CHITTAGONG UNIVERSITY OF ENGINEERING AND TECHNOLOGY

Team Notebook

CUET_Ɛ_St00ges

Index

1.1 Sieve

```
// sieve :- prime number generator;
// 0 for prime, 1 for not prime;

vector<int> primes;
vector<bool> mark(1000002);
void sieve(int n)
{
          int i, j, limit = sqrt(n*1.) + 2;
          mark[1] = 1;
          for(i = 4; i<=n; i+=2) mark[i] = 1;
          primes.push_back(2);
          for(i = 3; i <= n; i += 2){
              if(!mark[i]){
          primes.push_back(i);
                    if(i<=limit){
                        for(j = i*i; j <= n; j += i*2){
                            mark[j] = 1;
                        }
                    }
              }
          }
}
// Block/Segmented Sieve:
// Output: prime list in range {L, R}; (R-L+1) <= 1e7 && R <=
1e12;
vector<char> segmentedSieve(long long L, long long R) {  //
generate all primes up to sqrt(R)
          long long lim = sqrt(R);
          vector<char> mark(lim + 1, false);
          vector<long long> primes;
          for (long long i = 2; i <= lim; ++i) {
              if (!mark[i]) {
          primes.emplace_back(i);
          for (long long j = i * i; j <= lim; j += i) mark[j]
= true;
              }
          }
          vector<char> isPrime(R - L + 1, true);
          for (long long i : primes)
              for (long long j = max(i * i, (L + i - 1) / i *
i); j <= R; j += i) isPrime[j - L] = false;
          if (L == 1) isPrime[0] = false;
          return isPrime;
}
// Bitwise - sieve :- prime number generator;
// 0 for prime, 1 for not prime; >> Not sure ... #define MX
10000000
int marked[MX / 64 + 2];
vector<int> primes;
#define mark(x) marked[x >> 6] |= (1 << ((x & 63) >> 1))
#define check(x)(marked[x >> 6] & (1 << ((x & 63) >> 1)))
bool isPrime(int x) { return (x > 1) && ((x == 2) || ((x & 1)
&& (!(check(x))))); }
void sieve(int n) {
          int i, j;
          primes.push_back(2);
          for (i = 3; i * i <= n; i += 2) {
              if (!check(i)) {
                    primes.push_back(i);
          for (j = i * i; j <= n; j += i << 1) mark(j);
              }
          }
}
```

1.2 Linear Sieve

```
const int MX = 10000001;
int lp[MX];
vector<int> pr;
void linear_sieve()
{
    for(int i = 2; i < MX; ++i)
    {
        if(lp[i] == 0) {lp[i] = i; pr.PB(i);}
        for(int j = 0; j < (int)pr.size() && pr[j] <= lp[i]
&& (i * pr[j]) < MX; ++j) lp[i * pr[j]] = pr[j];
    }
}
```

1.3 Pascal Triangle

```
const int MX = 1e2;
ll C[MX + 5][MX + 5];
void pascal() {
          C[0][0] = 1;
          for (int K = 1; K <= MX; K++) {
              C[K][0] = C[K][K] = 1;
              for (int L = 1; L < K; L++) {
          C[K][L] = C[K - 1][L - 1] + C[K - 1][L];
              }
          }
}
```

1.4 nCr with array

```
// Complexity O(n)
const int MX = 1e6;
ll fact[MX + 5], inv[MX + 5];
void factorial() {
          fact[0] = fact[1] = 1;
          for (ll K = 1; K <= MX; K++) fact[K] = fact[K - 1] * K
% mod;
          inv[MX] = inverse(fact[MX], mod);
          for (ll K = MX - 1; K >= 1; K--) inv[K] = inv[K + 1] *
(K + 1) % mod;
          inv[0] = 1;
}
// function call ...
factorial();
cout << (fact[18] * inv[5] % mod) * inv[13] % mod << "\n";
```

Combinatorics problems can be done using stars and bars.The number of ways to put *n* identical objects into *k* labeled boxes is (n+k-1)C(n)

while counting nCr where n,r size is big to get rid of overflow we can do this :

Code:

```
minncr(k,n-k);
ll ans=1;
for(ll i=1;i<=k;i++){
   ans = (ans*(n-i+1) )/ i ;
}
```

1.5 divisors

Maximal number of divisors of any n-digit number:
First 18 numbers: 4, 12, 32, 64, 128, 240, 448, 768, 1344, 2304, 4032, 6720, 10752, 17280, 26880, 41472, 64512, 103680

```
// Number of divisors...
// Euler's totient function…
// first, run a sieve for value sqrt(n);
vector<pair<int, int> > divisors;
void divs(int n) {
          int cnt, tot = 1, i;
          for (i = 0; i < (int)primes.size() && (primes[i] *
primes[i]) <= n; i++) {
              if (n % primes[i] == 0) {
                    cnt = 1;
                    while (n % primes[i] == 0) {
                        n /= primes[i];
                        cnt++;
          }
                    divisors.push_back(make_pair(primes[i], cnt -
1));
                    tot *= cnt;
              }
          }
          if (n > 1) {
              tot *= 2;
              divisors.push_back(make_pair(n, 1));
          }
          printf("Number of divisors %d\n", tot);
          for (i = 0; i < (int)divisors.size(); i++) printf("%d
%d\n", divisors[i].first, divisors[i].second);
```

```
}
// Number of divisors...
ll NOD(int n) {
        ll ans = 1;
        for (int K = 0; K < sz(divisors); K++) {
            ans *= (ll)(divisors[K].se + 1);
        }
        return ans;
}
// Sum of divisors...
ll SOD(int n) {
        ll ans = 1, cnt;
        for (int K = 0; K < sz(divisors); K++) {
            cnt = divisors[K].fi;
            while (divisors[K].se--) cnt *=
(ll)divisors[K].fi;
            ans *= (ll)(cnt - 1) / (divisors[K].fi - 1);
        }
        return ans;
}
```

1.6 Totient Function

```
#define MX 1e5+6;
// phi pre-calculation function . . .
ll phi[MX];
void phi_function()
{
    memset(phi, 0, sizeof phi);
    phi[1] = 1;
    for(int i=2; i<=MAX; ++i){
        if(phi[i]==0){
            phi[i]=i-1;
            for(int j=i+i; j<=MAX; j+=i){
                if(phi[j]==0)   phi[j] = j;
                phi[j]= phi[j]/i * (i-1);
            }
        }
    }
}
// gcd sum calculation function . . .
ll table[MAX];
void div()
{
        memset(table, 0, sizeof table);
        for(int i=1;i<MAX;i++){
            for(int j=i+i;j<MAX;j=j+i){
                table[j]+=(i*phi[j/i]);
            }
        }
        for(int i=2;i<MAX;i++)
            table[i]+=table[i-1];
}
```

1.7 Some theory

GCD-LCM

GCD sum function $g(n) = \prod_{i=0}^{k}((a_i+1)p_i^{a_i} - a_i p^{a_i-1})$ where $g(n) = gcd(1,n) + gcd(2,n) + gcd(3,n) + \cdots + gcd(n,n) = \sum_{i=1}^{n} gcd(i,n)$

LCM sum function $SUM = \frac{n}{2}(\sum_{d|n}(\phi(d) \times d) + 1)$ where $SUM = lcm(1,n) + lcm(2,n) + lcm(3,n) + \cdots + lcm(n,n) = \sum_{i=1}^{n} lcm(i,n)$

Sum of coprimes of $n = \frac{n \cdot \phi(n)}{2}$

5.20 Totient Function

- $\phi(n) = n \times \frac{p_1-1}{p_1} \times \frac{p_2-1}{p_2} \cdots \times \frac{p_k-1}{p_k}$

5.15 NOD-SOD

Let $n = p_1^{a_1} p_2^{a_2} \cdots p_k^{a_k}$, then, $NOD(n) = (a_1+1)(a_2+1)\cdots(a_k+1)$ and $SOD = (1+p_1+p_1^2+\cdots p_1^{a_1}) \cdot (1+p_2+p_2^2+\cdots p_2^{a_2}) \cdots (1+p_k^1+p_k^2+\cdots p_k^{a_k}) = \frac{p_1^{a_1+1}-1}{p_1-1} \cdot \frac{p_2^{a_2+1}-1}{p_2-1} \cdot \cdots \cdot \frac{p_k^{a_k+1}-1}{p_k-1}$

5 Count divisors of n in cubic-root complexity

- Split number n in two numbers x and y such that $n = x \cdot y$ where x contains only prime factors in range $2 \le x \le n^{\frac{1}{3}}$ and y deals with higher prime factors greater than $n^{\frac{1}{3}}$.
- Count total factors of x using the naive trial division method. Let this count be F(x).
  - If y is a prime number then factors will be 1 and y itself. That implies, F(y) = 2.
  - If y is square of a prime number, then factors will be 1, sqrt(y) and y itself. That implies, F(y) = 3.
  - If y is the product of two distinct prime numbers, then factors will be 1, both prime numbers and number y itself. That implies, F(y) = 4.
- Since F(x*y) is a multiplicative function and gcd(x, y) = 1, that implies, F(x*y) = F(x)*F(y) which gives the count of total distinct divisors of n.

Catalan Numbers

$$C_n = \binom{2n}{n} - \binom{2n}{n-1} = \frac{1}{n+1}\binom{2n}{n}, n \ge 0$$

5.1 Application of Catalan Numbers

1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012, 742900, 2674440, 9694845,..

- Number of correct bracket sequence consisting of n opening and n closing brackets.
- The number of rooted full binary trees with $n+1$ leaves (vertices are not numbered). A rooted binary tree is full if every vertex has either two children or no children.
- The number of ways to completely parenthesize $n+1$ factors.
- The number of triangulations of a convex polygon with $n+2$ sides (i.e. the number of partitions of polygon into disjoint triangles by using the diagonals).
- The number of ways to connect the $2n$ points on a circle to form n disjoint chords.
- The number of non-isomorphic full binary trees with n internal nodes (i.e. nodes having at least one son).
- The number of monotonic lattice paths from point $(0,0)$ to point (n,n) in a square lattice of size $n \times n$, which do not pass above the main diagonal (i.e. connecting $(0,0)$ to (n,n)).
- Number of permutations of length n that can be stack sorted (i.e. it can be shown that the rearrangement is stack sorted if and only if there is no such index $i < j < k$, such that $a_k < a_i < a_j$).
- The number of non-crossing partitions of a set of n elements.
- The number of ways to cover the ladder $1 \ldots n$ using n rectangles (The ladder consists of n columns, where ith column has a height i).

If p is a prime number, then $gcd(p,q) = 1$ for all $1 \le q < p$. Therefore we have: $\phi(p) = p-1$.

If p is a prime number and $k \ge 1$, then there are exactly p^k/p numbers between 1 and p^k that are divisible by p. Which gives us: $\phi(p^k) = p^k - p^{k-1}$.

If a and b are relatively prime, then: $\phi(ab) = \phi(a) \cdot \phi(b)$.

In general, for not co-prime a and b, the equation $\phi(ab) = \phi(a) \cdot \phi(b) \cdot \frac{d}{\phi(d)}$ with $d = \gcd(a,b)$ holds.

Sum of co-primes of a number n is $\frac{n \cdot \phi(n)}{2}$.

3.9 Pick's Theorem

Given a certain lattice polygon with non-zero area.

We denote its area by S, the number of points with integer coordinates lying strictly inside the polygon by I and the number of points lying on polygon sides by B.

$$S = I + \frac{B}{2} - 1$$

B can be calculated using $GCD(|x_1 - x_2|, |y_1 - y_2|) + 1$

Lucas Theorem:

For non negative integers n and r and a prime p, 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

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

1.8 Extended Euclid

```
int gcd(int a, int b, int &x, int &y) {
    if (a == 0) {
        x = 0; y = 1;
        return b;
    }
    int x1, y1;
    int d = gcd(b%a, a, x1, y1);
    x = y1 - (b / a) * x1;
    y = x1;
    return d;
}
bool find_any_solution(int a, int b, int c, int &x0, int &y0,
int &g) {
    g = gcd(abs(a), abs(b), x0, y0);
    if (c % g) {
        return false;
    }
    x0 *= c / g;
    y0 *= c / g;
    if (a < 0) x0 = -x0;
    if (b < 0) y0 = -y0;
    return true;
}
```

While the Euclidean algorithm calculates only the greatest common divisor (GCD) of two integers x and y, the extended version also finds a way to represent GCD in terms of a and b, i.e. coefficients x and y for which:

$$a.x + b.y = GCD(x, y)$$

Now, we know Diophantine Equations.These are the polynomial equations for which integral solutions exist.Ex: $3x + 7y = 1$ or $x^2 - y^2 = z^3$.

For CP, we only need to deal with ax+by=c . Here, solutions exist only if gcd(a,b) divides c.

