DATA STRUCTURE & ALGORITHMS

Sorting, Searching Algorithm

- **Sorting Algorithms.**
- **Searching Algorithms.**

SORTING ALGORITHMS.

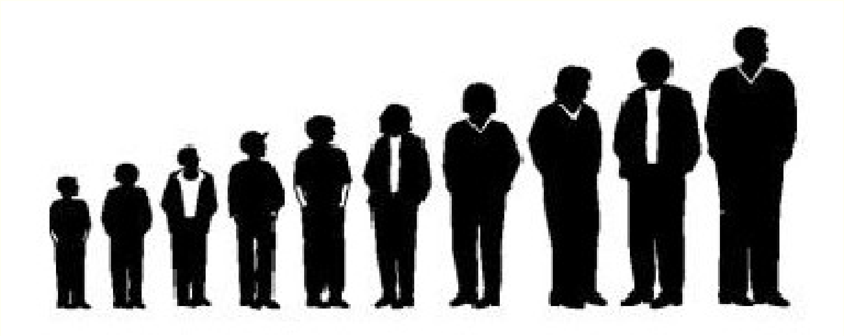
Sorting in Databases

- Many possibilities
  - Names in alphabetical order
  - Students by grade
  - Customers by zip code
  - Home sales by price
  - Cities by population

Example

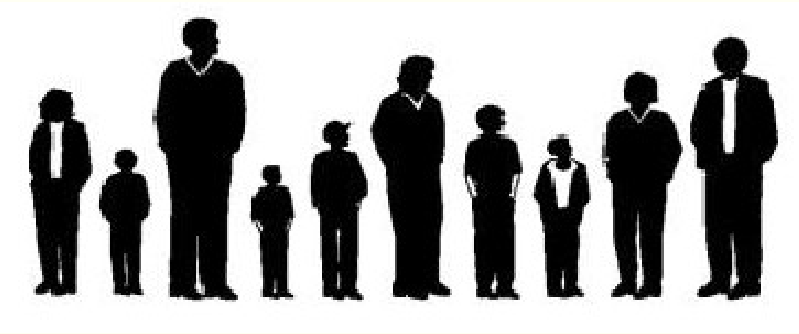
List sorted in ascending order:

20 25 30 40 55



List that is not sorted in ascending order:

20 25 15 40 12



Basic Sorting Algorithms

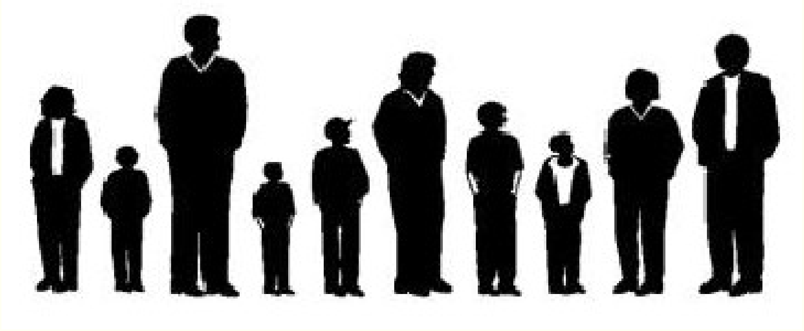
- Bubble
- Selection
- Insertion

- Although these are:
  - Simpler
  - Slower

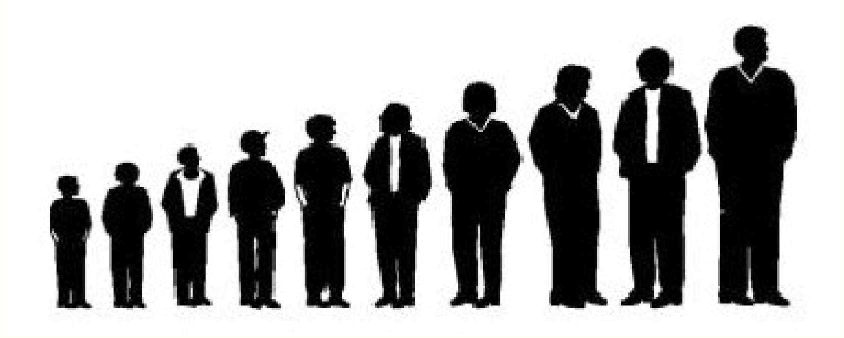
- They sometimes are better than advanced
- And sometimes the advanced methods build on them

