CSE 373: Data Structures and Algorithms

Lecture 20: More Sorting

Instructor: Lilian de Greef
Quarter: Summer 2017
Today: More sorting algorithms!

• Merge sort analysis
• Quicksort
• Bucket sort
• Radix sort
Divide and conquer

Very important technique in algorithm design

1. Divide problem into smaller parts

2. Independently solve the simpler parts
   • Think recursion
   • Or parallelism

3. Combine solution of parts to produce overall solution

Two great sorting methods are fundamentally divide-and-conquer
(Merge Sort & Quicksort)
Merge Sort

Merge Sort: repeatedly...
- Sort the left half of the elements
- Sort the right half of the elements
- Merge the two sorted halves into a sorted whole

To sort array from position $lo$ to position $hi$:
- If range is 1 element long, it is already sorted!
- Else:
  - Sort from $lo$ to $(hi+lo)/2$
  - Sort from $(hi+lo)/2$ to $hi$
  - Merge the two halves together
Linked lists and big data

We defined sorting over an array, but sometimes you want to sort linked lists

One approach:
  • Convert to array:
  • Sort:
  • Convert back to list:

Merge sort works very nicely on linked lists directly
  • Heapsort and quicksort do not
  • Insertion sort and selection sort do but they’re slower

Merge sort is also the sort of choice for external sorting
  • Linear merges minimize disk accesses
  • And can leverage multiple disks to get streaming accesses
Analysis

Having defined an algorithm and argued it is correct, we should analyze its running time and space:

To sort $n$ elements, we:
- Return immediately if $n=1$
- Else do 2 subproblems of size and then an merge

Recurrence relation:
Analysis intuitively

This recurrence is common, you just “know” it’s $O(n \log n)$

Merge sort is relatively easy to intuit (best, worst, and average):
• The recursion “tree” will have height
• At each level we do a total amount of merging equal to
Analysis more formally
(One of the recurrence classics)

For simplicity, ignore constants (let constants be )
T(1) = 1
T(n) = 2T(n/2) + n
    = 2(2T(n/4) + n/2) + n
    = 4T(n/4) + 2n
    = 4(2T(n/8) + n/4) + 2n
    = 8T(n/8) + 3n
    ....
    = 2^kT(n/2^k) + kn

We will continue to recurse until we reach the base case, i.e. T(1) for T(1), \( n/2^k = 1 \), i.e., \( \log n = k \)

So the total amount of work is \( 2^kT(n/2^k) + kn = 2^{\log n}T(1) + n \log n = n + n \log n = O(n \log n) \)
Divide-and-Conquer Sorting

Two great sorting methods are fundamentally divide-and-conquer

1. Merge Sort:
   • Sort the left half of the elements (recursively)
   • Sort the right half of the elements (recursively)
   • Merge the two sorted halves into a sorted whole

2. Quicksort:
   • Pick a “pivot” element
   • Divide elements into “less-than pivot” and “greater-than pivot”
   • Sort the two divisions (recursively on each)
   • Answer is “sorted-less-than”, followed by “pivot”, followed by ”sorted-greater-than”
Quicksort Overview

1. Pick a pivot element

2. Partition all the data into:
   A. The elements less than the pivot
   B. The pivot
   C. The elements greater than the pivot

3. Recursively sort A and C

4. The final answer is A-B-C
Think in Terms of Sets

- **Select Pivot Value**

- **Partition S**

- **Quicksort(S₁) and Quicksort(S₂)**

- **Presto! S is sorted**

[Weiss]
Example, Showing Recursion

Divide
Divide
Divide
1 Element
Conquer
Conquer
Conquer
Details

Have not yet explained:

• How to pick the pivot element
  • Any choice is correct: data will end up sorted
  • But as analysis will show, want the two partitions to be about

• How to implement partitioning
  • In linear time
  • In place
Pivots

• Best pivot?
  • Halve each time

• Worst pivot?
  • Greatest/least element
  • Partition of size n - 1
Potential pivot rules

While sorting \( arr \) from \( lo \) to \( hi-1 \) ...

