|  |
| --- |
|  |
| CODE LIBRARY |
| MD. SHAHADAT HOSSAIN SHAHIN |
|  |
| UNIVERSITY OF DHAKA |
|  |

Table of Contents

[Template 5](#_Toc507278095)

[Number Theory/Combinatorics 6](#_Toc507278096)

[Bigmod 6](#_Toc507278097)

[Modular Inverse 6](#_Toc507278098)

[Sieve 6](#_Toc507278099)

[Prime Power Factorization 7](#_Toc507278100)

[Mobius 8](#_Toc507278101)

[Binomial Coefficient(nCr % P) 8](#_Toc507278102)

[Chinese Remainder Theorem 9](#_Toc507278103)

[Euler Phi 9](#_Toc507278104)

[Wilson’s Theorem 9](#_Toc507278105)

[Lucas Theorem & Modulo Operation with Composite Number 9](#_Toc507278106)

[Zahin Vai’s Code 11](#_Toc507278107)

[Prime counting Functions 12](#_Toc507278108)

[Disjoint Set Union 14](#_Toc507278109)

[Minimum Spanning Tree (Kruskal) 16](#_Toc507278110)

[Directed Minimum Spanning Tree 16](#_Toc507278111)

[Dijkstra 18](#_Toc507278112)

[Bellman Ford 18](#_Toc507278113)

[Floyd Warshal 19](#_Toc507278114)

[Strongly Connected Component 20](#_Toc507278115)

[Articulation Point 21](#_Toc507278116)

[Bridge 22](#_Toc507278117)

[Bridge Tree 22](#_Toc507278118)

[Biconnected Component 24](#_Toc507278119)

[Euler Trail/Circuit 26](#_Toc507278120)

[Directed Graph 26](#_Toc507278121)

[Undirected Graph 27](#_Toc507278122)

[Centroid Decomposition 29](#_Toc507278123)

[Flow and Matching 31](#_Toc507278124)

[Maximum Flow 31](#_Toc507278125)

[Edmonds Karp 31](#_Toc507278126)

[Dinic 32](#_Toc507278127)

[Notes 33](#_Toc507278128)

[Minimum Cost Maximum Flow 34](#_Toc507278129)

[Maximum Bipartite Matching 36](#_Toc507278130)

[Posets (Partially Ordered Sets) 37](#_Toc507278131)

[Antichain 37](#_Toc507278132)

[Chain 37](#_Toc507278133)

[Mirsky’s Theorem 37](#_Toc507278134)

[Dilworth’s Theorem 37](#_Toc507278135)

[Konig’s Theorem 37](#_Toc507278136)

[Finding Minimum Vertex Cover 37](#_Toc507278137)

[Node Disjoint Path Cover 38](#_Toc507278138)

[General Path Covers 38](#_Toc507278139)

[Hungarian Algorithm(Assignment Problem) 38](#_Toc507278140)

[Dominator Tree 39](#_Toc507278141)

[Notes 39](#_Toc507278142)

[DSU on Tree 41](#_Toc507278143)

[Using Map 41](#_Toc507278144)

[Using Vector 41](#_Toc507278145)

[Heavy Light Trick 42](#_Toc507278146)

[Segment Tree 43](#_Toc507278147)

[Point Update, Range Query 43](#_Toc507278148)

[Range Update, Range Query 44](#_Toc507278149)

[How Many Non Zero Elements in the Tree 45](#_Toc507278150)

[Implicit Segment Tree 46](#_Toc507278151)

[Point Update, Range Query 46](#_Toc507278152)

[Range Update, Range Query 47](#_Toc507278153)

[Persistent Segment Tree 49](#_Toc507278154)

[Point Update Range Query( Static Array ) 49](#_Toc507278155)

[Binary Indexed Tree 50](#_Toc507278156)

[Mo’s Algorithm 51](#_Toc507278157)

[Query on Path of A Tree 52](#_Toc507278158)

[Way 1 52](#_Toc507278159)

[Way 2 52](#_Toc507278160)

[Lowest Common Ancestor 52](#_Toc507278161)

[Trie 53](#_Toc507278162)

[Static Array Implementation 53](#_Toc507278163)

[Finding Maximum and Minimum Xor Match 54](#_Toc507278164)

[Heavy Light Decomposition 54](#_Toc507278165)

[Treap 56](#_Toc507278166)

[Normal Treap 56](#_Toc507278167)

[Implicit Treap 58](#_Toc507278168)

[Maximum Contiguous Sum Merging 60](#_Toc507278169)

[Matrix Exponentiation 61](#_Toc507278170)

[Extended Euclid (Solving Linear Diophantine Equation) 62](#_Toc507278171)

[Discrete Logarithm 65](#_Toc507278172)

[Fraction Class 66](#_Toc507278173)

[Gaussian Elimination 67](#_Toc507278174)

[Gauss-Jordan Elimination 68](#_Toc507278175)

[Gauss Related Problem 69](#_Toc507278176)

[Multiplying Two Polynomials 71](#_Toc507278177)

[Notes 71](#_Toc507278178)

[FFT 71](#_Toc507278179)

[Karatsuba 72](#_Toc507278180)

[Knuth Morris Pratt 74](#_Toc507278181)

[Notes 75](#_Toc507278182)

[Suffix Array 75](#_Toc507278183)

[Notes 75](#_Toc507278184)

[n \* lg2n \* lg2n 75](#_Toc507278185)

[n \* lg n 76](#_Toc507278186)

[n \* lg n (Zobayer Vai) 77](#_Toc507278187)

[Longest Increasing Subsequence (nlog(n)) 79](#_Toc507278188)

[Sorting an Array with Minimum Moves 80](#_Toc507278189)

[Type 1 80](#_Toc507278190)

[Moves 80](#_Toc507278191)

[Solution 80](#_Toc507278192)

[Type 2 80](#_Toc507278193)

[Moves 80](#_Toc507278194)

[Solution 80](#_Toc507278195)

[Type 3 80](#_Toc507278196)

[Move 80](#_Toc507278197)

[Solution 80](#_Toc507278198)

[Fast IO 80](#_Toc507278199)

[Game Theory 81](#_Toc507278200)

[Impartial Game 81](#_Toc507278201)

[Nim Game 81](#_Toc507278202)

[Turning Turtles 81](#_Toc507278203)

[Grundy Number 81](#_Toc507278204)

