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]
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]) && (i > left) j--;
        if (i <= j) {
            y = items[j];
            items[j] = items[i];
            items[i] = y;
            i++; j--;
        }
    } while (i <= j);
    if (left < i) quicksort(items, left, j);
    if (i < right) quicksort(items, i, right);
}
Quicksort t-shirt

```java
public class Quick {
    public static void sort(Comparable[] a) { 
        if (a.length <= 0) return; 
        boolean isSorted = false; 
        int hi = a.length - 1; 
        while (!isSorted) { 
            int lo = 0; 
            while (lo < hi) { 
                if (less(a[lo], a[hi])) { 
                    swap(a, lo, hi); 
                    hi--; 
                } else { 
                    lo++; 
                }
            } 
            isSorted = true; 
        }
    }

    public static int partition(Comparable[] a) { 
        int lo = 0; 
        int hi = a.length - 1; 
        while (lo < hi) { 
            if (less(a[lo], a[hi])) { 
                swap(a, lo, hi); 
                hi--; 
            } else { 
                lo++; 
            }
        }
        return lo; 
    }

    public static boolean less(Comparable v, Comparable w) { 
        return v.compareTo(w) < 0; 
    }

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

    private static void swap(Object[] a, int i, int j) { 
        Object t = a[i]; 
        a[i] = a[j]; 
        a[j] = t; 
    }

    private static boolean isSorted(Comparable[] a, int lo, int hi) { 
        for (int i = lo; i < hi; i++) { 
            if (!less(a[i], a[i + 1])) return false; 
        }
        return true; 
    }

    public static void main(String[] args) { 
        String[] a = Stein.remainderString(); 
        int i = 0; 
        while (i < a.length) { 
            System.out.println(a[i]); 
            i++; 
        }
    }
}
```

CS @ Princeton
2.3 QUICKSORT

- quicksort
- selection
- duplicate keys
- system sorts
**Quicksort overview**

**Step 1.** Shuffle the array.

**Step 2.** Partition the array so that, for some \( j \):
- Entry \( a[j] \) is in its eventual sorted position.
- No larger entry to the left of \( j \).
- No smaller entry to the right of \( j \).

**Step 3.** Sort each subarray recursively.
Quicksort overview

input

QUICKSORT EXAMPLE
Quicksort overview

Step 1. Shuffle the array.

shuffle

QuickSortExample
QuickSort overview

Step 1. Shuffle the array.

shuffled
**Quicksort overview**

**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 $j$.

---

**partition**

```
K     R     A     T     E     L     E     P     U     I     M     Q     C     X     O     S
```
Quick sort overview

**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 $j$.

```
partition
```

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

\[<= K\] \[>= K\]
Quicksort overview

**Step 3.** Sort each subarray recursively.

sort the left subarray

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

Step 3. Sort each subarray recursively.

sort the left subarray

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

sorted
Quicksort overview

**Step 3.** Sort each subarray recursively.

sort the right subarray

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

sorted
Quicksort overview

**Step 3.** Sort each subarray recursively.

**sort the right subarray**

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

sorted

sorted
Quicksort overview

sorted array

A C E E I K L M O P Q R S T U X
• 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
1980 Turing Award

**Tony Hoare**

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

---

**ALGORITHM 64**

**QUICKSORT**

C. A. R. Hoare

Elliott Brothers Ltd., Borehamwood, Hertfordshire, Eng.

**procedure** quicksort (A,M,N); **value** M,N;

**array** A; **integer** M,N;

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

**begin**

**integer** I,J;

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

quicksort (A,M,I);

quicksort (A, I, N)

end

end quicksort

---

Communications of the ACM (July 1961)
Tony Hoare

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

“*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.*”
Refined and popularized quicksort.
Analysed many versions of quicksort.
Quicksort partitioning: first try

1. Pick $a[0]$ as the partitioning element
2. Create an auxiliary array $aux$
3. Scan the array and copy each item less than $a[0]$ to $aux$
4. Scan the array and copy each item not less than $a[0]$ to $aux$
5. Copy $aux$ back to $a$

Problems

- Requires space for auxiliary array
- Requires multiple scans of the array
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].

stop i scan because a[i] >= a[lo]
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 \( a[j] \).
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 \(a[j]\).

stop j scan and 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 \( a[j] \).
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 \( a[j] \).

stop i scan because \( a[i] \geq a[lo] \)
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].
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 $a[j]$.