- **Pick** \( arr[lo] \) or \( arr[hi-1] \)
  - Fast, but worst-case occurs with mostly sorted input

- **Pick random element in the range**
  - Does as well as any technique, but (pseudo)random number generation can be slow
  - Still probably the most elegant approach

- **Median of 3, e.g.,** \( arr[lo], \) \( arr[hi-1], \) \( arr[(hi+lo)/2] \)
  - Common heuristic that tends to work well
Partitioning

Conceptually simple, but hardest part to code up correctly

- After picking pivot, need to partition in linear time in place

One approach (there are slightly fancier ones):

1. Swap pivot with \( \text{arr}[lo] \)
2. Use two fingers \( i \) and \( j \), starting at \( lo+1 \) and \( hi-1 \)
3. while (\( i < j \))
   - if (\( \text{arr}[j] > \text{pivot} \)) \( j-- \)
   - else if (\( \text{arr}[i] < \text{pivot} \)) \( i++ \)
   - else swap \( \text{arr}[i] \) with \( \text{arr}[j] \)
4. Swap pivot with \( \text{arr}[i] \) *

*skip step 4 if pivot ends up being least element
Example

• Step one: pick pivot as median of 3
  • $l_0 = 0$, $h_i = 10$

```
0 1 2 3 4 5 6 7 8 9
8 1 4 9 0 3 5 2 7 6
```

• Step two: move pivot to the $l_0$ position

```
0 1 2 3 4 5 6 7 8 9
6 1 4 9 0 3 5 2 7 8
```
Example

Now partition in place

Move fingers

Swap

Move fingers

Move pivot

Often have more than one swap during partition – this is a short example
Analysis

• Best-case: Pivot is always the median
  \[ T(0) = T(1) = 1 \]
  \[ T(n) = \quad \quad \text{-- linear-time partition} \]
  Same recurrence as merge sort:

• Worst-case: Pivot is always smallest or largest element
  \[ T(0) = T(1) = 1 \]
  \[ T(n) = \]
  Basically same recurrence as selection sort:

• Average-case (e.g., with random pivot)
  • \( O(n \log n) \), not responsible for proof (in text)
Cutoffs

• For small $n$, all that recursion tends to cost more than doing a quadratic sort
  • Remember asymptotic complexity is for

• Common engineering technique: switch algorithm below a cutoff
  • Reasonable rule of thumb: use insertion sort for $n < 10$

• Notes:
  • Could also use a cutoff for merge sort
  • Cutoffs are also the norm with parallel algorithms
    • Switch to sequential algorithm
  • None of this affects asymptotic complexity
Cutoff pseudocode

```c
void quicksort(int[] arr, int lo, int hi)
{
    if(hi - lo < CUTOFF)
        insertionSort(arr,lo,hi);
    else
        ...
}
```

Notice how this cuts out the vast majority of the recursive calls
   - Think of the recursive calls to quicksort as a tree
   - Trims out the bottom layers of the tree
Practice with comparison sort!

A comparison sorting algorithm is operating on an array of 8 integers. After its 4th loop or recursive call, the array looks like:

```
4  8  11  15  42  29  18  37
```

Which of these sorting algorithms can it be?
A) Heapsort
B) Merge sort
C) Insertion sort
D) Quicksort using Median of 3
(space for notes/scratch)
How Fast Can We Sort?

• Heapsort & mergesort have $O(n \log n)$ worst-case running time

• Quicksort has $O(n \log n)$ average-case running time

• These bounds are all tight, actually $\Theta(n \log n)$

• Comparison sorting in general is $\Omega(n \log n)$
  • An amazing computer-science result: proves all the clever programming in the world cannot comparison-sort in linear time
The Big Picture

Surprising amount of juicy computer science: 2-3 lectures...

- Simple algorithms: $O(n^2)$
  - Insertion sort
  - Selection sort
  - Shell sort
  ...  
- Fancier algorithms: $O(n \log n)$
  - Heap sort
  - Merge sort
  - Quick sort (avg)
  ...  
- Comparison lower bound: $\Omega(n \log n)$
- Specialized algorithms: $O(n)$
  - Bucket sort
  - Radix sort
- Handling huge data sets

How???
- Change the model – assume more than “compare(a,b)”
Bucket Sort (a.k.a. BinSort)

- If all values to be sorted are known to be integers between 1 and $K$ (or any small range):
  - Create an array of size $K$
  - Put each element in its proper bucket (a.k.a. bin)
  - If data is only integers, no need to store more than a count of how times that bucket has been used
- Output result via linear pass through array of buckets

<table>
<thead>
<tr>
<th>count array</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
</tbody>
</table>

- Example:
  
