CSE 332: Sorting
Richard Anderson, Steve Seitz
Winter 2014
Announcements (2/3/14)

• Reading for this lecture: Chapter 7.
• HW 4 due Wednesday
  – no new HW out this week
• Midterm next Monday
Sorting

• Input
  – an array $A$ of data records
  – a key value in each data record
  – a comparison function which imposes a consistent ordering on the keys

• Output
  – “sorted” array $A$ such that
    • For any $i$ and $j$, if $i < j$ then $A[i] \leq A[j]$
Consistent Ordering

- The comparison function must provide a *consistent ordering* on the set of possible keys
  - You can compare any two keys and get back an indication of $a < b$, $a > b$, or $a = b$ (trichotomy)
  - The comparison functions must be consistent
    - If $\text{compare}(a, b)$ says $a < b$, then $\text{compare}(b, a)$ must say $b > a$
    - If $\text{compare}(a, b)$ says $a = b$, then $\text{compare}(b, a)$ must say $b = a$
    - If $\text{compare}(a, b)$ says $a = b$, then $\text{equals}(a, b)$ and $\text{equals}(b, a)$ must say $a = b$
Why Sort?

- Provides fast search: $O(\log n)$
- Find $k$th largest element in: $O(1)$
Space

• How much space does the sorting algorithm require?
  – In-place: no more than the array or at most $O(1)$ addition space
  – out-of-place: use separate data structures, copy back
  – External memory sorting – data so large that does not fit in memory
A sorting algorithm is **stable** if:

- Items in the input with the same value end up in the same order as when they began.

<table>
<thead>
<tr>
<th>Input</th>
<th>Unstable sort</th>
<th>Stable Sort</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adams</td>
<td>Adams 1</td>
<td>Adams 1</td>
</tr>
<tr>
<td>Black</td>
<td>Smith 1</td>
<td>Smith 1</td>
</tr>
<tr>
<td>Brown</td>
<td>Washington 2</td>
<td>Black 2</td>
</tr>
<tr>
<td>Jackson</td>
<td>Jackson 2</td>
<td>Jackson 2</td>
</tr>
<tr>
<td>Jones</td>
<td>Black 2</td>
<td>Washington 2</td>
</tr>
<tr>
<td>Smith</td>
<td>White 3</td>
<td>White 3</td>
</tr>
<tr>
<td>Thompson</td>
<td>Wilson 3</td>
<td>Wilson 3</td>
</tr>
<tr>
<td>Washington</td>
<td>Thompson 4</td>
<td>Brown 4</td>
</tr>
<tr>
<td>White</td>
<td>Brown 4</td>
<td>Jones 4</td>
</tr>
<tr>
<td>Wilson</td>
<td>Jones 4</td>
<td>Thompson 4</td>
</tr>
</tbody>
</table>

[Sedgewick]
Time

How fast is the algorithm?

– This means that you need to at least check on each element at the very minimum
  • Complexity is at least: $O(n)$
– And you could end up checking each element against every other element
  • Complexity could be as bad as: $O(n^2)$

The big question: How close to $O(n)$ can you get?
Sorting: The Big Picture

Simple algorithms: \( O(n^2) \)
- Insertion sort
- Selection 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
Demo (with sound!)

- http://www.youtube.com/watch?v=kPRA0W1kECg
Selection Sort: idea

1. Find the smallest element, put it 1\textsuperscript{st}
2. Find the next smallest element, put it 2\textsuperscript{nd}
3. Find the next smallest, put it 3\textsuperscript{rd}
4. And so on …
Try it out: Selection Sort

- 31, 16, 54, 4, 2, 17, 6
- 2 4 6 16 31 31 54 17
Selection Sort: Code

```c
void SelectionSort (Array a[0..n-1]) {
    for (i=0; i<n; ++i) {
        j = Find index of
            smallest entry in a[i..n-1]
        Swap(a[i],a[j])
    }
}
```

Runtime:

- worst case : \(O(n^2)\)
- best case : \(O(n)\) swap
- average case : \(O(n^2)\)
Bubble Sort

• Take a pass through the array
  – If neighboring elements are out of order, swap them.

• Repeat until no swaps needed

• Wost & avg case: $O(n^2)$
  – pretty much no reason to ever use this algorithm
Insertion Sort

1. Sort first 2 elements.
2. Insert $3^{rd}$ element in order.
   • (First 3 elements are now sorted.)
3. Insert $4^{th}$ element in order
   • (First 4 elements are now sorted.)
4. And so on…
How to do the insertion?

