Number Theory

Sieve:

```
int N = 10000;
int prime[10000];
int status[10000/32 + 1];

bool Check(int index,int bitNumber){
      int x = status[index] & (1<<bitNumber);
      if( x == 0 ) {
            return false; // prime
      }
      else {
            return true; // composite
      }
}

void Set(int index,int bitNumber){
      status[index] = status[index] | (1 << bitNumber);
}

void bitwiseSieve()
{
     int i, j, sqrtN;
     sqrtN = int( sqrt( N ) );
     for( i = 3; i <= sqrtN; i += 2 )
     {
             if( Check(i/32, i%32) == false )
             {
                   for( j = i*i; j <= N; j += 2*i )
                   {
                        Set(j/32, j%32);
                   }
             }
       }
}

bool isPrime( int num ) {
      if( num == 2 ) return true;
      else if( num % 2 == 0 ) return false;
      else {
            return !Check(num/32, num%32);
      }
}
```

Euler Phi:

```
int phi(int n) {
    int result = n;
    for (int i = 2; i * i <= n; i++) {
        if (n % i == 0) {
            while (n % i == 0)
                n /= i;
            result -= result / i;
        }
    }
    if (n > 1)
        result -= result / n;
    return result;
}
void phi_1_to_n(int n) {
    vector<int> phi(n + 1);
    for (int i = 0; i <= n; i++)
        phi[i] = i;

    for (int i = 2; i <= n; i++) {
        if (phi[i] == i) {
            for (int j = i; j <= n; j += i)
                phi[j] -= phi[j] / i;
        }
    }
}
void phi_1_to_n(int n) {
    vector<int> phi(n + 1);
    phi[0] = 0;
    phi[1] = 1;
    for (int i = 2; i <= n; i++)
        phi[i] = i - 1;

    for (int i = 2; i <= n; i++)
        for (int j = 2 * i; j <= n; j += i)
              phi[j] -= phi[i];
}
```

Sum of Divisors:

$$\sigma(n) = \frac{p_1^{e_1+1}-1}{p_1-1} \cdot \frac{p_2^{e_2+1}-1}{p_2-1} \cdots \frac{p_k^{e_k+1}-1}{p_k-1}$$

Number of Divisors:

$$d(n) = (e_1+1) \cdot (e_2+1) \cdots (e_k+1)$$

Pseudo Code : O($\sqrt[3]{n}$)

```
N = input()
primes = array containing primes till 10^6
ans = 1
for all p in primes :
      if p*p*p > N:
         break
      count = 1
      while N divisible by p:
         N = N/p
         count = count + 1
      ans = ans * count
if N is prime:
      ans = ans * 2
else if N is square of a prime:
      ans = ans * 3
else if N != 1:

      ans = ans * 4
```

EGCD:

```
int egcd(int a, int b, int& x, int& y) {
    if (b == 0) {
        x = 1;
        y = 0;
        return a;
    }
    int x1, y1;
    int d = egcd(b, a % b, x1, y1);
    x = y1;
    y = x1 - y1 * (a / b);
    return d;
}
```

Modular Multiplicative Inverse:

```
int mmi(int a, int m){
      int x, y;
      int g = egcd(a, m, x, y);
      if (g != 1) {
          return -1; //No solution!
      }
      else {
          x = (x % m + m) % m;
          return x;
      }
}
```

CRT:

```
ll CRT_2( ll a1, ll n1, ll a2, ll n2 ) // CRT for 2 equations
{
      ll d = __gcd(n1, n2);
      a1 = a1 % n1;
      a2 = a2 % n2;
      if( ( ((a1-a2) % d) + d ) % d != 0 ) return -1;

      ll x, x1, y1;
      ll lcm = n1 * n2 / d;
      if( n1 > n2 ) {
            ll tmp1 = n1; n1 = n2, n2 = tmp1;
            ll tmp2 = a1; a1 = a2, a2 = tmp2;
      }
      Egcd(n1, n2, x1, y1);

      ll a = x1 * (a2 - a1) / d;
      ll b = n2 / d;
      ll c = n1;

      return x = (((a1 + (a%b) * c) % lcm) + lcm) % lcm;
}


ll CRT_t( std::vector<ll> a, std::vector<ll> n ) // CRT for t
equations
```

```
{
      ll a1 = CRT_2(a[0], n[0], a[1], n[1]);
      if( a1 == -1 ) return -1; // no solution

      ll n1 = n[0] * n[1] / __gcd(n[0], n[1]);
      ll a2, n2, sz = a.size();

      for( int i = 2; i < sz; i++ ) {
            a2 = a[i], n2 = n[i];
            a1 = CRT_2(a1, n1, a2, n2);
            if( a1 == -1 ) return -1;
            ll d = __gcd(n1, n2);
            n1 = n1 * n2 / d;
      }
      return a1;
}
```

Big Mod:

```
LL bigmod(LL a, LL b, LL M) {
      if(b == 0) return 1 % M;
      LL x = bigmod(a, b/2, M);
      x = (x*x)%M;
      if(b%2 == 1) x = (x*a)%M;
      return x;
}
```

