Algorithms and Data Structures: Counting sort and Radix sort

8th February, 2016

Special Cases of the Sorting Problem

In this lecture we assume that the sort keys are sequences of bits.

- Quite a natural special case. Doesn't cover everything:
  - eg, exact real number arithmetic doesn't take this form.
  - In certain applications, eg Biology, pairwise experiments may only return $>$ or $<$ (non-numeric).
- *Sometimes* the bits are naturally grouped, eq, as characters in a string or hexadecimal digits in a number (4 bits), or in general bytes (8 bits).
- Today's sorting algorithms are allowed access these bits or groups of bits, instead of just letting them compare keys ...
  This was NOT possible in comparison-based setting.

Easy results . . . Surprising results

Simplest Case:

Keys are integers in the range $1, \ldots, m$, where $m = O(n)$ (n is (as usual) the number of elements to be sorted). We can sort in $\Theta(n)$ time

Easy results ... Surprising results

Simplest Case:

Keys are integers in the range $1, \ldots, m$, where $m = O(n)$ (n is (as usual) the number of elements to be sorted). We can sort in $\Theta(n)$ time (big deal

Easy results ... Surprising results

Simplest Case:

Keys are integers in the range $1, \ldots, m$, where $m = O(n)$ (n is (as usual) the number of elements to be sorted). We can sort in $\Theta(n)$ time (big deal ... but will help later).

Easy results . . . Surprising results

Simplest Case:

Keys are integers in the range $1, \ldots, m$, where $m = O(n)$ (n is (as usual) the number of elements to be sorted). We can sort in $\Theta(n)$ time (big deal . . . but will help later).

Surprising case: (I think)

For any constant k, the problem of sorting n integers in the range $\{1, \ldots, n^k\}$ can be done in $\Theta(n)$ time.

Counting Sort

Assumption: Keys (attached to items) are Ints in range $1, \ldots, m$.

Counting Sort

Assumption: Keys (attached to items) are Ints in range $1, \ldots, m$.

Idea

1. Count for every key j, $1 \leq j \leq m$ how often it occurs in the input array. Store results in an array C.

Counting Sort

Assumption: Keys (attached to items) are Ints in range $1, \dots, m$.

Idea

1. Count for every key j, $1 \leq j \leq m$ how often it occurs in the input array. Store results in an array C.
2. The counting information stored in C can be used to determine the position of each element in the sorted array. Suppose we modify the values of the $C[j]$ so that *now*

   $C[j] =$ the number of keys *less than or equal* to j.

   Then we know that the elements with key "j" must be stored at the indices $C[j-1]+1, \dots, C[j]$ of the final sorted array.

Counting Sort

Assumption: Keys (attached to items) are Ints in range $1, \dots, m$.

Idea

1. Count for every key j, $1 \leq j \leq m$ how often it occurs in the input array. Store results in an array C.
2. The counting information stored in C can be used to determine the position of each element in the sorted array. Suppose we modify the values of the $C[j]$ so that *now*

   $C[j] =$ the number of keys *less than or equal* to j.

   Then we know that the elements with key "j" must be stored at the indices $C[j-1]+1, \dots, C[j]$ of the final sorted array.
3. We use a "trick" to move the elements to the right position of an auxiliary array. Then we copy the sorted auxiliary array back to the original one.

Implementation of Counting Sort

Algorithm $\textsc{Counting Sort}(A, m)$

$n \leftarrow A$.length
Initialise array $C[1 \ldots m]$
for $i \leftarrow 1$ **to** n **do**
    $j \leftarrow A[i]$.key
    $C[j] \leftarrow C[j] + 1$
for $j \leftarrow 2$ **to** m **do**
    $C[j] \leftarrow C[j] + C[j-1]$ $\triangleright$ $C[j]$ stores $\sharp$ of keys $\leq j$
Initialise array $B[1 \ldots n]$
for $i \leftarrow n$ **downto** 1 **do**
    $j \leftarrow A[i]$.key $\triangleright$ $A[i]$ highest w. key j
    $B[C[j]] \leftarrow A[i]$ $\triangleright$ Insert $A[i]$ into highest free index for j
    $C[j] \leftarrow C[j] - 1$
for $i \leftarrow 1$ **to** n **do**
    $A[i] \leftarrow B[i]$

Analysis of Counting Sort

- The loops in lines 3–5, 9–12, and 13–14 all require time $\Theta(n)$.
- The loop in lines 6–7 requires time $\Theta(m)$.
- Thus the overall running time is

$$O(n + m).$$

- This is *linear* in the number of elements if $m = O(n)$.

Analysis of Counting Sort

- The loops in lines 3–5, 9–12, and 13–14 all require time $\Theta(n)$.
- The loop in lines 6–7 requires time $\Theta(m)$.
- Thus the overall running time is

$$O(n + m).$$

- This is *linear* in the number of elements if $m = O(n)$.

