Lecture 3: Efficient Sorts

Mergesort
Quicksort
Analysis of Algorithms

Mergesort and Quicksort

Two great sorting algorithms.
- Full scientific understanding of their properties has enabled us to hammer them into practical system sorts.
- Occupies a prominent place in world’s computational infrastructure.
- Quicksort honored as one of top 10 algorithms for science and engineering of 20th century.

Mergesort.
- Java Arrays sort for type Object.
- Java Collections sort.
- Perl stable, Python stable.

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

Sorting Applications

- Sort a list of names.
- Organize an MP3 library.
- Display Google PageRank results.
- 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.
- Simulate a system of particles.
- Book recommendations on Amazon.
- Load balancing on a parallel computer.

Estimating the Running Time

Total running time is sum of cost $\times$ frequency for all of the basic ops.
- Cost depends on machine, compiler.
- Frequency depends on algorithm, input.

Cost for sorting.
- $A = \#$ function calls.
- $B = \#$ exchanges.
- $C = \#$ comparisons.
- Cost on a typical machine $= 35A + 11B + 4C$.

Frequency of sorting ops.
- $N = \#$ elements to sort.
- Selection sort: $A = 1, B = N-1, C = N(N-1) / 2$. 

Donald Knuth
Estimating the Running Time

An easier alternative.
(i) Analyze asymptotic growth as a function of input size N.
(ii) For medium N, run and measure time.
(iii) For large N, use (i) and (ii) to predict time.

Asymptotic growth rates.
- Estimate as a function of input size N.
  - $N$, $N \log N$, $N^2$, $N^3$, $2^N$, $N!$
- Ignore lower order terms and leading coefficients.
  - Ex. $6N^3 + 17N^2 + 56$ is asymptotically proportional to $N^3$

Big Oh Notation

Big Theta, Oh, and Omega notation.
- $O(N^3)$ means \( \{ N^3, 17N^3, N^3 + 17N^{1.5} + 3N, \ldots \} \)
  - ignore lower order terms and leading coefficients
- $O(N^2)$ means \( \{ N^2, 17N^2, N^2 + 17N^{1.5} + 3N, N^{1.5}, 100N, \ldots \} \)
  - $O(N^2)$ and smaller
  - use for upper bounds
- $\Omega(N^2)$ means \( \{ N^2, 17N^2, N^2 + 17N^{1.5} + 3N, N^3, 100N^3, \ldots \} \)
  - $O(N^2)$ and larger
  - use for lower bounds

Never say: insertion sort makes at least $O(N^2)$ comparisons.

Estimating the Running Time

Insertion sort is quadratic.
- $N^2 / 4 - N / 4$ comparisons on average.
- $\Theta(N^2)$.

On arizona: 1 second for $N = 10,000$.
- How long for $N = 100,000$? 100 seconds (100 times as long)
- $N = 1$ million? 2.78 hours (another factor of 100)
- $N = 1$ billion? 317 years (another factor of $10^6$)
- $N = 1$ trillion?

Why It Matters

<table>
<thead>
<tr>
<th>Run time in nanoseconds $\rightarrow$</th>
<th>$1.3N^3$</th>
<th>$10N^2$</th>
<th>$47N\log_2N$</th>
<th>$48N$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>1.3 seconds</td>
<td>10 msec</td>
<td>0.4 msec</td>
<td>0.048 msec</td>
</tr>
<tr>
<td>10,000</td>
<td>22 minutes</td>
<td>1 second</td>
<td>6 msec</td>
<td>0.48 msec</td>
</tr>
<tr>
<td>100,000</td>
<td>18 days</td>
<td>17 minutes</td>
<td>78 msec</td>
<td>4.8 msec</td>
</tr>
<tr>
<td>million</td>
<td>41 years</td>
<td>2.8 hours</td>
<td>0.94 seconds</td>
<td>48 msec</td>
</tr>
<tr>
<td>10 million</td>
<td>41 millennia</td>
<td>17 weeks</td>
<td>11 seconds</td>
<td>0.48 seconds</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Time to solve a problem of size $N$</th>
</tr>
</thead>
<tbody>
<tr>
<td>second</td>
</tr>
<tr>
<td>minute</td>
</tr>
<tr>
<td>hour</td>
</tr>
<tr>
<td>day</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Max size problem solved in one</th>
</tr>
</thead>
<tbody>
<tr>
<td>N multiplied by 10, time multiplied by</td>
</tr>
<tr>
<td>--------------------------------</td>
</tr>
<tr>
<td>1,000</td>
</tr>
</tbody>
</table>

