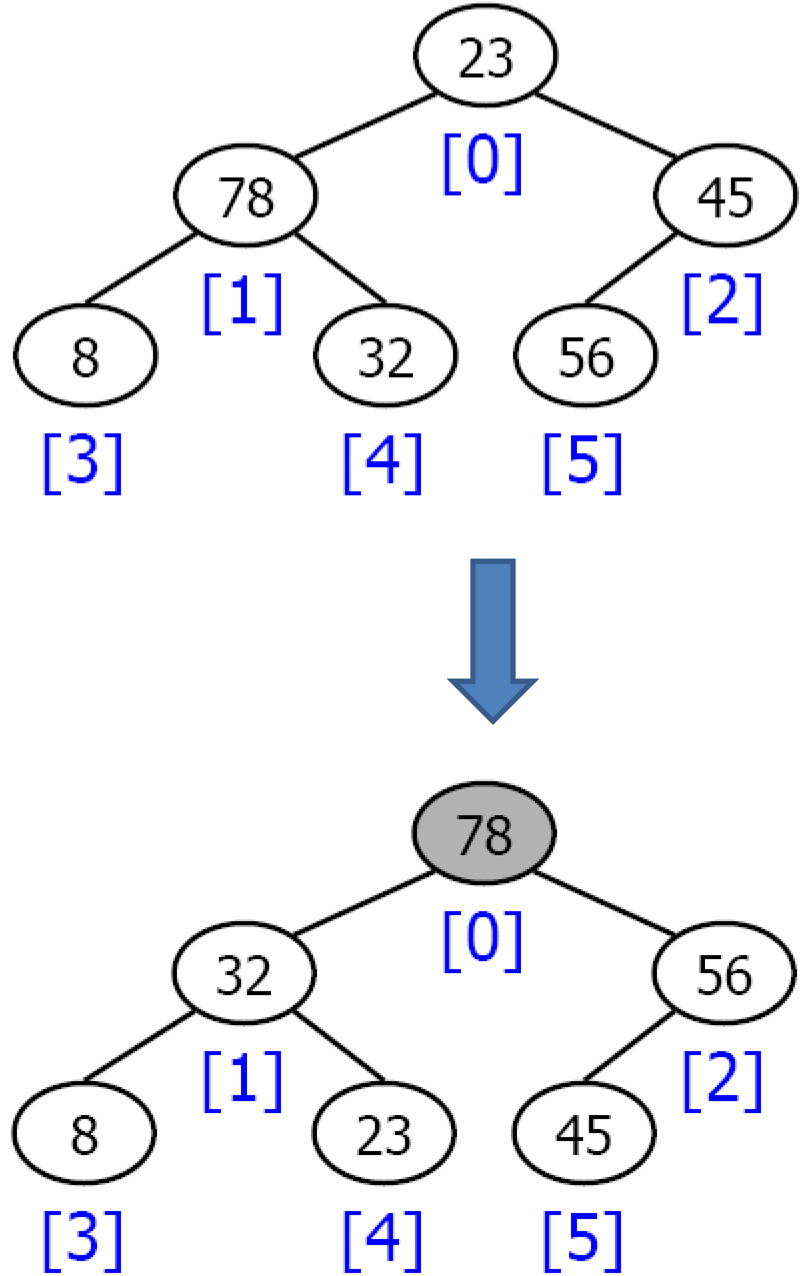
- In each pass, in the unsorted sublist, the largest element is selected and exchanged with the last element.

  Then the heap is reheaped.

