RadixSorts
CSE 373 Winter 2020

Instructor: Hannah C. Tang

Teaching Assistants:
Aaron Johnston   Ethan Knutson   Nathan Lipiarski
Amanda Park      Farrell Fileas   Sam Long
Anish Velagapudi Howard Xiao   Yifan Bai
Brian Chan       Jade Watkins    Yuma Tou
Elena Spasova    Lea Quan
Announcements

❖ COVID-19 is really something, huh?
  ▪ **HW8**: No change, still due
  ▪ **Drop-in Times**: we’ll switch to online DITs next week
  ▪ **Workshops**: cancelled
  ▪ **Quiz sections**: since topic is final prep, may switch to online format or cancel
  ▪ **Final review session**: keeping, but in online format
  ▪ **Final exam**: still happening; online exam during **usual time slot** (Thu, Mar 19 2:30-4:20)
    • Please ensure you have access to a quiet location with good internet connectivity at that time!
  ▪ **Lectures**: today’s lecture finishes the topics that’ll be on the final exam
    • We’ll post pre-recorded videos for next week’s 3 lectures
    • Fortunately, topics were review + enrichment. Do your best ... or just use the time to finish HW8

❖ I’m insanely behind on email, but contact us anyway with questions, requests, etc
  ▪ We’ll announce details related to format, tools, etc on Piazza
  ▪ You’ll probably need to install Zoom (video conferencing)
Feedback from Reading Quiz

❖ How to handle non-numeric keys like \{♣, ♥, ♦\}?  
  ▪ Map keys to numeric values; exact implementation can vary  
  ▪ Eg: ♣ → 0, ♥ → 1, ♦ → 3

❖ We’ll answer these in lecture today:  
  ▪ What’s the runtime of counting sort? Is it \(\Theta(N^2)\) or \(\Theta(2N)\)?  
  ▪ What’s a radix?  
  ▪ How does radix sort maintain stability?  
  ▪ Can we use radix sort techniques for comparison sorts?
Lecture Outline

❖ Generalizing CountingSort

❖ RadixSort
  ▪ LSD RadixSort
  ▪ MSD RadixSort
Comparison-Based Sorting

- **Definition**: A type of sorting algorithm that determines an element’s ordering using *comparison operations*
  - More simply: sorting using only `compareTo()` type operations

- We determined the best we can do with comparison-based sorting is $\Theta(N \log N)$ time complexity

- Can we do better? What if we don’t compare at all?
Radix: A Definition

- **Radix**: the number of “characters” in the “alphabet”
  - More formally: the number of elements in the domain

<table>
<thead>
<tr>
<th>Name</th>
<th>Radix</th>
<th>Characters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary</td>
<td>2</td>
<td>0,1</td>
</tr>
<tr>
<td>Decimal</td>
<td>10</td>
<td>0,1,2,3,4,5,6,7,8,9</td>
</tr>
<tr>
<td>Lowercase Latin Alphabet</td>
<td>26</td>
<td>a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z</td>
</tr>
<tr>
<td>ASCII</td>
<td>128</td>
<td><a href="http://www.ascii-table.com/">http://www.ascii-table.com/</a></td>
</tr>
</tbody>
</table>
Reading Review: Generalizing CountingSort

- We want Counting Sort to work for non-unique and/or non-consecutive keys!
  - Count the occurrences for each key value
  - Compute each key’s starting index using the counts array
  - For each [item, key] in the input do:
    - Get the destination index by checking the index array for key
    - Copy item into the result using this destination index
    - Increment the index for key
  - Copy items back to initial array (if needed)

- Demo: https://docs.google.com/presentation/d/1FTTxlsd-7EqbJ6Md40svCV9zjDL-XxGI00pXp4gXsr8/edit
What is the runtime for CountingSort on an input of N items and an alphabet of size ("radix") R? Treat R as a variable, not a constant.

A. \( \Theta(N) \)
B. \( \Theta(R) \)
C. \( \Theta(N + R) \)
D. \( \Theta(NR) \)
E. I’m not sure ...
CountingSort: Performance Analysis

Time Complexity:

- \( \Theta(N) \)
- \( \Theta(R) \)
- \( \Theta(N) \)
- Overall: \( \Theta(N + R) \)

CountingSort(a):

```cpp
map<char, int> counts
foreach key in a:
    counts[key]++
map<char, int> indices;
foreach key in counts:
    indices[key] =
    indices[key - 1] +
    indices[key]
foreach (key, item) in a:
    output[indices[key]] =
    item;
    indices[key]++
```
CountingSort: Performance Analysis

- CountingSort is stable because it processes then input in order
  - No long-distance swaps like SelectionSort or Hoare Partitioning

- Runtime and memory use is $\Theta(N + R)!$
  - $N =$ # of items, $R =$ radix of alphabet

- We “beat” comparison sorts by avoiding comparisons!
  - Aaaaccccccttttually ... empirical/performance testing is still needed to compare against QuickSort on real-world inputs
You have an array of 100 elements, consisting of a city’s name and its population. If you want to sort them by population, which algorithm’s worst-case runtime \textit{as measured in seconds} (ie, not asymptotically) is lower / faster?