Example

- Unordered:



- Ordered:



Simple Sorts

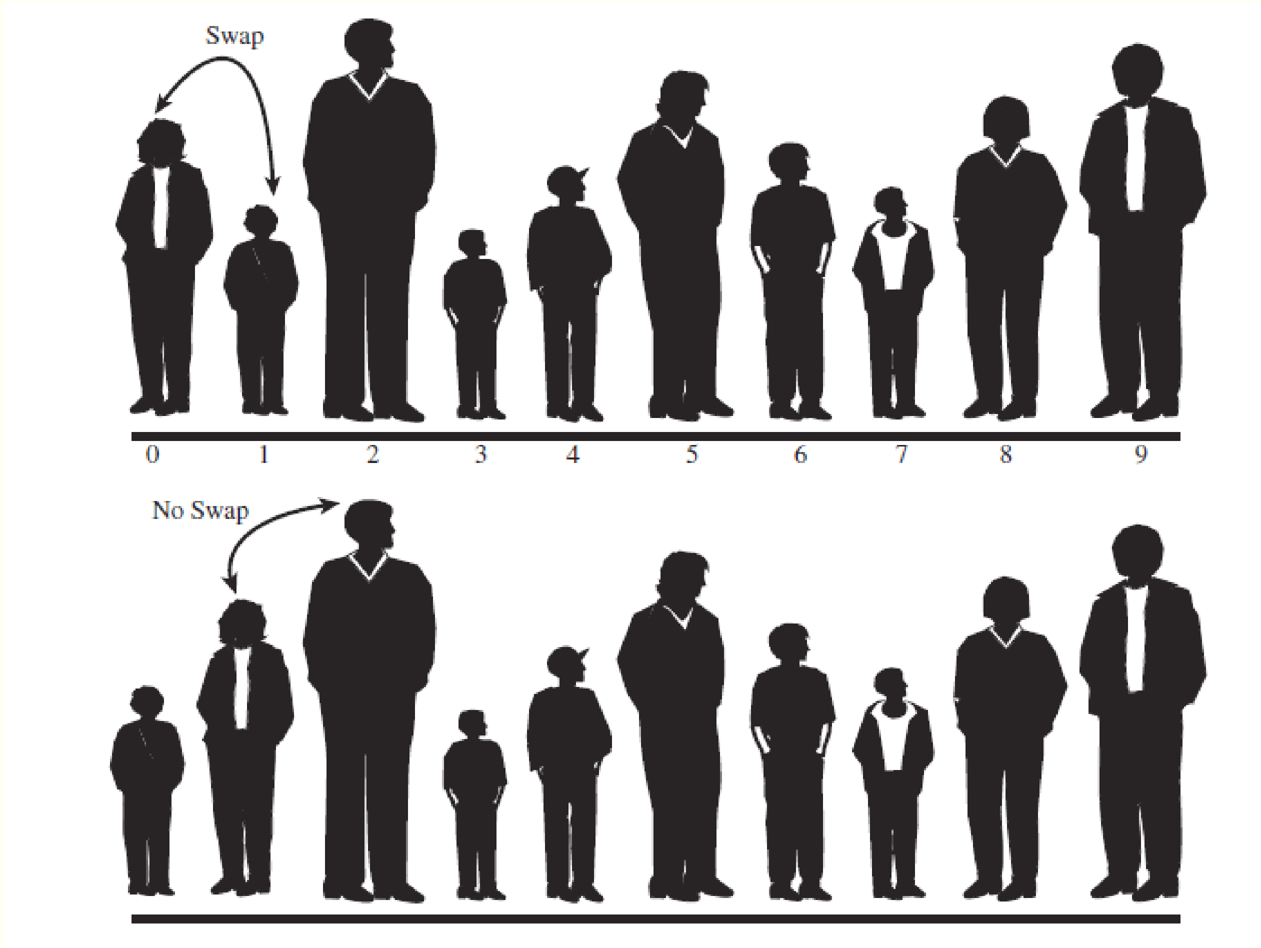
- All three algorithms involve two basic steps, which are executed repeatedly until the data is sorted
  - Compare two items
  - Either swap two items, or copy one item