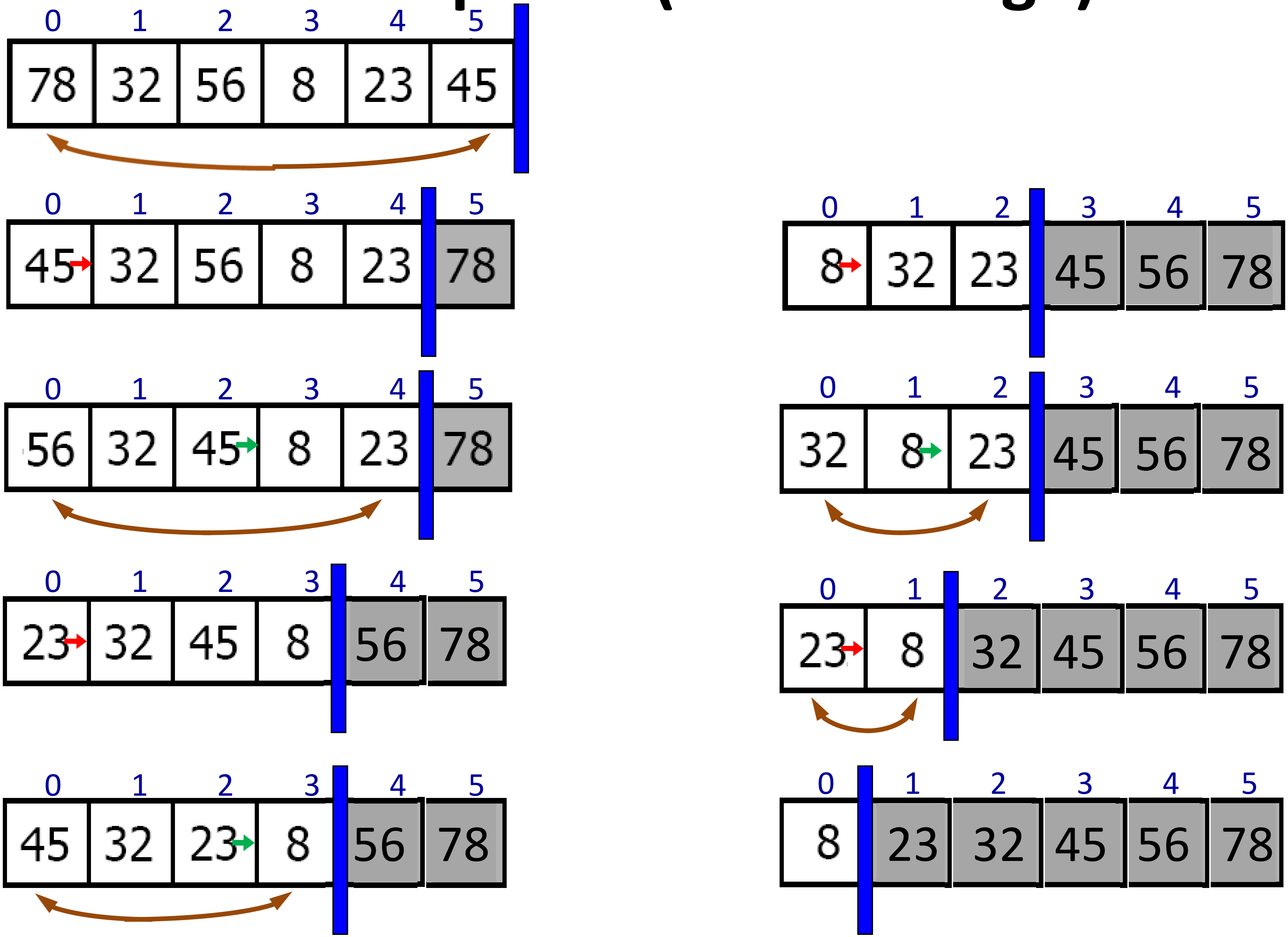


Build Heap (first stage)





Heap Sort (second stage)



Heap Sort

Algorithm **HeapSort** ()

Sorts the contiguous list using heap sort.

Post sorted list.

Uses Recursive function ReheapDown.

1. position = count / 2 -1 *// Build Heap*
2. **loop** (position >=0)
   1. ReheapDown(position, count-1)
   2. position = position - 1
3. last = count – 1 *// second stage of heapsort*
4. **loop** (last > 0)
   1. swap(0, last)
   2. last = last - 1
   3. ReheapDown(0, last - 1)

End HeapSort

Selection Sort Efficiency

- Straight selection sort: $O(n^2)$
- Heap sort: $O(n \log_2 n)$

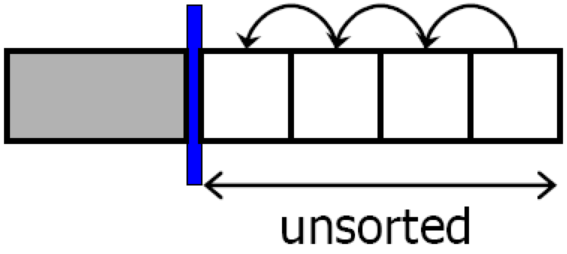
Exchange Sort

- In each pass, elements that are out of order are exchanged, until the entire list is sorted.

