GA HW2

Piyush Hinduja

September 2023

Answer 1

Q-1) a)

We are given an array 'A' of size 'n' which contains '+1's and '-1's.
Our goal is to find an index 'i' such that A[i] = +1 and A[i+1] = -1.
This can be done using simple binary search with slight modifications or additions.
We know that Binary search takes $\log_2(n)$ time in worst case.
But our main concern is how do we know which side to perform binary search at each step, since the given array is not sorted.
This can be solved using the given information that A[0] = +1 and A[n-1] = -1.
Case 1: A[mid] = +1
During Binary Search, if we get A[mid] = +1, we will compare it with it's neighbouring element A[mid+1] and if it is -1, we will return i = mid.
But in other case we will perform binary search on the right part of the array because we know that the end or last element is -1 and we will achieve answer in $\log_2(n)$ even if A[mid] = +1, A[n-1] = -1 and all the elements in between are +1.
Case 2: A[mid] = -1
During Binary Search, if we get A[mid] = -1, we will compare it with it's neighbouring element A[mid-1] and if it is +1, we will return i = mid-1.
But in other case we will perform binary search on the left part of the array because we know that the start or first element is +1 and we will achieve answer in $\log_2(n)$ even if A[mid] = -1, A[0] = +1 and all the elements in between are -1.

Algorithm:

```
int[] A = [+1, ......, -1]
function binary(int s, int e, int[] arr):
        mid = (s + e) / 2
        if(A[mid]==+1):
            if(A[mid+1]==-1):
                return mid
            else:
                binary(mid+1, e, arr)
```

```
    if(A[mid]==-1):
        if(A[mid-1]==+1):
            return mid-1
        else:
            binary(s, mid-1, arr)
```

Hence, the worst case time complexity of this algorithm will be same as binary search, i.e., $\log_2(n)$.

Q-1) b)

In this problem also, we are given an array of size 'n' but now it contains unsorted elements and we need to find a local minimum element such that A[i-1] > A[i] < A[i+1].
So we will start by applying binary search.
If arr[mid] satisfies the above condition of local minimum then we return that element.
But we can also encounter two other cases:
Case 1: A[mid-1] > A[mid] > A[mid+1]
In this case we will apply binary search to the right part of the array because we will find at least one local minimum even in the worst case i.e. even if all the elements are in decreasing order in right sub-array we can output the last element because it needs only one comparison with A[n-2] one.
Case 2: A[mid-1] < A[mid] < A[mid+1]
In this case we will apply binary search to the left part of the array because we will find at least one local minimum even in the worst case i.e. even if all the elements are in decreasing order in left sub-array we can output the first element because it needs only one comparison with A[1] one.

