CS 171: Introduction to Computer Science II

Quicksort
Announcements/Reminders

• Midterm
  – Mean: 88
  – Median: 86
• Quiz 3 on Thursday – sorting
• Hw4 due Thursday
Survey Results

• Most favorite topics
  – Stacks and queues (12+)
  – Linked list (7+)
  – Algorithm analysis (4+)
  – Interface (3+)
• Least favorite topics
  – Algorithm analysis/Big O (12-)
  – Interface/iterators (11-)
  – Linked list (5-)
• Most favorite hws
  – Hw3 maze (11+)
  – Hw2 Nqueens (9+)
  – Hw1 Guessing game (3+)
• Least favorite hws
  – Hw 3 maze (5-)
  – Hw1 Guessing game (4-)
  – Hw2 Nqueens (3-)

• Ranking
  – Stacks and queues (12)
  – Hw3 Maze (6)
  – Hw2 Nqueens (6)
  – Linked list (2)
  – Inheritance (0)
  – Hw1 Guessing game (-1)
  – Arrays (-1)
  – Algorithm analysis (-8)
  – Interface (-8)
Roadmap

• Elementary sorting
  – Bubble sort
  – Selection sort
  – Insertion sort

• Recursion (review) and analysis

• Advanced sorting
  – Mergesort
  – Quicksort
Quick Sort

• The most popular sorting algorithm.
• Divide and conquer.
• Uses recursion.
• Fast, and sort ‘in-place’ (i.e. does not require additional memory space)
Quicksort

Basic plan.

- **Shuffle** the array.
- **Partition** so that, for some $j$
  - entry $a[j]$ is in place
  - no larger entry to the left of $j$
  - no smaller entry to the right of $j$
- **Sort** each piece recursively.

![Diagram of Quicksort process]
Partition (Split)

• A **key step** in quicksort
• Given an input array, and a **pivot value**
• Partition the array to two groups: all elements smaller than the pivot are on the left, and those larger than the pivot are on the right

• Example: K R A T E L E P U I M Q C X O S
  pivot: K
Partition (Split)

• How to write code to accomplish partitioning?

1. Assume you are allowed additional memory space.

2. Assume you must perform in-place partition (i.e. no additional memory space allowed).

Quicksort uses in-place partitioning
Partition (Split)

• If additional memory space is allowed (using a workspace array)

Loop over the input array, copy elements smaller than the pivot value to the left side of the workspace array, copy elements larger than the pivot value to the right hand side of the array, and put the pivot value in the “middle”
Quicksort partitioning

Basic plan.

- Scan $i$ from left for an item that belongs on the right.
- Scan $j$ from right for an item that belongs on the left.
- Exchange $a[i]$ and $a[j]$.
- Repeat until pointers cross.

Partitioning trace (array contents before and after each exchange)
private static int partition(Comparable[] a, int lo, int hi)
{
    int i = lo, j = hi + 1;
    while (true)
    {
        while (less(a[++i], a[lo]))
            if (i == hi) break;

        while (less(a[lo], a[--j]))
            if (j == lo) break;

        if (i >= j) break;
        exch(a, i, j);
    }

    exch(a, lo, j);
    return j;
}
Partition (Split)

Some observations:

• The array is not necessarily partitioned in half.
  – This depends on the pivot value.

• The array is by no means sorted.
  – But we are getting closer to that goal.

• What’s the cost of partition?
Quick Sort

• Partition is the key step in quicksort.
• Once we have it, quicksort is pretty simple:
  – Partition (this splits the array into two: left and right)
  – Sort the left part, and sort the right part (how? What’s the base case?)
  – What about the element at the partition boundary?
public class Quick {
    private static int partition(Comparable[] a, int lo, int hi) {
        /* see previous slide */
    }

    public static void sort(Comparable[] a) {
        StdRandom.shuffle(a);
        sort(a, 0, a.length - 1);
    }

    private static void sort(Comparable[] a, int lo, int hi) {
        if (hi <= lo) return;
        int j = partition(a, lo, hi);
        sort(a, lo, j-1);
        sort(a, j+1, hi);
    }
}
Quicksort trace

<table>
<thead>
<tr>
<th>lo</th>
<th>j</th>
<th>hi</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>jhi</td>
<td>0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15</td>
</tr>
</tbody>
</table>

Quicksort trace (array contents after each partition)
Quicksort demonstrations

- [http://www.youtube.com/watch?v=ywWBy6J5gz8](http://www.youtube.com/watch?v=ywWBy6J5gz8)
Quicksort Cost Analysis

• Depends on the partitioning
  – What’s the best case?
  – What’s the worst case?
  – What’s the average case?
**Quicksort: best-case analysis**

