String Sorts

Strings in Java

> Sequence of characters

Char data type

C char data type. Typically an 8-bit integer.

- Supports 7-bit ASCII.
- Can represent only 256 characters.

Java char data type. A 16-bit unsigned integer.

- Supports original 16-bit Unicode.
- Supports 21-bit Unicode 3.0 (awkwardly).

String data type

String data type (in Java). Sequence of characters (immutable).
Underlying implementation. Immutable `char[]` array, offset, and length.

	String	
operation	guarantee	extra space
`length()`	1	1
`charAt()`	1	1
`substring()`	1	1
`concat()`	N	N

Memory. $40 + 2N$ bytes for a virgin `String` of length N.

can use `byte[]` or `char[]` instead of `String` to save space (but lose convenience of `String` data type)

StringBuilder data type

StringBuilder data type. Sequence of characters (mutable).
Underlying implementation. Resizing `char[]` array and length.

	String		StringBuilder	
operation	guarantee	extra space	guarantee	extra space
length()	1	1	1	1
charAt()	1	1	1	1
substring()	1	1	N	N
concat()	N	N	1 *	1 *

* amortized

Alphabets

- **Digital key**---sequence of digits over fixed alphabet
- **Radix**---number of digits R in alphabet

Key-indexed Counting

Baisc

- Sort an array a[] of N integers between 0 and $R-1$
  - count frequencies of each letter using key as index
  - Compute frequency cumulates which specify destinations
  - Access cumulates using keys as index to move items
  - Copy back into original array

Complexity

- Key-indexed counting uses ~$11N+4$ array accesses to sort N items whose keys are integers between 0 and $R-1$
- Key-indexed couning uses extra space proportional to $N+R$
- stable

LSD radix sort

> Least-significant-digit-first sort

Baisc

- Consider characters from right to left
- Stably sort using d^{th} character as the key (using key-indexed counting)

Implementation

```java
public class LSD {
    private static final int BITS_PER_BYTE = 8;

    // do not instantiate
    private LSD() { }

    public static void sort(String[] a, int w) {
        int n = a.length;
        int R = 256;   // extend ASCII alphabet size
        String[] aux = new String[n];

        for (int d = w-1; d >= 0; d--) {
            // sort by key-indexed counting on dth character

            // compute frequency counts
            int[] count = new int[R+1];
            for (int i = 0; i < n; i++)
                count[a[i].charAt(d) + 1]++;

            // compute cumulates
            for (int r = 0; r < R; r++)
                count[r+1] += count[r];

            // move data
            for (int i = 0; i < n; i++)
                aux[count[a[i].charAt(d)]++] = a[i];

            // copy back
            for (int i = 0; i < n; i++)
                a[i] = aux[i];
        }
    }

    public static void sort(int[] a) {
        final int BITS = 32;                 // each int is 32 bits
        final int R = 1 << BITS_PER_BYTE;    // each byte is between 0 and 255
        final int MASK = R - 1;              // 0xFF
        final int w = BITS / BITS_PER_BYTE;  // each int is 4 bytes

        int n = a.length;
        int[] aux = new int[n];

        for (int d = 0; d < w; d++) {

            // compute frequency counts
            int[] count = new int[R+1];
            for (int i = 0; i < n; i++) {
                int c = (a[i] >> BITS_PER_BYTE*d) & MASK;
                count[c + 1]++;
```

```
            }

            // compute cumulates
            for (int r = 0; r < R; r++)
                count[r+1] += count[r];

            // for most significant byte, 0x80-0xFF comes before 0x00-0x7F
            if (d == w-1) {
                int shift1 = count[R] - count[R/2];
                int shift2 = count[R/2];
                for (int r = 0; r < R/2; r++)
                    count[r] += shift1;
                for (int r = R/2; r < R; r++)
                    count[r] -= shift2;
            }

            // move data
            for (int i = 0; i < n; i++) {
                int c = (a[i] >> BITS_PER_BYTE*d) & MASK;
                aux[count[c]++] = a[i];
            }

            int[] temp = a;
            a = aux;
            aux = temp;
        }
    }

    public static void main(String[] args) {
        String[] a = StdIn.readAllStrings();
        int n = a.length;

        // check that strings have fixed length
        int w = a[0].length();
        for (int i = 0; i < n; i++)
            assert a[i].length() == w : "Strings must have fixed length";
        sort(a, w);
        for (int i = 0; i < n; i++)
            StdOut.println(a[i]);
    }
}
```

