Selected Sorting Algorithms

CS 165: Project in Algorithms and Data Structures
Michael T. Goodrich

Some slides are from J. Miller, CSE 373, U. Washington
Why Sorting?

• Practical application
  – People by last name
  – Countries by population
  – Search engine results by relevance

• Fundamental to other algorithms

• Different algorithms have different asymptotic and constant-factor trade-offs
  – No single ‘best’ sort for all scenarios
  – Knowing one way to sort just isn’t enough

• Many to approaches to sorting which can be used for other problems
Problem statement

There are $n$ comparable elements in an array and we want to rearrange them to be in increasing order

Pre:
– An array $A$ of data records
– A value in each data record
– A comparison function
  • $<, =, >$, compareTo

Post:
– For each distinct position $i$ and $j$ of $A$, if $i < j$ then $A[i] \leq A[j]$
– $A$ has all the same data it started with
Insertion sort

- **insertion sort**: orders a list of values by repetitively inserting a particular value into a sorted subset of the list

- more specifically:
  - ✓ consider the first item to be a sorted sublist of length 1
  - ✓ insert the second item into the sorted sublist, shifting the first item if needed
  - ✓ insert the third item into the sorted sublist, shifting the other items as needed
  - ✓ repeat until all values have been inserted into their proper positions
Insertion sort

- Simple sorting algorithm.
  - $n-1$ passes over the array
  - At the end of pass $i$, the elements that occupied $A[0]...A[i]$ originally are still in those spots and in sorted order.

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>15</th>
<th>8</th>
<th>1</th>
<th>17</th>
<th>10</th>
<th>12</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>after</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pass 2</td>
<td>2</td>
<td>8</td>
<td>15</td>
<td>1</td>
<td>17</td>
<td>10</td>
<td>12</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>8</th>
<th>15</th>
<th>17</th>
<th>10</th>
<th>12</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>after</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pass 3</td>
<td>1</td>
<td>2</td>
<td>8</td>
<td>15</td>
<td>17</td>
<td>10</td>
<td>12</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>
Insertion sort example

3 is sorted.

3 and 9 are sorted.
Shift 9 to the right. Insert 6.

3, 6, and 9 are sorted.
Shift 9, 6, and 3 to the right. Insert 1.

1, 3, 6, and 9 are sorted.
Shift 9, 6, and 3 to the right. Insert 2.
public static void insertionSort(int[] a) {
    for (int i = 1; i < a.length; i++) {
        int temp = a[i];
        int j = i;
        while (j > 0 && a[j - 1] > temp) {
            a[j] = a[j - 1];
            j--;
        }
        a[j] = temp;
    }
}
Insertion-sort Analysis

• An **inversion** in a permutation is the number of pairs that are out of order, that is, the number of pairs, \((i,j)\), such that \(i < j\) but \(x_i > x_j\).

• Each step of insertion-sort fixes an inversion or stops the while-loop.

• Thus, the running time of insertion-sort is \(O(n + k)\), where \(k\) is the number of inversions.
Insertion-sort Analysis

• The worst case for the number of inversions, \( k \), is

\[
\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \Theta(n^2)
\]

• This occurs for a list in reverse-sorted order.
Insertion-sort Analysis

- The average case for $k$ is

\[ x = \text{smallest element} \]

\[ \underbrace{A(\frac{n}{2})}_{\text{remove it}} \rightarrow A(n-1) \]

\[ E(\Sigma) = 0 \cdot \frac{1}{n} + 1 \cdot \frac{1}{n} + 2 \cdot \frac{1}{n} + \cdots + (n-1) \frac{1}{n} \]

\[ = \frac{1}{n} \sum_{k=1}^{n-1} k = \frac{1}{n} \left( \frac{n(n-1)}{2} \right) \]
\[ A(n) = A(n-1) + \frac{n-1}{2} \]

\[ = \sum_{i=1}^{\frac{n}{2}} i \]

\[ = \frac{n(n-1)}{4} \]

\[ = O(n^2) \]
Shell sort description

• **shell sort**: orders a list of values by comparing elements that are separated by a gap of >1 indexes
  – a generalization of insertion sort
  – invented by computer scientist Donald Shell in 1959

• based on some observations about insertion sort:
  ✓ – insertion sort runs fast if the input is almost sorted
  ✓ – insertion sort's weakness is that it swaps each element just one step at a time, taking many swaps to get the element into its correct position
Shell sort example

• Idea: Sort all elements that are 5 indexes apart, then sort all elements that are 3 indexes apart, ...
public static void shellSort(int[] a) {
    for (int gap = a.length / 2; gap > 0; gap /= 2) {
        for (int i = gap; i < a.length; i++) {
            // slide element i back by gap indexes
            // until it's "in order"
            int temp = a[i];
            int j = i;
            while (j >= gap && temp < a[j - gap]) {
                a[j] = a[j - gap];
                j -= gap;
            }
            a[j] = temp;
        }
    }
}
Shell sort Analysis

• Harder than insertion sort
• But certainly no worse than insertion sort
• Worst-case: $O(n^2)$
• Average-case: ????

