Advanced Topics in Sorting

- complexity
- selection
- system sorts
- equal keys
- comparators

References: Algorithms in Java, Chapter 8
Intro to Algs and Data Structs, Chapter 3

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.

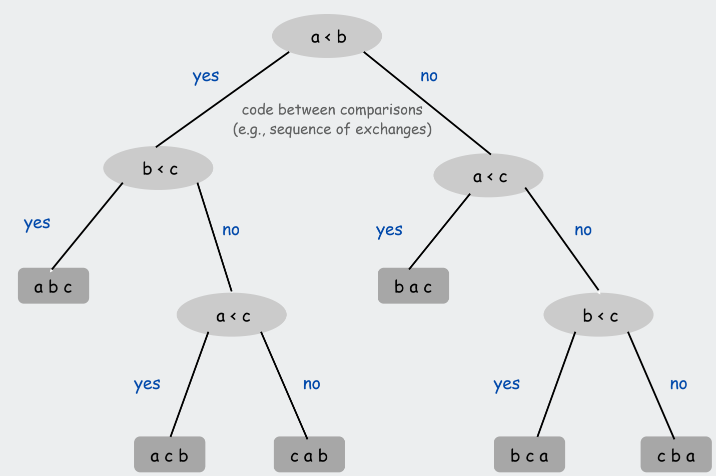
lower bound ~ upper bound

Example: sorting.

- Machine model = # comparisons ← access information only through compares
- Upper bound = N lg N from mergesort.
- Lower bound ?

complexity
selection
system sorts
duplicate keys
comparators

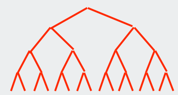
Decision Tree



Comparison-based lower bound for sorting

Theorem. Any comparison based sorting algorithm must use more than N lg N - 1.44 N comparisons in the worst-case.

Pf.
- Assume input consists of N distinct values a_1 through a_N.
- Worst case dictated by tree height h.
- N! different orderings.
- (At least) one leaf corresponds to each ordering.
- Binary tree with N! leaves cannot have height less than lg (N!)



$$\begin{aligned} h &\geq \lg N! \\ &\geq \lg (N / e)^N \quad \longleftarrow \text{Stirling's formula} \\ &= N \lg N - N \lg e \\ &\geq N \lg N - 1.44\, N \end{aligned}$$

Complexity of sorting

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
- Upper bound = N lg N (mergesort)
- Lower bound = N lg N - 1.44 N

Mergesort is optimal (to within a small additive factor)

lower bound ≈ upper bound

First goal of algorithm design: optimal algorithms

Drawbacks of complexity results

Mergesort is optimal (to within a small additive factor)

Other operations?
- statement is only about number of compares
- quicksort is faster than mergesort (lower use of other operations)

Space?
- mergesort is not optimal with respect to space usage
- insertion sort, selection sort, shellsort, quicksort are space-optimal
- is there an algorithm that is both time- and space-optimal?

stay tuned for radix sorts

Is my case the worst case?
- statement is only about guaranteed worst-case performance
- quicksort's probabilistic guarantee is just as good in practice

Lessons
- use theory as a guide ← don't try to design an algorithm that uses half as many compares as mergesort
- know your algorithms ← use quicksort when time and space are critical

More drawbacks of complexity results

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.

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

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

stay tuned for 3-way quicksort

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

stay tuned for radix sorts

complexity
selection
system sorts
duplicate keys
comparators

Selection

Find the k^{th} largest element.

- Min: k = 1.
- Max: k = N.
- Median: k = N/2.

Applications.

- Order statistics.
- Find the "top k"

Use theory as a guide

- easy O(N log N) upper bound: sort, return a[k]
- easy O(N) upper bound for some k: min, max
- easy O(N) lower bound: must examine every element

Which is true?

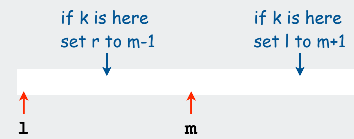
- O(N log N) lower bound? [is selection as hard as sorting?]
- O(N) upper bound? [linear algorithm for all k]

Selection: quick-select algorithm

Partition array so that:

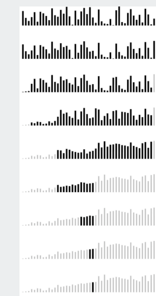
- element `a[m]` is in place
- no larger element to the left of `m`
- no smaller element to the right of `m`

Repeat in one subarray, depending on `m`.



