**MaTb3aa Reference**

**Math**

**Area of Triangle**

double AreaOfTri(ll x1, ll y1, ll x2, ll y2, ll x3, ll y3) {

double area = ((x1 \* (y2 - y3) + x2 \* (y3 - y1) + x3 \* (y1 - y2))) / 2.0;

return (area > 0) ? area : -area;

}

float findArea(float a, float b, float c)

{

// Length of sides must be positive

// and sum of any two sides

// must be smaller than third side.

if (a < 0 || b < 0 || c < 0 || (a + b <= c) || a + c <= b || b + c <= a)

{

cout << "Not a valid trianglen";

exit(0);

}

float s = (a + b + c) / 2;

return sqrt(s \* (s - a) \* (s - b) \* (s - c));

}

**Big Mod**

ll solve(ll b, ll p, ll mod) {

if (p == 0)return 1;

else if (p % 2 == 0) {

ll ans = solve(b, p / 2, mod);

return (ans % mod \* ans % mod) % mod;

}

else return (b % mod \* solve(b, p - 1, mod) % mod);

}

**find-divisors-from-prime-factors**

const int N = 10000000 + 5, mod = 1e9 + 7;

int spf[N];

void sieve() {

spf[1] = 1;

for (int i = 2; i < N; i++)spf[i] = i;

for (int i = 4; i < N; i += 2)spf[i] = 2;

for (int i = 3; i \* i < N; i++) {

if (spf[i] == i) {

for (int j = i \* i; j < N; j += i)

if (spf[j] == j)spf[j] = i;

}

}

}

int findDiv(int x) { // order prime fact

map<int, int>mp;

while (x != 1) {

mp[spf[x]]++;

x = x / spf[x];

}

int cnt = 1;

for (auto t : mp) {

cnt \*= (t.second + 1);

}

return cnt;

}

int main() {

sieve();

int x; cin >> x;

cout << findDiv(x) << endl;

}

**Inclusion-Exclusion Principle**

Most of counting involves duplicate counting

issue[count item more than once].

◼ IE principle is a generic sum rule to solve that

◼

◼ 2 ^ 3 - 1 = 7 subsets(exponential)

◼ General Computations

◼ Enumerate all subsets

◼ Compute each one intersection

◼ If odd subset add(include) it

◼ If even subset subtract(exclude) it

How many integers in {

1, 2 ..., 100

} are divisible by 2, 3, 5 or 7 ?

◼ How many divisible by 2 ? 100 / 2 = 50

◼ How many divisible by 3 ? 100 / 3 = 33

◼ How many divisible by 2, 3 ? 100 / (2 \* 3) = 16

◼ How many divisible by 2, 3, 7 ? 100 / 42 = 2 = > {42, 84}

◼ Answer : compute 2 ^ 4 - 1 terms = 15 terms

◼ F(2) + F(3) + F(5) + F(7)

◼ - F(2, 3) - F(2, 5) - F(2, 7) - F(3, 5) - F(3, 7) - F(5, 7)

◼ + F(2, 3, 5) + F(2, 3, 7) + F(2, 5, 7) + F(3, 5, 7)

◼ - F(2, 3, 5, 7)

int a[] = { 2,3,8,10 };

int solve(int idx, int d, int sign) {

if (idx == 4) {

if (d == 1) {

return 0;

}

return 100 \* sign / d;

}

return solve(idx + 1, d, sign) + solve(idx + 1, LCM(d, a[idx]), sign \* -1);

}

**intersection between two circle**

long double solve(double x0, double y0, double r0, double x1, double y1, double r1)

{

long double rr0 = r0 \* r0;

long double rr1 = r1 \* r1;

long double d = sqrt((x1 - x0) \* (x1 - x0) + (y1 - y0) \* (y1 - y0));

if (d > r1 + r0)

{

return 0;

}

else if (d <= abs(r0 - r1) && r0 >= r1)

{

return PI \* rr1;

}

else if (d <= abs(r0 - r1) && r0 < r1)

{

return PI \* rr0;

}

else

{

long double phi = (acos((rr0 + (d \* d) - rr1) / (2 \* r0 \* d))) \* 2;

long double theta = (acos((rr1 + (d \* d) - rr0) / (2 \* r1 \* d))) \* 2;

long double area1 = 0.5 \* theta \* rr1 - 0.5 \* rr1 \* sin(theta);

long double area2 = 0.5 \* phi \* rr0 - 0.5 \* rr0 \* sin(phi);

return area1 + area2;

}

}

**nCr**

double Rec(double n, double r) {

if (r == 0)

return 1;

return (n / r) \* Rec(n - 1, r - 1);

}

int main() {

double n, r;

double res;

cin >> n >> r;

res = round(Rec(n, r));

cout << long(res) << endl;

}

------------------------------------------ -

long long arr[100][100] = { 0 };

for (int i = 0; i <= 70; i++) { arr[i][i] = arr[i][0] = 1; }

for (int i = 1; i <= 70; i++) {

for (int j = 1; j < i; j++) {

arr[i][j] = arr[i - 1][j - 1] + arr[i - 1][j];

}

}

int x, y;

cin >> x >> y;

cout << arr[x][y];

**sieve**

void generate() {

memset(sieve, 1, sizeof sieve);

sieve[0] = sieve[1] = 0;

for (int i = 4; i < N; i += 2)

sieve[i] = 0;

for (int i = 3; i \* i < N; i += 2) {

if (!sieve[i])

continue;

for (int j = i \* i; j < N; j += i + i)

sieve[j] = 0;

