Divide and Sort!

Mergesort
Quicksort

How can we write sorting algorithms?

- Many common sorts consist of nested loops
  - Outer loop runs once per element to be sorted
  - Inner loop runs once per element that hasn’t yet been sorted
    - Averages half of the set to be sorted
- Examples
  - Selection sort
  - Insertion sort
  - Bubble sort

- Alternative: recursive sorting
  - Divide set to be sorted into two pieces
  - Sort each piece recursively
- Examples
  - Mergesort
  - Quicksort
Mergesort

- Basic idea
  - Divide set into two equal parts
  - Sort each part
  - Merge the (sorted) halves into a sorted whole

- This can be done very quickly
  - Merging is O(n): must examine two elements for each item in the sorted result
  - How long does it take to sort the two halves?

- First, let’s see the algorithm…

Mergesort: the algorithm

```c
void mergesort (int arr[], int sz) {
    int half = sz/2;
    int *arr2;
    int k1, k2, j;
    if (sz == 1) {
        return;
    }
    arr2 = (int *)malloc(sizeof (int) * sz);
    bcopy (arr, arr2, sz*sizeof(int));
    mergesort (arr2, half);
    mergesort (arr2+half, sz-half);
    for (j=0, k1=0, k2=half; j < sz; j++) {
        if ((k1 < half) && ((k2 >= sz) || (arr2[k1] < arr2[k2]))) {
            arr[j] = arr2[k1++];
        } else {
            arr[j] = arr2[k2++];
        }
    }
    free (arr2);
}
```

Any array of size 1 is sorted!

Make a copy of the data to sort

Recursively sort each half

Merge the two halves

Use the item from first half if any left and
- There are no more in the second half or
- The first half item is smaller

Free the duplicate array
How well does mergesort work?

- Code runs in \( O(n \log n) \)
  - \( O(n) \) for each “level”
  - \( O(\log n) \) levels
- Depending on the constant, it may be faster to sort small arrays (1–10 elements or so) using an \( n^2 \) sort
How well does mergesort work?

- Code runs in \( O(n \log n) \)
  - \( O(n) \) for each “level”
  - \( O(\log n) \) levels
- Depending on the constant, it may be faster to sort small arrays (1–10 elements or so) using an \( n^2 \) sort

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>53</td>
<td>53</td>
<td>53</td>
</tr>
<tr>
<td>14</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>27</td>
<td>27</td>
<td>27</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>31</td>
<td>31</td>
<td>31</td>
</tr>
<tr>
<td>85</td>
<td>85</td>
<td>85</td>
</tr>
<tr>
<td>30</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>67</td>
<td>67</td>
<td>67</td>
</tr>
<tr>
<td>50</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>39</td>
<td>39</td>
<td>39</td>
</tr>
</tbody>
</table>
How well does mergesort work?

- Code runs in $O(n \log n)$
  - $O(n)$ for each “level”
  - $O(\log n)$ levels
- Depending on the constant, it may be faster to sort small arrays (1–10 elements or so) using an $n^2$ sort
How well does mergesort work?

- Code runs in $O(n \log n)$
  - $O(n)$ for each “level”
  - $O(\log n)$ levels
- Depending on the constant, it may be faster to sort small arrays (1–10 elements or so) using an $n^2$ sort

<table>
<thead>
<tr>
<th>53</th>
<th>53</th>
<th>53</th>
<th>53</th>
<th>53</th>
<th>14</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>27</td>
<td>14</td>
</tr>
<tr>
<td>27</td>
<td>27</td>
<td>27</td>
<td>27</td>
<td>27</td>
<td>53</td>
<td>27</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>31</td>
</tr>
<tr>
<td>31</td>
<td>31</td>
<td>31</td>
<td>31</td>
<td>31</td>
<td>31</td>
<td>53</td>
</tr>
<tr>
<td>85</td>
<td>85</td>
<td>85</td>
<td>85</td>
<td>85</td>
<td>85</td>
<td>85</td>
</tr>
<tr>
<td>30</td>
<td>30</td>
<td>30</td>
<td>30</td>
<td>30</td>
<td>30</td>
<td>7</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>67</td>
<td>67</td>
<td>67</td>
<td>67</td>
<td>67</td>
<td>67</td>
<td>30</td>
</tr>
<tr>
<td>50</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>7</td>
<td>39</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>39</td>
<td>50</td>
</tr>
<tr>
<td>39</td>
<td>39</td>
<td>39</td>
<td>39</td>
<td>39</td>
<td>50</td>
<td>67</td>
</tr>
</tbody>
</table>

---

How well does mergesort work?

- Code runs in $O(n \log n)$
  - $O(n)$ for each “level”
  - $O(\log n)$ levels
- Depending on the constant, it may be faster to sort small arrays (1–10 elements or so) using an $n^2$ sort

<table>
<thead>
<tr>
<th>53</th>
<th>53</th>
<th>53</th>
<th>53</th>
<th>53</th>
<th>14</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>27</td>
<td>14</td>
</tr>
<tr>
<td>27</td>
<td>27</td>
<td>27</td>
<td>27</td>
<td>27</td>
<td>55</td>
<td>27</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>31</td>
</tr>
<tr>
<td>31</td>
<td>31</td>
<td>31</td>
<td>31</td>
<td>31</td>
<td>31</td>
<td>53</td>
</tr>
<tr>
<td>85</td>
<td>85</td>
<td>85</td>
<td>85</td>
<td>85</td>
<td>85</td>
<td>85</td>
</tr>
<tr>
<td>30</td>
<td>30</td>
<td>30</td>
<td>30</td>
<td>30</td>
<td>30</td>
<td>7</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>67</td>
<td>67</td>
<td>67</td>
<td>67</td>
<td>67</td>
<td>67</td>
<td>30</td>
</tr>
<tr>
<td>50</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>7</td>
<td>39</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>39</td>
<td>50</td>
</tr>
<tr>
<td>39</td>
<td>39</td>
<td>39</td>
<td>39</td>
<td>39</td>
<td>50</td>
<td>67</td>
</tr>
</tbody>
</table>
Problems with mergesort

- Mergesort requires two arrays
  - Second array dynamically allocated (in C)
  - May be allocated on stack in C++
    ```
    int arr2[sz];
    ```
  - This can take up too much space for large arrays!
- Mergesort is recursive
- These two things combined can be real trouble
  - Mergesort can have log n recursive calls
  - Each call requires O(n) space to be allocated
- Can we eliminate this need for memory?

Solution: mergesort “in place”

- Mergesort builds up “runs” of correctly ordered items and then merges them
- Do this “in place” using linked lists
  - Eliminates extra allocation
  - Eliminates need for recursion (!)
- Keep two lists, each consisting of runs of 1 or more elements in sorted order
  - Combine the runs at the head of the lists into a single (larger) run
  - Place the run at the back of one of the lists
  - Repeat until you’re done
Mergesort “in place” in action

53 2 44 85 11 67 7 39

14 27 80 50 29 72 95

27 80 50 29 72 95
Mergesort “in place” in action

2 44 85 11 67 7 39 14 27 53

80 50 29 72 95

11 67 7 39 14 27 53

50 29 72 95
Mergesort “in place” in action

<table>
<thead>
<tr>
<th>11</th>
<th>67</th>
<th>7</th>
<th>39</th>
<th>14</th>
<th>27</th>
<th>53</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>29</td>
<td>72</td>
<td>95</td>
<td>2</td>
<td>44</td>
<td>80</td>
</tr>
</tbody>
</table>