Suppose my sequence is:

16, 31, 54, 78, 32, 17, 6

And I’ve already sorted up to 78. How to insert 32?
Try it out: Insertion sort

- 31, 16, 54, 4, 2, 17, 6
void InsertionSort (Array a[0..n-1]) {
    for (i=1; i<n; i++) {
        for (j=i; j>0; j--) {
            if (a[j] < a[j-1])
                Swap(a[j], a[j-1])
            else
                break
        }
    }
}

Note: can instead move the “hole” to minimize copying, as with a binary heap.

Runtime:
worst case : $O(n^2)$
best case   : $O(n)$
average case: $O(n^2)$
Insertion Sort vs. Selection Sort

• Same worst case, avg case complexity
• Insertion better best-case
  – preferable when input is “almost sorted”
    • one of the best sorting algs for almost sorted case (also for small arrays)
Sorting: *The Big Picture*

- **Simple algorithms:** $O(n^2)$
  - Insertion sort
  - Selection 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: Sort with a Binary Heap

1. Build heap $O(n)$
2. deleteMin n times $O(n \log n)$

**Worst Case Runtime:** $O(n \log n)$
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]`
  - It’s not part of the heap anymore!

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

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

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

1. Build AVL tree (n inserts): $O(n \log n)$
2. In-order traversal: $O(n)$

Worst Case Runtime: $O(n \log n)$
“Divide and Conquer”

• Very important strategy in computer science:
  – Divide problem into smaller parts
  – Independently solve the parts
  – Combine these solutions to get overall solution

• **Idea 1**: Divide array in half, *recursively* sort left and right halves, then *merge* two halves
  → known as **Mergesort**

• **Idea 2**: Partition array into small items and large items, then recursively sort the two sets
  → known as **Quicksort**
Mergesort

- Divide it in two at the midpoint
- Sort each half (recursively)
- Merge two halves together
Mergesort Example

8 2 9 4 5 3 1 6

Divide

Divide

Divide

1 element

Merge

Merge

Merge

2 8

4 9

2 4 8 9

1 3 5 6

1 2 3 4 5 6 8 9
Merging: Two Pointer Method

• Perform merge using an auxiliary array
Merging: Two Pointer Method

- Perform merge using an auxiliary array
Merging: Two Pointer Method

- Perform merge using an auxiliary array

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

```
1  2  3  4  5
Auxiliary array
```
Merging: Finishing Up

Starting from here…

Left finishes up

or

Right finishes up
Merging: Two Pointer Method

• Final result

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
</table>

Auxiliary array

Complexity? $O(n)$

Stability? Yes if left wins on ties ($k$ keys)
Merging

```c
Merge(A[], Temp[], left, mid, right) {
    Int i, j, k, l, target
    i = left
    j = mid + 1
    target = left
    while (i < mid && j < right) {
        if (A[i] < A[j])
            Temp[target] = A[i++]
        else
            Temp[target] = A[j++]
        target++
    }
    if (i > mid) //left completed//
        for (k = left to target-1)
            A[k] = Temp[k];
    if (j > right) //right completed//
        k = mid
        l = right
        while (k > i)
            A[l--] = A[k--]
        for (k = left to target-1)
            A[k] = Temp[k]
}
```
Recursive Mergesort

MainMergesort(A[1..n], n) {
    Array Temp[1..n]
    Mergesort[A, Temp, 1, n]
}

Mergesort(A[], Temp[], left, right) {
    if (left < right) {
        mid = (left + right)/2
        Mergesort(A, Temp, left, mid)
        Mergesort(A, Temp, mid+1, right)
        Merge(A, Temp, left, mid, right)
    }
}

What is the recurrence relation?

\[ T(n) = \begin{cases} 
1 & \text{if } n = 1 \\
2T(n/2) + n & \text{otherwise}
\end{cases} \]
Mergesort: Complexity

\[ T(1) = 1 \quad T(n) = 2T(n/2) + n \]

\[ = 2(2T(n/4) + n/2) + n \]

\[ = 2(2(2T(n/8) + n/4) + n/2) + n \]

\[ = 8T(n/8) + 4n/4 + 2n/2 + n \]

\[ = 8T(n/8) + n + n + n \]

\[ = 8T(n/8) + 3n \]

\[ = 2^3T(n/2^3) + 3n \]

\[ = 2^K T(n/2^K) + Kn \]

\[ = n \cdot 1 + (\log_2 n)n \]

\[ \leq O(n \log n) \]
Iterative Mergesort

Merge by 1

Merge by 2

Merge by 4

Merge by 8
Iterative Mergesort reduces copying Complexity? $O(n \log n)$
Properties of Mergesort

- In-place? no
- Stable? yes
- Sorted list complexity? $O(n \log n)$
- Nicely extends to handle linked lists.
- Multi-way merge is basis of big data sorting.
- Java uses Mergesort on Collections and on Arrays of Objects.
Quicksort

Quicksort uses a divide and conquer strategy, but does not require the $O(N)$ extra space that MergeSort does.

