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 ()) 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.**

Let us first state the new modification.

To do merge sort, we check the problem size and if it is below some **LEAF-SIZE** we can perform insertion sort instead of further dividing it to perform of merge sort. Because insertion sort works better for smaller sizes.

When size will reach k we should have n/k problems.

If you still didn’t get why n/k? Go back to check merge sort. Let us go to merge sort. In given array. Let us assume that if size reduces to 4(k). We will stop dividing and use insertion sort. Now when we divide 8 we get 2 sub problems. Now we reached size 4.

We can see that no of sub-problems are 8/4 = 2 (n/k)

Ans.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  | 49 |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | 49 |

**Use Insertion Sort Use Insertion Sort**

a. Insertion Sort sorts a sequence of k items in T(k) = Θ(k2). Now we have n/k sequences, the overall running time will be

T(k) = Θ(k2) = = Θ(nk).

Note : Don’t think that **k** constant and we should discard it. K is not a constant.

b. Remember that merge process takes Θ(n) to merge two sub-sequences producing n sorted elements.

Let us assume that at rth level we have k elements. So time we need to merge elements will be

T(k) = Θ(k) = Θ(k) [ we have n/k lists]

At r-1th level

T() = Θ(k) = Θ(n)

Since we used in our analysis that we have lg() levels. In original merge sort we have **lg n** levels. But in modified algorithm we stop at k. If we won’t have discarded at k it would make a tree with lg n leaves. Hence these k could have made an addition of m(no. of levels)

⇒ m = lg k

Hence new height of the tree is **lg n – lg k = lg(**

Since we saw above that each level takes Θ(n) time.

lg( will take

**lg( \* Θ(n) = Θ(n lg ()**

c. Let us first see how we get Θ(nk + n lg ())

**Θ(nk) + Θ(n lg()) = Θ(nk + n lg ())**

Insertion sort time Recursive merging

As per problem we want to know largest k such that

Θ(nk + n lg ()) = Θ(n lg n )

Θ(nk + n lg n – n lg k ) = Θ(n lg n)

In order to make the equation true.

n lg n on left side must dominate nk.

Hence Θ(nk) = Θ(n lg n )

Hence **k ∈ Θ(lg n)**

d. To choose best k , we must run the two algorithm on required machine. And should note the output. Then we compare the noted data and try to choose the best k.

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]

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

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.

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

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

**Ans.**

a. 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.

b**. 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.

c. **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 part we 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.

d. Assume the worst case time is denoted by T(n) and S(n) denotes the worst running due to exchange and comparing in line 3 and 4 which takes Θ(1).

T(n) =

Outer loop Inner loop

T(n) =

T(n) = Θ(1)

T(n) = Θ(1)

T(n) = Θ(1)( )

T(n) = Θ1 ( n(n-1) – )

T(n) = Θ1 ( )

T(n) = Θ(n2)

Therefore, as we can see that bubble sort have the same asymptotic time of Θ(n2). But if we consider best case, insertion sort has best case when array is sorted has it has best running time of Θ(n) but in case of bubble sort there is not any termination till the loop is completed. Hence it has best case of Θ(n2).

We could add a condition to enhance it to detect best case where there are no swaps. But as a general requirement we don’t rely on best case. Hence it’s not worth much effort.

Q3. **Correctness of Horner’s rule**

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

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

b. 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?

c. 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

d. Conclude by arguing that the given code fragment correctly evaluates a polynomial characterized by the cn.

**Ans.**

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

b.

y = 0

**for** i=0 to n do

k = 1

**for** j=1 to i do

k = k\*x

y = y + ai\*k

**return** y

Analysis :

T(n) =

Outer loop Inner loop

T(n) =

T(n) = Θ(1)

T(n) = Θ(1)

T(n) = Θ(1)( )

T(n) = Θ(n2)

This code has runtime Θ(n2). This is much slower than Horner’s rule.

However we write another native yet efficient approach.

y = 0

k = 1

**for** i=0 to n do

y = y + a­i k

k = k\*x

**return** y

c.

Initialization : at the start of first iteration. i = n: therefore, the R.H.S upper limit evaluates to

n-i+1 = -1 which leads to 0 summation which true as y = 0 .

Maintenance:

**Let us try to prove this through induction. We denote yx to be the value of y when i = x. In this case let us assume that above variant holds for some i=j .**

Now as inductive step we need to prove it for j-1.

Now after next iteration

yj-1 = aj + x \* yj

Substituting yi­ , we get

yj-1 = aj + x \* -----(1)

yj-1 = aj +

yj-1 = aj + [ Substituting u = k +1]

At u = 0 we can notice that summation will yield aj . Hence we can combine two terms and make u start from 0 in summation.

=

Above summation can be written as by substituting k = u , we get

=

Note the similarity between above and 1st eqn.

Hence our invariant holds at i.

Termination : In termination step i = - 1, substituting it to the eqn we get our desired result.

d. From the termination step of above part we can get that the above algorithm at termination evaluates the polynomial.

