Announcements

• HW4: Due tomorrow!

• Final in EXACTLY 2 weeks.
  – Start studying
CSE373: Data Structure & Algorithms
Beyond Comparison Sorting

Hunter Zahn
Summer 2016
Introduction to Sorting

• Stacks, queues, priority queues, and dictionaries all focused on providing one element at a time

• But often we know we want “all the things” in some order
  – Humans can sort, but computers can sort fast
  – Very common to need data sorted somehow
    • Alphabetical list of people
    • List of countries ordered by population
    • Search engine results by relevance
    • …

• Algorithms have different asymptotic and constant-factor trade-offs
  – No single “best” sort for all scenarios
  – Knowing one way to sort just isn’t enough
More Reasons to Sort

General technique in computing:

*Preprocess data to make subsequent operations faster*

Example: Sort the data so that you can

- Find the $k^{th}$ largest in constant time for any $k$
- Perform binary search to find elements in logarithmic time

Whether the performance of the preprocessing matters depends on

- How often the data will change (and how much it will change)
- How much data there is
The main problem, stated carefully

For now, assume we have $n$ comparable elements in an array and we want to rearrange them to be in increasing order.

**Input:**
- An array $A$ of data records
- A key value in each data record
- A comparison function

**Effect:**
- Reorganize the elements of $A$ such that for any $i$ and $j$,
- if $i < j$ then $A[i] \leq A[j]$
- (Also, $A$ must have exactly the same data it started with)
- Could also sort in reverse order, of course

An algorithm doing this is a comparison sort.
Sorting: The Big Picture

Surprising amount of neat stuff to say about sorting:

**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**
- External sorting
Insertion Sort

• Idea: At step $k$, put the $k^{th}$ element in the correct position among the first $k$ elements

• Alternate way of saying this:
  – Sort first two elements
  – Now insert 3$^{rd}$ element in order
  – Now insert 4$^{th}$ element in order
  – ...

• “Loop invariant”: when loop index is $i$, first $i$ elements are sorted

• Time?
  
  Best-case $O(n)$  
  Worst-case $O(n^2)$  
  “Average” case $O(n^2)$

  start sorted  
  start reverse sorted  
  (see text)
Selection sort

- Idea: At step $k$, find the smallest element among the not-yet-sorted elements and put it at position $k$
- Alternate way of saying this:
  - Find smallest element, put it 1$^{st}$
  - Find next smallest element, put it 2$^{nd}$
  - Find next smallest element, put it 3$^{rd}$
  - ... 
- “Loop invariant”: when loop index is $i$, first $i$ elements are the $i$ smallest elements in sorted order
- Time?
  - Best-case $O(n^2)$
  - Worst-case $O(n^2)$
  - “Average” case $O(n^2)$
  - $T(1) = 1$ and $T(n) = n + T(n-1)$
Bubble Sort

• Not intuitive – It’s unlikely that you’d come up with bubble sort

• It doesn’t have good asymptotic complexity: $O(n^2)$

• It’s not particularly efficient with respect to common factors

Basically, almost everything it is good at some other algorithm is at least as good at
Heap sort

• Sorting with a heap is easy:
  – insert each \( arr[i] \), or better yet use \texttt{buildHeap}
  – for(i=0; i < arr.length; i++)
    \hspace{1em} arr[i] = \texttt{deleteMin}();

• Worst-case running time: \( O(n \log n) \)

• We have the array-to-sort and the heap
  – So this is not an in-place sort
  – There’s a trick to make it in-place...
In-place heap sort sort

- Treat the initial array as a heap (via \texttt{buildHeap})
- When you delete the \textit{i}^{th} element, put it at \texttt{arr[n-i]}
  - That array location isn’t needed for the heap anymore!

\begin{figure}[h]
\centering
\begin{tikzpicture}[scale=1]
\draw[very thick, ->] (0,0) -- (1,0);
\draw[very thick, ->] (1,0) -- (2,0);
\draw[very thick, ->] (2,0) -- (3,0);
\draw[very thick, ->] (3,0) -- (4,0);
\draw[very thick, ->] (4,0) -- (5,0);
\draw[very thick, ->] (5,0) -- (6,0);
\draw[very thick, ->] (6,0) -- (7,0);
\draw[very thick, ->] (7,0) -- (8,0);
\draw[very thick, ->] (8,0) -- (9,0);
\draw[very thick, ->] (9,0) -- (10,0);
\draw[very thick, ->] (10,0) -- (11,0);
\node at (0.5,0) {4}; \node at (1.5,0) {7}; \node at (2.5,0) {5}; \node at (3.5,0) {9}; \node at (4.5,0) {8}; \node at (5.5,0) {6}; \node at (6.5,0) {10}; \node at (7.5,0) {3}; \node at (8.5,0) {2}; \node at (9.5,0) {1};
\end{tikzpicture}
\caption{Array representation of heap sort.}
\end{figure}

\[ \texttt{arr[n-i]} = \texttt{deleteMin()} \]
Divide-and-Conquer Sorting

Two great sorting methods are fundamentally divide-and-conquer

1. Mergesort: 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 then pivot then 
   sorted-greater-than
Example, Showing Recursion

