Sorting

Insertion sort
Selection sort
Bubble sort

The efficiency of handling data can be substantially improved if the data are sorted according to some criteria of order. In a telephone directory we are able to locate a phone number, only because the names are alphabetically ordered. Same thing holds true for listing of directories created by us on the computer. It would be very annoying if we don't follow some order to store book indexes, payrolls, bank accounts, customer records, items inventory records, especially when the number of records is pretty large.

- We want to keep information in a sensible order.
  - alphabetical order
  - ascending/descending order
  - order according to names, ids, years, departments etc.

- The aim of sorting algorithms is to put unordered information in an ordered form.

- There are dozens of sorting algorithms. The more popular ones are listed below:
  - Selection Sort
  - Bubble Sort
  - Insertion Sort
  - Merge Sort
  - Quick Sort

As always, we want to decide which algorithms are best for a particular situation. The efficiency is decided by the number of comparisons and the number of data movements, using the big-O notation. The order of

magnitude can vary depending on the initial ordering of data.

How much time, does a computer spend on data ordering if the data are already ordered? We often try to compute the data movements, and comparisons for the following three cases:

best case (often, data is already in order),
worst case(sometimes, the data is in reverse order),
and average case(data in random order).

Some sorting methods perform the same operations regardless of the initial ordering of data. Why should we consider both comparisons and data movements?
If simple keys are compared, such as integers or characters, then the comparisons are relatively fast and inexpensive. If strings or arrays of numbers are compared, then the cost of comparisons goes up substantially.
If on the other hand, the data items moved are large, such as structures, then the movement measure may stand out as the determining factor in efficiency considerations.

Insertion Sort

The key idea is to pick up a data element and insert it into its proper place in the partial data considered so far. An outline of the algorithm is as follows:

Let the n data elements be stored in a array list[ ]. Then,

for (cur = 1; cur < n ; cur++)

move all elements list [j] greater than data [cur] by one position;

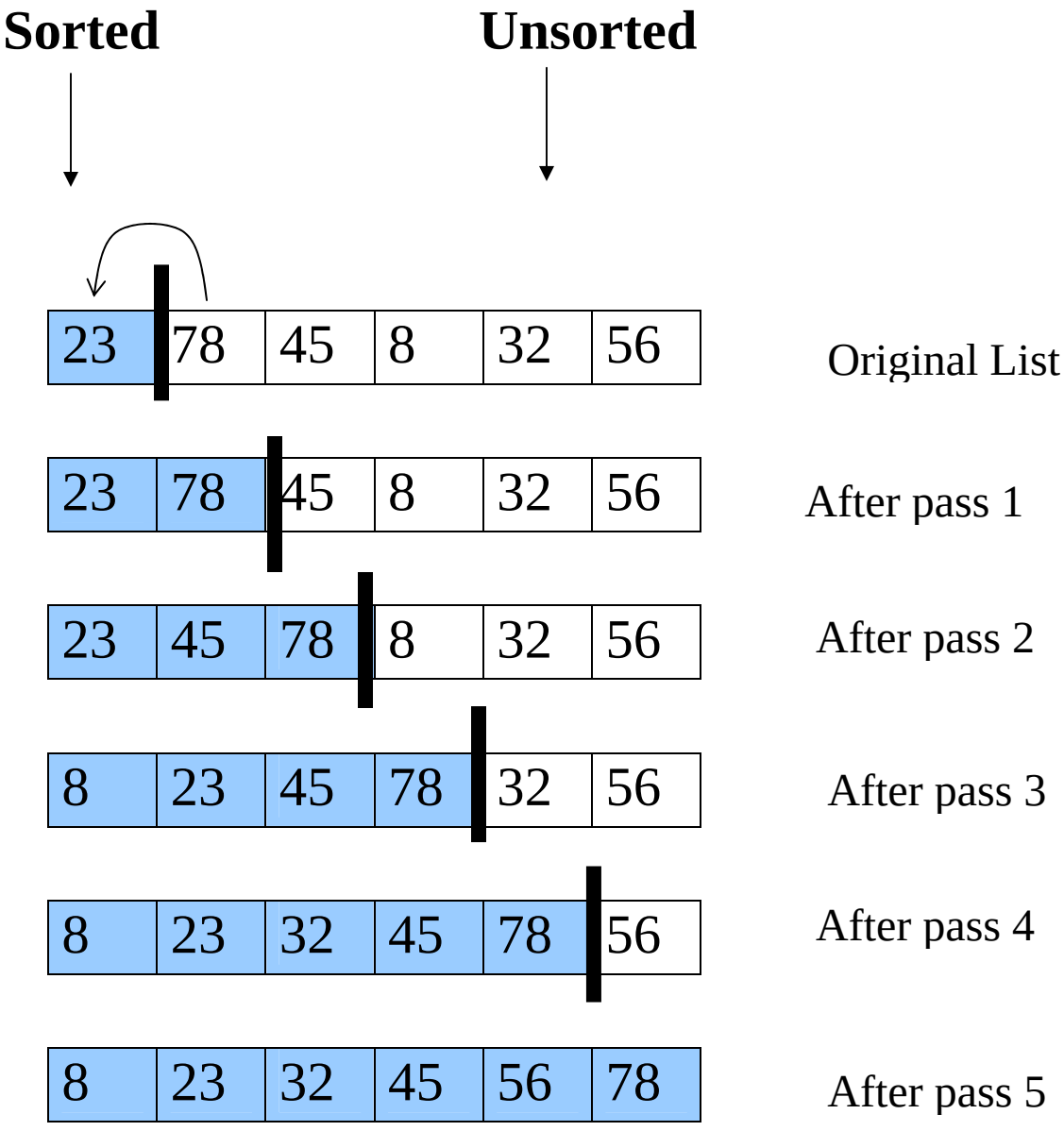
place data [cur] in its proper position.

