**SIEVE:**

1. #define M 1000000
2. **bool** marked[M];
3. vector < **int** > primes;
4. **void** sieve(**int** n) {
5. primes.push\_back(2);
6. **for** (**int** i = 3; i \* i <= n; i += 2) {
7. **if** (marked[i] == 0) {
8. primes.push\_back(i);
9. **for** (**int** j = i \* i; j <= n; j += i + i) {
10. marked[j] = 1;
11. }
12. }
13. }
14. }

**NUMBER OF DIVISOR:**

1. **int** NOD(**int** n) {
2. **int** sqrtn = sqrt(n);
3. **int** res = 1;
4. **for** (**int** i = 0; i < primes.size() && primes[i] <= sqrtn; i++) {
5. **if** (n % primes[i] == 0) {
6. **int** p = 0; /\*Counter for power of prime\*/
7. **while** (n % primes[i] == 0) {
8. n /= primes[i];
9. p++;
10. }
11. sqrtn = sqrt(n);
12. p++; /\*Increase it by one at end\*/
13. res \*= p; /\*Multiply with answer\*/
14. }
15. }
16. **if** (n != 1) {
17. res \*= 2; /\*Remaining prime has power p^1. So multiply with 2\*/
18. }
19. **return** res;
20. }

**ALL DIVISOR:**

1. vector < ll > mainfactor;
2. vector < **int** > countfactor;
3. vector < ll > allfactor;
4. **void** alldivisor(**int** idx, ll num) {
5. **if** (idx == mainfactor.size()) {
6. allfactor.push\_back(num);
7. **return**;
8. }
9. alldivisor(idx + 1, num);
10. //    alldivisor(idx+1,mainfactor[idx]);
11. **for** (**int** i = countfactor[idx]; i; i--) {
12. alldivisor(idx + 1, num \* mainfactor[idx]);
13. num \*= mainfactor[idx];
14. }
15. }

**BIG MOD: (Iterative)**

1. **int** bigmod(**int** b, **int** p, **int** m) {
2. **int** res = 1 % m, x = b % m;
3. **while** (p) {
4. **if** (p & 1) res = (res \* x) % m;
5. x = (x \* x) % m;
6. p >>= 1;
7. }
8. **return** res;
9. }

**BITWISE SIEVE:**

1. #define M 100000000
2. **int** marked[M / 64 + 2];
3. **void** sieve(**int** n) {
4. **for** (**int** i = 3; i \* i < n; i += 2) {
5. **if** (!bitCheck(i)) {
6. **for** (**int** j = i \* i; j <= n; j += i + i) {
7. bitOn(j);
8. }
9. }
10. }
11. }
12. **bool** isPrime(**int** num) {
13. **return** num > 1 && (num == 2 || ((num & 1) && !on(num)));
14. }

**MODULAR INVERSE + EXTENDED EUCLID:**

1. pii extendedEuclid(**int** a, **int** b) {
2. **if** (b == 0)
3. **return** pii(1, 0);
4. **else** {
5. pii d = extendedEuclid(b, a % b);
6. **return** pii(d.ss, d.ff - d.ss \* (a / b));
7. }
8. }
9. **int** modularInverse(**int** a, **int** n) { // returns a er modular Inverse ; n dara mod kore
10. pii ret = extendedEuclid(a, n);
11. **return** ((ret.ff % n) + n) % n;
12. }

**FACTORIAL:**

1. ///Digits of N! in Different Base
2. **int** factorialDigitExtended(**int** n, **int** base) {
3. **double** x = 0;
4. **for** (**int** i = 1; i <= n; i++) {
5. x += log10(i) / log10(base); ///Base Conversion
6. }
7. **int** res = ((**int**) x) + 1;
8. **return** res;
9. }
10. ///Prime Factorization of Factorial
11. **void** factFactorize(**int** n) {
12. **for** (**int** i = 0; i < primes.size() && primes[i] <= n; i++) {
13. **int** x = n;
14. **int** freq = 0;
15. **while** (x / primes[i]) {
16. freq += x / primes[i];
17. x = x / primes[i];
18. }
19. printf("%d^%d\n", primes[i], freq);
20. }
21. }
22. ///leading digits in factorial
23. ///Find the first K digits of N!
24. ///k=first k digits
25. **int** leadingDigitFact(**int** n, **int** k) {
26. **double** fact = 0;
27. ///Find log(N!)
28. **for** (**int** i = 1; i <= n; i++) {
29. fact += log10(i);
30. }
31. ///Find the value of q
32. **double** q = fact - floor(fact + EPS);
33. **double** B = pow(10, q);
34. ///Shift decimal point k-1 times
35. **for** (**int** i = 0; i < k - 1; i++) {
36. B \*= 10;
37. }
38. ///Don't forget to floor it
39. **return** floor(B + eps);
40. }
41. ///last digit of factorial
42. **int** last\_digit\_factorial(**int** N) {
43. **int** i, j, ans = 1, a2 = 0, a5 = 0, a;
45. **for** (i = 1; i <= N; i++) {
46. j = i;
47. //divide i by 2 and 5
48. **while** (j % 2 == 0) {
49. j /= 2;
50. a2++;
51. }
52. **while** (j % 5 == 0) {
53. j /= 5;
54. a5++;
55. }
56. ans = (ans \* (j % 10)) % 10;
57. }
58. a = a2 - a5;
59. **for** (i = 1; i <= a; i++)
60. ans = (ans \* 2) % 10;
62. **return** ans;
63. }

**LEADING DIGIT OF POWER: (N^M)**

1. ///leading digits of n^k
2. ll leadingdigit(ll n, ll k, ll dig) {
3. ///log10(x)=y
4. **double** y = (**double**) k \* log10(n);
5. ///if y=123.456 we are setting y=0.456
6. y = y - floor(y);
7. ///we are getting 10^y for reverse processing
8. y = pow(10, y);
9. ///now we are getting the digits by shifting the decimal to right
10. rep(i, dig - 1) y \*= 10;
11. **return** (ll) floor(y);
12. }

**LINEAR DIOPHANTINE EQN + EXTENDED EUCLID (ITERATIVE):**

1. **int** ext\_gcd(**int** A, **int** B, **int** \* X, **int** \* Y) {
2. **int** x2, y2, x1, y1, x, y, r2, r1, q, r;
3. x2 = 1;  y2 = 0;
4. x1 = 0;  y1 = 1;
5. **for** (r2 = A, r1 = B; r1 != 0; r2 = r1, r1 = r, x2 = x1, y2 = y1, x1 = x, y1 = y) {
6. q = r2 / r1;
7. r = r2 % r1;
8. x = x2 - (q \* x1);
9. y = y2 - (q \* y1);
10. } \* X = x2; \* Y = y2;
11. **return** r2;
12. }
13. **bool** linearDiophantine(**int** A, **int** B, **int** C, **int** \* x, **int** \* y) {
14. **int** g = gcd(A, B);
15. **if** (C % g != 0) **return** **false**; //No Solution
16. **int** a = A / g, b = B / g, c = C / g;
17. ext\_gcd(a, b, x, y); //Solve ax + by = 1
18. **if** (g < 0) { //Make Sure gcd(a,b) = 1
19. a \*= -1;
20. b \*= -1;
21. c \*= -1;
22. } \* x \*= c; \* y \*= c; //ax + by = c
23. **return** **true**; //Solution Exists
24. }
25. **int** main() {
26. **int** x, y, A = 2, B = 3, C = 5;
27. **bool** res = linearDiophantine(A, B, C, & x, & y);
28. **if** (res == **false**) printf("No Solution\n");
29. **else** {
30. printf("One Possible Solution (%d %d) \n", x, y);
31. **int** g = gcd(A, B);
32. **int** k = 1; //Use different value of k to get different solutions
33. printf("Another Possible Solution (%d %d)\n", x + k \* (B / g), y - k \* (A / g));
34. }
35. }

**Simple Hyperbolic Diophantine Equation:**

1. **bool** isValidSolution(**int** a, **int** b, **int** c, **int** p, **int** div) {
2. **if** (((div - c) % a) != 0) **return** **false**; //x = (div - c) / a
3. **if** (((p - b \* div) % (a \* div)) != 0) **return** **false**; // y = (p-b\*div) /(a\*div)
4. **return** **true**;
5. }
6. **int** hyperbolicDiophantine(**int** a, **int** b, **int** c, **int** d) {
7. **int** p = a \* d + b \* c;
8. **if** (p == 0) { //ad + bc = 0
9. **if** (-c % a == 0) **return** -1; //Infinite solutions (-c/a, k)
10. **else** **if** (-b % a == 0) **return** -1; //Infinite solutions (k, -b/a)
11. **else** **return** 0; //No solution
12. } **else** {
13. **int** res = 0;
14. //For each divisor of p
15. **int** sqrtn = sqrt(p), div;
16. **for** (**int** i = 1; i <= sqrtn; i++) {
17. **if** (p % i == 0) { //i is a divisor
19. //Check if divisors i,-i,p/i,-p/i produces valid solutions
20. **if** (isValidSolution(a, b, c, p, i)) res++;
21. **if** (isValidSolution(a, b, c, p, -i)) res++;
22. **if** (p / i != i) { //Check whether p/i is different divisor than i
23. **if** (isValidSolution(a, b, c, p, p / i)) res++;
24. **if** (isValidSolution(a, b, c, p, -p / i)) res++;
25. }
26. }
27. }
28. **return** res;
29. }
30. }

**EULER PHI/TOTIENT:**

1. **int** phi(**int** n) // Oyler er Tochient Function
2. {
3. **int** ret = n;
4. **for** (**int** i = 2; i \* i <= n; i++) {
5. **if** (n % i == 0) {
6. **while** (n % i == 0) {
7. n /= i;
8. }
9. ret -= ret / i;
10. }
11. }
12. **if** (n > 1) ret = ret - (ret / n);
13. **return** ret;
14. }
15. ///another if this method needs to be called several times
16. #define M 1000005
17. **int** phi[M];
18. **void** calculatePhi() {
19. **for** (**int** i = 1; i < M; i++) {
20. phi[i] = i;
21. }
22. **for** (**int** p = 2; p < M; p++) {
23. **if** (phi[p] == p) { // p is a prime
24. **for** (**int** k = p; k < M; k += p) {
25. phi[k] -= phi[k] / p;
26. }
27. }
28. }
29. }

**SEGMENTED SIEVE:**

1. #define SIZE 1000005
2. **int** arr[SIZE];
3. **int** segmentedSieve(**int** a, **int** b) {
4. **if** (a == 1) a++;
5. **int** sqrtn = sqrt(b);
6. CLR(arr);
7. **for** (**int** i = 0; i < primes.size() && primes[i] <= sqrtn; i++) {
8. **int** p = primes[i];
9. **int** j = p \* p;
10. ///If j is smaller than a, then shift it inside of segment [a,b]
11. **if** (j < a) j = ((a + p - 1) / p) \* p;
12. **for** (; j <= b; j += p) {
13. arr[j - a] = 1; ///mark them as not prime
14. }
15. }
16. **int** res = 0;
17. **for** (**int** i = a; i <= b; i++) {
18. ///If it is not marked, then it is a prime
19. **if** (arr[i - a] == 0) res++;
20. }
21. **return** res;
22. }

**Sum of Divisor (SOD):**

1. **int** SOD(**int** n) {
2. **int** res = 1;
3. **int** sqrtn = sqrt(n);
4. **for** (**int** i = 0; i < primes.size() && primes[i] <= sqrtn; i++) {
5. **if** (n % primes[i] == 0) {
6. **int** tempSum = 1; //Contains value of (p^0+p^1+...p^a)
7. **int** p = 1;
8. **while** (n % primes[i] == 0) {
9. n /= primes[i];
10. p \*= primes[i];
11. tempSum += p;
12. }
13. sqrtn = sqrt(n);
14. res \*= tempSum;
15. }
16. }
17. **if** (n != 1) {
18. res \*= (n + 1); //Need to multiply (p^0+p^1)
19. }
20. **return** res;
21. }
22. ///SOD(N)=(p01+p11+p21...pa11)×(p02+p12+p22...pa22)×...(p0k+p1k+p2k...pakk)

**SUM OF NUMBER OF DIVISOR (SNOD):**

1. **int** SNOD(**int** n) {
2. **int** res = 0;
3. **int** u = sqrt(n);
4. **for** (**int** i = 1; i <= u; i++) {
5. res += (n / i) - i; //Step 1
6. }
7. res \*= 2; //Step 2
8. res += u; //Step 3
9. **return** res;
10. }

**SUM OF SUM OF DIIVISOR (SSOD):**

1. ll ssod(ll n) {
2. ll ans = 0;
3. **for** (ll i = 2; i \* i <= n; i++) {
4. ll j = n / i;
5. ans += (i + j) \* (j - i + 1) / 2;
6. ans += i \* (j - i);
7. }
8. **return** ans;
9. }

**XOR 1 TO N:**

1. ll f(**long** **long** a) {
2. **long** **long** res[] = {a,1,a + 1,0};
3. **return** res[a % 4];
4. }
5. ll getXor(**long** **long** a, **long** **long** b) {
6. **return** f(b) ^ f(a - 1);
7. }

**LCM SUM:**

1. /\*Given n, calculate the sum LCM(1,n) + LCM(2,n) + .. + LCM(n,n)\*/
2. ll res[1000010];
3. ll phi[1000010];
4. **void** precal(**int** n) {
5. ///Calculate phi from 1 to n using sieve
6. FOR(i, 1, n) phi[i] = i;
7. FOR(i, 2, n) {
8. **if** (phi[i] == i) {
9. **for** (**int** j = i; j <= n; j += i) {
10. phi[j] /= i;
11. phi[j] \*= i - 1;
12. }
13. }
14. }
15. ///Calculate partial result using sieve
16. ///For each divisor d of n, add phi(d)\*d to result array
17. FOR(i, 1, n) {
18. **for** (**int** j = i; j <= n; j += i) {
19. res[j] += (i \* phi[i]);
20. }
21. }
22. }
23. **int** main() {
24. precal(1000000);
25. **int** kase;
26. scanf("%d", & kase);
27. **while** (kase--) {
28. ll n;
29. getL(n);
30. ///We already have partial result in res[n]
31. ll ans = res[n] + 1;
32. ans \*= n;
33. ans /= 2;
34. printf("%lld\n", ans);
35. }
36. }

**MATRIX EXPONENTIAL:**

1. **struct** matrix {
2. **int** v[5][5];
3. **int** row, col; // number of row and column
4. };
5. **int** mod = 10000;
6. // multiplies two matrices and returns the result
7. matrix multiply(matrix a, matrix b) {
8. assert(a.col == b.row);
9. matrix r;
10. r.row = a.row;
11. r.col = b.col;
12. **for** (**int** i = 0; i < r.row; i++) {
13. **for** (**int** j = 0; j < r.col; j++) {
14. **int** sum = 0;
15. **for** (**int** k = 0; k < a.col; k++) {
16. sum += a.v[i][k] \* b.v[k][j];
17. sum %= mod;
18. }
19. r.v[i][j] = sum;
20. }
21. }
22. **return** r;
23. }
24. // returns mat^p
25. matrix power(matrix mat, **int** p) {
26. assert(p >= 1);
27. **if** (p == 1) **return** mat;
28. **if** (p % 2 == 1)
29. **return** multiply(mat, power(mat, p - 1));
30. matrix ret = power(mat, p / 2);
31. ret = multiply(ret, ret);
32. **return** ret;
33. }
34. **int** main() {
35. **int** tcase;
36. **int** a, b, n, m;
37. cin >> tcase;
38. **while** (tcase--) {
39. // input routine
40. cin >> a >> b >> n >> m;
41. // preparing the matrix
42. matrix mat;
43. mat.row = mat.col = 2;
44. mat.v[0][0] = mat.v[0][1] = mat.v[1][0] = 1;
45. mat.v[1][1] = 0;
46. // preparing mod value
47. mod = 1;
48. **for** (**int** i = 0; i < m; i++) mod \*= 10;
49. a %= mod, b %= mod;
50. **if** (n < 3) {
51. **if** (n == 0) cout << a << endl;
52. **if** (n == 1) cout << b << endl;
53. **if** (n == 2) cout << (a + b) % mod << endl;
54. } **else** {
55. mat = power(mat, n - 1);
56. **int** ans = b \* mat.v[0][0] + a \* mat.v[0][1];
57. ans %= mod;
58. cout << ans << endl;
59. }
60. }
61. }

