1 2-1

Although merge sort runs in $\Theta(n * lg(n))$ worst-case time and insertion sort runs in $\Theta(n^2)$ worst-case time, the constant factors in insertion sort can make it faster in practice for small problem sizes on many machines. Thus, it makes sense to coarsen the leaves of the recursion by using insertion sort within merge sort when subproblems become sufficiently small. Consider a modification to merge sort in which n/k sublists of length k are sorted using insertion sort and then merged using the standart merging mechanism, where k is a value to be determined.

a. Show that insertion sort can sort the n/k sublists, each of length k, in $/Theta(nk)$ worst-case time.

Insertion sort of array length k takes $\Theta(k^2)$ worst-case time. To sort n/k sublists it takes $n/k * \Theta(k^2) = \Theta(n * k)$

b. Show how to merge the sublists in $\Theta(n * lg(n/k))$ worst-case time.

Recurrence:

$$T(n) = \begin{cases} 0, & \text{if } n = k \\ 2T(n/2) + n, & \text{if } n = k * 2^m, \text{ for } n > k \end{cases}$$

if $n = k$: $T(k) = 0 = lg(1) = k * lg(k/k) = \Theta(n * lg(n/k))$

if $n = k * 2$: $T(k * 2) = 0 + 2 * k = 2 * k + 2 * lg2 = 2 * k * lg(2 * k/k) = \Theta(n * lg(n/k))$

if $n = m * 2$: $T(m * 2) = 2 * m * lg(m/k) + 2m = 2m * (lg(m/k) + 1) = 2m * (lg(m/k) + lg(2)) = 2m * lg(2m/k) = \Theta(n * lg(n/k))$

c. Given that the modified algorithm runs in $\Theta(n*k+n*lg(n/k))$ worst-case time, what is the largest value of k as a function of n for which the modified algorithm has the same running time as standart merge sort, in terms of Θ-notation.

$k = lg(n)$ if $k > lg(n)$, $\Theta(n * k + n * lg(n/k)) > \Theta(n * lg(n))$

d. How should we choose k in practice?

Choose, where insertion sort beat merge sort.

2 2-2

Correctness of bubblesort

Bubblesort is a popular, but inefficient, sorting algorithm. It works by repeatedly swapping adjacent elements that are out of order.

a. Let A' denote the output of BUBBLESORT(A). To prove that BUBBLESORT is correct, we need to prove that if terminates and that $A'[1] \leq A'[2] \leq ... \leq A'[n]$, where n = A.length. In order to show that BUBBLESORT actually sorts, what else do we need to prove?

We need to choose inveriant and proove that it holds.

The next two parts will prove inequality(2.3).

b. State precisely a loop invariant for the for loop in lines 2-4, and prove that this loop invariant holds. Your prove shoult use the structureof the loop invariant proof presented in this chapter.

subarray A[j+1, A.length] does not containt the smallest element.

Initialization: prior the first iteration subarray A[] is empty.

Maintenance: Let us suppuse that A[j+2, A.length] does not contain the smallest element. If $A[j] > A[j+1]$ then swap two elements. So A[j+1, A.length] does not contain max element.

Termination: At termination subarray A[i+1, A.length] doest not contains the biggest element so A[i] conntains the largest element.

c.Using the termination condition of the loop invariant proved int part (b), state a loop invariant for the for loop in lines 1-4 that will allow you to prove inequality(2.3). Your prove should use the structure of the loop invariant proof presented in this section.

Before each loop exection A[0..i-1] in a sorted state.

Initialization: prior the first iteration subarray A[] is empty.

Maintenance: Let us suppuse that A[1, k-1] in a sorted order. Than on the next iteration biggest element floated from the subarray A[k, A.length].

Termination: At termination subarray Array[1, A.length] in a sorted order and unsorted subarray is empty. Invariant holds.

d. What is the worst-case running time of bubblesort? How does it compare to the running time of insertion sort?

$T(n) = \Theta(n^2)$

Compare to insertion sort could run slower because of swap quantity.

3 2-3

Correctness of Horner's rule.

a. In the terms of Θ-notation, what is the running time of this code fragment for Horner's rule? $T(n) = \Theta(n)$

b. Write pseudocode to implement the naive polynomial-evaluation algorithm that computes each term of the polynomial from scratch. What is the running time of this algorithm? How does it compare to Horner's rule.

```
y = 0
for  i= 1  to  n
          x^k = 1;
          for  j = 1  to  i
                    x^k = x^k*x;
          y = y + ak + x^k;
```

$T(n) = \Theta(n^2)$

c. Consider the following loop invariant:

At the start of each iteration of the **for** loop of lines 2-3, $y = \sum_{k=0}^{n-(i+1)} a_{k+i+1}x^k$

Interpret a summation with no terms as equaling 0. Following the structure of the loop invariant proof presented in this chapter, use this loop invariant to

show that, at termination, use this loop invariant to show that, at termination, $y = \sum_{k=0}^{n} a_k x^k$

At initialization $i = n$: $y = \sum_{k=0}^{n-(n+1)} a_{k+n+1} x^k = 0$

At termination $i = -1$: $y = \sum_{k=0}^{n-(-1+1)} a_{k+-1+1} x^k = \sum_{k=0}^{n} a_k x^k$

d. Conclude by arguing that the given code fragment correctly evaluates a polynomial characterized by coefficients $a_0, a_1, ...a_n$.

Invariant holds.

4 2-4

Inversion

Let A[1..n] be an array of n distinct numbers. If $i < j$ and $A[i] > A[j]$, than the pair (i,j) is called inversion of A.

a. Let the five inversions of the array (2,3,8,6,1) : (1,5), (2, 5), (3, 5), (4,5), (3,4)

b. What array with elements from set (1,2,..,n) has the most inversions? How many does it have?

Reverse sorting array, $n/2 * (n-1)$

c. What is the relationship between the running time of insertion sort and the number of inversions in the input array? Justiry your answer.

$\Theta(n+k)$, where k-inversion count.

d. Give an algorithm that determines the number of inversions in any permutatuin on n elements in $\Theta(nlg(n))$ worst-case time. (Hint: Modify merge sort.)

```
def _merge(A,p,r,q):
    L = A[p:r+1];
    R = A[r+1:q+1];
    index = p;
    permutation_count = 0;
    while len(R)>0 and len(L)>0:
        if (L[0]<R[0]):
            A[index] = L.pop(0);
        else:
            A[index] = R.pop(0);
            permutation_count = permutation_count  + len(L);
        index = index+1;
    for i in range(0, len(L),1):
        A[index] = L.pop(0);
        index = index + 1;
    for i in range(0, len(R), 1):
```

```
        A[index] = R.pop(0);
        index = index + 1;
    return permutation_count;

def _sort(array, l, r):
    permutation_count = 0;
    if r >l:
        permutation_count = permutation_count + _sort(array,l, int((l+r)/2));
        permutation_count = permutation_count + _sort(array, int((l+r)/2+1),r);
        permutation_count = permutation_count + _merge(array, l, int((l+r)/2),r)
    return permutation_count;

def sort(array):
    permutation_count = _sort(array,0, len(array)-1);
    return permutation_count;

print(sort([1,2,3,4,5]));
print(sort([2,1]));
print(sort([5,4,3,2,1]));
print(sort([5,4,1,2,3]));
print(sort([1]));
print(sort([]));
```