Simple Sorting

TRAN THANH TUNG

Objectives

- Understand and know how to use basic sorting methods.
  - Bubble Sort
  - Selection Sort
  - Insertion Sort
- Compare their performance

Major Topics

- Introductory Remarks
- Bubble Sort
- Selection Sort
- Insertion Sort
- Sorting Objects
- Comparing Sorts

Introduction

- Why do we need to sort data?
  - To get the lowest price
  - To get the most crowded country
  - etc
- So many lists are better dealt if ordered.

- Sorting data may be a preliminary step to searching

Introduction

- Sorting is time-consuming task
  → many sorting algorithms are developed

- Will look at simple sorting first.
  - Note: there are books written on many advanced sorting techniques. E.g. shell sort; quicksort; heapsort; etc.
- Will start with 'simple' sorts.
  - Relatively slow,
  - Easy to understand, and
  - Excellent performance under circumstances.
- All these are $O(n^2)$ sorts.

How would you do it?

Arrange player in order of increasing height

Basic idea of these simple sorts:

- Compare two items
- Swap or Copy over
  - Depending on the specific algorithm…
- Don't need additional space

Bubble sort

Bubble Sort

- Very slow but simple
- Basic idea:
  - Compare the first item in the first two positions (e.g. the left most)
  - If the first one is larger, swap with the second
  - Move one position right.

→At end, the largest item is in the last position. (e.g. the right most)

→ That why is name Bubble sort

Bubble Sort process

- After first pass, we made
  - n -1 comparisons
  - 0 to n-1 swaps (depend on data)

- Continue this process.

- Next time we do not check the last entry (n-1), because we know it is in the right spot. We stop comparing at (n-2).
- See the workshop
- See Some code (p85-86).

Hand-on

- Try with some examples

89	58	29	40	12	42	10	1
1	32	12	53	11	76	23	89

BubbleSort procedure

```
for(out=nElems-1; out>0;out--)   // outer loop (backward)
   for(in=0; in<out; in++)       // inner loop (forward)
      if( a[in] > a[in+1] )      // out of order?
         swap(in, in+1);         // swap them
```

Homework

Rewrite the code

1. To sort from Right to Left
2. To sort array descending
3. To use 2 forward loops

Efficiency of the Bubble Sort - **Comparisons**

- Can readily see that there are fewer comparisons each 'pass.'
- Thus number of comparisons is computed as: $(n-1)+(n-2)+\ldots + 1 = n(n-1)/2$;
  - For 10 elements, the number is $10*9/2 = 45$.
- So, the algorithm makes about $n^2/2$ comparisons

(ignoring the -1 which is negligible especially if N is large)

Efficiency of the Bubble Sort - **Swaps**

- Fewer SWAPS than COMPARISONS

  since every comparison does not result in a swap.

- In general, a swap will occur half the time.
  - For **$n^2/2$** comparisons,
    we have **$n^2/4$** swaps.

- Worst case, every compare results in a swap (which case?)

Overall – Bubble Sort

- Both swaps and compares are proportional to n^2.
  - Ignore the 2 and 4

→ Complexity of Bubble Sort is $O(n^2)$

→ Rather slow.

- Hint to determine Big-O:

  2 nested-loop → $O(n^2)$

  - Outer loop executes n times and inner loop executes in n times PER execution of the outer n times: hence n^2

Question

- The bubble sort algorithm alternates between:
  - A) Comparing and swapping
  - B) Moving and copying
  - C) Moving and comparing
  - D) Copying and comparing

- What can be improved in Bubble Sort algorithm?

Selection sort

Selection Sort

Fewer swaps, same comparisons.

- Swaps $O(n^2) \rightarrow O(n)$.
- Comparisons $O(n^2)$.

→Important while dealing with large records

→Reduction in swap time is more important than one in comparison time

How does the Selection Sort work?

- Start from the first element (e.g. the left end)
- Scan all elements to selecting the smallest (largest) item.
- Swap with the first element
- Next pass, move one position right
- Repeat until all are sorted.

Selection Sort – in more detail

- So, in one pass, you have made n comparisons but possibly ONLY ONE Swap!
- With each succeeding pass,
  - one more item is sorted and in place;
  - one fewer item needs to be considered.
- Java code for the Selection Sort (p93-94).

