CS2010: Data Structures and Algorithms II

Strings

Ivana.Dusparic@scss.tcd.ie

Outline of String algorithms

- String sorting algorithms
  - Exploit properties of strings to speed up sorting
  - Radix sort
- Tries
  - Data structures for searching with string keys – more efficient than general purpose ones previously discussed, eg hashtables, search trees
- Substring search
  - Interesting problem- multiple approaches illustrating different algorithm design techniques
- Regular expressions
  - Searching for incomplete patterns rather than exact substrings
- Data compression
  - Save storage, faster network transfer
  - Run length encoding, Huffman compression

What are strings?

- Sequences of characters
  - Text
  - Genome sequences
- What are characters then?
  - In C – char data type is 8 bit integer, 7-bit ascii, 256 characters max
  - In Java – char data type, 16 bit unsigned int, range '\u0000' to '\uffff' (0 to 65,535)

java.lang.String

- Immutable sequence of characters
- Implements Comparable
- Implements CharSequences
  - Methods length() – number of characters
  - charAt(i) – returns the i-th character

  - Concatenation- concatenate one string to the end of another java.lang.String
  - Substring – extract a subsequence of characters

Immutable?

› String cannot be modified
› What happens here then?

String test = “blah”;
test = “meh”;
System.out.println(test); //what does this print?

New string is created and reference test now points to it instead of the old one – reference to “blah” text has been lost, it’s still unchanged somewhere in memory

Memory leaks/garbage collection

Other implications of immutability

- Security
  - Parameters in many methods which could introduce vulnerability – security threats, eg network connection is passed a string – it could be modified to connect to a different machine, or a modified file name can be passed in etc
- Thread-safe
  - No need for synchronisation if shared between threads – no thread can modify it
- Can be used as keys in symbol tables
- Can calculate and save hashcode – efficiency

String implementation

char [] value

int offset – index of first char in the array (substrings)

int length - saved for efficiency so it doesn't have to be calculated every time we need it

int hash - calculated as s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1]

Constructors String(), String (char[] value, int offset, int count)

Cost of String operations

operation	Java	running time
length	`s.length()`	1
indexing	`s.charAt(i)`	1
concatenation	`s + t`	$M + N$

Because String is immutable, concat creates a new copy of the string and adds new String to it

Cost of String operations

› How long does it take to reverse a String?

```
public static String reverse(String s)
{
   String rev = "";
   for (int i = s.length() - 1; i >= 0; i--)
      rev += s.charAt(i);
   return rev;
}
```

quadratic time

› Alternative – mutable sequence of characters, StringBuilder

```
public static String reverse(String s)
{
   StringBuilder rev = new StringBuilder();
   for (int i = s.length() - 1; i >= 0; i--)
      rev.append(s.charAt(i));
   return rev.toString();
}
```

linear time

Substring operation

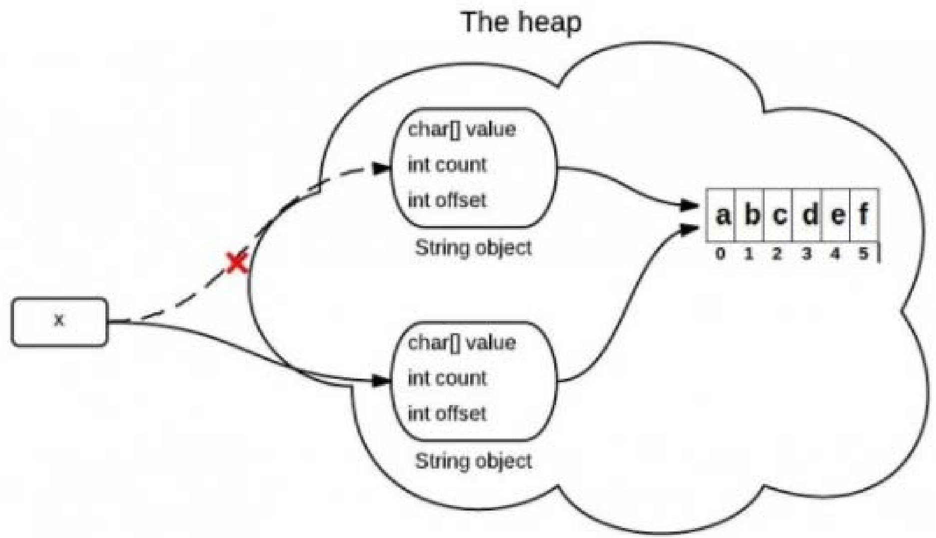
› Java 6 vs Java 7

```
String x = "abcdef";
x = x.substring(1,3);
System.out.println(x);
```

