Searching and Sorting Algorithms UNIT 4

CpE 1202L: Data Structures and Algorithms

COMPUTER ENGINEERING
UNIVERSITY of SAN CARLOS

Searching Algorithms

Searching

is the process of finding a certain information from a list.

Sequential Search

- Also known as **Linear search**

- the simplest, but most inefficient algorithm

- involves searching from the start of a list, for a match until one is found or no more item in the list is left
  - If the comparison shows the element is the one being searched for, return it's index.
  - if not, then move to the second element and compare it.
  - If we reach the end of the array, the search value is not in the array.

Sequential Search

```
Algorithm seqSearch (list, last, target, locn)
Locate the target in an unordered list of elements.
   Pre    list must contain at least one element
          last is index to last element in the list
          target contains the data to be located
          locn is address of index in calling algorithm
   Post   if found: index stored in locn & found true
          if not found: last stored in locn & found false
   Return found true or false
1  set looker to 0
2  loop (looker < last AND target not equal list[looker])
   1   increment looker
3  end loop
4  set locn to looker
5  if (target equal list[looker])
   1   set found to true
6  else
   1   set found to false
7  end if
8  return found
end seqSearch
```

Richard F. Gilberg and Behrouz A. Forouzan, Data Structures: A Pseudocode Approach with C, 2nd ed.
Thomson Learning, Inc. © 2005

Binary Search

- involves the repeated division and testing of the middle element for the proper location on item is most likely to be found

- a binary search or **half-interval search** algorithm locates the position of an item in a sorted array

Binary Search

- Binary search works by comparing an input value to the middle element of the array.
  - The comparison determines whether the element equals the input, less than the input or greater.
  - When the element being compared to equals the input the search stops and typically returns the position of the element.
  - If the element is not equal to the input then a comparison is made to determine whether the input is less than or greater than the element.
  - Depending on which it is the algorithm then starts over but only searching the top or bottom subset of the array's elements.
  - If the input is not located within the array the algorithm will usually output a unique value indicating this.

Binary Search

```
Algorithm binarySearch (list, last, target, locn)
Search an ordered list using Binary Search
   Pre    list is ordered; it must have at least 1 value
          last is index to the largest element in the list
          target is the value of element being sought
          locn is address of index in calling algorithm
   Post   FOUND: locn assigned index to target element
                 found set true
          NOT FOUND: locn = element below or above target
                     found set false
   Return found true or false
1  set begin to 0
2  set end to last
3  loop (begin <= end)
   1  set mid to (begin + end) / 2
   2  if (target > list[mid])
         Look in upper half
      1  set begin to (mid + 1)
   3  else if (target < list[mid])
         Look in lower half
      1  set end to mid - 1
   4  else
         Found: force exit
      1  set begin to (end + 1)
   5  end if
4  end loop
5  set locn to mid
6  if (target equal list [mid])
   1  set found to true
7  else
   1  set found to false
8  end if
9  return found
end binarySearch
```

Binary Search

index	0	1	2	3	4	5	6	7	8
data	5	12	17	23	38	44	77	84	90

- Example 1 | **Target : 44**
- Example 2 | **Target: 20**

Analyzing Search Algorithm

- Sequential Search

The efficiency of the sequential search is $O(n)$.

- Binary Search

The efficiency of the binary search is $O(\log n)$.

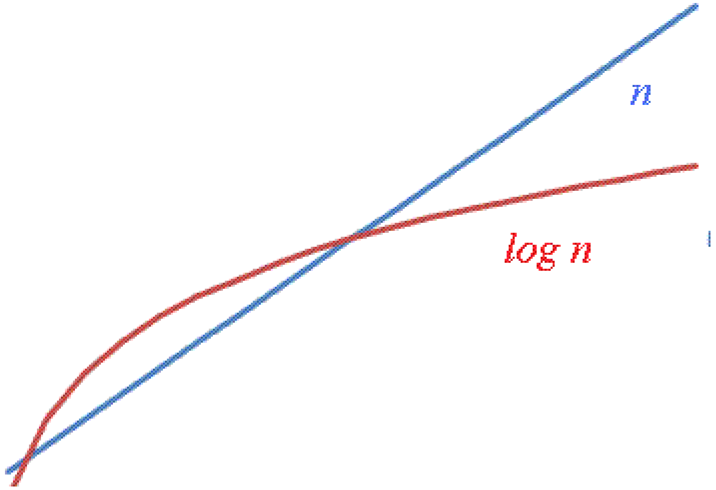
Comparison of Binary and Sequential Searches

List size	Iterations	
	Binary	Sequential
16	4	16
50	6	50
256	8	256
1000	10	1000
10,000	14	10,000
100,000	17	100,000
1,000,000	20	1,000,000

Search Analysis

- Binary search requires a more complex program than our original search and thus for **small n** it may run slower than the simple linear search. However, for **large n**,

$$\lim_{n \to \infty} \frac{\log n}{n} = 0$$



- Thus at large n, **log n is much smaller than n**, consequently an **Θ(log n) algorithm is much faster than an Θ(n)** one.

Sorting Algorithms

Sorting

is the process of arranging a set of similar information into an increasing or decreasing order.

Sorting

SORT

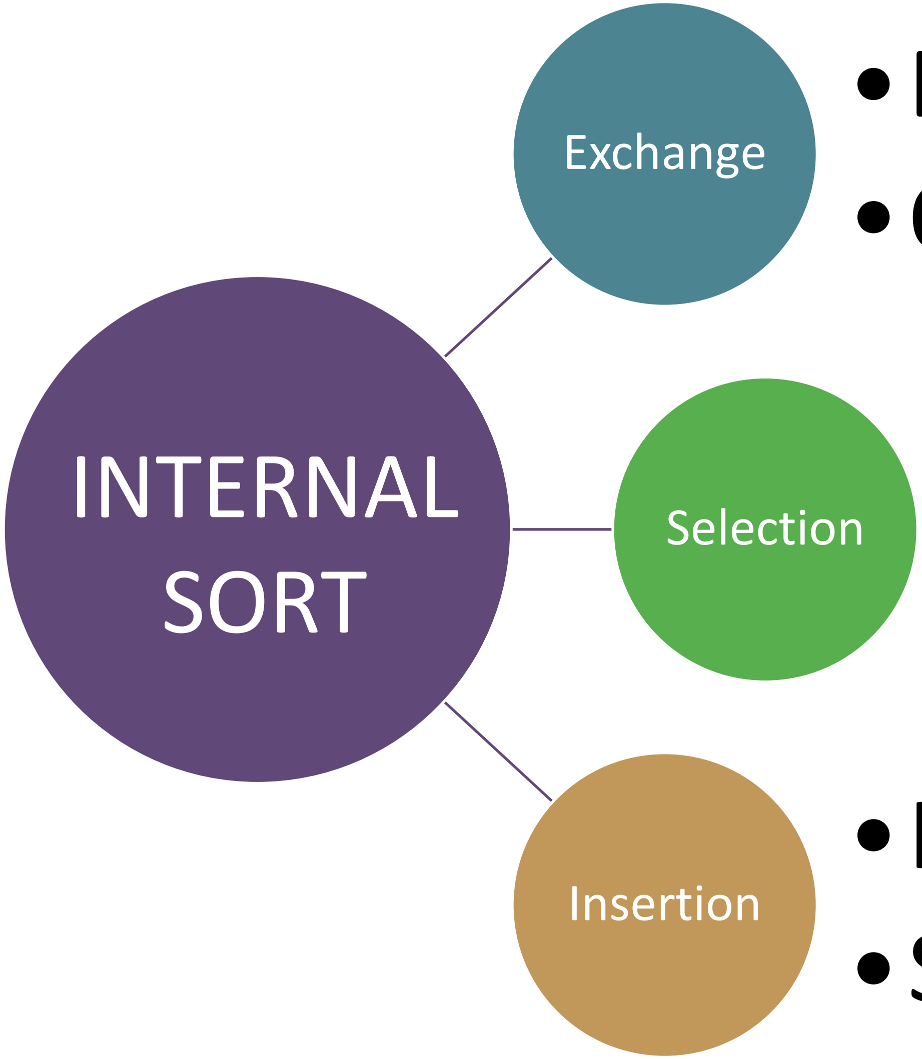
Internal

all of the data are held in primary storage during the sorting process

External

uses primary storage for the data currently being sorted and secondary storage for any data that does not fit in primary memory

Sorting



- Bubble
- Quick

- Selection
- Heap

- Insertion
- Shell

Sort Efficiency

- **Sort efficiency** is a measure of the relative efficiency of a sort. It is usually an estimate of the number of comparisons and moves required to order an unordered list.
- Three of the sorts (Exchange, Selection, Insertion) are $O(n^2)$.
- Best possible sorting algorithms are on the order of $n \log n$, that is $O(n \log n)$ sorts which is Quick Sort

Exchange Sort

In exchange sorts, exchange elements that are out of order until the entire list is sorted. Although virtually every sorting method uses some form of exchange, this kind of sort use it extensively.

