CSE 332 Summer 2024
Lecture 12: Sorting

Nathan Brunelle

http://www.cs.uw.edu/332
Properties To Consider

• Worst case running time
• In place:
  • We only need to use the pre-existing array to do sorting
  • Constant extra space (only some additional variables needed)
  • Selection Sort, Insertion Sort, Heap Sort
• Adaptive
  • The running improves as the given list is closer to being sorted
  • It should be linear time for a pre-sorted list, and nearly linear time if the list is nearly sorted
  • Insertion Sort
• Online
  • We can start sorting before we have the entire list.
  • Insertion Sort
• Stable
  • “Tied” elements keep their original order
Insertion Sort

• **Idea:** Maintain a sorted list prefix, extend that prefix by “inserting” the next element.
Heap Sort

- **Idea:** When “removing” an element from the heap, swap it with the last item of the heap then “pretend” the heap is one item shorter.
Divide And Conquer Sorting

• Divide and Conquer:
  • Recursive algorithm design technique
  • Solve a large problem by breaking it up into smaller versions of the same problem
Merge Sort

- **Base Case:**
  - If the list is of length 1 or 0, it’s already sorted, so just return it

- **Divide:**
  - Split the list into two “sublists” of (roughly) equal length

- **Conquer:**
  - Sort both lists recursively

- **Combine:**
  - **Merge** sorted sublists into one sorted list
Merge Sort In Action!

Sort between indices $low$ and $high$

Base Case: if $low == high$ then that range is already sorted!

Divide and Conquer: Otherwise call mergesort on ranges $\left( low, \frac{low + high}{2} \right)$ and $\left( \frac{low + high}{2} + 1, high \right)$

After Recursion:
Create a new array to merge into, and 3 pointers/indices:

• **L_next**: the smallest “unmerged” thing on the left
• **R_next**: the smallest “unmerged” thing on the right
• **M_next**: where the next smallest thing goes in the merged array

One-by-one: put the smallest of **L_next** and **R_next** into **M_next**, then advance both **M_next** and whichever of L/R was used.
Properties of Merge Sort

- Worst Case Running time:
  - $\Theta(n \log n)$
- In-Place?
  - No!
- Adaptive?
  - No!
- Stable?
  - Yes!
  - As long as in a tie you always pick l_next
Quicksort

• Like Mergesort:
  • Divide and conquer
  • $O(n \log n)$ run time (kind of...)

• Unlike Mergesort:
  • Divide step is the “hard” part
  • *Typically* faster than Mergesort
Quicksort

Idea: pick a pivot element, recursively sort two sublists around that element

- **Divide**: select pivot element $p$, $\text{Partition}(p)$
- **Conquer**: recursively sort left and right sublists
- **Combine**: Nothing!
Partition (Divide step)

Given: a list, a pivot $p$

Start: unordered list

Goal: All elements $< p$ on left, all $> p$ on right
Partition, Procedure

If $\text{Begin} \text{ value} < p$, move $\text{Begin}$ right
Else swap $\text{Begin}$ value with $\text{End}$ value, move $\text{End}$ left
Done when $\text{Begin} = \text{End}$
Partition, Procedure

If **Begin** value $< p$, move **Begin** right
Else swap **Begin** value with **End** value, move **End** Left
Done when **Begin** = **End**
Partition, Procedure

If Begin value $< \ p$, move Begin right
Else swap Begin value with End value, move End Left
Done when Begin = End

Case 1: meet at element $\leq \ p$
Swap $\ p$ with pointer position (2 in this case)
Partition, Procedure

If Begin value $< p$, move Begin right
Else swap Begin value with End value, move End Left
Done when Begin = End

Case 2: meet at element $> p$

Swap $p$ with value to the left (2 in this case)
Partition Summary

1. Put \( p \) at beginning of list

2. Put a pointer (Begin) just after \( p \), and a pointer (End) at the end of the list

3. While Begin < End:
   1. If Begin value < \( p \), move Begin right
   2. Else swap Begin value with End value, move End Left

4. If pointers meet at element < \( p \): Swap \( p \) with pointer position

5. Else If pointers meet at element > \( p \): Swap \( p \) with value to the left

Run time? \( O(n) \)
Conquer

Recursively sort Left and Right sublists

All elements < \( p \)

All elements > \( p \)

Exactly where it belongs!
Quicksort Run Time (Best)

If the pivot is always the median:

If the pivot is always the median:

Then we divide in half each time

\[ T(n) = 2T \left( \frac{n}{2} \right) + n \]

\[ T(n) = \mathcal{O}(n \log n) \]
Quicksort Run Time (Worst)

If the pivot is always at the extreme:

Then we shorten by 1 each time

\[ T(n) = T(n - 1) + n \]

\[ T(n) = O(n^2) \]
Quicksort Run Time (Worst)

\[
T(n) = T(n - 1) + n
\]

\[
T(n) = 1 + 2 + 3 + \cdots + n
\]

\[
T(n) = \frac{n(n + 1)}{2}
\]

\[
T(n) = O(n^2)
\]
Quicksort on a (nearly) Sorted List

First element always yields unbalanced pivot

So we shorten by 1 each time

\[ T(n) = T(n - 1) + n \]

\[ T(n) = O(n^2) \]
Good Pivot

• What makes a good Pivot?
  • Roughly even split between left and right
  • Ideally: median

• There are ways to find the median in linear time, but it’s complicated and slow and you’re better off using mergesort

• In Practice:
  • Pick a random value as a pivot
  • Pick the middle of 3 random values as the pivot
Properties of Quick Sort

• Worst Case Running time:
  • $\Theta(n^2)$
  • But $\Theta(n \log n)$ average! And typically faster than mergesort!

• In-Place?
  • ....Debatable

• Adaptive?
  • No!

• Stable?
  • No!
Improving Running time

• Recall our definition of the sorting problem:
  • Input:
    • An array $A$ of items
    • A comparison function for these items
      • Given two items $x$ and $y$, we can determine whether $x < y$, $x > y$, or $x = y$
  • Output:
    • A permutation of $A$ such that if $i \leq j$ then $A[i] \leq A[j]$

• Under this definition, it is impossible to write an algorithm faster than $n \log n$ asymptotically.

• Observation:
  • Sometimes there might be ways to determine the position of values without comparisons!
“Linear Time” Sorting Algorithms

• Useable when you are able to make additional assumptions about the contents of your list (beyond the ability to compare)
  • Examples:
    • The list contains only positive integers less than $k$
    • The number of distinct values in the list is much smaller than the length of the list

• The running time expression will always have a term other than the list’s length to account for this assumption
  • Examples:
    • Running time might be $\Theta(k \cdot n)$ where $k$ is the range/count of values
BucketSort

• Assumes the array contains integers between 0 and $k - 1$ (or some other small range)

• Idea:
  • Use each value as an index into an array of size $k$
  • Add the item into the “bucket” at that index (e.g. linked list)
  • Get sorted array by “appending” all the buckets
BucketSort Running Time

• Create array of $k$ buckets
  • Either $\Theta(k)$ or $\Theta(1)$ depending on some things...
• Insert all $n$ things into buckets
  • $\Theta(n)$
• Empty buckets into an array
  • $\Theta(n + k)$
• Overall:
  • $\Theta(n + k)$
• When is this better than mergesort?
Properties of BucketSort

• In-Place?
  • No

• Adaptive?
  • No

• Stable?
  • Yes!
RadixSort

• Radix: The base of a number system
  • We’ll use base 10, most implementations will use larger bases

• Idea:
  • BucketSort by each digit, one at a time, from least significant to most significant

| 103 | 801 | 401 | 323 | 255 | 823 | 999 | 101 | 113 | 901 | 555 | 512 | 245 | 800 | 018 | 121 |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0   | 1   | 2   | 3   | 4   | 5   | 6   | 7   | 8   | 9   | 10  | 11  | 12  | 13  | 14  | 15  |

Place each element into a “bucket” according to its 1’s place
RadixSort

• Radix: The base of a number system
  • We’ll use base 10, most implementations will use larger bases

• Idea:
  • BucketSort by each digit, one at a time, from least significant to most significant

<table>
<thead>
<tr>
<th></th>
<th>800</th>
<th>801</th>
<th>401</th>
<th>101</th>
<th>901</th>
<th>121</th>
<th>512</th>
<th>103</th>
<th>323</th>
<th>823</th>
<th>113</th>
<th>255</th>
<th>555</th>
<th>245</th>
<th>018</th>
<th>999</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Place each element into a “bucket” according to its 10’s place
RadixSort

• Radix: The base of a number system
  • We’ll use base 10, most implementations will use larger bases

• Idea:
  • BucketSort by each digit, one at a time, from least significant to most significant

Place each element into a “bucket” according to its 100’s place
RadixSort

- Radix: The base of a number system
  - We’ll use base 10, most implementations will use larger bases

- Idea:
  - BucketSort by each digit, one at a time, from least significant to most significant

Convert back into an array
RadixSort Running Time

• Suppose largest value is $m$
• Choose a radix (base of representation) $b$
• BucketSort all $n$ things using $b$ buckets
  • $\Theta(n + k)$
• Repeat once per each digit
  • $\log_b m$ iterations
• Overall:
  • $\Theta(n \log_b m + b \log_b m)$
• In practice, you can select the value of $b$ to optimize running time
• When is this better than mergesort?