Now, how to find x and y if we have to find the value of x and y if we are given the equation or we can come in a situation where we need to solve this equation.

$$ax + by = gcd(a, b)$$
$$gcd(a, b) = gcd(b, a\%b)$$
$$gcd(b, a\%b) = bx_1 + (a\%b)y_1$$

Now, $a\%b = a - (a/b) * b$

From above, $ax + by = bx_1 + (a\%b)y_1$

$$ax + by = bx_1 + (a - (a/b) * b)y_1$$
$$ax + by = ay_1 + b(x_1 - (a/b) * y_1)$$

So, comparing the coefficients of a and b,

$$x = y_1 \quad and \quad y = x_1 - (a/b) * y_1$$

Code:

```
struct Triplet{
    int x,y,gcd;
};
Triplet extendedEuclid(int a, int b)
{
    if(b == 0){
        Triplet ans;
        ans.gcd = a;
        ans.x = 1;
        ans.y = 0;
        return ans;
    }
    Triplet next = extendedEuclid(b, a%b);
    Triplet ans;
    ans.gcd = next.gcd;
    ans.x = next.y;
    ans.y = next.x - (a/b)*next.y;
    return ans;
}
```

1.8.1 chinese remainder theorem

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

```
class ChineseRemainderTheorem
{
  typedef long long vlong;
  typedef pair<vlong,vlong> pll;
  /** CRT Equations stored as pairs of vectors. See
  addEquation()*/
  vector<pll> equations;
public:
  void clear()
  {
    equations.clear();
  }
  /** Add equation of the form x = r (mod m)*/
  void addEquation( vlong r, vlong m )
  {
    equations.push_back({r, m});
  }
  pll solve()
  {
    if (equations.size() == 0)
      return {-1,-1}; /// No
    equations to solve
    vlong a1 = equations[0].first;
    vlong m1 = equations[0].second;
    a1 %= m1;
    /** Initially x = a_0 (mod m_0)*/
    /** Merge the solution with remaining equations */
    for ( int i = 1; i < equations.size(); i++ )
      {
        vlong a2 = equations[i].first;
        vlong m2 = equations[i].second;
        vlong g = __gcd(m1, m2);
        if ( a1 % g != a2 % g )
          return {-1,-1}; ///
        Conflict in equations
        /** Merge the two equations*/
        vlong p, q;
        ext_gcd(m1/g, m2/g, &p, &q);
        vlong mod = m1 / g * m2;
        vlong x = ( (__int128)a1 * (m2/g) % mod *q % mod
                + (__int128)a2 * (m1/g) % mod * p % mod ) %
              mod;
        /** Merged equation*/
        a1 = x;
        if ( a1 < 0 )
          a1 += mod;
        m1 = mod;
      }
    return {a1, m1};
  }
}
```

1.9 Bit Manipulation

⇒ $(a\&b) + (a|b) = a + b$

⇒ **Distributive operator :**

- AND over OR : $a \& (b | c) = (a \& b) | (a \& c)$
- AND over XOR : $a \& (b \wedge c) = (a \& b) \wedge (a \& c)$
- OR over AND : $a | (b \& c) = (a | b) \& (a | c)$

However, xor does not distribute over AND or OR and neither does OR distribute over XOR.

1.10 Miller Rabin - prime testing

```
// call using miller_rabin(n); returns 0 or 1
using ll = long long;
ll bigMod(ll a, ll e, ll mod) {
        if (e == 0) return 1LL;
        ll ret = bigMod(a, e >> 1LL, mod);
        ret = (__int128)ret * ret % mod;
        if (e & 1) {
        return (__int128)ret * a % mod;
        }
        return ret;
}
bool is_composite(ll n, ll a, ll d, ll s) {
        ll x = bigMod(a % n, d, n);

        if (x == 1 or x == n - 1) return false;

        for (int r = 1; r < s; r++) {
        x = (__int128)x * x % n;
        if (x == n - 1) return false;
        }
        return true;
}
bool miller_rabin(ll n, int trial = 10) {
        if (n < 4) {
        return n == 2 or n == 3;
        }

        if (n % 2 == 0) {
        return false;
        }

        int s = 0;
        ll d = n - 1;
        while ((d & 1) == 0) {
        s++;
        d >>= 1LL;
        }

        while (trial--) {
        ll a = 2 + rand() % (n - 2);
        if (is_composite(n, a, d, s)) return false;
        }
        return true;
}
```

1.11 Counting divisors O(n^⅓)

```
// using miller rabin algo.
// complexity O(n^1/3)
// Counting Divisors in O(N^(1/3))
// use miller rabin code here.
bool square(ll n) {
        ll root = sqrt(n);
            for (ll from = root - 2; from <= root + 2; from++)
{
            if (from * from == n) return true;
        }
        return false;
}
ll count_divisors(ll n) {
        // divide n with primes <= n^(1/3) and update answer
        ll ans = 1LL;

        for (int p = 2; 1LL * p * p * p <= n; p++) {
            int exp = 0;
            while (n % p == 0) {
                n /= p;
                exp++;
            }
            ans *= (exp + 1);
        }

        // If n doesn't contain any prime factors anymore then
we are done
        if (n == 1) return ans;

        // Check if n is prime
        if (miller_rabin(n)) {
            ans *= 2LL;  // add contribution to answer
        }
        // Check if n is a square of a prime
        else if (square(n)) {
            ans *= 3LL;
        }
        // else n is product of two different primes
        else {
            ans *= 4LL;
        }

        return ans;
}
```

1.12 Diophantine Equation

```
 // returns (d, x, y) such that ax + by = gcd(a, b) = d
 tuple<int, int, int> exgcd(int a, int b) {
          if (b == 0) {
          return {a, 1, 0};
          }
          auto [d, _x, _y] = exgcd(b, a % b);
          // b * _x + (a - b (a/b) ) * _y
          // a * _y + b(_x - (a/b)_y)
          int x = _y;
          int y = _x - (a / b) * _y;
          return {d, x, y};
}

// returns (true, x, y) if solution exists
// returns (false, 0, 0) otherwise
tuple<bool, int, int> diophantine(int a, int b, int c) {
        auto [d, _x, _y] = exgcd(a, b);

        if (c % d) {
        return {false, 0, 0};
        } else {
        int x = (c / d) * _x;
        int y = (c / d) * _y;
        return {true, x, y};
        }
}
int main() {
        int a = 5, b = -2, c = -1;
        auto [solution_exists, x, y] = diophantine(a, b,
c);
        if (solution_exists) {
        cout << x << ' ' << y << endl;
        }
        return 0;
}
```

1.13 Locas Theorem

```
const int N = 1e5 + 5;
int f[N], inv_f[N];

void init(int n, int mod) {
        // first calucalte i^(-1)
        inv_f[1] = 1;
        for (int i = 2; i <= n; i++) {
        inv_f[i] = mod - 1LL * (mod / i) * inv_f[mod % i] %
mod;
        }

        // Calculate Inverse factorial and factorial
        inv_f[0] = f[0] = f[1] = 1;
        for (int i = 2; i <= n; i++) {
        f[i] = (1LL * f[i - 1] * i) % mod;
        inv_f[i] = (1LL * inv_f[i] * inv_f[i - 1]) % mod;
        }
}

int nCr(int n, int r, int mod) {
        if (r > n) return 0;
```

```
        return (((1LL * f[n] * inv_f[n - r]) % mod) *
inv_f[r]) % mod;
}

// Convert n to some base
vector<int> toBase(ll n, int base) {
        vector<int> digits;
        while (n) {
        digits.push_back(n % base);
        n /= base;
        }
        return digits;
}

int lucas(ll n, ll r, int mod) {
        if (r > n) return 0;

        // convert n and r to base mod
        vector<int> N = toBase(n, mod);
        vector<int> R = toBase(r, mod);

        // make lengths equal by filling leading digits of
R with zeros
        while (R.size() < N.size()) {
        R.push_back(0);
        }

        // Calculate answer
        int ans = 1;
        for (int i = 0; i < N.size(); i++) {
        ans = (1LL * ans * nCr(N[i], R[i], mod)) % mod;
        }
        return ans;
}

int main() {
        ll n, r, p;
        cin >> n >> r >> p;
        init(p - 1, p);
        cout << lucas(n, r, p) << '\n';
}
```

1.14 Modular inverse from 1 to n in O(n)

```
const int N = 1e5 + 5;
int inv[N];
void range_mod_inverse(int n, int mod) {
        inv[1] = 1;
        for (int i = 2; i <= n; i++) {
        inv[i] = mod - 1LL * (mod / i) * inv[mod % i] %
mod;
        }
}
```

```
}
```

1.15 Gaussian elimination

```
const int N = 3e5 + 9;

const double eps = 1e-9;
int Gauss(vector<vector<double>> a, vector<double> &ans) {
 int n = (int)a.size(), m = (int)a[0].size() - 1;
 vector<int> pos(m, -1);
 double det = 1; int rank = 0;
 for(int col = 0, row = 0; col < m && row < n; ++col) {
  int mx = row;
  for(int i = row; i < n; i++) if(fabs(a[i][col]) > fabs(a[mx][col])) mx = i;
  if(fabs(a[mx][col]) < eps) {det = 0; continue;}
  for(int i = col; i <= m; i++) swap(a[row][i], a[mx][i]);
  if (row != mx) det = -det;
  det *= a[row][col];
  pos[col] = row;
  for(int i = 0; i < n; i++) {
   if(i != row && fabs(a[i][col]) > eps) {
    double c = a[i][col] / a[row][col];
    for(int j = col; j <= m; j++) a[i][j] -= a[row][j] * c;
   }
  }
  ++row; ++rank;
 }
 ans.assign(m, 0);
 for(int i = 0; i < m; i++) {
  if(pos[i] != -1) ans[i] = a[pos[i]][m] / a[pos[i]][i];
 }
 for(int i = 0; i < n; i++) {
  double sum = 0;
  for(int j = 0; j < m; j++) sum += ans[j] * a[i][j];
  if(fabs(sum - a[i][m]) > eps) return -1; //no solution
 }
 for(int i = 0; i < m; i++) if(pos[i] == -1) return 2; //infinte solutions
 return 1; //unique solution
}
```

2.1 Kadane (1D)

```
 sum = ans = ara[0];
 int u = 0, v = 0;
 for (int K = 1; K < n; K++) {
         if (sum + ara[K] >= ara[K]) {
             sum += ara[K];
             v++;
         } else {
             sum = ara[K];
             u = v = K;
         }
         if (sum > ans) {
             ans = sum;
             ans_l = u;
             ans_r = v;
         }
 }
```

2.2 Kadane (2D)

```
// Kadane's algo for Maximum Sub sub-Matrix (including
Max.sum sub-array)...
vector<vector<int> > ara(n, vector<int> (n));
void MaxSumSubMatrix(int n) {
         int max_sum = 0, sum, max_;
         // test_input(n);
         int ansup = 0, ansdown = 0, ansleft = 0, ansright =
0, ul, dl, u, v;
         for (int K = 0; K < n; K++) {
             vector<int> help(n);
             for (int L = K; L < n; L++) {
         for (int M = 0; M < n; M++) {
                     help[M] += ara[M][L];
         }
                 // Kadane's Algo for a 1D single array...
                 sum = max_ = help[0];
                  ul = dl = u = v = 0;
         for (int M = 1; M < n; M++) {
                     if (sum + help[M] >= help[M]) {
                         sum += help[M];
                         v++;
                     } else {
                         sum = help[M];
                         u = M;
                  v = M;
                     }
                     if (sum > max_) {
                  max_ = sum;
                  ul = u;
                           dl = v;
                     }
                 }
                if (max_sum < max_) {
                     max_sum = max_;
                     ansup = ul;
                     ansdown = dl;
                     ansleft = K;
                     ansright = L;
                 }
             }
         }
         cout << "\n" << max_sum << '\n';
         cout << ansleft << ' ' << ansright << ' ' << ansup
<< ' ' << ansdown << "\n";
}
```

2.3 LCS

```
int lcs(string& a, string& b, int len_a, int len_b) {
        int dp[len_a + 1][len_b + 1];
        int i, j;
        for (i = 0; i <= len_a; i++) {
            for (j = 0; j <= len_b; j++) {
                if (i == 0 || j == 0)
                    dp[i][j] = 0;
                else if (a[i - 1] == b[j - 1])
                    dp[i][j] = dp[i - 1][j - 1] + 1;
                else
                    dp[i][j] = max(dp[i - 1][j], dp[i][j -
1]);
                }
        }
        return dp[len_a][len_b];  // ans...
}
```

2.4 LIS

```
int n;
int a[10010], b[10010];
int bs(int h, int x) {
        int l = 1, ans = 0;
        while (l <= h) {
            int m = (l + h) >> 1;
            if (x > a[b[m]]) {
                ans = m;
                  l = m + 1;
            } else
        h = m - 1;
        }
        return ans + 1;
}
int lis() {
        int len = 0;
        int p[n];
        for (int i = 0; i < n; ++i) {
            int tmp = bs(len, a[i]);
            b[tmp] = i;
            p[i] = b[tmp - 1];
            len = max(tmp, len);
        }
        int k = b[len];
        int s[len];
        for (int i = len - 1; i >= 0; --i) {
            s[i] = a[k];
            k = p[k];
        }
        for (int i = 0; i < len; ++i) cout << s[i] << ' ';  //
        prints answer array.cout << endl;
        return len;  // returns answer length();
}
```

2.5 sum over subset

```
/**
 * Description: Fast sum over subsets $g_i = \sum_{j\subseteq
i} f_j$ and supersets $g_i=\sum_{i\subseteq j} f_j$.
 * Time: $O(n 2^n)$
 */
for (int mask = 0; mask < 1 << n; mask++) f[mask] = a[mask];
// sum over subsets
for (int i = 0; i < n; ++i) {
  for (int mask = 0; mask < 1 << n; ++mask) {
        if (mask & 1 << i) f[mask] += f[mask ^ 1 << i];
  }
}
// sum over supersets
for (int i = 0; i < n; ++i) {
  for (int mask = (1 << n) - 1; mask >= 0; --mask) {
        if (~mask & 1 << i) f[mask] += f[mask | 1 << i];
  }
}
```

3.1 BFS

```
vector<vector<int> > edge(MX + 5);
int bfs(int source, int destination) {
        queue<int> Q;
        vector<int> level(200, -1);
        int u, v;
        Q.push(source);
        level[source] = 0;
        while (!Q.empty()) {
            u = Q.front();
            Q.pop();
            for (int K = 0; K < edge[u].size(); K++) {
                v = edge[u][K];
                  if (level[v] == -1) {
                    level[v] = level[u] + 1;
                    Q.push(v);
                    // can be matched with destination
here to get the distance.
                      // Or, can be calculated new persons or
nodes    comes in
                    // consideration for the first time.
        }
            }
        }
        return distance;
}
```

Uses:

1. Graph traversal
2. Finding connected component [undirected graph]
3. Flood fill - Labelling/colouring the connected components
    3.1. Usually used in implicit-graph/2D-grid
    3.2. dividing grid into colouring to solve problem
4. Topological sort [Directed acyclic graph]
    4.1. Needs a priority list
    4.2. Used in preprocessing step for 'DP on DAG'
5. Bicoloring / Bipartite Graph check
6. Finding cycles
7. Finding Articulation Point/Bridge
8. Finding SCC [directed graph]
    8.1. Used as subproblem involves directed graph that 'requires transformation' to DAG

