Name: Sindhuja Yerramalla
UUID : U00839259
mailid : syrrmlla@memphis.edu.

Algorithms | Problem Solving - Homework 1

Problem 1: P. 2.20 from DPV book.

* We will first solve this question by taking an example.

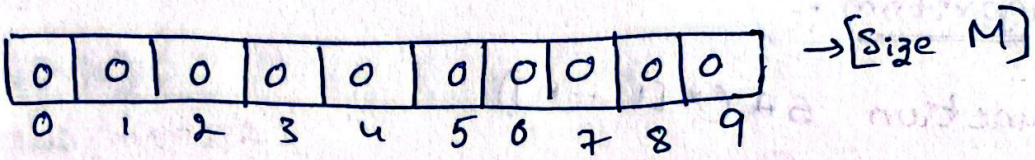
let the given array x be [1, 3, 8, 5, 10, 9, 2, 7, 7]
↳ size n

step 1 :- finding min and max values of x.

here minimum (min) is 1 and max = 10

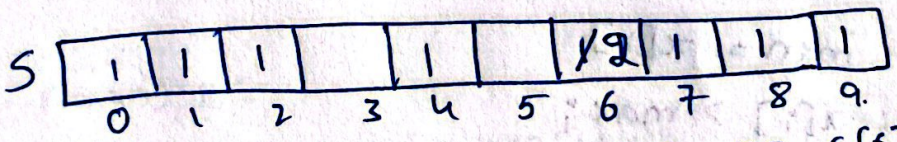
M = max − min = 9

step 2 : create another array say 'S' with size "max-min" and initialize values at every index to '0'



step 3 : By iterating through x, let us increment S at S[x[i] − min], [for i.e, x[1] = 1 and min = 1

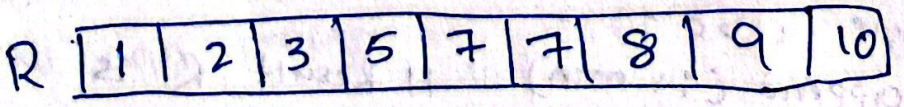
∴ S[x[1] − 1] = S[0] = 1]



• we have two sevens, so S[6] is incremented twice.

step 4 : Now we will iterate through S, create a new array R of size 'n' [if size of x is n], while iterating through S, if S[i] value is 1 then 'i+1' will be put in R, if S[i] value is 2 then 'i+1' will be put twice in R.

Basically, we are counting the # of the element occurs in our array and placing in R.



Scanned with CamScanner

Proof :-

In step 2 and step 3, we scan the array of size n and M, spent a contant amount of time. on each element., so it takes $O(n+M)$ time.

* This algorithm do not use comparision, so it is not bounded by $\Omega(n \log n)$. Hence it is proved.

Algorithm :-

```
Function SA (x[1---n])

max = x[1]
min = x[1]
for i=1 to size of x:
    if x[i] < min:
        min = x[k]
    if x[i] > max:
        max = x[k]      // got min & max.

S = array of zeroes of size [max-min]

for k=1 to size of x:
    S[x[k] - min]++     // Array 'S' in inirumented accordingly

R = [ ]                 // creating an empty array

for k=0 to size of S:
    c = S[k]
    while c>0:
        R.append (i+ min)   // Result R is retrieved as a sorted
        c--                                  array.
```

Problem 2: P. 2.22 from DPV book.

Algorithm for computing k^{th} smallest element in the Union of two lists:

We use Recurssion to solve such cases

* Function ksmallest(s[1,---m], t[1,---n], k)

```
if m=0:
    return t[k]
if n=0:
    return s[k]
if s[m/2] > t[n/2]:
    if k < (m+n)/2:
        return ksmallest(s[1----m/2], t[1,----n], k)
    else:
        return ksmallest(s[1,---m], t[(n/2)+1,----n], k-n/2)
else:
    if k < (m+n)/2:
        return ksmallest(s[1,----m], t[1,----n/2], k)
    else:
        return ksmallest(s[m/2+1,----m], t[1,---n], k-n/2)
```

Explanation:-

* It first checks if the middle value of first array is greater than second one. If yes, it checks whether the value of k is less than the combined middle value, If again yes, then we can eliminate second part of first array and can process 1st half of 1st array and

second array to the ksmallest function, if k is greater than the combined middle value then entire 1st array and 2nd half of second array is passed to the ksmallest function.

* If middle value of 1st array is less than the middle value of 2nd array then, we again check if the k is less than the combined middle value, If yes, only the first half of the 2nd array and entire first array is processed to ksmallest function. If no, then 2nd half of 1st array & entire 1st array is passed to ksmallest function.

