Strings: String Sorts

Neerja Mhaskar

Dept. of Computing and Software, McMaster University, Canada

Acknowledgments: Material mainly based on the textbook Algorithms by Robert Sedgewick and Kevin Wayne (Chapters 5.1)

Summary of the performance of sorting algorithms

Frequency of operations. [* in the below figure indicates probabilistic]

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()

Lower bound. $\sim N \log_2 N$ 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.

What is a string?

A **string** is just a sequence of "**letters**" (symbols) drawn from some (finite or infinite) "**alphabet**" (set):

- a word in the English language, whose letters are the upper and lower case English letters;
- a text file, whose letters are the ASCII characters;
- the binary code over $\Sigma = \{0, 1\}$;
- a number in base 10, whose digits are drawn from $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.
- a DNA sequence, perhaps three *billion* letters long, over the alphabet $\Sigma = \{A, C, G, T\}$;

Alphabets

Radix. Number of characters R in alphabet.

Map characters (in the alphabet) to distinct numbers in $[0..R-1]$ to form a numeric string. This mapping helps in achieving cleaner code as we can use keys as array indices.

$\log R$ - number of bits required to represent a character in the alphabet.

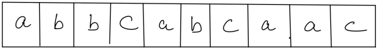
name	R()	lgR()	characters
BINARY	2	1	01
OCTAL	8	3	01234567
DECIMAL	10	4	0123456789
HEXADECIMAL	16	4	0123456789ABCDEF
DNA	4	2	ACTG
LOWERCASE	26	5	abcdefghijklmnopqrstuvwxyz
UPPERCASE	26	5	ABCDEFGHIJKLMNOPQRSTUVWXYZ
PROTEIN	20	5	ACDEFGHIKLMNPQRSTVWY
BASE64	64	6	ABCDEFGHIJKLMNOPQRSTUVWXYZabcdef ghijklmnopqrstuvwxyz0123456789+/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

Key-indexed counting: Basic idea

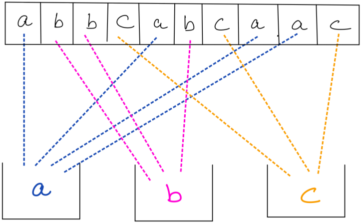
Assumption. Keys are integers between 0 and $R-1$.

Implication. Can use key as an array index.

Below is an example array we want to sort.



We can create buckets with the characters as labels and each time we encounter a character with the same label we put it in the bucket.



Key-indexed counting: Basic idea

- These buckets can be represented by using an array A, indexed by the keys 0 to $R - 1$.
- Store the number of occurrences of a given key i at $A[i]$.
- We can then sort the input array by overwriting it, such that the first $A[0]$ positions in the input array are filled with 0, the next $A[1]$ positions are filled with 1, and so on.

Key-indexed counting - applications

Applications.

- Sort strings by first letter.
- Sort class roster by section.
- Subroutine in a sorting algorithm. [stay tuned]

input name	section	sorted result (by 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 Counting Sort Algorithm

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.

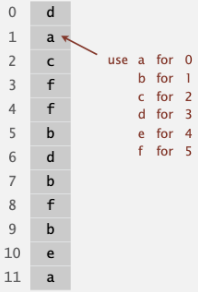
```
int N = a.length;
int[] count = new int[R+1];

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];
```



Key Counting Sort Algorithm

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.

count frequencies

```
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

offset by 1
[stay tuned]

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

Key Counting Sort Algorithm

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.

compute cumulates

```
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

r	count[r]
a	0
b	2
c	5
d	6
e	8
f	9
-	12

6 keys < d, 8 keys < e
so d's go in a[6] and a[7]

Key Counting Sort Algorithm

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]	r	count[r]	i	aux[i]
0	d			0	a
1	a			1	a
2	c			2	b
3	f	a	2	3	b
4	f	b	5	4	b
5	b	c	6	5	c
6	d	d	8	6	d
7	b	e	9	7	d
8	f	f	12	8	e
9	b	-	12	9	f
10	e			10	f
11	a			11	f

18

Key Counting Sort Algorithm

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

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

19

Key-indexed counting: analysis

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

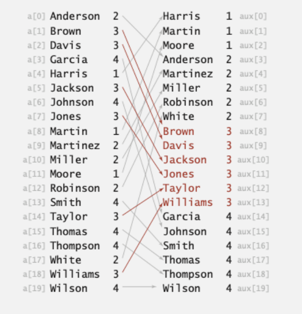
Proposition. Key-indexed counting uses extra space proportional to N + R.

Stable: Yes!