Graph

Floyd-Warshall Algorithm: All pair shortest path

```
LL N = 1000;
LL d[N][N];
void floyd_warshall(int n) {
      for (int k = 0; k < n; ++k) {
          for (int i = 0; i < n; ++i) {
              for (int j = 0; j < n; ++j) {
                  d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
              }
          }
      }
}
```

Bellman-Ford Algorithm: Single source, negative edges

```
struct edge
{
    int a, b, cost;
};

int n, m, v;
vector<edge> e;
const int INF = 1000000000;
void bellman_ford(){
    vector<int> d (n, INF);
    d[v] = 0;
    while ( true ){
        bool any = false;
        for (int j=0; j<m; ++j)
            if (d[e[j].a] < INF)
                if (d[e[j].b] > d[e[j].a] + e[j].cost)
                {
                    d[e[j].b] = d[e[j].a] + e[j].cost;
                    any = true;
                }
        if (!any) break;
    }
   // retrieve path:
   if (d[t] == INF)
        cout << "No path from " << v << " to " << t << ".";
    else
    {
        vector<int> path;
        for (int cur = t; cur != -1; cur = p[cur])
            path.push_back (cur);
        reverse (path.begin(), path.end());

        cout << "Path from " << v << " to " << t << ": ";
        for (size_t i=0; i<path.size(); ++i)
            cout << path[i] << ' ';
    }
}
```

Shortest Path Faster Algorithm:

```
const int INF = 1000000000;
```

```
vector<vector<pair<int, int>>> adj;

bool spfa(int s, vector<int>& d) {
    int n = adj.size();
    d.assign(n, INF);
    vector<int> cnt(n, 0);
    vector<bool> inqueue(n, false);
    queue<int> q;

    d[s] = 0;
    q.push(s);
    inqueue[s] = true;
    while (!q.empty()) {
        int v = q.front();
        q.pop();
        inqueue[v] = false;

        for (auto edge : adj[v]) {
            int to = edge.first;
            int len = edge.second;

            if (d[v] + len < d[to]) {
                d[to] = d[v] + len;
                if (!inqueue[to]) {
                    q.push(to);
                    inqueue[to] = true;
                    cnt[to]++;
                    if (cnt[to] > n)
                        return false;  // negative cycle
                }
            }
        }
    }
    return true;
}
```

Longest Common Subsequence:

```
const int m_ = 101;
const int n_ = 101;
int dp[m_][n_];
int lcs_easy( string A, string B, int m, int n ) {
      int LCS[m+1][n+1];

      for(int i = 0; i <= m; i++ ) {
          for( int j = 0; j <= m; j++ ) {
              LCS[i][j] = 0;
          }
      }

      for( int i = 1; i <= m; i++ ) {
            for( int j = 1; j <= n; j++ ) {
                  if( A[i-1] == B[j-1] ) LCS[i][j] = 1 +
LCS[i-1][j-1];
                  else LCS[i][j] = max( LCS[i-1][j],
LCS[i][j-1] );
            }
      }
      return LCS[m][n];
}
int main()
{
    string s1 = "axyt";
    string s2 = "ayxb";
    std::cin >> s1 >> s2;
    for( int i = 0; i < m_; i++ ) {
        for( int j = 0; j < n_; j++ ) {
            dp[i][j] = -1;
        }
    }
    cout << lcs_easy( s1, s2, s1.size(), s2.size() ) <<
endl;
}
```

0/1 knapsack:

```
int knapSack(int W, int w[], int v[], int n) {
   int i, wt;
   int K[n + 1][W + 1];
   for (i = 0; i <= n; i++) {
      for (wt = 0; wt <= W; wt++) {
         if (i == 0 || wt == 0)
         K[i][wt] = 0;
         else if (w[i - 1] <= wt)
            K[i][wt] = max(v[i - 1] + K[i - 1][wt - w[i - 1]], K[i - 1][wt]);
         else
        K[i][wt] = K[i - 1][wt];
      }
   }
   return K[n][W];
}
```

KMP:

```
void lps_func(string txt, vector<int>&Lps){
    Lps[0] = 0;
    int len = 0;
    int i=1;
    while (i<txt.length()){
        if(txt[i]==txt[len]){
            len++;
            Lps[i] = len;
            i++;
            continue;
        }
        else{
            if(len==0){
                Lps[i] = 0;
                i++;
                continue;
            }
            else{
                len = Lps[len-1];
                continue;
            }
        }
    }
}
void KMP(string pattern,string text){
    int n = text.length();
    int m = pattern.length();
    vector<int>Lps(m);

    lps_func(pattern,Lps); // This function constructs the Lps array.

    int i=0,j=0;
    while(i<n){
        if(pattern[j]==text[i]){i++;j++;} // If there is a match continue.
        if (j == m) {
            cout<<i - m <<' ';    // if j==m it is confirmed that we have found the pattern and we output the index
                                  // and update j as Lps of last matched character.
            j = Lps[j - 1];
        }
        else if (i < n && pattern[j] != text[i]) {  // If there is a mismatch
            if (j == 0)          // if j becomes 0 then simply increment the index i
                i++;
            else
                j = Lps[j - 1];  //Update j as Lps of last matched character
        }
    }
}
```