Efficient Sorting Algorithms

- mergesort
- sorting complexity
- quicksort
- animations

**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.**
- Divide array into two halves.
- Recursively sort each half.
- Merge two halves.

---

Mergesort overview

**input**

<table>
<thead>
<tr>
<th>M E R G E S O R T E X A M P L E</th>
</tr>
</thead>
<tbody>
<tr>
<td>E E G M O R R S T E X A M P L E</td>
</tr>
</tbody>
</table>

**sort left half**

<table>
<thead>
<tr>
<th>E E G M O R R S</th>
</tr>
</thead>
</table>

**sort right half**

<table>
<thead>
<tr>
<th>A E E L M P T X</th>
</tr>
</thead>
</table>

**merge results**

| A E E E E G L M M O P R R S T X |
Mergesort trace

Goal. Combine two sorted subarrays into a sorted whole.

Q. How to merge efficiently?
A. Use an auxiliary array.

Mergesort: Java implementation

```java
public class Merge
{
   private static Comparable[] aux;

   private static void merge(Comparable[] a, int lo, int m, int hi)
   {
      for (int i = lo; i <= m; i++)
         aux[i] = a[i];
      for (int j = m+1; j <= hi; j++)
         aux[j] = a[hi-j+m+1];

      int i = lo, j = m+1;
      for (int k = lo; k <= hi; k++)
         if (less(aux[j], aux[i])) a[k] = aux[j--];
         else a[k] = aux[i++];
   }

   private static void sort(Comparable[] a)
   {
      aux = new Comparable[a.length];
      sort(a, 0, a.length - 1);
   }
}
```

Merging: Java implementation

```java
public static void sort(Comparable[] a)
{
   aux = new Comparable[a.length];
   sort(a, 0, a.length - 1);
}
```
Mergesort visualization

Running time estimates:
• Home pc executes \(10^8\) comparisons/second.
• Supercomputer executes \(10^{12}\) comparisons/second.

Lesson. Good algorithms are better than supercomputers.

Mergesort: mathematical analysis

Proposition. Mergesort uses \(N \lg N\) compares to sort any array of size \(N\).

Def. \(T(N)\) = number of compares to mergesort an array of size \(N\).

\[
T(N) = T(N/2) + T(N/2) + N
\]

Mergesort recurrence. \(T(N) = 2T(N/2) + N\) for \(N > 1\), with \(T(1) = 0\).

• Not quite right for odd \(N\).
• Same recurrence holds for many divide-and-conquer algorithms.

Solution. \(T(N) \sim N \lg N\).

• For simplicity, we’ll prove when \(N\) is a power of 2.
• True for all \(N\). [see COS 340]

Mergesort recurrence: proof 1

Proposition. If \(N\) is a power of 2, then \(T(N) = N \lg N\).

Pf.
**Mergesort recurrence: proof 2**

**Mergesort recurrence.** \( T(N) = 2 T(N/2) + N \) for \( N > 1 \), with \( T(1) = 0 \).

**Proposition.** If \( N \) is a power of 2, then \( T(N) = N \log_2 N \).

**Pf.**

\[
T(N) = 2 T(N/2) + N \\
\frac{T(N)}{N} = 2 \frac{T(N/2)}{N} + 1 \\
= \frac{T(N/2)}{N/2} + 1 \\
= \frac{T(N/4)}{(N/4)/2} + 1 \\
\vdots \\
= \frac{T(N/N)}{N/N} + 1 + \ldots + 1 \\
= \log N
\]

given

divide both sides by \( N \)

algebra

apply to first term

apply to first term again

stop applying, \( T(1) = 0 \)

---

**Mergesort recurrence: proof 3**

**Mergesort recurrence.** \( T(N) = 2 T(N/2) + N \) for \( N > 1 \), with \( T(1) = 0 \).

**Proposition.** If \( N \) is a power of 2, then \( T(N) = 2 N \log (2N) \).

**Pf.** [by induction on \( N \)]

- **Base case:** \( N = 1 \).
- **Inductive hypothesis:** \( T(N) = N \log N \).
- **Goal:** show that \( T(2N) = 2N \log (2N) \).

\[
T(2N) = 2 T(N) + 2N \\
= 2 N \log N + 2N \\
= 2 N (\log (2N) - 1) + 2N \\
= 2 N \log (2N)
\]

given

inductive hypothesis

algebra

QED

---

**Mergesort analysis: memory**

**Proposition G.** Mergesort uses extra space proportional to \( N \).

**Pf.** The array aux[] needs to be of size \( N \) for the last merge.

**Def.** A sorting algorithm is **in-place** if it uses \( O(\log N) \) extra memory.

**Ex.** Insertion sort, selection sort, shellsort.

**Challenge for the bored.** In-place merge. [Kronrud, 1969]

---

**Mergesort: practical improvements**

**Use insertion sort for small subarrays.**

- Mergesort has too much overhead for tiny subarrays.
- Cutoff to insertion sort for \( \approx 7 \) elements.

