Data Structure

Lecture#16: Internal Sorting
(Chapter 7)

U Kang
Seoul National University
In This Lecture

- Definition and evaluation measures of sorting
- Exchange sorting algorithms and their limitations
- Shellsort and how to exploit the best-case behavior of other algorithm
Sorting

- Sorting: puts elements of a list in a certain order (increasing or decreasing)
  - Many applications: scores, documents, search results, …
  - One of the most fundamental tasks in Computer Science

- Sorting in offline world
Sorting

- We will discuss many sorting algorithms
  - insertion sort, bubble sort, selection sort, shell sort, merge sort, quicksort, heap sort, bin sort, radix sort

- Measures of cost:
  - # of Comparisons
  - # of Swaps
Insertion Sort (1)

- Initially, the output is empty
- Insert each item one by one to the output
  - Insert it in a correct place to make the output in a sorted order
Insertion Sort (2)

Input

Output
Insertion Sort (3)

```
static <E extends Comparable<? super E>>
void Sort(E[] A) {
    for (int i=1; i<A.length; i++)
        for (int j=i;
             (j>0) && (A[j].compareTo(A[j-1])<0);
             j--)
            DSutil.swap(A, j, j-1);
}
```

# of Swaps, # of Comparisons

- Best Case:
- Worst Case:
- Average Case:
Insertion Sort (4)

```
static <E extends Comparable<? super E>>
void Sort(E[] A) {
    for (int i=1; i<A.length; i++)
        for (int j=i;
            (j>0) && (A[j].compareTo(A[j-1])<0);
            j--)
            DSutil.swap(A, j, j-1);
}
```

- Best Case: 0 swaps, n – 1 comparisons
- Worst Case: n²/2 swaps and comparisons
- Average Case: n²/4 swaps and comparisons

Insertion Sort is very efficient when the array is near-sorted. This characteristic is used later in other sorting algorithms.
Bubble Sort (1)

- Maybe, one of the most popular sorting algorithms
  - Appears in many computer language introduction books

- Main Idea
  - Initially, the output is empty
  - At each iteration
    - “Bubble up” the smallest element from the input to the output (= move the smallest element from the input to the output)
  - Using an array for both input and output
    - At iteration $k$, $k$ th smallest element is located in the array[$k$]
  - Given an array, how to move the smallest element to the beginning of the array?
    - One idea is to swap neighbors repeatedly, from the end of the array
## Bubble Sort (2)

<table>
<thead>
<tr>
<th>i=0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>20</td>
<td></td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>17</td>
<td></td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td>17</td>
<td>17</td>
<td>17</td>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td>28</td>
<td></td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>14</td>
<td></td>
<td>17</td>
<td>17</td>
<td>17</td>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td>23</td>
<td></td>
<td>23</td>
<td>23</td>
<td>23</td>
<td>23</td>
<td>23</td>
</tr>
<tr>
<td>15</td>
<td></td>
<td>28</td>
<td>28</td>
<td>28</td>
<td>28</td>
<td>28</td>
</tr>
</tbody>
</table>

- After each pass, the largest unsorted element is moved to its correct position.
- The process continues until the entire list is sorted.
Bubble Sort (3)

```
static <E extends Comparable<? super E>>
void Sort(E[] A) {
    for (int i=0; i<A.length-1; i++)
        for (int j=A.length-1; j>i; j--)
            if ((A[j].compareTo(A[j-1]) < 0))
                DSutil.swap(A, j, j-1);
}
```

- **Best Case:**
- **Worst Case:**
- **Average Case:**

# of Swaps, # of Comparisons
Bubble Sort (4)

```java
static <E extends Comparable<? super E>>
void Sort(E[] A) {
    for (int i=0; i<A.length-1; i++)
        for (int j=A.length-1; j>i; j--)
            if ((A[j].compareTo(A[j-1]) < 0))
                DSutil.swap(A, j, j-1);
}
```

- **Best Case:** 0 swaps, \(n^2/2\) comparisons
- **Worst Case:** \(n^2/2\) swaps and comparisons
- **Average Case:** \(n^2/4\) swaps and \(n^2/2\) comparisons
Selection Sort (1)

- Essentially, a bubble sort

- Given an array, how to move the smallest element to the beginning of the array?
  - [Bubble Sort] swap neighbors repeatedly
  - [Selection Sort] scan the array, find the smallest element, and swap it with the first item in the array
## Selection Sort (2)

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>i=0</td>
<td>42</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>20</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>17</td>
<td>17</td>
<td>17</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>42</td>
<td>42</td>
<td>42</td>
<td>17</td>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>28</td>
<td>28</td>
<td>28</td>
<td>28</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>28</td>
<td>23</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>23</td>
<td>23</td>
<td>23</td>
<td>23</td>
<td>28</td>
<td>28</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>15</td>
<td>17</td>
<td>42</td>
<td>42</td>
<td>42</td>
<td>42</td>
</tr>
</tbody>
</table>
Selection Sort (3)

static <E extends Comparable<? super E>>
void Sort(E[] A) {
    for (int i=0; i<A.length-1; i++) {
        int lowindex = i;
        for (int j=A.length-1; j>i; j--)
            if (A[j].compareTo(A[lowindex]) < 0)
                lowindex = j;
        DSutil.swap(A, i, lowindex);
    }
}