Note that the sorting is restricted only to fraction of the array in each iteraion.

The list is divided into two parts: sorted and unsorted.

⇨ In each pass, the first element of the unsorted part is picked up, transferred to the sorted sublist, and inserted at the appropriate place. A list of *n* elements will take at most *n-1* passes to sort the data.

Insertion Sort Example



Insertion Sort Algorithm

```
/* With each pass, first element in unsorted
   sublist is inserted into sorted sublist.

*/

void insertionSort(int list[], int n)
{
   int cur, located, temp, j;

   for (cur = 1; cur <= n ; cur++){
     located = 0;
     temp = list[cur];
     for ( j = cur - 1; j >= 0 && !located;)
        if(temp < list[ j ]){
           list[j + 1]= list[ j ];
           j--;
        }
        else
           located = 1;
     list[j + 1] = temp;
   }
   return;
}
```

An advantage of this method is that it sorts the array only when it is really necessary.

If the array is already in order, no moves are performed. However, it overlooks the fact that the elements may already be in their proper positions . When an item is to be inserted, all elements greater than this have to be moved. There may be large number of redundant moves, as an element (properly located) may be moved , but later brought back to its position.

The best case is when the data are already in order. Only one comparison is made for each position, so the comparison complexity is O(n), and the data movement is 2n – 1 , i.e. it is O(n).

The worst case is when the data are in reverse order. Each data element is to be moved to new position and for that each of the other elements have to be shifted. This works out to be complexity of $O(n^2)$.

What happens when the elements are in **random order**? Does the over complexity is nearer to the best case , or nearer to the worst case? It turns out that both number of comparisons and movements turn out to be closer to the worst case .

Selection Sort

Selection sort is an attempt to localize the exchanges of array elements by finding a misplaced element first and putting it in its final place.

⇨ The list is divided into two sublists, *sorted* and *unsorted*, which are divided by an imaginary wall.

⇨ We select the smallest element from the unsorted sublist and swap it with the element at the beginning of the unsorted data.

