2.3 Quicksort

- quicksort
- selection
- duplicate keys
- system sorts

partitioning

Quicksort

array
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
- selection
- duplicate keys
- system sorts
Quicksort

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 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;  // find item on left to swap

        while (less(a[lo], a[--j]))
            if (j == lo) break;  // find item on right to swap

        if (i >= j) break;     // check if pointers cross
        exch(a, i, j);          // swap
    }
    exch(a, lo, j);          // swap with partitioning item
    return j;                // return index of item now known to be in place
}
public class Quick
{
    private static int partition(Comparable[] a, int lo, int hi)
    {
        /* see previous slide */
    }

    public static void sort(Comparable[] a)
    {
        StdRandom.shuffle(a);
        sort(a, 0, a.length - 1);
    }

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

<table>
<thead>
<tr>
<th>lo</th>
<th>j</th>
<th>hi</th>
</tr>
</thead>
<tbody>
<tr>
<td>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>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td></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></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></td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>15</td>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>

Quicksort trace (array contents after each partition)

---

Quicksort example:

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

Quicksort trace

**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 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 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 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>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 2.8 hours 317 years</td>
<td>instant 1 second 18 min</td>
<td>instant 0.6 sec 12 min</td>
</tr>
<tr>
<td>super</td>
<td>instant 1 second 1 week</td>
<td>instant instant instant</td>
<td>instant instant 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 $\sim N \lg N$. 

\[
\begin{array}{cccccccccccccc}
& & & & & & & & & & & & & & \\
lo & j & hi & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\
\hline
\text{initial values} & H & A & C & B & F & E & G & D & L & I & K & J & N & M & O \\
\text{random shuffle} & H & A & C & B & F & E & G & D & L & I & K & J & N & M & O \\
0 & 7 & 14 & D & A & C & B & F & E & G & H & L & I & K & J & N & M & O \\
0 & 3 & 6 & B & A & C & D & F & E & G & H & L & I & K & J & N & M & O \\
0 & 1 & 2 & A & B & C & D & F & E & G & H & L & I & K & J & N & M & O \\
0 & 2 & 2 & A & B & C & D & F & E & G & H & L & I & K & J & N & M & O \\
6 & 6 & 1 & A & B & C & D & E & F & G & H & L & I & K & J & N & M & O \\
\end{array}
\]
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 $\sim \frac{1}{3} N \ln N$).

**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:

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

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

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

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

- 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 $N$.
**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 $2 / |j - i + 1|$.

![Binary Search Tree Representation]

3 and 6 are compared (when 3 is partition)
1 and 6 are not compared (because 3 is partition)
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 $\frac{2}{|j - i + 1|}$.

- 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.
- Can delay insertion sort until 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

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

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

```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: practical improvements

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

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

Optimize parameters.
• Median-of-3 (random) elements.
• Cutoff to insertion sort for \( \approx 10 \) elements.
Quicksort with median-of-3 and cutoff to insertion sort: visualization
- quicksort
- selection
- duplicate keys
- system sorts
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. 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 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 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:
  \[ C_N = 2N + k \ln \left( \frac{N}{k} \right) + (N - k) \ln \left( \frac{N}{N - k} \right) \]

Ex. \((2 + 2 \ln 2)N\) compares 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

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

**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 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 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 keys equal to the partitioning element on one side.

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

```
B A A B A B B B C C C       A A A A A A A A A A A
```

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

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

```
B A A B A B C C B C B       A A A A A A A A A A A
```

**Desirable.** Put all keys equal to the partitioning element in place.

```
A A A B B B B B C C C       A A A A A A A A A A A
```
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 \( \gt \).

![Diagram of 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 element $a[lo]$.
- 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 $i$

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)
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

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

compares in the worst case.

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

Pf. [beyond scope of course]

Bottom line. Randomized quicksort 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.

