Day 30 / 100 :

Topic - Binary Tree

1 Problem statement: Pow(x, n) (Medium)

Implement pow(x, n), which calculates x raised to the power n (i.e., xn).

Example 1:
Input: x = 2.00000, n = 10
Output: 1024.00000

Example 2:
Input: x = 2.10000, n = 3
Output: 9.26100

Example 3:
Input: x = 2.00000, n = -2
Output: 0.25000
Explanation: 2-2 = 1/22 = 1/4 = 0.25

Solutions :

Approach 1 - Maths

Explanation:

- Initialize ans as 1.0 and store a duplicate copy of n i.e power_of_x and make it positive.
- In a while loop keep on iterating until power_of_x becomes zero
- Now if power_of_x is an odd power then multiply x with ans ans reduce power_of_x by 1. Else multiply x with itself and divide power_of_x by 2.
- Now the entire binary exponentiation is complete and power_of_x becomes zero
- Check if n is a negative value, return 1/ans else return ans.

Time Complexity : O(logN)

```cpp
class Solution {
public:

    double myPow(double x, int n) {
        double ans = 1.0;
        long long power_of_x = abs(n);

        while(power_of_x>0)
        {
            // if power is odd
            if(power_of_x%2==1)
            {
                ans = ans*x;
                power_of_x = power_of_x - 1;
            }
            // if power is even
            else if(power_of_x%2==0)
            {
                x = x*x;
                power_of_x = power_of_x / 2;
            }
        }

        if(n<0)
            return (double)1/(double)ans;

        return (double)ans;
    }
};
```

Solutions :

Approach 1 - Iterative + Maths

1. The function myPow takes two parameters, x (base) and n (exponent), and returns a double value representing the result of x raised to the power n.
2. It initializes a variable res to 1.0, which will store the final result.
3. The function uses a loop to iteratively calculate the result by reducing the exponent n in each iteration.
4. In each iteration, it checks whether the least significant bit of the current exponent n is set (i.e., whether n is odd) using the bitwise AND operator (i&1). If n is odd, it multiplies the current result res by x.
5. After checking the least significant bit, the function right-shifts the exponent n by one bit (i.e., i/=2) to effectively reduce the exponent by half.
6. In each iteration, it also squares the base x (x*=x) to update the value of x for the next iteration.
7. The loop continues until the exponent n becomes zero.
8. After the loop, the function returns res as the final result if n is non-negative. If n is negative, it returns 1/res as the result, which is equivalent to calculating the reciprocal of res.

```
class Solution {
public:
    double myPow(double x, int n) {
        double res=1.0;
        for(int i=n;i;i/=2){
            if(i&1)res*=x;
            x*=x;

        }
        return n>=0?res:1/res;
    }
};
```

2 Problem statement:[Valid Parenthesis](#) (Easy)

Given a string s containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid.
An input string is valid if:
Open brackets must be closed by the same type of brackets.
Open brackets must be closed in the correct order.
Every close bracket has a corresponding open bracket of the same type.

Example 1:
Input: s = "()"
Output: true

Example 2:
Input: s = "()[]{}"
Output: true

Solutions :

Approach 1 -
Intuition
Input string is valid if -:

- for every opening bracket we have a corresponding closing bracket of the same kind
- opening brackets of a kind comes before the closing
- Order of closing is maintained (inner ones are closed before the outer ones)

Approach:
If we get any type of opening bracket we push it into the stack

If we get a bracket of closing type we compare it with the top element of the stack

```
opening bracket -> push
closing bracket -> compare
```

While comparing with the top of the stack, there can be two cases

1. The stack can be empty
2. The stack can be non-empty

1.If the stack is not empty

Then we compare the top element of the stack

- If the top element and the current element of the string i.e. the closing bracket are of the same type -> pop the stack
- If not of the same type then -> return false

2.If the stack is empty and we get a closing bracket -> means the string is not valid -> return false

In the end after all the iterations

- If the stack is empty i.e., all the opening brackets found their corresponding closing brackets and have been popped
- Then we return true
- Else return false

Complexity:

Time complexity:O(n) - n is the length of the string

Space complexity:O(n) - in the worst case when all the brackets of the string are of opening type our loop will just push all the characters of the string into the stack.

```
class Solution {
public:
    bool isValid(string s) {
        stack<char> st ;
        for (int i = 0 ;  i< s.length() ; i++)
        {
            char ch = s[i];

            if (ch == '(' || ch == '{' || ch == '[')
            {
                st.push(ch) ;  }
```

```
            else {
                if (!st.empty())
                {
                    char top = st.top() ;
                    if ((ch == ')' && top == '(') ||
                        (ch == '}' && top == '{') ||
                        (ch == ']' && top == '['))
                        {
                            // if matches then pop
                            st.pop() ;
                        }
                        else
                        {
                            return false ;
                        }
                }
                else
                {

                    return false ;
                }
            }
        }

        if (st.empty())
        {
            return true ;
        }
        return false ;
    }
};
```