CSE 373

Sorting 3: Merge Sort, Quick Sort
reading: Weiss Ch. 7

slides created by Marty Stepp
http://www.cs.washington.edu/373/

© University of Washington, all rights reserved.
Merge sort

- **merge sort**: Repeatedly divides the data in half, sorts each half, and combines the sorted halves into a sorted whole.

The algorithm:
- Divide the list into two roughly equal halves.
- Sort the left half.
- Sort the right half.
- Merge the two sorted halves into one sorted list.

- Often implemented recursively.
- An example of a "divide and conquer" algorithm.
  - Invented by John von Neumann in 1945

- Runtime: $O(N \log N)$. Somewhat faster for asc/descending input.
Merge sort example

<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>22</td>
<td>18</td>
<td>12</td>
<td>-4</td>
<td>58</td>
<td>7</td>
<td>31</td>
<td>42</td>
</tr>
</tbody>
</table>

merge  
split  
split  
mmerge  
split  
mmerge  
mmerge  
mmerge  
mmerge
### Merging sorted halves

<table>
<thead>
<tr>
<th>Subarrays</th>
<th>Next include</th>
<th>Merged array</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3</td>
<td>0 1 2 3</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>14 32 67 76</td>
<td>14 from left</td>
<td>14 i</td>
</tr>
<tr>
<td>14 32 67 76</td>
<td>23 from right</td>
<td>14 23 i</td>
</tr>
<tr>
<td>14 32 67 76</td>
<td>32 from left</td>
<td>14 23 32 i</td>
</tr>
<tr>
<td>14 32 67 76</td>
<td>41 from right</td>
<td>14 23 32 41 i</td>
</tr>
<tr>
<td>14 32 67 76</td>
<td>58 from right</td>
<td>14 23 32 41 58 i</td>
</tr>
<tr>
<td>14 32 67 76</td>
<td>67 from left</td>
<td>14 23 32 41 58 67 i</td>
</tr>
<tr>
<td>14 32 67 76</td>
<td>76 from left</td>
<td>14 23 32 41 58 67 76 i</td>
</tr>
<tr>
<td>14 32 67 76</td>
<td>85 from right</td>
<td>14 23 32 41 58 67 76 85 i</td>
</tr>
<tr>
<td></td>
<td></td>
<td>i</td>
</tr>
</tbody>
</table>
// Merges the left/right elements into a sorted result.
// Precondition: left/right are sorted
public static void merge(int[] result, int[] left,
                          int[] right) {

    int i1 = 0;  // index into left array
    int i2 = 0;  // index into right array

    for (int i = 0; i < result.length; i++) {
        if (i2 >= right.length ||
            (i1 < left.length && left[i1] <= right[i2])) {
            result[i] = left[i1];  // take from left
            i1++;
        } else {
            result[i] = right[i2];  // take from right
            i2++;
        }
    }
}
// Rearranges the elements of a into sorted order using
// the merge sort algorithm.
public static void mergeSort(int[] a) {
    if (a.length >= 2) {
        // split array into two halves
        int[] left = Arrays.copyOfRange(a, 0, a.length/2);
        int[] right = Arrays.copyOfRange(a, a.length/2, a.length);

        // recursively sort the two halves
        mergeSort(left);
        mergeSort(right);

        // merge the sorted halves into a sorted whole
        merge(a, left, right);
    }
    }
}
Merge sort runtime

- What is the complexity class (Big-Oh) of merge sort?

<table>
<thead>
<tr>
<th>N</th>
<th>Runtime (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0</td>
</tr>
<tr>
<td>2000</td>
<td>0</td>
</tr>
<tr>
<td>4000</td>
<td>0</td>
</tr>
<tr>
<td>8000</td>
<td>0</td>
</tr>
<tr>
<td>16000</td>
<td>0</td>
</tr>
<tr>
<td>32000</td>
<td>15</td>
</tr>
<tr>
<td>64000</td>
<td>16</td>
</tr>
<tr>
<td>128000</td>
<td>47</td>
</tr>
<tr>
<td>256000</td>
<td>125</td>
</tr>
<tr>
<td>512000</td>
<td>250</td>
</tr>
<tr>
<td>1e6</td>
<td>532</td>
</tr>
<tr>
<td>2e6</td>
<td>1078</td>
</tr>
<tr>
<td>4e6</td>
<td>2265</td>
</tr>
<tr>
<td>8e6</td>
<td>4781</td>
</tr>
<tr>
<td>1.6e7</td>
<td>9828</td>
</tr>
<tr>
<td>3.3e7</td>
<td>20422</td>
</tr>
<tr>
<td>6.5e7</td>
<td>42406</td>
</tr>
<tr>
<td>1.3e8</td>
<td>88344</td>
</tr>
</tbody>
</table>

