|  |  |  |
| --- | --- | --- |
| Number Theory | | |
| 1. | Sieve | 2 |
| 2. | Number of Divisor | 2 |
| 3. | All Divisor | 2 |
| 4. | Big MOD (Iterative) | 2 |
| 5. | Bitwise Sieve | 2 |
| 6. | Modular Inverse + Extended Euclid | 2 |
| 7. | Factorial | 2 |
| 8. | Leading Digit of Power (n^m) | 3 |
| 9. | Linear Diphantine Eqn+Extd. Euclid (Iterative) | 3 |
| 10. | Simple Hyperbolic Diphantine Eqn. | 3 |
| 11. | Euler Phi / Totient | 3 |
| 12. | GCD(I, N) <= m | 3 |
| 13. | Segmented Sieve | 4 |
| 14. | Sum of Divisor (SOD) | 4 |
| 15. | Sum of Number of Divisor (SNOD) | 4 |
| 16. | Sum of Sum of Divisor (SSOD) | 4 |
| 17. | XOR of 1 to N | 4 |
| 18. | LCM SUM | 4 |
| 19. | Matrix Exponential | 5 |
| 20. | NCR | 5 |
| 21. | Negative Base | 5 |
| 22. | Construct N from SOD | 5 |
| 23. | Derangement | 5 |
| 24. | String Multiply | 5 |
| 25. | Wilson Theorem | 6 |
| 26. | Catalan Number | 6 |
| Graph Theory | | |
| 27. | Direction Array | 6 |
| 28. | Articulation Bridge | 6 |
| 29. | Articulation Point | 6 |
| 30. | Strongly Connected Component (SCC) | 6 |
| 31. | Shrink (mamun4122) | 7 |
| 32. | Lowest Common Ancestor (LCA) | 7 |
| 33. | LCA + MST | 7 |
| 34. | Stable Marriage | 8 |
| 35. | Minimum Vertex Cover | 8 |
| 36. | Kth Best Shortest Path | 9 |
| 37. | Djkstra | 9 |
| 38. | Bellman Ford | 9 |
| 39. | Floyd Warshall | 9 |
| 40. | Tree Diameter | 9 |
| 41. | Farthest node from a Given node | 10 |
| 42. | Topological Sort | 10 |
| 43. | Prufer Code to Tree | 10 |
| 44. | Tree to Prufer Code | 10 |
| 45. | Euler Circuit | 10 |
| 46. | Euler Path | 10 |
| 47. | MST (Kruskal) | 10 |
| 48. | MST (Prims) | 10 |
| 49. | Maximum Bipartite Matching | 11 |
| 50. | 2-SAT | 11 |
| 51. | Erdos & Gallai Theorem | 11 |
| 52. | Bi-connected Component | 12 |
| Dynamic Programming | | |
| 53. | LIS (N log K) | 12 |
| 54. | 2d LIS (N log N) | 12 |
| 55. | LCS (1d) | 12 |
| 56. | Matrix Chain Multiplication (MCM) | 13 |
| 57. | Histogram | 13 |
| 58. | Lonfest Palindrome (Manacher Algo) | 13 |
| 59. | 2d Max Sum (dipta007) | 13 |
| 60. | 2d Max Sum (mamun4122) | 14 |
| 61. | Coin Change (II) | 14 |
| 62. | Coin Change (III) | 14 |
| 63. | Digit DP (dipta007) | 14 |
| 64. | Digit Dp (mamun4122) | 15 |
| 65. | Edit Distance | 15 |
| Game Theory | | |
| 66. | Nim | 15 |
| 67. | Spurge Grundy Theorem | 15 |
| 68. | MinMax | 16 |
| Data Structure | | |
| 69. | Union Find/ Disjoint Set | 16 |
| 70. | Union Find (mamun4122) | 16 |
| 71. | Segment Tree | 16 |
| 72. | Binary Indexed Tree (BIT) | 17 |
| 73. | Range Minimum Query | 17 |
| 74. | Prefix Trie | 17 |
| String Algorithms | | |
| 75. | KMP | 18 |
| 76. | Z Algorithm | 18 |
| 77. | Aho Corasick | 18 |
| 78. | Suffix Array | 19 |
| Geometry | | |
| 79. | Misc Geometry Formula | 21 |
| 80. | Misc Trigonometric Func. & Formula | 21 |
| 81. | Misc Integration Formula | 21 |
| 82. | Misc Differential Formula | 21 |
| 83. | Mirror Point of a point W.r.to line | 21 |
| 84. | Determining if a point lies on the interior of a 3d convex Polygon | 21 |
| 85. | Misc. Geometry | 21 |
| 86. | Distance (Point, Point) | 22 |
| 87. | Distance^2 (Point, Point) | 22 |
| 88. | Distance (Point, Line) | 22 |
| 89. | Distance (Point, Segment) | 22 |
| 90. | IsLeft | 22 |
| 91. | Intersection (Line, Line) | 22 |
| 92. | Intersection (Segment, Segment) | 22 |
| 93. | Perpendicular line of a given line through a Point | 22 |
| 94. | Area of a 2d-Polygon | 22 |
| 95. | Point inside Polygon | 22 |
| 96. | Intersection (Circle, Line) | 22 |
| 97. | Find Points that are r1 unit away from A, r2 unit away from B | 22 |
| 98. | Intersection area between 2 circles | 22 |
| 99. | Circle through the Points | 23 |
| 100. | Rotating a point anti-clockwise by theta w.r.to origin | 23 |
| 101. | Convex hull (Graham Scan) O(N log N) | 23 |
| 102. | Angle between Vectors | 23 |
| Misc | | |
| 103. | Formula | 23 |
| 104. | Catalan Number Properties | 23 |
| 105. | Nth Permutation | 23 |
| 106. | Backtracking (N queen Problem) | 24 |
| 107. | Strtok | 24 |
| 108. | Bitwise Operator | 24 |
| 109. | Permutation & Combination upto 30 | 24 |
| 110. | Josephus Problem | 25 |
| Some Notes | | |
| 111. | mamun4122 | 25 |
| 112. | dipta007 | 25 |
| 113. | howcum | 25 |
| 114. | Palindromic Index | 25 |
| 115. | Ternary Search | 25 |

**NUMBER THEORY:**

**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. }

**GCD(I,N) <= M :**

1. **int** main() {
2. calculatePhi();
3. sieve(M - 1);
4. **int** t;
5. getI(t);
6. rep(cs, t) {
7. ll n, m;
8. **int** q;
9. CLR(cum);
10. getL(n);
11. getI(q);
12. ///getting the prime factor of n
13. divisor(n);
14. ///getting all the factor of n
15. alldivisor(0, 1);
16. sort(ALL(allfactor));
17. ///generating phi value for all the factor
18. repI(i, allfactor.size()) {
19. ans.push\_back(make\_pair(allfactor[i], eulerPhi(n / allfactor[i])));
20. }
21. printf("Case %d\n", cs);
22. **int** sz = ans.size();
23. ///generating cumalitive sum of phi value
24. repI(i, sz) {
25. **if** (i) cum[i] = cum[i - 1] + ans[i].ss;
26. **else** cum[i] = ans[i].ss;
27. }
28. rep(i, q) {
29. getL(m);
30. ///binary searching the answer based on the gcd see khata for explaination
31. **int** low = 0, high = sz - 1, flag = -1;
32. **while** (low <= high) {
33. **int** mid = (low + high) / 2;
34. **if** (ans[mid].ff <= m) {
35. flag = mid;
36. low = mid + 1;
37. } **else** high = mid - 1;
38. }
39. ///flag==-1 means m is negative or zero
40. printf("%lld\n", flag == -1 ? 0 : cum[flag]);
41. }
42. }
43. ///GCD(i,n)<=m
44. ///koita i ase jader n er sathe gcd m er cheya choto or equal
45. ///q = query
46. }

**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. }

**DERANGEMENT:**

1. /\*d(n)=(n−1)∗(d(n−1)+d(n−2))d(n)=(n−1)∗(d(n−1)+d(n−2))
2. বেস কেস: d(1)=0,d(2)=1\*/

**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:**

# 

**GRAPH THEORY:**

**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};

**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. }
26. **public**:
27. vector < pair < **int**, **int** > > adj[BRIDGENODE]; ///Enter target and edge number as pair
28. **void** clear(**int** n) {
29. cc += 3; ///cc is now 0. cc+1 is 1
30. **for** (**int** i = 0; i <= n; i++) {
31. adj[i].clear();
32. }
33. }
34. **void** findBridge(**int** n, **int** start = 0) {
35. **for** (**int** i = start; i <= n; i++) {
36. **if** (col[i] <= cc) {
37. tarjan(i, -1);
38. }
39. }
40. }
41. }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. } art;
48. ///remaining component after removing x
49. **int** compo = 0;
50. **if** ((s == root && child > 1) || (s != root && art)) {
51. **if** (s == root) compo = child;
52. **else** compo = art + 1;
53. } **else** { ///s is not articulation point
54. **if** (p != -1 || adj[s].size()) compo = 1; ///It is not singleton
55. }

**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:**

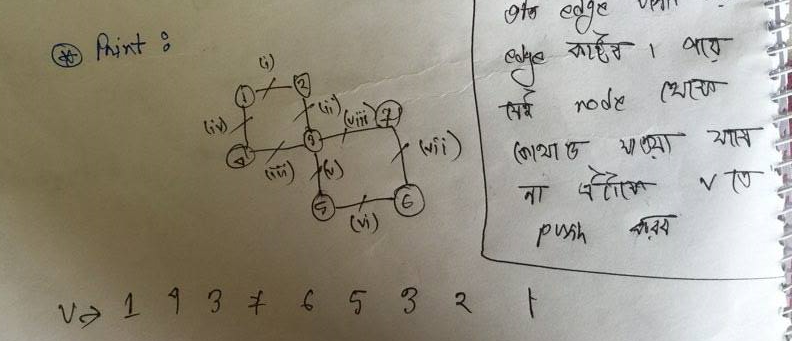
1. vector < **int** > prufer;
2. **void** pruferCodeToTree() {
3. /\*Stores number count of nodes in the prufer code\*/
4. map < **int**, **int** > mp;
5. /\* Set of integer absent in prufer code\*/
6. set < **int** > st;
7. **int** len = prufer.size();
8. **int** n = len + 2;
9. /\*Count frequency of 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. }

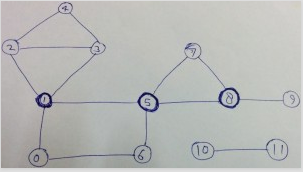
**2-SAT:**

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. \*/

**DYNAMIC PROGRAMMING:**

**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. }

**GAME THEORY:**

**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. }

**DATA STRUCTURE:**

**UNION FIND/DISJOINT SET:**

1. **class** UnionFind { // OOP style
2. **private**:
3. vi p, rank, setSize; // remember: vi is vector<int>
4. **int** numSets;
5. **public**:
6. UnionFind(**int** N) {
7. setSize.assign(N, 1);
8. numSets = N;
9. rank.assign(N, 0);
10. p.assign(N, 0);
11. **for** (**int** i = 0; i < N; i++) p[i] = i;
12. }
13. **int** findSet(**int** i) {
14. **return** (p[i] == i) ? i : (p[i] = findSet(p[i]));
15. }
16. **bool** isSameSet(**int** i, **int** j) {
17. **return** findSet(i) == findSet(j);
18. }
19. **void** unionSet(**int** i, **int** j) {
20. **if** (!isSameSet(i, j)) {
21. numSets--;
22. **int** x = findSet(i), y = findSet(j);
23. // rank is used to keep the tree short
24. **if** (rank[x] > rank[y]) {
25. p[y] = x;
26. setSize[x] += setSize[y];
27. } **else** {
28. p[x] = y;
29. setSize[y] += setSize[x];
30. **if** (rank[x] == rank[y]) rank[y]++;
31. }
32. }
33. }
34. **int** numDisjointSets() {
35. **return** numSets;
36. }
37. **int** sizeOfSet(**int** i) {
38. **return** setSize[findSet(i)];
39. }
40. };

**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. }

**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. }

**STRING ALGORITHMS:**

**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):**

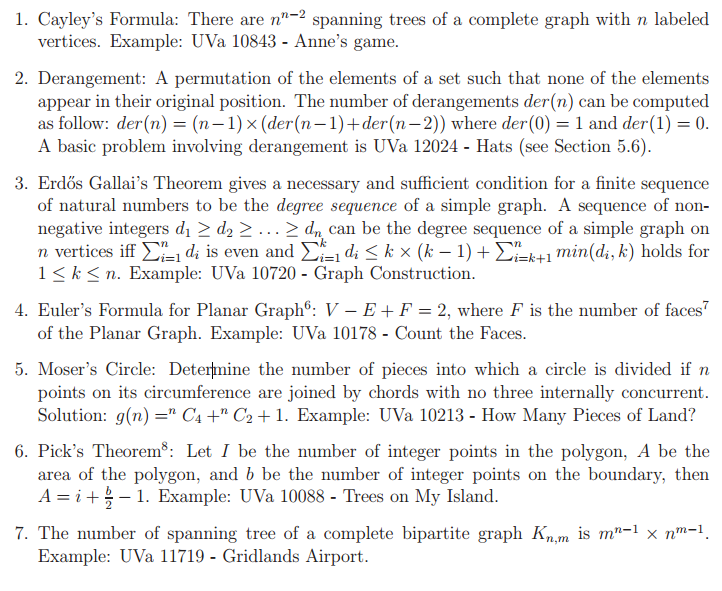
1. // compare Function for qsort in convex hull
2. point Firstpoint;
3. **int** cmp(**const** **void** \* a,
4. **const** **void** \* b) {
5. **double** x, y;
6. point aa, bb;
7. aa = \* (point \* ) a;
8. bb = \* (point \* ) b;
9. x = isleft(Firstpoint, aa, bb);
10. **if** (x > eps) **return** -1;
11. **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:**

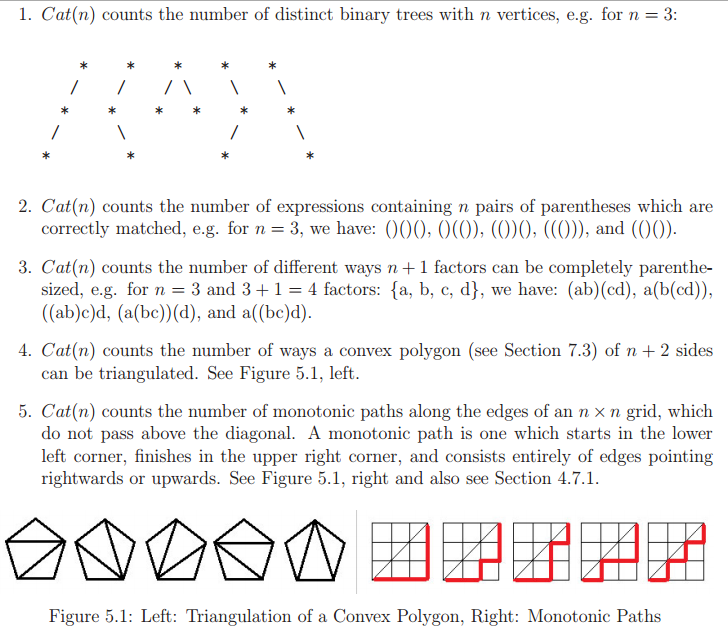
1. **inline** **double** angleBetweenVectors(point O, point A, point B) { // vector OA to OB
2. point t1, t2;
3. t1.x = A.x - O.x;
4. t1.y = A.y - O.y;
5. t2.x = B.x - O.x;
6. t2.y = B.y - O.y;
7. **double** theta = (atan2(t2.y, t2.x) - atan2(t1.y, t1.x));
8. **if** (theta < 0) theta += 2 \* pi;
9. **return**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

**BACKTRACKING (N QUEEN PROBLEM):**

1. **int** board[8][8];
2. **int** soln[100][8];
3. **int** tot = 0;
4. **bool** isSafe(**int** row, **int** col) {
5. **int** i, j;
6. **for** (i = 0; i < 8; i++) {
7. **if** (board[row][i] && i != col)
8. **return** **false**;
9. }
10. **for** (**int** i = 0; i < 8; i++) {
11. **if** (i == row) **continue**;
12. **for** (**int** j = 0; j < 8; j++) {
13. **if** (j == col) **continue**;
14. **if** (abs(row - i) == abs(col - j) && board[i][j])
15. **return** **false**;
16. }
17. }
18. **return** **true**;
19. }
20. **void** solveNQUtil(**int** col) {
21. **if** (col == 8) {
22. **for** (**int** i = 0; i < 8; i++) {
23. **for** (**int** j = 0; j < 8; j++) {
24. **if** (board[i][j])
25. soln[tot][i] = j;
26. }
27. }
28. tot++;
29. **return**;
30. }
31. **for** (**int** i = 0; i < 8; i++) {
32. **if** (isSafe(i, col)) {
33. board[i][col] = 1;
34. solveNQUtil(col + 1);
35. board[i][col] = 0;
36. }
37. }
38. }

**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;

**PERMUTATION & COMBINATION UPTO 30:**

1. ll dp\_permu[31][31], dp\_combi[31][31];
2. **void** permu() {
3. **for** (**int** i = 0; i <= 30; i++) {
4. dp\_permu[i][0] = 1;
5. dp\_permu[i][1] = i;
7. **for** (**int** j = 2; j <= i; j++)
8. dp\_permu[i][j] = dp\_permu[i][j - 1] \* (ll)(i - j + 1);
9. }
10. }
11. **void** combi() {
12. **for** (**int** i = 0; i < 31; i++) {
13. dp\_combi[i][0] = dp\_combi[i][i] = 1;
14. **for** (**int** j = 1; j < i; j++) {
15. dp\_combi[i][j] = dp\_combi[i - 1][j - 1] + dp\_combi[i - 1][j];
16. }
17. }
18. }

JOSEPHUS PROBLEM:

1. **int** joseph(**int** n, **int** k) {
2. **if** (n == 1) **return** 0;
3. **return** ((joseph(n - 1, k) + k) % n);
4. }

