Radix Sorting

LSD radix sort

MSD radix sort

3-way radix quicksort

Suffix sorting

Reference: Chapter 13, Algorithms in Java, 3rd Edition, Robert Sedgewick.

Radix Sorting

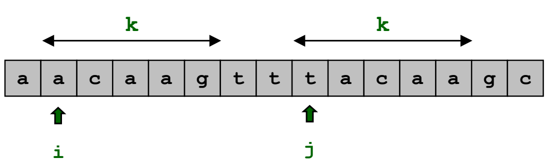
Radix sorting.
- Specialized sorting solution for strings.
- Same ideas for bits, digits, etc.

Applications.
- Sorting strings.
- Full text indexing.
- Plagiarism detection.
- Burrows-Wheeler transform. stay tuned
- Computational molecular biology.

An Application: Redundancy Detector

Longest repeated substring.
- Given a string of N characters, find the longest repeated substring.
- Ex: a a c a a g t t t a c a a g c
- Application: computational molecular biology.

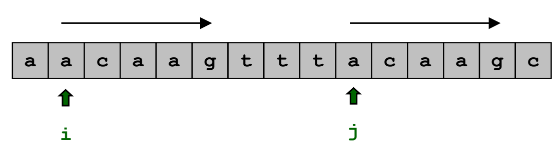
Dumb brute force.
- Try all indices i and j, and all match lengths k and check.
- $O(W N^3)$ time, where W is length of longest match.



An Application: Redundancy Detector

Longest repeated substring.
- Given a string of N characters, find the longest repeated substring.
- Ex: a a c a a g t t t a c a a g c
- Application: computational molecular biology.

Brute force.
- Try all indices i and j for start of possible match, and check.
- $O(W N^2)$ time, where W is length of longest match.



A Sorting Solution

Suffix sort.

- Form N suffixes of original string.
- Sort to bring longest repeated substrings together.

```
a a c a a g t t t a c a a g c        a a c a a g t t t a c a a g c
a c a a g t t t a c a a g c          a a g c
c a a g t t t a c a a g c            a a g t t t a c a a g c
a a g t t t a c a a g c              a c a a g c
a g t t t a c a a g c                a c a a g t t t a c a a g c
g t t t a c a a g c                  a g c
t t t a c a a g c                    a g t t t a c a a g c
t t a c a a g c                      c
t a c a a g c                        c a a g c
a c a a g c                          c a a g t t t a c a a g c
c a a g c                            g c
a a g c                              g t t t a c a a g c
a g c                                t a c a a g c
g c                                  t t a c a a g c
c                                    t t t a c a a g c
```


Suffix Sorting: Java Implementation

Java implementation.

- We use Java `String` library functions to simplify code.
- Could use `byte` array to store ASCII string, and array of pointers into the `byte` array to save memory.

```
public class SuffixSorter {
   public static void main(String[] args) {
      In stdin = new In();                          // read input
      String s = stdin.readAll();
      int N = s.length();

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

      Arrays.sort(suffixes);                        // sort and find
      findLongestMatch(suffixes);                   // longest match
                                                    // (bottleneck)
   }
}
```


Diversion: String Implementation in Java

Java implementation of String.

- Immutability: use as Key in symbol table, fast substring.
- Memory for virgin string: 28 + 2N bytes (!)

```
public final class String implements Comparable {
   private char[] value;   // characters
   private int offset;     // index of first char into array
   private int count;      // length of string
   private int hash;       // cache of hashCode

   private String(int offset, int count, char[] value) {
       this.offset = offset;
       this.count  = count;
       this.value  = value;
   }
   public String substring(int from, int to) {
      return new String(offset + from, to - from, value);
   }
   . . .
}
```


String Sorting Performance

	String Sort	Suffix (sec)
	Worst Case	Moby Dick
Brute	$W N^2$	36,000 §
Quicksort	$W N \log N$ †	9.5

N = number of strings.
1.2 million for Moby Dick.
191 thousand for Aesop's Fables.

§ estimate
† probabilistic guarantee.

String Sorting

Notation.

- String = variable length sequence of characters.
- W = max # characters per string.
- N = # input strings.
- R = radix (256 for extended ASCII, 65,536 for UNICODE).

Java syntax.

- Array of strings: `String[] a;`
- The i^{th} string: `a[i]`
- The d^{th} character of the i^{th} string: `a[i].charAt(d)`
- Strings to be sorted: `a[lo], ..., a[hi]`

Key Indexed Counting

Key indexed counting.