Reference: More Programming Pearls by Jon Bentley
Orders of Magnitude

<table>
<thead>
<tr>
<th>Seconds</th>
<th>Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 second</td>
</tr>
<tr>
<td>10</td>
<td>10 seconds</td>
</tr>
<tr>
<td>$10^2$</td>
<td>1.7 minutes</td>
</tr>
<tr>
<td>$10^3$</td>
<td>17 minutes</td>
</tr>
<tr>
<td>$10^4$</td>
<td>2.8 hours</td>
</tr>
<tr>
<td>$10^5$</td>
<td>11 days</td>
</tr>
<tr>
<td>$10^6$</td>
<td>1.6 weeks</td>
</tr>
<tr>
<td>$10^7$</td>
<td>3.8 months</td>
</tr>
<tr>
<td>$10^8$</td>
<td>3.1 years</td>
</tr>
<tr>
<td>$10^9$</td>
<td>3.1 decades</td>
</tr>
<tr>
<td>...</td>
<td>forever</td>
</tr>
<tr>
<td>$10^{17}$</td>
<td>age of universe</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Meters Per Second</th>
<th>Imperial Units</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^{-1}$</td>
<td>1.2 in / decade</td>
<td>Continental drift</td>
</tr>
<tr>
<td>$10^{-2}$</td>
<td>1 ft / year</td>
<td>Hair growing</td>
</tr>
<tr>
<td>$10^{-3}$</td>
<td>3.4 in / day</td>
<td>Glacier</td>
</tr>
<tr>
<td>$10^{-4}$</td>
<td>1.2 ft / hour</td>
<td>Gastro-intestinal tract</td>
</tr>
<tr>
<td>$10^{-5}$</td>
<td>2 ft / minute</td>
<td>Ant</td>
</tr>
<tr>
<td>1</td>
<td>2.2 mi / hour</td>
<td>Human walk</td>
</tr>
<tr>
<td>$10^0$</td>
<td>220 mi / hour</td>
<td>Propeller airplane</td>
</tr>
<tr>
<td>$10^1$</td>
<td>370 mi / min</td>
<td>Space shuttle</td>
</tr>
<tr>
<td>$10^2$</td>
<td>620 mi / sec</td>
<td>Earth in galactic orbit</td>
</tr>
<tr>
<td>$10^3$</td>
<td>62,000 mi / sec</td>
<td>1/3 speed of light</td>
</tr>
</tbody>
</table>

References: More Programming Pearls by Jon Bentley

Mergesort

**Mergesort (divide-and-conquer)**
- Divide array into two halves.
- Recursively sort each half.
- Merge two halves to make sorted whole.

Mergesort Implementation in Java

```java
public static void mergesort(Comparable[] a, int low, int high) {
    Comparable temp[] = new Comparable[a.length];
    for (int i = 0; i < a.length; i++) temp[i] = a[i];
    mergesort(temp, a, low, high);
}

private static void mergesort(Comparable[] from, Comparable[] to, int low, int high) {
    if (high <= low) return;
    int mid = (low + high) / 2;
    mergesort(to, from, low, mid);
    mergesort(to, from, mid+1, high);

    int p = low, q = mid+1;
    for(int i = low; i <= high; i++) {
        if (q > high) to[i] = from[p++];
        else if (p > mid) to[i] = from[q++];
        else if (less(from[q], from[p])) to[i] = from[q++];
        else to[i] = from[p++];
    }
}
```

Mergesort Analysis

**Stability?** Yes, if underlying merge is stable.

**How much memory does array implementation of mergesort require?**
- Original input = N.
- Auxiliary array for merging = N.
- Local variables: constant.
- Function call stack: $\log_2 N$.
- Total = 2N + O(log N).

**How much memory do other sorting algorithms require?**
- N + O(1) for insertion sort, selection sort, bubble sort.
- In-place = N + O(log N).
Mergesort Analysis

How long does mergesort take?
- Bottleneck = merging (and copying).
  - merging two files of size $N/2$ requires $\leq N$ comparisons
- $T(N)$ = comparisons to mergesort $N$ elements.
  - assume $N$ is a power of 2
  - assume merging requires exactly $N$ comparisons

Claim. $T(N) = N \log_2 N$.

Note: same number of comparisons for ANY file.
We’ll give several proofs to illustrate standard techniques.

Proof by Picture of Recursion Tree

$T(N) = \begin{cases} 0 & \text{if } N = 1 \\ 2T(N/2) + N & \text{merging otherwise} \end{cases}$

Proof by Telescoping

Claim. If $T(N)$ satisfies this recurrence, then $T(N) = N \log_2 N$.