3.2 DFS

```
vector<vi> edge(n + 1);
vi color(n + 1, 1);
int time = 0;
vi Enter(n + 1, -1), Exit(n + 1, -1);
bool dfs(int u, int destination) {
        int v;
        time++;
        Enter[u] = time;  // Enter time.
        if (u == destination) return 1;
        color[u] = 0;
        // 0 - grey
        for (int K = 0; K < (int)edge[u].size(); K++) {
            v = edge[u][K];
            if (color[v] == 1) dfs(v, destination);  // 1 -
white
        }
        color[u] = -1;
        // -1 - black...
        time++;
        Exit[u] = time;  // exit time.
        return 0;
}
```

3.3 Bicoloring

```
/// Bicoloring using BFS...
/// Note: If a graph contains cycle of odd length, then it
isn't bicolorable.
vector<vector<int> > edge;
bool bicoloring(int n, int source) {
        queue<int> q;
        int u, v;
        q.push(source);
        vector<int> color(n);  // 0 for white... 1 for
red... 2 for blue..
        color[0] = 1;
        // bool flag = 0;
```

```
        while (!q.empty()) {
            u = q.front();
            q.pop();
            for (int K = 0; K < edge[u].size(); K++) {
                v = edge[u][K];
                if (color[v] == 0) {
                    if (color[u] == 1) color[v] = 2;  //
color[u] is red than color[v] will be blue.
                    else color[v] = 1;                //
else color[v] will be red.q.push(v);
                }
                if (color[u] == color[v]) return false;  //
If the color of both source u ans source v ara seme, than
graph isn.t bicolorable..
            }
        }
        return true;
}
```

3.4 Top Sort

```
const int MX = 1e4;
vi ara[MX+5], in_deg(MX+5);
vector<int> order;
void topsort(int n)
{
        queue<int> q;
        // maybe a queue is enough sometimes or need
priority queue???
        order.clear();
        for(int K=1; K<=n; K++){
            if(!in_deg[K]){
                q.push(K);

                 // As want to proccess smaller first
                // multiplying with -1 makes the small
number bigger(!)
            }
        }
        while(!q.empty()){
            int u = q.front();
            q.pop();
            order.PB(u);
            // updating solution list
            for(int K=0; K<sz(ara[u]); K++){
                 in_deg[ara[u][K]]--;
                 if(in_deg[ara[u][K]] == 0){
                     q.push(ara[u][K]);
                     // child node's level is 1 step lower
                     // so have to add 1 to parent node's
level;
                 }
            }
        }
}
```

3.5 Dijkstra

```
struct Node {
        int at, cost;
        Node(int _at, int _cost) {
        at = _at;
        cost = _cost;
        }
};
bool operator<(Node A, Node B) {
        // Priority queue returns the greatest value
        // So we need to write the comparator in a wayy
        // So that cheapest value bacomes greatest value
        return A.cost > B.cost;
}
struct Edge {
        int v, w;
        Edge(int _v, int _w) {
        v = _v;
        w = _w;
        }
};
const int MX = 2e5;
vector<Edge> adj[MX+5];  // adjacency list of weighted graph
priority_queue<Node> PQ;
int dist[MX+5];
int n;
int dijkstra(int source, int dest) {
        memset(dist, -1, sizeof dist);
        dist[source] = 0;
        while (!PQ.empty()) PQ.pop();
        PQ.push(Node(source, 0));

        while (!PQ.empty()) {
        Node u = PQ.top();
        PQ.pop();
        if (u.cost != dist[u.at]) {
        continue;
        }
        for (Edge e : adj[u.at]) {
        if (dist[e.v] == -1 || dist[e.v] > u.cost + e.w) {
                dist[e.v] = u.cost + e.w;
                PQ.push(Node(e.v, dist[e.v]));
        }
        }
        }
        return dist[dest];
}
```

3.5.1 Dijkstra bruteforce

```
const int N = 2e5;
vector < int > d(N + 5, -1), p(N + 5, -1);
vector < pair < int, int >> edge[N + 5];

void dijkstra(int s) {
        d[s] = 0;
        set < pair < int, int > > q;
        q.insert({0, s});

        while (!q.empty()) {
        int u = q.begin() -> second;
        q.erase(q.begin());

                for (auto[to, w]: edge[u]) {
                  if (d[u] + w < d[to]) {
                        q.erase({d[to], to});

                        d[to] = d[u] + w;
                        d[to] = u;

                       q.insert({d[to], to});
                   }
                }
        }
}
```

3.5.2 0-1 BFS

```
const int N = 2e5;
vector<int> d(N+5, -1), p(N+5, -1);
vector<pair<int, int>> edge[N+5];

void dijkstra(int s)
{
        d[s] = 0;
        deque<int> q;
        q.push_back(s);

        while(!q.empty()) {
        int u = q.front();
        q.pop_front();

        for(auto [to, w] : edge[u]) {
                if(d[u] + w < d[to]) {
                d[to] = d[u] + w;

                      if(w == 1) q.push_back(to);
                      else q.push_front(to);
```

```
                    }
            }
            }
}
```

3.6 Bellman Ford

```
// Complexity: O(VE) for Bellman Ford + O(v^2) for checking
negative cycles.
// Not used much as for high complexity.
struct Edge {
          int u, v, w;
          Edge(int _u, int _v, int _w) {
              u = _u;
              v = _v;
              w = _w;
          }
};
vector<Edge> E;  // weighted edge list
int dist[100], n;
int bellman_ford(int s, int dest) {
          for (int i = 1; i <= n; i++) dist[i] = MXI;
          dist[s] = 0;
          for (int i = 1; i < n; i++) {
              for (Edge e : E) {
          if (dist[e.v] > dist[e.u] + e.w) {
                    dist[e.v] = dist[e.u] + e.w;
                    }
              }
          }
          for (Edge e : E) {
              if (dist[e.v] > dist[e.u] + e.w) {
          return -1;
             }
          }
          return dist[dest];
}
```

3.7 Floyd Warshall

```
// dist[] <- Infinity
for (int K = 1; K <= n; K++) ara[K][K] = 0LL;
```

- Sometimes dijkstra can be used to solve these kinds of problems.
- Complexity: Dijkstra: [O(V^3*logV)] && FWA: [V^4], where E = V^2
- Is usually used for fast typing/coding; [For n <= 400]

Uses:

- Solving SCC problems on a small weighted graph. (n <= 400)
- **Printing the shortest path** (a bit different from other path printings)

```
for(int K = 0; K < n; K++){
          for(int L = 0; L < n; L++) p[// dist[] <-
InfinityK][L] = K;
}

for(int M = 0; M < n; M++){
    for(int K = 0; K < n; K++){
        for(int L = 0; L < n; L++){
               if(dist[K][M] + dist[M][L] < dist[K][L]){
                      dist[K][L] = dist[K][M] +
dist[M][L];
                p[K][L] = p[M][L];
            }
        }
    }
}

// path print . .
Void print(int i, int j)
{
    if(i != j) print(i, p[i][j]);
    printf(" %d", j);
}
```

- Transitive Closure (Warshall's Algorithm)
  - Instead of weight in the adjacency matrix, we use 1 for directly connected edges and 0 for directly not connected edges.
  - Then, run floyd warshall algo.
- Minimax and Maximin (Revisited - after MST)
  - The minimum cost AdjMat[K][L] is the minimum of either (itself) or (the maximum between AdjMat[K][M] or AdjMat[M][L])
  - Initially AdjMat[][] = {INF} except the given edges.
- Finding the Cheapest / Negative cycle:
  - If AdjMat[K][K] < INF, then we have a cycle. The minimum value for AdjMat[K][K] is the cheapest cycle
  - If AdjMat[K][K] < 0, then we have a negative cycle.
- Finding the graph diameter
  - Maximum value in AdjMat[][]
- Finding SCC

3.8 Shortest Cycle

```
#include <bits/stdc++.h>
using namespace std;
#define N 100200
vector<int> gr[N];

// Function to add edge
void Add_edge(int x, int y) {
          gr[x].push_back(y);
          gr[y].push_back(x);
}

// Function to find the length of
// the shortest cycle in the graph
int shortest_cycle(int n) {
          // To store length of the shortest cycle
          int ans = INT_MAX;

          // For all vertices
          for (int i = 0; i < n; i++) {
          vector<int> dist(n, (int)(1e9)); // Make distance
maximum
          vector<int> par(n, -1);  // Take a imaginary parent
          dist[i] = 0;       // Distance of source to source
is 0
          queue<int> q;
          q.push(i);         // Push the source element

          while (!q.empty()) { // Continue until queue is not
empty
          // Take the first element
                    int x = q.front();
                    q.pop();
                    for (int child : gr[x]) {
                              // If it is not visited yet
                             if (dist[child] == (int)(1e9)) {
                                  dist[child] = 1 + dist[x]; //
Increase distance by 1
                                   par[child] = x;   // Change
parent
                                  q.push(child);   // Push into
the queue
                             }
                             else if (par[x] != child and
par[child] != x)  // If it is already visited
                                       ans = min(ans, dist[x]
+ dist[child] + 1);
                    }
          }
          }

          if (ans == INT_MAX) return -1;       // If graph
contains no cycle\
          else return ans;   // If graph contains cycle
}
```

3.9 Kruskal Algorithm

```
const int MX = 2e5;
int dsu[MX+5];
int Find(int x)
{
    if(dsu[x] == x) return x;

    return dsu[x] = Find(dsu[x]);
}
void Union(int a, int b)
{
    a = Find(a);
    b = Find(b);

    dsu[b] = a;
}

vector<pair<int, pair<int, int> > > Edges;
int kruskal()
{
    sort(Edges.begin(), Edges.end());
    int E = (int)Edges.size();
    int mst_cost = 0;

    for(int K = 0; K < E; K++){
          if(Find(Edges[K].second.first) !=
Find(Edges[K].second.second)){
                Union(Edges[K].second.first,
Edges[K].second.second);
                mst_cost += Edges[K].first;
          }
    }

    return mst_cost;
}
```

Uses:

1. 'Maximum' spanning tree
2. 'Minimum' spanning subgraph [some edges are given fixed from the start]
3. Minimum spanning forest
4. Second best spanning tree :O(VE)
5. minimax/maximin :(build MST + graph traversal)

Notes:

- Not too many variations
- Hard to find weather it's an MST problem (so the main task is to find we can solve the problem using MST)
- For future: Arborescence problem, Steiner tree, degree constrained MST, k-MST etc.

3.10 Prim's Algorithm

```
vi taken;                    // global boolean flag to avoid
cycle
priority_queue<ii> pq;  // priority queue to help choose
shorter edges
// note: default setting for C++ STL priority_queue is a max
heap
void process(int vtx) {  // so, we use -ve sign to reverse
the sort order
          taken[vtx] = 1;
          for (int j = 0; j < (int)AdjList[vtx].size(); j++)
{
          ii v = AdjList[vtx][j];
          if (!taken[v.first]) pq.push(ii(-v.second, -
v.first));
          }
}  // sort by (inc) weight then by (inc) id
// inside int main()---assume the graph is stored in AdjList,
pq is empty
taken.assign(V, 0);  // no vertex is taken at the beginning
process(0);        // take vertex 0 and process all edges
incident to vertex 0
mst_cost = 0;
while (!pq.empty()) {  // repeat until V vertices (E=V-1
edges) are taken
          ii front = pq.top();
          pq.pop();
          u = -front.second, w = -front.first;  // negate the
id and weight again
          if (!taken[u])                        // we have not
connected this vertex yet
          mst_cost += w, process(u);            // take u,
process all edges incident to u
}  // each edge is in pq only once!
printf("MST cost = %d (Prim's)\n", mst_cost
```

3.11 Centroid Decomposition

```
// dis[l][u] denotes the distance in original tree of node u
to the anscentor of u that is at depth l in centroid tree
const long long MX = 1e5 + 10;
const long long MX1 = 20;
VI g[MX];
int lvl[MX], sz[MX], p[MX], dis[MX1][MX];
void cal_subtree_size(int u, int pu) {
          sz[u] = 1;
          for (int v : g[u]) {
          if (v != pu && lvl[v] == -1) {
          cal_subtree_size(v, u);
          sz[u] += sz[v];
          }
          }
}
int find_centroid(int u) {
          for (int v : g[u]) {
          if (lvl[v] == -1 && sz[v] > sz[u] / 2) {
          sz[u] -= sz[v];
          sz[v] += sz[u];
          return find_centroid(v);
          }
          }
          return u;
}
void cal_dis(int u, int pu, int l) {
          for (int v : g[u]) {
          if (v != pu && lvl[v] == -1) {
          dis[l][v] = dis[l][u] + 1;
          cal_dis(v, u, l);
          }
          }
}
int build_centroid_tree(int root, int l) {
          cal_subtree_size(root, -1);
          int croot = find_centroid(root);
          lvl[croot] = l;
          dis[l][croot] = 0;
          cal_dis(croot, -1, l);

          for (int v : g[croot]) {
          if (lvl[v] == -1) {
          int cv = build_centroid_tree(v, l + 1);
          p[cv] = croot;
          }
          }
          return croot;
}
void centroid_decomposition(int n, int root) {
          loop(i, 0, n + 1) lvl[i] = -1;
          int croot = build_centroid_tree(root, 0);
          p[croot] = -1;
}
```

