Topic 06: Quicksort and Merge Sort

- Quicksort
- Merge Sort
- Stability
- Time and Space Complexity
QuickSort

- Quicksort is recursive sort (divide and conquer)
  - Splits array into two pieces, sorts each piece
- Often, the fastest sort (on average) in practice
- But can be slow in the worst case
QuickSort

• QuickSort works in three phases:
  – Choose a pivot and partition the array
  – Recurse to sort the left side
  – Recurse to sort the right side

• But, use a special case for short arrays
What is a Pivot?

- Value from the array
- Chosen arbitrarily
- Partition the array such that:
  - Everything to the left of the pivot is \( \leq \) the pivot
  - Everything to the right of the pivot is \( \geq \) the pivot
What is a Pivot?

The Original Array

pivot

100 13 10 42 0 -1 10 9
What is a Pivot?

The Original Array

After Partitioning

pivot
What is a Pivot?

The Original Array

-100-13-10-42-0-1-109

After Partitioning

-10-0-910100-131042

After Recurse Left

-100910-131042
What is a Pivot?

The Original Array

After Partitioning

After Recurse Left

After Recurse Right
Quicksort (if it's perfect)

127 values

Running Quicksort on the entire array...

Stack quicksort(0,127)
Quicksort
(if it's perfect)

Choose a pivot...

127 values

Running Quicksort on the entire array...

Stack
quicksort(0,127)
Quicksort
(if it's perfect)

Choose a pivot...and partition the array.

NOTE:
The pivot is in its final position (kind of like selection sort)
Quicksort
(if it's perfect)

63 values

Recurse!
Run Quicksort on the left half

Stack
quicksort(0,127)
quicksort(0,63)
Quicksort (if it's perfect)

Choose a new pivot...

Recurse!
Run Quicksort on the left half

Stack
quicksort(0, 127)
quicksort(0, 63)
Quicksort
(if it's perfect)

31 values  31 values  63 values

Choose a new pivot...and partition the array again.

Recurse!
Run Quicksort on the left half

Stack
quicksort(0,127)
quicksort(0,63)
Quick sort (if it's perfect)

Run Quicksort on the left quarter

Stack
quicksort(0,127)
quicksort(0,63)
quicksort(0,31)
Quicksort
(if it's perfect)

15  15  31 values  63 values

Stack
quicksort(0,127)
quicksort(0,63)
quicksort(0,31)
quicksort(0,15)
When the blocks get small enough, we stop recursing.

In the base case, we use an $n^2$ sort.
Quicksort (if it's perfect)

We return to the sort-15 call...

Stack
quicksort(0,127)
quicksort(0,63)
quicksort(0,31)
quicksort(0,15)
Quicksort
(if it's perfect)

We return to the sort-15 call... and recurse into its right-hand partition.

Stack
quicksort(0,127)
quicksort(0,63)
quicksort(0,31)
quicksort(0,15)
quicksort(8,7)
Quicksort
(if it's perfect)

\[ n^2 \] sort again.

Stack
quicksort(0,127)
quicksort(0,63)
quicksort(0,31)
quicksort(0,15)
quicksort(8,7)
We return to the sort-15 call, and now all of its values are sorted...
We return to the sort-15 call, and now all of its values are sorted... so it returns to the sort-31 call...

Stack
quicksort(0,127)
quicksort(0,63)
quicksort(0,31)
We return to the sort-15 call, and now all of its values are sorted... so it returns to the sort-31 call... which recurses into its own right-hand half.
Quicksort
(if it's perfect)

127 values

63 values

31 values

15 15 15 15 15 15 15

63 values

31 values

7 7 7 7 7 7 7 7 7 7 7 7 7
How Expensive is Quicksort?

(Assume perfect pivot choices at first)

• Cost to choose the pivot: $\Theta(1)$
• Cost to partition the array:
• Cost of recursion:
Cost to Partition the Array

- Lower Bound: \( \Theta(n) \)
  - Why?