$T(N) = \begin{cases} 0 & \text{if } N = 1 \\ 2T(N/2) + N & \text{merging otherwise} \end{cases}$

Proof. For $N > 1$:

$\frac{T(N)}{N} = \frac{2T(N/2)}{N} + 1$

$= \frac{T(N/2)}{N/2} + 1$

$= \frac{T(N/4)}{N/4} + 1 + 1$

$\cdots$

$= \frac{T(N/N)}{N/N} + 1 + \cdots + 1$

$= \log_2 N$

Mathematical Induction

Mathematical induction.
- Powerful and general proof technique in discrete mathematics.
- To prove a theorem true for all integers $k \geq 0$:
  - base case: prove it to be true for $N = 0$
  - induction hypothesis: assuming it is true for arbitrary $N$
  - induction step: show it is true for $N + 1$

Claim: $0 + 1 + 2 + 3 + \ldots + N = \frac{N(N+1)}{2}$ for all $N \geq 0$.

Proof: (by mathematical induction)
- Base case ($N = 0$).
  - $0 = 0(0+1)/2$.
- Induction hypothesis: assume $0 + 1 + 2 + \ldots + N = \frac{N(N+1)}{2}$.
- Induction step: $0 + 1 + \ldots + N + N + 1 = \frac{N(N+1)}{2} + N + 1$

$= \frac{N(N+1)}{2} + N + 1$

$= \frac{(N+2)(N+1)}{2}$
Proof by Induction

Claim. If $T(N)$ satisfies this recurrence, then $T(N) = N \log_2 N$.

\[
T(N) = \begin{cases} 
0 & \text{if } N = 1 \\
2T(N/2) + N & \text{otherwise}
\end{cases}
\]

Proof. (by induction on $N$)
- Base case: $N = 1$.
- Inductive hypothesis: $T(N) = N \log_2 N$.
- Goal: show that $T(2N) = 2N \log_2 (2N)$.

\[
T(2N) = 2T(N) + 2N \\
= 2N\log_2 N + 2N \\
= 2N(\log_2 (2N) - 1) + 2N \\
= 2N \log_2 (2N)
\]

Proof by Induction

Q. What if $N$ is not a power of 2?
Q. What if merging takes at most $N$ comparisons instead of exactly $N$?

A. $T(N)$ satisfies following recurrence.

\[
T(N) \leq \begin{cases} 
0 & \text{if } N = 1 \\
T([N/2]) + T([N/2]) + N & \text{otherwise}
\end{cases}
\]

Claim. $T(N) \leq N \lceil \log_2 N \rceil$.

Proof. Challenge for the bored.

Mergesort: Practical Improvements

Eliminate recursion. Bottom-up mergesort. Sedgewick Program 8.5

Stop if already sorted.
- Is biggest element in first half $\leq$ smallest element in second half?
- Helps for nearly ordered lists.

Insertion sort small files.
- Mergesort has too much overhead for tiny files.
- Cutoff to insertion sort for $< 7$ elements.

Use sentinels.
- Two of four statements in inner loop are bounds checking.
- "Superoptimization requires mindbending recursive switchery."

Sorting By Different Fields

Design challenge: enable sorting students by email or section.

```
// sort by email
Student.setSortKey(Student.EMAIL);
ArraySort.mergesort(students, 0, N-1);

// then by precept
Student.setSortKey(Student.SECTION);
ArraySort.mergesort(students, 0, N-1);
```

Mergesort is stable

<table>
<thead>
<tr>
<th>Name</th>
<th>Grade</th>
<th>Email</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anand Dharan</td>
<td>1</td>
<td>adharan</td>
</tr>
<tr>
<td>Ashley Evans</td>
<td>1</td>
<td>amevans</td>
</tr>
<tr>
<td>Alicia Myers</td>
<td>1</td>
<td>amyers</td>
</tr>
<tr>
<td>Arthur Shum</td>
<td>1</td>
<td>ashum</td>
</tr>
<tr>
<td>Amy Transe</td>
<td>1</td>
<td>stranger</td>
</tr>
<tr>
<td>Bryant Chen</td>
<td>1</td>
<td>bryantc</td>
</tr>
<tr>
<td>Charles Alden</td>
<td>1</td>
<td>calden</td>
</tr>
<tr>
<td>Cole Deforest</td>
<td>1</td>
<td>cdeforest</td>
</tr>
<tr>
<td>David Astle</td>
<td>1</td>
<td>dastle</td>
</tr>
<tr>
<td>Elinor Keith</td>
<td>1</td>
<td>ekeith1</td>
</tr>
<tr>
<td>Kira Hohensee</td>
<td>1</td>
<td>hohensee</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tom Brennan</td>
<td>5</td>
<td>tbrenna</td>
</tr>
<tr>
<td>Timothy Ruse</td>
<td>5</td>
<td>truse5</td>
</tr>
<tr>
<td>Yiting Jin</td>
<td>5</td>
<td>yjin</td>
</tr>
</tbody>
</table>

