CMPSC112
Lecture 35: Quicksort

Prof. John Wenskovitch
04/25/2016
Last Time

- Merge sort:
  - Split an array into two halves
  - Sort each half
    - Split into two halves
    - Sort each half
      - Split into two halves
      - Sort each half
        - Split into two halves
        - Sort each half
          - Merge them back into a single sorted array
        - Merge them back into a single sorted array
    - Merge them back into a single sorted array
Quicksort vs. Mergesort

• In Mergesort, we always split the array in half (as best we could). In Quicksort, we split the array depending on input.
  – Makes sense that this would improve things – worry about what the input we’re sorting is rather than making it arbitrary.

• In Mergesort, we did our recursive calls before we touched the whole array in the same operation. In Quicksort, our recursive calls come after the whole array in partitioned.
  – This also seems like an improvement – instead of merging things that are far apart, let’s partially order the array first.
public static void sort(Comparable a[]) {
    StdRandom.shuffle(a);
    sort(a, 0, a.length-1);
} //sort1

private static void sort(Comparable a[], int lo, int hi) {
    if (hi <= lo) {
        return;
    } //if
    int j = partition(a, lo, hi);
    sort(a, lo, j-1);
    sort(a, j+1, hi);
} //sort
private static int partition(Comparable a[], int lo, int hi) {
    int i = lo, j = hi+1;
    Comparable v = a[lo];
    while (true) {
        while (less(a[++i], v)) {
            if (i == hi) {
                break;
            } //if
        } //while
        while (less(v, a[--j])) {
            if (j == lo) {
                break;
            } //if
        } //while
        if (i >= j) {
            break;
        } //if
        exch(a, i, j);
    } //while
    exch(a, lo, j);
    return j;
} //sort
# Quicksort Partition Visual

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<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>0</td>
<td>16</td>
<td>KR</td>
<td>R</td>
<td>A</td>
<td>T</td>
<td>E</td>
<td>L</td>
<td>E</td>
<td>P</td>
<td>U</td>
<td>I</td>
<td>M</td>
<td>Q</td>
<td>C</td>
<td>X</td>
<td>O</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>12</td>
<td>K</td>
<td>R</td>
<td>A</td>
<td>T</td>
<td>E</td>
<td>L</td>
<td>E</td>
<td>P</td>
<td>U</td>
<td>I</td>
<td>M</td>
<td>Q</td>
<td>C</td>
<td>X</td>
<td>O</td>
<td>S</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td>K</td>
<td>C</td>
<td>A</td>
<td>T</td>
<td>E</td>
<td>I</td>
<td>E</td>
<td>L</td>
<td>P</td>
<td>U</td>
<td>I</td>
<td>M</td>
<td>Q</td>
<td>R</td>
<td>X</td>
<td>O</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td>K</td>
<td>C</td>
<td>A</td>
<td>I</td>
<td>E</td>
<td>L</td>
<td>E</td>
<td>P</td>
<td>U</td>
<td>T</td>
<td>M</td>
<td>Q</td>
<td>R</td>
<td>X</td>
<td>O</td>
<td>S</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>K</td>
<td>C</td>
<td>A</td>
<td>I</td>
<td>E</td>
<td>E</td>
<td>L</td>
<td>P</td>
<td>U</td>
<td>T</td>
<td>M</td>
<td>Q</td>
<td>R</td>
<td>X</td>
<td>O</td>
<td>S</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>K</td>
<td>C</td>
<td>A</td>
<td>I</td>
<td>E</td>
<td>E</td>
<td>L</td>
<td>P</td>
<td>U</td>
<td>T</td>
<td>M</td>
<td>Q</td>
<td>R</td>
<td>X</td>
<td>O</td>
<td>S</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>K</td>
<td>C</td>
<td>A</td>
<td>I</td>
<td>E</td>
<td>E</td>
<td>L</td>
<td>P</td>
<td>U</td>
<td>T</td>
<td>M</td>
<td>Q</td>
<td>R</td>
<td>X</td>
<td>O</td>
<td>S</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>E</td>
<td>C</td>
<td>A</td>
<td>I</td>
<td>E</td>
<td>E</td>
<td>K</td>
<td>L</td>
<td>P</td>
<td>U</td>
<td>T</td>
<td>M</td>
<td>Q</td>
<td>R</td>
<td>X</td>
<td>O</td>
</tr>
</tbody>
</table>