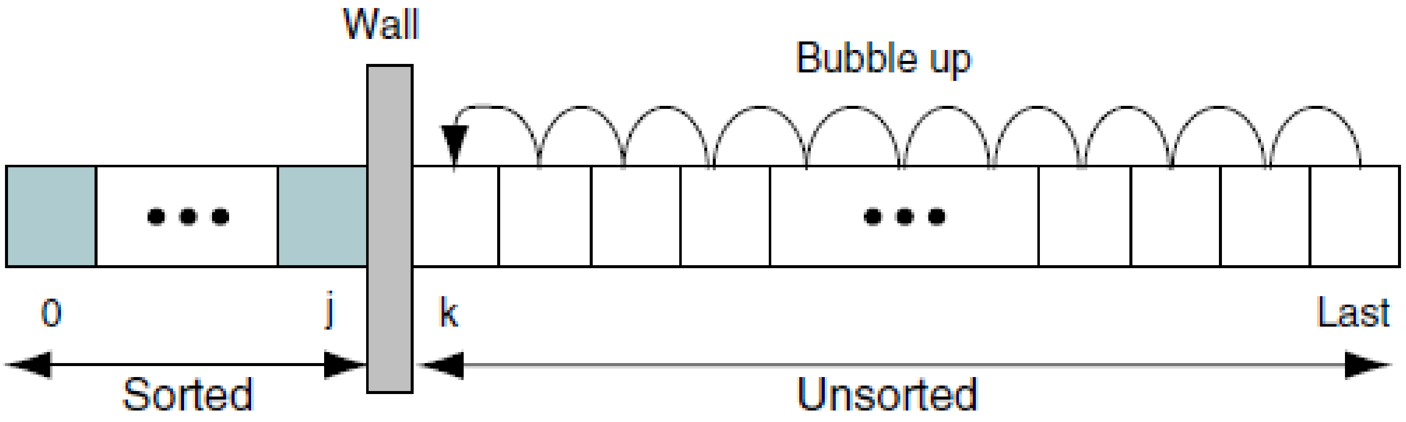
- Two exchange sorts: **Bubble Sort** and **Quick Sort**

Bubble Sort

- In the **bubble sort**, the list at any moment is divided into two sublists: sorted and unsorted.

- In each pass the smallest element is bubbled up from the unsorted sublist and moved to the sorted sublist.

Bubble Sort



Richard F. Gilberg and Behrouz A. Forouzan, Data Structures: A Pseudocode Approach with C, 2nd ed. Thomson Learning, Inc. © 2005

Bubble Sort

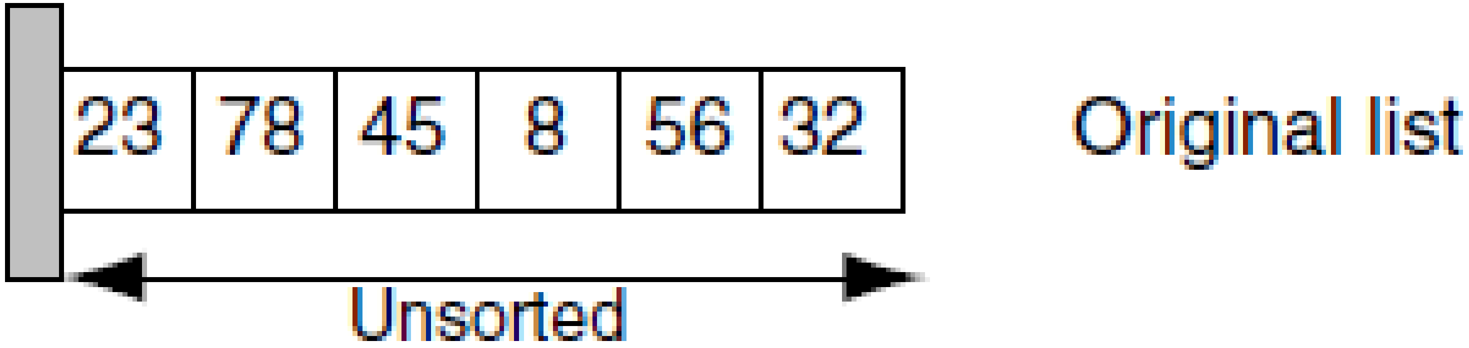
```
Algorithm  bubbleSort (list, last)
Sort an array using bubble sort. Adjacent
elements are compared and exchanged until list is
completely ordered.
   Pre  list must contain at least one item
        last contains index to last element in the list
   Post list has been rearranged in sequence low to high
 1 set current to 0
 2 set sorted  to false
 3 loop (current <= last AND sorted false)
      Each iteration is one sort pass.
   1  set walker to last
   2  set sorted to true
   3  loop (walker > current)
      1  if (walker data < walker - 1 data)
            Any exchange means list is not sorted.
         1  set sorted to false
         2  exchange (list, walker, walker - 1)
      2  end if
      3  decrement walker
   4  end loop
   5  increment current
 4 end loop
end bubbleSort
```

Richard F. Gilberg and Behrouz A. Forouzan, Data Structures: A Pseudocode Approach with C, 2nd ed.

