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 20th century in science and engineering.

Mergesort. [last lecture]

Quicksort. [this lecture]
Quicksort t-shirt

```java
public static void quickSort(Comparable[] a) {
    if (a == null) return;
    quickSort(a, 0, a.length - 1);
}

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

private static int partition(Comparable[] a, int lo, int hi) {
    int pi = lo + (hi - lo) / 2;
    swap(a, lo, pi);
    Comparable v = a[lo];
    for (int i = lo + 1; i <= hi; i++) {
        if (a[i].compareTo(v) < 0) {
            swap(a, i, hi--);
        }
    }
    return lo;
}

private static boolean isSorted(Comparable[] a) {
    for (int i = 0; i < a.length - 1; i++) {
        if (a[i].compareTo(a[i + 1]) > 0) return false;
    }
    return true;
}

private static void show(Comparable[] a) {
    for (int i = 0; i < a.length; i++) {
        System.out.println(a[i]);
    }
}

public static void main(String[] args) {
    String[] a = {"CS", "at", "Princeton"};
    quickSort(a, 0, a.length - 1);
    System.out.println(Arrays.toString(a));
}
```

CS @ Princeton
2.3 Quicksort

- quicksort
- selection
- duplicate keys
- system sorts
Quicksort overview demo
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.
Tony Hoare

- Invented quicksort to translate Russian into English.
  - [but couldn’t explain his algorithm or implement it!]
- Learned Algol 60 (and recursion).
- Implemented quicksort.

**Algorithm 64**

QuickSort

C. A. R. Hoare

Elliott Brothers Ltd., Borehamwood, Hertfordshire, Eng.

procedure quicksort (A,M,N); value M,N;
array A; integer M,N;
comment Quicksort is a very fast and convenient method of sorting an array in the random-access store of a computer. The entire contents of the store may be sorted, since no extra space is required. The average number of comparisons made is 2(M−N) ln (N−M), and the average number of exchanges is one sixth this amount. Suitable refinements of this method will be desirable for its implementation on any actual computer;

begin integer I,J;
if M < N then begin partition (A,M,N,I,J);
  quicksort (A,M,I);
  quicksort (A, I, N)
end quicksort

Communications of the ACM (July 1961)
Tony Hoare

- Invented quicksort to translate Russian into English.
  - [ but couldn't explain his algorithm or implement it! ]
- Learned Algol 60 (and recursion).
- Implemented quicksort.

“There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.”

“I call it my billion-dollar mistake. It was the invention of the null reference in 1965… This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.”
Bob Sedgewick

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

Implementing Quick sort Programs
Robert Sedgewick
Brown University

This paper is a practical study of how to implement the Quick sort sorting algorithm and its best variants on real computers, including how to apply various code optimization techniques. A detailed implementation combining the most effective improvements to Quick sort is given, along with a discussion of how to implement it in assembly language. Analytic results describing the performance of the programs are summarized. A variety of special situations are considered from a practical standpoint to illustrate Quick sort’s wide applicability as an internal sorting method which requires negligible extra storage.

Key Words and Phrases: Quick sort, analysis of algorithms, code optimization, sorting

