05. Linear-Time Sorting and Selection

CPSC 535

Kevin A. Wortman

CALIFORNIA STATE UNIVERSITY FULLERTON

Big Idea in Computer Science

"What is old is new again": CS principles are solved science; society's needs, economic factors, and fads dictate which are prominent and which are in the background

- thin clients dominated the mainframe era; thick clients dominated the PC era; thin clients dominate the web app era
- memory conservation was critical prior to the 90s; programmer labor was more important, until mobile phones
- Unix rose (70s), fell (80s-90s), rose again (MacOS, iOS, Android, ChromeOS, Linux, PlayStation, embedded)
- algorithms were considered ivory-tower theory until recently

Protip: the CS material that seems irrelevant now, will probably become extremely marketable later in your career

Big Idea in Algorithm Design

Paramaterized complexity: algorithm complexity measured both in terms of input size n, **and** some parameter describing the values in the input

- machine word size W (e.g. $W = 64$ on modern PCs)
- $\#$ distinct values k

Pseudopolynomial: polynomial over both n, and also parameters

- radix sort takes $\Theta(nW)$ time
- strictly speaking W could be as large as n, so $\Theta(nW) = \Theta(n^2)$, unimpressive
- in practice all real-world computers have $W \in \Theta(1)$ so $\Theta(nW) = \Theta(n)$, faster than $\Theta(n \log n)$
- arguably defying the spirit of the Random Access Model

Tool to circumvent lower bounds, NP-hardness

The Lower Bound for the Sorting Problem

Recall the precise phrasing of the theorem:
Any comparison sort algorithm requires $\Omega(n \log n)$ *comparisons in the worst case.*

Bad news: $O(n \log n)$ “speed limit” for this important problem

Good news:

- optimal $\Theta(n \log n)$-time algorithms: mergesort, heapsort, quicksort
- loophole: theorem only applies to “comparison sorts”
- loophole: theorem applies to the general sorting problem, but we could make the problem more specific

Counting Sort Problem

Recall the classical *sorting problem:*
input: a sequence of n numbers $A = \langle a_1, a_2, \ldots, a_n \rangle$
output: a permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \ldots, a'_n$.

What if the inputs a_i are all bounded integers?

counting sort problem: (changes are underlined)
input: an integer $k \geq 0$, and a sequence of n integers $A = \langle a_1, a_2, \ldots, a_n \rangle$, where each $a_i \in [0, k]$
output: same

Turns out this change admits a $\Theta(n)$-time algorithm.

Counting Sort Idea

- each a_i can work as an array index
- count the number of occurrences of each value; create array C where

$$C[x] = \text{ the number of times that } x \text{ appears in A}$$

- let B be the output array
- use the counts in C to plan out which indices of B will hold each a_i
- fill in B using this information

Counting Sort Pseudocode

```
function COUNTING-SORT(A, B, k)                      ▷ A[1..n], B[1..n]
    allocate new array C[0, . . . , k], initialized to all zeroes
    for j from 1 to n do
        C[A[j]] + +                        ▷ C[i] is the number of elements = i
    end for
    for i from 1 to k do
        C[i] = C[i] + C[i − 1]                ▷ C[i] = count of elements ≤ i
    end for
    for j from n down to 1 do
        B[C[A[j]]] = A[j]
        C[A[j]] = C[A[j]] − 1
    end for
end function
```

Counting Sort Analysis

- allocate C: $\Theta(k)$ time
- first **for** loop: $\Theta(n)$ time
- second **for** loop: $\Theta(k)$ time
- third **for** loop: $\Theta(n)$ time
- total $= \Theta(k + n + k + n) = \Theta(2k + 2n) = \Theta(k + n)$ time

When Counting Sort Wins

If $k \in O(n)$, then counting sort takes $\Theta(k+n) = \Theta((n)+n) = \Theta(n)$ time.

