2.3 QuickSort

- quicksort
- selection
- duplicate keys
- system sorts
Two classic sorting algorithms: mergesort and quicksort

**Critical components in the world’s computational infrastructure.**

- Full scientific understanding of their properties has enabled us to develop them into practical system sorts.
- Quicksort honored as one of top 10 algorithms of 20^{th} century in science and engineering.

**Mergesort.** [last lecture]

**Quicksort.** [this lecture]
Quicksort t-shirt

```java
public static void quicksort(Comparable[] a) { // private static void exch(Comparable[] a, int i, int j) {
    private static boolean isSorted(Comparable[] a) { return true; }
    private static void show(Comparable[] a) {
        for (int i = 0; i < a.length; i++) {
            System.out.println(a[i]);
        }
    }
    public static void sort(Comparable[] a) {
        if (a.length == 0) return;
        if (a.length <= 1) return;
        int lo = 0, hi = a.length - 1;
        Comparable v = a[hi];
        while (lo < hi) {
            int i = lo;
            int j = hi;
            Comparable w = a[i];
            while (less(w, v)) { // while (i < hi) {
                a[i++] = v;
            }
            while (less(v, w)) { // while (lo < j) {
                a[j--] = w;
            } // a[i] = v;
            if (i == j) break;
        }
        if (less(a[i], v)) { // if (i <= j) {
            a[i] = v;
            a[i++] = a[j];
            a[j] = a[i-1];
            a[i-1] = v;
        }
        show(a); // return a;
    }
    public static void main(String[] args) { String[] a = StdIn.readString();
        int i = 0;
        int length = a.length;
        while (i < length) {
            System.out.println(a[i]);
            i++;
        }
    }
    public class QuickSort {
        public static void sort() {
            int length = a.length;
            int i = 0;
            int j = length - 1;
            Comparable v = a[i];
            while (i < j) {
                a[i++] = v;
            }
            while (j > i) {
                a[j--] = v;
            }
            System.out.println(a);
        }
    }
```
A brief history

Tony Hoare.

- Invented quicksort to translate Russian into English.
- Learned Algol 60 (and recursion) to implement it.

Bob Sedgewick.

- Refined and popularized quicksort.
- Analyzed many versions of quicksort.
2.3 Quicksort

- quicksort
- selection
- duplicate keys
- system sorts
Quicksort overview

**Step 1.** Shuffle the array.

**Step 2.** Partition the array 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 $j$.

**Step 3.** Sort each subarray recursively.
Quicksort partitioning demo

Repeat until i and j pointers cross.

- Scan i from left to right so long as (a[i] < a[lo]).
- Scan j from right to left so long as (a[j] > a[lo]).
- Exchange a[i] with a[j].
Quicksort partitioning demo

Repeat until i and j pointers cross.

- Scan i from left to right so long as \( a[i] < a[lo] \).
- Scan j from right to left so long as \( a[j] > a[lo] \).
- Exchange \( a[i] \) with \( a[j] \).

When pointers cross.

- Exchange \( a[lo] \) with \( a[j] \).

partitioned!
In the worst case, how many compares and exchanges to partition an array of length $n$, respectively?

A. $\sim \frac{1}{2} n$ and $\sim \frac{1}{2} n$

B. $\sim \frac{1}{2} n$ and $\sim n$

C. $\sim n$ and $\sim \frac{1}{2} n$

D. $\sim n$ and $\sim n$

<table>
<thead>
<tr>
<th>M</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>V</th>
<th>W</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
</tbody>
</table>

Scan until $\geq M$

Scan until $\leq M$
The music of quicksort partitioning (by Brad Lyon)

The value was larger than the pivot, so the lower one waits while the upper one comes down

We will now start coming down from the right

https://learnforeverlearn.com/pivot_music
Quicksort partitioning: Java implementation

```java
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;
}
```

https://algs4.cs.princeton.edu/23quick/Quick.java.html
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></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

initial values

random shuffle

<table>
<thead>
<tr>
<th>lo</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>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>15</td>
</tr>
<tr>
<td>7</td>
<td>9</td>
<td>15</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>13</td>
<td>15</td>
</tr>
<tr>
<td>10</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>14</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>15</td>
<td>15</td>
<td>15</td>
</tr>
</tbody>
</table>

Quicksort trace (array contents after each partition)

