4-SIMPLE SORTING ALGORITHMS

FIGURE 3.1 The unordered baseball team.

Sorting Algorithm

FIGURE 3.2 The ordered baseball team.

Simple Sorting Algorithms

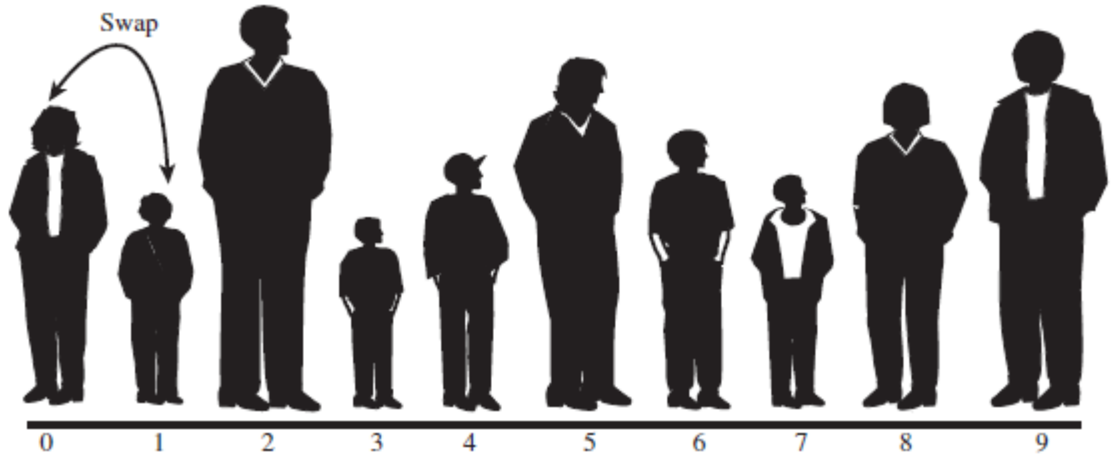
- We will introduce a number of simple sorting algorithms, which are:
  - Bubble Sort
  - Selection Sort
  - Insertion Sort

- For each sorting algorithm, we will discuss
  - How it works
  - Complexity
  - Implementation in Java (some of it will be given as an assignment)

1. Bubble sort

- Compare each element (except the last one) with its neighbor to the right
  - If they are out of order, swap them
  - This puts the largest element at the very end
  - The last element is now in the correct and final place
- Compare each element (except the last *two*) with its neighbor to the right
  - If they are out of order, swap them
  - This puts the second largest element next to last
  - The last two elements are now in their correct and final places
- Compare each element (except the last *three*) with its neighbor to the right
  - Continue as above until you have no unsorted elements on the left