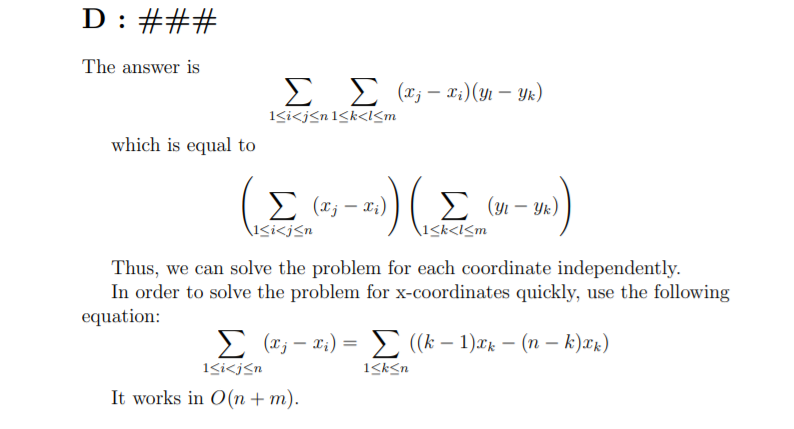
}

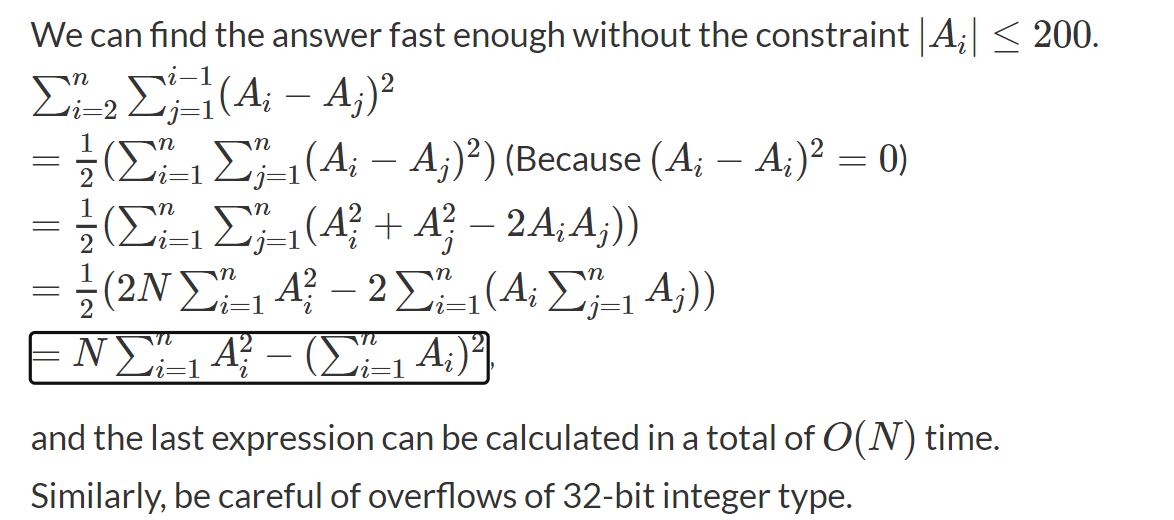
primes.push\_back(2);

for (int i = 3; i < N; i += 2)

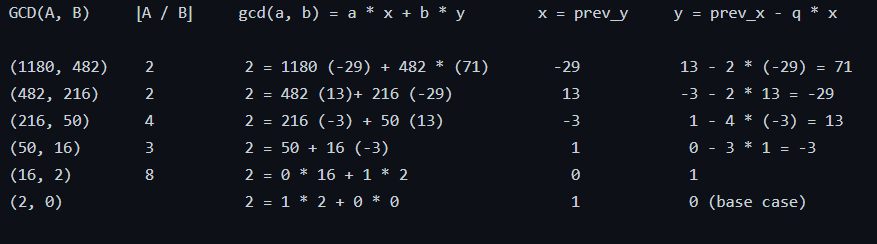
if (sieve[i]) primes.push\_back(i);

}





**Extended Euclidean algorithm**



For each input line the output line should consist of three integers X, Yand D, separated with space.

If there are several such Xand Y, you should output that pair for which | X | +| Y | is the minimal.If

there are several X and Y satisfying the minimal criteria, output the pair for which X ≤ Y .

// ax+by = g = gcd(a,b)

ll extended\_eculid(ll a, ll b, ll & x, ll & y) {

if (b == 0) {

x = 1; y = 0;

return a;

}

ll g = extended\_eculid(b, a % b, y, x);

// now our x = previous y

y -= (a / b) \* x;

return g;

}

int main() {

run();

ll x = 0, y = 0;

extended\_eculid(1180, 482, x, y);

cout << x << " " << y << endl;

}

Bézout's identity

Assume a > 0 and b > 0

◼ ax + by = g = gcd(a, b) = > we know that

◼ Can we generate further solutions ?

◼ Is following valid :

◼ a(x + b) + b(y - a) = g

◼ Yes, we added ab - ab, so same equation

◼ a(x + b / g) + b(y - a / g) = g

◼ a(x + kb / g) + b(y - ka / g) = g

◼ With easy math, we can generate!

**Factorial**

n! % x where x <= n -> 0

Wilson's theorem: (p-1)!%p = p-1 IFF p is prime

2! is the only prime factorial

Number of factorial digits?

how many digit for 1000? 1050? 9000 ? 9999? 4 digits. Use 1 + (int)log10(n)

pow(10, log10(X) ) = X

log(a\*b) = log(a)+log(b)

Then: double d = log(n!) = log(1) + log(2) .... log(n)

cout << floor(d) + 1

// What about number of bits for factorial? Think as above. Remember, 8 needs only 4 bits

//log2(120) = 6.906, rounded up become 7 (bits)

log2(2) + log2(3) + log2(4) + log2(5) = 6.906, which gives same result

// Given a prime p, and n!, what is max x such that n! divisble by p^x

n!% (p ^ x) == 0

Ex : n = 5 -- > 5!= 120 find max x such that n!% 2 ^ x == 0

1 2 3 4 5

2 ^ 0 2 ^ 1 3 ^ 1 2 ^ 2 5 ^ 1

// Very important function

int FactN\_PrimePower(int n, int p) { // O(log(n) base p)

int pow = 0;

for (int i = p; i <= n; i = i \* p)

pow += n / i;

return pow;

}

Factorial Factorization: n! = p1^a \* p2^b ...

What are possible primes in n!? Factorial is exponential, but finally it consist of [1-n]

So primes in range n

Seive on n, get primes in range n

For each prime number p

call FactN\_PrimePower(n, p)

Given m, what is max x such that m^x divides n!?

Again, think in prime representation.

let's simplify, uch that m^x divides g?

Let m = 2^3 \* 5^4

Let g = 2^10 \* 5^8 \* 11^3

Then 1st we can with 2 and 5 only.

10/3 = 3, then max for p1=2 is (2^3)^3

8/4 =2, so max is (5^4)^2

Then let x = min(3, 2) = 2...so m^2 divides g

what if m = 2^3 \* 5^4 \* 7^2 ? then it doesn't divide g, and x = 0

what about n!?