Selection sort

```
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()
```

Selection Sort itself - more

- Algorithm implies it is an $O(n^2)$ sort (and it is).
  - How did we see this?

In comparison with Bubble Sort

- Same number of comparisons **$n^2/2$**
- Fewer swap **n**

→ Faster than Bubble Sort

Question

- In the selection sort:
  - A) the largest keys accumulate on the left (low indices)
  - B) A minimum key is repeatedly discovered
  - C) A number of items must be shifted to insert each item in its correctly sorted position
  - D) The sorted items accumulate on the right

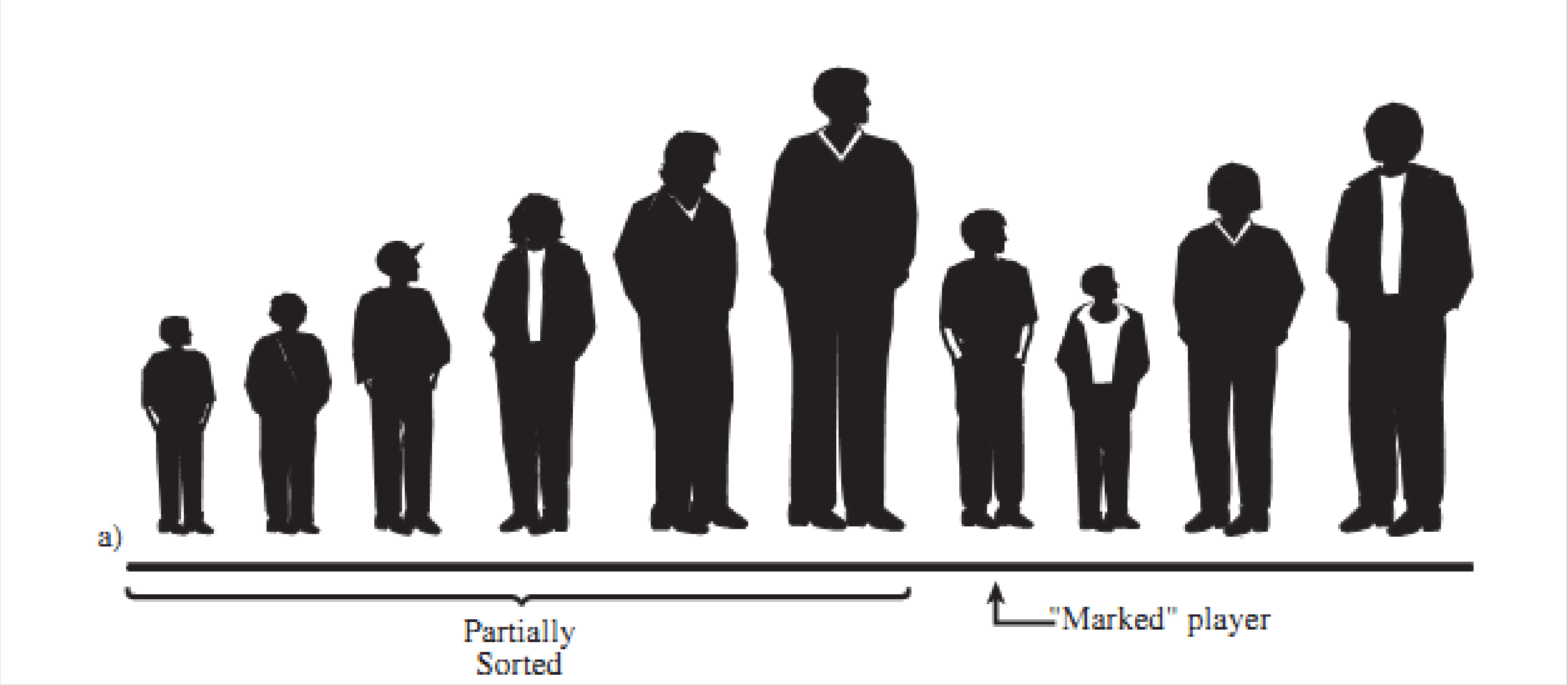
Insertion sort

Insertion Sort

- In many cases, this sort is considered the best of these elementary sorts.
- Still an $O(n^2)$ but:
  - about twice as fast as bubble sort and
  - somewhat faster than selection sort in most situations.
- Easy, but a bit more complex than the others
- Sometimes used as **final stage** of some more sophisticated sorts, such as a QuickSort (coming).