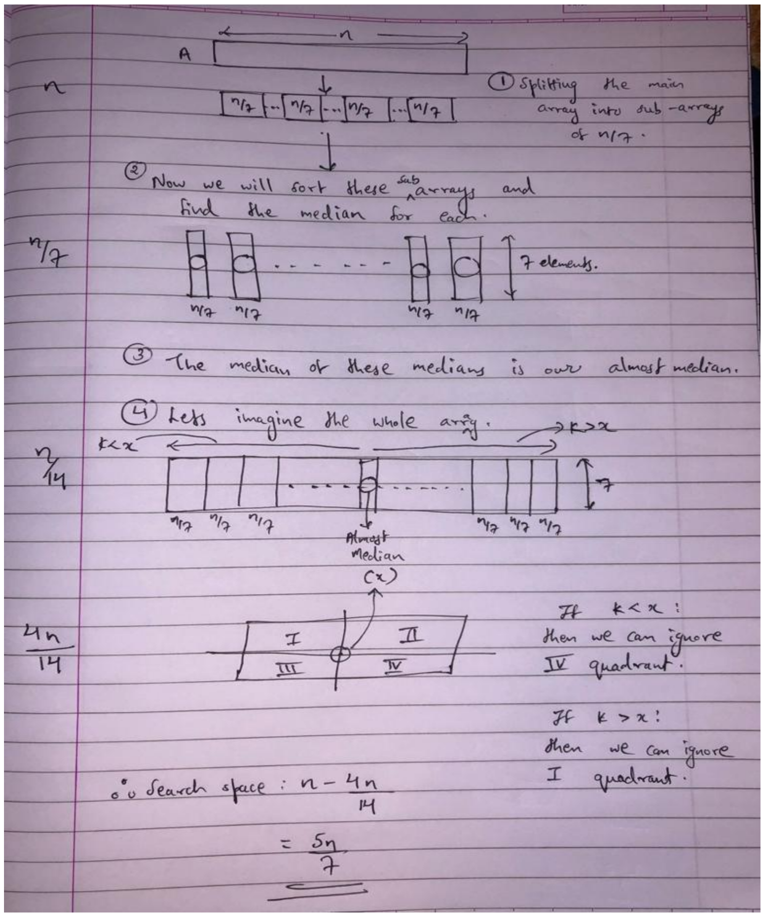
```
Algorithm:
int[] A
function binary(int s, int e, int[] arr):
        mid = (s + e) / 2
    if(A[mid-1] > A[mid] < A[mid+1]):
        return mid
        if(A[mid-1] > A[mid] > A[mid+1]):
            binary(mid+1, e, arr)
        if(A[mid-1] < A[mid] < A[mid+1]):
            binary(s, mid-1, arr)
```

Hence, the worst case time complexity of this algorithm will be same as binary search, i.e., $\log_2(n)$.

Answer 2

Q-2) a)

If we divide the main array into sub-arrays of size 7, then the recurrence relation would become: First of all size of each sub-array would be, $\frac{n}{7}$
Here depending upon the value of k (index to be found), whether it is $>$ or $<$ x (the derived almost median), we will eliminate half of the sub-arrays.
Therefore, our search space changes from $\frac{n}{7}$ to $\frac{n}{14}$.
Also, we know that, among all the elements from the above remaining sub-arrays, half of them can be eliminated (again depending on the same above condition whether $k < x$ or $k > x$) including their medians.
Half of the elements are $\frac{7}{2} = 3$ and one median, together 4 elements.
Therefore, the updated search space for elimination is $4 * \frac{n}{14}$.
Remaining searching space becomes $n - \frac{4n}{14} = \frac{5n}{7}$.
Above sentences can be clearly understood using below image.



① Splitting the main array into sub-arrays of n/7.

② Now we will sort these sub arrays and find the median for each.

③ The median of these medians is our almost median.

④ Lets imagine the whole array.

If $k < x$: then we can ignore IV quadrant.

If $k > x$: then we can ignore I quadrant.

$\therefore$ Search space : $n - \frac{4n}{14}$

$= \underline{\underline{\frac{5n}{7}}}$

Now, updated recurrence relation is,

$$T(n) = T(\frac{5n}{7}) + T(\frac{n}{7}) + cn$$

Bonus:

$$T(n) = T(\frac{5n}{7}) + T(\frac{n}{7}) + cn$$

Comparing the equation with,

$$T(n) = a1 * T(b1 * n) + a2 * T(b2 * n) + g(n)$$

We get,$a1 = 1, a2 = 1, b1 = \frac{5}{7}, b2 = \frac{1}{7}$ and $g(n) = cn$
Substituting the above values in,