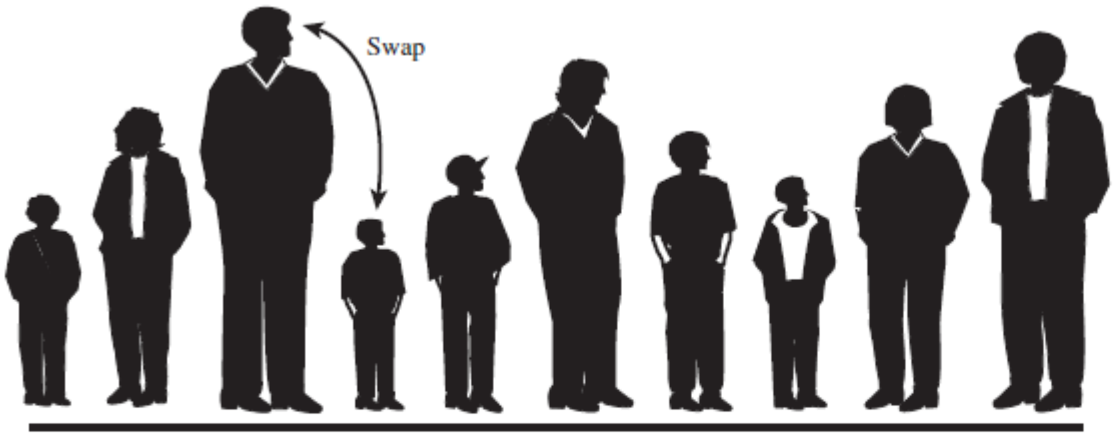
Step 1



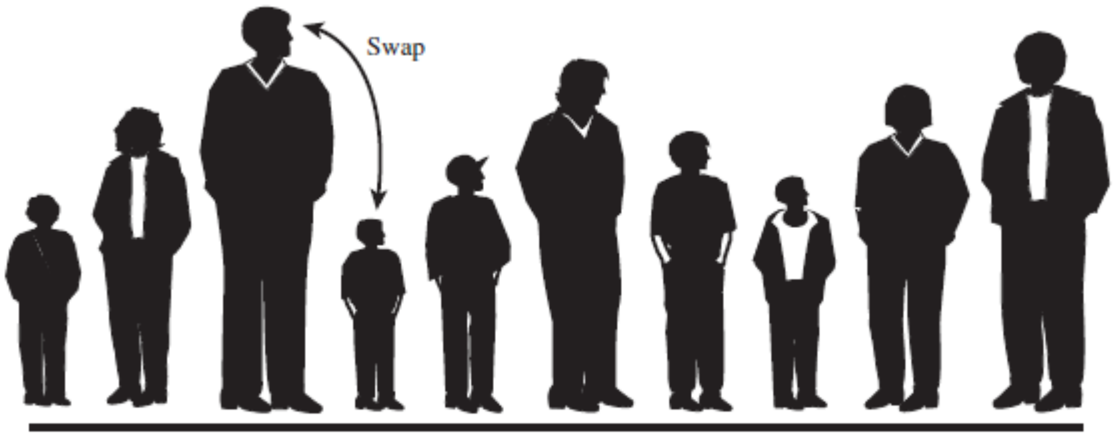
Step 2

Step 2

