CHAPTER 02

Ex 2.1

Q1. Using Figure 2.2 as a model, illustrate the operation of INSERTION-SORT on the array A = {31,41,59,26,41,58}.

Ans.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Iteration(j) | Iteration(i) | Item1 | Item2 | Item3 | Item4 | Item5 | Item6 | Key | Comments |
|  |  | 31 | 41 | 59 | 26 | 41 | 58 | N/A | Initial condition |
| **2** | **1** | **31** | **41** | 59 | 26 | 41 | 58 | **41** | Loop passed as 41>31 |
| **3** | **2** | 31 | **41** | **59** | 26 | 41 | 58 | **59** | Loop passed as 59>41 |
| **4** | **3** | 31 | 41 | **59** | **26** | 41 | 58 | **26** | Loop Enters |
| **4** | **3** | 31 | 41 | **59** | **59** | 41 | 58 | **26** | A[i+1] = A[i] |
| **4** | **2** | 31 | **41** | **41** | 59 | 41 | 58 | **26** | A[i+1] = A[i] |
| **4** | **1** | **31** | **31** | 41 | 59 | 41 | 58 | **26** | A[i+1] = A[i] |
| **4** | **0** | **31** | **31** | 41 | 59 | 41 | 58 | **26** | Loop Terminates |
| **4** | **0** | **26** | 31 | 41 | 59 | 41 | 58 | **26** | A[i+1] = key |
| **5** | **4** | 26 | 31 | 41 | **59** | **41** | 58 | **41** | Loop Enters |
| **5** | **4** | 26 | 31 | 41 | **59** | **59** | 58 | **41** | A[i+1] = A[i] |
| **5** | **3** | 26 | 31 | 41 | **59** | **59** | 58 | **41** | Loop Terminates |
| **5** | **3** | 26 | 31 | 41 | **41** | 59 | 58 | **41** | A[i+1] = key |
| **6** | **5** | 26 | 31 | 41 | 41 | **59** | **58** | **58** | Loop Enters |
| **6** | **5** | 26 | 31 | 41 | 41 | **59** | **59** | **58** | A[i+1] = A[i] |
| **6** | **4** | 26 | 31 | 41 | 41 | **59** | **59** | **58** | Loop Terminates |
| **6** | **4** | 26 | 31 | 41 | 41 | **58** | 59 | **58** | A[i+1] = key |
|  |  | 26 | 31 | 41 | 41 | 58 | 59 | N/A | Final Sorted array. |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31/**i** | 41/**j** | 59 | 26 | 41 | 58 |

↓(No swap as A[j] > A[i])

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 | 41/**i** | 59/**j** | 26 | 41 | 58 |

↓( No swap as A[j] > A[i])

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 | 41 | 59/**i** | 26/**j** | 41 | 58 |

↓(3 forward insertion )

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 26 | 31 | 41 | 59/**i** | 41/**j** | 58 |

↓(1 swaps)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 26 | 31 | 41 | 41 | 59/**i** | 58/**j** |

↓(1 swaps)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 26 | 31 | 41 | 41 | 58 | 59 |

Q2. Rewrite the INSERTION-SORT procedure to sort into non-increasing instead of non- decreasing order.

Ans. Pseudo Code:-

INSERTION-SORT(A)

for j = 2 to A.length

key = A[j]

i = j - 1

while i > 0 and A[i] < key

A[i + 1] = A[i]

i = i - 1

A[i + 1] = key

Q3. Consider the searching problem:

Input: A sequence of n numbers **A = <a1 ,a2,a3,a4 ……an>** and a value v.

Output: An index i such that v = A[i] or the special value **NIL** if v does not appear in A.

Write pseudocode for **linear search**, which scans through the sequence, looking for v. Using a loop invariant, prove that your algorithm is correct. Make sure that your loop invariant fulfills the three necessary properties.

Ans.

Pseudocode:

LINEAR-SEARCH(A, v)

for i = 1 to A.length

if A[i] == v

return i

return NIL

**Loop invariant** :- ∀ j <i, A[j] ≠ v.

Speaking informally – All the items that proceeds the item positioned at i , is not equal to v.

**Initialization:-**  Before the start of loop , the subarray is empty. Hence the invariant holds.

**Maintenance:-** During each loop iteration we compare **v** with **A[i]** . If they are same we return it which is true result. Hence it’s the ith elements that matches but the array A[1..i-1] still has item which doesn’t equal with v. If we assume that algorithm has completed it means line 3 has not been executed. Then A[i] ≠ v is true and our invariant still holds.

