CS1102 Semester 1 2002/2003
Tutorial 8: Sorting
23-38 September 2002

1. You and Bob are arguing over how to pick a good pivot for Quicksort. For Bob to win the argument, he must come up with a strategy such that you cannot find a *worst-case input*, i.e., an input that causes Quicksort to run in O(N^2) time. Otherwise, you win.

   Describe a worst-case input, if possible, for each of the following pivot picking strategies (a) – (f) proposed by Bob. If you failed to come up with a worst-case input, then you must argue why Bob's pivot picking strategy is flawed.

   (a) pick the first element as pivot
   (b) pick the last element as pivot
   (c) pick the middle element as pivot
   (d) pick the median of the first, middle, last element as pivot
   (e) pick the median of the whole array as pivot
   (f) pick a random element as pivot

2. Which sorting algorithm should you use to solve each of the following problems? Justify your answer.

   (a) Sort an array of 10 random numbers.
   (b) Sort a large array of numbers, 90% of which is already sorted.
   (c) Sort a singly linked list of numbers.

3. **[Merging with Constant Storage]** You are given an array A of size N. Elements A[0] to A[N/2] are sorted. Elements A[N/2+1] to A[N] are sorted as well. Give an algorithm that sorts A *using only O(1) additional storage*. What is the running time of your algorithm?

4. **[Stable Sorting Algorithm]** A sorting algorithm is *stable* if elements with equal keys are left in the same order as they occur in the input. Determine if the following two algorithms are stable. If an algorithm is not stable, modify the code, so that it becomes stable.

   (a)
```
public static void mergeSort(
Comparable []a, Comparable []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 );
   }
}
```

```
private static void merge(
   Comparable []a, Comparable []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 ].compareTo( a[ rightPos ] ) < 0 )
        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 ];
}
```

(b)

```
   public static void insertionSort( Comparable [ ] a )
   {
      for( int p = 1; p < a.length; p++ ) {
        Comparable tmp = a[ p ];
        int j = p;

        for( ; j > 0 && tmp.compareTo( a[ j - 1 ] ) < 0; j-- )
           a[ j ] = a[ j - 1 ];
        a[ j ] = tmp;
      }
    }
```

5. **[Sorting in Linear Time]** If you know more about the items being sorted, you can sort them in linear time. Describe a *linear time*, *stable* sorting algorithm to sort a collection of 8-bit integers.

CS1102 Semester 1 2002/2003
Tutorial 8: Sorting
23-28 September 2002
Answers

1.
(a) A sorted array.
(b) A sorted array.
(c) An array where the middle element is (recursively) maximum. Example: 2 4 6 7 1 5 3
(d) An array where the first/middle/last element contains the maximum and the 2nd largest element. Example: 1 2 4 6 5 3 7
(e) No worst case input. But requires O(N) average time just to pick the pivot. (Note: However, this does not effect the running time of quicksort)
(f) No worst case input. But random number generation is expensive.

2.
(a) insertion sort. Good at sorting small arrays.
(b) (i) insertion sort. Good at sorting almost sorted array. (ii) quicksort. Since the array is almost sorted, if we choose the middle element as the pivot, the chances are we will get a very balanced partition.
(c) (i) insertion sort. Merge/quick sort requires random access to elements.
(ii) mergesort. Only need O(N) time to access the middle element. This does not effect the running time. Merging can be done without using O(N) temporary storage.

(The point of this question is that there is no "best" sorting algorithm. Which one to use always depends on the specific situation.)

3.
Students might come up with modified merge(), which is actually insertion sort. This takes $O(N^2)$ time. The faster way is to just use quickSort(). If students suggest quickSort, then they should indicate how they choose the pivot. (They should not use middle element or median of three, but should use the A[N/4] or A[3N/4] instead.) Other algorithms are acceptable but students should analyze the running time, and ensure that only constant storage is used. Students are not expected to give anything better than O(N log N).

4.
(a) is not stable. replace < with <= in merge() to make the algorithm stable.
(b) is stable

5.
Build an array of 256 queues. For each item *i*, enqueue *i* into the queue A[*i*]. In the end, for each queue A[0] to A[255], dequeue all items. Items will be dequeued in sorted order.