**Stop if already sorted.**

- Is biggest element in first half \( \leq \) smallest element in second half?
- Helps for nearly ordered lists.

**Eliminate the copy to the auxiliary array.** Save time (but not space) by switching the role of the input and auxiliary array in each recursive call.

**Ex.** See Program 8.4 or Arrays.sort().
Bottom-up mergesort

Basic plan.
- Pass through array, merging subarrays of size 1.
- Repeat for subarrays of size 2, 4, 8, 16, ...

Bottom line. No recursion needed!

Bottom-up mergesort: visualization

Bottom-up mergesort: Java implementation

```java
public class MergeBU {
    private static Comparable[] aux;
    private static void merge(Comparable[] a, int lo, int m, int hi) {
        /* as before */
    }
    public static void sort(Comparable[] a) {
        int N = a.length;
        aux = new Comparable[N];
        for (int m = 1; m < N; m = m+m)
            for (int i = 0; i < N-m; i += m+m)
                merge(a, i, i+m, Math.min(i+m+m, N));
    }
}
```

Bottom line. Concise industrial-strength code, if you have the space.
**Complexity of sorting**

**Computational complexity.** Framework to study efficiency of algorithms for solving a particular problem X.

**Machine model.** Focus on fundamental operations.

**Upper bound.** Cost guarantee provided by some algorithm for X.

**Lower bound.** Proven limit on cost guarantee of any algorithm for X.

**Optimal algorithm.** Algorithm with best cost guarantee for X.

**Example: sorting.**
- Machine model = # compares.
- Upper bound = $N \log_2 N$ from mergesort.
- Lower bound = $N \log_2 N$ ?
- Optimal algorithm = mergesort ?

---

**Compare-based lower bound for sorting**

**Proposition.** Any compare-based sorting algorithm must use more than $N \log_2 N - 1.44 \cdot N$ comparisons in the worst-case.

**Pf.**
- Assume input consists of $N$ distinct values $a_1$ through $a_N$.
- Worst case dictated by height $h$ of decision tree.
- Binary tree of height $h$ has at most $2^h$ leaves.
- $N!$ different orderings $\Rightarrow$ at least $N!$ leaves.

\[
2^h \geq N! \\
h \geq \log_2 N! \\
\geq \log_2 (N/e) \cdot N \\
= N \log_2 N - N \log_2 e \\
\geq N \log_2 N - 1.44 \cdot N
\]
Complexity of sorting

Machine model. Focus on fundamental operations.
Upper bound. Cost guarantee provided by some algorithm for X.
Lower bound. Proven limit on cost guarantee of any algorithm for X.
Optimal algorithm. Algorithm with best cost guarantee for X.

Example: sorting.
• Machine model = # compares.
• Upper bound = \(N \log N\) from mergesort.
• Lower bound = \(N \log N - 1.44N\).
• Optimal algorithm = mergesort.

First goal of algorithm design: optimal algorithms.

Complexity results in context

Other operations? Mergesort optimality is only about number of compares.

Space?
• Mergesort is not optimal with respect to space usage.
• Insertion sort, selection sort, and shellsort are space-optimal.
• Is there an algorithm that is both time- and space-optimal?

Lessons. Use theory as a guide.
Ex. Don’t try to design sorting algorithm that uses \(\frac{1}{2} N \log N\) compares.

Complexity results in context (continued)

Lower bound may not hold if the algorithm has information about
• The key values.
• Their initial arrangement.

Partially ordered arrays. Depending on the initial order of the input,
we may not need \(N \log N\) compares.

Duplicate keys. Depending on the input distribution of duplicates,
we may not need \(N \log N\) compares.

Digital properties of keys. We can use digit/character comparisons instead
of key comparisons for numbers and strings.

insertion sort requires \(O(N)\) compares on
an already sorted array

stay tuned for 3-way quicksort

stay tuned for radix sorts
Quicksort

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

Sir Charles Antony Richard Hoare
1980 Turing Award

Quicksort

<table>
<thead>
<tr>
<th>Input</th>
<th>Quicksort Example</th>
<th>Shuffle</th>
<th>Partition</th>
<th>Sort Left</th>
<th>Sort Right</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERATESLPUIMQ CXOK</td>
<td>ECAIEKLPUTMRXOS</td>
<td>not greater</td>
<td>partitioning element</td>
<td>ACEEIKLPMOPQRSTUX</td>
<td>ACEEIKLPMOPQRSTUX</td>
<td></td>
</tr>
</tbody>
</table>

Quicksort partitioning

Basic plan.
• Scan from left for an item that belongs on the right.
• Scan from right for item item that belongs on the left.
• Exchange.
• Continue until pointers cross.

