CSE 373: Data Structures and Algorithms

Lecture 6: Sorting
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
# Sorting Classification

<table>
<thead>
<tr>
<th>In memory sorting</th>
<th>External sorting</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Comparison sorting</strong></td>
<td><strong>Specialized Sorting</strong></td>
</tr>
<tr>
<td>$\Omega(N \log N)$</td>
<td></td>
</tr>
<tr>
<td><strong>O($N^2$)</strong></td>
<td><strong>O($N \log N$)</strong></td>
</tr>
<tr>
<td><strong>O($N$)</strong></td>
<td><strong># of tape accesses</strong></td>
</tr>
</tbody>
</table>

- Bubble Sort
- Selection Sort
- Insertion Sort
- Shellsort Sort
- Merge Sort
- Quick Sort
- Heap Sort
- Bucket Sort
- Radix Sort
- Simple External Merge Sort
- Variations

in place? stable?
Comparison Sorting

comparison-based sorting: determine order through comparison operations on the input data:
<, >, compareTo, ...
• **bogo sort**: orders a list of values by repetitively shuffling them and checking if they are sorted

• more specifically:
  – scan the list, seeing if it is sorted
  – if not, shuffle the values in the list and repeat

• This sorting algorithm has terrible performance!
  – Can we deduce its runtime?
Bogo sort code

```java
public static void bogoSort(int[] a) {
    while (!isSorted(a)) {
        shuffle(a);
    }
}

// Returns true if array a's elements
// are in sorted order.
public static boolean isSorted(int[] a) {
    for (int i = 0; i < a.length - 1; i++) {
        if (a[i] > a[i+1]) {
            return false;
        }
    }
    return true;
}
```

// Shuffles an array of ints by randomly swapping each
// element with an element ahead of it in the array.
public static void shuffle(int[] a) {
    for (int i = 0; i < a.length - 1; i++) {
        // pick random number in [i+1, a.length-1] inclusive
        int range = a.length-1 - (i + 1) + 1;
        int j = (int)(Math.random() * range + (i + 1));
        swap(a, i, j);
    }
}

// Swaps a[i] with a[j].
private static void swap(int[] a, int i, int j) {
    if (i == j)
        return;

    int temp = a[i];
a[i] = a[j];
a[j] = temp;
}
Bogo sort runtime

• How long should we expect bogo sort to take?
  – related to probability of shuffling into sorted order
  – assuming shuffling code is fair, probability equals $1 / (\text{number of permutations of } n \text{ elements})$

$$P_n^n = n!$$

– bogo sort takes roughly factorial time to run
  • note that if array is initially sorted, bogo finishes quickly!
– it should be clear that this is not satisfactory...
$O(n^2)$ Comparison Sorting
Bubble sort

• **bubble sort**: orders a list of values by repetitively comparing neighboring elements and swapping their positions if necessary

• more specifically:
  – scan the list, exchanging adjacent elements if they are not in relative order; this bubbles the highest value to the top
  – scan the list again, bubbling up the second highest value
  – repeat until all elements have been placed in their proper order
"Bubbling" largest element

- Traverse a collection of elements
  - Move from the front to the end
  - "Bubble" the largest value to the end using pair-wise comparisons and swapping

![Diagram showing array elements and highlighted elements for swapping]

- Example:
  - Initial array: [1, 2, 3, 4, 5, 6]
  - After first pass: [1, 2, 77, 42, 101, 5]
"Bubbling" largest element

- Traverse a collection of elements
  - Move from the front to the end
  - "Bubble" the largest value to the end using pairwise comparisons and swapping
"Bubbling" largest element

• Traverse a collection of elements
  – Move from the front to the end
  – "Bubble" the largest value to the end using pair-wise comparisons and swapping
"Bubbling" largest element

• Traverse a collection of elements
  – Move from the front to the end
  – "Bubble" the largest value to the end using pair-wise comparisons and swapping

1 2 3 4 5 6
42 35 12 77 101 5

No need to swap
"Bubbling" largest element

• Traverse a collection of elements
  – Move from the front to the end
  – "Bubble" the largest value to the end using pairwise comparisons and swapping
"Bubbling" largest element

- Traverse a collection of elements
  - Move from the front to the end
  - "Bubble" the largest value to the end using pairwise comparisons and swapping

