26/04/2023

Q1 Insert an element at the bottom of the stack.

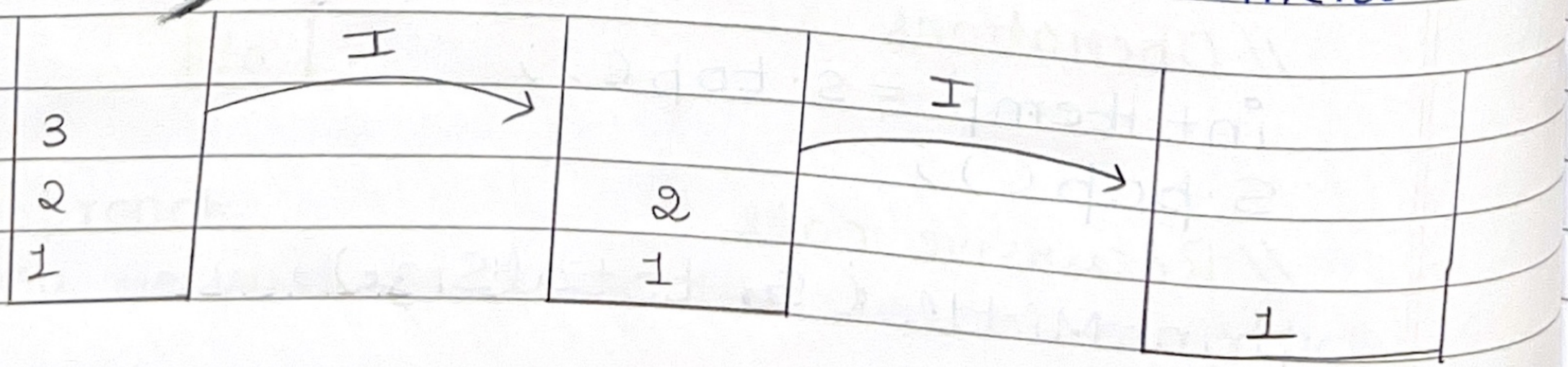
i/p →
5
4
3
2

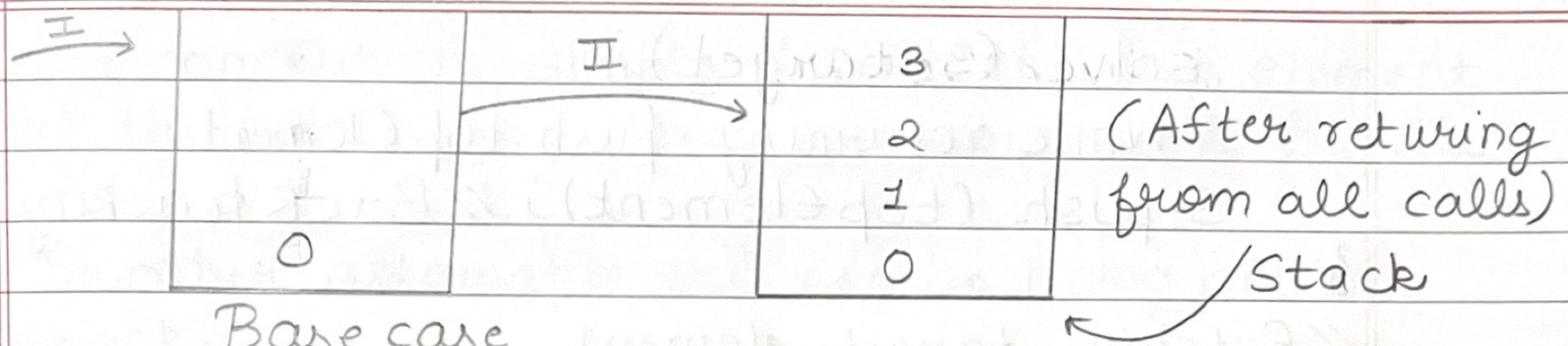
Key = 1/ Top element

o/p →
5
4
3
2
1

4
3
2
5

We will be solving this with the help of recursion and the base case here will be when the stack becomes empty. As we reach the base case, insert the element.





At the base case simply push the key element and return. While returning from each recursive call push the top element at each step.

I ⇒ Store the top element. Pop it and then simply call the function for next element.

II ⇒ Simply push the top element which was stored at the returning side i.e returning from the recursive call.

Note → Also remember when we are trying to access s.top(), just make sure that stack is not empty.

