Stack_2

@kbbhatt04

@September 18, 2023

Maximal Rectangle

- Approach
  - Optimal
    - For each element of a row, we add the element above it to it if it is 1 else set the element to its current value
    - Then for each row, calculate largest rectangle in histogram
    - Time Complexity: $O(n^2)$
    - Space Complexity: $O(n)$

```
# Python3
# Optimal Solution
class Solution:
    def max_area_histogram(self, heights):
        n = len(heights)
        ans = 0
        stack = []

        for i in range(n+1):
            while stack and (i == n or heights[stack[-1]] >= hei
                height = heights[stack.pop()]
                width = i if not stack else i - stack[-1] -1
                ans = max(ans, height * width)
            stack.append(i)
        return ans

    def maximalRectangle(self, matrix: List[List[str]]) -> int:
```

```
        m, n = len(matrix), len(matrix[0])
        cur_row = [int(i) for i in matrix[0]]
        ans = self.max_area_histogram(cur_row)

        for i in range(1, m):
            for j in range(n):
                if matrix[i][j] == "0":
                    cur_row[j] = 0
                else:
                    cur_row[j] += 1

            ans = max(ans, self.max_area_histogram(cur_row))
        return ans
```

```
// C++
// Optimal Solution
int maximalRectangle(vector<vector<char> > &matrix) {
    if(matrix.empty()){
        return 0;
    }
    int maxRec = 0;
    vector<int> height(matrix[0].size(), 0);
    for(int i = 0; i < matrix.size(); i++){
        for(int j = 0; j < matrix[0].size(); j++){
            if(matrix[i][j] == '0'){
                height[j] = 0;
            }
            else{
                height[j]++;
            }
        }
        maxRec = max(maxRec, largestRectangleArea(height));
    }
    return maxRec;
}
```

```
int largestRectangleArea(vector<int> &height) {
    stack<int> s;
    height.push_back(0);
    int maxSize = 0;
    for(int i = 0; i < height.size(); i++){
        if(s.empty() || height[i] >= height[s.top()]){
            s.push(i);
        }
        else{
            int temp = height[s.top()];
            s.pop();
            maxSize = max(maxSize, temp * (s.empty() ? i : i - 1
            i--;
        }
    }
    return maxSize;
}
```

Online Stock Span

Design an algorithm that collects daily price quotes for some stock and returns **the span** of that stock's price for the current day.

The **span** of the stock's price in one day is the maximum number of consecutive days (starting from that day and going backward) for which the stock price was less than or equal to the price of that day.

- Approach
  - Brute-force
    - Using two nested loops, count for how many days was the current stock price ≥ stock price before that day consecutively
    - Time Complexity: $O(n^2)$
    - Space Complexity: $O(1)$
  - Optimal

- Push every pair of `<price, result>` to a stack
- Pop lower price from the stack and accumulate the count
- Time Complexity: $O(n)$
- Space Complexity: $O(n)$

```
# Python3
# Brute-force Solution
class StockSpanner:
    def __init__(self):
        self.arr = []

    def next(self, price):
        self.arr.append(price)
        count = 0
        for i in range(len(self.arr)-1, -1, -1):
            if self.arr[i] <= price:
                count += 1
            else:
                return count
        return count
```

```
# Python3
# Optimal Solution
class StockSpanner:
    def __init__(self):
        self.stack = []

    def next(self, price):
        res = 1
        while self.stack and self.stack[-1][0] <= price:
            res += self.stack.pop()[1]
        self.stack.append([price, res])
        return res
```

```cpp
// C++
// Optimal Solution
class StockSpanner {
public:
    StockSpanner() {}

    stack<pair<int, int>> s;
    int next(int price) {
        int res = 1;
        while (!s.empty() && s.top().first <= price) {
            res += s.top().second;
            s.pop();
        }
        s.push({price, res});
        return res;
    }
};
```

The Celebrity Problem

There are `N` people at a party. Each person has been assigned a unique id between `0 to N-1` (both inclusive). A celebrity is a person who is known to everyone but does not know anyone at the party.

Given a helper function `knows(A, B)`, It will return `true` if the person having id `A` know the person having id `B` in the party, `false` otherwise. Your task is to find out the celebrity at the party. Print the id of the celebrity, if there is no celebrity at the party then print -1.

- Approach
  - Brute-force
    - Maintain 2 arrays, in_degree and out_degree for all person
    - Check for person having `N-1` in_degrees and `0` out_degrees
    - Time Complexity: $O(n^2)$

        - Space Complexity: $O(2 * n)$
    - Space Optimized Brute-force
        - Check for every person, whether he/she is celebrity or not
        - Time Complexity: $O(n^2)$
        - Space Complexity: $O(1)$
    - Better
        - If for any pair (i, j) such that `i != j`, if `knows(i, j)` returns true, then it implies that the person having id `i` cannot be a celebrity as it knows the person having id `j`. Similarly if `knows(i, j)` returns false, then it implies that the person having id `j` cannot be a celebrity as it is not known by a person having id `i`. We can use this observation to solve this problem
        - Create a stack and push all ids in it
        - While there are more than one element in the stack
            - Pop two elements from the stack, let these elements be `id1` and `id2`
            - If the person with `id1` knows the person with `id2` i.e `knows(id1, id2)` return true, then the person with `id1` cannot be a celebrity, so push `id2` in the stack
            - Otherwise, if the person with `id1` doesn't know the person with `id2` i.e `knows(id1, id2)` return false, then the person with `id2` cannot be a celebrity, so push `id1` in the stack
        - Only one id remains in the stack, you need to check whether the person having this id is a celebrity or not, this can be done by running two loops. One checks whether this person is known to everyone or not and another loop will check whether this person knows anyone or not
        - Time Complexity: $O(n)$
        - Space Complexity: $O(n)$