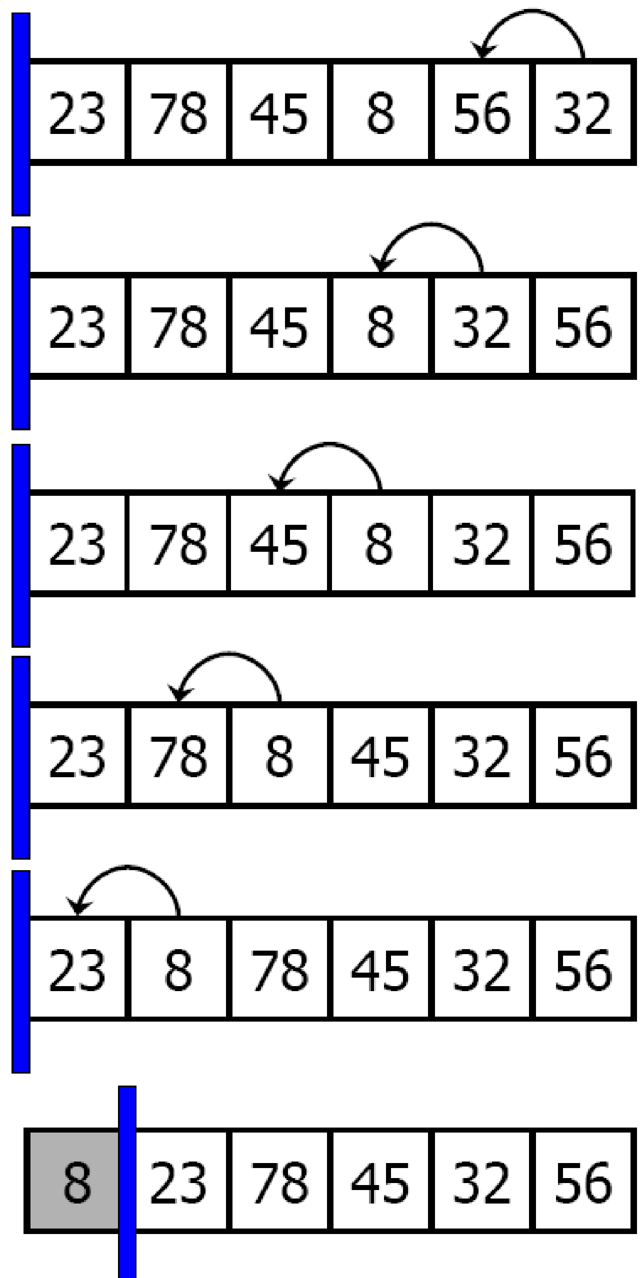
- Exchange is extensively used.

Bubble Sort

- The list is divided into two parts: sorted and unsorted.
- In each pass, the smallest element is bubbled from the unsorted sublist and moved to the sorted sublist.

