Session 06

Sorting - I

Overview

- Need for sorting.
- Simple sorting algorithms
  - Insertion
  - Selection
  - Shell
  - Bubble
- Comparison and complexity analysis.

Introduction

- We realize need for keeping information in some order in our daily life for *convenience.*
- Imagine the mess created if dictionaries/ telephone directories were not arranged in some order.
- Similarly, programmers realized that it is often necessary to organize data before processing, to achieve *efficiency*.

Approach

Step I:

Choose the *criteria* for sorting (for natural numbers, criteria can be ascending or descending order).

Step II:

How to put data in order using the criterion selected.

Analysis

- Final ordering of data can be obtained in a variety of ways.
- Some are meaningful and efficient.
- Depends on many aspects of an application: type of data, randomness of data, run time constraints, size of the data, nature of criteria, etc.
- Many algorithms exist.
- How to decide which algorithm is the best?

Analysis

- To make comparisons, certain properties of sorting algorithms should be defined.
- Properties which are used to compare algorithms without depending on the type and speed of the machines are:
  - number of *comparisons*.
  - number of *data movements*.
  - Use of *auxiliary storage*.

Analysis

- Practical reason must aid the choice of algorithm.
- Different dimensions may rate an algorithm

  differently.

  E.g. An algorithm can be efficient on data movements and can perform poorly on number of comparisons or vice versa.

The Problem

- Problem is to arrange the records of a file in ascending order as per the specified criteria.
- '>' defined by the domain. E.g. strings, numbers etc.
- Internal Sort
  - The data is completely in the main memory.
- External Sort
  - The data is at least partially in the secondary storage (e.g. hard disk)

