2.3 Quicksort

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, ...

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

Sir Charles Antony Richard Hoare 1980 Turing Award
Quicksort partitioning

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

```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 >= j) break;
        exch(a, i, j);
    }
    exch(a, lo, j);
    return j;
}
```

public class Quick {
    private static int partition(Comparable[] a, int lo, int hi) {
        /* see previous slide */
    }
    ...
}

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 >= j) break;
        exch(a, i, j);
    }
    exch(a, lo, j);
    return j;
}
```

public class Quick {
    private static int partition(Comparable[] a, int lo, int hi) {
        /* see previous slide */
    }
    ...
}

Quicksort trace

```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 >= j) break;
        exch(a, i, j);
    }
    exch(a, lo, j);
    return j;
}
```

public class Quick {
    private static int partition(Comparable[] a, int lo, int hi) {
        /* see previous slide */
    }
    ...
}
Quick sort: 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 \((i = hi)\) 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 elements equal to the partitioning element.

Quick sort: empirical analysis

**Running time estimates:**
- Home PC executes \(10^9\) compares/second.
- Supercomputer executes \(10^{12}\) compares/second.

<table>
<thead>
<tr>
<th>Computer</th>
<th>Insertion sort (N²)</th>
<th>Mergesort (N log N)</th>
<th>Quick sort (N log N)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Home</td>
<td>thousand million billion</td>
<td>thousand million billion</td>
<td>thousand million billion</td>
</tr>
<tr>
<td>Super</td>
<td>instant 1 second 1 week</td>
<td>instant instant instant</td>
<td>instant instant instant</td>
</tr>
</tbody>
</table>

**Lesson 1.** Good algorithms are better than supercomputers.

**Lesson 2.** Great algorithms are better than good ones.

Quick sort: best-case analysis

**Best case.** Number of compares is \(-N \log N\).

<table>
<thead>
<tr>
<th>(A[1])</th>
<th>(n)</th>
<th>(j)</th>
<th>(i)</th>
<th>(k)</th>
<th>(m)</th>
<th>(n)</th>
<th>(l)</th>
<th>(o)</th>
<th>(p)</th>
<th>(q)</th>
<th>(r)</th>
<th>(s)</th>
<th>(t)</th>
<th>(u)</th>
<th>(v)</th>
<th>(w)</th>
<th>(x)</th>
<th>(y)</th>
<th>(z)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial values</td>
<td></td>
<td>H</td>
<td>A</td>
<td>C</td>
<td>B</td>
<td>F</td>
<td>E</td>
<td>G</td>
<td>D</td>
<td>L</td>
<td>I</td>
<td>K</td>
<td>J</td>
<td>N</td>
<td>M</td>
<td>O</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Random shuffle</td>
<td></td>
<td>H</td>
<td>A</td>
<td>C</td>
<td>B</td>
<td>F</td>
<td>E</td>
<td>G</td>
<td>D</td>
<td>L</td>
<td>I</td>
<td>K</td>
<td>J</td>
<td>N</td>
<td>M</td>
<td>O</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>14</td>
<td>D</td>
<td>A</td>
<td>C</td>
<td>B</td>
<td>F</td>
<td>E</td>
<td>G</td>
<td>H</td>
<td>L</td>
<td>I</td>
<td>K</td>
<td>J</td>
<td>N</td>
<td>M</td>
<td>O</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>6</td>
<td>B</td>
<td>A</td>
<td>C</td>
<td>D</td>
<td>F</td>
<td>E</td>
<td>G</td>
<td>H</td>
<td>L</td>
<td>I</td>
<td>K</td>
<td>J</td>
<td>N</td>
<td>M</td>
<td>O</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>F</td>
<td>E</td>
<td>G</td>
<td>H</td>
<td>L</td>
<td>I</td>
<td>K</td>
<td>J</td>
<td>N</td>
<td>M</td>
<td>O</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>D</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>F</td>
<td>E</td>
<td>G</td>
<td>H</td>
<td>L</td>
<td>I</td>
<td>K</td>
<td>J</td>
<td>N</td>
<td>M</td>
<td>O</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>F</td>
<td>E</td>
<td>G</td>
<td>H</td>
<td>L</td>
<td>I</td>
<td>K</td>
<td>J</td>
<td>N</td>
<td>M</td>
<td>O</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>L</td>
<td>I</td>
<td>K</td>
<td>J</td>
<td>N</td>
<td>M</td>
<td>O</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>L</td>
<td>I</td>
<td>K</td>
<td>J</td>
<td>N</td>
<td>M</td>
<td>O</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>L</td>
<td>I</td>
<td>K</td>
<td>J</td>
<td>N</td>
<td>M</td>
<td>O</td>
<td></td>
<td></td>
<td></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>K</td>
<td>L</td>
<td>N</td>
<td>M</td>
<td>O</td>
<td></td>
<td></td>
<td></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>J</td>
<td>K</td>
<td>L</td>
<td>N</td>
<td>M</td>
<td>O</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>N</td>
<td>M</td>
<td>O</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>N</td>
<td>M</td>
<td>O</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>N</td>
<td>O</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>12</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>N</td>
<td>O</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>14</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>N</td>
<td>O</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Quicksort: worst-case analysis

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

<table>
<thead>
<tr>
<th>N</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>
</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>
</tr>
<tr>
<td>0</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>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>1</td>
