You have `n` boxes labeled from `0` to `n - 1`. You are given four arrays: `status`, `candies`, `keys`, and `containedBoxes` where:

- `status[i]` is `1` if the `i^th` box is open and `0` if the `i^th` box is closed,
- `candies[i]` is the number of candies in the `i^th` box,
- `keys[i]` is a list of the labels of the boxes you can open after opening the `i^th` box.
- `containedBoxes[i]` is a list of the boxes you found inside the `i^th` box.

You are given an integer array `initialBoxes` that contains the labels of the boxes you initially have. You can take all the candies in **any open box** and you can use the keys in it to open new boxes and you also can use the boxes you find in it.

Return *the maximum number of candies you can get following the rules above.*

open/closed

off box 1 2 3

0th box contains no keys. 1 no keys.

Example 1:

```
Input: status = [1,0,1,0], candies = [7,5,4,100], keys = [[],[],[1],[]],
containedBoxes = [[1,2],[3],[],[]], initialBoxes = [0]
Output: 16
Explanation: You will be initially given box 0. You will find 7 candies in it and
boxes 1 and 2.
Box 1 is closed and you do not have a key for it so you will open box 2. You will find
4 candies and a key to box 1 in box 2.
In box 1, you will find 5 candies and box 3 but you will not find a key to box 3 so
box 3 will remain closed.
Total number of candies collected = 7 + 4 + 5 = 16 candy.
```

2nd box contains key of 1st box

Example 2:

```
Input: status = [1,0,0,0,0,0], candies = [1,1,1,1,1,1], keys = [[1,2,3,4,5],[],[],[],
[],[]], containedBoxes = [[1,2,3,4,5],[],[],[],[],[]], initialBoxes = [0]
Output: 6
Explanation: You have initially box 0. Opening it you can find boxes 1,2,3,4 and 5 and
their keys.
The total number of candies will be 6.
```

box no → 0 1 2

keys = [[ ], [ ], [1]]

each element tells what keys (which box's key) they contain & each element's index represent the box no.

Ex: box 2 contains box 1's key.

Thought Process

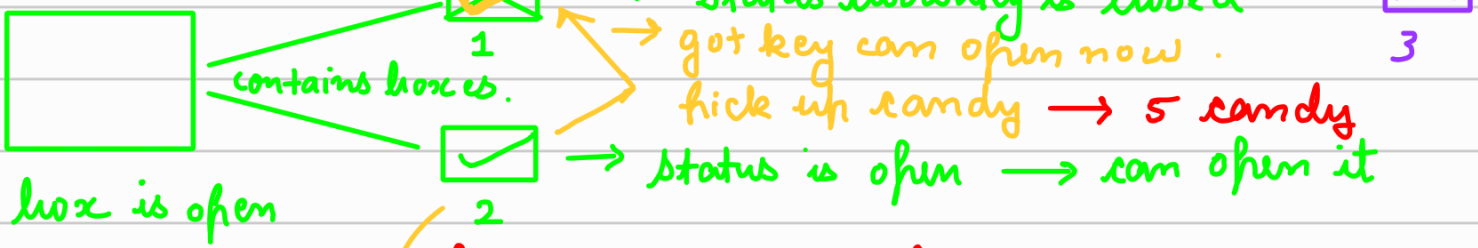
Status = [1, 0, 1, 0] (indices 0 1 2 3)

Candies = [7, 5, 4, 100] (indices 0 1 2 3)

Keys = [( ), ( ), (1), ( )] (indices 0 1 2 3)

Contained Boxes = [(1,2), (3), ( ), ( )] (indices 0 1 2 3)

initial Boxes = [0]