<table>
<thead>
<tr>
<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>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>U</td>
<td>I</td>
<td>C</td>
<td>K</td>
<td>S</td>
<td>O</td>
<td>R</td>
<td>T</td>
<td>E</td>
<td>X</td>
<td>A</td>
<td>M</td>
<td>P</td>
<td>L</td>
<td>E</td>
</tr>
<tr>
<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>E</td>
<td>C</td>
<td>A</td>
<td>I</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>
<td>S</td>
</tr>
<tr>
<td>E</td>
<td>C</td>
<td>A</td>
<td>E</td>
<td>I</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>
<td>S</td>
</tr>
<tr>
<td>A</td>
<td>C</td>
<td>E</td>
<td>E</td>
<td>I</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>
<td>S</td>
</tr>
<tr>
<td>A</td>
<td>C</td>
<td>E</td>
<td>E</td>
<td>I</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>
<td>S</td>
</tr>
<tr>
<td>A</td>
<td>C</td>
<td>E</td>
<td>E</td>
<td>I</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>
<td>S</td>
</tr>
<tr>
<td>A</td>
<td>C</td>
<td>E</td>
<td>E</td>
<td>I</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>
<td>S</td>
</tr>
<tr>
<td>A</td>
<td>C</td>
<td>E</td>
<td>E</td>
<td>I</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>O</td>
<td>P</td>
<td>T</td>
<td>Q</td>
<td>R</td>
<td>X</td>
<td>U</td>
<td>S</td>
</tr>
<tr>
<td>A</td>
<td>C</td>
<td>E</td>
<td>E</td>
<td>I</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>O</td>
<td>P</td>
<td>T</td>
<td>Q</td>
<td>R</td>
<td>X</td>
<td>U</td>
<td>S</td>
</tr>
<tr>
<td>A</td>
<td>C</td>
<td>E</td>
<td>E</td>
<td>I</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>O</td>
<td>P</td>
<td>T</td>
<td>Q</td>
<td>R</td>
<td>X</td>
<td>U</td>
<td>S</td>
</tr>
<tr>
<td>A</td>
<td>C</td>
<td>E</td>
<td>E</td>
<td>I</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>O</td>
<td>P</td>
<td>Q</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>U</td>
<td>X</td>
</tr>
<tr>
<td>A</td>
<td>C</td>
<td>E</td>
<td>E</td>
<td>I</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>O</td>
<td>P</td>
<td>Q</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>U</td>
<td>X</td>
</tr>
<tr>
<td>A</td>
<td>C</td>
<td>E</td>
<td>E</td>
<td>I</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>O</td>
<td>P</td>
<td>Q</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>U</td>
<td>X</td>
</tr>
<tr>
<td>A</td>
<td>C</td>
<td>E</td>
<td>E</td>
<td>I</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>O</td>
<td>P</td>
<td>Q</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>U</td>
<td>X</td>
</tr>
</tbody>
</table>

Quicksort trace (array contents after each partition)

result

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

no partition for subarrays of size 1

Quicksort trace (array contents after each partition)
Quicksort animation

50 random items

http://www.sorting-algorithms.com/quick-sort
Quicksort: implementation details

Partitioning in-place. Using an extra array makes partitioning easier (and stable), but it is not worth the cost.

Loop termination. Terminating the loop is trickier than it appears.

Equal keys. Handling duplicate keys is trickier than it appears. [stay tuned]

Preserving randomness. Shuffling is needed for performance guarantee.

Equivalent alternative. Pick a random partitioning item in each subarray.
Quicksort: empirical analysis (1962)

Running time estimates:

- Algol 60 implementation.
- National Elliott 405 computer.

<table>
<thead>
<tr>
<th>NUMBER OF ITEMS</th>
<th>MERGE SORT</th>
<th>QUICKSORT</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>2 min 8 sec</td>
<td>1 min 21 sec</td>
</tr>
<tr>
<td>1,000</td>
<td>4 min 48 sec</td>
<td>3 min 8 sec</td>
</tr>
<tr>
<td>1,500</td>
<td>8 min 15 sec*</td>
<td>5 min 6 sec</td>
</tr>
<tr>
<td>2,000</td>
<td>11 min 0 sec*</td>
<td>6 min 47 sec</td>
</tr>
</tbody>
</table>

* These figures were computed by formula, since they cannot be achieved on the 405 owing to limited store size.