Q4. **Inversion**

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 five inversions of array {2,3,8,6,1}

b. What array with elements from the set {1,2,3…n} has the most no. of inversion? How many does it have?

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

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

**Ans.**

a. The five inversion are (2,1) , (3,1) , (8,1) , (8,6) , (6,1) . Since in all these cases i < j and A[i] > A[j].

b. Let us develop intuition before answering. If the first element of the array will be the largest. It need to be swap through all remaining elements.(n-1). If the second element is largest element it need to be swap through the remaining elements on it right side. (n-2). As may have guessed now that if we continue above argument. In order to generate most no. of inversion. 1st element must be largest , 2nd must be 2nd biggest and so on. Hence a reverse sorted array(n,n-1,n-2….1) has most no. of inversion. And the no. of inversions are

Sn =  n-1 + n-2 + n-3 +..+ 1

=

= -

= n(n-1) -

=

c. We know that the inner while loop of insertion sort shift the elements to left to their right position. So if there is more inversion in an array, then we need to shift more elements. Hence as the number of inversions increases, running time of insertion sort increases. Hence if g(n) denotes the no. of inversion then

T(n) = Θ(g(n)) , where n is the size of array.

Justification:

The running time of insertion sort is a constant times the number of inversions. Let I(i) denote the number of j < i such that A[j] > A[i]. Then ∑ni = 1 I(i) equals the number of inversions in A. Now consider the while loop on lines 5-7 of the insertion sort algorithm. The loop will execute once for each element of A which has index less than j is larger than A[j]. Thus, it will execute I(j) times. We reach this while loop once for each iteration of the for loop, so the number of constant time steps of insertion sort is ∑ni = 1 I(j) which is exactly the inversion number of A.

d. Explanation:

We will use our regular merge sort but with minor modification to meet our requirements. We will call our main procedure COUNT-INVERSIONS. Since during counting original array will be destroyed. We can create a copy of array before starting our main process. (I have not mentioned it in code because it depends on you. If you want to preserve it or not).

To count the total no. of inversion we need to count the inversion from right , left and then add them up to give total. This give the reason why we use Divide and Conquer.

Now an inversion occurs when we copy something from right sub-array while there are elements in left subarray. And the no. of inversions will be no. of elements left in left sub-array because only those elements will be greater than that right sub-array element.

Since there is no extra task other than trivial arithmetic its runtime is same that of merge sort.

T(n) = Θ(n) + Θ(n lg n ) = Θ(n lg n)

Pseudocode:-

COUNT-INVERSIONS(A, p, r)

if p < r

q = floor((p + r) / 2)

left = COUNT-INVERSIONS(A, p, q)

right = COUNT-INVERSIONS(A, q + 1, r)

inversions = MERGE-INVERSIONS(A, p, q, r) + left + right

return inversions

MERGE-INVERSIONS(A, p, q, r)

n1 = q - p + 1

n2 = r - q

let L[1..n1 + 1] and R[1..n2 + 1] be new arrays

for i = 1 to n1

L[i] = A[p + i - 1]

for j = 1 to n2

R[j] = A[q + j]

L[n1 + 1] = ∞

R[n2 + 1] = ∞

i = 1

j = 1

inversions = 0

for k = p to r

if L[i] <= R[j]

A[k] = L[i]

i = i + 1

else

inversions = inversions + n1 - i + 1 //Important step to be noted.

// When the right is smaller, there are elements on the left this indicates inversion.

// each remained element in the left makes an inversion pair.

A[k] = R[j]

j = j + 1

return inversions

Since we need to count the inversion where i<j & A[i]>A[j]. We can easily count this during the merge process. Since in merge when we compare left and right sub-array if at any step we find that element in the right subarray is smaller than the element in the left. We perform swap **or inversion** are there. Now to calculate **How**  much inversion.

E.g.

Imagine this scenario

L = low

M = mid

H = high

i j

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 4/**l** | 9 | 15/**m** | 5 | 7 | 16/**h** |  |  |

Let us assume that A[l….m] is left array and A[m+1…...h] is right array. Initially **i** and **j** are at 4 and 5 respectively. Since j>I and A[j] > A[i] there won’t be any inversion. So we insert 4 in original array and increase **i.** Now j is at 5 and i at 9. Since j > i **but** A[i] > A[j], there will be an inversion. To calculate much many inversion. We can calculate that 5 will be at the position of 9 and hence it will swap with 15 and then 9 to reach its position.

No. of swap of will be no. of elements he need to cross in left array for swap. Since we have already moved i position total no. of swaps will be, swap = (size1 – i +1 ). Extra one because j will be at i's positon.

If we want we can also print the number which will be swapped. Swapped number will be (15,5) and (9,5).

To visualize this add following lines in the else part(where inversion count is calculates)

for l= i to size1

print (left[l] and right[j]))

l = l +1;