https://www.programcreek.com/2013/09/the-substring-method-in-jdk-6-and-jdk-7/

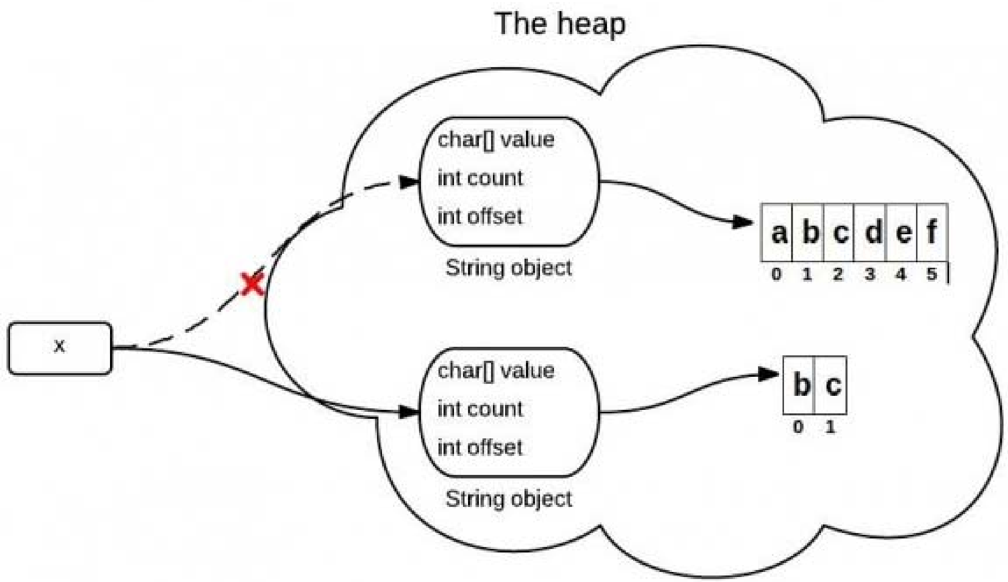
Substring operation in Java 6

- Cheap (cost of 1)
- Memory leaks – unused portion of the string cant be garbage collected
- Points to same value
- Count modified
- Offset modified



Substring operation in Java 7 (now)

- Cost of N, new string needs to be created- characters copied into it
- Old string can be garbage collected



Comparing 2 Strings

Q. How many character compares to compare two strings of length W ?

p	r	e	f	e	t	c	h
0	1	2	3	4	5	6	7
p	r	e	f	i	x	e	s

Running time. Proportional to length of longest common prefix.

- Proportional to W in the worst case.
- But, often sublinear in W.

Different Alphabets

- Performance depends on the size of the alphabet (i.e., unique characters)

Digital key. Sequence of digits over fixed alphabet.

Radix. Number of digits R in alphabet.

name	R()	lgR()	characters
BINARY	2	1	01
OCTAL	8	3	01234567
DECIMAL	10	4	0123456789
HEXADECIMAL	16	4	0123456789ABCDEF
DNA	4	2	ACTG
LOWERCASE	26	5	abcdefghijklmnopqrstuvwxyz
UPPERCASE	26	5	ABCDEFGHIJKLMNOPQRSTUVWXYZ
PROTEIN	20	5	ACDEFGHIKLMNPQRSTVWY
BASE64	64	6	ABCDEFGHIJKLMNOPQRSTUVWXYZabcdef ghijklmnopqrstuvwxyz0123456789+/
ASCII	128	7	*ASCII characters*
EXTENDED_ASCII	256	8	*extended ASCII characters*
UNICODE16	65536	16	*Unicode characters*

Key-Indexed Counting

Review: summary of the performance of sorting algorithms

Frequency of operations.

algorithm	guarantee	random	extra space	stable?	operations on keys
insertion sort	½ N^2	¼ N^2	1	✔	`compareTo()`
mergesort	$N \lg N$	$N \lg N$	N	✔	`compareTo()`
quicksort	$1.39\ N \lg N$ *	$1.39\ N \lg N$	$c \lg N$		`compareTo()`
heapsort	$2\ N \lg N$	$2\ N \lg N$	1		`compareTo()`

* probabilistic

Lower bound. $\sim N \lg N$ compares required by any compare-based algorithm.

Q. Can we do better (despite the lower bound)?

A. Yes, if we don't depend on key compares. ← use array accesses to make R-way decisions (instead of binary decisions)

Key-indexed counting

- Basis for other more complex sorting algorithms
  - LSD and MSD (least and most significant digit)
- Specialized sorting algorithm which works best when the following conditions are met:
  - Input consists of collection of n items
  - Maximum possible value of each of the individual item is K

Key-indexed counting: assumptions about keys

Assumption. Keys are integers between 0 and $R - 1$.

Implication. Can use key as an array index.

Applications.