Then factroize m

for each p^x in m

check power in n!, and follow as above

What about calc gcd(!m, n)

gcd is greatest common divisor

Let m = 2^3 \* 5^7

Let n = 2^10 \* 5^3 \* 11^3

then gcd is min of each power, so that it divides both, and largest

gcd = 2 ^ min(3, 10) \* 5 ^ min(7, 3) \* 11 ^ min(0, 3)

Then factroize n

for each p^x in n

check power in m!, and use min of powers

How many trailing zeros in n!? E.g. 15! = 1307674368000 -> 3 zeros

How to calculate them? How zero come in base 10? 2\*5

Then, if w know that n! = 2^a \* 5^b \* reminder...then we have min(a, b) zeros.

More simpler, count of 2s > count of 5s in n!...then min(a, b) = b always

What about n! base X? how many zeros?

Again, Let X = 16 (hexidecimal), when zero appears?...notice (10 in base 16 equals 16 in base 10)

Again, every time n! has X, we have a nother Zero

So how many Xs there in n!?

Factorize X, check its primes power in n!

Get right most non zero digit of factorial N? So in 15! will be 8

Let's simplify. Last digit of X = X%10

Let X = 123000, then last non zero digit of X = 3

Let X = 123 \* 10^3, so 3 = 123%10

Note (a\*b\*c)%D = (a%D \* b%D \* c%D) % D

If in n! we could represent it as = 10^x \* reminder...then reminder%10 is the answer

again, Let n! = 2^a \* 5^b \* reminder1

Then n! = 10^b \* reminder2

So to calculate reminder2 we need to cacl n! such that we don't consider b 2's and b5's

In fact, all 5's wont't be used, and we will use overall a-b 2's

Let n = 15, n! = 1307674368000 = 2^11 \* 5^3 \* 5108103

So n! has 3 zeros

We want to calculate n! with only 11-3 2's

Let X = 2^8 \* 5108103 = 1307674368 so X%10 = 8

**Factorization**

//Counting the divisors

//2 ^ 4 has 5 divisors 2 ^ 0, 2 ^ 1...2 ^ 4

//p ^ n has n + 1 divisors for any pprime number

//what about p1^ a p1^ b(a + 1) (b + 1)

//E.g. 12 2 ^ 2 3 ^ 1 has 3 2 divisors.

//12 = 2 ^ 0 3 ^ 0

//12 = 2 ^ 0 3 ^ 1

//12 = 2 ^ 1 3 ^ 0

//12 = 2 ^ 1 3 ^ 1

//12 = 2 ^ 2 3 ^ 0

//12 = 2 ^ 2 3 ^ 1

//So if we modified factorization to return (p1 ^ a, p2 ^ b...)

//We could develop a simple RECURSIVE code to build the divisors.

//Simply pick a power from current prime, and move to next prime number

//Any iterative code is also possible, but a bit challenging

//what about Factorizing n^ power

//Simply if n = p1 ^ a p2 ^ b p3 ^ c

//Then n ^ z = p1 ^ az p2 ^ bz p3 ^ cz

//

//Divisors of n = (a + 1) (b + 1) (c + 1)

//Divisors of n ^ z = (az + 1) (bz + 1) (cz + 1)

//Let D(i) is number of divisors of i.Return sum D(i) in range n

int rangeFactorization1(int n) // Forward thinking

{

int s = 0;

for (int i = 1; i = n; i++)

s += generate\_divisors(i).size();

return s;

}

int rangeFactorization2(int n) // backward thinking

{

//suitable for range 210 ^ 6

vector <int> numFactors(n + 1);

for (int i = 1; i = n; i++) // For each divisor

for (int k = i; k = n; k += i) // For each divisble number

numFactors[k]++; // i divides k

int s = 0;

for (int i = 1; i = n; i++)

s += numFactors[i]; // sure you can do it without an array

return s;

}

**Modular multiplicative inverse**

ax ≡ 1 (% m)

◼ Which means ax % m = 1 % m

◼ m = 11, a = 8, x = 7 = > 8 \* 7 = 1 (mod 11)

◼ Then, a is multiplicative inverse of x for% m

◼ Also a = 1 / x(mod m)

◼ Exists IFF gcd(a, m) = 1

◼(119 / 7) % 11 = > 17 % 11 = > 6

◼ Recall 8 \* 7 = 1 (mod 11) … then 1 / 7 == 8 % 11

◼(119 \* 8) % 11 = (119 % 11 \* 8) % 11 = 6

ax ≡ 1 (% m)

◼ Then(ax - 1) % m = 0, then ax - 1 = qm

◼ m = 11, a = 8, x = 7 = > 8 \* 7 = 1 (mod 11)

◼ 56 - 1 = 5 \* 11

◼ Rearrange : ax + m(-q) = 1

◼ This is similar to ax + my = gcd(a, m) = 1

◼ That is, the solution to extended(a, m) giving

that gcd(a, m) = 1

◼ So just 1 call to extended, x is the answer

ll extended\_eculid(ll a, ll b, ll & x, ll & y) {

if (b == 0) {

x = 1; y = 0;

return a;

}

ll g = extended\_eculid(b, a % b, y, x);

// now our x = previous y

y -= (a / b) \* x;

return g;

}

ll modInvrse(ll a, ll m) {

ll x, y;

ll d = extended\_eculid(a, m, x, y);

if (d > 1)return -1;

return (x + m) % m;

}

Solution 2: Euler's theorem

if gcd(a, m) = 1 = > a ^ (φ(m))≡ 1 (% m)

◼ φ(m) is Euler's totient function

◼ As a result(divide both sides by a)

◼ a ^ (φ(m) - 1) ≡ a ^ (-1)

◼ if m is prime a ^ (-1) ≡ a ^ (m - 2)

◼ Computations amount in GCD vs Euler ?

