CSE 326: Data Structures
Sorting

Brian Curless
Spring 2008

Announcements (5/9/08)

- Project 3 is now assigned.
- Partnerships due by 3pm
  - We will not assume you are in a partnership unless you sign up!
- Homework #5 will be ready after class, due in a week.
- Reading for this lecture: Chapter 7.

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
  - reorganize the elements of $A$ 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?

- Allows binary search of an N-element array in $O(\log N)$ time
- Allows $O(1)$ time access to $k$th largest element in the array for any $k$
- Sorting algorithms are among the most frequently used algorithms in computer science

Space

- How much space does the sorting algorithm require in order to sort the collection of items?
  - Is copying needed?
    - In-place sorting algorithms: no copying or at most $O(1)$ additional temp space.
  - External memory sorting – data so large that does not fit in memory

Stability

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</td>
<td>Adams</td>
</tr>
<tr>
<td>Black</td>
<td>Smith</td>
<td>Smith</td>
</tr>
<tr>
<td>Brown</td>
<td>Washington</td>
<td>Black</td>
</tr>
<tr>
<td>Jackson</td>
<td>Jackson</td>
<td>Jackson</td>
</tr>
<tr>
<td>Jones</td>
<td>Black</td>
<td>Washington</td>
</tr>
<tr>
<td>Smith</td>
<td>White</td>
<td>White</td>
</tr>
<tr>
<td>Thompson</td>
<td>Wilson</td>
<td>Wilson</td>
</tr>
<tr>
<td>Washington</td>
<td>Thompson</td>
<td>Brown</td>
</tr>
<tr>
<td>White</td>
<td>Brown</td>
<td>Jones</td>
</tr>
<tr>
<td>Wilson</td>
<td>Jones</td>
<td>Thompson</td>
</tr>
</tbody>
</table>

[Sedgewick]

Time

How fast is the algorithm?
- The definition of a sorted array $A$ says that for any $i < j$, $A[i] \leq A[j]$
- This means that you need to at least check on each element at the very minimum
  - Complexity is at least: $\mathcal{O}(n)$
- And you could end up checking each element against every other element
  - Complexity could be as bad as: $\mathcal{O}(n^2)$

The big question is: How close to $O(n)$ can you get?
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 ...

Selection Sort: Code

\begin{verbatim}
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])
    }
}
\end{verbatim}

Runtime:
- worst case : $O(n^2)$
- best case : $O(n^2)$
- average case : $O(n^2)$
Bubble Sort Idea

- Take a pass through the array
  - If neighboring elements are out of order, swap them.
- Take passes until no swaps needed.

Try it out: Bubble Sort

- 31, 16, 54, 4, 2, 17, 6
  
  10 4 2 17 6 31 54

Bubble Sort: Code

```python
void BubbleSort (Array a[0..n-1]) {
    swapPerformed = 1
    while (swapPerformed) {
        for (i=0, i<n-1; i++) {
            if (a[i] < a[i+1]) {
                Swap(a[i], a[i+1])
                swapPerformed = 1
            }
        }
    }
}
```

2, 3, 4, 5, 7, 8, 9

Insertion Sort: Idea

1. Sort first 2 elements.
2. Insert 3rd element in order.
   - (First 3 elements are now sorted.)
3. Insert 4th element in order
   - (First 4 elements are now sorted.)
4. And so on...

Runtime:

- worst case : $O(n^2)$
- best case : $O(n)$
- average case : $O(n^2)$

Cocktail sort?
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?

Example

Try it out: Insertion sort

- 31, 16, 54, 4, 2, 17, 6

   16, 31, 54
   16, 31, 54
   4, 16, 31, 54
   4, 16, 31, 54

   16, 31, 54
   16, 31, 54
   4, 16, 31, 54
   4, 16, 31, 54
Insertion Sort: Code

```c
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: $O(n^2)$

Shell Sort: Idea

A small element at end of list takes a long time to percolate to front.

**Idea:** take bigger steps at first to percolate faster.

1. Choose offset $k$:
   a. Insertion sort over array: $a[0], a[k], a[2k], a[3k], ...$
   b. Insertion sort over array: $a[1], a[1+k], a[1+2k], a[1+3k], ...$
   c. Insertion sort over array: $a[2], a[2+k], a[2+2k], a[2+3k], ...$
   d. Do this until all elements touched

2. Choose smaller offset $m$, where $m$ is smaller than $k$, and do another set of insertion sort passes, stepping by $m$ through the array.

3. Repeat for smaller offsets until last pass uses offset $= 1$

[Named after the algorithm's inventor, Donald Shell.]

Try it out: Shell Sort

- Offsets: 3, 2, 1
- Input array: 31, 16, 54, 4, 2, 17, 6

Shell Sort: Code

```c
void ShellSort (Array a[0..n-1]) {
    determine good offsets based on n
    for (i=0, i<numOffsets; i++) {
        for (j=0, j<offsets[i]; j++) {
            insertionSort(a, j, offsets[i])
        }
    }
}
```

```c
void InsertionSkipSort (Array a[0..n-1],
                        Int start, Int offset) {
    Do insertion sort on array
    a[start], a[start+offset], a[start+2*offset],...
}
```
Shell Sort Offsets

The key to good Shell sort performance: **good offsets.**

Shell started the offset at ceil(n/2) and halved the offset each time. **Not good.**

Sedgewick proposed this offset sequence:
- Lowest offset is 1.
- Others are: $1 + 3 \cdot 2^i + 4^i$ for $i \geq 0$
- Looks like: 1, 8, 23, 77, 281, 1073, 4193, ...
- (Put in the offset array in reverse order to work with pseudocode on previous slide.)

Result:
- Worst case complexity is $O(n^{4/3})$
- Average case is believed to be $O(n^{7/6})$

Comb Sort

Could you do something like Shell Sort with bubble sort instead of insertion sort?

Yes! Called "Comb Sort". **Dobosiewicz Sort**

Complexity not well understood.

$$\text{Offset} = \sqrt[4]{1.3}$$

Heap Sort: Sort with a Binary Heap

1. BuildHeap() — $O(n)$
2. DeleteMin() until empty — $O(n \log n)$

Runtime: $O(n \log n)$

Try it out: Heap Sort

- 31, 16, 54, 4, 2, 17, 6
Binary Tree Sort

1. Insert all elements into AVL tree. $O(n\log n)$
2. Do in-order traversal. $O(n)$

Try it out: Binary Tree Sort

- 31, 16, 54, 4, 2, 17, 6

Runtime: $O(n\log n)$