2.3 Quicksort

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, Python stable sort.

Quicksort.
- Java sort for primitive types.
- C qsort, Unix, g++, Visual C++, Python.

Basic plan.
- Shuffle the array.
- Partition so that, for some j
  - element a[j] is in place
  - no larger element to the left of j
  - no smaller element to the right of j
- Sort each piece recursively.

Sir Charles Antony Richard Hoare
1980 Turing Award
Quicksort partitioning

Basic plan:
- Scan i from left for an item that belongs on the right.
- Scan j from right for item that belongs on the left.
- Exchange a[i] and a[j].
- Continue until pointers cross.

Quicksort: 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;
    exch(a, i, j);
  }
  exch(a, lo, j);
  return j;
}
```

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);
  }
  ...
}

Quicksort: Java code for partitioning

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

Quicksort trace

Initial values:
-1  5  12  9  4  3  5  6  7  8  9  10  11  12  13  14  15

Scan left, scan right:
-1  5  12  9  4  3  5  6  7  8  9  10  11  12  13  14  15
  Exchange

Scan right, scan left:
6  5  12  9  4  3  5  6  7  8  9  10  11  12  13  14  15
  Exchange

Scan left, scan right:
5  6  12  9  4  3  5  6  7  8  9  10  11  12  13  14  15
  Exchange

Scan right, scan left:
5  6  12  9  4  3  5  6  7  8  9  10  11  12  13  14  15
  Exchange

Final exchange:
0  5  6  12  9  4  3  5  6  7  8  9  10  11  12  13  14  15

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

Quick sort example

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

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

Quick sort trace (array contents after each partition)
Quicksort animation

50 random elements

http://www.sorting-algorithms.com/quick-sort

Quicksort: implementation details

Partitioning in-place. Using a spare 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 == hi) test is redundant (why?), but the (i == lo) test is not.

Preserving randomness. Shuffling is needed for performance guarantee.

Equal keys. When duplicates are present, it is (counter-intuitively) best to stop on elements equal to the partitioning element.

Quicksort: empirical analysis

Running time estimates:

• Home pc executes $10^8$ compares/second.
• Supercomputer executes $10^{12}$ compares/second.

<table>
<thead>
<tr>
<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>
</tr>
<tr>
<td>home</td>
<td>instant</td>
<td>2.8 hours</td>
</tr>
<tr>
<td>super</td>
<td>instant</td>
<td>1 second</td>
</tr>
</tbody>
</table>

Lesson 1. Good algorithms are better than supercomputers.

Lesson 2. Great algorithms are better than good ones.

Quicksort: best case analysis

Best case. Number of compares is ~ N log N.