Mergesort is stable
### Sorting By Different Fields

```java
public class Student implements Comparable {
    private String first, last, email;
    private int section;

    public final static int FIRST = 0;
    public final static int LAST = 1;
    public final static int EMAIL = 2;
    public final static int SECTION = 3;

    private static int sortKey = SECTION;

    public static void setSortKey(int k) { sortKey = k; }

    public int compareTo(Object x) {
        Student a = this;
        Student b = (Student) x;
        if (sortKey == FIRST) return a.first.compareTo(b.first);
        else if (sortKey == LAST) return a.last.compareTo(b.last);
        else if (sortKey == EMAIL) return a.email.compareTo(b.email);
        else return a.section - b.section;
    }
}
```

---

### Computational Complexity

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

**Machine model.** Count 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 = # comparisons on random access machine.
- Upper bound = \( N \log_2 N \) from mergesort.
- Lower bound = \( N \log_2 N - N \log_2 e \)
- Optimal algorithm = mergesort.

---

### Decision Tree

```
\( a_1 < a_2 \)

\( a_2 < a_3 \)

\( a_1 < a_3 \)

\( a_2 < a_3 \)

\( \text{print } a_1, a_2, a_3 \)

\( \text{print } a_2, a_1, a_3 \)

\( \text{print } a_3, a_1, a_2 \)

\( \text{print } a_1, a_3, a_2 \)

\( \text{print } a_2, a_3, a_1 \)

\( \text{print } a_3, a_2, a_1 \)
```

---

### Comparison Based Sorting Lower Bound

**Theorem.** Any comparison based sorting algorithm must use \( \Omega(N \log_2 N) \) comparisons.

**Proof.** Worst case dictated by tree height \( h \).

- \( N! \) different orderings.
- One (or more) leaves corresponding to each ordering.
- Binary tree with \( N! \) leaves must have height

\[
h \geq \log_2(N!)
\]

\[
\geq \log_2(N/e)^N
\]

\[
= N \log_2 N - N \log_2 e
\]

---

What if we don’t use comparisons? Stay tuned for radix sort.
### Sorting Analysis Summary

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>
</tr>
</thead>
<tbody>
<tr>
<td>Home pc</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>1.6 weeks</td>
</tr>
</tbody>
</table>

Insertion Sort ($N^2$)

Mergesort ($N \log N$)

<table>
<thead>
<tr>
<th></th>
<th>Thousand</th>
<th>Million</th>
<th>Billion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instant</td>
<td>1 sec</td>
<td>18 min</td>
<td></td>
</tr>
</tbody>
</table>

**Lesson 1: good algorithms are better than supercomputers.**

---

### Quicksort

- Partition array so that:
  - some pivot element $a[m]$ is in its final position
  - no larger element to the left of $m$
  - no smaller element to the right of $m$

**Partitioning element**

```
QuickSort
```

- Sort each "half" recursively.

**Partitioned array**

```
QuickSort
```

**Sort each piece**

Sir Charles Antony Richard Hoare, 1960
Quicksort: Java Implementation

Quicksort.
- Partition array so that:
  - some pivot element a[m] is in its final position
  - no larger element to the left of m
  - no smaller element to the right of m
- Sort each "half" recursively.

```java
public static void quicksort(Comparable[] a, int L, int R) {
    if (R <= L) return;
    int m = partition(a, L, R);
    quicksort(a, L, m-1);
    quicksort(a, m+1, R);
}
```

How do we partition in-place efficiently?

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

Quicksort Example

Partitioning Quicksort

Quicksort: Worst Case

Number of comparisons in worst case is quadratic.
- \( N + (N-1) + (N-2) + \ldots + 1 = N(N+1)/2 \)

Worst-case inputs.
- Already sorted!
- Reverse sorted.

What about all equal keys or only two distinct keys?
- Many textbook implementations go quadratic.
- Sedgewick partitioning algorithm stops on equal keys.
- Stay tuned for 3-way quicksort.
Quicksort: **Average Case**

Average case running time.
- Roughly \(2N \ln N\) comparisons.  
- Assumption: file is randomly shuffled.
- Equivalent assumption: pivot on random element.

