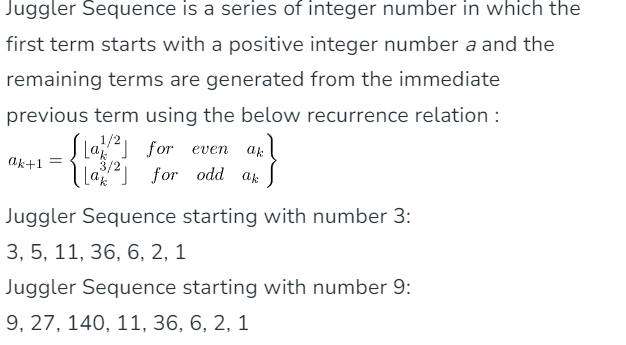
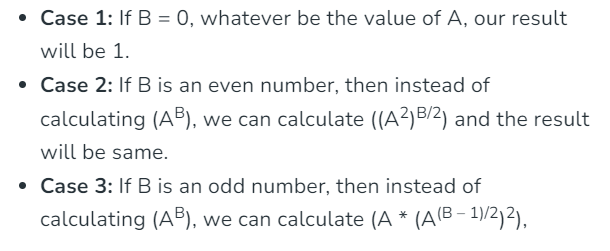
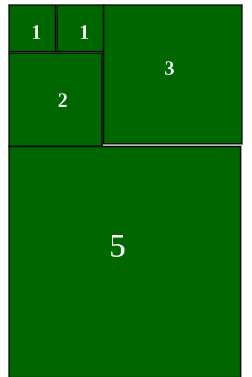
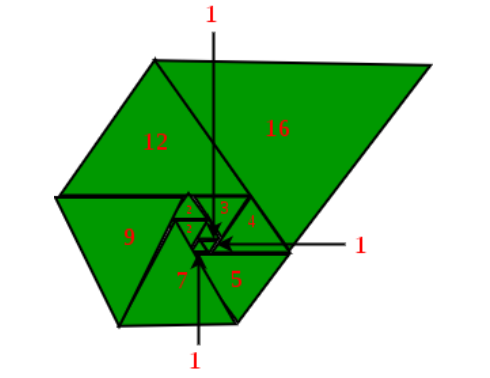
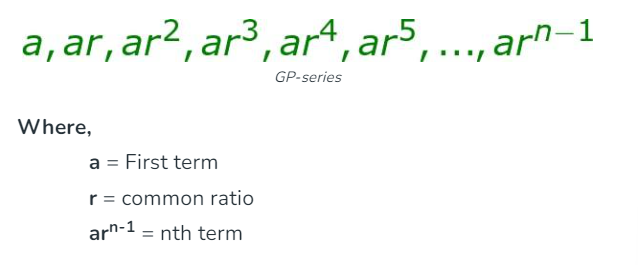
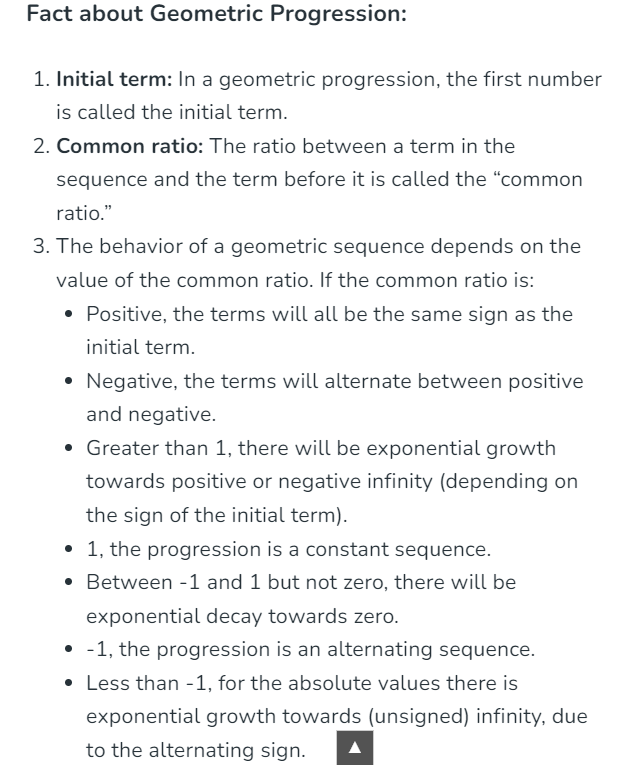
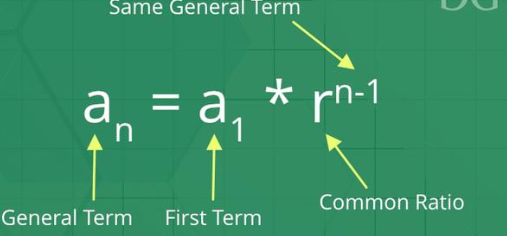
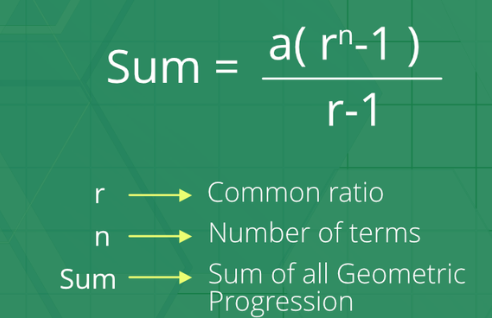
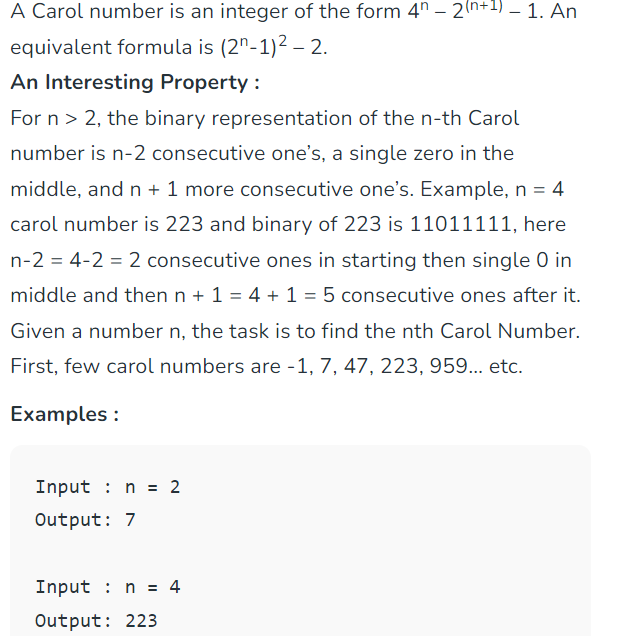
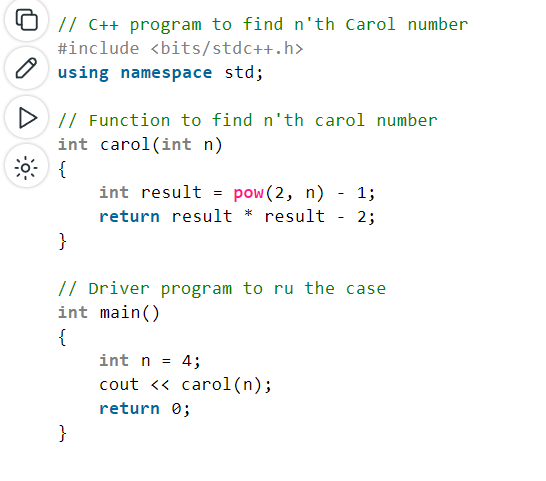
**Series**

1. Juggler Sequence.
   1. 
   2. Given a number n we have to print the juggler Sequence for this number as the first term of the sequence.
      1. Input: 9
      2. Output: 9, 27, 140, 11, 36, 6, 2, 1
      3. We start with 9 and use above formula to get
      4. next terms.
