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, ...
```java
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[i] < x) && (i < right)) i++;
        while ((x < items[j]) && (j > 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, i);
    if (i < right) quicksort(items, i, right);
}
```
› quicksort
› selection
› duplicate keys
› system sorts
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
Quicksort partitioning

Basic plan.
• Scan \( i \) from left for an item that belongs on the right.
• Scan \( j \) from right for an item that belongs on the left.
• Exchange \( a[i] \) and \( a[j] \).
• Repeat until pointers cross.

Partitioning trace (array contents before and after each exchange)
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;
}
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

Quicksort trace (array contents after each partition)
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 a bit trickier than it might seem.

Staying in bounds. The \((j == 10)\) 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: empirical analysis

Running time estimates:

- Home PC executes $10^8$ compares/second.
- Supercomputer executes $10^{12}$ compares/second.

<table>
<thead>
<tr>
<th>Computer</th>
<th>insertion sort ($N^2$)</th>
<th>mergesort ($N \log N$)</th>
<th>quicksort ($N \log N$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>home</td>
<td>instant</td>
<td>instant</td>
<td>instant</td>
</tr>
<tr>
<td></td>
<td>2.8 hours</td>
<td>1 second</td>
<td>0.6 sec</td>
</tr>
<tr>
<td></td>
<td>317 years</td>
<td>18 min</td>
<td>12 min</td>
</tr>
<tr>
<td>super</td>
<td>instant</td>
<td>instant</td>
<td>instant</td>
</tr>
<tr>
<td></td>
<td>1 second</td>
<td>instant</td>
<td>instant</td>
</tr>
<tr>
<td></td>
<td>1 week</td>
<td>instant</td>
<td>instant</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 $\sim N \lg N$.
**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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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>
</tbody>
</table>
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 $\sim \frac{1}{3} N \ln N$).

**Pf 1.** $C_N$ satisfies the recurrence $C_0 = C_1 = 0$ and for $N \geq 2$:

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

- 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 $N(N + 1)$:

\[
\frac{C_N}{N+1} = \frac{C_{N-1}}{N} + \frac{2}{N+1}
\]
QuickSort: average-case analysis

- Repeatedly apply above equation:

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

previous equation

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

substitute previous equation

\[
= \frac{C_{N-3}}{N-2} + \frac{2}{N-1} + \frac{2}{N} + \frac{2}{N+1}
\]

\[
= \frac{2}{3} + \frac{2}{4} + \frac{2}{5} + \ldots + \frac{2}{N+1}
\]

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

**Pf 2.** Consider BST representation of keys 1 to $N$. 

![BST representation](attachment:image.png)
**Quickselect: 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 $\sim \frac{1}{3} N \ln N$).

**Pf 2.** Consider BST representation of keys 1 to $N$.

- A key is compared only with its ancestors and descendants.
- Probability $i$ and $j$ are compared equals $2 / |j - i + 1|$.
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 $\sim \frac{1}{3} N \ln N$).

Pf 2. Consider BST representation of keys 1 to $N$.
- A key is compared only with its ancestors and descendants.
- Probability $i$ and $j$ are compared equals $2 / |j - i + 1|$.

- Expected number of compares
  $\sum_{i=1}^{N} \sum_{j=i+1}^{N} \frac{2}{j - i + 1} = 2 \sum_{i=1}^{N} \sum_{j=2}^{N-i+1} \frac{1}{j} \leq 2N \sum_{j=1}^{N} \frac{1}{j}$

  $\sim 2N \int_{x=1}^{N} \frac{1}{x} \, dx = 2N \ln N$
QuickSort: summary of performance characteristics

**Worst case.** Number of compares is quadratic.
- \(N + (N - 1) + (N - 2) + \ldots + 1 \sim \frac{1}{2} N^2.\)
- 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: 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.

```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);
}
```
Quicksort with median-of-3 partitioning and cutoff to insertion sort: visualization

input

result of first partition

partitioning element

left subarray partially sorted

both subarrays partially sorted

result
- quicksort
- selection
- duplicate keys
- system sorts
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 $O(N \log N)$ upper bound. How?
- Easy $O(N)$ upper bound for $k = 1, 2, 3$. How?
- Easy $\Omega(N)$ lower bound. Why?

