Linear Sorting (Chapter 8)

Comparison-based sorting problem has a worst-case lower bound of $\Omega(n \log n)$.

So to achieve better runtimes even in the worst case, we have to change the model.

Can we add more assumptions that an algorithm can use to do less work?

- Add more restrictions? (Recall that our sorts so far work on any comparable data)

- Could we have certain types of data where the runtime is vastly improved?
Huh? What do you mean, change the model?

**Spaghetti Sort**

- Take box of spaghetti.
- Cut each spaghetti stick to the size of the sort key.
- Take a piece of tape and tape it to the top of the highest stick.
- Take the stick attached to the sticky tape and put aside.
- Repeat, moving sticky tape down.
after a few iterations sorted to be sorted.
Memory “Sort” aka Hashtable \{ on non-negative integers \\
- allocate array \hat{H}[\text{MAXINT}] \\
- initialize all values in \hat{H} to 0 \\
- go through input array A \\
  \text{if } A[i] = k \text{ then } \hat{H}[k] ++; \}
  \hat{H}[A[i]] ++; \\
- go through \hat{H}, print out values. \\
\} \text{ If a cell has value } v > 0, \text{ then print the cell index } v \text{ times.}

# data comparisons? \\
# array reads? \\
\text{Side Note: Do we still require UNIQUE values for our input?}
Memory Sort:

- Number of comparisons: None!
  (Is this correct? What about integer comparisons? Okay, so maybe $O(n)$.)

- Number of array reads?

  $O(n)$?

  $O(\text{MAXINT})$?

  - MAXINT is a constant—so $O(1)$.
  - Hey, wait a minute!!!(How big do you think MAXINT is wrt $n$?)

So is this supposed to be our promised Linear Sort??
Well... maybe not...

What are the issues with Memory Sort?

1. Not Stable.
   (Debugged in class - ignore previous slide that indicated stability.)

2. MAXINT.

Can we address these issues?
Counting Sort: Intuition

- Done on integers
- Values do not need to be unique
- Three arrays:

\[ A: \begin{array}{cccccc}
2 & 5 & 3 & 0 & 2 & 3 \end{array} \quad \text{input}
\]

\[ B: \begin{array}{cccccc}
0 & 0 & 2 & 2 & 3 & 3 \end{array} \quad \text{output}
\]

\[ C: \begin{array}{cccccc}
2 & 0 & 2 & 3 & 0 & 1 \end{array} \quad \text{Scratch work array. Its size is based on the range of numbers in array A.}
\]

- Determine, for each input element \( x \) (in A), the number of elements less than \( x \). This information can be used to place element \( x \) directly into its position in output array B.

- What is \( C \) for? Count up how many occurrences of each input value. Then modify so each cell contains number of elements \( \leq \) to the cell index:

\[ \begin{array}{cccccc}
2 & 2 & 4 & 7 & 7 & 8 \end{array} \]

\[ \begin{array}{cccccc}
0 & 1 & 2 & 3 & 4 & 5 \end{array} \]
Counting Sort.

// input must be nonnegative ints
// A is input, B is output

// for each input A[i], find
// all values less than it.

1. Find MaxVal = Max(A) ← O(n)
2. Initialize all C[0,...,MaxVal] to zero.
3. For j = 1 to n
   C[A[j]] ++;
   // C[i] is #elements in A equal to i
4. For i = 1 to MaxVal
   C[i] = C[i] + C[i-1].
   // Now C[i] is #elements in A ≤ i
5. For k = n to 1
   Let val = A[k].  // value in A
   Let count = C[val].  // #elements ≤ val
   B[count] = val;
   C[val] = count - 1;
Counting Sort

What if our lowest value is much greater than \( 0 \)?

OR: What if we have some negative integers?
Counting Sort

What if our lowest value is $>0$?

OR: What if we have some negative integers?

- compute Max Val
- compute Min Val

\[ \text{We were already doing this.} \]

- give \( C \) size \( (\text{Max} - \text{Min}) \).
  \[ C[0... (\text{Max} - \text{Min})] \]

- subtract min from all values
- do Counting Sort as before.

- add min to all elements in \( B \).

Alternatively, subtract min from values used as indices to \( C \).
Counting Sort: Desirable Features.