COMPUTER ENGINEERING
UNIVERSITY of SAN CARLOS

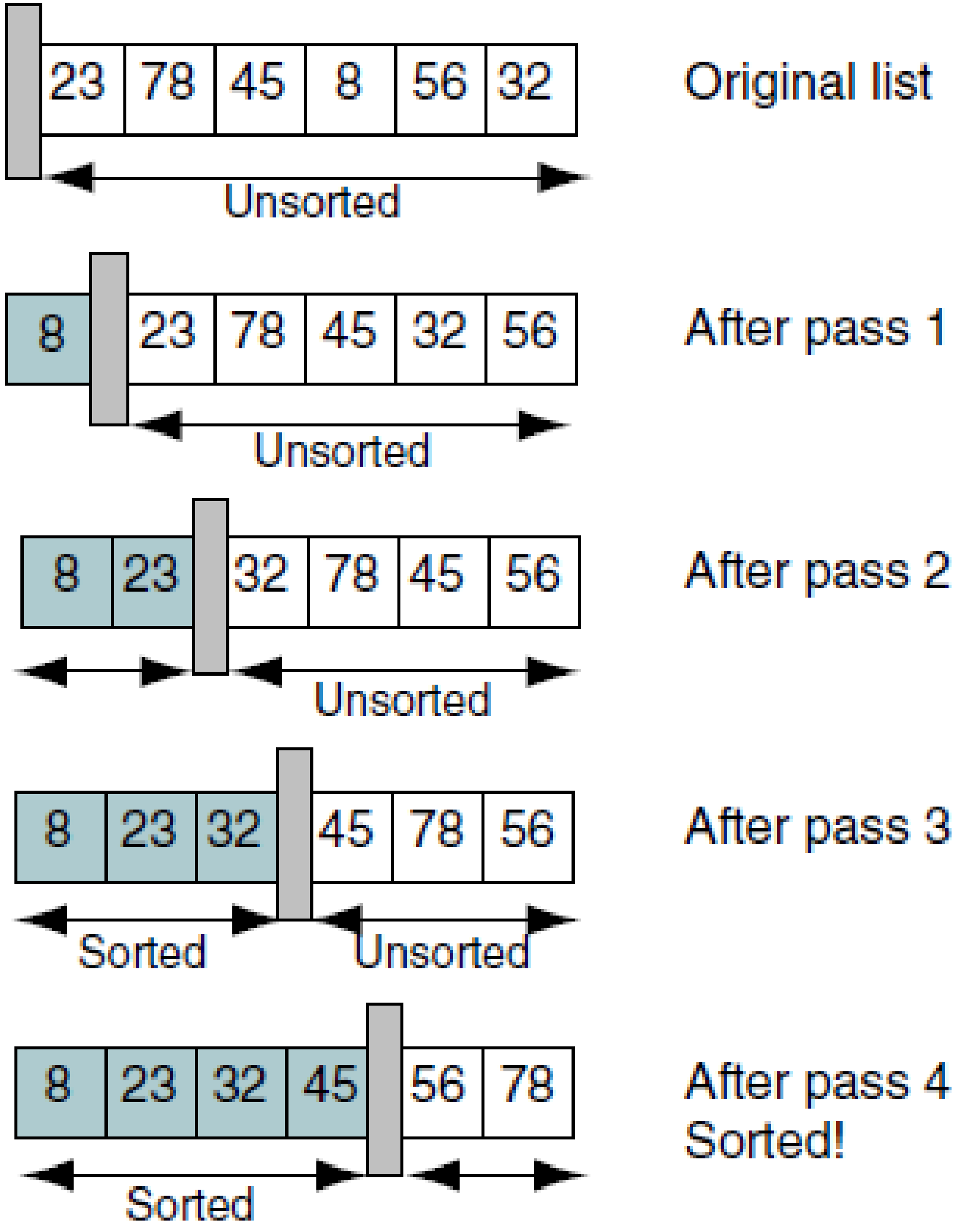
Bubble Sort

- Example



Bubble Sort

- Example



Selection Sort

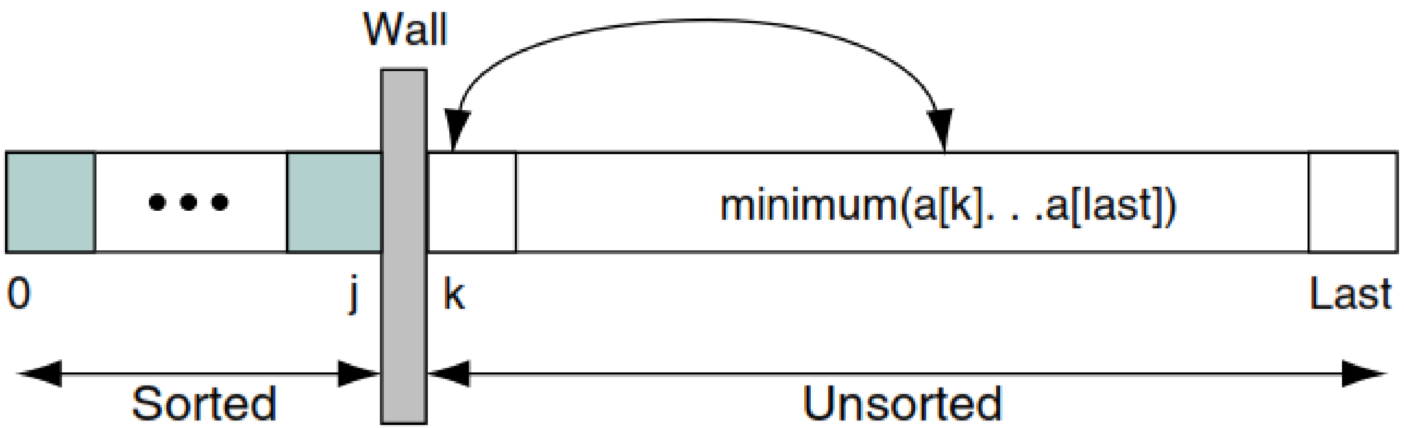
In each pass of the selection sort, the smallest element is selected from the unsorted sublist and exchanged with the element at the beginning of the unsorted list.

- Two selection sorts: **Straight Selection Sort** and **Heap Sort**.

Straight Selection Sort

- Concept:
  - The list at any moment is divided into two sublists, **sorted** and **unsorted**, which are divided by an imaginary wall.
  - Select the smallest element from the unsorted sublist and exchange it with the element at the beginning of the unsorted data.
  - After each selection and exchange, the wall between the two sublists moves one element, increasing the number of sorted elements and decreasing the number of unsorted ones.
  - Each time a move of one element from the unsorted sublist to the sorted sublist, it has completed one sort pass.
  - For a list of *n elements, therefore,* it need *n – 1 passes to completely rearrange the data.*

Straight Selection Sort



Selection Sort Concept

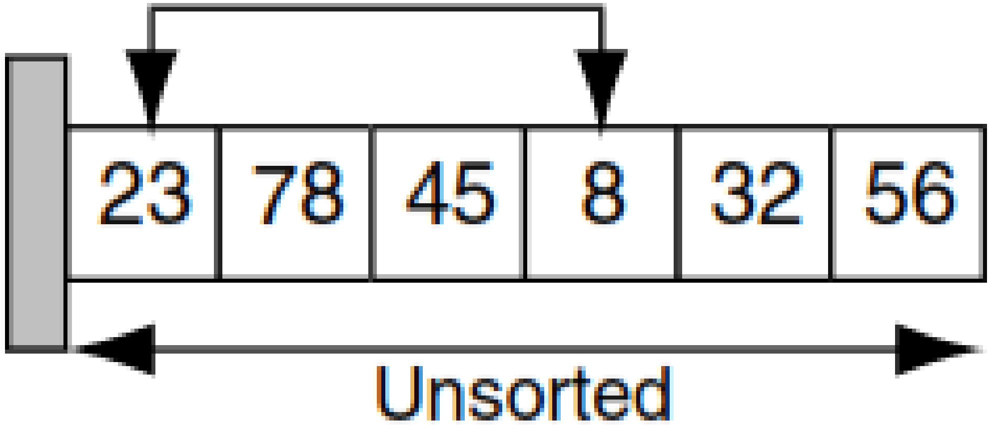
Straight Selection Sort

```
Algorithm selectionSort (list, last)
Sorts list array by selecting smallest element in
unsorted portion of array and exchanging it with element
at the beginning of the unsorted list.
   Pre  list must contain at least one item
        last contains index to last element in the list
   Post list has been rearranged smallest to largest
1  set current to 0
2  loop (until last element sorted)
   1  set smallest to current
   2  set walker   to current + 1
   3  loop (walker <= last)
      1  if (walker key < smallest key)
         1  set smallest to walker
      2  increment walker
   4  end loop
      Smallest selected: exchange with current element.
   5  exchange (current, smallest)
   6  increment current
3  end loop
end selectionSort
```

Richard F. Gilberg and Behrouz A. Forouzan, Data Structures: A Pseudocode Approach with C, 2nd ed. Thomson Learning, Inc. © 2005