Finished when m = k ← a[k] is in place, no larger element to the left, no smaller element to the right

```
public static void select(Comparable[] a, int k)
{
    StdRandom.shuffle(a);
    int l = 0;
    int r = a.length - 1;
    while (r > l)
    {
       int i = partition(a, l, r);
       if      (m > k) r = m - 1;
       else if (m < k) l = m + 1;
       else return;
    }
}
```



Selection analysis

Theorem. Quick-select takes linear time on average.

Pf.

- Intuitively, each partitioning step roughly splits array in half.
- N + N/2 + N/4 + … + 1 ≈ 2N comparisons.
- Formal analysis similar to quicksort analysis:

$$C_N = 2N + k \ln(N/k) + (N-k)\ln(N/(N-k))$$

Ex: (2 + 2 ln 2) N comparisons to find the median

Note. Might use $\sim N^2/2$ comparisons, but as with quicksort, the random shuffle provides a probabilistic guarantee.

Theorem. [Blum, Floyd, Pratt, Rivest, Tarjan, 1973] There exists a selection algorithm that take linear time in the worst case.
Note. Algorithm is far 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

complexity
selection
system sorts
duplicate keys
comparators

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 news items in reverse chronological order.

obvious applications

- Find the median.
- Find the closest pair.
- Binary search in a database.
- Identify statistical outliers.
- Find duplicates in a mailing list.

problems become easy once items are in sorted order

- Data compression.
- Computer graphics.
- Computational biology.
- Supply chain management.
- Load balancing on a parallel computer.
- . . .

non-obvious applications

Every system needs (and has) a system sort!

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.

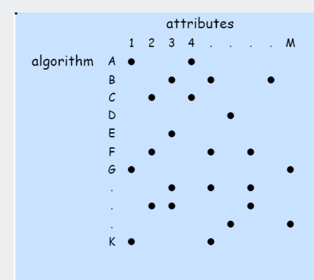
radix sorts.
- Distribution, MSD, LSD.
- 3-way radix quicksort.

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?
- 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?



many more combinations of attributes than algorithms

Elementary sort may be method of choice for some combination.
Cannot cover all combinations of attributes.

Q. Is the system sort good enough?
A. Maybe (no matter which algorithm it uses).

complexity
selection
system sorts
duplicate keys
comparators

Duplicate keys

Often, purpose of sort is to bring records with duplicate keys together.
- Sort population by age.
- Finding 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!
- [many textbook and system implementations have this problem]
- 1990s Unix user found this problem in qsort()

Duplicate keys: the problem

Assume all keys are equal.
Recursive code guarantees that case will predominate!

Mistake: Put all keys equal to the partitioning element on one side
- easy to code
- guarantees N^2 running time 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

Recommended: Stop scans on keys equal to the partitioning element
- easy to code
- guarantees 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

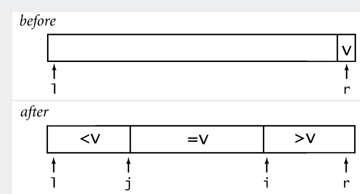
Desirable: Put all keys equal to the partitioning element in place

A A A B B B B B C C C   A A A A A A A A A A A

Common wisdom to 1990s: not worth adding code to inner loop

3-Way Partitioning

3-way partitioning. Partition elements into 3 parts:
- Elements between `i` and `j` equal to partition element `v`.
- No larger elements to left of `i`.
- No smaller elements to right of `j`.

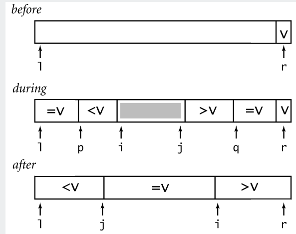


Dutch national flag problem.
- not done in practical sorts before mid-1990s.
- new approach discovered when fixing mistake in Unix qsort()
- now incorporated into Java system sort

Solution to Dutch national flag problem.

3-way partitioning (Bentley-McIlroy).

- Partition elements into 4 parts:
  - no larger elements to left of `i`
  - no smaller elements to right of `j`
  - equal elements to left of `p`
  - equal elements to right of `q`
- Afterwards, swap equal keys into center.



All the right properties.

- in-place.
- not much code.
- linear if keys are all equal.
- small overhead if no equal keys.