MSD radix sort

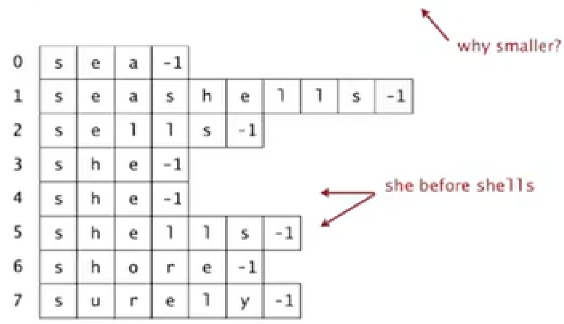
> most-significant-digit-first sor

Basic

- Partition array into R pieces according to first character (use key-indexed counting)
- Recursively sort all strings that start with each character (key-indexed counts delineate subarrays to sort)

Variable-length strings

Treat strings as if they had an extra char at end (smaller than any char).




```
private static int charAt(String s, int d)
{
   if (d < s.length()) return s.charAt(d);
   else return -1;
}
```

Improvement

Problem

Observation 1. Much too slow for small subarrays.

- Each function call needs its own `count[]` array.
- ASCII (256 counts): 100x slower than copy pass for $N = 2$.
- Unicode (65,536 counts): 32,000x slower for $N = 2$.

Observation 2. Huge number of small subarrays because of recursion.

Solution

> Cutoff to insertion sort

Performance

- MSD examines just enough characters to sort the keys
- Number of characters examined depends on keys
- Can be sublinear in input size

MSD vs. Quicksort

- Disadvantages of MSD
  - access memory randomly (cache inefficient)
  - Inner loop has a lot of instructions
  - Extra space for count[] and aux[]
- Disadvantage of quicksort
  - Linearithmic number of string compares (not linear)
  - Has to rescan many characters in keys wih long prefix matches

Implementation

```java
public class MSD {
    private static final int BITS_PER_BYTE =   8;
    private static final int BITS_PER_INT  =  32;   // each Java int is 32 bits
    private static final int R             = 256;   // extended ASCII alphabet size
    private static final int CUTOFF        =  15;   // cutoff to insertion sort

    // do not instantiate
    private MSD() { }

    public static void sort(String[] a) {
        int n = a.length;
        String[] aux = new String[n];
        sort(a, 0, n-1, 0, aux);
    }

    // return dth character of s, -1 if d = length of string
    private static int charAt(String s, int d) {
        assert d >= 0 && d <= s.length();
        if (d == s.length()) return -1;
        return s.charAt(d);
    }

    // sort from a[lo] to a[hi], starting at the dth character
    private static void sort(String[] a, int lo, int hi, int d, String[] aux) {

        // cutoff to insertion sort for small subarrays
        if (hi <= lo + CUTOFF) {
            insertion(a, lo, hi, d);
            return;
        }

        // compute frequency counts
```

```
        int[] count = new int[R+2];
        for (int i = lo; i <= hi; i++) {
            int c = charAt(a[i], d);
            count[c+2]++;
        }

        // transform counts to indices
        for (int r = 0; r < R+1; r++)
            count[r+1] += count[r];

        // distribute
        for (int i = lo; i <= hi; i++) {
            int c = charAt(a[i], d);
            aux[count[c+1]++] = a[i];
        }

        // copy back
        for (int i = lo; i <= hi; i++)
            a[i] = aux[i - lo];


        // recursively sort for each character (excludes sentinel -1)
        for (int r = 0; r < R; r++)
            sort(a, lo + count[r], lo + count[r+1] - 1, d+1, aux);
    }


    // insertion sort a[lo..hi], starting at dth character
    private static void insertion(String[] a, int lo, int hi, int d) {
        for (int i = lo; i <= hi; i++)
            for (int j = i; j > lo && less(a[j], a[j-1], d); j--)
                exch(a, j, j-1);
    }

    // exchange a[i] and a[j]
    private static void exch(String[] a, int i, int j) {}

    // is v less than w, starting at character d
    private static boolean less(String v, String w, int d) {
        // assert v.substring(0, d).equals(w.substring(0, d));
        for (int i = d; i < Math.min(v.length(), w.length()); i++) {
            if (v.charAt(i) < w.charAt(i)) return true;
            if (v.charAt(i) > w.charAt(i)) return false;
        }
        return v.length() < w.length();
    }

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

```java

    // MSD sort from a[lo] to a[hi], starting at the dth byte
    private static void sort(int[] a, int lo, int hi, int d, int[] aux) {

        // cutoff to insertion sort for small subarrays
        if (hi <= lo + CUTOFF) {
            insertion(a, lo, hi);
            return;
        }

        // compute frequency counts (need R = 256)
        int[] count = new int[R+1];
        int mask = R - 1;   // 0xFF;
        int shift = BITS_PER_INT - BITS_PER_BYTE*d - BITS_PER_BYTE;
        for (int i = lo; i <= hi; i++) {
            int c = (a[i] >> shift) & mask;
            count[c + 1]++;
        }