[Colon principle 82](#_Toc507278205)

[Fusion Principle 82](#_Toc507278206)

[Sudoku Solver 83](#_Toc507278207)

[Knight Distance in Infinite Chessboard 84](#_Toc507278208)

[STL 84](#_Toc507278209)

[Vector 85](#_Toc507278210)

[Priority Queue 85](#_Toc507278211)

[Set 85](#_Toc507278212)

[Policy Based Data Structures 85](#_Toc507278213)

[Bitset 86](#_Toc507278214)

[Convex Hull 87](#_Toc507278215)

[Important Mathematical Formulas 88](#_Toc507278216)

[Modular Arithmatic 90](#_Toc507278217)

[Miscellaneous 92](#_Toc507278218)

[String Hash + Seg Tree 93](#_Toc507278219)

[Discrete Log(General) 95](#_Toc507278220)

[Discrete Log 97](#_Toc507278221)

[Random Code 98](#_Toc507278222)

[Circle Operations 101](#_Toc507278223)

[Line Operations 105](#_Toc507278224)

Template

1. #include <bits/stdc++.h>
3. **using** **namespace** std;
5. **typedef** **long** **long** ll;
6. **typedef** unsigned **long** **long** ull;
7. **typedef** **long** **double** ld;
9. #define si(a)           scanf("%d",&a)
10. #define sii(a,b)        scanf("%d %d",&a,&b)
11. #define siii(a,b,c)     scanf("%d %d %d",&a,&b,&c)
13. #define sl(a)           scanf("%lld",&a)
14. #define sll(a,b)        scanf("%lld %lld",&a,&b)
15. #define slll(a,b,c)     scanf("%lld %lld %lld",&a,&b,&c)
17. #define pb              push\_back
18. #define PII             pair <int,int>
19. #define PLL             pair <ll,ll>
20. #define mp              make\_pair
21. #define xx              first
22. #define yy              second
23. #define all(v)          v.begin(),v.end()
24. #define un(x)           x.erase(unique(all(x)), x.end())
26. #define D(x)            cerr << #x " = " << (x) << '\n'
27. #define DBG             cerr << "In" << '\n'
29. #define CLR(a)          memset(a,0,sizeof(a))
30. #define SET(a)          memset(a,-1,sizeof(a))
32. #define eps             1e-9
33. #define PI              acos(-1.0)
34. #define MAX             300010
35. #define MOD             1000000007
36. #define INF             2000000000
38. **int** setBit(**int** n,**int** pos){ **return** n = n | (1 << pos); } //sets the pos'th bit to 1
39. **int** resetBit(**int** n,**int** pos){ **return** n = n & ~(1 << pos); } //sets the pos'th bit to 0
40. **bool** checkBit(**int** n,**int** pos){ **return** (**bool**)(n & (1 << pos)); } //returns the pos'th bit
42. //int dx[] = {+0, +0, +1, -1, -1, +1, -1, +1};
43. //int dy[] = {-1, +1, +0, +0, +1, +1, -1, -1}; //Four & Eight Direction

Number Theory/Combinatorics

## Bigmod

1. /// returns (a^b) % m
2. ll bigMod(ll a,ll b,ll m){
3. ll ret = 1LL;
4. a %= m;
5. **while** (b){
6. **if** (b & 1LL) ret = (ret \* a) % m;
7. a = (a \* a) % m;
8. b >>= 1LL;
9. }
10. **return** ret;
11. }

## Modular Inverse

1. /// returns (x,y) of the equation ax + by = gcd(a,b)
2. PLL extEuclid(ll a,ll b) {
3. **if**(b==0LL) **return** make\_pair(1LL,0LL);
4. PLL ret,got;
5. got = extEuclid(b,a%b);
6. ret = make\_pair(got.yy,got.xx-(a/b)\*got.yy);
7. **return** ret;
8. }
10. /// returns modular invers of a with respect to m
11. /// inverse exists if and only if a and m are co-prime
12. ll modularInverse(ll a, ll m){
13. ll x, y, inv;
14. PLL sol = extEuclid(a,m);
15. inv = (sol.xx + m) % m;
16. **return** inv;
17. }

## Sieve

1. **bool** isComp[MAX];  // ara[i] is true if i is composite
2. vector <**int**> primes;
4. **void** Sieve(**int** N){
5. **int**  i,j,sq = sqrt(N);
6. **for**(i=4;i<=N;i+=2) isComp[i] = **true**;
7. **for**(i=3;i<=sq;i+=2){
8. **if**(!isComp[i]){
9. **for**(j=i\*i;j<=N;j+=i+i) isComp[j] = 1;
10. }
11. }
12. **for**(i=2;i<=N;i++) **if**(!isComp[i]) primes.pb(i);
13. }

## Prime Power Factorization

1. /\*\*\*
2. A call to generatePPF(int N) will generate the prime power
3. factorization of numbers upto N
4. TLE khele vector ke array diye replace korte hobe
5. Overall complexity is almost n(log n)
6. \*\*\*/
8. vector <PII> factor[MAX];
9. // factor[x] contains (p,i) if prime p divides x i times
10. **bool** isComp[MAX]; // true if a number is composite
11. **int** lp[MAX]; // least prime factor
13. **void** Sieve(**int** N){
14. **int**  i,j,sq = sqrt(N);
15. **for**(i=1;i<=N;i++) lp[i] = i;
16. **for**(i=4;i<=N;i+=2) isComp[i] = **true** , lp[i] = 2;
17. **for**(i=3;i<=sq;i+=2){
18. **if**(isComp[i]) **continue**;
19. **for**(j=i\*i;j<=N;j+=i+i) isComp[j] = 1 , lp[j] = min(lp[j],i);
20. }
21. }
23. **void** generatePPF(**int** N) {
24. Sieve(N);
25. **int** now;
26. vector <**int**> temp;
27. **for**(**int** num=2;num<=N;num++) {
28. now = num;
29. temp.clear();
30. **while**(lp[now]!=1){
31. temp.pb(lp[now]);
32. now = now/lp[now];
33. }
34. **int** cnt = 1;
35. **for**(**int** i=1;i<temp.size();i++){
36. **if**(temp[i]==temp[i-1]) cnt++;
37. **else**{
38. factor[num].push\_back(mp(temp[i-1],cnt));
39. cnt = 1;
40. }
41. }
42. factor[num].push\_back(mp(temp[temp.size()-1],cnt));
43. }
44. }
46. /\*\*\*
47. Another Way
48. Takes more time than the previous but soto code
49. \*\*\*/
51. vector <PII> factor[MAX];
52. // factor[x] contains (p,i) if prime p divides x i times
53. **bool** isComp[MAX]; // true if a number is composite