- Count frequencies of each letter. (0^{th} character)

```
int[] count = new int[256+1];
for (int i = L; i <= R; i++) {
   char c = a[i].charAt(d);
   count[c+1]++;
}
```

frequencies

d = 0;

a

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

count

a	0
b	2
c	3
d	1
e	2
f	1
g	3

Key Indexed Counting

Key indexed counting.

- Count frequencies of each letter. (0^{th} character)
- Compute cumulative frequencies.

```
for (int i = 1; i < 256; i++)
   count[i] += count[i-1];
```

cumulative counts

a

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

count

a	0	a	0
b	2	b	2
c	3	c	5
d	1	d	6
e	2	e	8
f	1	f	9
g	3	g	11

Key Indexed Counting

Key indexed counting.

- Count frequencies of each letter. (0^{th} character)
- Compute cumulative frequencies.
- Use cumulative frequencies to rearrange strings.

```
for (int i = L; i <= R; i++) {
   char c = a[i].charAt(d);
   temp[count[c]++] = a[i];
}
```

rearrange

d = 0;

a

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

count

a	0	a	0
b	2	b	2
c	3	c	5
d	1	d	6
e	2	e	8
f	1	f	9
g	3	g	11

temp

0			
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			

Key Indexed Counting

Key indexed counting.

- Count frequencies of each letter. (0th character)
- Compute cumulative frequencies.
- Use cumulative frequencies to rearrange strings.

```
for (int i = L; i <= R; i++) {
   char c = a[i].charAt(d);
   temp[count[c]++] = a[i];
}
```

rearrange

```
d = 0;
```

a

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

count

a	0	a	0
b	2	b	2
c	3	c	5
d	1	d	7
e	2	e	8
f	1	f	9
g	3	g	11

temp

0			
1			
2			
3			
4			
5			
6	d	a	b
7			
8			
9			
10			
11			

Key Indexed Counting

Key indexed counting.

- Count frequencies of each letter. (0th character)
- Compute cumulative frequencies.
- Use cumulative frequencies to rearrange strings.

```
for (int i = L; i <= R; i++) {
   char c = a[i].charAt(d);
   temp[count[c]++] = a[i];
}
```

rearrange

```
d = 0;
```

a

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

count

a	0	a	2
b	2	b	5
c	3	c	6
d	1	d	8
e	2	e	9
f	1	f	12
g	3	g	11

temp

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

Key Indexed Counting

Key indexed counting.

- Count frequencies of each letter. (0th character)
- Compute cumulative frequencies.
- Use cumulative frequencies to rearrange strings.

```
for (int i = L; i <= R; i++)
  a[i] = temp[i - L];
```

copy back

a

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

count

a	0	a	2
b	2	b	5
c	3	c	6
d	1	d	8
e	2	e	9
f	1	f	12
g	3	g	11

temp

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

LSD Radix Sort

Least significant digit radix sort.

- Ancient method used for card-sorting.
- Consider digits from right to left:
  - use key-indexed counting to STABLE sort by character

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

LSD Radix Sort

Least significant digit radix sort.

- Ancient method used for card-sorting.
- Consider digits from right to left:
  - use key-indexed counting to STABLE sort by character

```
public static void lsd(String[] a, int lo, int hi) {
   for (int d = W-1; d >= 0; d--) {
      // do key-indexed counting sort on digit d
      ...
   }
}
```

Fixed length strings (length = W)

LSD Radix Sort: Correctness

Proof 1. (left-to-right).

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

Proof 2. (right-to-left)

- If the characters not yet examined differ, it doesn't matter what we do now.
- If the characters not yet examined agree, later pass won't affect order.

now	sob	cab	ace
for	nob	wad	ago
tip	cab	tag	and
ilk	wad	jam	bet
dim	and	rap	cab
tag	ace	tap	caw
jot	wee	tar	cue
sob	cue	was	dim
nob	fee	caw	dug
sky	tag	raw	egg
hut	egg	jay	fee
ace	gig	ace	few
bet	dug	wee	for
men	ilk	fee	gig
egg	owl	men	hut
few	dim	bet	ilk
jay	jam	few	jam
owl	men	egg	jay
joy	ago	ago	jot
rap	tip	gig	joy
gig	rap	dim	men
wee	tap	tip	nob
was	for	sky	now
cab	tar	ilk	owl
wad	was	and	rap
tap	jot	sob	raw
caw	hut	nob	sky
cue	bet	for	sob
fee	you	jot	tag
raw	now	you	tap
ago	few	now	tar
tar	caw	joy	tip
jam	raw	cue	wad
dug	sky	dug	was
you	jay	hut	wee
and	joy	owl	you

LSD Radix Sort Correctness

Running time. $\Theta(W(N + R))$.

why doesn't it violate N log N lower bound?