- Sort string by first letter.
- Sort class roster by section.
- Sort phone numbers by area code.
- Subroutine in a sorting algorithm. [stay tuned]

Remark. Keys may have associated data $\Rightarrow$ can't just count up number of keys of each value.

input		sorted result (by section)	
name	*section*		
Anderson	2	Harris	1
Brown	3	Martin	1
Davis	3	Moore	1
Garcia	4	Anderson	2
Harris	1	Martinez	2
Jackson	3	Miller	2
Johnson	4	Robinson	2
Jones	3	White	2
Martin	1	Brown	3
Martinez	2	Davis	3
Miller	2	Jackson	3
Moore	1	Jones	3
Robinson	2	Taylor	3
Smith	4	Williams	3
Taylor	3	Garcia	4
Thomas	4	Johnson	4
Thompson	4	Smith	4
White	2	Thomas	4
Williams	3	Thompson	4
Wilson	4	Wilson	4

keys are small integers

Key-indexed counting demo

Goal. Sort an array a[] of N integers between 0 and $R - 1$.

R = 6

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move items.
- Copy back into original array.

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
   count[a[i]+1]++;

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

for (int i = 0; i < N; i++)
   aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
   a[i] = aux[i];
```

i	a[i]
0	d
1	a
2	c
3	f
4	f
5	b
6	d
7	b
8	f
9	b
10	e
11	a

use a for 0
b for 1
c for 2
d for 3
e for 4
f for 5

Key-indexed counting demo

Goal. 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 key as index to move items.
- Copy back into original array.

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
   count[a[i]+1]++;

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

for (int i = 0; i < N; i++)
   aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
   a[i] = aux[i];
```

count
requencies

i	a[i]
0	d
1	a
2	c
3	f
4	f
5	b
6	d
7	b
8	f
9	b
10	e
11	a

offset by 1
[stay tuned]

r	count[r]
a	0
b	2
c	3
d	1
e	2
f	1
-	3

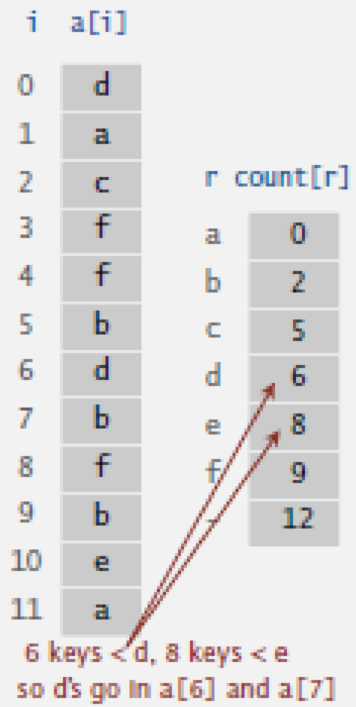
Key-indexed counting demo

Goal. 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 key as index to move items.
- Copy back into original array.

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
   count[a[i]+1]++;

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

for (int i = 0; i < N; i++)
   aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
   a[i] = aux[i];
```

compute cumulates



Key-indexed counting demo

Goal. 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 key as index to move items.**
- Copy back into original array.

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
   count[a[i]+1]++;

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

for (int i = 0; i < N; i++)
   aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
   a[i] = aux[i];
```

move items

i	a[i]
0	d
1	a
2	c
3	f
4	f
5	b
6	d
7	b
8	f
9	b
10	e
11	a

r	count[r]
a	2
b	5
c	6
d	8
e	9
f	12
-	12

i	aux[i]
0	a
1	a
2	b
3	b
4	b
5	c
6	d
7	d
8	e
9	f
10	f
11	f

Key-indexed counting demo

Goal. 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 key as index to move items.
- Copy back into original array.

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
   count[a[i]+1]++;