* So every time, a constant amount of time is required in recursive call to half the arrays with m & n elements respectively, it takes logm + logn times
∴ the algorithm of "O(logm + logn)" runtime. algorithm is achieved.

Problem 3 :- P. 2.23 from DPV.

a) To solve this in O(nlogn), we will first divide the array into two halves, then we get the majority element from first half and majority element from the 2nd half, we then compare the count of both of them, If m_1 & m_2 are majority elements of 1st & 2nd half respectively then we will check whether the count of m_1 or m_2 is greater than the size of the array [main array].

Algorithm :-

We use divide-and-conquer and recurssion both techniques.

* Function Maj(A[1, ---- n]) → Algo to find majority o(logn) time.

```
if (n == 1):
    return (A[1])

mid = n/2
left_maj_element = Maj(A[1, ---- mid])
right_maj_element = Maj(A[mid+1, ---- n])
if (left_maj_element == right_maj_element):
    return left_maj_element
left_sum = Frequency(A[1, ---- n], left_maj_element)
right_sum = Frequency(A[1, ---- n], right_maj_element)
if (left_sum > mid+1):
    return left_maj_element
```

```
else if (right_sum > mid+1):
    return right_maj_element
else:
    return no_majority
```

```
Function Frequency (A[1, ---- m], ele)
{   for i=1 to n:
        if (A[i] = ele)
            count++
    return count
```

→ Algorithm for finding the frequency, has O(n) time.

→ Function Maj( ) has O(logn) which used devide-and-conquer and Function Frequency( ) has O(n) which has O(n). In total it is O(nlogn).

b)

Algorithm

```
Function Majority (A[1, ---- m])
• if (m = = 2):
      if (m[1] = m[2]):
          return (A[1])
      else:
          return no_majority
  temp = array[
```

```
for k=1 to m:
    if A[k] = A[k+1] :
        temp.append(A[k])
k+1   k+1
return Majority(temp)
```

Algorithm to find frequency of algorithm:-

```
Function Lfrequency[A[1, ---- m]]
q = Majority(A[1, ---- m])
count = Frequency(A[1, ---- m], q)
if count > ⌊n/2⌋ + 1 :
    return q
else
    return no-majority.
```

In entire algorithm, we used only one 'for loop' so hence. it has linear-time. This algorithm also uses the divide-and conquer approach, we first pair up the elements and. then look at each pair, if the two elements are different, we discard both of them, if they are same then we keep one of them. we then store the result in an array and do the same process.

Problem 4 :-

a) Let us consider that the given points are split into two halves called L & R. and our objective is to prove that L contains at most four points in any d x d square.

Suppose a square lies in dxd plane and there are five or more points in the square. Now if we divide the square into four parts, these are the smaller squares. The smaller squares must also contain a pair of points. The distance between any two points from the smaller square must be $T(n) = 2T(n/2) + O(n \log n)$. This is a contradiction. for the statement "Every pair of points is at least at a distance of "d". clear explanation is even the four points take four corners, the fifth point position will be a proof for contradiction. Therefore, we can conclude that dxd square contains at most four points.

b) while doing the dividing part of the divide-and-conquer, we make sure that the division does not result in a sub-problem containing only one point. Also, the calculation of distance of seven points is sufficient from the list. Because, at most eight points can reside in a d x 2d rectangle. Suppose if we consider dxd square froms the left rectangle. and then four points can reside in it at maximum. similar for right rectangle (dxd square). as per (a). Thus a dx2d rectangle can have at most 8 points. it is clear now that to check the distance of seven subsequent points in the list., it is sufficient.

∴ Divide-and conquer given to solve the problem of finding the closest pair is correct.

c) Consider that the objective is to show that the pseudo code of the closest pair algorithm has a recurrence relation $T(n) = 2T(n/2) + O(n \log n)$. which results in a running time of $O(n \log^2 n)$.

Pseudo code:

* The first step is to find a line that divides the set of points into two halves L & R. The points are stored in the array X & Y. X has increasingly sorted x-co-ord and y has increasingly sorted y-co-ordinates.
* In the second step, we use recursive call to find the required pair in the L and another recursive call to find in R. The finall will be the ~~minimum~~ minimum among these two.
* In final step, the closest is found among the 3 pairs of points. It can be one of the pair of points calculated by recursive call or it could be a pair with one point on each side. The recursion has sorting so it has $O(n \log n)$ time. and the overall time for the pseudocode is "$O(n \log^2 n)$" [for dividing & continuous recursion calling. $\log n$]

d) Yes, we can calculate the closest pair in $O(n \log n)$ running. If we send the sorted array to the first recursive call, we can save the running time. This way all the next recursive calls have linear time. So for the first call time is $O(\log n)$ and for n recursive calls the running time will be $O(n \log n)$.

