CHAPTER 02

Q1. ***Insertion sort on small arrays in merge sort***

Although merge sort runs in Θ(n lg n) worst-case time and insertion sort runs in Θ(n2) 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 sub problems become sufficiently small. Consider a modification to merge sort in which n/k sub lists of length k are sorted using insertion sort and then merged using the standard merging mechanism, where k is a value to be determined.

a. Show that insertion sort can sort the n/k sub lists, each of length k, in Θ(nk) worst-case time.

b. Show how to merge the sub lists in Θ(n lg (n/k)) worst-case time.

c. Given that the modified algorithm runs in Θ(nk +n lg()) 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 standard merge sort, in terms of Θ-notation?

d. How should we choose k in practice?

Ans.

1. Sorting a list of size ***n*** , insertion sort takes Θ(n2) time. So to sort n/k list it will take ***Θ( k2) = Θ(nk)***
2. Let us assume the coarseness be k. This mean we will use merge process except starting at a point when the size of list is ***k***. Hence the depth of tree will be reduced to lg n – lg k = lg ( ). At each level merging time is still ***cn*** .

Hence ***cn(lg ( ))*** leads to worst case of Θ(n lg ( )).

1. As per problem Θ(nk +n lg()) = Θ(nk +n lgn – n lg k ) should be equal to Θ(n lg n). To satisfy this nk cannot grow faster than ***n lg n*** otherwise it will run worse than Θ(n lg n). So nk < n lg n ⇒ k < Θ(lg n) or k ∈ Θ(lg n)

Q2.  ***Correctness of bubblesort***

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

for i=1 to A.length-1

for j= A.length down to i+1

if A[j] < A[j-1]

exchange A[j] with A[j-1]

1. Let A’ denote the output of BUBBLESORT(A). To prove that BUBBLESORT is correct, we need to prove that it terminates and that A[1]≤ A[2]≤ A[3]…… A[n] where n = A.length. In order to show that BUBBLESORT actually sorts, what else do we need to prove?
2. State precisely a loop invariant for the for loop in lines 2–4, and prove that this loop invariant holds. Your proof should use the structure of the loop invariant proof presented in this chapter.
3. Using the termination condition of the loop invariant proved in part (b), state a loop invariant for the for loop in lines 1–4 that will allow you to prove in-equality (2.3). Your proof should use the structure of the loop invariant proof presented in this chapter.
4. What is the worst-case running time of bubblesort? How does it compare to the running time of insertion sort?

Ans.

1. We also need to prove that elements in the array are same and then array has same no of elements. This can be easily proved as only modification algorithm does is ***swap*** which surely don’t change the array’s element. So the resulting array is just a permutation of original array but with the above constrain.
2. ***Loop invariant*** :- At the start of each iteration, the position of the smallest element of A[i….n] is at most j.

* Initialization:- This is clearly true prior to the first iteration because the position of any element is at most A.length.
* Maintenance:- Suppose that the position of smallest element is at most k and j = k . Then we compare A[k] to A[k − 1]. If A[k] < A[k − 1] then A[k − 1] is not the smallest element of A[i..n], so when we swap A[k] and A[k − 1] we know that the smallest element of A[i..n] must occur in the first k − 1 positions of the subarray, the maintaining the invariant. On the other hand, if A[k] ≥ A[k − 1] then the smallest element can’t be A[k]. Since we do nothing, we conclude that the smallest element has position at most k − 1. Upon termination, the smallest element of A[i..n] is in position i.
* Termination :- At termination the j will be having an exchange in i and i+1 hence smallest element of A[i..n] is in position i which is within the bound of j.

1. ***Loop invariant***:- At the start of each iteration A[1..i-1] contains the i-1 elements but in sorted order.

* Initialization:- At start i= 1. And hence the first 1-0 element are trivially sorted.
* Maintenance:- Let us assume that A[1..i-1] contains sorted elements. Now in above partwe shows that at the end of line 4, ***i*** will be having smallest element. Since i-1 elements are already sorted , A[i[ must be ith smallest element. Therefore A[1..i] contains sorted elements.
* Termination:- Upon termination i = A.length -1 . This means A[1….n-1] elements are already sorted and hence the nth remaining element must be nth smallest. Hence A[1…n] contains sorted elements upon termination.

1. The ith iteration in outer loop will cause n-I iteration to inner loop. Hence in worst case this will lead to running time of Θ(n2). This is same that of insertion sort but bubble sort’s best case is also Θ(n2) whereas insertion sort has best case of Θ(n).

Q4. The following code fragment implements Horner’s rule for evaluating a polynomial.

Given the coefficient a0,a1….an and a value for x:

y = 0

for i = n down to 0

y = ai + x\*y

1. In terms of Θ-notation, what is the running time of this code fragment for Horner’rule?
2. Write pseudocode to implement the naïve 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?
3. Consider the following loop invariant:

At the start of each iteration for loop of line 2-3

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

1. Conclude by arguing that the given code fragment correctly evaluates a polynomial characterized by the coefficient a0,a1…an.

Ans.

1. If we assume that the trivial arithmetic calls run in constant time. Then the turn of iterations are n hence it’s Θ(n).