Largest value correctly placed
public static void bubbleSort(int[] a) {
    for (int i = 0; i < a.length; i++) {
        for (int j = 1; j < a.length - i; j++) {
            // swap adjacent out-of-order elements
            if (a[j-1] > a[j]) {
                swap(a, j-1, j);
            }
        }
    }
}
Bubble sort runtime

• Running time (# comparisons) for input size $n$:

$$\sum_{i=0}^{n-1} \sum_{j=1}^{n-i} 1 = \sum_{i=0}^{n-1} (n - i)$$

$$= \sum_{i=0}^{n-1} i$$

$$= \frac{(n - 1)n}{2}$$

$$= \mathcal{O}(n^2)$$

— number of actual swaps performed depends on the data; out-of-order data performs many swaps
Selection sort

- **selection sort**: orders a list of values by repetitively putting a particular value into its final position

- more specifically:
  - find the smallest value in the list
  - switch it with the value in the first position
  - find the next smallest value in the list
  - switch it with the value in the second position
  - repeat until all values are in their proper places
Selection sort example

Scan right starting with 3.
1 is the smallest. Exchange 1 and 3.

Scan right starting with 9.
2 is the smallest. Exchange 9 and 2.

Scan right starting with 6.
3 is the smallest. Exchange 6 and 3.

Scan right starting with 6.
6 is the smallest. Exchange 6 and 6.
## Selection sort example 2

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>27</td>
<td>63</td>
<td>1</td>
<td>72</td>
<td>64</td>
<td>58</td>
<td>14</td>
<td>9</td>
</tr>
<tr>
<td>1&lt;sup&gt;st&lt;/sup&gt; pass</td>
<td>1</td>
<td>63</td>
<td>27</td>
<td>72</td>
<td>64</td>
<td>58</td>
<td>14</td>
<td>9</td>
</tr>
<tr>
<td>2&lt;sup&gt;nd&lt;/sup&gt; pass</td>
<td>1</td>
<td>9</td>
<td>27</td>
<td>72</td>
<td>64</td>
<td>58</td>
<td>14</td>
<td>63</td>
</tr>
<tr>
<td>3&lt;sup&gt;rd&lt;/sup&gt; pass</td>
<td>1</td>
<td>9</td>
<td>14</td>
<td>72</td>
<td>64</td>
<td>58</td>
<td>27</td>
<td>63</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Selection sort code

```java
public static void selectionSort(int[] a) {
    for (int i = 0; i < a.length; i++) {
        // find index of smallest element
        int min = i;
        for (int j = i + 1; j < a.length; j++) {
            if (a[j] < a[min]) {
                min = j;
            }
        }

        // swap smallest element with a[i]
        swap(a, i, min);
    }
}
```
Selection sort runtime

• Running time for input size \( n \):
  
  — in practice, a bit faster than bubble sort. Why?

\[
\sum_{i=0}^{n-1} \sum_{j=i+1}^{n} 1 = \sum_{i=0}^{n-1} (n - (i + 1) + 1)
\]

\[
= \sum_{i=0}^{n-1} (n - i)
\]

\[
= \sum_{i=0}^{n-1} i
\]

\[
= \frac{(n - 1)n}{2}
\]

\[
= O(n^2)
\]
**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>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</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>after</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>
<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];

        // slide elements down to make room for a[i]
        int j = i;
        while (j > 0 && a[j - 1] > temp) {
            a[j] = a[j - 1];
            j--;
        }

        a[j] = temp;
    }
}
Insertion sort runtime

• worst case: reverse-ordered elements in array.
  \[ \sum_{i=1}^{n-1} i = 1 + 2 + 3 + \ldots + (n-1) = \frac{(n-1)n}{2} \]
  \[ = O(n^2) \]

• best case: array is in sorted ascending order.
  \[ \sum_{i=1}^{n-1} 1 = n - 1 = O(n) \]

• average case: each element is about halfway in order.
  \[ \sum_{i=1}^{n-1} \frac{i}{2} = \frac{1}{2} (1 + 2 + 3 \ldots + (n-1)) = \frac{(n-1)n}{4} \]
  \[ = O(n^2) \]
Comparing sorts

- We've seen "simple" sorting algos. so far, such as:
  - selection sort
  - insertion sort