Stop $j$ scan and 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 \(a[j]\).
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 \(a[j]\).

\[\]

\[
\]

stop \(i\) scan because \(a[i] \geq a[lo]\)
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 \(a[i] < a[\text{lo}]\).
- Scan j from right to left so long as \(a[j] > a[\text{lo}]\).
- 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 \( a[j] \).

stop j scan and 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 $a[j]$.
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 \( a[j] \).

stop i scan because \( a[i] \geq a[lo] \)
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 \(a[j]\).

---

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

\[ \uparrow \]

\[ \text{lo} \]

\[ \uparrow \]

\[ \text{j} \]

\[ \uparrow \]

\[ \text{i} \]

stop j scan because \(a[j] \leq a[lo]\)
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 \( a[j] \).

<table>
<thead>
<tr>
<th>K</th>
<th>C</th>
<th>A</th>
<th>I</th>
<th>E</th>
<th>E</th>
<th>L</th>
<th>P</th>
<th>U</th>
<th>T</th>
<th>M</th>
<th>Q</th>
<th>R</th>
<th>X</th>
<th>O</th>
<th>S</th>
</tr>
</thead>
</table>

pointers cross: exchange \( a[lo] \) with \( a[j] \)
Quicksort partitioning demo

Repeat until i and j pointers cross.
- Scan i from left to right so long as \(a[i] < a[lo]\).
- Scan j from right to left so long as \(a[j] > a[lo]\).
- Exchange \(a[i]\) with \(a[j]\).

When pointers cross.
- Exchange \(a[lo]\) with \(a[j]\).

partitioned!
Quicksort partitioning demo

Repeat until i and j pointers cross.
- Scan i from left to right so long as \( a[i] < a[lo] \).
- Scan j from right to left so long as \( a[j] > a[lo] \).
- Exchange \( a[i] \) with \( a[j] \).

When pointers cross.
- Exchange \( a[lo] \) with \( a[j] \).

