Chapter 2 Problems

2-1 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(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 standard 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 Θ(nk) worst-case time.

Solution:

Insertion sort sorts the sequence of k elements in   
T(k) = Θ(k^2)= c\_1 \* k^2 + c\_2 \* k + c\_3 s.t. c\_1 ≠ 0 and c\_1,c\_2,c\_3 are constants. Since we have n / k sequences, running time will be

n/k \* T(k)=n/k \* Θ(k^2)=n/k \* (c\_1 \* k^2 + c\_2 \* k + c\_3) = c\_1 \* nk + c\_2 \* n + c\_3 \* n/k

Therefore, running time is Θ(nk)

b. Show how to merge the sublists in Θ(nlg(n/k)) worst-case time.

Solution:

Merge-sort has lg(n) levels, but since there are n/k sublists, we will have lg(n/k) levels. We stop dividing at the level, where the number of elements is k.

Hence, we would subtract the number of canceled levels: lg(k), from the total number of levels: lg(n).   
lg(n) - lg(k) = lg(n/k)

We would use merge time Θ(n) at each level. Therefore, running time will be

lg(n/k) \* Θ(n) = Θ(nlg(n/k))

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

Solution:

We want to know the largest value of that makes Θ(nk + nlg(n/k)) = Θ(nlg(n))

In order to make Θ(nlg(n)) the most dominant, it must be bigger than nk.

Therefore, Θ(nk)= Θ(nlg(n)) ⇒ k = Θ(lg(n))

If k becomes bigger than lgn

Θ(nk + nlg(n) - nlgk)= Θ(nk) and would dominate nlgn.

d. How should we choose k in practice?

Solution:

We can use machines to run the two algorithms to figure out the value of k that has to be the largest input size, for which insertion sort runs faster than merge sort.

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 it terminates and that

A' [1] ≤ A' [2] ⋯ A' [n],

where n=A.length. In order to show that BUBBLESORT actually sorts, what else do we need to prove?

Solution:

We also need to prove that A' consists of the elements of A but in sorted order.

b. 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.

Solution:

Initialization: Initially the subarray contains only the last element A[n] and this is the smallest element of the subarray. Elements are the same because nothing happened yet, therefore the invariant holds.

Maintenance: In every step we compare A[j] with A[j - 1] and make A[j - 1] the smallest among them. So, after the iteration, the length of the subarray increases by one and the first element is the smallest of the subarray. The fourth line obviously holds because elements are neither added or deleted, only swapping. At the k^th iteration, where 2 ≤ k ≤ (n - i), assume that there exists a value m that comes after j and A[m] is the smallest element in [j ...n]. It means that when we compare m^th element with (m - 1)^th element, this iteration is completed since m > j. Since, A[m] is the smallest element, it means that swapping is not executed, which means the condition A[m] < A[m - 1] is false. However, if A[m] ≥ A[m - 1] is true, it will contradict that A[m] is the smallest element. It means that our assumption is wrong and invariant holds.

Termination: The loop terminates when j = i + 1. At that point also the length of the subarray increases by one and the first element is the smallest of the subarray as we swap A[i + 1] with A[i].

c. 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 inequality (2.3). Your proof should use the structure of the loop invariant proof presented in this chapter.

Solution:

Initialization: Initially the subarray A[1… i - 1] is empty and trivially this is the smallest element of the subarray.

Maintenance: The inner loop, which is executed at each iteration, would move the smallest element from A[i…n] to A[i] making the subarray A[1…n] sorted.

Termination: The loop terminates when i = A.length. At that point, the array A[1…n] will consist of all elements in sorted order.

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

Solution:

In the worst-case (reverse sorted array), bubblesort will iterate over the whole array for each element and will perform n comparisons and swaps. Therefore, worst-case running time of bubblesort is Θ(n^2).

Both Insertion and Bubble sorts have the same asymptotic time complexity Θ(n^2). However, for the best case, insertion sort would not enter the inner loop if the array is sorted and will perform with time complexity Θ(n). To the contrary, Bubble sort would always enter the inner loop and perform with time complexity Θ(n^2) at best case.

2-3 Correctness of Horner’s rule

a. In terms of Θ-notation, what is the running time of this code fragment for Horner’s rule?

Solution:

The loop fragment would pass through all the elements and it will take Θ(n) time complexity.

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?

Solution:

Naive-Horner (A, x)

1. y = 0

2. for i = 0 to n

3. k = 1

4. for j = 1 to i

5. k = k \* x

6. y = y + A[i] \* m

The above algorithm runs with a for loop of (n - 1) elements (lines 4 - 5) inside another for loop (lines 2 - 6) of n elements. Therefore, the algorithm runs at Θ(n^2) time. This algorithm is obviously worse than Horner’s rule which runs at linear time.

c. Consider the following loop invariant:

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.

Solution:

Initialization: At the start of the first iteration, there are no terms in the summation, so the sum is zero.

Maintenance: The loop would multiply the previous result by x. Then, it adds the element A[i]. Iteration starts from n, the k^th added factor, A[n - k - 1], would be multiplied by x (n – k – 1) times before the loop stops. Therefore, stopping after the m^th iteration would produce   
a\_n \* x^(m - 1) + a\_(n-1) \* x^(m - 2) + .. + a\_(n + 1 - m) which is equivalent to our invariant.

Termination: The loop terminates after i = -1; substituting this value within our invariant yields.

y - 1 = (a\_k \* x^k)

d. Conclude by arguing that the given code fragment correctly evaluates a polynomial characterized by the coefficients a\_0,a\_1,…,a\_n.

Solution:

Upon termination of our code according to the invariant, the y would be the correctly computed polynomial.

y= (a\_k \* x^k)

2-4 Inversions

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

a. List the five inversions of the array ⟨2, 3, 8, 6, 1⟩

Solution:

Inversions in the given array are: (2, 1), (3, 1), (8, 6), (8, 1), and (6, 1).

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

Solution:

The array with elements from the set 1, 2, …, n with the most inversions will have the elements in reverse sorted order ⟨n, n − 1,…, 2 , 1⟩. As the array has n unique elements in reverse sorted order, for every unique pair of (i, j), there will be an inversion. We would have (n - 1) for the 1st element, the largest, (n – 2) for the 2nd until having 1 inversion for the (n-1)^th element.

Therefore, total number of inversions is equal to (n - 1) + (n - 2) + ⋯ + 1 = (n(n - 1)) / 2.

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

Solution:

If we take a look at the pseudocode for insertion sort, we will see that the more the number of inversions in an array, the more times the inner while loop will run. Maximum number of inversions are possible when the array is reverse sorted. So, the higher the number of inversions in an array, the longer insertion sort will take to sort the array.

d. Give an algorithm that determines the number of inversions in any permutation on n elements in Θ(nlgn) worst-case time. (Hint: Modify merge sort.)

We need to recursively divide the array into halves and count the number of inversions in the sub-arrays. This will result in lgn steps and Θ(n) operations in each step to count the inversions. All in all Θ(nlgn) algorithm.