CSSE 220 Day 27

Finish the Sorting Intro
Work on Spellchecker Project

CSSE 220 Day 27

- Mini-project is due at the beginning of Day 30 class (no late days), **so ready for presentation**
- There will be time in class to work with your team every day. Do not miss it!

- Questions?

- Today:
  - Finish the Sorting intro
  - Work on Spellchecker

Knowledge of Elementary Sorts

- What should you know/be able to do by the end of this course?
  - The basic idea of how each sort works
    - insertion, selection, bubble, shell, merge
  - Can write the code in a few minutes
    - insertion, bubble, selection
    - perhaps with a minor error or two
    - not because you memorized it, but because you understand it
  - What are the best case and worst case orderings of N data items? For each of these:
    - Number of comparisons
    - Number of data movements

Elementary Sort summary

- Insertion sort
  - for (i=1; i< N; i++)
    - place a[i] in its correct position relative to a[0] …a[i-1]
      - move "right" each of those items that is less than a[i].
- Selection sort
  - for (i=N-1; i>0; i--)
    - maxPos = location of largest element among a[0] … a[i]
    - a[i]↔a[maxPos]
- Bubble sort
  - for (i=0; i< N-1; i++)
  - for (j=0; j≤ i; j++)
      - if (a[j] > a[j+1]) a[j]↔a[j+1]
- Demonstrations:
  - http://www.cs.ubc.ca/~harrison/Java/sorting-demo.html
  - http://www.geocities.com/siliconvalley/network/1854/Sort1.html

Analyzing Sorts

- Def: An *inversion* is any pair of inputs that are out of order:
  - [5,8,3,9,6] has 4 inversions: (5,3), (8,3), (8,6), (9,6)
  - [5,3,8,9,6] has 3 inversions: (5,3), (8,6), (9,6)
- Swapping a pair of adjacent elements removes exactly one inversion
- Worst case?
  - all $n(n-1)/2$ pairs are out of order, so $n(n-1)/2$ swaps.
- Average case?
  - Consider any array, a, and its reverse, r. Then $inv(a) + inv(r) = n(n-1)/2$
  - So on average, $n(n-1)/4$ inversions.

Demo

- Conclusion: if few inversions (almost sorted), then few swaps
- Yesterday we looked at a quick demo of selection, bubble, and insertion sorts…
  - Completely random data
  - Nearly sorted data

How do we beat $O(n^2)$?

- If swapping a pair of adjacent elements removes exactly one inversion…
- Would swapping elements that are farther apart remove more inversions?

- ShellSort
- MergeSort

Shell sort

- 1959, Donald Shell
- Based on insertion sort
- http://www.cs.princeton.edu/~rs/shell/animate.html
- Faster because it compares elements with a gap of several positions
- For example, if the gap size is 8,
  - Insertion sort elements 0, 8, 16, 24, 32, 40, …
  - Insertion sort elements 1, 9, 17, 25, 33, 41, …
  - …
  - Insertion sort elements 7, 15, 23, 31, 39, 47, …
- Elements that are far out of order are quickly moved closer to where they are supposed to go.

ShellSort example

Original	32	95	16	82	24	66	35	19	75	54	40	43	93	68	
After 5-sort	32	35	16	68	24	40	43	19	75	54	66	95	93	82	6 swaps
After 3-sort	32	19	16	43	24	40	54	35	75	68	66	95	93	82	5 swaps
After 1-sort	16	19	24	32	35	40	43	54	66	68	75	82	93	95	15 swaps

ShellSort Code

```java
public static final int[] GAPS = {1, 4, 10, 23, 57, 132, 301, 701};

public static void shellSort(int[] a) {
  for (int gapIndex = GAPS.length - 1; gapIndex >= 0; gapIndex--) {
    int increment = GAPS[gapIndex];
    if (increment < a.length)
      for (int i = increment; i < a.length; i++) {
        int temp = a[i];
        for (int j = i;
             j >= increment && a[j - increment] > temp;
             j -= increment) {
             a[j] = a[j - increment];
         }
        a[j] = temp;

  }
}
```