<td>1</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>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>2</td>
<td>2</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>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>3</td>
<td>3</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>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</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>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>7</td>
<td>8</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>8</td>
<td>9</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>9</td>
<td>10</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>10</td>
<td>11</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>11</td>
<td>12</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>12</td>
<td>12</td>
<td>12</td>
<td>13</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
</tr>
<tr>
<td>13</td>
<td>13</td>
<td>13</td>
<td>14</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
</tr>
<tr>
<td>14</td>
<td>14</td>
<td>14</td>
<td>15</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</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 $\sim \frac{1}{2} N \ln N$).

Pf 1. $C_N$ satisfies the recurrence $C_0 = C_1 = 0$ and for $N \geq 2$:

$$C_N = (N+1) + \frac{C_0}{N} + \frac{C_1}{N} + \ldots + \frac{C_{N-1}}{N}$$

- 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 $N(N+1)$:

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

- Repeatedly apply above equation:

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

- Approximate sum by an integral:

$$C_N \sim 2(N+1) \left( \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \ldots + \frac{1}{N+1} \right)$$

$$\sim 2(N+1) \int_3^{N+1} \frac{1}{x} \, dx$$

- Finally, the desired result:

$$C_N \sim 2(N+1) \ln N \approx 1.39 N \ln N$$

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 $\sim \frac{1}{2} N \ln N$).

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

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

<table>
<thead>
<tr>
<th>first partitioning element in left subarray</th>
<th>first partitioning element</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5 6 7 8 9 10 11 12 13</td>
<td>1 2 3 4 5 6 7 8 9 10 11 12 13</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 $\sim \frac{1}{2} N \ln N$).

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

\[
\sum_{i=1}^{N} \sum_{j=i+1}^{N} \frac{2}{j - i + 1} = 2 \sum_{i=1}^{N} \sum_{j=2}^{N-i+1} \frac{1}{j} \\
\leq 2N \sum_{j=1}^{N} \frac{1}{j} \\
\sim 2N \int_{1}^{N} \frac{1}{x} \, dx \\
= 2N \ln N
\]

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 $\sim \frac{1}{2} N \ln N$).

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

\[
\sum_{i=1}^{N} \sum_{j=i+1}^{N} \frac{2}{j - i + 1} = 2 \sum_{i=1}^{N} \sum_{j=2}^{N-i+1} \frac{1}{j} \\
\leq 2N \sum_{j=1}^{N} \frac{1}{j} \\
\sim 2N \int_{1}^{N} \frac{1}{x} \, dx \\
= 2N \ln N
\]

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: practical improvements

Insertion sort small subarrays.

• Even quicksort has too much overhead for tiny subarrays.
• Can delay insertion sort until end.

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

Insertion sort small subarrays.
- Even quicksort has too much overhead for tiny subarrays.
- Can delay insertion sort until end.

Median of sample.
- Best choice of pivot element = median.
- Estimate true median by taking median of sample.

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);
}

Optimize parameters.
- Median-of-3 (random) elements.
- Cutoff to insertion sort for \( \approx 10 \) elements.
**Selection**

**Goal.** Find the $k^{\text{th}}$ largest element.

**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 $O(N \log N)$ upper bound. How?
- Easy $O(N)$ upper bound for $k = 1, 2, 3$. How?
- Easy $\Omega(N)$ lower bound. Why?

Which is true?
- $\Omega(N \log N)$ lower bound? 
  
  Is selection as hard as sorting?
- $O(N)$ upper bound? 
  
  Is there a linear-time algorithm for all $k$?

**Quick-select: mathematical analysis**

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

**Pf sketch.**
- Intuitively, each partitioning step splits array approximately in half: $N + N/2 + N/4 + \ldots + 1 = 2N$ compares.
- Formal analysis similar to quicksort analysis yields:
  \[
  C_N = 2N + k \ln(N/k) + (N-k) \ln(N/(N-k))
  \]

**Ex.** $(2 + 2 \ln 2)N$ compares to find the median.

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

**Quick-select**

**Partition array so that:**
- Element $a[i]$ is in place.
- No larger element to the left of $i$.
- No smaller element to the right of $i$.

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

**Theoretical context for selection**

**Proposition.** [Blum, Floyd, Pratt, Rivest, Tarjan, 1973] There exists a compare-based selection algorithm whose worst-case running time is linear.

**Time bounds for selection**

by:
- Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ronald L. Rivest, and Robert E. Tarjan

**Notes:**
- 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 — quickselect. Specifically, at most \[5.5305 \ln n\] comparisons are ever required. This bound is improved for

**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.
Generic methods

In our `select()` implementation, client needs a cast.

