R · I · T
Department of Computer Science

Introduction to Sorting

or, How to get all your ducks in a row....

Searching review

- We looked at two approaches to searching
  - Linear (serial) search
    - Best case: $\Theta(1)$
    - Worst case: $\Theta(n)$
    - Average case: $\Theta(n)$
  - Binary search
    - Best case: $\Theta(1)$
    - Worst case: $\Theta(\log n)$
    - Average case: $\Theta(\log n)$

The need for sorting

- Why should we worry about sorting data?
  - Binary search needs to have the data in sorted order
  - Being able to sort data is good for other reasons, too, including presentation to human beings.

Options in sorting

- There are a lot of algorithms for sorting, including:
  - Selection sort
  - Insertion sort
  - Bubble sort
  - Merge sort
  - Quick sort
- Some of these are **much** better to use than others....

Some not-so-good algorithms

- Selection sort
- Insertion sort
- Bubble sort

Selection sort

- You may have done this in one of your labs for CS1
- The algorithm is as follows:
  1) Find the smallest integer in the list
  2) Swap it with the first element in the list
  3) Repeat steps 1 and 2 with the remaining data in the list

Write the swap method

```
// A convenient function to use in our discussions of
// sorting data
public void swap( int [] data, int first, int second )
{
    [code written in class]



}
```

Selection sort algorithm

- [See SelectionSort.java]

Analysis of selection sort

- What is the worst-case time for selection sort?
  - Answer: **$O(n^2)$**
- What is the best-case time for selection sort?
  - Answer: **$O(n^2)$**
- Overall performance: **$\Theta(n^2)$**

Insertion sort algorithm

- Algorithm makes (N-1) passes through the data
  - During passes P = 1 through N-1:
    - we're trying to find the correct position for element #P in the list
    - we assume that everything before element #P is already in sorted order
    - we move everything that's bigger than the value at P up one spot, and then put element #P into the gap this opens
- [See InsertionSort.java]

Analysis of insertion sort

- What is the worst-case time for insertion sort?
  - Answer: **O(n^2)**
- What is the best-case time for insertion sort?
  - Answer: **O(n)**
  - Occurs when the data is already sorted

Bubble sort algorithm

- Repeat the following steps until the data is sorted:
  - Go through the array from left to right
  - If an array element is larger than its right neighbor, swap the two elements
  - If you make it all the way through the array without making a swap, the data is sorted.
- [See BubbleSort.java]

Analysis of bubble sort

- What is the worst-case time for bubble sort?
  - Answer: **$O(n^2)$**
- What is the best-case time for bubble sort?
  - Answer: **$O(n)$**
  - Occurs when the data is already sorted

Ranking the algorithms thus far

- The ranking of these "not so good" algorithms is as follows:
  - Selection sort is worst
    - Best/worst case performance is $O(n^2)$
  - Bubble sort is not quite as bad
    - $O(n)$ best case, $O(n^2)$ worst case
  - Insertion sort is somewhat better still
    - $O(n)$ best case, $O(n^2)$ worst case
    - Lower multiplicative constant than Bubble sort

Some *good* algorithms

- Merge sort
- Quick sort

Merge sort algorithm

- Take the array to be sorted
  - If its size is 1 (or 0), it's already sorted, so we're done
  - Otherwise:
    - Split it into two halves of roughly equal size
    - Sort each of the halves (recursively)
    - Create a new (temporary) array, big enough to hold a copy of the original array
    - Merge the two sorted halves together into the new array
    - Copy the contents of the new array back into the original one
- [See MergeSort.java]

Analysis of merge sort

- What is the worst-case time for merge sort?
  - Answer: **O(n log n)**
- What is the best-case time for merge sort?
  - Answer: **O(n log n)**
- Overall performance: **Θ(n log n)**

One caveat....

- Merge sort is a reasonably efficient sort, unlike the others so far, but there's a catch: can you see it?
- The problem with this algorithm that it can require a <u>lot</u> of space (due to the need to make a short-lived copy of the entire data set while merging)
  - If you're going to be working with really big data sets <u>in memory</u>, you typically won't use this.
  - On the other hand, merge sort is frequently used if you're doing an "external sort" (e.g., sorting data on disk, etc.)

Quick Sort

- The fastest known (general) sorting algorithm in practice
  - Average running time is O(n log n)
  - Worst-case is $O(N^2)$, but you can code the algorithm so that this is unlikely to occur
- This algorithm uses a recursive "divide and conquer", similar to merge sorting

Quick sort algorithm

- Given some set of data to be quick-sorted:
  - If the number of elements to be sorted is 0 or 1, then return
  - Pick some element in the data set.
    - This is called the "pivot".
  - Reorder the elements in the set so that:
    - Every value *less than* (or equal to) the pivot is to its left
    - Every value *larger than* the pivot is to its right
  - Finally:
    - quick-sort the sub-array to the left of the pivot
    - quick-sort the sub-array to the right of the pivot

Quick sort vs. Merge sort

- Advantages of quick sorting:
  - The memory issue with merge sort
  - The "hidden constant" (in the "O(n log n)") is smaller for quick sort than for merge sort
- Advantages of merge sorting:
  - better "worst case" behavior
- In general?
  - They're both optimal, in that <u>any</u> general sorting algorithm can't do better than O(n log n) for average performance

Department of
Computer Science

Any questions?