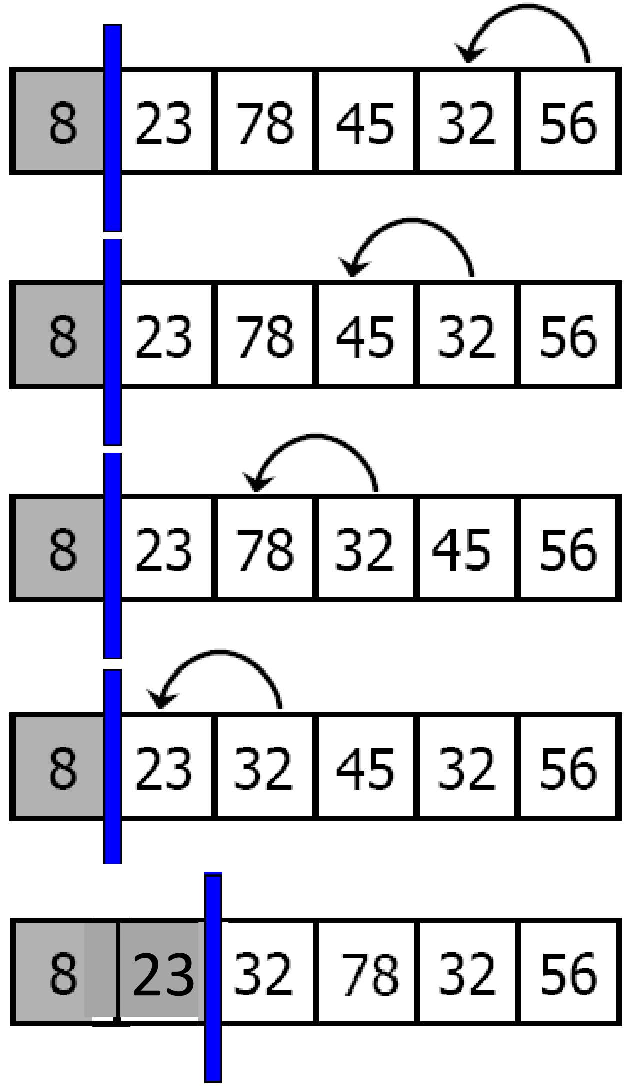


Bubble Sort



23 78 45 8 56 32
23 78 45 8 32 56
23 78 45 8 32 56
23 78 8 45 32 56
23 8 78 45 32 56
8 23 78 45 32 56

Bubble Sort


8 23 78 45 32 56
8 23 78 45 32 56
8 23 78 32 45 56
8 23 32 45 32 56
8 23 32 78 32 56

Bubble Sort

Algorithm **BubbleSort** ()

Sorts the contiguous list using straight bubble sort

Post sorted list.

1. current = 0
2. flag = FALSE
3. **loop** (current < count) AND (flag = FALSE)
    1. walker = count - 1
    2. flag = TRUE
    3. **loop** (walker > current)
        1. if (data [walker].key < data [walker-1].key)
            1. flag = FALSE
            2. swap(walker, walker – 1)
        2. walker = walker - 1
    4. current = current + 1

End BubbleSort

Exchange Sort efficiency

- Bubble sort:

$$f(n) = n(n + 1)/2 = O(n^2)$$

Divide-and-conquer sorting

Algorithm **DivideAndConquer**()

1. **if** (the list has length greater than 1)
    1. partition the list into lowlist, highlist
    2. lowlist. DivideAndConquer()
    3. highlist. DivideAndConquer()
    4. combine(lowlist, highlist)

End DivideAndConquer

Divide-and-conquer sorting

	Partition	Combine
Merge Sort	easily	hard
Quick Sort	hard	easily

Quick Sort

Algorithm **QuickSort**()

Sorts the contiguous list using quick sort.

Post Sorted list.

Uses function recursiveQuickSort.

1. recursiveQuickSort(0, count -1)

End QuickSort

Quick Sort

Algorithm **recursiveQuickSort**(val low <int>, val high <int>)

Sorts the contiguous list using quick sort.

Pre low and high are valid positions in contiguous list.

Post Sorted list.

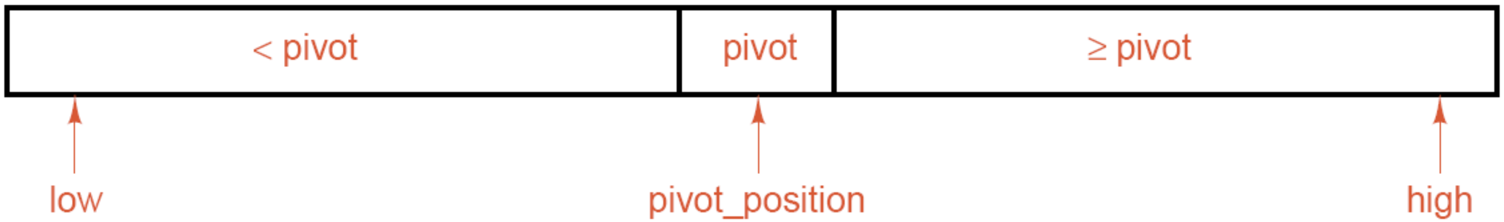
Uses functions recursiveQuickSort, Partition.