Step 3



Step 3



Step 4

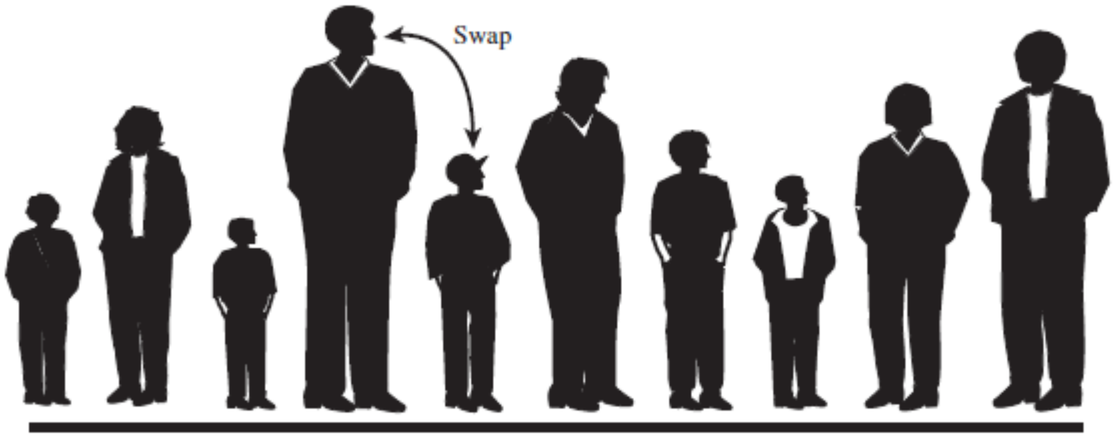
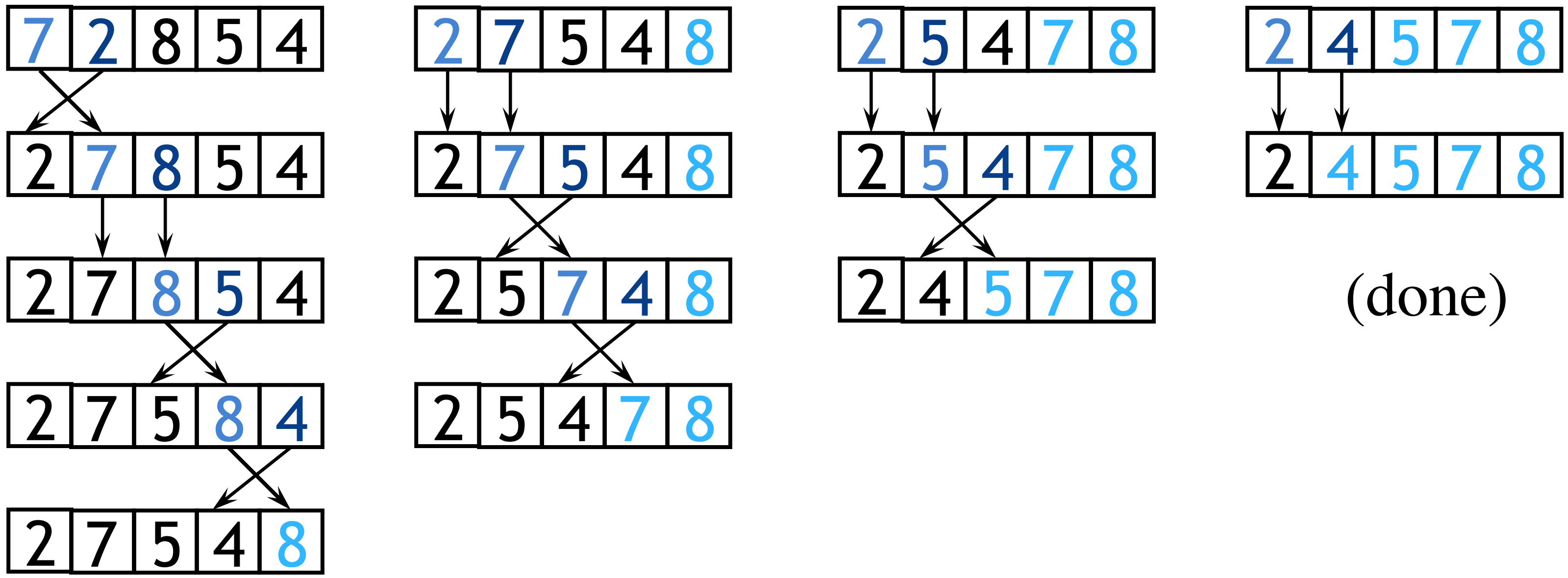