3.12 Flow

```
// Edmonds-Karp algorithm: O(VE^2)
int n;
vector<vector<int>> capacity;
vector<vector<int>> adj;

int bfs(int s, int t, vector<int>& parent) {
          fill(parent.begin(), parent.end(), -1);
          parent[s] = -2;
          queue<pair<int, int>> q;
```

```
        q.push({s, INF});

        while (!q.empty()) {
        int cur = q.front().first;
        int flow = q.front().second;
        q.pop();

            for (int next : adj[cur]) {
                if (parent[next] == -1 &&
capacity[cur][next]) {
                            parent[next] = cur;
                            int new_flow = min(flow,
capacity[cur][next]);
                            if (next == t)
                  return new_flow;
                            q.push({next, new_flow});
                            }
                }
        }

        return 0;
}

int maxflow(int s, int t) {
        int flow = 0;
        vector<int> parent(n);
        int new_flow;

        while (new_flow = bfs(s, t, parent)) {
            flow += new_flow;
            int cur = t;
            while (cur != s) {
                int prev = parent[cur];
                capacity[prev][cur] -= new_flow;
                capacity[cur][prev] += new_flow;
                cur = prev;
            }
        }

        return flow;
}
```

3.13 SCC

```
# Not Tested
/// Finding SCC
// For directed edges
// Need a reverse edge graph
const int MX = 1e5;
vector<int> edge[MX+5], reverse_edge[MX+5], component[MX+5];
bool vis[MX+5], color[MX+5];
stack<int> stk;

void dfs(int u)
{
    color[u] = 1;

    for(int K = 0; K < (int)edge[K].size(); K++){
        if(color[edge[u][K]] == 0) dfs(edge[u][K]);
    }
    stk.push(u);
}

void dfs2(int u, int mark)
{
    component[mark].push_back(u);
    vis[u] = 1;

    for(int K = 0; K < (int)reverse_edge[u].size(); K++){
        if(!vis[reverse_edge[u][K]])
dfs2(reverse_edge[u][K], mark);
    }
}

int find_scc(int n)
{
    int mark = 0;

    for(int K = 1; K <= n; K++){
        if(color[K] == 0) dfs(K);
    }
    while(!stk.empty()){
        int u = stk.top();
        stk.pop();

        if(vis[u] == 0){
                mark++;
                dfs2(u, mark);
        }
    }
    // We get components as result
    return mark; // Number for SCC subgraph.
}
```

3.14 2-SAT

```
// Usage:
// TwoSat two_sat(n);
// two_sat.either(i, j); // i, j both 1-indexed
// two_sat.either(i, -j); // i, j both 1-indexed
// two_sat.solve(); // solves and returns true if solution
exists
// two_sat.values; // get assignments in the solution

struct TwoSat {
  int n;
  vector<vector<int>> adj;
  vector<int> values; // 0 = false, 1 = true

  TwoSat(int n = 0) : n(n), adj(2 * n) {}

  void either(int i, int j) {
        i = 2 * (abs(i) - 1) + (i < 0);
        j = 2 * (abs(j) - 1) + (j < 0);
        adj[i ^ 1].push_back(j);
        adj[j ^ 1].push_back(i);
  }

  vector<int> enter, comp, curr_comp;
  int time = 0;
  int dfs(int at) {
        int low = enter[at] = ++time;
        curr_comp.push_back(at);
        for (int to : adj[at]) {
        if (!comp[to]) {
        low = min(low, enter[to] ? enter[to] : dfs(to));
        }
        }
        if (low == enter[at]) {
        int v;
        do {
        v = curr_comp.back();
        curr_comp.pop_back();
        comp[v] = low;
        if (values[v >> 1] == -1) values[v >> 1] = !(v &
1);
        } while (v != at);
        }
        return enter[at] = low;
  }

  bool solve() {
        values.assign(n, -1);
        enter.assign(2 * n, 0);
        comp = enter;
        for (int i = 0; i < 2 * n; i++) {
        if (!comp[i]) dfs(i);
        }
        for (int i = 0; i < n; i++) {
        if (comp[2 * i] == comp[2 * i + 1]) return false;
        }
        return true;
  }
};
```

3.15 Maximum bipartite matching (root V*E)

```
const int mx=1e4+10;
vector<int>adj[mx];
map<pair<int,int>,int>cost;
int pairU[mx],pairV[mx],dist[mx],a,b;
bool bfs()
{
   queue<int> Q;
   for (int u=1; u<=a; u++)
   {
      if (pairU[u]==0)
      {
         dist[u] = 0;
         Q.push(u);
      }
      else dist[u] = inf;
   }
   dist[0] = inf;
   while (!Q.empty())
   {
      int u = Q.front();
      Q.pop();
      if (dist[u] < dist[0])
      {
         for (auto it : adj[u])
         {
            int v = it;
            if (dist[pairV[v]] == inf)
            {
               dist[pairV[v]] = dist[u] + 1;
               Q.push(pairV[v]);
            }
         }
      }
   }
   return (dist[0] != inf);
}
bool dfs(int u)
{
   if (u != 0)
   {
      for (auto it : adj[u])
         {
            int v = it;
         if (dist[pairV[v]] == dist[u]+1)
         {

            if (dfs(pairV[v]) == true)
            {
               pairV[v] = u;
               pairU[u] = v;
               return true;
            }
         }
      }
      return false;
   }
   return true;
}
```

```
int hopcroftKarp()
{
   for (int u=0; u<=a; u++)
      pairU[u] = 0;
   for (int v=0; v<=a; v++)
      pairV[v] = 0;
   int result = 0;
   while (bfs())
   {
      for (int u=1; u<=a; u++)

         if (pairU[u]==0 && dfs(u)){
            result++;
         }
   }
   return result;
}
main()
{
   cin>>a>>b;
   int x,y;
   for(int i=0;i<b;i++){
      cin>>x>>y;
      adj[y].push_back(x);
      int cost;
      cin>>cost;
      cost[{y,x}]=cost;
   }
   int ans=hopcroftKarp();
}
```

3.16 Min cost max flow (V^3*E)

```
struct Edge
{
   int from, to, capacity, cost;
};
vector<vector<int>> adj, cost, capacity;
const int INF = 1e9;
void shortest_paths(int n, int v0, vector<int>& d, vector<int>& p) {
   d.assign(n, INF);
   d[v0] = 0;
   vector<bool> inq(n, false);
   queue<int> q;
   q.push(v0);
   p.assign(n, -1);

   while (!q.empty()) {
      int u = q.front();
      q.pop();
      inq[u] = false;
      for (int v : adj[u]) {
         if (capacity[u][v] > 0 && d[v] > d[u] + cost[u][v]) {
            d[v] = d[u] + cost[u][v];
            p[v] = u;
            if (!inq[v]) {
               inq[v] = true;
               q.push(v);
            }
         }
      }
   }
}
int min_cost_flow(int N, vector<Edge> edges, int K, int s, int t) {
   adj.assign(N, vector<int>());
   cost.assign(N, vector<int>(N, 0));
   capacity.assign(N, vector<int>(N, 0));
   for (Edge e : edges) {
      adj[e.from].push_back(e.to);
      adj[e.to].push_back(e.from);
      cost[e.from][e.to] = e.cost;
      cost[e.to][e.from] = -e.cost;
      capacity[e.from][e.to] = e.capacity;
   }

   int flow = 0;
   int cost = 0;
   vector<int> d, p;
   while (flow < K) {
      shortest_paths(N, s, d, p);
      if (d[t] == INF)
         break;

      // find max flow on that path
      int f = K - flow;
      int cur = t;
      while (cur != s) {
         f = min(f, capacity[p[cur]][cur]);
         cur = p[cur];
      }
      // apply flow
      flow += f;
      cost += f * d[t];
      cur = t;
      while (cur != s) {
         capacity[p[cur]][cur] -= f;
         capacity[cur][p[cur]] += f;
         cur = p[cur];
      }
   }
   if (flow < K)
      return -1;
   else
      return cost;
}
```

3.17 Hungarian algorithm

```
/**
   Hungarian algorithm for minimum weighted bipartite matching. (1-indexed)
   For max cost, negate cost matrix and negate output.
   Complexity: O(n^2 m). n must not be greater than m.

   Input: (n+1) x (m+1) cost matrix. (0th row and column are useless)
   Output: (ans, ml), where ml[i] = match for node i on the left.
*/
#include<bits/stdc++.h>
using namespace std;

template<typename T>
pair<T, vector<int>> Hungarian(const vector<vector<T>> &cost){
```

```
  const T INF = numeric_limits<T>::max();
  int n = cost.size()-1,  m = cost[0].size()-1;
  vector<T> U(n+1), V(m+1);
  vector<int> mr(m+1), way(m+1), ml(n+1);

  for(int i = 1; i<=n; i++){
    mr[0] = i;
    int lastJ = 0;
    vector<T> minV(m+1, INF);
    vector<bool> used(m+1);
    do{
      used[lastJ] = true;
      int lastI = mr[lastJ], nextJ;
      T delta = INF;
      for(int j = 1; j<=m; j++){
        if(used[j]) continue;
        T diffCost = cost[lastI][j] - U[lastI] - V[j];
        if(diffCost < minV[j]) minV[j] = diffCost, way[j] = lastJ;
        if(minV[j] < delta) delta = minV[j], nextJ = j;
      }
      for(int j = 0; j<=m; j++){
        if(used[j]) U[mr[j]] += delta, V[j] -= delta;
        else       minV[j] -= delta;
      }
      lastJ = nextJ;
    } while(mr[lastJ] != 0);
    do{
      int prevJ = way[lastJ];
      mr[lastJ] = mr[prevJ];
      lastJ = prevJ;
    } while(lastJ != 0);
  }
  for (int i=1; i<=m; i++)    ml[mr[i]] = i;
  return {-V[0], ml} ;
}
```

3.18 Articulation Point

```
struct articulation_point
{
          int n; // number of nodes
          vector<vector<int>> adj; // adjacency list of graph
          vector<bool> visited;
          vector<int> tin, low;
          vector<int> ans;
          int timer;
          articulation_point (int N, vector<vector<int>> v) {
            n = N;
            adj = v;
            visited.resize(n + 5, false);
            tin.resize(n + 5, 0); low.resize(n + 5, 0);
            timer = 0;
          }
          void IS_CUTPOINT(int a) {
            // process the fact that vertex A is an
articulation point
            ans.PB(a);
          }
          void dfs(int v, int p = -1) {
            visited[v] = true;
            tin[v] = low[v] = timer++;
            int children = 0, point = 0;
            for (int to : adj[v]) {
              if (to == p) continue;
              if (visited[to]) {
                    low[v] = min(low[v], tin[to]);
              } else {
                    dfs(to, v);
                    low[v] = min(low[v], low[to]);
                    if (low[to] >= tin[v] && p!=-1)
                    point = 1;// this can be called multiple
time for each vertex
                    ++children;
              }
            }
            if(p == -1 && children > 1)
            point = 1;
            if(point) IS_CUTPOINT(v);
          }
          void find_cutpoints() {
            timer = 0;
            visited.assign(n + 5, false);
            tin.assign(n + 5, -1);
            low.assign(n + 5, -1);
            for (int i = 0; i < n; ++i) {
              // check index 0 or 1
              if (!visited[i])
                    dfs(i);
            }
          }
};
```

3.19 Articulation Bridge

```
struct bridge
{
          int n; // number of nodes
          vector<vector<int>> adj; // adjacency list of graph
          vector<bool> visited;
          vector<int> tin, low;
          vector<pii> edge;
          int timer;
          bridge (int N, vector<vector<int>> v) {
            n = N;
            adj = v;
          }
          void IS_BRIDGE(int x, int y) {
            edge.PB({x, y});
          }
          void dfs(int v, int p = -1) {
            visited[v] = true;
            tin[v] = low[v] = timer++;
            for (int to : adj[v]) {
            if (to == p) continue;
            if (visited[to]) {
                        low[v] = min(low[v], tin[to]);
            } else {
                        dfs(to, v);
                        low[v] = min(low[v], low[to]);
                        if (low[to] > tin[v])
                          IS_BRIDGE(v, to);
              }
            }
          }
          void find_bridge() {
            timer = 0;
            visited.assign(n + 5, false);
            tin.assign(n + 5, -1);
            low.assign(n + 5, -1);
            for (int i = 0; i < n; ++i) {
            if (!visited[i])
                        dfs (i);
          }
          }
 };
```

4.1 Segment Tree

```
// Some notes:
// 1. Check tree[] size .
// 2. Check the range for build(1, 1, n).
const int MX = 1e5;
int sum[4 * MX + 5], num[MX + 5];
// keep num size 4*n;
/// Building segment trees...
void build(int at, int l, int r) {
          sum[at] = 0;
          if (l == r) {
          // keeping the sum...
          // Or, maybe other queries...
          sum[at] = num[l];
          return;
          }
          int mid = (l + r) / 2;
          build(at * 2, l, mid);
          build(at * 2 + 1, mid + 1, r);
          // keeping the sum...
          // Or, maybe other queries...
          sum[at] = sum[at * 2] + sum[at * 2 + 1];
}
/// Updating segment tree...
void update(int at, int l, int r, int idx, int num) {
```

```
        if (l == r) {
        sum[at] += num;
        return;
        }
        int mid = (l + r) / 2;
        // instead of if-else
        // we can use
        // if(pos < L || R < pos) return;
        if (pos <= mid)
        update(at * 2, l, mid, idx, num);
        else
        update(at * 2 + 1, mid + 1, r, idx, num);
        sum[at] = sum[at * 2] + sum[at * 2 + 1];
}
/// Making queries in segment tree...
int query(int at, int l, int r, int L, int R) {
        // query from l to r.
        if (R < l || L > r) return 0;
        if (L <= l && r <= R) return sum[at];
        int mid = (l + r) / 2;
        int x = query(at * 2, l, mid, L, R);
        int y = query(at * 2 + 1, mid + 1, r, L, R);
        return x + y;
}
int main()
{
    int n, u, q, pos, value, l, r;
    // Input data(s).
    cin >> n;
    for(int K=1; K<=n; K++) cin >> num[K];

    // build...
    build(1, 1, n);

    // update...
    # handle indexes for 0 based ones. ( This Tamplate is 1
based...)
    # consider overflow for sum.
    cin >> u;
    while(u--){
        cin >> pos >> value;
        update(1, 1, n, pos, value);
    }
    // queries...
    cin >> q;
    while(q--){
        cin >> l >> r;
        cout << query(1, 1, n, l, r) << "\n";
    }
    return 0;
}
```