```java
Double[] a = new Double[N];
for (int i = 0; i < N; i++)
    a[i] = StdRandom.uniform();
Double median = (Double) Quick.select(a, N/2);
```

The compiler complains.

```
% javac Quick.java
Note: Quick.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
% javac Quick.java
```

Q. How to fix?

Pedantic (safe) version. Compiles cleanly, no cast needed in client.

```java
public class QuickPedantic
{
    public static <Key extends Comparable<Key>> Key select(Key[] a, int k)
    {
        /* as before */
    }
    public static <Key extends Comparable<Key>> void sort(Key[] a)
    {
        /* as before */
    }
    private static <Key extends Comparable<Key>> int partition(Key[] a, int lo, int hi)
    {
        /* as before */
    }
    private static <Key extends Comparable<Key>> boolean less(Key v, Key w)
    {
        /* as before */
    }
    private static <Key extends Comparable<Key>> void exch(Key[] a, int i, int j)
    {
        Key swap = a[i]; a[i] = a[j]; a[j] = swap;
    }
}
```

Remark. Obnoxious code needed in system sort; not in this course (for brevity).

Duplicate keys

Often, purpose of sort is to bring records with duplicate keys together.
- Sort population by age.
- Find collinear points.
- 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. Always 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!
- 1990s C user found this defect in \texttt{qsort()}.

Several textbook and system implementation also have this defect.

Desirable. Put all keys equal to the partitioning element in place.

Dijkstra 3-way partitioning algorithm

3-way partitioning.
- Let \( v \) be partitioning element \( a[lo] \).
- Scan \( i \) from left to right.
  - \( a[i] \) less than \( v \): exchange \( a[lt] \) with \( a[i] \) and increment both \( lt \) and \( i \).
  - \( a[i] \) greater than \( v \): exchange \( a[gt] \) with \( a[i] \) and decrement \( gt \).
  - \( a[i] \) equal to \( v \): increment \( i \).

All the right properties.
- In-place.
- Not much code.
- Small overhead if no equal keys.

Dutch national flag problem. [Edsger Dijkstra]
- Conventional wisdom until mid 1990s: not worth doing.
- New approach discovered when fixing mistake in C library \texttt{qsort()}.
- Now incorporated into \texttt{qsort()} and Java system sort.
3-way partitioning: trace

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

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

**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

$$
\log \left( \frac{N!}{x_1! x_2! \cdots x_n!} \right) \sim - \sum_{i=1}^{n} x_i \log \frac{x_i}{N}
$$

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.
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.
- Find the closest pair.
- Binary search in a database.
- Identify statistical outliers.
- Find duplicates in a mailing list.

- Data compression.
- Computer graphics.
- Computational biology.
- Supply chain management.
- Load balancing on a parallel computer.

Every system needs (and has) a system sort!

Java system sorts

Java uses both mergesort and quicksort.

- `Arrays.sort()` sorts an array of `Comparable` or of any primitive type.
- Uses tuned quicksort for primitive types; tuned mergesort for objects.

Q. Why use different algorithms, depending on type?

War story (C `qsort` function)

AT&T Bell Labs (1991). Allan Wilks and Rick Becker discovered that a `qsort()` call that should have taken a few minutes was consuming hours of CPU time.

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.