**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

**mamun4122:**

1. /\* Given a number N, let d be a divisor of N. Then the number of pairs {a,N}, where 1≤a≤N and gcd(a,N)=d, is ϕ(N/d)
2. \*\* Approximate number of primes under n= (n/ln(n))
3. \*\* Approximate upper limit of number of divisor =2∛N
4. \*\* Diphonite eqn gulai negative number niye hisab korte hbe
5. \*\* Once we find a pair (x,y) using ext\_gcd, we can generate infinite pairs of Bezout coefficients using the formula: (x+(k\*b)/gcd(a,b),y−(k\*a)/gcd(a,b))
6. \*\* Goldbach’s Conjecture: For any integer n (n ≥ 4) there exist two prime numbers p1 and p2 such that p1 + p2 = n.
7. \*\* For a given positive integer n (0 < n < 231) we need to find the number of such m that 1 ≤ m ≤ n, GCD(m, n) ≠ 1 and GCD(m, n) ≠ m
8. n – φ(n) – (a1 + 1) \* (a2 + 1) \* … \* (ak + 1) + 1 \*/

**dipta007:**

1. /\*Area of a triangle :
2. Let K be the triangle's area and let a, b and c, be the lengths of its sides. By  Heron’s Formula , the area of the triangle is K = sqrt( s \* (s-a) \* (s-b) \* (s –c) ).
3. where S is the semiperimeter s = (a+b+c)/2.
4. \*\* length of median to side c = sqrt(2\*(a\*a+b\*b)-c\*c)/2
5. \*\* length of bisector of angle C =   sqrt(ab[(a+b)\*(a+b)-c\*c])/(a+b)
6. \*\* Radius of a In-cicle: The radius of the incircle is r = (2\*k)/P = sqrt((s-a)\*(s-b)\*(s-c)/s). Thus, the area K of a triangle may be found by multiplying the inradius by the semiperimeter: K = rs.
7. \*\* Angle : For a regular convex n-gon, each interior angle has a measure of: (n-2)\*180/n degrees.
8. \*\* Apothem: The apothem of a regular polygon is a line segment from the center to the midpoint of one of its sides.Equivalently, it is the line drawn from the center of the polygon that is perpendicular to one of its sides.
9. \*\* Circumradius: The circumradius from the center of a regular polygon to one of the vertices is related to the side length s or to the apothem a by r = s/(2sin(PI/n)) = a/(cos(PI/n))
10. \*\* Area : The area A of a convex regular n-sided polygon having Side s, circumradius r, apothem a, and perimeter p is given by \*/



**howcum:**

1. /\* Hockeystick pattern: (2c2 \* 3c2 \* ……. \* (n+1)c2) = (n+2)c(2+1)
2. \*\* Equation for the angle of the hour hand: angle\_hour= (½) \* (60H + M)
3. \*\* Equation for the angle of the minute hand: angle\_minute = 6M
4. \*\* Equation for the angle of the both hand : angle = abs(angle\_hour - angle\_minute) \*/

**PALINDROMIC INDEX:**

1. /\* The position of a palindrome within the sequence can be determined almost without calculation: If the palindrome has an even number of digits, prepend a 1 to the front half of the palindrome's digits. If the number of digits is odd, prepend the value of front digit + 1 to the digits from position 2 ... central digit. Examples: 98766789=a(19876), 515=a(61), 8206028=a(9206), 9230329=a(10230). \*/

**TERNARY SEARCH:**

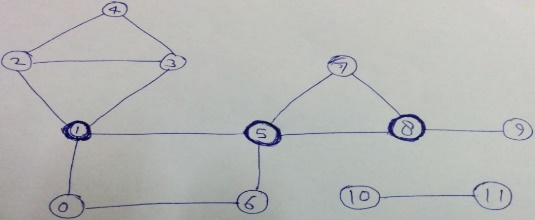
1. **double** ts() {
2. **double** min = 0;
3. **double** max = 1;
4. **int** c = 100; //for higher precision have to increase
5. **double** k, l, f, g;
6. **while** (c--) {
7. f = min + (max - min) / (**double**) 3.0;
8. g = min + (**double**) 2.0 \* ((max - min) / (**double**) 3.0);
9. k = fun(f);
10. l = fun(g);
11. **if** (k < l) {
12. max = g;
13. } **else** {
14. min = f;
15. }
16. }
17. **return** (min + max) / 2.0;
18. }

**TEMPLATE:**

1. #pragma comment(linker, "/stack:640000000")
3. #include <algorithm>
4. #include <bitset>
5. #include <cassert>
6. #include <cctype>
7. #include <climits>
8. #include <cmath>
9. #include <cstdio>
10. #include <cstdlib>
11. #include <cstring>
12. #include <fstream>
13. #include <iostream>
14. #include <iomanip>
15. #include <iterator>
16. #include <list>
17. #include <map>
18. #include <numeric>
19. #include <queue>
20. #include <set>
21. #include <sstream>
22. #include <stack>
23. #include <string>
24. #include <utility>
25. #include <vector>
26. **using** **namespace** std;
28. **const** **double** EPS = 1e-9;
29. **const** **int** INF = 0x7f7f7f7f;
30. **const** **double** PI=acos(-1.0);
32. #define    READ(f)           freopen(f, "r", stdin)
33. #define    WRITE(f)          freopen(f, "w", stdout)
34. #define    MP(x, y)          make\_pair(x, y)
35. #define    PB(x)             push\_back(x)
36. #define    rep(i,n)          for(int i = 1 ; i<=(n) ; i++)
37. #define    repI(i,n)         for(int i = 0 ; i<(n) ; i++)
38. #define    FOR(i,L,R)        for (int i = L; i <= R; i++)
39. #define    ROF(i,L,R)        for (int i = L; i >= R; i--)
40. #define    FOREACH(i,t)      for (typeof(t.begin()) i=t.begin(); i!=t.end(); i++)
41. #define    ALL(p)            p.begin(),p.end()
42. #define    ALLR(p)           p.rbegin(),p.rend()
43. #define    SET(p)            memset(p, -1, sizeof(p))
44. #define    CLR(p)            memset(p, 0, sizeof(p))
45. #define    MEM(p, v)         memset(p, v, sizeof(p))
46. #define    getI(a)           scanf("%d", &a)
47. #define    getII(a,b)        scanf("%d%d", &a, &b)
48. #define    getIII(a,b,c)     scanf("%d%d%d", &a, &b, &c)
49. #define    getL(a)           scanf("%lld",&a)
50. #define    getLL(a,b)        scanf("%lld%lld",&a,&b)
51. #define    getLLL(a,b,c)     scanf("%lld%lld%lld",&a,&b,&c)
52. #define    getC(n)           scanf("%c",&n)
53. #define    getF(n)           scanf("%lf",&n)
54. #define    getS(n)           scanf("%s",n)
55. #define    bitCheck(a,k)     ((bool)(a&(1<<(k))))
56. #define    bitOff(a,k)       (a&(~(1<<(k))))
57. #define    bitOn(a,k)        (a|(1<<(k)))
58. #define    iseq(a,b)         (fabs(a-b)<EPS)
59. #define    vi    vector < int >
60. #define    vii   vector < vector < int > >
61. #define    pii   pair< int, int >
62. #define    ff    first
63. #define    ss    second
64. #define    ll    long long
65. #define    ull   unsigned long long
67. **template**< **class** T > **inline** T \_abs(T n) { **return** ((n) < 0 ? -(n) : (n)); }
68. **template**< **class** T > **inline** T \_max(T a, T b) { **return** (!((a)<(b))?(a):(b)); }
69. **template**< **class** T > **inline** T \_min(T a, T b) { **return** (((a)<(b))?(a):(b)); }
70. **template**< **class** T > **inline** T \_swap(T &a, T &b) { a=a^b;b=a^b;a=a^b;}
71. **template**< **class** T > **inline** T gcd(T a, T b) { **return** (b) == 0 ? (a) : gcd((b), ((a) % (b))); }
72. **template**< **class** T > **inline** T lcm(T a, T b) { **return** ((a) / gcd((a), (b)) \* (b)); }
73. **template** <**typename** T> string NumberToString ( T Number ) { ostringstream ss; ss << Number; **return** ss.str(); }
75. #ifdef CSE13
76. #define debug(args...) {cerr<<"\*: "; dbg,args; cerr<<endl;}
77. #else
78. #define debug(args...)  // Just strip off all debug tokens
79. #endif
81. **struct** debugger{
82. **template**<**typename** T> debugger& operator , (**const** T& v){
83. cerr<<v<<" ";
84. **return** \***this**;
85. }
86. }dbg;
87. ///\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* template ends here \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
88. **int** t,n,m;
90. **int** main() {
91. ///check for 0 or -1 if input not specified
92. #ifdef CSE13
93. //        READ("in.txt");
94. //        WRITE("out.txt");
95. #endif // mamun
96. }
97. //    clock\_t begin, end;
98. //    double time\_spent;
99. //    begin = clock();
100. //
101. //    end = clock();
102. //    time\_spent = (double)(end - begin) / CLOCKS\_PER\_SEC;
103. //    cerr<<"Time spent = "<<time\_spent<<endl;

**Bi-Connected Component:**

1. // 4–2 3–4 3–1 2–3 1–2
2. // 8–9
3. // 8–5 7–8 5–7
4. // 6–0 5–6 1–5 0–1
5. // 10–11



1. /\*\*
2. G[][]: undirected graph
3. Separates bi-connected component by edges.
4. \*\*/
5. vector < **int** > G[MAX];
6. stack < pii > S;
7. **int** dfstime;
8. **int** low[MAX], vis[MAX], used[MAX];
9. **void** dfs(**int** u, **int** par) {
10. **int** v, i, sz = G[u].size();
11. pii e, curr;
12. used[u] = 1;
13. vis[u] = low[u] = ++dfstime;
14. **for** (i = 0; i < sz; i++) {
15. v = G[u][i];
16. **if** (v == par) **continue**;
17. **if** (!used[v]) {
18. S.push(pii(u, v));
19. dfs(v, u);
20. **if** (low[v] >= vis[u]) {
21. // new component
22. curr = pii(u, v);
23. **do** {
24. e = S.top();
25. S.pop();
26. // e is an edge in current bcc
27. } **while** (e != curr);
28. }
29. low[u] = min(low[u], low[v]);
30. } **else** **if** (vis[v] < vis[u]) {
31. S.push(pii(u, v));
32. low[u] = min(low[u], vis[v]);
33. }
34. }
35. }

**HLD with SEG Tree (LOJ 1348):**

1. #define root 0
2. #define mx 30005
3. #define LN 14
4. vector < **int** > adj[mx], costs;
5. **int** parent[mx], sparse[mx][22], depth[mx];
6. **int** baseArray[mx], ptr;
7. **int** chainNo, chainInd[mx], chainHead[mx], posInBase[mx];
8. **int** subsize[mx];
10. ///dfs is used to set parent,depth,subtree size
11. **void** dfs(**int** from, **int** u, **int** dep = 0) {
12. depth[u] = dep;
13. parent[u] = from;
14. subsize[u] = 1;
15. repI(i, adj[u].size()) {
16. **int** v = adj[u][i];
17. **if** (v == from) **continue**;
18. dfs(u, v, dep + 1);
19. subsize[u] += subsize[v];
20. }
21. }
22. ///LCA er code hbe aikhan e
23. /\*
24. \* Actual HL-Decomposition part
25. \* Initially all entries of chainHead[] are set to -1.
26. \* So when ever a new chain is started, chain head is correctly assigned.
27. \* As we add a new node to chain, we will note its position in the baseArray.
28. \* In the first for loop we find the child node which has maximum sub-tree size.
29. \* The following if condition is failed for leaf nodes.
30. \* When the if condition passes, we expand the chain to special child.
31. \* In the second for loop we recursively call the function on all normal nodes.
32. \* chainNo++ ensures that we are creating a new chain for each normal child.
33. \*/
34. **void** HLD(**int** curNode, **int** cost, **int** from) {
35. **if** (chainHead[chainNo] == -1) {
36. chainHead[chainNo] = curNode; /// Assign chain head
37. }
38. chainInd[curNode] = chainNo;
39. posInBase[curNode] = ptr; /// Position of this node in baseArray which we will use in Segtree
40. baseArray[ptr++] = cost;
42. **int** sc = -1, ncost; ///sc = subchild
43. /// Loop to find special child
44. repI(i, adj[curNode].size()) {
45. **int** v = adj[curNode][i];
46. **if** (v == from) **continue**;
47. **if** (sc == -1 || subsize[sc] < subsize[v]) {
48. sc = v;
49. ncost = costs[v];
50. }
51. }
52. **if** (sc != -1) {
53. /// Expand the chain
54. HLD(sc, ncost, curNode);
55. }
56. repI(i, adj[curNode].size()) {
57. **int** v = adj[curNode][i];
58. **if** (v == from || v == sc) **continue**;
59. /// New chains at each normal node
60. chainNo++;
61. HLD(v, costs[v], curNode);
62. }
63. }
65. ///segment tree
66. **int** tree[mx \* 3];
68. **void** init(**int** node, **int** b, **int** e) {
69. **if** (b == e) {
70. tree[node] = baseArray[b];
71. **return**;
72. }
73. tree[node] = tree[Left] + tree[Right];
74. }
75. **int** query(**int** node, **int** b, **int** e, **int** i, **int** j) {
76. same return tree[node];
77. }
78. **void** update(**int** node, **int** b, **int** e, **int** i, **int** newvalue) {
79. //same  tree[node]=newvalue;
80. }
81. /\*
82. \* query\_up:
83. \* It takes two nodes u and v, condition is that v is an ancestor of u
84. \* We query the chain in which u is present till chain head, then move to next chain up
85. \* We do that way till u and v are in the same chain, we query for that part of chain and break
86. \*/
87. **int** query\_up(**int** u, **int** v) {
88. **if** (u == v) **return** query(1, 0, ptr - 1, posInBase[u], posInBase[v]); /// Trivial
89. **int** uchain, vchain = chainInd[v], ans = 0;
90. /// uchain and vchain are chain numbers of u and v
91. **while** (1) {
92. uchain = chainInd[u];
93. **if** (uchain == vchain) {
94. /// Both u and v are in the same chain, so we need to query from u to v
95. /// We break because we came from u up till v, we are done
96. ans += query(1, 0, ptr - 1, posInBase[v], posInBase[u]);
97. **break**;
98. }
99. **int** uchainhed = chainHead[uchain]; /// move u to u's chainHead
100. /// Above is call to segment tree query function. We do from chainHead of u till u. That is the whole chain from
101. /// start till head. We then update the answer
102. ans += query(1, 0, ptr - 1, posInBase[uchainhed], posInBase[u]);
103. u = parent[uchainhed]; ///Then move to its parent, that means we changed chains
104. }
105. **return** ans;
106. }
107. **void** query(**int** u, **int** v) {
108. ///We have a query from u to v, we break it into two queries
109. /// u to LCA(u,v) and LCA(u,v) to v
110. **int** lca = lca\_query(u, v);
111. **int** ans1 = query\_up(u, lca); /// One part of path
112. **int** ans2 = query\_up(v, lca); /// another part of path
113. **int** ans3 = query(1, 0, ptr - 1, posInBase[lca], posInBase[lca]);
114. //    debug(ans1,ans2,ans3)
115. //    debug(lca,posInBase[u],posInBase[v],posInBase[lca])
116. printf("%d\n", ans1 + ans2 - ans3);
117. }
119. ///find pos in segtree and update
120. **void** change(**int** u, **int** val) {
121. **int** indx = posInBase[u];
122. update(1, 0, ptr - 1, indx, val);
123. }
125. **void** clear(**int** n) {
126. ptr = 0;
127. chainNo = 0;
128. costs.clear();
129. repI(i, n + 1) adj[i].clear();
130. SET(chainHead);
132. }
134. **int** main() {
135. getI(t);
136. rep(cs, t) {
137. getI(n);
138. /// Cleaning step, new test case
139. clear(n);
141. repI(i, n) {
142. **int** c;
143. getI(c);
144. costs.PB(c);
145. }
146. rep(i, n - 1) {
147. **int** u, v;
148. getII(u, v);
149. adj[u].push\_back(v);
150. adj[v].push\_back(u);
151. }
153. dfs(-1, root); /// We set up subsize, depth and parent for each node
154. HLD(root, costs[root], -1); /// We decomposed the tree and created baseArray
155. init(1, 0, ptr - 1); /// We use baseArray and construct the needed segment tree
156. lca\_init(n);
158. printf("Case %d:\n", cs);
159. **int** q;
160. getI(q);
161. rep(i, q) {
162. **int** type, u, v;
163. getIII(type, u, v);
164. **if** (type) change(u, v);
165. **else** query(u, v);
166. }
168. }
169. }

**LCA (sgtlaugh):**

1. # define LOG 20
2. #define MAX 200010
3. **namespace** lca {
4. **long** **long** sum[MAX];
5. vector < **int** > ar[MAX], weight[MAX];
6. **int** n, r, parent[MAX], depth[MAX], lg[MAX], dp[MAX][LOG];
8. **void** init(**int** nodes, **int** root) {
9. n = nodes, r = root, lg[0] = lg[1] = 0;
10. **for** (**int** i = 2; i <= n; i++) lg[i] = lg[i >> 1] + 1;
11. **for** (**int** i = 0; i <= n; i++) ar[i].clear(), weight[i].clear();
12. }
14. **void** add\_edge(**int** u, **int** v, **int** w) {
15. ar[u].push\_back(v), weight[u].push\_back(w);
16. ar[v].push\_back(u), weight[v].push\_back(w);
17. }
19. **int** lca(**int** a, **int** b) {
20. **if** (a == b) **return** a;
21. **if** (depth[a] < depth[b]) swap(a, b);
22. **for** (**int** i = lg[depth[a] - depth[b]]; i >= 0; i--) {
23. **if** ((depth[a] - (1 << i)) >= depth[b]) a = dp[a][i];
24. }
25. **if** (a == b) **return** a;
26. **for** (**int** i = lg[depth[a]]; i >= 0; i--) {
27. **if** (dp[a][i] != dp[b][i]) {
28. a = dp[a][i];
29. b = dp[b][i];
30. }
31. }
32. **return** (a == b) ? a : parent[a];
33. }
35. **long** **long** dis(**int** u, **int** v) {
36. **int** l = lca(u, v);
37. **long** **long** res = sum[u] + sum[v] - (sum[l] << 1 LL);
38. **return** res;
39. }
41. **void** dfs(**int** i, **int** p) {
42. **int** j, len = ar[i].size();
43. **for** (j = 0, parent[i] = p; j < len; j++) {
44. **if** (ar[i][j] != p) {
45. sum[ar[i][j]] = sum[i] + weight[i][j];
46. depth[ar[i][j]] = depth[i] + 1;
47. dfs(ar[i][j], i);
48. }
49. }
50. }
52. **void** build() {
53. depth[r] = 0, sum[r] = 0;
54. dfs(r, r);
55. **for** (**int** l = 0; l <= lg[n]; l++) {
56. **for** (**int** i = 0; i < n; i++) {
57. **if** (!l) dp[i][l] = parent[i];
58. **else** dp[i][l] = dp[dp[i][l - 1]][l - 1];
59. }
60. }
61. }
62. }
63. **int** n, P[MAX];
64. **int** main() {
65. **int** i, j, k, x, y, u, v, d1, d2, diameter;
66. scanf("%d", & n);
67. lca::init(n, 0);
68. **for** (i = 1; i < n; i++) {
69. scanf("%d", & x);
70. P[i] = --x;
71. lca::add\_edge(i, x, 1);
72. }
73. lca::build();
75. d1 = 0, d2 = 0;
76. **for** (i = 1; i < n; i++) {
77. u = i, v = P[i];
78. diameter = lca::dis(d1, d2);
79. x = lca::dis(d1, v) + 1;
80. y = lca::dis(d2, v) + 1;
81. **if** (x > diameter) d2 = i;
82. **else** **if** (y > diameter) d1 = i;
83. printf("%lld ", lca::dis(d1, d2));
84. }
85. puts("");
86. **return** 0;
87. }

