Fundamentals of Data Structure

Mahesh Shirole

VJTI, Mumbai-19

Slides are prepared from

1. Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

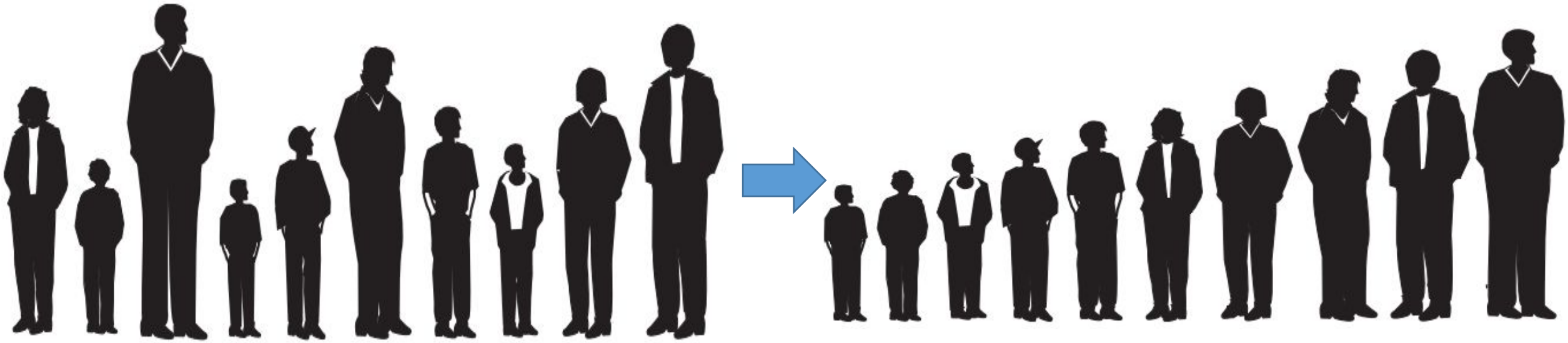
2.Data Structures and Algorithms in Java, by Robert Lafore, Second Edition, Sams Publishing

Why we need Sorting?

- In many application we need to arrange data in specific order.
- For example, we need to arrange
  - names in alphabetical order,
  - students by grade,
  - customers by ZIP code,
  - home sales by price,
  - cities in order of increasing population,
  - countries by GDP,
  - stars by magnitude, and so on.
- Sorting data may also be a preliminary step to searching it.

Sorting

- Because sorting is so important and potentially so timeconsuming.
- It has been the subject of extensive research in computer science and some very sophisticated methods have been developed.
- In this class, we look at three of the simpler algorithms: the bubble sort, the selection sort, and the insertion sort.
- Imagine that your kids-league baseball team, arrange the players in order of increasing height.

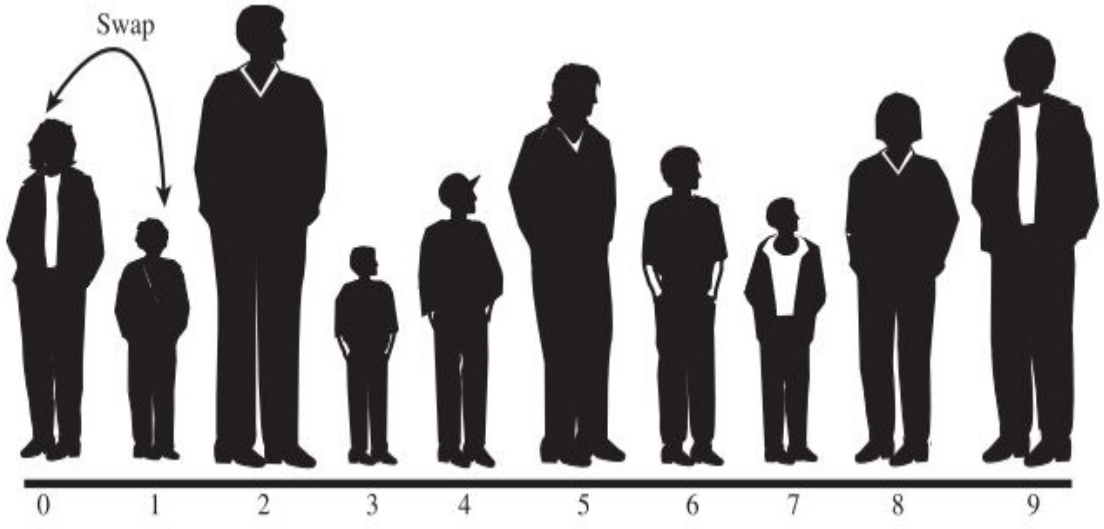


How computer can do this task?