Note: This does not contradict Theorem 3 from Lecture 7 - that's a result about the general case, where keys have an arbitary size (and need not even be numeric).

Analysis of Counting Sort

- The loops in lines 3–5, 9–12, and 13–14 all require time $\Theta(n)$.
- The loop in lines 6–7 requires time $\Theta(m)$.
- Thus the overall running time is

$$O(n+m).$$

- This is *linear* in the number of elements if $m = O(n)$.

Note: This does not contradict Theorem 3 from Lecture 7 - that's a result about the general case, where keys have an arbitary size (and need not even be numeric).

Note: COUNTING-SORT is STABLE.

- (*After* sorting, 2 items with the same key have their *initial relative order*).

Radix Sort

Basic Assumption

Keys are sequences of digits in a fixed range $0, \ldots, R-1$, all of equal length d.

Examples of such keys

- 4 digit hexadecimal numbers (corresponding to 16 bit integers)
  $R = 16, d = 4$
- 5 digit decimal numbers (for example, US post codes)
  $R = 10, d = 5$
- Fixed length ASCII character sequences
  $R = 128$
- Fixed length byte sequences
  $R = 256$

Stable Sorting Algorithms

Definition 1

A sorting algorithm is stable if it always leaves elements with equal keys in their original order.

Stable Sorting Algorithms

Definition 1

A sorting algorithm is stable if it always leaves elements with equal keys in their original order.

Examples

- COUNTING-SORT, MERGE-SORT, and INSERTION SORT are all stable.
- QUICKSORT is not stable.
- If keys and elements are exactly the same thing (in our setting, an element is a structure containing the key as a sub-element) then we have a much easier (non-stable) version of COUNTING-SORT. (How? ... CLASS?).

Radix Sort (cont'd)

Idea

Sort the keys digit by digit, *starting with the least significant digit*.

Example

`now`		`sob`		`tag`		`ace`
`for`		`nob`		`ace`		`bet`
`tip`		`ace`		`bet`		`dim`
`ilk`		`tag`		`dim`		`for`
`dim`		`ilk`		`tip`		`hut`
`tag`		`dim`		`sky`		`ilk`
`jot`	$\longrightarrow$	`tip`	$\longrightarrow$	`ilk`	$\longrightarrow$	`jot`
`sob`		`for`		`sob`		`nob`
`nob`		`jot`		`nob`		`now`
`sky`		`hut`		`for`		`sky`
`hut`		`bet`		`jot`		`sob`
`ace`		`now`		`now`		`tag`
`bet`		`sky`		`hut`		`tip`

Each of the three sorts is carried out with respect to the digits in that column. "Stability" (and having previously sorted digits/suffixes to the right), means this achieves a sorting of the suffixes starting at the current column.

Radix Sort (cont'd)

Algorithm $\textsc{Radix-Sort}(A, d)$

1. **for** $i \leftarrow 0$ **to** d **do**
2. use stable sort to sort array A using digit i as key

Most commonly, $\textsc{Counting Sort}$ is used in line 2 - this means that once a set of digits is already in sorted order, then (by stability) performing $\textsc{Counting Sort}$ on the *next-most significant* digits preserves that order, within the "blocks" constructed by the new iteration.

Then each execution of line 2 requires time $\Theta(n + R)$.
Thus the overall time required by $\textsc{Radix-Sort}$ is

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

Sorting Integers with Radix-Sort

Theorem 2

An array of length n whose keys are b-bit numbers can be sorted in time

$$\Theta(n\lceil b/\lg n\rceil)$$

using a suitable version of $\textsc{Radix-Sort}$.

Proof: Let the digits be blocks of $\lceil \lg n \rceil$ bits. Then $R = 2^{\lceil \lg n \rceil} = \Theta(n)$ and $d = \lceil b/\lceil \lg n \rceil \rceil$. Using the implementation of $\textsc{Radix-Sort}$ based on $\textsc{Counting Sort}$ the integers can be sorted in time

$$\Theta(d(n+R)) = \Theta(n\lceil b/\lg n\rceil).$$

Note: If all numbers are at most n^k, then $b = k \lg n \ldots \Rightarrow$ Radix Sort is $\Theta(n)$ (assuming k is some constant, eg 3, 10).

Reading Assignment

[CLRS] Sections 8.2, 8.3

Problems

1. Think about the qn. on slide 7 - how do we get a very easy (non-stable) version of COUNTING-SORT if there are no items attached to the keys?
2. Can you come up with another way of achieving counting sort's $O(m+n)$-time bound and stability (you will need a different data structure from an array).
3. Exercise 8.3-4 of [CLRS].