# Sorting Algorithms

# Sorting

- ***Sorting*** is a process that organizes a collection of data into either ascending or descending order.
- An ***internal sort*** requires that the collection of data fit entirely in the computer's main memory.
- We can use an ***external sort*** when the collection of data cannot fit in the computer's main memory all at once but must reside in secondary storage such as on a disk.
- We will analyze only internal sorting algorithms.
- Any significant amount of computer output is generally arranged in some sorted order so that it can be interpreted.
- Sorting also has indirect uses. An initial sort of the data can significantly enhance the performance of an algorithm.
- Majority of programming projects use a sort somewhere, and in many cases, the sorting cost determines the running time.
- A comparison-based sorting algorithm makes ordering decisions only on the basis of comparisons.

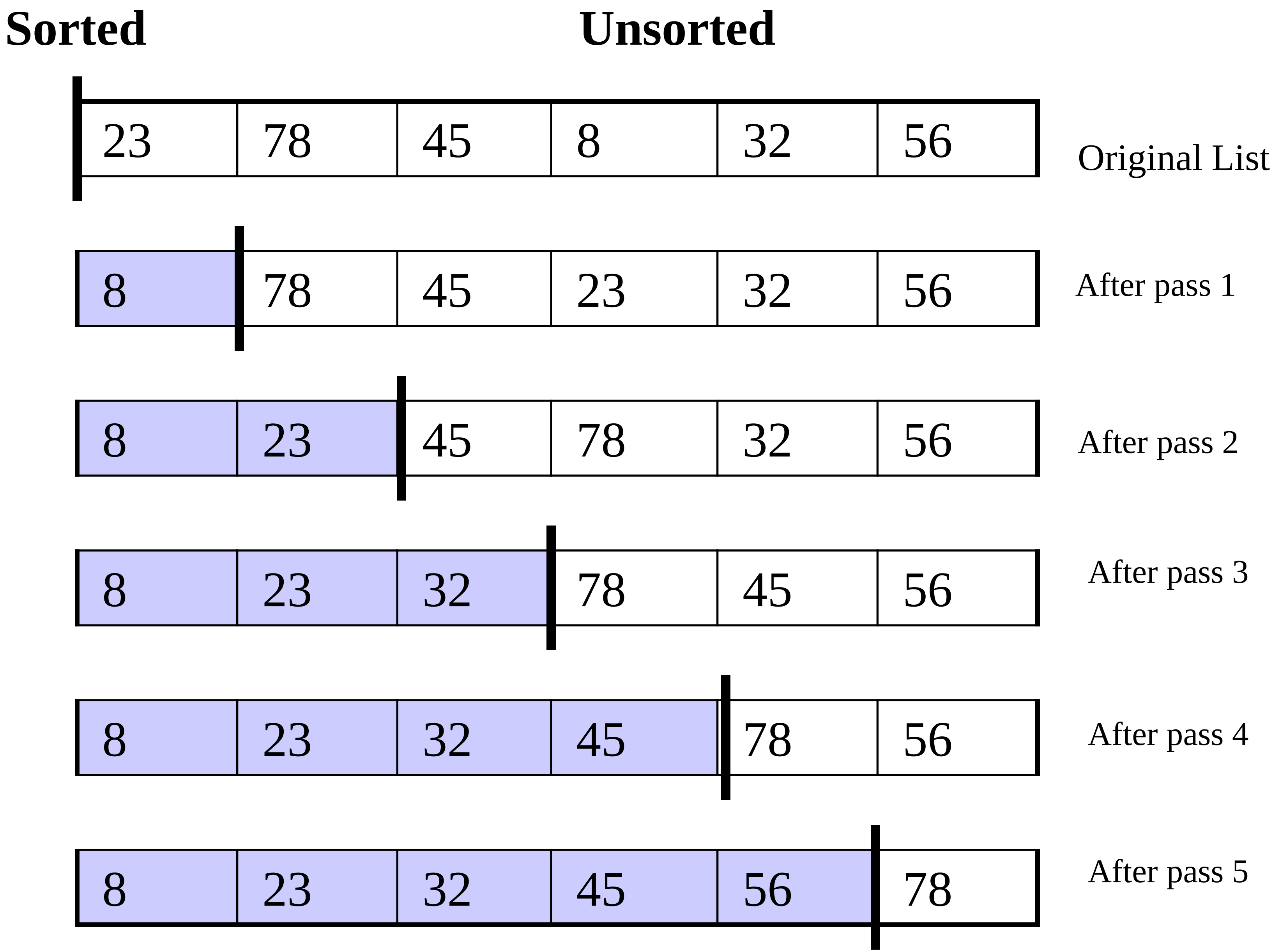
# Sorting Algorithms

- There are many sorting algorithms, such as:
  - Selection Sort
  - Insertion Sort
  - Bubble Sort
  - Merge Sort
  - Quick Sort

- The first three are the foundations for faster and more efficient algorithms.

# Selection Sort

- The list is divided into two sublists, *sorted* and *unsorted*, which are divided by an imaginary wall.
- We find the smallest element from the unsorted sublist and swap it with the element at the beginning of the unsorted data.
- 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.


Sorted
Unsorted
23 78 45 8 32 56
Original List
8 78 45 23 32 56
After pass 1
8 23 45 78 32 56
After pass 2
8 23 32 78 45 56
After pass 3
8 23 32 45 78 56
After pass 4
8 23 32 45 56 78
After pass 5

# Selection Sort (cont.)

```
template <class Item>
void selectionSort( Item a[], int n) {
  for (int i = 0; i < n-1; i++) {
    int min = i;
    for (int j = i+1; j < n; j++)