A more formal approach will be by contradiction. Let us assume ∀ k <i , A[k] = v. Since we are in ith iteration and k is less than i. Line 3 must not have executed because it leads to termination. Since 3rd line is not executed means Line 2 must have been a false statement i.e. A[k] ≠ v. But this contradicts our first assumption. Hence it proves our invariant.

**Termination:-**  Our algorithm can terminate in two ways.

a. The loop terminates when i = A.length+1 . At that time i would be n+1. Substituting it in A[1…i-1], we find that A[1…n] contains elements which are different from v. Thus we return **nil** as per requirement. Hence our algorithm is correct.

b. If line 3 executes successfully. In that case A[i] must have been equal to v. Hence we return i, which is correct answer. Hence our algorithm is correct.

Q4. Consider the problem of adding two n-bit binary integers, stored in two n-element arrays A and B. The sum of the two integers should be stored in binary form in an (n+1)-element array C. State the problem formally and write pseudocode for adding the two integers.

Ans. Let us first state the problem formally.

**Input** : Two n-bit arrays, A= {a1,a2,…,an} and B = {b1,b2,…,bn} and 1 (n+1) bit array C = {c1,c2,…,cn}

where ai , bi , ci are bits i.e. ai , bi , ci ∈ {0,1}

**Output** : Modified C array, ci ∈ {0,1} and Cholds the binary addition of A and B.

Pseudocodes:-

1.

**Input:-**  Two arrays A and B containing binary digits of two number **a**  and **b**

**Output :-** an (n+1) element array C containing a+b in binary form.

ADD-BINARY(A, B)

C = new integer[A.length + 1]

carry = 0

for i = 1 to A.length

C[i] = (A[i] + B[i] + carry) % 2 // remainder

carry = (A[i] + B[i] + carry) / 2 // quotient

C[i + 1] = carry

return C

2.

**Input:-**  Two arrays A and B containing binary digits of two number **a**  and **b**

**Output :-** an (n+1) element array C containing a+b in binary form.

ADD-BINARY(A, B)

C = new integer[A.length + 1]

carry = 0

for i = 1 to A.length

carry = (A[i] **XOR** B[i] **XOR** carry)

C[i] = (A[i] **AND** B[i]) **XOR** (A[i] **AND** carry) **XOR** (B[i] **AND** carry)

C[i] = carry

return C

Note : AND, XOR are basic bitwise operator.

Ex 2.2

Q1. Express the function **n3/1000-100n2-100n+3** in terms of Θ-notation.

Ans. From our understanding we know that while calculating Θ-notations. We mostly worry about the terms can dominates the expression whenever input size grow to infinity. We care this way because other smaller terms becomes insignificant when compared with the larger term.

f(n) = ʘ(g(n)) ⇒ ∃ c1 , c2 , n0 : ∀n ≥ n0 **0 ≤ c1g(n) ≤ f(n) ≤ c2g(n)**

To take it further

Let f(n) = 2n +2n .

Now consider what is going to happen when n → 100.

2100  will be **1,267,650,600,228,229,401,496,703,205,376**.

While the other being **200**.

I think now it’s clear why we care about larger terms.

Coming to our original problem.

If we **ignore** the **low order terms** and **constant factors**. It will reduce to **Θ(n3)**.

Q2. Consider sorting n numbers stored in array A by first finding the smallest element of A and exchanging it with the element in A[1]. Then find the second smallest element of A, and exchange it with A[2]. Continue in this manner for the first n-1 elements of A. Write pseudocode for this algorithm, which is known as selection sort. What loop invariant does this algorithm maintain? Why does it need to run for only the first n-1 elements, rather than for all n elements? Give the best-case and worst-case running times of selection sort in Θ-notation.

Ans. **Pseudocode:-**

n = A.length

for i = 1 to n - 1

smallest\_pos = i

for j = i + 1 to n

if A[j] < A[smallest\_pos]

smallest\_pos = j

swap(A[i], A[smallest\_pos])

**Loop invariant** :- **∀ k < i , ∀ j ≥i , A[1] ≤ A[2] ≤ …. ≤ A[j] ≤ A[j]**

Our invariant says that all the item which proceeds the ith are in order(sorted) and they are smaller than any element whose index is not less than i.

**Initialization:-**  Before the start of iteration of loop , the subarray is hence, so the variant holds.

**Maintenance:-** During each loop iteration we compare mark smallest element’s position as i’s positon. Then we iterate over remaining array A[i+1…n]. If we find any element smaller than marked smallest we modify the marker. In the end of inner loop we swap. If marked element was already smallest we swap same element otherwise we insert A[j] in it proper position by swap in A[1…i-1].