sorting n 6-word items with 1-word keys
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>
<tr>
<td></td>
<td>thousand</td>
<td>million</td>
<td>billion</td>
</tr>
<tr>
<td>home</td>
<td>instant</td>
<td>1 second</td>
<td>18 min</td>
</tr>
<tr>
<td>super</td>
<td>instant</td>
<td>instant</td>
<td>instant</td>
</tr>
<tr>
<td></td>
<td>thousand</td>
<td>million</td>
<td>billion</td>
</tr>
<tr>
<td>home</td>
<td>instant</td>
<td>0.6 sec</td>
<td>12 min</td>
</tr>
<tr>
<td>super</td>
<td>instant</td>
<td>instant</td>
<td>instant</td>
</tr>
</tbody>
</table>

**Lesson 1.** Good algorithms are better than supercomputers.
**Lesson 2.** Great algorithms are better than good ones.
Worst case. Number of compares is $\sim \frac{1}{2} n^2$. 

![Table showing worst-case analysis of Quicksort after random shuffle]
Quicksort: worst-case analysis

**Worst case.** Number of compares is $\sim \frac{1}{2} n^2$.

**Good news.** Worst case for quicksort is irrelevant in practice.

- Exponentially small chance of occurring (unless bug in shuffling or no shuffling).
- More likely that computer is struck by lightning bolt during execution.
**Quicksort: probabilistic analysis**

**Proposition.** The expected number of compares $C_n$ to quicksort an array of $n$ distinct keys is $\sim 2n \ln n$ (and the number of exchanges is $\sim \frac{1}{3} n \ln n$).

**Recall.** Any algorithm with the following structure takes $n \log n$ time.

```java
public static void f(int n)
{
    if (n == 0) return;
    f(n/2);  // solve two problems
    f(n/2);  // of half the size
    linear(n);  // do a linear amount of work
}
```

**Intuition.** Each partitioning step divides the problem into two subproblems, each of approximately one-half the size.

“close enough”
Quicksort: probabilistic analysis

**Proposition.** The expected number of compares $C_n$ to quicksort an array of $n$ distinct keys is $\sim 2n \ln n$ (and the number of exchanges is $\sim \frac{1}{3} n \ln n$).

**Pf.** $C_n$ satisfies the recurrence $C_0 = C_1 = 0$ and for $n \geq 2$:

$$C_n = (n + 1) + \left( \frac{C_0 + C_{n-1}}{n} \right) + \left( \frac{C_1 + C_{n-2}}{n} \right) + \ldots + \left( \frac{C_{n-1} + C_0}{n} \right)$$

- Multiply both sides by $n$ and collect terms:
  $$n C_n = n(n + 1) + 2(C_0 + C_1 + \ldots + C_{n-1})$$

- Subtract from this equation the same equation for $n - 1$:
  $$n C_n - (n - 1) C_{n-1} = 2n + 2 C_{n-1}$$

- Rearrange terms and divide by $n(n + 1)$:
  $$\frac{C_n}{n + 1} = \frac{C_{n-1}}{n} + \frac{2}{n + 1}$$
Quicksort: probabilistic analysis

• Repeatedly apply previous equation:

\[
\frac{C_n}{n+1} = \frac{C_{n-1}}{n} + \frac{2}{n+1}
\]

\[
= \frac{C_{n-2}}{n-1} + \frac{2}{n} + \frac{2}{n+1}
\]

\[
= \frac{C_{n-3}}{n-2} + \frac{2}{n-1} + \frac{2}{n} + \frac{2}{n+1}
\]

\[
= \frac{2}{3} + \frac{2}{4} + \frac{2}{5} + \ldots + \frac{2}{n+1}
\]

• Approximate sum by an integral:

\[
C_n = 2(n+1) \left( \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \ldots + \frac{1}{n+1} \right)
\]

\[
\approx 2(n+1) \int_3^{n+1} \frac{1}{x} \, dx
\]

• Finally, the desired result:

\[
C_n \sim 2(n+1) \ln n \approx 1.39n \lg n
\]
Quicksort: performance characteristics

Quicksort performance summary.

- Expected number of compares is $\sim 1.39 \ n \ lg \ n$.
- Minimum number of compares is $\sim n \ lg \ n$.
- Maximum number of compares is $\sim \frac{1}{2} \ n^2$.
- Faster than mergesort in practice because of less data movement.

