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]
2.3 QUICKSORT

quicksort

Selection

Duplicate keys

System sorts

quicksort

Selection

Duplicate keys

System sorts

Quick sort 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.

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.

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].

The music of quicksort partitioning (by Brad Lyon)

https://drive.google.com/host/0B2GQktu-wcTicKraRjV1mRFN1U/index.html
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]))
            find item on left to swap
        if (i == hi) break;

        while (less(a[lo], a[--j]))
            find item on right to swap
        if (j == lo) break;

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

Quicksort trace

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

Quicksort trace (array contents after each partition)
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.

Quicksort: empirical analysis (1961)

Running time estimates:

- Algol 60 implementation.
- National Elliott 405 computer.

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

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

Table 1

<table>
<thead>
<tr>
<th>INSERTION SORT (n^2)</th>
<th>MERGESORT (n log n)</th>
<th>QUICKSORT (n log n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>computer</td>
<td>thousand</td>
<td>million</td>
</tr>
<tr>
<td>home</td>
<td>instant</td>
<td>2.8 hours</td>
</tr>
<tr>
<td>super</td>
<td>instant</td>
<td>1 second</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 \( \sim n \lg n \).

#### Table

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

### QuickSort: worst-case analysis

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

#### Table

<table>
<thead>
<tr>
<th>le</th>
<th>j</th>
<th>hi</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>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>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>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>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>
<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>
</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.** \( C_n \) 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 \( n(n + 1) \):
  \[
  \frac{C_N}{N + 1} = \frac{C_{N-1}}{N} + \frac{2}{N + 1}
  \]

- Repeatedly apply previous equation:
  \[
  \frac{C_N}{N + 1} = \frac{C_{N-1}}{N} + \frac{2}{N + 1} = \frac{C_{N-2}}{N - 1} + \frac{2}{N} + \frac{2}{N + 1} = \ldots
  \]

- Approximate sum by an integral:
  \[
  C_N \sim 2(N + 1) \int_3^{N + 1} \frac{1}{x} \, dx
  \]

- Finally, the desired result:
  \[
  C_N \sim 2(N + 1) \ln N \approx 1.39N \ln N
  \]
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 ]

Quicksort: practical improvements

Insertion sort small subarrays.
- Even quicksort has too much overhead for tiny subarrays.
- Cutoff to insertion sort for $\sim 10$ items.

```
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.

```
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?
- \( n \) upper bound?

Quick-select: mathematical analysis

Proposition. Quick-select takes linear time on average.

Pf sketch.
- Intuitively, each partitioning step splits array approximately in half:
  \( n + n/2 + n/4 + \ldots + 1 \approx 2n \) 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).

**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":

```
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
```

```
Chicago 0:25:52
Chicago 0:03:13
Chicago 0:21:05
Chicago 0:19:46
Chicago 0:19:32
Chicago 0:00:00
Chicago 0:35:21
Chicago 0:00:59
Houston 0:01:10
Houston 0:00:13
Phoenix 0:37:44
Phoenix 0:00:03
Phoenix 0:14:25
Seattle 0:10:25
Seattle 0:36:14
Seattle 0:22:43
Seattle 0:10:11
Seattle 0:22:54
```
War story (system sort in C)

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

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.

---

**QuickSort quiz 2**

What is the result of partitioning the following array (skip over equal keys)?

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
A A A A A A A A A A A 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
```

**C.**

```
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)?

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
A A A A A A A A A A A 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
```

**C.**

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

\[
a[i] = \begin{bmatrix}
\end{bmatrix}
\]

**Duplicate keys: partitioning strategies**

**Bad.** Don't stop scans on equal keys.
[\( \sim \frac{1}{2} n^2 \) compares when all keys equal]

\[
\begin{bmatrix}
\end{bmatrix}
\]

**Good.** Stop scans on equal keys.
[\( \sim n \lg n \) compares when all keys equal]

\[
\begin{bmatrix}
\end{bmatrix}
\]

**Better.** Put all equal keys in place. How?
[\( \sim n \) compares when all keys equal]

\[
\begin{bmatrix}
\end{bmatrix}
\]

**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**

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

**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.

**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 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 $i$

```java
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: Java implementation

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, then any compare-based sorting algorithm must use at least

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

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 applications

Sorting algorithms are essential in a broad variety of applications:

- Sort a list of names.
- Organize an MP3 library.
- Display Google PageRank results.
- List RSS feed in reverse chronological order.

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

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

...
Engineering a system sort (in 1993)

**Bentley-McIlroy quicksort.**

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

**Similar to Dijkstra 3-way partitioning (but fewer exchanges when not many equal keys)**

Very widely used. C, C++, Java 6, ...

A beautiful mailing list post (Yaroslavskiy, September 2009)

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

Hello A11,

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 ]
```

...
Dual-pivot partitioning demo

**Initializaton.**
- Choose $a[lo]$ and $a[hi]$ as partitioning items.
- Exchange if necessary to ensure $a[lo] \leq a[hi]$.

**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>$? \leq p_2 &gt; p_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\uparrow$ $lo$</td>
<td>$\uparrow$ $lt$</td>
<td>$\uparrow$ $i$</td>
<td>$\uparrow$ $gt$ $\uparrow$ $hi$</td>
</tr>
</tbody>
</table>

**Finalize.**
- Exchange $a[hi]$ with $a[+gt]$.

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

**Dual-pivot quicksort**

**Now widely used.** Java 7, Python unstable sort, Android, ...
Three-pivot quicksort

Use three partitioning items $p_1$, $p_2$, and $p_3$ and partition into four subarrays:

- Keys less than $p_1$.
- Keys between $p_1$ and $p_2$.
- Keys between $p_2$ and $p_3$.
- Keys greater than $p_3$.

$$< p_1 \quad p_1 \; \geq \; p_2 \quad \geq \; p_2 \; \leq \; p_3 \quad p_3 \; > \; p_3$$

Quicksort quiz 4

Why do 2-pivot (and 3-pivot) quicksort perform better than 1-pivot?

A. Fewer compares.

B. Fewer exchanges.

C. Fewer cache misses.

D. I don’t know.

System sort in Java 7

Arrays.sort().

- 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.

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

Bottom line. Use the system sort!

Bottom line. Caching can have a significant impact on performance.