**max flow (dinik):**

1. /\*\*
2. max flow (dinitz algorithm)
3. works on undirected graph
4. can have loops, multiple edges, cycles
5. \*\*/
7. **int** src, snk, nNode, nEdge;
8. **int** Q[MAXN], fin[MAXN], pro[MAXN], dist[MAXN];
9. **int** flow[MAXE], cap[MAXE], next[MAXE], to[MAXE];
11. **inline** **void** init(**int** \_src, **int** \_snk, **int** \_n) {
12. src = \_src, snk = \_snk, nNode = \_n, nEdge = 0;
13. SET(fin);
14. }
16. **inline** **void** add(**int** u, **int** v, **int** \_cap) {
17. to[nEdge] = v, cap[nEdge] = \_cap, flow[nEdge] = 0;
18. next[nEdge] = fin[u], fin[u] = nEdge++;
19. to[nEdge] = u, cap[nEdge] = \_cap, flow[nEdge] = 0;
20. next[nEdge] = fin[v], fin[v] = nEdge++;
21. }
23. **bool** bfs() {
24. **int** st, en, i, u, v;
25. SET(dist);
26. dist[src] = st = en = 0;
27. Q[en++] = src;
28. **while** (st < en) {
29. u = Q[st++];
30. **for** (i = fin[u]; i >= 0; i = next[i]) {
31. v = to[i];
32. **if** (flow[i] < cap[i] && dist[v] == -1) {
33. dist[v] = dist[u] + 1;
34. Q[en++] = v;
35. }
36. }
37. }
38. **return** dist[snk] != -1;
39. }
41. **int** dfs(**int** u, **int** fl) {
42. **if** (u == snk) **return** fl;
43. **for** (**int** & e = pro[u], v, df; e >= 0; e = next[e]) {
44. v = to[e];
45. **if** (flow[e] < cap[e] && dist[v] == dist[u] + 1) {
46. df = dfs(v, min(cap[e] - flow[e], fl));
47. **if** (df > 0) {
48. flow[e] += df;
49. flow[e ^ 1] -= df;
50. **return** df;
51. }
52. }
53. }
54. **return** 0;
55. }
57. i64 dinitz() {
58. i64 ret = 0;
59. **int** df;
60. **while** (bfs()) {
61. **for** (**int** i = 1; i <= nNode; i++) pro[i] = fin[i];
62. **while** (**true**) {
63. df = dfs(src, INF);
64. **if** (df) ret += (i64) df;
65. **else** **break**;
66. }
67. }
68. **return** ret;
69. }

**min cost max flow (bellman ford):**

1. /\*\*
2. min cost flow (bellman ford)
3. works only on directed graphs
4. handles multiple edges, cycles, loops
5. \*\*/
7. **int** src, snk, nNode, nEdge;
8. **int** fin[MAXN], pre[MAXN], dist[MAXN];
9. **int** cap[MAXE], cost[MAXE], next[MAXE], to[MAXE], from[MAXE];
10. **inline** **void** init(**int** \_src, **int** \_snk, **int** nodes) {
11. SET(fin);
12. nNode = nodes, nEdge = 0;
13. src = \_src, snk = \_snk;
14. }
16. **inline** **void** addEdge(**int** u, **int** v, **int** \_cap, **int** \_cost) {
17. from[nEdge] = u, to[nEdge] = v, cap[nEdge] = \_cap, cost[nEdge] = \_cost;
18. next[nEdge] = fin[u], fin[u] = nEdge++;
19. from[nEdge] = v, to[nEdge] = u, cap[nEdge] = 0, cost[nEdge] = -(\_cost);
20. next[nEdge] = fin[v], fin[v] = nEdge++;
21. }
23. **bool** bellman() {
24. **int** iter, u, v, i;
25. **bool** flag = **true**;
26. MEM(dist, 0x7f);
27. SET(pre);
28. dist[src] = 0;
29. **for**(iter = 1; iter < nNode && flag; iter++) {
30. flag = **false**;
31. **for**(u = 0; u < nNode; u++) {
32. **for**(i = fin[u]; i >= 0; i = next[i]) {
33. v = to[i];
34. **if**(cap[i] && dist[v] > dist[u] + cost[i]) {
35. dist[v] = dist[u] + cost[i];
36. pre[v] = i;
37. flag = **true**;
38. }
39. }
40. }
41. }
42. **return** (dist[snk] < INF);
43. }
45. **int** mcmf(**int** &fcost) {
46. **int** netflow, i, bot, u;
47. netflow = fcost = 0;
48. **while**(bellman()) {
49. bot = INF;
50. **for**(u = pre[snk]; u >= 0; u = pre[from[u]]) bot = min(bot, cap[u]);
51. **for**(u = pre[snk]; u >= 0; u = pre[from[u]]) {
52. cap[u] -= bot;
53. cap[u^1] += bot;
54. fcost += bot \* cost[u];
55. }
56. netflow += bot;
57. }
58. **return** netflow;
59. }

**Maximum Bipartite matching (Hopcroft algo):**

1. /// max must be double of nodes
2. /// in edges u,v  v must be n+v
3. #define MAX 100001
4. #define NIL 0
5. #define INF(1 << 28)
7. vector < **int** > G[MAX];
8. **int** n, m, match[MAX], dist[MAX];
9. // n: number of nodes on left side, nodes are numbered 1 to n
10. // m: number of nodes on right side, nodes are numbered n+1 to n+m
11. // G = NIL[0] ? G1[G[1---n]] ? G2[G[n+1---n+m]]
13. **bool** bfs() {
14. **int** i, u, v, len;
15. queue < **int** > Q;
16. **for** (i = 1; i <= n; i++) {
17. **if** (match[i] == NIL) {
18. dist[i] = 0;
19. Q.push(i);
20. } **else** dist[i] = INF;
21. }
22. dist[NIL] = INF;
23. **while** (!Q.empty()) {
24. u = Q.front();
25. Q.pop();
26. **if** (u != NIL) {
27. len = G[u].size();
28. **for** (i = 0; i < len; i++) {
29. v = G[u][i];
30. **if** (dist[match[v]] == INF) {
31. dist[match[v]] = dist[u] + 1;
32. Q.push(match[v]);
33. }
34. }
35. }
36. }
37. **return** (dist[NIL] != INF);
38. }
40. **bool** dfs(**int** u) {
41. **int** i, v, len;
42. **if** (u != NIL) {
43. len = G[u].size();
44. **for** (i = 0; i < len; i++) {
45. v = G[u][i];
46. **if** (dist[match[v]] == dist[u] + 1) {
47. **if** (dfs(match[v])) {
48. match[v] = u;
49. match[u] = v;
50. **return** **true**;
51. }
52. }
53. }
54. dist[u] = INF;
55. **return** **false**;
56. }
57. **return** **true**;
58. }
60. **int** hopcroft\_karp() {
61. CLR(match);
62. **int** matching = 0, i;
63. // match[] is assumed NIL for all vertex in G
64. **while** (bfs())
65. **for** (i = 1; i <= n; i++)
66. **if** (match[i] == NIL && dfs(i))
67. matching++;
68. **return** matching;
69. }

**maximum vertex cover:**

1. /// same as minimum vertex cover
2. **int** call(**int** u, **int** isGuard) {
3. visit[u] = 1;
4. **if** (edges[u].size() == 0) **return** 1;
5. **if** (dp[u][isGuard] != -1) **return** dp[u][isGuard];
6. **int** sum = 0;
7. **for** (**int** i = 0; i < (**int**) edges[u].size(); i++) {
8. **int** v = edges[u][i];
9. **if** (v != par[u]) {
10. par[v] = u;
11. **if** (isGuard == 1) sum += call(v, 0);
12. **else** sum += max(call(v, 1), call(v, 0));
13. }
14. }
15. **return** dp[u][isGuard] = sum + isGuard;
16. }

**Circle intersection area:**

1. /\*\*
2. This code assumes the circle center and radius to be integer.
3. Change this when necessary.
4. \*\*/
5. **inline** **double** commonArea(**const** Circle & a,
6. **const** Circle & b) {
7. **int** dsq = sqDist(a.c, b.c);
8. **double** d = sqrt((**double**) dsq);
9. **if** (sq(a.r + b.r) <= dsq) **return** 0;
10. **if** (a.r >= b.r && sq(a.r - b.r) >= dsq) **return** pi \* b.r \* b.r;
11. **if** (a.r <= b.r && sq(b.r - a.r) >= dsq) **return** pi \* a.r \* a.r;
12. **double** angleA = 2.0 \* acos((a.r \* a.r + dsq - b.r \* b.r) / (2.0 \* a.r \* d));
13. **double** angleB = 2.0 \* acos((b.r \* b.r + dsq - a.r \* a.r) / (2.0 \* b.r \* d));
14. **return** 0.5 \* (a.r \* a.r \* (angleA - sin(angleA)) + b.r \* b.r \* (angleB - sin(angleB)));
15. }

**Convex Hull (All):**

1. /// main convex hull algorithm
3. /\*\*
4. ConvexHull : Graham's Scan O(n lg n), integer implementation
5. P[]: holds all the points, C[]: holds points on the hull
6. np: number of points in P[], nc: number of points in C[]
7. to handle duplicate, call makeUnique() before calling convexHull()
8. call convexHull() if you have np >= 3
9. to remove co-linear points on hull, call compress() after convexHull()
10. \*\*/
11. point P[MAX], C[MAX], P0;
13. **inline** **int** triArea2(**const** point &a, **const** point &b, **const** point &c) {
14. **return** (a.x\*(b.y-c.y) + b.x\*(c.y-a.y) + c.x\*(a.y-b.y));
15. }
17. **inline** **int** sqDist(**const** point &a, **const** point &b) {
18. **return** ((a.x-b.x)\*(a.x-b.x) + (a.y-b.y)\*(a.y-b.y));
19. }
21. **inline** **bool** comp(**const** point &a, **const** point &b) {
22. **int** d = triArea2(P0, a, b);
23. **if**(d < 0) **return** **false**;
24. **if**(!d && sqDist(P0, a) > sqDist(P0, b)) **return** **false**;
25. **return** **true**;
26. }
28. **inline** **bool** normal(**const** point &a, **const** point &b) {
29. **return** ((a.x==b.x) ? a.y < b.y : a.x < b.x);
30. }
32. **inline** **bool** issame(**const** point &a, **const** point &b) {
33. **return** (a.x == b.x && a.y == b.y);
34. }
36. **inline** **void** makeUnique(**int** &np) {
37. sort(&P[0], &P[np], normal);
38. np = unique(&P[0], &P[np], issame) - P;
39. }
41. **void** convexHull(**int** &np, **int** &nc) {
42. **int** i, j, pos = 0;
43. **for**(i = 1; i < np; i++)
44. **if**(P[i].y<P[pos].y || (P[i].y==P[pos].y && P[i].x<P[pos].x))
45. pos = i;
46. swap(P[0], P[pos]);
47. P0 = P[0];
48. sort(&P[1], &P[np], comp);
49. **for**(i = 0; i < 3; i++) C[i] = P[i];
50. **for**(i = j = 3; i < np; i++) {
51. **while**(triArea2(C[j-2], C[j-1], P[i]) < 0) j--;
52. C[j++] = P[i];
53. }
54. nc = j;
55. }
56. **void** compress(**int** &nc) {
57. **int** i, j, d;
58. C[nc] = C[0];
59. **for**(i=j=1; i < nc; i++) {
60. d = triArea2(C[j-1], C[i], C[i+1]);
61. **if**(d || (!d && issame(C[j-1], C[i+1]))) C[j++] = C[i];
62. }
63. nc = j;
64. }
66. /// point in convex hull
68. /\*\*
69. C[] array of points of convex polygon in ccw order,
70. nc number of points in C, p target points.
71. returns true if p is inside C (including edge) or false otherwise.
72. complexity O(lg n)
73. \*\*/
75. **inline** **bool** inConvexPoly(point \*C, **int** nc, **const** point &p) {
76. **int** st = 1, en = nc - 1, mid;
77. **while**(en - st > 1) {
78. mid = (st + en)>>1;
79. **if**(triArea2(C[0], C[mid], p) < 0) en = mid;
80. **else** st = mid;
81. }
82. **if**(triArea2(C[0], C[st], p) < 0) **return** **false**;
83. **if**(triArea2(C[st], C[en], p) < 0) **return** **false**;
84. **if**(triArea2(C[en], C[0], p) < 0) **return** **false**;
85. **return** **true**;
86. }
88. /// polygon area
90. /\*\*
91. P[] holds the points, must be either in cw or ccw
92. function returns double of the area.
93. \*\*/
95. **inline** **int** dArea(**int** np) {
96. **int** area = 0;
97. **for**(**int** i = 0; i < np; i++) {
98. area += p[i].x\*p[i+1].y - p[i].y\*p[i+1].x;
99. }
100. **return** abs(area);
101. }
103. /// closest pair
105. /\*\*
106. closestPair(Point \*X, Point \*Y, int n);
107. X contains the points sorted by x co-ordinate,
108. Y contains the points sorted by y co-ordinate,
109. One additional item in Point structure is needed, the original index.
110. \*\*/
112. **typedef** **long** **long** i64;
113. **typedef** **struct** { **int** x, y, i; } Point;
115. **int** flag[MAX];
117. **inline** i64 sq(**const** i64 &x) {
118. **return** x\*x;
119. }
121. **inline** i64 sqdist(**const** Point &a, **const** Point &b) {
122. **return** sq(a.x-b.x) + sq(a.y-b.y);
123. }
125. **inline** i64 closestPair(Point \*X, Point \*Y, **int** n) {
126. **if**(n == 1) **return** INF;
127. **if**(n == 2) **return** sqdist(X[0], X[1]);
129. **int** i, j, k, n1, n2, ns, m = n >> 1;
130. Point Xm = X[m-1], \*XL, \*XR, \*YL, \*YR, \*YS;
131. i64 lt, rt, dd, tmp;
133. XL = **new** Point[m], YL = **new** Point[m];
134. XR = **new** Point[m+1], YR = **new** Point[m+1];
135. YS = **new** Point[n];
137. **for**(i = 0; i < m; i++) XL[i] = X[i], flag[X[i].i] = 0;
138. **for**(; i < n; i++) XR[i - m] = X[i], flag[X[i].i] = 1;
139. **for**(i = n2 = n1 = 0; i < n; i++) {
140. **if**(!flag[Y[i].i]) YL[n1++] = Y[i];
141. **else** YR[n2++] = Y[i];
142. }
144. lt = closestPair(XL, YL, n1);
145. rt = closestPair(XR, YR, n2);
146. dd = min(lt, rt);
148. **for**(i = ns = 0; i < n; i++)
149. **if**(sq(Y[i].x - Xm.x) < dd)
150. YS[ns++] = Y[i];
151. **for**(j = 0; j < ns; j++)
152. **for**(k = j + 1; k < ns && sq(YS[k].y - YS[j].y) < dd; k++)
153. dd = min(dd, sqdist(YS[j], YS[k]));
155. **delete**[] XL; **delete**[] XR;
156. **delete**[] YL; **delete**[] YR;
157. **delete**[] YS;
159. **return** dd;
160. }

**Ternary Search:**

1. **double** ts()
2. {
3. **double** min = 0;
4. **double** max = 1;
5. **int** c = 100; //for higher precision have to increase
6. **double** k, l, f, g;
7. **while** (c--)
8. {
9. f = min + (max - min) / (**double**) 3.0;
10. g = min + (**double**) 2.0 \* ((max - min) / (**double**) 3.0);
11. k = fun(f);
12. l = fun(g);
13. **if** (k < l)
14. max = g;
15. **else**
16. min = f;
17. }
18. **return** (min + max) / 2.0;
19. }

**Linear Diophantine Equation(structure):**

1. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
2. Returns char according to kind of solution.
3. run and rise variable to generate more solution.
4. Solution stored in x and y variable.
5. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
7. **struct** linearDiphontine {
8. vlong a, b, c, x, y, run, rise;
9. **char** solution;
11. **char** solve() {
12. **if** (a == 0 && b == 0) {
13. **if** (c == 0) {
14. //Infinite solutions. Anything works
15. **return** solution = 'i';
16. } **else** **return** solution = '0'; //No solution
17. }
18. **if** (a == 0 || b == 0) {
19. //Vertical or Horizontal Line
20. **if** (!a) {
21. **if** (c % b != 0) **return** solution = '0'; /// No Solution
22. run = 1;
23. rise = 0;
24. **return** solution = 'h'; /// ( anything, c / b )
25. } **else** {
26. **if** (c % a != 0) **return** solution = '0'; ///No Solution
27. run = 0;
28. rise = 1;
29. **return** solution = 'v'; /// ( c / a , anything )
30. }
31. }
32. ///Existence of solution
33. vlong g = ext\_gcd(a, b, & x, & y);
34. **if** (c % g != 0) {
35. //No solution
36. **return** solution = '0';
37. }
38. ///Simplification of equation
39. a /= g;
40. b /= g;
41. c /= g;
42. ///Bezout's Identity to Original Equation
43. ext\_gcd(a, b, & x, & y);
44. x \*= c;
45. y \*= c;
46. ///run and rise calculation
47. run = b;
48. rise = -a;
49. **if** (run < 0) {
50. run \*= -1;
51. rise \*= -1;
52. }
53. **return** solution = '1';
54. }
56. ///Move solution near to vertical line x = p
57. **void** moveNearVerticalLine(**int** p) {
58. **if** (run == 0) **return**; /// Both are vertical
59. vlong jump = (p - x) / run;
60. x += jump \* run;
61. y += jump \* rise;
63. **if** (x < p) { ///Keep solution on right of the line
64. x += run;
65. y += rise;
66. }
67. }
69. **void** moveNearHorizontalLine(**int** p) {
70. **if** (rise == 0) **return**; /// Both are horizontal
71. vlong jump = (p - y) / rise;
72. x += jump \* run;
73. y += jump \* rise;
74. **if** (y < p) { ///Keep solution above the line
75. **if** (rise > 0) {
76. x += run;
77. y += rise;
78. } **else** {
79. x -= run;
80. y -= rise;
81. }
82. }
83. }
84. };