**NCR:**

1. ncr[0][0] = 1;
2. **int** limncr = 10;
3. FOR(i, 1, limncr)
4. FOR(j, 0, limncr) {
5. **if** (j > i) ncr[i][j] = 0;
6. **else** **if** (j == i || j == 0) ncr[i][j] = 1;
7. **else** ncr[i][j] = ncr[i - 1][j - 1] + ncr[i - 1][j];
8. }

**NEGATIVE BASE:**

1. string negaBase(**int** n, **int** b) {
2. **int** i, tmp;
3. string a;
4. **for** (i = 0; n; i++) {
5. tmp = n % b;
6. n = n / b;
7. **if** (tmp < 0) {
8. tmp += (-b), n++;
9. }
10. a += '0' + tmp;
11. }
12. **for** (n = 0; n < (i / 2); n++) swap(a[n], a[i - n - 1]);
13. **if** (i) **return** a;
14. **return** "0";
15. }

**CONSTRUCT N FROM SOD:**

1. // powi64(a, b) computes a^b, rememver that prime upto i-1 are used
2. i64 table[NN + 1][NN + 1]; // if there is an overflow, table[i][j] = inf;
3. **void** preprocessTable() {
4. **for** (**int** i = 0; i <= NN; i++) table[0][i] = 1;
5. **for** (**int** i = 1; i <= NN; i++) {
6. table[i][0] = 1;
7. **for** (**int** j = 1; j < NN; j++) table[i][j] = table[i][j - 1] + powi64(pr[i - 1], j);
8. }
9. }
10. vector < i64 > calculateXFromSumOfDivisors(**int** sum) {
11. vector < i64 > res;
12. i64 val = 1, prevD = 1;
13. **for** (**int** i = NN;; i--) {
14. **if** (sum == 1) {
15. res.push\_back(val); // Here the value is saved
16. sum \*= prevD, val = 1;
17. }
18. **if** (i <= 0 || sum == 1) **break**;
19. **for** (**int** j = NN - 1; j >= 0; j--) {
20. **if** (table[i][j] > 1 && (sum % table[i][j] == 0)) {
21. val \*= powi64(pr[i - 1], j);
22. sum /= table[i][j], prevD = table[i][j];
23. **break**;
24. }
25. }
26. }
27. **return** res;
28. }

**STRING MULTIPLY:**

1. string multiply(string a, **int** b) {
2. // a contains the biginteger in reversed form
3. **int** carry = 0;
4. **for** (**int** i = 0; i < a.size(); i++) {
5. carry += (a[i] - 48) \* b;
6. a[i] = (carry % 10 + 48);
7. carry /= 10;
8. }
9. **while** (carry) {
10. a += (carry % 10 + 48);
11. carry /= 10;
12. }
13. **return** a;
14. }

**WILSON THEOREM:**

Wilson's theorem states that a natural number n>1 is a prime number if and only if (n−1)!≡−1 (mod n).

This asserts that (n−1)! is exactly 1 less than a multiple of n when n is prime.

If N is a composite number (except for 1 and 4), then (N−1)! ≡0 (mod N)

**CATALAN NUMBER:**

# 

**DIRECTION ARRAY:**

1. // 4 direction
2. **int** dx[]={-1,1,0,0};
3. **int** dy[]={0,0,-1,1};
4. // 8 direction
5. **int** dx[]={-1,1,0,0,-1,-1,1,1};
6. **int** dy[]={0,0,-1,1,-1,1,1,-1};
7. // horse
8. **int** dx[] = {-2,-2,2,2,-1,-1,1,1};
9. **int** dy[] = {1,-1,-1,1,2,-2,-2,2};

**UNION FIND/DISJOINT SET:**

1. **class** UnionFind {
2. **private**:
3. vi p, rank;
4. **public**:
5. UnionFind(**int** n) {
6. rank.assign(n + 1, 0);
7. p.assign(n + 1, 0);
8. **for** (**int** i = 0; i <= n; i++)
9. p[i] = i;
10. }
11. **int** findSet(**int** i) {
12. **return** (p[i] == i) ? i : (p[i] = findSet(p[i]));
13. }
14. **bool** isSameSet(**int** i, **int** j) {
15. **return** findSet(i) == findSet(j);
16. }
17. **void** unionSet(**int** i, **int** j) {
18. **if** (!isSameSet(i, j)) {
19. **int** x = findSet(i);
20. **int** y = findSet(j);
21. **if** (rank[x] > rank[y]) p[y] = x;
22. **else** {
23. p[x] = y;
24. **if** (rank[x] == rank[y])
25. rank[y]++;
26. }
27. }
28. }
29. };

**UNION FIND (MAMUN4122) :**

1. **int** find\_representative(**int** r) {
2. **if** (par[r] == r) **return** r;
3. **else** {
4. **return** par[r] = find\_representative(par[r]);
5. }

**ARTICULATION BRIDGE:**

1. #define BRIDGENODE 10010
2. **class** BridgeFinding {
3. **int** disc[BRIDGENODE];
4. **int** low[BRIDGENODE];
5. **int** col[BRIDGENODE];
6. **int** cnt; ///Timer
7. **int** cc; ///Color
8. **void** tarjan(**int** s, **int** parentEdge) {
9. disc[s] = low[s] = cnt++;
10. col[s] = cc + 1;
11. **for** (**int** i = 0; i < adj[s].size(); ++i) {
12. **int** t = adj[s][i].ff;
13. **int** edgeNumber = adj[s][i].ss;
14. **if** (edgeNumber == parentEdge) **continue**;
15. **if** (col[t] <= cc) { ///New node. Discovery.
16. tarjan(t, edgeNumber);
17. low[s] = min(low[s], low[t]); ///Update back edge extension for S
18. **if** (low[t] > disc[s]) { ///Back edge of T did not go above S
19. ///This edge is Bridge
20. }
21. } **else** **if** (col[t] == cc + 1) { ///Back Edge
22. low[s] = min(low[s], disc[t]);
23. }
24. }
25. }
27. **public**:
28. vector < pair < **int**, **int** > > adj[BRIDGENODE]; ///Enter target and edge number as pair
29. **void** clear(**int** n) {
30. cc += 3; ///cc is now 0. cc+1 is 1
31. **for** (**int** i = 0; i <= n; i++) {
32. adj[i].clear();
33. }
34. }
35. **void** findBridge(**int** n, **int** start = 0) {
36. **for** (**int** i = start; i <= n; i++) {
37. **if** (col[i] <= cc) {
38. tarjan(i, -1);
39. }
40. }
41. }
42. }bridge;

**ARTICULATION POINT:**

1. #define ARTNODE 10010
2. **class** ArticulationPoint {
3. **int** disc[ARTNODE], low[ARTNODE], col[ARTNODE];
4. **int** cnt; ///Timer
5. **int** cc; ///Color
6. **int** root; ///Root of tree
7. **void** tarjan(**int** s, **int** p) {
8. disc[s] = low[s] = cnt++;
9. col[s] = cc + 1;
10. **int** child = 0; ///Needed for root only
11. **int** art = 0;
12. **for** (**int** i = 0; i < adj[s].size(); ++i) {
13. **int** t = adj[s][i];
14. **if** (t == p) **continue**; ///Don't go to parent
15. **if** (col[t] <= cc) { ///New node. Discovery.
16. child++;
17. tarjan(t, s);
18. low[s] = min(low[s], low[t]); ///Update back edge extension for S
19. **if** (low[t] >= disc[s]) { ///Back edge of T did not go above S
20. art++; ///S is articulation point for T
21. }
22. } **else** **if** (col[t] == cc + 1) { ///Back Edge
23. low[s] = min(low[s], disc[t]);
24. }
25. }
26. **if** ((s == root && child > 1) || (s != root && art)) {
27. ///Edit in this block
28. printf("This is a articulation point: %d\n", s);
29. }
30. }
31. **public**:
32. vector < **int** > adj[ARTNODE];
33. **void** clear(**int** n) {
34. cc += 3; ///cc is now 0. cc+1 is 1
35. **for** (**int** i = 0; i <= n; i++) {
36. adj[i].clear();
37. }
38. }
39. **void** findArt(**int** n, **int** start = 0) {
40. **for** (**int** i = start; i <= n; i++) {
41. **if** (col[i] <= cc) {
42. root = i;
43. tarjan(i, -1);
44. }
45. }
46. }
47. }
48. art;
49. ///remaining component after removing x
50. **int** compo = 0;
51. **if** ((s == root && child > 1) || (s != root && art)) {
52. **if** (s == root) compo = child;
53. **else** compo = art + 1;
54. } **else** { ///s is not articulation point
55. **if** (p != -1 || adj[s].size()) compo = 1; ///It is not singleton
56. }

**Strongly Connected Component (SCC) :**

1. //Cycle contains which scc node belongs too.
2. **struct** SCC {
3. **int** num[NODE], low[NODE], col[NODE], cycle[NODE], st[NODE];
4. **int** tail, cnt, cc;
5. vi adj[NODE];
6. SCC(): tail(0), cnt(0), cc(0) {}
7. **void** clear() {
8. cc += 3;
9. FOR(i, 0, NODE - 1) adj[i].clear();
10. tail = 0;
11. }
12. **void** tarjan(**int** s) {
13. num[s] = low[s] = cnt++;
14. col[s] = cc + 1;
15. st[tail++] = s;
16. FOR(i, 0, SZ(adj[s]) - 1) {
17. **int** t = adj[s][i];
18. **if** (col[t] <= cc) {
19. tarjan(t);
20. low[s] = MIN(low[s], low[t]);
21. }
22. /\*Back edge\*/
23. **else** **if** (col[t] == cc + 1)
24. low[s] = MIN(low[s], low[t]);
25. }
26. **if** (low[s] == num[s]) {
27. **while** (1) {
28. **int** temp = st[tail - 1];
29. tail--;
30. col[temp] = cc + 2;
31. cycle[temp] = s;
32. **if** (s == temp) **break**;
33. }
34. }
35. }
36. **void** shrink(**int** n) {
37. FOR(i, 0, n) {
38. FOR(j, 0, SZ(adj[i]) - 1) {
39. adj[i][j] = cycle[adj[i][j]]; ///Careful. This will create self-loop. Just ignore i->i edges when processing.
40. }
41. }
42. FOR(i, 0, n) {
43. **if** (cycle[i] == i) **continue**;
44. **int** u = cycle[i];
45. FOR(j, 0, SZ(adj[i]) - 1) {
46. **int** v = adj[i][j];
47. adj[u].pb(v);
48. }
49. adj[i].clear();
50. }
51. FOR(i, 0, n) { ///Not always necessary
52. sort(ALL(adj[i]));
53. UNIQUE(adj[i]);
54. }
55. }
56. **void** findSCC(**int** n) {
57. FOR(i, 0, n) {
58. **if** (col[i] <= cc) {
59. tarjan(i);
60. }
61. }
62. }
63. };

**SHRINK : (mamun4122)**

1. **bool** shrink(**int** node) {
2. repI(i, node) {
3. repI(j, adj[i].size()) {
4. **if** (cycle[i] != cycle[adj[i][j]]) {
5. scc[cycle[i]].push\_back(cycle[adj[i][j]]);
6. }
7. }
8. }
9. ///this inserts same edge multiple times
10. }

**Lowest Common Ancesstor (LCA):**

1. #define mx 100002
2. **int** depth[mx]; //লেভেল
3. **int** parent[mx][22]; //স্পার্স টেবিল
4. **int** T[mx]; //প্যারেন্ট
5. vector < **int** > g[mx];
6. **void** dfs(**int** from, **int** u, **int** dep) {
7. T[u] = from;
8. depth[u] = dep;
9. **for** (**int** i = 0; i < (**int**) g[u].size(); i++) {
10. **int** v = g[u][i];
11. **if** (v == from) **continue**;
12. dfs(u, v, dep + 1);
13. }
14. }
15. **int** lca\_query(**int** N,**int** p,**int** q)//N=নোড সংখ্যা
16. {
17. **int** tmp, log, i;
18. **if** (depth[p] < depth[q])
19. tmp = p, p = q, q = tmp;
20. log = 1;
21. **while** (1) {
22. **int** next = log + 1;
23. **if** ((1 << next) > depth[p]) **break**;
24. log++;
25. }
26. **for** (i = log; i >= 0; i--)
27. **if** (depth[p] - (1 << i) >= depth[q])
28. p = parent[p][i];
29. **if** (p == q)
30. **return** p;
31. **for** (i = log; i >= 0; i--)
32. **if** (parent[p][i] != -1 && parent[p][i] != parent[q][i])
33. p = parent[p][i], q = parent[q][i];
34. **return** T[p];
35. }
36. **void** lca\_init(**int** N) {
37. memset(parent, 1, **sizeof**(parent)); //শুরুতে সবগুলো ঘরে -১ থাকবে
38. **int** i, j;
39. **for** (i = 0; i < N; i++)
40. parent[i][0] = T[i];
41. //can be modified here by looping only to depth from dfs
42. **for** (j = 1; 1 << j < N; j++)
43. **for** (i = 0; i < N; i++)
44. **if** (parent[i][j - 1] != -1)
45. parent[i][j] = parent[parent[i][j - 1]][j - 1];
46. }
47. **int** main(**void**) {
48. g[0].pb(1);g[0].pb(2);g[2].pb(3);g[2].pb(4);
49. dfs(0, 0, 0);
50. lca\_init(5);
51. printf("%d\n", lca\_query(5, 3, 4));
52. **return** 0;
53. }

