2.3 Quicksort

- quicksort
- selection
- duplicate keys
- system sorts
Two classic sorting algorithms

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.

- Java sort for objects.
- Perl, C++ stable sort, Python stable sort, Firefox JavaScript, ...

Quicksort.

- Java sort for primitive types.
- C qsort, Unix, Visual C++, Python, Matlab, Chrome JavaScript, ...
public static void quicksort(char[] items, int left, int right)
{
    int i, j;
    char x, y;
    i = left; j = right;
    x = items[(left + right) / 2];
    do
    {
        while ((items[j] < x) && (i < right)) i++;
        while ((x < items[i]) && (j > left)) j--;
        if (i <= j)
        {
            y = items[i];
            items[i] = items[j];
            items[j] = y;
            i++; j--;
        }
    } while (i <= j);
    if (left < j) quicksort(items, left, j);
    if (i < right) quicksort(items, i, right);
}
2.3 QUICKSORT

- quicksort
- selection
- duplicate keys
- system sorts
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.

![QuickSort Diagram]

Sir Charles Antony Richard Hoare
1980 Turing Award
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 \(a[j]\).
Quicksort: Java code for partitioning

```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;
}
```

Quicksort partitioning overview

before

<table>
<thead>
<tr>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>↑</td>
</tr>
<tr>
<td>lo</td>
</tr>
<tr>
<td>↑</td>
</tr>
<tr>
<td>hi</td>
</tr>
</tbody>
</table>

during

| ≤ V |
| V |
| ≥ V |
| ↑ |
| i |
| ↑ |
| j |
| ↑ |
| hi |

after

| ≤ V |
| V |
| ≥ V |
| ↑ |
| lo |
| ↑ |
| j |
| ↑ |
| hi |
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

Quicksort trace (array contents after each partition)

<table>
<thead>
<tr>
<th>lo</th>
<th>0</th>
<th>j</th>
<th>1</th>
<th>hi</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>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>
<td></td>
<td></td>
</tr>
<tr>
<td></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>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<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>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<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>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<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>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<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>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<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>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<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>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<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>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<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>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<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>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<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>S</td>
<td>Q</td>
<td>R</td>
<td>T</td>
<td>U</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<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>R</td>
<td>Q</td>
<td>S</td>
<td>T</td>
<td>U</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<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>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<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>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<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>
<td></td>
<td></td>
</tr>
</tbody>
</table>

no partition for subarrays of size 1

result

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

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 a bit trickier than it might seem.

Staying in bounds. The \( j = 10 \) test is redundant (why?), but the \( i = hi \) test is not.

Preserving randomness. Shuffling is needed for performance guarantee.

Equal keys. When duplicates are present, it is (counter-intuitively) better to stop on keys equal to the partitioning item's key.
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.
# Quicksort: best-case analysis

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

![Quicksort Best Case Example](image)

<table>
<thead>
<tr>
<th>a[]</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>lo</td>
<td>j</td>
<td>hi</td>
<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>
<td>11</td>
<td>12</td>
</tr>
<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>
</tr>
<tr>
<td>initial values</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>
</tr>
<tr>
<td>random shuffle</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>
</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>
</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>
</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>
</tr>
<tr>
<td>0</td>
<td>0</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>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>6</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>
</tr>
<tr>
<td>4</td>
<td>4</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>
</tr>
<tr>
<td>4</td>
<td>6</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>
</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>
</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>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>9</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>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>9</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>
</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>
</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>
</tr>
<tr>
<td>14</td>
<td>14</td>
<td>13</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>
</tr>
</tbody>
</table>
Quicksort: worst-case analysis

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

<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>initial values</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>
<td></td>
<td></td>
</tr>
<tr>
<td>random shuffle</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>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</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>1</td>
<td>1</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>2</td>
<td>2</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>3</td>
<td>3</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>4</td>
<td>4</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>5</td>
<td>5</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>6</td>
<td>6</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>7</td>
<td>7</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>8</td>
<td>8</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>9</td>
<td>9</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>10</td>
<td>10</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>11</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>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>13</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>
</tbody>
</table>
Quicksort: average-case analysis

**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 $\sim \frac{1}{3} N \ln N$).

**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:
  $$NC_N = N(N + 1) + 2(C_0 + C_1 + \ldots + C_{N-1})$$

- Subtract this from 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 above equation:

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

previous equation

= \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:

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

\[
\sim 2(N + 1) \int_{3}^{N+1} \frac{1}{x} \, dx
\]

- Finally, the desired result:

\[
C_N \sim 2(N + 1) \ln N \approx 1.39N \lg N
\]
QuickSort: average-case analysis

**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 $\sim \frac{1}{3} N \ln N$).

**Pf 2.** Consider BST representation of keys 1 to $N$. 

```
shuffle

9 10 2 5 8 7 6 1 11 12 13 3 4
```

```
first partitioning item

first partitioning item in left subarray
```

```
3
/ \\
1 2

4
\ / \
6 5

8
/ \
10

11
/ \
12

13

11
/ \
13
```

18
**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 $\sim \frac{1}{3} N \ln N$).

**Pf 2.** Consider BST representation of keys 1 to $N$.

- A key is compared only with its ancestors and descendants.
- Probability $i$ and $j$ are compared equals $2 / |j - i + 1|$.

![Diagram of BST representation with annotations showing comparison probabilities and partitioning items.](image-url)
Quicksort: average-case analysis

**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 $\sim \frac{1}{3} N \ln N$).

**Pf 2.** Consider BST representation of keys 1 to $N$.

- A key is compared only with its ancestors and descendants.
- Probability $i$ and $j$ are compared equals $2/|j - i + 1|$.

- **Expected number of compares**
  
  $$
  E_n = \sum_{i=1}^{N} \sum_{j=i+1}^{N} \frac{2}{j-i+1} = 2 \sum_{i=1}^{N} \sum_{j=2}^{N-i+1} \frac{1}{j} \\
  \leq 2N \sum_{j=1}^{N} \frac{1}{j} \\
  \sim 2N \int_{x=1}^{N} \frac{1}{x} \, dx \\
  = 2N \ln N
  $$

  all pairs $i$ and $j$
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 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.**

<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>0</td>
<td>1</td>
<td>A₁</td>
<td>B₁</td>
<td>C₂</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>
</tbody>
</table>

can guarantee logarithmic depth by recurring on smaller subarray before larger subarray


Quicksort: practical improvements

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 improvements

Median of sample.