partitioned!
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;  // find item on left to swap
        while (less(a[lo], a[--j]))
            if (j == lo) break;  // find item on right to swap
        if (i >= j) break;  // check if pointers cross
        exch(a, i, j);  // swap
    }
    exch(a, lo, j);  // swap with partitioning item
    return j;  // return index of item now known to be in place
}
```

**Before**

\[
\begin{array}{c}
\text{i} \\
\downarrow \\
\text{lo} \\
\text{hi}
\end{array}
\]

**During**

\[
\begin{array}{c|c|c|c}
\text{v} & \leq \text{v} & \geq \text{v} \\
\downarrow & \downarrow & \downarrow \\
\text{lo} & \text{i} & \text{j} \\
\text{hi}
\end{array}
\]

**After**

\[
\begin{array}{c|c|c|c}
\leq \text{v} & \text{v} & \geq \text{v} \\
\downarrow & \downarrow & \downarrow \\
\text{lo} & \text{j} & \text{hi}
\end{array}
\]
Quicksort quiz 1

Are the array bounds checks in the previous slide necessary?

A. Yes
B. No
C. Both of the above
D. Neither of the above
E. I don't know.

Trick question! One of them is necessary and the other isn’t.
Quicksort quiz 2

How many compares to partition an array of length $N$?

A. $\sim \frac{1}{4} N$
B. $\sim \frac{1}{2} N$
C. $\sim N$
D. $\sim N \lg N$
E. I don't know.
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

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

Initial values
random shuffle

Quicksort trace (array contents after each partition)

Initial values:
- QuickSort Example
- Key: RatelePuiMqCxs

Random shuffle:
- 0 5 15 ECAIEKLPUMQRXOS
- 0 3 4 ECAEIKLPUTMQRXOS
- 0 2 2 ACEEIKLPUTMQRXOS
- 0 0 1 ACEEIKLPUTMQRXOS
- 1 1 ACEEIKLPUTMQRXOS
- 4 4 ACEEIKLPUTMQRXOS
- 6 6 15 ACEEIKLPUMQRXOS
- 7 9 15 ACEEIKLMOPTQRXUS
- 7 7 8 ACEEIKLMOPTQRXUS
- 8 8 ACEEIKLMOPTQRXUS
- 10 13 15 ACEEIKLMOPSRQTUX
- 10 12 12 ACEEIKLMOPRQSCTX
- 10 11 11 ACEEIKLMOPRQSCTX
- 10 10 ACEEIKLMOPQRSTUX
- 14 14 15 ACEEIKLMOPQRSTUX
- 15 15 ACEEIKLMOPQRSTUX

Result:
- ACEEIKLMOPQRSTUX

Quicksort trace (array contents after each partition)

No partition for subarrays of size 1.
Quicksort animation

50 random items

http://www.sorting-algorithms.com/quick-sort
Quicksort: implementation details

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

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

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

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.

sorting N 6–word items with 1–word keys

Elliott 405 magnetic disc (16K words)
Quicksort: best-case analysis

Best case. Number of compares is $\sim N \log N$. 

<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>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td></td>
</tr>
</tbody>
</table>

initial values:
H A C B F E G D L I K J N M O

random shuffle:
H A C B F E G D L I K J N M O
0 7 14 D A C B F E G H L I K J N M O
0 3 6 B A C D F E G H L I K J N M O
0 1 2 A B C D F E G H L I K J N M O
0 0 A B C D F E G H L I K J N M O
2 2 A B C D F E G H L I K J N M O
4 5 6 A B C D E F G H L I K J N M O
4 4 A B C D E F G H L I K J N M O
6 6 A B C D E F G H L I K J N M O
8 11 14 A B C D E F G H J I K L N M O
8 9 10 A B C D E F G H I J K L N M O
8 8 A B C D E F G H I J K L N M O
10 10 A B C D E F G H I J K L N M O
12 13 14 A B C D E F G H I J K L M N O
12 12 A B C D E F G H I J K L M N O
14 14 A B C D E F G H I J K L M N O
A B C D E F G H I J K L M N O
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></td>
<td></td>
<td></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>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  0  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>1  1  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>2  2  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>3  3  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>4  4  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>5  5  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>6  6  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>7  7  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>8  8  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>9  9  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>10 10 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>11 11 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 12 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>13 13 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>14 14 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>
</tbody>
</table>

Due to bad randomness, not bad input
QuickSort: analysis of expected running time

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

**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: analysis of expected running time

- 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:

\[
C_N \sim 2(N+1) \ln N \approx 1.39N \lg N
\]
Quicksort: worst case is exponentially unlikely

Probability (\# compares $> 0.1 N^2$) $< 1/2^N$ for large $N$.

Things more likely than quicksort being quadratic on a million-item array:
- Lightning bolt strikes computer during execution.
- Get trampled by a herd of zebra above the Arctic Circle, while being hit by a meteor.
- I become the next president of these United States.

The probability of needing even $2N \lg N$ compares (instead of \( \sim 1.39 N \lg N \)) is negligible for large $N$.

**Bottom line.** Assuming good randomness and no implementation bugs, this is as good as a worst-case $\sim 1.39 N \lg N$ guarantee.
Quicksort: summary of performance characteristics

Quicksort is a randomized algorithm.

- Guaranteed to be correct.
- Running time depends on random shuffle.

Expected running time.

- Expected number of compares is $\sim 1.39 \, N \lg N$.
- Independent of the input.

Comparison to mergesort.

- 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 ]
How much extra space does quicksort use?

A. $\Theta(1)$
B. $\Theta(\ln N)$
C. $\Theta(N)$
D. $\Theta(N \ln N)$
E. I don't know.
**Quicksort properties**

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

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

---

can guarantee logarithmic depth by recurring on smaller subarray before larger subarray (but requires using an explicit stack)

---

**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 improvements

Insertion sort small subarrays.

- Like mergesort, quicksort has too much overhead for tiny subarrays.
- Cutoff to insertion sort for $\approx 10$ items.

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

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

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

    int median = medianOf3(a, lo, lo + (hi - lo)/2, hi);
    swap(a, lo, median);

    int j = partition(a, lo, hi);
    sort(a, lo, j-1);
    sort(a, j+1, hi);
}
```
2.3 QUICKSORT

- quicksort
- selection
- duplicate keys
- system sorts
Selection

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

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

**Applications.**
- Order statistics.
- Find the "top $k$.”

**Use theory as a guide.**
- Easy $N \log N$ upper bound. How?
- Easy $N$ upper bound for $k = 1, 2, 3$. How?
- Easy $N$ lower bound. Why?