Straight Selection Sort

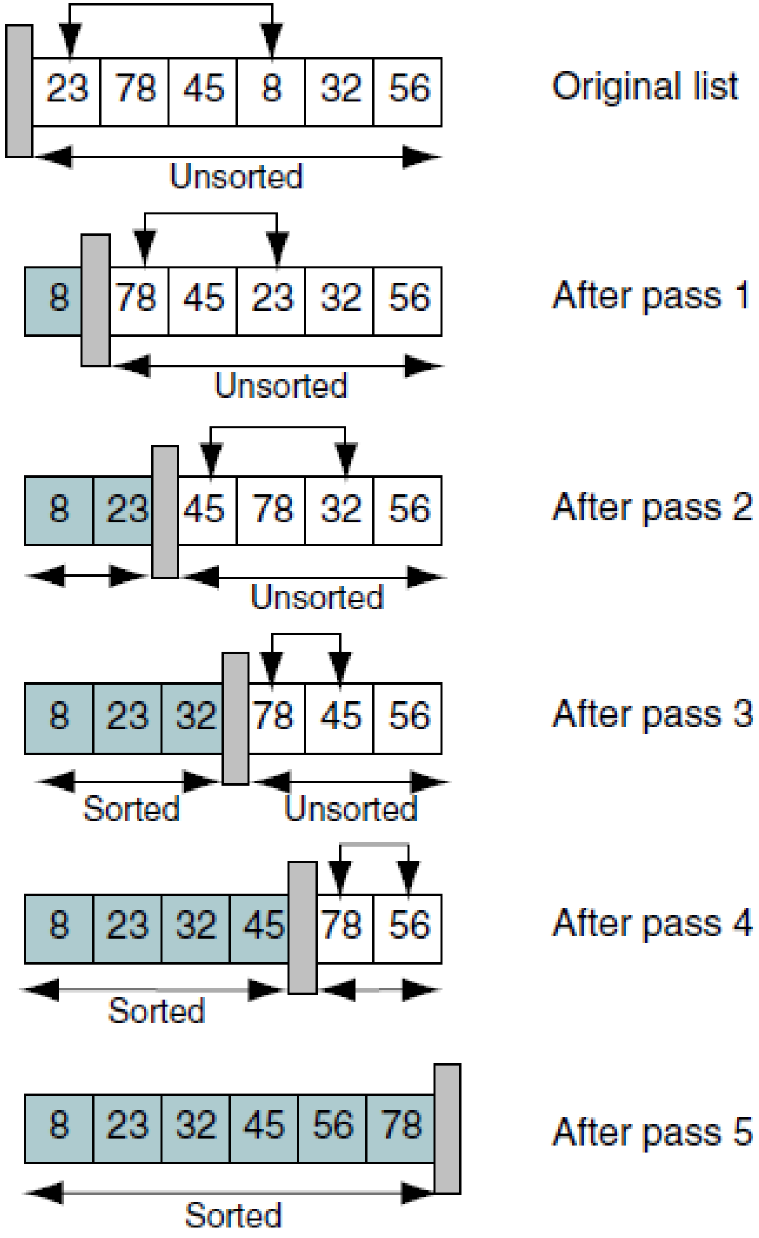
- Example



Original list

Straight Selection Sort

- Example



Insertion Sort

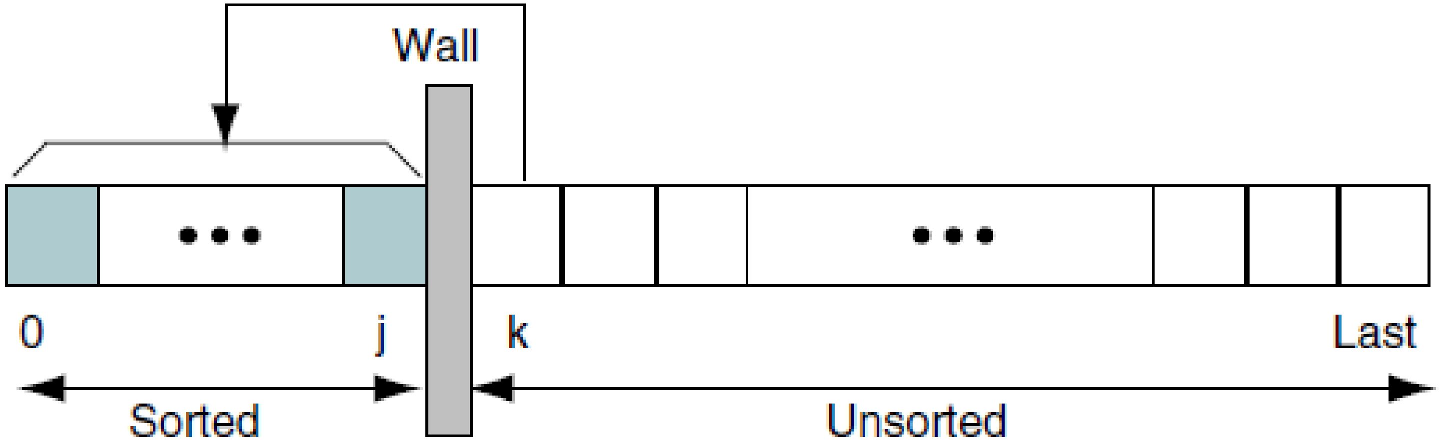
In each pass of an insertion sort, one or more pieces of data are inserted into their correct location in an ordered list.

- Two insertion sorts: **Straight Insertion Sort** and **Shell Sort**

Straight Insertion Sort

- In a **straight insertion sort**, the list at any moment is divided into two sublists: sorted and unsorted.
- In each pass the first element of the unsorted sublist is transferred to the sorted sublist by inserting it at the appropriate place.

Straight Insertion Sort



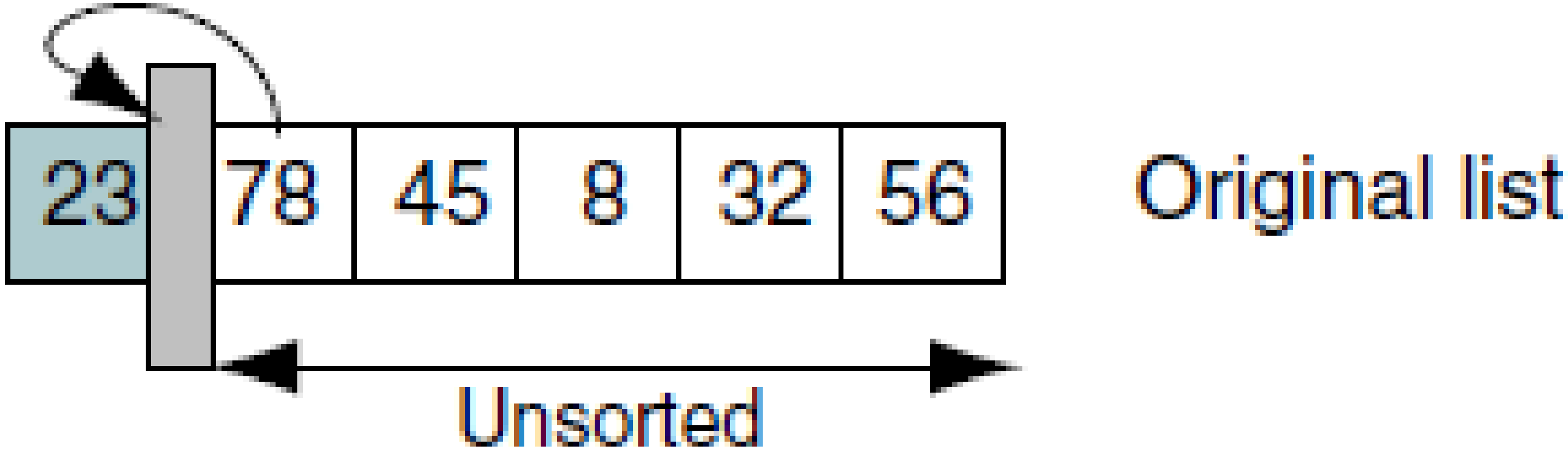
Insertion Sort Concept

Straight Insertion Sort

```
Algorithm insertionSort (list, last)
Sort list array using insertion sort. The array is
divided into sorted and unsorted lists. With each pass, the
first element in the unsorted list is inserted into the
sorted list.
   Pre  list must contain at least one element
        last is an index to last element in the list
   Post list has been rearranged
1 set current to 1
2 loop (until last element sorted)
   1 move current element to hold
   2 set walker to current - 1
   3 loop (walker >= 0 AND hold key < walker key)
      1 move walker element right one element
      2 decrement walker
   4 end loop
   5 move hold to walker + 1 element
   6 increment current
3 end loop
end insertionSort
```

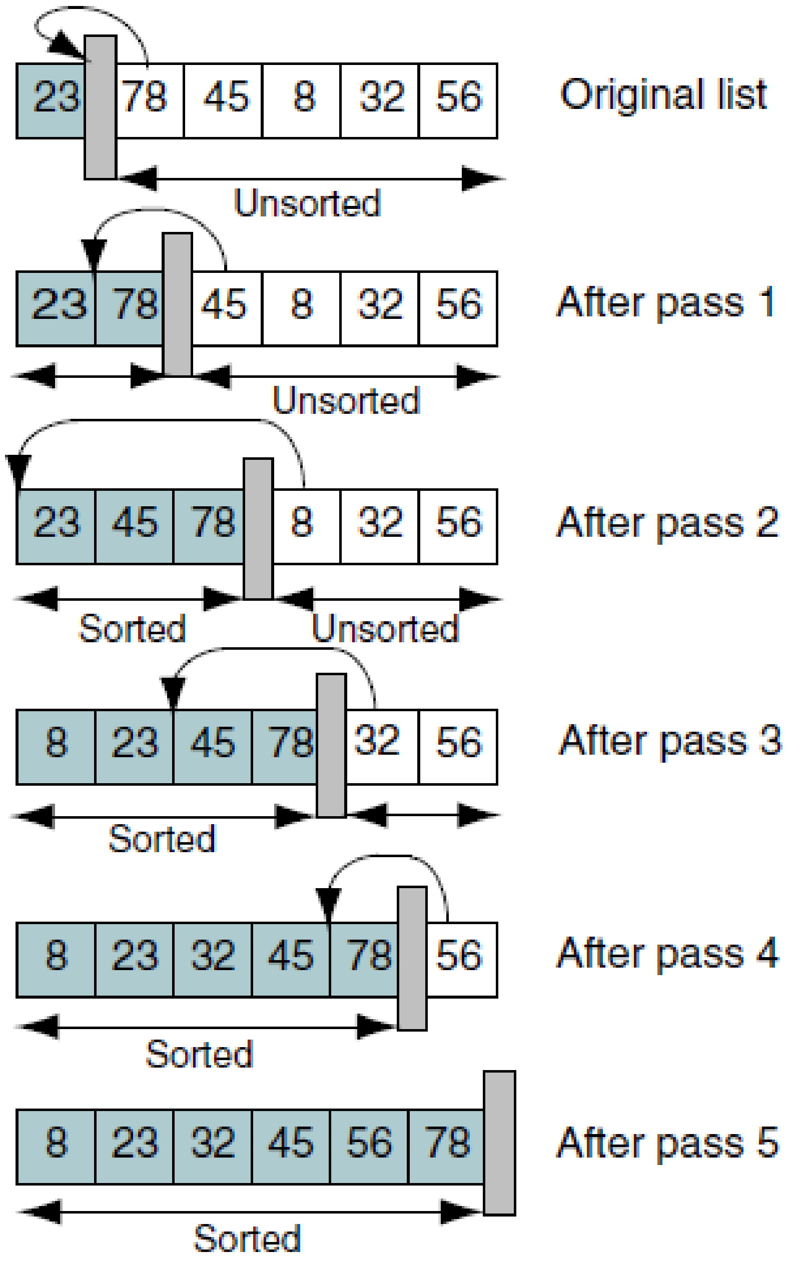
Straight Insertion Sort

- Example



Straight Insertion Sort

- Example



Sort Efficiency

Bubble Sort

- The number of loops in the inner loop depends on the current location in the outer loop. It therefore loops through half the list on the average. The total number of loops is the product of both loops, making the bubble sort efficiency
- $n\left(\frac{n+1}{2}\right)$
- The bubble sort efficiency is **O(n²)**.

