Assignment 3

Master theorem

$T(n) = aT\left[\left(\frac{n}{b}\right) + O(n^d)\right]$

for $a > 0$, $b > 1$, $d \geq 0$, then

$$T(n) = \begin{cases} \text{Case 1} \\ O(n^d) & \text{if } d > \log_b^a \\ \text{Case 2} \\ O(n^d \log n) & \text{if } d = \log_b^a \\ \text{case 3} \\ O(n^{\log_b^a}) & \text{if } d < \log_b^a \end{cases}$$

1. A) $T(n) = 2T(n/4) + 1$

$a = 2, \quad b = 4, \quad d = 0$

$\log_b^a = \log_4^2 = 0.5 > d$

Therefore case 3: $O(n^{0.5})$

B) $T(n) = 2T(n/4) + \sqrt{n}$

$a = 2, \quad b = 4, \quad c = 1/2$

$\log_b^a = \log_4^2 = 0.5$

C) $T(n) = 2T(n/4) + n$

$a = 2, \quad b = 4, \quad d = 1$

$\log_b^a = \log_4^2 = 0.5 < d$

Therefore case 1: $O(n)$

d) $T(n) = 2T(n/4) + n^2$

$a = 2, \quad b = 4, \quad d = 2$

$\log_4^2 = 0.5 < d$

Therefore case 1: $O(n^2)$

e) $T(n) = 2T(n/3) + 1$

$a = 2, \quad b = 3, \quad d = 0$

$\log_b^a = \log_3^2 > d$

Therefore case 3, $O(n^{\log_3^2})$

F) $5T(n/4) + n$

$a = 5, b = 4, d = 1$

$\log_b^a = \log_4^5 > d$

Therefore case 3, $(n^{\log_4^5})$

G) $T(n) = 7T(n/7) + n$

$a = 7, b = 7, d = 1$

$\log_b^a = \log_7^7 = 1 = d$

Therefore case 2 : $O(n \log_7 n)$

H) $T(n) = 9T(n/3) + n^2$

$a = 9, b = 3, d = 2$

$\log_b^a = \log_3^9 = 2 = d$

Therefore case 2 : $O(n^2 \log_3 n)$

I) $T(n) = 8T(n/2) + n^3$

$a = 8, b = 2, d = 3$

$\log_b^a = \log_2^8 = 3 = d$

Therefore case 2 : $O(n^3 \log_2 n)$

J) $T(n) = 49T(n/25) + n^{3/2} \log n$

Using Substitution

$T(n) = 49T(\frac{n}{25}) + n^{3/2} \log n$

$a = 49, b = 25, d = 3/2$

$\log_b^a = \log_{25}^{49} = 1.2 < d$

Therefore $O(n^{3/2} \log n)$

(2)

2.14

Sol:-

-> We can sort the element of the array using mergesort in $O(n \log n)$. Then in one linear pass copy of the element to a new array, eliminating the duplicates by traversing the sorted array.

Let the Array A has the following numbers: 2 3 1 3 1 4 Once the duplicated number are removed the output array should be 2 3 1 4. But that the order of elements in the final output array maintained. this algorithm sort the output array

Function remove-duplicate (a[1.....n])
Input : An array of number a[1 .... n]
output : Array A with duplicate remove
Construct an array temp [1.... n]:
    temp[i].value = a[i]
    temp[i].key = i
Sort temp based on value
remove duplicates from temp based on value:
    Keep the entry with minimum key
Sort temp based on key
Construct array A from temp:
    A[i] = temp[i].value
Return A

The field key helps in keeping the order of elements in output array. Two sort takes $O(n \log n)$. Duplicate removal considering key is $O(n)$. Therefore the algorithm is $O(n \log n)$.

If we don't consider maintaing the order of the original array in the output array the algorithm can be simply given as

function remove duplicate ($a[1 \ldots n]$)
Input : An array of number $a[1 \ldots n]$
Output : Array A with duplicates removed
Sort a
Construct array A from a
by removing duplicate entries from a return A.

2.16

Solution

To search an element in the sorted array of n integers in an infinite array, follow these steps:

* Search for the value of n, so that search of the element can be applied to n elements only.
* Apply binary search over n elements, if element is not found, return error.

Algorithm:

1. Take an Array A[] with infinite size
   right -> 1
2. left -> 1 // Assign index 1 to both right and left and 0 to mid
   mid -> 0
3. While A[right] ≠ ∞
   left = right // check for infinity in the array
   right = 2 * right
4. When left - right > 1

$$mid = left + \left[\frac{(right - left)}{2}\right]$$

if A[mid] == ∞

right = mid // if infinity is found, then move to left to find //first infinity

else

left = mid // if infinity is not found, move right

5. λ = left + 1 // λ is the size of finite Integers array.