**Termination:-**  The loop terminates when i > A.length-1. At that time A[1…n-1] will have smallest i-1 elements but in sorted order. Hence remaining nth will be largest among all. Hence we terminate the loop with assurance that A[1…n] is now sorted.

As mentioned above after loop termination we have n-1 smallest element in the A[1..n-1]. It itself implies that **nth**is **largest** among all. Hence we run only it n-1 times.

**Analysis** **:**

We must note that in any case we must swap n-1 times (7 lines) .

Hence total no comparisons is :-

(n-1)+ (n-2) +….+ 1 = ∑1n-1

By using AP

∑1n-1= (n-1)+1/2 \* (n-1) = ½n(n-1) = ½(n2-n) [(n-1) because last element is already sorted.]

Ignoring constant we get complexity is

**Θ(n2)**

Q3. Consider linear search again (see Exercise 2.1-3). How many elements of the input sequence need to be checked on the average, assuming that the element being searched for is equally likely to be any element in the array? How about in the worst case? What are the average-case and worst-case running times of linear search in Θ-notation? Justify your answers.

Ans.

**No. of checks** :

**Average case**

Let us assume that all the elements are equally likely. Hence on average we need to check n/2.

This is because any element has the chance to be the target.

Thus for 1 element, no of steps will be 1

For 2nd element it would be 2.

Since we have n case result would be 1 + 2+ …+ n divided by n.

⇒ ⇒

**Worst case**

In Worst case, element will not be in array. Hence it’s easy to count to the no. of check which is **n**.

**Running time**

Since we can see that time depend on no. of checks. But since we are looking for Θ-notation, which discards all low order terms and constant.

Discarding 2 in from average case check we get,

Running time in both average and worst case is Θ(n).

Q4. How can we modify almost any algorithm to have a good best-case running time?

Ans. There are mainly two cases

* When we don’t care about correctness of the algorithm. In that case we can return some valid solution for some cases.

E.g. In case of sorting we simply return the inputted array and do nothing. Since we did nothing it’s Θ(1) solution. We will rely on the fact that the input is already sorted.

* When we care about correctness. In that case we can pre-compute some valid solutions for some inputs. And whenever we receive any input firstly we can check that, inputted input matches with the pre-computed solution. If that’s the case we can simply return our **hard-coded** , pre-computed solution.

E.g.

1. In case of exponent calculation. We can pre-compute 2100 and for every input if input matches with 2100 we can return the cached solution. Otherwise we can run the normal algorithm for solution.
2. Another analogy can be sorting. For every array we can first check if it’s already sorted. If it is simply return it. Checking it would take Θ(n). Otherwise run the normal sort procedure.

Ex 2.3

Q1. Using Figure 2.4 as a model, illustrate the operation of merge sort on the array A{3,41,52,26,38,57,9,49}.

Ans.

DIVIDE STEP

Ans.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 3 | 41 | 52 | 26 | 38 | 57 | 9 | 49 |

|  |  |  |  |
| --- | --- | --- | --- |
| 3 | 41 | 52 | 26 |

|  |  |  |  |
| --- | --- | --- | --- |
| 38 | 57 | 9 | 49 |

|  |  |
| --- | --- |
| 9 | 49 |

|  |  |
| --- | --- |
| 38 | 57 |

|  |  |
| --- | --- |
| 3 | 41 |

|  |  |
| --- | --- |
| 52 | 26 |

|  |
| --- |
| 49 |

|  |
| --- |
| 9 |

|  |
| --- |
| 57 |

|  |
| --- |
| 38 |

|  |
| --- |
| 26 |

|  |
| --- |
| 52 |

|  |
| --- |
| 3 |

|  |
| --- |
| 41 |

MERGE STEP

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 3 | 9 | 26 | 38 | 41 | 49 | 52 | 57 |

|  |  |  |  |
| --- | --- | --- | --- |
| 3 | 26 | 41 | 52 |

|  |  |  |  |
| --- | --- | --- | --- |
| 9 | 38 | 49 | 57 |

|  |  |
| --- | --- |
| 9 | 49 |

|  |  |
| --- | --- |
| 38 | 57 |

|  |  |
| --- | --- |
| 3 | 41 |

|  |  |
| --- | --- |
| 26 | 52 |

|  |
| --- |
| 49 |

|  |
| --- |
| 9 |

|  |
| --- |
| 57 |

|  |
| --- |
| 38 |

|  |
| --- |
| 26 |

|  |
| --- |
| 52 |

|  |
| --- |
| 3 |

|  |
| --- |
| 41 |

Q2. Rewrite the MERGE procedure so that it does not use sentinels, instead stopping once either array L or R has had all its elements copied back to A and then copying the remainder of the other array back into A.

