ANS 1-

Explanation:

1. We initialize an array quad_residue to keep track of which numbers between 0 and n−1 are quadratic residues.
2. We loop through every number x between 0 and n−1, compute x^2 mod n, and mark the result in the quad_residue array.
3. After processing all x, the unmarked indices in the quad_residue array correspond to the quadratic nonresidues.
4. We count these unmarked values and return the total number of quadratic nonresidues.

Algorithm: CountQuadraticNonresidues

Input: n (modulus)

Output: count of quadratic nonresidues mod n

1. Initialize an array quad_residue of size n with False(It is false by default)
2. For x = 0 to n-1:
   a. Compute r = (x^2) mod n
   b. Set quad_residue[r] = True // r will give true to the value in array
3. Initialize count = 0
4. For r = 0 to n-1:
   a. If quad_residue[r] == False:
      i. Increment count
5. Return count

Boolean Array is the Data Structure (is_residue): This is an efficient choice since we only need to track whether each number r is a quadratic residue or not. A boolean array allows for constant-time updates and lookups.

Time complexity would be O(n) which is linear.

ANS 2-

2a)

Algorithm: Union-Find.Initialize

Input: n (size of the Union-Find to initialize)

Output: Union-Find data structure with min array

1. uf = Array(n)
2. Initialize uf to [0, 1, 2, ..., n-1] (each element is its own parent initially)
3. size = Array(n)
4. Initialize size to 1 (each element is its own set initially)
5. min = Array(n)
6. Initialize min to [0, 1, 2, ..., n-1] (each element is the minimum in its own set)
7. Return (uf, size, min)

Explanation:

1. **LINE 2**: We initialize uf (the parent array) such that every element is its own parent. This means each element starts as the root of its own set.
2. **LINE 4**: We initialize the size array such that the size of each set is 1. This indicates that each element is its own set (of size 1).
3. **LINE 6**: We initialize the min[ ] array such that each element is the minimum of its own set. Since each element is its own set initially, the minimum element of the set is just the element itself.
4. This min[ ] array will allow us to track the minimum element of each set throughout the Union-Find operations.
5. When we perform a union of two sets, we will need to update the min[] array by setting the root of the newly merged set to have the minimum value of the two original sets.

2b)

Worst-case Time Complexity for Initialize

1. Time Complexity:
    - The initialization of uf, size, and min takes O(n),O(n),and O(n), since we are initializing arrays of size n. Therefore, the worst-case time complexity of the initialization is O(n).

2c)

Algorithm: Union

Input: (uf, size, min): the Union-Find structure to modify

Input: a, b (indexes of elements to union)

1. ra = uf.Find(a)

2. rb = uf.Find(b)

3. if ra == rb:

    return (already in the same set)

4. if size[ra] > size[rb]:

    Swap ra and rb (to ensure smaller tree is merged into larger one)

5. uf[ra] = rb (merge tree of ra into rb)

6. size[rb] = size[ra] + size[rb] (update size of rb's tree)

7. min[rb] = min(min[ra], min[rb]) (update minimum in new root)

8. Return modified (uf, size, min)

Explanation: When merging two trees, we update the size of the new root and set the minimum value in the new root to be the smaller of the two original roots.

2d) Amortized Time Complexity of Union

- **Amortized Time Complexity**: The amortized time complexity of the Union operation is O(alpha(n)), where alpha(n) is the inverse Ackermann function. This comes from the fact that we are using the optimized Union-Find structure, and the Union operation only adds constant work for updating the min array.

ANS 3-

```
   Input: data: set of data points
   Input: n: size of data
   Input: distance: distance function that takes two data points and
          returns a nonnegative real number
   Input: c: desired number of clusters; must be an integer between 1 and
          n
   Output: single-linkage hierarchical clusters for data
 1 Algorithm: SingleHClust
 2 heap = MinHeap()
 3 for i = 1 to n − 1 do
 4 |   for j = i + 1 to n do
 5 |   |   Insert distance(data[i], data[j]) into heap, along with the
   |   |    corresponding indexes i and j
 6 |   end
 7 end
 8 uf = UnionFind(n)
 9 count = n
10 while count > c do
11 |   (i, j, dist) = heap.DeleteMin()
12 |   if uf.Find(i) ≠ uf.Find(j) then
13 |   |   uf.Union(i, j)
14 |   |   count = count − 1
15 |   end
16 end
17 return uf
```

Lines 1-2:

- A MinHeap is initialized in O(1) time.

Lines 3-7:

- There is a double loop over all pairs of data points i and j.
- this step involves inserting O(n^2) elements into the heap.
- **Distance** takes Theta(1), as given in the question.

- Heap insertion takes O(log k), where k is the number of elements in the heap at that time. The heap will eventually contain O(n^2) elements, so each insertion takes O(log n^2)=O(log n).
- Time complexity for this step: O(n^2 log n).

Line 8:

- **Time complexity**: O(n).

Lines 9-16:

- The loop runs O(n−c) = O(n) times since c is usually much smaller than n.
- heap deletion takes O(log n).
- Total complexity for the main loop: O(n logn).

Final Time Complexity:

- O(n^2 log n)
- O(n)
- O(n logn)

Therefore, worst-case time complexity of the hierarchical clustering algorithm is

O (n^2 logn)