W4231: Analysis of Algorithms

9/23/1999 (revised 9/29)

- Sorting in linear time (sometimes).

A trivial example

An array of integers $a_1 \cdots a_n$ is given such that $1 \leq a_i \leq n$ and all the elements are distinct.

Solution: output $1, \ldots, n$.

Repetitions are allowed

An array of integers $a_1 \cdots a_n$ is given such that $1 \leq a_i \leq n$ and elements may be repeated.

Create a vector $c_1, \ldots, c_n$, where

$$c_i = |\{j : a_j = i\}|$$

If $A = [2, 4, 1, 2, 5, 8, 3, 1]$ then

$C = [2, 2, 1, 1, 1, 0, 0, 1]$.

Scan C, for every i, write i for c_i times.

Implementation

```
sort(int a[], int n){
  int c[n],i,j,k;
  // initialize c[]
  for (j=0; j<n; j++)
    c[j]=0;
  // fill in the entries of c[]
  for (i=0; i<n; i++)
    c[a[i]]++;
  // sort a[]
  i=0;
  for (j=0; j<n; j++)
    for (k=0; k<c[j]; k++){
       a[i]=j; i++;}
}
```


Stability

A sorting algorithm is stable if

on input $a_1 \cdots a_n$ it outputs the sorted sequence $a_{\pi(1)} \cdots a_{\pi(n)}$

with the property that if $i < j$ and $a_{\pi(i)} \leq a_{\pi(j)}$

then $\pi(i) < \pi(j)$.

An example of non-stability

The difference between stable and non-stable algorithms is important only if each item has a key used for sorting and some other information; and the keys can be repeated.

E.g. sort the pairs

$$(1997, \text{LA Confidential}), (1998, \text{Life is Beautiful}),$$

$$(1993, \text{Schindler's List}), (1997, \text{Titanic}), (1993, \text{The Piano})$$

using the first number as a key.

If the algorithm reports

$$(1993, \text{Schindler's List}), (1993, \text{The Piano}),$$

$(1997, \text{Titanic}), (1997, \text{LA Confidential}), (1998, \text{Life is Beautiful})$

Then it is not stable

A Stable Version of Counting Sort

Each c_j is a queue.

For every i, we copy a_i in the queue c_j, where j is the key of a_i.

At the end we patch the queues together. Impossible to have an inversion.

Alternative method in CLR.

Analysis

Let c_j be the number of items of key j. Then $\sum_{j=1}^{m} c_j = n$.

Running time; $O(m)$ to initialize c; $O(n)$ to fill c; $\sum_{j=1}^{m} O(c_j) + O(1) = O(\sum_j c_j) + O(m) = O(m+n)$ total time is $O(n+m)$.

Better than `mergesort` when $m = o(n \log n)$.

Radix Sort

Suppose we have in input n integers that are b-digits binary numbers.

Put the numbers whose last digit is 0 before those whole last digit is 1.

Proceed like that for every digit using a stable sorting.

Dealing with each digit takes $O(n)$ time.

Total time: $O(nb)$.

More on Radix Sort

Generalization: each number has b digits in base k.

Do b passes of a stable sort.

For integers in the range $1, \ldots, m$, we can view these integers as having $\log_n m$ digits in base n.

Do $\log_n m$ passes of stable counting sort. Each one takes time $O(n)$.

Sort in time $O(n \log m / \log n)$.

Summary of Sorting Algs for Integers

Input: n integers in the range $1, \ldots, m$.

- Mergesort $O(n \log n)$-time independent of m (assuming unit-cost RAM model).
- Radix Sort $O(n \log m / \log n)$.
- Counting Sort $O(n + m)$.

Counting sort is preferable only if $m = O(n)$. Radix sort works well for bigger m, provided $m = O(n^{\log n})$. For bigger values of m, Mergesort is better.

Lexicographic order

Consider strings over a certain alphabet set S on which an order $<$ is defined. E.g. S is the set of Roman characters $a, b, \ldots, z$ and the order $<$ is the alphabetic order.