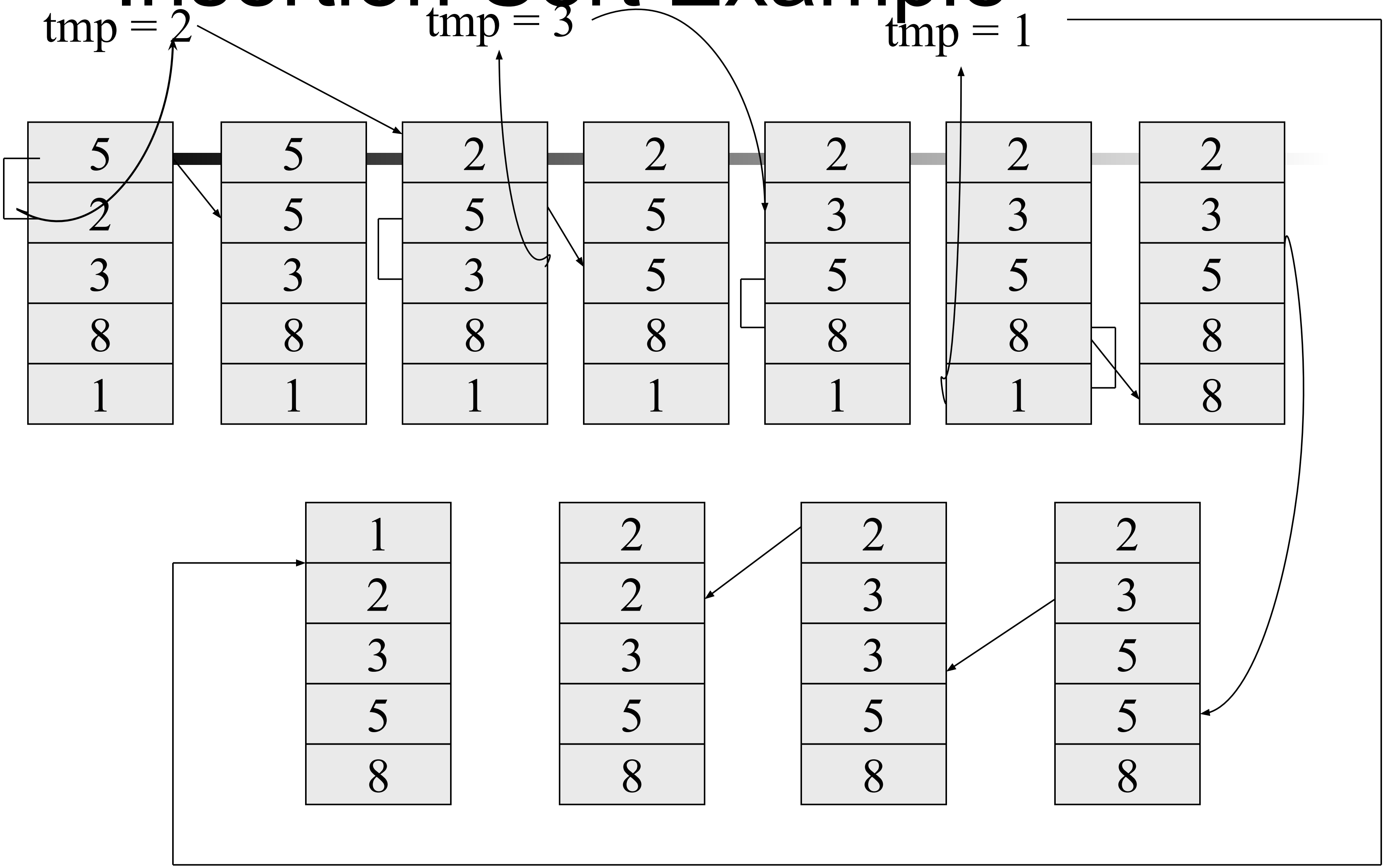
Insertion Sort

Consider an array a[N] of N elements.

for next 1 to N

- array 0 to next-1 is sorted.
- Insert a[next] in this sorted array suitably, so that 0 to next becomes sorted.

Insertion Sort Example



Algorithm

for *next* = 1 to *N*

/* *data*[*next*] to be inserted in *data*[0]….*data*[*next*] */

x = *data*[*next*]

a. scan from *next* leftwards till an element less than or equal to *x* is found, shift all the elements greater than *x* one position right.

b. Insert *x* where the iteration stops.

Implementation

```
public void InsertionSort(int[] data){
   int tmp;
   int i, j;
   for(i = 1; i < data.length; i++){
    tmp = data[i];
    for(j=i; j>0 && tmp < data[j-1]; j--)
      data[j] = data[j-1];
    data[j] = tmp;
   }
}
```

Analysis

- We are assuming ascending order
- First element need not be inserted => hence the for-loop starts with 1.
- Check boundary cases for insertion.
  - What if data[next] is the largest so far?
    (It should stay at next itself - should not be moved).
  - What if data[next] is the smallest so far?

Analysis

- Outerloop executed N-1 times, for every run
- Inner loop *(step a)* depends on the next value. For a given A[next],
  - only once in best case
  - ‘next’ times in worst case
  - next/2 on an average

Analysis

- Best case complexity: (N-1) * 1 = O(N)
- Worst case complexity: 1+2+ ... +N-1 = $aN^2+bN+c = O(N^2)$
- Average case: $O(N^2)$

Observations

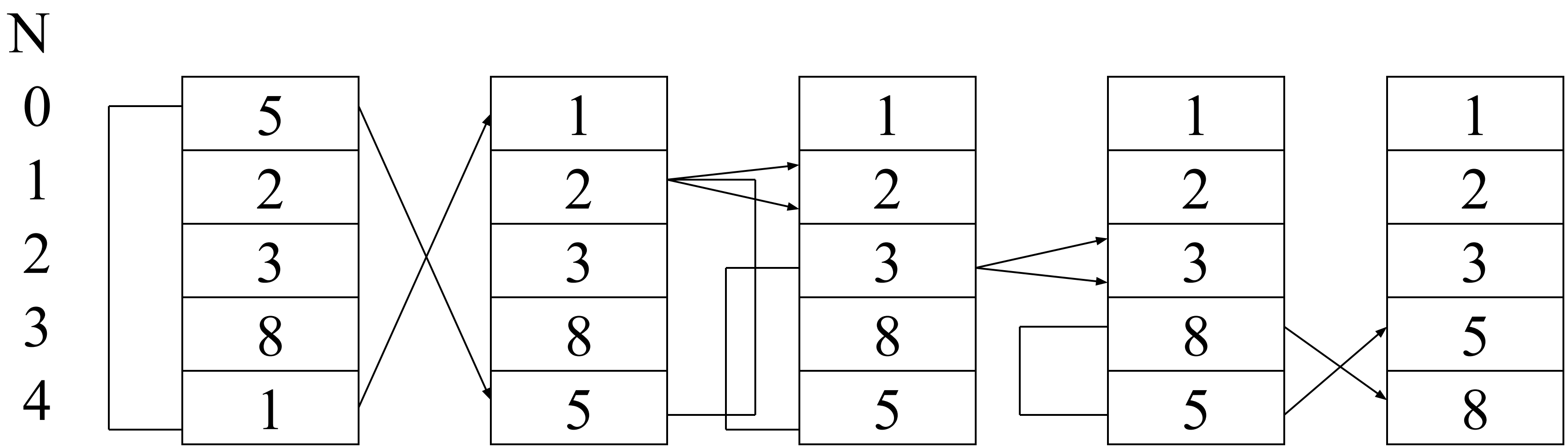
- Note that every step involves a comparison and data movement.
- If records have a complex structure, movements are expensive.
- Suitable for less elements, small record size
- Selection sort reduces data movement.

