Radix Sorting

Some of these lecture slides have been adapted from:

- *Algorithms in C, 3rd Edition*, Robert Sedgewick.

Radix Sorting

Radix sorting.

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

This lecture.

- LSD radix sort.
- MSD radix sort.
- Three-way radix quicksort.
- Suffix sorting.

An Application: Redundancy Detector

Longest repeated substring.

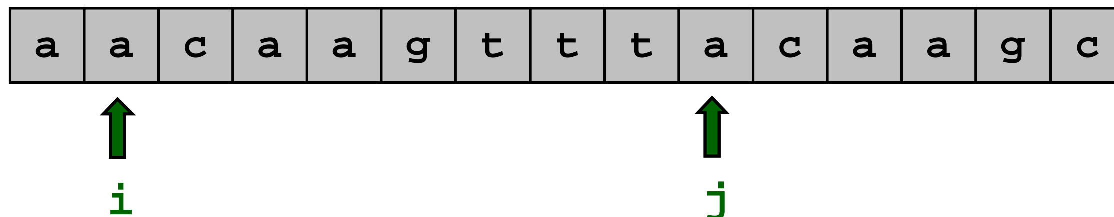
- Given a string of N characters, find the longest repeated substring.
- Example: `a a c a a g t t t a c a a g c`

Applications.

- Computational molecular biology.
- Data compression.
- Plagiarism detection.

Brute force.

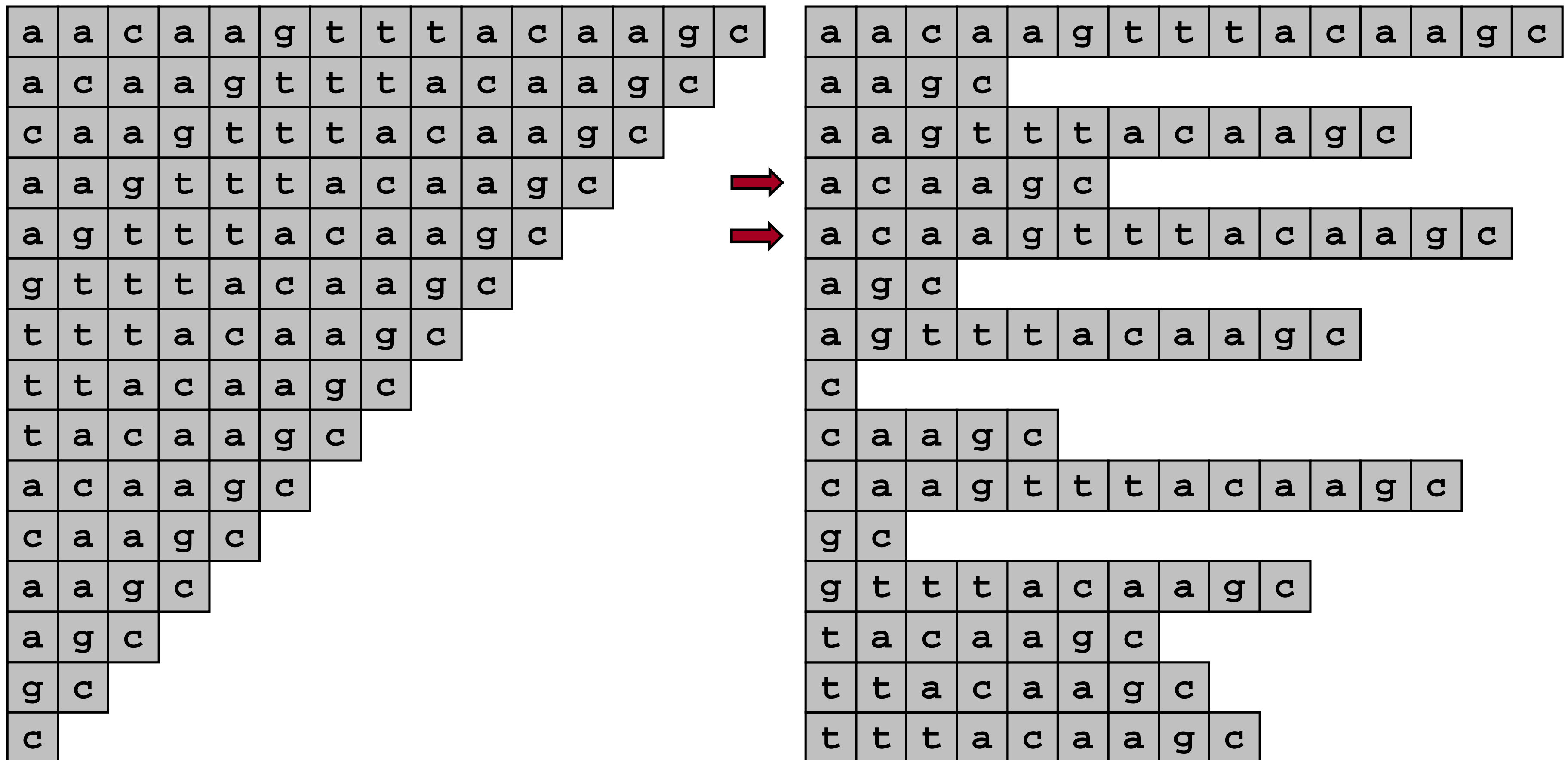
- Try all indices i and j for start of possible match and check.
- $O(W N^2)$ time, where W is length of longest match.