3-way Quicksort: Java Implementation

```
private static void sort(Comparable[] a, int l, int r)
{
   if (r <= l) return;
   int i = l-1, j = r;
   int p = l-1, q = r;

   while(true)                                        4-way partitioning
   {
      while (less(a[++i], a[r])) ;
      while (less(a[r], a[--j])) if (j == l) break;
      if (i >= j) break;
      exch(a, i, j);
      if (eq(a[i], a[r])) exch(a, ++p, i);   swap equal keys to left or right
      if (eq(a[j], a[r])) exch(a, --q, j);
   }
   exch(a, i, r);
                                                      swap equal keys back to middle
   j = i - 1;
   i = i + 1;
   for (int k = l  ; k <= p; k++) exch(a, k, j--);
   for (int k = r-1; k >= q; k--) exch(a, k, i++);

   sort(a, l, j);                                     recursively sort left and right
   sort(a, i, r);
}
```


Duplicate keys: lower bound

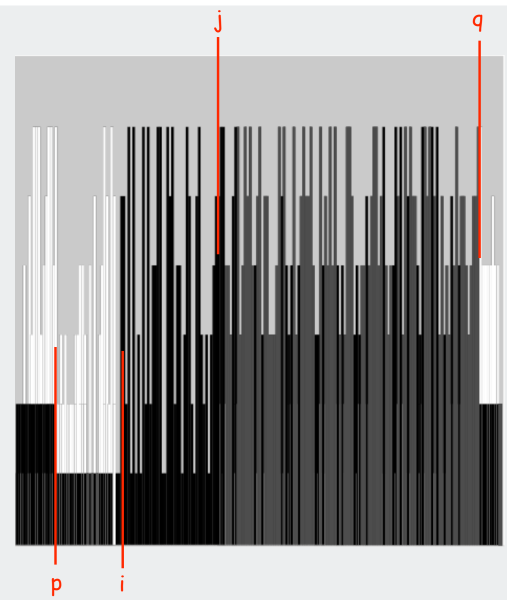
Theorem. [Sedgewick-Bentley] Quicksort with 3-way partitioning is optimal for random keys with duplicates.

Proof (beyond scope of 226).

- generalize decision tree
- tie cost to entropy
- note: cost is linear when number of key values is $O(1)$

Bottom line: Randomized Quicksort with 3-way partitioning reduces cost from quadratic to linear (!) in broad class of applications

3-way partitioning animation



complexity
selection
system sorts
duplicate keys
comparators

Generalized compare

Comparable interface: sort uses type's compareTo() function:

```
public class Date implements Comparable<Date>
{
   private int month, day, year;

   public Date(int m, int d, int y)
   {
      month = m;
      day   = d;
      year  = y;
   }

   public int compareTo(Date b)
   {
      Date a = this;
      if (a.year  < b.year ) return -1;
      if (a.year  > b.year ) return +1;
      if (a.month < b.month) return -1;
      if (a.month > b.month) return +1;
      if (a.day   < b.day  ) return -1;
      if (a.day   > b.day  ) return +1;
      return 0;
   }
}
```


Generalized compare

Comparable interface: sort uses type's compareTo() function:

Problem 1: May want to use a different order.
Problem 2: Some types may have no "natural" order.

Ex. Sort strings by:

- Natural order. `Now is the time`
- Case insensitive. `is Now the time`
- French. `real réal rico`
- Spanish. `café cuidado champiñón dulce`

ch and rr are single letters

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

`import java.text.Collator;`

Generalized compare

Comparable interface: sort uses type's compareTo() function:

Problem 1: May want to use a different order.
Problem 2: Some types may have no "natural" order.

Solution: Use Comparator interface

Comparator interface. Require a method `compare()` so that `compare(v, w)` is a total order that behaves like `compareTo()`.

Advantage. Separates the definition of the data type from definition of what it means to compare two objects of that type.

- add any number of new orders to a data type.
- add an order to a library data type with no natural order.

Generalized compare

Comparable interface: sort uses type's compareTo() function:

Problem 1: May want to use a different order.
Problem 2: Some types may have no "natural" order.

Solution: Use Comparator interface

Example:

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

reverse sense of comparison

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


Generalized compare

Easy modification to support comparators in our sort implementations

- pass comparator to sort(), less()
- use it in less

Example: (insertion sort)

```
public static void sort(Object[] a, Comparator 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 boolean less(Comparator c, Object v, Object w)
{   return c.compare(v, w) < 0;    }

private static void exch(Object[] a, int i, int j)
{   Object t = a[i]; a[i] = a[j]; a[j] = t;   }
```


Generalized compare

Comparators enable multiple sorts of single file (different keys)