**LCA + MST :**

1. #define MAXN 200005
2. **struct** edge {
3. **int** u, v, pos;
4. ll w;
5. **bool** operator < (**const** edge & p) **const** {
6. **return** w < p.w;
7. }
8. };
9. **int** pr[MAXN];
10. vector < edge > e;
11. **int** find(**int** r) {
12. **if** (pr[r] == r) **return** r;
13. **return** pr[r] = find(pr[r]);
14. }
15. vector < pair < **int**, ll > > g[MAXN];
16. ll ans[MAXN];
17. ///LCA HERE
18. **int** depth[MAXN];
19. **int** parent[MAXN][30];
20. **int** T[MAXN];
21. ll dist[MAXN][30];
22. **void** dfs(**int** from, **int** u, **int** dep) {
23. T[u] = from;
24. depth[u] = dep;
25. **for** (**int** i = 0; i < (**int**) g[u].size(); i++) {
26. **int** v = g[u][i].ff;
27. **if** (v == from) **continue**;
28. dist[v][0] = g[u][i].ss;
29. dfs(u, v, dep + 1);
30. }
31. }
32. **void** lca\_init(**int** N) {
33. memset(parent, -1, **sizeof**(parent));
34. **int** i, j;
35. **for** (i = 1; i <= N; i++)
36. parent[i][0] = T[i];
37. //can be modified here by looping only to depth from dfs
38. **for** (j = 1; 1 << j < N; j++)
39. **for** (i = 1; i <= N; i++)
40. **if** (parent[i][j - 1] != -1) {
41. dist[i][j] = max(dist[i][j - 1], dist[parent[i][j - 1]][j - 1]);
42. parent[i][j] = parent[parent[i][j - 1]][j - 1];
43. }
44. }
45. ll lca\_query(**int** N, **int** p, **int** q) //N=???? ??????
46. {
47. **int** tmp, log, i;
48. **if** (depth[p] < depth[q]) swap(p, q);
49. ll tmpans = 0;
50. log = 1;
51. **while** (1) {
52. **int** next = log + 1;
53. **if** ((1 << next) > depth[p]) **break**;
54. log++;
55. }
56. **for** (i = log; i >= 0; i--)
57. **if** (depth[p] - (1 << i) >= depth[q]) {
58. tmpans = max(tmpans, dist[p][i]);
59. p = parent[p][i];
60. }
61. **if** (p == q)
62. **return** tmpans;
63. **for** (i = log; i >= 0; i--)
64. **if** (parent[p][i] != -1 && parent[p][i] != parent[q][i]) {
65. tmpans = max(tmpans, dist[p][i]);
66. tmpans = max(tmpans, dist[q][i]);
67. p = parent[p][i];
68. q = parent[q][i];
69. }
70. tmpans = max(tmpans, dist[p][0]);
71. tmpans = max(tmpans, dist[q][0]);
72. **return** tmpans;
73. }
74. ///LCA END
75. **int** main() {
76. **int** n, m;
77. getII(n, m);
78. edge get;
79. rep(i, m) {
80. getII(get.u, get.v);
81. getL(get.w);
82. get.pos = i;
83. e.push\_back(get);
84. }
85. CLR(ans);
86. ///mst here
87. ll mst = 0;
88. sort(e.begin(), e.end());
89. **for** (**int** i = 1; i <= n; i++) pr[i] = i;
90. **int** count = 0;
91. **for** (**int** i = 0; i < (**int**) e.size(); i++) {
92. **int** u = find(e[i].u);
93. **int** v = find(e[i].v);
94. **if** (u != v) {
95. pr[u] = v;
96. count++;
97. mst += e[i].w;
98. ans[e[i].pos] = 1;
99. g[e[i].u].push\_back(make\_pair(e[i].v, e[i].w));
100. g[e[i].v].push\_back(make\_pair(e[i].u, e[i].w));
101. **if** (count == n - 1) **break**;
102. }
103. }
104. dfs(-1, 1, 1);
105. lca\_init(n);
106. repI(i, m) {
107. **if** (ans[e[i].pos]) ans[e[i].pos] = mst;
108. **else** {
109. ans[e[i].pos] = mst - lca\_query(n, e[i].u, e[i].v);
110. ans[e[i].pos] += e[i].w;
111. }
112. }
113. rep(i, m) printf("%lld\n", ans[i]);
114. }

**STABLE MARRIAGE:**

1. /\* A person has an integer preference for each of the persons of the opposite
2. \* sex, produces a matching of each man to some woman. The matching will follow:
3. \*          - Each man is assigned to a different woman (n must be at least m)
4. \*          - No two couples M1W1 and M2W2 will be unstable.
5. \* Two couples are unstable if (M1 prefers W2 over W1 and W1 prefers M2 over M1)
6. \* INPUT: m – number of man, n – number of woman (must be at least as large as m)
7. \*          - L[i][]: the list of women in order of decreasing preference of man i
8. \*          - R[j][i]: the attractiveness of i to j.
9. \* OUTPUTS: - L2R[]: the mate of man i (always between 0 and n-1)
10. \*          - R2L[]: the mate of woman j (or -1 if single)      \*/
11. /\*While there is a free man m: let w be the most-preferred woman to whom he has not yet proposed, and propose m to w. If w is free, or is engaged to someone whom she prefers less than m, match m with w, else deny proposal.\*/
12. **int** m, n, L[MAXM][MAXW], R[MAXW][MAXM], L2R[MAXM], R2L[MAXW], p[MAXM];
13. **void** stableMarriage() {
14. memset(R2L, -1, **sizeof**(R2L));
15. memset(p, 0, **sizeof**(p));
16. **for** (**int** i = 0; i < m; i++) { // Each man proposes...
17. **int** man = i;
18. **while** (man >= 0) {
19. **int** wom;
20. **while** (1) {
21. wom = L[man][p[man]++];
22. **if** (R2L[wom] < 0 || R[wom][man] > R[wom][R2L[wom]]) **break**;
23. }
24. **int** hubby = R2L[wom];
25. R2L[L2R[man] = wom] = man;
26. man = hubby;
27. }
28. }
29. }

**MINIMUM VERTEX COVER:**

1. #define MAXN 100002
2. **int** dp[MAXN][5];
3. **int** par[MAXN];
4. vectoredges[MAXN];
5. **int** f(**int** u, **int** isGuard) {
6. **if** (edges[u].size() == 0)
7. **return** 0;
8. **if** (dp[u][isGuard] != -1)
9. **return** dp[u][isGuard];
10. **int** sum = 0;
11. **for** (**int** i = 0; i < (**int**) edges[u].size(); i++) {
12. **int** v = edges[u][i];
13. **if** (v != par[u]) {
14. par[v] = u;
15. **if** (isGuard == 0)
16. sum += f(v, 1);
17. **else**
18. sum += min(f(v, 1), f(v, 0));
19. }
20. }
21. **return** dp[u][isGuard] = sum + isGuard;
22. }
23. **int** main() {
24. memset(dp, -1, **sizeof**(dp));
25. **int** n;
26. scanf("%d", & n);
27. **for** (**int** i = 1; i < n; i++) {
28. **int** u, v;
29. scanf("%d%d", & u, & v);
30. edges[u].push\_back(v);
31. edges[v].push\_back(u);
32. }
33. **int** ans = 0;
34. ans = min(f(1, 1), f(1, 0));
35. printf("%d\n";, ans);
36. **return** 0;
37. }

**KTH BEST SHORTEST PATH:**

1. **int** m, n, deg[MM], source, sink, K, val[MM][12];
2. **struct** edge {
3. **int** v, w;
4. }
5. adj[MM][500];
6. **struct** info {
7. **int** v, w, k;
8. **bool** operator < (**const** info & b) **const** {
9. **return** w > b.w;
10. }
11. };
12. priority\_queue < info, vector < info > > Q;
13. **void** kthBestShortestPath() {
14. **int** i, j;
15. info u, v;
16. **for** (i = 0; i < n; i++)
17. **for** (j = 0; j < K; j++) val[i][j] = inf;
18. u.v = source;
19. u.k = 0;
20. u.w = 0;
21. Q.push(u);
22. **while** (!Q.empty()) {
23. u = Q.top();
24. Q.pop();
25. **for** (i = 0; i < deg[u.v]; i++) {
26. v.v = adj[u.v][i].v;
27. **int** cost = adj[u.v][i].w + u.w;
28. **for** (v.k = u.k; v.k < K; v.k++) {
29. **if** (cost == inf) **break**;
30. **if** (val[v.v][v.k] > cost) {
31. swap(cost, val[v.v][v.k]);
32. v.w = val[v.v][v.k];
33. Q.push(v);
34. **break**;
35. }
36. }
37. **for** (v.k++; v.k < K; v.k++) {
38. **if** (cost == inf) **break**;
39. **if** (val[v.v][v.k] > cost) swap(cost, val[v.v][v.k]);
40. }
41. }
42. }
43. }

**DIJKSTRA:**

1. #define mx 100005
2. vector < **int** > adj[mx], cost[mx];
3. **struct** node {
4. **int** u, w;
5. node(**int** a, **int** b) { u = a;w = b;}
6. **bool** operator < (**const** node & p) **const** {
7. **return** w > p.w;
8. }
9. };
10. **int** dist[mx], par[mx];
11. **int** dijkstra(**int** dest) {
12. MEM(dist, 63);
13. SET(par);
14. priority\_queue < node > q;
15. q.push(node(1, 0));
16. dist[1] = 0;
17. **while** (!q.empty()) {
18. node top = q.top();
19. q.pop();
20. **int** u = top.u;
21. **if** (u == dest) **return** dist[dest];
22. repI(i, adj[u].size()) {
23. **int** v = adj[u][i];
24. **if** (dist[u] + cost[u][i] < dist[v]) {
25. dist[v] = dist[u] + cost[u][i];
26. par[v] = u;
27. q.push(node(v, dist[v]));
28. }
29. }
30. }
31. **return** -1;
32. }

**BELLMAN FORD:**

1. #define MAXE 10005
2. #define MAXN 105
3. **int** dist[MAXN], edge\_u[MAXE], edge\_v[MAXE], edge\_cost[MAXE];
4. **int** main() {
5. **int** n, m;
6. getII(n, m);
7. MEM(dist, 63);
8. dist[1] = 0;
9. rep(i, m) getIII(edge\_u[i], edge\_v[i], edge\_cost[i]);
10. **int** neg\_cycle = **false**;
11. rep(step, n) {
12. **int** updated = **false**;
13. rep(i, m) {
14. **int** u = edge\_u[i], v = edge\_v[i];
15. **if** (dist[u] + edge\_cost[i] < dist[v]) {
16. updated = **true**;
17. **if** (step == n) neg\_cycle = **true**;
18. dist[v] = dist[u] + edge\_cost[i];
19. }
20. }
21. **if** (updated == **false**) **break**;
22. }
23. **if** (neg\_cycle == **false**) {
24. rep(i, n) cout << dist[i] << endl;
25. } **else** puts("Negative Cycle");
26. }

**FLOYD WARSHALL:**

1. **int** d[100][100]; // d[i][j] = distance from i to j
2. **int** midMan[100][100] //first set -1
3. rep(k, n) rep(i, n) rep(j, n)
4. **if** (d[i][j] > d[i][k] + d[k][j]) {
5. d[i][j] = d[i][k] + d[k][j];
6. next[i][j] = next[i][k];
7. }
8. // at 1st in all position = INF
9. // After floyd if diagonal has <INF -> cycle
10. // After floyd if diagonal has <0 -> Neg cycle
11. **void** print(**int** a, **int** b) {
12. **int** k = midMan[a][b];
13. **if** (k == -1) v.PB(b);
14. **else** {
15. print(k, b);
16. print(a, k);
17. }
18. }

**Tree Diameter:**

1. // From any node 1st find the farthest node(f1) from that node
2. // Then from F1 the farthest node(f2)
3. // Tree diameter = dist[f1][f2];

**Farthest node from a Given Node:**

1. /\* First find the tree diameter (f1,f2)
2. Then from given node dist[a][f1] & dist[a][f2] which one is maximum \*/

**Topological Sort:**

1. /\* In degree 0 gula k queue te rakhbo then oi gula 1 ta kore queue theke ber kore tader adjancent node er indegree 1 minus korbo. If any in degree decreases to 0 then push the node in queue. jodi n ta node na hoi then topological sort nei \*/

**Prufer Code to Tree:**

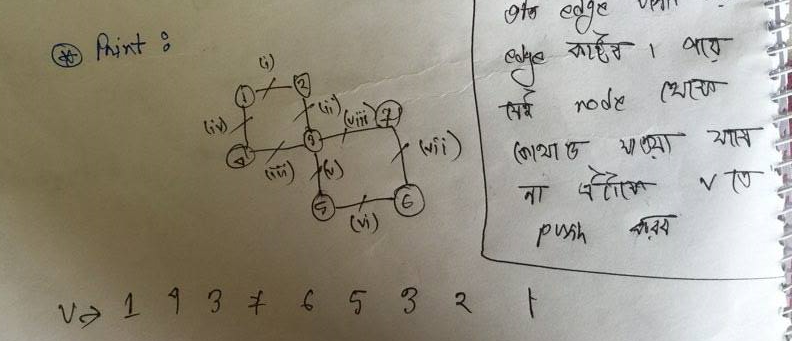
1. vector < **int** > prufer;
2. **void** pruferCodeToTree() {
3. /\*Stores number count of nodes in the prufer code\*/
4. map < **int**, **int** > mp;
5. /\* Set of integer absent in prufer code\*/
6. set < **int** > st;
7. **int** len = prufer.size();
8. **int** n = len + 2;
9. /\*Count frequency of nodes\*/
10. **for** (**int** i = 0; i < len; i++) {
11. **int** t = prufer[i];
12. mp[t]++;
13. }
14. /\*Find the absent nodes\*/
15. **for** (**int** i = 1; i <= n; i++) {
16. **if** (mp.find(i) == mp.end()) st.insert(i);
17. }
18. /\*Connect Edges\*/
19. **for** (**int** i = 0; i < len; i++) {
20. **int** a = prufer[i]; // First node
21. /\*Find the smallest number which is not present in prufer code now\*/
22. **int** b = \* st.begin(); // Second node
23. printf("%d %d\n", a, b); // Edge of the tree
24. st.erase(b); // Remove absent list
25. mp[a]--; // Remove from prufer code
26. **if** (mp[a] == 0) st.insert(a); // If a becomes absent
27. }
28. /\*The final edge\*/
29. printf("%d %d\n", \* st.begin(), \* st.rbegin());
30. }
31. // If the tree has n node then we can make n^(n-2) numbers of tree

**Tree to Prufer Code:**

1. /\* Among all the leaves which one is minimum we will push that in the queue. And cut the edge from the tree. It will be continue until the length of prufer code is (n-2) \*/

**Euler Circuit:**

1. /\* Euler Circuit: If we can visit every edge ONCE and can come back to the first node then this path is called Euler Circuit
2. Condition(Undirected):
3. (i) All degree even
4. (ii) graph connected
5. Condition(Directed):
6. (i) for each node InDegree = OutDegree
7. (ii) SCC
8. Print: Proti edge visit kore edge katbo( mark kore rakhbo). Pore jei node theke kothaou jawa jabena take v te push korbo. \*/

****

**Euler Path:**

1. /\* Euler Path: If we can visit every edge ONCE and can go to the another node then this path is called Euler Path
2. Condition(Undirected):
3. (i) Except 2 every node has even degree
4. (ii) graph connected
5. (iii) Source & Destination have odd degree
6. Condition(Directed):
7. (i) for source (out - in) = +1
8. (ii) for destination (out - in) = -1
9. (iii) for others (out - in) = 0
10. (iv) after adding an edge from Destination to Source the graph will be SCC
11. Print: For undirected, graph traversal will be started from any odd degree node like euler circuit.
12. For directed, graph traversal will be strated from that node which has (out-in) = +1 like euler circuit\*/

**MST(KRUSKAL):**

1. #define MAXN 100005
2. **struct** edge {
3. **int** u, v, w;
4. **bool** operator < (**const** edge & p) **const** {
5. **return** w < p.w;
6. }
7. };
8. **int** par[MAXN];
9. vector < edge > e;
10. **int** find(**int** r) {
11. **if** (par[r] == r) **return** r;
12. **return** par[r] = find(par[r]);
13. }
14. **int** mst(**int** n) {
15. sort(e.begin(), e.end());
16. **for** (**int** i = 1; i <= n; i++) par[i] = i;
17. **int** count = 0, s = 0;
18. **for** (**int** i = 0; i < (**int**) e.size(); i++) {
19. **int** u = find(e[i].u);
20. **int** v = find(e[i].v);
21. **if** (u != v) {
22. par[u] = v;
23. count++;
24. s += e[i].w;
25. **if** (count == n - 1) **break**;
26. }
27. }
28. ///check if a mst exist or not by count
29. **return** s;
30. }

