Advanced Topics in Sorting

- selection
- duplicate keys
- system sorts
- comparators

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 $a[i]$ is in place.
- No larger element to the left of $i$.
- No smaller element to the right of $i$.

Repeat in one subarray, depending on $i$: finished when $i$ 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 i = partition(a, lo, hi);
        if (i < k) lo = i + 1;
        else if (i > k) hi = i - 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:
  $N + N/2 + N/4 + \ldots + 1 \approx 2N$ 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 might use $\sim N^2/2$ compares, but as with quicksort, the random shuffle provides a probabilistic guarantee.
Theoretical context for selection

**Challenge.** Design a selection algorithm whose running time is linear in the worst-case.

**Theorem.** [Blum, Floyd, Pratt, Rivest, Tarjan, 1973] There exists a compare-based selection algorithm that takes linear time in the worst case.

**Remark.** 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

**Safe version.** Compiles cleanly, no cast needed in client.

```java
public class Quick {
    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).

---

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 is also unhappy.**

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

**Q. How to fix?**
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 file.
• Small number of key values.

Mergesort with duplicate keys. Always ∼ 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().

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]
• Convention 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 $a[lt]$ 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.

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: Java implementation

3-way quicksort: visual trace
Duplicate keys: lower bound

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

**Pf.** [beyond scope of course]
- Generalize decision tree.
- Tie cost to Shannon entropy.

**Ex.** Linear-time when only a constant number of distinct keys.

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

---

**Comparable interface:** sort uses type’s natural order.

```java
class Date implements Comparable<Date> {
    private final int month, day, year;
    public Date(int m, int d, int y) {
        month = m;
        day = d;
        year = y;
    }
    ... public int compareTo(Date that) {
        if (this.year < that.year ) return -1;
        if (this.year > that.year ) return +1;
        if (this.month < that.month) return -1;
        if (this.month > that.month) return +1;
        if (this.day < that.day) return -1;
        if (this.day > that.day) return +1;
        return 0;
    }
}
```

---

**Generalized compare**

**Comparable interface:** sort uses type’s natural order.

**Problem 1.** May want to use a non-natural order.
**Problem 2.** Desired data type may not come with a “natural” order.

**Ex.** Sort strings by:
- Natural order.
- Case insensitive.
- Spanish.
- British phone book.

```java
String[] a;
...
Arrays.sort(a, String.CASE_INSENSITIVE_ORDER);
Arrays.sort(a, Collator.getInstance(Locale.SPANISH));
```

---

Java text collator: Pre-1994 order for digraphs ch and ll and rr.

```java
import java.text.Collator;
```
Comparators

Solution. Use Java’s Comparator interface.

```java
public interface Comparator<Key>
{
    public int compare(Key v, Key w);
}
```

Remark. The `compare()` method implements a total order like `compareTo()`.

Advantages. Decouples the definition of the data type from the definition of what it means to compare two objects of that type.
- Can add any number of new orders to a data type.
- Can add an order to a library data type with no natural order.

Comparator example

Reverse order. Sort an array of strings in reverse order.

```java
public class ReverseOrder implements Comparator<String>
{
    public int compare(String a, String b)
    {
        return b.compareTo(a);
    }
}
```

```
Arrays.sort(a, new ReverseOrder());
```

Sort implementation with comparators

To support comparators in our sort implementations:
- Pass Comparator to `sort()` and `less()`.
- Use it in `less()`.

Ex. Insertion sort.

```java
public static <Key> void sort(Key[] a, Comparator<Key> comparator)
{
    int N = a.length;
    for (int i = 0; i < N; i++)
        for (int j = i; j > 0; j--)
            if (less(comparator, a[j], a[j-1]))
                exch(a, j, j-1);
            else break;

    private static <Key> boolean less(Comparator<Key> c, Key v, Key w)
    { return c.compare(v, w) < 0; }

    private static <Key> void exch(Key[] a, int i, int j)
    { Key swap = a[i]; a[i] = a[j]; a[j] = swap; }
```

Generalized compare

Comparators enable multiple sorts of a single file (by different keys).

Ex. Sort students by name or by section.

```java
Arrays.sort(students, Student.BY_NAME);
Arrays.sort(students, Student.BY_SECT);
```
Enable sorting students by name or by section.

```java
public class Student
{
    private final String name;
    private final int section;

    private static class ByName implements Comparator<Student>
    {
        public int compare(Student a, Student b)
        {  return a.name.compareTo(b.name);  }
    }

    private static class BySect implements Comparator<Student>
    {
        public int compare(Student a, Student b)
        {  return a.section - b.section;  }
    }
}
```

Generalized compare

A typical application. First, sort by name; then sort by section.

@#%&@!! Students in section 3 no longer in order by name.

A stable sort preserves the relative order of records with equal keys.

Stability

Q. Which sorts are stable?
- Selection sort?
- Insertion sort?
- Shellsort?
- Quicksort?
- Mergesort?

Open problem. Stable, inplace, N log N, practical sort??
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.

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.

Why use Tukey’s ninther?

- Better partitioning than sampling.
- Less costly than random.
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 file is randomly ordered 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, splay sort, 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.
• GPU sort.

Applications have diverse attributes.
• Stable?
• Multiple keys?
• Deterministic?
• Keys all distinct?
• Multiple key types?
• Linked list or arrays?
• Large or small records?
• Is your file 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^{1/2} )</td>
<td>( N^{1/2} )</td>
<td>( N^{1/2} )</td>
<td>( N ) exchanges</td>
</tr>
<tr>
<td>insertion</td>
<td>x</td>
<td>x</td>
<td>( N^{1/2} )</td>
<td>( N^{1/4} )</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>quick</td>
<td>x</td>
<td>( N^{1/2} )</td>
<td>( 2 N \ln N )</td>
<td>( N \lg N )</td>
<td>( N \lg N ) probabilistic guarantee fastest in practice</td>
</tr>
<tr>
<td>3-way quick</td>
<td>x</td>
<td>( N^{1/2} )</td>
<td>( 2 N \ln N )</td>
<td>( N \lg N )</td>
<td>improves quicksort in presence of duplicate keys</td>
</tr>
<tr>
<td>merge</td>
<td>x</td>
<td>( N \lg N )</td>
<td>( N \lg N )</td>
<td>( N \lg N )</td>
<td>( N \lg N ) guarantee, stable</td>
</tr>
<tr>
<td>???</td>
<td>x</td>
<td>( N \lg N )</td>
<td>( N \lg N )</td>
<td>( N \lg N )</td>
<td>holy sorting grail</td>
</tr>
</tbody>
</table>