A Sorting Solution

Suffix sort.

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



Suffix Sorting

main()

```
char text[MAXN + 1];
Item suffixes[MAXN];

int main(void) {
   int c, N = 0;

   // read in text string
   while ((c = getchar()) != EOF)
      text[N++] = c;
   text[N] = '\0';


   // compute pointer to ith suffix
   for (i = 0; i < N; i++)
      suffixes[i] = text + i;

   // suffix sort and find longest repeated substring
   sort(suffixes, 0, N - 1);
   find(suffixes, 0, N - 1);
   return 0;
}
```

Implicitly form suffixes!

String Sorting Performance

	String Sort	Suffix (sec)
	Worst Case	Moby Dick
Brute	$W N^2$	36,000 §
Quicksort	$W N \log N$ †	694
Quicksort with cutoff	$W N \log N$ †	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

C syntax.

- Array of strings: `char *a[];`
- The ith string: `a[i]`
- The dth character of the ith string: `a[i][d]`
- Strings to be sorted: `a[lo], ..., a[hi]`

Key Indexed Counting

Key indexed counting.

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

```
// rearrange
for (i = lo; i <= hi; i++) {
   c = a[i][d];
   temp[count[c]++] = a[i];
}
```

```
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	1
b	2	b	5
c	3	c	6
d	1	d	8
e	2	e	8
f	1	f	12
g	3	g	11

temp

0	a	d	d
1			
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			
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.

```
// rearrange
for (i = lo; i <= hi; i++) {
   c = a[i][d];
   temp[count[c]++] = a[i];
}
```

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	1
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			
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

`d = 0;`

Key Indexed Counting

Key indexed counting.

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

```
// rearrange
for (i = lo; i <= hi; i++) {
   c = a[i][d];
   temp[count[c]++] = a[i];
}
```

```
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.