Example. Enable sorting students by name or by section.

```
Arrays.sort(students, Student.BY_NAME);
Arrays.sort(students, Student.BY_SECT);
```

sort by name

Andrews	3	A	664-480-0023	097 Little
Battle	4	C	874-088-1212	121 Whitman
Chen	2	A	991-878-4944	308 Blair
Fox	1	A	884-232-5341	11 Dickinson
Furia	3	A	766-093-9873	101 Brown
Gazsi	4	B	665-303-0266	22 Brown
Kanaga	3	B	898-122-9643	22 Brown
Rohde	3	A	232-343-5555	343 Forbes

then sort by section

Fox	1	A	884-232-5341	11 Dickinson
Chen	2	A	991-878-4944	308 Blair
Andrews	3	A	664-480-0023	097 Little
Furia	3	A	766-093-9873	101 Brown
Kanaga	3	B	898-122-9643	22 Brown
Rohde	3	A	232-343-5555	343 Forbes
Battle	4	C	874-088-1212	121 Whitman
Gazsi	4	B	665-303-0266	22 Brown

Generalized compare

Comparators enable multiple sorts of single file (different keys)

Example. Enable sorting students by name or by section.

```
public class Student
{
   public static final Comparator<Student> BY_NAME = new ByName();
   public static final Comparator<Student> BY_SECT = new BySect();

   private String name;
   private 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;  }
   }
}
```

only use this trick if no danger of overflow

Generalized compare problem

A typical application
- first, sort by name
- then, sort by section

`Arrays.sort(students, Student.BY_NAME);`

Andrews	3	A	664-480-0023	097 Little
Battle	4	C	874-088-1212	121 Whitman
Chen	2	A	991-878-4944	308 Blair
Fox	1	A	884-232-5341	11 Dickinson
Furia	3	A	766-093-9873	101 Brown
Gazsi	4	B	665-303-0266	22 Brown
Kanaga	3	B	898-122-9643	22 Brown
Rohde	3	A	232-343-5555	343 Forbes

`Arrays.sort(students, Student.BY_SECT);`

Fox	1	A	884-232-5341	11 Dickinson
Chen	2	A	991-878-4944	308 Blair
Kanaga	3	B	898-122-9643	22 Brown
Andrews	3	A	664-480-0023	097 Little
Furia	3	A	766-093-9873	101 Brown
Rohde	3	A	232-343-5555	343 Forbes
Battle	4	C	874-088-1212	121 Whitman
Gazsi	4	B	665-303-0266	22 Brown

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

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

Stability

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

A. Careful look at code required.

Annoying fact. Many useful sorting algorithms are unstable.

Easy solutions.
- add an integer rank to the key
- careful implementation of mergesort

Open: Stable, inplace, optimal, practical sort??

Java system sorts

Use theory as a guide: Java uses both mergesort and quicksort.
- Can sort array of type `Comparable` or any primitive type.
- Uses quicksort for primitive types.
- Uses mergesort for objects.

```
import java.util.Arrays;
public class IntegerSort
{
    public static void main(String[] args)
    {
        int N = Integer.parseInt(args[0]);
        int[] a = new int[N];
        for (int i = 0; i < N; i++)
            a[i] = StdIn.readInt();
        Arrays.sort(a);
        for (int i = 0; i < N; i++)
            System.out.println(a[i]);
    }
}
```

Q. Why use two different sorts?
A. Use of primitive types indicates time and space are critical: quicksort
A. Use of objects indicates time and space not so critical:
mergesort provides worst-case guarantee and stability.

Arrays.sort() for primitive types

Bentley-McIlroy. [Engineeering a Sort Function]
- Original motivation: improve `qsort()` function in C.
- Basic algorithm = 3-way quicksort with cutoff to insertion sort.
- Partition on Tukey's ninther: median-of-3 elements, each of which is a median-of-3 elements.

approximate median-of-9

nine evenly spaced elements

R		A		M		G		X		K		B		J		E

groups of 3

R	A	M	G	X	K	B	J	E

medians

M	K	E

ninther

K

Why use ninther?
- better partitioning than sampling
- quick and easy to implement with macros
- less costly than random ← Good idea? Stay tuned.

A final caution

Based on all this research, Java's system sort is solid, right?

McIlroy's devious idea. [A Killer Adversary for Quicksort]

- Construct malicious input while running system quicksort, in response to elements compared.
- If `p` is pivot, commit to `(x < p)` and `(y < p)`, but don't commit to `(x < y)` or `(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.

A final caution

A killer input. Blows function call stack in Java and crashes program.

more disastrous possibilities in C

```
% more 250000.txt
0
218750
222662
11
166672
247070
83339
156253
...
```

```
% java IntegerSort < 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.

Achilles heel: no guarantees in implementation (randomization is required)