FIGURE 3.4 Bubble sort: the end of the first pass.

1.1 Example of bubble sort



See Workshop applet on BubbleSort

1.2 Code for bubble sort

```
public static void bubbleSort(int[] a) {
  int outer, inner;
  for (outer = a.length - 1; outer > 0; outer--) {  // counting down
    for (inner = 0; inner < outer; inner++) {        // bubbling up
      if (a[inner] > a[inner + 1]) {  // if out of order...
        int temp = a[inner];          // ...then swap
        a[inner] = a[inner + 1];
        a[inner + 1] = temp;
      }
    }
  }
}
```

1.3 Analysis of bubble sort

- ```
  for (outer = a.length - 1; outer > 0; outer--) {
    for (inner = 0; inner < outer; inner++) {
      if (a[inner] > a[inner + 1]) {
        // code for swap omitted
  } } }
  ```
- Let n = a.length = size of the array
- The outer loop is executed n-1 times (call it n, that's close enough)
- Each time the outer loop is executed, the inner loop is executed
  - Inner loop executes n-1 times at first, linearly dropping to just once
  - On average, inner loop executes about n/2 times for each execution of the outer loop
  - In the inner loop, the comparison is always done (constant time), the swap might be done (also constant time)
- Result is $n * n/2 * k$, that is, $O(n^2/2 * k) = O(n^2)$
  ```

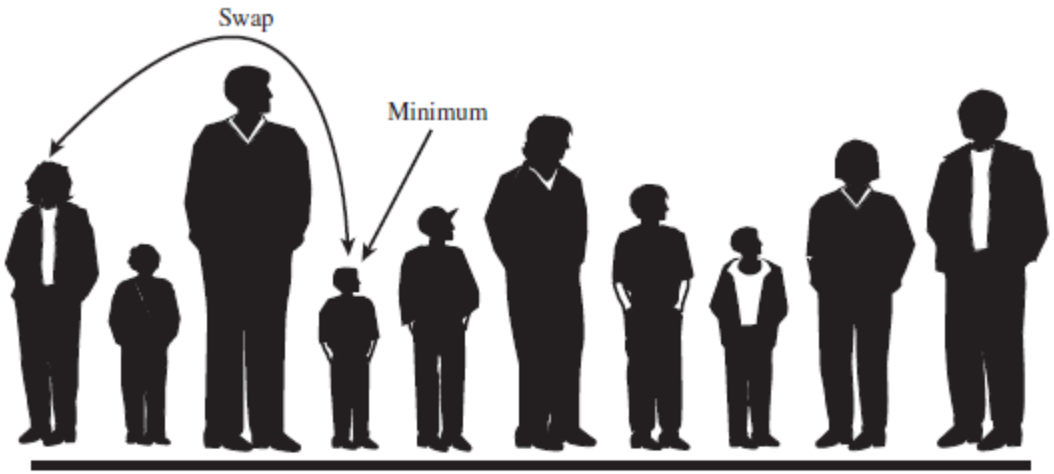
1.4 Loop invariants

- You run a loop in order to change things
