2.3 QUICKSORT

- quicksort
- selection
- duplicate keys
- system sorts

https://algs4.cs.princeton.edu
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]
A brief history

Tony Hoare.

- Invented quicksort in 1960 to translate Russian into English.
- Later learned Algol 60 (and recursion) to implement it.

Bob Sedgewick.

- Refined and popularized quicksort in 1970s.
- Analyzed many versions of quicksort.
2.3 Quicksort

- quicksort
- selection
- duplicate keys
- system sorts
Quick sort overview

Step 1. Shuffle the array.

Step 2. Partition the array so that, for some index $j$:
- Entry $a[j]$ is in place. "pivot" or "partitioning item"
- No larger entry to the left of $j$.
- No smaller entry to the right of $j$.

Step 3. Sort each subarray recursively.
Quicksort partitioning demo

Repeat until 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 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!
The music of quicksort partitioning (by Brad Lyon)

https://learnforeverlearn.com/pivot_music
private static int partition(Comparable[] a, int lo, int hi) {
    Comparable p = a[lo];
    int i = lo, j = hi+1;
    while (true) {
        while (less(a[++i], p))
            if (i == hi) break;
        while (less(p, 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
In the worst case, how many compares and exchanges does partition() make to partition a subarray 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 \( \sim n \)
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

**Quicksort Trace (Array Contents After Each Partition)**

<table>
<thead>
<tr>
<th>i</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>1</td>
<td>1</td>
</tr>
<tr>
<td>0</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**

- **No Partition for Subarrays of Size 1**

- **Result**

Quicksort Trace

<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
<th>hi</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</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>1</td>
<td>1</td>
</tr>
<tr>
<td>0</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>

**Result**

- **ACEEIKLMOPQRSSTUX**
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 it is not worth the cost.

- **Loop termination.** Terminating the loop (when pointers cross) is more subtle than it appears.

- **Equal keys.** Handling duplicate keys is trickier than it appears. [stay tuned]

- **Preserving randomness.** Shuffling is needed for performance guarantee.

- **Equivalent alternative.** Pick a random pivot 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></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>computer</td>
<td>thousand</td>
<td>million</td>
</tr>
<tr>
<td>home</td>
<td>instant</td>
<td>instant</td>
<td>2.8 hours</td>
</tr>
<tr>
<td>super</td>
<td>instant</td>
<td>instant</td>
<td>1 second</td>
</tr>
</tbody>
</table>
Why is quicksort typically faster than mergesort in practice?

A. Fewer compares.
B. Fewer array accesses.
C. Both A and B.
D. Neither A nor B.
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>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>14</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>14</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>14</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>14</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>14</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>14</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>14</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>14</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>14</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>14</td>
</tr>
<tr>
<td>10</td>
<td>9</td>
<td>14</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>14</td>
</tr>
<tr>
<td>12</td>
<td>11</td>
<td>14</td>
</tr>
<tr>
<td>13</td>
<td>12</td>
<td>14</td>
</tr>
<tr>
<td>14</td>
<td>13</td>
<td>14</td>
</tr>
</tbody>
</table>

after random shuffle
Quicksort: worst-case analysis

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

<table>
<thead>
<tr>
<th></th>
<th>lo</th>
<th>j</th>
<th>hi</th>
</tr>
</thead>
<tbody>
<tr>
<td>a[ ]</td>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
</tbody>
</table>

after random shuffle

Good news. Worst case for randomized quicksort is mostly irrelevant in practice.

- Exponentially small chance of occurring.
  (unless bug in shuffling or no shuffling)
- More likely that computer is struck by lightning bolt during execution.
QuickSort: probabilistic 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 $\sim \frac{1}{3} n \ln n$).

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

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

**Intuition.** Each partitioning step divides the problem into two subproblems, each of approximately one-half the size.

probabilistically “close enough”
**Quicksort: probabilistic 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 $\sim \frac{1}{3} n \ln n$).

