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
private static int partition(int[] a, int lo, int hi) {
    int i = lo - 1;
    for (int j = lo; j < hi; j++) {
        if (a[j] <= a[hi]) {
            i++;
        } else {
            swap(a, i, j);
            i--;
        }
    }
    swap(a, i + 1, hi);
    return i + 1;
}
```

```java
public 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;
}
```

```java
public static void main(String[] args) {
    // QuickSort code
}
```

```
private static void sort(Comparable[] a, int lo, int hi) {
    sort(a, lo, hi - 1);
    sort(a, lo + 1, hi);
}
```

```java
private static void swap(Comparable[] a, int i, int j) {
    Comparable temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}
```

CS @ Princeton
A brief history

Tony Hoare

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

Bob Sedgewick

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

```
input  QUICKSORTEXAMPLE
shuffle KRATELEPUIMQCSOS
partition ECAIEKLMPUTMQRXOS
sort left ACEEIKLPUTMQRXOS
sort right ACEEIKLMOQPQRSUX
result ACEEIKLMOQPQRSUX
```
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 \( a[j] \).

\[\begin{array}{cccccccccccccccc}
K & R & A & T & E & L & E & P & U & I & M & Q & C & X & O & S \\
\uparrow & \uparrow & & & & & & & & & & & & & & \\
lo & i & & & & & & & & & & & & & & \\
\end{array}\]

stop i scan because \( a[i] \geq a[lo] \)
In the worst case, how many compares and exchanges to partition an array of length $n$?

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

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

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

D. $\sim n$ and $\sim n$
Quicksort partitioning: Java implementation

```java
private static int partition(Comparable[] a, int lo, int hi) {
    int i = lo, j = hi+1;
    while (true)
    {
        while (less(a[++i], a[lo]))
            if (i == hi) break;

        while (less(a[lo], a[--j]))
            if (j == lo) break;

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

https://algs4.cs.princeton.edu/23quick/Quick.java.html
public class Quick
{
    private static int partition(Comparable[] a, int lo, int hi)
    { /* see previous slide */ }

    public static void sort(Comparable[] a)
    {
        StdRandom.shuffle(a);
        sort(a, 0, a.length - 1);
    }

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

https://algs4.cs.princeton.edu/23quick/Quick.java.html
### Quicksort trace

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

initial values
random shuffle

Quicksort trace (array contents after each partition)