1. **if** (low < high) *// Otherwise, no sorting is needed.*
    1. pivot_position = Partition(low, high)
    2. recursiveQuickSort(low, pivot_position -1)
    3. recursiveQuickSort(pivot_position +1, high)

End recursiveQuickSort

Partition Algorithm

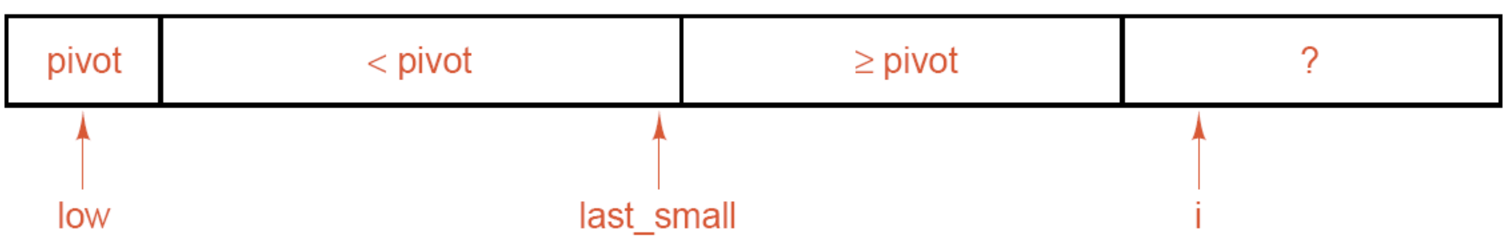
- Given a pivot value, the partition rearranges the entries in the list as below:



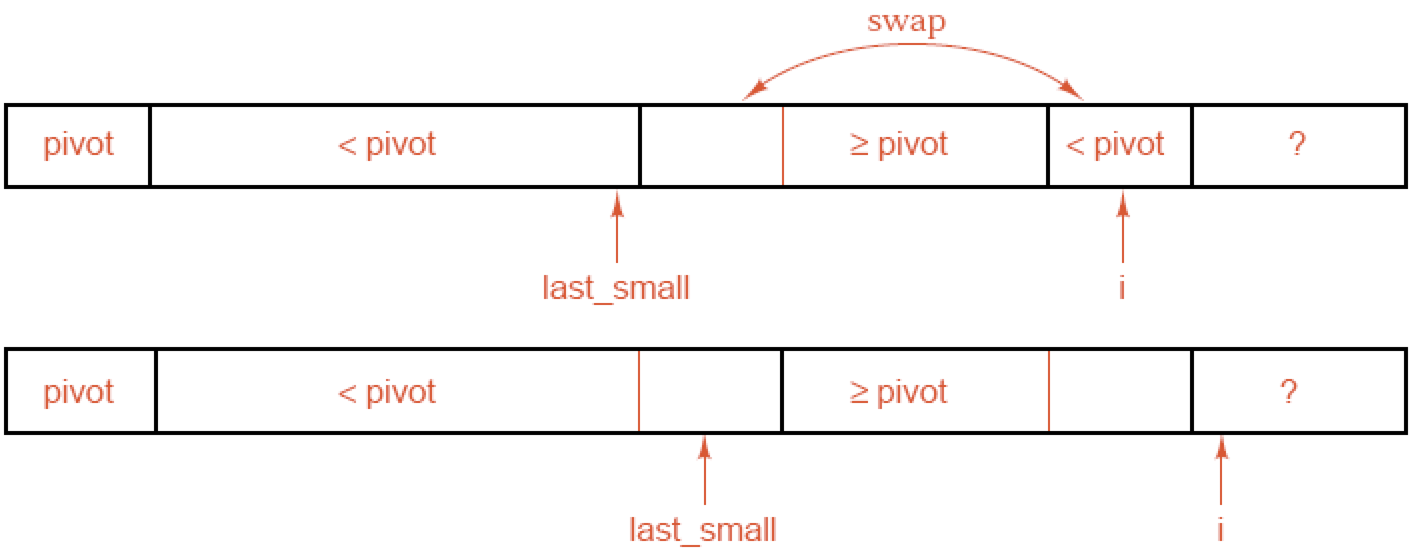
Partition Algorithm

Algorithm:

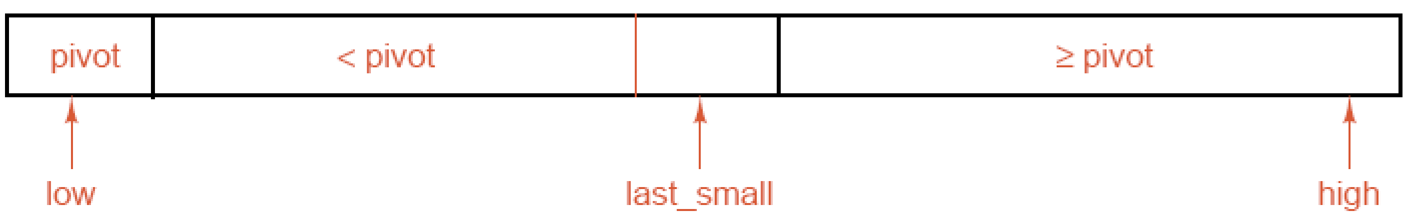
- Temporarily leave the pivot value at the first position.
- use a for loop running on a variable i, last_small is the position all entries at or before it have keys less than pivot.
- if the entry at i >= pivot, i can be increased.
- Otherwise, last_small is increased and two entries at position last_small and i are swapped:



Partition Algorithm



- When the loop terminates:



- At last, swap the pivot from position low to position last_small.

Partition in Quick Sort

<integer> **Partition**(val low <integer>, val high <integer>)

Partitions the entries between indices low and high to two sublists.

Pre low and high are valid positions in contiguous list, with low<=high.

Post The center entry in the range between indices low and high of the list has been chosen as a pivot.

All entries of the list between indices low and high, inclusive, have been rearranged so that those with keys less than the pivot come before the pivot , and the remaining entries come after the pivot. The final position of the pivot is returned.

Uses Function swap(val i <integer>, val j <integer>) interchanges entries in positions i and j.

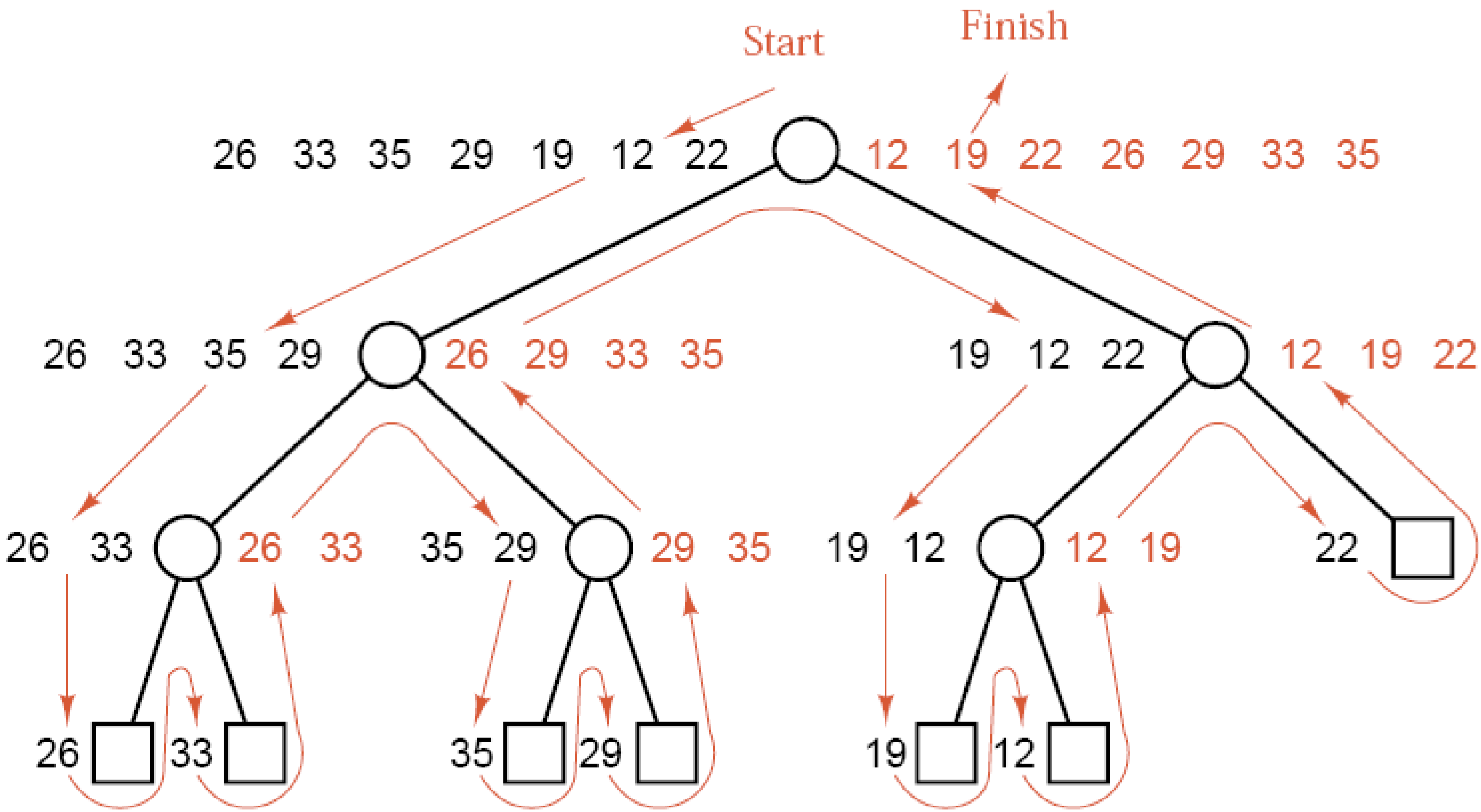
```
<integer> Partition(val low <integer>, val high <integer>)
// i  is used to scan through the list.
// last_small is the position of the last key less than pivot
1.   swap (low, (low+high)/2) // First entry is now pivot.
2.   pivot = entry_low
3.   last_small = low
4.   i = low + 1
5.   loop (i <= high)
     //entry_j.key  < pivot, when low < j <= last_small
     // entry_j.key >= pivot, when last_small < j < i
     1.   if (data_i < pivot)
          1.   last_small = last_small + 1
          2.   swap(last_small, i)  // Move large entry to right and small to left.
6.   swap(low, last_small)  // Put the pivot into its proper position.
7.   return last_small
End Partition
```