**Pf.** $C_n$ satisfies the recurrence $C_0 = C_1 = 0$ and for $n \geq 2$:

\[
C_n = (n+1) + \frac{C_0 + C_{n-1}}{n} + \frac{C_1 + C_{n-2}}{n} + \ldots + \frac{C_{n-1} + C_0}{n}
\]

- Multiply both sides by $n$ and collect terms:

\[
n C_n = n(n+1) + 2(C_0 + C_1 + \ldots + C_{n-1})
\]

- Subtract from this equation the same equation for $n - 1$:

\[
n C_n - (n - 1) C_{n-1} = 2n + 2 C_{n-1}
\]

- Rearrange terms and divide by $n(n+1)$:

\[
\frac{C_n}{n+1} = \frac{C_{n-1}}{n} + \frac{2}{n+1}
\]

*Analysis beyond scope of this course*
Quicksort: probabilistic 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 properties

Quicksort analysis summary.

- Expected number of compares is $\sim 1.39 \, n \log_2 n$.
  [ standard deviation is $\sim 0.65 \, n$ ]
- Expected number of exchanges is $\sim 0.23 \, n \log_2 n$.  ← much less than mergesort
- Min number of compares is $\sim n \log_2 n$.  ← never less than mergesort
- Max number of compares is $\sim \frac{1}{2} \, n^2$.  ← but never happens

Context.  Quicksort is a (Las Vegas) randomized algorithm.

- Guaranteed to be correct.
- Running time depends on outcomes of random coin flips (shuffle).
Quicksort properties

**Proposition.** Quicksort is an *in-place* sorting algorithm.
- Partitioning: \( \Theta(1) \) extra space.
- Function–call stack: \( \Theta(\log n) \) 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>

...can guarantee \( \Theta(\log n) \) depth by recurring on smaller subarray before larger subarray (but this requires using an explicit stack)
Insertion sort small subarrays.

- Even 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.

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

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

    int median = medianOf3(a, lo, (lo + hi) >>> 1, 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 item of rank $k$.

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

Applications.

- Order statistics.
- Find the “top $k$.”

Use complexity theory as a guide.

- Easy $O(n \log n)$ algorithm. How?
- Easy $O(n)$ algorithm for $k = 0$ or $1$. How?
- Easy $\Omega(n)$ lower bound. Why?

Which is true?

- $O(n)$ algorithm? [is there a linear-time algorithm?]
- $\Omega(n \log n)$ lower bound? [is selection as hard as sorting?]
Quickselect demo

Partition 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 \( j \).

Repeat in one subarray, depending on \( j \); stop when \( j \) equals \( k \).

select element of rank \( k = 5 \)

<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>50</td>
<td>21</td>
<td>28</td>
<td>65</td>
<td>39</td>
<td>59</td>
<td>56</td>
<td>22</td>
<td>95</td>
<td>12</td>
<td>90</td>
<td>53</td>
<td>32</td>
<td>77</td>
<td>33</td>
</tr>
</tbody>
</table>

\( k = 5 \)
Quickselect

Partition 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 \( j \).

Repeat in one subarray, depending on \( j \); stop 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];
}
```
Quickselect: probabilistic analysis

**Proposition.** The expected number of compares $C_n$ to quickselect the item of rank $k$ in an array of length $n$ is $\Theta(n)$.

**Intuition.** Each partitioning step approximately halves the length of the array.

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

```java
public static void f(int n)
{
    if (n == 0) return;
    linear(n);   // do $\Theta(n)$ work
    f(n/2);     // solve one subproblem of half the size
}
```

$n + n/2 + n/4 + \ldots + 1 \sim 2n$

**Careful analysis yields:**

$$C_n \sim 2n + 2k \ln(n/k) + 2(n-k) \ln(n/(n-k))$$

$$\leq (2 + 2 \ln 2) n$$

$$\approx 3.38 n \quad \text{max occurs for median } (k = n/2)$$
Theoretical context for selection

Q. Compare-based selection algorithm that makes $\Theta(n)$ compares in the worst case?
A. Yes! [ingenious divide-and-conquer]

\[ T(n) = T(n/5) + T(7n/10) + \Theta(n) \]

\[ \uparrow \]

find pivot

\[ \uparrow \]

that eliminates 30% of items

Caveat. Constants are high $\Rightarrow$ not used in practice.

Use theory as a guide.

- Open problem: practical algorithm that makes $\Theta(n)$ compares in the worst case.
- Until one is discovered, use quickselect (if you don’t need a full sort).
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.
When partitioning, how to handle keys equal to pivot?

A. [Diagram of sequence with a pivot value P]

B. [Diagram of sequence with a pivot value P]

C. Either A or B.
War story (system sort in C)

Bug. A `qsort()` call in C that should have taken seconds was taking minutes to sort a random array of 0s and 1s.

Why is `qsort()` so slow?

