Counting sort treats the element as just one digit

Radix Sort

From least to most significant digit
$d = 0$ to $b-1$
apply counting sort where we access digit b

ex

329		720		720		329
457		355		329		355
657		436	phase 1	436		436
839	→	457	→	839	→	457
436	phase 0 using counting sort on digit 0	657	place 1	355	look at place 2	657
720		329		457		720
355		839		657		839

Radix

~~Counting~~ Sort (input, output, k) {

for d= 0 to #digits-1

```
    int n= input.length;
    for (i=0; i<k; i++)
        count[i]=0;
    for (j=0; j<n; j++)
        count[input[j]]++;
    for (i=1; i<k; i++)
        count[i] += count[i-1];
    for (j=n-1; j≥0; j--)
        output[--count[input[j]]] = input[j];
}
```

radix sort

digitizer.getDigit(input[j], d)   current digit

array of same length

at end count[i] is # of elements that are i

count[i] is # of elements $\le i$

Correctness Highlights.

Prove following holds using induction

After first p phases, numbers when looking at least significant p digits are sorted

base: true after 1 phase by correctness of counting sort

inductive step: combines inductive hyp, correctness of counting sort + stability of counting sort

What's the asymptotic time complexity?

n # elements

d max # of digits of elements (pad accordingly within digitizer)

b base of each digit $(0, 1, \ldots, b-1)$

$\Theta(d(n+b))$

Return to the ex.

n = # social security #s to sort

$d = 9$
$b = 10$
) treat each base-10 digit as a digit for radix sort

$$\Theta(9(n+10)) = \Theta(n)$$

$$\Theta(9(n+10))$$

$$\underbrace{XXX}_{\text{base 1000 number}}\ \underbrace{XXX}_{\text{digit}}\ \underbrace{XXX}_{\text{digit}}$$

$0, \ldots, 999$

roughly 3 times faster for "large" n

Time complexity with this digitizer

$$\Theta(3(n+1000))$$

Consider sorting #s that begin in binary (base 2)

n #s

b bits ($b = 32$, 32-bit #)

group $\underline{r}$ bits into a digit

$\# \text{digits} = \frac{b}{r}$

base per digit $= 2^r$

$$\Theta\left(\frac{b}{r}\left(n+2^r\right)\right)$$ time complexity

$$\frac{c \cdot b}{r}\left(n+2^r\right)$$ could even work out constants more carefully

min with respect to r.

roughly optimizes when $r = \Theta(\log_2 n)$

ADT Taxonomy Part II

Note Title 9/28/2007

Plan for today

- Finish discussion of radix sort
- Briefly discuss bucket sort
- Return to ADT Taxonomy
- Introduce Set ADT
  (if time permits)

Counting Sort

```
protected void radixsortImpl(Digitizer<? super E> digitizer) {
    Object[] from = new Object[getSize()];                    //elements start in from
    Object[] to = new Object[getSize()];                      //placed in sorted order into to
    int b = digitizer.getBase();                              //base for digitizer
    int count[] = new int[b];                                 //counter for each possible value
    int numDigits = 0;                                        //maximum number of digits in any element
    for (int i = 0; i < getSize(); i++) {
        from[i] = a[getPosition(i)];                          //move position i into index i of from
        numDigits = max(numDigits, digitizer.numDigits((E) from[i]));
    }
    for (int d = 0; d < numDigits; d++) {                     //digit to use in current pass
        Arrays.fill(count, 0);                                //reset all counts to 0
        for (Object x : from)                                 //count # elements with
            count[digitizer.getDigit((E) x, d)]++;            //each value for digit d
        for (int i = 1; i < b; i++)                           //update to cumulative count
            count[i] += count[i-1];
        for (int i = getSize()-1; i ≥ 0; i--)                 //put elements in array to
            to[--count[digitizer.getDigit((E) from[i], d)]] = from[i];
        Object[] temp = from; from = to; to = temp;           //swap the "from" and "to" arrays
    }
    for (int i = 0; i < getSize(); i++)                       //put sorted elements back into the collection
        put(getPosition(i), from[i]);
    version.increment();                                      //invalidate locators for iteration
}
```

Optimizing radix sort

bits

base for a r-bit digit

time complexity $\quad c \cdot \frac{b}{r}\underbrace{(n + 2^r)}$

bits/digit

intuitively want to set r so $n = 2^r$ so you reduce # digits while not making base too high

Solving for r in $n=2^r$ yields $r = \log_2 n$

Example: let $c = 5$, $n = 1{,}000{,}000$

32 bit numbers

	# bits/digit	# digits	base $\leftarrow k$	$c \cdot d(n+k)$
basic radix sort →	1	32	2	160,000,300
	2	16	4	80,000,320
	4	8	16	40,000,640
	8	4	256	20,005,120
	16	2	65536	10,655,360
counting sort →	32	1	$>4 \times 10^9$	$>4{,}000{,}000{,}000$

What are the limitations of radix sort?

Requires that you can digitize all elements.

Instead of using a comparator to compare entire elements, a digitizer is used to extract each digit.

Think about these costs.