Advantage. Fastest sorting method for random fixed length strings.

Disadvantages.

- Accesses memory "randomly."
- Inner loop has a lot of instructions.
- Wastes time on low-order characters.
- Doesn't work for variable-length strings.
- Not much semblance of order until very last pass.

Goal: find fast algorithm for variable length strings.

MSD Radix Sort

Most significant digit radix sort.

- Partition file into 256 pieces according to first character.
- Recursively sort all strings that start with the same character, etc.

How to sort on d^{th} character?

- Use key-indexed counting.

now	a	ce	ac	e	ace
for	a	go	ag	o	ago
tip	a	nd	an	d	and
ilk	b	et	be	t	bet
dim	c	ab	ca	b	cab
tag	c	aw	ca	w	caw
jot	c	ue	cu	e	cue
sob	d	im	di	m	dim
nob	d	ug	du	g	dug
sky	e	gg	eg	g	egg
hut	f	or	fe	w	fee
ace	f	ee	fe	e	few
bet	f	ew	fo	r	for
men	g	ig	gi	g	gig
egg	h	ut	hu	t	hut
few	i	lk	il	k	ilk
jay	j	am	ja	y	jam
owl	j	ay	ja	m	jay
joy	j	ot	jo	t	jot
rap	j	oy	jo	y	joy
gig	m	en	me	n	men
wee	n	ow	no	w	nob
was	n	ob	no	b	now
cab	o	wl	ow	l	owl
wad	r	ap	ra	p	rap
caw	s	ob	sk	y	sky
cue	s	ky	so	b	sob
fee	t	ip	ta	g	tag
tap	t	ag	ta	p	tap
ago	t	ap	ta	r	tar
tar	t	ar	ti	p	tip
jam	w	ee	wa	d	wad
dug	w	as	wa	s	was
and	w	ad	we	e	wee

MSD Radix Sort Implementation

```
public static void msd(String[] a, int lo, int hi) {
   msd(a, lo, hi, 0);
}

private static void msd(String[] a, int lo, int hi, int d) {
   if (hi <= lo) return;

   // do key-indexed counting sort on digit d
   int[] count = new int[256+1];
   ...

   // recursively sort 255 subfiles - assumes '\0' terminated
   for (int i = 0; i < 255; i++)
      msd(a, L + count[i], L + count[i+1] - 1, d+1);
}
```


String Sorting Performance

	String Sort	Suffix (sec)
	Worst Case	Moby Dick
Brute	$W N^2$	36,000 §
Quicksort	W N log N †	9.5
LSD *	W(N + R)	-
MSD	W(N + R)	395
MSD with cutoff	W(N + R)	6.8

R = radix.
W = max length of string.
N = number of strings.

§ estimate
* assumes fixed length strings.
† probabilistic guarantee.

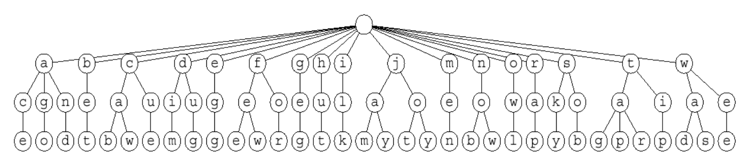
MSD Radix Sort Analysis

Disadvantages.
- Too slow for small files.
  - ASCII: 100x slower than insertion sort for N = 2
  - UNICODE: 30,000x slower for N = 2
- Huge number of recursive calls on small files.

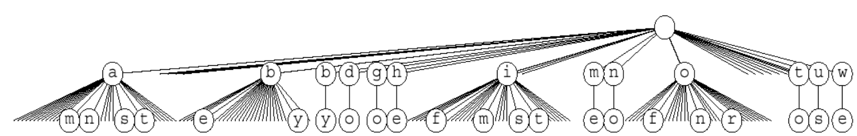
Solution: cutoff to insertion sort for small N.
- Competitive with quicksort for string keys.

Recursive Structure of MSD Radix Sort

Trie structure to describe recursive calls in MSD radix sort.



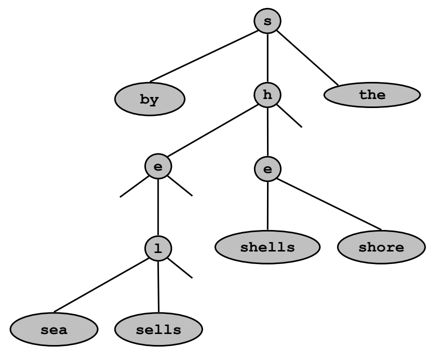
Problem: algorithm touches lots of empty nodes ala R-way tries.
- Tree can be as much as 256 times bigger than it appears!