Achilles heel in Bentley-McIlroy implementation (Java system sort)

Q. Based on all this research, Java’s system sort is solid, right?

A. No: a killer input.
- Overflows function call stack in Java and crashes program.
- Would take quadratic time if it didn’t crash first.

Now widely used. C, C++, Java,...
System sort: Which algorithm to use?

Applications have diverse attributes.

- Stable?
- Parallel?
- Deterministic?
- Keys all distinct?
- Multiple key types?
- Linked list or arrays?
- Large or small records?
- Is your array randomly ordered?
- Need guaranteed performance?

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

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

Sorting summary

<table>
<thead>
<tr>
<th>inplace?</th>
<th>stable?</th>
<th>worst</th>
<th>average</th>
<th>best</th>
<th>remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>selection</td>
<td>x</td>
<td>N³ / 2</td>
<td>N³ / 2</td>
<td>N² / 2</td>
<td>N exchanges</td>
</tr>
<tr>
<td>insertion</td>
<td>x x</td>
<td>N³ / 2</td>
<td>N⁴ / 4</td>
<td>N</td>
<td>use for small N or partially ordered</td>
</tr>
<tr>
<td>shell</td>
<td>x</td>
<td>?</td>
<td>?</td>
<td>N</td>
<td>tight code, subquadratic</td>
</tr>
<tr>
<td>merge</td>
<td>x</td>
<td>N lg N</td>
<td>N lg N</td>
<td>N lg N</td>
<td>N log N guarantee, stable</td>
</tr>
<tr>
<td>quick</td>
<td>x</td>
<td>N² / 2</td>
<td>2 N ln N</td>
<td>N lg N</td>
<td>N log N probabilistic guarantee fastest in practice</td>
</tr>
<tr>
<td>3-way quick</td>
<td>x</td>
<td>N² / 2</td>
<td>2 N ln N</td>
<td>N</td>
<td>improves quicksort in presence of duplicate keys</td>
</tr>
<tr>
<td>???</td>
<td>x x</td>
<td>N lg N</td>
<td>N lg N</td>
<td>N lg N</td>
<td>holy sorting grail</td>
</tr>
</tbody>
</table>

Which sorting algorithm?
Convex hull

The **convex hull** of a set of \( N \) points is the smallest convex set containing all the points.

Convex hull output. Sequence of extreme points in counterclockwise order.

Non-degeneracy assumption. No three points on a line.

Graham scan

- Choose point \( p \) with smallest \( y \)-coordinate (break ties by \( x \)-coordinate).
- Sort points by polar angle with respect to \( p \).
- Consider points in order, and discard unless they would create a \( ccw \) turn.

Graham scan: demo

http://www.cs.princeton.edu/courses/archive/fall08/cos226/demos/ah/GrahamScan.html

Convex hull: brute-force algorithm

Observation 1. Edges of convex hull of \( P \) connect pairs of points in \( P \).
Observation 2. Edge \( p \rightarrow q \) is on convex hull if all other points are \( ccw \) of \( pq \).

\( O(N^3) \) algorithm. For all pairs of points \( p \) and \( q \):
- Compute \( \text{Point.ccw}(p, q, x) \) for all other points \( x \).
- \( p \rightarrow q \) is on hull if all values are positive.

Degeneracies. Three (or more) points on a line.
**Graham scan: demo**

**Graham scan: implementation**

**Simplifying assumptions.** No three points on a line; at least 3 points.

```
Stack<Point> hull = new Stack<Point>();
Quick.sort(p, Point.BY_Y);  // p[0] is new point with lowest y-coordinate
Quick.sort(p, p[0].BY_POLAR_ANGLE);  // sort by polar angle with respect to p[0]

hull.push(p[0]);  // definitely on hull
hull.push(p[1]);

for (int i = 2; i < N; i++) {
    Point top = hull.pop();
    while (Point.ccw(top, hull.peek(), p[i]) <= 0)  // discard points that would create clockwise turn
        top = hull.pop();
    hull.push(top);
    hull.push(p[i]);  // add p[i] to putative hull
}
```

**Running time.** $N \log N$ for sorting and linear for rest.