1381. Design a Stack With Increment Operation

Design a stack that supports increment operations on its elements.

Implement the `CustomStack` class:

- `CustomStack(int maxSize)` Initializes the object with `maxSize` which is the maximum number of elements in the stack.
- `void push(int x)` Adds `x` to the top of the stack if the stack has not reached the `maxSize`.
- `int pop()` Pops and returns the top of the stack or `-1` if the stack is empty.
- `void inc(int k, int val)` Increments the bottom `k` elements of the stack by `val`. If there are less than `k` elements in the stack, increment all the elements in the stack.

Example 1:

```
Input
["CustomStack","push","push","pop","push","push","push","increment","increment","po
p","pop","pop","pop"]
[[3],[1],[2],[],[2],[3],[4],[5,100],[2,100],[],[],[],[]]
Output
[null,null,null,2,null,null,null,null,null,103,202,201,-1]
Explanation
CustomStack stk = new CustomStack(3); // Stack is Empty []
stk.push(1);                          // stack becomes [1]
stk.push(2);                          // stack becomes [1, 2]
stk.pop();                            // return 2 --> Return top of the stack 2,
stack becomes [1]
stk.push(2);                          // stack becomes [1, 2]
stk.push(3);                          // stack becomes [1, 2, 3]
stk.push(4);                          // stack still [1, 2, 3], Do not add another
elements as size is 4
stk.increment(5, 100);                // stack becomes [101, 102, 103]
stk.increment(2, 100);                // stack becomes [201, 202, 103]
stk.pop();                            // return 103 --> Return top of the stack
103, stack becomes [201, 202]
stk.pop();                            // return 202 --> Return top of the stack
202, stack becomes [201]
stk.pop();                            // return 201 --> Return top of the stack
201, stack becomes []
stk.pop();                            // return -1 --> Stack is empty return -1.
```

Constraints:

- `1 <= maxSize, x, k <= 1000`
- `0 <= val <= 100`
- At most `1000` calls will be made to each method of `increment`, `push` and `pop` each separately.

1381. Design a Stack With Increment Operation

```
/*
    Array based approach
    Time complexity: O(n)
    Space complexity: O(n)
    n: size of the array
    m: number of calls of `increment` function
*/
class CustomStack{
    public:
        std::vector<int> st;
        int _capacity;
    public:
        // O(1)
        CustomStack(int maxSize){
            _capacity=maxSize;
        }

        // O(1)
        void push(int x) {
            if(st.size()<_capacity) st.push_back(x);
        }

        // O(1)
        int pop(){
            if(st.empty()) return -1;
            int x=st[st.size()-1];
            st.pop_back();
            return x;
        }

        // O(n)
        void increment(int k, int val) {
            for(int i=0;i<std::min(k,(int)st.size());++i) st[i]+=val;
        }
};
```

1381. Design a Stack With Increment Operation

```
/*
    Array based approach + Lazy propagation
    Time complexity: O(1) , overall Time complexity: O(m)
    Space complexity: O(2n)
    n: size of the array
    m: number of calls of `increment` function
*/
class CustomStack{
    public:
        std::vector<int> st,inc;
        int _capacity;
    public:
        // O(1)
        CustomStack(int maxSize){
           _capacity=maxSize;
        }
        // O(1)
        void push(int x){
            if(st.size()<_capacity){
                st.push_back(x);
                inc.push_back(0);
            }
        }
        // O(1)
        int pop(){
            if(st.empty()) return -1;
            int i=st.size()-1;
            if(i>0) inc[i-1]+=inc[i]; // Lazy propagation
            int x=st[i]+inc[i];
            st.pop_back();
            inc.pop_back();
            return x;
        }
        // O(1)
        void increment(int k, int val){
          int i=std::min(k,(int)st.size())-1;
          if(i>=0) inc[i]+=val;
        }
};
```