Lecture 17: Sorting

Dr Kieran T. Herley

Department of Computer Science
University College Cork

2013/14

Sorting Problem

Sorting Problem

Input List of n *keys* in no particular order

Output List rearranged in increasing order of key

Motivation fundamental algorithmic tool in many real-world applications

Problem Variations

- Sorting in decreasing order
- Sorting *items* = keys + other stuff

Some Sorting Algorithms

There are literally dozens of main sorting algorithms including

- Bubblesort, Selectionsort, Insertionsort, Quicksort, Shellsort, Mergesort, Heapsort, Radix-sort
- And many more plus a myriad of variations, combinations and so on.

Bubblesort

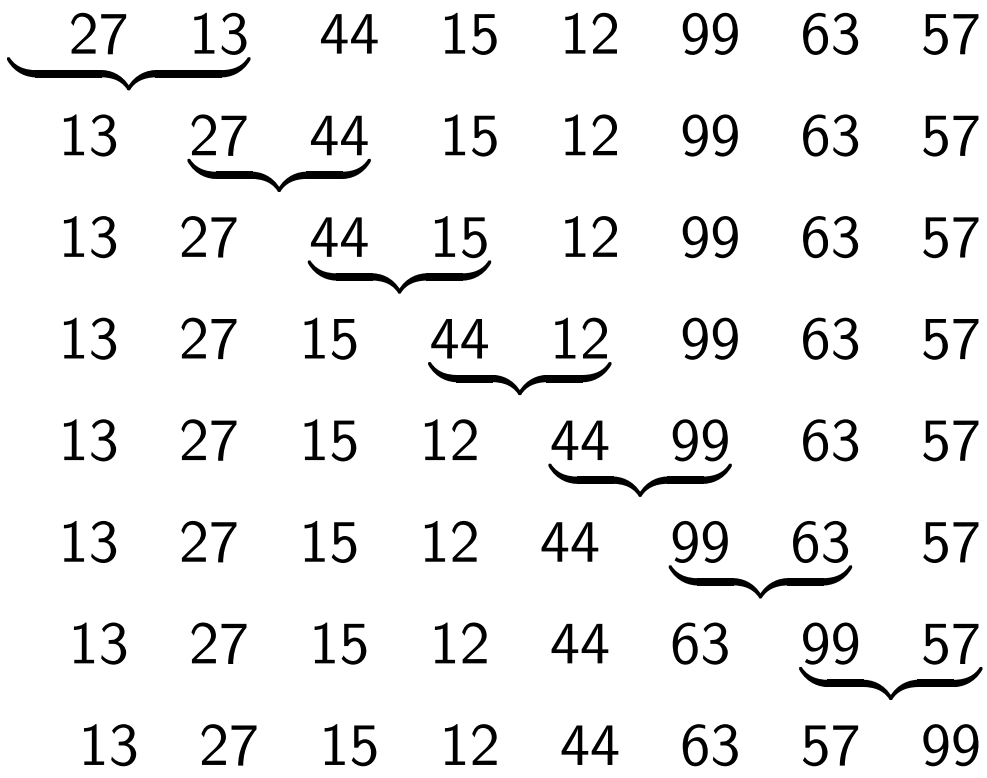
- Bubblesort sorts a list (ADT List) of values
- Based on a structured pattern of *comparison-exchange* operations
- *ComparisonExchange*(i): Take value in two adjacent slots in the list and if the values are out of order (i.e. the larger before the smaller), then swap them around:

$$\cdots \underbrace{27 \quad 13} \cdots \quad \rightarrow \quad \cdots 13 \quad 27 \cdots \text{(Swap)}$$

$$\cdots \underbrace{27 \quad 44} \cdots \quad \rightarrow \quad \cdots 27 \quad 44 \cdots \text{(No swap)}$$

Bubblesort cont'd

- Bubblesort involves multiple *sweeps* through list
- *Sweep*: For an n-element list, apply $n-1$ comparison-exchanges to each pair of adjacent position in left-to-right order.



Bubblesort cont'd

- Bubblesort involves $n - 1$ sweeps through the list
-

sweep = 0	27	13	44	15	12	99	63	57
sweep = 1	13	27	15	12	44	63	57	**99**
sweep = 2	13	15	12	27	44	57	**63**	**99**
sweep = 3	13	12	15	27	44	**57**	**63**	**99**
sweep = 4	12	13	15	27	**44**	**57**	**63**	**99**
sweep = 5	12	13	15	**27**	**44**	**57**	**63**	**99**
sweep = 6	12	13	**15**	**27**	**44**	**57**	**63**	**99**
End :	12	**13**	**15**	**27**	**44**	**57**	**63**	**99**

(Each line represents the state of the list at the start of the sweep.)