<table>
<thead>
<tr>
<th></th>
<th>comparisons</th>
<th>swaps</th>
</tr>
</thead>
<tbody>
<tr>
<td>selection</td>
<td>$n^2/2$</td>
<td>$n$</td>
</tr>
<tr>
<td>insertion</td>
<td>worst: $n^2/2$</td>
<td>worst: $n^2/2$</td>
</tr>
<tr>
<td></td>
<td>best: $n$</td>
<td>best: $n$</td>
</tr>
</tbody>
</table>

- They all use nested loops and perform approximately $n^2$ comparisons
- They are relatively inefficient
Average case analysis

• Given an array $A$ of elements, an *inversion* is an ordered pair $(i, j)$ such that $i < j$, but $A[i] > A[j]$. (out of order elements)
• Assume no duplicate elements.

• Theorem: The average number of inversions in an array of $n$ distinct elements is $n (n - 1) / 4$.
• Corollary: Any algorithm that sorts by exchanging adjacent elements requires $O(n^2)$ time on average.
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 code
Sorting practice problem

• Consider the following array of int values.

\[ [22, 11, 34, -5, 3, 40, 9, 16, 6] \]

(a) Write the contents of the array after 3 passes of the outermost loop of bubble sort.

(b) Write the contents of the array after 5 passes of the outermost loop of insertion sort.

(c) Write the contents of the array after 4 passes of the outermost loop of selection sort.

(d) Write the contents of the array after a pass of bogo sort. (Just kidding.)
$O(n \log n)$ Comparison Sorting
Merge sort

• **merge sort**: orders a list of values by recursively dividing the list in half until each sub-list has one element, then recombining
  – Invented by John von Neumann in 1945

• more specifically:
  – divide the list into two roughly equal parts
  – recursively divide each part in half, continuing until a part contains only one element
  – merge the two parts into one sorted list
  – continue to merge parts as the recursion unfolds

• This is a "divide and conquer" algorithm.
Merge sort

- Merge sort idea:
  - Divide the array into two halves.
  - Recursively sort the two halves (using merge sort).
  - Use merge to combine the two arrays.

\[
\text{mergeSort}(0, \frac{n}{2}-1) \quad \text{mergeSort}(\frac{n}{2}, n-1)
\]

\[
\text{sort} \quad \text{sort}
\]

\[
\text{merge}(0, \frac{n}{2}, n-1)
\]
<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>98</td>
<td>23</td>
<td>45</td>
<td>14</td>
<td>6</td>
<td>67</td>
<td>33</td>
<td>42</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>98</td>
<td>23</td>
<td>45</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>98</td>
<td>23</td>
<td></td>
<td></td>
<td></td>
<td>45</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>98</td>
<td>23</td>
<td></td>
<td>45</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>23</td>
<td>98</td>
<td></td>
<td>14</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Merge**
Merge
98 23 45 14 6 67 33 42

98 23 45 14

98 23 45 14

98 23 45 14

23 98 14 45

14 23 45

Merge
Merge
Merge
Merge
98 23 45 14 6 67 33 42
98 23 45 14 6 67 33 42
98 23 45 14 6 67 33 42
98 23 45 14 6 67 33 42
23 98 14 45 6 67 33 42
14 23 45 98

Merge
<table>
<thead>
<tr>
<th>98</th>
<th>23</th>
<th>45</th>
<th>14</th>
<th>6</th>
<th>67</th>
<th>33</th>
<th>42</th>
</tr>
</thead>
<tbody>
<tr>
<td>98</td>
<td>23</td>
<td>45</td>
<td>14</td>
<td></td>
<td>6</td>
<td>67</td>
<td>33</td>
</tr>
<tr>
<td>98</td>
<td>23</td>
<td>45</td>
<td>14</td>
<td></td>
<td>6</td>
<td>67</td>
<td>33</td>
</tr>
<tr>
<td>98</td>
<td>23</td>
<td>45</td>
<td>14</td>
<td></td>
<td>6</td>
<td>67</td>
<td>33</td>
</tr>
<tr>
<td>23</td>
<td>98</td>
<td>14</td>
<td>45</td>
<td></td>
<td>6</td>
<td>67</td>
<td>33</td>
</tr>
<tr>
<td>14</td>
<td>23</td>
<td>45</td>
<td>98</td>
<td></td>
<td>6</td>
<td>33</td>
<td>42</td>
</tr>
<tr>
<td>6</td>
<td>14</td>
<td>23</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Merge
Merge
Merge sort example 2