56. **void** generatePPF(**int** N) {
57. **int** i,j,cnt,tmp,x;
58. **for**(i=2;i<=N;i++) {
59. **if**(!isComp[i]) {
60. **for**(j=i;j<=N;j+=i) {
61. isComp[j] = **true**;
62. tmp = j , cnt = 0;
63. **while**(**true**) {
64. x = tmp/i;
65. **if**(x\*i!=tmp) **break**;
66. tmp = x;
67. cnt++;
68. }
69. factor[j].pb({i,cnt});
70. }
71. }
72. }
73. }

## Mobius

1. /\*\*\*
2. mu[1] = 1, mu[n] = 0 if n has a squared prime factor,
3. mu[n] = 1 if n is square-free with even number of prime factors
4. mu[n] = -1 if n is square-free with odd number of prime factors
6. \*\*\* sum of mu[d] where d | n is 0 ( For n=1, sum is 1 )\*\*\*
7. \*\*\*/
9. **int** mu[MAX] = {0};
11. **void** Mobius(**int** N){
12. **int** i, j;
13. mu[1] = 1;
14. **for** (i = 1; i <= N; i++){
15. **if** (mu[i]){
16. **for** (j = i + i; j <= N; j += i){
17. mu[j] -= mu[i];
18. }
19. }
20. }
21. }

## Binomial Coefficient(nCr % P)

1. **const** **int** MOD = some prime;
2. **int** inv[MAX],fact[MAX];
4. **void** precal(**int** N) {
5. fact[0] = 1;
6. **for**(**int** i=1;i<=N;i++)  fact[i] = ( (**long** **long**)fact[i-1]\*i )  % MOD;
7. inv[N] = bigMod(fact[N],MOD-2,MOD);
8. **for** (**int** i = N - 1 ; i >= 0; i--)
9. inv[i] = ( (**long** **long**)inv[i + 1]\*(i + 1) ) % MOD;
10. }
11. /// returns nCr
12. **int** bin(**int** n,**int** r) {
13. **if**(n<r) **return** 0;
14. ll ret = fact[n];
15. ret \*= inv[r] , ret %= MOD;
16. ret \*= inv[n-r] , ret %= MOD;
17. **return** ret;
18. }

## Chinese Remainder Theorem

1. /\*
2. X = a\_1 % m\_1
3. X = a\_2 % m\_2
4. X = a\_3 % m\_3
6. m\_1,m\_2,m\_3 are pair wise co-prime
8. M = m\_1\*m\_2\*m\_3
10. u\_i = Modular inverse of (M/m\_i) with respect m\_i
12. X = ( a\_1 \* (M/m\_1) \* u\_1 + a\_2 \* (M/m\_2) \* u\_2 + a\_3 \* (M/m\_3) \* u\_3 ) % M
13. \*/

## Euler Phi

1. /\*
2. ( a ^ b ) % m = ( a ^ (phi [m] + b % phi [m]) ) % m
3. where (b >= phi [m])
4. \*/

## Wilson’s Theorem

1. /\*
2. (p-1)! = -1 (mod p) if p is a prime
3. \*/

## Lucas Theorem & Modulo Operation with Composite Number

1. /\*
2. If we need to find nCr % P where P is a prime but P can be
3. less than n or r, we can use Lucas Theorem.
5. nCm = ((n\_0 C r\_0) \* (n\_1 C r\_1) \* (n\_2 C r\_2) \* ... \* (n\_k C r\_k)) % P
7. Where n\_i is the i'th digit in P based representation of n
8. and r\_i is the i'th digit in P based representation of r
10. \*\* What if P is a composite number? \*\*
12. P = (p\_0 ^ a\_0) \* (p\_1 ^ a\_1) \* ... \* (p\_k ^ a\_k)
13. where all p\_i are prime numbers.
15. nCr = (n!)/((r)!\*(n-r)!)
17. If all a\_i are 1, then we can use lucas to find individual mods for
18. each p\_i and combine those using CRT
20. If any a\_i is greater than 1,
22. Let's Suppose,  n! = (p\_i ^ u) \* x
23. (n-r)! = (p\_i ^ v) \* y
24. (r)! = (p\_i ^ w) \* z
25. (See the code for calculation of x,y,z when
26. n or r has large value)
28. Let's suppose p\_i ^ a\_i = t,
29. gcd(t,x) = gcd(t,y) = gcd(t,z) = 1, so, x,y,z will have modular inverse
30. with respect to t (see Note 1)
32. So, we will find ( x /(y\*z) ) % (p\_i^a\_i) and then multiply the
33. result by (p\_i ^ s) where s = u - v - w;
34. If, s is not smaller than a\_i, then the result is 0.
36. Then, we will use CRT to combine the result.
37. Actually, we don't need Lucas theorem anymore. This technique
38. will work for a\_i = 1 also.

41. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Note 1\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
42. phi(p^a) = (p^a) - (p^(a-1)) if p is prime
44. a ^ phi(p^x) = 1 (mod p^x)  if gcd(a,p) = 1
45. modular inverse of a with respect to p^a is
46. a ^ ( phi(p^x) - 1 ) % (p^x)
47. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
48. \*/


52. // returns factorail(n) % (p^a) ignoring prime number p
53. // can be done using a loop if n is small
54. // complexity p^a \* log(p^a)
56. ll fact[MAX]; // size at least p^a
58. ll call(ll n,ll p,ll a)
59. {
60. ll ret = 1LL;
61. ll y,x,m = 1;
63. //m = p^a
64. **for**(ll i=1;i<=a;i++) m \*= p;
66. fact[0] = 1;
67. **for**(ll i=1;i<=m;i++){
68. **if**(i%p==0) fact[i] = fact[i-1];
69. **else** fact[i] = (fact[i-1]\*i)%m;
70. }
72. **while**(**true**){
73. **if**(n==0) **break**;
75. y = n/m;
76. ret \*= bm(fact[m],y,m);
77. ret %= m;
79. y = n%m;
81. ret \*= fact[y];
82. ret %= m;
83. n /= p;
84. }
85. **return** ret;