**Matrix Expo structure:**

1. **struct** Mat {
2. **int** a[2][2];
3. **void** clear() {
4. memset(a, 0, **sizeof**(a));
5. }
6. **void** one() {
7. a[0][0] = a[1][1] = 1;
8. a[0][1] = a[1][0] = 0;
9. }
10. Mat operator + (**const** Mat & b) **const** {
11. Mat tmp;
12. tmp.clear();
13. **for** (**int** i = 0; i < 2; i++) {
14. **for** (**int** j = 0; j < 2; j++) {
15. tmp.a[i][j] = a[i][j] + b.a[i][j];
16. **if** (tmp.a[i][j] >= MOD) {
17. tmp.a[i][j] -= MOD;
18. }
19. }
20. }
21. **return** tmp;
22. }
23. Mat operator \* (**const** Mat & b) **const** {
24. Mat tmp;
25. tmp.clear();
26. **for** (**int** i = 0; i < 2; i++) {
27. **for** (**int** j = 0; j < 2; j++) {
28. **for** (**int** k = 0; k < 2; k++) {
29. tmp.a[i][k] += (**long** **long**) a[i][j] \* b.a[j][k] % MOD;
30. **if** (tmp.a[i][k] >= MOD) {
31. tmp.a[i][k] -= MOD;
32. }
33. }
34. }
35. }
36. **return** tmp;
37. }
38. Mat pw(**int** x) {
39. Mat ans, num = \* **this**;
40. ans.one();
41. **while** (x > 0) {
42. **if** (x & 1) {
43. ans = ans \* num;
44. }
45. num = num \* num;
46. x >>= 1;
47. }
48. **return** ans;
49. }
50. };

**Ncr (with mod):**

1. #define MOD 1000003
2. **typedef** pair < ll, ll > pll;#
3. define x first# define y second
5. pii extended\_euclid(ll a, ll b) {
6. **if** (b == 0) {
7. **return** pll(1, 0);
8. } **else** {
9. pll d = extended\_euclid(b, a % b);
10. **return** pll(d.y, d.x - d.y \* (a / b));
11. }
12. }
13. ll modular\_Inverse(ll a, ll n) {
14. pll ret = extended\_euclid(a, n);
15. **return** ((ret.x % n) + n) % n;
16. }
18. ll N, K;
19. ll fact[1000000];
21. ll NCR(ll N, ll R) {
22. ll top = fact[N];
23. ll dwn = (fact[R] \* fact[N - R]) % MOD;
24. dwn = modular\_Inverse(dwn, MOD);
25. **return** (top \* dwn) % MOD;
26. }
28. **int** main() {
29. fact[0] = 1;
30. **for** (ll i = 1; i < 1000000; i++) {
31. fact[i] = (fact[i - 1] \* i) % MOD;
32. }
33. }

**NOD of 10^18 solution With MillerRabin:**

1. // multiplication (beware for OVERFLOW): a \* b mod m, a >= 0, b >= 0, m >= 2
2. ll mulMod(ll a, ll b, ll m) {
3. // determine largest bit which is set in b
4. **int** largestBit = 0;
5. **while** ((b >> largestBit) != 0) {
6. largestBit++;
7. }
9. // multiplication
10. ll result = 0;
11. **for** (**int** currentBit = largestBit - 1; currentBit >= 0; currentBit--) {
12. result = (result + result); // result \*= 2;
14. **if** (result >= m) result -= m; // result %= m;
16. **if** ((b >> currentBit) & 1) { // check whether current bit is set in b
17. result += a;
18. **if** (result >= m) result -= m; // result %= m;
19. }
20. }
21. **return** result;
22. }
24. //fast exponentiation: a^x mod m, a >= 0, x >= 0, m >= 2
25. ll exp\_mod(ll a, ll x, ll m) {
26. **if** (x == 0) **return** 1;
27. ll res = exp\_mod(a, x / 2, m);
28. res = mulMod(res, res, m); //(res \* res) % m;
29. **if** (x % 2 == 1) res = mulMod(res, a, m); // (res \* a) % m
30. **return** res;
31. }
33. // returns random number in intervall [a..b], a <= b
34. // overflows in this method lead to valid results!!
35. ll random(ll a, ll b) {
36. ll intervallLength = b - a + 1;
37. **int** neededSteps = 0;
38. ll base = RAND\_MAX + 1 LL;
39. **while** (intervallLength > 0) {
40. intervallLength /= base;
41. neededSteps++;
42. }
43. intervallLength = b - a + 1;
44. ll result = 0;
45. **for** (**int** stepsDone = 0; stepsDone < neededSteps; stepsDone++) {
46. result = (result \* base + rand());
47. }
48. result %= intervallLength;
49. **if** (result < 0) result += intervallLength;
50. **return** result + a;
51. }
52. //checks whether a is a witness that n is not prime, 1< a < n
53. **bool** witness(ll a, ll n) {
54. // check as in Miller Rabin Primality Test described
55. ll u = n - 1;
56. **int** t = 0;
57. **while** (u % 2 == 0) {
58. t++;
59. u /= 2;
60. }
61. ll next = exp\_mod(a, u, n);
62. **if** (next == 1) **return** **false**;
63. ll last;
64. **for** (**int** i = 0; i < t; i++) {
65. last = next;
66. next = mulMod(last, last, n); //(last \* last) % n;
67. **if** (next == 1) {
68. **return** last != n - 1;
69. }
70. }
71. **return** next != 1;
72. }
74. // Miller-Rabin-Test, s = # iterations
75. // probability for an error <= 2^(-s)
76. // [method is only called with n > 10^6
77. // and n is not divisible by primes < 10^6]
78. **bool** millerRabin(ll n, **int** s) {
79. **if** (n <= 1) **return** **false**;
80. **if** (n == 2) **return** **true**;
81. **if** (n % 2 == 0) **return** **false**;
82. **for** (**int** i = 0; i < s; i++) {
83. ll a = random(1, n - 1);
84. **if** (witness(a, n)) **return** **false**;
85. }
86. **return** **true**;
87. }
89. // uses Miller-Rabin
90. **bool** isPrime(ll val) {
91. **return** millerRabin(val, 20);
92. }
94. // precalculate all primes < 10^6 with sieve of Eratosthenes
95. // store primes in array primes - the number of stored primes is P
96. **const** **int** MAXP = 1000000; // = 10^6
97. **int** primes[MAXP / 10], P;
98. **bool** sieve[MAXP];
100. **void** init() {
101. memset(sieve, **false**, **sizeof**(sieve));
102. P = 0;
103. **for** (**int** p = 2; p < MAXP; p++) {
104. **if** (sieve[p]) **continue**;
105. primes[P++] = p;
106. **if** (MAXP / p >= p) {
107. **for** (**int** i = p \* p; i < MAXP; i += p) {
108. sieve[i] = **true**;
109. }
110. }
111. }
112. }
114. // checks whether val is a square with binary search, val >= 1
115. **bool** isSquare(ll val) {
116. ll lo = 1, hi = val;
117. **while** (lo <= hi) {
118. ll mid = lo + (hi - lo) / 2;
119. ll tmp = (val / mid) / mid; // be careful with overflows!!
120. **if** (tmp == 0) hi = mid - 1;
121. **else** **if** (mid \* mid == val) **return** **true**;
122. **else** **if** (mid \* mid < val) lo = mid + 1;
123. }
124. **return** **false**;
125. }
127. // calculates the number of positive divisors of val
128. ll calc(ll val) {
129. ll res = 1;
130. **for** (**int** i = 0; i < P; i++) {
131. **if** (val == 1) **break**;
132. **int** p = primes[i];
133. **if** (val % p == 0) { // checks whether p is a divisor of val
134. **int** num = 0;
135. **while** (val % p == 0) {
136. val /= p;
137. ++num;
138. }
139. // p^num divides initial val but p^(num + 1) does not divide initial val
140. // => p can be taken 0 to num times => num + 1 possibilities!!
141. res \*= num + 1;
142. }
143. }
144. // val has no divisors less or equal than 10^6 and val <= 10^18
145. // => this leads to the following 3 cases
146. // val is either 1
147. // or val = p with p prime
148. // or val = p \* q with p and q prime
149. **if** (val == 1) **return** res; // first case
150. **else** **if** (isPrime(val)) **return** res \* 2; // second case
151. **else** **if** (isSquare(val)) **return** res \* 3; // third case but with p == q
152. **else** **return** res \* 4; // third case with p != q
153. }
155. // reading input and writing output:
156. **int** main(**int** argc, **char** \* argv[]) {
157. **if** (argc == 3) {
158. freopen(argv[1], "r", stdin);
159. freopen(argv[2], "w", stdout);
160. }
161. init();
162. ll next;
163. **while** (cin >> next) {
164. cout << calc(next) << endl;
165. }
166. **return** 0;
167. }

**Finding nth palindrome:**

1. **int** t, n, m;
2. ll pal[25];
3. **void** palindrome() {
4. ll range = 9, cum = 0;
5. **int** i = 1;
6. **while** (cum <= 2000000000) {
7. pal[i] = range;
8. cum += range;
9. i++;
10. **if** (i % 2 == 1) range \*= 10;
11. }
12. }
13. **int** main() {
14. palindrome();
15. ll n;
16. **while** (~getL(n) && n) {
17. **if** (n < 10) {
18. printf("%lld\n", n);
19. **continue**;
20. }
21. **int** dig = 1;
22. **while** (n > pal[dig]) {
23. n -= pal[dig++];
24. }
25. **int** mid = (dig + 1) / 2;
26. stack < **int** > st;
27. **int** gun = 1;
28. rep(i, mid - 1) gun \*= 10;
29. **int** flag = 1;
30. **int** hoise = 1;
31. **while** (hoise <= mid) {
32. hoise++;
33. **int** cnt = flag;
34. flag = 0;
35. **while** (n > gun) {
36. n -= gun;
37. cnt++;
38. }
39. gun /= 10;
40. printf("%d", cnt);
41. st.push(cnt);
42. }
43. **if** (dig % 2 != 0 && !st.empty()) st.pop();
44. **while** (!st.empty()) {
45. printf("%d", st.top());
46. st.pop();
47. }
48. puts("");
49. }
50. }

**Egg Dropping:**

1. **int** t, n, m;
2. **int** dp[55][1005];
3. **int** call(**int** dim, **int** height) {
4. **if** (dim == 1 || height == 0) **return** height;
5. **int** & ret = dp[dim][height];
6. **if** (ret != -1) **return** ret;
7. **int** ans = INF;
8. ///1 theka height sb floor e try korbo ai dim dia
9. rep(i, height) {
10. ///dim vangse ai floor e so 1 theka height i-1 er modde
11. ///dim-1 ta die try korbo
12. **int** tmp = call(dim - 1, i - 1);
13. ///dim vange nai tai i+1 theka baki floor e
14. ///dim ta die try korbo
15. tmp = max(tmp, call(dim, height - i));
16. ans = min(ans, tmp);
17. }
18. **return** ret = 1 + ans;
19. }
20. **void** solveiterative() {
21. CLR(dp);
22. rep(i, m) dp[1][i] = i;
23. rep(i, n) {
24. **if** (i == 1) **continue**;
25. ///if(i>j)then dp[i][j]=dp[i-1][j]
26. rep(j, m) {
27. **int** ans = INF;
28. rep(k, j) {
29. ans = min(ans, max(dp[i - 1][k - 1], dp[i][j - k]));
30. }
31. dp[i][j] = 1 + ans;
32. }
33. }
34. ///        printf("%d %d\n",cs,dp[n][m]);
35. }
36. **int** main() {
37. getI(t);
38. SET(dp);
39. rep(cs, t) {
40. getI(cs);
41. getII(n, m);
42. **int** ans = call(n, m);
43. printf("%d %d\n", cs, ans);
44. //        solveiterative();
46. }
47. }

**1147 - Tug of War (DP):**

1. /\*A tug of war is to be arranged at the local office picnic. For the tug of war, the picnickers must be divided into two teams. Each person must be on one team or the other; the number of people on the two teams must not differ by more than 1; the total weight of the people on each team should be as nearly equal as possible.
2. \*/
3. **int** t, n, m;
4. **int** val[105];
5. ll dp[100005]; //,need[100005];
7. **int** main() {
8. getI(t);
9. rep(cs, t) {
10. getI(n);
11. **int** sum = 0;
12. rep(i, n) getI(val[i]);
13. rep(i, n) sum += val[i];
14. **int** ans = sum;
15. sum /= 2;
16. CLR(dp);
17. ///0 pos e 1 set korsi
18. dp[0] = 1;
19. ///shift kore kore hisab rakhtesi j koivabe number ta banano gese
20. ///limit choto dekhe kora jaitese
21. rep(i, n) {
22. **for** (**int** j = sum; j >= val[i]; j--) {
23. dp[j] |= dp[j - val[i]] << 1 LL;
24. //                dp[j] = dp[j]<<1LL;
25. ///pore shift korle WA dibe bcz tkhn ager gulao shift hoye jai
26. }
27. }
28. **int** tot = n / 2;
29. ///bitcheck er jonno 1 ll nite hbe upore
30. **if** (n % 2) {
31. **while** (bitCheck(dp[sum], tot) == 0 && bitCheck(dp[sum], tot + 1) == 0) sum--;
32. } **else**
33. **while** ((bitCheck(dp[sum], tot) == 0)) sum--;
34. printf("Case %d: %d %d\n", cs, sum, ans - sum);
35. }
36. }

**Some tricks:**

1. /\*  Summation of any series with equal interval like:
2. 2+4+5+6 or 3+6+9+12 is equal
3. Sum=n\*(first number+last number)/2 \*/
4. /\* Upper limit for n\*(n+1)/2 is 1414213563 \*/
5. /\* **Right Angle Triangle** :
6. If the given side is an even number, then find (N^2)/4. The integer before and after this value will make a right angled triangle. Example, if 8 is the given side, then (8^2)/4 = 16. So the other two sides of the right angled triangle will be 15 and (edit)17.
7. Now if the given side is an odd number, then find (N^2)/2. Here also, the integers before and after the found out value will make the right angled triangle. For example, if 3 is the number. Then (3^2)/2 = 4.5 So the other two sides will be 4 and 5. \*/
8. /\* **bitflip**: #define    bitFlip(a,k)      (a^(1<<(k))) \*/
9. /\* **bitset sum with mod value**:
10. const int M = 100;
12. bitset<M> all,tmp;
13. ///m=meaning mod value
14. inline void apply(bitset<M> &b, int sh) {
15. b = ((b << sh) & all) | (b >> (m - sh));
16. }
17. \*/
18. /\* **mod negative**: res=((res%mod)+mod)%mod; \*/
19. /\*  **LCS**: jdi 2ta sequence er lcs ber korte ble jar length n(<10^5) and n distinct
20. ধরে নিলাম আমাদের দুইটি sequence হচ্ছে { ৪ , ২ , ১ , ৩ } এবং অপরটা হচ্ছে { ১ , ৪  , ২ , ৩ } । যদি আমরা প্রথমটাকে আমাদের বেস ধরি তাহলে ,
21. ৪ -> ১
22. ২ -> ২
23. ১ -> ৩
24. ৩ -> ৪
25. যদি এই ভ্যালু গুলা দিয়ে সেকেন্ড লিস্টটা রিপ্লেস করি তাহলে { ১ , ৪ , ২ , ৩ } হয়ে যাবে { ৩ , ১ , ২ , ৪ } যার মধ্যে যদি LIS দেখি { ১ , ২ , ৪ } হচ্ছে আমাদের বৃহত্তর sequence যা { ৪ , ২ , ৩ } আমাদের LCS দুইটা sequence এর মধ্যে  ।\*/
26. /\*  Bitcheck er smy ll hle upore 1LL declare kore aste hbe and \_\_builtin\_popcountll hbe \*/
27. /\*  Dp te coin change type problem hbe loj 1233 or 1147 er concept dekhte hbe….jdi 1 ta number koivabe banana jai ta lage and limit choto hoi tahole amra bit use korte parbo like 1147 \*/

**Java buffer reader:**

1. /\*\* Class for buffered reading int and double values \*/
2. **class** Reader {
3. **static** BufferedReader reader;
4. **static** StringTokenizer tokenizer;
6. /\* call this method to initialize reader for InputStream \*/
7. **static** **void** init(InputStream input) {
8. reader = **new** BufferedReader(
9. **new** InputStreamReader(input));
10. tokenizer = **new** StringTokenizer("");
11. }
13. /\*\* get next word \*/
14. **static** String next() **throws** IOException {
15. **while** (!tokenizer.hasMoreTokens()) {
16. //TODO add check for eof if necessary
17. tokenizer = **new** StringTokenizer(
18. reader.readLine());
19. }
20. **return** tokenizer.nextToken();
21. }
23. **static** **int** nextInt() **throws** IOException {
24. **return** Integer.parseInt(next());
25. }
27. **static** **double** nextDouble() **throws** IOException {
28. **return** Double.parseDouble(next());
29. }
30. }
32. ///in main func
33. Reader.init(System.in);
34. **double** x = Reader.nextDouble();
35. **int** n = Reader.nextInt();

**RMQ With Sparse Table:**

1. **int** parent[100005];
2. **int** sparse[100005][25];
3. **int** height;
4. **void** build\_sparse(**int** N) {
5. MEM(sparse, 63);
6. **for** (**int** i = 0; i < N; i++) sparse[i][0] = parent[i];
7. **for** (**int** j = 1;
8. (1 << j) < N; j++) {
9. height = j;
10. **int** prev = 1 << (j - 1);
11. **for** (**int** i = 0; i < N; i++) {
12. **int** k = i + prev;
13. k = min(k, n - 1);
14. sparse[i][j] = min(sparse[i][j - 1], sparse[k][j - 1]);
15. }
16. }
17. height++;
18. }
19. ///0 index e pathaite hbe
20. **int** rmq(**int** i, **int** j) {
21. **int** len = j - i + 1;
22. **int** col = height;
23. **int** scnd = pow(2, height);
24. **while** (scnd > len) {
25. scnd /= 2;
26. col--;
27. }
28. **int** res = min(sparse[i][col], sparse[j - scnd + 1][col]);
29. **return** res;
30. }