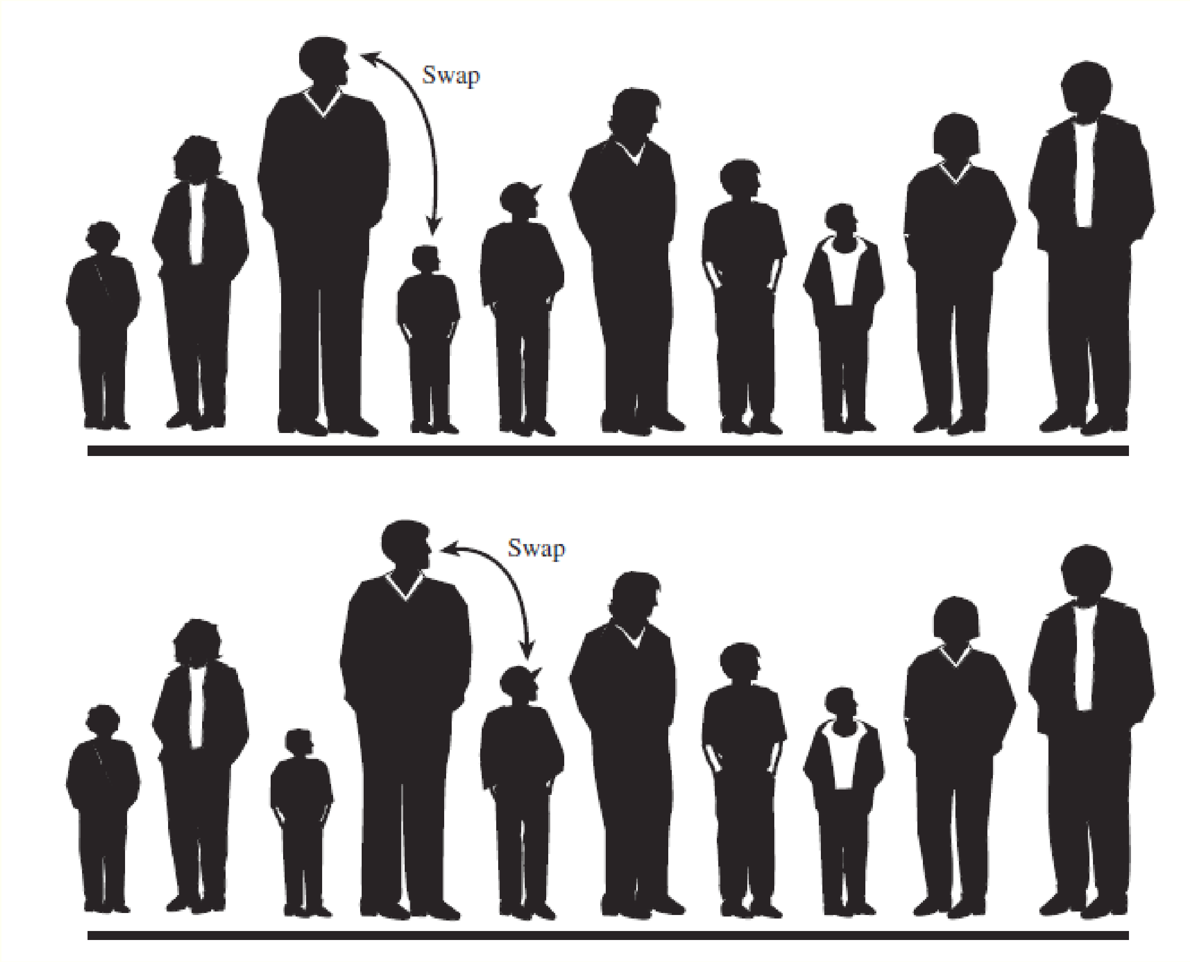
- They differ in the details and order of operations