**Which is true?**
- $\Omega(N \log N)$ lower bound? is selection as hard as sorting?
- $O(N)$ upper bound? is there a linear-time algorithm for each $k$?
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 linear time on average.

Pf sketch.

• Intuitively, each partitioning step splits array approximately in half:
  \[ N + \frac{N}{2} + \frac{N}{4} + \ldots + 1 \sim 2N \text{ compares.} \]

• Formal analysis similar to quicksort analysis yields:

\[ C_N = 2N + k \ln \left( \frac{N}{k} \right) + (N - k) \ln \left( \frac{N}{N - k} \right) \]

(2 + 2 ln 2) N 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


Remark. But, constants are too high ⇒ 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.
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 complains.

```
% 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).

› quicksort
› selection
› duplicate keys
› system sorts
Duplicate keys

Often, purpose of sort is to bring items with equal 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 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!
- 1990s C user found this defect in `qsort()`.

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

Mistake. Put all items equal to the partitioning item on one side.

Consequence. $\sim \frac{1}{2} N^2$ compares when all keys equal.

\[
\begin{array}{cccccccc}
C & C & C & C & & & &
\end{array}
\quad
\begin{array}{cccccccc}
\end{array}
\]

Recommended. Stop scans on items equal to the partitioning item.

Consequence. $\sim N \log N$ compares when all keys equal.

\[
\begin{array}{cccccccc}
C & B & C & B & & & &
\end{array}
\quad
\begin{array}{cccccccc}
\end{array}
\]

Desirable. Put all items equal to the partitioning item in place.

\[
\begin{array}{cccccccc}
C & C & C & & & & &
\end{array}
\quad
\begin{array}{cccccccc}
\end{array}
\]
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 \( g_t \).

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

**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 algorithm

3-way partitioning.
• Let $v$ be partitioning item $a[lo]$.
• Scan $i$ from left to right.
  - $a[i]$ less than $v$: exchange $a[lt]$ with $a[i]$ 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 $i$

Most of the right properties.
• In-place.
• Not much code.
• Linear time if keys are all equal.
Dijkstra's 3-way partitioning: demo
Dijkstra's 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

equal to partitioning element
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

$$\lg \left( \frac{N!}{x_1! \cdot x_2! \cdot \ldots \cdot x_n!} \right) \sim - \sum_{i=1}^{n} x_i \lg \frac{x_i}{N}$$

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.
- selection
- duplicate keys
- comparators
- 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.
• 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 an array of `Comparable` or of any primitive type.
- Uses tuned quicksort for primitive types; tuned mergesort for objects.

Q. Why use different algorithms, depending on type?
War story (C qsort function)

AT&T Bell Labs (1991). Allan Wilks and Rick Becker discovered that a `qsort()` call that should have taken a few minutes was consuming hours of CPU time.

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.
Basic algorithm = quicksort.

- Cutoff to insertion sort for small subarrays.
- Partitioning scheme: Bentley-McIlroy 3-way partitioning. [ahead]
- Partitioning item.
  - small arrays: middle entry
  - medium arrays: median of 3
  - large arrays: Tukey’s ninther [next slide]

Now widely used. C, C++, Java, ....
Tukey's ninther

Tukey's ninther. Median of the median of 3 samples, each of 3 entries.
- Approximates the median of 9.
- Uses at most 12 compares.

Q. Why use Tukey's ninther?
A. Better partitioning than random shuffle and less costly.
Bentley-McIlroy 3-way partitioning

Partition items into *four* parts:
- No larger entries to left of $i$.
- No smaller entries to right of $j$.
- Equal entries to left of $p$.
- Equal entries to right of $q$.

Afterwards, swap equal keys into center.

All the right properties.
- In-place.
- Not much code.
- Linear time if keys are all equal.
- Small overhead if no equal keys.
Achilles heel in Bentley-McIlroy implementation (Java system sort)

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

A. No: a killer input.
   • **Overflows** function call stack in Java and crashes program.
   • **Would take quadratic time** if it didn’t crash first.