      5. Input: 6
      6. Output: 6, 2, 1
   3. Time complexity: - O(nlogN)
   4. Space complexity: - O (1)
   5. The terms in juggler sequence first increase to a peak value and then start decreasing
   6. The last term in juggler Sequence is always 1.
2. Binary Exponentiation.
   1. Binary Exponentiation or Exponentiation by squaring is the process of calculating a number raised to the power another number (A^B) in Logarithmic time of the exponent or power, which speeds up the execution time of the program.
   2. 
   3. ***EX: 2 12****= (2) 2 \* 6  
      = (4) 6  
      = (4) 2 \* 3  
      = (16) 3  
      = 16 \* (16) 2  
      = 16 \* (256) 1*
   4. **Use Case**
      1. Fast Computation of Nth Fibonacci Number
         * We can compute Nth Fibonacci Number by simply running a loop till N and in every iteration i, we calculate the ith Fibonacci number using (i-1)th and (i-2)th iteration. But this approach runs in linear time complexity, that is O(N). But, if we are concerned with simply the Nth Fibonacci number and not every number before it, then we can compute it in O(logN) by using matrix exponentiation, where we build a **2⨯2 matrix** to transition from (i-2)th and (i-1)th Fibonacci number to (i-1)th and ith Fibonacci number
         * [Click here to expand your knowledge about Matrix Exponentiation](https://www.geeksforgeeks.org/matrix-exponentiation/)
      2. **Compute a large number modulo M:**
         * It is hardly to see any problem which is all about to compute **AB**, but Binary Exponentiation comes in handy when our answer becomes too large to be stored as an integer, that is greater than **INT\_MAX.**There are many problems which asks us to count the number of ways to do something and as the number of ways are too large to be stored in an Integer variable, the question asks us to print the answer modulo **1000000007**or modulo **998244353**. Since, it is proved [here](https://www.geeksforgeeks.org/modular-multiplication/), that
         * ***(A \* B) mod M = ((A mod M) \* (B mod M)) mod M***
      3. Apply Permutation of a give Sequence large number of times.
         * Let’s suppose, we are given a Permutation **P** and a Sequence **S** and we need to apply **P** to **S** for a large number of times (say **K**), then we can easily compute the final sequence by using Binary Exponentiation.
         * ***P****= [2, 3, 4, 1],****S****= [2, 1, 3, 4]  
           After applying permutation P to Sequence S once,****S’****= [1, 3, 4, 2]****Explanation:*** *S'[1] = S[P[1]] = S[2] = 1  
           S'[2] = S[P[2]] = S[3] = 3  
           S'[3] = S[P[3]] = S[4] = 4  
           S'[4] = S[P[4]] = S[1] = 2*
         * *After applying permutation P to Sequence S twice,****S”****= [3, 4, 2, 1]****Explanation:*** *S”[1] = S'[P[1]] = S'[2] = S[P[2]] = S[3] = 3  
           S”[2] = S'[P[2]] = S'[3] = S[P[3]] = S[4] = 4  
           S”[3] = S'[P[3]] = S'[4] = S[P[4]] = S[1] = 2  
           S”[4] = S'[P[4]] = S'[1] = S[P[1]] = S[2] = 1*
         * **Observations:**If we observe carefully, in the above example instead of applying permutation **P** to **S** twice, if we apply permutation **P** in itself (**P’**) and then apply it on **S** once, we will get the same result.