Ans.

Pseudocode:-

MERGE(A, p, q, r)

n1 = q - p + 1

n2 = r - q

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

for i = 1 to n1

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

for j = 1 to n2

R[j] = A[q + j]

i = 1

j = 1

for k = p to r

if i > n1

A[k] = R[j]

j = j + 1

else if j > n2

A[k] = L[i]

i = i + 1

else if L[i] ≤ R[j]

A[k] = L[i]

i = i + 1

else

A[k] = R[j]

j = j + 1

Q3. Use mathematical induction to show that when n is an exact power of 2, the solution of the recurrence

2 if n = 2,

T(n) = 2T(n/2) + n if n = 2k , for k > 1

is T(n) = n lg n.

Ans.

* Base Case

For n = 21 , T(n) = 2 lg 2 = 2 [ ∵ lg22 = 1]

* Let us assume it hold for some constant k

⇒ T(n) = n lg n = 2k lg 2k = k2k  [Eq-1]

* Let us try to prove for k+1

T(2k+1) = 2 T(2k+1/2) + 2k+1

= 2 T(2k) + 2k+1

From eqn 1

= 2 (k 2k) + 2k+1

= k 2k+1 + 2k+1

= 2k+1(k+1)

= 2k+1 lg 2k+1 [a = lg22a]

= n lg n

Q4. We can express insertion sort as a recursive procedure as follows. In order to sort A[1…n] , we recursively sort A[1…n-1] and then insert A[n] into the sorted array A[1..n-1]. Write a recurrence for the running time of this recursive version of insertion sort.

Ans.

Let us derive a recurrence for insertion sort.

**Divide** : Divide the n-element array into two sub-arrays with one part containing 1 element and other with n-1

**Conquer** : Sort each sub-array recursively; the single element is already sorted.

**Combine** : Combine the two sub-arrays by inserting the single element into it’s appropriate position that keeps the array sorted.

Let T(n) : Worst running time to sort n-elements.

D(n) : Time required to divide the sub-problem. Calculating index

C(n) : Time required to combine the solutions. Insertion element in its proper position.

* Recurrence :

Θ(1) if n = 1

T(n) = T(n-1) + T(1) + D(n) + C(n) n > 1

Since division step only do index calculation it’s Θ(1) and so do sorting a single element. Hence recurrence reduces to

Θ(1) if n = 1

T(n) = T(n-1) + Θ(1) + Θ(1) + C(n) n > 1

Since in combine step we need to find appropriate position of the element. Hence we have to traverse the array and in worst case we may need to traverse n-1 items. Hence C(n) = Θ(n).

Θ(1) If n =1

T(n) = T(n-1) + Θ(n) n >1

Q5. Referring back to the searching problem (see Exercise 2.1-3), observe that if the sequence A is sorted, we can check the midpoint of the sequence against **v** and eliminate half of the sequence from further consideration. The binary search algorithm repeats this procedure, halving the size of the remaining portion of the sequence each time. Write pseudocode, either iterative or recursive, for binary search. Argue that the worst-case running time of binary search is Θ(lg n).

Ans.

* Iterative:

ITERATIVE-BINARY-SEARCH(A, v, low, high)

while low ≤ high

mid = floor((low + high) / 2)

if v == A[mid]

return mid

else if v > A[mid]

low = mid + 1

else high = mid - 1

return NIL

* Recursive:

RECURSIVE-BINARY-SEARCH(A, v, low, high)

if low > high

return NIL

mid = floor((low + high) / 2)

if v == A[mid]

return mid

else if v > A[mid]

return RECURSIVE-BINARY-SEARCH(A, v, mid + 1, high)

else return RECURSIVE-BINARY-SEARCH(A, v, low, mid - 1)

**Recurrence** :

Let us derive a recurrence for insertion sort.

**Divide** : Divide the n-element array into two sub-arrays with one part containing 1 element and other with n-1

**Conquer** : Sort each sub-array recursively; the single element is already sorted.

**Combine** : Combine the two sub-arrays by inserting the single element into it’s appropriate position that keeps the array sorted.

Let T(n) : Worst running time to sort n-elements.

D(n) : Time required to divide the sub-problem. Index calculation (calculation of mid)

C(n) : Time required to combine the solutions. Return the solution.

D(n) we only need to calculate the mid which takes constant time. Hence D(n) is Θ(1).

C(n) : In combine step or better say. In resulting step all we need to do is return the index if we find the value which is independent of the size of array. Hence it’s Θ(1).

T(n) : In recursive step we work with half part ( Either A[low….mid] or A[mid+1….high]). Hence though it divides our problem in two part from which we consider only **one** hence a = 1 .