◼ In addition, the theorem can be used to help

reducing large powers evaluations

int phi(int n) {

int cnt = 0;

for (int i = 1; i < n; ++i) {

cnt += \_\_gcd(i, n) == 1;

}

return cnt;

}

ll solve(ll b, ll p, ll mod) {

if (p == 0)return 1;

else if (p % 2 == 0) {

ll ans = solve(b, p / 2, mod);

return (ans % mod \* ans % mod) % mod;

}

else return (b % mod \* solve(b, p - 1, mod) % mod);

}

ll modInvrseP(ll a, ll p) {

return solve(a, p - 2, p);

}

ll modInvrse(ll a, ll m) {

return solve(a, phi(m) - 1, m);

}

Modinverse range for prime

Given P, compute all mod inv for range 1 - (p - 1)

◼ p % i = p - (p / i) \* i = > % equation

◼(p % i) % p = p % i

◼ p % p = 0

◼ p % i = -(p / i) \* i(mod p) = > % P

◼ Now, divide by i \* (p % i)

◼ 1 / i = -(p / i) \* 1 / (p % i) % p

◼ inv[i] = -(p / i) \* inv[p % i] % p

◼ Add + p to convert to + ve

◼ inv[i] = p - (p / i) \* inv[p % i] % p

vector<int>ModInvRange(int p) {

vector<int>inv(p - 1, 1);

for (int i = 2; i < p; ++i) {

inv[i] = (p - (p / i) \* inv[p % i] % p) % p;

}

return inv;

}

**Totient and Möbius Functions φ(n)**

φ(n), the Phi Function

◼ Count integers i < n such that gcd(i, n) = 1

◼ gcd(a, b) = 1 = > then coprimes : gcd(5, 7), gcd(4, 9)

◼ gcd(prime, i) = 1 for i < prime

◼ φ(10) = 4 = > 1, 3, 7, 9

◼ φ(5) = 4 = > 1, 2, 3, 4 ….φ(prime) = prime - 1

◼ If a, b, c are pairwise coprimes, then

◼ φ(a \* b \* c) = φ(a) \* φ(b) \* φ(c)

if k >= 1

φ(p ^ k) = p ^ k - p ^ (k - 1) = p ^ (k - 1) \* (p - 1) = p ^ k(1 - (1 / p)) such that p is prime

φ(n) = 1, 1, 2, 2, 4, 2, 6, 4, 6, 4, 10, 4, 12, 6, 8,8, 16, 6, 18, 8, 12, 10, 22, 8, 20, 12, 18, 12, 28,8, 30, 16, 20, 16, 24, 12, 36, 18, 24, 16, 40, 12

φ(1) = φ(2) = 1. φ(5) = 4

◼ φ(n) is even for n > 2

◼ sqrt(n) <= φ(n) <= n - sqrt(n) : Except 2, 6

◼ φ(nk) = nk - 1 \* φ(n)

◼ n = ∑iφ(di) where di are the divisors of n

//code

int phi(int n) {

int cnt = 0;

for (int i = 1; i < n; ++i) {

cnt += \_\_gcd(i, n) == 1;

}

return cnt;

}

// factorize and use fact p^(k-1) \* (p-1) ->p is prime

int phi(int n) {

int p\_to\_k = 0, ans = 1;

for (int i = 2; i \* i <= n; i++) {

if (n % i == 0) {

p\_to\_k = 1;//powers

while (n % i == 0) {

p\_to\_k \*= i;

n /= i;

}

ans \*= ((p\_to\_k / i) \* (i - 1));

}

}

if (n != 1)

ans \*= (n - 1);

return ans;

}

/// phi in range with sieve()

void phiRange(int n) {

bool prime[n];

int phi[n];

for (int i = 0; i <= n; ++i) {

phi[i] = prime[i] = 1;

}

for (int i = 2; i <= n; ++i) {

if (prime[i]) {

phi[i] = i - 1; //phi(prime) = p-1

for (int j = i + i; j <= n; j += i) {

prime[j] = 0;

int tmp = j, pow = 1;

while (tmp % i == 0) {

tmp /= i;

pow \*= i;

}

phi[j] \*= (pow / i) \* (i - 1);

}

}

}

}

// phi(N!) = (N is prime ? N-1 : N) \* phi(N-1)!

ll phi\_factn(int n) {

ll ret = 1;

for (int i = 2; i <= n; ++i) {

ret = ret \* (isprime(i) ? i - 1 : i);

}

return ret;

}

// phi for 1e9

v->vector of prime numbers

int phi(int n) {

if (n == 1)return 0;

int ans = n;

for (int i = 0; i < sz(v) && v[i] \* v[i] <= n; ++i) {

if (n % v[i] == 0) {

while (n % v[i] == 0)n /= v[i];

ans -= (ans / v[i]);

}

}

if (n != 1)

ans -= (ans / n);

return ans;

}

Square - free integer

Is not divisible by perfect square(except 1)

◼ perfect square : sqrt(n) = is integer.sqrt(16) = 4

◼ SQ : e.g. not divisible by 16 = 4x4... or 49 = 7x7...etc

In other words, no prime number occurs more

than once : e.g.n = 2 \* 5 \* 11 is square free, but n

= 2 \* 3 \* 3 \* 3 \* 7 is not (divisible by 9 = 3x3)

I - th square free : 1, 2, 3, 5, 6, 7, 10, 11, 13, 14,

15, 17, 19, 21, 22, 23, 26, 29, 30, 31, 33, 34

◼ F(13) = 19

Möbius function

μ(1) = 1

◼ μ(n) = 1 if n is a square - free positive integer

with an even number of prime factors.

◼ E.g.μ(2 \* 3 \* 5 \* 7) = 1

◼ μ(n) = −1 if n is a square - free positive integer

with an odd number of prime factors.

◼ E.g.μ(2 \* 3 \* 5) = -1

◼ μ(n) = 0 if n has a squared prime factor.

◼ E.g.μ(2 \* 3 \* 3 \* 7) = 0

int meobius(int n) {

int mebval = -1;

for (int i = 2; i \* i < n; ++i)

if (n % i == 0) {

if (n % (i \* i) == 0)return 0;

n /= i; mebval = -mebval;

}

if (n)

mebval = -mebval;

return mebval;

}

void moebius\_generator() {

bool prime[N];

int moebus[N];

for (int i = 2; i <= N; ++i) {

moebus[i] = -1, prime[i] = 1;

}

for (int i = 0; i < N; ++i) {

moebus[i] = 1;

for (int j = 0; j < N; ++j) {

prime[j] = 0, moebus[j] = j % (i \* i) == 0 ? 0 : -moebus[j];

}

}

}

Count the triples(a, b, c) such a, b, c <= n, and

gcd(a, b, c) = 1

◼ Reverse thinking, total - (# triples gcd > 1)

◼ How many triples with gcd multiple of 2: (n / 2)3

◼ How many triples with gcd multiple of 3 : (n / 3)3

◼ and 4 ? Ignore any numbers of internal duplicate primes

◼ and 6 ? already computed in 2, 3. Remove it : -(n / 6)3

◼ This is Inclusion Exclusion

int n = 4;