Straight Selection Sort

- The outer loop executes n – 1 times. The inner loop also executes n – 1 times.
- The straight selection sort efficiency is **O(n²)**.

Sort Efficiency

Straight Insertion Sort

- Dependent quadratic loop, which is mathematically stated as
- $f(n) = n\left(\frac{n+1}{2}\right)$

- The straight insertion sort efficiency is **O(n²)**.

Sort Comparisons

n	Number of Loops Bubble, Straight Selection, Straight Insertion
25	625
100	10,000
500	250,000
1,000	1,000,000
2,000	4,000,000

Some Variation of Sorts

Improved version of Exchange, Selection and Insertion Sorts

Quick Sort

- The **quick sort** is the new version of the exchange sort in which the list is continuously divided into smaller sublists and exchanging takes place between elements that are out of order.

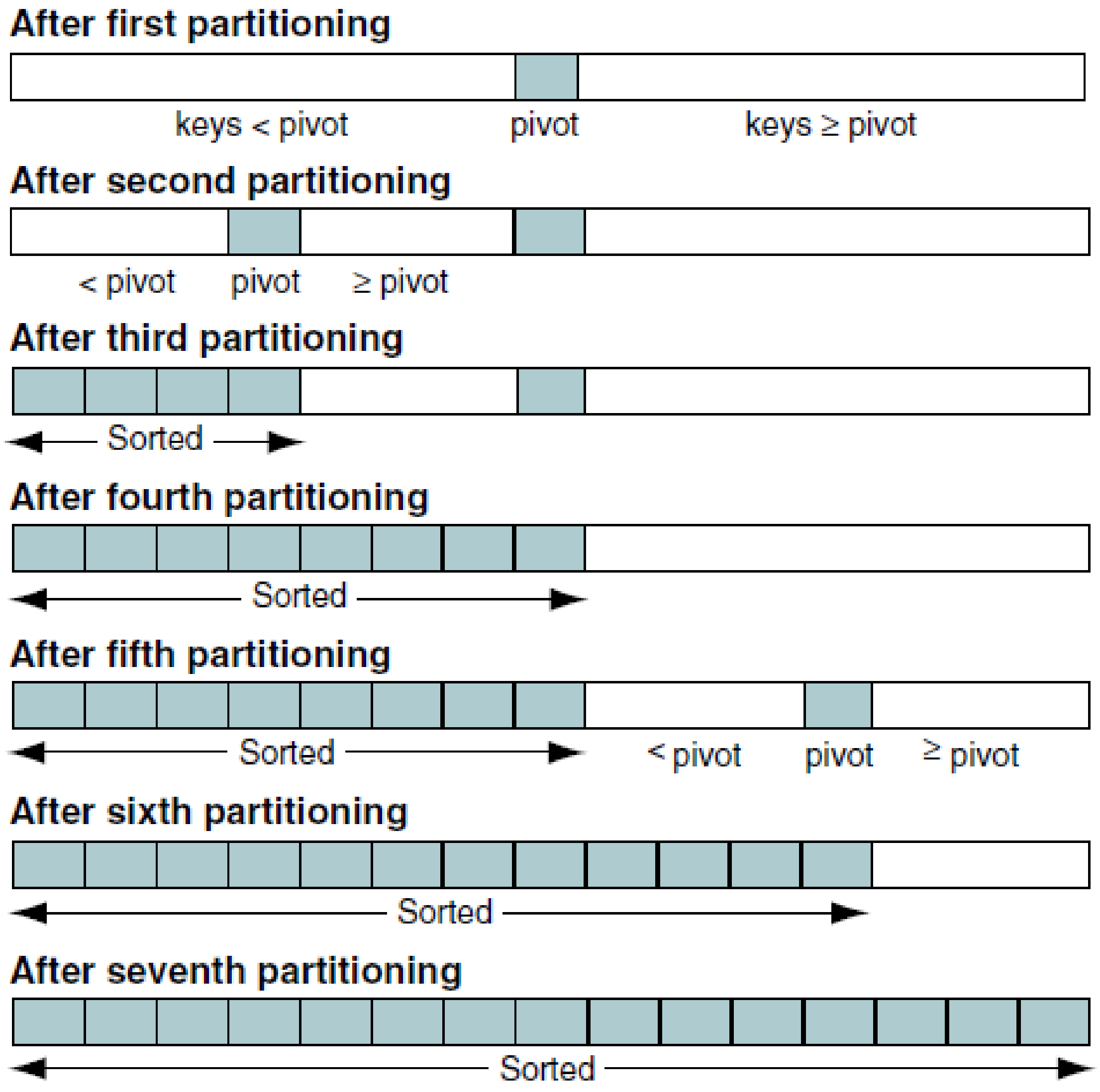
- Each pass of the quick sort selects a pivot and divides the list into three groups: a partition of elements whose key is less than the pivot's key, the pivot element that is placed in its ultimate correct position, and a partition of elements greater than or equal to the pivot's key. The sorting then continues by quick sorting the left partition followed by quick sorting the right partition.

Quick Sort

- **Quick sort** is an exchange sort in which a pivot key is placed in its correct position in the array while rearranging other elements widely dispersed across the list. Uses the idea of dived and conquer.

- Quick Sort in three recursive steps:
  - Find the pivot key (leftmost, rightmost or middle element) that divides the array into two partitions
  - Quick sort the left partition
  - Quick sort the right partition

Quick Sort



Richard F. Gilberg and Behrouz A. Forouzan, Data Structures: A Pseudocode Approach with C, 2nd ed.
Thomson Learning, Inc. © 2005

Quick Sort

```
Algorithm quickSort (list, left, right)
An array, list, is sorted using recursion.
   Pre  list is an array of data to be sorted
        left and right identify the first and last
        elements of the list, respectively
   Post list is sorted
 1 if ((right - left) > minSize)
      Quick sort
   1  medianLeft (list, left, right)
   2  set pivot     to left element
   3  set sortLeft  to left + 1
   4  set sortRight to right
   5  loop (sortLeft    <= sortRight)
         Find key on left that belongs on right
      1  loop (sortLeft key < pivot key)
         1  increment sortLeft
      2  end loop
         Find key on right that belongs on left
      3  loop (sortRight key >= pivot key)
         1  decrement sortRight
      4  end loop
      5  if (sortLeft <= sortRight)
         1  exchange(list, sortLeft, sortRight)
         2  increment sortLeft
         3  decrement sortRight
      6  end if
   6  end loop
```

Quick Sort

```
          Prepare for next pass
      7   move sortLeft - 1 element to left element
      8   move pivot element to sortLeft - 1 element
      9   if  (left < sortRight)
          1  quickSort (list, left, sortRight - 1)
     10   end if
     11   if (sortLeft < right)
          1  quickSort (list, sortLeft, right)
     12   end if
   2  else
      1   insertionSort (list, left, right)
   3  end if
  end quickSort
```

Quick Sort

- Example 1

17	41	5	22	54	6	29	3	13

17	41	5	22	54	6	29	3	13
3	41	5	22	54	6	29	17	13
3	41	5	22	54	6	29	17	13
3	6	5	22	54	41	29	17	13
3	6	5	22	54	41	29	17	13
3	6	5	22	54	41	29	17	13
3	6	5	13	54	41	29	17	22

Left Partition
all < pivot

Right Partition
all > pivot

Quick Sort

- Example 1 (con't)

3	6	5	13	54	41	29	17	22
3	6	5	13	54	41	29	17	22
3	5	6	13	54	41	29	17	22

3	5	6	13	54	41	29	17	22
3	5	6	13	17	41	29	54	22
3	5	6	13	17	41	29	54	22
3	5	6	13	17	22	29	54	41