By recurrence relation

Θ(1) if n = 2,

T(n) = aT(n/b) + D(n) + C(n) if n = 2k , for k > 1 {where **a** is no. of sub-problems each of size **1/b**}.

Above statement recurrence reduces to

Θ(1) n = 1

T(n) = T() + Θ(1) n >1

Θ(1) +n = 1

T(n) ≤ T() + Θ(1) n >1

Since there are many methods to solve this but we haven’t studies them yet. Let’s us try to solve this through from our previous knowledge.

T(n) ≤ T() + Θ(1), 1

T() + Θ(1) + Θ(1), 2

T() + Θ(1) + Θ(1) + Θ(1), 3

……………………………………………………….

T(2) + Θ(1) + Θ(1) +……….+ Θ(1) k--1

T(1) + Θ(1) + Θ(1) +……….+ Θ(1) k

Θ(1) + Θ(1) + Θ(1) +……….+ Θ(1) k+1

From above steps we can derive relation

T(n) ≤ Θ(1) + Θ(1) + Θ(1) +……….+ Θ(1)

Now all we need to do is to calculate the no. of Θ(1) terms. Let us see how we can do that.

Notice that there is only 1 such term after step 1 and the denominator of problem size become 2 = 21

In step 2 there are two such terms and the denominator of problem size becomes 4= 22.

Same referring to 3 we can see the denominator size is 8 = 23 ( 3 being the no. of Θ terms.)

Hence in kth step .

Denominator will be 2k , using this statement we get

T(n) ≤ T + Θ(1) + …+ Θ(1)

In order to have size 1 as we get from k +1 step.

= 1

n = 2k

lg n = lg 2k

lg n = k lg 2

k = lg n

Substituting back we get

T(n) ≤ T + Θ(1) + …+ Θ(1)

lg n terms

T(n) ≤ (lg n + 1) Θ(1)

**T(n) = Θ(lg n)**

Q6. Observe that the while loop of lines 5–7 of the INSERTION-SORT procedure in Section 2.1 uses a linear search to scan (backward) through the sorted subarray A[1….j-1]. Can we use a binary search (see Exercise 2.3-5) instead to improve the overall worst-case running time of insertion sort to Θ(n lg n)?

Ans. We can use binary search **but** it **won’t** help us in any way. In insertion sort we need to copy element to its neighbor. Binary search can tell us how many shift we need to do but it won’t get rid us of copying we need to do.

Q7. Describe a (n lg n) time algorithm that, given a set S of n integers and another integer x, determines whether or not there exist two elements in S whose sum is exactly x.

Ans.

**Solution 1**-

Explanation :

We first sort the array. After sorting we take two pointer and point them to the first and the last element. Since array is sorted we are sure that 1st will point to smallest element and 2nd to largest element.

Now we compute the sum of value pointed by pointer.

* If sum happens to be **desired result** we simply return it.
* **If Sum is smaller**. Since we are already summing the smallest and largest element. To produce a bigger result. We either need to point to 2nd to a bigger element or 1st to bigger element. Since increasing 2nd will lead to out of bound. We increase 1st pointer to point just bigger element. If that matches we return.
* **If sum is larger**. We either need to decrease the 1st to point a smaller element or 2nd to point a smaller element. Since 1st already point to smallest , decreasing it will lead to out of bound. Hence we reduce the 2nd pointer to point just smaller element. If that matches we return otherwise will do the same process again and again. We do this process while 2nd pointer pointing position is greater than 1st ‘s location. Once 1st crosses second we simply return false (or null as per implementation) to indicate that sum pair doesn’t exist.

Use Merge Sort to sort the array A in time Θ(n lg(n))

i = 1

j = n

while i < j do

if A[j] + A[j] = S

return true

if A[i] + A[j] < S

i = i + 1

if A[i] + A[j] > S then

j = j − 1

return false

**Solution 2**

Use Merge Sort to sort the Array A in time Θ(n lg n)

for A[i] in A[1,n]

Complement = sum – x;

if binarySearch(A,complement,i,n)

return true

First, sort S, which takes Θ(n lgn). Then, for each element si ​ in S , i = 1..…,n, search A[i+1….n] for si' = x – si  by **binary search**, which takes **Θ(lgn).**

* If si' is found, return its position;
* otherwise, continue for next iteration.

T(n) = Θ(n lg n) + Θ(lg n) + ….+ Θ(lg n)

Merge Sort Time Time to perform binary search

T(n) = Θ(n lg n) + n Θ(lg n)

In Worst case there will be n searches.

T(n) = Θ(n lg n).