Quick Sort Efficiency

- Quick sort:

  $O(n \log_2 n)$

Merge Sort


Start
Finish
26 33 35 29 19 12 22
12 19 22 26 29 33 35
26 33 35 29
26 29 33 35
19 12 22
12 19 22
26 33
26 33
35 29
29 35
19 12
12 19
22
26
33
35
29
19
12

Merge Sort

Algorithm **MergeSort**() *// for linked list*

Sorts the linked list using merge sort

Post sorted list.

Uses recursiveMergeSort.

1. recursiveMergeSort(head)

End MergeSort

Merge Sort

Algorithm **recursiveMergeSort**(ref sublist <pointer>)

Sorts the linked list using recursive merge sort.

Post The nodes referenced by sublist have been reaaranged so that their keys are sorted into nondecreasing order.
The pointer parameter sublist is reset to point at the node containing the smallest key.

Uses functions recursiveMergeSort, Divide, Merge.

1. **if** (sublist is not NULL) AND (sublist->link is not NULL)
   1. Divide(sublist, second_list)
   2. recursiveMergeSort(sublist)
   3. recursiveMergeSort(secondlist)
   4. Merge(sublist, secondlist)

End recursiveMergeSort

Merge Sort

Algorithm **Divide**(val sublist <pointer>, ref secondlist <pointer>)

Divides the list into two halves.