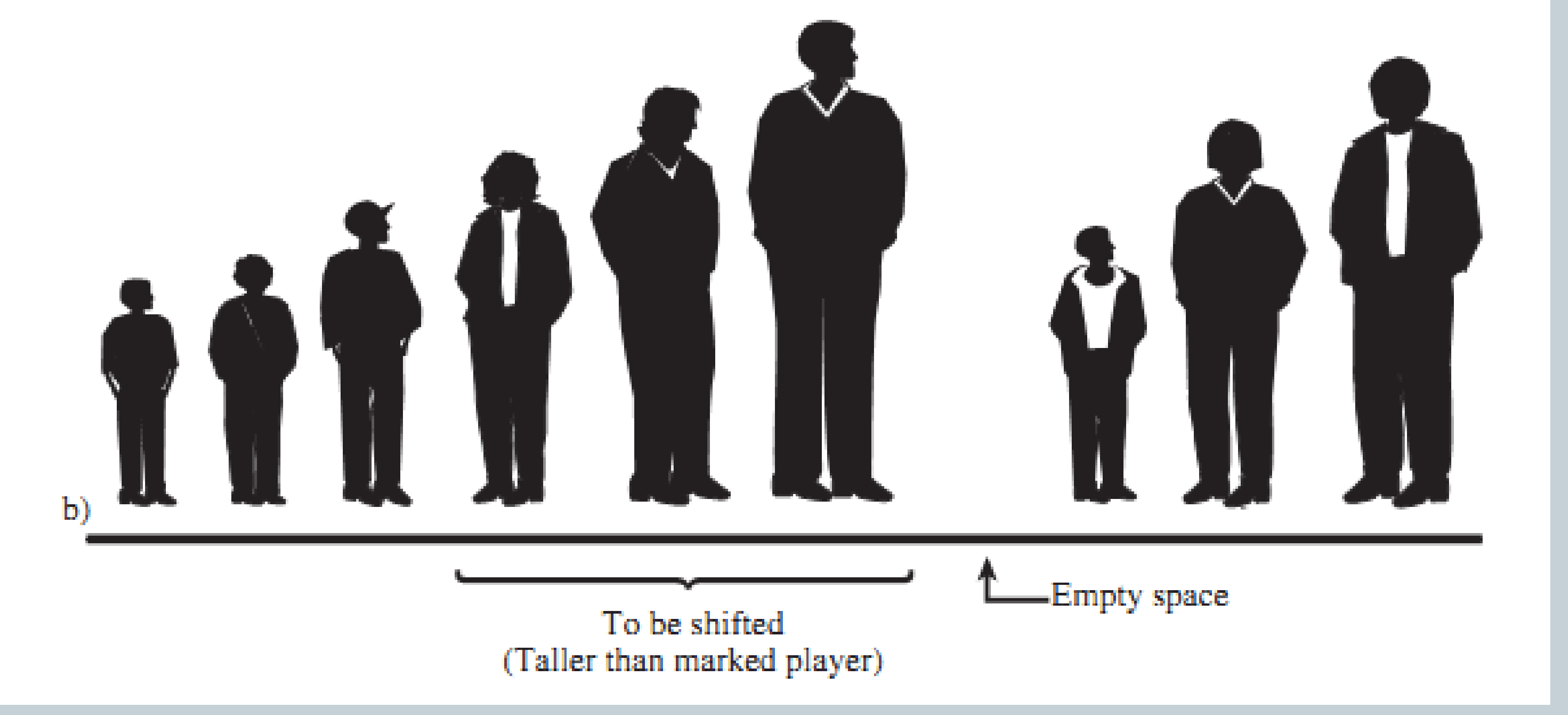
Insertion Sort – The idea

- Thinking that 'half' of the list of items to be sorted. (*Partially sorted* , *Marked*, *Unsorted* items)