Recall Vibha's last lecture:

**DATA STABILITY** means that items with the same keys stay in the same relative positions.

Why important?

- If satellite data carried around with element being sorted, we don't want to lose relative order.
- Sometimes we require stability for a sort because it is used as a subroutine for another sort that has this requirement. (We will see: RadixSort uses CountingSort)

- Unique keys - Not required.

- Are non-primitive keys allowed?
Runtime of Counting Sort (array reads, array writes)

\[ \Theta(\max(n, \text{MaxVal})) \]

or

\[ \Theta(\max(n, \text{range})) \]

Max Val - Min Val + 1 equivalent

= \Theta(n + \text{range})

If range is \( O(n) \), then algorithm is \( \Theta(n) \).

- This sort is good for a large set of values in a small range. (Lots of duplicates.)
- Is it stable?
- Is it efficient for sorting SS numbers?
Radix Sort for nonneg. integers

Sort one digit at a time

\[ \begin{align*}
329 & \quad 457 & \quad 329 & \quad 329 \\
457 & \quad 657 & \quad 839 & \quad 457 \\
657 & \quad 329 & \quad 457 & \quad 657 \\
839 & \quad 839 & \quad 657 & \quad 839
\end{align*} \]

What would happen if we sorted on most significant digit first?

sort rightmost digit first

sort leftmost digit last

Digit sorts must be STABLE.

- MaxVal on digit sort is 9.
  - So digit sort is \( O(\max(n,10)) \) \( \rightarrow O(n) \).
- So runtime of radix sort is \( O(dn) \)
  - \( d = \# \text{ digits} \)

Use Counting Sort on each column
Question about Format of Digits

What if we have different numbers of digits?

738
59
132
7
561

Need to “pad” with zeros: 738, 059, 132, 007, 561
A trivia moment...

What can this be used for?

For us old-timers...

- Used by card-sorting machines now found only in computer museums

- Sorter mechanically "programmed" to examine a given column of each card and distribute in one of 12 bins depending on where punched.
**General Radix Sort Runtime**

- $d = \# \text{ "digits"}$ (could be other data) \[\text{only } d \text{ passes are required.}\]
- $r = \text{ range of each digit}$
- $n = \# \text{ values}$.

Radix sort runtime is $O(d(n+r))$ (equivalent to $O(d \cdot \max(n, r))$)

If $d$ is fixed and $r \in O(n)$, then radix sort is linear.
Question 1: If we have $n$ $b$-bit integers, can we sort them in $\Theta(b \cdot n)$ time?

Yes, trivially:
- $d = b$ (# of bits)
- $r = 2$ (0, 1)
- $n = n$

$\Theta(d(n+r)) \Rightarrow \Theta(b(n+2)) \Rightarrow \Theta(bn)$
Question 2: How many bits are used to represent the numbers in the range 0...n-1?

$\log_2 n$, so $\Theta(n \log n)$ sort!
Question 3: What if we group the bits into clusters of size $r$?

Radix Sort $\in \Theta \left( \frac{b}{r} (n + 2^r) \right)$

Trade off is Speed vs. Memory used in Counting Sort Part
Claim: Given the number of bits to represent $n$ numbers is $O(\log n)$, then if we do all of the bits in a single grouping, Radix Sort runs in $\Theta(n)$ time.

$b \in O(\log n)$

Let $r = \log_2 n$ (i.e., # of bits)

Radix Sort is $\Theta\left(\frac{b}{r} (n + 2^r)\right)$ where $\frac{b}{r} = 1$

- $\Theta(n + 2^r)$
- $\Theta(n + 2^{\log_2 n})$
- $\Theta(n + n)$
- $\Theta(n)$

Lots of hidden constants + memory.
Thought Question

Can we sort n values that are in the range 0...n² in O(n) time?
Can we sort $n$ values that are in the range $0 \ldots n^2$ in $O(n)$ time?

- $n$ values in range $0$ to $n \times n$.  

- Can we take each value and "cut it in half"?

\[ \Theta(d(n+r)) \quad d=2 \]

\[ \Theta(2(n+n)) \quad r=0 \text{ to } n \]

\[ \Theta(n) \]

$\Rightarrow$ Hides work/memory for internal stable sort