- A computer program isn’t able to glance over the data in this way.
- It can compare only two players at one time because that’s how the comparison operators work.
- The algorithms we discuss involve two steps, executed over and over until the data is sorted:
  - Compare two items.
  - Swap two items, or copy one item
- However, each algorithm handles the details in a different way.

Bubble Sort

- The bubble sort is very slow, but it's conceptually the simplest of the sorting algorithms
- The bubble sort routine works like this
  1. Compare two players (start from left zero position)
  2. If the one on the left is taller, swap them
  3. Move one position right



Bubble Sort

- As the algorithm progresses, the biggest items "bubble up" to the top end of the array
- After this first pass through all the data, you've made N-1 Comparisons and somewhere between 0 and N-1 swaps
- Now you go back and start another pass from the left end of the line
- However, this time you can stop one player short of the end of the line, at position N-2, because you know the last position
- You continue this process until all the players are in order

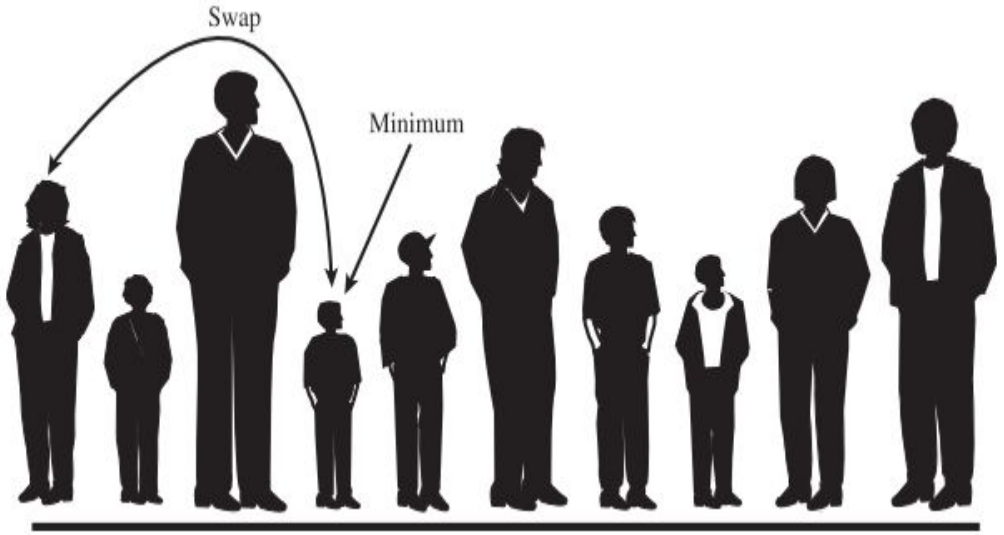
Bubble Sort Code

```
1.  public void bubbleSort() {
2.      int out;                    //outerLoopCount,
3.      int in;                     //innerLoopCount;
4.      for(out=nElems-1; out>1; out--)     // outer loop (backward)
        for(in=0; in<out; in++)             // inner loop (forward)
5.          if( a[in] > a[in+1] )           // out of order?
6.              swap(in, in+1);  //swap them
7.  }                                   // end bubbleSort()
```

Number of operations are $O(n^2)$, where 'n' is number of elements in the

Selection Sort

- The selection sort improves on the bubble sort by reducing the number of swaps necessary from $O(n^2)$ to $O(n)$
- The selection sort routine works like this
  1. Pass through all the players and picking the shortest one
  2. This shortest player is then swapped with the player on the left end of the line, at position where we started (zero)
  3. Move one position right



Selection Sort

- Notice that in this algorithm the sorted players accumulate on the left (lower indices), whereas in the bubble sort they accumulated on the right.
- Unfortunately, the number of comparisons remains $O(n^2)$

Selection Sort Java Code

```
1.  public void selectionSort() {
2.      int out, in, min;
3.      for(out=0; out<nElems-1; out++) {       // outer loop
4.          min = out;                          // minimum
5.          for(in=out+1; in<nElems; in++)      // inner loop
6.              if(a[in] < a[min] )             // if min greater,
7.                  min = in;                   // we have a new min
8.          swap(out, min);
9.      }                                       // end for outer loop
10.  }                                          // end selectionSort()
```

The selection sort performs the same number of comparisons as the bubble sort: N*(N-1)/2.