**Which is true?**
- $N \log N$ lower bound? **is selection as hard as sorting?**
- $N$ upper bound? **is there a linear-time algorithm?**
Quick-select

Partition array so that:

- Entry $a[j]$ is in place.
- No larger entry to the left of $j$.
- No smaller entry to the right of $j$.

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

```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 expected linear time.

Pf.
Omitted, similar to the analysis of expected running time of quicksort.

There exists a deterministic algorithm with linear running time, but we don’t use it because the constants are bad.
2.3 Quicksort

- quicksort
- selection
- duplicate keys
- system sorts
Duplicate keys

Often, purpose of sort is to bring items with equal keys together.

- Sort population by age.
- Remove duplicates from mailing list.
- Sort job applicants by college attended.

Typical characteristics of such applications.

- Huge array.
- Small number of key values.
War story (system sort in C)

**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.

Q. Why not continue scans on equal keys?
QuickSort quiz 4

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

<table>
<thead>
<tr>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<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>
</tbody>
</table>

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

C. 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></th>
<th></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>i</td>
<td>j</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<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></td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<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></td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
</tbody>
</table>
Duplicate keys: partitioning strategies

**Bad.** Don't stop scans on equal keys.

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

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

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

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

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

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

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

\[
\begin{array}{cccccccccccc}
\end{array}
\]
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 \( \gt \).
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 3-way partitioning demo

- Let \( v \) be partitioning item \( a[10] \).
- Scan \( i \) from left to right.
  - \((a[i] < v)\): exchange \( a[lt] \) with \( a[i] \); increment both \( lt \) and \( i \)
  - \((a[i] > v)\): exchange \( a[gt] \) with \( a[i] \); decrement \( gt \)
  - \((a[i] == v)\): increment \( i \)

\[
\begin{array}{cccccccccccccccc}
\text{lt} & \text{i} & \downarrow & \downarrow & \text{gt} \\
\end{array}
\]

\[\text{equal} \quad \text{unknown}\]
Dijkstra 3-way partitioning demo

- Let \( v \) be partitioning item \( a[10] \).
- 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 \( a[10] \).
- 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 3-way partitioning demo

- Let v be partitioning item a[10].
- 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 $a[i_0]$.
- Scan $i$ from left to right.
  - $(a[i] < v)$: exchange $a[\text{lt}]$ with $a[i]$; increment both $\text{lt}$ and $i$
  - $(a[i] > v)$: exchange $a[\text{gt}]$ with $a[i]$; decrement $\text{gt}$
  - $(a[i] == v)$: increment $i$
Dijkstra 3-way partitioning demo

- Let \( v \) be partitioning item \( a[10] \).
- 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 3-way partitioning demo

- Let \( v \) be partitioning item \( a[10] \).
- Scan \( i \) from left to right.
  - \((a[i] \ < \ v)\): exchange \( a[\text{lt}] \) with \( a[i] \); increment both \( \text{lt} \) and \( i \)
  - \((a[i] \ > \ v)\): exchange \( a[\text{gt}] \) with \( a[i] \); decrement \( \text{gt} \)
  - \((a[i] \ == \ v)\): increment \( i \)