}

### Zahin Vai’s Code

1. #include <bits/stdtr1c++.h>
3. #define MAXP 100010
4. #define clr(ar) memset(ar, 0, sizeof(ar))
5. #define read() freopen("lol.txt", "r", stdin)
6. #define dbg(x) cout << #x << " = " << x << endl
8. **using** **namespace** std;
10. /// Lucas theorem to calculate binomial co-efficients modulo a prime
12. **namespace** lc{
13. **int** MOD = 1000000007;
14. **int** fact[MAXP], inv[MAXP];
16. /// Call once with the modulo prime
17. **void** init(**int** prime){
18. MOD = prime;
19. fact[0] = 1, inv[MOD - 1] = MOD - 1;
20. **for** (**int** i = 1; i < MOD; i++) fact[i] = ((**long** **long**)fact[i - 1] \* i) % MOD;
21. **for** (**int** i = MOD - 2; i >= 0; i--) inv[i] = ((**long** **long**)inv[i + 1] \* (i + 1)) % MOD;
22. }
24. **inline** **int** count(**int** n, **int** k){
25. **if** (k > n) **return** 0;
26. **int** x = ((**long** **long**)inv[n - k] \* inv[k]) % MOD;
27. **return** ((**long** **long**)x \* fact[n]) % MOD;
28. }
30. /// Lucas theorem, calculates binomial(n, k) modulo MOD, MOD must be a prime
31. **inline** **int** binomial(**long** **long** n, **long** **long** k){
32. **if** (k > n) **return** 0;
34. **int** res = 1;
35. k = min(k, n - k);
36. **while** (k && res){
37. res = ((**long** **long**)res \* count(n % MOD, k % MOD)) % MOD;
38. n /= MOD, k /= MOD;
39. }
40. **return** res;
41. }
43. /\*\*\* Alternate and extended functionalities \*\*\*/
45. /// Must call init with prime before (Or set lc::MOD = prime)
46. /// Computes (n! / (p ^ (n / p))) % p in O(p log(n)) time, p MUST be a prime
47. /// That is, calculating n! without p's powers
48. /// For instance factmod(9, 3) = (1 \* 2 \* 4 \* 5 \* 2 \* 7 \* 8 \* 1) % 3 = 1
49. **inline** **int** factmod(**long** **long** n, **int** p){
50. **int** i, res = 1;
51. **while** (n > 1) {
52. **if** ((n / p) & 1) res = ((**long** **long**)res \* (p - 1)) % p;
53. **for** (i = n % p; i > 1; i--) res = ((**long** **long**)res \* i) % p;
54. n /= p;
55. }
56. **return** (res % p);
57. }
59. **inline** **int** expo(**int** a, **int** b){
60. **int** res = 1;
62. **while** (b){
63. **if** (b & 1) res = (**long** **long**)res \* a % MOD;
64. a = (**long** **long**)a \* a % MOD;
65. b >>= 1;
66. }
67. **return** res;
68. }
70. /// Trailing zeros of n! in base p, p is a prime
71. **inline** **long** **long** fact\_ctz(**long** **long** n, **long** **long** p){
72. **long** **long** x = p, res = 0;
73. **while** (n >= x){
74. res += (n / x);
75. x \*= p;
76. }
77. **return** res;
78. }
80. /// Calculates binomial(n, k) modulo MOD, MOD must be a prime
81. **inline** **int** binomial2(**long** **long** n, **long** **long** k){
82. **if** (k > n) **return** 0;
83. **if** (fact\_ctz(n, MOD) != (fact\_ctz(k, MOD) + fact\_ctz(n - k, MOD))) **return** 0;
84. **int** a = factmod(n - k, MOD), b = factmod(k, MOD), c = factmod(n, MOD);
85. **int** x = ((**long** **long**)expo(a, MOD - 2) \* expo(b, MOD - 2)) % MOD;
86. **return** ((**long** **long**)x \* c) % MOD;
87. }
88. }
90. **int** main(){
91. lc::init(997);
92. printf("%d\n", lc::binomial(10, 5));
93. printf("%d\n", lc::binomial(1996, 998));
95. lc::MOD = 10007;
96. printf("%d\n", lc::binomial2(10, 5));
97. printf("%d\n", lc::binomial2(1996, 998));
98. **return** 0;
99. }

## Prime counting Functions

[Efficient Prime Counting with the Meissel-Lehmer Algorithm](http://acganesh.com/blog/2016/12/23/prime-counting)

[Legendre's Formula](http://mathworld.wolfram.com/LegendresFormula.html)

1. **namespace** pcf{
2. ///   Prime-Counting Function
3. ///   initialize once by calling init()
4. ///   Legendre(n) and Lehmer(n) returns number of primes less than or equal to m
5. ///   Lehmer(n) is faster
7. #define MAXN 1000010 /// initial sieve limit
8. #define MAX\_PRIMES 1000010 /// max size of the prime array for sieve
9. #define PHI\_N 100000 ///
10. #define PHI\_K 100
12. **int** len = 0; /// total number of primes generated by sieve
13. **int** primes[MAX\_PRIMES];
14. **int** counter[MAXN]; /// counter[m] --> number of primes <= i
15. **int** phi\_dp[PHI\_N][PHI\_K]; /// precal of phi(n,k)