```bash
% more 250000.txt
0
218750
222662
11
166672
247070
83339
...
```

```bash
% java IntegerSort 250000 < 250000.txt
Exception in thread "main"
java.lang.StackOverflowError
   at java.util.Arrays.sort1(Arrays.java:562)
   at java.util.Arrays.sort1(Arrays.java:606)
   at java.util.Arrays.sort1(Arrays.java:608)
   at java.util.Arrays.sort1(Arrays.java:608)
   at java.util.Arrays.sort1(Arrays.java:608)
   ...
```

250,000 integers between 0 and 250,000
Java’s sorting library crashes, even if you give it as much stack space as Windows allows

more disastrous consequences in C
Achilles heel in Bentley-McIlroy implementation (Java system sort)

McIlroy's devious idea. [A Killer Adversary for Quicksort]
• Construct malicious input on the fly while running system quicksort, in response to the sequence of keys compared.
• Make partitioning item compare low against all items not seen during selection of partitioning item (but don't commit to their relative order).
• Not hard to identify partitioning item.

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

Good news. Attack is not effective if sort() shuffles input array.

Q. Why do you think Arrays.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.

**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.
System sort: Which algorithm to use?

Applications have diverse attributes.
- Stable?
- Parallel?
- Deterministic?
- Keys all distinct?
- Multiple key types?
- Linked list or arrays?
- Large or small items?
- 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.
## Sorting Summary

<table>
<thead>
<tr>
<th></th>
<th>inplace?</th>
<th>stable?</th>
<th>worst</th>
<th>average</th>
<th>best</th>
<th>remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>selection</td>
<td>x</td>
<td></td>
<td>$N^2/2$</td>
<td>$N^2/2$</td>
<td>$N^2/2$</td>
<td>$N$ exchanges</td>
</tr>
<tr>
<td>insertion</td>
<td>x</td>
<td>x</td>
<td>$N^2/2$</td>
<td>$N^2/4$</td>
<td>$N$</td>
<td>use for small $N$ or partially ordered</td>
</tr>
<tr>
<td>shell</td>
<td>x</td>
<td></td>
<td>?</td>
<td>?</td>
<td>$N$</td>
<td>tight code, subquadratic</td>
</tr>
<tr>
<td>merge</td>
<td>x</td>
<td></td>
<td>$N \log N$</td>
<td>$N \log N$</td>
<td>$N \log N$</td>
<td>$N \log N$ guarantee, stable</td>
</tr>
<tr>
<td>quick</td>
<td>x</td>
<td></td>
<td>$N^2/2$</td>
<td>$2N \log N$</td>
<td>$N \log N$</td>
<td>$N \log N$ probabilistic guarantee, fastest in practice</td>
</tr>
<tr>
<td>3-way quick</td>
<td>x</td>
<td></td>
<td>$N^2/2$</td>
<td>$2N \log N$</td>
<td>$N$</td>
<td>improves quicksort in presence of duplicate keys</td>
</tr>
<tr>
<td>???</td>
<td>x</td>
<td>x</td>
<td>$N \log N$</td>
<td>$N \log N$</td>
<td>$N \log N$</td>
<td>holy sorting grail</td>
</tr>
</tbody>
</table>
### Which sorting algorithm?

<table>
<thead>
<tr>
<th>lifo</th>
<th>find</th>
<th>data</th>
<th>data</th>
<th>data</th>
<th>data</th>
<th>hash</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>fifo</td>
<td>fifo</td>