A. CountingSort
B. QuickSort
C. I’m not sure ...
CountingSort: Performance Analysis

- **Runtime and memory use** is $\Theta(N + R)!$
  - $N = \# \text{ of items}, R = \text{radix of alphabet}$

- But did we *actually* beat comparison sorts?
  - If $N \geq R$: performance is reasonable
  - If $N >> R$: $R$ is negligible, performance is great!
  - What if $N << R$?
    - In other words: When is our alphabet large?
    - Integers, strings, ...
Sorting Cities by Population

- CountingSort builds an array of size ~30,000,000 -- the largest city’s population -- to sort the input

- ... which is a very large and very sparse array
  - Most indices are unused because we are sorting only 100 cities!
Lecture Outline

❖ Generalizing CountingSort

❖ RadixSort
  ▪ LSD RadixSort
  ▪ MSD RadixSort
RadixSort’s Raison D'être

❖ We want to be able to sort keys that don’t belong to a finite alphabet, such as strings
   ▪ Strings don’t belong to a finite alphabet, but they **consist of characters** from a finite alphabet!
   ▪ Numbers do too

❖ RadixSort’s idea is similar to tries’:
   ▪ Subdivide the key; it’s not an atomic indivisible “whole”?
   ▪ Sort each chunk/character/digit independently using CountingSort

❖ How should we “chunk”? In what order should we process the chunks?
Least Significant Digit (LSD) RadixSort

❖ LSD RadixSort: Sort each chunk independently, from rightmost to leftmost

❖ Example:

<table>
<thead>
<tr>
<th>Key</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>Stitch</td>
</tr>
<tr>
<td>12</td>
<td>Gantu</td>
</tr>
<tr>
<td>31</td>
<td>Nani</td>
</tr>
<tr>
<td>23</td>
<td>Lilo</td>
</tr>
<tr>
<td>11</td>
<td>David</td>
</tr>
</tbody>
</table>

Alphabet: \{1, 2, 3\}
LSD RadixSort: Correctness

- Does LSD RadixSort create correct results?
  - What property of CountingSort enables that?
  - Can you give an example of what could go wrong?

<table>
<thead>
<tr>
<th>Key</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>Stitch</td>
</tr>
<tr>
<td>12</td>
<td>Gantu</td>
</tr>
<tr>
<td>31</td>
<td>Nani</td>
</tr>
<tr>
<td>23</td>
<td>Lilo</td>
</tr>
<tr>
<td>11</td>
<td>David</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>Nani</td>
</tr>
<tr>
<td>11</td>
<td>David</td>
</tr>
<tr>
<td>12</td>
<td>David</td>
</tr>
<tr>
<td>22</td>
<td>Stitch</td>
</tr>
<tr>
<td>12</td>
<td>Gantu</td>
</tr>
<tr>
<td>23</td>
<td>Lilo</td>
</tr>
<tr>
<td>22</td>
<td>Stitch</td>
</tr>
<tr>
<td>31</td>
<td>Nani</td>
</tr>
</tbody>
</table>
LSD RadixSort Correctness: More Formally

- If the **unexamined chunks** are **different**, the examined chunks don’t matter!
  - A later pass will sort correctly on more significant chunks

- If the **unexamined chunks** are **identical**, the keys are already properly ordered
  - Since the sort is stable, they will remain so
LSD RadixSort: Non-equal Key Lengths 🤔

- If keys are of unequal length, treat empty spaces as less-than all other chunks in the alphabet/domain

- Example:

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>is</td>
</tr>
<tr>
<td>31</td>
<td>fun!</td>
</tr>
<tr>
<td>23</td>
<td>duper</td>
</tr>
<tr>
<td>12</td>
<td>super</td>
</tr>
<tr>
<td>1</td>
<td>sorting</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>fun!</td>
</tr>
<tr>
<td>1</td>
<td>sorting</td>
</tr>
<tr>
<td>12</td>
<td>super</td>
</tr>
<tr>
<td>3</td>
<td>is</td>
</tr>
<tr>
<td>23</td>
<td>duper</td>
</tr>
<tr>
<td>31</td>
<td>fun!</td>
</tr>
</tbody>
</table>

Alphabet: {1, 2, 3}
LSD RadixSort: Runtime

- \( N = \# \text{ items}, \ R = \text{radix}, \ L = \# \text{ chunks in longest item} \)
  - We have to run CountingSort for each chunk
  - CountingSort has runtime on the order of \( \Theta(N + R) \)
  - Therefore, LSD RadixSort’s runtime: \( \Theta(LN + LR) \)
LSD RadixSort: Summary

❖ Use CountingSort on each chunk, from right to left
   ▪ Now we can sort non-alphabetic keys that consist of alphabetic keys!