Sort #1: **Bubble Sort**

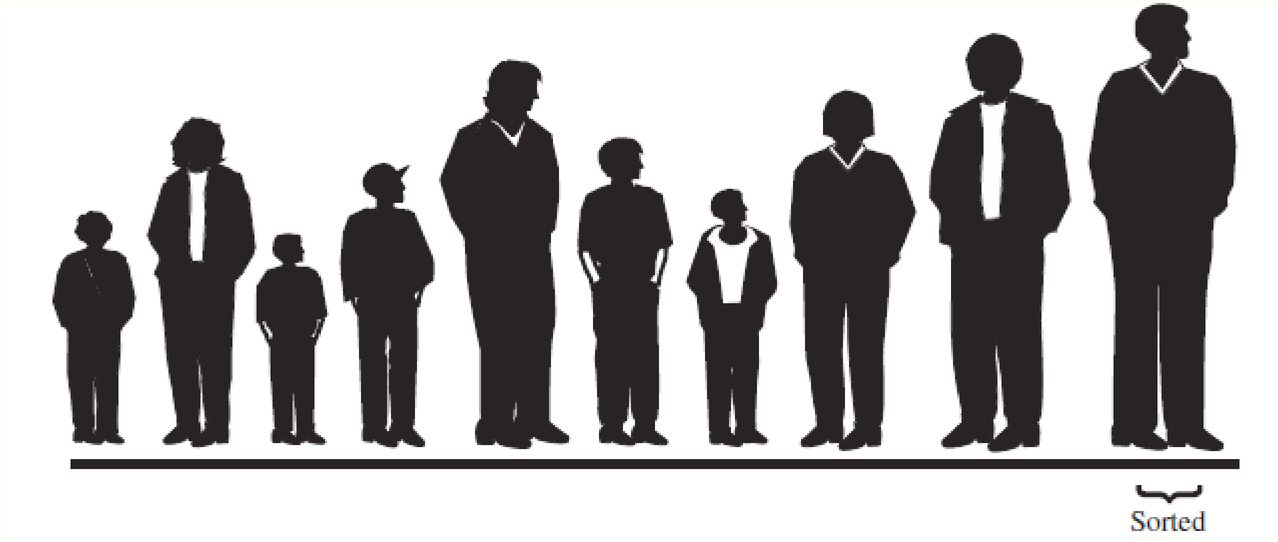
Bubble Sort

- **Bubble Sort on the Baseball Players**
- Here are the rules you’re following:
  1. Compare two players.
  2. If the one on the left is taller, swap them.
  3. Move one position right.