Selection Sort

We pick the elements in required order and place at successive locations. (N = number of elements –1).

- Pick the smallest from 0 to N, and exchange with *data*[0].
- Pick the smallest from 1 to N, and exchange with *data*[1].
- Pick the smallest from 2 to N, and exchange with *data*[2].
- At step i, pick the smallest from i to N, and exchange with $A[i]$.
- Repeat till $i = N$-2.

Selection Sort Example



Implementation

```
public void selectionSort(int[] data){
  int i,j,least;
  for(i = 0; i < data.length-1; i++)
  {
   for(j = i+1,least = i;j < data.length;j++)
     if(data[j] < data[least])
      least = j;
   swap(data,least,i);
  }
}
```

Analysis

- Count of iterations predictable - no best case, worst case, etc.
- Outer loop N-1 times, inner loop i times.
- Each step involves a comparison primarily.
- Three record movements for each iteration of outer loop (done in swap).

Analysis

- Time: $1+2+3+4+...+N-1 = O(N^2)$
- Best/Worst/Average are all $O(N^2)$.
- Data movement is $O(N)$, it was $O(N^2)$ in insertion sort.
- Independent of initial order of data
- More comparisons, less movements

Observations

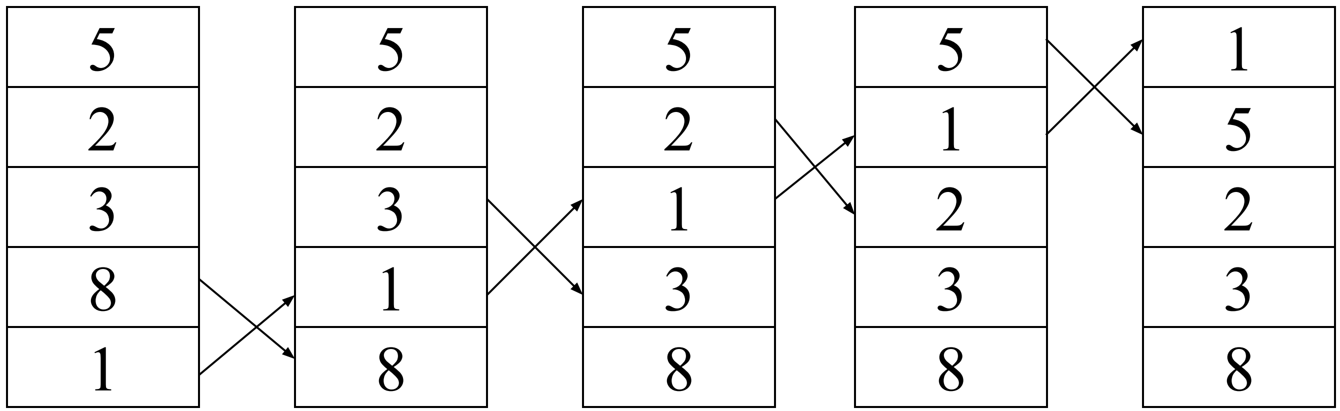
- Insertion sort moves data only one step at a time.
- Can reduce movement, if movement could take larger strides.
- Selection sort does perfect movement, but too many comparisons.

Bubble Sort

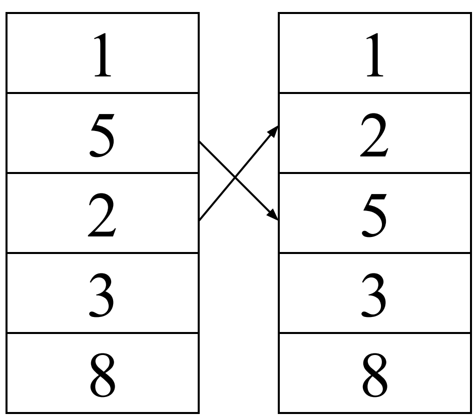
- The array is scanned and adjacent elements are interchanged if they are out of order with respect to each other.
- This way, after first pass smallest element is bubbled up to the top, i.e., data[0].
- The array is scanned again up to *data*[2] and *data*[1] and second smallest element bubbles to its position and it continues until the last pass which involves only *data*[*n-1*] and *data*[*n-2*], and possibly one interchange is performed.