4.2 Lazy

```
ll seg[400005], a[100005], lazy[400005];
// can lazy be zero by default ,check question
// if we have multiple test case then check seg,a,lazy array
// 0 index or 1 index for query and update

void build(int pos, int l, int r) {
        lazy[pos] = 0;
        if (l == r) {
        seg[pos] = a[l];
        return;
        }
        ll mid = (l + r) >> 1;
        build(2 * pos, l, mid);
        build(2 * pos + 1, mid + 1, r);
        seg[pos] = seg[2 * pos] + seg[2 * pos + 1];
}
void lazy_update(int pos, int l, int r) {
        seg[pos] += (r - l + 1) * lazy[pos];
        if (l != r) {
        lazy[2 * pos] += lazy[pos];
        lazy[2 * pos + 1] += lazy[pos];
        }
        lazy[pos] = 0;
}
void update(int pos, int l, int r, int x, int y, ll val) {
        if (lazy[pos] != 0) lazy_update(pos, l, r);

        if (l > y || x > r) return;
        if (l >= x && r <= y) {
        lazy[pos] += val;
        lazy_update(pos, l, r);
        return;
        }
        int mid = (l + r) >> 1;
        update(2 * pos, l, mid, x, y, val);
        update(2 * pos + 1, mid + 1, r, x, y, val);
        seg[pos] = seg[2 * pos] + seg[2 * pos + 1];
}
ll query(int pos, int l, int r, int x, int y) {
        if (lazy[pos] != 0) lazy_update(pos, l, r);

        if (l > y || x > r) return 0;
        if (l >= x && r <= y) return seg[pos];
        int mid = (l + r) >> 1;
        ll q1 = query(2 * pos, l, mid, x, y);
        ll q2 = query(2 * pos + 1, mid + 1, r, x, y);
        return q1 + q2;
}
```

4.3 BIT / Fenwick Tree

```
// copied ...
int n, a[MX], tree[MX]; //tree is 1-indexed
void update(int idx, int val) {//add val to index idx
        while(idx <= n){
            tree[idx] += val;
            idx += (idx & -idx);
        }
}
int query(int idx) {//returns sum from index 1 to index idx
        int sum = 0;
        while(idx) {
            sum += tree[idx];
            idx -= (idx & -idx);
        }
        return sum;
}
int query(int idi, int idj) {//return sum from index idi to
index idj
        int sum = 0;
        while(idi <= idj) {
            sum += tree[idj];
            idj -= (idj & -idj);
        }
        idi--;
        while(idi != idj) {
            sum -= tree[idi];
            idi -= (idi & -idi);
        }
        return sum;
}
int bS(int sum) {//normal binary search, return the index for
which element 1-n sum equals this parameter 'sum'
        int idx = 0;
        while(bitMask != 0) {
            int mid = idx + bitMask;
            if(mid <= n) {
                if(tree[mid] == sum) {
                  return mid;
                }
                else if(tree[mid] < sum) {
                  sum -= tree[mid];
                  idx = mid;
                }
            }
            bitMask >>= 1;
        }
        if(sum != 0) return -1; //not found
        return idx;
}
int bS(int sum) {//returns the greatest index which equals sum
        int idx = 0;
        while(bitMask != 0) {
            int mid = idx + bitMask;
            if(mid <= n && (tree[mid] <= sum)) {
                sum -= tree[mid];
```

```
                idx = mid;
            }
            bitMask >>= 1;
        }
        if(sum != 0) return -1; //not found return idx;
}
```

4.4 DSU

```
// code 1
const int MX = 2e5;
int dsu[MX+5], sz[MX+5];
// sz[] is initialized by value 1;
// dsu[] if initialized by dsu[K] = K

int Find(int x)
{
    if(dsu[x] == x) return dsu[x];
    return dsu[x] = Find(dsu[x]);
}
void Union(int a, int b)
{
    a = Find(a);
    b = Find(b);

    if(a != b){
        if(sz[a] > sz[b]) swap(a, b);

        dsu[a] = b;
        sz[b] += sz[a];
    }
}
```

```
 // code 2
 void init(int N) {
          for (int i = 0; i < N; ++i) {
          pre[i] = i;
          height[i] = 0;
          }
 }

 int find(int node) {
          if (pre[node] != node) {
                   pre[node] = find(pre[node]);
          }
          return pre[node];
 }

 void unite(int A, int B) {
          A = find(A);
          B = find(B);
          if (A == B) return;
          if (height[A] > height[B]) {
          pre[B] = A;
          height[A] = max(height[A], height[B] + 1);
          } else {
          pre[A] = B;
          height[B] = max(height[B], height[A] + 1);
          }
}
```

Smaller to larger (DSU on tree)

Trees are a very special kind of graph and many procedures on trees can be optimized. One of these optimizations is called “Smaller to Larger”, also known as “Dsu on Trees”.A way to merge two sets efficiently.

let's prove that there's at most O(N log N) moves, each move can be done in O(log N) so total complexity is O(N log^2 N)

when you are merging two sets you move elements from smaller set(assume its size is K) to bigger one, so every element in the smaller set now is in a set with at least size 2K

In other words every element that has been moved T times is in a set with at least size 2^T , thus every element will be moved at most O(log N) and we have N elements so in total there will be at most O(N log N) move operations.

Here ,duplicate values aren’t considered because we are assuming the worst case scenario.If we found any duplicate value then that node will not be counted in further operation.So, duplicate values will reduce complexity.
Complexity : $O(nlog^2n)$

4.5 LCA

```
const int N= 1e5+5,LOG= 20;
int depth[N],up[N][LOG];
vector<int> v[N];
void dfs(int pos,int pre)
{
   for(auto it:v[pos])
   {
      if(it==pre) continue;
      depth[it]=depth[pos]+1;
      up[it][0] = pos;
      for(int j = 1; j < LOG; j++)
      {
        up[it][j] = up[up[it][j-1]][j-1];
      }
      dfs(it,pos);
   }
   return ;
}
```

```
int kthancestor(int pos,int k)
{
    for(int i=LOG-1;i>=0;i--)
    {
        if(k&(1<<i))
        pos=up[pos][i];
    }
    return pos;
}
int get_lca(int a, int b)
{
    if(depth[a] < depth[b]) {
        swap(a, b);
    }
    // 1) Get same depth.
    int k = depth[a] - depth[b];
    a=kthancestor(a,k);
    // 2) if b was ancestor of a then now a==b
    if(a == b) {
        return a;
    }
    // 3) move both a and b with powers of two
    for(int j = LOG - 1; j >= 0; j--) {
        if(up[a][j] != up[b][j]) {
            a = up[a][j];
            b = up[b][j];
        }
    }
    return up[a][0];
}
```

4.6 Articulation Bridge / Point

```
void dfsCut(int par, int u) {
         low[u] = dfstime[u] = ++cnt;
         for (auto v : adj[u]) {
             if (dfstime[v] == 0) {
               if (u == dfsroot) rc++;
               dfsCut(u, v);
               if (low[v] >= dfstime[u]) cutnode[u] = true;
               if (low[v] > dfstime[u]) brdg.emplace_back(u,
v);
                   low[u] = min(low[u], low[v]);
               } else if (v != par) {
                   low[u] = min(low[u], dfstime[v]);
               }
         }
}
int main() {
         cnt = 0;
         cutnode.assign(n + 2, 0);
```

```
        for (int i = 1; i <= n; i++) {
            if (dfstime[i] > 0) continue;
            dfsroot = i;
            rc = 0;
            dfsCut(-1, i);
            cutnode[dfsroot] = (rc > 1);
        }
}
```

4.7 Sparse Table

```
// sparse table
const int MX = 2e5;
const int lg = 20;
int spt[lg+1][MX+5], ara[MX+5];

int n;

void build_spt()
{
        for(int K = 0; K < n; K++) spt[0][K] = ara[K];

        for(int K = 1; K < lg; K++){
            for(int L = 0; L < n; L++){
                if(L+(1<<K) > n) break;
                spt[K][L] = spt[K-1][L] + spt[K-
1][L+(1<<(K-1))];
            }
        }
}
int get(int l, int r)
{
        int ans = 0;
        for(int K = lg; K >= 0; K--){
            if(l+(1<<K)-1 <= r){
                ans += spt[K][l];
                l += (1<<K);
            }
        }
        // For idempodent functions, we can calculate it in
O(n). then, ans = gcd(spt[K-1][l], spt[K-1][l+(1<<(K-1))]);
        return ans;
}
```

4.8 Matrix Multiplication

```
vector<vl> matrix_multi(vector<vl> a, vector<vl> b, int n) {
        vector<vl> ans(n + 5, vl(n + 5));
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                ll sum = 0;
                for (int k = 0; k < n; k++) {
                    sum = (sum + a[i][k] * b[k][j]) % mod;
                }
                ans[i][j] = sum;
            }
        }
        return ans;
}
```

4.9 Persistant Segment Tree

```
namespace path_copying_segree {
int L[N * LOGN], R[N * LOGN], ST[N * LOGN], blen, root[N];
// sparse persistent-segtree. range sum, initially 0
int update(int pos, int add, int l, int r, int id) {
        if (l > pos || r <= pos) return id;
        int ID = ++blen, m = l + (r - l) / 2;
        if (l == r - 1) return (ST[ID] = ST[id] + add, ID);
        L[ID] = update(pos, add, l, m, L[id]);
        R[ID] = update(pos, add, m, r, R[id]);
        return (ST[ID] = ST[L[ID]] + ST[R[ID]], ID);
}
void update(int A[], int n) {
        root[0] = ++blen;
        for (int i = 1; i <= n; i++) root[i] = update(A[i],
1, 1, MX, root[i - 1]);
}
int query(int qL, int qR, int l, int r, int x) {
        if (!x || r <= qL || qR <= l) return 0;
        if (l >= qL && r <= qR) return ST[x];
        int m = l + (r - l) / 2;
        return query(qL, qR, l, m, L[x]) + query(qL, qR, m,
r, R[x]);
}
int query(int l, int r, int k) {
    return query(k + 1, MX, 1, MX, root[r]) - query(k + 1,
MX, 1, MX, root[l - 1]); }
}  // namespace path_copying_segree

// code 2
const int mx_q = 2e5;
const int N = 2e5;
int ara[N + 5];
struct node {
        int val;
        node *left, *right;
        node(int a = 0, node *b = NULL, node *c =
NULL): val(a), left(b), right(c) {}
        void build(int l, int r) {
            if (l == r) { val = ara[l]; return; }
            left = new node();
            right = new node();

            int mid = (l + r) / 2;
            left->build(l, mid);
            right->build(mid + 1, r);
            val = left->val + right->val;
        }

        node *update(int l, int r, int idx, int v) {
            if (idx < l || r < idx) return this;
            if (l == r) {
                node *ret = new node(val, left,
right);
                ret->val += v;
                return ret;
            }
            int mid = (l + r) / 2;
            node *ret = new node(val);
            ret->left = left->update(l, mid, idx, v);
            ret->right = right->update(mid + 1, r,
idx, v);

            ret->val = ret->left->val + ret->right-
>val;
            return ret;
        }

        int query(int l, int r, int L, int R) {
            if (r < L || R < l) return 0;
            if (L <= l && r <= R) return val;
            int mid = (l + r) / 2;
            return left->query(l, mid, L, R) + right-
>query(mid + 1, r, L, R);
        }
} * root[mx_q + 5];

int main() {
        int n, idx, val;
        cin >> n;
        for (int K = 1; K <= n; K++) cin >> ara[K];

        root[0] = new node();
        root[0]->build(1, n);

        cin >> idx >> val;
        root[1] = root[0]->update(1, n, idx, val);
```

```
// for creating a new version of tree
        // root[0] = root[0] -> update(1, n, idx,
val); // for updating a old version of tree

        cout << root[0]->query(1, n, 1, 5) << "\n";
        cout << root[0]->query(1, n, 5, 10) << "\n";
        cout << root[1]->query(1, n, 1, 5) << "\n";
        cout << root[1]->query(1, n, 5, 10) << "\n";
        return 0;
}
```

4.10 Sliding window range min structure

```
struct MinQ {
         int window;
         deque<pair<int, int> > dq;
         MinQ(int x) { window = x; }
         void push(int idx, int x) {
              while (!dq.empty() && dq.back().first >= x)
dq.pop_back();
              dq.push_back(make_pair(x, idx));
         }
         int qry(int idx) {
             while (idx - dq.front().second >= window)
dq.pop_front();
              return dq.front().first;
         }
};
/* MinQ a(x);x=window size

a.push(i,x);inserting x at idx=i
i must be sequentiallay added;//1 2 3 like this
a.qry(i)=min value in range(i-window+1,i);
if(a.dq.size()==k)range(i-window+1,i) e sob already sorted
,size neyar age pop korte qry koro
vector<MinQ>a(M,window);Can be used like this
*/
```

4.11 Mo's Algorithm

```
const int MX = 2e5;   // query size . . .
const int MXI = 1e6;  // maximum value in array . . .
ll cnt[MXI+5];
ll block_size, range_ans;

struct Query{
    int id, l, r;
    Query() {}
    Query(int _id, int _l, int _r){
        id = _id;
        l = _l;
        r = _r;
    }
    bool operator<(Query &other) const{
        int curr_size = l/block_size;
        int other_size = other.l/block_size;

        if(curr_size == other_size) return r < other.r;
        return curr_size < other_size;
    }
} query[MX+5];

void add(ll x)
{
    if(cnt[x]) range_ans -= cnt[x]*cnt[x]*x;
    cnt[x]++;
    range_ans += cnt[x]*cnt[x]*x;
}
void rmv(ll x)
{
    range_ans -= cnt[x]*cnt[x]*x;
    cnt[x]--;
    if(cnt[x]) range_ans += cnt[x]*cnt[x]*x;
}
ll get_solution()
{
    return range_ans;
}

void mos_algo()
{
    int n, t, l, r;
    cin >> n >> t;
    // setting block size ...
    block_size = sqrt(n);

    int ara[n];
    for(int K = 0; K < n; K++) cin >> ara[K];
    for(int K = 0; K < t; K++){
        cin >> l >> r;
        query[K] = Query(K, l-1, r-1); // 0-based indexing
. . .
    }
    sort(query, query+t);

    int L = 0, R = -1;
    ll v[t];
    for(int K = 0; K < t; K++){
        l = query[K].l;
        r = query[K].r;
        while(L < l){
            rmv(ara[L]);
            L++;
        }
        while(L > l){
            L--;
            add(ara[L]);
        }
        while(R < r){
            R++;
            add(ara[R]);
        }
        while(R > r){
            rmv(ara[R]);
            R--;
        }
        v[query[K].id] = get_solution();
    }
    for(int K = 0; K < t; K++) cout << v[K] << "\n";
}
```