Swap
0
1
2
3
4
5
6
7
8
9
No Swap


Swap
Swap

4. When you reach the first sorted player, start over at the left end of the line.



Sort #2: Selection Sort

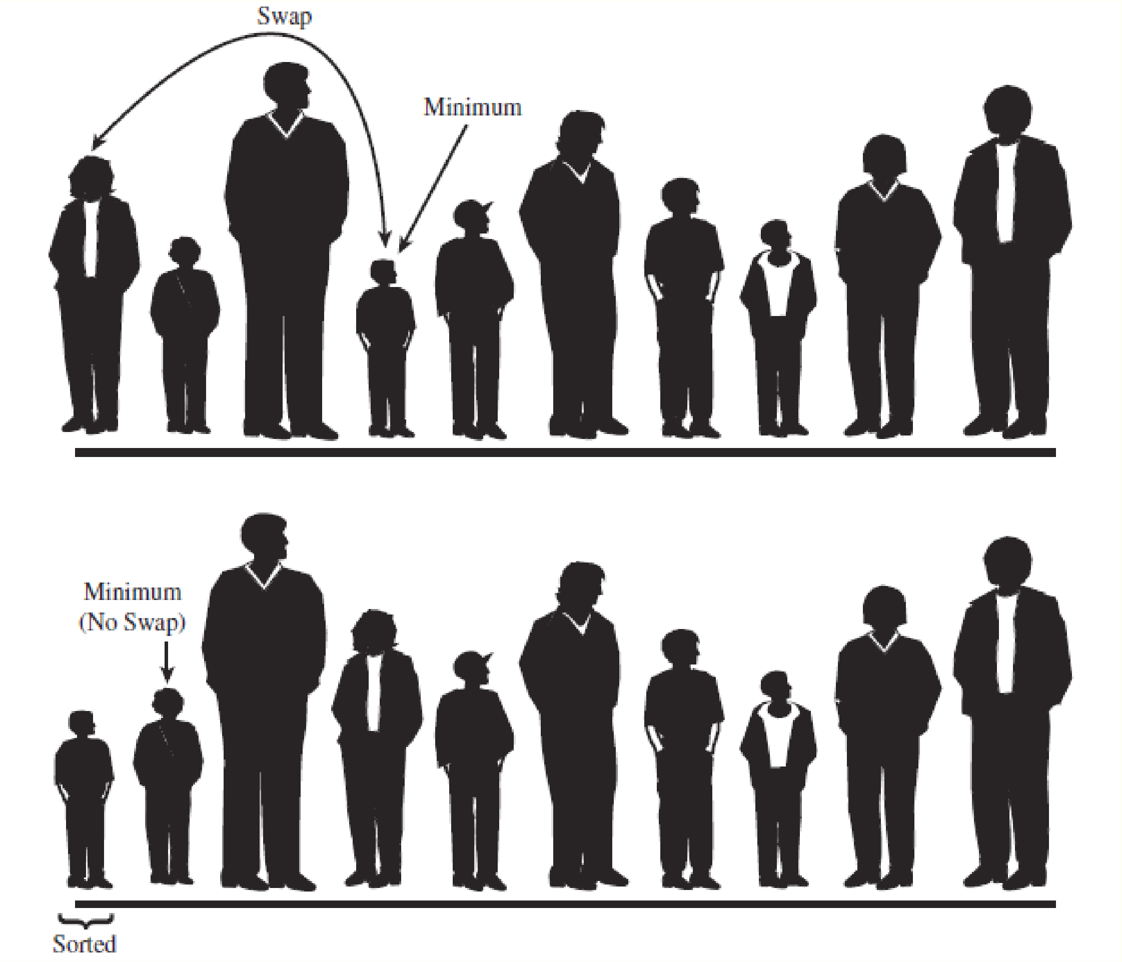
Sort #2: Selection Sort

- Purpose:
  - Improve the speed of the bubble sort
- We'll go back to the baseball team…
  - Now, we no longer compare only players next to each other
  - Rather, we must 'remember' heights
  - So what's another tradeoff of selection sort?