Remarks.
- 39% more comparisons than mergesort.
- Faster than mergesort in practice because of lower cost of other high-frequency instructions.
- Worst case still proportional to \(N^2\) but more likely that you are struck by lightning and meteor at same time.
- Caveat: many textbook implementations have best case \(N^2\) if duplicates, even if randomized!

\[\begin{align*}
\text{Theorem.} \quad & \text{The average number of comparisons } C_N \text{ to quicksort a random file of } N \text{ elements is about } 2N \ln N. \\
\text{Running time estimates:} \quad & \begin{array}{l}
\text{Home pc executes } 10^8 \text{ comparisons/second.} \\
\text{Supercomputer executes } 10^{12} \text{ comparisons/second.}
\end{array}
\end{align*}\]

\[\begin{array}{|c|c|c|c|}
\hline
& \text{thousand} & \text{million} & \text{billion} \\
\hline
\text{Insertion Sort (}\!N^2\!) & \text{home} & \text{instant} & 2.8 hours & 317 years \\
\text{super} & \text{instant} & 1 second & 1.6 weeks \\
\hline
\text{Mergesort (}\!N \ln N\!)} & \text{instant} & \text{1 sec} & 18 min \\
\text{instant} & \text{instant} & \text{1 sec} & \text{1.6 weeks} \\
\hline
\end{array}\]

\[\begin{align*}
\text{Quicksort (}\!N \ln N\!)} & \begin{array}{|c|c|c|}
\hline
& \text{thousand} & \text{billion} \\
\hline
\text{instant} & 0.3 sec & \text{6 min} \\
\text{instant} & \text{instant} & \text{instant} \\
\hline
\end{array}
\end{align*}\]

\[\begin{align*}
C_N \quad & = 2(N + 1) \ln N = 1.39 N \log_2 N.
\end{align*}\]
Quicksort: Practical Improvements

Median of sample.
- Best choice of pivot element = median.
- But how would you compute the 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 elements.
- Cutoff to insertion sort for < 10 elements.

Non-recursive version.
- Use explicit stack.
- Always sort smaller half first. \(\Rightarrow\) guarantees \(O(\log N)\) stack size

Engineering a System Sort

Samplesort.
- Basic algorithm = quicksort.
- Sort a relatively large random sample from the array.
- Use sorted elements as pivots.
- Pivots are (probabilistically) good estimates of true medians.

Bentley-McIlroy.
- Original motivation: improve \texttt{qsort} function in \texttt{C}.
- Basic algorithm = quicksort.
- Partition on Tukey’s \texttt{ninther}: Approximate median-of-9.
  - used median-of-3 elements, each of which is median-of-3
  - idea borrowed from statistics, useful in many disciplines
- 3-way quicksort to deal with equal keys.
  \[
  \text{stay tuned}
  \]

Reference: \textit{Engineering a Sort Function} by Jon L. Bentley and M. Douglas McIlroy.

System Sorts

Java’s \texttt{Arrays.sort} library function for arrays.
- Uses Bentley-McIlroy quicksort implementation for objects.
- Uses \texttt{mergesort} for primitive types.

\texttt{Arrays.sort(students, 0, N)}; \[\text{starting index is inclusive, ending index is exclusive}\]
\[\text{http://java.sun.com/j2se/1.4.2/docs/api/}\]

- To access library, need following line at beginning of program.

\texttt{import java.util.Arrays;}

Why the difference for objects and primitive types?

Breaking Java’s System Sort

Is it possible to make system sort go quadratic?
- No, for \texttt{mergesort}.
- Yes, for deterministic \texttt{quicksort}. \(\Rightarrow\) so, why are most system sorts deterministic?

McIlroy’s devious idea.
- Construct malicious input WHILE running system quicksort in response to elements compared.
  - If \(p\) is partition element, commit to \(x < p, y < p\), but don’t commit to any order on \(x, y\) until \(x\) and \(y\) are compared.

Consequences.
- Confirms theoretical possibility.
- Algorithmic complexity attack: you enter linear amount of data; server performs quadratic amount of work.
- Blows function call stack and crashes program. \(\Rightarrow\) more disastrous possibilities in \texttt{C}

Reference: McIlroy. \textit{A Killer Adversary for Quicksort}. 
Lots of Sorting Algorithms

Internal sorts.
  - Insertion sort, selection sort, bubblesort, shellsort, shaker sort.
  - Quicksort, mergesort, heapsort.
  - Samplesort, introsort.
  - Solitaire sort, red-black sort, splaysort, 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.