Code

```
void solve (stack<int> &s, int target) {
    // Base case
    if (s.empty()) {
        s.push(target); → Simply push
        return; → return as only single element
                  to be added.
    }
    //Step I
    int topElement = s.top();
    s.pop();
    //Recursive call for next element
```

```
        solve (s, target);
        //While returning push top Element
        s.push (topElement); //Back tracking
}
//Fetching target element
void insertAtBottom (stack<int> & s){
    //Making sure stack is empty or not
    if (s.empty()){
            cout << "Empty Stack";
            return;
    }
    int target = s.top();
    s.pop();
    solve (s, target);
}
```

Insert top element at bottom of stack

Q2 Reverse the stack.

i/p →	50
	40
	30
	20
	10

o/p →	10
	20
	30
	40
	50

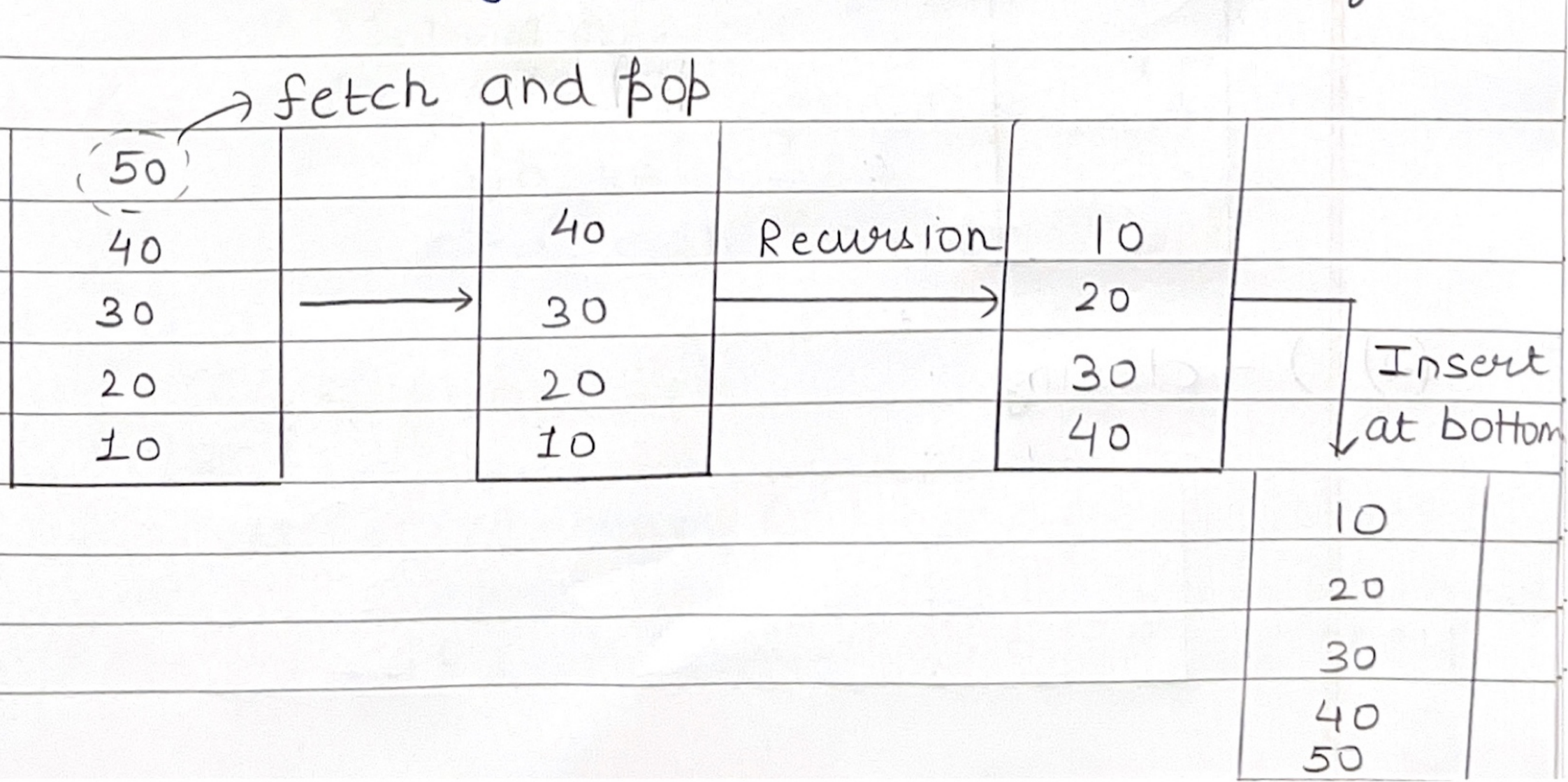
We will be using the concept of recursion i.e solve one case and rest recursion will handle.