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

for (int i = 0; i < N; i++)
   aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
   a[i] = aux[i];
```

copy back

i	a[i]
0	a
1	a
2	b
3	b
4	b
5	c
6	d
7	d
8	e
9	f
10	f
11	f

r	count[r]
a	2
b	5
c	6
d	8
e	9
f	12
-	12

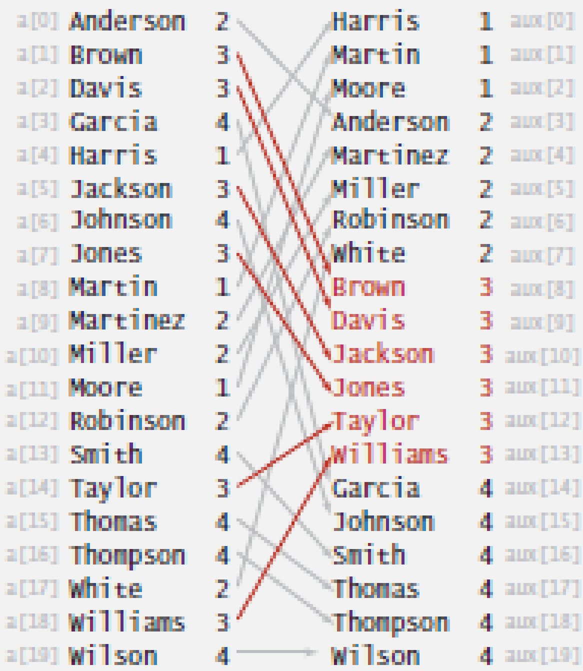
i	aux[i]
0	a
1	a
2	b
3	b
4	b
5	c
6	d
7	d
8	e
9	f
10	f
11	f

Key-indexed counting: analysis

Proposition. Key-indexed takes time proportional to $N + R$.

Proposition. Key-indexed counting uses extra space proportional to $N + R$.

Stable? ✔



Radix Sorts – LSD and MSD

Radix sorts

- Non-comparative integer sorting algorithm
- Sorts data with integer keys by grouping keys by the individual digits which share the same significant position and value.
- LSD radix sort
  - short keys come before longer keys
  - keys of the same length are sorted lexicographically
  - Ie, normal order of integer representations, such as the sequence 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11
- MSD radix sort
  - lexicographic order, which is suitable for sorting strings, such as words, or fixed-length integer representations
  - A sequence such as "b, c, d, e, f, g, h, i, j, ba" would be lexicographically sorted as "b, ba, c, d, e, f, g, h, i, j"
  - (1 to 10 would be output as 1, 10, 2, 3, 4, 5, 6, 7, 8, 9,)

LSD Sort

LSD Sort – Sort by Least Significant Digit first

LSD string (radix) sort.

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

	input	sort key (d = 2)	sort key (d = 1)	sort key (d = 0)
0	d a b	d a b	d a b	a c e
1	a d d	c a b	c a b	a d d
2	c a b	e b b	f a d	b a d
3	f a d	a d d	b a d	b e d
4	f e e	f a d	d a d	b e e
5	b a d	b a d	e b b	c a b
6	d a d	d a d	a c e	d a b
7	b e e	f e d	a d d	d a d
8	f e d	b e d	f e d	e b b
9	b e d	f e e	b e d	f a d
10	e b b	b e e	f e e	f e d
11	a c e	a c e	b e e	f e e

sort is stable
(arrows do not cross)

LSD stability and correctness

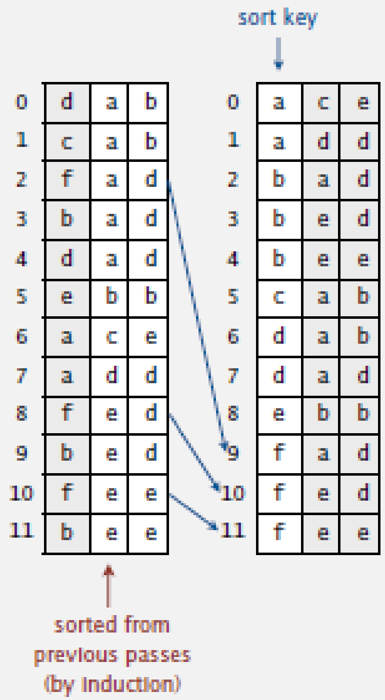
LSD string sort: correctness proof

Proposition. LSD sorts fixed-length strings in ascending order.

Pf. [by induction on i]

After pass *i*, strings are sorted by last *i* characters.

- If two strings differ on sort key, key-indexed sort puts them in proper relative order.
- If two strings agree on sort key, stability keeps them in proper relative order.



Proposition. LSD sort is stable.

Pf. Key-indexed counting is stable.

LSD in Java

```java
public class LSD
{
   public static void sort(String[] a, int W)
   {
      int R = 256;
      int N = a.length;
      String[] aux = new String[N];