Quick Sort

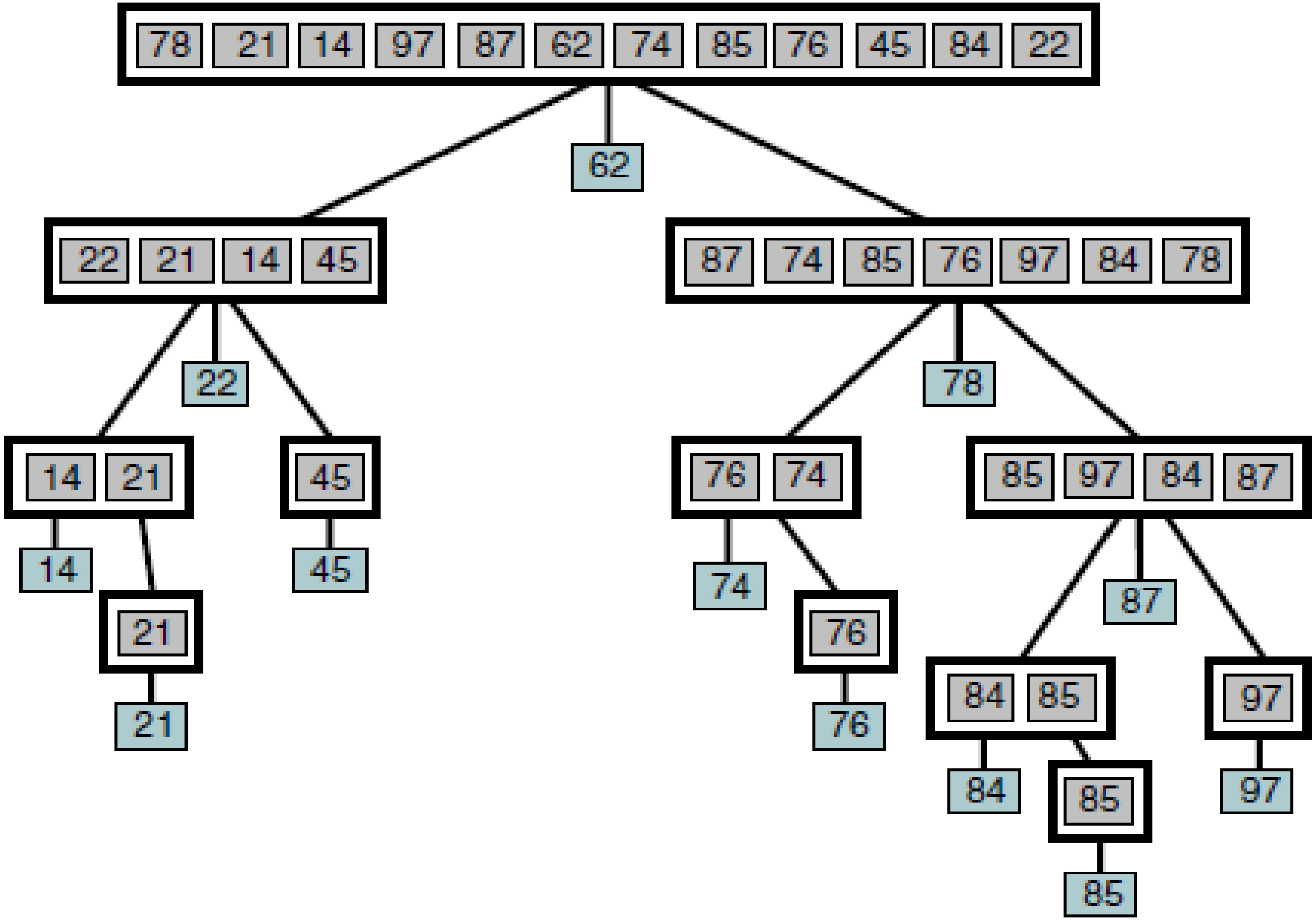
- Example 1 (con't)

3	5	6	13	17	22	29	54	41

3	5	6	13	17	22	29	54	41
3	5	6	13	17	22	29	54	41
3	**5**	**6**	**13**	**17**	**22**	**29**	**41**	**54**

Quick Sort

- Example 2



Exchange Sort Efficiency

Bubble Sort

- The number of loops in the inner loop depends on the current location in the outer loop. It therefore loops through half the list on the average. The total number of loops is the product of both loops, making the bubble sort efficiency
- $n\left(\frac{n+1}{2}\right)$
- The bubble sort efficiency is **$O(n^2)$**.

Quick Sort

- Quick sort uses pivot key which is in the middle of the array and used a median value. Assuming that it is located relatively close to the center, it divides the list into two sublists of roughly the same size.
- Because it divides by 2, the number of loops is logarithmic. The total sort effort is therefore the product of the first loop times the recursive loops, or n log n.
- The quick sort efficiency is **O(n log n)**.

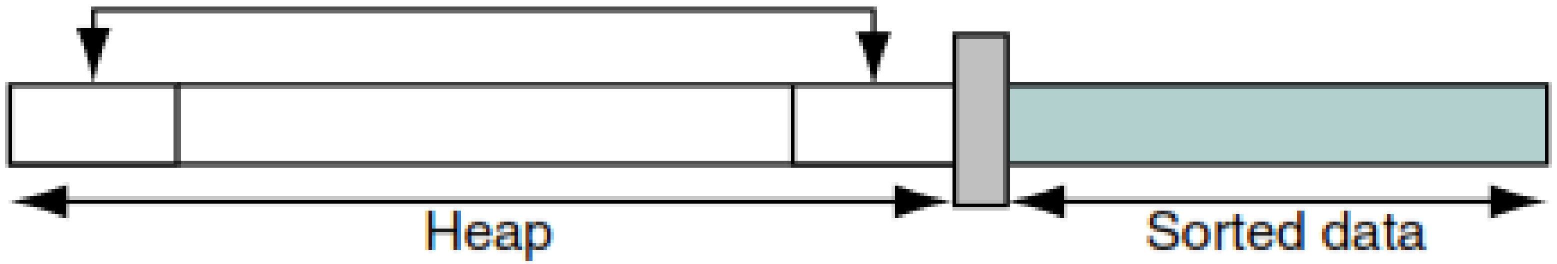
Exchange Sort

- Comparison of Exchange Sorts

n	Number of loops	
	Bubble	Quick
25	625	116
100	10,000	664
500	250,000	4482
1000	1,000,000	9965
2000	4,000,000	10,965

Heap Sort

- The **heap sort** is an improved version of the selection sort in which the largest element (the root) is selected and exchanged with the last element in the unsorted list.