Here’s the idea for sorting array $S$:

1. Pick an element $v$ in $S$. This is the *pivot* value.
2. Partition $S-\{v\}$ into two disjoint subsets, $S_1$ and $S_2$ such that:
   - elements in $S_1$ are all $\leq v$
   - elements in $S_2$ are all $\geq v$
3. Return concatenation of QuickSort($S_1$), $v$, QuickSort($S_2$)

Recursion ends when Quicksort( ) receives an array of length 0 or 1.
The steps of Quicksort

select pivot value

partition S

QuickSort(S₁) and QuickSort(S₂)

Presto! S is sorted

[Weiss]
Quicksort Example

1  2  3  4
  5  6  8  9

4  6  3  8  1  9  2  5

Divide
Conquer
1 element

Divide
Conquer

Divide
Conquer

Divide
Conquer

Divide
Conquer

Divide
Conquer

Divide
Conquer
The tricky parts are:

• **Picking the pivot**
  – Goal: pick a pivot value so that $|S_1|$ and $|S_2|$ are roughly equal in size.

• **Partitioning**
  – Preferably in-place
  – Dealing with duplicates
Picking the Pivot

median is ideal - too expensive
middle - could be really bad
median of 3 - good
more than 3, hierarchical
## Median of Three Pivot

Choose the pivot as the median of three.

Place the pivot and the largest at the right and the smallest at the left.
Quicksort Partitioning

• Partition the array into left and right sub-arrays such that:
  – elements in left sub-array are $\leq$ pivot
  – elements in right sub-array are $\geq$ pivot
• Can be done in-place with another “two pointer method”
  – Sounds like mergesort, but here we are partitioning, not sorting…
  – …and we can do it in-place.
Partitioning In-place

Setup: $i =$ start and $j =$ end of un-partitioned elements:

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

Advance $i$ until element $\geq$ pivot:

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

Advance $j$ until element $\leq$ pivot:

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

If $j > i$, then swap:

\[
\begin{array}{ccccccccc}
0 & 1 & 4 & 2 & 7 & 3 & 5 & 9 & 6 & 8
\end{array}
\]
Partitioning In-place

Advance $i$:

$\begin{array}{cccccccc}
0 & 1 & 4 & 2 & 7 & 3 & 5 & 9 & 6 & 8
\end{array}$

Advance $j$:

$\begin{array}{cccccccc}
0 & 1 & 4 & 2 & 7 & 3 & 5 & 9 & 6 & 8
\end{array}$

Swap:

$\begin{array}{cccccccc}
0 & 1 & 4 & 2 & 5 & 3 & 7 & 9 & 6 & 8
\end{array}$

Advance $i$:

$\begin{array}{cccccccc}
0 & 1 & 4 & 2 & 5 & 3 & 7 & 9 & 6 & 8
\end{array}$

Advance $j$:

$\begin{array}{cccccccc}
0 & 1 & 4 & 2 & 5 & 3 & 7 & 9 & 6 & 8
\end{array}$

$i > j$, swap in pivot, partition done!

$\begin{array}{cccccccc}
0 & 1 & 4 & 2 & 5 & 3 & 6 & 9 & 7 & 8
\end{array}$

$S_1 \leq \text{pivot}$ \hspace{1cm} \text{pivot} \hspace{1cm} S_2 \geq \text{pivot}$
Partition Pseudocode

Partition(A[], left, right) {
    v = A[right]; // Assumes pivot value currently at right
    i = left;    // Initialize left side, right side pointers
    j = right-1;

    // Do i++, j-- until they cross, swapping values as needed
    while (1) {
        while (A[i] < v) i++;  // Increment i
        while (A[j] > v) j--;  // Decrement j
        if (i < j) {
            Swap(A[i], A[j]);  // Swap i and j
            i++;  // Increment i
            j--;  // Decrement j
        } else
            break;
    }

    Swap(A[i], A[right]); // Swap pivot value into position
    return i;              // Return the final pivot position
}