      for (int d = W-1; d >= 0; d--)
      {
         int[] count = new int[R+1];
         for (int i = 0; i < N; i++)
            count[a[i].charAt(d) + 1]++;
         for (int r = 0; r < R; r++)
            count[r+1] += count[r];
         for (int i = 0; i < N; i++)
            aux[count[a[i].charAt(d)]++] = a[i];
         for (int i = 0; i < N; i++)
            a[i] = aux[i];
      }
   }
}
```

fixed-length W strings

radix R

do key-indexed counting
for each digit from right to left

key-indexed counting

Performance wrt other sorting algorithms

Frequency of operations.

algorithm	guarantee	random	extra space	stable?	operations on keys
insertion sort	½ N^2	¼ N^2	1	✔	compareTo()
mergesort	$N \lg N$	$N \lg N$	N	✔	compareTo()
quicksort	1.39 $N \lg N$ *	1.39 $N \lg N$	$c \lg N$		compareTo()
heapsort	2 $N \lg N$	2 $N \lg N$	1		compareTo()
LSD sort †	2 $W (N + R)$	2 $W (N + R)$	$N + R$	✔	charAt()

* probabilistic
† fixed-length W keys

LSD Performance

- Key indexed counting
  - 11n+4R+1
    - Initialize arrays n+R+1
    - First loop 3n
    - Second loop 3R
    - Third loop 5n
    - Fourth loop 2n
- LSD
  - 10wn+n+WR
  - W x LSD apart from third loop which is just 1 x rather than LSD x
  - R usually much smaller than N, so proportional to wn – linear!

Example - sorting large integer arrays

String sorting interview question

Problem. Sort one million 32-bit integers.

Ex. Google (or presidential) interview.

Which sorting method to use?

- Insertion sort.
- Mergesort.
- Quicksort.
- Heapsort.
- LSD string sort.

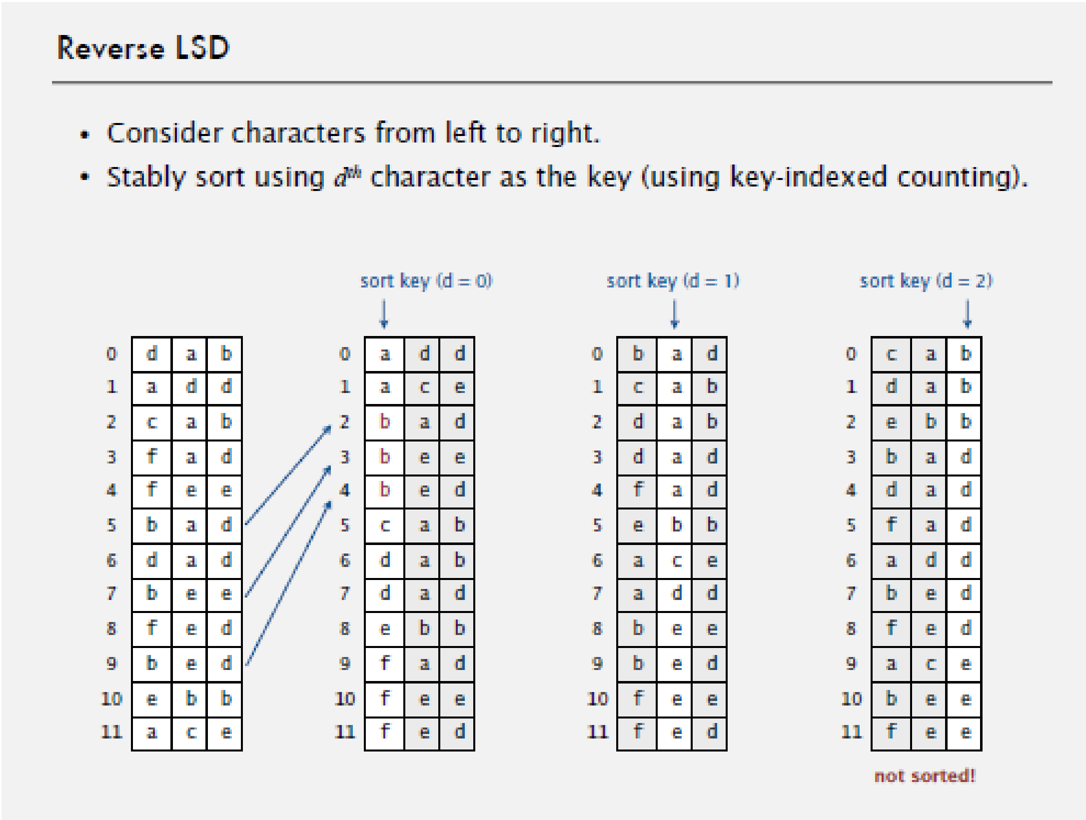
LSD Sorting large integer arrays

- https://algs4.cs.princeton.edu/51radix/LSD.java.html
- Break up 32-bit integer in 4 8-bit characters
- Use bit shifting and masking to isolate characters to sort by
- What if strings are not same length? MSD!

MSD Sort

MSD – Sort by Most Significant Digit first

› Does simply reversing LSD work?



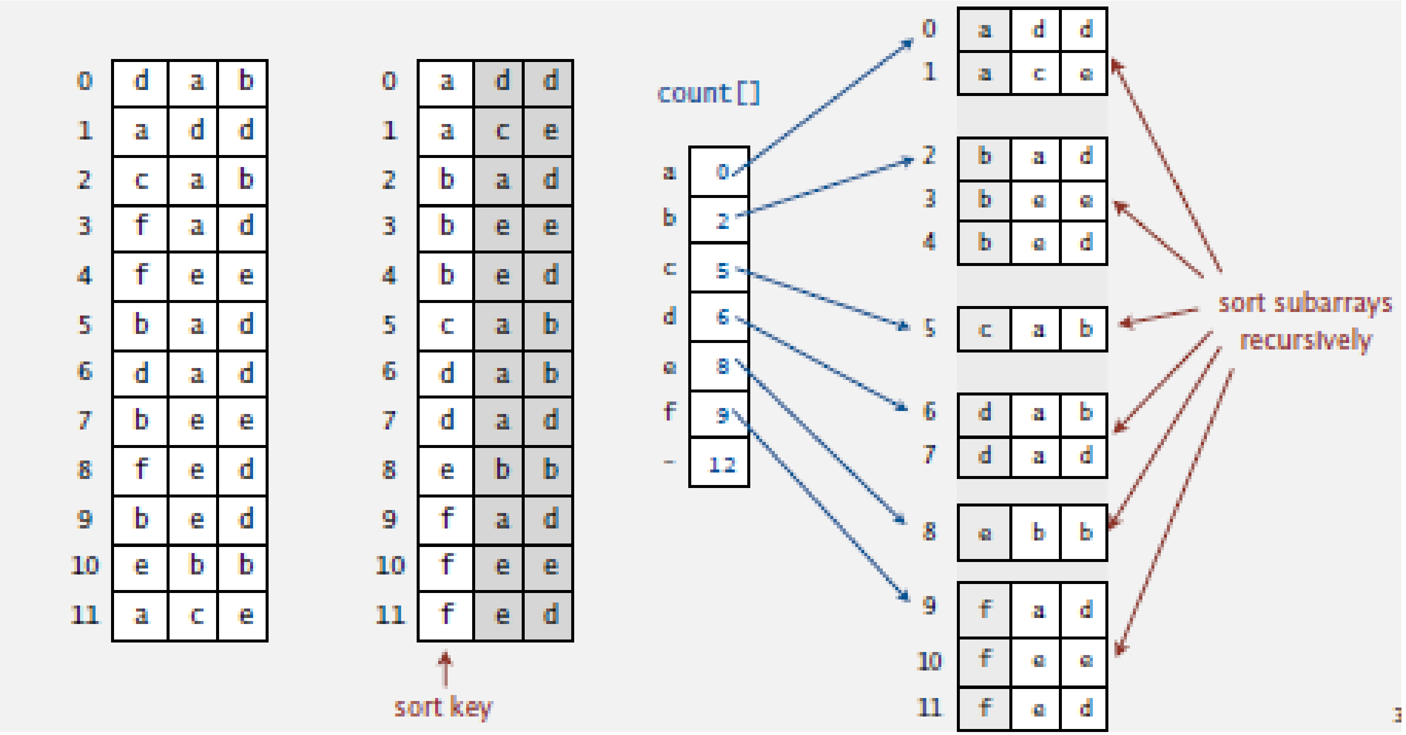
MSD string sort

› Similar to quicksort

› Partition array into R (radix) pieces according to the first character (most significant digit) using key-indexed counting

› Recursively sort all strings that start with each character (key-indexed counts delineate subarrays to sort)

MSD String sort



MSD string sort: example

Trace of recursive calls for MSD string sort (no cutoff for small subarrays, subarrays of size 0 and 1 omitted)

Variable length Strings

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

why smaller?

0	s	e	a	-1						
1	s	e	a	s	h	e	l	l	s	-1
2	s	e	l	l	s	-1				
3	s	h	e	-1						
4	s	h	e	-1						
5	s	h	e	l	l	s	-1			
6	s	h	o	r	e	-1				
7	s	u	r	e	l	y	-1			

she before shells

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

C strings. Have extra char '\0' at end ⇒ no extra work needed.

MSD String sort – Java Implementation

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

private static void sort(String[] a, String[] aux, int lo, int hi, int d)
{
   if (hi <= lo) return;
   int[] count = new int[R+2];
   for (int i = lo; i <= hi; i++)
      count[charAt(a[i], d) + 2]++;
   for (int r = 0; r < R+1; r++)
      count[r+1] += count[r];
   for (int i = lo; i <= hi; i++)
      aux[count[charAt(a[i], d) + 1]++] = a[i];
   for (int i = lo; i <= hi; i++)
      a[i] = aux[i - lo];

   for (int r = 0; r < R; r++)
      sort(a, aux, lo + count[r], lo + count[r+1] - 1, d+1);
}
```