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

\[ \sim \frac{12}{7} \text{ N In N compares (slightly fewer)} \]
\[ \sim \frac{12}{35} \text{ N In N exchanges (slightly more)} \]

```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

- Input
- Result of first partition
- Left subarray partially sorted
- Both subarrays partially sorted
- Result
2.3 Quicksort

- quicksort
- selection
- duplicate keys
- system sorts
Selection

Goal. Given an array of $N$ items, find the $k^{th}$ largest.

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 for each $k$?
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 \textit{linear} time on average.

\textbf{Pf 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:
  \[ C_N = 2N + 2k \ln (N/k) + 2(N-k) \ln (N/(N-k)) \]
  \[ (2 + 2 \ln 2)N \text{ to find the median} \]

\textbf{Remark.} Quick-select uses \( \sim \frac{1}{2} N^2 \) compares in the worst case, but (as with quicksort) the random shuffle provides a probabilistic guarantee.
Theoretical context for selection


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 \(5.4305^n\) 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.** But, constants are too high \(\Rightarrow\) 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
War story (C qsort function)

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 (C qsort function)

AT&T Bell Labs (1991). Allan Wilks and Rick Becker discovered that 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

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.
Duplicate keys

Mergesort with duplicate keys. Between $\frac{1}{2} N \lg N$ and $N \lg N$ compares.

Quicksort with duplicate keys. Algorithm goes quadratic unless partitioning stops on equal keys!

which is why ours does!
(but many textbook implementations do not)

S T O P O N E Q U A L K E Y S

swap if we don't stop on equal keys if we stop on equal keys
Duplicate keys: the problem

Mistake. Put all items equal to the partitioning item on one side.
Consequence. $\sim \frac{1}{2} N^2$ compares when all keys equal.

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

Recommended. Stop scans on items equal to the partitioning item.
Consequence. $\sim N \lg N$ compares when all keys equal.

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

Desirable. Put all items equal to the partitioning item in place.

\[
\begin{array}{cccccccc}
\end{array}
\]
3-way partitioning

**Goal.** Partition array into 3 parts so that:

- Entries between \( l_t \) and \( g_t \) equal to partition item \( v \).
- No larger entries to left of \( l_t \).
- No smaller entries to right of \( g_t \).

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

- Conventional wisdom until mid 1990s: not worth doing.
- New approach discovered when fixing mistake in C library `qsort()`.
- Now incorporated into `qsort()` and Java system sort.
Dijkstra 3-way partitioning 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 3-way partitioning 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: trace

<table>
<thead>
<tr>
<th>lt</th>
<th>i</th>
<th>gt</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>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>R</td>
<td>B</td>
<td>W</td>
<td>W</td>
<td>R</td>
<td>W</td>
<td>B</td>
<td>R</td>
<td>R</td>
<td>W</td>
<td>B</td>
<td>R</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11</td>
<td>R</td>
<td>B</td>
<td>W</td>
<td>W</td>
<td>R</td>
<td>W</td>
<td>B</td>
<td>R</td>
<td>R</td>
<td>W</td>
<td>B</td>
<td>R</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>11</td>
<td>B</td>
<td>R</td>
<td>W</td>
<td>W</td>
<td>R</td>
<td>W</td>
<td>B</td>
<td>R</td>
<td>R</td>
<td>W</td>
<td>B</td>
<td>R</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>10</td>
<td>B</td>
<td>R</td>
<td>R</td>
<td>W</td>
<td>R</td>
<td>W</td>
<td>B</td>
<td>R</td>
<td>R</td>
<td>W</td>
<td>B</td>
<td>W</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>10</td>
<td>B</td>
<td>R</td>
<td>R</td>
<td>W</td>
<td>R</td>
<td>W</td>
<td>B</td>
<td>R</td>
<td>R</td>
<td>W</td>
<td>B</td>
<td>W</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>9</td>
<td>B</td>
<td>R</td>
<td>R</td>
<td>B</td>
<td>R</td>
<td>W</td>
<td>B</td>
<td>R</td>
<td>R</td>
<td>W</td>
<td>W</td>
<td>W</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>9</td>
<td>B</td>
<td>B</td>
<td>R</td>
<td>R</td>
<td>B</td>
<td>R</td>
<td>W</td>
<td>B</td>
<td>R</td>
<td>R</td>
<td>W</td>
<td>W</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>9</td>
<td>B</td>
<td>B</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>W</td>
<td>B</td>
<td>R</td>
<td>R</td>
<td>W</td>
<td>W</td>
<td>W</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>8</td>
<td>B</td>
<td>B</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>W</td>
<td>B</td>
<td>R</td>
<td>R</td>
<td>W</td>
<td>W</td>
<td>W</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>7</td>
<td>B</td>
<td>B</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>B</td>
<td>R</td>
<td>R</td>
<td>W</td>
<td>W</td>
<td>W</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>7</td>
<td>B</td>
<td>B</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>B</td>
<td>R</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>7</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>7</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>W</td>
<td>W</td>
<td>W</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>7</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>W</td>
<td>W</td>
</tr>
</tbody>
</table>

3-way partitioning trace (array contents after each loop iteration)
3-way quicksort: Java implementation

```java
class ThreeWayQuickSort {
  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 partitioning:

- **before**
  - Elements before `v` are less than `v`.
  - Elements after `v` are greater than `v`.
  - Elements equal to `v` are between `v` and another partition.

- **during**
  - Splits `lo` and `hi` into `lt`, `i`, and `gt` partitions.

- **after**
  - Final partitions are complete.
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, any compare-based sorting algorithm must use at least

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

compares in the worst case.

**Proposition.** [Sedgewick-Bentley, 1997]
Quicksort with 3-way partitioning is entropy-optimal.

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

**Bottom line.** Randomized quicksort with 3-way partitioning reduces running time from linearithmic to linear in broad class of applications.
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.  \hspace{1cm} obvious applications
- Display Google PageRank results.
- List RSS feed in reverse chronological order.

- Find the median.
- Identify statistical outliers.  \hspace{1cm} problems become easy once items are in sorted order
- Binary search in a database.
- Find duplicates in a mailing list.

- Data compression.
- Computer graphics.  \hspace{1cm} non-obvious applications
- Computational biology.
- Load balancing on a parallel computer.

\ldots
Java system sorts

Arrays.sort().