Input size (N)
Recursive code and runtime

- It is difficult to look at a recursive method and estimate its runtime.
  - Let $T(N) =$ Runtime for merge sort to process an array of size $N$.

```java
mergeSort(a, length=N):
    if $N \geq 2$:
        left = copyOfRange(0, $N/2$)  // approx. $N/2$ time
        right = copyOfRange($N/2$, $N$)  // approx. $N/2$ time
        mergeSort(left)  // $T(N/2)$
        mergeSort(right)  // $T(N/2)$
        merge(a, left, right)  // approx. $N$ time
```

- $T(N) = N/2 + N/2 + T(N/2) + T(N/2) + N$
- $T(N) = 2T(N/2) + 2N$, when $N \geq 2$
- $T(1) = 1$, when $N = 1$

- **recurrence relation**: An equation that recursively defines a sequence, specifically the runtime of a recursive algorithm.
Recurrence relations

- Intuition about recurrence relations: Use *repeated substitution*.
- \( T(N) = 2 \ T(N/2) + 2N \)
  - \( T(N/2) = 2 \ T(N/4) + 2(N/2) \)
- \( T(N) = 2 \ ( \ 2 \ T(N/4) + 2(N/2) \ ) + 2N \)
  - \( T(N/4) = 2 \ T(N/8) + 2(N/4) \)
- \( T(N) = 4 \ ( \ 2 \ T(N/8) + 2(N/4) \ ) + 4N \)
  - \( T(N/8) = 2 \ T(N/16) + 6N \)
- \( T(N) = 16 \ T(N/16) + 8N \)
- \( T(N) = 32 \ T(N/32) + 10N \)
- ...
- \( T(N) = 2^k \ T(N/2^k) + 2kN \)
  - At what value of \( k \) will we hit \( T(1) \) ?
- Let \( k = \log_2 N \).
- \( T(N) = 2^{\log_2 N} \ T(N/2^{\log_2 N}) + 2 \ (\log_2 N) \ N \)
- \( T(N) = N \ T(N/N) + 2 \ N \log_2 N \)
- \( T(N) = N \ T(1) + 2 \ N \log_2 N \)
- \( T(N) = 2 \ N \log_2 N + N \)
- \( T(N) = O(N \log N) \)
More runtime intuition

- Merge sort performs $O(N)$ operations on each level.
  - Each level splits the array in 2, so there are $\log_2 N$ levels.
  - Product of these $= N \times \log_2 N = O(N \log N)$.
  - Example: $N = 32$. Performs $\sim \log_2 32 = 5$ levels of $N$ operations each:

  \[
  \begin{align*}
  &32 \\
  &16 \\
  &8 \\
  &4 \\
  &2 \\
  &1
  \end{align*}
  \]
Quick sort

- **quick sort**: Orders a list of values by partitioning the list around one element called a *pivot*, then sorting each partition.
  - invented by British computer scientist C.A.R. Hoare in 1960

- Quick sort is another divide and conquer algorithm:
  - Choose one element in the list to be the pivot.
  - *Divide* the elements so that all elements less than the pivot are to its left and all greater (or equal) are to its right.
  - *Conquer* by applying quick sort (recursively) to both partitions.

- Runtime: $O(N \log N)$ average, $O(N^2)$ worst case.
  - Generally somewhat faster than merge sort.
Choosing a "pivot"

• The algorithm will work correctly no matter which element you choose as the pivot.
  ▪ A simple implementation can just use the first element.

• But for efficiency, it is better if the pivot divides up the array into roughly equal partitions.
  ▪ What kind of value would be a good pivot? A bad one?

| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|-------|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|
| value | 8 | 18| 12| -4| 27| 30| 36| 50| 7 | 68| 91 | 56 | 2  | 85 | 42 | 98 | 25 |
Partitioning an array