The concept is simply insert the top element at the bottom and then recursion will handle.

The other alternative soln can be using another stack..

Code

```
void reverseStack (Stack <int> &s){
    //Base case → Stack empty & return as
    //we can't reverse
    if (s.empty ()){
        return;
    }
    //Handle single case → Fetch top element
    int target = s.top();
    s.pop();
    //Recursive call
    reverseStack (s);
    //After we have received reversed stack
    //simply insert top element at bottom
    solve (s, target); // Same function of Q1
}
```



→ V.V.V. Imp Question

Q3 Valid Paranthesis problem.

i/p → ( ) [ ] { } | ( ]

o/p → true. | false.

Valid paranthesis means

(i) Open bracket must be closed by same type of bracket.

(ii) Open bracket must be closed in correct order

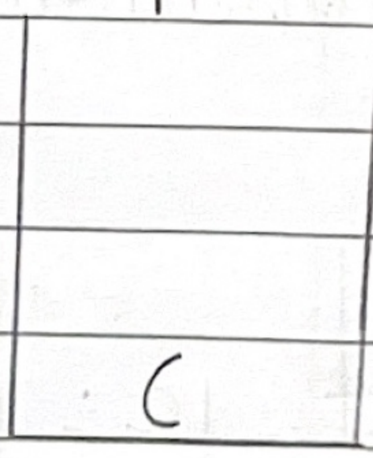
(iii) Every close bracket has corresponding open bracket of same type.

Dry run

i/p → ( ) [ ] { }

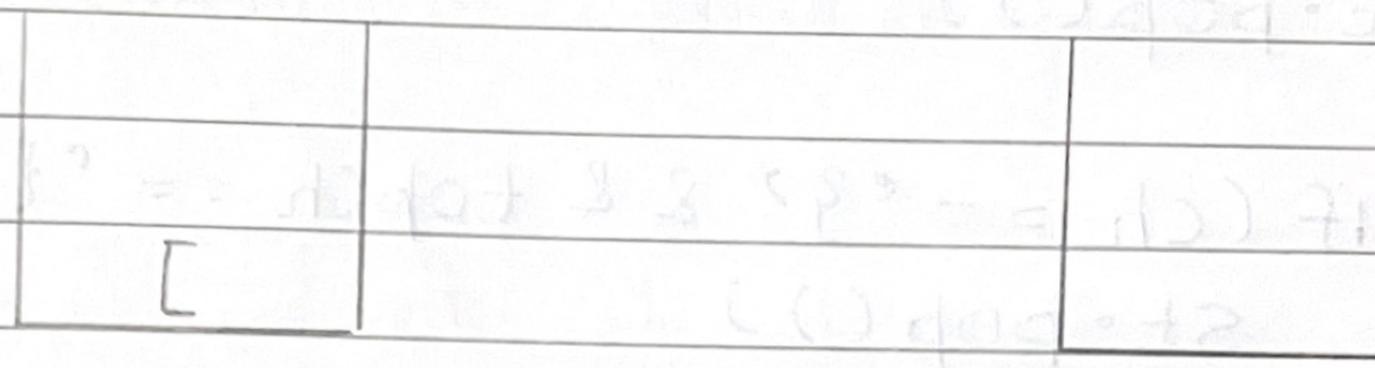
* Whenever we get the opening bracket simply push it in stack.
* Whenever we get the closing bracket, is there any opening bracket. If present then simply pop & if not found then return false.