Merge

Divide

Divide

Divide

1 Element

Merge

Merge

Merge

1 Element
Quicksort

• Also uses divide-and-conquer
  – Recursively chop into two pieces
  – Instead of doing all the work as we merge together, we will do all the work as we recursively split into halves
  – Unlike merge sort, does not need auxiliary space

• $O(n \log n)$ on average 😊, but $O(n^2)$ worst-case ☹️

• Faster than merge sort in practice?
  – Often believed so
  – Does fewer copies and more comparisons, so it depends on the relative cost of these two operations!
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 answer is, “as simple as A, B, C”

(Alas, there are some details lurking in this algorithm)
Think in Terms of Sets

1. Select pivot value
2. Partition S
3. Quicksort(S₁) and Quicksort(S₂)
4. 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 equal in size

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

• Best pivot?
  – Median
  – Halve each time

• Worst pivot?
  – Greatest/least element
  – Problem of size n - 1
  – $O(n^2)$
Potential pivot rules

While sorting `arr` from `lo` (inclusive) to `hi` (exclusive)...

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

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

• **Step one:** pick pivot as median of 3
  - \( lo = 0, hi = 10 \)

  ![Array](image)

  - **Step two:** move pivot to the \( lo \) position

  ![Updated Array](image)
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
  
  \[ T(0)=T(1)=1 \]
  
  \[ T(n)=2T(n/2) + n \] -- linear-time partition
  
  Same recurrence as mergesort: \( O(n \log n) \)

• **Worst-case**: Pivot is always smallest or largest element
  
  \[ T(0)=T(1)=1 \]
  
  \[ T(n) = 1T(n-1) + n \]
  
  Basically same recurrence as selection sort: \( O(n^2) \)

• **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 large $n$

• 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 skeleton

```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
The Big Picture

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

<table>
<thead>
<tr>
<th>Simple algorithms: O(n²)</th>
<th>Fancier algorithms: O(n log n)</th>
<th>Comparison lower bound: Ω(n log n)</th>
<th>Specialized algorithms: O(n)</th>
<th>Handling huge data sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion sort</td>
<td>Heap sort</td>
<td>Bucket sort</td>
<td>Specialized algorithms: O(n)</td>
<td>Handling huge data sets</td>
</tr>
<tr>
<td>Selection sort</td>
<td>Merge sort</td>
<td>Radix sort</td>
<td></td>
<td>External sorting</td>
</tr>
<tr>
<td>Shell sort</td>
<td>Quick sort (avg)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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 $\Theta(n \log n)$.

• So maybe we need to dream up another algorithm with a lower asymptotic complexity, such as $O(n)$ or $O(n \log \log n)$.
  
  – Instead: we know that this is impossible.

  • Assuming our comparison model: The only operation an algorithm can perform on data items is a 2-element comparison.
A General View of Sorting

• Assume we have *n* elements to sort
  – For simplicity, assume none are equal (no duplicates)

• How many *permutations* of the elements (possible orderings)?

• Example, *n*=3
  
  \[
  \]

• In general, *n* choices for least element, *n*-1 for next, *n*-2 for next, ...
  
  - \( n(n-1)(n-2)\cdots(2)(1) = n! \) possible orderings
Counting Comparisons

- So every sorting algorithm has to “find” the right answer among the $n!$ possible answers
  - Starts “knowing nothing”, “anything is possible”
  - Gains information with each comparison
  - **Intuition**: Each comparison can at best eliminate half the remaining possibilities
  - Must narrow answer down to a single possibility

- **What we can show**: Any sorting algorithm must do at least $(1/2)n \log n - (1/2)n$ (which is $\Omega(n \log n)$) comparisons
  - Otherwise there are at least two permutations among the $n!$ possible that cannot yet be distinguished, so the algorithm would have to guess and could be wrong [incorrect algorithm]
Optional: Counting Comparisons

• Don’t know what the algorithm is, but it cannot make progress without doing comparisons
  - Eventually does a first comparison “is $a < b$ ?"
  - Can use the result to decide what second comparison to do
  - Etc.: comparison $k$ can be chosen based on first $k-1$ results

• Can represent this process as a decision tree
  - Nodes contain “set of remaining possibilities”
    • Root: None of the $n!$ options yet eliminated
  - Edges are “answers from a comparison”
  - The algorithm does not actually build the tree; it’s what our proof uses to represent “the most the algorithm could know so far” as the algorithm progresses
Optional: One Decision Tree for n=3

- The leaves contain all the possible orderings of a, b, c
- A different algorithm would lead to a different tree
Optional: Example if $a < c < b$