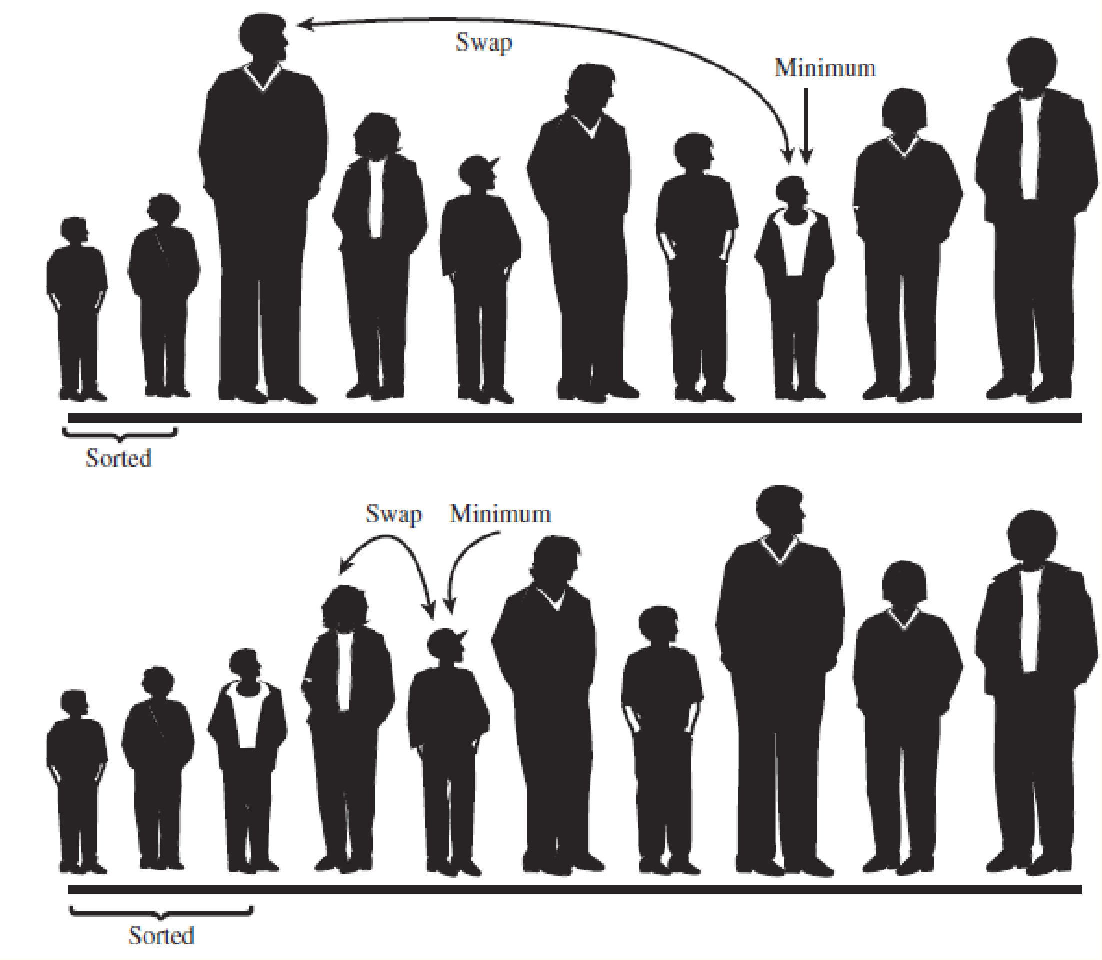
What's Involved

- Make a pass through all the players
  - Find the shortest one
- Swap that one with the player at the left of the line
  - At position 0
- Now the leftmost is sorted
- Find the shortest of the remaining (n-1) players
- Swap that one with the player at position 1
- And so on and so forth…

Selection Sort



Selection Sort



Selection Sort Functoin

```
public void selectionSort()

{

int out, in, min;

for(out=0; out<nElems-1; out++) // outer loop

{min = out; // minimum

for(in=out+1; in<nElems; in++) // inner loop

if(a[in] < a[min] ) // if min greater,

min = in; // we have a new min

swap(out, min); // swap them

} // end for(out)

} // end selectionSort()
```