<table>
<thead>
<tr>
<th>lo</th>
<th>j</th>
<th>hi</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>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>initial values</strong></td>
<td>H</td>
<td>A</td>
<td>C</td>
<td>B</td>
<td>F</td>
<td>E</td>
<td>G</td>
<td>D</td>
<td>L</td>
<td>I</td>
<td>K</td>
<td>J</td>
<td>N</td>
<td>M</td>
<td>O</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>random shuffle</strong></td>
<td>H</td>
<td>A</td>
<td>C</td>
<td>B</td>
<td>F</td>
<td>E</td>
<td>G</td>
<td>D</td>
<td>L</td>
<td>I</td>
<td>K</td>
<td>J</td>
<td>N</td>
<td>M</td>
<td>O</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>7</td>
<td>14</td>
<td>D</td>
<td>A</td>
<td>C</td>
<td>B</td>
<td>F</td>
<td>E</td>
<td>G</td>
<td>H</td>
<td>L</td>
<td>I</td>
<td>K</td>
<td>J</td>
<td>N</td>
<td>M</td>
<td>O</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>6</td>
<td>B</td>
<td>A</td>
<td>C</td>
<td>D</td>
<td>F</td>
<td>E</td>
<td>G</td>
<td>H</td>
<td>L</td>
<td>I</td>
<td>K</td>
<td>J</td>
<td>N</td>
<td>M</td>
<td>O</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>F</td>
<td>E</td>
<td>G</td>
<td>H</td>
<td>L</td>
<td>I</td>
<td>K</td>
<td>J</td>
<td>N</td>
<td>M</td>
<td>O</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>3</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>F</td>
<td>E</td>
<td>G</td>
<td>H</td>
<td>L</td>
<td>I</td>
<td>K</td>
<td>J</td>
<td>N</td>
<td>M</td>
<td>O</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>5</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>L</td>
<td>I</td>
<td>K</td>
<td>J</td>
<td>N</td>
<td>M</td>
<td>O</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>6</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>L</td>
<td>I</td>
<td>K</td>
<td>J</td>
<td>N</td>
<td>M</td>
<td>O</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>8</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>L</td>
<td>I</td>
<td>K</td>
<td>J</td>
<td>N</td>
<td>M</td>
<td>O</td>
</tr>
<tr>
<td>8</td>
<td>11</td>
<td>14</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>J</td>
<td>I</td>
<td>K</td>
<td>L</td>
<td>N</td>
<td>M</td>
<td>O</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>N</td>
<td>M</td>
<td>O</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>10</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>N</td>
<td>M</td>
<td>O</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>12</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>N</td>
<td>M</td>
<td>O</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>N</td>
<td>O</td>
</tr>
<tr>
<td>12</td>
<td>12</td>
<td>14</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>N</td>
<td>O</td>
</tr>
<tr>
<td>14</td>
<td>14</td>
<td>14</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>N</td>
<td>O</td>
</tr>
</tbody>
</table>
Quicksort Cost Analysis – Best case

• The best case is when each partition splits the array into two equal halves
• Overall cost for sorting N items
  – Partitioning cost for N items: N comparisons
  – Cost for recursively sorting two half-size arrays
• Recurrence relations
  – $C(N) = 2 \times C(N/2) + N$
  – $C(1) = 0$
Quicksort Cost Analysis – Best case

• Simplified recurrence relations
  – \( C(N) = 2 \ C(N/2) + N \)
  – \( C(1) = 0 \)

• Solving the recurrence relations
  – \( N = 2^k \)
  – \( C(N) = 2 \ C(2^{k-1}) + 2^k \)
    = \( 2 \ (2 \ C(2^{k-2}) + 2^{k-1}) + 2^k \)
    = \( 2^2 \ C(2^{k-2}) + 2^k + 2^k \)
    = ...
    = \( 2^k \ C(2^{k-k}) + 2^k + ... 2^k + 2^k \)
    = \( 2^k + ... 2^k + 2^k \)
    = \( k \times 2^k \)
    = \( O(N \log N) \)
# Quicksort: worst-case analysis

<table>
<thead>
<tr>
<th>lo</th>
<th>j</th>
<th>hi</th>
<th></th>
<th>a[]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>A B C D E F G H I J K L M N O</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>A B C D E F G H I J K L M N O</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>A B C D E F G H I J K L M N O</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td>A B C D E F G H I J K L M N O</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>A B C D E F G H I J K L M N O</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>5</td>
<td>A B C D E F G H I J K L M N O</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>6</td>
<td>A B C D E F G H I J K L M N O</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>7</td>
<td>A B C D E F G H I J K L M N O</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>8</td>
<td>A B C D E F G H I J K L M N O</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>9</td>
<td>A B C D E F G H I J K L M N O</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>10</td>
<td>A B C D E F G H I J K L M N O</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>11</td>
<td>A B C D E F G H I J K L M N O</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>12</td>
<td>A B C D E F G H I J K L M N O</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>13</td>
<td>A B C D E F G H I J K L M N O</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>14</td>
<td>A B C D E F G H I J K L M N O</td>
</tr>
</tbody>
</table>