❖ Performance (N = # items, R = radix, L = # chunks in longest item):
   ▪ Runtime: $\Theta(LN + LR)$
   ▪ Memory use: $\Theta(N + R)$
     • Output array: N
     • Need L counts array (R) and L starting indices array (R), but can reuse them between chunks

❖ If R is small (CountingSort’s restriction) and L is small (an LSD RadixSort restriction), the runtime isn’t shabby!

*If only the runtime didn’t depend on the longest key ... 😞*
Most Significant Digit (MSD) RadixSort

- By definition, LSD RadixSort examines the least significant chunk first!
  - ie, may do more computation than necessary

- MSD RadixSort Idea: similar to LSD, but leftmost to rightmost
  - Handles keys that are much longer than the rest, eg:

```
349499234
4589245
132954351638273762
62302213
2934592
432035235
```
MSD RadixSort: Example

Suppose we sort each chunk left to right. Will we arrive at the correct result? Why or why not?

<table>
<thead>
<tr>
<th>a</th>
<th>d</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>f</td>
<td>a</td>
<td>d</td>
</tr>
<tr>
<td>f</td>
<td>e</td>
<td>e</td>
</tr>
<tr>
<td>b</td>
<td>a</td>
<td>d</td>
</tr>
<tr>
<td>f</td>
<td>e</td>
<td>d</td>
</tr>
<tr>
<td>b</td>
<td>e</td>
<td>d</td>
</tr>
<tr>
<td>a</td>
<td>c</td>
<td>e</td>
</tr>
</tbody>
</table>
MSD RadixSort: Example

❖ No! Items that were previously ordered by a more-significant chunk may get swapped!
MSD RadixSort: Example

Solution: sort each subproblem separately, rejoin at the end
**MSD RadixSort: Example**

- Optimization: don’t subdivide or sort already-sorted singletons

```
| a | d | d |
| c | a | b |
| f | a | d |
| f | e | e |
| b | a | d |
| f | e | d |
| b | e | d |
| a | c | e |

| a | c | e |
| a | d | d |
| b | a | d |
| c | a | b |
| f | a | d |
| f | e | e |
| f | e | d |
| a | c | e |

| a | c | e |
| a | d | d |
| b | a | d |
| b | e | d |
| b | e | d |
| f | a | d |
| f | e | e |
| f | e | e |
```
MSD RadixSort: Runtime

- Best-case runtime of MSD RadixSort, expressed in N, R, L?
- What type of input leads to this best-case?
  - One CountingSort pass, looking only at the first chunk: $\Theta(N + R)$
  - Every input has a unique most-significant chunk

- Worst-case runtime of MSD RadixSort, expressed in N, R, L?
  - L CountingSort passes to look at every chunk (i.e., degenerates to LSD RadixSort): $\Theta(LN + LR)$
  - Every key is the same or only differs in the least-significant chunk
MSD RadixSort: Memory

- Memory usage: $\Theta(N + R)$
  - Output array: $N$
  - Each chunk requires <=R CountingSorts for each subproblem, and each CountingSort requires $N+R$ memory. However, we can reuse that memory between each CountingSort
MSD RadixSort: Analysis

❖ Runtime:
  ▪ Best case: \( \Theta(N + R) \)
  ▪ Worst case: \( \Theta(LN + LR) \)

❖ Memory usage: \( \Theta(N + R) \)

❖ In practice, long strings are rarely random; they may contain structure
  ▪ Eg, HTML has tags: <html>, <p>, <li>

❖ Structured strings may benefit from specialized sorting algorithms or, minimally, specialized “chunkers”
  ▪ Eg, a HTML-tag-aware chunking

From Algorithms, 4th edition by Sedgewick and Wayne
### tl;dr

<table>
<thead>
<tr>
<th>Sorting Algorithm</th>
<th>Time Complexity</th>
<th>Space Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Counting Sort</td>
<td>$\Theta(N+R)$</td>
<td>$\Theta(N+R)$</td>
</tr>
<tr>
<td>LSD Radix Sort</td>
<td>$\Theta(LN + LR)$</td>
<td>$\Theta(N + R)$</td>
</tr>
</tbody>
</table>
| MSD Radix Sort        | Best: $\Theta(N + R)$  
Worst: $\Theta(LN + LR)$ | $\Theta(N + LR)$  |
And, finally ...

- Thank you for your understanding and patience re: COVID-19
- Thank you for being a great class!
- Good luck on HW8 and the final. Stay in touch!