CSE 332: Data Abstractions
Sorting I
Spring 2016

Announcements

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:
  • Find 4th largest element in:

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
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 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>Brown 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>

Time

How fast is the algorithm?

- requirement: 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:
    - And you could end up checking each element against every other element
      - Complexity could be as bad as:

  The big question: How close to \(O(n)\) can you get?

Sorting: The Big Picture

<table>
<thead>
<tr>
<th>Simple algorithms: (O(n^2))</th>
<th>Fancier algorithms: (O(n \log n))</th>
<th>Comparison lower bound: (\Omega(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>Comparison sort</td>
<td>Bucket sort</td>
<td>External sorting</td>
</tr>
<tr>
<td>Selection sort</td>
<td>Merge sort</td>
<td>Radix sort</td>
<td>Extreme sorting</td>
<td></td>
</tr>
</tbody>
</table>

Demo (with sound!)

- [http://www.youtube.com/watch?v=kPRA0W1kECq](http://www.youtube.com/watch?v=kPRA0W1kECq)

Selection Sort: idea

1. Find the smallest element, put it 1\(^{st}\)
2. Find the next smallest element, put it 2\(^{nd}\)
3. Find the next smallest, put it 3\(^{rd}\)
4. And so on …

Try it out: Selection Sort

- 31, 16, 54, 4, 2, 17, 6
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 :
- best case :
- average case :

Bubble Sort

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

- Worst & avg case: $O(n^2)$
  - pretty much no reason to ever use this algorithm

Insertion Sort

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...

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

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
        }
    }
}
```

**Runtime:**
- worst case :
- best case :
- average case :

Note: can instead move the “hole” to minimize copying, as with a binary heap.
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 algos for almost sorted case (also for small arrays)

Sorting: The Big Picture

- Simple algorithms: $O(n^2)$
- Fancier algorithms: $O(n \log n)$
- Comparison lower bound: $\Omega(n \log n)$
- Specialized algorithms: $O(n)$
- Handling huge data sets

Heap Sort: Sort with a Binary Heap

Worst Case Runtime:

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!

AVL Sort

Insert nodes into an AVL Tree
Conduct an In-order traversal to extract nodes in sorted order

Worst Case Runtime:

“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

Divide
Divide
Divide

Sort

Merge

Merging: Two Pointer Method

- Perform merge using an auxiliary array

Merging: Two Pointer Method

- Perform merge using an auxiliary array

Merging: Finishing Up

Starting from here…

Left finishes up

or

Right finishes up

Auxiliary array

Auxiliary array

Auxiliary array

Auxiliary array

Auxiliary array
Merging: Two Pointer Method

- Final result

1 2 3 4 5 6 8 9

Auxiliary array

Complexity? Stability?

Merging

Mergesort(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?

Mergesort: Complexity

Iterative Mergesort

Iterative Mergesort reduces copying
Complexity?
Properties of Mergesort

- In-place?
- Stable?
- Sorted list complexity?
- 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.