- Oddly enough, what is usually most important in understanding a loop is finding an invariant: that is, *a condition that doesn't change*
- In bubble sort, we put the largest elements at the end, and once we put them there, we don't move them again
  - The variable outer starts at the last index in the array and decreases to 0
  - Our invariant is: Every element to the right of outer is in the correct place
  - That is, for all j > outer, if i < j, then a[i] <= a[j]
  - When this is combined with the loop exit test, outer == 0, we know that *all* elements of the array are in the correct place

2. Selection sort

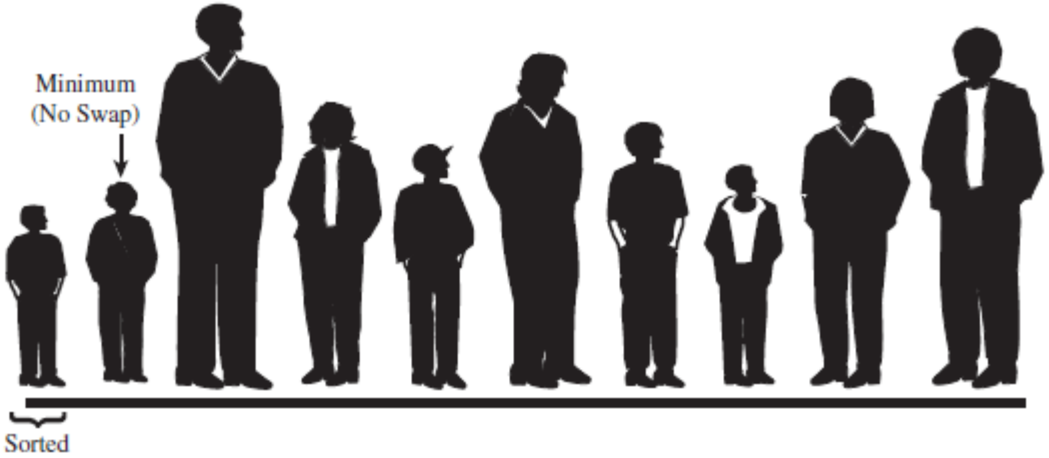
- Given an array of length n,
  - Search elements 0 through n-1 and select the smallest
    - Swap it with the element in location 0
  - Search elements 1 through n-1 and select the smallest
    - Swap it with the element in location 1
  - Search elements 2 through n-1 and select the smallest
    - Swap it with the element in location 2
  - Search elements 3 through n-1 and select the smallest
    - Swap it with the element in location 3
  - Continue in this fashion until there's nothing left to search

Step 1

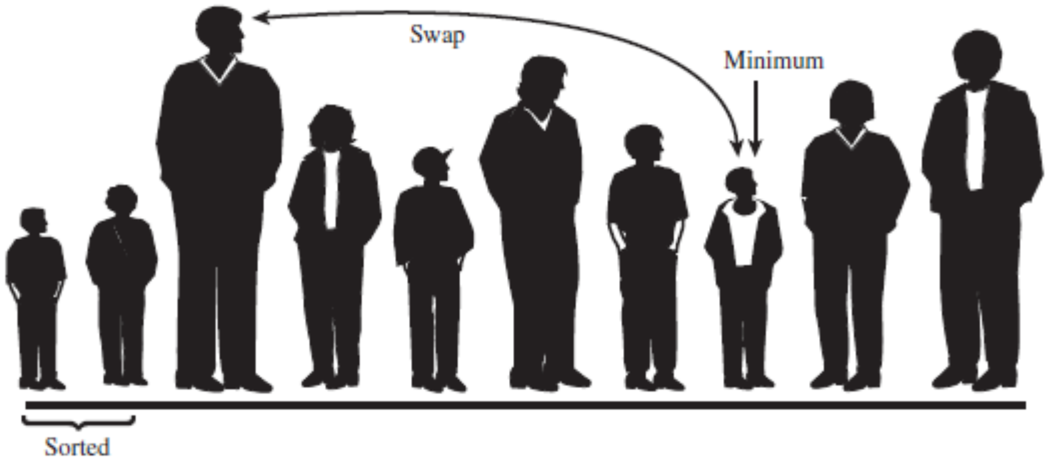


Step 2

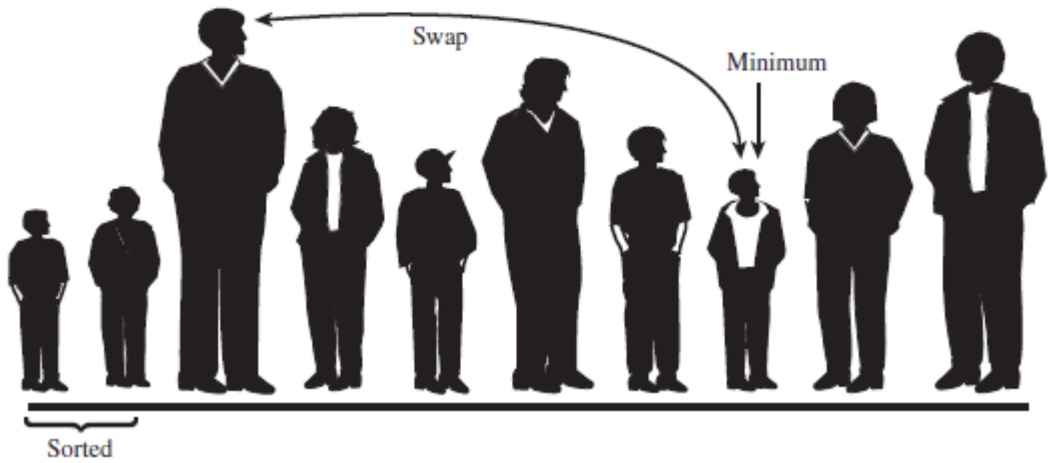
Step 2



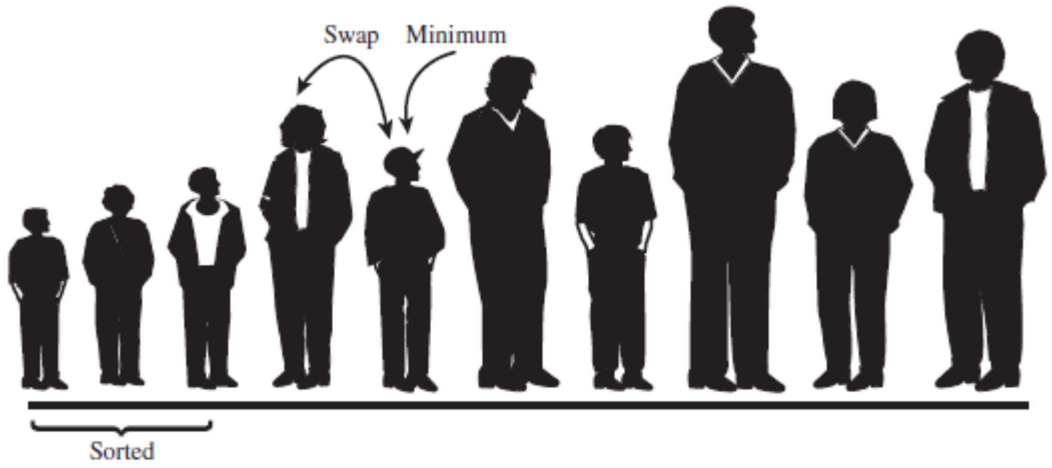
Step 3



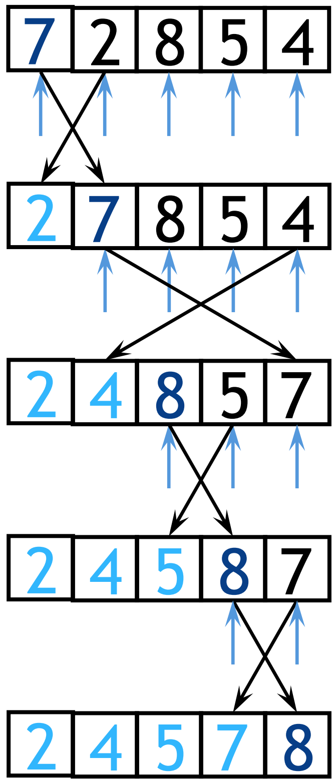
Step 3



Step 4



2.1 Example



See SelectSort Workshop Applet

2.2 Code for selection sort

