2.3 QUICKSORT

- quicksort
- selection
- duplicate keys
- system sorts

Two classic sorting algorithms

Critical components in the world's computational infrastructure.

- Full scientific understanding of their properties has enabled us to develop them into practical system sorts.
- Quicksort honored as one of top 10 algorithms of 20th century in science and engineering.

Mergesort.

- Java sort for objects.
- Perl, C++ stable sort, Python stable sort, Firefox JavaScript, ...

Quicksort.

- Java sort for primitive types.
- C qsort, Unix, Visual C++, Python, Matlab, Chrome JavaScript, ...

Quicksort t-shirt
**Quicksort**

**Basic plan.**
- Shuffle the array.
- **Partition** 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 $j$
- Sort each piece recursively.

**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: 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;
        while (less(a[lo], a[--j]))
            if (j == lo) break;
        if (i >= j) break;
        exch(a, i, j);
    }
    exch(a, lo, j);
    return j;
}
```

**Quick Sort Example**

Sir Charles Antony Richard Hoare
1980 Turing Award

**Quick Sort: Demo**

input: Q U I C K S O R T E X A M P L E
shuffle: K R A T E L E P U I M Q C X O S
partition: E C A I E K L P U T M Q R X O S
sort left: A C E E I K L P U T M Q R X O S
sort right: A C E E I K L M O P Q R S T U X
result: A C E E I K L M O P Q R S T U X

**Quick Sort Partitioning Overview**

**Private static int partition(Comparable[] a, int lo, int hi)**

```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;
}
```

**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: 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;
        while (less(a[lo], a[--j]))
            if (j == lo) break;
        if (i >= j) break;
        exch(a, i, j);
    }
    exch(a, lo, j);
    return j;
}
```

**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;
        while (less(a[lo], a[--j]))
            if (j == lo) break;
        if (i >= j) break;
        exch(a, i, j);
    }
    exch(a, lo, j);
    return j;
}
```
QuickSort: Java implementation

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

    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);
    }
}
```

Shuffle needed for performance guarantee (stay tuned)

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 a bit trickier than it might seem.

**Staying in bounds.** The \((j == lo)\) test is redundant (why?), but the \((i == hi)\) test is not.

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

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

QuickSort trace

```
lo  j  hi   0  1  2  3  4  5  6  7  8  9  10  11  12  13  14  15
K  R  A  T  E  L  E  P  U  I  M  Q  C  X  O  S
0  15  E  C  A  I  E  K  L  P  U  T  M  Q  R  X  O  S
0  3  4  E  C  A  E  I  K  L  P  U  T  M  Q  R  X  O  S
0  2  2  A  C  E  E  I  K  L  P  U  T  M  Q  R  X  O  S
0  0  1  A  C  E  E  I  K  L  P  U  T  M  Q  R  X  O  S
0  4  4  A  C  E  E  I  K  L  P  U  T  M  Q  R  X  O  S
6  6  15  A  C  E  E  I  K  L  P  U  T  M  Q  R  X  O  S
7  9  15  A  C  E  E  I  K  L  M  O  P  T  Q  R  X  U  S
7  7  8  A  C  E  E  I  K  L  M  O  P  T  Q  R  X  U  S
6  8  8  A  C  E  E  I  K  L  M  O  P  T  Q  R  X  U  S
10  13  15  A  C  E  E  I  K  L  M  O  P  S  Q  R  T  X
10  12  12  A  C  E  E  I  K  L  M  O  P  S  Q  R  T  X
10  11  11  A  C  E  E  I  K  L  M  O  P  Q  R  S  T  X
10  10  10  A  C  E  E  I  K  L  M  O  P  Q  R  S  T  X
14  14  15  A  C  E  E  I  K  L  M  O  P  Q  R  S  T  U  X
15  15  15  A  C  E  E  I  K  L  M  O  P  Q  R  S  T  U  X
```

QuickSort trace (array contents after each partition)

**Initial values**

- Random shuffle

**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: 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)</th>
<th>mergesort (N log N)</th>
<th>quicksort (N log N)</th>
</tr>
</thead>
<tbody>
<tr>
<td>computer</td>
<td>thousand</td>
<td>million</td>
<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: best-case analysis

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

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

Quicksort: worst-case analysis

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

Quicksort: average-case analysis

Proposition. The average number of compares $C_N$ to quicksort an array of $N$ distinct keys is $\sim 2N \ln N$ (and the number of exchanges is $\sim \frac{1}{2} N \ln N$).

