Final Exam Stuff

CS170 Final Exam: **April 30th, 6:30-9:00 PM, White Hall 208**
(Except for those with conflicts, who should have received an e-mail with the date, time, and location for their CS170 Final)

CS170 Review Session (note the new time and room!!):
   **Wednesday April 29th, 3:00-5:00 PM, MSC w201**
This will be a joint review session for all sections given by Dr. Swenson and Clarissa Garvey

Final Exam Study Guide now available via course calendar page!!
Lecture 32:
Merge Sort
And analysis of algorithms
Apr 17 2015
Merging

• The key to Merge Sort is merging two sorted lists into one, such that if you have two lists \( X (x_1 \leq x_2 \leq \cdots \leq x_m) \) and \( Y(y_1 \leq y_2 \leq \cdots \leq y_n) \) the resulting list is \( Z(z_1 \leq z_2 \leq \cdots \leq z_{m+n}) \)

• Example:

\[ L_1 = \{3 \ 8 \ 9\} \quad L_2 = \{1 \ 5 \ 7\} \]

\[ \text{merge}(L_1, L_2) = \{1 \ 3 \ 5 \ 7 \ 8 \ 9\} \]
Merging (cont.)

X: 3 10 23 54
Y: 1 5 25 75

Result:
Merging (cont.)

X: 3 10 23 54

Y: 5 25 75

Result: 1
Merging (cont.)

X: [10, 23, 54]  Y: [5, 25, 75]

Result: [1, 3]
Merging (cont.)

X: 10 23 54

Y: 25 75

Result: 1 3 5
Merging (cont.)

X: [ ] [23] [54] Y: [ ] [25] [75]

Result: [1] [3] [5] [10] [ ] [ ] [ ] [ ]
Merging (cont.)

X: 

54

Y: 

25 75

Result: 

1 3 5 10 23
Merging (cont.)

X: [ ] [ ] [ ] 54
Y: [ ] [ ] [ ] 75

Result: [ ] [ ] [ ] [ ] [ ] 1 3 5 10 23 25
Merging (cont.)

X: [ ] [ ] [ ]

Y: [ ] [ ] [ ] 75

Result: 1 3 5 10 23 25 54
Merging (cont.)

X: 

Y: 

Result: 1 3 5 10 23 25 54 75
Divide And Conquer

- Merging a two lists of one element each is the same as sorting them.

- Merge sort divides up an unsorted list until the above condition is met and then sorts the divided parts back together in pairs.

- Specifically this can be done by recursively dividing the unsorted list in half, merge sorting the right side then the left side and then merging the right and left back together.
Merge Sort Algorithm

Given a list L with a length k:

• If \( k == 1 \) \( \rightarrow \) the list is sorted

• Else:
  – Merge Sort the left side (1 thru \( k/2 \))
  – Merge Sort the right side (\( k/2+1 \) thru \( k \))
  – Merge the right side with the left side
### Merge Sort Example

<table>
<thead>
<tr>
<th>99</th>
<th>6</th>
<th>86</th>
<th>15</th>
<th>58</th>
<th>35</th>
<th>86</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
</table>
## Merge Sort Example

<table>
<thead>
<tr>
<th>99</th>
<th>6</th>
<th>86</th>
<th>15</th>
<th>58</th>
<th>35</th>
<th>86</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>99</td>
<td>6</td>
<td>86</td>
<td>15</td>
<td>58</td>
<td>35</td>
<td>86</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>
## Merge Sort Example

<table>
<thead>
<tr>
<th>99</th>
<th>6</th>
<th>86</th>
<th>15</th>
<th>58</th>
<th>35</th>
<th>86</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>99</td>
<td>6</td>
<td>86</td>
<td>15</td>
<td>58</td>
<td>35</td>
<td>86</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>99</td>
<td>6</td>
<td>86</td>
<td>15</td>
<td>58</td>
<td>35</td>
<td>86</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>99</td>
<td>6</td>
<td>86</td>
<td>15</td>
<td>58</td>
<td>35</td>
<td>86</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>
Merge Sort Example

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>99</td>
<td>6</td>
<td>86</td>
<td>15</td>
<td>58</td>
<td>35</td>
<td>86</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>99</td>
<td>6</td>
<td>86</td>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>99</td>
<td>6</td>
<td></td>
<td></td>
<td>86</td>
<td>15</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>99</td>
<td>6</td>
<td>86</td>
<td>15</td>
<td>58</td>
<td>35</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>99</td>
<td>6</td>
<td>86</td>
<td>15</td>
<td>58</td>
<td>35</td>
<td>86</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>
Merge Sort Example

99  6  86  15  58  35  86  4  0

99  6  86  15
58  35  86  4  0

99  6  86  15
58  35
86  4  0

99  6  86  15
58  35
86  4  0

4  0
Merge Sort Example

Merge
Merge Sort Example

Merge
Merge Sort Example

Merge
Merge Sort Example

0 4 6 15 35 58 86 86 99

6 15 86 99 0 4 35 58 86

Merge
## Merge Sort Example

<table>
<thead>
<tr>
<th>0</th>
<th>4</th>
<th>6</th>
<th>15</th>
<th>35</th>
<th>58</th>
<th>86</th>
<th>86</th>
<th>99</th>
</tr>
</thead>
</table>
Implementing Merge Sort