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

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

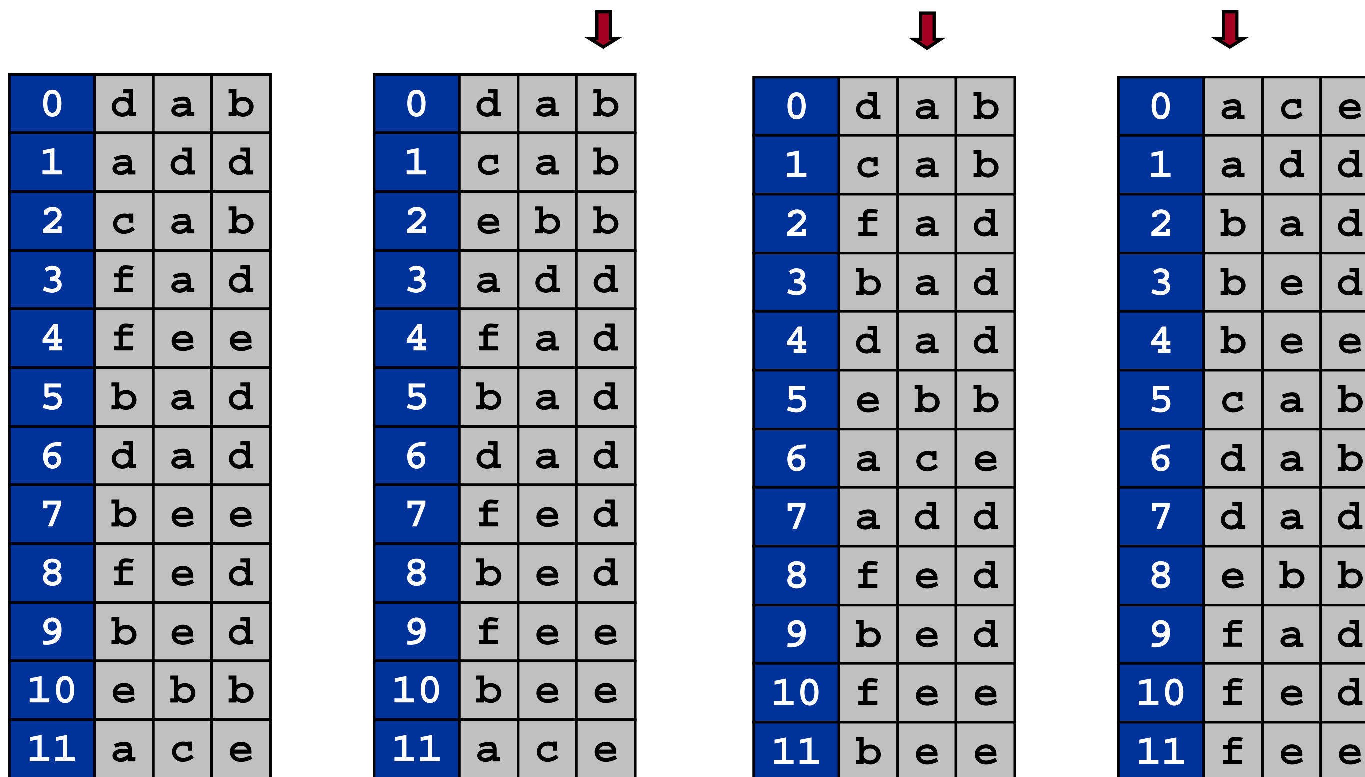
temp

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

LSD Radix Sort, Sedgewick Program 10.4

```
void lsd(char *a[], int lo, int hi) {
   int d;
   for (d = W-1; d >= 0; d--)
      keyindex(a, lo, hi, d);
}
```

Fixed length strings (length = W)

LSD Radix Sort Correctness

Two proofs of correctness.

- 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
- 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.

- $O(W(N + R))$.
- Why doesn't it violate N log N lower bound?

Advantage.

- Fastest sorting method for random fixed length strings.

Disadvantages.

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

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

MSD Radix Sort, Sedgewick Program 10.2

```
void msdR(char *a[], int lo, int hi, int d) {
   int i;
   int count[256+1] = {0};

   if (hi <= lo) return;

   keyindex(a, lo, hi, d, count);

   // recursively sort 255 pieces
   for (i = 0; i < 255; i++)
      msdR(a, lo + count[i], lo + count[i+1] - 1, d + 1);
}

void msd(char *a[], int lo, int hi) {
   msdR(a, lo, hi, 0);
}
```

assumes '\0'-terminated strings

String Sorting Performance

	String Sort	Suffix (sec)
	Worst Case	Moby Dick
Brute	$W N^2$	36,000 §
Quicksort	$W N \log N$ †	694
Quicksort with cutoff	$W N \log N$ †	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.

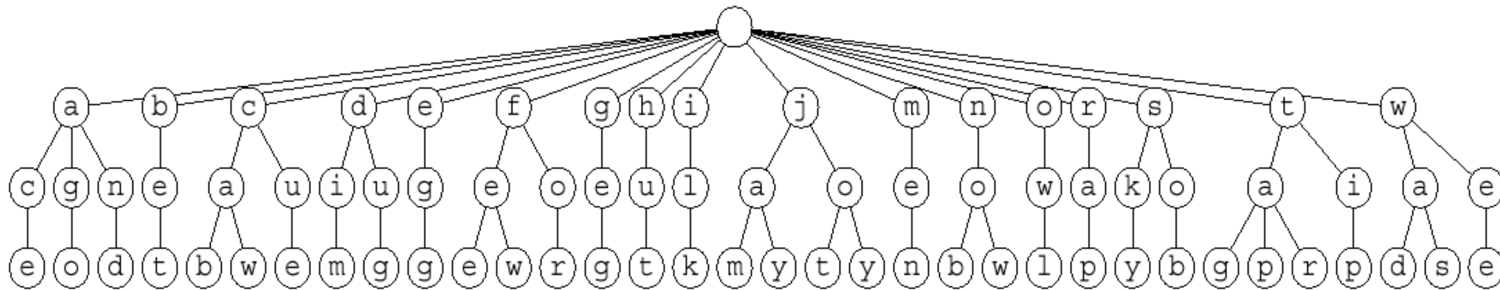
- Too slow for small files.
  - ASCII: 100x slower than insertion sort for N = 2
  - UNICODE: 30,000x slower for N = 2
- Huge number of recursive calls on small files.

Solution: cutoff to insertion sort for small N.

- Competitive with quicksort for string keys.