Applications where $k \ll n$

- DNA sequences have only $k = 4$ bases A, C, G, T; human genome has $n \approx 3$ billion bases
- n-character ASCII string has only $k = 127$ character values
- log of website analytics has n hits but only k distinct web page URLs; each page is visited many times so $n \gg k$

When Counting Sort Loses

In the general sorting problem, each a_i is unbounded, so the maximum element could use $\Theta(n)$ bits, have value

$$a_i = 2^n = k$$

and force counting sort to take

$$\Theta(k + n) = \Theta((2^n) + n) = \Theta(2^n)$$

time, which is **exponential** in n and much more expensive than $\Theta(n \log n)$

$\implies$ counting sort is optimal when the software designer knows that the input is always a set of k integers with $k \in O(n)$

$\implies$ **but** if that is not guaranteed, comparison sorts are still optimal

Stable Sorting

stable sorting algorithm: does not swap order of ties

if $a_i = a_j$ and $i < j$ then $i' < j'$

Ex.: suppose we sort

$$[(47, a), (13, c), (28, b), (13, d)]$$

by the first element of each pair; a *stable* sort guarantees $(13, c)$ comes before $(13, d)$

Stability is a convenient, desirable property

Stable: insertion sort, mergesort, counting sort
Not stable: heapsort, quicksort

“Radix”

Vocabulary quiz:

- what does **radix** mean?
- where else do we use the word “radix”?

Radix Sort Overview

- make counting sort more robust to large elements
- sort one digit at a time
- i.e. sort by least-significant-digit, then by second-least-significant-digit, . . ., sort by most-significant digit
- e.g. to sort names in a spreadsheet: sort by first name, then by last name
- originally used by pre-digital punchcard sorting machines (what's old...)
- now used for parallel sort in GPU (...is new again)

Radix Sort Worked Example

(Sorting one base-10 digit at a time.)

7	7	5		7	9	1		5	2	1		0	5	3
0	5	3		5	2	1		2	2	5		2	2	5
7	6	2		7	6	2		3	3	4		3	3	4
7	9	1		0	5	3		0	5	3		5	2	1
6	7	4		6	7	4		7	6	2		6	7	4
5	2	1		3	3	4		6	7	4		7	6	2
3	3	4		7	7	5		7	7	5		7	7	5
2	2	5		2	2	5		7	9	1		7	9	1

Radix Sort, 1 bit at a time

function RADIX-SORT-1(A, W)
    for i from 1 to W **do**
        use a stable sort to sort A based only on bit position i
    end for
end function

Using counting sort as the stable sort, we have $k = 2$ (bit values 0 or 1) so each loop iteration takes $\Theta(k + n) = \Theta(2 + n) = \Theta(n)$ time

Clearly W iterations $\implies \Theta(nW)$ total time

Radix Sort, 8 bits at a time

```
function RADIX-SORT-8(A, W)
    for i from 1 to ⌈W/8⌉ do
        stably sort A on bits 8i − 7 through 8i
    end for
end function
```

$k = 2^8 = 256 \in \Theta(1)$
number of iterations is $\lceil nW/8 \rceil \in \Theta(n)$
$\implies$ still $\Theta(nW)$ time, but with different constant factors

Let r =#bits per pass; optimal choice of r minimizes

$$\lceil W/r \rceil \cdot (2^r + n)$$

Minimum and Maximum

```
function MINIMUM(A)
    min = A[1]
    for i from 2 to A.length do
        if min > A[i] then
            min = A[i]
        end if
    end for
    return min
end function
```

$\Theta(n)$ time

can also find maximum in $\Theta(n)$ time, or both in $\Theta(n)$ time

Selection Problem and Baseline Algorithm

input: array of n numbers $A = \langle a_1, a_2, \ldots, a_n \rangle$; index $i \in \{1, 2, \ldots, n\}$
output: the ith smallest element of A

```
function SELECTION-BY-SORTING(A, i)
    return MERGE − SORT(A)[i]
end function
```

Clearly $\Theta(n \log n)$ time

Surprise: selection can be solved in only $\Theta(n)$ time

Randomized Quicksort Review

```
function RQSORT(A, p, r)
    if p < r then
        q = RPART(A, p, r)
        RQSORT(A, p, q − 1)
        RQSORT(a, q + 1, r)
    end if
end function
```

Non-stable sort in $\Theta(n \log n)$ expected time but $\Theta(n^2)$ worst-case time

```
function RPART(A, p, r)
    k = random in [p, r]          ▷ pivot index
    swap(A[k], A[r])              ▷ move pivot aside
    pivot = A[r]                  ▷ pivot value
    i = p − 1                     ▷ i is first ≤pivot index
    for j from p to r − 1 do
        if A[j] ≤ pivot then
            i + +
            swap(A[i], A[j])
        end if
    end for
    swap(A[i + 1], A[r])          ▷ move pivot back
    return i + 1
end function
```