  $K=5$
  
  input $(5, 1, 3, 4, 3, 2, 1, 1, 5, 4, 5)$
  
  output
Analyzing Bucket Sort

• Overall: $O(n+K)$
  • Linear in $n$, but also linear in $K$
  • $\Omega(n \log n)$ lower bound does not apply because this is not a comparison sort

• Good when $K$ is smaller (or not much larger) than $n$
  • We don’t spend time doing comparisons of duplicates

• Bad when $K$ is much larger than $n$
  • Wasted space; wasted time during linear $O(K)$ pass

• For data in addition to integer keys, use list at each bucket
Bucket Sort with Data

• Most real lists aren’t just keys; we have data
• Each bucket is a list (say, linked list)
• To add to a bucket, insert in $O(1)$ (at beginning, or keep pointer to last element)

Example: spice level; scale 1-5;

1 = mild, 5 = very spicy

Input=

5: Habanero
3: Jalapeño
5: Ghost pepper
1: Bell pepper

<table>
<thead>
<tr>
<th>count array</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
</tbody>
</table>

• Result:
• Easy to keep ‘stable’; Habanero still before Ghost pepper
Radix sort

• Radix = “the base of a number system”
  • Examples will use 10 because we are used to that
  • In implementations use larger numbers
    • For example, for ASCII strings, might use 128

• Idea:
  • Bucket sort on one digit at a time
    • Number of buckets = radix
    • Starting with least significant digit
    • Keeping sort stable
  • Do one pass per digit
  • Invariant: After k passes (digits), the last k digits are sorted

• Aside: Origins go back to the 1890 U.S. census
### Radix Sort: Example

First pass: bucket sort by one’s digit

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>478</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>537</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>
</tr>
</tbody>
</table>

Second pass: **stable** bucket sort by ten’s digit

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>721</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>
</tr>
<tr>
<td>38</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Third pass: **stable** bucket sort by hundred’s digit

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>143</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>67</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Input: 478, 537, 9, 721, 3, 38, 143, 67
Output:
Analysis

Input size: $n$
Number of buckets = Radix: $B$
Number of passes = “Digits”: $P$

Work per pass is 1 bucket sort:

Total work is

Compared to comparison sorts, sometimes a win, but often not
  • Example: Strings of English letters up to length 15
    • Run-time proportional to: $15*(52 + n)$
    • This is less than $n \log n$ only if $n > 33,000$
    • Of course, cross-over point depends on constant factors of the implementations
      • And radix sort can have poor locality properties
Interactive Visualizations

Comparison Sort (including quicksort):
• http://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html

Bucket Sort:
• http://www.cs.usfca.edu/~galles/visualization/BucketSort.html
• http://www.cs.usfca.edu/~galles/visualization/CountingSort.html

Radix Sort:
• http://www.cs.usfca.edu/~galles/visualization/RadixSort.html
Sorting massive data

• Need sorting algorithms that minimize disk/tape access time:
  • Quicksort and Heapsort both jump all over the array, leading to expensive random disk accesses
  • Merge sort scans linearly through arrays, leading to (relatively) efficient sequential disk access

• Merge sort is the basis of massive sorting

• Merge sort can leverage multiple disks
External Merge Sort

• Sort 900 MB using 100 MB RAM
  • Read 100 MB of data into memory
  • Sort using conventional method (e.g. quicksort)
  • Write sorted 100MB to temp file
  • Repeat until all data in sorted chunks (900/100 = 9 total)

• Read first 10 MB of each sorted chuck, merge into remaining 10MB
  • writing and reading as necessary
  • Single merge pass instead of log n
  • Additional pass helpful if data much larger than memory

• Parallelism and better hardware can improve performance
• Distribution sorts (similar to bucket sort) are also used
Last Slide on Sorting

- Simple $O(n^2)$ sorts can be fastest for small $n$
  - Insertion sort (latter linear for mostly-sorted)
  - Good “below a cut-off” for divide-and-conquer sorts
- $O(n \log n)$ sorts
  - Heap sort, in-place, not stable, not parallelizable
  - Merge sort, not in place but stable and works as external sort
  - Quick sort, in place, not stable and $O(n^2)$ in worst-case
    - Often fastest, but depends on costs of comparisons/copies
- $\Omega (n \log n)$ is worst-case and average lower-bound for sorting by comparisons
- Non-comparison sorts
  - Bucket sort good for small number of possible key values
  - Radix sort uses fewer buckets and more phases
- Best way to sort?