```
[101x456]OpMonal:
  if
  a
  <
  c
  <
  b

Summer 2016
33
CSE373: Data Structures & Algorithms
```

possible orders

- $a < b < c$, $b < c < a$, $a < c < b$, $c < a < b$
- $b < a < c$, $c < b < a$

actual order

- $a < b < c$
- $a < c < b$
- $b < c$
- $b > c$
- $c < a$
- $c > a$

- $b < c < a$
- $b < a < c$
Optional: What the Decision Tree Tells Us

• A binary tree because each comparison has 2 outcomes
  – (We assume no duplicate elements)
  – (Would have 1 outcome if algorithm asks redundant questions) This means that poorly implemented algorithms could yield deeper trees (categorically bad)

• Because any data is possible, any algorithm needs to ask enough questions to produce all $n!$ answers
  – Each answer is a different leaf
  – So the tree must be big enough to have $n!$ leaves
  – Running *any* algorithm on *any* input will *at best* correspond to a root-to-leaf path in *some* decision tree with $n!$ leaves
  – So no algorithm can have worst-case running time better than the height of a tree with $n!$ leaves
    • Worst-case number-of-comparisons for an algorithm is an input leading to a longest path in algorithm’s decision tree
Optional: Where are we

- **Proven**: No comparison sort can have worst-case running time better than the height of a binary tree with $n!$ leaves
  - A comparison sort could be worse than this height, but it cannot be better

- **Now**: a binary tree with $n!$ leaves has height $\Omega(n \log n)$
  - Height could be more, but cannot be less
  - Factorial function grows very quickly

- **Conclusion**: Comparison sorting is $\Omega (n \log n)$
  - An amazing computer-science result: proves all the clever programming in the world cannot comparison-sort in linear time
Optional: Height lower bound

- The height of a binary tree with \( L \) leaves is at least \( \log_2 L \)
- So the height of our decision tree, \( h \):

\[
h \geq \log_2 (n!)
\]

- property of binary trees

\[
= \log_2 (n*(n-1)*(n-2)...(2)(1))
\]

- definition of factorial

\[
= \log_2 n + \log_2 (n-1) + ... + \log_2 1
\]

- property of logarithms

\[
\geq \log_2 n + \log_2 (n-1) + ... + \log_2 (n/2)
\]

- drop smaller terms

\[
\geq \log_2 (n/2) + \log_2 (n/2) + ... + \log_2 (n/2)
\]

- shrink terms to \( \log_2 (n/2) \)

\[
= (n/2)\log_2 (n/2)
\]

- arithmetic

\[
= (n/2)(\log_2 n - \log_2 2)
\]

- property of logarithms

\[
= (1/2)n \log_2 n - (1/2)n
\]

- arithmetic

\[
=“\ \Omega (n \log n)
\]

Height, or \# of comparisons made bounded by \( n \log n \)
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
- External sorting

How???
- Change the model – assume more than “compare(a,b)”
BucketSort (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: 1,1,1,2,3,3,4,4,5,5,5
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)

<table>
<thead>
<tr>
<th>count array</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Rocky V</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Harry Potter</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Casablanca, Star Wars</td>
</tr>
</tbody>
</table>

• Example: Movie ratings; scale 1-5; 1=bad, 5=excellent

Input=

5: Casablanca
3: Harry Potter movies
5: Star Wars Original Trilogy
1: Rocky V

• Result: 1: Rocky V, 3: Harry Potter, 5: Casablanca, 5: Star Wars
• Easy to keep ‘stable’; Casablanca still before Star Wars

Summer 2016
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 \textit{least} significant digit
    • Keeping sort \textit{stable}
  – Do one pass per digit
  – Invariant: After $k$ passes (digits), the last $k$ digits are sorted
**Example**

**Radix = 10**

<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></td>
<td>721</td>
<td>3</td>
<td>143</td>
<td>537</td>
<td>67</td>
<td>478</td>
<td>38</td>
<td>9</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Input:**
- 478
- 537
- 9
- 721
- 3
- 38
- 143
- 67

First pass:
- bucket sort by ones digit

Order now:
- 721
- 3
- 143
- 537
- 67
- 478
- 38
- 9
Example

Radix = 10

Order was:

Second pass:

stable bucket sort by tens digit

Order now:
### Example

Radix = 10

Order was:

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>9</td>
<td>721</td>
<td>537</td>
<td>38</td>
<td>143</td>
<td>67</td>
<td>478</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Order now:

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>9</td>
<td>143</td>
<td>478</td>
<td>537</td>
<td>721</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Third pass:

**stable** bucket sort by 100s digit
Analysis

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

Work per pass is 1 bucket sort: $O(B+n)$
Total work is $O(P(B+n))$

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
Sorting massive data

• Note: If data is on disk (ie too big to fit in main memory), reading and writing are much slower

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

• Mergesort is the basis of massive sorting

• Mergesort can leverage multiple disks
Last Slide on Sorting

• Simple $O(n^2)$ sorts can be fastest for small $n$
  – Selection sort, Insertion sort (latter linear for mostly-sorted)
  – Good for “below a cut-off” to help divide-and-conquer sorts

• $O(n \log n)$ sorts
  – Heap sort, in-place but not stable nor parallelizable
  – Merge sort, not in place but stable and works as external sort
  – Quick sort, in place but 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? It depends!