TEST CODE:

```java
public static void main(String[] args) {
       int SIZE = 31;
       int [] nums = new int[SIZE];
       for (int i=0; i<SIZE; i++) {
          nums[i] = (SIZE/2 + 5*i) % SIZE;
       }
       printArray("Before sort", nums);
       shellSort(nums);
       printArray("After sort", nums);
```

Shell sort gap sizes

- Start with a large gap
- Do it again with a smaller gap
- Keep decreasing the gap size
- The last time, the gap must be 1 (why?)
- No gap size should be a multiple of another (except all are multiples of 1)
- If proper gaps are chosen, worst-case performance is $O(N (\log N)^2)$
- An example of shellsort analysis (not for the faint of heart):
  - http://www.cs.princeton.edu/~rs/shell/paperF.pdf

Merge Sort

- Divide and conquer
- Sort each half, merge halves together
- How to sort each half?
  - Use Merge sort
- Running time to merge two sorted arrays whose total length is N:
  - O(N)

```java
public static void mergeSort( int [ ] a )
  {
      int [ ] tmpArray = new int[ a.length ];
      mergeSort( a, tmpArray, 0, a.length - 1 );
  }

  /**
   * Internal method that makes recursive calls.
   * @param a an array of Comparable items.
   * @param tmpArray an array to place the merged result.
   * @param left the left-most index of the subarray.
   * @param right the right-most index of the subarray.
   */
  private static void mergeSort( int [ ] a, int [ ] tmpArray,
             int left, int right )
  {
      if( left < right )
      {
          int center = ( left + right ) / 2;
          mergeSort( a, tmpArray, left, center );
          mergeSort( a, tmpArray, center + 1, right );
          merge( a, tmpArray, left, center + 1, right );
      }
  }
```

Mergesort Analysis

```
private static void mergeSort(a, left, right ) {
   if( left < right ) {
            int center = ( left + right ) / 2
            mergeSort( a, left, center )
            mergeSort( a, center + 1, right )
            merge( a, left, center + 1, right )
   }
}
```

- Need to answer:
  - How deep is the recursion?
  - How much work is done in each level of the recursion?

```java
/**
  * Internal method that merges two sorted halves of a subarray.
  * @param a an array of Comparable items.
  * @param tmpArray an array to place the merged result.
  * @param leftPos the left-most index of the subarray.
  * @param rightPos the index of the start of the second half.
  * @param rightEnd the right-most index of the subarray.
  */
 private static void merge( int [ ] a, int [ ] tmpArray,
                            int leftPos, int rightPos, int rightEnd ) {
     int leftEnd = rightPos - 1;
     int tmpPos = leftPos;
     int numElements = rightEnd - leftPos + 1;

     // Main loop
     while( leftPos <= leftEnd && rightPos <= rightEnd )
         if( a[ leftPos ] <=  a[ rightPos ] )
             tmpArray[ tmpPos++ ] = a[ leftPos++ ];
         else
             tmpArray[ tmpPos++ ] = a[ rightPos++ ];

     while( leftPos <= leftEnd )    // Copy rest of first half
         tmpArray[ tmpPos++ ] = a[ leftPos++ ];

     while( rightPos <= rightEnd )  // Copy rest of right half
         tmpArray[ tmpPos++ ] = a[ rightPos++ ];

     // Copy tmpArray back
     for( int i = 0; i < numElements; i++, rightEnd-- )
         a[ rightEnd ] = tmpArray[ rightEnd ];
 }
```

Analysis of merge()

- Merging two sorted arrays of length O(n/2) each is ~n steps

- Why?
  - After each comparison, one element is moved into the sorted array, so there are only n comparisons

- What about merging two sorted arrays of length n/2 each?

Visual analysis

Mergesort Analysis

- For simplicity, assume that N is a power of 2.
- N = Time for merging the sorted halves
- N = (N/2)*2 = time for merging four sorted "quarters" into two sorted "halves"
- N = (N/4)*4 = time for merging four sorted "eighths" into two sorted "quarters"
- …
- N = (2)*N/2 = time for merging N single elements into N/2 sorted pairs
- Total =

Project time

- Proceed according to your IEP.