- Can we write code that achieves the lower bound?
  - Need to “do something” to each element at most once

In Class:
Can we come up with this code?
Partition Algorithm
(a first draft)

\[ \text{data[]} = \ldots \]

\[ p = 0 \quad \text{// position of the pivot} \]
\[ l = 0 \quad \text{// number of “low” elements} \]
\[ h = 0 \quad \text{// number of “high” elements} \]

while (l+h < arr.length-1)
  if (arr[p+1] <= arr[p])
    swap(arr, p, p+1)
    p++
    l++
  else
    swap(arr, p+1, arr.length-1-h)
    h++
Partition Algorithm  
(a first draft)

\textbf{data[]} = ...

\begin{align*}
p & = 0 \quad \text{// position of the pivot} \\
l & = 0 \quad \text{// number of “low” elements} \\
h & = 0 \quad \text{// number of “high” elements}
\end{align*}

\begin{algorithm}
\begin{algorithmic}
  \While{(l+h < arr.length-1)}
    \If{(arr[p+1] <= arr[p])}
      \text{swap}(arr, p,p+1)
      p++
      l++
    \Else
      \text{swap}(arr, p+1,arr.length-1-h)
      h++
    \EndIf
  \EndWhile
\end{algorithmic}
\end{algorithm}

\textbf{By the way...}
Each of these variables represents a loop invariant.
Partition Algorithm (better)

```
data[] arr = ...
    // arr[0] is the pivot, until the end
l = 0   // number of “low” elements
h = 0   // number of “high” elements

while (l+h < arr.length-1)
    while (l+h < arr.length-1 && arr[1+l] <= arr[0])
        l++
    while (l+h < arr.length-1 && arr[arr.length-1-h] >= arr[0])
        h++

    if (l+1+h < arr.length)
        swap(arr, 1+l, arr.length-1-h);

swap(arr, 0,l)   // move the pivot
```

This partitioning algorithm is more efficient than the first one.
How Expensive is Quicksort?

(Assume perfect pivot choices at first)

• Cost to choose the pivot: $\Theta(1)$
• Cost to partition the array: $\Theta(n)$
• Cost of recursion:
Cost of Quicksort Recursion

- Write a recurrence for the cost
  - Assume perfect pivots for now
- Use Master Method to solve
Cost of Quicksort Recursion

- Write a recurrence for the cost
  - Assume perfect pivots for now
- Use Master Method to solve

\[ T(n) = 2 \cdot T\left(\frac{n}{2}\right) + \Theta(n) \]
Cost of Quicksort Recursion

- Write a recurrence for the cost
  - Assume perfect pivots for now
- Use Master Method to solve

\[ T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) \]

\[ T(n) = \Theta(n \lg n) \]
How Expensive is Quicksort?

(Assume perfect pivot choices at first)

- Cost to choose the pivot: $\Theta(1)$
- Cost to partition the array: $\Omega(n)$
- Cost of recursion: (master method)

Total Cost: $\Theta(n \ lg \ n)$
Quicksort's Worst Case

- Worst Case:
  - Choose a pivot which is either the max or min
  - One partition is empty, the other is $n-1$

- If this happens often, then it's an $n^2$ sort!
Quicksort's Worst Case

- The worst case is usually rare
  - Given random data, it is **highly unlikely** to happen
  - If happens once, usually doesn't happen again

- Noteworthy exception: a sorted array (!)
Choosing a Better Pivot

• Median-of-three rule:
  - Choose three pivot candidates (first, last, middle)
  - Use the median of the three as the pivot
  - Works very well for sorted arrays
  - Works well in random case
  - But still might hit the worst case!
Quicksort: Conclusion

• A good Quicksort implementation works very well in practice
  – Usually the fastest (on average)
  – But has a very-rare $n^2$ explosion
Topic 06: Quicksort and Merge Sort

• Quicksort
• Merge Sort
• Stability
• Time and Space Complexity
Merge Sort

- Can we avoid the bad worst-case performance of Quicksort?

