CSE 326: Data Structures
Sorting

Hal Perkins
Spring 2007
Lecture 17-18

Sorting: The Big Picture

Given \( n \) comparable elements in an array, sort them in an increasing (or decreasing) order.

Simple algorithms: \( O(n^2) \)
- Insertion sort
- Selection sort
- Bubble sort
- Shell sort
- ...

Fancier algorithms: \( O(n \log n) \)
- Heap sort
- Merge sort
- Quick sort
- ...

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 the \( k \)th step, put the \( k \)th input element in the correct place among the first \( k \) elements
- Result: After the \( k \)th step, the first \( k \) elements are sorted.

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

Selection Sort: idea

- Find the smallest element, put it 1st
- Find the next smallest element, put it 2nd
- Find the next smallest, put it 3rd
- And so on …
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 : 5
- best case   :
- average case : 5

HeapSort: Using Priority Queue ADT (heap)

Shove all elements into a priority queue, take them out smallest to largest.

Runtime:

Merge Sort

1. Split Array in half
2. Recursively sort each half
3. Merge two halves together

```
MergeSort (Array [1..n])
```

`Merge (a1[1..n], a2[1..n])`

```
Merge (a1[1..n], a2[1..n])
  i1=1, i2=1
  While (i1<n, i2<n) {
    if (a1[i1] < a2[i2]) {
      Next is a1[i1]
      i1++
    } else {
      Next is a2[i2]
      i2++
    }
  }
  Now throw in the dregs...
```

Merge Sort: Complexity
The steps of QuickSort

QuickSort Example

QuickSort Example
Recursive Quicksort

Quicksort(A[]): integer array, left,right : integer): {
    pivotindex : integer;
    if left + CUTOFF ≤ right then
        pivot := median3(A,left,right);
        pivotindex := Partition(A,left,right-1,pivot);
        Quicksort(A, left, pivotindex – 1);Quicksort(A, pivotindex + 1, right);
    else
        Insertionsort(A,left,right);
}

Don’t use quicksort for small arrays.
CUTOFF = 10 is reasonable.

QuickSort: Best case complexity

QuickSort: Worst 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.
Features of Sorting Algorithms

- **In-place**
  - Sorted items occupy the same space as the original items. (No copying required, only O(1) extra space if any.)
- **Stable**
  - Items in input with the same value end up in the same order as when they began.

Sort Properties

<table>
<thead>
<tr>
<th>Are the following:</th>
<th>stable?</th>
<th>in-place?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion Sort?</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Selection Sort?</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>MergeSort?</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>QuickSort?</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

How fast can we sort?

- Heapsort, Mergesort, and Quicksort all run in O(N log N) best case running time
- Can we do any better?
- No, if the basic action is a comparison.

Sorting Model

- Recall our basic assumption: we can only compare two elements at a time
  - we can only reduce the possible solution space by half each time we make a comparison
- Suppose you are given N elements
  - Assume no duplicates
- How many possible orderings can you get?
  - Example: a, b, c (N = 3)
Permutations

- How many possible orderings can you get?
  - Example: a, b, c (N = 3)
  - (a b c), (a c b), (b a c), (c a b), (c b a)
  - 6 orderings = 3·2·1 = 3! (i.e., “3 factorial”)
  - All the possible permutations of a set of 3 elements
- For N elements
  - N choices for the first position, (N-1) choices for the second position, …, (2) choices, 1 choice
  - N(N-1)(N-2) = N! possible orderings

Decision Tree

- The leaves contain all the possible orderings of a, b, c
- A binary tree of height h has at most how many leaves?
  - L
- A binary tree with L leaves has height at least:
  - h
- The decision tree has how many leaves:
  - 
- So the decision tree has height:
  - h

log(N!) is \( \Omega(N \log N) \)

\[
\log(N!) = \log(N \cdot (N-1) \cdot (N-2) \cdots (2) \cdot (1))
\]

\[
= \log N + \log(N - 1) + \log(N - 2) + \cdots + \log 2 + \log 1
\]

\[
\geq \log N + \log(N - 1) + \log(N - 2) + \cdots + \log \frac{N}{2}
\]

\[
\geq \frac{N}{2} \log \frac{N}{2}
\]

\[
\geq \frac{N}{2} (\log N - \log 2) = \frac{N}{2} \log N - \frac{N}{2}
\]

\[
= \Omega(N \log N)
\]
\[ \Omega(N \log N) \]
- Run time of any comparison-based sorting algorithm is \( \Omega(N \log N) \)
- Can we do better if we don’t use comparisons?

BucketSort (aka BinSort)
If all values to be sorted are known to be between 1 and \( K \), create an array \( \text{count} \) of size \( K \), increment counts while traversing the input, and finally output the result.

Example \( K=5 \). Input = (5,1,3,4,3,2,1,1,5,4,5)

<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>

Running time to sort \( n \) items?

BucketSort Complexity: \( O(n+K) \)
- Case 1: \( K \) is a constant
  - BinSort is linear time
- Case 2: \( K \) is variable
  - Not simply linear time
- Case 3: \( K \) is constant but large (e.g. \( 2^{32} \))
  - ???

Fixing impracticality: RadixSort
- Radix = “The base of a number system”
  - We’ll use 10 for convenience, but could be anything
- Idea: BucketSort on each digit, least significant to most significant (lsd to msd)
Radix Sort Example (1st pass)

Input data

<table>
<thead>
<tr>
<th>Input data</th>
<th>After 1st pass</th>
</tr>
</thead>
<tbody>
<tr>
<td>478</td>
<td>721</td>
</tr>
<tr>
<td>537</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>123</td>
</tr>
<tr>
<td>721</td>
<td>67</td>
</tr>
<tr>
<td>3</td>
<td>478</td>
</tr>
<tr>
<td>38</td>
<td>3</td>
</tr>
<tr>
<td>123</td>
<td>721</td>
</tr>
<tr>
<td>67</td>
<td>2</td>
</tr>
</tbody>
</table>

Bucket sort by 1's digit

This example uses B=10 and base 10 digits for simplicity of demonstration. Larger bucket counts should be used in an actual implementation.

Radix Sort Example (2nd pass)

Radix Sort Example (3rd pass)

RadixSort

- Input: 126, 328, 636, 341, 416, 131, 328

Your Turn

BucketSort on lsd:

BucketSort on next-higher digit:

BucketSort on msd:
Radixsort: Complexity

- How many passes?
- How much work per pass?
- Total time?
- Conclusion?

In practice
- RadixSort only good for large number of elements with relatively small values
- Hard on the cache compared to MergeSort/QuickSort

Internal versus External Sorting

- Need sorting algorithms that minimize disk/tape access time
- **External sorting** – Basic Idea:
  - Load chunk of data into RAM, sort, store this “run” on disk/tape
  - Use the Merge routine from Mergesort to merge runs
  - Repeat until you have only one run (one sorted chunk)
- Text gives some examples