Context. Quicksort is a (Las Vegas) randomized algorithm.

- Guaranteed to be correct.
- Running time depends on outcomes of random coin flips (shuffle).
Quicksort properties

Proposition. Quicksort is an in-place sorting algorithm.

Proof. (Pf.)
- Partitioning: constant extra space.
- Function-call stack: logarithmic extra space (with high probability).

Proposition. Quicksort is not stable.

Proof. (Pf.) [by counterexample]

<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>B₀</td>
<td>C₀</td>
<td>C₁</td>
<td>A₀</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>A₀</td>
<td>B₀</td>
<td>C₀</td>
<td>C₁</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>B₁</td>
<td>C₁</td>
<td>C₂</td>
<td>A₁</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>B₁</td>
<td>A₁</td>
<td>C₂</td>
<td>C₁</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>A₁</td>
<td>B₁</td>
<td>C₂</td>
<td>C₁</td>
</tr>
</tbody>
</table>
**Quicksort: practical improvements**

**Insertion sort small subarrays.**

- Even quicksort has too much overhead for tiny subarrays.
- Cutoff to insertion sort for \( \approx 10 \) items.

```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 improvements

Median of sample.

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

\[ \sim \frac{12}{7} \ n \ln n \text{ compares (14\% fewer)} \]
\[ \sim \frac{12}{35} \ n \ln n \text{ exchanges (3\% more)} \]

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

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

    int j = partition(a, lo, hi);
    sort(a, lo, j-1);
    sort(a, j+1, hi);
}
```
2.3 QUICKSORT

- quicksort
- selection
- duplicate keys
- system sorts
Selection

Goal. Given an array of $n$ items, find item of rank $k$.
Ex. Min ($k = 0$), max ($k = n - 1$), median ($k = n / 2$).

Applications.
- Order statistics.
- Find the “top $k$.”

Use theory as a guide.
- Easy $n \log n$ upper bound. How?
- Easy $n$ upper bound for $k = 0, 1, 2$. How?
- Easy $n$ lower bound. Why?

Which is true?
- $n \log n$ lower bound? [ is selection as hard as sorting? ]
- $n$ upper bound? [ is there a linear-time algorithm? ]
Quick-select

Partition array so that:

- Entry $a[j]$ is in place.
- No larger entry to the left of $j$.
- No smaller entry to the right of $j$.

Repeat in one subarray, depending on $j$; finished when $j$ equals $k$.

```java
public static Comparable select(Comparable[] a, int k) {
    StdRandom.shuffle(a);
    int lo = 0, hi = a.length - 1;
    while (hi > lo) {
        int j = partition(a, lo, hi);
        if (j < k) lo = j + 1;
        else if (j > k) hi = j - 1;
        else return a[k];
    }
    return a[k];
}
```
Quick-select: mathematical analysis

**Proposition.** Quick-select takes linear time on average.

**Recall.** Any algorithm with the following structure takes linear time.

```java
public static void f(int n)
{
    if (n == 0) return;
    linear(n);
    f(n/2);
}
```

\[ n + n/2 + n/4 + \ldots + 1 \sim 2n \]

**Intuition.** Each partitioning step approximately halves the size of array.

**Formal analysis.**

\[
C_n = 2n + 2k \ln(n/k) + 2(n-k) \ln(n/(n-k))
\]

\[ \leq (2 + 2 \ln 2) n \]

\[ \approx 3.38 n \quad \text{max occurs for median (}k = n/2\text{)} \]
Theoretical context for selection


**Time Bounds for Selection**

**MANUEL BLUM, ROBERT W. FLOYD, VAUGHAN PRATT, RONALD L. RIVEST, AND ROBERT E. TARJAN**

Department of Computer Science, Stanford University, Stanford, California 94305
Received November 14, 1972

The number of comparisons required to select the $i$-th smallest of $n$ numbers is shown to be at most a linear function of $n$ by analysis of a new selection algorithm—PICK. Specifically, no more than $5.4305n$ comparisons are ever required. This bound is improved for extreme values of $i$, and a new lower bound on the requisite number of comparisons is also proved.