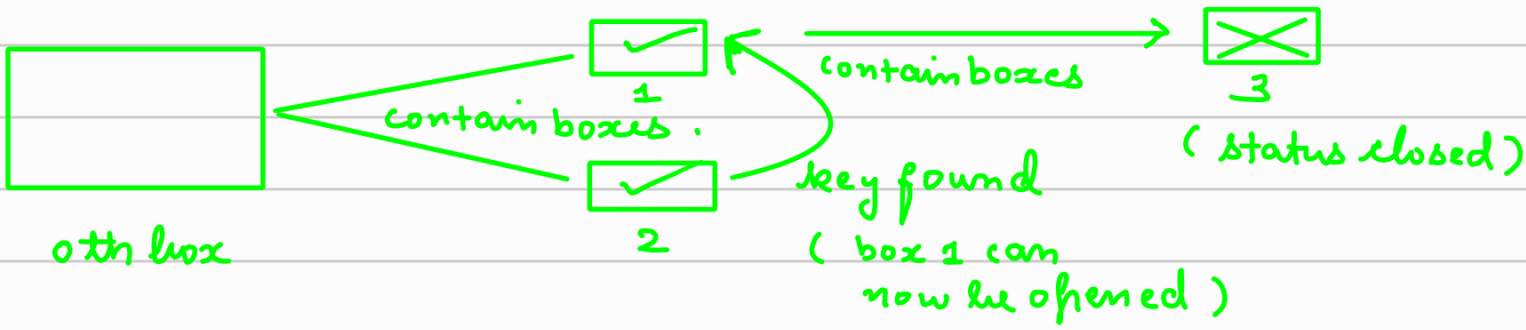
open it & take candies.
4 candies.

box 2 contains no boxes.
but contains key to box 1.

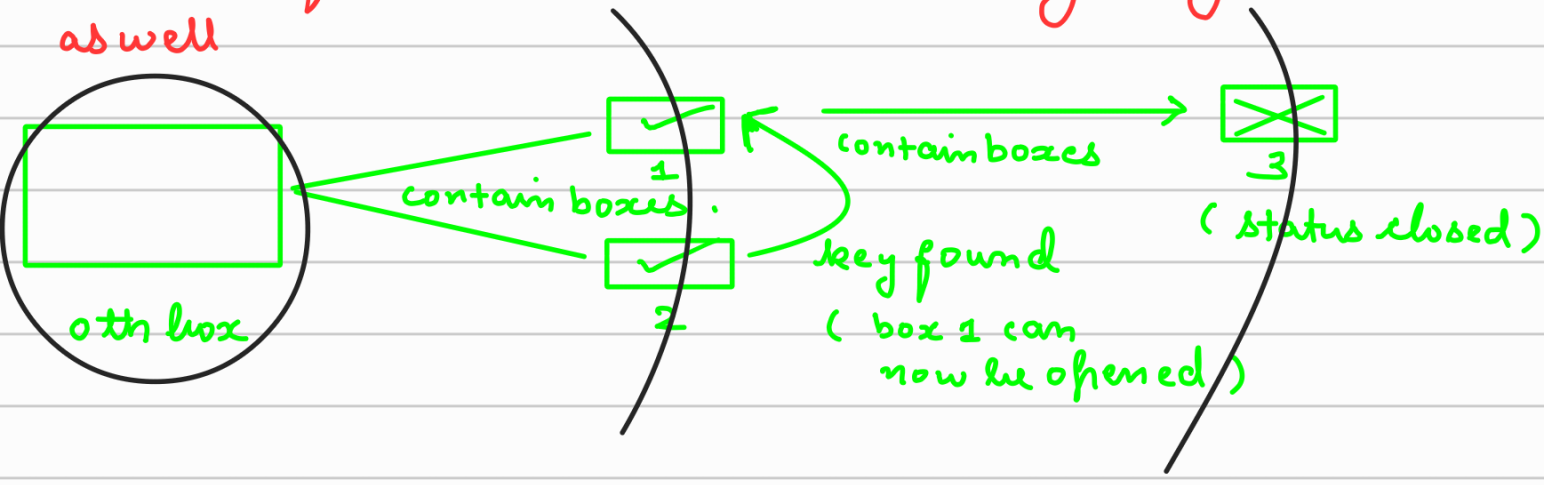
∴ can box 1 with key.

7 + 4 + 5 = 16.

box 1 contains box 3 inside
box 3 status → closed.
no key found for box 3
∴ box 3 cannot be opened.



This diagram looks familiar to graph traversal.
dfs traversal → we are going int dipth
(of boxes)
can be bfs traversal → Exploring neighbours.
as well



levels. Bfs.