<td>fifo</td>
<td>fifo</td>
<td>exch</td>
<td>fifo</td>
<td>fifo</td>
<td>exch</td>
</tr>
<tr>
<td>data</td>
<td>data</td>
<td>find</td>
<td>find</td>
<td>fifo</td>
<td>lifo</td>
<td>data</td>
<td>fifo</td>
</tr>
<tr>
<td>type</td>
<td>exch</td>
<td>hash</td>
<td>hash</td>
<td>find</td>
<td>type</td>
<td>link</td>
<td>find</td>
</tr>
<tr>
<td>hash</td>
<td>hash</td>
<td>heap</td>
<td>heap</td>
<td>hash</td>
<td>hash</td>
<td>leaf</td>
<td>hash</td>
</tr>
<tr>
<td>heap</td>
<td>heap</td>
<td>lifo</td>
<td>lifo</td>
<td>heap</td>
<td>heap</td>
<td>heap</td>
<td>heap</td>
</tr>
<tr>
<td>sort</td>
<td>less</td>
<td>link</td>
<td>link</td>
<td>leaf</td>
<td>link</td>
<td>exch</td>
<td>leaf</td>
</tr>
<tr>
<td>link</td>
<td>left</td>
<td>list</td>
<td>list</td>
<td>left</td>
<td>sort</td>
<td>node</td>
<td>left</td>
</tr>
<tr>
<td>list</td>
<td>leaf</td>
<td>push</td>
<td>push</td>
<td>less</td>
<td>find</td>
<td>lifo</td>
<td>less</td>
</tr>
<tr>
<td>push</td>
<td>lifo</td>
<td>root</td>
<td>root</td>
<td>lifo</td>
<td>list</td>
<td>left</td>
<td>lifo</td>
</tr>
<tr>
<td>find</td>
<td>push</td>
<td>sort</td>
<td>sort</td>
<td>link</td>
<td>push</td>
<td>find</td>
<td>link</td>
</tr>
<tr>
<td>root</td>
<td>root</td>
<td>type</td>
<td>type</td>
<td>list</td>
<td>root</td>
<td>path</td>
<td>list</td>
</tr>
<tr>
<td>leaf</td>
<td>list</td>
<td>leaf</td>
<td>leaf</td>
<td>sort</td>
<td>leaf</td>
<td>list</td>
<td>next</td>
</tr>
<tr>
<td>tree</td>
<td>tree</td>
<td>left</td>
<td>tree</td>
<td>null</td>
<td>tree</td>
<td>null</td>
<td>next</td>
</tr>
<tr>
<td>null</td>
<td>null</td>
<td>node</td>
<td>null</td>
<td>null</td>
<td>path</td>
<td>less</td>
<td>node</td>
</tr>
<tr>
<td>path</td>
<td>path</td>
<td>null</td>
<td>path</td>
<td>path</td>
<td>tree</td>
<td>root</td>
<td>path</td>
</tr>
<tr>
<td>node</td>
<td>node</td>
<td>path</td>
<td>node</td>
<td>node</td>
<td>exch</td>
<td>sink</td>
<td>push</td>
</tr>
<tr>
<td>left</td>
<td>link</td>
<td>tree</td>
<td>left</td>
<td>type</td>
<td>left</td>
<td>swim</td>
<td>root</td>
</tr>
<tr>
<td>less</td>
<td>sort</td>
<td>exch</td>
<td>less</td>
<td>root</td>
<td>less</td>
<td>null</td>
<td>sink</td>
</tr>
<tr>
<td>exch</td>
<td>type</td>
<td>less</td>
<td>exch</td>
<td>push</td>
<td>node</td>
<td>sort</td>
<td>sort</td>
</tr>
<tr>
<td>sink</td>
<td>sink</td>
<td>next</td>
<td>sink</td>
<td>sink</td>
<td>next</td>
<td>type</td>
<td>swap</td>
</tr>
<tr>
<td>swim</td>
<td>swim</td>
<td>sink</td>
<td>swim</td>
<td>swim</td>
<td>sink</td>
<td>tree</td>
<td>swim</td>
</tr>
<tr>
<td>next</td>
<td>next</td>
<td>swap</td>
<td>next</td>
<td>next</td>
<td>swap</td>
<td>push</td>
<td>tree</td>
</tr>
<tr>
<td>swap</td>
<td>swap</td>
<td>swim</td>
<td>swap</td>
<td>swap</td>
<td>swim</td>
<td>swap</td>
<td>type</td>
</tr>
</tbody>
</table>

## Which sorting algorithm?

<table>
<thead>
<tr>
<th>lifo</th>
<th>find</th>
<th>data</th>
<th>data</th>
<th>data</th>
<th>data</th>