**Remark.** Constants are high ⇒ not used in practice.

**Use theory as a guide.**
- Still worthwhile to seek practical linear-time (worst-case) algorithm.
- Until one is discovered, use quick-select (if you don’t need a full sort).
2.3 Quicksort

- quicksort
- selection
- duplicate keys
- system sorts
Duplicate keys

Often, purpose of sort is to bring items with equal keys together.

- Sort population by age.
- Remove duplicates from mailing list.
- Sort job applicants by college attended.

Typical characteristics of such applications.

- Huge array.
- Small number of key values.

<table>
<thead>
<tr>
<th>City</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chicago</td>
<td>09:00:00</td>
</tr>
<tr>
<td>Chicago</td>
<td>09:00:59</td>
</tr>
<tr>
<td>Chicago</td>
<td>09:03:13</td>
</tr>
<tr>
<td>Chicago</td>
<td>09:19:32</td>
</tr>
<tr>
<td>Chicago</td>
<td>09:19:46</td>
</tr>
<tr>
<td>Chicago</td>
<td>09:21:05</td>
</tr>
<tr>
<td>Chicago</td>
<td>09:25:52</td>
</tr>
<tr>
<td>Chicago</td>
<td>09:35:21</td>
</tr>
<tr>
<td>Chicago</td>
<td>09:00:03</td>
</tr>
<tr>
<td>Houston</td>
<td>09:01:10</td>
</tr>
<tr>
<td>Houston</td>
<td>09:00:13</td>
</tr>
<tr>
<td>Phoenix</td>
<td>09:37:44</td>
</tr>
<tr>
<td>Phoenix</td>
<td>09:00:00</td>
</tr>
<tr>
<td>Phoenix</td>
<td>09:14:25</td>
</tr>
<tr>
<td>Seattle</td>
<td>09:10:25</td>
</tr>
<tr>
<td>Seattle</td>
<td>09:36:14</td>
</tr>
<tr>
<td>Seattle</td>
<td>09:22:43</td>
</tr>
<tr>
<td>Seattle</td>
<td>09:10:11</td>
</tr>
<tr>
<td>Seattle</td>
<td>09:22:54</td>
</tr>
</tbody>
</table>
When partitioning, how should we handle keys equal to partitioning key?

A. 

B. 

C. Either A or B.
War story (system sort in C)

**Bug.** A `qsort()` call in C that should have taken seconds was taking minutes to sort a random array of 0s and 1s.

Why is `qsort()` so slow?

---

<table>
<thead>
<tr>
<th>A₀</th>
<th>A₁</th>
<th>A₂</th>
<th>A₃</th>
<th>A₄</th>
<th>A₅</th>
<th>A₆</th>
<th>A₇</th>
<th>A₈</th>
<th>A₉</th>
<th>A₁₀</th>
<th>A₁₁</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A₁</td>
<td>A₂</td>
<td>A₃</td>
<td>A₄</td>
<td>A₅</td>
<td>A₆</td>
<td>A₇</td>
<td>A₈</td>
<td>A₉</td>
<td>A₁₀</td>
<td>A₁₁</td>
</tr>
</tbody>
</table>

**skip over equal keys**

i

j

---

<table>
<thead>
<tr>
<th>A₀</th>
<th>A₁</th>
<th>A₂</th>
<th>A₃</th>
<th>A₄</th>
<th>A₅</th>
<th>A₆</th>
<th>A₇</th>
<th>A₈</th>
<th>A₉</th>
<th>A₁₀</th>
<th>A₁₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₀</td>
<td>A₁</td>
<td>A₂</td>
<td>A₃</td>
<td>A₄</td>
<td>A₅</td>
<td>A₆</td>
<td>A₇</td>
<td>A₈</td>
<td>A₉</td>
<td>A₁₀</td>
<td>A₁₁</td>
</tr>
</tbody>
</table>

**stop on equal keys**

i

j

---

35
Duplicate keys: partitioning strategies

**Bad.** Don’t stop scans on equal keys.

[ quadratic number of compares when all keys equal ]

\[ \text{B A A B A B B C C C A A A A A A A A} \]

**Good.** Stop scans on equal keys.

[ \sim n \lg n \text{ compares when all keys equal } ]

\[ \text{B A A B A B C C B C B A A A A A A} \]

**Better.** Put all equal keys in place. How?

[ \sim n \text{ compares when all keys equal } ]

\[ \text{A A A B B B B C C C A A A A A A} \]
Problem. [Edsger Dijkstra] Given an array of $n$ buckets, each containing a red, white, or blue pebble, sort them by color.