       if (a[j] < a[min]) min = j;
    swap(a[i], a[min]);
  }
}

template < class Object>
void swap( Object &lhs, Object &rhs )
{
  Object tmp = lhs;
  lhs = rhs;
  rhs = tmp;
}
```

# Selection Sort -- Analysis

- In general, we compare keys and move items (or exchange items) in a sorting algorithm (which uses key comparisons).

➔ **So, to analyze a sorting algorithm we should count the number of key comparisons and the number of moves.**

  - Ignoring other operations does not affect our final result.

- In selectionSort function, the outer for loop executes n-1 times.
- We invoke swap function once at each iteration.

➔ Total Swaps: n-1

➔ Total Moves: 3*(n-1) (Each swap has three moves)

# Selection Sort – Analysis (cont.)

- The inner for loop executes the size of the unsorted part minus 1 (from 1 to n-1), and in each iteration we make one key comparison.

  ➔ # of key comparisons = 1+2+...+n-1 = n*(n-1)/2

  ➔ **So, Selection sort is O($n^2$)**

- The best case, the worst case, and the average case of the selection sort algorithm are same. ➔ all of them are **O($n^2$)**
  - This means that the behavior of the selection sort algorithm does not depend on the initial organization of data.
  - Since O($n^2$) grows so rapidly, the selection sort algorithm is appropriate only for small n.
  - Although the selection sort algorithm requires O($n^2$) key comparisons, it only requires O(n) moves.
  - A selection sort could be a good choice if data moves are costly but key comparisons are not costly (short keys, long records).

# Comparison of *N*, *logN* and *N*$^2$

| N | O(LogN) | O($N^2$) |
|---|---|---|
| 16 | 4 | 256 |
| 64 | 6 | 4K |
| 256 | 8 | 64K |
| 1,024 | 10 | 1M |
| 16,384 | 14 | 256M |
| 131,072 | 17 | 16G |
| 262,144 | 18 | 6.87E+10 |
| 524,288 | 19 | 2.74E+11 |
| 1,048,576 | 20 | 1.09E+12 |
| 1,073,741,824 | 30 | 1.15E+18 |