4.12 Sqrt Decomposition

```
// 0-based indexing . . .
int main()
{
        int n;
        cin >> n;
        int block_size = sqrt(n);
        int ara[n], block_ans[n] = {0};
        for(int K = 0; K < n; K++){
            cin >> ara[K];
            block_ans[K/block_size] += ara[K];
        }

        int q, l, r;
        cin >> q;
        while(q--){
            cin >> l >> r;
            l--;
            r--;
            // query operation
            int block_l = l/block_size;
            int block_r = r/block_size;
            int ans = 0;
            if(block_l == block_r){
                for(int K = l; K <= r; K++) ans += ara[K];
            }
            else{
                for(int K = l; K < (block_l+1)*block_size;
K++) ans += ara[K];
                for(int K = block_l+1; K < block_r; K++)
ans += block_ans[K];
                for(int K = block_r*block_size; K <= r; K++)
```

```
ans += ara[K];
            }
            cout << ans << "\n";
        }
        return 0;
}
```

4.13 Trie

```
struct Trie{
        const int A = 26;
        int N;
        vector<vector<int> > next;
        vector<int> cnt;

        Trie() : N(0) {
                node();
        }
        int node(){
                next.emplace_back(A, -1);
                cnt.emplace_back(0);
                return N++;
        }

        inline int get(char c){
                return c-'a';
        }
        inline void insert(string s){
                int cur = 0;
                for(char c : s){
                        int to = get(c);
                        if(next[cur][to] == -1)
next[cur][to] = node();
                        cur = next[cur][to];
                }
                cnt[cur]++;
        }
        inline bool find(string s){
                int cur = 0;
                for(char c : s){
                        int to = get(c);
                        if(next[cur][to] == -1) return
false;
                        cur = next[cur][to];
                }
                return cnt[cur]!=0;
        }
        // Doesn't check for existance
        inline void erase(string s){
                int cur = 0;
                for(char c : s){
                        int to =get(c);
                        cur = next[cur][to];
                }
                cnt[cur]--;
        }
        vector<string> dfs(){
        stack<pair<int, int> > st;
        string s;
        vector<string> ret;
            for(st.push({0, -1}), s.push_back('$');
!st.empty(); ){
                auto [u,  c] = st.top();
                st.pop();
                s.pop_back();
                c++;
                if(c <  A){
                st.push({u, c});
                s.push_back(c+'a');
                int v = next[u][c];
                if(~v){
                if(cnt[v]) ret.emplace_back(s);
                st.push({v, -1});
                s.push_back('$');
                }
                }
                }
                return ret;
        }
};
```

4.14 Implicit Segment Tree

```
// Implicit/Dynamic Segment tree
// Stress Test : "Passed"
// Call it by: Vertex* node = new Vertex(1, n);

struct Vertex{
        int left, right;
        ll sum = 0;
        Vertex *left_child = nullptr, *right_child =
nullptr;

        Vertex(){}
        Vertex(int lb, int rb){
            left = lb;
            right = rb;
        }

        void extend(){
            if(!left_child && left < right){
                int mid = (left+right)/2;
                  left_child = new Vertex(left, mid);
                  right_child = new Vertex(mid+1, right);
            }
        }
        void add(int idx, int x){
            if(left == right){
                  sum += x;
                  return;
            }
            extend();
            if(left_child){
                  if(idx <= left_child->right) left_child-
>add(idx, x);
                  else right_child->add(idx, x);
            }
            sum = left_child->sum + right_child->sum;
        }

        ll get_sum(int lq, int rq){
            if(lq <= left && right <= rq) return sum;
            if(rq < left || lq > right) return 0;
            extend();
            return left_child->get_sum(lq, rq) +
right_child->get_sum(lq, rq);
        }
};
```

4.15 BIT 2D

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

class BIT {
public:
  int n;
  vector<int> f;
  BIT(int n) : n(n) {
        f.resize(n + 1, 0);
  }
  void update(int p, int v) {
        while (p <= n) f[p] += v, p += p & -p;
  }
  int pref(int p) {
        p++;
        int ret = 0;
        while (p) ret += f[p], p -= p & -p;
        return ret;
  }
  int range(int l, int r) {
        return pref(r) - pref(l - 1);
  }
};
class BIT2D {
public:
  int n, m;
```

```
  vector<BIT> f;
  BIT2D(int n, int m) : n(n) {
        f.resize(n + 1, BIT(m));
  }
  void update(int p, int q, int v) {
        while (p <= n) f[p].update(q, v), p += p & -p;
  }
  int pref(int p, int q) {
        int ret = 0;
        while (p) ret += f[p].pref(q), p -= p & -p;
        return ret;
  }
  int range(int x1, int y1, int x2, int y2) {
        return pref(x2, y2) - pref(x2, y1 - 1) - pref(x1 - 1,
y2) + pref(x1 - 1, y1 - 1);
  }
};
```

4.16 Suffix array & lcp

```
vector<int> sort_cyclic_shifts(string const& s) {
  int n = s.size();
  const int alphabet = 256;
  vector<int> p(n), c(n), cnt(max(alphabet, n), 0);
  for (int i = 0; i < n; i++)
    cnt[s[i]]++;
  for (int i = 1; i < alphabet; i++)
    cnt[i] += cnt[i-1];
  for (int i = 0; i < n; i++)
    p[--cnt[s[i]]] = i;
  c[p[0]] = 0;
  int classes = 1;
  for (int i = 1; i < n; i++) {
    if (s[p[i]] != s[p[i-1]])
      classes++;
    c[p[i]] = classes - 1;
  }
  vector<int> pn(n), cn(n);
  for (int h = 0; (1 << h) < n; ++h) {
    for (int i = 0; i < n; i++) {
      pn[i] = p[i] - (1 << h);
      if (pn[i] < 0)
        pn[i] += n;
    }
    fill(cnt.begin(), cnt.begin() + classes, 0);
    for (int i = 0; i < n; i++)
      cnt[c[pn[i]]]++;
    for (int i = 1; i < classes; i++)
      cnt[i] += cnt[i-1];
    for (int i = n-1; i >= 0; i--)
      p[--cnt[c[pn[i]]]] = pn[i];
    cn[p[0]] = 0;
    classes = 1;
    for (int i = 1; i < n; i++) {
      pair<int, int> cur = {c[p[i]], c[(p[i] + (1 << h)) % n]};
      pair<int, int> prev = {c[p[i-1]], c[(p[i-1] + (1 << h)) % n]};
      if (cur != prev)
        ++classes;
      cn[p[i]] = classes - 1;
    }
    c.swap(cn);
    if(classes==n)break;//jodi logn iteration er aghei sort hoye jay
  }
  return p;
}
vector<int> suffix_array_construction(string s) {
  s += "$";
  vector<int> sorted_shifts = sort_cyclic_shifts(s);
  sorted_shifts.erase(sorted_shifts.begin());
  return sorted_shifts;
}
//longest common prefix of string suffix
vector<int> lcp_construction(string const& s, vector<int> const& p) {
  int n = s.size();
  vector<int> rank(n, 0);
  for (int i = 0; i < n; i++)
    rank[p[i]] = i;

  int k = 0;
  vector<int> lcp(n-1, 0);
  for (int i = 0; i < n; i++) {
    if (rank[i] == n - 1) {
      k = 0;
      continue;
    }
    int j = p[rank[i] + 1];
    while (i + k < n && j + k < n && s[i+k] == s[j+k])
      k++;
    lcp[rank[i]] = k;
    if (k)
      k--;
  }
  return lcp;
}
int main()
{
    string s;
    cin>>s;
    vector<int>v=suffix_array_construction(s);
    for(int i=0;i<v.size();i++)cout<<v[i]<<" ";
    cout<<endl;
    vector<int>lcp=lcp_construction(s,v);
    for(int i=0;i<lcp.size();i++)cout<<lcp[i]<<" ";
}
```

5.1 Point

```
const double inf = 1e100;
const double eps = 1e-9;
const double PI = acos((double)-1.0);
int sign(double x) { return (x > eps) - (x < -eps); }
struct PT {
  double x, y;
  PT() { x = 0, y = 0; }
  PT(double x, double y) : x(x), y(y) {}
  PT(const PT &p) : x(p.x), y(p.y)    {}
  PT operator + (const PT &a) const { return PT(x + a.x, y + a.y); }
  PT operator - (const PT &a) const { return PT(x - a.x, y - a.y); }
  PT operator * (const double a) const { return PT(x * a, y * a); }
  friend PT operator * (const double &a, const PT &b) { return PT(a * b.x, a *
b.y); }
  PT operator / (const double a) const { return PT(x / a, y / a); }
  bool operator == (PT a) const { return sign(a.x - x) == 0 && sign(a.y - y) == 0; }
  bool operator != (PT a) const { return !(*this == a); }
  bool operator < (PT a) const { return sign(a.x - x) == 0 ? y < a.y : x < a.x; }
  bool operator > (PT a) const { return sign(a.x - x) == 0 ? y > a.y : x > a.x; }
  double norm() { return sqrt(x * x + y * y); }
  double norm2() { return x * x + y * y; }
  PT perp() { return PT(-y, x); }
  double arg() { return atan2(y, x); }
  PT truncate(double r) { // returns a vector with norm r and having same
direction
    double k = norm();
    if (!sign(k)) return *this;
    r /= k;
    return PT(x * r, y * r);
  }
};
inline double dot(PT a, PT b) { return a.x * b.x + a.y * b.y; }
inline double dist2(PT a, PT b) { return dot(a - b, a - b); }
inline double dist(PT a, PT b) { return sqrt(dot(a - b, a - b)); }
inline double cross(PT a, PT b) { return a.x * b.y - a.y * b.x; }
inline double cross2(PT a, PT b, PT c) { return cross(b - a, c - a); }
inline int orientation(PT a, PT b, PT c) { return sign(cross(b - a, c - a)); }
PT perp(PT a) { return PT(-a.y, a.x); }
PT rotateccw90(PT a) { return PT(-a.y, a.x); }
PT rotatecw90(PT a) { return PT(a.y, -a.x); }
PT rotateccw(PT a, double t) { return PT(a.x * cos(t) - a.y * sin(t), a.x * sin(t) +
a.y * cos(t)); }
PT rotatecw(PT a, double t) { return PT(a.x * cos(t) + a.y * sin(t), -a.x * sin(t) +
a.y * cos(t)); }
double SQ(double x) { return x * x; }
double rad_to_deg(double r) { return (r * 180.0 / PI); }
double deg_to_rad(double d) { return (d * PI / 180.0); }
double get_angle(PT a, PT b) {
  double costheta = dot(a, b) / a.norm() / b.norm();
  return acos(max((double)-1.0, min((double)1.0, costheta)));
}
bool is_point_in_angle(PT b, PT a, PT c, PT p) { // does point p lie in angle <bac
  assert(orientation(a, b, c) != 0);
  if (orientation(a, c, b) < 0) swap(b, c);
  return orientation(a, c, p) >= 0 && orientation(a, b, p) <= 0;
}
bool half(PT p) {
  return p.y > 0.0 || (p.y == 0.0 && p.x < 0.0);
}
```

```
void polar_sort(vector<PT> &v) { // sort points in counterclockwise
  sort(v.begin(), v.end(), [](PT a,PT b) {
    return make_tuple(half(a), 0.0, a.norm2()) < make_tuple(half(b), cross(a,
b), b.norm2());
  });
}
```

5.2 Line

```
struct line {
  PT a, b; // goes through points a and b
  PT v; double c; //line form: direction vec [cross] (x, y) = c
  line() {}
  //direction vector v and offset c
 line(PT v, double c) : v(v), c(c) {
    auto p = get_points();
    a = p.first; b = p.second;
 }
 // equation ax + by + c = 0
 line(double _a, double _b, double _c) : v({-_b, _a}), c(-_c) {
  auto p = get_points();
    a = p.first; b = p.second;
 }
 // goes through points p and q
 line(PT p, PT q) : v(q - p), c(cross(v, p)), a(p), b(q) {}
 pair<PT, PT> get_points() { //extract any two points from this line
  PT p, q; double a = -v.y, b = v.x; // ax + by = -c
  if (sign(a) == 0) {
      p = PT(0, -c / b);
      q = PT(1, -c / b);
    }
    else if (sign(b) == 0) {
      p = PT(-c / a, 0);
      q = PT(-c / a, 1);
    }
    else {
      p = PT(0, -c / b);
      q = PT(1, (-c - a) / b);
    }
    return {p, q};
 }
  //ax + by + c = 0
  array<double, 3> get_abc() {
    double a = -v.y, b = v.x;
    return {a, b, c};
  }
  // 1 if on the left, -1 if on the right, 0 if on the line
  int side(PT p) { return sign(cross(v, p) - c); }
  // line that is perpendicular to this and goes through point p
  line perpendicular_through(PT p) { return {p, p + perp(v)}; }
  // translate the line by vector t i.e. shifting it by vector t
  line translate(PT t) { return {v, c + cross(v, t)}; }
  // compare two points by their orthogonal projection on this line
  // a projection point comes before another if it comes first according to
vector v
  bool cmp_by_projection(PT p, PT q) { return dot(v, p) < dot(v, q); }
 line shift_left(double d) {
  PT z = v.perp().truncate(d);
  return line(a + z, b + z);
 }
};
```