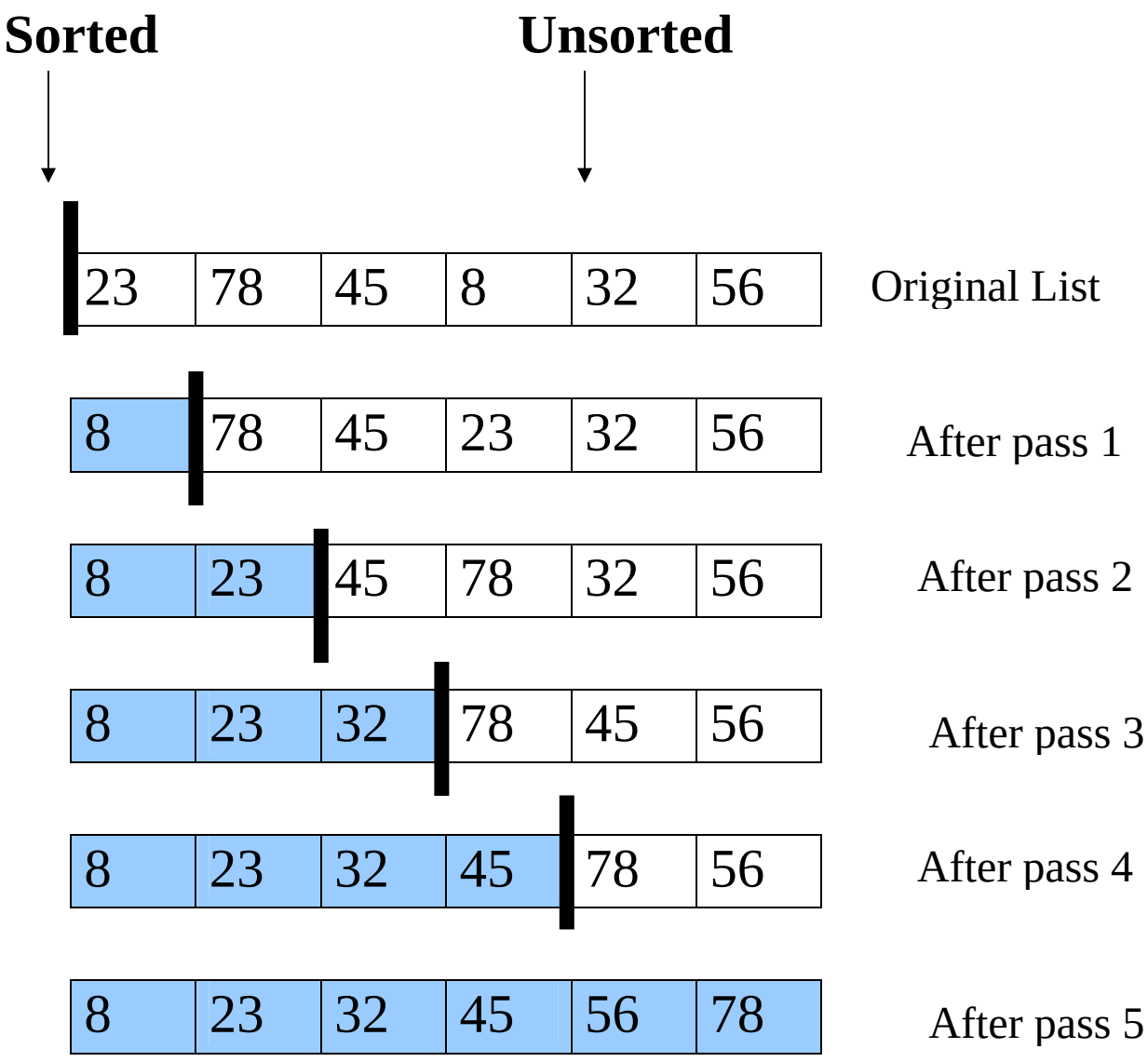
⇨ Then the smallest value among the remaining elements is selected and put in the second position and so on.

