NUMBER THEORY

Lesson - 5

Topics to be covered:

1. Prime Numbers and Fibonacci
2. Find the highest occurring digit in Prime number in Range
3. Compute nCr % p | Set 1 (Introduction and Dynamic Programming Solution)
4. Compute nCr % p | Set 2 (Lucas Theorem)

1.Prime numbers and Fibonacci

Given a number, find the numbers (smaller than or equal to n) which are both Fibonacci and prime.

Examples:

```
Input : n = 40                          Output: 2 3 5 13
Explanation :
Here, range(upper limit) = 40
Fibonacci series upto n is, 1,
1, 2, 3, 5, 8, 13, 21, 34.
Prime numbers in above series = 2, 3, 5, 13.

Input : n = 100                         Output: 2 3 5 13 89
Explanation :
Here, range(upper limit) = 40
Fibonacci series upto n are 1, 1, 2,
3, 5, 8, 13, 21, 34, 55, 89.
```

```
Prime numbers in Fibonacci upto n : 2, 3,
5, 13, 89.
```

An efficient solution is to use Sieve to generate all Prime numbers up to n. After we have generated prime numbers, we can quickly check if a prime is Fibonacci or not by using the property that a number is Fibonacci if it is of the form $5i^2 + 4$ or in the form $5i^2 – 4$.

CODE:

```
#include <bits/stdc++.h>
using namespace std;

// Function to check perfect square
bool isSquare(int n)
{
        int sr = sqrt(n);
        return (sr * sr == n);
}

// Prints all numbers less than or equal to n that
// are both Prime and Fibonacci.
void printPrimeAndFib(int n)
{
        // Using Sieve to generate all primes
        // less than or equal to n.
        bool prime[n + 1];
        memset(prime, true, sizeof(prime));
        for (int p = 2; p * p <= n; p++) {
```

```

                // If prime[p] is not changed, then
                // it is a prime
                if (prime[p] == true) {

                    // Update all multiples of p
                    for (int i = p * 2; i <= n; i += p)
                        prime[i] = false;
                }
            }

            // Now traverse through the range and print numbers
            // that are both prime and Fibonacci.
            for (int i=2; i<=n; i++)
            if (prime[i] && (isSquare(5 * i * i + 4) > 0 ||
                                isSquare(5 * i * i - 4) > 0))
                        cout << i << " ";
}

// Driver function
int main()
{
    int n = 30;
    printPrimeAndFib(n);
    return 0;
}
```

2. Find the highest occurring digit in prime numbers in a range

Given a range L to R, the task is to find the highest occurring digit in prime numbers lie between L and R (both inclusive). If multiple digits have the same highest frequency print the largest of them. If no prime number occurs between L and R, output -1.

Examples:
Input : L = 1 and R = 20.
Output : 1
Prime numbers between 1 and 20 are 2, 3, 5, 7, 11, 13, 17, 19.
1 occurs maximum i.e 5 times among 0 to 9.

The idea is to start from L to R, check if the number is prime or not. If prime then increment the frequency of digits (using array) present in the prime number. To check if a number is prime or not we can use Sieve of Eratosthenes.
Below is the implementation of this approach:

```
// C++ program to find the highest occurring digit
// in prime numbers in a range L to R.
#include<bits/stdc++.h>
using namespace std;

// Sieve of Eratosthenes
void sieve(bool prime[], int n)
{
        prime[0] = prime[1] = true;
        for (int p = 2; p * p <= n; p++)
        {
                if (prime[p] == false)
                        for (int i = p*2; i <= n; i+=p)
                                prime[i] = true;
        }
    }

    // Returns maximum occurring digits in primes
    // from l to r.
    int maxDigitInPrimes(int L, int R)
    {
        bool prime[R+1];
        memset(prime, 0, sizeof(prime));

        // Finding the prime number up to R.
        sieve(prime, R);
```

```

    // Initialse frequency of all digit to 0.
    int freq[10] = { 0 };
    int val;

    // For all number between L to R, check if prime
    // or not. If prime, incrementing the frequency
    // of digits present in the prime number.
    for (int i = L; i <= R; i++)
    {
        if (!prime[i])
        {
            int p = i; // If i is prime
            while (p)
            {
                freq[p%10]++;
                p /= 10;
            }
        }
    }

    // Finding digit with highest frequency.
    int max = freq[0], ans = 0;
    for (int j = 1; j < 10; j++)
    {
        if (max <= freq[j])
        {
```

```
                    max = freq[j];
                    ans = j;
            }
    }

    return ans;
}

// Driven Program
int main()
{
    int L = 1, R = 20;

    cout << maxDigitInPrimes(L, R) << endl;
    return 0;
}
```

