CSCI 136 Data Structures & Advanced Programming

Simple Sorting Algorithms

Introduction to Sorting

- Along with search, sorting is among the most ubiquitous computations
- Simple Examples
 - Reordering a list of integers so that they are in increasing order
 - Reordering a list of strings so that they are in alphabetical (*lexicographic*) order
 - Ordering the results of a Google search by decreasing relevance

Introduction to Sorting

More Examples

- Reordering a list of strings so that they are in lexicographical order by length
 - Shorter strings come before longer strings and
 - Equal-length strings are in lexicographic order
- Ordering tracks in a music collection by artist
 - Breaking ties by title
 - Breaking remaining ties by release date
- If you can order it, you can sort it!
- So, let's look at some simple sorting methods

Introduction to Sorting

In this video we will

- Introduce three simple sorting algorithms
 - Bubble Sort, Insertion Sort, and Selection Sort
- Discuss their implementation
- Discuss their time and space complexity
 - Identify some sorting-specific measures of complexity
- Introduce some notation for describing lower bounds

- First Pass:
 - $(5 \underline{1} 3 2 9) \rightarrow (\underline{1} 5 3 2 9)$
 - $(| 5 \underline{3} 29) \rightarrow (| \underline{3} 5 29)$
 - $(| 3 5 \underline{2} 9) \rightarrow (| 3 \underline{2} 5 9)$
 - $(| 3 2 5 \underline{9}) \rightarrow (| 3 2 5 \underline{9})$
- Second Pass:
 - $(| \underline{3} 2 5 9) \rightarrow (| \underline{3} 2 5 9)$
 - $(| 3 \underline{2} 5 9) \rightarrow (| \underline{2} 3 5 9)$
 - $(| 2 3 \underline{5} 9) \rightarrow (| 2 3 \underline{5} 9)$

- Third Pass:
 - (| <u>2</u>359) -> (| <u>2</u>359)
 - (|**2**<u>3</u>59)->(|**2**<u>3</u>59)
- Fourth Pass:
 - (| <u>2</u>359) -> (| <u>2</u>359)
- Finished!

http://www.visualgo.net/sorting http://www.youtube.com/watch?v=lyZQPjUT5B4

Bubble sort uses a utility method swap

```
private static void swap(int[]A, int i, int j) {
       int temp = a[i];
      A[i] = A[j];
      A[j] = temp;
}
public static void bubbleSort(int[] A) {
       for(int i = 1; i < A.length; i++)</pre>
              // Process all but last i-1 elements
              for(int j = 0; j < A.length -i; j++)
                     if( A[j] > A[j+1]) swap(A, j, j+1);
}
```

The only subtlety: Do loops start and end at reasonable points!

- Repeatedly scans through the list to be sorted, comparing two items at a time and swapping them if they are in the wrong order
 - Works on smaller initial slice each time
 - Can be improved to stop after a "swap-free" scan
- Gets its name from the way larger elements "bubble" to the end of the list
- Time complexity?
 - O(n²) : Might perform O(n²) compares and O(n²) swaps
- Space complexity?
 - O(n) total (very little additional space is required)
- It's a Stable sorting method
 - Equal elements remain in same relative positions

Counting Operations (where n = A.length)

Outer loop executes n-1 times

}

- for ith iteration: inner loop executes n-i times
 - Performing, say at most, say, 5 operations each time

$$\sum_{i=1}^{n-1} 5(n-i) = \sum_{i=1}^{n-1} 5n - \sum_{i=1}^{n-1} 5i = (5n(n-1)) - (5\sum_{i=1}^{n-1} i) = 5n(n-1) - (5n(n-1))/2$$

• This equals $\frac{5n(n-1)}{2}$ which is $O(n^2)$

Aside: Lower Bound Notation

There are situations in which bubble sort must necessarily perform a quadratic number of operations.

- Any "almost reverse-sorted" list will cause this This observation describes a lower bound on the (worst-case) running time of the algorithm
- It's useful to have notation for lower-bound claims, similar to the Big-O notation for upper bound
- It exists: It's called "Big-Ω" (Big Omega) notation

Aside: Lower Bound Notation

Definition: A function f(n) is $\Omega(g(n))$ if for some constant c > 0 and all $n \ge n_0$ $f(n) \ge c g(n)$

So, f(n) is $\Omega(g(n))$ exactly when g(n) is O(f(n))

All three sorting algorithms have time complexity

- O(n²) : Never use more than cn² operations
- Ω(n²) : Sometimes use at least cn² operations
 When f(n) is O(g(n)) and f(n) is Ω(g(n)) we write: f(n) is Θ(g(n))
 (pronounced "Big-Theta")

Bubble Sort Complexity

Time complexity?