Randomized Selection Overview

- combining ideas from binary search and quicksort
- recursively search for ith smallest element
- do randomized partition; then
- three cases
  - pivot happens to be ith smallest
  - need to keep searching before pivot
  - need to keep searching after pivot
- expected runtime is $T(n) \approx T(n/2) + \Theta(n)$
- counterintuitively, that solves to $\Theta(n)$

Randomized Selection Pseudocode

```
function RSELECT(A, p, r, i)
    if p == r then
        return A[p]                          ▷ base case, done
    end if
    q = RPART(A, p, r)                       ▷ partition, q is pivot index
    k = q − p + 1                            ▷ k = number of elements before pivot
    if i == k then
        return A[q]                          ▷ pivot is answer
    else if i < k then
        return RSELECT(A, p, q − 1, i)
    else
        return RSELECT(A, q + 1, r, i − k)   ▷ i decreases by k
    end if
end function
```

Randomized Selection Analysis

- at most one recursive call, on $n/2$ elements on average
- partitioning takes $\Theta(n)$ time
- rest of algorithm takes $\Theta(1)$ time
- expected running time

  $$T(n) = T(n/2) + \Theta(n)$$

  which is only $\Theta(n)$ by master theorem
- worst case is the same for quicksort, extreme pivot at each step, $\Theta(n^2)$ time
- **takeaway:** randomized selection takes $\Theta(n)$ expected time and $\Theta(n^2)$ worst-case time

Deterministic Selection Overview

- **deterministic:** perfectly predictable; not randomized
- recall that $T(n) = T(fn) + \Theta(n)$ is $\Theta(n)$ for any fraction $0 < f < 1$, not just $f = 1/2$
- need: deterministic process to find a not-terrible pivot
- i.e. need at least fn elements on each side of the pivot, so that the worst-case recursive call is $T((1-f)n)$
- e.g. need at least $\frac{1}{3}n$ elements on each side of the pivot, so that there is a $T(\frac{1}{3}n)$ or $T(\frac{2}{3}n)$ call; worst-case is $T(\frac{2}{3}n)$; so

$$T(n) = T(\frac{2}{3}n) + \Theta(n)$$

which is still $\Theta(n)$ (though with worse constants)

Deterministic Selection Process

1. divide n elements into $\approx n/5$ groups of 5 elements each
2. find the median of each group with $SELECTION - BY - SORTING$; $\Theta(n(5 \log 5)) = \Theta(n)$ time
3. form a new array of the medians, and recursively select the median of this array = "median-of-medians"; $T(n/5)$ time
4. partition as usual, using median-of-medians as the pivot; $\Theta(n)$ time
5. same three cases: either pivot is answer, or recurse before pivot, or recurse after pivot; $T(\text{max. } \# \text{ elements on either side of pivot})$

Deterministic Selection Analysis

- let x be the median-of-medians; count elements $\geq x$
- suppose W.L.O.G. that input elements are distinct
- $\therefore$ at least half of the group-medians are $\geq x$
- $\therefore$ at least half of the groups contain at least 3 elements $\geq x$ each; except for the group containing x, and possibly one group with < 5 elements
- $\therefore$ #elements $\geq x$ is at least

$$3\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{5} \right\rceil \right\rceil - 2\right) \geq \frac{3}{10}n - 6$$

- symmetrically there are at least $\frac{3}{10}n - 6$ elements $\leq x$
- $\therefore$ recursively select at most $n - (\frac{3}{10}n - 6) = \frac{7}{10}n + 6$ elements

Deterministic Selection Analysis (continued)

For some $t \in \Theta(1)$,

$$T(n) \le \begin{cases} O(1) & n < t \\ T(\lceil n/5 \rceil) + T(\frac{7}{10}n + 6) + O(n) & n \ge t. \end{cases}$$

The master theorem does not apply, but the substitution method can be used to show $T(n) \in O(n)$.

Takeaway: Deterministic selection takes $O(n)$ worst-case time.

Surprise: selection can be **derandomized** from $O(n)$ expected time to $O(n)$ worst-case time with no asymptotic overhead.

Impractical; much worse constant factors, not usually worth it.