```
A B C P P P P V P D W Y Z X
less equal unknown greater
```
Dijkstra 3-way partitioning demo

- Let $v$ be partitioning item $a[10]$.
- Scan $i$ from left to right.
  - $(a[i] < v)$: exchange $a[\text{lt}]$ with $a[i]$; increment both $\text{lt}$ and $i$
  - $(a[i] > v)$: exchange $a[\text{gt}]$ with $a[i]$; decrement $\text{gt}$
  - $(a[i] == v)$: 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 3-way partitioning demo

- Let $v$ be partitioning item $a[10]$.
- 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 $a[10]$.
- 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 \( a[10] \).
- 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 \( i \)

---

![Partitioning Diagram]

- \( lt \) less
- \( i \) equal
- \( gt \) greater
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 $i$

![Diagram showing partitioning process with labels for lo, hi, lt, i, gt, and partitioning invariant]
3-way quicksort: visual trace

equal to partitioning element
2.3 QUICKSORT

- quicksort
- selection
- duplicate keys
- system sorts
Sorting applications

Sorting algorithms are essential in a broad variety of applications:

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

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

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

...
Dual-pivot quicksort

Use two partitioning items $p_1$ and $p_2$ and partition into three subarrays:

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

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

degenerates to Dijkstra's 3-way partitioning

Recursively sort three subarrays.

**Note.** Skip middle subarray if $p_1 = p_2$. 
Dual-pivot quicksort

Use two partitioning items $p_1$ and $p_2$ and partition into three subarrays:

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

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

Now widely used. Java 7, Python unstable sort, Android, ...
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!
Review: three types of averages in measuring efficiency of algorithms

Average-case. Average over all possible inputs.

Expected.* Average over all possible values of RNG.

Worst-case over all possible inputs.

Amortized. Average over a sequence of inputs.

(Must be stateful, such as a data structure.)

Example 1. The _______ running time of quicksort is $O(N \lg N)$. But if we omitted the shuffling step, only the _______ running time would be $O(N \lg N)$.

Example 2. The _______ running time of selection is $O(N)$ with quick-select, but if we only care about the _______ running time, we’d first sort the array.

*Some people use average-case to refer to both.

If you do, it’s important to always know which one you’re talking about.
Quicksort quiz 6

The ______ running time of quicksort is $O(N \lg N)$. But if we omitted the shuffling step, only the ______ running time would be $O(N \lg N)$.

A. Average-case, expected
B. Expected, average-case
C. Amortized, expected
D. Expected, amortized
E. I don't know.
The ______ running time of selection is $O(N)$ with quick-select, but if we only care about the ______ running time, we’d first sort the array.

A. Average-case, amortized
B. Amortized, average-case
C. Amortized, expected
D. Expected, amortized
E. I don't know.
## Sorting summary

<table>
<thead>
<tr>
<th></th>
<th>inplace?</th>
<th>stable?</th>
<th>best</th>
<th>average</th>
<th>worst</th>
<th>remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>selection</td>
<td>✔️</td>
<td></td>
<td>(\frac{1}{2} N^2)</td>
<td>(\frac{1}{2} N^2)</td>
<td>(\frac{1}{2} N^2)</td>
<td>(N) exchanges</td>
</tr>
<tr>
<td>insertion</td>
<td>✔️</td>
<td>✔️</td>
<td>(N)</td>
<td>(\frac{1}{4} N^2)</td>
<td>(\frac{1}{2} N^2)</td>
<td>use for small (N) or partially ordered</td>
</tr>
<tr>
<td>merge</td>
<td>✔️</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>timsort</td>
<td>✔️</td>
<td></td>
<td>(N)</td>
<td>(N \lg N)</td>
<td>(N \lg N)</td>
<td>improves mergesort when preexisting order</td>
</tr>
<tr>
<td>quick</td>
<td>✔️</td>
<td></td>
<td>(N \lg N)</td>
<td>(2 N \ln N) (expected)</td>
<td>(\frac{1}{2} N^2)</td>
<td>(N \log N) probabilistic guarantee; fastest in practice</td>
</tr>
<tr>
<td>3-way quick</td>
<td>✔️</td>
<td></td>
<td>(N)</td>
<td>(2 N \ln N) (expected)</td>
<td>(\frac{1}{2} N^2)</td>
<td>improves quicksort when duplicate keys</td>
</tr>
<tr>
<td>?</td>
<td>✔️</td>
<td>✔️</td>
<td>(N)</td>
<td>(N \lg N)</td>
<td>(N \lg N)</td>
<td>holy sorting grail</td>
</tr>
</tbody>
</table>