- There are two basic ways to implement merge sort:
  - In Place: Merging is done with only the input array
    - Pro: Requires only the space needed to hold the array
    - Con: Takes longer to merge because if the next element is in the right side then all of the elements must be moved down.
  - Double Storage: Merging is done with a temporary array of the same size as the input array.
    - Pro: Faster than In Place since the temp array holds the resulting array until both left and right sides are merged into the temp array, then the temp array is appended over the input array.
    - Con: The memory requirement is doubled.
The actual code

```java
private void recMergeSort(int[] workspace, int lowerBound, int upperBound) {
    if (lowerBound == upperBound) return;
    int mid = (lowerBound + upperBound) / 2;
    recMergeSort(workspace, lowerBound, mid);
    recMergeSort(workspace, mid + 1, upperBound);
    merge(workspace, lowerBound, mid + 1, upperBound);
}
```

from Lafore’s, “Data Structures and Algorithms in Java”
The *merge* method

- The merge method is too ugly to show
- It requires:
  - The index of the first element in the left subarray
  - The index of the last element in the right subarray
  - The index of the first element in the right subarray
  - That is, the dividing line between the two subarrays
  - The index of the next value in the left subarray
  - The index of the next value in the right subarray
  - The index of the destination in the workspace

- But *conceptually*, it isn't difficult!
Analysis of the merge operation

• The basic operation is: compare two elements, move one of them to the workspace array
  – This takes constant time
• We do this comparison \( n \) times
  – Hence, the whole thing takes \( O(n) \) time
• Now we move the \( n \) elements back to the original array
  – Each move takes constant time
  – Hence moving all \( n \) elements takes \( O(n) \) time
• Total time: \( O(n) + O(n) = O(n) \)
But wait...there’s more

• So far, we’ve found that it takes $O(n)$ time to merge two sorted halves of an array.
• How did these subarrays get sorted?
• At the next level down, we had four subarrays, and we merged them pairwise.

Since merging arrays is linear time, when we cut the array size in half, we cut the time in half.
• But there are two arrays of size $n/2$.
• Hence, all merges combined at the next lower level take the same amount of time as a single merge at the upper level.
Analysis II

• So far we have seen that it takes
  - $O(n)$ time to merge two subarrays of size $n/2$
  - $O(n)$ time to merge four subarrays of size $n/4$
    into two subarrays of size $n/2$
  - $O(n)$ time to merge eight subarrays of size $n/8$
    into four subarrays of size $n/4$
  - Etc.

• How many levels deep do we have to proceed?

• How many times can we divide an array of size $n$ into two halves? (answer: $\log_2 n$)
Analysis III

• So if our recursion goes $\log n$ levels deep...
• ...and we do $O(n)$ work at each level...
• ...our total time is: $\log n \times O(n)$...
• ...or in other words, $O(n \log n)$

• For large arrays, this is much better than Bubblesort, Selection sort, or Insertion sort, all of which are $O(n^2)$
• Mergesort does, however, require a “workspace” array as large as our original array
Stable sort algorithms

- A stable sort keeps equal elements in the same order.
- This may matter when you are sorting data according to some characteristic.
- Example: sorting students by test scores.

<table>
<thead>
<tr>
<th>Name</th>
<th>Score</th>
<th>Original</th>
<th>Stably Sorted</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ann</td>
<td>98</td>
<td></td>
<td>Ann 98</td>
</tr>
<tr>
<td>Bob</td>
<td>90</td>
<td></td>
<td>Joe 98</td>
</tr>
<tr>
<td>Dan</td>
<td>75</td>
<td></td>
<td>Bob 90</td>
</tr>
<tr>
<td>Joe</td>
<td>98</td>
<td></td>
<td>Sam 90</td>
</tr>
<tr>
<td>Pat</td>
<td>86</td>
<td></td>
<td>Pat 86</td>
</tr>
<tr>
<td>Sam</td>
<td>90</td>
<td></td>
<td>Zöe 86</td>
</tr>
<tr>
<td>Zöe</td>
<td>86</td>
<td></td>
<td>Dan 75</td>
</tr>
</tbody>
</table>

- Original array:
- Stably sorted array:
Unstable sort algorithms

- An unstable sort may or may not keep equal elements in the same order
- Stability is usually not important, but sometimes it matters

<table>
<thead>
<tr>
<th>Ann</th>
<th>98</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob</td>
<td>90</td>
</tr>
<tr>
<td>Dan</td>
<td>75</td>
</tr>
<tr>
<td>Joe</td>
<td>98</td>
</tr>
<tr>
<td>Pat</td>
<td>86</td>
</tr>
<tr>
<td>Sam</td>
<td>90</td>
</tr>
<tr>
<td>Zöe</td>
<td>86</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Joe</th>
<th>98</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ann</td>
<td>98</td>
</tr>
<tr>
<td>Bob</td>
<td>90</td>
</tr>
<tr>
<td>Sam</td>
<td>90</td>
</tr>
<tr>
<td>Zöe</td>
<td>86</td>
</tr>
<tr>
<td>Pat</td>
<td>86</td>
</tr>
<tr>
<td>Dan</td>
<td>75</td>
</tr>
</tbody>
</table>

original array  | unstably sorted
Is mergesort stable?

```java
private void recMergeSort(int[] workspace,
                        int lowerBound,
                        int upperBound) {
  if (lowerBound == upperBound) return;
  int mid = (lowerBound + upperBound) / 2;
  recMergeSort(workspace, lowerBound, mid);
  recMergeSort(workspace, mid + 1, upperBound);
  merge(workspace, lowerBound, mid + 1, upperBound);
}
```

• Is this sort stable?
• `recMergeSort` does none of the actual work of moving elements around—that’s done in `merge`
Stability of merging

- Stability of mergesort depends on the *merge* method
- After all, this is the only place where elements get moved

What if we encounter equal elements when merging?
  - If we always choose the element from the left subarray, mergesort is stable