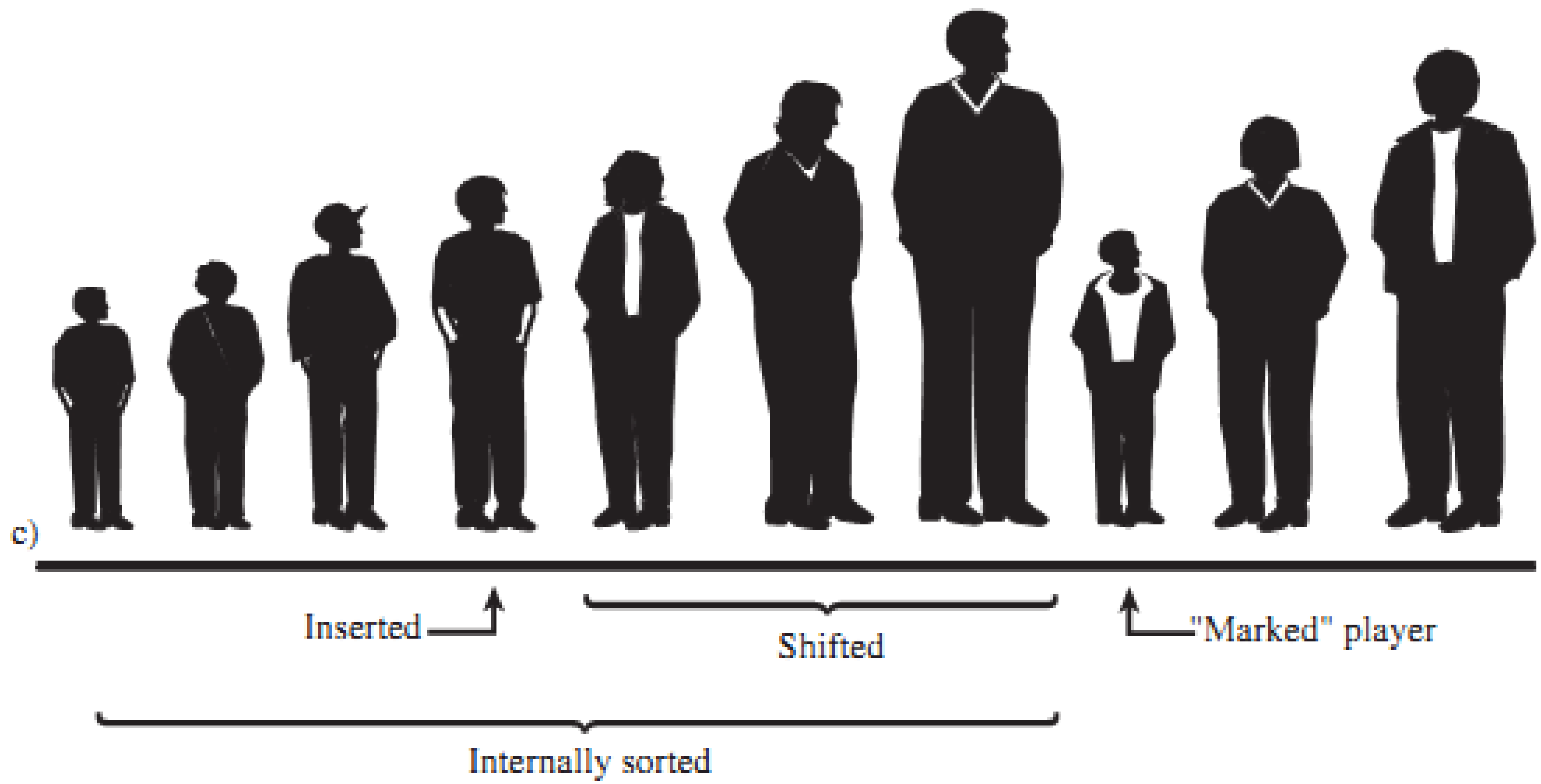


Insert *marked* item to *partially sorted* list

- Take out of line → Shift right until appropriate place → Insert



Insertion sort – Intermediate result



Insertion Sort

- Result:
  - partially-ordered list is one item larger and
  - the unsorted list is one item smaller.
  - Marked item moves one slot to the right, so once more it is again in front of the leftmost unsorted item.