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

$$C_N = (N + 1) + \left( \frac{C_0 + C_{N-1}}{N} \right) + \left( \frac{C_1 + C_{N-2}}{N} \right) + \ldots + \left( \frac{C_{N-1} + C_0}{N} \right)$$

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

$$NC_N = N(N + 1) + 2(C_0 + C_1 + \ldots + C_{N-1})$$

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

$$NC_N - (N-1)C_{N-1} = 2N + 2C_{N-1}$$

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

$$\frac{C_N}{N+1} = \frac{C_{N-1}}{N} + \frac{2}{N+1}$$
Repeatedly apply above equation:
\[
\frac{C_N}{N+1} = \frac{C_{N-1}}{N} + \frac{2}{N+1}
\]
\[
= \frac{C_{N-2}}{N-1} + \frac{2}{N} + \frac{2}{N+1}
\]
\[
= \frac{C_{N-3}}{N-2} + \frac{2}{N-1} + \frac{2}{N} + \frac{2}{N+1}
\]
\[
= \frac{2}{3} + \frac{2}{4} + \frac{2}{5} + \ldots + \frac{2}{N+1}
\]

Approximate sum by an integral:
\[
C_N = 2(N+1) \left( \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \ldots + \frac{1}{N+1} \right)
\]
\[
\sim 2(N+1) \int_{\frac{1}{3}}^{N+1} \frac{1}{x} \, dx
\]

Finally, the desired result:
\[
C_N \sim 2(N+1) \ln N \approx 1.39N \ln N
\]
Quicksort: summary of performance characteristics

**Worst case.** Number of compares is quadratic.
- \( N + (N - 1) + (N - 2) + \ldots + 1 \approx \frac{1}{2} N^2. \)
- More likely that your computer is struck by lightning bolt.

**Average case.** Number of compares is \( \sim 1.39 N \lg N. \)
- 39% more compares than mergesort.
- But faster than mergesort in practice because of less data movement.

**Random shuffle.**
- Probabilistic guarantee against worst case.
- Basis for math model that can be validated with experiments.

**Caveat emptor.** Many textbook implementations go quadratic if array
- Is sorted or reverse sorted.
- Has many duplicates (even if randomized!)

---

Quicksort properties

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

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

---

**Proposition.** Quicksort is not stable.

**Pf.**

---

Quicksort: practical improvements

**Insertion sort small subarrays.**
- Even quicksort has too much overhead for tiny subarrays.
- Cutoff to insertion sort for \( \approx 10 \) items.
- Note: could delay insertion sort until one pass at end.

---

```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.

\[ \sim 12/7 N \ln N \text{ compares (slightly fewer)} \]
\[ \sim 12/35 N \ln N \text{ exchanges (slightly more)} \]

```java
private static void sort(Comparable[] a, int lo, int hi)
{
    if (hi <= lo) return;
    int m = medianOf3(a, lo, lo + (hi - lo)/2, hi);
    swap(a, lo, m);
    int j = partition(a, lo, hi);
    sort(a, lo, j-1);
    sort(a, j+1, hi);
}
```
Selection

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

**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 for each $k$?
Quick-select: mathematical analysis

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

**Proof sketch.**
- Intuitively, each partitioning step splits array approximately in half: 
  \[ N + N/2 + N/4 + \ldots + 1 \sim 2N \text{ compares}. \]
- Formal analysis similar to quicksort analysis yields:
  \[ C_N = 2N + 2k \ln (N/k) + 2(N-k) \ln (N/(N-k)) \]
  \[ (2 + 2 \ln 2)N \text{ to find the median} \]

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

Theoretical context for selection


**Abstract.** The number of comparisons required to select the i-th smallest of n numbers is shown to be at most a linear function of n by analysis of a new selection algorithm -- PICK. Specifically, no more than \( 5.4305 n \) comparisons are ever required. This bound is improved for extreme values of i, and a new lower bound on the requisite number of comparisons is also proved.

**Remark.** But, constants are too high \( \Rightarrow \) not used in practice.

Use theory as a guide.
- Still worthwhile to seek practical linear-time (worst-case) algorithm.
- Until one is discovered, use quick-select if you don’t need a full sort.

War story (C qsort function)

**A beautiful bug report.** [Allan Wilks and Rick Becker, 1991]

We found that qsort is unbearably slow on "organ-pipe" inputs like "01233210":

```c
int n = atoi(argv[1]), i, x[100000];
for (i = 0; i < n; i++)
    x[i] = i;
for (; i < 2*n; i++)
    x[i] = 2*n-1-i;
qsort(x, 2*n, sizeof(int), intcmp);
```

Here are the timings on our machine:

```
$ time a.out 2000
real  5.85s
```

```
$ time a.out 4000
real  21.64s
```

```
$ time a.out 8000
real  85.11s
```
War story (C qsort function)

AT&T Bell Labs (1991). Allan Wilks and Rick Becker discovered that a qsort() call that should have taken seconds was taking minutes.

Why is qsort() so slow?

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

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: the problem

Mergesort with duplicate keys. Between $\frac{1}{2} N \lg N$ and $N \lg N$ compares.

Quicksort with duplicate keys. Algorithm goes quadratic unless partitioning stops on equal keys!

which is why ours does!
(but many textbook implementations do not)

STOP ONE EQUAL KEYS

swap
if we don't stop on equal keys
if we stop on equal keys

Mistake. Put all items equal to the partitioning item on one side.
Consequence. $\sim \frac{1}{2} N^2$ compares when all keys equal.

Recommended. Stop scans on items equal to the partitioning item.
Consequence. $\sim N \lg N$ compares when all keys equal.

Desirable. Put all items equal to the partitioning item in place.
### 3-way partitioning

**Goal.** Partition array into 3 parts so that:
- Entries between \( l_t \) and \( g_t \) equal to partition item \( v \).
- No larger entries to left of \( l_t \).
- No smaller entries to right of \( g_t \).

---

**Dutch national flag problem.** [Edsger Dijkstra]
- Conventional wisdom until mid 1990s: not worth doing.
- New approach discovered when fixing mistake in C library `qsort()`.
- Now incorporated into `qsort()` and Java system sort.

---

### Dijkstra 3-way partitioning demo

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

---

### Dijkstra's 3-way partitioning: trace

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

---

3-way partitioning trace (array contents after each loop iteration)
private static void sort(Comparable[] a, int lo, int hi) {
    if (hi <= lo) return;
    int lt = lo, gt = hi;
    Comparable v = a[lo];
    int i = lo;
    while (i <= gt) {
        int cmp = a[i].compareTo(v);
        if      (cmp < 0) exch(a, lt++, i++);
        else if (cmp > 0) exch(a, i, gt--);
        else        i++;
    }
    sort(a, lo, lt - 1);
    sort(a, gt + 1, hi);
}

3-way quicksort: Java implementation

duplicate keys: lower bound

Sorting lower bound. If there are $n$ distinct keys and the $i^{th}$ one occurs $x_i$ times, any compare-based sorting algorithm must use at least

$$\log \left( \frac{n!}{x_1! x_2! \cdots x_n!} \right) \sim - \sum_{i=1}^{n} x_i \log \frac{x_i}{n} \text{ compares in the worst case.}$$

Proposition. [Sedgewick-Bentley, 1997] Quick sort with 3-way partitioning is entropy-optimal.

Proof. [beyond scope of course]

Bottom line. Randomized quicksort with 3-way partitioning reduces running time from linearithmic to linear in broad class of applications.

3-way quicksort: visual trace
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>$p_1 &lt; p_1$</th>
<th>$p_1$</th>
<th>$p_1 \leq$ and $\leq p_2$</th>
<th>$p_2$</th>
<th>$&gt; p_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>↑ lo</td>
<td>↑ lt</td>
<td>↑ i</td>
<td>↑ gt</td>
<td>↑ hi</td>
</tr>
</tbody>
</table>

Recursively sort three subarrays.

Note. Skip middle subarray if $p_1 = p_2$.

Dual-pivot partitioning demo

Initialization.

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

exchange $a[lo]$ and $a[hi]$
**Dual-pivot partitioning demo**

**Finalize.**
- Exchange \( a[lo] \) with \( a[-1t] \).
- Exchange \( a[hi] \) with \( a[+gt] \).

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

\[ C \quad E \quad A \quad F \quad K \quad L \quad M \quad R \quad Q \quad S \quad Z \quad V \quad Y \quad T \]

3-way partitioned

**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>( p_1 &lt; p_1 )</th>
<th>( p_1 )</th>
<th>( p_1 \leq ) and ( \leq p_2 )</th>
<th>( p_2 )</th>
<th>( &gt; p_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \uparrow ) lo</td>
<td>( \uparrow ) lt</td>
<td>( \uparrow ) gt</td>
<td>( \uparrow ) hi</td>
<td></td>
</tr>
</tbody>
</table>

**Proposition.** [Wild and Nevel] \( 1.9 \ln N \) compares and \( 0.6 \ln N \) exchanges.

**Improvements.** Take 5 random items, use 2\(^{nd}\) and 4\(^{th}\) largest for \( p_1 \) and \( p_2 \).

**Now widely used.** Java 7, Python unstable sort, ...

**Which sorting algorithm to use?**

Many sorting algorithms to choose from:

**Internal (in-memory) sorts.**
- Insertion sort, selection sort, bubblesort, shaker sort.
- Quicksort, mergesort, heapsort, samplesort, shellsort.
- Solitaire sort, red-black sort, splaysort, dual-pivot quicksort, timsort, ...

**External sorts.** Poly-phase mergesort, cascade-merge, psort, ....

**String/radix sorts.** Distribution, MSD, LSD, 3-way string quicksort.

**Parallel sorts.**
- Bitonic sort, Batcher even-odd sort.
- Smooth sort, cube sort, column sort.
- GPUsort.

**Applications have diverse attributes.**
- Stable?
- Parallel?
- Deterministic?
- Duplicate keys?
- Multiple key types?
- Linked list or arrays?
- Large or small items?
- Randomly-ordered array?
- Guaranteed performance?

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

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

**Which sorting algorithm to use?**

- Many more combinations of attributes than algorithms