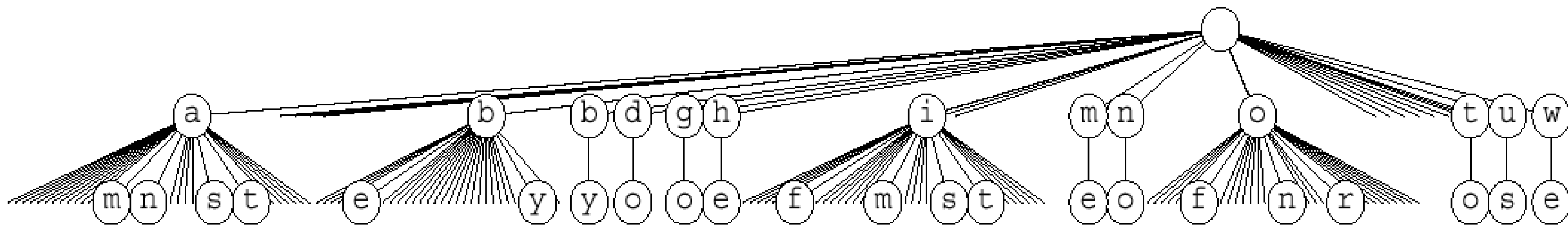
Recursive Structure of MSD Radix Sort

Tree structure to describe recursive calls in MSD radix sort.



Problem: algorithm touches lots of empty nodes.

- Tree can be as much as 256 times bigger than it appears!



3-Way Radix Quicksort

Idea 1. Use dth character to "sort" into 3 pieces instead of 256!

Idea 2. Keep all duplicates together in partitioning step.

Sort each piece recursively.

actinian	coenobite	actinian
jeffrey	conelrad	bracteal
coenobite	actinian	coenobite
conelrad	bracteal	conelrad
secureness	secureness	cumin
cumin	dilatedly	chariness
chariness	inkblot	centesimal
bracteal	jeffrey	cankerous
displease	displease	circumflex
millwright	millwright	millwright
repertoire	repertoire	repertoire
dourness	dourness	dourness
centesimal	southeast	southeast
fondler	fondler	fondler
interval	interval	interval
reversionary	reversionary	reversionary
dilatedly	cumin	secureness
inkblot	chariness	dilatedly
southeast	centesimal	inkblot
cankerous	cankerous	jeffrey
circumflex	circumflex	displease

Partition

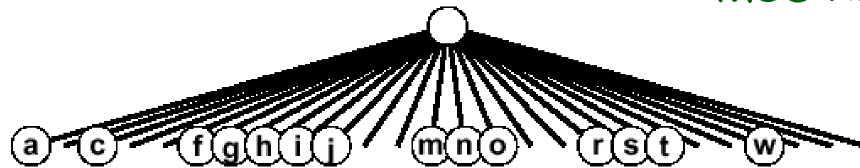
now	gig	ace	ago	a go	
for	for	bet	bet	a ce	
tip	dug	dug	and	a nd	
ilk	ilk	cab	ace	b et	
dim	dim	dim	c ab		
tag	ago	ago	c aw		
jot	and	and	c ue		
sob	fee	egg	egg		
nob	cue	cue	dug		
sky	caw	caw	dim		
hut	hut	f ee			
ace	ace	f or			
bet	bet	f ew			
men	cab	ilk			
egg	egg	gig			
few	few	hut			
jay	j ay	ja m			
owl	j ot	ja y			
joy	j oy	jo y			
rap	j am	jo t			
gig	owl	owl	m en		
wee	wee	now	owl		
was	was	nob	nob		
cab	men	men	now		
wad	wad	r ap			
caw	sky	sky	sky	sky	
cue	nob	was	tip	sob	
fee	sob	sob	sob	t ip	ta r
tap	tap	tap	tap	t ap	ta p
ago	tag	tag	tag	t ag	ta g
tar	tar	tar	tar	t ar	ti p
dug	tip	tip	w as		
and	now	wee	w ee		
jam	rap	wad	w ad		

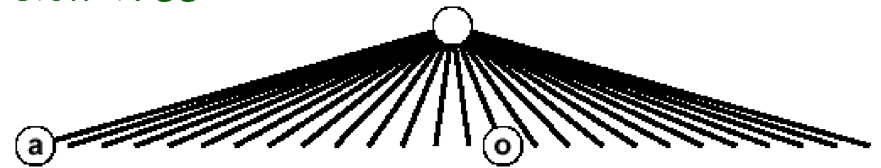
Algorithm

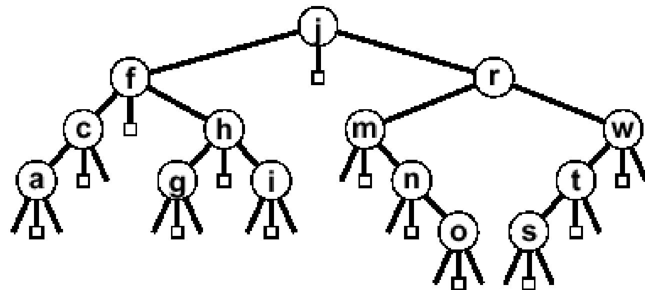
Recursive Structure of MSD Radix Sort vs. 3-Way Quicksort