⇨ After each selection and swapping, the imaginary wall between the two sublists move one element ahead, increasing the number of sorted elements and decreasing the number of unsorted ones.

⇨ Each time we move one element from the unsorted sublist to the sorted sublist, we say that we have completed a sort pass.

⇨ A list of *n* elements requires *n-1* passes to completely rearrange the data.

Selection Sort Example



Selection Sort Algorithm

```
/* Sorts by selecting smallest element in
unsorted
   portion of array and exchanging it with
element
   at the beginning of the unsorted list.
*/

void selectionSort(int list[], int n)
{
   int cur, j, smallest, tempData;

   for (cur = 0; cur <= n; cur ++){
      smallest = cur;

       for ( j = cur +1; j <= n ; j++)
         if(list[ j ] < list[smallest])
            smallest =  j ;

      // Smallest selected; swap with current
element

      tempData  = list[cur];
      list[cur] = list[smallest];
      list[smallest] = tempData;
   }
}
```

The outer loop is executed n times, and the inner loop iterates
j = (n –1 – cur) times.

So it is a n times summation of j. Thus the order turns out to be $O(n^2)$.

The number of comparisons remain the same in all cases.

There may be some saving in terms of data movements (swappings).

Bubble Sort

Imagine all elements are objects of various sizes and are placed in a vertical column. If the objects are allowed to float, then the smallest element will bubble its way to the top. This is how the algorithm gets its name.

⇨ The list scanned from the bottom up, and two adjacent elements are interchanged if they are found to be out of order with respect to each other.

Then next pair of adjacent elements are considered and so on.

⇨ Thus the smallest element is bubbled from the unsorted list to the top of the array.

⇨ After that, the wall moves one element ahead, increasing the number of sorted elements and decreasing the number of unsorted ones.

⇨ Each time an element moves from the unsorted part to the sorted part one sort pass is completed.

⇨ Given a list of n elements, bubble sort requires up to n-1 passes to sort the data.