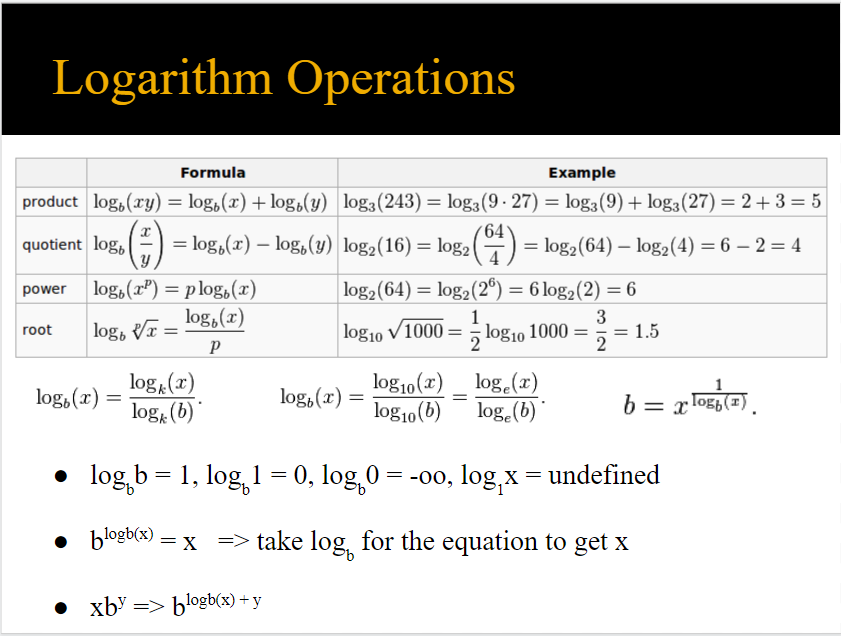
ll sum = n \* n \* n;

for (int i = 0; i < n; ++i) {

sum -= moebus[i] \* (n / i) \* (n / i) \* (n / i);

}

**LOG**

****

**Dynamic programming**

**dp Digit**

// Given two integers a and b. The task is to print

// sum of all the digits appearing in the

// integers between a and b

#include "bits/stdc++.h"

using namespace std;

// Memoization for the state results

long long dp[20][180][2];

// Stores the digits in x in a vector digit

long long getDigits(long long x, vector <int>& digit)

{

while (x)

{

digit.push\_back(x % 10);

x /= 10;

}

}

// Return sum of digits from 1 to integer in

// digit vector

long long digitSum(int idx, int sum, int tight,vector <int>& digit)

{

// base case

if (idx == -1)

return sum;

// checking if already calculated this state

if (dp[idx][sum][tight] != -1 and tight != 1)

return dp[idx][sum][tight];

long long ret = 0;

// calculating range value

int k = (tight) ? digit[idx] : 9;

for (int i = 0; i <= k; i++)

{

// caclulating newTight value for next state

int newTight = (digit[idx] == i) ? tight : 0;

// fetching answer from next state

ret += digitSum(idx - 1, sum + i, newTight, digit);

}

if (!tight)

dp[idx][sum][tight] = ret;

return ret;

}

// Returns sum of digits in numbers from a to b.

int rangeDigitSum(int a, int b)

{

// initializing dp with -1

memset(dp, -1, sizeof(dp));

// storing digits of a-1 in digit vector

vector<int> digitA;

getDigits(a - 1, digitA);

// Finding sum of digits from 1 to "a-1" which is passed

// as digitA.

long long ans1 = digitSum(digitA.size() - 1, 0, 1, digitA);

// Storing digits of b in digit vector

vector<int> digitB;

getDigits(b, digitB);

// Finding sum of digits from 1 to "b" which is passed

// as digitB.

long long ans2 = digitSum(digitB.size() - 1, 0, 1, digitB);

return (ans2 - ans1);

}

// driver function to call above function

int main()

{

long long a = 123, b = 1024;

cout << "digit sum for given range : "

<< rangeDigitSum(a, b) << endl;

return 0;

}

**game theory**

bool isWinning(int pos) {

if (pos == 0)

return false; // can't move = terminal position

int moves[3] = { 1, 3, 4 };

// play optimally: try all against his optimality

for (int i = 0; i < 3; ++i) {

if (pos >= moves[i] && !isWinning(pos - moves[i]))

// opponent will lose from this move

return true; // ANY lose = I win

}

return false; // ALL moves make opponent win

}

**Ternary Search**

/\*

\* How to think ternary? If it is ternary, say we search for min value, then you know we keep decreasing, then increasing starts.

\* Think in the order of x ~-> f(x) that you need to try, and have intuitive if the given function behaves as we need or not.

\*

\* An intuitive of x's typically will be sorted numbers. Say sorted numbers are 1 2 3 4 5 6 7

\*

\* Say we evaluated the value at first position, we expect sum to be so high -> 21

\* Now, think what happens when we move to right 1 step?

\*

\* We got -6 from the left 6 numbers and got +1 for right number -> 16

\*

\* So the more we move to left, we become nearer to left numbers, and further from right numbers

\*

\* F function gives: 21 16 13 12 13 16 21.

\*

\* So search seems ternable search, one more validation is required, is inc/dec strict? Homework

\*

\* Note, although 16 is duplicate, it is strict dec, then strict enc

\*

\* What if numbers are: 1 2 3 4 4 4 4 5 6 7 8

\*

\* F function is : 37 28 21 16 16 16 16 19 24 31 40

\*

\* Number 16 is repeated 4 times! So definitely this is not strict and ternary won't work!

\*

\* True, but algorithm will work if the repetition in the mode value only. Homework: Validate! Don't take every thing as fact :)

\*

\* Homework: why ternary needs f space to be strict?

\*

\* E.g. if we want to find minimum value in given list 5 1 5 5 5 5 5, answer may be 5 not 1

\*/

pair<double, double> ternaryReal() {

double left = v[0], right = v.back(); // set your range

while (right - left > EPS) { // stop when reaching almost right = left

double g = left + (right - left) / 3, h = left + 2 \* (right - left) / 3;

if (f(g) < f(h)) // use > if f increase then decrease

right = h;

else

left = g;

}

return make\_pair(left, f(left));

}

pair<int, int> ternaryDiscrete() {

int left = v[0], right = v.back(); // set your range

while (right - left > 3) { // We need 4 different positions

int g = left + (right - left) / 3, h = left + 2 \* (right - left) / 3;

if (f(g) < f(h)) // use > if f increase then decrease

right = h;

else

left = g;

}

int solIdx = left, answer = f(left++);

for (int i = left; i <= right; ++i) // iterate on the remaining

if (answer > f(i))

answer = f(i), solIdx = i;

return make\_pair(solIdx, answer);

}

/\*

\* What if we are given set of points (x, y) and need to find point p that has minimum Manhattan distance summation?

