CSE332: Data Abstractions
Lecture 10: Comparison Sorting

James Fogarty
Winter 2012
**Introduction to Sorting**

- We have covered 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
  - Anyone can sort, but a computer can sort faster
  - Very common to need data sorted somehow
    - Alphabetical list of people
    - List of countries ordered by population

- Algorithms have different asymptotic and constant-factor trade-offs
  - No single “best” sort for all scenarios
  - Knowing “one way to sort” is not sufficient
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
  
  – How much data there is
Careful Statement of the Basic Problem

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 (potentially a set of fields)
- A comparison function (must be consistent and total)
  - Given keys \( a \) and \( b \), what is their relative ordering? \(<, =, >\)?

Effect:
- Reorganize the elements of \( A \) such that for any \( i \) and \( j \),
  \[ \text{if } i < j \text{ then } A[i] \leq A[j] \]
- Unspoken assumption: \( A \) must have all the data it started with

An algorithm doing this is a comparison sort
Variations on the basic problem

1. Maybe elements are in a linked list (could convert to array and back in linear time, but some algorithms need not do so)

2. Maybe ties need to be resolved by “original array position”
   - Sorts that do this naturally are called stable sorts
   - Others could tag each item with its original position and adjust their comparisons (non-trivial constant factors)

3. Maybe we must not use more than $O(1)$ “auxiliary space”
   - Sorts meeting this requirement are called in-place sorts

4. Maybe we can do more with elements than just compare
   - Sometimes leads to faster algorithms

5. Maybe we have too much data to fit in memory
   - Use an “external sorting” algorithm
Sorting: The Big Picture

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 input element in the correct position among the first $k$ elements

- Alternate way of saying this:
  - Sort first element (this is easy)
  - Now insert 2nd element in order
  - Now insert 3rd element in order
  - Now insert 4th element in order
  - ...

- "Loop invariant": when loop index is $i$, first $i$ elements are sorted

- **Time?**
  
  Best-case _____  Worst-case _____  "Average" case _____
**Insertion Sort**

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

- **Alternate way of saying this:**
  - Sort first element (this is easy)
  - Now insert 2$^{nd}$ element in order
  - 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 unsorted 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 _____  Worst-case _____  “Average” case _____
Selection Sort

- Idea: At step $k$, find the smallest element among the unsorted elements and put it at position $k$

- Alternate way of saying this:
  - Find smallest element, put it 1$^{\text{st}}$
  - Find next smallest element, put it 2$^{\text{nd}}$
  - Find next smallest element, put it 3$^{\text{rd}}$
  - ...

- “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 $O(n^2)$  
  
  Always $T(1) = 1$ and $T(n) = n + T(n-1)$
**Mystery Sort**

This is one implementation of which sorting algorithm (shown for ints)?

```java
void mystery(int[] arr) {
    for(int i = 1; i < arr.length; i++) {
        int tmp = arr[i];
        int j;
        for(j=i; j > 0 && tmp < arr[j-1]; j--)
            arr[j] = arr[j-1];
        arr[j] = tmp;
    }
}
```

**Note:** As with heaps, “moving the hole” is faster than unnecessary swapping (impacts constant factor)
**Insertion Sort vs. Selection Sort**

- They are different algorithms
- They solve the same problem
- Have the same worst-case and average-case asymptotic complexity
  - Insertion-sort has better best-case complexity; preferable when input is “mostly sorted”
- Other algorithms are more efficient
  *for non-small arrays that are not already almost sorted*
  - Small arrays may do well with Insertion sort
Aside: We Will Not Cover Bubble Sort

- It does not have good asymptotic complexity: $O(n^2)$

- It is not particularly efficient with respect to constant factors

- Almost everything it is good at, some other algorithm is at least as good at

- Perhaps some people teach it just because it was taught to them

- For fun see: “Bubble Sort: An Archaeological Algorithmic Analysis”, Owen Astrachan, SIGCSE 2003
Sorting: The Big Picture

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
Heap Sort

- As you are seeing in Project 2, sorting with a heap is easy:
  - insert each arr[i], or better yet do a buildHeap
  - for(i=0; i < arr.length; i++)
    arr[i] = deleteMin();

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

- 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**

- Treat the initial array as a heap (via `buildHeap`)
- When you delete the \(i\)th element, put it at `arr[n-i]`
  - That array location is not part of the heap anymore!

```
4 7 5 9 8 6 10 3 2 1
```

```
heap part  sorted part
```

```
arr[n-i]=
deleteMin()
```

```
5 7 6 9 8 10 4 3 2 1
```

```
heap part  sorted part
```

But this reverse sorts – how would you fix that?