CR Categories: 4.0, 4.6, 5.25, 5.31, 5.5

Acta Informatica 7, 327—355 (1977)
© by Springer-Verlag 1977

The Analysis of Quick sort Programs*

Robert Sedgewick

Received January 19, 1976

Summary. The Quick sort sorting algorithm and its best variants are presented and analyzed. Results are derived which make it possible to obtain exact formulas describing the total expected running time of particular implementations on real computers of Quick sort and an improvement called the median-of-three modification. Detailed analysis of the effect of an implementation technique called loop unwrapping is presented. The paper is intended not only to present results of direct practical utility, but also to illustrate the intriguing mathematics which arises in the complete analysis of this important algorithm.
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!
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;
}
```

before

<table>
<thead>
<tr>
<th>≤</th>
<th>≤</th>
<th>≥</th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

↑ lo

↑ hi

during

<table>
<thead>
<tr>
<th>≤</th>
<th>≤</th>
<th>≥</th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

↑ i

↑ j

after

<table>
<thead>
<tr>
<th>≤</th>
<th>≤</th>
<th>≥</th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
<td>v</td>
<td>v</td>
</tr>
</tbody>
</table>

↑ lo

↑ j

↑ hi

http://algs4.cs.princeton.edu/23quick/Quick.java.html
How many compares (in worst case) to partition an array of length \( n \)?

A. \( \sim \frac{1}{4} n \)

B. \( \sim \frac{1}{2} n \)

C. \( \sim n \)

D. \( \sim n \lg n \)
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);
    }
}

http://algs4.cs.princeton.edu/23quick/Quick.java.html
Quicksort trace (array contents after each partition)

<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></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td></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></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></td>
</tr>
<tr>
<td>14</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>15</td>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>

<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>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)
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 is not worth the cost.

Terminating the loop. Testing whether the pointers cross is trickier than it might seem.

Equal keys. When duplicates are present, it is (counter-intuitively) better to stop scans on keys equal to the partitioning item’s key. stay tuned

Preserving randomness. Shuffling is needed for performance guarantee.
Equivalent alternative. Pick a random partitioning item in each subarray.
Running time estimates:

- Algol 60 implementation.
- National Elliott 405 computer.

### Table 1

<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>Computer</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.
Worst case. Number of compares is $\sim \frac{1}{2} n^2$. 

![Quicksort worst-case analysis](image)
Quicksort: best-case analysis

Best case. Number of compares is $\sim n \lg n$. 

![Table and Diagram](image-url)
Proposition. The average number of compares $C_n$ to quicksort an array of $n$ distinct keys is $\sim 2n \ln n$ (and the number of exchanges is $\sim \sqrt{3} n \ln n$).

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:

\[
nC_n = n(n+1) + 2(C_0 + C_1 + \ldots + C_{n-1})
\]

- Subtract from this equation the same equation for $n - 1$:

\[
nC_n - (n-1)C_{n-1} = 2n + 2C_{n-1}
\]

- Rearrange terms and divide by $n(n+1)$:

\[
\frac{C_n}{n+1} = \frac{C_{n-1}}{n} + \frac{2}{n+1}
\]
Quicksort: average-case 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:

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

Quicksort is a (Las Vegas) **randomized algorithm**.
- Guaranteed to be correct.
- Running time depends on random shuffle.

**Average case.** Expected number of compares is $\sim 1.39 \, n \, \lg n$.
- 39% more compares than mergesort.
- Faster than mergesort in practice because of less data movement.

**Best case.** Number of compares is $\sim n \, \lg n$.

**Worst case.** Number of compares is $\sim \frac{1}{2} \, n^2$.
[ but more likely that lightning bolt strikes computer during execution ]
Quicksort properties

**Proposition.** Quicksort is an *in-place* sorting algorithm.

**Pf.**

- Partitioning: constant extra space.
- Depth of recursion: logarithmic extra space (with high probability).

---

**Proposition.** Quicksort is *not* stable.

**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>B₁</td>
<td>C₁</td>
<td>C₂</td>
<td>A₁</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>
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);
}
```
Median of sample.

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

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

```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 the \( k^{th} \) smallest item.

**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 \( k = 1, 2, 3 \). 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 $k$.

```
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.

**Proof sketch.**

- Intuitively, each partitioning step splits array approximately in half:
  \[ n + n/2 + n/4 + \ldots + 1 \sim 2n \text{ compares.} \]
- Formal analysis similar to quicksort analysis yields:

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

- Ex: \((2 + 2 \ln 2) n \approx 3.38 n\) compares to find median \((k = n/2)\).
Theoretical context for selection


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.
War story (system sort in C)

A beautiful bug report.  [Allan Wilks and Rick Becker, 1991]

We found that qsort is unbearably slow on "organ-pipe" inputs like "01233210":

```c
main (int argc, char**argv) {
    int n = atoi(argv[1]), i, x[100000];
    for (i = 0; i < n; i++)
        x[i] = i;
    for ( ; i < 2*n; i++)
        x[i] = 2*n-i-1;
    qsort(x, 2*n, sizeof(int), intcmp);
}
```

Here are the timings on our machine:

$ time a.out 2000
real  5.85s
$ time a.out 4000
real  21.64s
$ time a.out 8000
real  85.11s
War story (system sort in C)

**Bug.** A `qsort()` call that should have taken seconds was taking minutes.

Why is `qsort()` so slow?