\* Now problem became 2D :(

\*

\* Remember the Independence feature

\* Minimize Manhattan Distance = Minimize on Xs + Minimize on Ys

\*

\* Hence once in 1D find best x, and once find best y

\*

\* What if function is square of the Euclidean distance?

\*

\* Similarly: Sum(Pi, p) = Sum (Pi.x-p.x)^2 + Sum (Pi.y-p.y)^2 for i [1-n], and p is target point

\*

\*

\* What if function is the Euclidean distance?!!!

\* Now it is really a 2D problem! You can't do a split!

\*

\* This where Nested Ternary appears. We have 2 F functions.

\* First ternary works on X in x ranges and call FX(x)

\*

\* FX is a nested ternary function that works on y, given x and evaluates FXY(x, y)

\*

\* WAIT! For both functions, you must guarantee that their spaces are strict dec/enc!

\*/

**Graph**

**Tarjan**

vector<vector<int>>g;

vector<int>idx, low, vis;

vector<pair<int, int>>bridges;

bool cut[1000];

int T, chiled;

void DFS(int u, int p) {

idx[u] = low[u] = ++T;

for (auto t : g[u]) {

if (!idx[t]) {

if (p == -1)chiled++;

DFS(t, u);

low[u] = min(low[u], low[t]);

if (low[t] > idx[u])bridges.push\_back({ t,u });

if (low[t] >= idx[u])cut[u] = 1;

}

else if (t != p) {

low[u] = min(low[u], idx[t]);

}

}

}

int main() {

file();

int n, m; cin >> n >> m;

g.resize(n); vis.resize(n);

idx.resize(n); low.resize(n);

for (int i = 0; i < m; i++) {

int u, v; cin >> u >> v; --u; --v;

g[u].push\_back(v);

g[v].push\_back(u);

}

DFS(0, -1);

for (auto t : bridges)cout << t.first << " " << t.second << endl;

cut[0] = (chiled > 1);

for (int i = 0; i < n; i++)if (cut[i])cout << i + 1 << endl;

}

**Detecte Cycle**

int start[1000], finish[10000];

vector<vector<int>>g;

int n, timer; bool cycle, forword;

// directed or undirected

void DFS(int u) {

start[u] = timer++;

for (auto t : g[u]) {

if (start[t] == -1)

DFS(t);

else {

if (finish[t] == -1) { cycle = 1; }

else if (start[u] < start[t]) { forword = 1; }

//else cross

}

}

finish[u] = timer++;}

**Print - SCC**

vector<vector<int>>g;

vector<int>S, vis, cur;

vector<vector<int>>comp;

int idx[N], low[N];

int T, ID;

void DFS(int u) {

idx[u] = low[u] = ++T;

vis[u] = 1;

S.push\_back(u);

for (auto t : g[u]) {

if (!idx[t])DFS(t);

if (vis[t])

low[u] = min(low[u], low[t]);

}

if (idx[u] == low[u]) {

while (true) {

int v = S.back();

cur.push\_back(v + 1);

vis[v] = 0; // why ??????

S.pop\_back();

if (v == u)break;

}

ID++;

comp.push\_back(cur); cur.clear();

}

}

int main() {

file();

int n, m;

cin >> n >> m;

g.resize(n);

vis.resize(n);

for (int i = 0; i < m; i++) {

int u, v; cin >> u >> v;

--u; --v;

g[u].push\_back(v);

}

for (int i = 0; i < n; i++)

if (!idx[i])DFS(i);

for (auto t : comp) {

for (auto tt : t)cout << tt << " ";

cout << endl;

}

}

**Bellmanford SASA**

struct edge {

int from, to, w;

edge(int from, int to, int w) : from(from), to(to), w(w) {}

bool operator < (const edge& e) const {

return w > e.w;

}

};

vi buildPath(vi prev, int src) {

vector<int> path; // make sure to test case self edge. E.g. 2 --> 2

for (int i = src; i > -1 && sz(path) <= sz(prev); i = prev[i])

path.push\_back(i);

reverse(all(path));

return path;

}

bool BellmanPrcoessing(vector<edge>& edgeList, int n, vi& dist, vi& prev, vi& pos) {

if (sz(edgeList) == 0) return false;

for (int it = 0, r = 0; it < n + 1; ++it, r = 0) {

for (int j = 0; j < sz(edgeList); ++j) {

edge ne = edgeList[j];

if (dist[ne.from] >= OO || ne.w >= OO) continue;

if (dist[ne.to] > dist[ne.from] + ne.w) {

dist[ne.to] = dist[ne.from] + ne.w;

prev[ne.to] = ne.from, pos[ne.to] = j, r++;

if (it == n) return true;

}

}

if (!r) break;

}

return false;

}

pair<int, bool> BellmanFord(vector<edge>& edgeList, int n, int src, int dest)// O(NE)

{

vector<int> dist(n, OO), prev(n, -1), reachCycle(n), path, pos(n);

dist[src] = 0;

bool cycle = BellmanPrcoessing(edgeList, n, dist, prev, pos);

if (cycle) {

vector<int> odist = dist;

BellmanPrcoessing(edgeList, n, dist, prev, pos);

for (int i = 0; i < n; ++i)

reachCycle[i] = (odist[i] != dist[i]);

}

else

path = buildPath(prev, dest);

return make\_pair(dist[dest], cycle);

}

**Dijkstra\_Algorithm**

struct edge {

int from, to, w;

edge(int from, int to, int w) : from(from), to(to), w(w) {}

bool operator < (const edge& e) const {

return w > e.w;

}

};

int Dijkstra2(vector< vector< edge > > adjList, int src, int dest = -1)// O(E logV)