Correspondence With Sorting Algorithms

Correspondence between trees and sorting algorithms.

- BSTs correspond to quicksort recursive partitioning structure.
- R-way tries corresponds to MSD radix sort.
- What corresponds to ternary search tries?

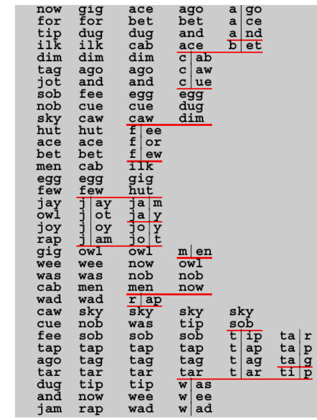


3-Way Radix Quicksort

Idea 1. Use d^{th} character to "sort" into 3 pieces instead of 256, and sort each piece recursively.

Idea 2. Keep all duplicates together in partitioning step.

Partition

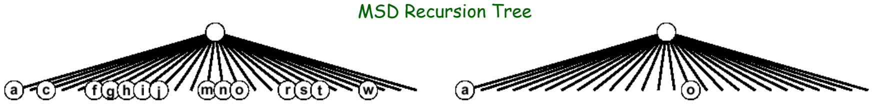


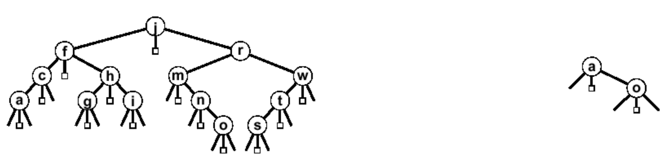
Algorithm

Recursive Structure of MSD Radix Sort vs. 3-Way Quicksort

3-way radix quicksort collapses empty links in MSD tree.

MSD Recursion Tree



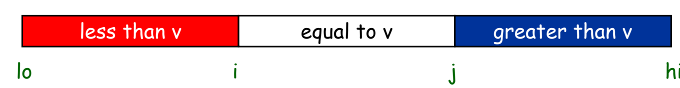


3-Way Radix Quicksort Recursion Tree

3-Way Partitioning

3-way partitioning.

- Natural way to deal with equal keys.
- 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



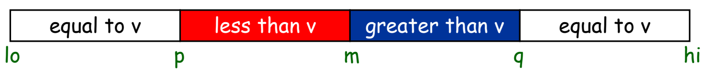
Dutch national flag problem.

- Not easy to implement efficiently. (Try it!)
- Not done in practical sorts before mid-1990s.
- Incorporated into Java system sort, C qsort.

3-Way Partitioning

Elegant solution to Dutch national flag problem.

- Partition elements into 4 parts:
  - no larger elements to left of m
  - no smaller elements to right of m
  - equal elements to left of p
  - equal elements to right of q



- Afterwards, swap equal keys into center.

All the right properties.

- Not much code.
- In-place.
- Linear if keys are all equal.
- Small overhead if no equal keys.

3-Way Radix Quicksort

```
private static void quicksortX(String a[], int lo, int hi, int d) {
   if (hi - lo <= 0) return;
   int i = lo-1, j = hi, p = lo-1, q = hi;
   char v = a[hi].charAt(d);

   while (i < j) {
      while (a[++i].charAt(d) < v) ;
      while (v < a[--j].charAt(d))
         if (j == lo) break;
      if (i > j) break;
      exch(a, i, j);
      if (a[i].charAt(d) == v) { p++; exch(a, p, i); }
      if (a[j].charAt(d) == v) { q--; exch(a, j, q); }
   }
   if (p == q) {
      if (v != '\0') quicksortX(a, lo, hi, d+1);
      return;
   }
   if (a[i].charAt(d) < v) i++;
   for (int k = lo; k <= p; k++, j--) exch(a, k, j);
   for (int k = hi; k >= q; k--, i++) exch(a, k, i);
   quicksortX(a, lo, j, d);
   if ((i == hi) && (a[i].charAt(d) == v)) i++;
   if (v != '\0') quicksortX(a, j+1, i-1, d+1);
   quicksortX(a, i, hi, d);
}
```

repeat until pointers cross

find i on left and j on right to swap

swap equal chars to left or right

special case for all equal chars

swap equal ones back to middle

sort 3 pieces recursively

Significance of 3-Way Partitioning

Equal keys omnipresent in applications when purpose of sort is to bring records with equal keys together.

- Finding collinear points.
- Sort population by age.
- Remove duplicates from mailing list.
- Sort job applicants by college attended.