At the time, almost all `qsort()` implementations based on those in:

- Version 7 Unix (1979): quadratic time to sort organ-pipe arrays.
- BSD Unix (1983): quadratic time to sort random arrays of 0s and 1s.
Duplicate keys: stop on equal keys

Our partitioning subroutine stops both scans on equal keys.

Q. Why not continue scans on equal keys?
What is the result of partitioning the following array (skip over equal keys)?

A. A A A A A A A A A A A A A A A A

B. A A A A A A A A A A A A A A A A A A

C. A A A A A A A A A A A A A A A A A A

D. I don't know.
QuickSort: quiz 3

What is the result of partitioning the following array (stop on equal keys)?

A. [A A A A A A A A A A A A A A A A A A A]

B. [A A A A A A A A A A A A A A A A A A A]

C. [A A A A A A A A A A A A A A A A A A A]

D. I don't know.
### Partitioning an array with all equal keys

<table>
<thead>
<tr>
<th>i</th>
<th>j</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>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>15</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>15</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>14</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>14</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>3</td>
<td>13</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>3</td>
<td>13</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>4</td>
<td>12</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>4</td>
<td>12</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>5</td>
<td>11</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>5</td>
<td>11</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>7</td>
<td>9</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>7</td>
<td>9</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>7</td>
<td>9</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>7</td>
<td>9</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
</tbody>
</table>
Duplicate keys: partitioning strategies

Bad. Don’t stop scans on equal keys.

\[
\sim \frac{1}{2} n^2 \text{ compares when all keys equal }
\]

\[
\begin{array}{cccccccc}
\end{array}
\]

Good. Stop scans on equal keys.

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

\[
\begin{array}{cccccccc}
\end{array}
\]

Better. Put all equal keys in place. How?

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

\[
\begin{array}{cccccccc}
\end{array}
\]
**Dutch National Flag Problem**

**Problem.** [Edsger Dijkstra] Given an array of $n$ buckets, each containing a red, white, or blue pebble, sort them by color.

- **Input:**
  - Input array:
    - Red, white, blue pebbles.
  - Sorted array:
    - Red, white, blue pebbles.

- **Operations allowed.**
  - $swap(i, j)$: swap the pebble in bucket $i$ with the pebble in bucket $j$.
  - $color(i)$: color of pebble in bucket $i$.

- **Requirements.**
  - Exactly $n$ calls to $color()$.
  - At most $n$ calls to $swap()$.
  - Constant extra space.
3-way partitioning

**Goal.** Partition array into three parts so that:
- Entries between $lt$ and $gt$ equal to the partition item.
- No larger entries to left of $lt$.
- No smaller entries to right of $gt$.

---

**Dutch national flag problem.** [Edsger Dijkstra]
- Conventional wisdom until mid 1990s: not worth doing.
- Now incorporated into C library `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 $i$

![Diagram of 3-way partitioning algorithm]

**Invariant**

<table>
<thead>
<tr>
<th>$&lt;$ $v$</th>
<th>$=$ $v$</th>
<th>$&gt;$ $v$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$lt$</td>
<td>$i$</td>
<td>$gt$</td>
</tr>
</tbody>
</table>
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

\[
\begin{array}{cccccccccccccccc}
\uparrow & \uparrow & \uparrow & \uparrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow \\
lo & hi \\
\end{array}
\]

invariant

\[
\begin{array}{cccc}
< v & = v & \text{gray} & > v \\
\uparrow & \uparrow & \uparrow & \uparrow \\
lt & i & gt \\
\end{array}
\]
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;
    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
Duplicate keys: lower bound