18. bitset <MAXN> isComp;
19. **void** Sieve(**int** N){
20. **int**  i,j,sq = sqrt(N);
21. isComp[1] = **true**;
22. **for**(i=4;i<=N;i+=2) isComp[i] = **true**;
23. **for**(i=3;i<=sq;i+=2){
24. **if**(!isComp[i]){
25. **for**(j=i\*i;j<=N;j+=i+i) isComp[j] = 1;
26. }
27. }
28. **for** (i = 1; i <= N; i++){
29. **if** (!isComp[i]) primes[len++] = i;
30. counter[i] = len;
31. }
32. }
34. **void** init(){
35. Sieve(MAXN - 1);
36. /// precalculation of phi upto size (PHI\_N,PHI\_K)
37. **int** k , n , res;
38. **for**(n = 0; n < PHI\_N; n++) phi\_dp[n][0] = n;
39. **for** (k = 1; k < PHI\_K; k++){
40. **for** (n = 0; n < PHI\_N; n++){
41. phi\_dp[n][k] = phi\_dp[n][k - 1] - phi\_dp[n / primes[k - 1]][k - 1];
42. }
43. }
44. }
46. /// returns number of integers less or equal n which are
47. /// not divisible by any of the first k primes
48. /// recurrence --> phi( n , k ) = phi( n , k-1 ) - phi( n / p\_k , k-1)
49. **long** **long** phi(**long** **long** n, **int** k){
50. **if** (n < PHI\_N && k < PHI\_K) **return** phi\_dp[n][k];
51. **if** (k == 1) **return** ((++n) >> 1);
52. **if** (primes[k - 1] >= n) **return** 1;
53. **return** phi(n, k - 1) - phi(n / primes[k - 1], k - 1);
54. }
56. **long** **long** Legendre(**long** **long** n){
57. **if** (n < MAXN) **return** counter[n];
59. **int** lim = sqrt(n) + 1;
60. **int** k = upper\_bound(primes, primes + len, lim) - primes;
61. **return** phi(n, k) + (k - 1);
62. }
64. **long** **long** Lehmer(**long** **long** n){
65. **if** (n < MAXN) **return** counter[n];
67. **long** **long** w , res = 0;
68. **int** i, j, a, b, c, lim;
69. b = sqrt(n), c = Lehmer(cbrt(n)), a = Lehmer(sqrt(b)), b = Lehmer(b);
70. res = phi(n, a) + (((b + a - 2) \* (b - a + 1)) >> 1);
72. **for** (i = a; i < b; i++){
73. w = n / primes[i];
74. lim = Lehmer(sqrt(w)), res -= Lehmer(w);
76. **if** (i <= c){
77. **for** (j = i; j < lim; j++){
78. res += j;
79. res -= Lehmer(w / primes[j]);
80. }
81. }
82. }
83. **return** res;
84. }
85. }

Disjoint Set Union

1. //Time complexity = 5\*number of operations
2. **struct** DisjointSet{
3. **int** \*root,\*rnk,n;
4. DisjointSet(){}
5. DisjointSet(**int** sz){
6. root = **new** **int**[sz+1];
7. rnk = **new** **int**[sz+1];
8. n = sz;
9. }
10. ~DisjointSet(){
11. **delete**[] root;
12. **delete**[] rnk;
13. }
14. **void** init(){
15. **for**(**int** i=1;i<=n;i++){
16. root[i] = i;
17. rnk[i] = 0;
18. }
19. }
20. **int** findRoot(**int** u){
21. **if**(u!=root[u]) root[u] = findRoot(root[u]);
22. **return** root[u];
23. }
24. **void** Merge(**int** u,**int** v){
25. **int** ru = findRoot(u); **int** rv = findRoot(v);
26. **if**(rnk[ru]>rnk[rv]) root[rv] = ru;
27. **else** root[ru] = rv;
28. **if**(rnk[ru]==rnk[rv]) rnk[rv]++;
29. }
30. };
32. **int** main(){
33. DisjointSet \*S;
34. S = **new** DisjointSet(n);
35. S->init();
36. **int** ru = S->findRoot(u);
37. S->Merge(u,v);
38. **delete** S;
40. //or
42. DisjointSet S(n);
43. S.init();
44. **int** ru = S.findRoot(u);
45. S.Merge(u,v);
46. **return** 0;
47. }

Minimum Spanning Tree (Kruskal)

1. **struct** edge{
2. **int** u,v,c;
3. }ara[MAX];
5. **bool** cmp(edge a,edge b) { **return** a.c<b.c;}
6. **int** par[MAX];
8. **int** findParent(**int** u){
9. **if**(par[u]==u) **return** u;
10. **else** **return** par[u] = findParent(par[u]);
11. }
12. **int** kruskal(**int** n,**int** m){
13. sort(ara+1,ara+m+1,cmp);
14. **int** i,mst;
15. mst = 0;
16. **for**(i=1;i<=n;i++) par[i] = i;
17. **for**(i=1;i<=m;i++){
18. edge x = ara[i];
19. par[x.u] = findParent(x.u);
20. par[x.v] = findParent(x.v);
21. **if**(par[x.u]!=par[x.v]){
22. par[par[x.u]] = par[x.v];
23. mst += x.c;
24. }
25. }
26. **return** mst;
27. }