(1) (→ opening

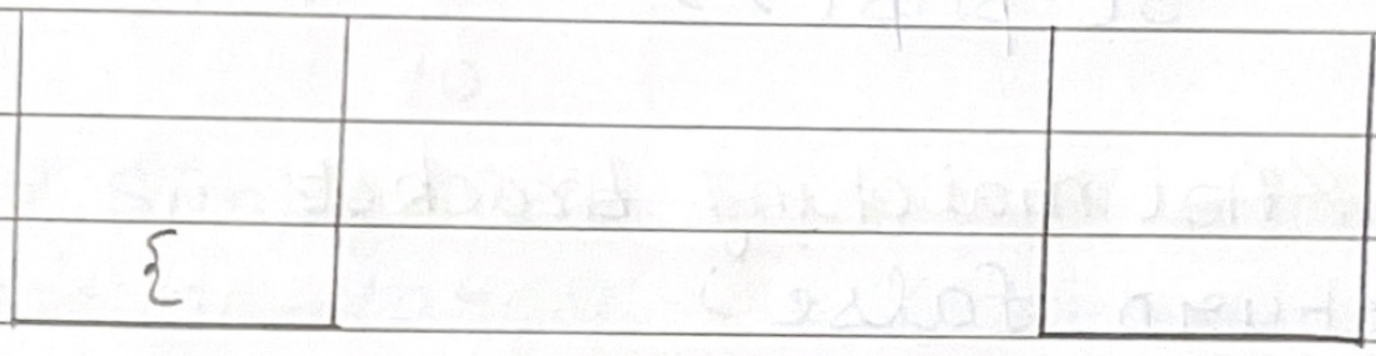


(2)) → closing

(3) [→ Opening (4)] → closing



(5) { → opening (6) } → closing



If after traversing the whole i/p string, the stack is empty then it is a valid paranthesis else not valid.

Code

```
bool isValid (string &s) {
    stack <char> st; //Stack of characters
    //Traverse the string
    for (int i=0; i<s.length(); i++) {
        char ch = s[i]; //Consider character
        //Opening bracket
        if (ch == '{' || ch == '[' ||
                ch == '(') {
            st.push (ch); //Push
    }
    //Closing bracket
    else {
        if (!st.empty()) { // Stack not empty
            char topCh = st.top(); //Fetch top
            if (ch == '}' && topCh == '('){
```

```
        //Matching bracket
            st.pop();
        }
        else if (ch == '}' && top(ch == '{') {
                st.pop();
        }
        else if (ch == ']' && top(ch == '[') {
                st.pop();
        }
        else { //Not matching bracket
            return false;
        }
    }
    else {
        //Stack empty but closing bracket is there.
        return false;
    }
    }
    }
    //Stack empty after traversing whole string
    if (st.empty()) {
            return true; //Valid Paranthesis
    }
    return false;
    }
```

Q4 Sort a stack.

i/p →

9
5
3
11
7

o/p →

3
5
7
9
11

Before solving this question, we can solve insert the element in a sorted stack.

		target = 5
	2	
	4	
	6	
	8	
	10	

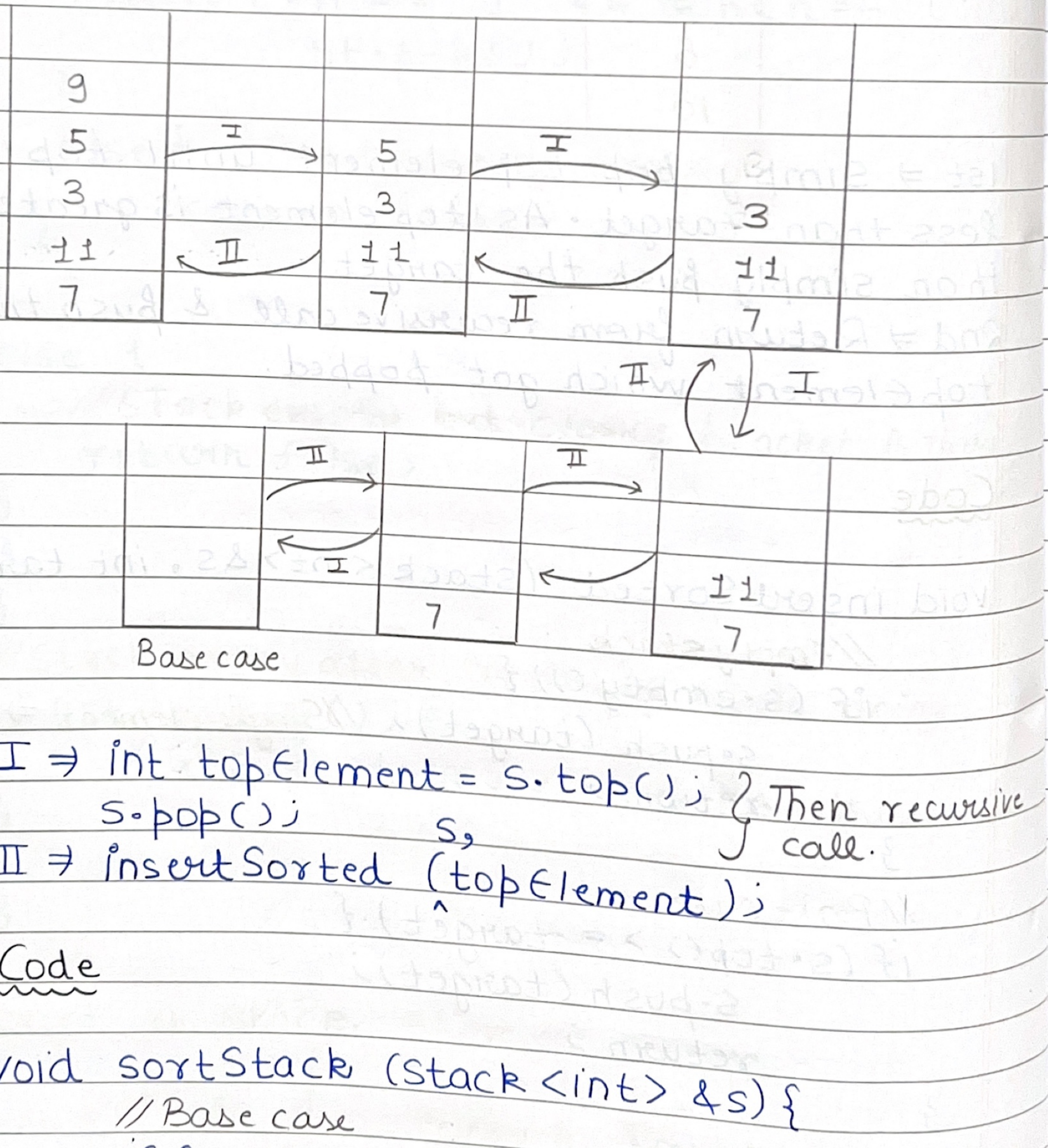
1st ⇒ Simply pop top element until top is less than target. As top element is greater than simply push the target.
2nd ⇒ Return from recursive call & push the topElement which got popped.