**SQRT Decomposition (MO’s Algo):**

1. /// O( (N+M) \* sqrt N )
2. #define N 311111
3. #define A 1111111
4. #define BLOCK 555 // ~sqrt(N)
6. **int** cnt[A], a[N], ans[N], answer = 0;
8. **struct** node {
9. **int** L, R, i;
10. }
11. q[N];
13. **bool** cmp(node x, node y) {
14. **if** (x.L / BLOCK != y.L / BLOCK) {
15. // different blocks, so sort by block.
16. **return** x.L / BLOCK < y.L / BLOCK;
17. }
18. // same block, so sort by R value
19. **return** x.R < y.R;
20. }
22. **void** add(**int** position) {
23. cnt[a[position]]++;
24. **if** (cnt[a[position]] == 1) {
25. answer++;
26. }
27. }
29. **void** remove(**int** position) {
30. cnt[a[position]]--;
31. **if** (cnt[a[position]] == 0) {
32. answer--;
33. }
34. }
36. **int** main() {
37. **int** n;
38. scanf("%d", & n);
39. **for** (**int** i = 0; i < n; i++)
40. scanf("%d", & a[i]);
42. **int** m;
43. scanf("%d", & m);
44. **for** (**int** i = 0; i < m; i++) {
45. scanf("%d%d", & q[i].L, & q[i].R);
46. q[i].L--;
47. q[i].R--;
48. q[i].i = i;
49. }
51. sort(q, q + m, cmp);
53. **int** currentL = 0, currentR = 0;
54. **for** (**int** i = 0; i < m; i++) {
55. **int** L = q[i].L, R = q[i].R;
56. **while** (currentL < L) {
57. remove(currentL);
58. currentL++;
59. }
60. **while** (currentL > L) {
61. add(currentL - 1);
62. currentL--;
63. }
64. **while** (currentR <= R) {
65. add(currentR);
66. currentR++;
67. }
68. **while** (currentR > R + 1) {
69. remove(currentR - 1);
70. currentR--;
71. }
72. ans[q[i].i] = answer;
73. }
75. **for** (**int** i = 0; i < m; i++)
76. printf("%d\n", ans[i]);
77. }

**2D BIT:**

1. ///\*\* 2d BIT
2. ll bit[1030][1030];
3. **int** arr[1030][1030];
5. **void** update(**int** x, **int** y, **int** v) {
6. **while** (x <= n) {
7. **int** tmp = y;
8. **while** (tmp <= n) {
9. bit[x][tmp] += v;
10. tmp += tmp & (-tmp);
11. }
12. x += x & (-x);
13. }
14. }
16. ll query(**int** x, **int** y) {
17. ll res = 0;
18. **while** (x) {
19. **int** tmp = y;
20. **while** (tmp) {
21. res += bit[x][tmp];
22. tmp -= tmp & (-tmp);
23. }
24. x -= x & (-x);
25. }
26. **return** res;
27. }
28. ///ans+=query(a,b)+query(x-1,y-1);
29. ///ans-=query(a,y-1);
30. ///ans-=query(x-1,b);

**Persistent Segment Tree (Kth smallest number):**

1. /\* Given an array a[1 … N] of different integer numbers, your program must answer a series of questions Q(i, j, k) in the form: “What would be the k-th number in a[i … j] segment, if this segment was sorted? \*/
2. /\*  O( (N+M) \* log N ) \*/
4. #define sz size()
5. #define rep(i, n) for (int i = 0; i < n; i++)
6. #define fd(i, a, b) for (int i = a; i >= b; i--)
7. #define N 111111
8. **struct** node {
9. **int** count;
10. node \* left, \* right;
12. node(**int** count, node \* left, node \* right):
13. count(count), left(left), right(right) {}
15. node \* insert(**int** l, **int** r, **int** w);
16. };
17. node \* null = **new** node(0, NULL, NULL); //see line 135
19. node \* node::insert(**int** l, **int** r, **int** w) {
20. **if** (l <= w && w < r) {
21. // With in the range, we need a new node
22. **if** (l + 1 == r) {
23. **return** **new** node(**this** - > count + 1, null, null);
24. }
25. **int** m = (l + r) >> 1;
26. **return** **new** node(**this** - > count + 1, **this** - > left - > insert(l, m, w), **this** - > right - > insert(m, r, w));
27. }
28. // Out of range, we can use previous tree node.
29. **return** **this**;
30. }
32. **int** query(node \* a, node \* b, **int** l, **int** r, **int** k) {
33. **if** (l + 1 == r) {
34. **return** l;
35. }
36. **int** m = (l + r) >> 1;
37. **int** count = a - > left - > count - b - > left - > count;
38. **if** (count >= k)
39. **return** query(a - > left, b - > left, l, m, k);
41. **return** query(a - > right, b - > right, m, r, k - count);
42. }
44. **int** a[N], RM[N];
45. node \* root[N];
46. **int** main() {
47. **int** n, m;
48. scanf("%d%d", & n, & m);
49. map < **int**, **int** > M;
50. rep(i, n) {
51. scanf("%d", & a[i]);
52. M[a[i]];
53. }
54. **int** maxi = 0;
55. **for** (map < **int**, **int** > ::iterator it = M.begin(); it != M.end(); it++) {
56. M[it - > first] = maxi;
57. RM[maxi] = it - > first;
58. maxi++;
59. }
60. null - > left = null - > right = null;
61. rep(i, n) {
62. // Build a tree for each prefix using segment tree of previous prefix
63. root[i] = (i == 0 ? null : root[i - 1]) - > insert(0, maxi, M[a[i]]);
64. }
65. **while** (m--) {
66. **int** u, v, k;
67. scanf("%d%d%d", & u, & v, & k);
68. u--;
69. v--;
70. **int** ans = query(root[v], (u == 0 ? null : root[u - 1]), 0, maxi, k);
71. printf("%d\n", RM[ans]);
72. }
73. }

**Variation of Seg Tree:**

1. /// prefix suffix
2. #define MAX 100005
3. **int** arr[MAX];
4. **struct** info {
5. **int** pre, suf, sum, tot;
6. }
7. tree[MAX \* 3], demo;
8. info call(info a, info b) {
9. info tmp;
11. tmp.sum = max(a.sum, b.sum);
12. tmp.sum = max(tmp.sum, a.suf + b.pre);
13. tmp.sum = max(tmp.sum, a.tot + b.pre);
14. tmp.sum = max(tmp.sum, a.suf + b.tot);
15. tmp.sum = max(tmp.sum, a.tot + b.tot);
17. tmp.tot = a.tot + b.tot;
19. tmp.pre = a.pre;
20. tmp.suf = b.suf;
22. tmp.pre = max(tmp.pre, a.tot + b.pre);
23. tmp.pre = max(tmp.pre, a.tot + b.tot);
25. tmp.suf = max(tmp.suf, a.suf + b.tot);
26. tmp.suf = max(tmp.suf, a.tot + b.tot);
28. **return** tmp;
30. }
31. **void** init(**int** node, **int** b, **int** e) {
32. **if** (b == e) {
33. tree[node].sum = tree[node].suf = tree[node].pre = arr[b];
34. tree[node].tot = arr[b];
35. **return**;
36. }
37. **int** Left = node \* 2;
38. **int** Right = node \* 2 + 1;
39. **int** mid = (b + e) / 2;
40. init(Left, b, mid);
41. init(Right, mid + 1, e);
42. tree[node] = call(tree[Left], tree[Right]);
43. }
44. info query(**int** node, **int** b, **int** e, **int** i, **int** j) {
45. **if** (b >= i && e <= j) **return** tree[node];
46. **int** Left = node \* 2;
47. **int** Right = node \* 2 + 1;
48. **int** mid = (b + e) / 2;
49. **if** (j <= mid) **return** query(Left, b, mid, i, j);
50. **else** **if** (i > mid) **return** query(Right, mid + 1, e, i, j);
51. info p1 = query(Left, b, mid, i, j);
52. info p2 = query(Right, mid + 1, e, i, j);
53. **return** call(p1, p2);
54. }
55. **int** main() {
56. demo.suf = demo.pre = demo.sum = -INF;
57. demo.tot = 0;
58. **while** (~getI(n)) {
59. rep(i, n) getI(arr[i]);
60. init(1, 1, n);
61. getI(m);
62. rep(i, m) {
63. **int** x, y;
64. getII(x, y);
65. printf("%d\n", query(1, 1, n, x, y).sum);
66. }
67. }
68. }
70. /// vector augmented seg tree
72. #define MAX 200005
73. vector < **int** > arr[MAX];
74. **struct** info {
75. vector < **int** > v;
76. }
77. tree[MAX \* 4];
78. info call(info a, info b) {
79. info tmp;
80. tmp.v.resize(a.v.size() + b.v.size());
81. merge(ALL(a.v), ALL(b.v), tmp.v.begin());
82. **return** tmp;
83. }
84. **void** init(**int** node, **int** b, **int** e) {
85. **if** (b == e) {
86. tree[node].v.resize(arr[b].size());
87. tree[node].v = arr[b];
88. ///sort korsi
89. sort(ALL(tree[node].v));
90. **return**;
91. }
92. tree[node] = call(tree[Left], tree[Right]);
93. }

**Gaussian Elimination (DU Resonance):**

1. **void** gauss( **int** N, **long** **double** mat[NN][NN] ) {
2. **int** i, j, k;
3. **for** (i = 0; i < N; i++) {
4. k = i;
5. **for** (j = i+1; j < N; j++) **if** (fabs(mat[j][i]) > fabs(mat[k][i])) k = j;
6. **if** (k != i) **for** (j = 0; j <= N; j++) swap(mat[k][j], mat[i][j]);
7. **for** (j = i+1; j <= N; j++) mat[i][j] /= mat[i][i];
8. mat[i][i] = 1;
9. **for** (k = 0; k < N; k++) **if**( k != i ) {
10. **long** **double** t = mat[k][i];
11. **if** (t == 0.0L) **continue**;
12. **for** (j = i; j <= N; j++) mat[k][j] -= t \* mat[i][j];
13. mat[k][i] = 0.0L;
14. }
15. }
16. }

**Matrix Expo (Samiul Vaia) :**

1. /\*\*
2. 1. Assign MATMOD
3. 2. Declare MAT win row and col.
4. \*\*/
5. ll MATMOD = 1000000000 + 7;
6. **struct** MAT {
7. ll a[3][3]; ///Maximum dimension of the matrix
8. **int** row, col;
9. MAT() {}
10. MAT(**int** r, **int** c) {
11. row = r;
12. col = c;
13. mem();
14. }
15. **void** assign(**int** r, **int** c) {
16. row = r;
17. col = c;
18. }
20. **void** identity() {
21. mem();
22. **for** (**int** i = 0; i < row; i++) a[i][i] = 1;
23. }
24. **void** mem() {
25. memset(a, 0, **sizeof** a);
26. }
27. **void** print() { ///For debugging purpose
28. printf("\*\*\*\*\*\*\*\*\n");
29. **for** (**int** i = 0; i < row; i++) {
30. **for** (**int** j = 0; j < col; j++) printf("%llu ", a[i][j]);
31. printf("\n");
32. }
33. }
34. MAT operator \* (MAT b) { ///Matrix Multiplication N^3
35. MAT res(row, b.col);
37. **for** (**int** i = 0; i < row; i++) {
38. **for** (**int** j = 0; j < b.col; j++) {
39. **for** (**int** k = 0; k < col; k++)
40. res.a[i][j] = (res.a[i][j] + a[i][k] \* b.a[k][j]) % MATMOD; ///Change here if no mod.
41. }
42. }
43. **return** res;
44. }
45. MAT operator ^ (ll p) { ///Matrix Exponentiation
46. MAT res(row, col);
47. res.identity();
48. MAT x = \* **this**;
49. **while** (p) {
50. **if** (p & 1) {
51. res = res \* x;
52. }
53. x = x \* x;
54. p >>= 1;
55. }
56. **return** res;
57. }
58. };

**Josephus:**

1. **int** josephus(**int** n, **int** k)
2. {
3. **if**(n==1) **return** 1;
4. /\*\* The position returned by josephus(n - 1, k) is adjusted because the
5. recursive call josephus(n - 1, k) considers the original position
6. k%n + 1 as position 1 \*\*/
7. **return** (josephus(n-1, k)+k-1)%n+1;
9. }
10. **int** main() {
11. **int** t;
12. getI(t);
13. FOR(ci,1,t)
14. {
15. **int** n,k;
16. getII(n,k);
17. printf("Case %d: %d\n",ci,josephus(n,k));
18. }
19. }

**Minimum Expression:**

1. /\*
2. In computer science, the lexicographically minimal string rotation or lexicographically least circular substring is the problem of finding the rotation of a string possessing the lowest lexicographical order of all such rotations. For example, the lexicographically minimal rotation of "bbaaccaadd" would be "aaccaaddbb". It is possible for a string to have multiple lexicographically minimal rotations, but for most applications this does not matter as the rotations must be equivalent. Finding the lexicographically minimal rotation is useful as a way of normalizing strings. If the strings represent potentially isomorphic structures such as graphs, normalizing in this way allows for simple equality checking.[1] A common implementation trick when dealing with circular strings is to concatenate the string to itself instead of having to perform modular arithmetic on the string indices. \*/
3. **inline** **int** minimumExpression(**char** \*s) {
4. **int** i, j, k, n, len, p, q;
5. len = n = strlen(s), n <<= 1, i = 0, j = 1, k = 0;
6. **while**(i + k < n && j + k < n) {
7. p = i+k >= len ? s[i+k-len] : s[i+k];
8. q = j+k >= len ? s[j+k-len] : s[j+k];
9. **if**(p == q) k++;
10. **else** **if**(p > q) { i = i+k+1; **if**(i <= j) i = j+1; k = 0; }
11. **else** **if**(p < q) { j = j+k+1; **if**(j <= i) j = i+1; k = 0; }
12. }
13. **return** i < j ? i : j;
14. }

**Persistent Segment Tree:**

Given an array a[1 … N] of different integer numbers, your program must answer a series of questions Q(i, j, k) in the form: “What would be the k-th number in a[i … j] segment, if this segment was sorted?

There are solutions which work in O( (N + M) \* log N \* log N ) or O( ( N + M \* log N ) \* log N \* log N ), those solutions will give AC. I will describe O( (N+M) \* log N ) solution.

Let us assume we have a segment tree for all N \* N ranges. That is for each i, j such that 1 <= i <= j <= N we have a segment tree ready to be used. In this case we can answer the query in O( log N ) which is what we need to do. But building those segment trees will take O(N\*N\*N) time and memory. Like in previous problem let me concentrate on reducing memory which will in turn reduce time complexity.

Trick 1 : In segment tree for range (i, j) each node can be calculated from respective nodes in segment tree for range (1, i-1) and range (1, j). That is for each node in segment tree for range (i, j) : node for (i, j) = node for (1, j) – node for (1, i-1). Awesome. We only need O(N) segment trees now. One segment tree for each of the prefixes. (This trick works only because all input values are distinct, see why it wont work in other case)

Trick 2 : Segment tree for prefix i is almost same as segment tree for prefix i-1, except some O( log N ) nodes that will change. So once again we can reduce the memory to O( N \* log N ).

Great, we are done with it. From O( N\*N\*N ) memory we reduced to O( N \* log N ). Time complexity will be O( N \* log N ). There you go, O( (N+M) \* log N ) solution.