**MST(PRIMS):**

1. **typedef** vector < vector < pii > > Graph;
2. **long** **long** prim(Graph & g, vector < **int** > & pred) {
3. **int** n = g.size();
4. pred.assign(n, -1);
5. vector < **bool** > vis(n);
6. vector < **int** > prio(n, INT\_MAX);
7. prio[0] = 0;
8. priority\_queue < pii, vector < pii > , greater < pii > > q;
9. q.push(make\_pair(0, 0));
10. **long** **long** res = 0;
11. **while** (!q.empty()) {
12. **int** d = q.top().first;
13. **int** u = q.top().second;
14. q.pop();
15. **if** (vis[u])
16. **continue**;
17. vis[u] = **true**;
18. res += d;
19. **for** (**int** i = 0; i < (**int**) g[u].size(); i++) {
20. **int** v = g[u][i].first;
21. **if** (vis[v])
22. **continue**;
23. **int** nprio = g[u][i].second;
24. **if** (prio[v] > nprio) {
25. prio[v] = nprio;
26. pred[v] = u;
27. q.push(make\_pair(nprio, v));
28. }
29. }
30. }
31. **return** res;
32. }
33. **int** main() {
34. Graph g(3);
35. g[0].push\_back(make\_pair(1, 10));
36. vector < **int** > prio;
37. **long** **long** res = prim(g, prio);
38. cout << res << endl;
39. }

**MAXIMUM BIPARTITE MATCHING:**

1. /\* যদি কোন গ্রাফ এর N টা নোড থাকে , এমন এই N টা নোডকে U, V দুইটা Independent   set এ বিভক্ত করা যাবে যাতে U Set  এর প্রত্যেকটা নোড এর সাথে V set  এর কোন না কোন নোড এর সাথে connect থাকবে । এই গ্রাফ এ কোন odd cycle থাকবে না। যেহেতু U,V দুইটা মাত্রই set এ নোড গুলো বিভক্ত odd cycle থাকা possible ও না। আর Independent set U,V এর নিজেদের মধ্যে কোন কানেকশন থাকবে না। মানে U set এর কোন নোড নিজেদের মধ্যে connected থাকবে না।
2. \*/
3. **int** adj[MAX][MAX], deg[MAX], Left[MAX], Right[MAX], m, n;
4. **bool** visited[MAX];
5. **bool** bpm(**int** u) {
6. **for** (**int** i = 0, v; i < deg[u]; i++) {
7. v = adj[u][i];
8. **if** (visited[v]) **continue**;
9. visited[v] = **true**;
10. **if** (Right[v] == -1 || bpm(Right[v])) {
11. Right[v] = u, Left[u] = v;
12. **return** **true**;
13. }
14. }
15. **return** **false**;
16. }
17. **int** bipartiteMatching() { // Returns Maximum Matching
18. memset(Left, -1, **sizeof**(Left));
19. memset(Right, -1, **sizeof**(Right));
20. **int** i, cnt = 0;
21. **for** (i = 0; i < m; i++) {
22. memset(visited, 0, **sizeof**(visited));
23. **if** (bpm(i)) cnt++;
24. }
25. **return** cnt;
26. }

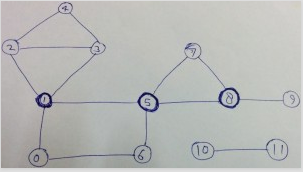
**2-SAT:**

1. /\* 1. The nodes need to be split. So change convert() accordingly.
2. 2. Using clauses, populate scc edges.
3. 3. Call possible, to find if a valid solution is possible or not.
4. 4. Dont forget to keep space for !A variables \*/
5. **struct** SAT2 {
6. SCC scc; /// This is from SCC Class
7. SAT2(): bfscc(1) {}
8. **void** clear() {
9. scc.clear();
10. }
11. **int** convert(**int** n) { ///Change here. Depends on how input is provided
12. **int** x = ABS(n);
13. x--;
14. x \*= 2;
15. **if** (n < 0) x ^= 1;
16. **return** x;
17. }
18. **void** mustTrue(**int** a) { ///A is True
19. scc.adj[a ^ 1].pb(a);
20. }
21. **void** orClause(**int** a, **int** b) { /// A || B clause
22. //!a->b !b->a
23. scc.adj[a ^ 1].pb(b);
24. scc.adj[b ^ 1].pb(a);
25. }
26. /// Out of all possible option, only one is true
27. **void** atMostOneClause(**int** a[], **int** n, **int** flag) {
28. **if** (flag == 0) { /// At most one can be false
29. FOR(i, 0, n) {
30. a[i] = a[i] ^ 1;
31. }
32. }
33. FOR(i, 0, n) {
34. FOR(j, i + 1, n) {
35. orClause(a[i] ^ 1, a[j] ^ 1); /// !a || !b both being true not allowed
36. }
37. }
38. }
39. ///Send n, total number of nodes, after expansion
40. **bool** possible(**int** n) {
41. scc.findSCC(n);
43. FOR(i, 0, n) {
44. **int** a = i, b = i ^ 1;
45. ///Falls on same cycle a and !a.
46. **if** (scc.cycle[a] == scc.cycle[b]) **return** **false**;
47. }
48. ///Valid solution exists
49. **return** **true**;
50. }
51. ///To determine if A can be true. It cannot be true, if a path exists from A to !A.
52. **int** vis[SAT2NODE], qqq[SAT2NODE], bfscc;
53. **void** bfs(**int** s) {
54. bfscc++;
55. **int** qs = 0, qt = 0;
56. vis[s] = bfscc;
57. qqq[qt++] = s;
58. **while** (qs < qt) {
59. s = qqq[qs++];
60. FOR(i, 0, SZ(scc.adj[s]) - 1) {
61. **int** t = scc.adj[s][i];
62. **if** (vis[t] != bfscc) {
63. vis[t] = bfscc;
64. qqq[qt++] = t;
65. }
66. }
67. }
68. }
70. }
71. sat2;

**ERDOS AND GALLAI THEOREM:**

1. // Given the degrees of the vertices of a graph, is it possible to construct such graph Input - the deg[] array
2. **int** deg[MM], n, degSum[MM], ind[MM], minVal[MM];
3. **bool** ErdosGallai() { // 1 indexed
4. **bool** poss = **true**;
5. **int** i, sum = 0, j, r;
6. **for** (i = 1; i <= n; i++) {
7. **if** (deg[i] >= n) poss = **false**;
8. sum += deg[i];
9. }
10. //Summation of degrees has to be ODD and all degrees has to be < n - 1
11. **if** (!poss || (sum & 1) || (n == 1 && deg[1] > 0)) **return** **false**;
12. sort(deg + 1, deg + n + 1, greater < **int** > ());
13. degSum[0] = 0;
14. j = n;
15. **for** (i = 1; i <= n; i++) {
16. degSum[i] = degSum[i - 1] + deg[i]; //CONSTRUCTING: degSum
17. **for** (; j >= 1 && deg[j] < i; j--); //CONSTRUCTING: ind
18. ind[i] = j + 1;
19. }
20. //CONSTRUCTING : minVal
21. **for** (r = 1; r < n; r++) {
22. j = ind[r];
23. **if** (j == n + 1) minVal[r] = (n - r) \* r;
24. **else** **if** (j <= r) minVal[r] = degSum[n] - degSum[r];
25. **else** {
26. minVal[r] = degSum[n] - degSum[j - 1];
27. minVal[r] += (j - r - 1) \* r;
28. }
29. }
30. //Checking : Erdos & Gallai Theorem
31. **for** (r = 1; r < n; r++)
32. **if** (degSum[r] > (r \* (r - 1) + minVal[r])) **return** **false**;
33. **return** **true**;
34. }

**BICONNECTED COMPONENT:**



1. /\*In above graph, following are the biconnected components:
2. 4–2 3–4 3–1 2–3 1–2
3. 8–9
4. 8–5 7–8 5–7
5. 6–0 5–6 1–5 0–1
6. 10–11
7. An undirected graph is called Biconnected if there are two vertex-disjoint paths between any two vertices.
8. Idea is to store visited edges in a stack while DFS on a graph and keep looking for Articulation Points (highlighted in above figure). As soon as an Articulation Point u is found, all edges visited while DFS from node u onwards will form one biconnected component. When DFS completes for one connected component, all edges present in stack will form a biconnected component.
9. If there is no Articulation Point in graph, then graph is biconnected and so there will be one biconnected component which is the graph itself.
10. \*/

**LIS NlogK:**

1. ///input must be 0 indexed
2. vector < **int** > Sequence, I, L;
3. **int** LisNlogK() {
4. **int** i;
5. I.clear();
6. L.clear();
7. I.push\_back(-INF);
8. **for** (i = 1; i <= n; i++) I.push\_back(INF);
9. **int** LisLength = 0;
10. **for** (i = 0; i < n; i++) {
11. **int** low, high, mid;
12. low = 0;
13. high = LisLength;
14. **while** (low <= high) {
15. mid = (low + high) / 2;
16. **if** (I[mid] < Sequence[i])
17. low = mid + 1;
18. **else**
19. high = mid - 1;
20. }
21. I[low] = Sequence[i];
22. **if** (LisLength < low)
23. LisLength = low;
24. L.push\_back(low);
25. }
26. **return** LisLength;
27. }
28. **void** findSequence(**int** length) {
29. **int** ind;
30. **for** (**int** j = L.size() - 1; j >= 0; j--) {
31. **if** (L[j] == length) {
32. ind = j;
33. **break**;
34. }
35. }
36. stack < **int** > st;
37. **int** mx = length - 1;
38. st.push(Sequence[ind]);
39. **for** (**int** i = ind - 1; i >= 0; i--) {
40. **if** (L[i] == mx && Sequence[ind] > Sequence[i]) {
41. st.push(Sequence[i]);
42. ind = i;
43. mx--;
44. }
45. }
46. **while** (!st.empty()) {
47. cout << st.top() << endl;
48. st.pop();
49. }
51. }

**2D LIS (N log N):**

1. **typedef** pair < **int**, **int** > pii;
2. pii p[100005];
3. set < pii > s[100005];
4. set < pii > ::iterator it, it1;
5. **int** main() {
6. **int** n, i, lo, hi, mid, lb, k, t, cs = 1;
7. scanf("%d", & n);
8. **for** (i = 0; i < n; i++) scanf("%d %d", & p[i].first, & p[i].second);
9. s[0].insert(p[0]);
10. k = 0;
11. **for** (i = 1; i < n; i++) {
12. lo = 0;
13. hi = k, lb = -1;
14. **while** (lo <= hi) {
15. mid = (lo + hi) / 2;
16. it = s[mid].lower\_bound(p[i]);
17. **if** (it != s[mid].begin()) {
18. it1 = it, it1--;
19. **if** (( \* it1).first == p[i].first) it--;
20. }
21. **if** (it != s[mid].begin() && ( \* (--it)).second < p[i].second)
22. lo = mid + 1, lb = max(lb, mid);
23. **else** hi = mid - 1;
24. }
25. lb++;
26. k = max(k, lb);
27. it = s[lb].lower\_bound(pii(p[i].first, -inf));
28. **if** (it == s[lb].end() || (( \* it).first > p[i].first || ( \* it).second > p[i].second))
29. s[lb].insert(p[i]);
30. it = s[lb].upper\_bound(p[i]);
31. **while** (it != s[lb].end()) {
32. **if** (( \* it).first >= p[i].first && ( \* it).second >= p[i].second) {
33. it1 = it, it1++;
34. s[lb].erase(it);
35. it = it1;
36. } **else** **break**;
37. }
38. }
39. printf("%d\n", k + 1);
40. }

**LCS 1D:**

1. Outline: O(nm) algorithm foR the LCS With O(n) spAce
2. **int** m[2][1000]; // instead of [1000][1000]
3. **for** (i = M; i >= 0; i--) {
4. ii = i & 1;
5. **for** (j = N; j >= 0; j--) {
6. **if** (i == M || j == N) {
7. m[ii][j] = 0;
8. **continue**;
9. }
10. **if** (s1[i] == s2[j]) m[ii][j] = 1 + m[1 - ii][j + 1];
11. **else** m[ii][j] = max(m[ii][j + 1], m[1 - ii][j]);
12. }
13. }
14. cout << m[0][0]; // if you want m[x][y], write m[x&1][y]

**Matrix Chain Multiplication (MCM) O(MAX \* MAX):**

1. #define MAX 100
2. **int** row[MAX], col[MAX];
3. **int** dp[MAX][MAX];
4. **bool** visited[MAX][MAX];
5. **int** f(**int** beg, **int** end) {
6. **if** (beg >= end) **return** 0;
7. **if** (visited[beg][end]) **return** dp[beg][end];
8. **int** ans = 1 << 30; //২^৩০ কে ইনফিনিটি ধরছি
9. **for** (**int** mid = beg; mid < end; mid++) //দুইভাগে ভাগ করছি
10. {
11. **int** opr\_left = f(beg, mid); //opr = multiplication operation
12. **int** opr\_right = f(mid + 1, end);
13. **int** opr\_to\_multiply\_left\_and\_right = row[beg] \* col[mid] \* col[end];
14. **int** total = opr\_left + opr\_right + opr\_to\_multiply\_left\_and\_right;
15. ans = min(ans, total);
16. }
17. visited[beg][end] = 1;
18. dp[beg][end] = ans;
19. **return** dp[beg][end];
20. }
21. **int** main() {
22. **int** n; cin >> n;
23. FOR(i,0, n-1) cin >> row[i] >> col[i];
24. cout << f(0, n - 1) << endl;
25. }

**HISTOGRAM:**

1. #define SIZE 120
2. **int** arr[SIZE][SIZE];
3. **int** cum[SIZE][SIZE];
4. **int** histogram(**int** row, **int** col) {
5. stack < pii > st;
6. st.push(pii(-INF, 0));
7. **int** lft[SIZE], rght[SIZE];
8. rep(i, col) {
9. **while** (st.top().ff >= cum[row][i]) st.pop();
10. lft[i] = st.top().ss;
11. st.push(pii(cum[row][i], i));
12. }
13. **while** (!st.empty()) st.pop();
14. st.push(pii(-INF, col + 1));
15. ROF(i, col, 1) {
17. **while** (st.top().ff >= cum[row][i]) st.pop();
18. rght[i] = st.top().ss;
19. st.push(pii(cum[row][i], i));
20. }
21. **int** res = -INF;
22. rep(i, col) {
23. res = max(res, cum[row][i] \* (rght[i] - lft[i] - 1));
24. }
25. **return** res;
26. }

**LONGEST PALINDROME (MANACHER ALGORITHM):**

1. // Transform S into T.
2. // For example, S = "abba", T = "^#a#b#b#a#$".
3. // ^ and $ signs are sentinels appended to each end to avoid bounds checking
4. string preProcess(string s) {
5. **int** n = s.length();
6. **if** (n == 0) **return** "^$";
7. string ret = "^";
8. **for** (**int** i = 0; i < n; i++)
9. ret += "#" + s.substr(i, 1);
11. ret += "#$";
12. **return** ret;
13. }
14. string longestPalindrome(string s) {
15. string T = preProcess(s);
16. **int** n = T.length();
17. **int** \* P = **new** **int**[n];
18. **int** C = 0, R = 0;
19. **for** (**int** i = 1; i < n - 1; i++) {
20. **int** i\_mirror = 2 \* C - i; // equals to i' = C - (i-C)
22. P[i] = (R > i) ? min(R - i, P[i\_mirror]) : 0;
24. // Attempt to expand palindrome centered at i
25. **while** (T[i + 1 + P[i]] == T[i - 1 - P[i]])
26. P[i]++;
28. // If palindrome centered at i expand past R,
29. // adjust center based on expanded palindrome.
30. **if** (i + P[i] > R) {
31. C = i;
32. R = i + P[i];
33. }
34. }
35. // Find the maximum element in P.
36. **int** maxLen = 0;
37. **int** centerIndex = 0;
38. **for** (**int** i = 1; i < n - 1; i++) {
39. **if** (P[i] > maxLen) {
40. maxLen = P[i];
41. centerIndex = i;
42. }
43. }
44. **delete**[] P;
45. **return** s.substr((centerIndex - 1 - maxLen) / 2, maxLen);
46. }