Another Example :

status = [1, 1, 1] (0, 1, 2)

candies = [100, 1, 100] (0, 1, 2)

Keys = [( ), (0, 2), ( )] (0, 1, 2)

contained Boxes = [( ), ( ), ( )] (0, 1, 2)

initial Boxes = [1]

Initially box 1, candies = 1.

box 1.

no contained boxes inside box 1. ✗

[status: box 1 is open.]

1 candy

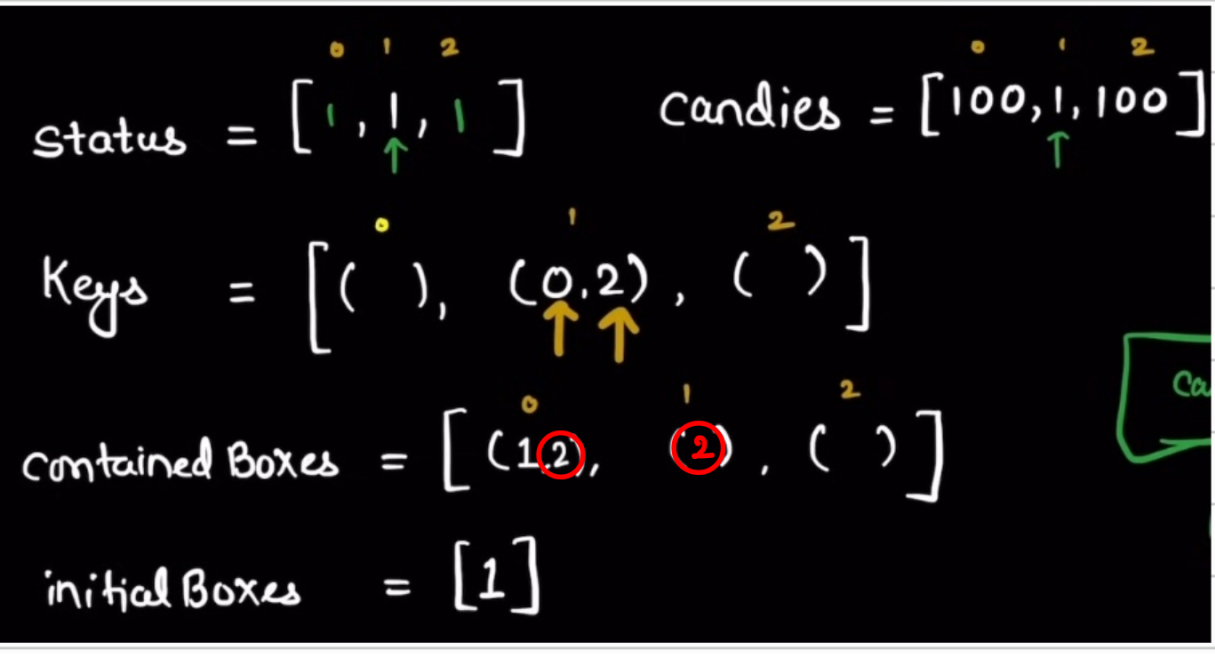
even though box 1 contains key of boxes 0 & 2.
(so if it was closed we could have opened it)
but there is no way to reach those boxes (as box 1 contains no boxes inside it)

∴ candies = 1

Observation: Even though found keys to box 0 & 2
But no way to reach them (box 0 & 2)
∴ cannot open them (box 0 & 2)

so we will apply dfs from initial box given,

edge case



in box 0, we got box 1 & box 2
in box 1, we got box 2 again.

so we won't visit box 2 again at box 1
(as we visited box already at box 2)

∴ no duplicates → don't visit visited box again
as candies will be taken from it during the first visit
∴ no point in visiting again (no candies will be present to collect in second visit)

∴ keep a visited array → for visited boxes.

DFS

```
for ( int &box : initialBoxes ) {
        candies += dfs ( box ) ;
}
```

```
int dfs ( box ) {
if ( visited [ box ] == True ) {
        return 0 ;
}
if ( status [ box ] == 0 ) {          // check status of box
        foundBoxes . insert ( box )      if closed won't get
        return 0 ;                       anything .
}                                        ∴ return 0 ;
visited [ Box ] = True ; // mark
candies += candies [ Box ]   box as visited
                             // add candies of current box .
```

→ currently box is closed in future we might get key & status will be open

∴ keep a tally of found boxes → take a ds which stores unique values. (no duplicates)

∴ unordered_set .

