CSE 241 Class 11

Jeremy Buhler

October 7, 2015

Today: sorting in linear time!

1 Counting Sort

Counting sort is a correct $\Theta(n)$ sort.

- How is this possible?
- Must not fit the comparison sort model
- Indeed, never actually compares two elements of array

Input: array A of n integers between 0 and $k-1$
Uses: an auxiliary array "counts" of size k

```
CountingSort(A, n, k)
  for j in 0 ... k-1 do
     counts[j] ← 0

  for i in 0 ... n-1 do
     counts[A[i]]++

  i ← 0
  for j in 0 ... k-1 do
     for m in 1 ... counts[j] do
        A[i] ← j
        i++
```

- *Correctness*: clearly, resulting array is sorted, as procedure writes lower values before higher ones.
- Moreover, each distinct value j in A occurs exactly counts[j] times in both input and output

What is running time? Split into three parts.

1. Initialization of "counts" = $\Theta(k)$
2. Counting up array elements = $\Theta(n)$

3. Refilling array? Inner loop body is executed how many times?

$$\sum_{j=0}^{k-1} \text{counts}[j] = n$$

   Hence, complexity of refilling operation is $\Theta(n)$.

Total time is thus $\Theta(k+n)$. If $k = O(n)$, time is $\Theta(n)$.

2 Counting Sort for Arbitrary Values

CountingSort is interesting, but is it more than just a curiosity?

- not hard to extend to sort records with integer keys in a fixed range
- **ask class**: intuitively, how might we do it?
- (see homework problem for formal solution)
- **Example**: sort phone numbers by area code
- $k = 1000$ (3-digit area code)

One important property of CountingSort on arbitrary records is that it can be made **stable**.

- **Defn**: A sorting algorithm is *stable* if it preserves the order of elements with equal-valued keys.
- That is, if $A[i] = A[j]$ and $i < j$ before sorting, then $A[i]$ will occur before $A[j]$ in the final sorted array.

3 Radix Sort

CountingSort works great if range of values is small, but what if it is very large?

- **Example**: Social Security Numbers: $k = 1$ billion
- **Example**: 32-bit integers: $k \approx 4$ billion

- Do we really want to allocate a "counts" array this big?

Fortunately, we can sort large numbers incrementally – one digit at a time!

- Break up numbers into d "digits" (could be base 10, base 2, in general, base k)
- Could use CountingSort to sort records by any one digit.
- Will sort by each digit in turn from least to most sigificant
- **Stability property** of CountingSort guarantees that records with same ith digit remain in correct order after ith sort.

RadixSort(A, d, n)
  for i **in** $1 \ \ldots \ d$ **do**
    sort records by i**th least significant digit** with CountingSort

Example:

4 Correctness of Radix Sort

Will prove by induction on number of digits d.
Base: when $d = 1$, result is same as applying CountingSort to A, hence correct.
Ind: suppose RadixSort correctly sorts $d - 1$ digit numbers.

- Consider array elements $A[i]$, $A[j]$ after sorting on dth digit.
- If $A[i]$, $A[j]$ have different dth digits (the most significant digit), they are correctly ordered by correctness of CountingSort.

- If $A[i]$, $A[j]$ have *same* dth digit, we have by inductive hyp. that they were correctly ordered before dth call to CountingSort.
- Moreover, by **stability** of CountingSort, the order of $A[i]$ and $A[j]$ does not change in dth sorting pass, so they remain correctly sorted. QED

5 Efficiency of Radix Sort

- Suppose input values contain d base-k digits.
- Each call to CountingSort takes time $\Theta(n + k)$.
- One call for each digit.
- Total cost is then $\Theta(d(n + k))$.
- If k is a constant (e.g. 10, 2, etc.), cost is $\Theta(dn)$.

6 A Hack: Digit Grouping

What if we must sort large numbers in a small base ($d >> k$)?

- **Example**: array of 64-bit integers
- $d = 64$, $k = 2$
- needs 64 sorting passes, each on one bit
- can we spend less time?

Suppose we **group** the bits of each integer into a smaller number of "superdigits"

- **Example**: three bits for base-8 digits
- **Example**: four bits for base-16 digits
- In general, b bits can be grouped into b/r digits in base 2^r.

Let each pass of CountingSort operate on one superdigit.

- $d = b/r$
- $k = 2^r$
- Hence, cost is

$$\Theta(d(n + k)) = \Theta\left(\frac{b}{r}\left[n + 2^r\right]\right)$$

7 Digit Grouping Tradeoff

As we increase superdigit size r, number of passes decreases, but each pass becomes more expensive (more time to walk through larger "counts" array).

- must try to balance total work performed by algorithm
- optimal balance depends on constants of CountingSort implementation, i.e.

$$T(n, b, r) = \frac{b}{r}(c_1 n + c_2 2^r)$$

- a good balance can often be had when $r = \log n$
- with this assumption, time is $\Theta\left(b \frac{n}{\log n}\right)$

8 Why Not Always use Radix Sort?

- Radix sort is good to know if you need linear time
- It is a good way to sort records by hand
- However, it cannot sort in place – CountingSort needs extra memory
- In practice, lack of in-placeness and generality mean that comparison sorts are more popular, even though they are asymptotically slower