```
<table>
<thead>
<tr>
<th>A0</th>
<th>A1</th>
<th>A2</th>
<th>A3</th>
<th>A4</th>
<th>A5</th>
<th>A6</th>
<th>A7</th>
<th>A8</th>
<th>A9</th>
<th>A10</th>
<th>A11</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

skip over equal keys

```
<table>
<thead>
<tr>
<th>A0</th>
<th>A1</th>
<th>A2</th>
<th>A3</th>
<th>A4</th>
<th>A5</th>
<th>A6</th>
<th>A7</th>
<th>A8</th>
<th>A9</th>
<th>A10</th>
<th>A11</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

stop scan on equal keys
Duplicate keys: partitioning strategies

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

\[ \Theta(n^2) \text{ compares when all keys equal} \]

\[
\begin{array}{ccccccc}
B & A & A & B & A & B & B \hline \\
C & C & C & & & & A \\
\end{array}
\]

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

\[ \sim n \log_2 n \text{ compares when all keys equal} \]

\[
\begin{array}{ccccccc}
B & A & A & B & A & B & C \hline \\
C & B & C & B & & & A \\
\end{array}
\]

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

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

\[
\begin{array}{ccccccc}
A & A & A & B & B & B & B \hline \\
C & C & C & & & & A \\
\end{array}
\]
**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.

- **input**
  - Red
  - Blue
  - White
  - Red
  - Blue
  - Red

- **sorted**
  - Red
  - Red
  - Blue
  - Blue
  - White

**Operations allowed.**

- \( \text{swap}(i, j) \): swap the pebble in bucket \( i \) with the pebble in bucket \( j \).
- \( \text{getColor}(i) \): determine the color of the pebble in bucket \( i \).

**Performance requirements.**

- Exactly \( n \) calls to \( \text{getColor}() \).
- At most \( n \) calls to \( \text{swap}() \).
- \( \Theta(1) \) extra space.
3-way partitioning

**Goal.** Use pivot $p = a[lo]$ to partition array into three parts so that:

- Red: smaller entries to the left of $lt$.
- White: equal entries between $lt$ and $gt$.
- Blue: larger entries to the right of $gt$. 

![Diagram showing 3-way partitioning](chart)
Dijkstra’s 3-way partitioning algorithm: demo

- Let $p = a[lo]$ be pivot.
- Scan $i$ from left to right and compare $a[i]$ to $p$.
  - less: exchange $a[i]$ with $a[lt]$; increment both $lt$ and $i$
  - greater: exchange $a[i]$ with $a[gt]$; decrement $gt$
  - equal: increment $i$

\[\begin{array}{c|cccccccccccc}
\hline
\text{lo} & \text{lt} & \text{i} & \text{gt} & \text{hi} \\
\hline
\downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
\text{P}_1 & \text{D} & \text{B} & \text{X} & \text{W} & \text{P}_2 & \text{P}_3 & \text{V} & \text{P}_4 & \text{A} & \text{P}_5 & \text{C} & \text{Y} & \text{Z} \\
\hline
\end{array}\]
Dijkstra’s 3-way partitioning algorithm: demo

- Let \( p = a[lo] \) be pivot.
- Scan \( i \) from left to right and compare \( a[i] \) to \( p \).
  - less: exchange \( a[i] \) with \( a[lt] \); increment both \( lt \) and \( i \)
  - greater: exchange \( a[i] \) with \( a[gt] \); decrement \( gt \)
  - equal: increment \( i \)
private static void sort(Comparable[] a, int lo, int hi) {
    if (hi <= lo) return;
    Comparable p = a[lo];

    int lt = lo, gt = hi;
    int i = lo + 1;
    while (i <= gt) {
        int cmp = a[i].compareTo(p);
        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);
}
What is the worst-case number of compares to 3-way quicksort an array of length \( n \) containing only 7 distinct values?