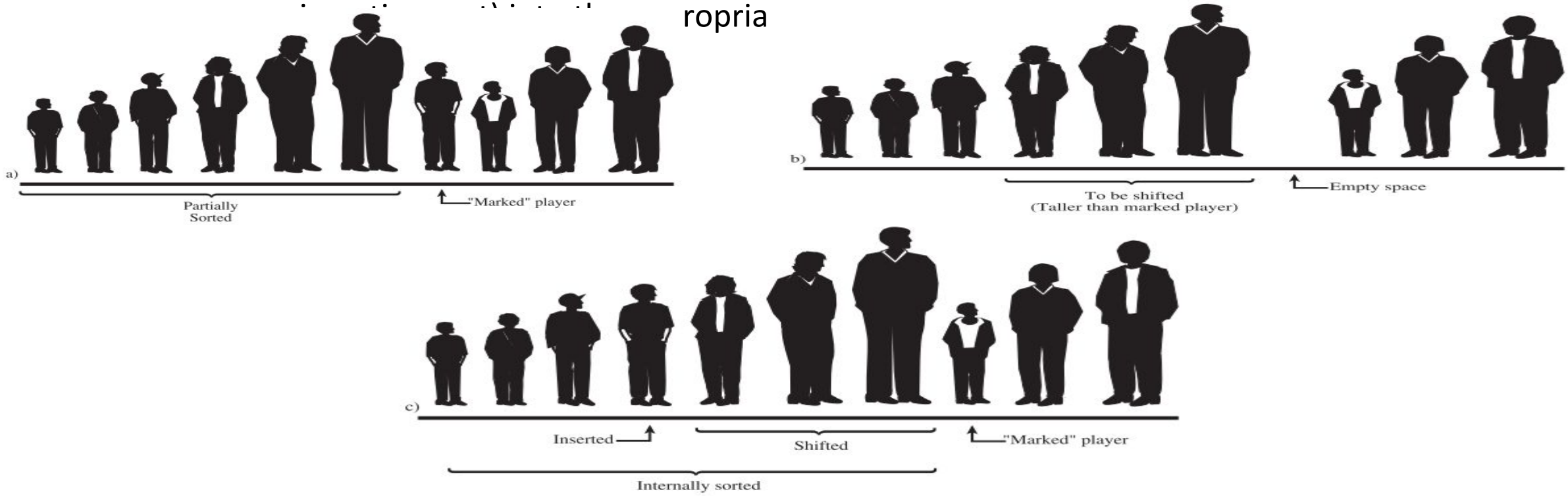
Insertion Sort

- Insertion sort inserts an element at its appropriate position.
- Insertion sort executes in $O(n^2)$ time, but it's about twice as fast as the bubble sort and somewhat faster than the selection sort in normal situations.
- It's easier to think about the insertion sort if we begin in the middle of the process, when the team is half sorted.
- At this point there's an imaginary marker somewhere in the middle of the line.
- The players to the left of this marker are *partially sorted*.
- However, the players aren't necessarily in their final positions because they may still need to be moved when previously unsorted players are inserted between them.

Insertion Sort

- The selection sort routine works like this
  1. An element is marked for sorting, who's right-side elements are unsorted
  2. Find an appropriate place for marked player in the (partially) sorted group. (We remove marked player to make room)
  3. Now we shift the sorted elements to make room for marked player. Insert marked player at its appropriate place. The largest sorted element moves into the marked elemet's position.
  4. This process is repeated until all the unsorted elements have been inserted (hence the

ropria



Insertion Sort Java Code

```
1.  public void insertionSort() {
2.    int in, out;
3.    for(out=1; out<nElems; out++) {          // out is dividing line
4.          long temp = a[out];              // remove marked item
5.          in = out;                        // start shifts at out
6.          while(in>0 && a[in-1] >= temp){  // until one is smaller,
7.                a[in] = a[in-1];           // shift item right,
8.                --in;                      // go left one position
9.          }
10.   a[in] = temp;                          // insert marked item
11.   } // end for
12.  } // end insertionSort()
```

The insertion sort runs in $O(n^2)$ time for random data. In best case, algorithm runs in O(n) time