```java
import java.util.Arrays;

public class StringSort
{
    public static void main(String[] args)
    {
       String[] a = StdIn.readAll().split("\s+"),
       Arrays.sort(a);
       for (int i = 0; i < N; i++) 
          StdOut.println(a[i]);
    }
}
```

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.

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: optimized 3-way partitioning.
• Partitioning element.
  - small arrays: middle element
  - medium arrays: median of 3
  - large arrays: Tukey's ninther [median of 3 medians of 3]

Now widely used. C, C++, Java, ....
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.

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

% 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)
   ...

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
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 element compare low against all keys not seen during selection of partitioning element (but don't commit to their relative order).
- Not hard to identify partitioning element.

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 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.
## Sorting summary

<table>
<thead>
<tr>
<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>$N^2/2$</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>$N$ use for small $N$ or partially ordered</td>
</tr>
<tr>
<td>shell</td>
<td>x</td>
<td>?</td>
<td>?</td>
<td>$N$</td>
<td>tight code, subquadratic</td>
</tr>
<tr>
<td>merge</td>
<td>x</td>
<td>x</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>$2N \ln N$</td>
<td>$N \log N$ $N \log N$ probabilistic guarantee fastest in practice</td>
</tr>
<tr>
<td>3-way quick</td>
<td>x</td>
<td></td>
<td>$N^2/2$</td>
<td>$2N \ln N$</td>
<td>$N$ 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>$N \log N$ holy sorting grail</td>
</tr>
</tbody>
</table>
Which sorting algorithm?

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>lifo</td>
<td>find</td>
<td>data</td>
<td>data</td>
<td>data</td>
<td>data</td>
<td>hash</td>
<td>data</td>
<td></td>
<td></td>
</tr>
<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>
<td></td>
<td></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>
<td></td>
<td></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>
<td></td>
<td></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>
<td></td>
<td></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>
<td></td>
<td></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>
<td></td>
<td></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>
<td></td>
<td></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>
<td></td>
<td></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>
<td></td>
<td></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>
<td></td>
<td></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>
<td></td>
<td></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>
<td></td>
<td></td>
</tr>
<tr>
<td>tree</td>
<td>tree</td>
<td>left</td>
<td>tree</td>
<td>tree</td>
<td>null</td>
<td>next</td>
<td>node</td>
<td></td>
<td></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>null</td>
<td></td>
<td></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>
<td></td>
<td></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>
<td></td>
<td></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>
<td></td>
<td></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>
<td></td>
<td></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>
<td></td>
<td></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>
<td></td>
<td></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>
<td></td>
<td></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>
<td></td>
<td></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>
<td></td>
<td></td>
</tr>
</tbody>
</table>
› selection
› duplicate keys
› comparators
› system sorts
› application: convex hull
**Convex hull**

The **convex hull** of a set of $N$ points is the smallest convex set containing all the points.

**Convex hull output.** Sequence of extreme points in counterclockwise order.

**Non-degeneracy assumption.** No three points on a line.
Convex hull: brute-force algorithm

Observation 1. Edges of convex hull of $P$ connect pairs of points in $P$.
Observation 2. Edge $p\rightarrow q$ is on convex hull if all other points are ccw of $pq$.

$O(N^3)$ algorithm. For all pairs of points $p$ and $q$:
- Compute $\text{Point.ccw}(p, q, x)$ for all other points $x$.
- $p\rightarrow q$ is on hull if all values are positive.

Degeneracies. Three (or more) points on a line.
Graham scan

- Choose point $p$ with smallest $y$-coordinate (break ties by $x$-coordinate).
- Sort points by polar angle with respect to $p$.
- Consider points in order, and discard unless they would create a ccw turn.
Graham scan: demo

http://www.cs.princeton.edu/courses/archive/fall08/cos226/demo/ah/GrahamScan.html
Graham scan: demo

http://www.cs.princeton.edu/courses/archive/fall08/cos226/demo/ah/GrahamScan.html
Graham scan: implementation

Simplifying assumptions. No three points on a line; at least 3 points.

```java
Stack<Point> hull = new Stack<Point>();

Quick.sort(p, Point.BY_Y);
Quick.sort(p, p[0].BY_POLAR_ANGLE);

hull.push(p[0]);
hull.push(p[1]);

for (int i = 2; i < N; i++) {
    Point top = hull.pop();
    while (Point.ccw(top, hull.peek(), p[i]) <= 0)
        top = hull.pop();
    hull.push(top);
    hull.push(p[i]);
}
```

**Running time.** $N \log N$ for sorting and linear for rest.