- What's good about Quicksort:
  - Recursive ($\log n$ levels)

- What's bad about Quicksort:
  - Can't always choose a good pivot
  - Unstable sort (we'll show this later)
Merge Sort

• Idea:
  - Choose fixed-size bottom-level blocks
  - Combine pairs of blocks after they are sorted (“merging”)
Merge Sort

64 values
Merge Sort

Choose a first block, and sort it, using an $n^2$ sort.
Merge Sort

Sort a second block.
Merge the first two blocks.
To merge two sorted sections, take the minimum of the first two, and repeat.
Merge Sort

Sort another small block.
Merge Sort

16  8  8  32

And another.
Merge Sort

16 16 32

Merge the smallest blocks.
Then merge the 2\textsuperscript{nd} layer.
Merge Sort

More and more blocks build up – since we only merge duplicate sized blocks.
Merge Sort

Sometimes, we will merge many times in a row.

(As the last step, we always merge everything, smallest to largest, even if things aren't nice powers of 2.)
Merge Sort

| 32 | 16 | 16 |

Sometimes, we will merge many times in a row.

(As the last step, we always merge everything, smallest to largest, even if things aren't nice powers of 2.)
Merge Sort

| 32 | 32 |

Sometimes, we will merge many times in a row.

(As the last step, we always merge everything, smallest to largest, even if things aren't nice powers of 2.)
Merge Sort

Sometimes, we will merge many times in a row.

(As the last step, we always merge everything, smallest to largest, even if things aren't nice powers of 2.)
Merge Algorithm

merge(area1[], area2[]):
    output[] = new array
    while both inputs have values left
        compare first elem of area1 to area2
        move smaller to output
    dump leftover values to output
    return output
**Merge Sort**

Group Size: 2

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>17</td>
<td>0</td>
<td>-10</td>
<td>13</td>
<td>100</td>
<td>24</td>
<td>13</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>17</td>
<td>10</td>
<td>13</td>
<td>100</td>
<td>24</td>
<td>13</td>
<td>11</td>
<td>12</td>
<td>17</td>
<td>10</td>
</tr>
</tbody>
</table>

First group (use $n^2$ sort)
### Merge Sort

**Group Size:** 2

| 13 | 17 | -10 | 0  | 13 | 100 | 24 | 13 | 11 | 12 | 17 | 10 |

Second group
(use $n^2$ sort)
Merge Sort

Group Size: 2

merge()

| 13 | 17 | -10 | 0  | 13 | 100 | 24 | 13 | 11 | 12 | 17 | 10 |
Merge Sort

Group Size: 2

```
[13, 17, 0, 13, 100, 24, 13, 11, 12, 17, 10]
```

merge()

```
-10
```
Merge Sort

Group Size: 2

merge()
Merge Sort

Group Size: 2

merge()
Merge Sort

Group Size: 2

merge()

13 100 24 13 11 12 17 10

-10 0 13 17
Merge Sort

Group Size: 2

-10 0 13 17 13 100 24 13 11 12 17 10

Copy
Back

-10 0 13 17
<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>-10</td>
<td>0</td>
<td>13</td>
<td>17</td>
<td>13</td>
<td>100</td>
<td>13</td>
</tr>
<tr>
<td>24</td>
<td>11</td>
<td>12</td>
<td>17</td>
<td>10</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Group Size: 2

\[ n^2 \text{ sort} \quad n^2 \text{ sort} \]
Merge Sort

Group Size: 2

\[
\begin{array}{cccccccc}
-10 & 0 & 13 & 17 & 13 & 100 & 13 & 24 & 11 & 12 & 17 & 10 \\
\end{array}
\]

merge()
Merge Sort

Group Size: 2

-10 0 13 17 100 13 24 11 12 17 10

merge()

13
Merge Sort

Group Size: 2

-10 0 13 17 100 24 11 12 17 10

merge()

13 13
Merge Sort
Group Size: 2

-10  0  13  17  100  100  11  12  17  10

merge()

13  13  24
Merge Sort

Group Size: 2

-10  0  13  17  11  12  17  10

merge()

13  13  24  100
Merge Sort

Group Size: 2

-10  0  13  17  13  13  24  100  11  12  17  10

Copy
Back

13  13  24  100
## Merge Sort

**Group Size: 2**

<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>-10</td>
<td>0</td>
<td>13</td>
<td>17</td>
<td>13</td>
<td>13</td>
<td>24</td>
<td>100</td>
<td>11</td>
</tr>
</tbody>
</table>

```plaintext
merge()
```
Merge Sort