recycles aux[] array but not count[] array

key-indexed counting

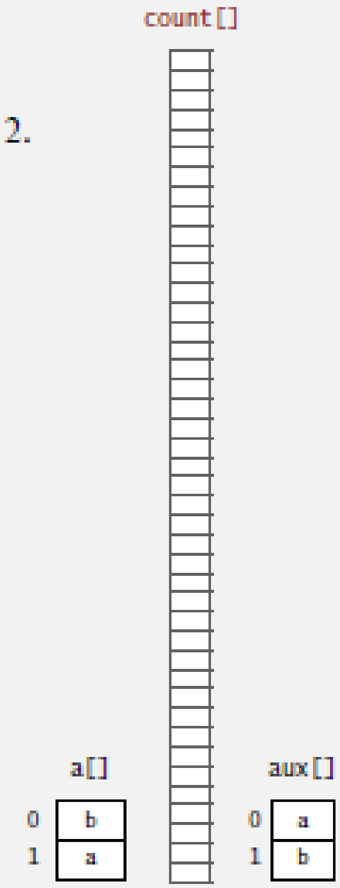
sort R subarrays recursively

MSD – improvements

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.



Yep, you guessed it – cutoff to Insertion sort

Cutoff to insertion sort

Solution. Cutoff to insertion sort for small subarrays.