Sort #3: Insertion Sort

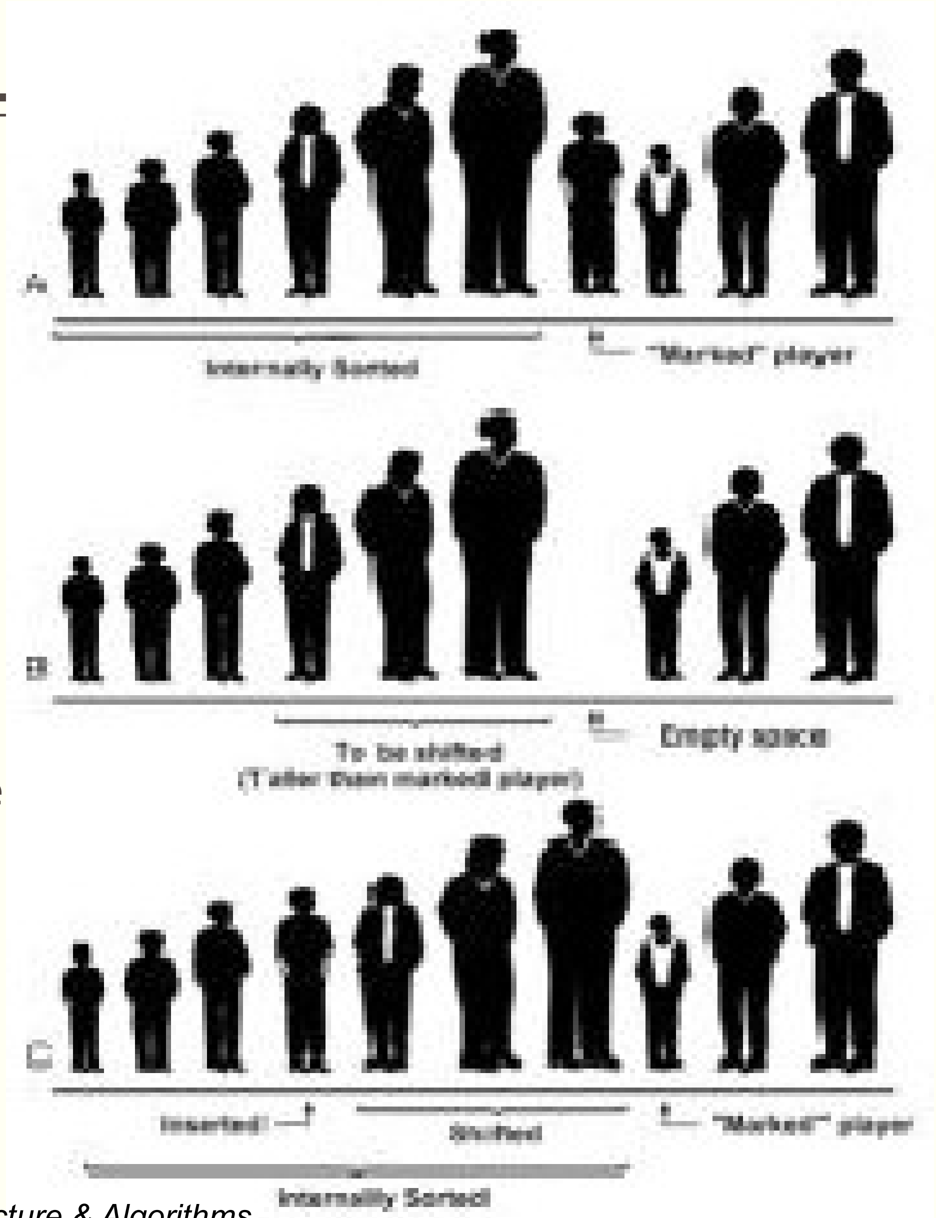
Sort #3: Insertion Sort

- In most cases, the best one…
  - 2x as fast as bubble sort
  - Somewhat faster than selection in MOST cases

- Slightly more complex than the other two
- More advanced algorithms (quicksort) use it as a stage

Proceed..

- A subarray to the left is ‘partially sorted’
  - Start with the first element
- The player immediately to the right is ‘marked’.
- The ‘marked’ player is inserted into the correct place in the partially sorted array
  - Remove first
  - Marked player ‘walks’ to the left
  - Shift appropriate elements until we hit a smaller one



Count Operations

- First Pass, for an array of size n:
  - How many comparisons were made?
  - How many swaps were made?
    - Were there any? What were there?
- Now we have to start again at position two and do the same thing
  - Move the marked player to the correct spot
- Keep doing this until all players are in order

insertion Sort Function

```
public void insertionSort()
{
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()
```

- Any question?

DATA STRUCTURE & ALGORITHMS

Sorting, Searching Algorithm

SEARCHING ALGORITHMS.

Array Searching Algorithms

Two methods for searching an array for a given item:

1. The **Sequential Search** method can be used with any array.
2. The **Binary Search** method can only be used with arrays that are known to be sorted, but is much faster than Sequential Search.

Linear search

Sequential Search

- A **sequential search** of a list/array begins at the beginning of the list/array and continues until the item is found or the entire list/array has been searched

Sequential Search Algorithms

//Search an array A[0..N-1] for X

INPUT : A[0..N-1] an array of integers, floats or chars

item element.

OUTPUT : true if ***item*** is found or false other wise.

bool LinSearch(double x[ ], double item){

b ← true

for i ← 0 to N-1

if (x[i]==item)