- Has different method for each primitive type.
- Has a method for data types that implement Comparable.
- Has a method that uses a Comparator.
- Uses tuned quicksort for primitive types; tuned mergesort for objects.

```java
import java.util.Arrays;

public class StringSort
{
    public static void main(String[] args)
    {
        String[] a = StdIn.readStrings();
        Arrays.sort(a);
        for (int i = 0; i < N; i++)
            StdOut.println(a[i]);
    }
}
```

Q. Why use different algorithms for primitive and reference types?
Basic algorithm = quicksort.

- Cutoff to insertion sort for small subarrays.
- Partitioning scheme: Bentley-McIlroy 3-way partitioning.
- Partitioning item.
  - small arrays: middle entry
  - medium arrays: median of 3
  - large arrays: Tukey's ninther

Now very widely used.  C, C++, Java 6, ....
Tukey's ninther

Tukey's ninther. Median of the median of 3 samples, each of 3 entries.
- Approximates the median of 9.
- Uses at most 12 compares.

Q. Why use Tukey's ninther?
A. Better partitioning than random shuffle and less costly.
A beautiful mailing list post (Yaroslavskiy, September 2011)

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.

Description
----------
The classical Quicksort algorithm uses the following scheme:

1. Pick an element P, called a pivot, from the array.
2. Reorder the array so that all elements, which are less than the pivot, come before the pivot and all elements greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot element is in its final position.
3. Recursively sort the sub-array of lesser elements and the sub-array of greater elements.

The invariant of classical Quicksort is:

\[ [ \leq p | \geq p ] \]

There are several modifications of the schema:

\[ [ < p | = p | > p ] \text{ or } [ = p | < p | > p | = p ] \]

But all of them use *one* pivot.

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 \( \leq \) 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:

\[ [ < P1 | P1 \leq & \leq P2 | > P2 ] \]

...
**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>$p_1 &lt; p_1$</th>
<th>$p_1$</th>
<th>$p_1 \leq$ 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>hi</td>
</tr>
<tr>
<td>lt</td>
<td>↑</td>
<td>↑</td>
<td>↑</td>
<td></td>
</tr>
<tr>
<td>gt</td>
<td></td>
<td>↑</td>
<td></td>
<td></td>
</tr>
<tr>
<td>hi</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Recursively sort three subarrays.

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

similar to Dijkstra's 3-way partitioning
Dual-pivot partitioning demo

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></th>
<th>(&lt; p_1)</th>
<th>(p_1 \leq \text{ and } \leq p_2)</th>
<th>(?)</th>
<th>(&gt; p_2)</th>
<th>(p_2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(\uparrow)</td>
<td>(\uparrow)</td>
<td>(\uparrow)</td>
<td>(\uparrow)</td>
<td>(\uparrow)</td>
</tr>
<tr>
<td>(p_1)</td>
<td>(p_2)</td>
<td>(i)</td>
<td>(lt)</td>
<td>(lo)</td>
<td></td>
</tr>
</tbody>
</table>
Finalize.

- Exchange a[lo] with a[--lt].
- Exchange a[hi] with a[+gt].
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>$p_1 &lt; p_1$</th>
<th>$p_1$</th>
<th>$p_1 \leq \text{ and } \leq p_2$</th>
<th>$p_2$</th>
<th>$&gt; p_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>lo</td>
<td>lt</td>
<td></td>
<td></td>
<td>hi</td>
</tr>
</tbody>
</table>

**Proposition.** [Wild and Nevel] 1.9 $N \ln N$ compares and 0.6 $N \ln N$ exchanges.

**Improvements.** Take 5 random items, use 2nd and 4th largest for $p_1$ and $p_2$.

**Now widely used.** Java 7, Python unstable sort, ...
Which sorting algorithm to use?

Many sorting algorithms to choose from:

**Internal (in-memory) sorts.**
- Insertion sort, selection sort, bubblesort, shaker sort.
- Quicksort, mergesort, heapsort, samplesort, shellsort.
- Solitaire sort, red-black sort, splay sort, dual-pivot quicksort, timsort, ...

**External sorts.** Poly-phase mergesort, cascade-merge, psort, ....

**String/radix sorts.** Distribution, MSD, LSD, 3-way string quicksort.

**Parallel sorts.**
- Bitonic sort, Batcher even-odd sort.
- Smooth sort, cube sort, column sort.
- GPUsort.
Which sorting algorithm to use?

Applications have diverse attributes.
- Stable?
- Parallel?
- Deterministic?
- Duplicate keys?
- Multiple key types?
- Linked list or arrays?
- Large or small items?
- Randomly-ordered array?
- Guaranteed performance?

Elementary sort may be method of choice for some combination but cannot cover all combinations of attributes.

Q. Is the system sort good enough?
A. Usually.