CHAPTER 02

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.

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

↓(No swap)

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

↓(No swap)

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

↓(3 swaps)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 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*** :- At the start of each iteration of the **for** loop, the subarray A[1..i-1] that are different than ***v***.

***Initialization:-*** Before the start of iteration of loop ( i = 1) , the subarray is hence, so the variant holds.

***Maintenance:-*** During each loop iteration we compare ***v*** with ***A[i]*** . If they are same we return it which is true result. Otherwise, we continue to the next iteration. At the end of the loop , we conclude the ***v*** is not an element of A.

**Termination:-**  The loop terminates when i > A.length. At that time i would be n+1. substituting it 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.

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. Pseudocodes:-

***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

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

Ans. 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:-***

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

***Initialization:-*** Before the start of iteration of loop ( i = 1) , 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.

Assuming equal probability of occurrence 1/n, average number of elements which need to be checked is 1/n \* (1 + 2 + ... +n) = (n+1)/2. Running time is Θ(n)

Worst case, the element to search is dead last in the array. In that case n elements need to be searched. Running time is Θ(n)

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

Ans. Modify the algorithm so it checks whether the input satisfies some special case condition. If it does, output a pre-computed answer.

E.g.: You can modify any algorithm to have a best case time complexity of O(n) by adding a special case, that if the input matches this special case - return a cached hard coded answer (or some other easily obtained answer.

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.

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

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

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

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

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

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

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

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

***Merging of Sorted***

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

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

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

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

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

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

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

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

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.

* Recurrence :

Θ(1) if n ≤ c

T(n) = T(n-1) + I(n) otherwise

Where I(n) denotes the amount required to insert in the sorted A[1…n-1] array. Since we may have to shift all n-1 element. Upper bound for I(n) = Θ(n).

Hence recurrence can be written as :

Θ(1) if n ≤ c

T(n) = T(n-1) + Θ(n) otherwise

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 :
  + During recursive binary search. We find mid of the mid of the array and if mid has the required key , we simply return.
  + If array has size 1 we simply check that position and return if key is found.(Best Case)
  + Otherwise , We calculate mid , compare it and ***discard*** other part of the array.

Ans.

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***}

In case of binary search though we have two sub-problem, since we divide array in two part. ***But***  we discard one part of the array. Hence ***a = 1* ,** since we divide in 2 part hence size of each is ***half of whole size***

∴ b = 2

Since Division and combine step (D(n) & C(n)) requires only to find mid. It takes Θ(1) time.

From a ,b and above statement recurrence reduces to

Θ(1) if n = 1

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

Solution of above recurrence is 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***-

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

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

***Solution 2***

First, sort S, which takes Θ(*n* lg*n*). 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.

The time complexity of the algorithm is Θ (n lg n) + n. Θ(lg n) = Θ(*n* lg *n*).