```
public static void selectionSort(int[] a) {
        int inner, outer, min;
        for (outer = 0; outer < a.length - 1; outer++) {
            min = outer;
            for (inner = outer + 1; inner < a.length; inner++) {
                if (a[inner] < a[min]) {
                    min = inner;
                }
                // Invariant: for all i, if outer <= i <= inner, then a[min] <= a[i]
            }
            // a[min] is least among a[inner]..a[a.length - 1]
            int temp = a[outer];
            a[outer] = a[min];
            a[min] = temp;
            // Invariant: for all i <= outer, if i < j then a[i] <= a[j]
        }
    }
```

2.3 Analysis of selection sort

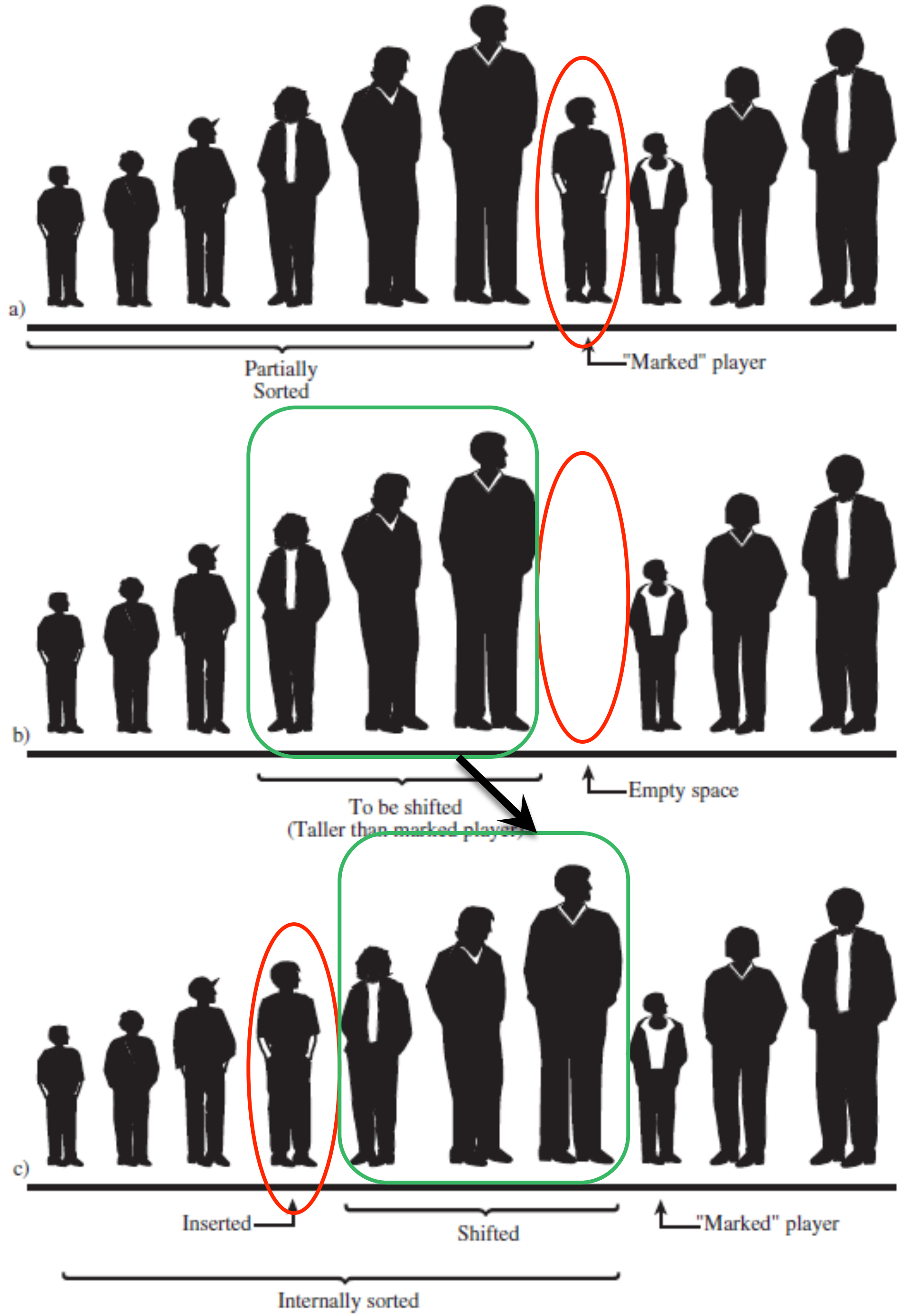
- The selection sort might swap an array element with itself--this is harmless, and not worth checking for
- Analysis:
  - The outer loop executes $n-1$ times
  - The inner loop executes about $n/2$ times on average (from $n-1$ to 1 time)
  - Work done in the inner loop is constant (swap two array elements)
  - Time required is roughly $(n-1)*(n/2)$
  - You should recognize this as $O(n^2)$

2.4 Loop Invariants for selection sort

- For the inner loop:
  - This loop searches through the array, incrementing inner from its initial value of outer+1 up to a.length-1
  - As the loop proceeds, min is set to the index of the smallest number found so far
  - Our invariant is:
    for all i such that outer <= i <= inner then a[min] <= a[i]
- For the outer (enclosing) loop:
  - The loop counts up from outer = 0
  - Each time through the loop, the minimum remaining value is put in a[outer]