Experiments
Divide-and-Conquer

- Divide-and conquer is a general algorithm design paradigm:
  - Divide: divide the input data $S$ in two disjoint subsets $S_1$ and $S_2$
  - Recur: solve the subproblems associated with $S_1$ and $S_2$
  - Conquer: combine the solutions for $S_1$ and $S_2$ into a solution for $S$

- The base case for the recursion are subproblems of size 0 or 1
Merge-Sort

- Merge-sort is a sorting algorithm based on the divide-and-conquer paradigm
  - It has $O(n \log n)$ running time
The Merge-Sort Algorithm

- Merge-sort on an input sequence $S$ with $n$ elements consists of three steps:
  - Divide: partition $S$ into two sequences $S_1$ and $S_2$ of about $n/2$ elements each
  - Recur: recursively sort $S_1$ and $S_2$
  - Conquer: merge $S_1$ and $S_2$ into a unique sorted sequence

Algorithm mergeSort(S):

- **Input** array $S$ of $n$ elements
- **Output** array $S$ sorted

if $n > 1$ then
  
  $(S_1, S_2) \leftarrow \text{partition}(S, n/2)$
  
  mergeSort($S_1$)
  
  mergeSort($S_2$)
  
  $S \leftarrow \text{merge}(S_1, S_2)$
Merging Two Sorted Sequences

• The conquer step of merge-sort consists of merging two sorted sequences $A$ and $B$ into a sorted sequence $S$ containing the union of the elements of $A$ and $B$.

• Merging two sorted sequences, each with $n/2$ elements and implemented by means of a doubly linked list, takes $O(n)$ time.

Algorithm merge($S_1, S_2, S$):

Input: Two arrays, $S_1$ and $S_2$, of size $n_1$ and $n_2$, respectively, sorted in non-decreasing order, and an empty array, $S$, of size at least $n_1 + n_2$

Output: $S$, containing the elements from $S_1$ and $S_2$ in sorted order

1. $i \leftarrow 1$
2. $j \leftarrow 1$
3. While $i \leq n$ and $j \leq n$ do
   4. If $S_1[i] \leq S_2[j]$ then
      5. $S[i + j - 1] \leftarrow S_1[i]$
      6. $i \leftarrow i + 1$
   7. Else
      8. $S[i + j - 1] \leftarrow S_2[j]$
      9. $j \leftarrow j + 1$
10. While $i \leq n$ do
   11. $S[i + j - 1] \leftarrow S_1[i]$
   12. $i \leftarrow i + 1$
13. While $j \leq n$ do
   14. $S[i + j - 1] \leftarrow S_2[j]$
   15. $j \leftarrow j + 1$
Merge-Sort Tree

- An execution of merge-sort is depicted by a binary tree
  - each node represents a recursive call of merge-sort and stores
    - unsorted sequence before the execution and its partition
    - sorted sequence at the end of the execution
  - the root is the initial call
  - the leaves are calls on subsequences of size 0 or 1
Analysis of Merge-Sort

• The height \( h \) of the merge-sort tree is \( O(\log n) \)
  - at each recursive call we divide in half the sequence,
• The overall amount or work done at the nodes of depth \( i \) is \( O(n) \)
  - we partition and merge \( 2^i \) sequences of size \( n/2^i \)
  - we make \( 2^{i+1} \) recursive calls
• Thus, the best/worst/average running time of merge-sort is \( O(n \log n) \)
A Hybrid Sorting Algorithm

- A **hybrid** sorting algorithm is a blending of two different sorting algorithms, typically, a divide-and-conquer algorithm, like merge-sort, combined with an incremental algorithm, like insertion-sort.
- The algorithm is parameterized with hybridization value, $H$, and an example with merge-sort and insertion-sort would work as follow:
  - Start out performing merge-sort, but switch to insertion sort when the problem size goes below H.
A Hybrid Sorting Algorithm

• Pseudo-code:

• Running time:
  – Depends on $H$
  – Interesting experiments:
    1. $H = n^{1/2}$
    2. $H = n^{1/3}$
    3. $H = n^{1/4}$

Algorithm HybridMergeSort($S, H$):

Input array $S$ of $n$ elements
Output array $S$ sorted

if $n \geq H$ then
  $(S_1, S_2) \leftarrow \text{partition}(S, n/2)$
  HybridMergeSort($S_1, H$)
  HybridMergeSort($S_2, H$)
  $S \leftarrow \text{merge}(S_1, S_2)$
else
  InsertionSort($S$)
Hybrid Merge-sort Analysis

• Hint: combine the tree-based merge-sort analysis and the insertion-sort analysis...