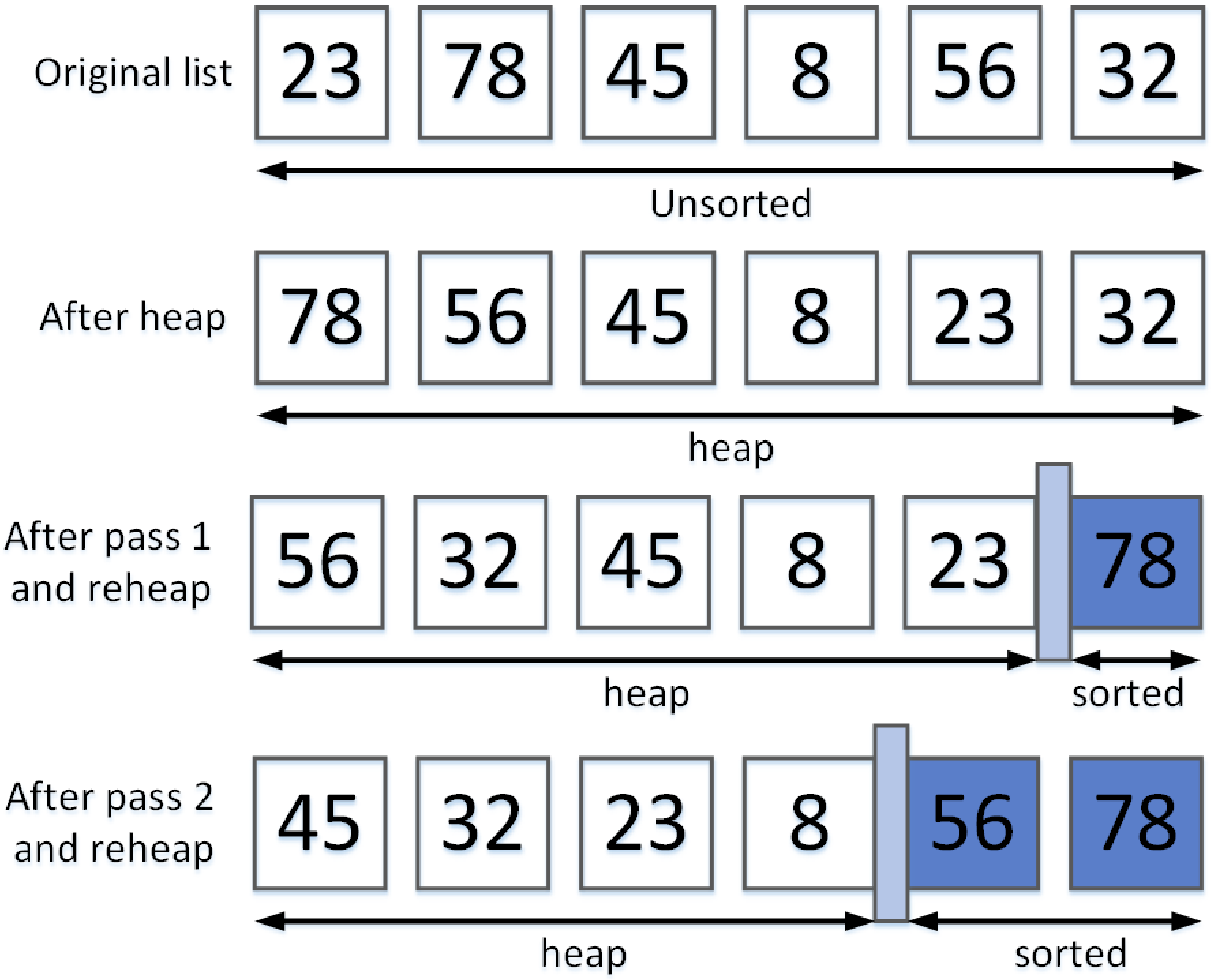
(a) Heap sort exchange process

Heap Sort

```
Algorithm heapSort (heap, last)
Sort an array, using a heap.
   Pre  heap array is filled
        last is index to last element in array
   Post heap array has been sorted
   Create heap
1  set walker to 1
2  loop (heap built)
   1  reheapUp (heap, walker)
   2  increment walker
3  end loop
   Heap created. Now sort it.
4  set sorted to last
5  loop (until all data sorted)
   1  exchange (heap, 0, sorted)
   2  decrement sorted
   3  reheapDown (heap, 0, sorted)
6  end loop
end heapSort
```

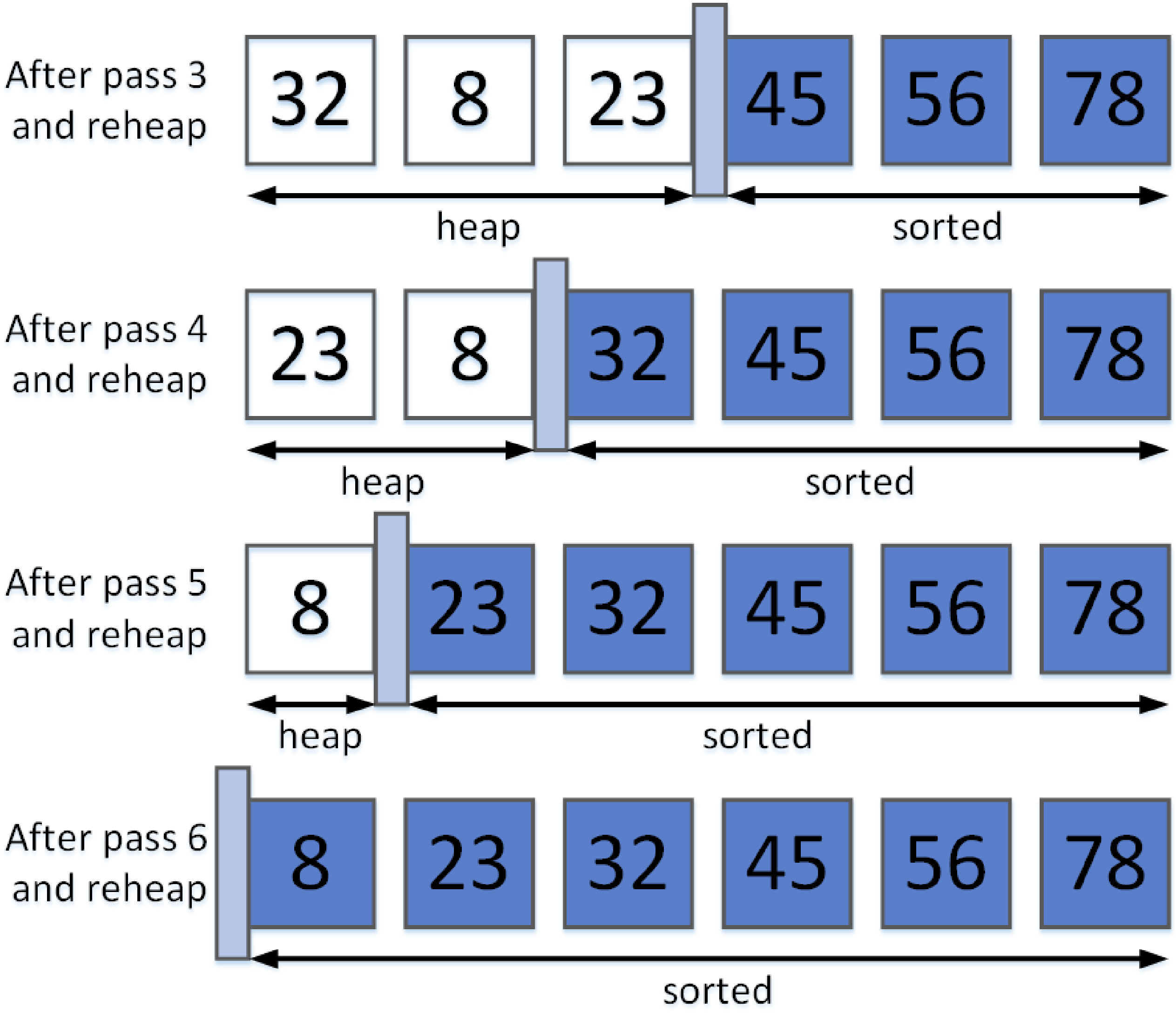
Heap Sort

- Example



Heap Sort

- Example (con't)



Selection Sort Efficiency

Straight Selection Sort

- The outer loop executes n – 1 times. The inner loop also executes n – 1 times.
- The straight selection sort efficiency is **O(n^2)**.

Heap Sort

- The outer loop starts at the end of the array and moves through the heap one element at a time until it reaches the first element. It therefore loops n times. The inner loop follows a branch down a binary tree from the root to a leaf or until the parent and child data are in heap order.
- The heap sort efficiency is **O(n log n)**.

Selection Sort Efficiency

- Comparison of Selection Sort

n	Number of loops	
	Straight Selection	Heap
25	625	116
100	10,000	664
500	250,000	4482
1000	1,000,000	9965
2000	4,000,000	10,965