```
// find a point from a through b with distance d
PT point_along_line(PT a, PT b, double d) {
  return a + (((b - a) / (b - a).norm()) * d);
}
// projection point c onto line through a and b assuming a != b
PT project_from_point_to_line(PT a, PT b, PT c) {
  return a + (b - a) * dot(c - a, b - a) / (b - a).norm2();
}
// reflection point c onto line through a and b assuming a != b
PT reflection_from_point_to_line(PT a, PT b, PT c) {
  PT p = project_from_point_to_line(a,b,c);
  return point_along_line(c, p, 2.0 * dist(c, p));
}
// minimum distance from point c to line through a and b
double dist_from_point_to_line(PT a, PT b, PT c) {
  return fabs(cross(b - a, c - a) / (b - a).norm());
}
// returns true if point p is on line segment ab
bool is_point_on_seg(PT a, PT b, PT p) {
  if (fabs(cross(p - b, a - b)) < eps) {
    if (p.x < min(a.x, b.x) || p.x > max(a.x, b.x)) return false;
    if (p.y < min(a.y, b.y) || p.y > max(a.y, b.y)) return false;
    return true;
  }
  return false;
}
// minimum distance point from point c to segment ab that lies on segment ab
PT project_from_point_to_seg(PT a, PT b, PT c) {
  double r = dist2(a, b);
  if (fabs(r) < eps) return a;
  r = dot(c - a, b - a) / r;
  if (r < 0) return a;
  if (r > 1) return b;
  return a + (b - a) * r;
}
// minimum distance from point c to segment ab
double dist_from_point_to_seg(PT a, PT b, PT c) {
  return dist(c, project_from_point_to_seg(a, b, c));
}
// 0 if not parallel, 1 if parallel, 2 if collinear
bool is_parallel(PT a, PT b, PT c, PT d) {
  double k = fabs(cross(b - a, d - c));
  if (k < eps){
    if (fabs(cross(a - b, a - c)) < eps && fabs(cross(c - d, c - a)) < eps) return 2;
    else return 1;
  }
  else return 0;
}
// check if two lines are same
bool are_lines_same(PT a, PT b, PT c, PT d) {
  if (fabs(cross(a - c, c - d)) < eps && fabs(cross(b - c, c - d)) < eps) return true;
  return false;
}
// bisector vector of <abc
PT angle_bisector(PT &a, PT &b, PT &c){
  PT p = a - b, q = c - b;
  return p + q * sqrt(dot(p, p) / dot(q, q));
}
// 1 if point is ccw to the line, 2 if point is cw to the line, 3 if point is on the line
int point_line_relation(PT a, PT b, PT p) {
  int c = sign(cross(p - a, b - a));
  if (c < 0) return 1;
  if (c > 0) return 2;
  return 3;
}
// intersection point between ab and cd assuming unique intersection exists
bool line_line_intersection(PT a, PT b, PT c, PT d, PT &ans) {
  double a1 = a.y - b.y, b1 = b.x - a.x, c1 = cross(a, b);
  double a2 = c.y - d.y, b2 = d.x - c.x, c2 = cross(c, d);
  double det = a1 * b2 - a2 * b1;
  if (det == 0) return 0;
  ans = PT((b1 * c2 - b2 * c1) / det, (c1 * a2 - a1 * c2) / det);
  return 1;
}
// intersection point between segment ab and segment cd assuming unique
intersection exists
bool seg_seg_intersection(PT a, PT b, PT c, PT d, PT &ans) {
  double oa = cross2(c, d, a), ob = cross2(c, d, b);
  double oc = cross2(a, b, c), od = cross2(a, b, d);
  if (oa * ob < 0 && oc * od < 0){
    ans = (a * ob - b * oa) / (ob - oa);
    return 1;
  }
  else return 0;
}
// intersection point between segment ab and segment cd assuming unique
intersection may not exists
// se.size()==0 means no intersection
// se.size()==1 means one intersection
// se.size()==2 means range intersection
set<PT> seg_seg_intersection_inside(PT a, PT b, PT c, PT d) {
  PT ans;
  if (seg_seg_intersection(a, b, c, d, ans)) return {ans};
  set<PT> se;
  if (is_point_on_seg(c, d, a)) se.insert(a);
  if (is_point_on_seg(c, d, b)) se.insert(b);
  if (is_point_on_seg(a, b, c)) se.insert(c);
  if (is_point_on_seg(a, b, d)) se.insert(d);
  return se;
}
// intersection between segment ab and line cd
// 0 if do not intersect, 1 if proper intersect, 2 if segment intersect
int seg_line_relation(PT a, PT b, PT c, PT d) {
  double p = cross2(c, d, a);
  double q = cross2(c, d, b);
  if (sign(p) == 0 && sign(q) == 0) return 2;
  else if (p * q < 0) return 1;
  else return 0;
}
// intersection between segament ab and line cd assuming unique intersection
exists
bool seg_line_intersection(PT a, PT b, PT c, PT d, PT &ans) {
  bool k = seg_line_relation(a, b, c, d);
  assert(k != 2);
  if (k) line_line_intersection(a, b, c, d, ans);
  return k;
}
// minimum distance from segment ab to segment cd
double dist_from_seg_to_seg(PT a, PT b, PT c, PT d) {
  PT dummy;
  if (seg_seg_intersection(a, b, c, d, dummy)) return 0.0;
```

```
    else return min({dist_from_point_to_seg(a, b, c), dist_from_point_to_seg(a, b,
d),
        dist_from_point_to_seg(c, d, a), dist_from_point_to_seg(c, d, b)});
}
// minimum distance from point c to ray (starting point a and direction vector
b)
double dist_from_point_to_ray(PT a, PT b, PT c) {
    b = a + b;
    double r = dot(c - a, b - a);
    if (r < 0.0) return dist(c, a);
    return dist_from_point_to_line(a, b, c);
}
// starting point as and direction vector ad
bool ray_ray_intersection(PT as, PT ad, PT bs, PT bd) {
    double dx = bs.x - as.x, dy = bs.y - as.y;
    double det = bd.x * ad.y - bd.y * ad.x;
    if (fabs(det) < eps) return 0;
    double u = (dy * bd.x - dx * bd.y) / det;
    double v = (dy * ad.x - dx * ad.y) / det;
    if (sign(u) >= 0 && sign(v) >= 0) return 1;
    else return 0;
}
double ray_ray_distance(PT as, PT ad, PT bs, PT bd) {
    if (ray_ray_intersection(as, ad, bs, bd)) return 0.0;
    double ans = dist_from_point_to_ray(as, ad, bs);
    ans = min(ans, dist_from_point_to_ray(bs, bd, as));
    return ans;
}
```

5.3 Convex Hull

```
struct pt {
    double x, y;
};

int orientation(pt a, pt b, pt c) {
    double v = a.x*(b.y-c.y)+b.x*(c.y-a.y)+c.x*(a.y-b.y);
    if (v < 0) return -1; // clockwise
    if (v > 0) return +1; // counter-clockwise
    return 0;
}

bool cw(pt a, pt b, pt c, bool include_collinear) {
    int o = orientation(a, b, c);
    return o < 0 || (include_collinear && o == 0);
}
bool ccw(pt a, pt b, pt c, bool include_collinear) {
    int o = orientation(a, b, c);
    return o > 0 || (include_collinear && o == 0);
}

void convex_hull(vector<pt>& a, bool include_collinear = false) {
    if (a.size() == 1)
        return;

    sort(a.begin(), a.end(), [](pt a, pt b) {
        return make_pair(a.x, a.y) < make_pair(b.x, b.y);
    });
    pt p1 = a[0], p2 = a.back();
    vector<pt> up, down;
    up.push_back(p1);
    down.push_back(p1);
    for (int i = 1; i < (int)a.size(); i++) {
        if (i == a.size() - 1 || cw(p1, a[i], p2, include_collinear)) {
            while (up.size() >= 2 && !cw(up[up.size()-2], up[up.size()-1], a[i],
include_collinear))
                up.pop_back();
            up.push_back(a[i]);
        }
        if (i == a.size() - 1 || ccw(p1, a[i], p2, include_collinear)) {
            while (down.size() >= 2 && !ccw(down[down.size()-2], down[down.size()-
1], a[i], include_collinear))
                down.pop_back();
            down.push_back(a[i]);
        }
    }

    if (include_collinear && up.size() == a.size()) {
        reverse(a.begin(), a.end());
        return;
    }
    a.clear();
    for (int i = 0; i < (int)up.size(); i++)
        a.push_back(up[i]);
    for (int i = down.size() - 2; i > 0; i--)
        a.push_back(down[i]);
}
```

5.4 area of polygon

```
double area(const vector<point>& fig) {
double res = 0;
for (unsigned i = 0; i < fig.size(); i++) {
point p = i ? fig[i - 1] : fig.back();
point q = fig[i];
res += (p.x - q.x) * (p.y + q.y);
}
return fabs(res) / 2;
}
```

5.5 Triangles and Circles

```
double perimeterTriangle(double a, double b, double c) { return
a + b + c; }
double areaTriangle(double a, double b, double c) { return
sqrt(s * (s - a) * (s - b) * (s - c)); }
double rInCircle(double ab, double bc, double ca) {
        // radius of inscribed circle in a triangle
        return areaTriangle(ab, bc, ca) / (0.5 *
perimeterTriangle(ab, bc, ca));
}
double rCircumCircle(double ab, double bc, double ca) { return
ab * bc * ca / (4.0 * areaTriangle(ab, bc, ca)); }
double rCircumCircle(point a, point b, point c) { return
rCircumCircle(dist(a, b), dist(b, c), dist(c, a)); }
point cCircumCircle(point a, point b, point c) {
        b.x -= a.x;
        b.y -= a.y;
        c.x -= a.x;
        c.y -= a.y;
        double d = 2.0 * (b.x * c.y - b.y * c.x);
        double p = (c.y * (b.x * b.x + b.y * b.y) - b.y * (c.x
* c.x + c.y * c.y)) / d;
        double q = (b.x * (c.x * c.x + c.y * c.y) - c.x * (b.x
* b.x + b.y * b.y)) / d;
        return point(p + a.x, q + a.y);
}
```

5.6 Point Inside Convex Polygon O(nlogn)

```
struct pt {
        long long x, y;
        pt() {}
        pt(long long _x, long long _y) : x(_x), y(_y) {}
        pt operator+(const pt &p) const { return pt(x +
p.x, y + p.y); }
        pt operator-(const pt &p) const { return pt(x -
p.x, y - p.y); }
        long long cross(const pt &p) const { return x * p.y
- y * p.x; }
        long long dot(const pt &p) const { return x * p.x +
y * p.y; }
        long long cross(const pt &a, const pt &b) const {
return (a - *this).cross(b - *this); }
        long long dot(const pt &a, const pt &b) const {
return (a - *this).dot(b - *this); }
        long long sqrLen() const { return this->dot(*this);
}
};
bool lexComp(const pt &l, const pt &r) { return l.x < r.x ||
(l.x == r.x && l.y < r.y); }
int sgn(long long val) { return val > 0 ? 1 : (val == 0 ? 0 :
-1); }
vector<pt> seq;
pt translation;
int n;
bool pointInTriangle(pt a, pt b, pt c, pt point) {
        long long s1 = abs(a.cross(b, c));
        long long s2 = abs(point.cross(a, b)) +
abs(point.cross(b, c)) + abs(point.cross(c, a));
        return s1 == s2;
}
void prepare(vector<pt> &points) {
        n = points.size();
        int pos = 0;
        for (int i = 1; i < n; i++) {
                if (lexComp(points[i], points[pos])) pos =
i;
        }
        rotate(points.begin(), points.begin() + pos,
points.end());
        n--;
```

```
        seq.resize(n);
        for (int i = 0; i < n; i++) seq[i] = points[i + 1]
- points[0];
        translation = points[0];
}
bool pointInConvexPolygon(pt point) {
        point = point - translation;
        if (seq[0].cross(point) != 1 &&
sgn(seq[0].cross(point)) != sgn(seq[0].cross(seq[n - 1])))
return false;
        if (seq[n - 1].cross(point) != 0 && sgn(seq[n -
1].cross(point)) != sgn(seq[n - 1].cross(seq[0]))) return
false;
        if (seq[0].cross(point) == 0) return
seq[0].sqrLen() >= point.sqrLen();
        int l = 0, r = n - 1;
        while (r - l > 1) {
        int mid = (l + r) / 2;
        int pos = mid;
        if (seq[pos].cross(point) >= 0)
                l = mid;
        else
                r = mid;
        }
        int pos = l;
        return pointInTriangle(seq[pos], seq[pos + 1],
pt(0, 0), point);
}
```

6.1 KMP

```
#define MAX_N 100010
char T[MAX_N], P[MAX_N];  // T = text, P = pattern
int b[MAX_N], n, m;         // b = back table, n = length of
T, m = length of P
void kmpPreprocess() {      // call this before calling
kmpSearch()
        int i = 0, j = -1;
        b[0] = -1;                                      //
starting values
        while (i < m) {                                 //
pre-process the pattern string P
            while (j >= 0 && P[i] != P[j]) j = b[j];  //
different, reset j using b
            i++;
            j++;           // if same, advance both pointers
            b[i] = j;  // observe i = 8, 9, 10, 11, 12, 13
with j = 0, 1, 2, 3, 4, 5
        }
}  // in the example of P = "SEVENTY SEVEN" above
void kmpSearch() {                          // this is
similar as kmpPreprocess(), but on string T
        int i = 0, j = 0;                               //
starting values
        while (i < n) {                                 //
search through string T
            while (j >= 0 && T[i] != P[j]) j = b[j];  //
different, reset j using b
            i++;
            j++;           // if same, advance both pointers
            if (j == m) {  // a match found when j == m
                printf("P is found at index %d in T\n", i
- j);
                j = b[j];  // prepare j for the next
possible match
            }
        }
}
```

6.2 Hashing