Directed Minimum Spanning Tree

1. /\*
2. Jaan Vai's code
3. Finds cost of forming DMST
4. Runs under V^2log(V) where V is the number of nodes
5. 0 based indexing
6. MM is the number of nodes
7. Put all the outgoing edges from u in E[u]
8. Just call Find\_DMST(root, number of nodes) and it will return the total cost of forming DMST
9. if it returns inf, then initial graph was disconnected
10. \*/
11. **const** **int** MM = ?
12. **const** **int** inf = ?
14. **struct** edge {
15. **int** v, w;
16. edge() {}
17. edge( **int** vv, **int** ww ) { v = vv, w = ww; }
18. **bool** operator < ( **const** edge &b ) **const** { **return** w < b.w; }
19. };
21. vector <edge> E[MM], inc[MM];
22. **int** DirectedMST( **int** n, **int** root, vector <edge> inc[MM] ) {
23. **int** pr[MM];
24. inc[root].clear();
26. /// if any node is not reachable from root, then no mst can be found
27. **for**( **int** i = 0; i < n; i++ ) {
28. sort( inc[i].begin(), inc[i].end() );
29. pr[i] = i;
30. }
31. **bool** cycle = **true**;
32. **while**( cycle ) {
33. cycle = **false**;
34. **int** vis[MM] = {0}, W[MM];
35. vis[root] = -1;
36. **for**( **int** i = 0, t = 1; i < n; i++, t++ ) {
37. **int** u = pr[i], v;
38. **if**( vis[u] ) **continue**;
39. **for**( v = u; !vis[v]; v = pr[inc[v][0].v] ) vis[v] = t;
40. **if**( vis[v] != t ) **continue**;
41. cycle = **true**;
42. **int** sum = 0, super = v;
43. **for**( ; vis[v] == t; v = pr[inc[v][0].v] ) {
44. vis[v]++;
45. sum += inc[v][0].w;
46. }
47. **for**( **int** j = 0; j < n; j++ ) W[j] = INT\_MAX;
48. **for**( ; vis[v] == t + 1; v = pr[inc[v][0].v] ) {
49. vis[v]--;
50. **for**( **int** j = 1; j < inc[v].size(); j++ ) {
51. **int** w = inc[v][j].w + sum - inc[v][0].w;
52. W[ inc[v][j].v ] = min( W[ inc[v][j].v ], w );
53. }
54. pr[v] = super;
55. }
56. inc[super].clear();
57. **for**( **int** j = 0; j < n; j++ ) **if**( pr[j] != pr[ pr[j] ] ) pr[j] = pr[ pr[j] ];
58. **for**( **int** j = 0; j < n; j++ ) **if**( W[j] < INT\_MAX && pr[j] != super ) inc[super].push\_back( edge( j, W[j] ) );
59. sort( inc[super].begin(), inc[super].end() );
60. }
61. }
62. **int** sum = 0;
63. **for**( **int** i = 0; i < n; i++ ) **if**( i != root && pr[i] == i ) sum += inc[i][0].w;
64. **return** sum;
65. }
67. **int** Find\_DMST(**int** root, **int** n) {
68. **bool** visited[MM] = {0};
69. queue <**int**> Q;
70. **for**( **int** i = 0; i < n; i++ ) inc[i].clear();
71. **for**( **int** i = 0; i < n; i++ ) **for**( **int** j = 0; j < E[i].size(); j++ ) {
72. **int** v = E[i][j].v, w = E[i][j].w;
73. inc[v].push\_back( edge( i, w ) );
74. }
75. visited[root] = **true**;
76. Q.push(root);
77. **while**( !Q.empty() ) {
78. **int** u = Q.front(); Q.pop();
79. **for**( **int** i = 0; i < E[u].size(); i++ ) {
80. **int** v = E[u][i].v;
81. **if**( !visited[v] ) {
82. visited[v] = **true**;
83. Q.push(v);
84. }
85. }
86. }
87. /// The given graph is disconnected. So forming any MST is not possible.
88. **for**( **int** i = 0; i < n; i++ ) **if**( !visited[i] ) **return** inf;
89. **return** DirectedMST( n, root, inc );
90. }

Dijkstra

1. vector <**int**> ed[MAX],co[MAX];
2. **int** dis[MAX];
3. **bool** vis[MAX];
5. **struct** node{
6. **int** city,cost;
7. };
9. **bool** operator < (node a,node b){**return** a.cost>b.cost;}
11. **void** dijkstra(**int** s,**int** n)
12. {
13. CLR(vis);
14. **int** i,x,u,v,c;
15. node a,b;
16. **for**(i=1;i<=n;i++) dis[i] = INF;
17. dis[s] = 0;
18. a = {s,0};
19. priority\_queue <node> q;
20. q.push(a);
21. **while**(!q.empty()){
22. a = q.top();
23. q.pop();
24. u = a.city;
25. **if**(!vis[u]){
26. vis[u] = **true**;
27. **for**(i=0;i<ed[u].size();i++){
28. v = ed[u][i];
29. c = co[u][i];
30. **if**(dis[v]>dis[u]+c){
31. dis[v] = dis[u]+c;
32. b = {v,dis[v]};
33. q.push(b);
34. }
35. }
36. }
37. }
38. }

Bellman Ford

1. **int** dis[MAX];
2. **struct** data{
3. **int** u,v,c;
4. } edge[MAX];
6. **bool** bellmanFord(**int** n,**int** e,**int** s){
7. **int** i,j;
8. **for**(i=1;i<=n;i++) dis[i] = INF;
9. dis[s] = 0;
10. **for**(j=1; j<=n-1; j++){
11. **for**(i=1; i<=e; i++){
12. **if**(dis[edge[i].u]!=INF && dis[edge[i].u]+edge[i].c<dis[edge[i].v])
13. dis[edge[i].v] = dis[edge[i].u]+edge[i].c;
14. }
15. }
16. **bool** negativeCycle = **false**;
17. **for**(i=1; i<=e; i++){
18. **if**(dis[edge[i].u]!=INF && dis[edge[i].u]+edge[i].c<dis[edge[i].v]){
19. negativeCycle = **true**;
20. }
21. }
22. **return** negativeCycle;
23. }

Floyd Warshal

1. **int** dis[MAX][MAX],P[MAX][MAX];
2. **void** warshall(**int** n){
3. **int** i,j,k;
4. **for**(i=0; i<n; i++)
5. **for**(j=0; j<n; j++){
6. **if**(dis[i][j]!=INF) P[i][j] = i;
7. **else** P[i][j] = -1;
8. }
9. **for**(k=0; k<n; k++){
10. **for**(i=0; i<n; i++){
11. **for**(j=0; j<n; j++){
12. **if**(dis[i][k]!=INF && dis[k][j]!=INF && dis[i][k]+dis[k][j]<=dis[i][j]){
13. dis[i][j] = dis[i][k]+dis[k][j]; P[i][j] = k;
14. }
15. }
16. }
17. }
18. }
19. **void** printPath(**int** s,**int** d)
20. {
21. **if**(P[s][d]==-1) puts("No Path!");
22. **else** **if**(P[s][d]==s) printf("%d\n",s);
23. **else**{
24. printPath(s,P[s][d]);
25. printPath(P[s][d],d);
26. }
27. }
29. /\* Print d when the function returns \*/

Strongly Connected Component

1. /\*
2. 1 based indexing
3. Step 1: Topsort All the nodes
4. Step 2: Run DFS from the unvisited nodes in topsorted order.
5. This will mark the component related to the node.
6. \*/
8. vector <**int**> edges[MAX],trans[MAX];
9. **int** compNum[MAX];
10. **bool** vis[MAX];
11. **int** cnum;
12. stack <**int**> topSortedNodes;
14. **void** topSort(**int** s){
15. **int** i,x;
16. vis[s] = 1;
17. **for**(i=0; i<edges[s].size(); i++) {
18. x = edges[s][i];
19. **if**(!vis[x]) topSort(x);
20. }
21. topSortedNodes.push(s);
22. }
24. **void** markComponent(**int** s)
25. {
26. **int** i,x;
27. vis[s] = 1;
28. compNum[s] = cnum;
29. **for**(i=0; i<trans[s].size(); i++) {
30. x = trans[s][i];
31. **if**(!vis[x]) markComponent(x);
32. }
33. }
35. // finds the SCC for nodes from 1 to n
36. **void** SCC(**int** n) {
37. **int** i,x;
38. CLR(vis);
39. **for**(**int** i=1; i<=n; i++)
40. **if**(!vis[i]) topSort(i);
42. cnum = 0;
43. CLR(vis);
45. **while**(!topSortedNodes.empty()) {
46. x = topSortedNodes.top();
47. topSortedNodes.pop();
48. **if**(!vis[x]) {
49. cnum++;
50. markComponent(x);
51. }
52. }
53. }