// keeping a tally of boxes

found, so if found a key in future we might open it.

// from current box which other boxes we can visit ?
(current box contains which other boxes)
(can be found from contained Boxes array)

```
for ( int insideBox : containedBoxes [box] ) {

    candies += dfs ( insideBox ) // apply dfs

}
```

// also what keys we found inside our current box.

```
for ( int boxKey : Keys [ box ] ) {

    status [ boxkey ] = 1 ;   // whichever box's key we find, change its status to 1 (can be opened).

// can only be opened if key found & also that box was found as well (from inside some other box)
    if ( foundBoxes.count ( boxKey ) ) {

        candies += dfs( boxkey ) ; // apply dfs on boxes whose keys are found & the boxes themselves are found too.

    }

}
```

return candies ; // return total candies .

DFS CODE

```cpp
//Approach-1 (Using DFS)
//T.C : O(n), where n =  number of boxes, we don't visit any box more than once
//S.C : O(n) ; visited & foundBoxes array .
class Solution {
public:

    int dfs(int box, vector<int>& status, vector<int>& candies, vector<vector<int>>& keys,
vector<vector<int>>& containedBoxes, unordered_set<int>& visited, unordered_set<int>& foundBoxes) {

        if(visited.count(box)) {
            return 0;
        }

        if(status[box] == 0) {
            foundBoxes.insert(box);
            return 0;
        }

        visited.insert(box);
        int candiesCollected = candies[box];


        for(int &insideBox : containedBoxes[box]) {
            candiesCollected += dfs(insideBox, status, candies, keys, containedBoxes, visited,
foundBoxes);
        }

        for(int &boxKey : keys[box]) {
            status[boxKey] = 1; //can be opened
            if(foundBoxes.count(boxKey)) {
                candiesCollected += dfs(boxKey, status, candies, keys, containedBoxes, visited,
foundBoxes);
            }
        }

        return candiesCollected;

    }

    int maxCandies(vector<int>& status, vector<int>& candies, vector<vector<int>>& keys,
vector<vector<int>>& containedBoxes, vector<int>& initialBoxes) {

        int candiesCollected = 0;
        unordered_set<int> visited;
        unordered_set<int> foundBoxes;
        //T.C : O(n) visiting all box only once , n = number of boxes
        //S.C : O(n)
        for(int &box : initialBoxes) {
            candiesCollected += dfs(box, status, candies, keys, containedBoxes, visited, foundBoxes);
        }

        return candiesCollected;
    }
};
```

BFS CODE

```cpp
//Approach-2 (Using BFS)
//T.C : O(n), where n =  number of boxes, we don't visit any box more than once
//S.C : O(n)
class Solution {
public:
    int maxCandies(vector<int>& status, vector<int>& candies, vector<vector<int>>& keys,
vector<vector<int>>& containedBoxes, vector<int>& initialBoxes) {

        int candiesCollected = 0;
        unordered_set<int> visited;
        unordered_set<int> foundBoxes;
        //T.C : O(n) visiting all box only once , n = number of boxes
        //S.C : O(n)

        queue<int> que; //insert those which you have now and you can open it
        for(int &box : initialBoxes) {
            foundBoxes.insert(box);
            if(status[box] == 1) {
                que.push(box);
                visited.insert(box);
                candiesCollected += candies[box];
            }
        }

        while(!que.empty()) {
            int box = que.front();
            que.pop();

            for(int &insideBox : containedBoxes[box]) {
                foundBoxes.insert(insideBox);
                if(status[insideBox] == 1 && !visited.count(insideBox)) {
                    que.push(insideBox);
                    visited.insert(insideBox);
                    candiesCollected += candies[insideBox];
                }
            }

            for(int &boxKey : keys[box]) {
                status[boxKey] = 1; //can be opened in future if we reach this box
                if(foundBoxes.count(boxKey) && !visited.count(boxKey)) {
                    que.push(boxKey);
                    visited.insert(boxKey);
                    candiesCollected += candies[boxKey];
                }
            }
        }

        return candiesCollected;
    }
};
```