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>U</td>
<td>I</td>
<td>C</td>
<td>K</td>
<td>S</td>
<td>O</td>
<td>R</td>
<td>T</td>
<td>E</td>
<td>X</td>
<td>A</td>
<td>M</td>
<td>P</td>
<td>L</td>
<td>E</td>
</tr>
<tr>
<td>K</td>
<td>R</td>
<td>A</td>
<td>T</td>
<td>E</td>
<td>L</td>
<td>E</td>
<td>P</td>
<td>U</td>
<td>I</td>
<td>M</td>
<td>Q</td>
<td>C</td>
<td>X</td>
<td>O</td>
<td>S</td>
</tr>
<tr>
<td>E</td>
<td>C</td>
<td>A</td>
<td>I</td>
<td>E</td>
<td>K</td>
<td>L</td>
<td>P</td>
<td>U</td>
<td>T</td>
<td>M</td>
<td>Q</td>
<td>R</td>
<td>X</td>
<td>O</td>
<td>S</td>
</tr>
<tr>
<td>E</td>
<td>C</td>
<td>A</td>
<td>E</td>
<td>I</td>
<td>K</td>
<td>L</td>
<td>P</td>
<td>U</td>
<td>T</td>
<td>M</td>
<td>Q</td>
<td>R</td>
<td>X</td>
<td>O</td>
<td>S</td>
</tr>
<tr>
<td>E</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>E</td>
<td>C</td>
<td>E</td>
<td>E</td>
<td>I</td>
<td>K</td>
<td>L</td>
<td>P</td>
<td>U</td>
<td>T</td>
<td>M</td>
<td>Q</td>
<td>R</td>
<td>X</td>
<td>O</td>
<td>S</td>
</tr>
<tr>
<td>A</td>
<td>C</td>
<td>E</td>
<td>E</td>
<td>I</td>
<td>K</td>
<td>L</td>
<td>P</td>
<td>U</td>
<td>T</td>
<td>M</td>
<td>Q</td>
<td>R</td>
<td>X</td>
<td>O</td>
<td>S</td>
</tr>
<tr>
<td>A</td>
<td>C</td>
<td>E</td>
<td>E</td>
<td>I</td>
<td>K</td>
<td>L</td>
<td>P</td>
<td>U</td>
<td>T</td>
<td>M</td>
<td>Q</td>
<td>R</td>
<td>X</td>
<td>O</td>
<td>S</td>
</tr>
<tr>
<td>A</td>
<td>C</td>
<td>E</td>
<td>E</td>
<td>I</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>O</td>
<td>P</td>
<td>T</td>
<td>Q</td>
<td>R</td>
<td>X</td>
<td>U</td>
<td>S</td>
</tr>
<tr>
<td>A</td>
<td>C</td>
<td>E</td>
<td>E</td>
<td>I</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>O</td>
<td>P</td>
<td>T</td>
<td>Q</td>
<td>R</td>
<td>X</td>
<td>U</td>
<td>S</td>
</tr>
<tr>
<td>A</td>
<td>C</td>
<td>E</td>
<td>E</td>
<td>I</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>O</td>
<td>P</td>
<td>T</td>
<td>Q</td>
<td>R</td>
<td>X</td>
<td>U</td>
<td>S</td>
</tr>
<tr>
<td>A</td>
<td>C</td>
<td>E</td>
<td>E</td>
<td>I</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>O</td>
<td>P</td>
<td>Q</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>U</td>
<td>X</td>
</tr>
<tr>
<td>A</td>
<td>C</td>
<td>E</td>
<td>E</td>
<td>I</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>O</td>
<td>P</td>
<td>Q</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>U</td>
<td>X</td>
</tr>
<tr>
<td>A</td>
<td>C</td>
<td>E</td>
<td>E</td>
<td>I</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>O</td>
<td>P</td>
<td>Q</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>U</td>
<td>X</td>
</tr>
</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
Another quicksort animation

https://en.wikipedia.org/wiki/Quicksort
Quicksort: implementation details

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

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

Equal keys. When duplicate keys 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

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: worst-case analysis

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

<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>
</tr>
</thead>
<tbody>
<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>
</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>
</tr>
</tbody>
</table>

After random shuffle
Quicksort: worst-case analysis

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

Good news. Worst case analysis of quicksort is irrelevant for practical purposes.

Worst case exponentially unlikely to occur (unless bug in shuffling method.)
More likely that lightning strikes computer during execution.
Quicksort: average-case analysis

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

**Intuition.** Each partitioning step splits array approximately in half.

**Recall:** Any algorithm with the following structure takes $\Theta(n \log n)$ time.

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

For quicksort, the two problems aren’t exactly half the size, but close enough.
Quicksort: average-case analysis

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

**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:

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

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

- Finally, the desired result:

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

Quicksort is a 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 ]
Three different types of average-case complexity

<table>
<thead>
<tr>
<th></th>
<th>Cost is averaged over...</th>
<th>Example</th>
<th>Impact of worst case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amortized</td>
<td>Sequence of operations</td>
<td>Stacks and queues using resizing arrays</td>
<td>Some operations take (far) longer than amortized running time</td>
</tr>
<tr>
<td>Expected</td>
<td>Internal randomness of implementation</td>
<td>Quicksort</td>
<td>Irrelevant</td>
</tr>
<tr>
<td>Average case</td>
<td>Possible inputs</td>
<td>Quicksort without shuffling</td>
<td>Worst case may occur if our model of “average” input is wrong</td>
</tr>
</tbody>
</table>

Frequent source of performance bugs in practice

In this course, for simplicity, we’ll ignore the distinction between average case and expected complexity.
Quicksort properties

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

**Pf.**
- Partitioning: constant extra space.
- Function-call stack: 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>
Quicksort: practical improvement

Insertion sort small subarrays.