b ← true

return false

Binary search

Binary Search

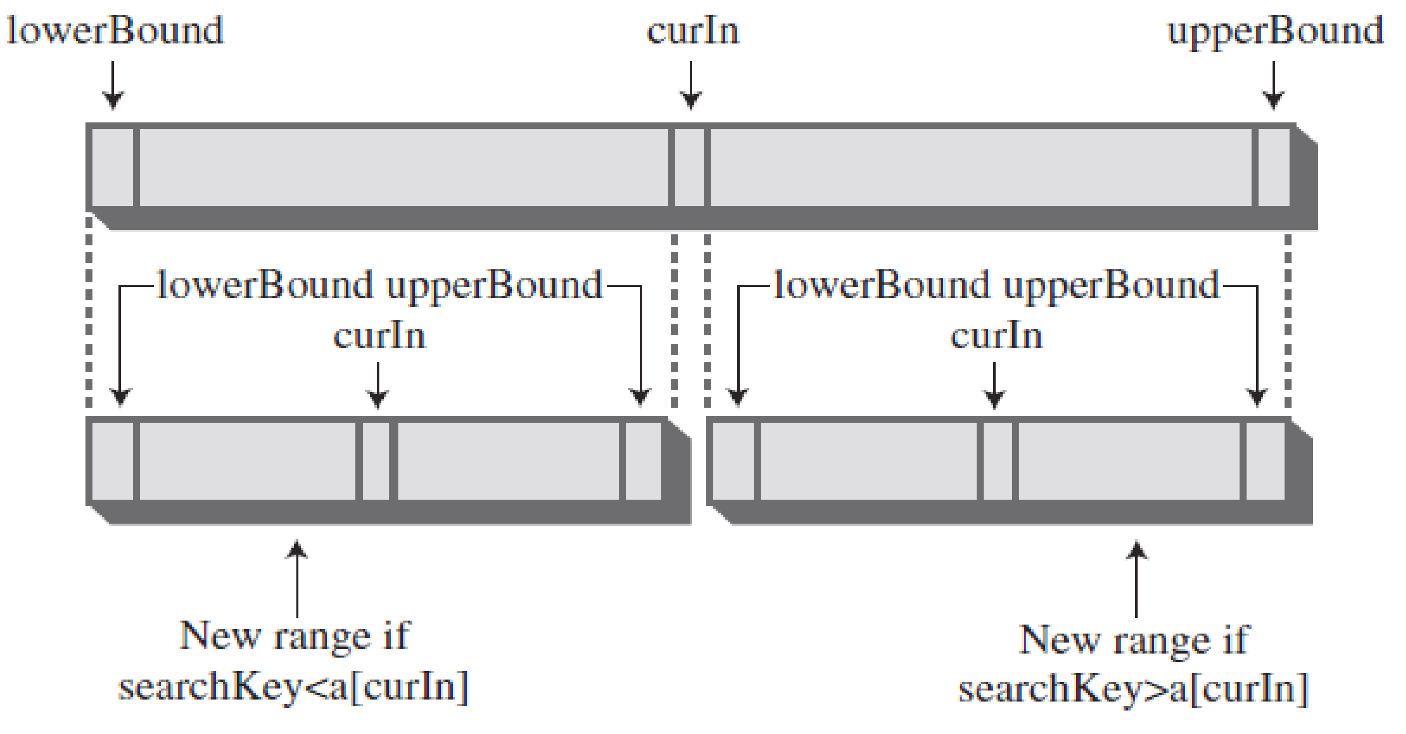
- Binary search algorithm assumes that the items in the array being searched are **sorted**
- The algorithm **begins at the middle** of the array in a binary search
- If the item for which we are searching **is less than the item in the middle**, we know that the item won't be in the second half of the array
- **Once again** we examine the "middle" element
- The process continues with each comparison cutting in half the portion of the array where the item might be.

Example

TABLE 2.2 Guessing a Number

Step Number	Number Guessed	Result	Range of Possible Values
0			1–100
1	50	Too high	1–49
2	25	Too low	26–49
3	37	Too high	26–36
4	31	Too low	32–36
5	34	Too high	32–33
6	32	Too low	33–33
7	33	Correct	

Binary Search (Cont'd)



Binary Search code

```
public int find(long searchKey)
{
int lowerBound = 0;
int upperBound = nElems-1;
int curIn;
while(true)
{
curIn = (lowerBound + upperBound ) / 2;
if(a[curIn]==searchKey)
return curIn; // found it
else if(lowerBound > upperBound)
return nElems; // can’t find it
else // divide range
{
if(a[curIn] < searchKey)
lowerBound = curIn + 1; // it’s in upper half
else
upperBound = curIn - 1; // it’s in lower half
} // end else divide range
} // end while
} // end find()
```

Binary Search code

```
public int find(long searchKey)
{
int lowerBound = 0;
int upperBound = nElems-1;
int curIn;
while(true)
{
curIn = (lowerBound + upperBound ) / 2;
if(a[curIn]==searchKey)
return curIn; // found it
else if(lowerBound > upperBound)
return nElems; // can’t find it
else // divide range
{
if(a[curIn] < searchKey)
lowerBound = curIn + 1; // it’s in upper half
else
upperBound = curIn - 1; // it’s in lower half
} // end else divide range
} // end while
} // end find()
```

Exercise:

- Write a program to convert from infix to postfix.