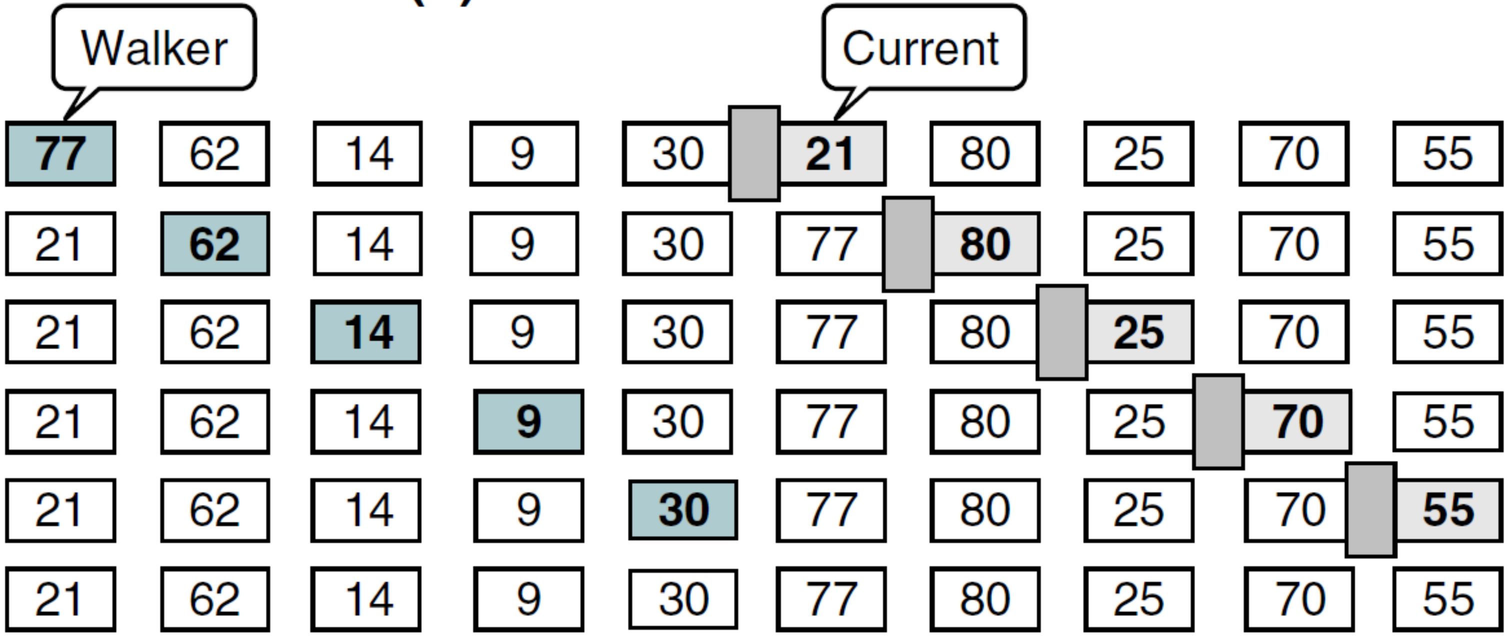
Shell Sort

- The **shell sort** is an improved version of the straight insertion sort in which diminishing partitions are used to sort the data.
- In the shell sort, a list of *N* elements is divided into *K* segments, where *K* is known as the increment. Each segment contains a minimum of integral *N/K*; if *N/K* is not an integral, some of the segments will contain an extra element.

Shell Sort

- Example

(a) First increment: $K = 5$



Shell Sort

- Example (con't)

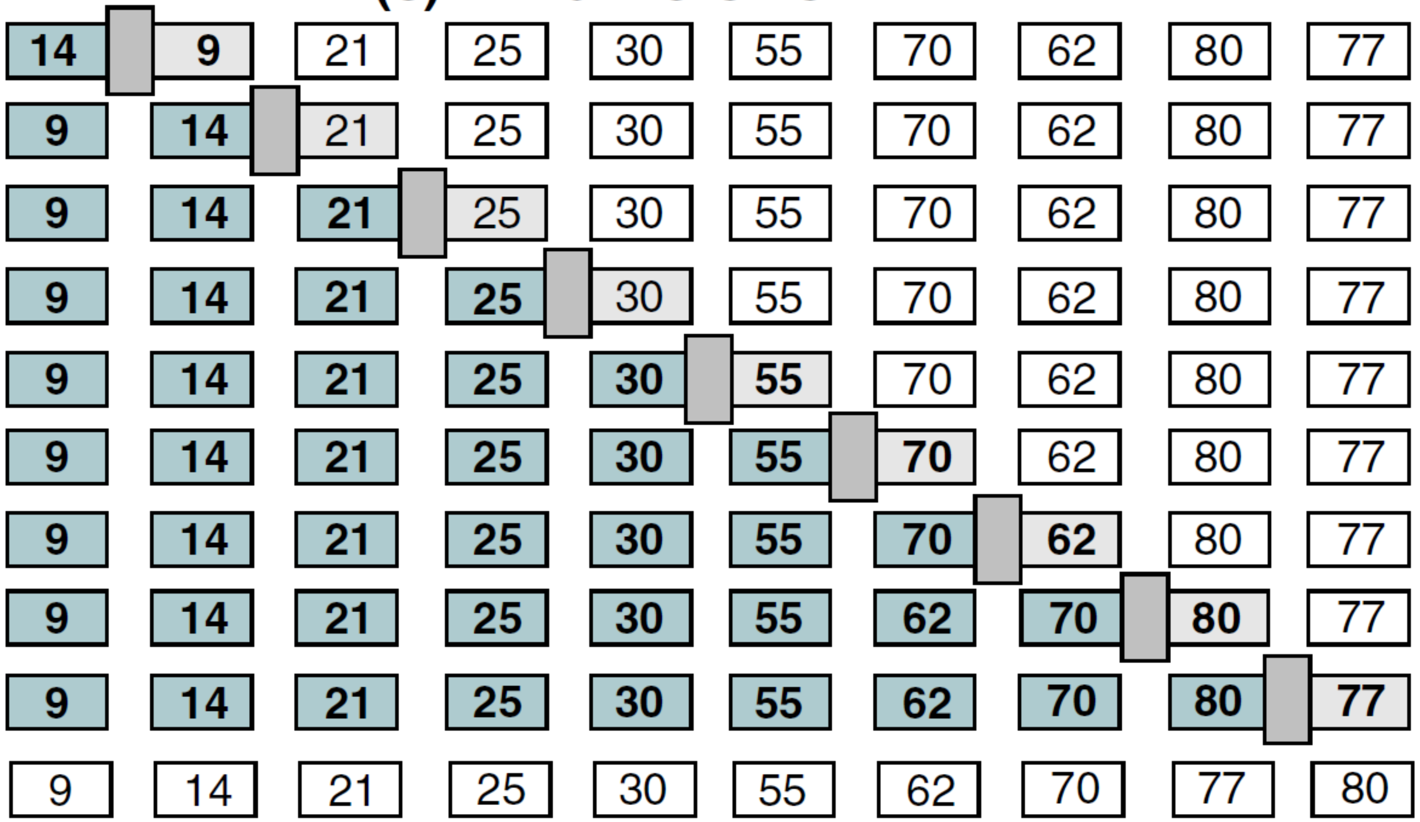
(b) Second increment: $K = 2$

21	62	14	9	30	77	80	25	70	55
14	9	21	62	30	77	80	25	70	55
14	9	21	62	30	77	80	25	70	55
14	9	21	62	30	77	80	25	70	55
14	9	21	62	30	77	80	25	70	55
14	9	21	62	30	77	80	25	70	55
14	9	21	25	30	62	80	77	70	55
14	9	21	25	30	62	70	77	80	55
14	9	21	25	30	55	70	62	80	77

Shell Sort

- Example (con't)

(c) Third increment: $K = 1$



Shell Sort

- Example (con't)

(d) Sorted array

9	14	21	25	30	55	62	70	77	80

Shell Sort

```
Algorithm shellSort (list, last)
Data in list array are sorted in
place. After the sort, their keys will be in order,
list[0] <= list[1] <= … <= list[last].
   Pre  list is an unordered array of records
        last is index to last record in array
   Post list is ordered on list[i].key
 1 set incre to last / 2
   Compare keys "increment" elements apart.
 2 loop (incre not 0)
   1  set current to incre
   2  loop (until last element sorted)
      1  move current element to hold
      2  set walker to current - incre
      3  loop (walker >= 0 AND hold key < walker key)
            Move larger element up in list.
         1  move walker element one increment right
            Fall back one partition.
         2  set walker to walker - incre
      4  end loop
         Insert hold record in proper relative location.
      5  move hold to walker + incre element
      6  increment current
   3  end loop
      End of pass--calculate next increment.
   4  set incre to incre / 2
 3 end loop
end shellSort
```

Insertion Sort Efficiency

Straight Insertion Sort

- Dependent quadratic loop, which is mathematically stated as
$$f(n) = n\left(\frac{n+1}{2}\right)$$
- The straight insertion sort efficiency is **O(n²)**.

Shell Sort

- The total number of iterations for the outer loop and the first inner loop is shown below:

$$logn \times \left[\left(n-\frac{n}{2}\right)+\left(n-\frac{n}{2}\right)+\left(n-\frac{n}{2}\right)+\cdots+1\right]$$
$$= n\log n$$

- With third loop. The result is something greater than O(n log n).
- Knuth's estimates from his empirical studies that the average sort effort is 15n1.25.
- The shell sort efficiency is **$O(n^{1.25})$**.

Insertion Sort

- Comparison of Insertion Sort

n	Number of loops	
	Straight Insertion	Shell
25	625	55
100	10,000	316
500	250,000	2364
1000	1,000,000	5623
2000	4,000,000	13,374

Summary | Sort Comparisons

n	Number of loops		
	Bubble, Straight Selection, Straight Insertion	Shell	Quick, Heap
25	625	55	116
100	10,000	316	664
500	250,000	2364	4482
1000	1,000,000	5623	9965
2000	4,000,000	13,374	10,965

References

- Richard F. Gilberg and Behrouz A. Forouzan, **Data Structures: A Pseudocode Approach with C**, 2nd ed. Thomson Learning, Inc. © 2005

COMPUTER ENGINEERING
UNIVERSITY *of* SAN CARLOS