Operations allowed.

- $\text{swap}(i, j)$: swap the pebble in bucket $i$ with the pebble in bucket $j$.
- $\text{color}(i)$: color of pebble in bucket $i$.

Requirements.

- Exactly $n$ calls to $\text{color}()$.
- At most $n$ calls to $\text{swap}()$.
- Constant extra space.
3-way partitioning

**Goal.** Partition array into three parts so that:

- White: entries between \( \lt \) and \( \gt \) equal to the partition item.
- Red: smaller entries to left of \( \lt \).
- Blue: larger entries to the right of \( \gt \).

**Dutch national flag problem.** [Edsger Dijkstra]

- Conventional wisdom until mid 1990s: not worth doing.
- Now incorporated into C library \texttt{qsort()} and Java 6 system sort.
Dijkstra’s 3-way partitioning algorithm: demo

- Let $v$ be partitioning item $a[lo]$.
- Scan $i$ from left to right.
  - $(a[i] < v)$: exchange $a[lt]$ with $a[i]$; increment both $lt$ and $i$
  - $(a[i] > v)$: exchange $a[gt]$ with $a[i]$; decrement $gt$
  - $(a[i] == v)$: increment $i$
Dijkstra’s 3-way partitioning algorithm: demo

- Let \( v \) be partitioning item \( a[lo] \).
- Scan \( i \) from left to right.
  - \((a[i] < v)\): exchange \( a[lt] \) with \( a[i] \); increment both \( lt \) and \( i \)
  - \((a[i] > v)\): exchange \( a[gt] \) with \( a[i] \); decrement \( gt \)
  - \((a[i] == v)\): increment \( i \)
private static void sort(Comparable[] a, int lo, int hi)
{
    if (hi <= lo) return;
    int lt = lo, gt = hi;
    Comparable v = a[lo];
    int i = lo + 1;
    while (i <= gt)
    {
        int cmp = a[i].compareTo(v);
        if (cmp < 0) exch(a, lt++, i++);
        else if (cmp > 0) exch(a, i, gt--);
        else i++;
    }
    sort(a, lo, lt - 1);
    sort(a, gt + 1, hi);
}
3-way quicksort: visual trace

equal to partitioning element
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>inplace?</th>
<th>stable?</th>
<th>best</th>
<th>average</th>
<th>worst</th>
<th>remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>selection</td>
<td>✔</td>
<td></td>
<td>$\frac{1}{2} n^2$</td>
<td>$\frac{1}{2} n^2$</td>
<td>$\frac{1}{2} n^2$</td>
<td>$n$ exchanges</td>
</tr>
<tr>
<td>insertion</td>
<td>✔</td>
<td>✔</td>
<td>$n$</td>
<td>$\frac{1}{4} n^2$</td>
<td>$\frac{1}{2} n^2$</td>
<td>use for small $n$ or partially sorted</td>
</tr>
<tr>
<td>shell</td>
<td>✔</td>
<td></td>
<td>$n \log_3 n$</td>
<td>?</td>
<td>$c n^{3/2}$</td>
<td>tight code; subquadratic</td>
</tr>
<tr>
<td>merge</td>
<td>✔</td>
<td>✔</td>
<td>$\frac{1}{2} n \log n$</td>
<td>$n \log n$</td>
<td>$n \log n$</td>
<td>$n \log n$ guarantee; stable</td>
</tr>
<tr>
<td>timsort</td>
<td>✔</td>
<td>✔</td>
<td>$n$</td>
<td>$n \log n$</td>
<td>$n \log n$</td>
<td>improves mergesort when pre-existing order</td>
</tr>
<tr>
<td>quick</td>
<td>✔</td>
<td></td>
<td>$n \log n$</td>
<td>$2 n \ln n$</td>
<td>$\frac{1}{2} n^2$</td>
<td>$n \log n$ probabilistic guarantee; fastest in practice</td>
</tr>
<tr>
<td>3-way quick</td>
<td>✔</td>
<td>✔</td>
<td>$n$</td>
<td>$2 n \ln n$</td>
<td>$\frac{1}{2} n^2$</td>
<td>improves quicksort when duplicate keys</td>
</tr>
<tr>
<td>?</td>
<td>✔</td>
<td>✔</td>
<td>$n$</td>
<td>$n \log n$</td>
<td>$n \log n$</td>
<td>holy sorting grail</td>
</tr>
</tbody>
</table>
2.3 Quicksort

- quicksort
- selection
- duplicate keys
- system sorts
Sorting applications

**Sorting algorithms are essential in a broad variety of applications:**

- Sort a list of names.
- Organize an MP3 library.
- Display Google PageRank results.
- List RSS feed in reverse chronological order.
  