Articulation Point

1. vector <**int**> edges[MAX];
2. **bool** vis[MAX] , isArt[MAX];
3. **int** st[MAX] , low[MAX] , Time = 0 , n;
5. **void** findArt(**int** s,**int** par){
6. **int** i,x,child = 0;
7. vis[s] = 1;
8. Time++;
9. st[s] = low[s] = Time;
10. **for**(i=0;i< edges[s].size();i++){
11. x = edges[s][i];
12. **if**(!vis[x]){
13. child++;
14. findArt(x,s);
15. low[s] = min(low[s],low[x]);
16. **if**(par!=-1 && low[x]>=st[s]) isArt[s] = 1;
17. }
18. **else**{
19. **if**(par!=x) low[s] = min(low[s],st[x]);
20. }
21. }
22. **if**(par==-1 && child>1) isArt[s] = 1;
23. }
25. **void** processArticulation(){
26. Time = 0;
27. **for**(**int** i=1;i=<n;i++) **if**(!vis[i]) findArt(i,-1);
28. }

Bridge

1. vector <**int**> ed[MAX];
2. vector <PII> res;
3. **bool** vis[MAX];
4. **int** st[MAX] , low[MAX] , Time = 0 , n;
6. **void** findBridge(**int** s,**int** par){
7. **int** i,x;
8. vis[s] = 1;
9. Time++;
10. st[s] = low[s] = Time;
11. **for**(i=0;i<ed[s].size();i++){
12. x = ed[s][i];
13. **if**(!vis[x]){
14. findBridge(x,s);
15. low[s] = min(low[s],low[x]);
16. **if**(low[x]>st[s]) res.pb(mp(s,x));
17. }
18. **else**{
19. **if**(par!=x) low[s] = min(low[s],st[x]);
20. }
21. }
22. }
24. **void** processBridge(){
25. Time = 0;
26. **for**(**int** i=1;i<=n;i++) **if**(!vis[i]) findBridge(i,-1);
27. }

Bridge Tree

1. /\*
2. 1 based indexing
4. call to processBridge(node,edges) generates bridge tree
5. and the edge list of that is brTree
7. Clear ed , isBridge , brTree per test case
8. \*/
10. **const** **int** MAXN = ?;
11. **const** **int** MAXE = ?;
13. **struct** edges {
14. **int** u,v;
15. } ara[MAXE];
17. vector <**int**> ed[MAXN]; // actual graph
18. vector <**int**> isBridge[MAXN]; // if the edge is a bridge, the entry will be 1
19. vector <**int**> brTree[MAXN]; // edges of the bridge tree
21. **bool** vis[MAXN];
22. **int** st[MAXN], low[MAXN], Time = 0;
23. **int** cnum; // number of nodes in bridge tree
24. **int** comp[MAXN];
26. **void** findBridge(**int** s,**int** par) {
27. **int** i,x,child = 0,j;
28. vis[s] = 1;
29. Time++;
30. st[s] = low[s] = Time;
31. **for**(i=0; i<ed[s].size(); i++) {
32. x = ed[s][i];
33. **if**(!vis[x]) {
34. child++;
35. findBridge(x,s);
36. low[s] = min(low[s],low[x]);
37. **if**(low[x] > st[s]) {
38. isBridge[s][i] = 1;
39. j = lower\_bound(ed[x].begin(),ed[x].end(),s)-ed[x].begin();
40. isBridge[x][j] = 1;
41. }
42. }
43. **else** **if**(par!=x)
44. low[s] = min(low[s],st[x]);
45. }
46. }
48. **void** dfs(**int** s) {
49. **int** i,x;
50. vis[s] = 1;
51. comp[s] = cnum;
52. **for**(i=0; i<ed[s].size(); i++) {
53. **if**(!isBridge[s][i]) {
54. x = ed[s][i];
55. **if**(!vis[x]) dfs(x);
56. }
57. }
58. }
60. **void** processBridge(**int** n,**int** m) {
61. CLR(vis);
62. Time = 0;
63. **for**(**int** i=1; i<=n; i++) **if**(!vis[i]) findBridge(i,-1);
65. cnum = 0;
66. CLR(vis);
67. **for**(**int** i=1; i<=n; i++) {
68. **if**(!vis[i]) {
69. cnum++;
70. dfs(i);
71. }
72. }
74. n = cnum; //number of nodes in the bridge tree
76. **for**(**int** i=1; i<=m; i++) {
77. **if**(comp[ara[i].u] != comp[ara[i].v]) {
78. brTree[comp[ara[i].u]].pb(comp[ara[i].v]);
79. brTree[comp[ara[i].v]].pb(comp[ara[i].u]);
80. }
81. }
82. }


86. **int** main() {
87. **int** n,m,u,v;
88. scanf("%d %d",&n,&m);
89. **for**(**int** i=1; i<=m; i++) {
90. sii(u,v);
92. ed[u].pb(v);
93. ed[v].pb(u);
95. isBridge[u].pb(0);
96. isBridge[v].pb(0);
98. ara[i].u = u;
99. ara[i].v = v;
100. }
101. **for**(**int** i=1; i<=n; i++) sort(all(ed[i]));
102. processBridge(n,m);
104. **return** 0;
105. }