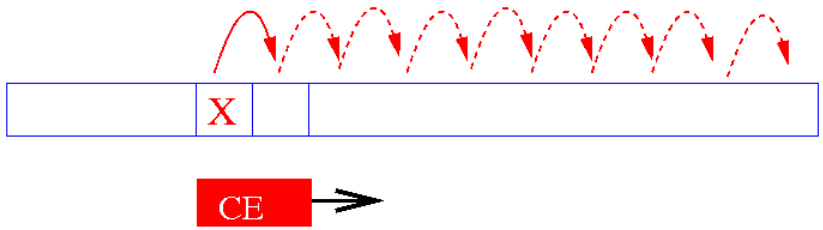
Bubblesort Pseudocode

```
(1) Algorithm Bubblesort(L):
(2)     n ←L.size()
(3)     for s ←1 to n − 1 do
(4)         for current ←0 to n−2 do
(5)             next ←current + 1
(6)             currentElt ←L.get(current)
(7)             nextElt ←L.get(next)
(8)             if currentElt > nextElt then
(9)                 L.set(current, nextElt)
(10)                L.set(next, currentElt)
```

- L is a list (ADT List)
- Lines (6-10) implement a comparison exchange
- Loop (4-10) implements one sweep
- Loop (3-) ensures multiple sweeps

Useful Observation

- Consider largest value X:
  - No CE can move X leftwards
  - Every CE with X on LHS moves it rightwards
- First sweep pushes X into very last slot in the list (where it belongs)



- CEs of subsequent sweeps leave it there

Correctness

Assume list elements distinct (argument generalizes)

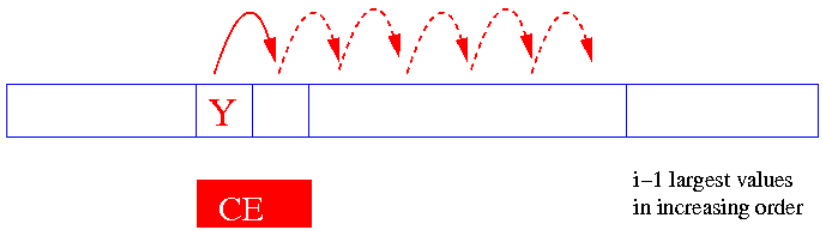
Proposition

After i sweeps, largest i values occupy last i positions in increasing order.

-
- Reasoning by induction on i
- Case $i = 1$: Consequence of Useful Observation

Correctness cont'd

- Case $i > 1$:
  - Following first $i - 1$ sweeps, largest $i - 1$ elements occupy last $i - 1$ positions in increasing order.



  - During i-th sweep,
    - largest $i - 1$ values already in position and will not move
    - Let Y be the ith largest value overall; "wins" every comparison exchange against smaller values and gets pushed rightwards
    - Y ends up in ith last position
    - Reasoning akin to Useful Observation for base case
- (Flesh out details as exercise)

Analysis

```
(1) Algorithm Bubblesort(L):
(2)     n ←L.size()
(3)     for s ←1 to n − 1 do
(4)       for current ←0 to n−2 do
(5)         next ←current + 1
(6)         currentElt ←L.get(current)
(7)         nextElt ←L.get(next)
(8)         if currentElt > nextElt then
(9)           L.set(current, nextElt)
(10)          L.set(next, currentElt)
```

- Assume list ops. contain no comparisons
- Number of comparisons equals
  - Comparisons of line (3)
    - $n - 1$ iterations
    - n comparisons
  - Plus comps. from $n - 1$ repetitions of inner loop

Analysis cont'd

```
(4)     for current ←0 to n−2 do
(5)         next ←current + 1
(6)         currentElt ←L.get(current)
(7)         nextElt ←L.get(next)
(8)         if currentElt > nextElt then
(9)           L.set(current, nextElt)
(10)          L.set(next, currentElt)
```

- Focus on *single execution* inner loop
  - # iterations $= n - 1$
  - # comparisons

$$\underbrace{n}_{(4)} + \underbrace{n-1}_{(8)} = 2n - 1$$

- But need to reflect fact that this is repeated

Analysis cont'd

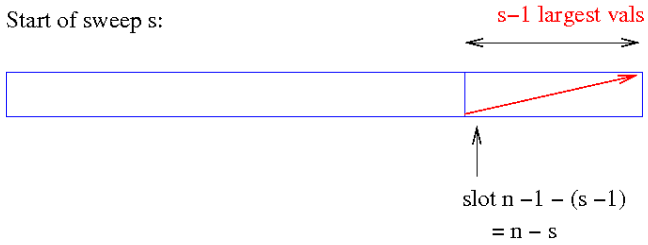
```
(1) Algorithm Bubblesort(L):
(2)     n ←L.size()
(3)     for s ←1 to n − 1 do
(4)         for current ←0 to n−2 do
(5)             next ←current + 1
(6)             currentElt ←L.get(current)
(7)             nextElt ←L.get(next)
(8)             if currentElt > nextElt then
(9)                 L.set(current, nextElt)
(10)                L.set(next, currentElt)
```

Number of comparisons:

$$\underbrace{n}_{(3)} + \underbrace{\underbrace{(2n-1)}_{\text{cost per.}} \cdot \underbrace{(n-1)}_{\#\text{ reps.}}}_{(4-10)} = 2n^2 - 2n + 1$$