{

int n = sz(adjList);

vi dist(n, OO), prev(n, -1);

dist[src] = 0;

priority\_queue<edge> q;

q.push(edge(-1, src, 0));

while (!q.empty()) {

edge e = q.top(); q.pop();

if (e.w > dist[e.to])continue; // some other state reached better

prev[e.to] = e.from;

for (int j = 0; j < adjList[e.to].size(); j++) {

edge ne = adjList[e.to][j];

if (dist[ne.to] > dist[ne.from] + ne.w) {

ne.w = dist[ne.to] = dist[ne.from] + ne.w;

q.push(ne);

}

}

}

return dest == -1 ? -1 : dist[dest];

}

**LCA**

int n, a, b, dp[100010][18], depth[100010];

vector<int> g[100100];

void dfs(int u, int parent) {

dp[u][0] = parent;

for (int i = 0; i < g[u].size(); ++i) {

int v = g[u][i];

if (v == parent)continue;

depth[v] = depth[u] + 1;

dfs(v, u);

}

}

int lca(int u, int v) {

if (depth[u] < depth[v])

swap(u, v);

for (int k = 17; k >= 0; --k) {

if (depth[u] - (1 << k) >= depth[v]) {

u = dp[u][k];

}

}

if (u == v)return u;

for (int k = 17; k >= 0; --k) {

if (dp[u][k] != dp[v][k]) {

u = dp[u][k];

v = dp[v][k];

}

}

return dp[u][0];

}

int shortestPath(int n, int p, int q) {

return depth[p] + depth[q] - 2 \* depth[lca(n, p, q)] + 1;

}

int main() {

file();

int n;

scanf("%d", &n);

for (int i = 0; i < n - 1; ++i) {

scanf("%d%d", &a, &b);

g[a].push\_back(b);

g[b].push\_back(a);

}

memset(dp, -1, sizeof dp);

dfs(1, -1);

for (int k = 1; k <= 17; ++k) {

for (int u = 1; u <= n; ++u) {

if (dp[u][k - 1] == -1)continue;

dp[u][k] = dp[dp[u][k - 1]][k - 1];

}

}

}

**MST**

struct unionFind {

vector < int > rankk, parent;

int forests;

unionFind(ll n) {

rankk = vector < int >(n), parent = vector < int >(n);

forests = n;

for (int i = 0; i < n; i++) {

parent[i] = i;

rankk[i] = 1;

}

}

int findSet(ll x) {

if (x == parent[x])return x;

return parent[x] = findSet(parent[x]);

}

void link(int x, int y) {

if (rankk[x] > rankk[y])swap(x, y);

parent[x] = y;

if (rankk[x] == rankk[y])rankk[y]++;

}

bool unionSets(int x, int y) {

x = findSet(x), y = findSet(y);

if (x != y) {

link(x, y);

forests--;

}

return x != y;

}

bool sameSet(int x, int y) {

return findSet(x) == findSet(y);

}

vector < vector < int > > connectedComponent() {

vector < vector < int > > comps(sz(parent));

for (int i = 0; i < sz(parent); i++) {

comps[findSet(i)].push\_back(i);

}

}

};

struct edge {

int from, to;

ll w;

edge(int from, int to, ll w) :from(from), to(to), w(w) {}

bool operator < (const edge& e)const {

return w > e.w;

}

};

vector < edge > MST\_kruskal(vector < edge > edgeList, int n) {

unionFind uf(n);

vector < edge > edges;

priority\_queue < edge > q;

for (auto p : edgeList)q.push(p);

while (!q.empty()) {

edge p = q.top(); q.pop();

if (uf.unionSets(p.from, p.to)) {

edges.push\_back(p);

}

}

return edges;

}

**Floyd**

int adj[MAX][MAX];

int path[MAX][MAX];

int n;

set<pair<int, int>>all;

void build\_path(int src, int dest)

{

if (path[src][dest] == -1) //So this is the last way

{

cout << src << " " << dest << endl;

return;

}

build\_path(src, path[src][dest]);

build\_path(path[src][dest], dest);

}

void floyd() {

for (int i = 1; i <= n; i++)adj[i][i] = 0;

for (int k = 1; k <= n; k++)

for (int i = 1; i <= n; i++)

for (int j = 1; j <= n; j++)

if (adj[i][j] > adj[i][k] + adj[k][j]) {

adj[i][j] = adj[i][k] + adj[k][j];

path[i][j] = k;

}

}

void TransitiveClosure()

{

// assume matrix is 0 for disconnect, 1 is connect

// we only care if a path exist or not, not a shortest path value

lp(k, n) lp(i, n) lp(j, n)

adj[i][j] |= (adj[i][k] & adj[k][j]);

}

void minimax()

{

// find path such that max value on road is minimum

lp(k, n) lp(i, n) lp(j, n)

adj[i][j] = min(adj[i][j], max(adj[i][k], adj[k][j]));

}

void maximin()

{

// find path such that min value on road is maximum

lp(k, n) lp(i, n) lp(j, n)

adj[i][j] = max(adj[i][j], min(adj[i][k], adj[k][j]));

}

void longestPathDAG()

{

lp(k, n) lp(i, n) lp(j, n)

adj[i][j] = max(adj[i][j], max(adj[i][k], adj[k][j]));

}

void countPaths()

{

lp(k, n) lp(i, n) lp(j, n) // Floyd warshal for counting #of paths

adj[i][j] += adj[i][k] \* adj[k][j];

// 1- assume graph is DAG

/\*

\* k = 5, i = 1, j = 2 we will use adj[ 1 ][ 5 ] with old value

\* k = 5, i = 1, j = 5 we will update adj[ 1 ] [ 5 ]

\* k = 5, i = 1, j = 7 we will use adj[ 1 ][ 5 ] with new value

\* Won't this give and incorrect result?

\*

\* NO. E.g. when k = 5, i = 1, j = 5:

\* adj[1][5] += adj[1][5] \* adj[5][5];

\* adj[5][5] = 0, so no update happens. In general, when k = i or j, no updates happens, so in-placement update is fine.

\*/

/\*

\* What if graph is not DAG.

\* if(adj[i][i] > 0) -> i has a cycle path on it

\*

\* Generally, for any node v that has a cycle,

\* if i reaches v, and v reaches j, then count of (i, j) is useless as count is OO. Remain adj[][] is valid

\*/

}

/\*

\* . If you had a cycle in a graph, the longest path wouldn't be

well defined, because you could keep going round and round the cycle

making the path longer.