Reverse your comparator, so you build a maxHeap
“AVL sort”

• We can also use a balanced tree to:
  – **insert** each element: total time $O(n \log n)$
  – Repeatedly **deleteMin**: total time $O(n \log n)$

• But this cannot be made in-place, and it has worse constant factors than heap sort
  – both are $O(n \log n)$ in worst, best, and average case
  – neither parallelizes well
  – heap sort is better

• Do not even think about trying to sort with a hash table
Divide and Conquer

Very important technique in algorithm design

1. Divide problem into smaller parts

2. Independently solve the simpler parts
   - Think recursion
   - Or potential parallelism

3. Combine solution of parts to produce overall solution
**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 ]
**Mergesort**

<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>
</tr>
</thead>
<tbody>
<tr>
<td><code>a</code></td>
<td>8</td>
<td>2</td>
<td>9</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>6</td>
</tr>
</tbody>
</table>

- To sort array from position `lo` to position `hi`:
  - If range is 1 element long, it is already sorted! (our base case)
  - Else, split into two halves:
    - Sort from `lo` to `(hi+lo)/2`
    - Sort from `(hi+lo)/2` to `hi`
    - Merge the two halves together

- Merging takes two sorted parts and sorts everything
  - $O(n)$ but requires auxiliary space…
Example: Focus on Merging

Start with:

After recursion:
(for now we just assume it works)

Merge:
Use 3 “fingers” and 1 more array

(After merge, copy back to original array)
Example: Focus on Merging

Start with:

\[
\begin{array}{cccccccc}
8 & 2 & 9 & 4 & 5 & 3 & 1 & 6 \\
\end{array}
\]

After recursion:
(For now we just assume it works)

\[
\begin{array}{cccccccc}
2 & 4 & 8 & 9 & 1 & 3 & 5 & 6 \\
\end{array}
\]

Merge:
Use 3 “fingers” and 1 more array

\[
\begin{array}{c}
1 \\
\end{array}
\]

(After merge, copy back to original array)
Example: Focus on Merging

Start with:

![Array 1](8, 2, 9, 4, 5, 3, 1, 6)

After recursion:
(for now we just assume it works)

![Array 2](2, 4, 8, 9, 1, 3, 5, 6)

Merge:
Use 3 “fingers” and 1 more array

(After merge, copy back to original array)
Example: Focus on Merging

Start with:

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

After recursion:

(for now we just assume it works)

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

Merge:
Use 3 “fingers” and 1 more array

(After merge, copy back to original array)
Example: Focus on Merging

Start with:

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

After recursion:

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

(for now we just assume it works)

Merge:
Use 3 “fingers” and 1 more array

(After merge, copy back to original array)
Example: Focus on Merging

Start with:

\[
\begin{array}{cccccccc}
8 & 2 & 9 & 4 & 5 & 3 & 1 & 6 \\
\end{array}
\]

After recursion:
(for now we just assume it works)

\[
\begin{array}{cccccccc}
2 & 4 & 8 & 9 & 1 & 3 & 5 & 6 \\
\end{array}
\]

Merge:
Use 3 “fingers” and 1 more array

(After merge, copy back to original array)
Example: Focus on Merging

Start with:

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

After recursion:
(for now we just assume it works)

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

Merge:
Use 3 “fingers” and 1 more array

```
1 2 3 4 5 6
```

(After merge, copy back to original array)
Example: Focus on Merging

Start with:

After recursion:
(for now we just assume it works)

Merge:
Use 3 “fingers” and 1 more array

(After merge, copy back to original array)
Example: Focus on Merging

Start with:

After recursion:
(for now we just assume it works)

Merge:
Use 3 “fingers” and 1 more array

(After merge, copy back to original array)
### Example: Focus on Merging

Start with:

| a | 8 | 2 | 9 | 4 | 5 | 3 | 1 | 6 |

After recursion: (for now we just assume it works)

| a | 2 | 4 | 8 | 9 | 1 | 3 | 5 | 6 |

Merge:

Use 3 “fingers” and 1 more array

| aux | 1 | 2 | 3 | 4 | 5 | 6 | 8 | 9 |

(After merge, copy back to original array)

| a | 1 | 2 | 3 | 4 | 5 | 6 | 8 | 9 |
Example: Mergesort Recursion
Mergesort: Some Time Saving Details

- What if the final steps of our merge looked like this:

  ![Diagram](image)

  - Wasteful to copy to the auxiliary array just to copy back…
Mergesort: Some Time Saving Details

- If left-side finishes first, just stop the merge and copy back:

- If right-side finishes first, copy dregs into right then copy back:
Mergesort: Saving Space and Copying

Simplest / Worst:
   Use a new auxiliary array of size \((\text{hi} - \text{lo})\) for every merge

Better:
   Use a new auxiliary array of size \(n\) for every merging stage

Better:
   Reuse same auxiliary array of size \(n\) for every merging stage

Best:
   Do not copy back after merge, instead swap usage of the original and auxiliary array (i.e., even levels move to auxiliary array, odd levels move back to original array)
      – Need one copy at end if number of stages is odd
Swapping Original and Auxiliary Array

- First recurse down to lists of size 1
- As we return from the recursion, swap between arrays

- Arguably easier to code without using recursion at all

Copy if Needed
Mergesort Analysis

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

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

Recurrence relation:
- $T(1) = c_1$
- $T(n) = 2T(n/2) + c_2 n$
Mergesort Analysis

This recurrence is common enough 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 $\log n$ height
- At each level we do a total amount of merging equal to $n$
Quicksort

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

- $O(n \log n)$ on average, but $O(n^2)$ worst-case
  - MergeSort is always $O(n \log n)$
  - So why use QuickSort at all?

- Can be faster than Mergesort
  - Believed by many to be faster
  - Quicksort 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
Quicksort: Think in Terms of Sets

S

S

S

13 81 43 31 57 75 0

select pivot value

partition S

QuickSort(S₁) and QuickSort(S₂)

Presto! S is sorted

[Weiss]
Example: Quicksort Recursion
Quicksort Details

We have not explained:

• How to pick the pivot element
  – Any choice is correct: data will end up sorted
  – But we 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 $n - 1$
  - $O(n^2)$
Quicksort: 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 approximately sorted input

• Pick random element in the range
  – Does as well as any technique
    • But 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 ($\text{arr}[j] \geq \text{pivot}$) $j--$
     else if ($\text{arr}[i] \leq \text{pivot}$) $i++$
     else swap $\text{arr}[i]$ with $\text{arr}[j]$
  4. Swap pivot with $\text{arr}[i]$
**Quicksort Example**

- **Step One:** Pick Pivot as Median of 3
  - \(lo = 0, \, hi = 10\)

  ![Array with pivots highlighted]

- **Step Two:** Move Pivot to the \(lo\) Position

  ![Array with pivot moved]

\[0 \, 1 \, 2 \, 3 \, 4 \, 5 \, 6 \, 7 \, 8 \, 9\]

\[\begin{array}{cccccccccc}
8 & 1 & 4 & 9 & 0 & 3 & 5 & 2 & 7 & 6 \\
\end{array}\]
Quicksort Example

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

Now partition in place

Move fingers

Swap

Move fingers

Move pivot
Quicksort Analysis

• Best-case: Pivot is always the median
  \[ T(0) = T(1) = 1 \]
  \[ T(n) = 2T(n/2) + n \quad \text{--- linear-time partition} \]
  Same recurrence as mergesort: \( O(n \log n) \)

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

• Average-case (e.g., with random pivot)
  – \( O(n \log n) \) (see text)