A. \( \Theta(n) \)

B. \( \Theta(n \log n) \)

C. \( \Theta(n^2) \)

D. \( \Theta(n^7) \)
## 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 sorted arrays</td>
</tr>
<tr>
<td>merge</td>
<td>✔</td>
<td></td>
<td>$\frac{1}{2}n \log_2 n$</td>
<td>$n \log_2 n$</td>
<td>$n \log_2 n$</td>
<td>$\Theta(n \log n)$ guarantee; stable</td>
</tr>
<tr>
<td>timsort</td>
<td>✔</td>
<td></td>
<td>$n$</td>
<td>$n \log_2 n$</td>
<td>$n \log_2 n$</td>
<td>improves mergesort when pre-existing order</td>
</tr>
<tr>
<td>quick</td>
<td>✔</td>
<td></td>
<td>$n \log_2 n$</td>
<td>$2n \ln n$</td>
<td>$\frac{1}{2}n^2$</td>
<td>$\Theta(n \log n)$ probabilistic guarantee; fastest in practice</td>
</tr>
<tr>
<td>3-way quick</td>
<td>✔</td>
<td>✔</td>
<td>$n$</td>
<td>$2n \ln n$</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>$n \log_2 n$</td>
<td>$n \log_2 n$</td>
<td>holy sorting grail</td>
</tr>
</tbody>
</table>

*number of compares to sort an array of $n$ elements*
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.

...
Bentley–McIlroy quicksort.

- Cutoff to insertion sort for small subarrays.
- Pivot selection: median of 3 or Tukey’s ninther.
- Partitioning scheme: Bentley–McIlroy 3-way partitioning.

In the wild. C, C++, Java 6, ….
Replacement of quicksort in java.util.Arrays with new dual-pivot quicksort

Hello All,

I'd like to share with you new Dual-Pivot Quicksort which is faster than the known implementations (theoretically and experimental). I'd like to propose to replace the JDK's Quicksort implementation by new one.

...

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 ]

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

Date: Thu, 29 Oct 2009 11:19:39 +0000
Subject: Replace quicksort in java.util.Arrays with dual-pivot implementation

Changeset: b05abb410c52
Author: alamb
Date: 2009-10-29 11:18 +0000
URL: http://hg.openjdk.java.net/jdk7/tl/jdk/rev/b05abb410c52

6880672: Replace quicksort in java.util.Arrays with dual-pivot implementation
Reviewed-by: jjb
Contributed-by: vladimir.yaroslavskiy at sun.com, joshua.bloch at google.com,
jbentley at avaya.com

! src/share/classes/java/util/Arrays.java
+ src/share/classes/java/util/DualPivotQuicksort.java

https://mail.openjdk.java.net/pipermail/compiler-dev/2009-October.txt
Use two pivots $p_1$ 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>$&lt; p_1$</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>↑</td>
<td>↑</td>
<td>↑</td>
<td>↑</td>
<td>↑</td>
</tr>
<tr>
<td>lo</td>
<td>lt</td>
<td>gt</td>
<td>hi</td>
<td></td>
</tr>
</tbody>
</table>

Recursively sort three subarrays (skip middle subarray if $p_1 = p_2$).

degenerates to Dijkstra’s 3-way partitioning

In the wild. Java 8, Java 11, Python unstable sort, Android, …
Suppose you are the lead architect of a new programming language. Which sorting algorithm(s) would you use for the system sort? Defend your answer.
System sorts in Java 8 and Java 11

Arrays.sort() and Arrays.parallelSort().

- Has one method for Comparable objects.
- Has an overloaded method for each primitive type.
- Has an overloaded method for use with a Comparator.
- Has overloaded methods for sorting subarrays.

Algorithms.

- Timsort for reference types.
- Dual-pivot quicksort for primitive types.
- Parallel mergesort for Arrays.parallelSort().

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

Bottom line. Use the system sort!