Code

```
void insertSorted (stack <int> &s, int target){
    //Empty stack
    if (s.empty()){
        s.push (target); //Single element ⇒ sorted
        return;
    }
    //Base case
    if (s.top() >= target){
        s.push(target);
        return;
    }
    //Normal case
    int topElement = s.top();
    s.pop();
    insertSorted (s, target); //Recursive call
```

```
// Backtracking
s.push(topElement);
}
```

The above code will be used in sorting a stack also.



I ⇒ int topElement = s.top(); } Then recursive call.
s.pop();
II ⇒ insertSorted (s, topElement);

Code

```
void sortStack (stack<int> &s) {
    // Base case
    if (s.empty()) {
        return;
    }
```

```
//Step-I
int topElement = s.top();
s.pop();
//Recursive call
sortStack(s);
// Step-II
insertSorted(s, topElement);
}
```

Q5 Remove redundant brackets.

i/p → ((a+b)) — Redundant brackets
o/p → (a+b) ↳ Useless brackets

We will taking help of operators. If there is an operator in b/w the brackets, then it is not redundant & is a useful bracket else it is a redundant bracket.

Dry run
expression → (a+b)

(I)

(

(II)

(

Don't push the operand

III

+
(

IV

+
(

(V) Now we have encountered the closing bracket we have to pop until we have the opening bracket & when we encounter an operator, simply make flag = true and pop again until we get opening bracket.

Now whole expression has been traversed & hence no redundant brackets is there.

Code

```
int checkRedundancy (string &s) {
    // Creation of stack
    stack <char> st;
    // Traverse the expression
    for (int i=0; i<s.length(); i++){
        // Opening bracket or operator → push
        if (s[i] == '(' || s[i] == '+' || s[i] == '-'
            || s[i] == '*' || s[i] == '/') {
                st.push(s[i]);
        }
        // Closing bracket or character
        else {
            if (s[i] == ')') { // Consider closing only
                // flag → true means redundant
                bool flag = true;
                // Pop until we get opening bracket
                while (!st.empty() && st.top() != '('){
                    // Operator comes → not redundant
                    if (st.top() == '+' || st.top() == '-'
                        || st.top() == '/' || st.top()
                        == '*')
```

```
    { // false → not redundant
      flag = false;
    }
    st.pop();
  }
  // Either opening bracket or empty stack
  if (!st.empty()) {
      st.pop(); // Pop opening bracket
    }
    if (flag == true) { // Redundant bracket is
        return true;    encountered as flag was
      }                 true.
    }
  }
}
return false; //Not redundant
}
```