1.0a:**Space complexity** refers to the total amount of memory space required by an algorithm to complete its execution. It quantifies the memory consumption as a function of the input size.

The components used to compute space complexity include:

1. **Fixed space:** This includes space for instruction storage, simple variables, constants, and other space that doesn't depend on the input size.

2.**Variable space:** The space that depends on the input size, including dynamically allocated memory, recursion stack space, and data structures whose size changes with input.

3. **Return space:** The space required to store the return values and parameters passed during function calls.

1.0b:The Find operation in Disjoint sets is used to determine which set a particular element belongs to. Here's the implementation with path compression:

```
FIND(x):

    if parent[x] ≠ x

        parent[x] = FIND(parent[x]) // Path compression

    return parent[x]
```

Path compression is an optimization technique that flattens the tree structure when Find operations are performed. When we locate the root of the tree, we update the parent pointers of all traversed nodes to point directly to the root. This significantly improves the average time complexity of subsequent operations by reducing the tree height.

1.0c:**Divide and Conquer** is an algorithmic strategy that breaks down a problem into smaller, more manageable subproblems of the same type, solves these subproblems independently, and then combines their solutions to solve the original problem.

The key steps in the Divide and Conquer approach are:

1. Divide: Break the original problem into smaller subproblems.

2. Conquer: Recursively solve these subproblems.

3. Combine: Merge the solutions of the subproblems to form the solution of the original problem.

2.06

Disjoint Sets Union and Find Operations with Performance Improvements

Disjoint sets are a data structure that maintains a collection of disjoint (non-overlapping) sets, each with a representative element. They efficiently support two primary operations: Union and Find.

1. Basic Operations:

Find Operation:

The Find operation determines which set an element belongs to by identifying the representative element (root) of its set.

```
FIND(x):
    if parent[x] ≠ x
        return FIND(parent[x])
    return x
```

Union Operation:

The Union operation merges two sets by connecting the root of one set to the root of another.

```
UNION(x, y):
    root_x = FIND(x)
```

```
root_y = FIND(y)
    if root_x ≠ root_y
        parent[root_x] = root_y
```

2. Performance Improvements:

To enhance the efficiency of these operations, two key optimizations are employed:

a) Path Compression:

During the Find operation, we update the parent of each visited node to point directly to the root, flattening the tree structure:

```
FIND-WITH-PATH-COMPRESSION(x):
    if parent[x] ≠ x
        parent[x] = FIND(parent[x]) // Path compression
    return parent[x]
```

b) Union by Rank/Size:

When performing Union operations, we attach the smaller tree under the larger one to minimize the resulting tree height:

```
UNION-BY-RANK(x, y):
    root_x = FIND(x)
    root_y = FIND(y)
    if root_x = root_y
        return
    if rank[root_x] < rank[root_y]
        parent[root_x] = root_y
    else if rank[root_x] > rank[root_y]
```

```
  parent[root_y] = root_x
   else
         parent[root_y] = root_x
         rank[root_x] = rank[root_x] + 1
```

The combination of these optimizations results in a highly efficient data structure with near-constant amortized time complexity. Specifically, for n elements with m operations, the time complexity approaches $O(m\ \alpha(n))$, where $\alpha(n)$ is the inverse Ackermann function, which grows extremely slowly and is practically constant for all realistic values of n.