```
| le  | j   | hi  | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 | 13 | 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  |    |
| 0   | 14  | D   | A  | B  | C  | F  | E  | G  | H  | L  | I  | K  | J  | N  | M  | O  |    |    |
| 3   | 6   | B   | A   | C  | D  | F  | E  | G  | H  | L  | I  | K  | J  | N  | M  | O  |    |    |
| 1   | 2   | 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   | A   | B   | C  | D  | E  | F  | G  | H  | L  | I  | K  | J  | N  | M  | O  |    |    |
| 4   | 5   | 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  | A   | B   | C  | D  | E  | F  | G  | H  | J  | K  | L  | N  | M  | O  |    |    |
| 8   | 10  | A   | B   | C  | D  | E  | F  | G  | H  | I  | K  | L  | N  | M  | O  |    |    |
| 8   | 8   | A   | B   | C  | D  | E  | F  | G  | H  | I  | K  | L  | N  | M  | O  |    |    |
| 10  | 10  | A   | B   | C  | D  | E  | F  | G  | H | I  | J  | K  | L  | M  | N  | O  |    |    |
| 12  | 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  |    |    |
```
Quicksort: worst case analysis

Worst case. Number of compares is \( \sim N^2 / 2 \).

<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>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</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>1</td>
<td>1</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>N</td>
<td>O</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>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>\vdots</td>
<td></td>
<td>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
</tr>
<tr>
<td>14</td>
<td>14</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>N</td>
<td>O</td>
</tr>
</tbody>
</table>

Quicksort: average-case analysis

Proposition I. The average number of compares \( C_N \) to quicksort an array of \( N \) elements is \( \sim 2N \ln N \) (and the number of exchanges is \( \sim \frac{1}{3} N \ln N \)).

\[ C_N = (N+1) + \frac{C_{N-2} + C_{N-3} + \ldots + C_1}{N} + \frac{C_{N-1} + C_{N-2} + \ldots + C_0}{N} \]

- 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 \( N(N+1) \):
  \[ \frac{C_N}{N+1} = \frac{C_{N-1}}{N} + \frac{2}{N+1} \]

\begin{itemize}
  \item Repeatedly apply above equation:
  \[ \frac{C_N}{N+1} = \frac{C_{N-1}}{N} + \frac{2}{N+1} \]
  \item Approximate sum by an integral:
  \[ C_N \sim 2(N+1) \left(1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{N} \right) \sim 2(N+1) \int_1^N \frac{1}{x} \, dx \]
  \item Finally, the desired result:
  \[ C_N \sim 2(N+1) \ln N \approx 1.39N \ln N \]
\end{itemize}

Quicksort: summary of performance characteristics

Worst case. Number of compares is quadratic.
\( \sim N + (N-1) + (N-2) + \ldots + 1 \sim N^2 / 2 \).
More likely that your computer is struck by lightning.

Average case. Number of compares is \( \sim 1.39 N \ln N \).
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 input:
- Is sorted or reverse sorted.
- Has many duplicates (even if randomized)! [stay tuned]
Quicksort: practical improvements

Median of sample.
• Best choice of pivot element = median.
• Estimate true median by taking median of sample.

Insertion sort small subarrays.
• Even quicksort has too much overhead for tiny subarrays.
• Can delay insertion sort until end.

Optimize parameters.
• Median-of-3 random elements.
• Cutoff to insertion sort for ≈ 10 elements.

Non-recursive version.
• Use explicit stack.
• Always sort smaller half first.

Quicksort with cutoff to insertion sort: visualization

Selection

Goal. Find the $k^{th}$ largest element.
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 $O(N \log N)$ upper bound.
• Easy $O(N)$ upper bound for $k = 1, 2, 3$.
• Easy $\Omega(N)$ lower bound.

Which is true?
• $\Omega(N \log N)$ lower bound?
• $O(N)$ upper bound?

is selection as hard as sorting?
is there a linear-time algorithm for all $k$?
Quick-select

Partition array so that:
• Element $a[j]$ is in place.
• No larger element to the left of $j$.
• No smaller element 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 linear time on average.

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

**Ex.** $(2 + 2 \ln 2) N$ compares to find the median.

**Remark.** Quick-select uses $\sim N^2/2$ compares in worst case, but as with quicksort, the random shuffle provides a probabilistic guarantee.

Theoretical context for selection

**Challenge.** Design algorithm whose worst-case running time is linear.

**Proposition.** [Blum, Floyd, Pratt, Rivest, Tarjan, 1973] There exists a compare-based selection algorithm whose worst-case running time is linear.

**Remark.** But, algorithm is too complicated to be useful 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.

Generic methods

In our `select()` implementation, client needs a cast.

```java
double[] a = new double[N];
for (int i = 0; i < N; i++)
a[i] = StdRandom.uniform();
Double median = (Double) Quick.select(a, N/2);
```

The compiler also complains.

```bash
% javac Quick.java
Note: Quick.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

Q. How to fix?
Generic methods

Pedantic (safe) version. Compiles cleanly, no cast needed in client.

```java
public class QuickPedantic{
    public  static <Key extends Comparable<Key>> Key select(Key[] a, int k) {
        /* as before */
    }
    public  static <Key extends Comparable<Key>> void sort(Key[] a) {
        /* as before */
    }
    private static <Key extends Comparable<Key>> int partition(Key[] a, int lo, int hi) {
        /* as before */
    }
    private static <Key extends Comparable<Key>> boolean less(Key v, Key w) {
        /* as before */
    }
    private static <Key extends Comparable<Key>> void exch(Key[] a, int i, int j) {
        Key swap = a[i]; a[i] = a[j]; a[j] = swap;
    }
}
```

Remark. Obnoxious code needed in system sort; not in this course (for brevity).

Duplicate keys

Often, purpose of sort is to bring records with duplicate keys together.
- Sort population by age.
- Find collinear points.
- 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

Mergesort with duplicate keys. Always \(\sim N \lg N\) compares.

Quicksort with duplicate keys.
- Algorithm goes quadratic unless partitioning stops on equal keys!
- 1990s C user found this defect in `qsort()`.

Several textbook and system implementations also have this defect.
Duplicate keys: the problem

**Mistake.** Put all keys equal to the partitioning element on one side.

**Consequence.** $\sim N^2/2$ compares when all keys equal.

$\begin{array}{lcl}
B & A & A \\
B & A & B \\
B & B & B \\
B & C & C \\
C & C & C \\
\end{array}$

$\begin{array}{lcl}
A & A & A \\
A & A & A \\
A & A & A \\
A & A & A \\
A & A & A \\
\end{array}$

**Recommended.** Stop scans on keys equal to the partitioning element.

**Consequence.** $\sim N \lg N$ compares when all keys equal.

3-way partitioning

**Goal.** Partition array into 3 parts so that:
- Elements between $lt$ and $gt$ equal to partition element $v$.
- No larger elements to left of $lt$.
- No smaller elements to right of $gt$.