- Insertion sort, but start at d^{th} character.

```
private static void sort(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);
}
```

- Implement `less()` so that it compares starting at d^{th} character.

```
private static boolean less(String v, String w, int 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();
}
```

MSD Performance

Frequency of operations.

algorithm	guarantee	random	extra space	stable?	operations on keys
insertion sort	½ N^2	¼ N^2	1	✔	compareTo()
mergesort	$N \lg N$	$N \lg N$	N	✔	compareTo()
quicksort	$1.39\, N \lg N$ *	$1.39\, N \lg N$	$c \lg N$		compareTo()
heapsort	$2\, N \lg N$	$2\, N \lg N$	1		compareTo()
LSD sort †	$2\, W (N + R)$	$2\, W (N + R)$	$N + R$	✔	charAt()
MSD sort ‡	$2\, W (N + R)$	$N \log_R N$	$N + D\, R$	✔	charAt()

D = function-call stack depth
(length of longest prefix match)

* probabilistic
† fixed-length W keys
‡ average-length W keys

MSD improvements - American flag sort

- Analogy to Dutch national flag, partition array into many "stripes"
- In-place variant of radix sort that distributes items into hundreds of buckets
- How?
- Cut off to insertion sort
- Replaces recursion with explicit stack
- In-place, eliminate auxiliary array (no stability)

MSD vs Quicksort for String sorting

Disadvantages of MSD string sort.

- Extra space for `aux[]`.
- Extra space for `count[]`.
- Inner loop has a lot of instructions.
- Accesses memory "randomly" (cache inefficient).

Disadvantage of quicksort.

- Linearithmic number of string compares (not linear).
- Has to rescan many characters in keys with long prefix matches.

doesn't rescan characters

tight inner loop, cache friendly

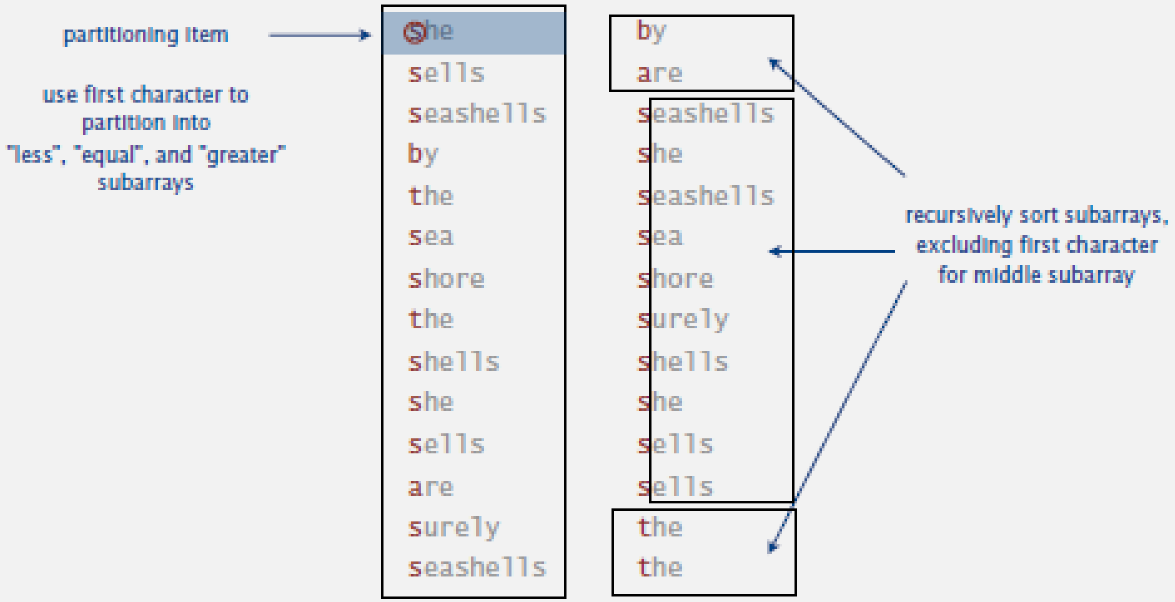
Goal. Combine advantages of MSD and quicksort.

