UNIT 5C
Merge Sort

Course Announcements

• Exam rooms for Lecture 1, 2:30 - 3:20
  – Sections A, B, C, D at Rashid
  – Sections E, F, G at Baker A51 (Giant Eagle Auditorium)

• Exam rooms for Lecture 2, 3:30 – 4:20
  – Sections H, I, J, K at Rashid
  – Sections L, M at PH125C
  – Section N at PH125B

• Bring your CMU id!
Divide and Conquer

- In the military: strategy to gain or maintain power
- In computation:
  - **Divide** the problem into “simpler” versions of itself.
  - **Conquer** each problem using the same process (usually recursively).
  - **Combine** the results of the “simpler” versions to form your final solution.
- Examples: Towers of Hanoi, fractals, Binary Search, Merge Sort

Merge Sort

- **Input**: Array A of n elements.
- **Result**: Returns a new array containing the same elements in non-decreasing order.
- General algorithm for merge sort:
  1. Sort the first half using merge sort. (recursive!)
  2. Sort the second half using merge sort. (recursive!)
  3. Merge the two sorted halves to obtain the final sorted array.
### Divide (Split)

1. Split the array into two halves:
   - First half: 84, 27, 49, 91
   - Second half: 32, 53, 63, 17

2. Recursively divide each half until the list is sorted.

### Conquer (Merge)

1. Merge the sorted halves:
   - Merge the first half: 17, 27, 32, 49, 53, 63, 84, 91
   - Merge the second half: 17, 32, 53, 63

2. Combine the merged lists to get the final sorted array.

---

15110 Principles of Computing, Carnegie Mellon University - CORTINA
Example 1: Merge

array a      array b      array c

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>44</td>
<td>58</td>
<td>62</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>12</td>
<td>29</td>
<td>31</td>
<td>74</td>
</tr>
<tr>
<td>31</td>
<td>74</td>
<td>80</td>
<td>12</td>
</tr>
</tbody>
</table>

Example 1: Merge (cont’d)

array a      array b      array c

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>44</td>
<td>58</td>
<td>62</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>12</td>
<td>29</td>
<td>31</td>
<td>44</td>
</tr>
<tr>
<td>31</td>
<td>44</td>
<td>58</td>
<td>62</td>
</tr>
<tr>
<td>62</td>
<td>74</td>
<td>80</td>
<td>12</td>
</tr>
</tbody>
</table>

12 29 31 44 58 62 74 80
Example 2: Merge

<table>
<thead>
<tr>
<th>array a</th>
<th>array b</th>
<th>array c</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3</td>
<td>0 1 2 3</td>
<td>0 1 2 3</td>
</tr>
<tr>
<td>58 67 74 90</td>
<td>19 26 31 44</td>
<td>19</td>
</tr>
<tr>
<td>0 1 2 3</td>
<td>0 1 2 3</td>
<td>0 1 2 3</td>
</tr>
<tr>
<td>58 67 74 90</td>
<td>19 26 31 44</td>
<td>19 26</td>
</tr>
<tr>
<td>0 1 2 3</td>
<td>0 1 2 3</td>
<td>0 1 2 3</td>
</tr>
<tr>
<td>58 67 74 90</td>
<td>19 26 31 44</td>
<td>19 26 31</td>
</tr>
<tr>
<td>0 1 2 3</td>
<td>0 1 2 3</td>
<td>0 1 2 3</td>
</tr>
<tr>
<td>58 67 74 90</td>
<td>19 26 31 44</td>
<td>19 26 31 44</td>
</tr>
<tr>
<td>0 1 2 3</td>
<td>0 1 2 3</td>
<td>0 1 2 3</td>
</tr>
<tr>
<td>58 67 74 90</td>
<td>19 26 31 44</td>
<td>19 26 31 44 58 67 74 90</td>
</tr>
</tbody>
</table>

Merge

- **Input**: Two arrays a and b.
  - Each array must be sorted already in non-decreasing order.
- **Result**: Returns a new array containing the same elements merged together into a new array in non-decreasing order.
- We’ll need two variables to keep track of where we are in arrays a and b: `index_a` and `index_b`.
  1. Set `index_a` equal to 0.
  2. Set `index_b` equal to 0.
  3. Create an empty array `c`. 

Merge (cont’d)

4. While index_a < the length of array a and index_b < the length of array b, do the following:
   a. If a[index_a] ≤ b[index_b], then do the following:
      i. append a[index_a] on to the end of array c
      ii. add 1 to index_a
   Otherwise, do the following:
      i. append b[index_b] on to the end of array c
      ii. add 1 to index_b

(Once we finish step 4, we’ve added all of the elements of either array a or array b to array c. The other array still has some elements left that need to be added to array c.)