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.

3-way partitioning: Dijkstra’s solution

3-way partitioning.
- Let $v$ be partitioning element $a[lo]$.
- Scan $i$ from left to right.
  - $a[i]$ less than $v$: exchange $a[i]$ with $a[lt]$ and increment both $lt$ and $i$.
  - $a[i]$ greater than $v$: exchange $a[gt]$ with $a[i]$ and decrement $gt$.
  - $a[i]$ equal to $v$: increment $i$.

All the right properties.
- In-place.
- Not much code.
- Small overhead if no equal keys.

3-way partitioning: trace

3-way partitioning trace (array contents after each loop iteration)
3-way quicksort: Java implementation

```java
private static void sort(Comparable[] a, int lo, int hi) {
    if (hi <= lo) return;
    int lt = lo, gt = hi;
    Comparable v = a[lo];
    int i = lo;
    while (i <= gt) {
        int cmp = a[i].compareTo(v);
        if      (cmp < 0) exch(a, lt++, i++);
        else if (cmp > 0) exch(a, i, gt--);
        else              i++;
    }
    sort(a, lo, lt - 1);
    sort(a, gt + 1, hi);
}
```

3-way quicksort: visual trace

![Visual trace of quicksort with 3-way partitioning](image)

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

\[
\log \left( N! \prod_{i=1}^{n} \frac{x_i}{x_i!} \right) \sim - \sum_{i=1}^{n} x_i \log \frac{x_i}{N} 
\]

compares in the worst case.

**Proposition.** [Sedgewick-Bentley, 1997]

Quick sort with 3-way partitioning is entropy-optimal.

**Pf.** [beyond scope of course]

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

Sorting algorithms are essential in a broad variety of applications:

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

• Find the median.
• Find the closest pair.
• Binary search in a database.
• Identify statistical outliers.
• Find duplicates in a mailing list.

• Data compression.
• Computer graphics.
• Computational biology.
• Supply chain management.
• Load balancing on a parallel computer.

Every system needs (and has) a system sort!

Java system sorts

Java uses both mergesort and quicksort.

• Arrays.sort() Sorts array of comparable or any primitive type.
• Uses quicksort for primitive types; mergesort for objects.

Java system sort for primitive types

Engineering a sort function. [Bentley-McIlroy, 1993]

• Original motivation: improve qsort().
• Basic algorithm = 3-way quicksort with cutoff to insertion sort.
• Partition on Tukey’s ninther: median of the medians of 3 samples, each of 3 elements.

Why use Tukey’s ninther?
• Better partitioning than random shuffle.
• Less costly than random shuffle.

Achilles heel in Bentley-McIlroy implementation (Java system sort)

Based on all this research, Java’s system sort is solid, right?

A killer input.
• Blows function call stack in Java and crashes program.
• Would take quadratic time if it didn’t crash first.
Achilles heel in Bentley-McIlroy implementation (Java system sort)

McIlroy’s devious idea. [A Killer Adversary for Quicksort]

• Construct malicious input while running system quicksort, in response to elements compared.
• If v is partitioning element, commit to \((v < a[i])\) and \((v < a[j])\), but don’t commit to \((a[i] < a[j])\) or \((a[j] > a[i])\) until \(a[i]\) and \(a[j]\) are compared.

Consequences.

• Confirms theoretical possibility.
• Algorithmic complexity attack: you enter linear amount of data; server performs quadratic amount of work.

Remark. Attack is not effective if array is shuffled before sort.

Q. Why do you think system sort is deterministic?

System sort: Which algorithm to use?

Many sorting algorithms to choose from:

Internal sorts.

• Insertion sort, selection sort, bubblesort, shaker sort.
• Quicksort, mergesort, heapsort, samplesort, shellsort.
• Solitaire sort, red-black sort, splaysort, Dobosiewicz sort, psort, ...

External sorts. Poly-phase mergesort, cascade-merge, oscillating sort.

Radix sorts. Distribution, MSD, LSD, 3-way radix quicksort.

Parallel sorts.

• Bitonic sort, Batcher even-odd sort.
• Smooth sort, cube sort, column sort.
• GPUsort.

Applications have diverse attributes.

• Stable?
• Parallel?
• Deterministic?
• Keys all distinct?
• Multiple key types?
• Linked list or arrays?
• Large or small records?
• Is your array randomly ordered?
• Need guaranteed performance?

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

Q. Is the system sort good enough?
A. Usually.
Which sorting algorithm?