  - Our invariant is:
    for all i <= outer, if i < j then a[i] <= a[j]

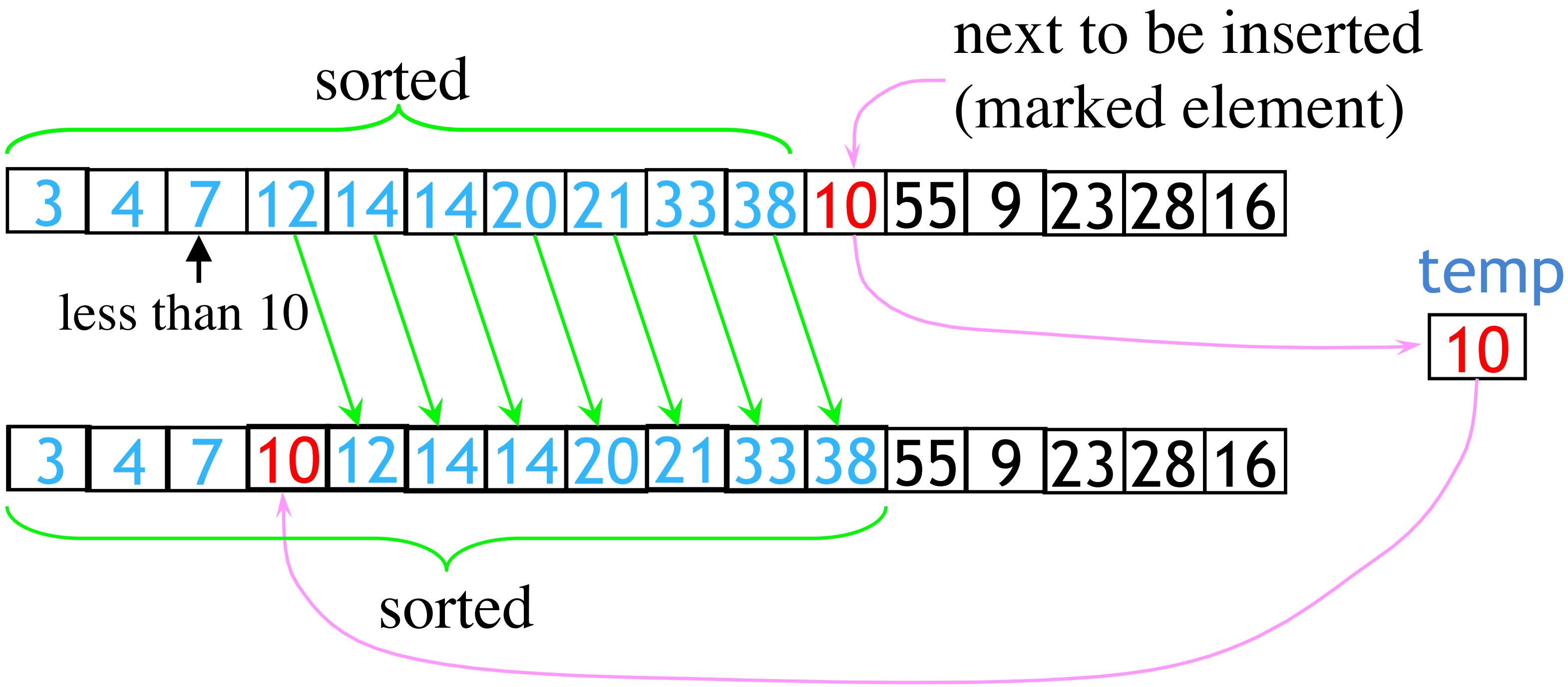
3. Insertion sort

- It's easier to think about the insertion sort if we begin in the middle of the process, when the list is half (partially) sorted.
- At this point there's an imaginary marker somewhere in the middle of the list.
- The elements to the left of this marker are *partially sorted*. This means that they are sorted among themselves; each one is taller than the person to his or her left.
- 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.



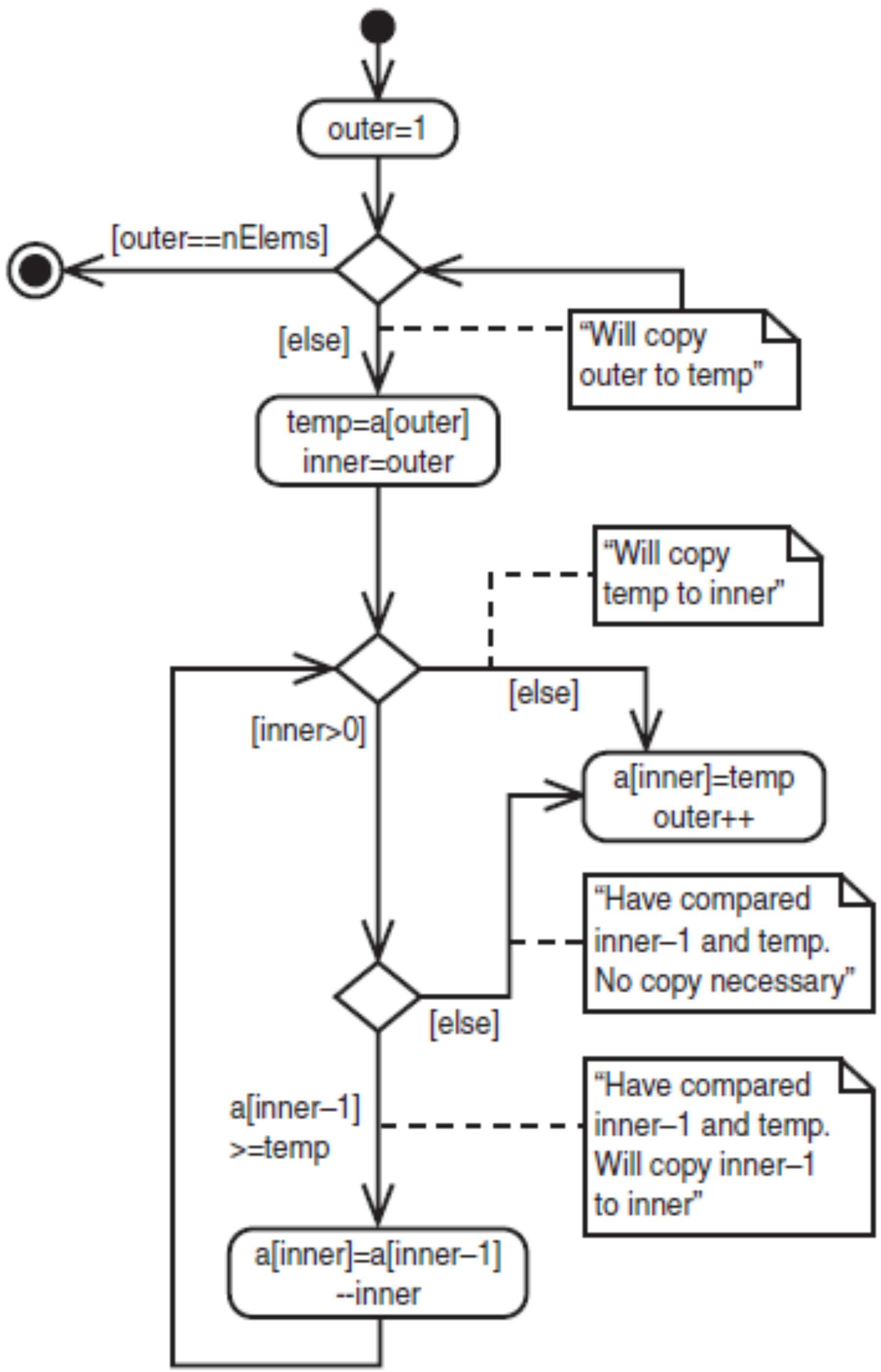
a)
Partially
Sorted
"Marked" player
b)
Empty space
To be shifted
(Taller than marked player)
c)
Inserted
Shifted
"Marked" player
Internally sorted

3.1 One step of insertion sort



See InsertionSort Workshop applet

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


outer=1
[outer==nElems]
[else]
"Will copy
outer to temp"
temp=a[outer]
inner=outer
"Will copy
temp to inner"
[else]
[inner>0]
a[inner]=temp
outer++
"Have compared
inner–1 and temp.
No copy necessary"
[else]
a[inner–1]
>=temp
"Have compared
inner–1 and temp.
Will copy inner–1
to inner"
a[inner]=a[inner–1]
--inner

3.2 Analysis of insertion sort

- The outer loop of insertion sort is:
  `for(out=1; out<nElems; out++) {…}`
- The invariant is that all the elements to the left of `out` are sorted with respect to one another
  - For all i < out, j < out, if i < j then a[i] <= a[j]
  - This does *not* mean they are all in their final correct place; the remaining array elements may need to be inserted