Problem .5 : Problem 3.7 from DPV

a) The algorithm mentioned below recursively partition the vertices of the graph "G" into two sets and check if the two-coloring property is maintained or not.

If a vertex finds out the violation of coloring property then algo will stop and returns 'False', if the graph has no such violations then algo returns 'True'.

Algorithm of linear time:

```
Procedure. isBipartite (G):

    For v in V:
        visited[v] = 0

    For v in V:
        if explore(G, v, 1) == 0:
            return False

        else:
            return True.
end.

Procedure explore(G, v, color)

    visited[v] = color

    For edge(v,u) in E:
        if visited[u] <> 0 && visited[u] + color <> 0:
            return false
```

```
else:
    explore (G, u, color * (-1))

end;
```

b) Proof of bipartite graph without odd length-cycle:

lemma: The undirected graph is bipartite if and only if. if it has no cycles of odd length.

Proof:- Let us assume that the graph $G = (V, E)$ is taken with no odd length cycles and if we run DFS on G, for each edge $e = (u, v) \in E$, "e" is either a tree edge or back edge.

* If 'e' is a tree edge, then the bipartite property is maintained until the colors are alternated.

* If 'e' is a back edge, then there exists a cycle with even number of vertices in graph.

Thus, u and v must be odd number of edges away in DFS search tree and it is colored with different colors That means, it's clear that the graph is two-colored and it is bipartite by from the above assumption.

To prove "only-if" part, let us assume that an undirected graph 'G' contains no odd length cycle if the graph is bipartite. If we assume that the graph G=(V,E), is bipartite and suppose it contains at least one cycle of odd length. Now, choose some other node "u" along with this cycle and let the node 'v' be the end node. to be reached before returning to 'u'.

Consider the 'u' begins by coloring with one color, and the v begins by coloring with another color at different points in the cycle. But the cycle is still now in odd length.

→ As a result, it is established that the assumption that the graph "G" is bipartite cannot be true for two-coloring of the graph.

∴ The graph cannot contain an odd length cycle. Hence, it proves that "an undirected graph is bipartite if & only if it contains no cycles of odd length".

c) In an undirected graph with exactly one-odd-length cycle, the number of colours needed to color the graph is at most three. This is because the cycle itself can be colored with three colors, and the rest of the graph can be colored with two colors. Any additional colors would not be necessary as two colors are sufficient for the remaining part of the graph.

Problem 6 - Problem 3.17 from DPV book

a) To prove that Inf(P) is a subset of a strongly connected component of G, we first need to define the notion of it. A strongly connected component of a directed graph $G = (V, E)$ is a subgraph $G' = (V', E')$ of G such that for any two vertices u, v in V', there exists a path from $u \rightarrow v$ & from $v \rightarrow u$

Let P be an infinite trace of G & let Inf(P) be the set of vertices that occur infinitely often in P. We will show that Inf(P) is a subset of a strongly connected component of G.

Since P is an infinite trace, every vertex in Inf(P) is visited infinitely often times, which means that there exists a sequence of vertices $V_0, V_1, V_2, \ldots$ such that $V_0 = S$, V_i is connected to V_{i+1} by an edge in E, and every vertex in Inf(P) occurs in the sequence infinitely often

∴ the set of vertices in Inf(P) occurs infinitely often, it follows that there exists a subset of vertices in Inf(P) that form a strongly connected component.

b) To determine if G has an infinite trace, we can use DFS to traverse the graph from the start vertex S and check if there is any cycle that can be reached from it. A cycle in the graph indicates the presence of an infinite trace, as the path can continue to visit the vertices in the cycle infinitely often.

c) To determine if G has an infinite trace that visits some good vertex in V_G infinitely often, we can modify the DFS algorithm to keep track of the visited vertices and the # times each vertex is visited. If a good vertex in V_G is visited infinitely often, then there is an infinite trace that visits a good vertex in V_G.

d) To determine if G has an infinite trace that visits some good vertex in V_G infinitely often, but visits no bad vertex in V_B infinitely often, we can use a modified version of the DFS algorithm that keeps track of the visited vertices and the # times each vertex is visited. Additionally, we can keep track of the good and bad vertices that are visited, and if a bad vertex is visited infinitely often, we can stop the algorithm and return false. If a good vertex.

infinitely often, and no bad vertex is visited infinitely often, then we can return true, as there is an infinite trace that visits a good vertex in V_G but visits no bad vertex in V_B.

Note :- I have gone through various resources like. Youtube videos, stack exchange, chegg and few random pages on internet and also referred. text book, lecture notes.