**Sorting lower bound.** If there are \( n \) distinct keys and the \( i^{th} \) one occurs \( x_i \) times, then any compare-based sorting algorithm must use at least

\[
\lg \left( \frac{n!}{x_1! \cdot x_2! \cdot \ldots \cdot x_n!} \right) \sim \sum_{i=1}^{n} -x_i \lg \frac{x_i}{n}
\]

\( n \lg n \) when all distinct; linear when only a constant number of distinct keys compares in the worst case.

**Proposition.** The expected number of compares to 3-way quicksort an array is **entropy optimal** (proportional to sorting lower bound).

**Pf.** [beyond scope of course]

**Bottom line.** Quicksort with 3-way partitioning reduces running time from linearithmic to linear in broad class of applications.
## Sorting summary

<table>
<thead>
<tr>
<th></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.

...
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:

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

...
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/FINES_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>$\uparrow$</td>
<td>$\uparrow$</td>
<td>$\uparrow$</td>
<td>$\uparrow$</td>
<td>$\uparrow$</td>
</tr>
<tr>
<td>lo</td>
<td>lt</td>
<td>gt</td>
<td>hi</td>
<td></td>
</tr>
</tbody>
</table>

Recursively sort three subarrays.

**Note.** Skip middle subarray if $p_1 = p_2$. 

degenerates to Dijkstra's 3-way partitioning
Initialization.

- Choose \( a[lo] \) and \( a[hi] \) as partitioning items.
- Exchange if necessary to ensure \( a[lo] \leq a[hi] \).
Dual-pivot partitioning demo

Main loop. Repeat until i and gt pointers cross.

- If \((a[i] < a[lo])\), exchange \(a[i]\) with \(a[lt]\) and increment \(lt\) and \(i\).
- Else if \((a[i] > a[hi])\), exchange \(a[i]\) with \(a[gt]\) and decrement \(gt\).
- Else, increment \(i\).

<table>
<thead>
<tr>
<th>&lt; (p_1)</th>
<th>(p_1)</th>
<th>(\geq p_1) and (\leq p_2)</th>
<th>?</th>
<th>(p_2)</th>
<th>&gt; (p_2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\uparrow)</td>
<td>(\uparrow)</td>
<td>(\uparrow)</td>
<td>(\uparrow)</td>
<td>(\uparrow)</td>
<td>(\uparrow)</td>
</tr>
<tr>
<td>(lo)</td>
<td>(lt)</td>
<td>(i)</td>
<td>(gt)</td>
<td>(hi)</td>
<td></td>
</tr>
</tbody>
</table>
Finalize.

- Exchange $a[hi]$ with $a[++]t$.

### 3-way partitioned

<table>
<thead>
<tr>
<th></th>
<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>$lo$</td>
<td>↑</td>
<td></td>
<td></td>
<td></td>
<td>↑ hi</td>
</tr>
<tr>
<td>$lt$</td>
<td>↑</td>
<td></td>
<td></td>
<td></td>
<td>↑ hi</td>
</tr>
<tr>
<td>$gt$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>↑ hi</td>
</tr>
<tr>
<td>$hi$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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></th>
<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>lo</td>
<td>↑</td>
<td>↑</td>
<td>↑</td>
<td>↑</td>
<td>↑</td>
</tr>
<tr>
<td>lt</td>
<td>↑</td>
<td>↑</td>
<td>↑</td>
<td>↑</td>
<td>↑</td>
</tr>
<tr>
<td>gt</td>
<td>↑</td>
<td>↑</td>
<td>↑</td>
<td>↑</td>
<td>↑</td>
</tr>
<tr>
<td>hi</td>
<td>↑</td>
<td>↑</td>
<td>↑</td>
<td>↑</td>
<td>↑</td>
</tr>
</tbody>
</table>

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.
System sort in Java 8

`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!