Refinement

- Can reduce upper limit of inner loop to $n - s - 1$




```
(1) Algorithm Bubblesort(L):
(2)     n ←L.size()
(3)     for s ←1 to n − 1 do
(4)         for current ←0 to n − s − 1 do
(5)             next ←current + 1
(6)             currentElt ←L.get(current)
(7)             nextElt ←L.get(next)
(8)             if currentElt > nextElt then
(9)                 L.set(current, nextElt)
(10)                L.set(next, currentElt)
```

- Refined analysis:

$$\underbrace{n}_{(3)} + \underbrace{\sum_{s=1}^{n-1} \underbrace{2(n-s)+1}_{\text{s-th sweep}}}_{(4-10)} = n + 2\sum_{s=1}^{n-1}(n-s) + \sum_{s=1}^{n-1} 1 = \cdots = n^2 + n - 1$$

- Small print: Have used identity $1 + 2 + \cdots + h = \sum_{i=1}^{h} i = h(h+1)/2$

BubbleSort in Java

Interface

```
public interface SortObject<KeyType>
{   public void sort ( List <KeyType> S, Comparator<KeyType> c);
}
```

Implementation

```
public class BubbleSort<KeyType>
        implements SortObject <KeyType>
{

    public void sort ( List <KeyType> lst, Comparator<KeyType> c)
    {. . .}
}
```

Notes

- Comparator allows algorithm to cope with different element types
- Common sorting interface allows different algorithms to be interchanged with ease

Bubblesort sort Method

```
public void sort(List<KeyType> lst, Comparator<KeyType> c)
  {  int numElts = lst.size();
     int sweepNum, current, next;

     for (sweepNum = 1; sweepNum < numElts; sweepNum++)
     {  current = 0;
        for (current = 0; current < numElts-1; current++)
        {  next = current + 1;
           KeyType currentElt = lst.get(current);
           KeyType nextElt = lst.get(next);
           if    (c.compare(currentElt, nextElt) > 0)
           {  lst.set(current, nextElt);
              lst.set(next, currentElt);
           }
        }
     }
  }
```

Reversing Sorting Order

- Could "re-engineer" algorithm, but easier to change comparator, but .
  .

Reversing Sorting Order

- Could "re-engineer" algorithm, but easier to change comparator, but . .
- Reverse comparator:

```
public class ReverseIntegerComparator
        implements Comparator<Integer>
{  public int compare(Integer a, Integer b)
   {  return -1 * comp.compare(a, b);
   }

   Comparator<Integer> comp = new IntegerComparator();
}
```

-

```
myList = . . . // a list of numbers
Comparator revComp = new ReverseIntegerComparator();

SortObject<Integer> sorter = new BubbleSort<Integer>();
sorter.sort(myList, revComp);
```

Array-Based Bubblesort Formulation

Interface

```
public interface SortObject2<KeyType>
{  public void sort(KeyType[] S, Comparator<KeyType> c);
}
```

Implementation

```
public class BubbleSort2<KeyType>
        implements SortObject2 <KeyType>
{  public void sort(KeyType[] lst , Comparator<KeyType> c)
   {. . .}
}
```

Notes

- Obvious changes to code, *e.g.*
  currentElt = lst[current] instead of
  currentElt = lst.get(current) *etc.*
- Complete as exercise.

SelectionSort Sketch

R	L
()	(7, 4, 8, 2, 5, 3, 9)
(9)	(7, 4, 8, 2, 5, 3)
(8, 9)	(7, 4, 2, 5, 3)
(7, 8, 9)	(4, 2, 5, 3)
(5, 7, 8, 9)	(4, 2, 3)
(4, 5, 7, 8, 9)	(2, 3)
(3, 4, 5, 7, 8, 9)	(2)
(2, 3, 4, 5, 7, 8, 9)	()

- At each step remove largest remaining element from L and add to the start of R
- When L exhausted, elements appear in R in increasing order
- Assign R to L and return

- Can be implemented using a single list/array; $O(n^2)$ [1] comparisons
- See website for details

[1] Intuitively, this means num of comarisons is "in the ballpark of n^2" for a particular notion of ballpark not defined here.

InsertionSort Sketch

R	L
$()$	$(7,4,8,2,5,3,9)$
(7)	$(4,8,2,5,3,9)$
$(4,7)$	$(8,2,5,3,9)$
$(4,7,8)$	$(2,5,3,9)$
$(2,4,7,8)$	$(5,3,9)$
$(2,4,5,7,8)$	$(3,9)$
$(2,3,4,5,7,8)$	(9)
$(2,3,4,5,7,8)$	$()$

- Maintain R in increasing order
- At each step remove next element from L and add to R (maintaining order)
- When L exhausted, elements appear in R in increasing order
- Assign R to L and return

- Can be implemented using a single list/array with $O(n^2)$ comparisons; See website
- Insertionsort is regarded as the best of the "simple" sorting algorithms (bubblesort, insertionsort, selectionsort)

Practical Notes

- InsertionSort generally considered superior to BubbleSort
- Array versions generally preferred in practice
- Code for various algorithms on website