```
// Computing hashing of a string . . .
long long polynomial_rolling_hash(string const& s) {
        const int p = 31;
        const int m = 1e9 + 9;

        long long hash_value = 0;
        long long p_pow = 1;

        for (char c : s) {
            hash_value = (hash_value + (c - 'a' + 1) *
p_pow) % m;
            p_pow = (p_pow * p) % m;
        }

        return hash_value;
}

// Search for duplicate strings in an array of strings . . .
// Puts the same strings in a group . . .
/// Example . .
/// 5
/// asaf
/// asad
/// asae
/// asaf
/// asad
/// 1 4
/// 2
/// 0 3
vector<vector<int> > group_identical_strings(vector<string>
const& s) {
        int n = s.size();
        vector<pair<long long, int>> hashes(n);
        for (int i = 0; i < n; i++)
          hashes[i] = {polynomial_rooling_hash(s[i]), i};

        sort(hashes.begin(), hashes.end());

        vector<vector<int>> groups;
        for (int i = 0; i < n; i++) {
            if (i == 0 || hashes[i].first != hashes[i-
1].first)
                groups.emplace_back();
            groups.back().push_back(hashes[i].second);
        }
        return groups;
}

// String Matching . . .
// Robin-Karp Algorithm . . .
vector<int> rabin_karp(string const& s, string const& t) {
        const int p = 31;
        const int m = 1e9 + 9;

        int S = s.size(), T = t.size();

        vector<long long> p_pow(max(S, T));
        p_pow[0] = 1;

        for (int i = 1; i < (int)p_pow.size(); i++)
            p_pow[i] = (p_pow[i-1] * p) % m;

        vector<long long> h(T + 1, 0);
        for (int i = 0; i < T; i++)
            h[i+1] = (h[i] + (t[i] - 'a' + 1) * p_pow[i]) %
m;

        long long h_s = 0;
        for (int i = 0; i < S; i++)
            h_s = (h_s + (s[i] - 'a' + 1) * p_pow[i]) % m;

        vector<int> occurences;
        for (int i = 0; i + S - 1 < T; i++) {
            long long cur_h = (h[i+S] + m - h[i]) % m;
            if (cur_h == h_s * p_pow[i] % m)
                occurences.push_back(i);
        }

        return occurences;
}
// Main function . . .
int main()
{
    vector<string> v;
    string s;
    int n;
```

```
    cin >> n;
    for(int K=0; K<n; K++){
          cin >> s;
          v.push_back(s);
    }

    vector<vector<int> > ara = group_identical_strings(v);

    for(int K=0; K<(int)ara.size(); K++){
          for(int L=0; L<(int)ara[K].size(); L++) cout <<
ara[K][L] << ' ';
          cout << "\n";
    }

    return 0;
}
```

6.4 z-algo

```
void z_algo(string st)
{
          int n=st.size();
          vector<int> z(n,0);
          int l=0,r=0;
          for(int i=1;i<n;i++) {
               if(i<=r) z[i]=min(r-i+1,z[i-l]);
               while(i+z[i]<n&&st[z[i]]==st[i+z[i]]) z[i]++;
               if(i+z[i]-1>r) l=i,r=i+z[i]-1;
          }
          for(auto i:z) cout<<i<< " ";
          /*
          ei function er kaj holo jodi ekta string abcdabc
hoy tahole proti positon
          theke koyta prefix oi position theke suffix
          er soman hoy ta oi position a dibe
          like
          abcdabc
pos=0123456
          0000300 er mane 4th positon theke 3 ta mane abc
prefix and abc suffix
          aaaaa
          04321
          aabaabcaabaabaab
          0103100610610310
          */
}
```

7.1 PBDS

```
#include <bits/stdc++.h>
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace std;
using namespace __gnu_pbds;
typedef long long ll;

typedef tree<int, null_type, less<int>, rb_tree_tag,
tree_order_statistics_node_update> indexed_set;
struct ordered_multiset {  // multiset supporting duplicating
values in set
          ll len = 0;
          const ll ADD = 1000010;
          const ll MAXVAL = 1000000010;
          unordered_map<ll, ll> mp;  // hash = 96814
          tree<ll, null_type, less<ll>, rb_tree_tag,
tree_order_statistics_node_update> T;

          ordered_multiset() {
              len = 0;
              T.clear(), mp.clear();
          }

          inline void insert(ll x) {
              len++, x += MAXVAL;
              ll c = mp[x]++;
              T.insert((x * ADD) + c);
          }

          inline void erase(ll x) {
              x += MAXVAL;
             ll c = mp[x];
             if (c) {
          c--, mp[x]--, len--;
          T.erase((x * ADD) + c);
              }
          }

          inline ll kth(ll k) {                  // 1-based
index,  returns the
              if (k < 1 || k > len) return -1;  // K'th
element in the treap,
              auto it = T.find_by_order(--k);   // -1 if none
exists
              return ((*it) / ADD) - MAXVAL;
          }

          inline ll lower_bound(ll x) {  // Count of value <x
in treap
              x += MAXVAL;
              ll c = 0;
              return (T.order_of_key((x * ADD) + c));
          }

          inline ll upper_bound(ll x) {  // Count of value
<=x in treap
              x += MAXVAL;
              ll c = mp[x];
              return (T.order_of_key((x * ADD) + c));
          }

          inline ll size() { return len; }  // Number of
elements in treap
};

int main() {
          indexed_set s1;
          ordered_multiset s2;

          s2.insert(2);
          s2.insert(2);
          s2.insert(3);

          cout << s2.kth(3) << endl;
          cout << s2.lower_bound(2) << endl;
          cout << s2.upper_bound(2) << endl;
          cout << s2.size() << endl;
          //  .. operations ..
          //  same ones as set..
          ///  extra: 1. s.order_of_key(x) // returns order
of x;
          ///  extra: 2. s.find_by_order(K) // returns K-th
element;
          return 0;
}
template<class T> using
oset=tree<T,null_type,less<T>,rb_tree_tag,tree_order_statisti
cs_node_update>;
```

8.1 Matrix Exponentiation

```
struct mat {
          ll a[3][3];
          mat() { mem(a, 0); }
          mat operator*(const mat &b) const {
              mat ret;
              rep(i, 3) rep(j, 3) rep(k, 3) ret.a[i][j] =
add(ret.a[i][j], mult(a[i][k], b.a[k][j]));
              return ret;
          }
};
mat power(mat a, ll b) {
          mat ret;
          rep(i, 3) ret.a[i][i] = 1;
          while (b) {
              if (b & 1) ret = ret * a;
```

```
            b >>= 1;
            a = a * a;
        }
        return ret;
}

// from dr.swad
const int MOD = 998244353;
typedef vector<int> row;
typedef vector<row> matrix;
inline int add(const int &a, const int &b) {
  int c = a + b;
  if (c >= MOD) c -= MOD;
  return c;
}
inline int mult(const int &a, const int &b) {
  return (long long)a * b % MOD;
}
matrix operator+(const matrix &m1, const matrix &m2) {
  int r = m1.size();
  int c = m1.back().size();
  matrix ret(r, row(c));
  for (int i = 0; i < r; i++) {
        for (int j = 0; j < c; j++) {
        ret[i][j] = add(m1[i][j], m2[i][j]);
        }
  }
  return ret;
}
matrix operator*(const matrix &m1, const int m2) {
  int r = m1.size();
  int c = m1.back().size();
  matrix ret(r, row(c));
  for (int i = 0; i < r; i++) {
        for (int j = 0; j < c; j++) {
        ret[i][j] = mult(m1[i][j], m2);
        }
  }
  return ret;
}
matrix operator*(const matrix &m1, const matrix &m2) {
  int r = m1.size();
  int m = m1.back().size();
  int c = m2.back().size();
  matrix ret(r, row(c, 0));
  for (int i = 0; i < r; i++) {
        for (int k = 0; k < m; k++) {
        for (int j = 0; j < c; j++) {
        ret[i][j] = add(ret[i][j], mult(m1[i][k],
m2[k][j]));
        }
        }
  }
  return ret;
 }
 matrix one(int dim) {
  matrix ret(dim, row(dim, 0));
  for (int i = 0; i < dim; i++) {
        ret[i][i] = 1;
  }
  return ret;
 }
 matrix operator^(const matrix &m, const int &e) {
  if (e == 0) return one(m.size());
  matrix sqrtm = m ^ (e / 2);
  matrix ret = sqrtm * sqrtm;
  if (e & 1) ret = ret * m;
  return ret;
 }
```

8.2 Submask

```
for(int submask = mask; ; submask =     (submask - 1) &
mask) {
          // do something
        if(submask == 0) break;
}
```

8.3 Divide and conquer optimization

```
int m, n;
vector<long long> dp_before(n), dp_cur(n);
long long C(int i, int j)
{
  return 1;//this function will be according to problem

}
// compute dp_cur[l], ... dp_cur[r] (inclusive)
void compute(int l, int r, int optl, int optr) {
  if (l > r)
     return;

  int mid = (l + r) >> 1;
  pair<long long, int> best = {LLONG_MAX, -1};

  for (int k = optl; k <= min(mid, optr); k++) {
     best = min(best, {(k ? dp_before[k - 1] : 0) + C(k, mid), k});
  }

  dp_cur[mid] = best.first;
  int opt = best.second;

  compute(l, mid - 1, optl, opt);
  compute(mid + 1, r, opt, optr);
}

int solve() {
  for (int i = 0; i < n; i++)
     dp_before[i] = C(0, i);

  for (int i = 1; i < m; i++) {
     compute(0, n - 1, 0, n - 1);
     dp_before = dp_cur;
  }

  return dp_before[n - 1];
}
int main()
{
   ios_base::sync_with_stdio(false);
   cin.tie(NULL);
}
```

Debugging

Input generator

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

int mrand(int a, int b)
{
    return a+rand()%(b-a+1);
}
//mt19937
rng(chrono::steady_clock::now().time_since_epoch().count());
//ll  mrand(ll l, ll r) {
         //return uniform_int_distribution<ll>(l,r) (rng);
//}
int main(int argc, char* argv[])
{
    srand(atoi(argv[1]));
    cout << 10000<< "\n";
    for(int K = 1; K <= 10000; K++) cout << mrand(10000000,
1000000000) << ' ' << 10 << "\n";
}
```

Sagor:

```
#include<bits/stdc++.h>
#define ll long long
#define yes cout << "YES" << endl
#define no cout << "NO" << endl
#define testing cout << "testing ";
#define mod 1000000007
#define optimize() ios_base::sync_with_stdio(0);cin.tie(0);cout.tie(0);
using namespace std;


void do_the_honour(){



}

int main(){
    optimize();
    int t=1;
    cin>>t;
    for(int z=1;z<=t;z++){


    do_the_honour();


}
    return 0;
}
```

NAHIAN

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

#define print(arr, n) for(int i = 0;i < n; i++) cout << arr[i] << ' ';
#define fastio() ios_base::sync_with_stdio(false);cin.tie(NULL);cout.tie(NULL)
#define dbg(x) cout << #x << " = " << x << "\n"

//typedef long long ll;
#define int long long

void gameover(int tt) {

}

signed main(void) {
    // #ifndef ONLINE_JUDGE
    // freopen("Error.txt", "w", stderr);
    // #endif
    fastio();
    int tt;
    cin >> tt;
    for(int i = 1;i <= tt; i++) {
        gameover(i);
    }
    return 0;
}
```

Floyd Warshall:

```
void FloydWarshall() {
    for(int k = 1;k <= n; k++) {
        for(int i = 1;i <= n; i++) {
            for(int j = 1;j <= n; j++) {
                dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);
            }
        }
    }
}
```

DSU:

```
const int N=2e3+10;
int parent[N];
int Size[N];

void make(int v) {parent[v]=v;Size[v]=1;}

int Find(int v){
    if(v==parent[v]) return v;
    return (parent[v]=Find(parent[v]));
}

void Union(int a,int b){
    a=Find(a);
    b=Find(b);
    if(Size[a]<Size[b]) swap(a,b);
    if(a!=b) parent[b]=a;
    Size[a]+=Size[b];
}
```

convex hull

```
struct Point {
  int x, y;
  bool operator<(Point P) const {
    if (x == P.x) return y < P.y;
    else return x < P.x;
  }
  bool operator==(Point p) const {
    return x == p.x && y == p.y;
  }
};

bool cw(Point a, Point b, Point c) {
  return a.x*(c.y-b.y)+b.x*(a.y-c.y)+c.x*(b.y-a.y)>0;
}
bool ccw(Point a, Point b, Point c) {
  return a.x*(c.y-b.y)+b.x*(a.y-c.y)+c.x*(b.y-a.y)<0;
}
bool cll(Point a, Point b, Point c) {
  return a.x*(c.y-b.y)+b.x*(a.y-c.y)+c.x*(b.y-a.y)==0;
}

vector<Point> get_convex_hull(vector<Point> points) {
  if (points.size() <= 2) return points;
  sort(points.begin(), points.end());

  Point first = points.front(), last = points.back();
  vector<Point> up{first, last}, down{first, last};

  for (int i=1; i < points.size(); i++) {
    if (i == points.size() - 1 || cw(first, points[i], last)) {
      // up set
      while (up.size() >= 2 && (!cw(up[up.size() - 2], up.back(), points[i]))) {
        up.pop_back();
      }
      up.push_back(points[i]);
    }
    if (i == points.size() - 1 || ccw(first, points[i], last)) {
      // down set
      while (down.size() >= 2 && (!ccw(down[down.size() - 2], down.back(),
points[i]))) {
        down.pop_back();
      }
      down.push_back(points[i]);
    }
  }

  points.clear();
  points.insert(points.end(), up.begin(), up.end());
  points.insert(points.end(), down.begin(), down.end());
  sort(points.begin(), points.end());
  points.resize(unique(points.begin(), points.end()) - points.begin());
  return points;
}
```