6. Apply binary_search (A, λ, x)

7. If x is found

return index of the element

else

infinity

Both the steps are logarithmic, thus the algorithm is taking $O(\log \lambda)$ time.

(4)

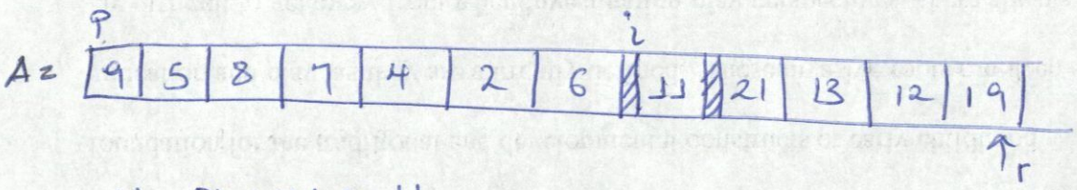
2.17

Solution

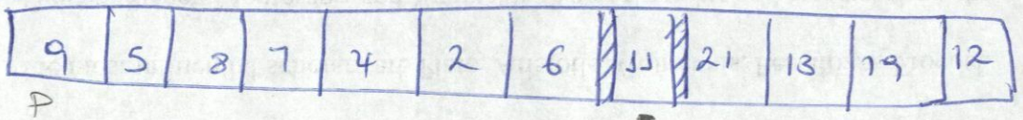
-> The algorithm always divide the array into 2 halves, and checks with both the halves. In this process we always will be left with single element eventually. and that element if index equal content, will return true, which means if at any index of the array, if the element satisfies it returns true, and false otherwise. So the algorithm divide the array into 2 halves and the algorithm runs for both the halves, the boundary running time $2^* \log n$ which belongs $O(\log n)$.

(5) A = {13, 19, 9, 5, 12, 8, 7, 4, 21, 2, 6, 11}

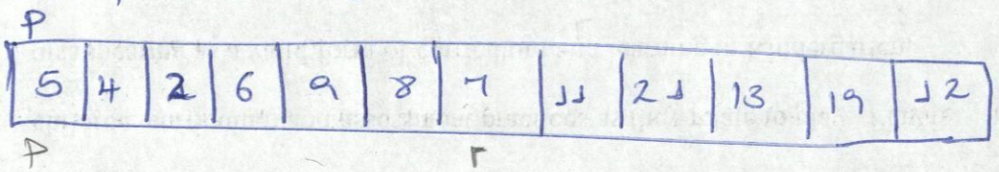
↑ pivot



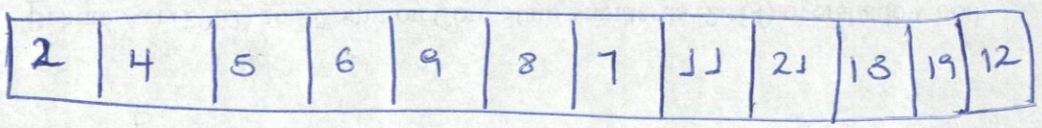
p=0, r=11, Pivot value = 11



p=0, r=6, pivot value = 6



p=0, r=2, pivot Value=1



p=1, r=2, pivot value=5

2	4	5	6	9	8	7	11	21	13	19	12

p=4, r=6, pivot value=7

2	4	5	6	7	8	9	11	21	13	19	12

p=5, r=6, pivot value=9

2	4	5	6	7	8	9	11	21	13	19	12

p=8, r=11, pivot value=12

2	4	5	6	7	8	9	11	12	13	19	21

p=9, r=11, pivot value=21

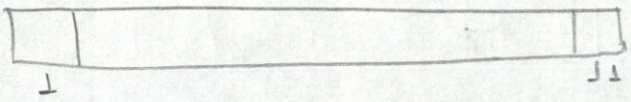
2	4	5	6	7	8	9	11	12	13	19	21

p=9, r=10, pivot value=19

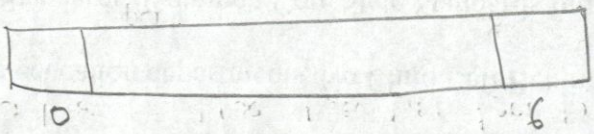
2	4	5	6	7	8	9	11	12	13	19	21

6) a) A = [6, 0, 2, 0, 1, 3, 4, 6, 1, 3, 2]