It is because while moving items as indicated in the below loop, equal keys/elements from $a[]$ are copied to $aux[]$ in the same order as they appear in $a[]$; that is, relative ordering of equal keys is maintained.

move items

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



Comparing 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

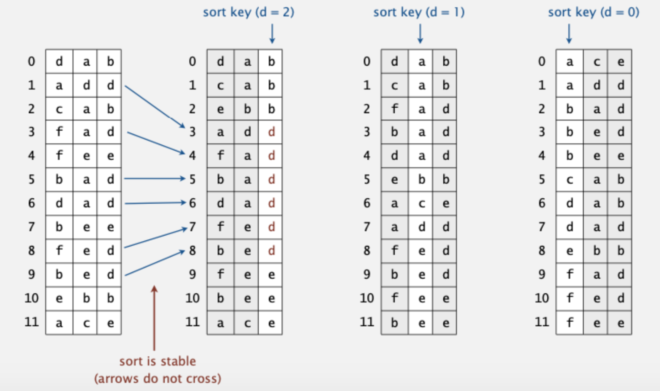
Running time. Proportional to length of longest common prefix.

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

Least-significant-digit-first string sort

LSD string (radix) sort.

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



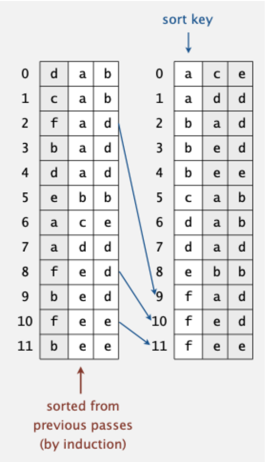
LSD string sort: correctness proof

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.



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

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

LSD string sort: Java implementation

```
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

LSD string sort: correctness proof

Proposition. LSD string sort takes time proportional to $W(N + R)$, where W is the size/length of the fixed length strings.

Reverse LSD

- Consider characters from left to right.
- Stably sort using dth character as the key (using key-indexed counting).

Reverse LSD does not work!

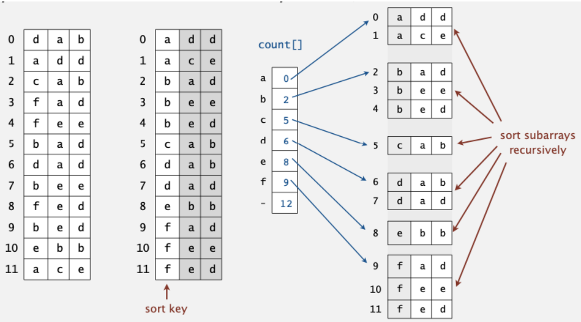
		sort key (d = 0)	sort key (d = 1)	sort key (d = 2)
0	d a b	a d d	b a d	c a b
1	a d d	a c e	c a b	d a b
2	c a b	b a d	d a b	e b b
3	f a d	b e e	d a d	b a d
4	f e e	b e d	f a d	d a d
5	b a d	c a b	e b b	f a d
6	d a d	d a b	a c e	a d d
7	b e e	d a d	a d d	b e d
8	f e d	e b b	b e e	f e d
9	b e d	f a d	b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

not sorted!

Most-significant-digit-first string sort

MSD string (radix) sort.

- 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).

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 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 string sort: potential for disastrous performance

Problem: Much too slow for small subarrays.

- Each function call needs its own count[] array.
- Huge number of small subarrays because of recursion.

For example:

- Suppose that we are sorting millions of ASCII strings (R = 256) that are all different, with no cutoff for small subarrays. Each string eventually finds its way to its own subarray, so you will sort millions of subarrays of size 1.
- But each such sort involves initializing the 258 entries of the count[] array to 0 and transforming them all to indices. This cost is likely to dominate the rest of the sort.
- With Unicode (R = 65536) the sort might be thousands of times slower.

Solution. Switch to insertion sort for small subarrays (cut-off at 10 or 15).

MSD string sort: performance

Number of characters examined.

- MSD examines just enough characters to sort the keys.
- Number of characters examined depends on keys.
- Can be sublinear in input size!
- A worst-case input is one with all strings equal.

Random (sublinear)	Non-random with duplicates (nearly linear)	Worst case (linear)
1EIO402	are	1DNB377
1HYL490	by	1DNB377
1ROZ572	sea	1DNB377
2HXE734	seashells	1DNB377
2IYE230	seashells	1DNB377
2XOR846	sells	1DNB377
3CDB573	sells	1DNB377
3CVP720	she	1DNB377
3IGJ319	she	1DNB377
3KNA382	shells	1DNB377
3TAV879	shore	1DNB377
4CQP781	surely	1DNB377
4QGI284	the	1DNB377
4YHV229	the	1DNB377

Characters examined by MSD string sort

MSD string sort: performance

Proposition. MSD string sort takes anywhere between $O(N + R)$ and $O(W(N + R))$, where W is the average size/length of the fixed length strings.