**Initial values:**

**Random shuffle:**

|    |    |    | 0    | A B C D E F G H I J K L M N O |
|    |    |    | 1    | A B C D E F G H I J K L M N O |
|    |    |    | 2    | A B C D E F G H I J K L M N O |
|    |    |    | 3    | A B C D E F G H I J K L M N O |
|    |    |    | 4    | A B C D E F G H I J K L M N O |
|    |    |    | 5    | A B C D E F G H I J K L M N O |
|    |    |    | 6    | A B C D E F G H I J K L M N O |
|    |    |    | 7    | A B C D E F G H I J K L M N O |
|    |    |    | 8    | A B C D E F G H I J K L M N O |
|    |    |    | 9    | A B C D E F G H I J K L M N O |
|    |    |    | 10   | A B C D E F G H I J K L M N O |
|    |    |    | 11   | A B C D E F G H I J K L M N O |
|    |    |    | 12   | A B C D E F G H I J K L M N O |
|    |    |    | 13   | A B C D E F G H I J K L M N O |
|    |    |    | 14   | A B C D E F G H I J K L M N O |

After continued sorting, the elements are sorted in ascending order.
Quicksort Cost Analysis – Worst case

• The worst case is when the partition does not split the array (one set has no elements)
• Ironically, this happens when the array is sorted!
• Overall cost for sorting N items
  – Partitioning cost for N items: N comparisons
  – Cost for recursively sorting the remaining (N-1) items
• Recurrence relations
  – C(N) = C(N-1) + N
  – C(1) = 0
Quicksort Cost Analysis – Worst case

• Recurrence relations
  \[ C(N) = C(N-1) + N \]
  \[ C(1) = 0 \]

• Solving the recurrence relations
  \[ C(N) = C(N-1) + N \]
  \[ = C(N-2) + N -1 + N \]
  \[ = C(N-3) + N-2 + N-1 + N \]
  \[ = \ldots \]
  \[ = C(1) + 2 + \ldots + N-2 + N-1 + N \]
  \[ = O(N^2) \]
Quicksort Cost Analysis – Average case

• Suppose the partition split the array into 2 sets containing k and N-k-1 items respectively (0<=k<=N-1)

• Recurrence relations
  – C(N) = C(k) + C(N-k-1) + N

• On average,
  – C(k) = C(0) + C(1) + … + C(N-1) /N
  – C(N-k-1) = C(N-1) + C(N-2) + … + C(0) /N

• Solving the recurrence relations (not required for the course)
  – Approximately, C(N) = 2NlogN
Quicksort: summary of performance characteristics

Worst case. Number of compares is quadratic.
- \( N + (N - 1) + (N - 2) + \ldots + 1 \sim \frac{1}{2} N^2. \)
- More likely that your computer is struck by lightning bolt.

Average case. Number of compares is \( \sim 1.39 N \lg N \).
- 39% more compares than mergesort.
- But faster than mergesort in practice because of less data movement.

Random shuffle.
- Probabilistic guarantee against worst case.
- Basis for math model that can be validated with experiments.

Caveat emptor. Many textbook implementations go quadratic if array
- Is sorted or reverse sorted.
- Has many duplicates (even if randomized!)
Quicksort: empirical analysis

Running time estimates:
- Home PC executes $10^8$ compares/second.
- Supercomputer executes $10^{12}$ compares/second.

<table>
<thead>
<tr>
<th></th>
<th>insertion sort ($N^2$)</th>
<th>mergesort ($N \log N$)</th>
<th>quicksort ($N \log N$)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>thousand</td>
<td>million</td>
<td>billion</td>
</tr>
<tr>
<td>home</td>
<td>instant</td>
<td>2.8 hours</td>
<td>317 years</td>
</tr>
<tr>
<td>super</td>
<td>instant</td>
<td>1 second</td>
<td>1 week</td>
</tr>
</tbody>
</table>

Lesson 1. Good algorithms are better than supercomputers.
Lesson 2. Great algorithms are better than good ones.
Insertion sort small subarrays.