Quicksort partitioning overview

Initial values
-1 15 E R A T E S L P U I M Q C X O K

Scan left, scan right
1 12 E R A T E S L P U I M Q C X O K

Exchange
1 12 E C A I E K L R X O S

Scan left, scan right
3 9 E C A I E K L P U I M Q R X O K

Exchange
3 9 E C A I E K L P U I M Q R X O K

Scan left, scan right
5 5 E C A I E K L P U I M Q R X O K

Final exchange
5 5 E C A I E K L P U I M Q R X O S

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

Partitioning trace (array contents before and after each exchange)

Quicksort: Java code for partitioning

```java
private static int partition(Comparable[] a, int lo, int hi)
{
   int i = lo - 1;
   int j = hi;
   while(true)
   {
      while (less(a[++i], a[hi]))
         if (i == hi) break;
      while (less(a[hi], a[--j]))
         if (j == lo) break;
      if (i >= j) break;
      exch(a, i, j);
   }
   exch(a, i, hi);
   return i;
}
```

Quicksort: Java implementation

```java
public class Quick
{
   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 i = partition(a, lo, hi);
      sort(a, lo, i-1);
      sort(a, i+1, hi);
   }
}
```
**Quicksort trace**

<table>
<thead>
<tr>
<th>lo</th>
<th>i</th>
<th>hi</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
</tbody>
</table>

**Quicksort: implementation details**

**Partitioning in-place.** Using a spare array makes partitioning easier, 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 \((i == hi)\) test is redundant, but the \((j == lo)\) test is not.

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

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

**Running time estimates:**
- Home pc executes \(10^8\) comparisons/second.
- Supercomputer executes \(10^{12}\) comparisons/second.

**Lesson 1.** Good algorithms are better than supercomputers.

**Lesson 2.** Great algorithms are better than good ones.

**QuickSort: empirical analysis**

Running time estimates:
- Home pc executes \(10^8\) comparisons/second.
- Supercomputer executes \(10^{12}\) comparisons/second.

<table>
<thead>
<tr>
<th>computer</th>
<th>thousand</th>
<th>million</th>
<th>billion</th>
<th>thousand</th>
<th>million</th>
<th>billion</th>
<th>thousand</th>
<th>million</th>
<th>billion</th>
</tr>
</thead>
<tbody>
<tr>
<td>home</td>
<td>instant</td>
<td>2.8 hours</td>
<td>317 years</td>
<td>instant</td>
<td>1 second</td>
<td>18 min</td>
<td>instant</td>
<td>0.3 sec</td>
<td>6 min</td>
</tr>
<tr>
<td>super</td>
<td>instant</td>
<td>1 second</td>
<td>1 week</td>
<td>instant</td>
<td>instant</td>
<td>instant</td>
<td>instant</td>
<td>instant</td>
<td>instant</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 \(\sim \frac{1}{2} N \ln N\)).

**Pf.** \(C_N\) satisfies the recurrence \(C_0 = C_1 = 0\) and for \(N > 2\):
\[
C_N = (N + 1) + (C_0 + C_1 + \ldots + C_{N-1})/N + (C_{N-2} + C_{N-3} + \ldots + C_0)/N
\]

* 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 \(N-1\):
\[
NC_N - (N - 1)C_N = 2N - 2C_{N-1}
\]

* Rearrange terms and divide by \(N(N+1)\):
\[
C_N / (N+1) = (C_{N-1} / N) + 2 / (N + 1)
\]
Quicksort: average-case analysis

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

- Repeatedly apply above equation:
  \[ C_N / (N+1) = C_{N+1} / N + 2 / (N+1) \]
  \[ = C_{N+2} / (N+1) + 2/N + 2/(N+1) \]
  \[ = C_{N+3} / (N+2) + 2/(N+1) + 2/N + 2/(N+1) \]
  \[ = 2 \left( \frac{1}{1+1/2} + \frac{1}{1+1/3} + \ldots + \frac{1}{N+1} \right) \]

- Approximate by an integral:
  \[ C_N \approx 2(N+1) \int_1^N \frac{dx}{x} \]
  \[ \approx 2(N+1) \ln N \approx 1.39 N \lg N \]

Quicksort: practical improvements

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

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

Optimize parameters.
- Median-of-3 random elements.
- Cutoff to insertion sort for \( \approx 10 \) elements.

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

Quicksort: summary of performance characteristics

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

Average case. Number of compares is \( \approx 1.39 N \lg N \).
- \( 39\% \) 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 input:
- Is sorted or reverse sorted
- Has many duplicates (even if randomized!) [stay tuned]

Quicksort with cutoff to insertion sort: visualization

input
result of first partition
both subfiles partially sorted
result
left subfile partially sorted
right subfile partially sorted
small

~ 12/7 N lg N comparisons

~ guarantees O(log N) stack size

 guarantees O(log N) stack size

 guarantees O(log N) stack size
Mergesort animation

Mergesort animation

Boton-up mergesort animation