Group Size: 2

merge()

0 13 17 13 13 24 100 11 12 17 10

-10
## Merge Sort

**Group Size:** 2

### Merge Function

```plaintext
merge()
```

<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>-10</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

13 17 13 13 24 100 11 12 17 10
```
Merge Sort

Group Size: 2

merge()

-10 0 13
Merge Sort
Group Size: 2

merge()
Why a $2^{nd}$ Buffer?

- To merge in place, we have to shift values around
  - Each chosen value would require $\Theta(n)$ shifts.
  - Merging two lists would be $\Theta(n^2)$!

- Instead, we use a 2$^{nd}$ buffer, and copy back
  - $\Theta(n)$ time per pass
  - But also $\Theta(n)$ space!
The Battle of the Century
https://www.youtube.com/watch?v=d0nERTFo-Sk

• Quicksort
  - $\Theta(n \ lg \ n)$ time (on average)
  - $\Theta(1)$ space
  - Fastest in practice, overall
  - Best name

• Merge Sort
  - $\Theta(n \ lg \ n)$ time (worst case)
  - $\Theta(n)$ space
  - No bad corner cases
  - Stable
Topic 06: Quicksort and Merge Sort

• Quicksort
• Merge Sort
• Stability
• Time and Space Complexity
Stability

• When there are duplicate keys:
  - A “stable” sort keeps the relative order of duplicates
  - An “unstable” sort might change the order
Quicksort is Unstable

Duplicate 13s
Quicksort is Unstable

After the partition step, the 13s and 17s have changed order.

Thus Quicksort is **unstable**.
In Merge Sort, we need to simply make sure that:

1) Our bottom-level sort is stable

2) When we merge, we always prefer the left over the right.
Merge Sort is Stable

\[ n^2 \text{ sort} \quad n^2 \text{ sort} \]
Merge Sort is Stable

<table>
<thead>
<tr>
<th>13₁</th>
<th>17₁</th>
<th>-10</th>
<th>0</th>
<th>13₂</th>
<th>100</th>
<th>24</th>
<th>13₃</th>
<th>11</th>
<th>12</th>
<th>17₂</th>
<th>10</th>
</tr>
</thead>
</table>

Merge

| -10 | 0  | 13₁ | 17₁ |
Merge Sort is Stable

<table>
<thead>
<tr>
<th>-10</th>
<th>0</th>
<th>13₁</th>
<th>17₁</th>
<th>13₂</th>
<th>100</th>
<th>24</th>
<th>13₃</th>
<th>11</th>
<th>12</th>
<th>17₂</th>
<th>10</th>
</tr>
</thead>
</table>

Copy back

<table>
<thead>
<tr>
<th>-10</th>
<th>0</th>
<th>13₁</th>
<th>17₁</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>

-10 0 13₁ 17₁
Merge Sort is Stable

| -10 | 0  | 13₁ | 17₁ | 13₂ | 100 | 13₃ | 24  | 11  | 12  | 17₂ | 10  |

\( n^2 \) sort \quad \text{and} \quad n^2 \text{ sort}
Merge Sort is Stable

When keys are equal...
Merge Sort is Stable

When keys are equal... always prefer the one on the left.
Merge Sort is Stable

When keys are equal... always prefer the one on the left.
Merge Sort is Stable

When keys are equal... always prefer the one on the left.
Merge Sort is Stable

When keys are equal... always prefer the one on the left.
Merge Sort is Stable

When keys are equal... always prefer the one on the left.
Topic 06: Quicksort and Merge Sort

- Quicksort
- Merge Sort
- Stability
- Time and Space Complexity
Time Complexity

• Quicksort:
  - $\Theta(n)$ cost for partitioning

• Merge Sort
  - $\Theta(n)$ cost for merge

• Same recurrence:

\[
T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) \\
T(n) = \Theta(n \log n)
\]
Space Complexity

• Quicksort:
  - No secondary space required (except stack!)

• Merge Sort
  - Reuse the same buffer for all the merge steps
  - $\Theta(n)$ space for last merge step
Topic 06: Quicksort and Merge Sort

• Quicksort
• Merge Sort
• Stability
• Time and Space Complexity

Summary