1. #define sz size()
2. #define pb push\_back
3. #define rep(i, n) for (int i = 0; i < n; i++)
4. #define fd(i, a, b) for (int i = a; i >= b; i--)
5. # define N 111111
7. **struct** node {
8. **int** count;
9. node \* left, \* right;
11. node(**int** count, node \* left, node \* right):
12. count(count), left(left), right(right) {}
14. node \* insert(**int** l, **int** r, **int** w);
15. };
17. node \* null = **new** node(0, NULL, NULL); //see line 135
19. node \* node::insert(**int** l, **int** r, **int** w) {
20. **if** (l <= w && w < r) {
21. // With in the range, we need a new node
22. **if** (l + 1 == r) {
23. **return** **new** node(**this** - > count + 1, null, null);
24. }
26. **int** m = (l + r) >> 1;
28. **return** **new** node(**this** - > count + 1, **this** - > left - > insert(l, m, w), **this** - > right - > insert(m, r, w));
29. }
31. // Out of range, we can use previous tree node.
32. **return** **this**;
33. }
35. **int** query(node \* a, node \* b, **int** l, **int** r, **int** k) {
36. **if** (l + 1 == r) {
37. **return** l;
38. }
40. **int** m = (l + r) >> 1;
41. **int** count = a - > left - > count - b - > left - > count;
42. **if** (count >= k)
43. **return** query(a - > left, b - > left, l, m, k);
45. **return** query(a - > right, b - > right, m, r, k - count);
46. }
48. **int** a[N], RM[N];
49. node \* root[N];
50. **int** main() {
51. **int** n, m;
52. scanf("%d%d", & n, & m);
54. map < **int**, **int** > M;
55. rep(i, n) {
56. scanf("%d", & a[i]);
57. M[a[i]];
58. }
60. **int** maxi = 0;
62. **for** (map < **int**, **int** > ::iterator it = M.begin(); it != M.end(); it++) {
63. M[it - > first] = maxi;
64. RM[maxi] = it - > first;
65. maxi++;
66. }
68. null - > left = null - > right = null;
69. rep(i, n) {
70. // Build a tree for each prefix using segment tree of previous prefix
71. root[i] = (i == 0 ? null : root[i - 1]) - > insert(0, maxi, M[a[i]]);
72. }
74. **while** (m--) {
75. **int** u, v, k;
76. scanf("%d%d%d", & u, & v, & k);
77. u--;
78. v--;
80. **int** ans = query(root[v], (u == 0 ? null : root[u - 1]), 0, maxi, k);
81. printf("%d\n", RM[ans]);
82. }
83. }

**Convex Hull: (Samiul vaia)**

1. /\*
2. 1. Assign hull.n
3. 2. Take input in hull.point
4. 3. Call findHull()
5. 4. Convex Hull is ready in hull.convex with hull.m points in it.
6. \*/
8. pll g;
10. vlong triArea(**const** pll & a,
11. **const** pll & b,
12. **const** pll & c) {
13. vlong area = a.ff \* b.ss + b.ff \* c.ss + c.ff \* a.ss;
14. area -= a.ff \* c.ss + b.ff \* a.ss + c.ff \* b.ss;
15. **return** area;
16. }
18. vlong sqDist(**const** pll & a,
19. **const** pll & b) {
20. **return** (SQ(a.ff - b.ff) + SQ(a.ss - b.ss));
21. }
23. **bool** convexCompare(**const** pll & a,
24. **const** pll & b) {
25. vlong area = triArea(g, a, b);
26. **if** (area > 0) **return** **true**;
27. **else** **if** (area == 0 && sqDist(g, a) < sqDist(g, b)) **return** **true**;
28. **else** **return** **false**;
29. }
31. **class** ConvexHull {
32. **void** convexSort() { ///CCW
33. g = point[0];
34. FOR(i, 0, n - 1) {
35. **if** (point[i].ff < g.ff) g = point[i];
36. **else** **if** (point[i].ff == g.ff && point[i].ss < g.ss) g = point[i];
37. }
38. sort(point, point + n, convexCompare);
39. }
41. **public**:
42. **int** n, m; ///N is number of points in polygon and M is number of points in convex hull
44. #define PPP adsf
45. pll point[PPP], convex[PPP];
47. **void** findHull() {
48. convexSort();
50. **if** (n == 1) {
51. convex[0] = convex[1] = point[0];
52. m = 1;
53. **return**;
54. }
55. convex[0] = point[n - 1];
56. convex[1] = point[0];
57. convex[2] = point[1];
58. **int** cur = 3;
59. **for** (**int** i = 2; i < n; i++) {
60. vlong area = triArea(convex[cur - 2], convex[cur - 1], point[i]);
61. **if** (area > 0) {
62. convex[cur] = point[i];
63. cur++;
64. } **else** **if** (area == 0) { ///Take action depending on what is required
65. /\*Left Vertical Line gets omitted. Manually handle it\*/
66. /\*convex[cur] = point[i];
67. cur++;\*/
68. ///If extra point needs to be removed
69. convex[cur - 1] = point[i];
70. } **else** {
71. cur--;
72. i--;
73. }
74. }
75. m = cur - 1;
76. }
77. }hull;

**Aho Cohrasick:**

1. /// Aho Corasick
2. /// !!! Works with lowercase letters !!!
3. /// !!! If RTE, bring struct node outside !!!
5. **class** AhoCorasick {
6. #define SIZE asdf
7. **struct** node {
8. **int** val;
9. **int** child[26];
10. **void** clear() {
11. memset(child, 0, **sizeof** child);
12. }
13. }
14. trie[SIZE];
16. **int** curNode, root, fail[SIZE], par[SIZE];
18. **public**:
19. **void** clear() {
20. root = 0;
21. curNode = 0;
22. trie[root].clear();
23. }
25. **void** addTrie(string s) {
26. **int** len = s.size();
28. **int** cur = root;
29. **for** (**int** i = 0; i < len; i++) {
30. **char** c = s[i] - 'a';
31. **if** (trie[cur].child[c] == 0) {
32. curNode++;
33. trie[curNode].clear();
34. trie[curNode].val = c;
35. trie[cur].child[c] = curNode;
36. }
37. cur = trie[cur].child[c];
38. }
40. ///Mark the node cur as finishing point, if needed
41. }
43. **void** calcFail() {
44. queue < **int** > q;
45. q.push(0);
47. **while** (!q.empty()) {
48. **int** s = q.front();
49. q.pop();
51. ///Push all child to queue
52. **for** (**int** i = 0; i < 26; i++) {
53. **int** t = trie[s].child[i];
54. **if** (t != 0) {
55. q.push(t);
56. par[t] = s;
57. }
58. }
60. ///Calculate failure of current node s
61. **if** (s == 0) { ///Special case
62. fail[s] = 0;
63. par[s] = 0;
64. **continue**;
65. }
67. **int** p = par[s];
68. **int** f = fail[p];
69. **int** v = trie[s].val;
71. **while** (f != 0 && trie[f].child[v] == 0) f = fail[f]; ///Keep on jumping until found
73. fail[s] = trie[f].child[v];
74. **if** (fail[s] == s) fail[s] = 0; ///Can't be fail of itself
75. }
76. }
78. **int** goTo(**int** state, **int** c) {
79. **if** (trie[state].child[c] != 0) {
80. **return** trie[state].child[c];
81. }
83. ///Fall back
84. **int** f = fail[state];
85. **while** (f != 0 && trie[f].child[c] == 0) {
86. f = fail[f];
87. }
89. **return** trie[f].child[c];
90. }
92. }aho;

**KMP (Samiul vai) :**

1. //WARNING: Arrays are 1-based index.
3. #define LENGTH XYZ
4. **char** text[LENGTH], pat[LENGTH];
5. **int** pre[LENGTH];
7. **void** compute() {
8. **int** plen = strlen(pat + 1), k = 0;
9. pre[1] = 0;
10. **for** (**int** i = 2; i <= plen; i++) {
11. k = pre[i - 1];
12. **while** (k && pat[k + 1] != pat[i]) k = pre[k];
13. **if** (pat[k + 1] == pat[i]) k++;
14. pre[i] = k;
15. }
16. }
17. **int** match() {
18. **int** tlen = strlen(text + 1), plen = strlen(pat + 1);
19. **int** q = 0;
20. **int** i, res = 0;
21. **for** (i = 1; i <= tlen; i++) {
22. **while** (q && pat[q + 1] != text[i]) q = pre[q];
24. **if** (pat[q + 1] == text[i]) q++;
25. **if** (q == plen) {
26. res++;
27. q = pre[q];
28. }
29. }
30. **return** res;
31. }
32. /// Application
33. /// Period of a string. N - pre[N] is the length of period iff period length divides N.

**MAX Flow MIN Cut:**

1. **struct** node {
2. **int** x, y, next, cap, cost;
3. };
5. /\*
6. 1. Clear graph
7. 2. Add edge
8. 3. Assign source and sink
9. 4. Pass highestNumberOfNode to maxFlow() or minCostMaxFlow()
10. Edge double number from the given constraint
11. \*/
12. **struct** FLOW {
13. **int** source, sink;
15. **int** head[NODE];
16. **void** clear() {
17. e = 0;
18. CLR(head, -1);
19. }
21. node edge[EDGE];
22. **int** e;
23. **void** addEdge(**int** u, **int** v, **int** cap, **int** cap2, **int** cost) {
24. //cap = forward dir, cap2 = backward dir
25. edge[e].x = u;
26. edge[e].y = v;
27. edge[e].cap = cap;
28. edge[e].cost = cost;
29. edge[e].next = head[u];
30. head[u] = e;
31. e++;
32. edge[e].x = v;
33. edge[e].y = u;
34. edge[e].cap = cap2;
35. edge[e].cost = -cost;
36. edge[e].next = head[v];
37. head[v] = e;
38. e++;
39. }
41. **int** vis[NODE], q[NODE], now[NODE];
42. **bool** bfs() {
43. memset(vis, -1, **sizeof** vis);
44. vis[source] = 0;
45. **int** ini = 0, qend = 0;
46. q[qend++] = source;
48. **while** (ini < qend && vis[sink] == -1) {
49. **int** s = q[ini++];
50. **int** i;
51. **for** (i = head[s]; i != -1; i = edge[i].next) {
52. **int** t = edge[i].y;
53. **if** (vis[t] == -1 && edge[i].cap) {
54. vis[t] = vis[s] + 1;
55. q[qend++] = t;
56. }
57. }
58. }
59. **if** (vis[sink] != -1) **return** **true**;
60. **else** **return** **false**;
61. }
62. **int** dfs(**int** s, **int** f) {
63. **if** (f == 0) **return** 0;
64. **if** (s == sink) **return** f;
65. **for** (**int** & i = now[s]; i != -1; i = edge[i].next) {
66. **int** t = edge[i].y;
67. **if** (vis[s] + 1 != vis[t]) **continue**;
68. **int** pushed = dfs(t, MIN(f, edge[i].cap));
69. **if** (pushed) {
70. edge[i].cap -= pushed;
71. edge[i ^ 1].cap += pushed;
72. **return** pushed;
73. }
74. }
75. **return** 0;
76. }
78. **int** maxFlow(**int** highestNumberOfNode, **int** flow) {
79. **int** res = 0;
80. **while** (1) {
81. **if** (flow == 0) **break**;
82. **if** (bfs() == **false**) **break**;
83. **int** i;
84. **for** (i = 0; i <= highestNumberOfNode; i++) now[i] = head[i];
85. **while** (**int** pushed = dfs(source, flow)) {
86. res += pushed; ///Can overflow depending on Max Flow
87. flow -= pushed;
88. }
89. }
90. **return** res;
91. }
93. **int** inq[NODE], par[NODE], record[NODE];
94. **int** minCostFlow(**int** highestNumberOfNode) {
95. **int** res = 0, i, j, k, fl = 0;
96. **while** (1) {
97. **for** (i = 0; i <= highestNumberOfNode; i++) vis[i] = inf;
98. vis[source] = 0;
99. deque < **int** > dq;
100. dq.pb(source);
101. **while** (!dq.empty()) {
102. **int** s = dq.front();
103. dq.pop\_front();
104. inq[s] = 0;
105. **for** (i = head[s]; i != -1; i = edge[i].next) {
106. **int** t = edge[i].y;
107. **if** (edge[i].cap) {
108. **if** (vis[s] + edge[i].cost < vis[t]) {
109. vis[t] = vis[s] + edge[i].cost;
110. par[t] = s;
111. record[t] = i;
112. **if** (inq[t] == 0) {
113. inq[t] = 1;
114. **if** (dq.empty() == **false** && vis[dq.front()] > vis[t])
115. dq.push\_front(t);
116. **else** dq.pb(t);
117. }
118. }
119. }
120. }
121. }
123. //if ( vis[sink] > 0 ) break; ///Cost Getting minimized, Change Here
124. **if** (vis[sink] == inf) **break**; //Flow getting maximized. Either this, or the one above
126. fl++; ///Total flow
127. res += vis[sink]; ///Cost of Flow
128. **for** (i = sink; i != source; i = par[i]) { //Travel from sink to source
129. **int** t = record[i]; //Record which edge was used to travel to t
130. edge[t].cap--;
131. edge[t ^ 1].cap++;
132. }
133. }
135. **return** res; ///Min cost of max flow
136. }
138. }graph;

**KUHN algo for finding maximum matching:**

1. //Kuhn's Algo for finding Maximum matching
2. **struct** KUHN {
3. **int** left[NODE], right[NODE], vis[2 \* NODE], cc; ///Node double in vis for vertex print
4. vector < **int** > adj[NODE];
6. KUHN(): cc(1) {}
8. **void** clear(**int** n) {
9. FOR(i, 0, n) adj[i].clear();
10. }
12. **bool** tryK(**int** v) {
13. **if** (vis[v] == cc) **return** **false**;
14. vis[v] = cc;
15. **for** (**int** i = 0; i < SZ(adj[v]); i++) {
16. **int** t = adj[v][i];
17. **if** (right[t] == -1) {
18. right[t] = v;
19. left[v] = t;
20. **return** **true**;
21. }
22. }
23. **for** (**int** i = 0; i < SZ(adj[v]); i++) {
24. **int** t = adj[v][i];
25. **if** (tryK(right[t])) {
26. right[t] = v;
27. left[v] = t;
28. **return** **true**;
29. }
30. }
31. **return** **false**;
32. }
34. **int** match(**int** n) {
35. **int** res = 0;
36. **bool** done;
37. CLR(left, -1);
38. CLR(right, -1);
39. **do** {
40. done = **true**;
41. cc++;
42. FOR(i, 0, n) {
43. **if** (left[i] == -1 && tryK(i)) {
44. done = **false**;
45. }
46. }
47. } **while** (!done);
48. FOR(i, 0, n) res += (left[i] != -1);
49. **return** res;
50. }
52. ///Careful. Loop runs from 0 to n-1
53. ///Make sure match() has been run
54. **int** lcover[NODE], rcover[NODE];
55. **void** findVertexCover(**int** n) {
56. queue < **int** > q;
57. cc++;
58. FOR(i, 0, n - 1) {
59. **if** (left[i] == -1) {
60. q.push(i);
61. vis[i] = cc;
62. }
63. };
65. **while** (!q.empty()) {
66. **int** s = q.front();
67. q.pop();
68. FOR(i, 0, SZ(adj[s]) - 1) {
69. **int** t = adj[s][i];
70. **if** (t == left[s]) **continue**;
72. **int** xt = t + n;
73. **if** (vis[xt] == cc) **continue**;
75. vis[xt] = cc;
76. xt = right[t];
78. **if** (xt != -1 && vis[xt] != cc) {
79. vis[xt] = cc;
80. q.push(xt);
81. }
83. };
84. }
86. FOR(i, 0, n - 1) {
87. **if** (vis[i] != cc) lcover[i] = 1;
88. **else** lcover[i] = 0;
89. };
90. FOR(i, 0, n - 1) {
91. **if** (vis[i + n] == cc) rcover[i] = 1;
92. **else** rcover[i] = 0;
93. };
95. }
96. }kuhn;

**BST using Set & Map:**

1. set < **int** > numbers;
2. map < **int**, **int** > left;
3. map < **int**, **int** > right;
5. **int** n, v; ///number of nodes =n, root = v
6. cin >> n >> v;
7. numbers.insert(v);
8. **for** (**int** i = 0; i < n - 1; i++) {
9. cin >> v;
10. auto it = numbers.upper\_bound(v);
11. **int** result;
12. **if** (it != numbers.end() && left.count( \* it) == 0) {
13. left[ \* it] = v;
14. result = \* it;
15. } **else** {
16. it--;
17. right[ \* it] = v;
18. result = \* it;
19. }
20. numbers.insert(v);
21. cout << result;
22. }

**Trie (Static Array) :**

1. /\*
2. Trie implementation using array, faster and takes less memory.
3. Each node can contain arbitrary data as needed for solving the problem.
4. The ALPHABET, MAX and scale() may need tuning as necessary.
5. \*/
7. **const** **int** ALPHABET = 26;
8. **const** **int** MAX = 100000;
10. // for mapping items form 0 to ALPHABET-1
11. #define scale(x)(x - 'a')
13. **struct** TrieTree {
14. **int** n, root;
15. **int** next[MAX][ALPHABET];
16. **char** data[MAX]; // there can be more data fields
18. **void** init() {
19. root = 0, n = 1;
20. data[root] = 0;
21. memset(next[root], -1, **sizeof**(next[root]));
22. }
24. **void** insert(**char** \* s) {
25. **int** curr = root, i, k;
26. **for** (i = 0; s[i]; i++) {
27. k = scale(s[i]);
28. **if** (next[curr][k] == -1) {
29. next[curr][k] = n;
30. data[n] = s[i]; // optional
31. memset(next[n], -1, **sizeof**(next[n]));
32. n++;
33. }
34. curr = next[curr][k];
35. }
36. data[curr] = 0; // sentinel, optional
37. }
39. **bool** find(**char** \* s) {
40. **int** curr = root, i, k;
41. **for** (i = 0; s[i]; i++) {
42. k = scale(s[i]);
43. **if** (next[curr][k] == -1) **return** **false**;
44. curr = next[curr][k];
45. }
46. **return** (data[curr] == 0);
47. }
49. }trieTree;

**Shank Baby Step Giant Step:**

1. /\*
2. Shanks baby step giant step - discrete logarithm algorithm
3. for the equation: b = a^x % p where a, b, p known, finds x
4. works only when p is an odd prime
5. \*/
7. **int** shank(**int** a, **int** b, **int** p) {
8. **int** i, j, m;
9. **long** **long** c, aj, ami;
10. map < **long** **long**, **int** > M;
11. map < **long** **long**, **int** > ::iterator it;
12. m = (**int**) ceil(sqrt((**double**)(p)));
13. M.insert(make\_pair(1, 0));
14. **for** (j = 1, aj = 1; j < m; j++) {
15. aj = (aj \* a) % p;
16. M.insert(make\_pair(aj, j));
17. }
18. ami = modexp(modinv(a, p), m, p);
19. **for** (c = b, i = 0; i < m; i++) {
20. it = M.find(c);
21. **if** (it != M.end()) **return** i \* m + it - > second;
22. c = (c \* ami) % p;
23. }
24. **return** 0;
25. }

**Heavy Light Decomposition (Zobayer vai):**

1. /\*
2. Rough code for heavy light decomposition. Input graph must be a tree.
3. Also includes the LCA method.
4. What to do with the chain is problem dependent.
5. \*/
7. vector < **int** > G[MAX];
8. **int** cost[MAX], lvl[MAX], parent[MAX];
9. **int** head[MAX], cnext[MAX], chainid[MAX], chainpos[MAX];
10. **int** nchain, temp[MAX];
12. **int** dfs(**int** u, **int** p, **int** d) {
13. **int** i, v, sz = G[u].size(), tmp, mx, id, tot, hd, k;
14. lvl[u] = d, mx = 0, id = u, tot = 1;
15. **for** (i = 0; i < sz; i++) {
16. v = G[u][i];
17. **if** (v != p) {
18. parent[v] = u;
19. tmp = dfs(v, u, d + 1);
20. tot += tmp;
21. **if** (tmp > mx) {
22. mx = tmp;
23. id = v;
24. }
25. }
26. }
27. **if** (tot == 1) cnext[u] = -1;
28. **else** cnext[u] = id;
29. **for** (i = 0; i < sz; i++) {
30. v = G[u][i];
31. **if** (v != p && v != id) {
32. **for** (hd = v, k = 0; v != -1; v = cnext[v], k++) {
33. head[v] = hd;
34. temp[k] = cost[v];
35. chainpos[v] = k;
36. chainid[v] = nchain;
37. }
38. // buff is the current chain of size k
39. nchain++;
40. }
41. }
42. **return** tot;
43. }
45. **void** hld(**int** v) {
46. **int** hd, k;
47. nchain = 0;
48. lvl[0] = -1;
49. dfs(v, 0, 0);
50. **for** (hd = v, k = 0; v != -1; v = cnext[v], k++) {
51. head[v] = hd;
52. temp[k] = cost[v];
53. chainpos[v] = k;
54. chainid[v] = nchain;
55. }
56. // buff is the current chain of size k
57. nchain++;
58. }
60. **int** lca(**int** a, **int** b) {
61. **while** (chainid[a] != chainid[b]) {
62. **if** (lvl[head[a]] < lvl[head[b]]) b = parent[head[b]];
63. **else** a = parent[head[a]];
64. }
65. **return** (lvl[a] < lvl[b]) ? a : b;
66. }