**2d MAX SUM: (dipta007)**

1. **struct** kadane //Structure for 1D-Kadane Algorithm
2. {
3. **int** u, d;
4. ll sum, area;
5. kadane() {
6. u = -1, d = -1, sum = 0, area = 0;
7. }
8. };
9. ll a[104][104], jaBerHoilo;
10. pii twoPoint(ll kad[104], **int** n, ll k) {
11. **int** low = 0, maxL = -1, maxR = -1, maxLen = 0;
12. ll maxSum = 0, sum = 0;
13. FOR(high, 0, n - 1) {
14. sum += kad[high];
15. **while** (sum > k) {
16. sum -= kad[low];
17. low++;
18. }
19. **int** len = high - low + 1;
20. **if** (len > maxLen) {
21. maxSum = sum; maxLen = len;
22. maxL = low; maxR = high;
23. } **else** **if** (len == maxLen && sum < maxSum) {
24. maxSum = sum; maxLen = len;
25. maxL = low; maxR = high;
26. }
27. }
28. jaBerHoilo = maxSum;
29. **return** pii(maxL, maxR);
30. }
31. **int** main() {
32. **int** t;
33. getI(t);
34. FOR(ci, 1, t) {
35. **int** n, m, k;
36. getIII(n, m, k);
37. FOR(i, 0, n - 1)
38. FOR(j, 0, m - 1)
39. getL(a[i][j]);
40. **int** r = n, c = m, maxL, maxR, maxU, maxD;
41. ll maxSum = 0, maxArea = 0, kad[r];
42. **for** (**int** left = 0; left < c; left++) {
43. CLR(kad);
44. **for** (**int** right = left; right < c; right++) {
45. **for** (**int** i = 0; i < r; i++) {
46. kad[i] = kad[i] + a[i][right];
47. }
48. pii res = twoPoint(kad, r, k);
49. **if** (res.ff == -1 || res.ss == -1) **continue**;
50. kadane maxy;
51. maxy.sum = jaBerHoilo;
52. maxy.u = res.ff; maxy.d = res.ss;
53. maxy.area = (ll) abs(maxy.d - maxy.u + 1) \* (ll) abs(right - left + 1);
54. **if** (maxy.area > maxArea) {
55. maxArea = maxy.area;
56. maxSum = maxy.sum;
57. maxL = left; maxR = right;
58. maxU = maxy.u; maxD = maxy.d;
59. } **else** **if** (maxy.area == maxArea && maxy.sum < maxSum) {
60. maxArea = maxy.area;
61. maxSum = maxy.sum;
62. maxL = left; maxR = right;
63. maxU = maxy.u; maxD = maxy.d;
64. }
65. }
66. } ///end of 2d kadane
67. printf("Case #%d: %lld %lld\n", ci, maxArea, maxSum);
68. }
69. }

**2d MAX SUM: (mamun4122)**

1. #define SIZE 105
2. **int** arr[SIZE][SIZE], vertical[SIZE][SIZE], cum[SIZE][SIZE];
3. **int** tmp[SIZE];
4. **int** twopointer(**int** frst, **int** scnd) {
5. CLR(tmp);
6. rep(i, m) tmp[i]=vertical[scnd][i]-vertical[frst-1][i];
7. **int** lft = 1, rght = 1, res = -INF, sum = 0;
8. **while** (rght <= m) {
9. **if** (sum + tmp[rght] >= tmp[rght]) sum += tmp[rght];
10. **else** {
11. **while**(lft< rght && sum + tmp[rght] < tmp[rght]) {
12. sum -= tmp[lft];
13. lft++;
14. }
15. sum += tmp[rght];
16. }
17. res = max(res, sum);
18. rght++;
19. }
20. res = max(res, sum);
21. **return** res;
22. }
23. **int** findmaximumsum(**int** r, **int** c) {
24. **int** n = r, m = c;
25. CLR(vertical);
26. CLR(cum);
27. ///first find the vertical cum sum of every row
28. **for** (**int** i = 1; i <= n; i++) {
29. **for** (**int** j = 1; j <= m; j++)
30. vertical[i][j] = vertical[i - 1][j] + arr[i][j];
31. }
32. **int** ans = -INT\_MAX;
33. **for** (**int** first = 1; first <= n; first++) {
34. **for** (**int** scnd = first; scnd <= n; scnd++) {
35. ans = max(ans, twopointer(first, scnd));
36. }
37. }
38. **return** ans;
39. }

**COIN CHANGE (II):**

1. /\*In a strange shop there are n types of coins of value A1, A2 ... An. You have to find the number of ways you can make M using the coins. You can use any coin at most M times.\*/
2. **int** t, n, m, val[105];
3. **int** dp[105][10005];
4. #define mod 100000007
5. **int** main() {
6. getI(t);
7. rep(cs, t) {
8. getII(n, m);
9. **int** x;
10. rep(i, n) getI(val[i]);
11. rep(i, n) dp[i][0] = 1;
12. rep(i, m) dp[0][i] = 0;
13. rep(i, n) {
14. rep(j, m) {
15. **if** (j < val[i]) dp[i][j] = dp[i - 1][j] % mod;
16. **else** dp[i][j] = ((dp[i - 1][j] % mod + dp[i][j - val[i]] % mod) % mod);
17. }
18. }
19. printf("Case %d: %d\n", cs, dp[n][m]);
20. }
21. }

**COIN CHANGE (III):**

1. /\*In a strange shop there are n types of coins of value A1, A2 ... An. C1, C2, ... Cn denote the number of coins of value A1, A2 ... An respectively. You have to find the number of different values (from 1 to m), which can be produced using these coins.\*/
2. **int** t, n, m;
3. **int** arr[105], val[105];
4. **int** dp[100005], need[100005];
5. **int** main() {
6. getI(t);
7. rep(cs, t) {
8. CLR(dp);
9. getII(n, m);
10. rep(i, n) getI(val[i]);
11. rep(i, n) getI(arr[i]);
12. dp[0] = 1;
13. **int** ans = 0;
14. rep(i, n) {
15. CLR(need);
16. **for** (**int** j = val[i]; j <= m; j++) {
17. **if** (!dp[j] && dp[j - val[i]] && need[j - val[i]] + 1 <= arr[i]) {
18. ans++;
19. dp[j] = 1;
20. need[j] = need[j - val[i]] + 1;
21. }
22. }
23. }
24. printf("Case %d: %d\n", cs, ans);
25. }
26. }

**DIGIT DP (dipta007) :**

1. **const** **int** NX = 70;
2. ll dp[2][2][NX][NX];
3. **int** vis[2][2][NX][NX];
4. **int** lim, tt;
5. vector < **int** > inp;
6. ll DP(**int** pos, **int** isSmall, **int** isStart, **int** value) {
7. **if** (pos == lim) **return** value;
8. ll & ret = dp[isSmall][isStart][pos][value];
9. **int** & v = vis[isSmall][isStart][pos][value];
10. **if** (v == tt) **return** ret;
11. v = tt;
12. **int** ses = isSmall ? 9 : inp[pos];
13. **int** i;
14. ret = 0;
15. **if** (!isStart) {
16. **for** (i = 0; i <= ses; i++) {
17. ret += DP(pos + 1, isSmall | i < inp[pos], 0, (i == 0) + value);
18. }
19. } **else** {
20. **for** (i = 1; i <= ses; i++) {
21. ret += DP(pos + 1, isSmall | i < inp[pos], 0, (i == 0) + value);
22. }
23. ret += DP(pos + 1, 1, 1, 0);
24. }
25. **return** ret;
26. }
27. ll Cal(ll x) {
28. **if** (x < 0) **return** 0;
29. **if** (x <= 9) **return** 1;
30. inp.clear();
31. **while** (x) {
32. inp.pb(x % 10);
33. x /= 10;
34. }
35. reverse(inp.begin(), inp.end());
36. lim = inp.size();
37. tt++;
38. **return** DP(0, 0, 1, 0) + 1; //for '0' case  (+1)
39. }
40. **int** main() {
41. **int** cs, t;
42. getI(t);
43. **for** (cs = 1; cs <= t; cs++) {
44. ll n, m;
45. getLL(n, m);
46. ll ans = Cal(m) - Cal(n - 1);
47. printf("Case %d: %lld\n", cs, ans);
48. }
49. }

**DIGIT DP (mamun4122) :**

1. **int** tot;
2. vector < **int** > dig;
3. ll dp[20][2][200][2];
4. ll call(**int** pos, **int** flag, **int** sum, **int** strt) {
5. **if** (pos == tot) **return** sum;
6. ll & ret = dp[pos][flag][sum][strt];
7. **if** (ret != -1) **return** ret;
8. ll ans = 0;
9. **if** (pos == 0) {
10. repI(i, dig[pos] + 1) {
11. ans += call(pos + 1, i == dig[pos], sum + (i == 0 && strt), (strt || i != 0));
12. }
13. } **else** {
14. **if** (flag) {
15. **for** (**int** i = 0; i <= dig[pos]; i++) {
16. ans += call(pos + 1, i == dig[pos], sum + (i == 0 && strt), (strt || i != 0));
17. }
18. } **else** {
19. repI(i, 10)
20. ans += call(pos + 1, 0, sum + (i == 0 && strt), (strt || i != 0));
21. }
22. }
23. **return** ret = ans;
24. }
25. **void** calc(ll num) {
26. dig.clear();
27. **while** (num) {
28. dig.push\_back(num % 10);
29. num /= 10;
30. }
31. reverse(ALL(dig));
32. tot = dig.size();
33. }
34. **int** main() {
35. **int** t, n, m;
36. getI(t);
37. rep(cs, t) {
38. SET(dp);
39. ll a, b;
40. getLL(a, b);
41. a--;
42. ll ansa, ansb;
43. **if** (a < 0) ansa = 0;
44. **else** **if** (a < 10) ansa = 1;
45. **else** {
46. calc(a);
47. ansa = call(0, 0, 0, 0) + 1;
48. }
49. SET(dp);
50. calc(b);
51. ansb = call(0, 0, 0, 0) + 1;
52. printf("Case %d: %lld\n", cs, ansb - ansa);
53. }
54. }

**EDIT DISTANCE:**

1. **int** MOD = 1000000007;
2. **int** dp[4000][4000];
3. **int** main() {
4. **int** t;
5. getI(t);
6. getchar();
7. **for** (**int** ci = 1; ci <= t; ci++) {
8. string a, b;
9. cin >> a >> b;
10. **int** la = a.size();
11. **int** lb = b.size();
12. **for** (**int** i = 0; i <= la; i++)
13. dp[0][i] = i;
14. **for** (**int** i = 0; i <= lb; i++)
15. dp[i][0] = i;
16. **for** (**int** i = 1; i <= lb; i++) {
17. **for** (**int** j = 1; j <= la; j++) {
18. **if** (a[j - 1] == b[i - 1]) {
19. dp[i][j] = dp[i - 1][j - 1];
20. } **else** {
21. dp[i][j] = min(dp[i - 1][j], min(dp[i - 1][j - 1], dp[i][j - 1])) + 1;
22. }
23. }
24. }
25. printf("%d\n", dp[lb][la]);
26. }
27. }

**NIM:**

1. /\*নিম-গেম এ দুইজন খেলোয়ার আর কিছু পাথরের স্তুপ(pile) থাকে। প্রতি চালে একজন খেলোয়াড় যেকোনো একটা স্তুপ থেকে এক বা একাধিক পাথর তুলে নিতে পারে। কেও চাল দিতে ব্যার্থ হলে হেরে যাবে। অর্থাৎ শেষ পাথরটা যে তুলে নিয়েছে সে গেমে জিতবে।
2. \*/
3. **if** (xorsum > 0) first win
4. **else** second win

**Spurge Grundy:**

1. **int** grundy[600][600];
2. **int** dirx[]={-2,-3,-2,-1,-1,1};
3. **int** diry[]={1,-1,-1,-2,-3,-2};
4. **int** calc(**int** x, **int** y) {
5. **if** (grundy[x][y] != -1)
6. **return** grundy[x][y];
7. set < **int** > st;
8. **for** (**int** i = 0; i < 6; i++) {
9. **int** posx = x + dirx[i];
10. **int** posy = y + diry[i];
11. **if** (posx >= 0 && posy >= 0)
12. st.insert(calc(posx, posy));
13. }
14. **int** ans = 0;
15. **while** (st.contains(ans)) ans++;
16. **return** grundy[x][y] = ans;
17. }
18. **int** main() {
19. **int** i, j, t, cs, n;
20. getI(t);
21. SET(grundy);
22. rep(cs, t) {
23. **int** ans = 0;
24. printf("Case %d: ", cs);
25. getI(n);
26. rep(i, n) {
27. **int** x, y;
28. getII(x, y);
29. ans ^= calc(x, y);
30. }
31. **if** (ans) puts("Alice");
32. **else** puts("Bob");
33. }
34. }

**MINMAX :**

1. **const** **int** MAXN = 100005;
2. **int** dp[MAXN];
3. **bool** vis[MAXN];
4. **int** moves[]={1,3,5};
5. **bool** valid\_move(**int** x) {
6. **return** x >= 0;
7. }
8. **bool** MINMAX(**int** x) {
9. **if** (x == 0) **return** **false**; ///LOSE
10. **if** (vis[x]) **return** dp[x];
11. vis[x] = 1;
12. FOR(i, 0, 2) {
13. **if** (valid\_move(x - moves[i]) && !MINMAX(x - moves[i]))
14. **return** dp[x] = **true**;
15. }
16. **return** dp[x] = **false**;
17. }
18. **int** main() {
19. **int** n;
20. getI(n);
21. CLR(vis);
22. **if** (MINMAX(n)) printf("First");
23. **else** printf("Second");
24. }

**SEGMENT TREE:**

1. #define mx 100001
2. **int** arr[mx];
3. **int** tree[mx \* 3];
4. **void** init(**int** node, **int** b, **int** e) {
5. **if** (b == e) {
6. tree[node] = arr[b];
7. **return**;
8. }
9. init(Left, b, mid);
10. init(Right, mid + 1, e);
11. tree[node] = tree[Left] + tree[Right];
12. }
13. **int** query(**int** node, **int** b, **int** e, **int** i, **int** j) {
14. **if** (i > e || j < b) **return** 0;
15. **if** (b >= i && e <= j) **return** tree[node];
16. **int** p1 = query(Left, b, mid, i, j);
17. **int** p2 = query(Right, mid + 1, e, i, j);
18. **return** p1 + p2;
19. }
20. **void** update(**int** node, **int** b, **int** e, **int** i, **int** newvalue) {
21. **if** (i > e || i < b) **return**;
22. **if** (b >= i && e <= i) {
23. tree[node] = newvalue;
24. **return**;
25. }
26. update(Left, b, mid, i, newvalue);
27. update(Right, mid + 1, e, i, newvalue);
28. tree[node] = tree[Left] + tree[Right];
29. }
30. ///lazy with propagation
31. **void** Propagate(**int** at, **int** L, **int** R) {
32. **int** mid = (L + R) / 2;
33. **int** left\_at = at \* 2, left\_L = L, left\_R = mid;
34. **int** right\_at = at \* 2 + 1, right\_L = mid + 1, right\_R = R;
35. toggle[at] = 0;
36. toggle[left\_at] ^= 1;
37. toggle[right\_at] ^= 1;
38. on[left\_at] = left\_R - left\_L + 1 - on[left\_at];
39. on[right\_at] = right\_R - right\_L + 1 - on[right\_at];
40. }
41. **void** update(**int** at, **int** L, **int** R, **int** l, **int** r) {
42. **if** (r < L || R < l) **return**;
43. **if** (l <= L && R <= r) {
44. toggle[at] ^= 1;
45. on[at] = R - L + 1 - on;
46. **return**;
47. }
48. **if** (toggle[at]) Propagate(at, L, R);
49. **int** mid = (L + R) / 2;
50. update(at \* 2, L, mid, l, r);
51. update(at \* 2 + 1, mid + 1, R, l, r);
52. on[at] = on[at \* 2] + on[at \* 2 + 1];
53. }
54. **int** query(**int** at, **int** L, **int** R, **int** l, **int** r) {
55. **if** (r < L || R < l) **return**;
56. **if** (l <= L && R <= r) **return** on[at];
57. **if** (toggle[at]) Propagate(at, L, R);
58. **int** mid = (L + R) / 2;
59. **int** x = query(at \* 2, L, mid, l, r);
60. **int** y = query(at \* 2 + 1, mid + 1, R, l, r);
61. **return** x + y;
62. }