        // transform counts to indices
        for (int r = 0; r < R; r++)
            count[r+1] += count[r];

        // for most significant byte, 0x80-0xFF comes before 0x00-0x7F
        if (d == 0) {
            int shift1 = count[R] - count[R/2];
            int shift2 = count[R/2];
            count[R] = shift1 + count[1];   // to simplify recursive calls later
            for (int r = 0; r < R/2; r++)
                count[r] += shift1;
            for (int r = R/2; r < R; r++)
                count[r] -= shift2;
        }

        // distribute
        for (int i = lo; i <= hi; i++) {
            int c = (a[i] >> shift) & mask;
            aux[count[c]++] = a[i];
        }

        // copy back
        for (int i = lo; i <= hi; i++)
            a[i] = aux[i - lo];

        // no more bits
        if (d == 3) return;

        // special case for most significant byte
        if (d == 0 && count[R/2] > 0)
            sort(a, lo, lo + count[R/2] - 1, d+1, aux);

```

```java
    // special case for other bytes
    if (d != 0 && count[0] > 0)
      sort(a, lo, lo + count[0] - 1, d+1, aux);

    // recursively sort for each character
    // (could skip r = R/2 for d = 0 and skip r = R for d > 0)
    for (int r = 0; r < R; r++)
      if (count[r+1] > count[r])
        sort(a, lo + count[r], lo + count[r+1] - 1, d+1, aux);
  }

  // insertion sort a[lo..hi]
  private static void insertion(int[] a, int lo, int hi) {
    for (int i = lo; i <= hi; i++)
      for (int j = i; j > lo && a[j] < a[j-1]; j--)
        exch(a, j, j-1);
  }

  public static void main(String[] args) {}
}
```

3-way radix quicksort

Basic

- Do 3-way partitioning on the d^{th} character
  - Less overhead than R-way partitioning in MSD strinmg sort
  - Doesn't re-examine characters equal to the partitioning char

vs. Quuck Sort

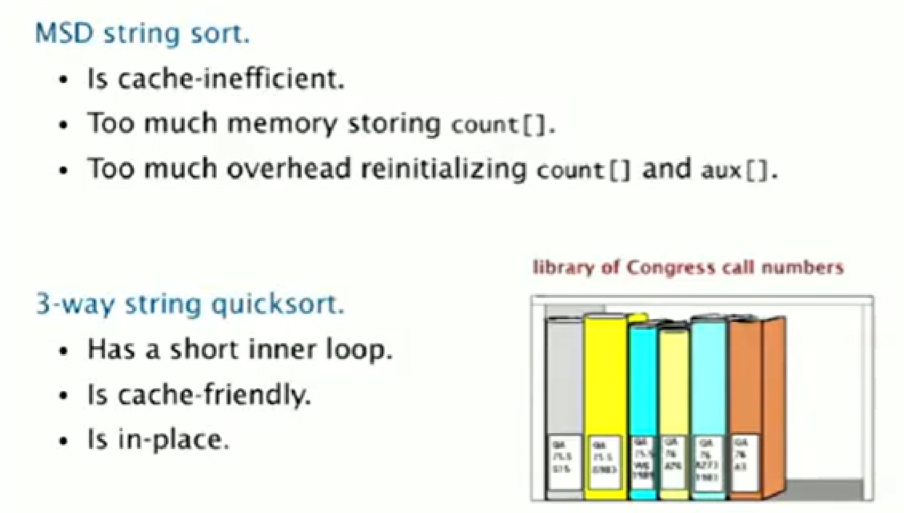
Standard quicksort.

- Uses ~ $2N \ln N$ string compares on average.
- Costly for keys with long common prefixes (and this is a common case!)

3-way string (radix) quicksort.

- Uses ~ $2N \ln N$ character compares on average for random strings.
- Avoids re-comparing long common prefixes.

vs. MDS sort



Implementation

```java
public class Quick3string {
    private static final int CUTOFF =  15;   // cutoff to insertion sort

    // do not instantiate
    private Quick3string() { }

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

    // return the dth character of s, -1 if d = length of s
    private static int charAt(String s, int d) {
        assert d >= 0 && d <= s.length();
        if (d == s.length()) return -1;
        return s.charAt(d);
    }


    // 3-way string quicksort a[lo..hi] starting at dth character
    private static void sort(String[] a, int lo, int hi, int d) {

        // cutoff to insertion sort for small subarrays
        if (hi <= lo + CUTOFF) {
            insertion(a, lo, hi, d);
            return;
        }

        int lt = lo, gt = hi;
        int v = charAt(a[lo], d);
        int i = lo + 1;
        while (i <= gt) {
            int t = charAt(a[i], d);
            if      (t < v) exch(a, lt++, i++);
```