**Initial values:**

- Value: 0
- Value: 16
- Value: K
- Value: R
- Value: A
- Value: T
- Value: E
- Value: L
- Value: E
- Value: P
- Value: U
- Value: I
- Value: M
- Value: Q
- Value: C
- Value: X
- Value: O
- Value: S

**Scan left, scan right:**

- Value: 1
- Value: 12
- Value: K
- Value: R
- Value: A
- Value: T
- Value: E
- Value: L
- Value: E
- Value: P
- Value: U
- Value: I
- Value: M
- Value: Q
- Value: C
- Value: X
- Value: O
- Value: S

**Exchange:**

- Value: 1
- Value: 12
- Value: K
- Value: C
- Value: A
- Value: T
- Value: E
- Value: L
- Value: E
- Value: P
- Value: U
- Value: I
- Value: M
- Value: Q
- Value: R
- Value: X
- Value: O

**Scan left, scan right:**

- Value: 3
- Value: 9
- Value: K
- Value: C
- A
- T
- E
- I
- E
- L
- P
- U
- T
- M
- Q
- R
- X

**Exchange:**

- Value: 3
- Value: 9
- Value: K
- C
- A
- I
- E
- T
- E
- L
- P
- U
- T
- M
- Q
- R
- X

**Scan left, scan right:**

- Value: 5
- Value: 6
- Value: K
- C
- A
- I
- E
- E
- L
- P
- U
- T
- M
- Q
- R
- X

**Exchange:**

- Value: 5
- Value: 6
- Value: K
- C
- A
- I
- E
- E
- L
- P
- U
- T
- M
- Q
- R
- X

**Scan left, scan right:**

- Value: 6
- Value: 5
- Value: K
- C
- A
- I
- E
- E
- L
- P
- U
- T
- M
- Q
- R
- X

**Final exchange:**

- Value: 6
- Value: 5
- Value: E
- C
- A
- I
- E
- K
- L
- P
- U
- T
- M
- Q
- R
- X

**Result:**

- Value: 5
- Value: E
- C
- A
- I
- E
- K
- L
- P
- U
- T
- M
- Q
- R
- X
- O
- S
Quicksort Sort Visual

Initial values
random shuffle

<table>
<thead>
<tr>
<th>l0</th>
<th>j</th>
<th>hi</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
<td>15</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>0</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>0</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>13</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>12</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>0</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>0</td>
<td>15</td>
<td>15</td>
</tr>
</tbody>
</table>

no partition for subarrays of size 1

result

AC E E I K L M O P Q R S T U X

AC E E I K L M O P Q R S T U X

AC E E I K L M O P Q R S T U X

AC E E I K L M O P Q R S T U X

AC E E I K L M O P Q R S T U X

AC E E I K L M O P Q R S T U X

AC E E I K L M O P Q R S T U X

AC E E I K L M O P Q R S T U X

AC E E I K L M O P Q R S T U X

AC E E I K L M O P Q R S T U X

AC E E I K L M O P Q R S T U X

AC E E I K L M O P Q R S T U X
Why Shuffle the Input?

• Quicksort is a \textit{randomized} algorithm.
  – After each \texttt{partition()} call, each subarray is in what is essentially a random order.
  – This random order turns out to be important in predicting the run time of Quicksort.
  – It then follows that we want to select keys randomly. We could either shuffle the array at the beginning, or we could pick a random key from the input instead of always picking the first key.
Quicksort Performance Characteristics

• Inner partition loop increments an index and compares an array entry against a fixed value. Mergesort and Shell Sort also do data movement in their inner loops.

• Quicksort doesn’t use many compares – the efficiency of the sort depends on how well the data is partitioned into subarrays, which hence depends on the choice of keys.

  – **Best case:** Each partitioning stage splits the array perfectly in half. $O(n \times \log(n))$

  – **Worst case:** Each partitioning stage picks the worst possible key, so that every data item needs to be exchanged. (what’s this complexity?)
Quicksort Performance Characteristics

• Wait, so the best case of Quicksort is the average case of Mergesort. How is this better?
  – Mergesort used $n \times \log(n)$ compares and $6n \times \log(n)$ array accesses.
  – Quicksort uses $2n \times \log(n)$ compares and $\frac{1}{3}n \times \log(n)$ exchanges.
Any Questions?