<table>
<thead>
<tr>
<th></th>
<th>13</th>
<th>6</th>
<th>21</th>
<th>18</th>
<th>9</th>
<th>4</th>
<th>8</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>9</td>
<td>4</td>
<td>8</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>13</td>
<td>6</td>
<td></td>
<td></td>
<td>9</td>
<td>4</td>
<td>8</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>21</td>
<td></td>
<td>18</td>
<td></td>
<td>8</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>13</td>
<td>6</td>
<td>21</td>
<td>18</td>
<td>9</td>
<td>4</td>
<td>8</td>
<td>20</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>13</td>
<td>18</td>
<td>21</td>
<td>8</td>
<td>20</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>8</td>
<td>9</td>
<td>20</td>
<td>6</td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>9</td>
<td>13</td>
<td>18</td>
<td>20</td>
<td>21</td>
</tr>
</tbody>
</table>

0
Merging two sorted arrays

• *merge* operation:
  – Given two sorted arrays, *merge* operation produces a sorted array with all the elements of the two arrays.

<table>
<thead>
<tr>
<th>A</th>
<th>6</th>
<th>13</th>
<th>18</th>
<th>21</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>4</td>
<td>8</td>
<td>9</td>
<td>20</td>
</tr>
<tr>
<td>C</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

Running time of *merge*: $O(n)$, where $n$ is the number of elements in the merged array.

When merging two sorted parts of the same array, we'll need a *temporary array* to store the merged whole.
public static void mergeSort(int[] a) {
    int[] temp = new int[a.length];
    mergeSort(a, temp, 0, a.length - 1);
}

private static void mergeSort(int[] a, int[] temp, int left, int right) {
    if (left >= right) {  // base case
        return;
    }

    // sort the two halves
    int mid = (left + right) / 2;
    mergeSort(a, temp, left, mid);
    mergeSort(a, temp, mid + 1, right);

    // merge the sorted halves into a sorted whole
    merge(a, temp, left, right);
}
private static void merge(int[] a, int[] temp, int left, int right) {
    int mid = (left + right) / 2;
    int count = right - left + 1;
    int l = left;                  // counter indexes for L, R
    int r = mid + 1;

    // main loop to copy the halves into the temp array
    for (int i = 0; i < count; i++) {
        if (r > right) {           // finished right; use left
            temp[i] = a[l++];
        } else if (l > mid) {      // finished left; use right
            temp[i] = a[r++];
        } else if (a[l] < a[r]) {  // left is smaller (better)
            temp[i] = a[l++];
        } else {                   // right is smaller (better)
            temp[i] = a[r++];
        }
    }

    // copy sorted temp array back into main array
    for (int i = 0; i < count; i++) {
        a[left + i] = temp[i];
    }
}

Merge code
Merge sort runtime

• let $T(n)$ be runtime of merge sort on $n$ items
  – $T(0) = 1$
  – $T(1) = 2 \times T(0) + 1$
  – $T(2) = 2 \times T(1) + 2$
  – $T(4) = 2 \times T(2) + 4$
  – $T(8) = 2 \times T(4) + 8$
  – ... 
  – $T(n/2) = 2 \times T(n/4) + n/2$
  – $T(n) = 2 \times T(n/2) + n$
Merge sort runtime

- $T(n) = 2T(n/2) + n$
- $T(n/2) = 2T(n/4) + n/2$

- $T(n) = 2(2T(n/4) + n/2) + n$
- $T(n) = 4T(n/4) + 2n$
- $T(n) = 8T(n/8) + 3n$
- ...
- $T(n) = 2^k T(n/2^k) + kn$

To get to a more simplified case, let's set $k = \log_2 n$.

- $T(n) = 2^{\log n} T(n/2^{\log n}) + (\log n) n$
- $T(n) = n \ast T(n/n) + n \log n$
- $T(n) = n \ast T(1) + n \log n$
- $T(n) = n \ast 1 + n \log n$
- $T(n) = n + n \log n$
- $T(n) = O(n \log n)$
Sorting practice problem

• Consider the following array of int values.

\[22, 11, 34, -5, 3, 40, 9, 16, 6]\n
(e) Write the contents of the array after all the recursive calls of merge sort have finished (before merging).