Bubble Sort Example

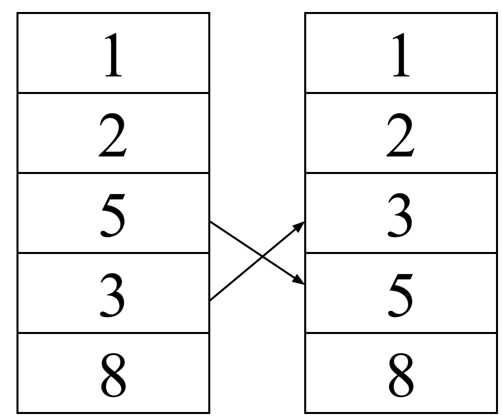
First pass



Second pass



Third pass



Implementation

```
public void bubblesort(int[] data){
  int i,j;
  for(i = 0; i < data.length-1; i++)
  for(j = data.length-1; j > i; --j)
      if(data[j] < (data[j-1])
        swap(data,j,j-1);
}
```

Analysis

- The number of comparisons are *same* in each case (best, average and worst cases) which is equal to $O(N^2)$.
- In the best case, when all the elements are already ordered; there are no swaps.

Analysis

- Disadvantages
  - If an element has to be moved from bottom to top, it is exchanged with every element in the array, even though some elements might be in their final position.
  - It looks at only two adjacent elements at a time.
- In average case, bubble sort makes
  - approximately twice as many comparisons and the same number of moves as insertion sort.
  - As many comparisons as selection sort, and n times more moves than selection sort.

Shell Sort

- Shell sort provides a balanced sort.
- Also called Diminishing Increment sort.
- Intention: to see that the data is either small or is nearly sorted so that insertion sort becomes affordable.

Shell Sort

- Choose an increment k, say 3
- Consider sub-files

  [1] (5) {8} [3] (4) {7} [10] ...

- Sort all the [], (), {}.

  [1] (4) {7} [3] (5) {8} [10] ...

Shell Sort

- Reduce the increment, say to 2

    [1] (4) [7] (3) [5] (8) [10] ...

- Sort all the [], and ().

    [1] (3) [5] (4) [7] (8) [10] ...

- Reduce the increment to 1 and sort

    [1] [3] [4] [5] [7] [8] [10] ...

The array to be sorted : [10 8 6 20 4 3 22 1 0 15 16]

Data before 5-sort	10	8	6	20	4	3	22	1	0	15	16
Five subarrays before sorting	10	-	-	-	-	3	-	-	-	-	16
		8	-	-	-	-	22				
			6	-	-	-	-	1			
				20	-	-	-	-	0		
					4	-	-	-	-	15	
Five subarrays after sorting	3	-	-	-	-	10	-	-	-	-	16
		8	-	-	-	-	22				
			1	-	-	-	-	6			
				0	-	-	-	-	20		
					4	-	-	-	-	15	
Data after 5-sort & before 3-sort	3	8	1	0	4	10	22	6	20	15	16
Three subarrays before sorting	3	-	-	0	-	-	22	-	-	15	
		8	-	-	4	-	-	6	-	-	16
			1	-	-	10	-	-	20		
Three subarrays after sorting	0	-	-	3	-	-	15	-	-	22	
		4	-	-	6	-	-	8	-	-	16
			1	-	-	10	-	-	20		
Data after 3-sort & before 1-sort	0	4	1	3	6	10	15	8	20	22	16
Data after 1-sort	0	1	3	4	6	8	10	15	16	20	22

Implementation

```
public void shellSort (int[] data){
   int i, j, k, h, hCnt ;
  increments[] = new int[20];
   int tmp;

   // Create an appropriate number of increments h
   for(h=1,i=0; h < data.length; i++){
      increments[i] = h;
      h = 3*h + 1;
   }
   // loop on the number of different increments h
   for(i--;i>=0;i--){
      h = increments[i];
```

Implementation…

```
        // loop on the number of subarrays h-sorted in ith pass
        for(hCnt = h; hCnt < 2*h; hCnt++){
        // insertion sort for subarray containing every hth element of
    array
          for(j = hCnt; j<data.length){
            tmp = data[j];
            k=j;
            while(k-h>=0 && tmp < data[k-h]){
           data[k] = data[k-h];
           k = k-h;
             }
            data[k] = tmp; j += h;
          }
        }
      }
}//shellsort
```

Analysis

- Large initial h makes data move by large distance.
- Files are small in initial stages (h is large).
- In later stages, files are longer, but nearly sorted.
- Hence, insertion sort does well.

Analysis

- Analysis very difficult to make.
- Empirical analysis indicates a complexity of approx $O(N^{1.25})$.
- Much better than simple insertion/selection sort.

Summary

- We saw four sorting algorithms
  - Insertion
  - Selection
  - Bubble
  - Shell
- We analyzed time and space complexities for each of them and analyzed relative trade offs.