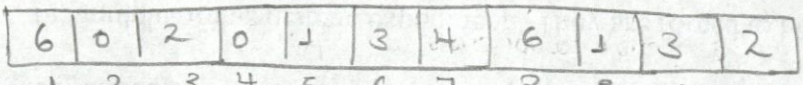
Creat B =



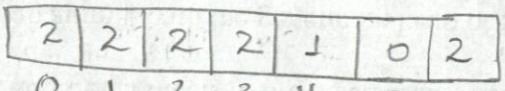
Creat C =



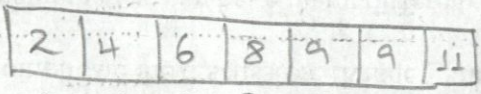
1. A =



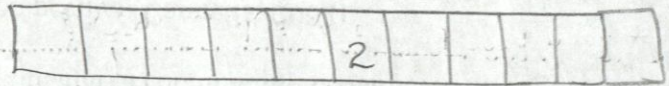
C =



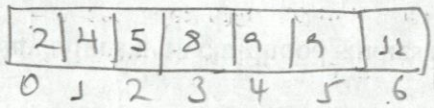
2. C =



3 B =

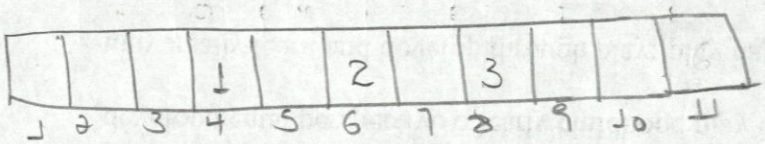


C =

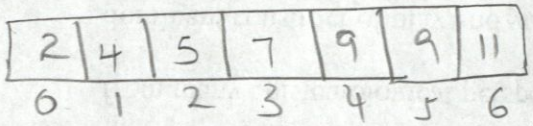


A[11] = 2
C[2] = 6
Set B[6] = 2
decrement C[2] by 1

4. B



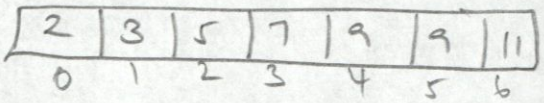
C.



A[10] = 3
C[3] = 8
Set B[8] = 3
decrement C[3] by 1

5. A[9] = 1
C[1] = 4
Set B[4] = 1
decrement C[1] by 1

C



6. A[8] = 6
C[6] = 11
Set B[11] = 6
decrement C[6] by 1

B

0	0	1	1	2	2	3	3	4	6	6
1	2	3	4	5	6	7	8	9	10	11

C

2	3	5	7	9	9	10
0	1	2	3	4	5	6

7. A[7] = 4
C[4] = 9
set B[9] = 4
decrement C[4] by 1

C

2	3	5	7	8	9	10
0	1	2	3	4	5	6

8. A[6] = 3
C[3] = 7
Set B[7] = 3
decrement C[3] by 1

C

2	3	5	6	8	9	10
0	1	2	3	4	5	6

9. A[5] = 1
C[1] = 3
Set B[3] = 1
decrement C[1] by 1

2	2	5	6	8	9	10
0	1	2	3	4	5	6

10. A[4] = 0
C[0] = 2
Set B[2] = 0
decrement C[0] by 1

1	2	5	6	8	9	10
0	1	2	3	4	5	6

11. A[3] = 2
C[2] = 5
Set B[5] = 2
decrement C[2] by 1

1	2	4	6	8	9	10
0	1	2	3	4	5	6

12. A[1] = 0
C[0] = 2
Set B[2] = 0
decrement C[0] by 1

0	2	4	6	8	9	10
0	1	2	3	4	5	6

13. $A[1] = 6$

$C[6] = 10$

set $B[10] = 6$

decrement $C[6]$ by 1

C

0	2	4	6	8	9	9

Therefore B =

0	0	1	1	2	2	3	3	4	6	6
1	2	3	4	5	6	7	8	9	10	11

B) Assume that our integers are stored in array $A[1 \ldots n]$ and thus, length $[A] = n$. We need two additional arrays $B[1 \ldots k]$ and $C[1 \ldots k]$. Our algorithm will first initialize all elements of array B to 0. This step requires $O(k)$ time. Next, for each element of array A (i.e, $A[i]$ for $i = 1, 2, \ldots, n$), we will increment $B[A[i]]$. observe that this step will require $O(n)$ time and $B[i]$ for $j = 1, 2, \ldots, k$, now contains the number of elements of A having value j. Finally make $C[1] = B[1]$ and for each element l of array C where $l = 2, 3, \ldots, k$, we will compute $C[l] = B[l] + B[l-1]$. This step takes $O(k)$ time.

Now, to answer any query about how many of n integers fall into a range $[a \ldots b]$, we simply compute $C[b] - C[a] + B[a]$ Observe that this computation requires only $O(1)$ time after the $O(n+k)$ preprocessing time.