         * *P = [2, 3, 4, 1], S = [2, 1, 3, 4]  
           After applying permutation P to itself once,  
           P’ = [3, 4, 1, 2]****Explanation:*** *P'[1] = P[P[1]] = P[2] = 3  
           P'[2] = P[P[2]] = P[3] = 4  
           P'[3] = P[P[3]] = P[4] = 1  
           P'[4] = P[P[4]] = P[1] = 2*
         * *Now, applying permutation P’ to S,  
           S”[1] = S[P'[1]] = S[3] = 3  
           S”[2] = S[P'[2]] = S[4] = 4  
           S”[3] = S[P'[3]] = S[1] = 2  
           S”[4] = S[P'[4]] = S[2] = 1*
         * Therefore, it is clear that applying a permutation **P** to a sequence **S** for **N** times is equal to applying permutation **P’** to sequence **S** for **N/2** times and we can simply solve this using Binary Exponentiation:
      4. Compute product of 2 very large Numbers modulo M.
         * If we have 2 very large number **A** and **B** and we need no compute **(A \* B)** mod **M** but**(A \* B)** in 64-bit integer, then we can use the concept of binary exponentiation to compute the product of these **2** numbers by adding **A** to the answer **B** times.
         * When we are calculating **(A \* B)**, we can have 3 possible positive values of **B**:
         * **Case 1:**If **B = 0**, whatever be the value of A, our result will be **0**.  
           **Case 2:** If **B**is an even number, then instead of calculating **(A \* B)** mod **M** , we can calculate**(((A \* 2) mod M)** **\* B/2)** mod **M**and the result will be same.  
           **Case 3:** If **B** is an odd number, then instead of calculating**(A \* B)**, we can calculate **(A + A \* (B-1))** mod **M**, which is same as case 2.
         * We can recursively follow the above steps to get our result.
         * ***(25 \* 10) mod 60****= ((25 \* 2) mod 60 \* 5) mod 60  
           = ((50 mod 60) \* 5) mod 60  
           = (50 + (50 mod 60) \* 4) mod 60  
           = (50 + (100 mod 60) \* 2) mod 60  
           = (50 + (40 mod 60) \* 2) mod 60  
           = (50 + (80 mod 60)) mod 60  
           = (50 + 20) mod 60  
           = 10*
         * Therefore, we can compute (A \* B) mod M using Binary Exponentiation:
         * **Brute Force:**O(1), but not possible for large numbers
         * **Binary Exponentiation:**O(log B), as we have distributed the multiplication operation to a series of log(B) addition operations.
3. .Padovan Sequence.
   1. Similar to Fibonacci sequence with similar recursive structure.
   2. P(n) = P(n-2) + P(n-3)
   3. P(0) = P(1) = P(2) = 1
   4. **Fibonacci Sequence:** 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55……   
      Spiral of squares with side lengths which follow the Fibonacci sequence.
   5. 
   6. **Padovan Sequence:** 1, 1, 1, 2, 2, 3, 4, 5, 7, 9, 12, 16, 21, 28, 37,…..   
      Spiral of equilateral triangles with side lengths which follow the Padovan sequence.
   7. 
   8. https://www.geeksforgeeks.org/problems/padovan-sequence2855/1?itm\_source=geeksforgeeks&itm\_medium=article&itm\_campaign=bottom\_sticky\_on\_article
   9. problems:
      * https://www.geeksforgeeks.org/problems/count-ways-to-express-n-as-the-sum-of-13-and-44024/1?itm\_source=geeksforgeeks&itm\_medium=article&itm\_campaign=bottom\_sticky\_on\_article
4. Geometric Progression.
   1. A sequence of number is called a Geometric progression. If the ration of any two consecutive terms is always the same.
   2. A geometric series is a list of numbers where each number, or term, is found by multiplying the previous term by a common ratio**r.**
   3. 
   4. In finite geometric progression contains a **finite** number of terms. The last term is always defined in this type of progression.
   5. Infinite geometric progression contains an infinite number of terms.  The last term is not defined in this type of progression.
   6. 
   7. Formula for nth term of GP –
   8. 
   9. Formula for sum of the nth term of GP
   10. 
5. Carol Number.
   1. 



1. Matrix Exponentiation.
   1. The terms in juggler sequence first increase to a peak value and then start decreasing
   2. The last term in juggler Sequence is always 1.
2. Juggler Sequence.
3. Juggler Sequence.
4. Juggler Sequence.