5. If index_a < the length of array a, then:
   append all remaining elements of array a on to the end of array c
Otherwise:
   append all remaining elements of array b on to the end of array c

6. Return array c as the result.
def merge(a, b)
    index_a = 0
    index_b = 0
    c = []
    while index_a < a.length and index_b < b.length do
        if a[index_a] <= b[index_b] then
            c << a[index_a]
            index_a = index_a + 1
        else
            c << b[index_b]
            index_b = index_b + 1
        end
    end
end

if (index_a < a.length) then
    for i in (index_a..a.length-1) do
        c << a[i]
    end
else
    for i in (index_b..b.length-1) do
        c << b[i]
    end
end
return c
end
Merge Sort: Base Case

- General algorithm for merge sort:
  1. Sort the first half using merge sort. (recursive!)
  2. Sort the second half using merge sort. (recursive!)
  3. Merge the two sorted halves to obtain the final sorted array.

- What is the base case?
  If the list has only 1 element, it is already sorted so just return the list as the result.

Merge Sort: Halfway Point

- General algorithm for merge sort:
  1. Sort the first half using merge sort. (recursive!)
  2. Sort the second half using merge sort. (recursive!)
  3. Merge the two sorted halves to obtain the final sorted array.

- How do we determine the halfway point where we want to split the array list?
  First half: \[0..\text{list.length}/2-1\]
  Second half: \[\text{list.length}/2..\text{list.length}-1\]
Merge Sort in Ruby

def msort(list)
    return list if list.length == 1  # base case
    halfway = list.length/2
    list1 = list[0..halfway-1]
    list2 = list[halfway..list.length-1]
    newlist1 = msort(list1)          # recursive!
    newlist2 = msort(list2)          # recursive!
    newlist = merge(newlist1, newlist2)
    return newlist
end

Analyzing Efficiency

- If you merge two lists of size i/2 into one new list of size i, what is the maximum number of appends that you must do?
- Clearly, each element must be appended to the new list at some point, so the total number of appends is i.
- If you have a set of pairs of lists that need to be merged (two pairs at a time), and the total number of elements in all of the lists combined is n, the total number of appends will be n.
How many group merges?

- How many group merges does it take to go from \( n \) groups of size 1 to 1 group of size \( n \)?
- Example: Merge sort on 32 elements.
  - Break down to groups of size 1 (base case).
  - Merge 32 lists of size 1 into 16 lists of size 2.
  - Merge 16 lists of size 2 into 8 lists of size 4.
  - Merge 8 lists of size 4 into 4 lists of size 8.
  - Merge 4 lists of size 8 into 2 lists of size 16.
  - Merge 2 lists of size 16 into 1 list of size 32.
- In general: \( \log_2 n \) group merges must occur.

Putting it all together

It takes \( \log_2 n \) iterations to go from \( n \) groups of size 1 to a single group of size \( n \).

Total number of elements per level is always \( n \).

It takes \( n \) appends to merge all pairs to the next higher level.
Big O

• In the worst case, merge sort requires $O(n \log n)$ time to sort an array with $n$ elements.

<table>
<thead>
<tr>
<th>Number of operations</th>
<th>Order of Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n \log_2 n$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>$4n \log_{10} n$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>$n \log_2 n + 2n$</td>
<td>$O(n \log n)$</td>
</tr>
</tbody>
</table>

For an $n \log_2 n$ algorithm, the performance is better than a quadratic algorithm but just a little worse than a linear algorithm.
Comparing Insertion Sort to Merge Sort
(Worst Case)

<table>
<thead>
<tr>
<th>n</th>
<th>isort ( n(n+1)/2 )</th>
<th>msort ( n \log_2 n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>36</td>
<td>24</td>
</tr>
<tr>
<td>16</td>
<td>136</td>
<td>64</td>
</tr>
<tr>
<td>32</td>
<td>528</td>
<td>160</td>
</tr>
<tr>
<td>(2^{10})</td>
<td>524,800</td>
<td>10,240</td>
</tr>
<tr>
<td>(2^{20})</td>
<td>549,756,338,176</td>
<td>20,971,520</td>
</tr>
</tbody>
</table>

For array sizes less than 100, there’s not much difference between these sorts, but for larger arrays sizes, there is a clear advantage to merge sort.

Sorting and Searching

- Recall that if we wanted to use binary search, the array must be sorted.
  - What if we sort the array first using merge sort?
    - Merge sort \( \mathcal{O}(n \log n) \) (worst case)
    - Binary search \( \mathcal{O}(\log n) \) (worst case)
    - Total time: \( \mathcal{O}(n \log n) + \mathcal{O}(\log n) = \mathcal{O}(n \log n) \) (worst case)
Comparing Big O Functions

Number of Operations

\(O(2^n)\)  \(O(n^2)\)  \(O(n \log n)\)  \(O(n)\)  \(O(\log n)\)  \(O(1)\)

\(n\) (amount of data)

Merge Sort: Iteratively
(optional)

• If you are interested, the textbook discusses an iterative version of merge sort which you can read on your own.

• This version uses an alternate version of the merge function that is not shown in the textbook but is given in the RubyLabs gem.
Quick Sort

• Uses the technique of divide-and-conquer
  1. Pick a pivot
  2. Divide the array into two subarrays, those that are smaller and those that are greater
  3. Put the pivot in the middle, between the two sorted arrays