\*/

bool isNegativeCycle() {

lp(i, n)

if (adj[i][i] < 0)

return true; // then node i got to i with overall cost < 0.

return false;

}

bool anyEffectiveCycle(int src, int dest)

{

lp(i, n)

if (adj[i][i] < 0 && adj[src][i] < OO && adj[i][dest] < OO)

return true;

return false; // there is a finite path although cycles if any

}

int graphDiameter()

{ // Longest path among all shortest ones

floyd2();

int mx = 0;

lp(i, n) lp(j, n) if (adj[i][j] < OO)

mx = max(mx, adj[i][j]);

return mx;

}

**Segment**

struct SegmentTree {

vector<ll> tree;vector<ll> lazy;

SegmentTree(int n) {

tree.resize(2 \* n);

lazy.resize(2 \* n);

for (int i = 0; i < sz(tree); i++)

tree[i] = 1e18, lazy[i] = 0;

}

void propagate(int cur, int l, int r, int x, int y) {

if (lazy[cur]) {

tree[cur] += lazy[cur];

if (l != r)

lazy[x] += lazy[cur], lazy[y] += lazy[cur];

lazy[cur] = 0;

}

}

void build(int cur, int l, int r, vector<ll>& arr) {

if (l == r) {

tree[cur] = arr[l];

return;

}

int left = 2 \* cur;

int right = 2 \* cur + 1;

int mid = (l + r) / 2;

build(left, l, mid, arr);

build(right, mid + 1, r, arr);

tree[cur] = min(tree[left], tree[right]);

}

void update(int cur, int l, int r, int x, int y, ll val) {

int left = 2 \* cur;

int right = 2 \* cur + 1;

int mid = (l + r) / 2;

propagate(cur, l, r, left, right);

if (l > y || r < x) return;

if (l >= x && r <= y) {

tree[cur] += val;

if (l != r)

lazy[left] += val, lazy[right] += val;

return;

}

update(left, l, mid, x, y, val);

update(right, mid + 1, r, x, y, val);

tree[cur] = min(tree[left], tree[right]);

}

ll solve(int cur, int l, int r, int x, int y) {

int left = 2 \* cur;

int right = 2 \* cur + 1;

int mid = (l + r) / 2;

propagate(cur, l, r, left, right);

if (x > r || y < l)

return 1e18;

if (l >= x && r <= y)

return tree[cur];

return min(solve(left, l, mid, x, y), solve(right, mid + 1, r, x, y));

}

};

**Lazy CP**

int t, n, m, lazy[4 \* N], l, r, val, f, sieve[N], tree[4 \* N], a[N];

void build(int n, int s, int e) {

if (s == e) {

if (sieve[a[s]])

tree[n] = 1;

else

tree[n] = 0;

return;

}

build(2 \* n, s, (s + e) / 2);

build(2 \* n + 1, (s + e) / 2 + 1, e);

tree[n] = (tree[2 \* n] + tree[2 \* n + 1]);

}

void push(int n, int s, int e) {

if (lazy[n]) {

if (sieve[lazy[n]])

tree[n] = (e - s + 1);

else

tree[n] = 0;

lazy[2 \* n] = lazy[n];

lazy[2 \* n + 1] = lazy[n];

}

lazy[n] = 0;

}

void update(int n, int s, int e) {

push(n, s, e);

if (s > r || e < l)return;

if (l <= s && e <= r) {

lazy[n] = val;

push(n, s, e);

return;

}

int md = s + e >> 1;

update(n \* 2, s, md);

update(2 \* n + 1, md + 1, e);

tree[n] = tree[2 \* n + 1] + tree[n \* 2];

}

ll get(int n, int s, int e) {

push(n, s, e);

if (s > r || e < l)return 0;

if (l <= s && e <= r)return tree[n];

int md = s + e >> 1;

return get(2 \* n, s, md) + get(2 \* n + 1, md + 1, e);

}

**Finding subsegments with the maximal sum**

struct data {

int sum, pref, suff, ans;

};

data combine(data l, data r) {

data res;

res.sum = l.sum + r.sum;

res.pref = max(l.pref, l.sum + r.pref);

res.suff = max(r.suff, r.sum + l.suff);

res.ans = max(max(l.ans, r.ans), l.suff + r.pref);

return res;

}

data make\_data(int val) {

data res;

res.sum = val;

res.pref = res.suff = res.ans = max(0, val);

return res;

}

void build(int a[], int v, int tl, int tr) {

if (tl == tr) {

t[v] = make\_data(a[tl]);

}

else {

int tm = (tl + tr) / 2;

build(a, v \* 2, tl, tm);

build(a, v \* 2 + 1, tm + 1, tr);

t[v] = combine(t[v \* 2], t[v \* 2 + 1]);

}

}

void update(int v, int tl, int tr, int pos, int new\_val) {

if (tl == tr) {

t[v] = make\_data(new\_val);

}

else {

int tm = (tl + tr) / 2;

if (pos <= tm)

update(v \* 2, tl, tm, pos, new\_val);

else

update(v \* 2 + 1, tm + 1, tr, pos, new\_val);

t[v] = combine(t[v \* 2], t[v \* 2 + 1]);

}

}

**Find the smallest number greater or equal to a specified number**

vector<int> t[4 \* MAXN];

void build(int a[], int v, int tl, int tr) {

if (tl == tr) {

t[v] = vector<int>(1, a[tl]);

}

else {

int tm = (tl + tr) / 2;

build(a, v \* 2, tl, tm);

build(a, v \* 2 + 1, tm + 1, tr);

merge(t[v \* 2].begin(), t[v \* 2].end(),

t[v \* 2 + 1].begin(), t[v \* 2 + 1].end(),back\_inserter(t[v]));

}

}

int query(int v, int tl, int tr, int l, int r, int x) {

if (l > r)

return INF;

if (l == tl && r == tr) {

vector<int>::iterator pos = lower\_bound(t[v].begin(), t[v].end(), x);

if (pos != t[v].end())

return \*pos;

return INF;

}

int tm = (tl + tr) / 2;

return min(query(v \* 2, tl, tm, l, min(r, tm), x),

query(v \* 2 + 1, tm + 1, tr, max(l, tm + 1), r, x));

}

void update(int v, int tl, int tr, int pos, int new\_val) {

t[v].erase(t[v].find(a[pos]));

t[v].insert(new\_val);

if (tl != tr) {

int tm = (tl + tr) / 2;

if (pos <= tm)

update(v \* 2, tl, tm, pos, new\_val);

else

update(v \* 2 + 1, tm + 1, tr, pos, new\_val);

}

else {

a[pos] = new\_val;

}

}