- Optimal
  - If for any pair (i, j) such that `i != j`, If `knows(i, j)` returns true, then it implies that the person having id `i` cannot be a celebrity as it knows the person having id `j`. Similarly if `knows(i, j)` returns false, then it implies that the person having id `j` cannot be a celebrity as it is not known by a person having id `i`. So, the Two Pointer approach can be used where two pointers can be assigned, one at the start and the other at the end of the elements to be checked, and the search space can be reduced
  - Initialize `P = 0` and `Q = N - 1`
  - while P < Q and in each iteration do the following
    - If `knows(P, Q)` returns true, then increment P by 1
    - If `knows(P, Q)` returns false, then decrement Q by 1
  - Check whether the person having id P is a celebrity or not, this can be done by running two loops. One checks whether this person is known to everyone or not and another loop will check whether this person knows anyone or not
  - Time Complexity: $O(n)$
  - Space Complexity: $O(1)$

```
# Python3
# Brute-force Solution
def findCelebrity(n, knows):
    #  Calculating indegree and outdegree of each nodes.
    indegree = [0] * n
    outdegree = [0] *  n

    for i in range(0, n):
        for j in range(0, n):
            if knows(i, j) == 1:
                indegree[j] += 1
                outdegree[i] += 1
```

```
    # Finding Celebrity
    celebrity = -1
    for i in range(0, n):
        if indegree[i] == n - 1 and outdegree[i] == 0:
            celebrity = i
            break

    return celebrity
```

```
# Python3
# Space Optimized Brute-force Solution
def findCelebrity(n, knows):
    celebrity = -1
    for i in range(0, n):
        knowAny = False
        knownToAll = True

        # Check whether perosn with id 'i' knows any other perso
        for j in range(0, n):
            if knows(i, j) == True:
                knowAny = True
                break

        # Check whether perosn with id 'i' is known to all the o
        for j in range(0, n):
            if i != j and knows(j, i) == False:
                knownToAll = False
                break

        if knowAny == False and knownToAll != False:
            celebrity = i
            break
```

```
    return celebrity
```

```
# Python3
# Better Solution
def findCelebrity(n, knows):
    # Create a stack and push all ids in it.
    ids = []
    for i in range(0, n):
        ids.append(i)
    #  Finding celebrity.
    while(len(ids) > 1):
        id1 = ids.pop()
        id2 = ids.pop()
        if knows(id1, id2) == True:
            # Because person with id1 can not be celebrity.
            ids.append(id2)
        else:
            # Because person with id2 can not be celebrity.
            ids.append(id1)

    celebrity = ids[len(ids)-1]

    knowAny = False
    knownToAll = True
    # Verify whether the celebrity knows any other person.
    for i in range(0, n):
        if knows(celebrity, i) == True:
            knowAny = True
            break

    # Verify whether the celebrity is known to all the other per
    for i in range(0, n):
        if i != celebrity and knows(i, celebrity) == False:
            knownToAll = False
```

```
            break

    if knowAny != False or knownToAll == False:
        # If verificatin failed, then it means there is no celeb
        celebrity = -1

    return celebrity
```

```
# Python3
# Optimal Solution
def findCelebrity(n, knows):
    # Two pointers pointing at start and end of search space.
    p = 0
    q = n-1
    while(p < q):
        # This means p cannot be celebrity.
        if knows(p, q) == True:
            p +=1
        # This means q cannot be celebrity.
        else:
            q -= 1

    celebrity = p
    knowAny = False
    knownToAll = True
    # Verify whether the celebrity knows any other person.
    for i in range(0, n):
        if knows(celebrity, i) == True:
            knowAny = True
            break
    # Verify whether the celebrity is known to all the other per
    for i in range(0, n):
        if i != celebrity and knows(i, celebrity) == False:
            knownToAll = False
            break
```

```
    if knowAny != False or knownToAll == False:
        # If verificatin failed, then it means there is no celeb
        celebrity = -1

    return celebrity
```

```
// C++
// Optimal Solution
int findCelebrity(int n) {
    // Two pointers pointing at start and end of search space.
    int p = 0, q = n-1;

    // Finding celebrity.
    while(p < q) {
        if(knows(p, q)) {
            // This means p cannot be celebrity.
            p++;
        }
        else {
            // This means q cannot be celebrity.
            q--;
        }
    }

    int celebrity = p;
    bool knowAny = false, knownToAll = true;

    // Verify whether the celebrity knows any other person.
    for(int i = 0; i < n; i++) {
        if(knows(celebrity, i)) {
            knowAny = true;
            break;
        }
    }
```

```
    // Verify whether the celebrity is known to all the other pe
    for(int i = 0; i < n; i++) {
        if(i != celebrity and !knows(i, celebrity)) {
            knownToAll = false;
            break;
        }
    }

    if(knowAny or !knownToAll) {
        // If verificatin failed, then it means there is no cele
        celebrity = -1;
    }

    return celebrity;
}
```

Template

- Approach
  - Brute-force
    -
    - Time Complexity: $O(n^3)$
    - Space Complexity: $O(1)$
  - Better
    -
    - Time Complexity: $O(n^3)$
    - Space Complexity: $O(1)$
  - Optimal
    -

- Time Complexity: $O(n^3)$
- Space Complexity: $O(1)$

```
# Python3
# Brute-force Solution
```

```
# Python3
# Better Solution
```

```
# Python3
# Optimal Solution
```

```
// C++
// Optimal Solution
```