  - When we increase `out`, `a[out-1]` becomes to its left; we must keep the invariant true by inserting `a[out-1]` into its proper place
  - This means:
    - Finding the element's proper place
    - Making room for the inserted element (by shifting over other elements)
    - Inserting the element

3.2 Analysis of insertion sort

- How many comparisons and copies does this algorithm require?
- On the first pass, it compares a maximum of one item.
- On the second pass, it's a maximum of two items, and so on, up to a maximum of N-1 comparisons on the last pass.
- This is $1 + 2 + 3 + \ldots + N-1 = N*(N-1)/2$
- However, because on each pass an average of only half of the maximum number of items are actually compared before the insertion point is found, we can divide by 2, which gives $N*(N-1)/4$
- Discarding constants, we find that insertion sort is $O(N^2)$

Summary

- Bubble sort, selection sort, and insertion sort are all $O(n^2)$
- As we will see later, we can do much better than this with somewhat more complicated sorting algorithms
- Within $O(n^2)$,
  - Bubble sort is very slow, and should probably never be used for anything. It makes large number of comparisons and swaps.
  - Selection sort is intermediate in speed. It has less number of swaps but still large number of comparisons.
  - Insertion sort is usually the fastest of the three--in fact, for small arrays (say, 10 or 15 elements), insertion sort is faster than more complicated sorting algorithms. It makes less comparisons and makes copies which is faster than swaps.
- Selection sort and insertion sort are “good enough” for small arrays

The End