For two strings $a = a_1 \cdots a_n$ and $b = b_1 \cdots b_m$, we write $a <_{lex} b$ if there is a j such that

- $a_i = b_i$ for $i = 1, \ldots j-1$ and
- $a_j < b_j$.

or if $a_i = b_i$ for $i = 1, \ldots, n$ and $m > n$.

E.g $platform < plausible$ ($j = 4$ in prev. definition — $t < u$).

p l a **t** f o r m
p l a **u** s i b l e

and also $platform < platforms$.

Sorting strings

disk dish blow true

1	2	3	4
d	i	s	k
d	i	s	h
b	l	o	w
t	r	u	e

We first sort the 4th component

1	2	3	4
t	r	u	e
d	i	s	h
d	i	s	k
b	l	o	w

Then the 3rd

1	2	3	4
b	l	o	w
d	i	s	h
d	i	s	k
t	r	u	e

Then the 2nd

1	2	3	4
d	i	s	h
d	i	s	k
b	l	o	w
t	r	u	e

Then the 1st

1	2	3	4
b	l	o	w
d	i	s	h
d	i	s	k
t	r	u	e

Running Time

If we have n strings of length l this takes *linear* and *optimal* time $O(nl)$, provided we can do each pass in $O(n)$ time.

This is possible if we sort the array of *pointers* to the strings.

Strings of different lengths

If the strings have different length $l_1, \ldots, l_n$, and $l_{\max}$ is the max length, the algorithm can be adapted to work in $O(nl_{\max})$ time. This is not linear (neither optimal) if there are only a few long strings.

A better algorithm takes time $O(l_{tot})$ where $l_{tot} = \sum_i l_i$.

Better Algorithm

Main idea: for l going from $l_{\max}$ to 1, sort all the strings *whose length is at least* l using l-th character as a key.

LexSort $(s_1, \ldots, s_n)$
  create queues $C_1, \ldots, C_{l_{\max}}$, where C_l contains strings of length l
  for $l = l_{\max}$ **down to** 2
    sort C_l using the l-th character as a key
    append C_l at the end of C_{l-1}
  sort C_1 using the 1-st character as a key
  return C_1

Analysis

For every $1 \leq l \leq l_{max}$, call c_l the number of strings of length $\geq l$.

Then $\sum_{l=1}^{l_{\max}} c_l = l_{tot}$.

Can you see why?

Then if we sort in time $O(c_l)$ the l-th entry of the strings who have an l-th entry, the algorithm takes time $O(l_{tot})$.

Example

mit, columbia, rutgers, harvard, princeton, yale

m i t
c o l u m b i a
r u t g e r s
h a r v a r d
p r i n c e t o n
y a l e

Entry 9

m i t
c o l u m b i a
r u t g e r s
h a r v a r d
p r i n c e t o **n**
y a l e

Entry 8

m i t
c o l u m b i **a**
r u t g e r s
h a r v a r d
p r i n c e t **o** n
y a l e

Entry 7

m i t
h a r v a r **d**
c o l u m b **i** a
p r i n c e **t** o n
r u t g e r **s**
y a l e

Entry 6

m i t
c o l u m **b** i a
p r i n c **e** t o n
h a r v a **r** d
r u t g e **r** s
y a l e

Entry 5

m i t
h a r v **a** r d
p r i n **c** e t o n
r u t g **e** r s
c o l u **m** b i a
y a l e

Entry 4

m i t
y a l **e**
r u t **g** e r s
p r i **n** c e t o n
c o l **u** m b i a
h a r **v** a r d

Entry 3

p r **i** n c e t o n
y a **l** e
c o **l** u m b i a
m i **t**
r u **t** g e r s
h a **r** v a r d

Entry 2

y **a** l e
h **a** r v a r d
m **i** t
c **o** l u m b i a
p **r** i n c e t o n
r **u** t g e r s

Entry 1

c o l u m b i a
h a r v a r d
m i t
p r i n c e t o n
r u t g e r s
y a l e