**Binary Indexed Tree (BIT):**

1. **int** query(**int** idx) {
2. **int** sum = 0;
3. **while** (idx > 0) {
4. sum += tree[idx];
5. idx -= idx & (-idx);
6. }
7. **return** sum;
8. }
9. **void** update(**int** idx, **int** x, **int** n)
10. //n is the size of the array, x is the number to add
11. {
12. **while** (idx <= n) {
13. tree[idx] += x;
14. idx += idx & (-idx);
15. }
16. }

**Range Minimum Query (RMQ) :**

1. **const** **int** inf = (1 << 28);
2. **template** < **typename** t > t MIN3(t a, t b, t c) {
3. **return** min(a, min(b, c));
4. }
5. **const** **int** sz = 100005;
6. **int** BLOCK[400];
7. **int** arr[sz];
8. **int** getId(**int** indx, **int** blockSZ) {
9. **return** indx / blockSZ;
10. }
11. **void** init(**int** sz) {
12. **for** (**int** i = 0; i <= sz; i++) BLOCK[i] = inf;
13. }
14. **void** update(**int** val, **int** indx, **int** blockSZ) {
15. **int** id = getId(indx, blockSZ);
16. BLOCK[id] = min(BLOCK[id], val);
17. }
18. **int** query(**int** L, **int** R, **int** blockSZ) {
19. **int** lid = getId(L, blockSZ);
20. **int** rid = getId(R, blockSZ);
21. **if** (lid == rid) {
22. **int** ret = inf;
23. **for** (**int** i = L; i <= R; i++) ret = min(ret, arr[i]);
24. **return** ret;
25. }
26. **int** m1 = inf, m2 = inf, m3 = inf;
27. **for** (**int** i = L; i < (lid + 1) \* blockSZ; i++) m1 = min(m1, arr[i]);
28. **for** (**int** i = lid + 1; i < rid; i++) m2 = min(m2, BLOCK[i]);
29. **for** (**int** i = rid \* blockSZ; i <= R; i++) m3 = min(m3, arr[i]);
30. **return** MIN3(m1, m2, m3);
31. }
32. **int** main() {
33. **int** N, Q;
34. scanf("%d %d", & N, & Q);
35. **int** blockSZ = sqrt(N);
36. init(blockSZ);
37. **for** (**int** i = 0; i < N; i++) {
38. **int** x;
39. scanf("%d", & x);
40. arr[i] = x;
41. update(x, i, blockSZ);
42. }
43. **while** (Q--) {
44. **int** x, y;
45. scanf("%d %d", & x, & y);
46. printf("%d\n", query(x, y, blockSZ));
47. }
48. }
49. //
50. //getId ফাংশনের কাজ হল কোন index কত নাম্বার block এ তা বের করে দেয়া।
51. //init ফাংশন সবগুলা block কে infinity ভ্যালু নিয়ে initialize করে নিচ্ছে
52. //update ফাংশন দিয়ে কোন একটা নির্দিষ্ট index এর ভ্যালু আপডেট করে দেয়া হচ্ছে।
53. //query ফাংশন দিয়ে x to y রেঞ্জের result calculation করা হচ্ছে।
54. //Line 25 এ যদি রেঞ্জ পুরোটা কোন একটা নির্দিষ্ট block এর sub part হয়ে থাকে, তাহলে main Array থেকে result calculation করে দিবে।
55. //Line 32, যদি রেঞ্জ এর lower bound এর কিছু অংশ নির্দিষ্ট একটা block এর sub part হয়ে থাকে তাহলে শুধু সেইটুক sub part এর result main Array থেকে calculate করে দিবে।
56. //Line 34, যদি রেঞ্জ এর upper bound এর কিছু অংশ নির্দিষ্ট একটা block এর sub part হয়ে থাকে তাহলে শুধু সেইটুক sub part এর result main Array থেকে নিবে।

**PREFIX TRIE:**

1. #define mx 26
2. **struct** node {
3. **bool** endmark;
4. node \* next[mx + 1];
5. node() {
6. endmark = 0;
7. **for** (**int** i = 0; i < mx; i++)
8. next[i] = NULL;
9. }
10. } \* root;
11. **void** insert(**char** \* str, **int** len) {
12. node \* curr = root;
13. **for** (**int** i = 0; i < len; i++) {
14. **int** id = str[i] - 'a';
15. **if** (curr - > next[id] == NULL)
16. curr - > next[id] = **new** node();
17. curr = curr - > next[id];
18. }
19. curr - > endmark = 1;
20. }
21. **bool** search(**char** \* str, **int** len) {
22. node \* curr = root;
23. **for** (**int** i = 0; i < len; i++) {
24. **int** id = str[i] - 'a';
25. **if** (curr - > next[id] == NULL) **return** **false**;
26. curr = curr - > next[id];
27. }
28. **return** curr - > endmark; /// returns 1 or 0
29. }
30. **void** del(node \* cur) /// send root here
31. {
32. **for** (**int** i = 0; i < mx; i++)
33. **if** (cur - > next[i])
34. del(cur - > next[i]);
35. **delete**(cur);
36. }
37. **int** main() {
38. root = **new** node();
39. **int** num\_word;
40. cin >> num\_word;
41. **for** (**int** i = 1; i <= num\_word; i++) {
42. **char** str[50];
43. scanf("%s", str);
44. insert(str, strlen(str));
45. }
46. **int** query;
47. cin >> query;
48. **for** (**int** i = 1; i <= query; i++) {
49. **char** str[50];
50. scanf("%s", str);
51. **if** (search(str, strlen(str))) puts("FOUND");
52. **else** puts("NOT FOUND");
53. }
54. del(root);
55. }

**KMP:**

1. **void** computeLPSArray(**char** \* pat, **int** M, **int** \* lps);
2. **void** KMPSearch(**char** \* pat, **char** \* txt) {
3. **int** M = strlen(pat);
4. **int** N = strlen(txt);
5. **int** \* lps = (**int** \* ) malloc(**sizeof**(**int**) \* M);
6. **int** j = 0;
7. computeLPSArray(pat, M, lps);
8. **int** i = 0; // index for txt[]
9. **while** (i < N) {
10. **if** (pat[j] == txt[i]) {
11. j++;
12. i++;
13. }
14. **if** (j == M) {
15. printf("Found pattern at index %d \n", i - j);
16. j = lps[j - 1];
17. } **else** **if** (pat[j] != txt[i]) {
18. **if** (j != 0)
19. j = lps[j - 1];
20. **else**
21. i = i + 1;
22. }
23. }
24. free(lps);
25. }
26. **void** computeLPSArray(**char** \* pat, **int** M, **int** \* lps) {
27. **int** len = 0;
28. **int** i;
29. lps[0] = 0;
30. i = 1;
31. // the loop calculates lps[i] for i = 1 to M-1
32. **while** (i < M) {
33. **if** (pat[i] == pat[len]) {
34. len++;
35. lps[i] = len;
36. i++;
37. } **else** // (pat[i] != pat[len])
38. {
39. **if** (len != 0) {
40. // This is tricky. Consider the example AAACAAAA and i = 7.
41. len = lps[len - 1];
42. } **else** {
43. lps[i] = 0;
44. i++;
45. }
46. }
47. }
48. }
49. **int** main() {
50. **char** \* txt = "ABABDABACDABABCABAB";
51. **char** \* pat = "ABABCABAB";
52. KMPSearch(pat, txt);
53. }

**Z ALGORITHM:**

1. **const** **int** NX = 1e5 + 10; // string size
2. **char** text[NX];
3. **int** Z[NX];
4. **void** Z\_Algorithm() {
5. **int** position, starting\_point, ending\_point;
6. **int** sz = strlen(text);
7. Z[0] = sz; // always ;
8. **for** (position = 1, starting\_point = 0, ending\_point = 0; position < sz; position++) {
9. **if** (position <= ending\_point) Z[position] = min(ending\_point - position + 1, Z[position - starting\_point]);
10. **while** (position + Z[position] < sz && text[Z[position]] == text[position + Z[position]]) ++Z[position];
11. **if** (position + Z[position] - 1 > ending\_point) // need to update
12. starting\_point = position, ending\_point = position + Z[position] - 1;
13. }
14. }
15. /\*\*\*\*\*\*prefix==suffix\*\*\*\*\*\*\*\*\*\*\*\*/
16. **for** (i = sz - 1; i >= 0; i--) {
17. **if** (Z[i] == sz - i) // suffix matches
18. }
19. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
20. **bool** zAlgorithm(string pattern, string target) {
21. string s = pattern + '$' + target;
22. **int** n = s.length();
23. vector < **int** > z(n, 0);
25. **int** goal = pattern.length();
26. **int** r = 0, l = 0, i;
27. **for** (**int** k = 1; k < n; k++) {
28. **if** (k > r) {
29. **for** (i = k; i < n && s[i] == s[i - k]; i++);
30. **if** (i > k) {
31. z[k] = i - k;
32. l = k;
33. r = i - 1;
34. }
35. } **else** {
36. **int** kt = k - l, b = r - k + 1;
37. **if** (z[kt] > b) {
38. **for** (i = r + 1; i < n && s[i] == s[i - k]; i++);
39. z[k] = i - k;
40. l = k;
41. r = i - 1;
42. }
43. }
44. **if** (z[k] == goal)
45. **return** **true**;
46. }
47. **return** **false**;
48. }

**AHO-CORASICK:**

1. #define MX 100 //small string length
2. **int** m, n, res;
3. **typedef** pair < **int**, **int** > Point;
4. **struct** NODE {
5. **int** cnt;
6. **bool** vis;
7. NODE \* next[27];
8. vector < NODE \* > out;
9. NODE() {
10. **for** (**int** i = 0; i < 27; i++) {
11. next[i] = NULL;
12. }
13. out.clear();
14. vis = **false**;
15. cnt = 0;
16. }~NODE() {
17. **for** (**int** i = 1; i < 27; i++)
18. **if** (next[i] != NULL && next[i] != **this**)
19. **delete** next[i];
20. }
21. } \* root;
22. **void** buildtrie(**char** dictionary[][MX], **int** n) // processing the dictionarytionary
23. {
24. root = **new** NODE();
25. /\*usual trie part\*/
26. **for** (**int** i = 0; i < n; i++) {
27. NODE \* p = root;
28. **for** (**int** j = 0; dictionary[i][j]; j++) {
29. **char** c = dictionary[i][j] - 'a' + 1;
30. **if** (!p - > next[c])
31. p - > next[c] = **new** NODE();
32. p = p - > next[c];
33. }
34. }
35. /\* Pushing the nodes adjacent to root into queue \*/
36. queue < NODE \* > q;
37. **for** (**int** i = 0; i < 27; i++) {
38. **if** (!root - > next[i])
39. root - > next[i] = root;
40. **else** {
41. q.push(root - > next[i]);
42. root - > next[i] - > next[0] = root; // ->next[0] = back Pointer
43. }
44. }
45. /\* Building Aho-Corasick tree \*/
46. **while** (!q.empty()) {
47. NODE \* u = q.front(); //parent node
48. q.pop();
49. **for** (**int** i = 1; i < 27; i++) {
50. **if** (u - > next[i]) {
51. NODE \* v = u - > next[i]; // child node
52. NODE \* w = u - > next[0]; // back pointer of parent node
53. **while** (!w - > next[i]) // Until the char(i+'a'-1) child is found
54. w = w - > next[0]; // go up and up to back pointer.
55. v - > next[0] = w = w - > next[i]; // back pointer of v will be found child above.
56. w - > out.push\_back(v); // out will be used in dfs step.
57. // here w is the new found match node.
58. q.push(v); // Push v into queue.
59. }
60. }
61. }
62. }
63. **void** aho\_corasick(NODE \* p, **char** \* word) // Third step, processing the text.
64. {
65. **for** (**int** i = 0; word[i]; i++) {
66. **char** c = word[i] - 'a' + 1;
67. **while** (!p - > next[c])
68. p = p - > next[0];
69. p = p - > next[c];
70. p - > cnt++;
71. }
72. }
73. **int** dfs(NODE \* p) // DFS for counting.
74. {
75. **if** (p - > vis) **return** p - > cnt;
76. **for** (**int** i = 0; i < p - > out.size(); i++)
77. p - > cnt += dfs(p - > out[i]);
78. p - > vis = **true**;
79. **return** p - > cnt;
80. }
81. **char** query[1000100];
82. **char** dictionary[MX][MX];
83. **int** main() {
84. **int** t, tc, y, z;
85. **int** i, j, k, l, h;
86. **char** ch;
87. scanf("%d", & tc);
88. **for** (t = 1; t <= tc; t++) {
89. **int** n;
90. scanf("%d", & n);
91. scanf("%s", query);
92. **for** (**int** i = 0; i < n; ++i) {
93. scanf("%s", dictionary[i]);
94. }
95. buildtrie(dictionary, n);
96. aho\_corasick(root, query);
97. printf("Case %d:\n", t);
98. **for** (**int** i = 0; i < n; i++) {
99. NODE \* p = root;
100. **for** (**int** j = 0; dictionary[i][j]; j++) {
101. **char** c = dictionary[i][j] - 'a' + 1;
102. p = p - > next[c];
103. }
104. printf("%d\n", dfs(p));
105. }
106. **delete** root;
107. }
108. }