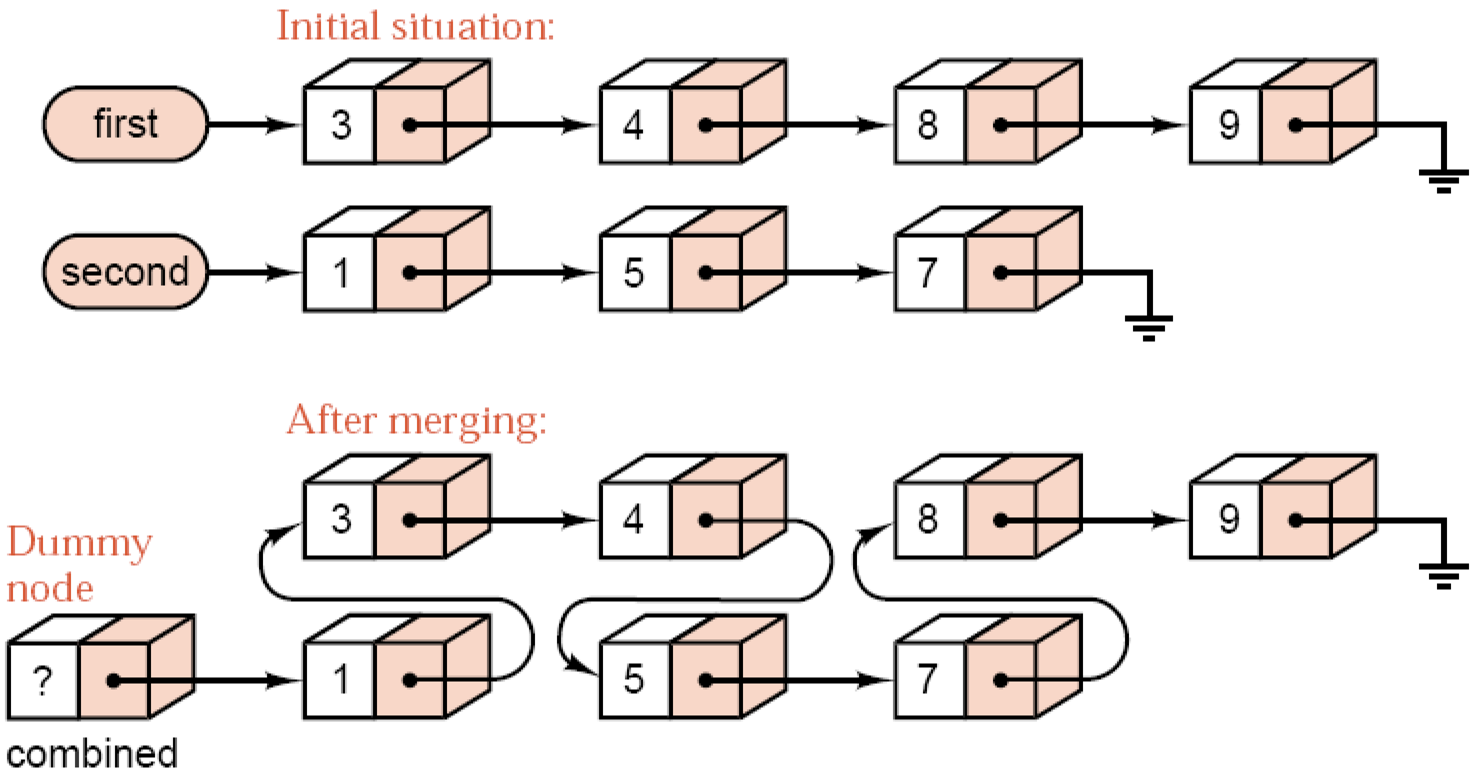
Pre sublist is not NULL.

Post The list of nodes referenced by sublist has been reduced to its first half, and secondlist points to the second half of the sublist. If the sublist has an odd number of entries, then its first half will be one entry larger than its second.

1. midpoint = sublist
2. position = sublist->link *// Traverse the entire list*
3. **loop** (position is not NULL) *// Move position twice for midpoint's one move.*
    1. position = position->link
    2. if (position is not NULL)
        1. midpoint = midpoint->link
        2. position = position->link
4. secondlist = midpoint->link
5. midpoint->link = NULL

End Divide

Merge two sublists



Merge two sublists

Algorithm **Merge** (ref first <pointer>, ref second <pointer>)

Merges two sorted lists to a sorted list.

Pre first and second point to ordered lists of nodes.

Post first points to an ordered list containing all nodes that were referenced by first and second. Second became NULL.

Algorithm **Merge** (ref first <pointer>, ref second <pointer>)

// lastSorted is a pointer points to the last node of sorted list.

// combined is a dummy first node, points to merged list.

1. lastSorted = address of combined
2. **loop** (first is not NULL) AND (second is not NULL) *// Attach node with smaller key*
    1. **if** (first->data.key <= second->data.key)
        1. lastSorted->link = first
        2. lastSorted = first
        3. first = first->link *// Advance to the next unmerged node*
    2. **else**
        1. lastSorted->link = second
        2. lastSorted = second
        3. second = second->link
3. **if** (first is NULL)
    1. lastSorted->link = second
    2. second = NULL
4. **else**
    1. lastSorted->link = first
5. first = combined.link

End Merge