**2d BIT:**

1. /\*
2. update and query function for 2D bit.
3. MAX is the maximum possible value.
4. bit[][] holds the 2D binary indexed tree
5. \*/
7. **void** update(**int** x, **int** y, **int** v) {
8. **int** y1;
9. **while** (x <= MAX) {
10. y1 = y;
11. **while** (y1 <= MAX) {
12. bit[x][y1] += v;
13. y1 += (y1 & -y1);
14. }
15. x += (x & -x);
16. }
17. }
19. **int** readsum(**int** x, **int** y) {
20. **int** v = 0, y1;
21. **while** (x > 0) {
22. y1 = y;
23. **while** (y1 > 0) {
24. v += bit[x][y1];
25. y1 -= (y1 & -y1);
26. }
27. x -= (x & -x);
28. }
29. **return** v;
30. }

**Double Hashing:**

1. /\*
2. M > N and should be close, better both be primes.
3. M should be as much large as possible, not exceeding array size.
4. HKEY is the Hash function, change it if necessary.
5. \*/
7. #define NIL - 1
8. #define M 1021
9. #define N 1019
10. #define HKEY(x, i)((x) % M + (i) \* (1 + (x) % N)) % M
12. **int** a[M + 1];
14. **inline** **int** hash(**int** key) {
15. **int** i = 0, j;
16. **do** {
17. j = **HKEY**(key, i);
18. **if** (a[j] == NIL) {
19. a[j] = key;
20. **return** j;
21. }
22. i++;
23. } **while** (i < M);
24. **return** -1;
25. }
27. **inline** **int** find(**int** key) {
28. **int** i = 0, j;
29. **do** {
30. j = **HKEY**(key, i);
31. **if** (a[j] == key) **return** j;
32. i++;
33. } **while** (a[j] != NIL && i < M);
34. **return** -1;
35. }

**String hasing using MAP:**

1. /\*
2. hash() takes the string and next free index as parameter.
3. if string is found in map, it returns its index.
4. else, it inserts in current free index and updates the free index.
5. \*/
7. **inline** **int** hash(**char** \* s, **int** & n) {
8. **int** ret;
9. it = M.find(s);
10. **if** (it == M.end()) M.insert(pair < string, **int** > (s, ret = n++));
11. **else** ret = it - > second;
12. **return** ret;
13. }

**Barn-string string hashing:**

1. /\*
2. u64 stands for unsigned long long type. Must use unsigned type.
3. hash = hash \* 33 ^ c, reliable hash
4. \*/
5. u64 hash(**const** **char** \* s) {
6. u64 hash = 0, c;
7. **while** ((c = \* s++)) hash = ((hash << 5) + hash) ^ c;
8. **return** hash;
9. }

**Colinear Point:**

1. **struct** Point {
2. **int** x, y;
3. Point() {
4. x = y = 0;
5. }
6. Point(**int** X, **int** Y): x(X), y(Y) {}
7. **bool** collinear(**const** Point & b,
8. **const** Point & c) **const** {
9. //return (b.y - y) \* (c.x - x) == (c.y - y) \* (b.x - x);
10. **return** ((b.x - x) \* (c.y - y) == (b.y - y) \* (c.x - x));
11. }
12. };

**Tarzan’s Offline LCA:**

1. /\*
2. Tarjan's offline LCA algorithm. For each pair of node's in P {u, v, qid},
3. it finds the LCA of the nodes in the rooted tree G (no edge to back to the parent.
4. The array ans holds the result for queries in orders defined by qid.
5. \*/
7. **void** lca(**int** u) {
8. **int** v, i, sz;
9. make\_set(u);
10. ancestor[find\_set(u)] = u;
11. sz = G[u].size();
12. **for** (i = 0; i < sz; i++) {
13. v = G[u][i];
14. lca(v);
15. union\_set(u, v);
16. ancestor[find\_set(u)] = u;
17. }
18. color[u] = 1;
19. sz = P[u].size();
20. **for** (i = 0; i < sz; i++) {
21. v = P[u][i].first;
22. **if** (color[v]) ans[P[u][i].second] = ancestor[find\_set(v)];
23. }
24. }

**Maximum Rectangle when binary 0 & 1 in n^2:**

1. // Rows and columns in input matrix
2. #define R 4
3. #define C 4
5. // Finds the maximum area under the histogram represented
6. // by histogram.  See below article for details.
7. // http://www.geeksforgeeks.org/largest-rectangle-under-histogram/
8. **int** maxHist(**int** row[]) {
9. // Create an empty stack. The stack holds indexes of
10. // hist[] array/ The bars stored in stack are always
11. // in increasing order of their heights.
12. stack < **int** > result;
14. **int** top\_val; // Top of stack
16. **int** max\_area = 0; // Initialize max area in current
17. // row (or histogram)
19. **int** area = 0; // Initialize area with current top
21. // Run through all bars of given histogram (or row)
22. **int** i = 0;
23. **while** (i < C) {
24. // If this bar is higher than the bar on top stack,
25. // push it to stack
26. **if** (result.empty() || row[result.top()] <= row[i])
27. result.push(i++);
29. **else** {
30. // If this bar is lower than top of stack, then
31. // calculate area of rectangle with stack top as
32. // the smallest (or minimum height) bar. 'i' is
33. // 'right index' for the top and element before
34. // top in stack is 'left index'
35. top\_val = row[result.top()];
36. result.pop();
37. area = top\_val \* i;
39. **if** (!result.empty())
40. area = top\_val \* (i - result.top() - 1);
41. max\_area = max(area, max\_area);
42. }
43. }
45. // Now pop the remaining bars from stack and calculate area
46. // with every popped bar as the smallest bar
47. **while** (!result.empty()) {
48. top\_val = row[result.top()];
49. result.pop();
50. area = top\_val \* i;
51. **if** (!result.empty())
52. area = top\_val \* (i - result.top() - 1);
54. max\_area = max(area, max\_area);
55. }
56. **return** max\_area;
57. }
59. // Returns area of the largest rectangle with all 1s in A[][]
60. **int** maxRectangle(**int** A[][C]) {
61. // Calculate area for first row and initialize it as
62. // result
63. **int** result = maxHist(A[0]);
65. // iterate over row to find maximum rectangular area
66. // considering each row as histogram
67. **for** (**int** i = 1; i < R; i++) {
69. **for** (**int** j = 0; j < C; j++)
71. // if A[i][j] is 1 then add A[i -1][j]
72. **if** (A[i][j]) A[i][j] += A[i - 1][j];
74. // Update result if area with current row (as last row)
75. // of rectangle) is more
76. result = max(result, maxHist(A[i]));
77. }
79. **return** result;
80. }

**Count Internal points of triangle:**

1. // Finds the no. of Integral points between
2. // two given points.
3. **int** getBoundaryCount(Point p,Point q)
4. {
5. // Check if line parallel to axes
6. **if** (p.x==q.x)
7. **return** abs(p.y - q.y) - 1;
8. **if** (p.y == q.y)
9. **return** abs(p.x-q.x) - 1;
11. **return** gcd(abs(p.x-q.x),abs(p.y-q.y))-1;
12. }
14. // Returns count of points inside the triangle
15. **int** getInternalCount(Point p, Point q, Point r)
16. {
17. // 3 extra integer points for the vertices
18. **int** BoundaryPoints = getBoundaryCount(p,q)+
19. getBoundaryCount(p,r)+
20. getBoundaryCount(q,r)+3;
22. // Calculate 2\*A for the triangle
23. **int** doubleArea = abs(p.x\*(q.y - r.y) + q.x\*(r.y - p.y)  +
24. r.x\*(p.y - q.y));
26. // Use Pick's theorem to calculate the no. of Interior points
27. **return** (doubleArea - BoundaryPoints + 2)/2;
28. }

**Lucas Theorem: nCR % p**

1. // Returns nCr % p.  In this Lucas Theorem based program,
2. // this function is only called for n < p and r < p.
3. **int** nCrModpDP(**int** n, **int** r, **int** p) {
4. // The array C is going to store last row of
5. // pascal triangle at the end. And last entry
6. // of last row is nCr
7. **int** C[r + 1];
8. memset(C, 0, **sizeof**(C));
10. C[0] = 1; // Top row of Pascal Triangle
12. // One by constructs remaining rows of Pascal
13. // Triangle from top to bottom
14. **for** (**int** i = 1; i <= n; i++) {
15. // Fill entries of current row using previous
16. // row values
17. **for** (**int** j = min(i, r); j > 0; j--)
19. // nCj = (n-1)Cj + (n-1)C(j-1);
20. C[j] = (C[j] + C[j - 1]) % p;
21. }
22. **return** C[r];
23. }
25. // Lucas Theorem based function that returns nCr % p
26. // This function works like decimal to binary conversion
27. // recursive function.  First we compute last digits of
28. // n and r in base p, then recur for remaining digits
29. **int** nCrModpLucas(**int** n, **int** r, **int** p) {
30. // Base case
31. **if** (r == 0)
32. **return** 1;
34. // Compute last digits of n and r in base p
35. **int** ni = n % p, ri = r % p;
37. // Compute result for last digits computed above, and
38. // for remaining digits.  Multiply the two results and
39. // compute the result of multiplication in modulo p.
40. **return** (nCrModpLucas(n / p, r / p, p) \* // Last digits of n and r
41. nCrModpDP(ni, ri, p)) % p; // Remaining digits
42. }

**BIT (MAX & MIN):**

1. #define LSOne(S)(S & (-S))
3. **class** FenwickTreeMAX {
4. **private**:
5. vi ft;
6. **int** N;
7. **public**:
8. FenwickTreeMAX() {}
9. // initialization: n + 1 zeroes, ignore index 0
10. FenwickTreeMAX(**int** n) {
11. ft.assign(1 << 18 + 1, -INF);
12. N = 1 << 17;
13. }
15. **int** rsq(**int** b) { // returns RSQ(1, b)
16. **int** sum = 0;
17. **for** (; b; b -= LSOne(b)) sum += ft[b];
18. **return** sum;
19. }
21. **int** rsq(**int** a, **int** b) { // returns RSQ(a, b)
22. **return** rsq(b) - (a == 1 ? 0 : rsq(a - 1));
23. }
25. // adjusts value of the k-th element by v (v can be +ve/inc or -ve/dec)
26. **void** adjust(**int** k, **int** v) { // note: n = ft.size() - 1
27. **for** (; k < (**int**) ft.size(); k += LSOne(k)) ft[k] = max(ft[k], v);
28. }
30. **void** SetMax(**int** pos, **int** x) {
31. **for** (**int** i = pos + N; i; i >>= 1)
32. ft[i] = max(ft[i], x);
33. }
34. **int** GetMax(**int** L, **int** R) **const** // [L, R) i.e. L <= i < R
35. {
36. **int** res = 0;
37. **for** (L += N, R += N; L < R; L >>= 1, R >>= 1) {
38. **if** (L & 1) {
39. res = max(res, ft[L]);
40. L++;
41. }
42. **if** (R & 1) {
43. R--;
44. res = max(res, ft[R]);
45. }
46. }
47. **return** res;
48. }
50. };
52. **class** FenwickTreeMIN {
53. **private**:
54. vi ft;
55. **int** N;
56. **public**:
57. FenwickTreeMIN() {}
58. // initialization: n + 1 zeroes, ignore index 0
59. FenwickTreeMIN(**int** n) {
60. ft.assign(1 << 18 + 1, INF);
61. N = 1 << 17;
62. }
64. **int** rsq(**int** b) { // returns RSQ(1, b)
65. **int** sum = 0;
66. **for** (; b; b -= LSOne(b)) sum += ft[b];
67. **return** sum;
68. }
70. **int** rsq(**int** a, **int** b) { // returns RSQ(a, b)
71. **return** rsq(b) - (a == 1 ? 0 : rsq(a - 1));
72. }
74. // adjusts value of the k-th element by v (v can be +ve/inc or -ve/dec)
75. **void** adjust(**int** k, **int** v) { // note: n = ft.size() - 1
76. **for** (; k < (**int**) ft.size(); k += LSOne(k)) ft[k] = min(ft[k], v);
77. }
79. **void** SetMin(**int** pos, **int** x) {
80. **for** (**int** i = pos + N; i; i >>= 1)
81. ft[i] = min(ft[i], x);
82. }
83. **int** GetMin(**int** L, **int** R) **const** // [L, R) i.e. L <= i < R
84. {
85. **int** res = INF;
86. **for** (L += N, R += N; L < R; L >>= 1, R >>= 1) {
87. **if** (L & 1) {
88. res = min(res, ft[L]);
89. L++;
90. }
91. **if** (R & 1) {
92. R--;
93. res = min(res, ft[R]);
94. }
95. }
96. **return** res;
97. }
99. };

**Permutation & Combination up to 30:**

1. ll dp\_permu[31][31], dp\_combi[31][31];
3. **void** permu() {
4. **for** (**int** i = 0; i <= 30; i++) {
5. dp\_permu[i][0] = 1;
6. dp\_permu[i][1] = i;
8. **for** (**int** j = 2; j <= i; j++)
9. dp\_permu[i][j] = dp\_permu[i][j - 1] \* (ll)(i - j + 1);
10. }
11. }
12. **void** combi() {
13. **for** (**int** i = 0; i < 31; i++) {
14. dp\_combi[i][0] = dp\_combi[i][i] = 1;
15. **for** (**int** j = 1; j < i; j++) {
16. dp\_combi[i][j] = dp\_combi[i - 1][j - 1] + dp\_combi[i - 1][j];
17. }
18. }
19. }

**Chinese Remainder Theorem:**

1. // Returns modulo inverse of a with respect to m using extended
2. // Euclid Algorithm. Refer below post for details:
3. // http://www.geeksforgeeks.org/multiplicative-inverse-under-modulo-m/
4. **int** inv(**int** a, **int** m) {
5. **int** m0 = m, t, q;
6. **int** x0 = 0, x1 = 1;
8. **if** (m == 1)
9. **return** 0;
11. // Apply extended Euclid Algorithm
12. **while** (a > 1) {
13. // q is quotient
14. q = a / m;
16. t = m;
18. // m is remainder now, process same as
19. // euclid's algo
20. m = a % m, a = t;
22. t = x0;
24. x0 = x1 - q \* x0;
26. x1 = t;
27. }
29. // Make x1 positive
30. **if** (x1 < 0)
31. x1 += m0;
33. **return** x1;
34. }
35. // k is size of num[] and rem[].  Returns the smallest
36. // number x such that:
37. //  x % num[0] = rem[0],
38. //  x % num[1] = rem[1],
39. //  ..................
40. //  x % num[k-2] = rem[k-1]
41. // Assumption: Numbers in num[] are pairwise coprime
42. // (gcd for every pair is 1)
43. **int** findMinX(**int** num[], **int** rem[], **int** k) {
44. // Compute product of all numbers
45. **int** prod = 1;
46. **for** (**int** i = 0; i < k; i++)
47. prod \*= num[i];
49. // Initialize result
50. **int** result = 0;
52. // Apply above formula
53. **for** (**int** i = 0; i < k; i++) {
54. **int** pp = prod / num[i];
55. result += rem[i] \* inv(pp, num[i]) \* pp;
56. }
58. **return** result % prod;
59. }

**A^B^C:**

1. /// a^x % m = ((a%m) ^ (x%phi(m))) %m (provided a and m coprime)
3. int64 call(**int** pos, **int** l, **int** cnt) {
4. **if** (adj[pos].size() == 0) **return** b[pos] % mod[cnt];
5. int64 & ret = a[l][pos];
6. **if** (ret != -1) **return** ret;
7. ret = 0;
8. **for** (auto x: adj[pos]) {
9. ret += BigMod(b[pos], call(x, l, cnt + 1), mod[cnt]);
10. ret %= mod[cnt];
11. }
12. **return** ret;
13. }
14. **int** main() {
15. mod[0] = MOD;
16. For(i, 20) mod[i] = phi(mod[i - 1]);
17. memo(a, -1);
18. n = II, m = II, k = II;
19. For(i, n) b[i] = IL;
20. For(i, k) {
21. **int** x = II;
22. rep(j, x) {
23. **int** y = II;
24. lev[y] = i;
25. }
26. }
27. rep(i, m) {
28. **int** u = II, v = II;
29. **if** (lev[u] > lev[v]) swap(u, v);
30. adj[u].pb(v);
31. }
33. **int** q = II;
34. **while** (q--) {
35. **int** y = II;
36. int64 x = call(y, lev[y], 0);
37. printf("%lld\n", x);
38. }
39. }

**Longest Balance String:**

1. **int** longestBalanceString(string s) {
2. **int** l = s.size();
3. vector < **int** > r(l);
4. vector < **int** > c(l);
5. stack < **int** > stk;
6. **for** (**int** i = 0; i < l; i++) {
7. **if** (s[i] == '1') stk.push(i);
8. **else** {
9. **if** (stk.size() > 0) {
10. **int** tmp = stk.top();
11. stk.pop();
12. r[tmp] = i + 1;
13. c[i] = tmp + 1;
14. }
15. }
16. }
17. **int** len = 0, ans = 0;
18. **for** (**int** i = 0; i < l; ++i) {
19. **if** (s[i] == '1' && r[i] == 0) len = 0;
20. **else** **if** (s[i] == '0' && c[i] == 0) len = 0;
21. **else** **if** (s[i] == '1') {
22. len += (r[i] - i);
23. i = r[i] - 1;
24. }
25. ans = max(ans, len);
26. }
27. **return** ans;
28. }

**Multiplication by Addition:**

1. /\*\* this function calculates (a\*b)%c taking into account
2. that a\*b might overflow \*\*/
3. **long** **long** mulmod(**long** **long** a,**long** **long** b,**long** **long** c){
4. **long** **long** x = 0,y=a%c;
5. **while**(b > 0){
6. **if**(b%2 == 1){
7. x = (x+y)%c;
8. }
9. y = (y\*2)%c;
10. b /= 2;
11. }
12. **return** x%c;
13. }