- Find the median.
- Identify statistical outliers.
- Binary search in a database.
- Find duplicates in a mailing list.

- Data compression.
- Computer graphics.
- Computational biology.
- Load balancing on a parallel computer.
  
...
Engineering a system sort (in 1993)

Bentley–McIlroy quicksort.

- Cutoff to insertion sort for small subarrays.
- Partitioning item: median of 3 or Tukey’s ninther.
- Partitioning scheme: Bentley–McIlroy 3-way partitioning.

Very widely used. C, C++, Java 6, ....
Replacement of quicksort in java.util.Arrays with new dual-pivot quicksort

Hello All,

I'd like to share with you new Dual-Pivot Quicksort which is faster than the known implementations (theoretically and experimental). I'd like to propose to replace the JDK's Quicksort implementation by new one.

...

The new Dual-Pivot Quicksort uses *two* pivots elements in this manner:

1. Pick an elements P1, P2, called pivots from the array.
2. Assume that P1 <= P2, otherwise swap it.
3. Reorder the array into three parts: those less than the smaller pivot, those larger than the larger pivot, and in between are those elements between (or equal to) the two pivots.
4. Recursively sort the sub-arrays.

The invariant of the Dual-Pivot Quicksort is:

$$[ < P1 | P1 <= & <= P2 } > P2 ]$$

...
Another Java mailing list post (Yaroslavskiy–Bloch–Bentley)

Replacement of quicksort in java.util.Arrays with new dual-pivot quicksort

Date: Thu, 29 Oct 2009 11:19:39 +0000
Subject: Replace quicksort in java.util.Arrays with dual-pivot implementation

Changeset: b05abb410c52
Author: alanb
Date: 2009-10-29 11:18 +0000
URL: http://hg.openjdk.java.net/jdk7/tl/jdk/rev/b05abb410c52

6880672: Replace quicksort in java.util.Arrays with dual-pivot implementation
Reviewed-by: jjb
Contributed-by: vladimir.yaroslavskiy at sun.com, joshua.bloch at google.com, jbentley at avaya.com

! make/java/java/FILES_java.gmk
! src/share/classes/java/util/Arrays.java
+ src/share/classes/java/util/DualPivotQuicksort.java

http://mail.openjdk.java.net/pipermail/compiler-dev/2009-October.txt
Dual-pivot quicksort

Use two partitioning items $p_1$ and $p_2$ and partition into three subarrays:

- Keys less than $p_1$.
- Keys between $p_1$ and $p_2$.
- Keys greater than $p_2$.

<table>
<thead>
<tr>
<th>$&lt; p_1$</th>
<th>$p_1$</th>
<th>$\geq p_1$ and $\leq p_2$</th>
<th>$p_2$</th>
<th>$&gt; p_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>↑</td>
<td>↑</td>
<td>↑</td>
<td>↑</td>
<td>↑</td>
</tr>
<tr>
<td>lo</td>
<td>lt</td>
<td>gt</td>
<td>hi</td>
<td></td>
</tr>
</tbody>
</table>

Recursively sort three subarrays (skip middle subarray if $p_1 = p_2$).

degenerates to Dijkstra’s 3-way partitioning

Now widely used. Java 8, Python unstable sort, Android, ...
Why does 2-pivot quicksort perform better than 1-pivot?

A. Fewer compares.
B. Fewer exchanges.
C. Both A and B.
D. Neither A nor B.
Arrays.sort(), Arrays.parallelSort().

- Has one method for objects that are Comparable.
- Has an overloaded method for each primitive type.
- Has an overloaded method for use with a Comparator.
- Has overloaded methods for sorting subarrays.

Algorithms.

- Dual-pivot quicksort for primitive types.
- Timsort for reference types.
- Parallel mergesort for Arrays.parallelSort().

Q. Why use different algorithms for primitive and reference types?

Bottom line. Use the system sort!