**SUFFIX ARRAY:**

1. #define MAX\_N 100010 // second approach: O(n log n)
2. **char** T[MAX\_N]; // the input string, up to 100K characters
3. **int** n; // the length of input string
4. **int** RA[MAX\_N], tempRA[MAX\_N]; // rank array and temporary rank array
5. **int** SA[MAX\_N], tempSA[MAX\_N]; // suffix array and temporary suffix array
6. **int** c[MAX\_N]; // for counting/radix sort
7. **char** P[MAX\_N]; // the pattern string (for string matching)
8. **int** m; // the length of pattern string
9. **int** Phi[MAX\_N]; // for computing longest common prefix
10. **int** PLCP[MAX\_N];
11. **int** LCP[MAX\_N]; // LCP[i] stores the LCP between previous suffix T+SA[i-1]
12. // and current suffix T+SA[i]
13. **bool** cmp(**int** a, **int** b) {
14. **return** strcmp(T + a, T + b) < 0;
15. } // compare
16. **void** constructSA\_slow() { // cannot go beyond 1000 characters
17. **for** (**int** i = 0; i < n; i++) SA[i] = i; // initial SA: {0, 1, 2, ..., n-1}
18. sort(SA, SA + n, cmp); // sort: O(n log n) \* compare: O(n) = O(n^2 log n)
19. }
20. **void** countingSort(**int** k) { // O(n)
21. **int** i, sum, maxi = max(300, n); // up to 255 ASCII chars or length of n
22. memset(c, 0, **sizeof** c); // clear frequency table
23. **for** (i = 0; i < n; i++) // count the frequency of each integer rank
24. c[i + k < n ? RA[i + k] : 0]++;
25. **for** (i = sum = 0; i < maxi; i++) {
26. **int** t = c[i];
27. c[i] = sum;
28. sum += t;
29. }
30. **for** (i = 0; i < n; i++) // shuffle the suffix array if necessary
31. tempSA[c[SA[i] + k < n ? RA[SA[i] + k] : 0]++] = SA[i];
32. **for** (i = 0; i < n; i++) // update the suffix array SA
33. SA[i] = tempSA[i];
34. }
35. **void** constructSA() { // this version can go up to 100000 characters
36. **int** i, k, r;
37. **for** (i = 0; i < n; i++) RA[i] = T[i]; // initial rankings
38. **for** (i = 0; i < n; i++) SA[i] = i; // initial SA: {0, 1, 2, ..., n-1}
39. **for** (k = 1; k < n; k <<= 1) { // repeat sorting process log n times
40. countingSort(k); // actually radix sort: sort based on the second item
41. countingSort(0); // then (stable) sort based on the first item
42. tempRA[SA[0]] = r = 0; // re-ranking; start from rank r = 0
43. **for** (i = 1; i < n; i++) // compare adjacent suffixes
44. tempRA[SA[i]] = // if same pair => same rank r; otherwise, increase r
45. (RA[SA[i]] == RA[SA[i - 1]] && RA[SA[i] + k] == RA[SA[i - 1] + k]) ? r : ++r;
46. **for** (i = 0; i < n; i++) // update the rank array RA
47. RA[i] = tempRA[i];
48. **if** (RA[SA[n - 1]] == n - 1) **break**; // nice optimization trick
49. }
50. }
51. **void** computeLCP\_slow() {
52. LCP[0] = 0; // default value
53. **for** (**int** i = 1; i < n; i++) { // compute LCP by definition
54. **int** L = 0; // always reset L to 0
55. **while** (T[SA[i] + L] == T[SA[i - 1] + L]) L++; // same L-th char, L++
56. LCP[i] = L;
57. }
58. }
59. **void** computeLCP() {
60. **int** i, L;
61. Phi[SA[0]] = -1; // default value
62. **for** (i = 1; i < n; i++) // compute Phi in O(n)
63. Phi[SA[i]] = SA[i - 1]; // remember which suffix is behind this suffix
64. **for** (i = L = 0; i < n; i++) { // compute Permuted LCP in O(n)
65. **if** (Phi[i] == -1) {
66. PLCP[i] = 0;
67. **continue**;
68. } // special case
69. **while** (T[i + L] == T[Phi[i] + L]) L++; // L increased max n times
70. PLCP[i] = L;
71. L = max(L - 1, 0); // L decreased max n times
72. }
73. **for** (i = 0; i < n; i++) // compute LCP in O(n)
74. LCP[i] = PLCP[SA[i]]; // put the permuted LCP to the correct position
75. }
76. pii stringMatching() { // string matching in O(m log n)
77. **int** lo = 0, hi = n - 1, mid = lo; // valid matching = [0..n-1]
78. **while** (lo < hi) { // find lower bound
79. mid = (lo + hi) / 2; // this is round down
80. **int** res = strncmp(T + SA[mid], P, m); // try to find P in suffix 'mid'
81. **if** (res >= 0) hi = mid; // prune upper half (notice the >= sign)
82. **else** lo = mid + 1; // prune lower half including mid
83. } // observe `=' in "res >= 0" above
84. **if** (strncmp(T + SA[lo], P, m) != 0) **return** pii(-1, -1); // if not found
85. pii ans;
86. ans.first = lo;
87. lo = 0;
88. hi = n - 1;
89. mid = lo;
90. **while** (lo < hi) { // if lower bound is found, find upper bound
91. mid = (lo + hi) / 2;
92. **int** res = strncmp(T + SA[mid], P, m);
93. **if** (res > 0) hi = mid; // prune upper half
94. **else** lo = mid + 1; // prune lower half including mid
95. } // (notice the selected branch when res == 0)
96. **if** (strncmp(T + SA[hi], P, m) != 0) hi--; // special case
97. ans.second = hi;
98. **return** ans;
99. } // return lower/upperbound as first/second item of the pair, respectively
100. pii LRS() { // returns a pair (the LRS length and its index)
101. **int** i, idx = 0, maxLCP = -1;
102. **for** (i = 1; i < n; i++) // O(n), start from i = 1
103. **if** (LCP[i] > maxLCP)
104. maxLCP = LCP[i], idx = i;
105. **return** pii(maxLCP, idx);
106. }
107. **int** owner(**int** idx) {
108. **return** (idx < n - m - 1) ? 1 : 2;
109. }
110. pii LCS() { // returns a pair (the LCS length and its index)
111. **int** i, idx = 0, maxLCP = -1;
112. **for** (i = 1; i < n; i++) // O(n), start from i = 1
113. **if** (owner(SA[i]) != owner(SA[i - 1]) && LCP[i] > maxLCP)
114. maxLCP = LCP[i], idx = i;
115. **return** pii(maxLCP, idx);
116. }
117. **int** main() {
118. //printf("Enter a string T below, we will compute its Suffix Array:\n");
119. strcpy(T, "GATAGACA");
120. n = (**int**) strlen(T);
121. T[n++] = '$';
122. // if '\n' is read, uncomment the next line
123. //T[n-1] = '$'; T[n] = 0;
124. constructSA\_slow(); // O(n^2 log n)
125. printf("The Suffix Array of string T = '%s' is shown below (O(n^2 log n) version):\n", T);
126. printf("i\tSA[i]\tSuffix\n");
127. **for** (**int** i = 0; i < n; i++) printf("%2d\t%2d\t%s\n", i, SA[i], T + SA[i]);
128. constructSA(); // O(n log n)
129. printf("\nThe Suffix Array of string T = '%s' is shown below (O(n log n) version):\n", T);
130. printf("i\tSA[i]\tSuffix\n");
131. **for** (**int** i = 0; i < n; i++) printf("%2d\t%2d\t%s\n", i, SA[i], T + SA[i]);
132. computeLCP(); // O(n)
133. // LRS demo
134. pii ans = LRS(); // find the LRS of the first input string
135. **char** lrsans[MAX\_N];
136. strncpy(lrsans, T + SA[ans.second], ans.first);
137. printf("\nThe LRS is '%s' with length = %d\n\n", lrsans, ans.first);
138. // stringMatching demo
139. //printf("\nNow, enter a string P below, we will try to find P in T:\n");
140. strcpy(P, "A");
141. m = (**int**) strlen(P);
142. // if '\n' is read, uncomment the next line
143. //P[m-1] = 0; m--;
144. pii pos = stringMatching();
145. **if** (pos.first != -1 && pos.second != -1) {
146. printf("%s is found SA[%d..%d] of %s\n", P, pos.first, pos.second, T);
147. printf("They are:\n");
148. **for** (**int** i = pos.first; i <= pos.second; i++)
149. printf("  %s\n", T + SA[i]);
150. } **else** printf("%s is not found in %s\n", P, T);
152. // LCS demo
153. //printf("\nRemember, T = '%s'\nNow, enter another string P:\n", T);
154. // T already has '$' at the back
155. strcpy(P, "CATA");
156. m = (**int**) strlen(P);
157. // if '\n' is read, uncomment the next line
158. //P[m-1] = 0; m--;
159. strcat(T, P); // append P
160. strcat(T, "#"); // add '$' at the back
161. n = (**int**) strlen(T); // update n
163. // reconstruct SA of the combined strings
164. constructSA(); // O(n log n)
165. computeLCP(); // O(n)
166. printf("\nThe LCP information of 'T+P' = '%s':\n", T);
167. printf("i\tSA[i]\tLCP[i]\tOwner\tSuffix\n");
168. **for** (**int** i = 0; i < n; i++)
169. printf("%2d\t%2d\t%2d\t%2d\t%s\n", i, SA[i], LCP[i], owner(SA[i]), T + SA[i]);
170. ans = LCS(); // find the longest common substring between T and P
171. **char** lcsans[MAX\_N];
172. strncpy(lcsans, T + SA[ans.second], ans.first);
173. printf("\nThe LCS is '%s' with length = %d\n", lcsans, ans.first);
174. **return** 0;
175. }

**GEOMETRY:**

# Misc Geometric Formula:

|  |  |
| --- | --- |
| **Triangle** | Circum Radius = a\*b\*c/(4\*area)  In Radius = area/s, where s = (a+b+c)/2  length of median to side c = sqrt(2\*(a\*a+b\*b)-c\*c)/2  length of bisector of angle C = sqrt(ab[(a+b)\*(a+b)-c\*c])/(a+b) |
| **Ellipse** | Area = PI\*a\*b  Circumference = 4a \*int(0,PI/2){sqrt(1-(k\*sint)\*(k\*sint))}dt  = 2\*PI\*sqrt((a\*a+b\*b)/2) approx  where k = sqrt((a\*a-b\*b)/a)  = PI\*(3\*(r1+r2)-sqrt[(r1+3\*r2)\*(3\*r1+r2)]) |
| **Spherical cap** | V = (1/3)\*PI\*h\*h\*(3\*r-h)  Surface Area = 2\*PI\*r\*h |
| **Spherical Sector** | V = (2/3)\*PI\*r\*r\*h |
| **Spherical Segment** | V = (1/6)\*PI\*h\*(3\*a\*a+3\*b\*b+h\*h) |
| **Torus** | V = 2\*PI\*PI\*R\*r\*r |
| **Truncated Conic** | V = (1/3)\*PI\*h\*(a\*a+a\*b+b\*b)  Surface Area = PI\*(a+b)\*sqrt(h\*h+(b-a)\*(b-a))  = PI\*(a+b)\*l |
| **Pyramidal frustum** | (1/3)\*h\*(A1+A2+sqrt(A1\*A2)) |

# Misc Trigonometric Functions and Formulas:

tan A/2 = +sqrt((1-cos A)/(1+cos A))

= sin A / (1+cos A)

= (1-cos A) / sin A

= cosec A – cot A

sin 3A = 3\*sin A – 4\*sincube A

cos 3A = 4\*coscube A – 3\*cos A

tan 3A = (3\*tan A-tancube A)/(1-3\*tansq A)

sin 4A = 4\*sin A\*cos A – 8\*sincube A\*cos A

cos 4A = 8\*cos4 A – 8\*cossq A + 1

[r\*(cost+i\*sint)]p = rp\*(cos pt+i\*sin pt)

**a**cos**x** + **b**sin**x = c,** x = 2nπ + α ± β, where

cosα = a / (sqrt(a^2+b^2)), cosβ = c / (sqrt(a^2+b^2));

2sinAcosB = sin(A+B) + sin(A-B)

2cosAsinB = sin(A+B) - sin(A-B)

2cosAcosB = cos(A-B) + cos(A+B)

2sinAsinB = cos(A-B) – cos(A+B)

sinC + sinD = 2sin[(C+D)/2]cos[(C-D)/2]

sinC - sinD = 2cos[(C+D)/2]sin[(C-D)/2]

cosD + cosC = 2cos[(C+D)/2]cos[(C-D)/2]

cosD - cosC = 2sin[(C+D)/2]sin[(C-D)/2]

# Misc Integration Formula:

a^x => a^x/ln(a)

1/sqrt(x\*x+a\*a) => ln(x+sqrt(x\*x+a\*a))

1/sqrt(x\*x-a\*a) => ln(x+sqrt(x\*x-a\*a))

1/(x\*sqrt(x\*x+a\*a) => -(1/a)\*ln([a+sqrt(x\*x+a\*a)]/x)

1/(x\*sqrt(a\*a-x\*x) => -(1/a)\*ln([a+sqrt(a\*a-x\*x)]/x)

# Misc Differentiation Formula:

asin x => 1/sqrt(1-x\*x) acos x => -1/sqrt(1-x\*x)

atan x => 1/(1+x\*x) acot x => -1/(1+x\*x)

asec x => 1/[x\*sqrt(x\*x-1)] acosec x => -1/[x\*sqrt(x\*x-1)]

a^x => a^x\*ln(x) cot x => -cosecsq x

sec x => sec x \* tan x cosec x => -cosec x \* cot x

# Mirror point(mx,my) of a point(x,y) w.r.to a line(ax+by+c=0):

1. **void** mirrorPoint(**double** a, **double** b, **double** c, **double** x, **double** y, **double** & mx, **double** & my) {
2. mx = x \* (a \* a - b \* b) - 2.0 \* a \* b \* y - 2.0 \* a \* c;
3. mx /= (a \* a + b \* b);
4. my = y \* (a \* a - b \* b) - 2.0 \* a \* b \* x - 2.0 \* b \* c;
5. my /= (a \* a + b \* b);
6. }

**Determining if a point lies on the interior of a 3D convex polygon:**

1. // To determine whether a point is on the interior of a convex polygon in 3D, one
2. // might be tempted to first determine whether the point is on the plane, then
3. // determine its interior status. Both of these can be accomplished at once by
4. // computing the sum of the angles between the test point (q below) and every pair of
5. // edge points p[i]->p[i+1]. This sum will only be twopi if both the point is on the
6. // plane of the polygon AND on the interior. The angle sum will tend to 0 the further
7. // away from the polygon point q becomes. The following code snippet returns the angle
8. // sum between the test point q and all the vertex pairs. The angle sum is in radians.
9. #define EPSILON 0.0000001
10. #define MODULUS(p)(sqrt(p.x \* p.x + p.y \* p.y + p.z \* p.z))
11. **const** **double** TWOPI = 6.283185307179586476925287,
12. RTOD = 57.2957795;
13. **double** CalcAngleSum(point3D q, point3D \* p, **int** n) {
14. **double** m1, m2, anglesum = 0, costheta;
15. point3D p1, p2;
16. **for** (**int** i = 0; i < n; i++) {
17. p1.x = p[i].x - q.x;
18. p1.y = p[i].y - q.y;
19. p1.z = p[i].z - q.z;
20. p2.x = p[(i + 1) % n].x - q.x;
21. p2.y = p[(i + 1) % n].y - q.y;
22. p2.z = p[(i + 1) % n].z - q.z;
23. m1 = MODULUS(p1), m2 = MODULUS(p2);
24. **if** (m1 \* m2 <= EPSILON) **return** (TWOPI); // We are on a node, consider this inside
25. **else** costheta = (p1.x \* p2.x + p1.y \* p2.y + p1.z \* p2.z) / (m1 \* m2);
26. anglesum += acos(costheta);
27. }
28. **return** (anglesum);
29. }

**MISC GEOMETRY:**

1. **const** **double** eps = 1e-11, pi = 2 \* acos(0.0);
2. **struct** point { // Creates normal 2D point
3. **double** x, y;
4. point() {}
5. point(**double** xx, **double** yy) {
6. x = xx, y = yy;
7. }
8. };
9. **struct** point3D { // Creates normal 3D point
10. **double** x, y, z;
11. };
12. **struct** line { // Creates a line with equation ax + by + c = 0
13. **double** a, b, c;
14. line() {}
15. line(point p1, point p2) {
16. a = p1.y - p2.y;
17. b = p2.x - p1.x;
18. c = p1.x \* p2.y - p2.x \* p1.y;
19. }
20. };
21. **struct** circle { // Creates a circle with point 'center' as center and r as radius
22. point center;
23. **double** r;
24. circle() {}
25. circle(point P, **double** rr) {
26. center = P;
27. r = rr;
28. }
29. };
30. **struct** segment { // Creates a segment with two end points -> A, B
31. point A, B;
32. segment() {}
33. segment(point P1, point P2) {
34. A = P1, B = P2;
35. }
36. };
37. **inline** **bool** eq(**double** a, **double** b) {
38. **return** fabs(a - b) < eps;
39. } //two numbers are equal

**Distance (Point, Point):**

1. **inline** **double** Distance(point a, point b) {
2. **return** sqrt((a.x - b.x) \* (a.x - b.x) + (a.y - b.y) \* (a.y - b.y));
3. }

**Distance ^ 2 (Point, Point):**

1. **inline** **double** sq\_Distance(point a, point b) {
2. **return** (a.x - b.x) \* (a.x - b.x) + (a.y - b.y) \* (a.y - b.y);
3. }

**Distance (Point, Line):**

1. **inline** **double** Distance(point P, line L) {
2. **return** fabs(L.a \* P.x + L.b \* P.y + L.c) / sqrt(L.a \* L.a + L.b \* L.b);
3. }