**Sibling DP (DP on TREE):**

1. /\*\*Given a map of the kingdom's  cities, find and print the number of
2. ways King Arthur can divide it between his two children such that they
3. will not invade each other. The first sibling will invade the second sibling's
4. city if the second sibling has no other cities directly connected to it.
5. \*/
6. #define NODE 100000
7. vector<**int**> adj[NODE+10];
8. **int** mod = 1e9+7;
9. vlong dp2 ( **int** s, **int** p, **int** c, **int** cur, **int** ally );
10. vlong dp ( **int** s, **int** p, **int** cur, **int** ally )   ///This is f() in editorial
11. {
12. vlong res = 0;
13. res = dp2 ( s, p, 0, cur, ally ); ///Go and fetch some ally from the children
14. **return** res;
15. }
16. **int** memo[NODE+10][2][2];
17. **int** done[NODE+10][2][2], cc = 1;
18. ///Instead of recording s and c separately, we can simply store adj[s][c] in memo.
19. ///No need to store p, since it is a rooted tree. Parent of a node is always same.
20. vlong dp2 ( **int** s, **int** p, **int** c, **int** cur, **int** ally )   ///This is g() in editorial
21. {
22. vlong res = 0;
23. **if** ( c == adj[s].size() )   ///Base case
24. {
25. **return** ally;
26. }
27. **int** t = adj[s][c];
28. ///Ignore back edge to parent of s. Rooted tree is not suppose to have this back edge anyway.
29. **if** ( t == p )
30. {
31. res = dp2 ( s, p, c + 1, cur, ally );
32. **return** res;
33. }
34. **if** ( done[t][cur][ally] == cc ) **return** memo[t][cur][ally];
35. done[t][cur][ally] = cc;
36. ///For each child, we can make it ally with its parent
37. ///And start independent sub-tree dp
38. res = dp ( t, s, cur, 1 ) \* dp2 ( s, p, c + 1, cur, 1 );
39. res %= mod;
40. ///Or, make it different
41. vlong temp = dp ( t, s, 1 - cur, 0 ) \* dp2 ( s, p, c + 1, cur, ally );
42. temp %= mod;
43. res += temp;
44. NORM(res);
45. **return** memo[t][cur][ally] = res;
46. }
47. **void** solution()
48. {
49. scanf ( "%d", &n );
50. FOR(i,1,n)adj[i].clear();
51. FOR(i,0,n-2)
52. {
53. **int** a, b;
54. scanf ( "%d %d", &a, &b );
55. adj[a].pb ( b );
56. adj[b].pb ( a );
57. }
58. cc++;
59. vlong res = dp ( 1, 0, 0, 0 );
60. res += dp ( 1, 0, 1, 0 );
61. NORM(res);/// meaning mod
62. printf ( "%lld\n", res );
63. }

**Tree DP:**

1. /\*\* The first problem we solve is as follows: Given a tree T of N nodes,
2. where each node i has Ci coins attached with it. You have to choose a
3. subset of nodes such that no two adjacent nodes(i.e. nodes connected directly
4. by an edge) are chosen and sum of coins attached with nodes in chosen subset
5. is maximum. \*\*/
7. vector < **int** > adj[N];
8. **int** dp1[N], dp2[N];
9. /// pV is parent of node V
10. **void** dfs(**int** V, **int** pV) {
11. /// for storing sums of dp1 and max(dp1, dp2) for all children of V
12. **int** sum1 = 0, sum2 = 0;
13. /// traverse over all children
14. **for** (auto v: adj[V]) {
15. **if** (v == pV) **continue**;
16. dfs(v, V);
17. sum1 += dp2[v];
18. sum2 += max(dp1[v], dp2[v]);
19. }
20. dp1[V] = C[V] + sum1;
21. dp2[V] = sum2;
22. }
24. **int** main() {
25. cin >> n;
26. **for** (**int** i = 1; i < n; i++) {
27. cin >> u >> v;
28. adj[u].push\_back(v);
29. adj[v].push\_back(u);
30. }
31. dfs(1, 0);
32. **int** ans = max(dp1[1], dp2[1]);
33. cout << ans << endl;
34. }

**New Template:**

1. #pragma comment(linker, "/stack:640000000")
2. #include <bits/stdc++.h>
3. **using** **namespace** std;
5. **const** **double** EPS = 1e-9;
6. **const** **int** INF = 0x7f7f7f7f;
7. **const** **double** PI=acos(-1.0);
9. #define    READ(f)           freopen(f, "r", stdin)
10. #define    WRITE(f)          freopen(f, "w", stdout)
11. #define    MP(x, y)          make\_pair(x, y)
12. #define    PB(x)             push\_back(x)
13. #define    rep(i,n)          for(int i = 1 ; i<=(n) ; i++)
14. #define    repI(i,n)         for(int i = 0 ; i<(n) ; i++)
15. #define    FOR(i,L,R)        for (int i = L; i <= R; i++)
16. #define    ROF(i,L,R)        for (int i = L; i >= R; i--)
17. #define    FOREACH(i,t)      for (typeof(t.begin()) i=t.begin(); i!=t.end(); i++)
18. #define    ALL(p)            p.begin(),p.end()
19. #define    ALLR(p)           p.rbegin(),p.rend()
20. #define    SET(p)            memset(p, -1, sizeof(p))
21. #define    CLR(p)            memset(p, 0, sizeof(p))
22. #define    MEM(p, v)         memset(p, v, sizeof(p))
23. #define    getI(a)           scanf("%d", &a)
24. #define    getII(a,b)        scanf("%d%d", &a, &b)
25. #define    getIII(a,b,c)     scanf("%d%d%d", &a, &b, &c)
26. #define    getL(a)           scanf("%lld",&a)
27. #define    getLL(a,b)        scanf("%lld%lld",&a,&b)
28. #define    getLLL(a,b,c)     scanf("%lld%lld%lld",&a,&b,&c)
29. #define    getC(n)           scanf("%c",&n)
30. #define    getF(n)           scanf("%lf",&n)
31. #define    getS(n)           scanf("%s",n)
32. #define    bitCheck(a,k)     ((bool)(a&(1<<(k))))
33. #define    bitOff(a,k)       (a&(~(1<<(k))))
34. #define    bitOn(a,k)        (a|(1<<(k)))
35. #define    bitFlip(a,k)      (a^(1<<(k)))
36. #define    iseq(a,b)         (fabs(a-b)<EPS)
37. #define    vi    vector < int >
38. #define    vii   vector < vector < int > >
39. #define    pii   pair< int, int >
40. #define    ff    first
41. #define    ss    second
42. #define    ll    long long
43. #define    ull   unsigned long long
44. #define    POPCOUNT           \_\_builtin\_popcount
45. #define    POPCOUNTLL         \_\_builtin\_popcountll
46. #define    RIGHTMOST          \_\_builtin\_ctzll
47. #define    LEFTMOST(x)        (63-\_\_builtin\_clzll((x)))
48. #define    UNIQUE(V) (V).erase(unique((V).begin(),(V).end()),(V).end())
49. **template**< **class** T > **inline** T gcd(T a, T b) { **return** (b) == 0 ? (a) : gcd((b), ((a) % (b))); }
50. **template**< **class** T > **inline** T lcm(T a, T b) { **return** ((a) / gcd((a), (b)) \* (b)); }
51. **template** <**typename** T> string NumberToString ( T Number ) { ostringstream ss; ss << Number; **return** ss.str(); }
52. /// template sayedgkm
53. #define    N       100010
54. #define    M       1000000007
55. **int** dx[]={0,0,1,-1,-1,-1,1,1};
56. **int** dy[]={1,-1,0,0,-1,1,1,-1};
57. **inline** **int** nxt(){**int** aaa;scanf("%d",&aaa);**return** aaa;}
58. **inline** ll lxt(){ll aaa;scanf("%I64d",&aaa);**return** aaa;}
59. **inline** **double** dxt(){**double** aaa;scanf("%lf",&aaa);**return** aaa;}
61. #ifdef mamun
62. #define debug(args...) {cerr<<"\*: "; dbg,args; cerr<<endl;}
63. #else
64. #define debug(args...)  // Just strip off all debug tokens
65. #endif
67. **struct** debugger{
68. **template**<**typename** T> debugger& operator , (**const** T& v){
69. cerr<<v<<" ";
70. **return** \***this**;
71. }
72. }dbg;
73. ///\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* template ends here \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
74. **int** t,n,m;

77. **int** main() {
78. ///check for 0 or -1 if input not specified
79. #ifdef mamun
80. //        READ("in.txt");
81. //        WRITE("out.txt");
82. #endif // mamun
83. //    ios\_base::sync\_with\_stdio(0);cin.tie(0);

86. **return** 0;
87. }
89. //    clock\_t begin, end;
90. //    double time\_spent;
91. //    begin = clock();
92. //
93. //    end = clock();
94. //    time\_spent = (double)(end - begin) / CLOCKS\_PER\_SEC;
95. //    cerr<<"Time spent = "<<time\_spent<<endl;

**LCIS:**

1. **int** n,m,A[505],B[505],prev[505],C[505];
2. **void** LCIS()
3. {
4. **int** i,j,last,cur,idx=-1,length=0;
5. vector <**int**> res;
6. rep(i,n)
7. {
8. cur=0,last=-1;
9. rep(j,m)
10. {
11. **if**(A[i]==B[j] && cur+1>C[j])
12. {
13. C[j]=cur+1;
14. prev[j]=last;
15. }
16. **if**(A[i]>B[j] && cur<C[j])
17. {
18. cur=C[j];
19. last=j;
20. }
21. }
22. }
23. rep(i,m)
24. {
25. //    printf("i:: %d\n",i,C[i]);
26. **if**(C[i]>length)
27. {
28. length=C[i];
29. idx=i;
31. }
32. }
33. printf("%d\n",length);
34. **while**(idx!=-1)
35. {
36. res.push\_back(B[idx]);
37. idx=prev[idx];
38. }
39. reverse(res.begin(),res.end());
40. **for**(i=0; i<res.size(); i++)
41. {
42. printf("%d%s",res[i],i+1==res.size()?"\n":" ");
43. }
44. }

**Persistent Segment Tree:**

1. /\*\* Problem : Given an array A[] and different point update operations.Considering
2. each point operation to create a new version of the array. We need to answer
3. the queries of type
4. Q v l r : output the sum of elements in range l to r just after the v-th update.
5. \*\*/
6. #define MAXN 100
7. **struct** node {
8. **int** val;
9. /// pointer to left and right children
10. node \* left, \* right;
11. node() {}
12. node(node \* l, node \* r, **int** v) {
13. left = l;
14. right = r;
15. val = v;
16. }
17. };
18. **int** arr[MAXN];
19. /// root pointers for all versions
20. node \* version[MAXN];
21. /// Constructs Version-0
22. /// Time Complexity : O(nlogn)
23. **void** build(node \* n, **int** low, **int** high) {
24. **if** (low == high) {
25. n - > val = arr[low];
26. **return**;
27. }
28. **int** mid = (low + high) / 2;
29. n - > left = **new** node(NULL, NULL, 0);
30. n - > right = **new** node(NULL, NULL, 0);
31. build(n - > left, low, mid);
32. build(n - > right, mid + 1, high);
33. n - > val = n - > left - > val + n - > right - > val;
34. }
36. /\*\*
37. \* Upgrades to new Version
38. \* @param prev : points to node of previous version
39. \* @param cur  : points to node of current version
40. \* Time Complexity : O(logn)
41. \* Space Complexity : O(logn)
42. \*/
43. **void** upgrade(node \* prev, node \* cur, **int** low, **int** high, **int** idx, **int** value) {
44. **if** (idx > high or idx < low or low > high)
45. **return**;
46. **if** (low == high) {
47. /// modification in new version
48. cur - > val = value;
49. **return**;
50. }
51. **int** mid = (low + high) / 2;
52. **if** (idx <= mid) {
53. /// link to right child of previous version
54. cur - > right = prev - > right;
55. /// create new node in current version
56. cur - > left = **new** node(NULL, NULL, 0);
57. upgrade(prev - > left, cur - > left, low, mid, idx, value);
58. } **else** {
59. /// link to left child of previous version
60. cur - > left = prev - > left;
61. /// create new node for current version
62. cur - > right = **new** node(NULL, NULL, 0);
63. upgrade(prev - > right, cur - > right, mid + 1, high, idx, value);
64. }
65. cur - > val = cur - > left - > val + cur - > right - > val;
66. }
68. **int** query(node \* n, **int** low, **int** high, **int** l, **int** r) {
69. **if** (l > high or r < low or low > high)
70. **return** 0;
71. **if** (l <= low and high <= r)
72. **return** n - > val;
73. **int** mid = (low + high) / 2;
74. **int** p1 = query(n - > left, low, mid, l, r);
75. **int** p2 = query(n - > right, mid + 1, high, l, r);
76. **return** p1 + p2;
77. }
79. **int** main(**int** argc, **char**
80. **const** \* argv[]) {
81. repI(i, n) arr[i] = A[i];
82. /// creating Version-0
83. node \* root = **new** node(NULL, NULL, 0);
84. build(root, 0, n - 1);
85. /// storing root node for version-0
86. version[0] = root;
87. /// upgrading to version-1
88. version[1] = **new** node(NULL, NULL, 0);
89. upgrade(version[0], version[1], 0, n - 1, 4, 1);
90. /// upgrading to version-2
91. version[2] = **new** node(NULL, NULL, 0);
92. upgrade(version[1], version[2], 0, n - 1, 2, 10);
93. cout << "In version 1 , query(0,4) : ";
94. cout << query(version[1], 0, n - 1, 0, 4) << endl;
96. cout << "In version 2 , query(3,4) : ";
97. cout << query(version[2], 0, n - 1, 3, 4) << endl;
98. }

**Hashing (sgtlaugh):**

1. /// from sgtlaugh
2. **const** **long** **long** string\_hash(**const** **char** \* str) {
3. **long** **long** h1 = 0, h2 = 0;
4. **for** (**int** i = 0; str[i] != 0; i++) {
5. h1 = ((h1 \* 10007) + str[i] + 97) % 1000000009;
6. h2 = ((h2 \* 997) + str[i] + 143) % 666666667;
7. }
8. **return** (h1 << 31) | h2;
9. }

**NCR:**

1. **long** **long** nCr(**long** **long** m, **long** **long** n) {
2. **long** **long** ans = 1;
3. **for** (**long** **long** i = 1; i <= n; i++) {
4. ans = ans \* (m - i + 1) / i;
5. }
6. **return** ans;
7. }

**Centroid Decomposition:**

1. **const** **int** N = **int**(1e5)+10;
2. **const** **int** LOGN = 20;
3. **const** **int** INF = **int**(1e9);
4. set<**int**> g[N];
5. **int** par[N],sub[N],level[N],ans[N];
6. **int** DP[LOGN][N];
7. **int** n,m;
8. /\*\* Using centroid Decomposition of a tree \*\*/
10. /\*\* ----------- Pre-Processing ------------ \*\*/
11. **void** dfs0(**int** u)
12. {
13. **for**(auto it=g[u].begin(); it!=g[u].end(); it++)
14. **if**(\*it!=DP[0][u])
15. {
16. DP[0][\*it]=u;
17. level[\*it]=level[u]+1;
18. dfs0(\*it);
19. }
20. }
21. **void** preprocess()
22. {
23. level[0]=0;
24. DP[0][0]=0;
25. dfs0(0);
26. **for**(**int** i=1; i<LOGN; i++)
27. **for**(**int** j=0; j<n; j++)
28. DP[i][j] = DP[i-1][DP[i-1][j]];
29. }
30. **int** lca(**int** a,**int** b)
31. {
32. **if**(level[a]>level[b])swap(a,b);
33. **int** d = level[b]-level[a];
34. **for**(**int** i=0; i<LOGN; i++)
35. **if**(d&(1<<i))
36. b=DP[i][b];
37. **if**(a==b)**return** a;
38. **for**(**int** i=LOGN-1; i>=0; i--)
39. **if**(DP[i][a]!=DP[i][b])
40. a=DP[i][a],b=DP[i][b];
41. **return** DP[0][a];
42. }
43. **int** dist(**int** u,**int** v)
44. {
45. **return** level[u] + level[v] - 2\*level[lca(u,v)];
46. }
47. /\*\* -----------------Decomposition Part--------------------------\*/
48. **int** nn;
49. **void** dfs1(**int** u,**int** p)
50. {
51. sub[u]=1;
52. nn++;
53. **for**(auto it=g[u].begin(); it!=g[u].end(); it++)
54. **if**(\*it!=p)
55. {
56. dfs1(\*it,u);
57. sub[u]+=sub[\*it];
58. }
59. }
60. **int** dfs2(**int** u,**int** p)
61. {
62. **for**(auto it=g[u].begin(); it!=g[u].end(); it++)
63. **if**(\*it!=p && sub[\*it]>nn/2)
64. **return** dfs2(\*it,u);
65. **return** u;
66. }
67. **void** decompose(**int** root,**int** p)
68. {
69. nn=0;
70. dfs1(root,root);
71. **int** centroid = dfs2(root,root);
72. **if**(p==-1)p=centroid;
73. par[centroid]=p;
74. **for**(auto it=g[centroid].begin(); it!=g[centroid].end(); it++)
75. {
76. g[\*it].erase(centroid);
77. decompose(\*it,centroid);
78. }
79. g[centroid].clear();
80. }
81. /\*\* ----------------- Handle the Queries -----------------\*/
83. **void** update(**int** u)
84. {
85. **int** x = u;
86. **while**(1)
87. {
88. ans[x] = min(ans[x],dist(x,u));
89. **if**(x==par[x])
90. **break**;
91. x = par[x];
92. }
93. }
94. **int** query(**int** u)
95. {
96. **int** x = u;
97. **int** ret=INF;
98. **while**(1)
99. {
100. ret = min(ret,dist(u,x) + ans[x]);
101. **if**(x==par[x])
102. **break**;
103. x = par[x];
104. }
105. **return** ret;
106. }
107. **int** main()
108. {
109. scanf("%d %d",&n,&m);
110. **for**(**int** i=0; i<n-1; i++)
111. {
112. **int** u,v;
113. scanf("%d %d",&u,&v);
114. g[u-1].insert(v-1);
115. g[v-1].insert(u-1);
116. }
117. preprocess();
118. decompose(0,-1);
119. **for**(**int** i=0; i<n; i++)
120. ans[i]=INF;
121. update(0);//first node is initially painted red
122. **while**(m--)
123. {
124. **int** t,v;
125. scanf("%d %d",&t,&v);
126. v--;
127. **if**(t==1)
128. update(v);
129. **else**
130. dout(query(v));
131. }
132. **return** 0;
133. }