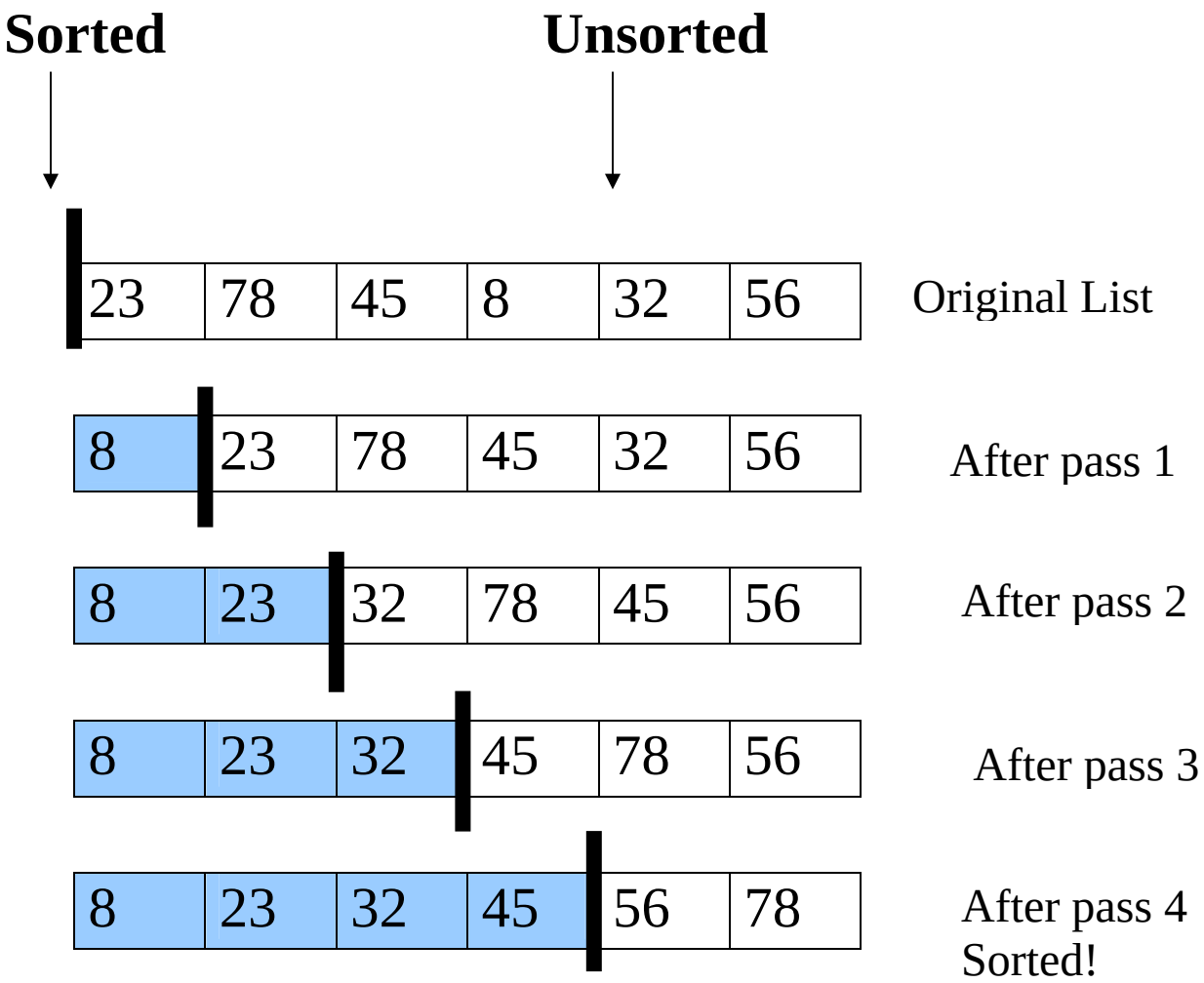
⇨ Bubble sort was originally written to “bubble up” the highest element in the list. From an efficiency point of

view it makes no difference whether the high element is bubbled or the low element is bubbled.

Bubble Sort Example

Trace of 1st pass of Bubble Sort Example:

23	78	45	8	32	56
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


Sorted
Unsorted
23 78 45 8 32 56
Original List
8 23 78 45 32 56
After pass 1
8 23 32 78 45 56
After pass 2
8 23 32 45 78 56
After pass 3
8 23 32 45 56 78
After pass 4
Sorted!

Bubble Sort Algorithm

```
/* Sorts list using bubble sort. Adjacent
elements
   are compared and exchanged until list is
   completely ordered.

*/
void bubbleSort(int list[], int n)
{
   int cur, j, temp;
   for (cur = 0; cur <= n; cur++){
      for ( j = n; j > cur; j--)
         if(list[j ] < list[ j - 1]){
            temp = list[ j ];
            list[ j ] = list[ j – 1];
            list[ j -1] = temp;
         }
   }
   return;
}
```

What is the time complexity of the Bubble sort algorithm?

The number of comparisons for the worst case (when the array is in the reverse order sorted) equals the number of iterations for the inner loop ,i.e. $O(n^2)$. How many comparisons are to be made for the best case? Well, it turns out that the number of comparisons remain the same.

What about the number of swaps? In the worst case, the number of movements is 3 n(n – 1)/2. In the best case, when all elements are ordered, no swaps are necessary. If it is a randomly ordered array, then number of movements are around half of the worst case figure.

Compare it with insertion sort. Here each element has to be bubbled one step at a time. It does not go directly to its proper place as in insertion sort. It could be said that insertion sort is twice as fast as the bubble sort , for the average case., because bubble sort makes approximately twice as many comparisons .

In summary, we can say that bubble sort, insertion sort and selection sort are not very efficient methods, as the complexity is $O(n^2)$.