Complexity for input size $n$?
Quicksort Pseudocode

Putting the pieces together:

```java
Quicksort(A[], left, right) {
    if (left < right) {
        medianOf3Pivot(A, left, right);
        pivotIndex = Partition(A, left+1, right-1);

        Quicksort(A, left, pivotIndex - 1);
        Quicksort(A, pivotIndex + 1, right);
    }
}
```
QuickSort: Best case complexity

```java
QuickSort(A[], left, right) {
    if (left < right) {
        medianOf3Pivot(A, left, right);
        pivotIndex = Partition(A, left+1, right-1);

        Quicksort(A, left, pivotIndex - 1);
        Quicksort(A, pivotIndex + 1, right);
    }
}
```
QuickSort: Worst case complexity

```c
void Quicksort(int A[], int left, int right) {
    if (left < right) {
        int medianOf3Pivot = medianOf3Pivot(A, left, right);
        int pivotIndex = Partition(A, left+1, right-1);

        Quicksort(A, left, pivotIndex - 1);
        Quicksort(A, pivotIndex + 1, right);
    }
}
```
QuickSort: **Average** case complexity

Turns out to be $O(n \log n)$.

See Section 7.7.5 for an idea of the proof. *Don’t need to know proof details for this course.*
Many Duplicates?

An important case to consider is when an array has many duplicates.

```
<table>
<thead>
<tr>
<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>8</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>0</td>
</tr>
</tbody>
</table>
```

medianOf3Pivot(...)

```
|   0   |   6   |   6   |   6   |   6   |   6   |   6   |   6   |   6   |   8   |
```
Partitioning with Duplicates

Setup: \( i = \) start and \( j = \) end of un-partitioned elements:

\[
\begin{array}{cccccccccc}
0 & 6 & 6 & 6 & 6 & 6 & 6 & 6 & 6 & 8 \\
\end{array}
\]

Advance \( i \) until element \( \geq \) pivot:

\[
\begin{array}{cccccccccc}
0 & 6 & 6 & 6 & 6 & 6 & 6 & 6 & 6 & 8 \\
\end{array}
\]

Advance \( j \) until element \( \leq \) pivot:

\[
\begin{array}{cccccccccc}
0 & 6 & 6 & 6 & 6 & 6 & 6 & 6 & 6 & 8 \\
\end{array}
\]

If \( j > i \), then swap:

\[
\begin{array}{cccccccccc}
0 & 6 & 6 & 6 & 6 & 6 & 6 & 6 & 6 & 8 \\
\end{array}
\]
Partitioning with Duplicates

<table>
<thead>
<tr>
<th>Advance $i, j$:</th>
<th>0</th>
<th>6</th>
<th>6</th>
<th>6</th>
<th>6</th>
<th>6</th>
<th>6</th>
<th>6</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Swap:</td>
<td>0</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>Advance $i, j$:</td>
<td>0</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>Swap:</td>
<td>0</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>Advance $i, j$:</td>
<td>0</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>Finish:</td>
<td>0</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>8</td>
</tr>
</tbody>
</table>
Partitioning with Duplicates: Take Two

Start \( i = \text{start and } j = \text{end of un-partitioned elements:} \)

\[
\begin{array}{cccccccccc}
0 & 6 & 6 & 6 & 6 & 6 & 6 & 6 & 6 & 8 \\
\end{array}
\]

Advance \( i \) until element > pivot (and in bounds):

\[
\begin{array}{cccccccccc}
0 & 6 & 6 & 6 & 6 & 6 & 6 & 6 & 6 & 8 \\
\end{array}
\]

Advance \( j \) until element < pivot (and in bounds):

\[
\begin{array}{cccccccccc}
0 & 6 & 6 & 6 & 6 & 6 & 6 & 6 & 6 & 8 \\
\end{array}
\]

Finish:

\[
\begin{array}{cccccccccc}
0 & 6 & 6 & 6 & 6 & 6 & 6 & 6 & 6 & 8 \\
\end{array}
\]

Is this better?
Partitioning with Duplicates: Upshot

It’s better to stop advancing pointers when elements are equal to pivot, and then just do swaps.

Complexity of quicksort on an array of identical values?

Can we do better?
Important Tweak

Insertion sort is actually better than quicksort on small arrays. Thus, a better version of quicksort:

```c
Quicksort(A[], left, right) {
    if (right - left ≥ CUTOFF) {
        medianOf3Pivot(A, left, right);
        pivotIndex = Partition(A, left+1, right-1);
        Quicksort(A, left, pivotIndex - 1);
        Quicksort(A, pivotIndex + 1, right);
    } else { 
        InsertionSort(A, left, right);
    }
}
```

CUTOFF = 10 is reasonable.
Properties of Quicksort

• $O(N^2)$ worst case performance, but $O(N \log N)$ average case performance.
• Pure quicksort not good for small arrays.
• No iterative version (without using a stack).
• “In-place,” but uses auxiliary storage because of recursive calls.
• Stable?
• Used by Java for sorting arrays of primitive types.