› 3-way string quicksort – addressed inefficiency of both

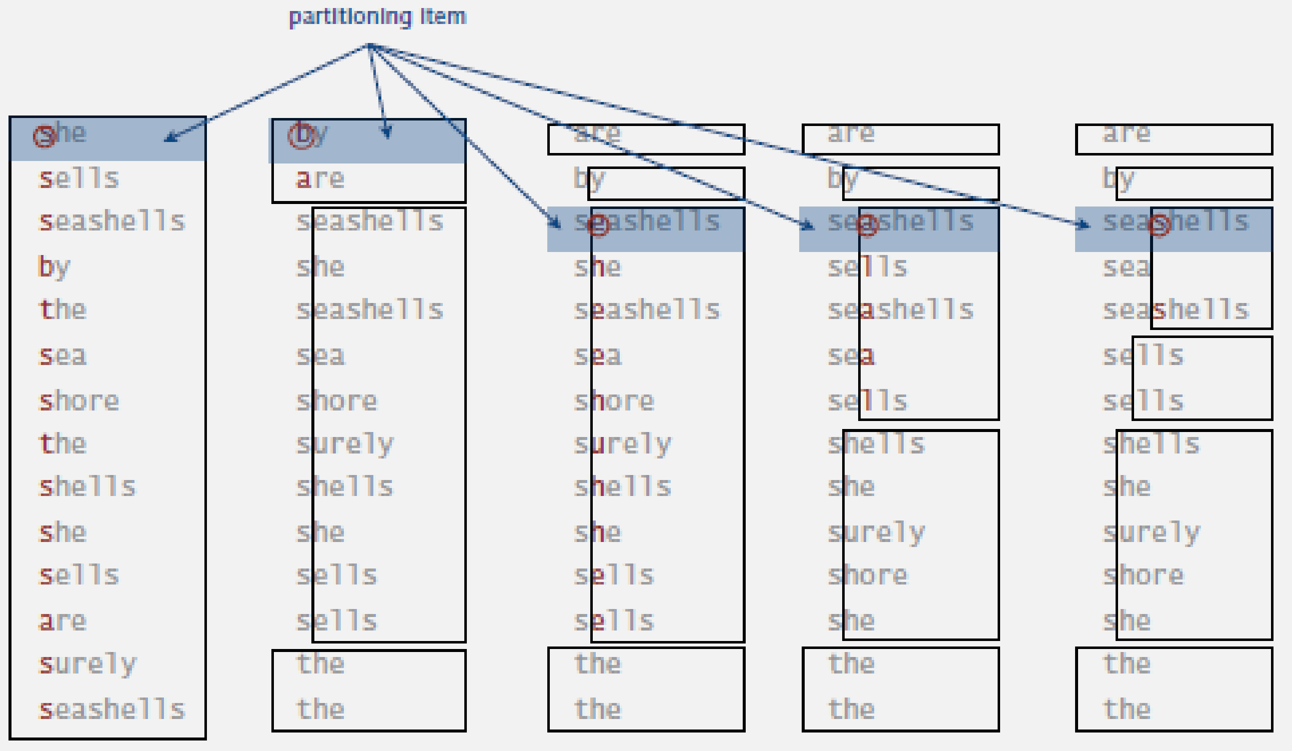
3-way string quicksort

Overview. Do 3-way partitioning on the d^{th} character.

- Less overhead than R-way partitioning in MSD string sort.
- Does not re-examine characters equal to the partitioning char.
  (but does re-examine characters not equal to the partitioning char)



3- way string quicksort



Trace of first few recursive calls for 3-way string quicksort (subarrays of size 1 not shown)

3-way string quicksort in java

- charAt instead of compare

```
private static void sort(String[] a)
{  sort(a, 0, a.length - 1, 0);  }

private static void sort(String[] a, int lo, int hi, int d)
{
   if (hi <= lo) return;                            3-way partitioning
   int lt = lo, gt = hi;                            (using dth character)
   int v = charAt(a[lo], d);
   int i = lo + 1;
   while (i <= gt)                       to handle variable-length strings
   {
      int t = charAt(a[i], d);
      if      (t < v) exch(a, lt++, i++);
      else if (t > v) exch(a, i, gt--);
      else            i++;
   }

   sort(a, lo, lt-1, d);
   if (v >= 0) sort(a, lt, gt, d+1);   <--  sort 3 subarrays recursively
   sort(a, gt+1, hi, d);
}
```

Summary of string sorts

Frequency of operations.

algorithm	guarantee	random	extra space	stable?	operations on keys
insertion sort	$\frac{1}{2} N^2$	$\frac{1}{4} N^2$	1	✔	compareTo()
mergesort	$N \lg N$	$N \lg N$	N	✔	compareTo()
quicksort	$1.39\, N \lg N$ *	$1.39\, N \lg N$	$c \lg N$		compareTo()
heapsort	$2\, N \lg N$	$2\, N \lg N$	1		compareTo()
LSD sort †	$2\, W (N + R)$	$2\, W (N + R)$	$N + R$	✔	charAt()
MSD sort ‡	$2\, W (N + R)$	$N \log_R N$	$N + D R$	✔	charAt()
3-way string quicksort	$1.39\, W N \lg R$ *	$1.39\, N \lg N$	$\log N + W$		charAt()

String sorting algorithms – when to use which?

- Insertion
  - Small arrays, arrays in (almost)order
- Quick
  - General purpose when space is tight
- Merge
  - General purpose stable
- 3-way quick
  - Large number of equal keys
- LSD
  - Short fixed-length strings
- MSD
  - Random strings
- 3-way string quicksort
  - General purpose, strings with long pre-fix matches

Check if 2 strings are anagrams

- two words are anagrams if they contain the same characters
  - “abc” and “cba” are anagrams