<th>hash</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>fifo</td>
<td>fifo</td>
<td>fifo</td>
<td>fifo</td>
<td>exch</td>
<td>fifo</td>
<td>fifo</td>
<td>exch</td>
</tr>
<tr>
<td>data</td>
<td>data</td>
<td>find</td>
<td>find</td>
<td>fifo</td>
<td>lifo</td>
<td>data</td>
<td>fifo</td>
</tr>
<tr>
<td>type</td>
<td>exch</td>
<td>hash</td>
<td>hash</td>
<td>find</td>
<td>type</td>
<td>link</td>
<td>find</td>
</tr>
<tr>
<td>hash</td>
<td>hash</td>
<td>heap</td>
<td>heap</td>
<td>hash</td>
<td>hash</td>
<td>leaf</td>
<td>hash</td>
</tr>
<tr>
<td>heap</td>
<td>heap</td>
<td>lifo</td>
<td>lifo</td>
<td>heap</td>
<td>heap</td>
<td>heap</td>
<td>heap</td>
</tr>
<tr>
<td>sort</td>
<td>less</td>
<td>link</td>
<td>link</td>
<td>leaf</td>
<td>link</td>
<td>exch</td>
<td>leaf</td>
</tr>
<tr>
<td>link</td>
<td>left</td>
<td>list</td>
<td>list</td>
<td>left</td>
<td>sort</td>
<td>node</td>
<td>left</td>
</tr>
<tr>
<td>list</td>
<td>leaf</td>
<td>push</td>
<td>push</td>
<td>less</td>
<td>find</td>
<td>lifo</td>
<td>less</td>
</tr>
<tr>
<td>push</td>
<td>lifo</td>
<td>root</td>
<td>root</td>
<td>lifo</td>
<td>list</td>
<td>left</td>
<td>lifo</td>
</tr>
<tr>
<td>find</td>
<td>push</td>
<td>sort</td>
<td>sort</td>
<td>link</td>
<td>push</td>
<td>find</td>
<td>link</td>
</tr>
<tr>
<td>root</td>
<td>root</td>
<td>type</td>
<td>type</td>
<td>list</td>
<td>root</td>
<td>path</td>
<td>list</td>
</tr>
<tr>
<td>leaf</td>
<td>list</td>
<td>leaf</td>
<td>leaf</td>
<td>sort</td>
<td>leaf</td>
<td>list</td>
<td>next</td>
</tr>
<tr>
<td>tree</td>
<td>tree</td>
<td>left</td>
<td>tree</td>
<td>null</td>
<td>next</td>
<td>node</td>
<td>path</td>
</tr>
<tr>
<td>null</td>
<td>null</td>
<td>node</td>
<td>node</td>
<td>null</td>
<td>path</td>
<td>less</td>
<td>null</td>
</tr>
<tr>
<td>path</td>
<td>path</td>
<td>null</td>
<td>path</td>
<td>path</td>
<td>root</td>
<td>null</td>
<td>sink</td>
</tr>
<tr>
<td>node</td>
<td>node</td>
<td>path</td>
<td>node</td>
<td>node</td>
<td>exch</td>
<td>sink</td>
<td>push</td>
</tr>
<tr>
<td>left</td>
<td>link</td>
<td>tree</td>
<td>left</td>
<td>type</td>
<td>left</td>
<td>swim</td>
<td>root</td>
</tr>
<tr>
<td>less</td>
<td>sort</td>
<td>exch</td>
<td>less</td>
<td>root</td>
<td>less</td>
<td>null</td>
<td>sink</td>
</tr>
<tr>
<td>exch</td>
<td>type</td>
<td>less</td>
<td>exch</td>
<td>push</td>
<td>node</td>
<td>sort</td>
<td>sort</td>
</tr>
<tr>
<td>sink</td>
<td>sink</td>
<td>next</td>
<td>sink</td>
<td>sink</td>
<td>next</td>
<td>type</td>
<td>swap</td>
</tr>
<tr>
<td>swim</td>
<td>swim</td>
<td>sink</td>
<td>swim</td>
<td>swim</td>
<td>sink</td>
<td>tree</td>
<td>swim</td>
</tr>
<tr>
<td>next</td>
<td>next</td>
<td>swap</td>
<td>next</td>
<td>next</td>
<td>swap</td>
<td>push</td>
<td>tree</td>
</tr>
<tr>
<td>swap</td>
<td>swap</td>
<td>swim</td>
<td>swap</td>
<td>swap</td>
<td>swim</td>
<td>swap</td>
<td>type</td>
</tr>
</tbody>
</table>

| original | quicksort | mergesort | insertion | selection | merge BU | shellsort | sorted |