Answers for Test 2, CISC 365 Fall 2010

1. If one NP-complete problem has an exponential lower bound, then all NP-complete problems have an exponential lower bound. This is because all NP-complete problems are reducible to one another in polynomial time. If we suppose that problem B, also NP complete, has a polynomial time algorithm, then we get a contradiction: A can be solved in polynomial time by first reducing to B and then running B's polynomial-time algorithm. This contradicts the proof that A has an exponential lower bound.

2.a= 4, b=2, f(n)=1
lg a / lg b = lg 4 / lg 2 = 2 / 1 = 2
f(n) grows slower than n^2 so this is case 1.
The answer is $\Theta(n^2)$

3. Try $T(n)=n^3$. Does $n^3 =? (n/2)^3+(n/4)^3+1 = n^3/8+n^3/64+1$
The left hand side is too big. We need a slower growing function.

Try T(n)=lg n. Does lg n =? (lg (n/2)) + (lg(n/4)) + 1 = lg n -1 + lg n -2 + 1 = 2 lg n - 2
The right hand side is too big. We need a faster growing function.

Try T(n)=lg n. Does n =? n/2 + n/4 + 1
The left hand side is too big. The solution to the recurrence is between lg n and n.

4. Find the kth largest element in an array, using divide and conquer. It is possible to do this without sorting the whole array; execution time in the average case is linear: $\Theta(n)$.

Solution based on quicksort -style partitioning

Standard quicksort uses a pivot to partition the array into two parts. The elements in the first part are smaller than the partition and the elements in the second part are larger. In our case, we don't need to sort, we only want to find the kth largest element. This can be done using only one recursive call: look at the number of items in each part to figure out which of the two parts contains the kth largest element, and then make the recursive call.

<u>Version 1</u> The first version uses new arrays to store the result of partitioning. This is easy to code, but there is overhead for allocating the arrays. For comparison, below I show code that does that partitioning in place, using only one array.

```
function findKth(array, k, n) {
    int pivot = array[0]; // Use the first array entry as pivot
    new int[] larger;     // New array for elements >= pivot
    new int[] smaller;    // New array for elements < pivot
    int L, S = 0;  // Array indices (for larger and smaller)

    if (n==1) {
        return array[0] }
    // Copy the array elements into “larger” and “smaller”
    for (int i=0; i<n; i++) {
        if (array[i] >= pivot) {
            larger[L]=array[i]; L++;
        } else {
            smaller[S]=array[i]; S++;
        }
    // Test whether the kth largest element is in
    // “larger” or “smaller”.
    if (k<=L) {  // The kth largest element is in “larger”
        return findKth(larger, k, L)
    } else {
        // The kth largest element is in “smaller”.
        // In the recursive call, use k – L as the new k value.
        // For example, we want to find the 10th largest element
        // and there are 7 items  in “larger”, so we look for
        // the 3rd largest element in “smaller”.
        return findKth(smaller, k-L, S);
    }}
```

Version 2 The second version uses one array, and does the partitioning in place. It is tricky to get the details of the partitioning correct. In some versions, the pivot ends up mixed in with the “larger” elements. Other styles of partitioning put the pivot into the middle of the array, separating “smaller” and “larger”. On the test, if you made a decent attempt at writing partitioning code, that was fine – I did not expect perfection. The pivot code below comes from Wikipedia…

Notice that in this code, the findKth routine has more parameters than it did in version 1. Version 1 uses new arrays all the time, so the findKth routine operates on the entire array that is passed to it. In version 2 (which is more efficient), we stick to using only one array, so we need to add the parameters first and last to indicate which part of the array should be worked on.

```
// Find the kth largest element in A[first..last]
function findKth(A, first, last, k) {
    if(first==last) return(A[first]);

    // Partition the array.  Use A[first] as the pivot.  Put
    // items smaller than the pivot into the first part of the
    // array, and put larger items into the second part.
    int pivotValue = A[first];
    swap A[first] and A[last] // Move pivot to end
    int storeIndex = first;
    for (i=first; i<last; i++) {
        if (A[i] ≤ pivotValue)  {
            swap A[i] and A[storeIndex];
            storeIndex = storeIndex + 1;
            }
    swap A[storeIndex] and A[last]; // Move pivot to final place

    // Compute how many large items there are.
    int largeItems = last-storeIndex;
    if (k==largeItems+1)
        return(A[storeIndex]); // the pivot is the answer
    else if (k<largeItems)
        // Make a recursive call to look for the kth smallest
        // item in the large part of the array.
        return(findKth(A,storeIndex+1,last,k));
    else  // Find the item in the first part; make k smaller by
          // subtracting the number of items in the second part
        return(findKth(A,first,storeIndex-1,k-largeItems)
    }
```

The runtime (for version 1 or version 2):
worst case: $T(n)=T(n-1)+n$ which is $\Theta(n^2)$
best case is $T(n)=T(n/2)+n$ which is $\Theta(n)$, value around 2n.
OK, in the best best case, no recursive call is made: if the first partition is the item we want, then the code takes n comparisons.

The average case is pretty close to the best case recurrence $T(n)=T(n/2)+n$: on average we can expect to get partitioning to create "pretty balanced" halves of the array.

Solution based on mergesort

Finding the maximum element can be done efficiently, using divide and conquer in the style of mergesort. Find the maximum in the first half, find the maximum in the second half, and return the larger of the two values. The recurrence for this is $T(n)=2T(n/2)+1$, which has solution $\Theta(n)$. The code for this is

```
int FindMax(A, i, j){ // Find the maximum value in A[i..j]
    int mid;          // The midpoint of the array
    int max, max2;    // Maxima in the first and second halves
    if (i==j) return A[i];
    mid = (i+j)/2;    // Integer division truncates (rounds down)
    max1 = FindMax(A, i, mid);
    max2 = FindMax(A, mid+1, j);
    // Now do the one comparison that appears in the recurrence.
    if (max1>max2) return(max1) else return(max2)
    }
```

However, it is difficult to extend this approach to finding the kth largest element. For example, if we are trying to find the third largest element, it does not help to recursively find the third largest element in the first half and find the third largest element in the second half. It's possible to use MergeSort to sort the array, and then pick out the kth largest element, but that has runtime $\Theta(n \log n)$. To find the kth largest element in linear expected time ($\Theta(n)$), go back to the solutions based on quicksort-style partitioning.