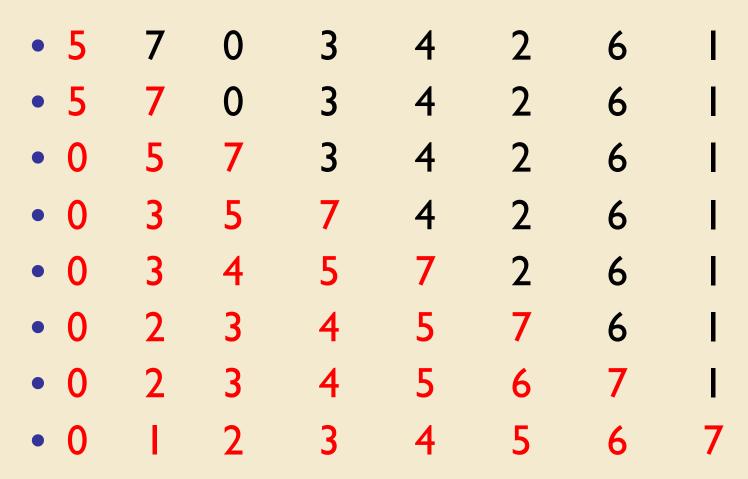
• $\Theta(n^2)$: That is, both $O(n^2)$ and $\Omega(n^2)$

- $O(n^2)$: Never performs more than c n^2 operations
- $\Omega(n^2)$: Sometimes uses at least cn^2 operations
 - Might perform O(n²) compares and O(n²) swaps

Space complexity?

- $\Theta(n)$: That is, both O(n) and $\Omega(n)$
 - Ω(n) : Needs to store an n-element array of constant-sized values
 - O(n) : Only stores the array plus a few other constant-sized variables

Insertion Sort

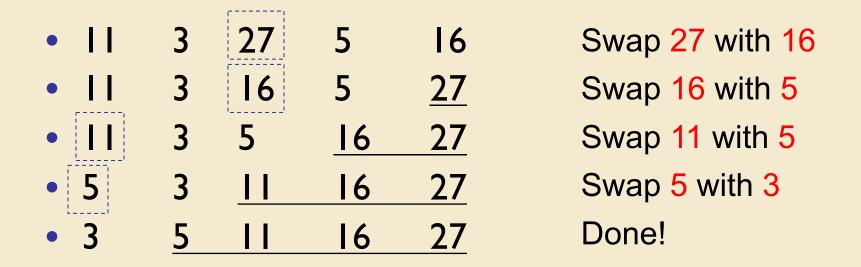


http://www.visualgo.net/sorting

Insertion Sort

- Simple sorting algorithm that works by building a sorted list one entry at a time
- Less efficient on large lists than more advanced algorithms
- Advantages:
 - Simple to implement and efficient on small lists
 - Efficient on data sets which are already mostly sorted
- Time complexity (Worst Case): $\Theta(n^2)$
 - $O(n^2)$: Only perform $O(n^2)$ compares and $O(n^2)$ moves
 - $\Omega(n^2)$: Could perform $\Omega(n^2)$ compares and $\Omega(n^2)$ moves
- Space complexity : θ(n)
- Stable: Yes

Selection Sort



- The algorithm works as follows:
 - Find the maximum value in the list
 - Swap it with the value in the last position
 - Repeat the steps above for remainder of the list (ending at the second to last position)
 - Continue on progressively smaller portions of array

Selection Sort

- Similar to insertion sort
- Noted for its simplicity and performance advantages when compared to complicated algorithms
- Time complexity (Worst Case): $\Theta(n^2)$
 - O(n²) : Only perform O(n²) compares and O(n) moves
 - $\Omega(n^2)$: Could perform $\Omega(n^2)$ compares and $\Omega(n)$ moves
- Space Complexity : $\Theta(n)$
- Stable : Yes

Implementation Details

Selection sort uses two utility methods

```
Uses a swap method
private static void swap(int[]A, int i, int j) {
    int temp = a[i];
    A[i] = A[j];
    A[j] = temp;
}
```

And a max-finding method

}

```
// Find position of largest value in A[0 .. last]
public static int findPosOfMax(int[] A, int last) {
    int maxPos = 0; // A wild guess
    for(int i = 1; i <= last; i++)
        if (A[maxPos] < A[i]) maxPos= i;
    return maxPos;</pre>
```

Iterative & Recursive Selection Sort

```
An Iterative Selection Sort
public static void selectionSort(int[] A) {
    for(int i = A.length - 1; i>0; i--)
        int big= findPosOfMax(A,i);
        swap(A, i, big);
    }
}
```

```
A Recursive Selection Sort (just the helper method)
public static void recSSHelper(int[] A, int last) {
    if(last == 0) return; // base case
```

```
int big= findPosOfMax(A, last);
swap(A,big,last);
recSSHelper(A, last-1);
```

}

Notes

Three simple algorithms: Bubble, Insertion, Selection

- All perform two basic operations: comparisons and swaps/moves
 Comparisons vs Swaps/Moves (worst case)
- Bubble Sort performs (naïve version)
 - quadratic number of comparisons in all cases
 - quadratic number of swaps for almost reverse-sorted data
- Insertion Sort performs
 - linear number of comparisons on almost-sorted data and quadratic number on almost reverse-sorted data
 - quadratic number of moves on almost reverse-sorted data
- Selection Sort
 - quadratic number of comparisons in all cases
 - at most (and sometimes) a linear number of swaps

Coming Up Next

How can we adapt our sorting algorithms to work on non-primitive data types? Can we break the $\Omega(n^2)$ performance bottleneck of our sorting algorithms? Spoiler: Yes!

Stay tuned....