- Even quicksort has too much overhead for tiny subarrays.
- Cutoff to insertion sort for \( \approx 10 \) items.
- Note: could delay insertion sort until one pass at end.

```java
private static void sort(Comparable[] a, int lo, int hi)
{
    if (hi <= lo + CUTOFF - 1)
    {
        Insertion.sort(a, lo, hi);
        return;
    }
    int j = partition(a, lo, hi);
    sort(a, lo, j-1);
    sort(a, j+1, hi);
}
```
QuickSort: practical improvement

• The basic QuickSort uses the first (or the last element) as the pivot value
• What’s the best choice of the pivot value?
• Ideally the pivot should partition the array into two equal halves
Median-of-Three Partitioning

• We don’t know the median, but let’s approximate it by the median of three elements in the array: the first, last, and the center.

• This is fast, and has a good chance of giving us something close to the real median.
Median of sample.

- Best choice of pivot item = median.
- Estimate true median by taking median of sample.
- Median-of-3 (random) items.

~ $12/7$ $N \ln N$ compares (slightly fewer)
~ $12/35$ $N \ln N$ exchanges (slightly more)

```java
private static void sort(Comparable[] a, int lo, int hi) {
    if (hi <= lo) return;

    int m = medianOf3(a, lo, lo + (hi - lo)/2, hi);
    swap(a, lo, m);

    int j = partition(a, lo, hi);
    sort(a, lo, j-1);
    sort(a, j+1, hi);
}
```
Quicksort with median-of-3 and cutoff to insertion sort: visualization
Quicksort Summary

• Quicksort partition the input array to two sub-arrays, then sort each subarray recursively.
• It sorts in-place.
• $O(N \cdot \log N)$ cost, but faster than mergesort in practice
• These features make it the most popular sorting algorithm.
Roadmap

• Quicksort algorithm
• Quicksort Analysis
• Practical improvements
• Sorting summary
• Java sorting methods
Sorting Summary

- Elementary sorting algorithms
  - Bubble sort
  - Selection sort
  - Insertion sort

- Advanced sorting algorithms
  - Merge sort
  - Quicksort

- Performance characteristics
  - Runtime
  - Space requirement
  - Stability
Stability

- A sorting algorithm is stable if it preserves the relative order of equal keys in the array.
- Stable: insertion sort and mergesort.
- Unstable: selection sort, quicksort.

Stability when sorting on a second key.
<table>
<thead>
<tr>
<th></th>
<th>inplace?</th>
<th>stable?</th>
<th>worst</th>
<th>average</th>
<th>best</th>
<th>remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>selection</td>
<td>x</td>
<td></td>
<td>(N^2/2)</td>
<td>(N^2/2)</td>
<td>(N^2/2)</td>
<td>(N) exchanges</td>
</tr>
<tr>
<td>insertion</td>
<td>x</td>
<td>x</td>
<td>(N^2/2)</td>
<td>(N^2/4)</td>
<td>N</td>
<td>use for small (N) or partially ordered</td>
</tr>
<tr>
<td>shell</td>
<td>x</td>
<td></td>
<td>?</td>
<td>?</td>
<td>N</td>
<td>tight code, subquadratic</td>
</tr>
<tr>
<td>merge</td>
<td>x</td>
<td></td>
<td>(N \log N)</td>
<td>(N \log N)</td>
<td>(N \log N)</td>
<td>(N \log N) guarantee, stable</td>
</tr>
<tr>
<td>quick</td>
<td>x</td>
<td></td>
<td>(N^2/2)</td>
<td>(2N \log N)</td>
<td>(N \log N)</td>
<td>(N \log N) probabilistic guarantee fastest in practice</td>
</tr>
<tr>
<td>3-way quick</td>
<td>x</td>
<td></td>
<td>(N^2/2)</td>
<td>(2N \log N)</td>
<td>N</td>
<td>improves quicksort in presence of duplicate keys</td>
</tr>
<tr>
<td>??</td>
<td>x</td>
<td>x</td>
<td>(N \log N)</td>
<td>(N \log N)</td>
<td>(N \log N)</td>
<td>holy sorting grail</td>
</tr>
</tbody>
</table>
Java system sort method

• Arrays.sort() in the java.util library represents a collection of overloaded methods:
  – Methods for each primitive type
    • e.g. sort(int[] a)
  – Methods for data types that implement Comparable.
    • sort(Object[] a)
  – Method that use a Comparator
    • sort(T[] a, Comparator<? super T> c)

• Implementation
  – quicksort (with 3-way partitioning) to implement the primitive-type methods (speed and memory usage)
  – mergesort for reference-type methods (stability)
Example

• Sorting transactions  
  – Who, when, transaction amount
• Use Arrays.sort() methods  
• Implement Comparable interface for a transaction  
• Define multiple comparators to allow sorting by multiple keys

• Transaction.java  