- Even quicksort has too much overhead for tiny subarrays.
- Cutoff to insertion sort for $\approx 10$ items.
2.3 **Quicksort**

- quicksort
- selection
- duplicate keys
- system sorts
Selection

**Goal.** Given an array of $n$ items, find item of rank $k$.

**Ex.** Min ($k = 0$), max ($k = n - 1$), median ($k = n/2$).

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

**Which is true?**
- $n \log n$ lower bound?
- $n$ upper bound?

---

is selection as hard as sorting?

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 $k$.

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

**Intuition:**
Each partitioning step splits array approximately in half:

$$n + n/2 + n/4 + \ldots + 1 \sim 2n$$ compares.
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.
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)?

Quicksort: quiz 2

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 A

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

D. I don't know.
What is the result of partitioning the following array (stop on equal keys)?

A.  

B.  

C.  

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>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.

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

B A A B A B B B C C C \hspace{2cm} A A A A A A A A A A A A

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

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

B A A B A B C C B C B \hspace{2cm} A A A A A A A A A A A A

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

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

A A A B B B B B C C C \hspace{2cm} A A A A A A A A A A A A
3-way partitioning

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

- Entries in the left part are less than the partitioning item.
- Entries in the left part are equal to the partitioning item.
- Entries in the left part are greater than the partitioning item.

![Diagram of 3-way partitioning](image)

**3-way partitioning algorithm.** [Edsger Dijkstra]

- 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 $i$

```
\begin{array}{cccccccccccc}
\text{lt} & \text{i} & \text{gt} \\
\downarrow & \downarrow & \downarrow \\
D & B & X & W & P_2 & P_3 & V & P_4 & A & P_5 & C & Y & Z \\
\uparrow & \uparrow & \uparrow \\
\text{lo} & \text{hi} \\
\end{array}
```

**Invariant**

```
\begin{array}{cccc}
< v & = v & \text{black} & > v \\
\uparrow & \uparrow & \uparrow \\
\text{lt} & \text{i} & \text{gt} \\
\end{array}
```
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 $i$

---

```
D  B  C  A  P_5  P_2  P_3  P_1  P_4  V  W  Y  Z  X
  \_\_\_\_
lo
```

---

```
\_\_\_\_
\_\_\_\_
\_\_\_\_
\_\_\_\_
\_\_\_\_
\_\_\_\_
\_\_\_\_
\_\_\_\_
\_\_\_\_
\_\_\_\_
\_\_\_\_
\_\_\_\_
\_\_\_\_
\_\_\_\_
\_\_\_\_
\_\_\_\_
\_\_\_\_
\_\_\_\_
\_\_\_\_
\_\_\_\_
```

---

```
< V  = V  \_\_\_\_\_\_  > V
  \_\_\_\_
lt  i  gt
```

---

**Invariant**
private static void sort(Comparable[] a, int lo, int hi) {
    if (hi <= lo) return;
    int lt = lo, gt = hi;
    Comparable v = a[lo];
    int i = lo + 1;
    while (i <= gt)
    {
        int cmp = a[i].compareTo(v);
        if      (cmp < 0) exch(a, lt++, i++);
        else if (cmp > 0) exch(a, i, gt--);
        else              i++;
    }
    sort(a, lo, lt - 1);
    sort(a, gt + 1, hi);
}
3-way quicksort: visual trace

equal to partitioning element
## Sorting summary

<table>
<thead>
<tr>
<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>$\frac{1}{2} n^2$</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>$n$</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>merge</td>
<td>✔️</td>
<td>$\frac{1}{2} n \lg n$</td>
<td>$n \lg n$</td>
<td>$n \lg n$</td>
<td>$n \log n$ guarantee; stable</td>
</tr>
<tr>
<td>quick</td>
<td>✔️</td>
<td>$n \lg n$</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>$n$</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>$n$</td>
<td>$n \lg n$</td>
<td>$n \lg n$</td>
<td>holy sorting grail</td>
</tr>
</tbody>
</table>
2.3 Quicksort

- quicksort
- selection
- duplicate keys
- system sorts
System sort in Java 8

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?
Q. Why so many overloaded methods?

Bottom line. Use the system sort!