- Continue process until all unsorted items have been inserted.
- Hence the name 'insertion sort.'
- Code page 99 - 100

Insertion sort

```
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()
```

Discussion– How really implemented!!

- Start with out = 1, which means there is only a single element to its 'left.'
  - We infer that this item to its left is sorted unto itself.
  - Hard to argue this is not true. (This is out = 0)

- a[out] is the marked item, and it is moved into temp.
  - a[out] is the leftmost unsorted item.

- Algorithm is an O(n2) sort.

- Let's consider some data…

Hand-on

24	19	42	12	41	29	50

Efficiency of the Insertion Sort

Comparisons:

- On pass one, max of one;
  pass two, max of two, etc.
  Up to a max of n-1 comparisons.

→ $1+2+3+\ldots+n-1 = n*(n-1)/2$ comparisons.

- But, on average

$n*(n-1)/4$.

Efficiency of the Insertion Sort (2 of 3)

Copy:

- Have lots of ‘copies’
  - (same as number of comparisons – about)

➔ But a **copy** is not nearly as time-consuming as a **swap. Think about this!!**

- For random data,
  - twice as fast as the bubble sort
  - faster than the selection sort.
- Still runs on $O(n^2)$ time for random data.

Efficiency of the Insertion Sort (3 of 3)

- If data is *nearly sorted* → quite well.
  - Condition in while loop is never true, so it almost runs in O(n) time; much better than O(n^2) time!

- If data is in *very unsorted* order (nearly backward) → No faster than the bubble sort
  - as every possible comparison and shift is carried out.

Question

- In the insertion sort, “partially sorted” means:
  - A) Some items are already sorted, but they may need to be moved
  - B) Most items are in their final sorted positions, but a few still need to be sorted
  - C) Only some of the items are sorted
  - D) Group items are sorted among themselves, but items outside the group may need to be inserted in it.

Sorting Objects

- Very important to be able to sort objects.
- Let's sort an array of Person objects.
- Must be careful, especially in noting that
  - An array are **objects**, and
  - The sort is based on **values of String attributes** inside of the object.
    - Use inherited String method, compareTo.

Sort objects

```
while(in>0 &&                        // until smaller one found,
      a[in-1].getLast().compareTo(temp.getLast())>0)
   {
   a[in] = a[in-1];                  // shift item to the right
   --in;                             // go left one position
   }
```

String comparison

TABLE 3.1 Operation of the `compareTo()` Method

`s2.compareTo(s1)`	**Return Value**
s1 < s2	< 0
s1 equals s2	0
s1 > s2	> 0

Secondary Sort Fields?

- Equal Keys Problems ? E.g., Last name
- Solutions
  - Using **Secondary Key**? E.g., ZIP code
  - Using **'Stable' Sort**? What does this mean?
    - Only sort what needs to be sorted
    - Leave everything else in its original order
- All of the algorithms so far are stable
- Think Windows:
  - Arrange by Type; then by date modified…

Comparing the Simple Sorts

Bubble Sort – simplest.

- Use only if you don't have other algorithms available and 'n' is small.

- Selection Sort
  - Minimizes number of swaps, but number of comparisons still high.
  - Useful when amount of data is small and swapping is very time consuming – like when sorting records in tables – internal sorts.
- Insertion Sort
  - The most versatile, and is usually best bet in most situations, if amount of data is small or data is almost sorted. For large n, other sorts are better. We will cover advanced sorts later…

Question

- Big O of bubble, selection and insertion?
  - Time/Speed?
  - Space/Memory?
- Which one has the smallest number of comparisons?
- Which one has the smallest number of swaps?
- Which one has the smallest number of copies?
- Put into ascending order the speed of the three types of sort: bubble, selection and insertion
- Questions & questions

Comparing the Simple Sorts

- All require very little space and they sort in place.
- Can’t see the real efficiencies / differences unless you apply the different sources to large amounts of data.
- Remember:
  - Doing Experiments
  - Doing Programming Projects