$$a1 * (b1)^p + a2 * (b2)^p = 1$$

We get,

$$p = 0.76$$

Now, we know $\theta(n)$ formula by Akra-Bazzi, i.e.,

$$T(n) = \theta(n^p + n^p \int_1^n \frac{g(u))}{u^{p+1}} du)$$

Substituting values of p and g(n) in above equation,

$$T(n) = \theta(n^{0.76} + n^{0.76} \int_1^n c * \frac{u}{u^{1.76}} du)$$

$$= \theta(n^{0.76} + n^{0.76} \int_1^n u^-0.76du)$$

$$= \theta(n^{0.76} + n^{0.76} * \left[\frac{u^{0.24}}{0.24}\right]_1^n)$$

$$= \theta(n^{0.76} + n^{0.76} * \left[\frac{n^{0.24}}{0.24} - \frac{1^{0.24}}{0.24}\right])$$

$$= \theta(0.24 * n^{0.76} + n^{0.76} * \left[n^{0.24} - 1\right])$$

$$= \theta(n - 0.76 * n^{0.76})$$

$$\theta(n)$$

Q-2) b)

$$T(n) = T(\frac{5n}{7}) + T(\frac{n}{5}) + cn$$

Comparing the equation with,

$$T(n) = a1 * T(b1 * n) + a2 * T(b2 * n) + g(n)$$

We get, $a1 = 1, a2 = 1, b1 = \frac{5}{7}, b2 = \frac{1}{5}$ and $g(n) = cn$
Substituting the above values in,

$$a1 * (b1)^p + a2 * (b2)^p = 1$$

We get,

$$p = 0.86$$

Now, we know $\theta(n)$ formula by Akra-Bazzi, i.e.,

$$T(n) = \theta(n^p + n^p \int_1^n \frac{g(u))}{u^{p+1}} du)$$

Substituting values of p and g(n) in above equation,

$$T(n) = \theta(n^{0.86} + n^{0.86} \int_1^n c * \frac{u}{u^{1.86}} du)$$

$$= \theta(n^{0.86} + n^{0.86} \int_1^n u^{-0.86} du)$$

$$= \theta(n^{0.86} + n^{0.86} * \left[\frac{u^{0.14}}{0.14}\right]_1^n)$$

$$= \theta(n^{0.86} + n^{0.86} * \left[\frac{n^{0.14}}{0.14} - \frac{1^{0.14}}{0.14}\right])$$

$$= \theta(0.14 * n^{0.86} + n^{0.86} * \left[n^{0.14} - 1\right])$$

$$= \theta(n - 0.86 * n^{0.86})$$

$$\theta(n)$$

Hence, we can say that the time complexity of new recurrence relation also remains the same.

Answer 3

We are given that there is an array A with 'n' elements and it compulsorily contains a majority element (x) such that it occurs more than $\frac{n}{2}$ times.

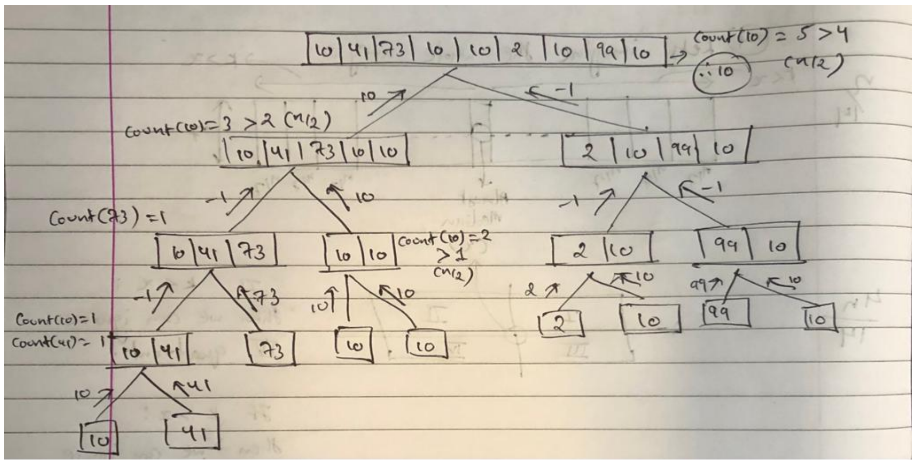
Q-3) a)

If we pick 'r' elements randomly, and if it's occurrence is less than $\frac{n}{2}$ then we output FAIL.
But we know that the probability of an element being the major element x is more than 50%.
Hence, for 'r' elements, the probability of 'r' elements being major elements are greater than 50%.
And automatically probability of outputting FAIL is less that $\frac{1}{2^r}$.
We can understand it using the below example,
If r = 5, then for each 'r', P(r!=x) ¡ $\frac{1}{2}$
Hence, eventually multiplying the probabilities in case of cumulative probabilities, we get P(r!=x) = P(FAIL) < $\frac{1}{2^r}$.