Typical application.

- Huge file.
- Small number of key values.
- Randomized 3-way quicksort is LINEAR time. (Try it!)

Theorem. Quicksort with 3-way partitioning is OPTIMAL.
Proof. Ties cost to entropy. Beyond scope of 226.

Quicksort vs. 3-Way Radix Quicksort

Quicksort.

- 2N ln N string comparisons on average.
- Long keys are costly to compare if they differ only at the end, and this is common case!
- Absolutism, absolut, absolutely, absolute.

3-way radix quicksort.

- Avoids re-comparing initial parts of the string.
- Uses just "enough" characters to resolve order.
- 2 N ln N character comparisons on average for random strings.
- Sub-linear sort for large W since input is of size NW.

String Sorting Performance

	String Sort	Suffix Sort
	Worst Case	Moby Dick
Brute	W N^2	36,000 §
Quicksort	W N log N †	9.5
LSD *	W(N + R)	-
MSD	W(N + R)	395
MSD with cutoff	W(N + R)	6.8
3-Way Radix Quicksort	W N log N †	2.8

R = radix.
W = max length of string.
N = number of strings.

§ estimate
* fixed length strings only
† probabilistic guarantee

Suffix Sorting: Worst Case Input

Length of longest match small.

- 3-way radix quicksort rules!

Length of longest match very long.

- 3-way radix quicksort is quadratic.
- Two copies of Moby Dick.

Can we do better?

- Θ(N log N) ?
- Θ(N) ?

Observation. Must find longest repeated substring WHILE suffix sorting to beat N^2.

```
abcdefghi
abcdefghiabcdefghi
bcdefghi
bcdefghiabcdefghi
cdefghi
cdefghiabcdefgh
defghi
efghiabcdefghi
efghi
fghiabcdefghi
fghi
ghiabcdefghi
fhi
hiabcdefghi
hi
iabcdefghi
i
```

Input: "abcdeghiabcdefghi"

Suffix Sorting in N log N Time: Key Idea

0	babaaaabcbabaaaaa0	17	0babaaaabcbabaaaaa
1	abaaaabcbabaaaaa0b	16	a0babaaaabcbabaaaa
2	baaaabcbabaaaaa0ba	15	aa0babaaaabcbabaaa
3	aaaabcbabaaaaa0bab	14	aaa0babaaaabcbabaa
4	aaabcbabaaaaa0baba	3	aaaabcbabaaaaa0bab
5	aabcbabaaaaa0babaa	12	aaaaa0babaaaabcbab
6	abcbabaaaaa0babaaa	⇨ 13	aaaa0babaaaabcbaba
7	bcbabaaaaa0babaaaa	⇨ 4	aaabcbabaaaaa0baba
8	cbabaaaaa0babaaaab	5	aabcbabaaaaa0babaa
9	babaaaaa0babaaaabc	1	abaaaabcbabaaaaa0b
10	abaaaaa0babaaaabcb	10	abaaaaa0babaaaabcb
11	baaaaa0babaaaabcba	6	abcbabaaaaa0babaaa
12	aaaaa0babaaaabcbab	2	baaaabcbabaaaaa0ba
13	aaaa0babaaaabcbaba	11	baaaaa0babaaaabcba
14	aaa0babaaaabcbabaa	0	babaaaabcbabaaaaa0
15	aa0babaaaabcbabaaa	9	babaaaaa0babaaaabc
16	a0babaaaabcbabaaaa	7	bcbabaaaaa0babaaaa
17	0babaaaabcbabaaaaa	8	cbabaaaaa0babaaaab

Input: "babaaaabcbabaaaaa"

Suffix Sorting in N log N Time

Manber's MSD algorithm.

- Phase 0: sort on first character using key-indexed sorting.
- Phase n: given list of suffixes sorted on first n characters, create list of suffixes sorted on first 2n characters
- Finishes after lg N phases.

Manber's LSD algorithm.

- Same idea but go from right to left.
- O(N log N) guaranteed running time.
- O(N) extra space.

String Sorting Performance

	String Sort	Suffix Sort (seconds)	
	Worst Case	Moby Dick	AesopAesop
Brute	$W N^2$	36,000 §	3,990 §
Quicksort	W N log N †	9.5	167
LSD *	W(N + R)	-	-
MSD	W(N + R)	395	memory
MSD with cutoff	W(N + R)	6.8	162
3-Way Radix Quicksort	W N log N †	2.8	400
Manber ‡	N log N	17	8.5

R = radix.
W = max length of string.
N = number of strings.

§ estimate
* fixed length strings only
† probabilistic guarantee
‡ suffix sorting only