**Distance (Point, Segment):**

1. **inline** **double** Distance(point P, segment S) {
2. line L1 = line(S.A, S.B), L2;
3. point P1;
4. L2 = findPerpendicularLine(L1, P);
5. **if** (intersection(L1, L2, P1))
6. **if** (eq(Distance(S.A, P1) + Distance(S.B, P1), Distance(S.A, S.B)))
7. **return** Distance(P, L1);
8. **return** min(Distance(S.A, P), Distance(S.B, P));
9. }

**IS Left Function:**

1. **inline** **double** isleft(point p0, point p1, point p2) {
2. **return** ((p1.x - p0.x) \* (p2.y - p0.y) - (p2.x - p0.x) \* (p1.y - p0.y));
3. }

**Intersection (Line, Line):**

1. **inline** **bool** intersection(line L1, line L2, point & p) {
2. **double** det = L1.a \* L2.b - L1.b \* L2.a;
3. **if** (eq(det, 0)) **return** **false**;
4. p.x = (L1.b \* L2.c - L2.b \* L1.c) / det;
5. p.y = (L1.c \* L2.a - L2.c \* L1.a) / det;
6. **return** **true**;
7. }

**Intersection (Segment, Segment):**

1. **inline** **bool** intersection(segment L1, segment L2, point & p) {
2. **if** (!intersection(line(L1.A, L1.B), line(L2.A, L2.B), p)) {
3. **return** **false**; // can lie on another, just check their equations, and check overlap
4. }
5. **return** (eq(Distance(L1.A, p) + Distance(L1.B, p), Distance(L1.A, L1.B)) &&
6. eq(Distance(L2.A, p) + Distance(L2.B, p), Distance(L2.A, L2.B)));
7. }

**Perpendicular Line of a Given Line Through a Point:**

1. **inline** line findPerpendicularLine(line L, point P) {
2. line res; //line perpendicular to L, and intersects with P
3. res.a = L.b, res.b = -L.a;
4. res.c = -res.a \* P.x - res.b \* P.y;
5. **return** res;
6. }

**Area of a 2 D Polygon:**

1. **double** areaPolygon(point P[], **int** n) {
2. **double** area = 0;
3. **for** (**int** i = 0, j = n - 1; i < n; j = i++) area += P[j].x \* P[i].y - P[j].y \* P[i].x;
4. **return** fabs(area) / 2;
5. }

**Point Inside Polygon:**

1. **bool** insidePoly(point & p, point P[], **int** n) {
2. **bool** inside = **false**;
3. **for** (**int** i = 0, j = n - 1; i < n; j = i++)
4. **if** (((P[i].x < p.x) ^ (P[j].x < p.x)) &&
5. (P[i].y - P[j].y) \* abs(p.x - P[j].x) < (p.y - P[j].y) \* abs(P[i].x - P[j].x))
6. inside = !inside;
7. **return** inside;
8. }

**Intersection - Circle, Line:**

1. **inline** **bool** intersection(circle C, line L, point & p1, point & p2) {
2. **if** (Distance(C.center, L) > C.r + eps) **return** **false**;
3. **double** a, b, c, d, x = C.center.x, y = C.center.y;
4. d = C.r \* C.r - x \* x - y \* y;
5. **if** (eq(L.a, 0)) {
6. p1.y = p2.y = -L.c / L.b;
7. a = 1;
8. b = 2 \* x;
9. c = p1.y \* p1.y - 2 \* p1.y \* y - d;
10. d = b \* b - 4 \* a \* c;
11. d = sqrt(fabs(d));
12. p1.x = (b + d) / (2 \* a);
13. p2.x = (b - d) / (2 \* a);
14. } **else** {
15. a = L.a \* L.a + L.b \* L.b;
16. b = 2 \* (L.a \* L.a \* y - L.b \* L.c - L.a \* L.b \* x);
17. c = L.c \* L.c + 2 \* L.a \* L.c \* x - L.a \* L.a \* d;
18. d = b \* b - 4 \* a \* c;
19. d = sqrt(fabs(d));
20. p1.y = (b + d) / (2 \* a);
21. p2.y = (b - d) / (2 \* a);
22. p1.x = (-L.b \* p1.y - L.c) / L.a;
23. p2.x = (-L.b \* p2.y - L.c) / L.a;
24. }
25. **return** **true**;
26. }

**Find Points that are r1 unit away from A, and r2 unit away from B:**

1. **inline** **bool** findpointAr1Br2(point A, **double** r1, point B, **double** r2, point & p1, point & p2) {
2. line L;
3. circle C;
4. L.a = 2 \* (B.x - A.x);
5. L.b = 2 \* (B.y - A.y);
6. L.c = A.x \* A.x + A.y \* A.y - B.x \* B.x - B.y \* B.y + r2 \* r2 - r1 \* r1;
7. C.center = A;
8. C.r = r1;
9. **return** intersection(C, L, p1, p2);
10. }

**Intersection Area between Two Circles:**

1. **inline** **double** intersectionArea2C(circle C1, circle C2) {
2. C2.center.x = Distance(C1.center, C2.center);
3. C1.center.x = C1.center.y = C2.center.y = 0;
4. **if** (C1.r < C2.center.x - C2.r + eps) **return** 0;
5. **if** (-C1.r + eps > C2.center.x - C2.r) **return** pi \* C1.r \* C1.r;
6. **if** (C1.r + eps > C2.center.x + C2.r) **return** pi \* C2.r \* C2.r;
7. **double** c, CAD, CBD, res;
8. c = C2.center.x;
9. CAD = 2 \* acos((C1.r \* C1.r + c \* c - C2.r \* C2.r) / (2 \* C1.r \* c));
10. CBD = 2 \* acos((C2.r \* C2.r + c \* c - C1.r \* C1.r) / (2 \* C2.r \* c));
11. res = C1.r \* C1.r \* (CAD - sin(CAD)) + C2.r \* C2.r \* (CBD - sin(CBD));
12. **return** .5 \* res;
13. }

**Circle Through Thee Points:**

1. circle CircleThrough3points(point A, point B, point C) {
2. **double** den;
3. circle c;
4. den = 2.0 \* ((B.x - A.x) \* (C.y - A.y) - (B.y - A.y) \* (C.x - A.x));
5. c.center.x = ((C.y - A.y) \* (B.x \* B.x + B.y \* B.y - A.x \* A.x - A.y \* A.y)–
6. (B.y - A.y) \* (C.x \* C.x + C.y \* C.y - A.x \* A.x - A.y \* A.y));
7. c.center.x /= den;
8. c.center.y = ((B.x - A.x) \* (C.x \* C.x + C.y \* C.y - A.x \* A.x - A.y \* A.y)–
9. (C.x - A.x) \* (B.x \* B.x + B.y \* B.y - A.x \* A.x - A.y \* A.y));
10. c.center.y /= den;
11. c.r = Distance(c.center, A);
12. **return** c;
13. }

**Rotating a Point anticlockwise by 'theta' radian w.r.t Origin:**

1. **inline** point rotate2D(**double** theta, point P) {
2. point Q;
3. Q.x = P.x \* cos(theta) - P.y \* sin(theta);
4. Q.y = P.x \* sin(theta) + P.y \* cos(theta);
5. **return** Q;
6. }

**Convex Hull(Graham Scan) O(nlogn):**

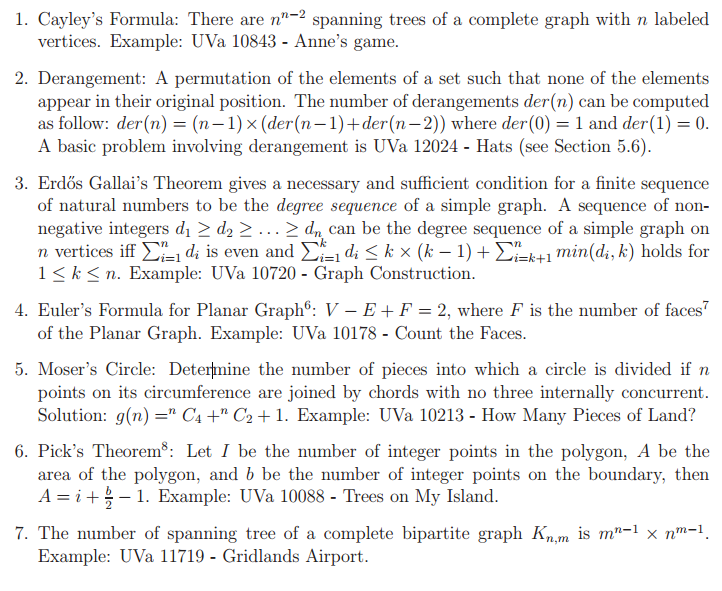
1. // compare Function for qsort in convex hull
2. point Firstpoint;
3. **int** cmp(**const** **void** \* a,
4. **const** **void** \* b) {
5. **double** x, y;
6. point aa, bb;
7. aa = \* (point \* ) a;
8. bb = \* (point \* ) b;
9. x = isleft(Firstpoint, aa, bb);
10. **if** (x > eps) **return** -1;
11. **else** **if** (x < -eps) **return** 1;
12. x = sq\_Distance(Firstpoint, aa);
13. y = sq\_Distance(Firstpoint, bb);
14. **if** (x + eps < y) **return** -1;
15. **return** 1;
16. }
17. // 'P' contains all the points, 'C' contains the convex hull
18. // 'nP' = total points of 'P', 'nC' = total points of 'C'
19. **void** ConvexHull(point P[], point C[], **int** & nP, **int** & nC) {
20. **int** i, j, pos = 0; // Remove duplicate points if necesary
21. **for** (i = 1; i < nP; i++)
22. **if** (P[i].y < P[pos].y || (eq(P[i].y, P[pos].y) && P[i].x > P[pos].x + eps))
23. pos = i;
24. swap(P[pos], P[0]);
25. Firstpoint = P[0];
26. qsort(P + 1, nP - 1, **sizeof**(point), cmp);
27. C[0] = P[0];
28. C[1] = P[1];
29. i = 2, j = 1;
30. **while** (i < nP) {
31. **if** (isleft(C[j - 1], C[j], P[i]) > -eps) C[++j] = P[i++];
32. **else** j--;
33. }
34. nC = j + 1;
35. }

**Angle between Vectors:**

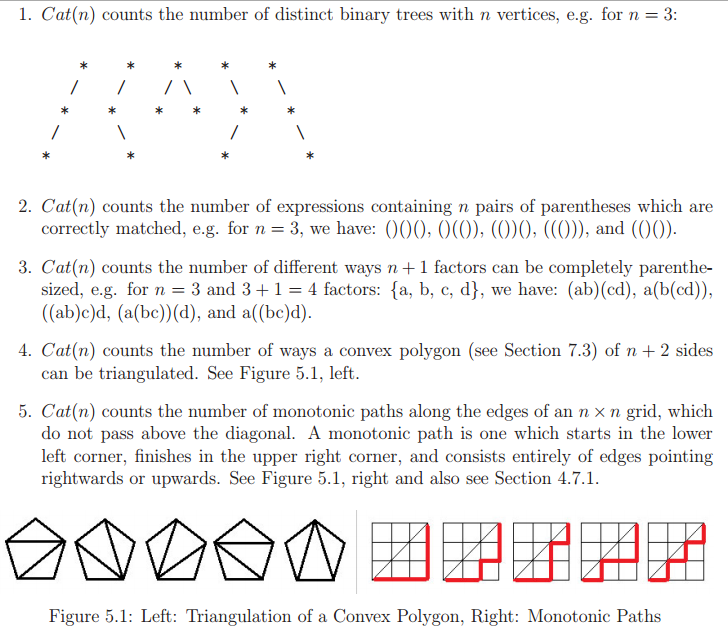
1. **inline** **double** angleBetweenVectors(point O, point A, point B) { // vector OA to OB
2. point t1, t2;
3. t1.x = A.x - O.x;
4. t1.y = A.y - O.y;
5. t2.x = B.x - O.x;
6. t2.y = B.y - O.y;
7. **double** theta = (atan2(t2.y, t2.x) - atan2(t1.y, t1.x));
8. **if** (theta < 0) theta += 2 \* pi;
9. **return** theta;
10. }

**MISC:**

**FORMULA:**



**CATALAN NUMBER PROPERTIES:**



**Nth Permutation:**

1. **const** **int** NX = 30;
2. Long fact[NX], nth\_value;
3. **int** freq[NX];
4. **char** inp[NX];
5. **void** ini() {
6. fact[0] = 1;
7. Long i;
8. **for** (i = 1; i <= 25; i++) fact[i] = (fact[i - 1] \* i);
9. }
10. Long occurence(**int** len) {
11. Long now = fact[len];
12. **int** i;
13. **for** (i = 0; i < 26; i++) now /= fact[freq[i]];
14. **return** now;
15. }
16. **void** solve(**int** sz) {
17. **while** (sz) {
18. Long upto = 0;
19. **bool** found = 0;
20. **int** i;
21. **for** (i = 0; i < 26 && !found; i++) {
22. **if** (freq[i] == 0) **continue**;
23. freq[i]--;
24. Long now = occurence(sz - 1);
25. **if** (now + upto >= nth\_value) {
26. nth\_value -= upto;
27. found = 1;
28. printf("%c", i + 'a');
29. sz--;
30. } **else** {
31. upto += now;
32. freq[i]++;
33. }
34. // printf("i :: %c sz :: %d now :: %lld upto :: %lld nth :: %lld \n",i+'a',sz,now,upto,nth\_value);
36. }
37. **if** (!found) **break**;
38. }
39. puts("");
40. }
41. **int** main() {
42. **int** cs, t;
43. getI(t);
44. ini();
45. **for** (cs = 1; cs <= t; cs++) {
46. scanf("%s %lld", inp, & nth\_value);
47. ms(freq, 0);
48. **int** sz = strlen(inp);
49. // rep( i , sz ) printf(" integer :: %d \n",inp[i]-'a');
50. rep(i, sz) freq[inp[i] - 'a']++;
51. Long need = occurence(sz);
52. printf("Case %d: ", cs);
53. **if** (need < nth\_value) puts("Impossible");
54. **else** solve(sz);
55. }
56. }

**STRTOK:**

1. **int** main ()
2. {
3. **char** str[] ="- This, a sample string.";
4. **char** \* pch;
5. pch = strtok (str," ,.-");          // by which charcters string is tokenized that should be given into "(here)"
6. **while** (pch != NULL)
7. {
8. printf ("%s\n",pch);
9. pch = strtok (NULL, " ,.-");    // by which charcters string is tokenized that should be given into "(here)"
10. }
11. **return** 0;
12. }

**BITWISE OPERATOR:**

1. // Odd - Even checking ==>>
2. **if**(x & 1) --> Odd
3. **else** --> Even
4. // 2^n dara gun or vag ==>>
5. gun --> x<<n
6. vag --> x>>n
7. // 2^n or 2 er power kina ==>>
8. **if**(x & (x-1)) --> 2 er power na
9. **else** --> 2 er power
10. // 2^n dara divisible naki ==>>
11. let, d=2^n
12. **int** x,d=8; // 8=2^3
13. **if**(x & (d-1)) --> x,d dara divisible
14. **else** --> x,d dara divisible na
15. //SWAP ==>>
16. **int** x,y;
17. x = x ^ y;
18. y = x ^ y;
19. x = x ^ y;

**SOME NOTES:**

* Gray Code এ যে বিট এ change হয় তার পর 1000…. থাকে
* Segment tree এর সময় 2 পাশের tree এর marge check করতে হয়।
* BIT সংক্রান্ত কিছু হলে TRIE কিনা দেখতে হবে
* Central Bionomial Coefficient =

(2n, n) = (n!) / ( (n-k)! \* (k!) )

N -> 3 = 20, 4 = 70, 5 = 252, 6 = 924, 7 = 3432