classfellow

CPPL
Date : / / 20
Page No.

Algorithms HW 5

Q3 ⇒ Find worst-case complexity. Distance function takes O(1) time.

A3 ⇒

line 2 (intialization of heap) takes O(1) linear time.
line 3 (outer loop) takes (n-1) iterations.
line 4 (inner loop) takes (n-i) iterations from i+1 to n.
So, total iterations become

$$(n-1)+(n-2)\text{ ---- }+2+1 = \frac{n(n-1)}{2} = O(n^2)$$

As distance function takes O(1) time for each iteration, heap insertion takes O(log m) m = current size of heap
So, total insertion times for $O(n^2)$ distances, $= O(n^2 \log m)$
m will go to n^2, so it becomes, $O(n^2 \log n)$

line 8 (unionFind initialize takes O(n) time).

(line 10-16)

Delete min operation time complexity $= O(\log m)$.
Total iterations for this is (n-c) times.
So, total time for delete operations $= O((n-c)\log m)$
$= O(n \log n)$

line 12 (Find operations) and line 13 (Union operations) :-
It takes linear time, $O(\alpha(n))$, α = Ackermann function
So, total complexity $= O(n \cdot \alpha(n))$ for union-Find operations.
So, for Heap & Distance time complexity $= O(n^2 \log n)$
for unionFind operations $= O(n \log n) + O(n \cdot \alpha(n)) = O(n \log n)$
As, $O(n^2 \log n)$ is dominant term over $O(n \log n)$,
So,
worst-case complexity $= O(n^2 \log n)$

Q 2-)

A 2-)

a) To implement improved data structure, we need to add an array called 'min' that keeps track of minimum element in each partition. During initialization we will set 'min' array such that each element point to itself as its own minimum.

Modified

1. Algorithm: UnionFind.Initialize
2. Input n: size of UnionFind to initialize
3. Output: a: UnionFind of size n where every element points to itself
4. uf = Array(n)
5. Initialize uf to 1..n
6. size = Array(n)
7. Initialize size to 1
8. min = Array(n)
9. Initialize min to 1..n
10. return(uf, size, min)

We add a new array 'min' to store min element in each set at line 8.

At line 9, we initialize 'min' such that min[i] = i for each index i, so that each element is initially its own minimum.

At line 10, we return additional 'min' array.

b) For Find operation, time complexity = $O(\alpha(n))$

For union operation, time complexity = $O(\alpha(n))$.

For Find min operation, time complexity = $O(\alpha(n))$.

Use of path compression in Find operation ensures that each time an element's root is found, the structure is updated so that queries are faster.

Union by size or rank ensures that smaller tree always points to root of larger tree, keeping overall tree height in terms of logarithmic.
'min' array is updated during Union operation, so it maintains minimum value correctly after unions.

c) To modify Union operation so that 'min' always stores the minimum element of tree at every root, we need to initialize 'min' array so each index of 'min' array should initially be set to value of element itself. Then, update Union operation to also update 'min' array when 2 sets are merged.
Here's modified pseudocode,

```
Input: (uf, size, min): unionFind to modify
Input: a: index of 1 element to union
Input: b: index of another element to union
Output: modified uf that merges trees that has a and b.
Algorithm: uf.Union
ra = uf.Find(a)
rb = uf.Find(b)
if ra == rb then
    return
end
if size[ra] > size[rb] then
    Swap ra and rb
end
uf[ra] = rb
size[rb] = size[ra] + size[rb]
min[rb] = min(min[rb], min[ra])
```

d) Time Complexity for Find operation = $O(\alpha(n))$
Time Complexity for a and b each = $O(\alpha(n))$ (constant time)
Time Complexity for swapping, updating = $O(1)$ each (linear time)
Total complexity, = $O(\alpha(n)) + \cdots + (O(1) + \cdots) =$
$= O(2\alpha(n)+5) = O(\alpha(n)$
(Ignoring constants)
$= O(\alpha(n))$

Q1→

A1→ Pseudocode:-

Input: n: an integer greater than 0
Output: count: no of quadratic non-residues modulo n
Algorithm: ComputeQuadraticNonresidues(n)
    Initialize empty set 'residues'
    for x from 0 to n-1 do:
        $r = x^2 \bmod n$
        Add r to residues
    end
    count = 0
    for r from 0 to n-1 do:-
        if r isnot in residues then:
            count += 1
        end
    end
    return count.

For datastructure, we are using 'set' because time complexity is O(1) for insertion and updation.
For given algorithm, time complexity is O(n).