- Swap the pivot to the last array slot, temporarily.
- Repeat until done partitioning (until \(i, j\) meet):
  - Starting from \(i = 0\), find an element \(a[i] \geq\) pivot.
  - Starting from \(j = N-1\), find an element \(a[j] \leq\) pivot.
  - These elements are out of order, so swap \(a[i]\) and \(a[j]\).
- Swap the pivot back to index \(i\) to place it between the partitions.

```
index  0  1  2  3  4  5  6  7  8  9
value  6  1  4  9  0  3  5  2  7  8

8 i  j 6
2  i  →  → j 8
5  i  → 9
```

```
Quick sort example

<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>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>65</td>
<td>23</td>
<td>81</td>
<td>43</td>
<td>92</td>
<td>39</td>
<td>57</td>
<td>16</td>
<td>75</td>
<td>32</td>
</tr>
</tbody>
</table>

choose pivot=65

swap pivot (65) to end

swap 81, 16

swap 57, 92

 recursively quicksort each half

pivot=32

swap 39, 16

swap pivot back in

pivot=81

swap 92, 75

swap 81 back in
public static void quickSort(int[] a) {
    quickSort(a, 0, a.length - 1);
}
private static void quickSort(int[] a, int min, int max) {
    if (min >= max) { // base case; no need to sort
        return;
    }

    // choose pivot; we'll use the first element (might be bad!)
    int pivot = a[min];
    swap(a, min, max); // move pivot to end

    // partition the two sides of the array
    int middle = partition(a, min, max - 1, pivot);

    swap(a, middle, max); // restore pivot to proper location

    // recursively sort the left and right partitions
    quickSort(a, min, middle - 1);
    quickSort(a, middle + 1, max);
}
// partitions a with elements < pivot on left and // elements > pivot on right; // returns index of element that should be swapped with pivot
private static int partition(int[] a, int i, int j, int pivot) {
    while (i <= j) {
        // move index markers i,j toward center // until we find a pair of out-of-order elements
        while (i <= j && a[i] < pivot) { i++; } while (i <= j && a[j] > pivot) { j--; }

        if (i <= j) {
            swap(a, i, j);
            i++; j--;
        }
    }

    return i;
}
Quick sort runtime

- Best-case analysis: If partition divides the array fairly evenly.
  - Let $T(N) = \text{Runtime for quick sort to process an array of size } N$.

  ```
  quickSort(a, length=N):
    swap pivot to end. // O(1)
    mid = partition(a, pivot). // approx. $N$ time
    swap pivot to mid. // O(1)
    quickSort(min, mid-1). // $T(N/2)$ if left size $\approx N/2$
    quickSort(mid+1, max). // $T(N/2)$ if right size $\approx N/2$
  
  \[
  T(N) = 2T(N/2) + k_1N + k_2(1)
  \]
  for some constants $k_1, k_2$
  
  $T(N) = O(N \log N)$

- Worst-case: What if the pivot is chosen poorly?
  What is the runtime?
  - $T(N) = T(N-1) + T(1) + k_1N + k_2(1) = O(N^2)$
Choosing a better pivot

• Choosing the first element as the pivot leads to very poor performance on certain inputs (ascending, descending)
  ▪ does not partition the array into roughly-equal size chunks

• Alternative methods of picking a pivot:
  ▪ *random*: Pick a random index from \([min .. max]\)
  ▪ *median-of-3*: look at left/middle/right elements and pick the one with the medium value of the three:
    • \(a[\text{min}], \ a[(\text{max}+\text{min})/2], \ \text{and} \ a[\text{max}]\)
    • better performance than picking random numbers every time
    • provides near-optimal runtime for almost all input orderings

| index | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|-------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| value | 8  | 18 | 91 | -4 | 27 | 30 | 86 | 50 | 65 | 78 | 5  | 56 | 2  | 25 | 42 | 98 | 31 |
Stable sorting

- **stable sort**: One that maintains relative order of "equal" elements.
  - important for secondary sorting, e.g.
    - sort by name, then sort again by age, then by salary, ...

- All of the $N^2$ sorts shown are stable, as is shell sort.
  - bubble, selection, insertion, shell

- Merge sort is stable.

- Quick sort is *not* stable.
  - The partitioning algorithm can reverse the order of "equal" elements.
  - For this reason, Java's Arrays/Collections.sort() use merge sort.
Unstable sort example

• Suppose you want to sort these points by Y first, then by X:
  ▪ \[ (4, 2), (5, 7), (3, 7), (3, 1) \]

• A stable sort like merge sort would do it this way:
  ▪ \[ (3, 1), (4, 2), (5, 7), (3, 7) \] sort by y
  ▪ \[ (3, 1), (3, 7), (4, 2), (5, 7) \] sort by x
  ▪ Note that the relative order of (3, 1) and (3, 7) is maintained.

• Quick sort might leave them in the following state:
  ▪ \[ (3, 1), (4, 2), (5, 7), (3, 7) \] sort by y
  ▪ \[ (3, 7), (3, 1), (4, 2), (5, 7) \] sort by x
  ▪ Note that the relative order of (3, 1) and (3, 7) has reversed.