# of Swaps, # of Comparisons

- Best Case:
- Worst Case:
- Average Case:
Selection Sort (4)

```java
static <E extends Comparable<? super E>>
void Sort(E[] A) {
    for (int i=0; i<A.length-1; i++) {
        int lowindex = i;
        for (int j=A.length-1; j>i; j--)
            if (A[j].compareTo(A[lowindex]) < 0)
                lowindex = j;
        DSutil.swap(A, i, lowindex);
    }
}
```

- Best Case: 0 swaps (n-1 swaps for bad swap()), \(n^2/2\) comparisons
- Worst Case: n-1 swaps and \(n^2/2\) comparisons
- Average Case: \(O(n)\) swaps and \(n^2/2\) comparisons

**Better than Bubble sort, since # of swap is much smaller**
Your array is probably storing a reference to the record instead of the record itself.
## Summary

<table>
<thead>
<tr>
<th></th>
<th>Insertion</th>
<th>Bubble</th>
<th>Selection</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Comparisons</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Best Case</td>
<td>(\Theta(n))</td>
<td>(\Theta(n^2))</td>
<td>(\Theta(n^2))</td>
</tr>
<tr>
<td>Average Case</td>
<td>(\Theta(n^2))</td>
<td>(\Theta(n^2))</td>
<td>(\Theta(n^2))</td>
</tr>
<tr>
<td>Worst Case</td>
<td>(\Theta(n^2))</td>
<td>(\Theta(n^2))</td>
<td>(\Theta(n^2))</td>
</tr>
<tr>
<td><strong>Swaps</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Best Case</td>
<td>0</td>
<td>0</td>
<td>0 or (\Theta(n))</td>
</tr>
<tr>
<td>Average Case</td>
<td>(\Theta(n^2))</td>
<td>(\Theta(n^2))</td>
<td>(\Theta(n))</td>
</tr>
<tr>
<td>Worst Case</td>
<td>(\Theta(n^2))</td>
<td>(\Theta(n^2))</td>
<td>(\Theta(n))</td>
</tr>
</tbody>
</table>
Exchange Sorting

- All of the sorting algorithms so far rely on exchanges of *adjacent* records.
  - Thus, they are called “exchange sorting” algorithms

- What is the average number of exchanges required in any exchange sorting of $n$ items?
  - There are $n!$ permutations
  - Consider a permutation $X$ and its reverse, $X'$
  - Together, all pairs require $n(n-1)/2$ exchanges (or “inversion”) in total.
  - On average, each permutation requires $n(n-1)/4 = \Omega(n^2)$ exchanges
Shell Sort (1)

Main idea

- Task: sort an array $x$ of size $n$
- Consider the following two sub arrays from $x$
  - $x_e$ (contains elements whose indexes are even)
  - $x_o$ (contains elements whose indexes are odd)
- Assume $x_e$ and $x_o$ are sorted, respectively
- Then, insertion sort on $x$ would be efficient (why?)
- Now, recursively consider the above process on the two subarrays

- Shell sort: go backward from the end of the above process
Shell Sort (2)

Procedure

- **Pass 1**
  - Make \( n/2 \) sublists of 2 elements each, where the array index of the 2 elements differs by \( n/2 \)
    - E.g., for \( n = 16 \), make 8 sublists: (0, 8), (1, 9), …, (7, 15)
  - Each list of 2 elements is sorted using Insertion Sort

- **Pass 2**
  - Make \( n/4 \) sublists of 4 elements each, where the array index of the 4 elements differs by \( n/4 \)
    - E.g., for \( n = 16 \), make 4 sublists: (0, 4, 8, 12), (1, 5, 9, 13), …
  - Each list of 4 elements is sorted using Insertion Sort

- …
Shell Sort (3)

- **Main Idea**
  - **Pass 3**
    - Make \( n/8 \) sublists of 8 elements each, where the array index of the 8 elements differs by \( n/8 \)
      - E.g., for \( n = 16 \), make 2 sublists: (even numbers), (odd numbers)
    - Each list of 8 elements is sorted using Insertion Sort
  - ... Final Pass (Pass (\( \log n \)))
    - Make 1 sublist of \( n \) elements (do nothing), and sort the sublist using insertion sort (apply the standard insertion sort on the array)
Shell Sort (4)
Shell Sort (5)

```java
static <E extends Comparable<? super E>>
void Sort(E[] A) {
    for (int i=A.length/2; i>=2; i/=2)
        for (int j=0; j<i; j++)
            inssort2(A, j, i);
    inssort2(A, 0, 1);
}

/** Modified version of Insertion Sort for varying increments */
static <E extends Comparable<? super E>>
void inssort2(E[] A, int start, int incr) {
    for (int i=start+incr; i<A.length; i+=incr)
        for (int j=i;(j >= start+incr) &&
            (A[j].compareTo(A[j-incr])<0);
                j-=incr)
        DSutil.swap(A, j, j-incr);
}
```
Shell Sort (6)

- **Correctness**: Shellsort always sorts an array correctly. Why?
  - Since it performs the insertion sort at the end

- **Efficiency**: Is Shellsort better than Insertion Sort?
  - Yes (in most cases), since each insertion sort operates on an “almost sorted” array
  - Fact: average-case performance of ShellSort takes $O(n^{1.5})$, which is much efficient than Insertion Sort
What you need to know

- **Sorting**: puts elements in a certain order
  - Evaluation: # of swaps, # of comparisons

- **Exchange sorting algorithms**
  - Insertion sort, bubble sort, and selection sort
  - Cost and limitations

- **Shellsort**
  - Main ideas
  - How it exploits insertion sort
Questions?