```
            else if (t > v) exch(a, i, gt--);
            else              i++;
        }

        // a[lo..lt-1] < v = a[lt..gt] < a[gt+1..hi].
        sort(a, lo, lt-1, d);
        if (v >= 0) sort(a, lt, gt, d+1);
        sort(a, gt+1, hi, d);
    }

    // sort from a[lo] to a[hi], starting at the dth character
    private static void insertion(String[] a, int lo, int hi, int d) {
        for (int i = lo; i <= hi; i++)
            for (int j = i; j > lo && less(a[j], a[j-1], d); j--)
                exch(a, j, j-1);
    }

    // exchange a[i] and a[j]
    private static void exch(String[] a, int i, int j) {
        String temp = a[i];
        a[i] = a[j];
        a[j] = temp;
    }

    private static boolean less(String v, String w, int d) {
        assert v.substring(0, d).equals(w.substring(0, d));
        for (int i = d; i < Math.min(v.length(), w.length()); i++) {
            if (v.charAt(i) < w.charAt(i)) return true;
            if (v.charAt(i) > w.charAt(i)) return false;
        }
        return v.length() < w.length();
    }

    // is the array sorted
    private static boolean isSorted(String[] a) {
        for (int i = 1; i < a.length; i++)
            if (a[i].compareTo(a[i-1]) < 0) return false;
        return true;
    }

    public static void main(String[] args) {

        // read in the strings from standard input
        String[] a = StdIn.readAllStrings();
        int n = a.length;

        // sort the strings
        sort(a);

        // print the results
        for (int i = 0; i < n; i++)
            StdOut.println(a[i]);
```

```
    }
}
```

Comparison

algorithm	guarantee	random	extra space	stable?	operations on keys
insertion sort	$N^2 / 2$	$N^2 / 4$	1	yes	compareTo()
mergesort	N lg N	N lg N	N	yes	compareTo()
quicksort	1.39 N lg N *	1.39 N lg N	c lg N	no	compareTo()
heapsort	2 N lg N	2 N lg N	1	no	compareTo()
LSD †	2 N W	2 N W	N + R	yes	charAt()
MSD ‡	2 N W	$N \log_R N$	N + D R	yes	charAt()
3-way string quicksort	1.39 W N lg N *	1.39 N lg N	log N + W	no	charAt()

Suffix arrays

Keyword-in-context search

- Steps
  - form suffixes
  - sort suffixes to bring repeated substrings together
  - bianry search for query; scan until mismatch

Longest repeated substring

- Applications
  - Bioinformatics
  - cryptanalysis
  - music notes

Brute-force Algorithm

- Steps
  - Try all indices i and j for start of possible match
  - Compute longest common prefix (LCP) for each pair
  - Running time $\leqslant DN^2$, where D is the length of longest match

A Sorting Solution

- Steps
  - form suffixes
  - sort suffixes to bring repeated substrings together

```java
public String lrs(String s)
{
  int N = s.length();

  String[] suffixes = new String[N];
  for (int i = 0; i < N; i++)
     suffixes[i] = s.substring(i, N);

  Arrays.sort(suffixes);

  String lrs = "";
  for (int i = 0; i < N-1; i++)
  {
     int len = lcp(suffixes[i], suffixes[i+1]);
     if (len > lrs.length())
        lrs = suffixes[i].substring(0, len);
  }
  return lrs;
}
```

create suffixes (linear time and space)

sort suffixes

find LCP between adjacent suffixes in sorted order

Challenge

Bad input: longest repeated substring very long.

- Ex: same letter repeated N times.
- Ex: two copies of the same Java codebase.

form suffixes

0	t w i n s t w i n s
1	w i n s t w i n s
2	i n s t w i n s
3	n s t w i n s
4	s t w i n s
5	t w i n s
6	w i n s
7	i n s
8	n s
9	s

sorted suffixes

9	i n s
8	i n s t w i n s
7	n s
6	n s t w i n s
5	s
4	s t w i n s
3	t w i n s
2	t w i n s t w i n s
1	w i n s
0	w i n s t w i n s

D = length of longest match

LRS needs at least 1 + 2 + 3 + ... + D character compares.

Running time. Quadratic (or worse) in D for LRS (also for sort).

Manber-Myers MSD Algorithm

> Suffix sorting in linearithmic time

- Step
  - Phase 0---sort on firts character using key-indexed counting sort
  - Phase i ---given array of suffixes sorted on first 2^{i-1} characters, create array of suffixes sorted on first 2^i characters
- constant-time string compare by indexing into inverse