Q-3) b)

Now for finding that element, we are asked to use divide and conquer.
We will take the main array A and keep dividing it into half till we get single elements.
Now while traversing upwards, at each level, we will count the number of times each element is received.
If the count of any element is greater than $\frac{n}{2}$, we will return that element and if there is no majority element at that step, we will return -1.
When we get a value 'x' and -1 at a particular node, we will ignore -1 and count the occurrences of 'x'.
We will follow this bottom up traversal till the root node and we will get the majority element.

Time complexity for this approach will be $O(n \log n)$, since dividing the array into two parts at each level takes $\log n$ time and comparing each element with its neighbouring node at leaf level takes 'n' time.
Hence, $O(n \log n)$ in total.



Algorithm:

Divide the array into halves till single elements.

```
function majority(int[] arr):
    if(arr.length == 1):
        return arr[0]
    else:
        if(count(x) in arr > n/2):
            return x
        else:
            return -1
```

Answer 4

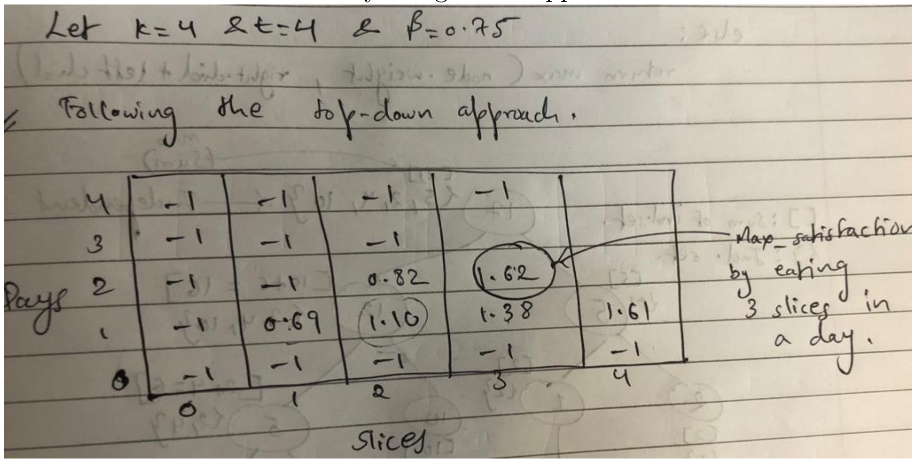
We will store 2 arrays,
Array 1 will contain all the max_satisfactions which we can achieve by consuming x number of slices in y days.

Now for the case that we have completed 3 slices till 2 days we have to choose max([2 slices at day 1 and 1 slice at day 2] AND [1 slice at day 1 and 2 slices at day 2]).

max([$\beta^{(1-1)} * \log(1+2)$ + $\beta^{(1-1)} * \log(1+1)$], [$\beta^{(2-1)} * \log(1+1)$ + $\beta^{(2-1)} * \log(1+2)$])

Putting, $\beta = 0.75$
i.e. max(1.10+0.52, 0.69+0.82) = 1.62

We can then fill the entire array using above approach.



Let k=4 & t=4 & β=0.75

Following the top-down approach.

Days \ Slices	0	1	2	3	4
4	-1	-1	-1	-1	
3	-1	-1	-1		
2	-1	-1	0.82	1.62	
1	-1	0.69	1.10	1.38	1.61
0	-1	-1	-1	-1	-1

Max_satisfaction by eating 3 slices in a day.