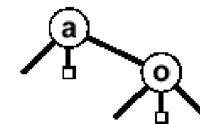
3-way radix quicksort collapses NULL links in MSD tree.

MSD Recursion Tree







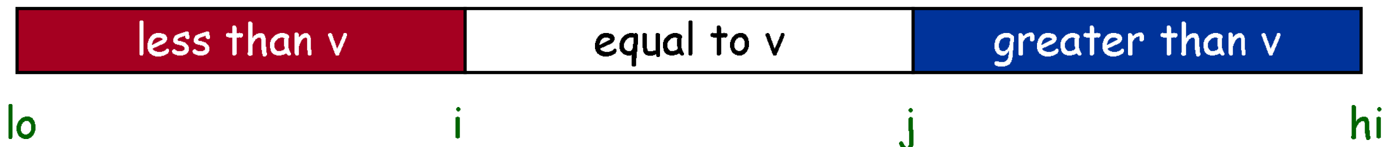


3-Way Radix Quicksort Recursion Tree

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



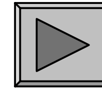
Dutch national flag problem.

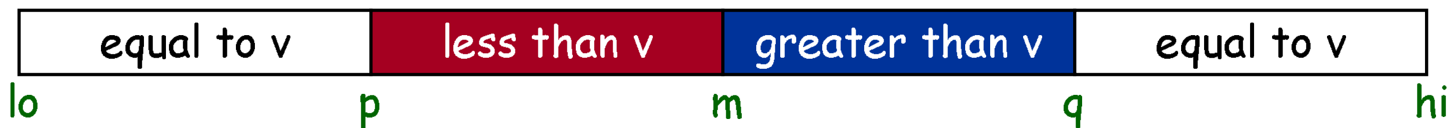
- Not easy to implement efficiently. (Try it!)
- Not done in practical sorts before mid-1990s.

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, Sedgewick Program 10.3

```
void quicksort3(Item a[], int lo, int hi, int d) {
    int i = lo-1, j = hi, k, p = lo-1, q = hi, v = a[hi][d];
    if (hi <= lo) return;
    while (i < j) {
        while (a[++i][d] < v)
            ;
        while (v < a[--j][d])
           if (j == lo) break;
        if (i > j) break;
        exch(a[i], a[j]);
        if (a[i][d] == v) { p++; exch(a[p], a[i]); }
        if (v == a[j][d]) { q--; exch(a[j], a[q]); }
    }
    if (p == q) {
        if (v != '\0') quicksort3(a, lo, hi, d+1);
        return;
    }
    if (a[i][d] < v) i++;
    for (k = lo; k <= p; k++, j--) exch(a[k], a[j]);
    for (k = hi; k >= q; k--, i++) exch(a[k], a[i]);
    quicksort3(a, lo, j, d);
    if ((i == hi) && (a[i][d] == v)) i++;
    if (v != '\0') quicksort3(a, j+1, i-1, d+1);
    quicksort3(a, i, hi, d);
}
```

swap equal keys to left or right

swap equal keys back to middle

Significance of 3-Way Partitioning

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

- Sort population by age.
- Sort job applicants by college attended.
- Remove duplicates from mailing list.
- Line recognition problem.

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.
  - independent of word length W for random strings
- Sublinear sort for large W since input is of size NW.

String Sorting Performance

	String Sort	Suffix (sec)
	Worst Case	Moby Dick
Brute	$W N^2$	36,000 §
Quicksort	W N log N †	694
Quicksort with cutoff	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 Qsort	W N log N †	2.8

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

§ estimate
* assumes fixed length strings.
† 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?

- N log N ?
- Linear time ?

Observation. Must find longest repeated substring WHILE suffix sorting to beat quadratic worst case.

```
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 †	694	320
Quicksort with cutoff	W N log N †	9.5	167
LSD *	W(N + R)	-	-
MSD	W(N + R)	395	crash (!)
MSD with cutoff	W(N + R)	6.8	162
3-Way Radix Qsort	W N log N †	2.8	400
Manber LSD	N log N ‡	17	8.5

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

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