3. Compute nCr % p | Set 1 (Introduction and Dynamic Programming Solution)

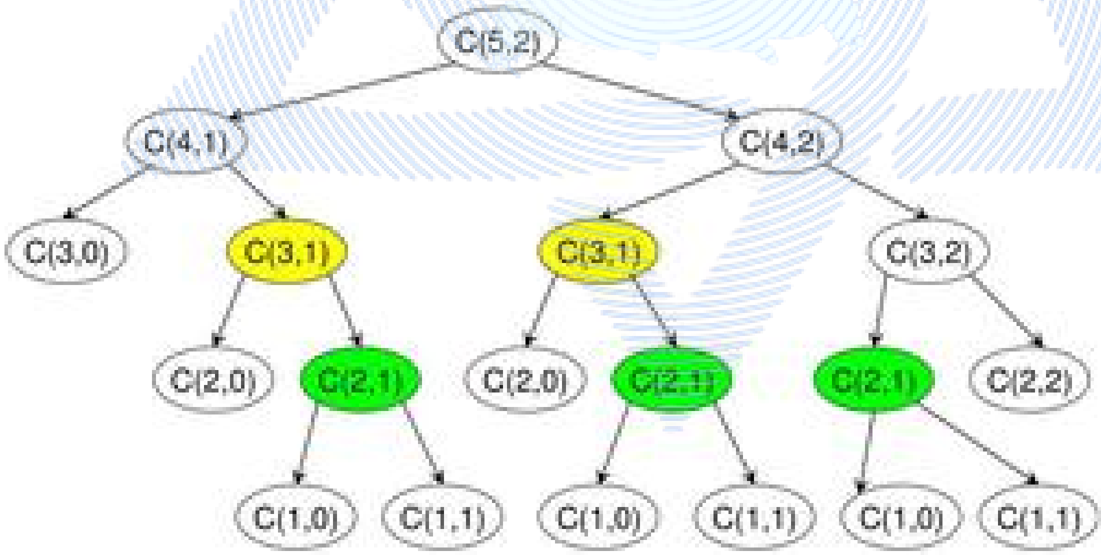
A binomial coefficient C(n, k) can be defined as the coefficient of x^k in the expansion of (1 + x)^n.

Write a function that takes two parameters n and k and returns the value of Binomial Coefficient C(n, k). For example, your function should return 6 for n = 4 and k = 2, and it should return 10 for n = 5 and k = 2.

The value of C(n, k) can be recursively calculated using the following standard formula for Binomial Coefficients.

C(n, k) = C(n-1, k-1) + C(n-1, k)

C(n, 0) = C(n, n) = 1



Code:

```
int binomialCoeff(int n, int k)
{
  int C[k + 1];
  memset(C, 0, sizeof(C));
  C[0] = 1; // nC0 is 1
  for (int i = 1; i <= n; i++) {
    for (int j = min(i, k); j > 0; j--)
      C[j] = C[j] + C[j - 1];
  }
  return C[k];
}
```

Compute nCr % p | Set 1 (Introduction and Dynamic Programming Solution)

We can use distributive property of modulo operator to find nCr % p using above formula.

C(n, r)%p = [C(n-1, r-1)%p + C(n-1, r)%p] % p

C(n, 0) = C(n, n) = 1

Code:

```
int nCrModp(int n, int r, int p)
{
  if (r > n - r)
    r = n - r;
  int C[r + 1];
  memset(C, 0, sizeof(C));
  C[0] = 1; // Top row of Pascal Triangle
```

```
    for (int i = 1; i <= n; i++) {
        for (int j = min(i, r); j > 0; j--)
            C[j] = (C[j] + C[j - 1]) % p;
    }
    return C[r];
}
```

4. Compute nCr % p | Set 2 (Lucas Theorem)

Given three numbers n, r and p, compute value of nCr mod p.

Lucas Theorem:
For non negative integers n and r and a prime p, the following congruence relation holds:

$$\binom{n}{r} = \prod_{i=0}^{k} \binom{n_i}{r_i} (mod\ p),$$

where

$$n = n_k p^k + n_{k-1} p^{k-1} + ... + n_1 p + n0,$$

and

$$r = r_k p^k + r_{k-1} p^{k-1} + ... + r_1 p + r0$$

Using Lucas Theorem for $_nC_r$ % p:
Lucas theorem basically suggests that the value of nCr can be computed by multiplying results of niCri where ni and ri are individual same-positioned digits in base p representations of n and r respectively..

Time complexity of this solution is $O(p^2 * \log_p n)$ and it requires only O(p) space.

```
#include<bits/stdc++.h>
```

```
using namespace std;
int nCrModpDP(int n, int r, int p)
{
if (r > n - r)
    r = n - r;
  int C[r+1];
  memset(C, 0, sizeof(C));
  C[0] = 1;
    for (int i = 1; i <= n; i++)
    {
      for (int j = min(i, r); j > 0; j--)
        C[j] = (C[j] + C[j-1])%p;
    }
    return C[r];
  }
  int nCrModpLucas(int n, int r, int p)
  {
   if (r==0)    return 1;
   int ni = n%p, ri = r%p;
   return (nCrModpLucas(n/p, r/p, p)*nCrModpDP(ni,ri,p))% p;
  }
  int main()
  {
    int n = 1000, r = 900, p = 13;
    cout << "Value of nCr % p is " << nCrModpLucas(n, r, p);
    return 0;
  }
```