In array 2, we will store the number of slices through which we are getting max_satisfaction for each day.

While filling the first array, we will save the [number of slices] by eating which we have got that max value on that particular day.
That values of slices which give us max_satisfaction will be stored in second array.
For example in above case of eating 3 slices in 2 days, we got the max_satisfaction by eating 1 slice on day 2 (and other 2 slices on day 1) we will put value 1 in cell (2, 3) or equivalent cell of array 1 where we have put 1.62.

Then using these two tables and back tracking them we can find the optimal solution, i.e., we will select a value from table 1 with maximum satisfaction and then keep on subtracting the slices day by day.

Algorithm:
1) Make array 1 as, A[i][j] = max($\beta^{(i-1)} * \log(1 + j)$)
(This can be done by iterating over all the possible combinations of days and slices).
2) Make array 2 such as A[i][j] is that [# of slices] from all the above combinations which gave us max_satisfaction.
3) Finally once we have both the arrays, we can backtrack from that cell in array 2 which has the max_satisfaction in array 1 for each day.
4) Return the best combination of days and slices got from last step.

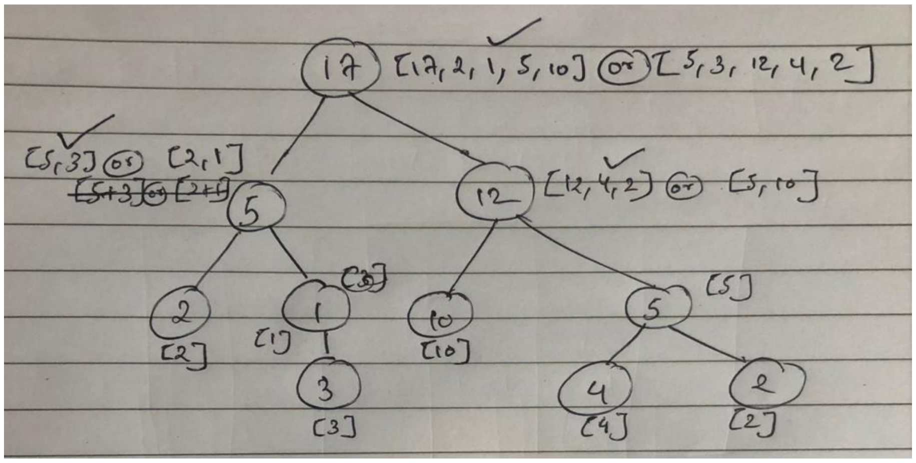
Answer 5

Here, we are given that an independent set is a collection of that nodes which are not connected to each other and we need to find such an independent set which has the maximum sum (if we consider value of nodes as their weights for simplicity), for a rooted tree.
We can perform this by writing a recursive function that returns max of that [node's weight] and [sum of independent set (or sum) for each of the child nodes at that level].
BUT in this case we are just comparing parent and child at each level whereas a parent is also not directly connected to it's grandchildren.
Hence we will have to choose a maximum of [node's weight + sum of it's max_independent of all the grand-children] and [sum of independent set (or sum) for each of the child nodes at that level]

Algorithm:
We will start with node = root (since it is a rooted node).

```
function max_independent(node):
    if(node == leaf):
        return node.weight
    else:
        return max[(node.weight + max_independent(node.right-grand-child) + max_independent(node.left-grand-child)), max_independent(node.right-child) + max_independent(node.left-child)]
```

We can refer the below tree example to understand it better (All the ticked sets are independent sets at that level):



In this approach, since we are traversing each node once, we can say it runs in time complexity of polynomial n.

References

1) Akra Bazzi by Kunal Kushwaha - YouTube

2) Eating cake problem by Econ John - YouTube

3) Maximum Independent Set by Easy Theory - YouTube