Biconnected Component

1. /\*
2. 1 based indexing
4. A graph is biconnected if every node is reachable from every other node even after
5. removing a single node.
6. Algorithm of checking Biconnectivity :
7. 1) The graph is connected.
8. 2) There is no articulation point in the graph.
10. In the following code
11. bcc\_counter --> Total number of biconnected components
12. bcc[i] keeps the list of nodes in the i'th BCC
14. edges  should be cleared per test case
16. call to prcoessBCC(n = total number of nodes) will construct bcc
18. \*/
20. **const** **int** MAX = ?; // maximum number of nodes
22. vector <**int**> edges[MAX];
23. **bool** vis[MAX], isArt[MAX];
24. **int** Time;
25. **int** low[MAX],st[MAX];
26. vector <**int**> bcc[MAX];
27. **int** bcc\_counter;
28. stack <**int**> S;
30. **void** popBCC(**int** s,**int** x) {
31. isArt[s] = 1;
32. bcc[bcc\_counter].pb(s);
33. **while**(**true**) {
34. bcc[bcc\_counter].pb(S.top());
35. **if**(S.top()==x) {
36. S.pop();
37. **break**;
38. }
39. S.pop();
40. }
41. bcc\_counter++;
42. }
44. **void** findBCC(**int** s,**int** par) {
45. S.push(s);
46. **int** i,x,child = 0;
47. vis[s] = 1;
48. Time++;
49. st[s] = low[s] = Time;
50. **for**(i=0; i< edges[s].size(); i++) {
51. x = edges[s][i];
52. **if**(!vis[x]) {
53. child++;
54. findBCC(x,s);
55. low[s] = min(low[s],low[x]);
56. **if**(par!=-1 && low[x]>=st[s]) popBCC(s,x);
57. **else** **if**(par==-1) **if**(child>1) popBCC(s,x);
58. }
59. **else** **if**(par!=x) low[s] = min(low[s],st[x]);
60. }
61. **if**(par==-1 && child>1) isArt[s] = 1;
62. }

65. // Finds biconnected components for nodes from 1 to n
66. **void** processBCC(**int** n) {
67. **for**(i=1;i<=MAX;i++)
68. bcc[i].clear();
70. CLR(vis); CLR(isArt);
72. bcc\_counter = 1;
74. **for**(**int** i=1; i<=n; i++) {
75. **if**(!vis[i]) {
76. Time = 0;
77. findBCC(i,-1);
78. **bool** lala = **false**;
79. **while**(!S.empty()) {
80. lala = **true**;
81. bcc[bcc\_counter].push\_back(S.top());
82. S.pop();
83. }
84. **if**(lala) bcc\_counter++;
85. }
86. }
87. bcc\_counter--;
88. }

Euler Trail/Circuit

## Directed Graph

1. /// 1 based graph input
2. /// Fill the edge list ed
3. /// Call findEuler()
5. **const** **int** MAX = ?;
7. vector <**int**> ed[MAX+5], sltn;
9. **int** inDeg[MAX+5], outDeg[MAX+5];
10. **bool** vis[MAX+5];
12. **void** dfs(**int** nd) {
13. vis[nd] = **true**; /// used to check the connectivity of the graph
14. **while**(ed[nd].size()) {
15. **int** v = ed[nd].back();
16. ed[nd].pop\_back();
17. dfs(v);
18. }
19. sltn.pb(nd);
20. }
22. /// returns 0 if no Euler path or circuit exists
23. /// returns 1 if a Euler trail exists
24. /// returns 2 if a Euler circuit exists
25. **int** findEuler (**int** n) {
26. **int** src , snk , ret = 1;
27. **bool** found\_src = **false**, found\_snk = **false**;
29. CLR(inDeg); CLR(outDeg);
31. **for**(**int** u = 1; u <= n; u++) {
32. **for**(**int** i = 0; i<ed[u].size(); i++) {
33. **int** v = ed[u][i];
34. outDeg[u]++;
35. inDeg[v]++;
36. }
37. }
39. **int** diff;
40. **for**(**int** i = 1; i<=n; i++) {
41. diff = outDeg[i] - inDeg[i];
43. **if**(diff == 1) {
44. **if**(found\_src) **return** 0;
45. found\_src = **true**;
46. src = i;
47. }
49. **else** **if** (diff == -1) {
50. **if**(found\_snk) **return** 0;
51. found\_snk = **true**;
52. snk = i;
53. }
55. **else** **if**(diff != 0) **return** 0;
56. }
58. **if**(!found\_src) {
59. /// there actually exists a euler cycle. So you need to pick a random node with non-zero degrees.
60. ret = 2;
61. **for**(**int** i = 1 ; i <= n ; i++) {
62. **if**( outDeg[i] ) {
63. found\_src = **true**;
64. src = i;
65. **break**;
66. }
67. }
68. }
70. **if**(!found\_src) **return** ret; /// every node has out-degree 0
72. CLR(vis);
73. sltn.clear();
74. dfs(src);
75. **for**(**int** i = 1; i<=n; i++) {
76. /// the underlying graph is not even weakly connected.
77. **if**(outDeg[i] && !vis[i]) **return** 0;
78. }
80. /// printing path
81. **for**(**int** i = (**int**)sltn.size()-1; i>=0; i--) printf("%d ",sltn[i]);
82. puts("");
84. **return** ret;
85. }

## Undirected Graph

1. /// 1 based graph input
2. /// Fill the edge list ed
3. /// Call findEuler()
5. **const** **int** MAX = ?;
7. vector <**int**> ed[MAX+5], sltn;
9. **int** deg[MAX+5];
10. **bool** vis[MAX+5];
12. **void** dfs(**int** nd) {
13. vis[nd] = **true**; /// used to check the connectivity of the graph
14. **while**(ed[nd].size()) {
15. **int** v = ed[nd].back();
16. ed[nd].pop\_back();
17. dfs(v);
18. }
19. sltn.pb(nd);
20. }
22. /// returns 0 if no Euler path or circuit exists
23. /// returns 1 if a Euler trail exists
24. /// returns 2 if a Euler circuit exists
25. **int** findEuler (**int** n) {
26. **int** src , snk , ret = 1;
27. **bool** found\_src = **false**, found\_snk = **false**;
29. CLR(deg);
31. **for**(**int** u = 1; u <= n; u++) {
32. **for**(**int** i = 0; i<ed[u].size(); i++) {
33. **int** v = ed[u][i];
34. deg[u]++;
35. deg[v]++;
36. }
37. }
39. **for**(**int** i = 1; i<=n; i++) {
40. **if**( deg[i]&1 ){
41. **if**( !found\_src ) {
42. found\_src = **true**;
43. src = i;
44. }
45. **else** **if**( !found\_snk ) {
46. found\_snk = **true**;
47. snk = i;
48. }
49. **else** **return** 0; // more than two nodes with odd degree
50. }
51. }
53. **if**(!found\_src) {
54. /// there actually exists a euler cycle. So you need to pick a random node with non-zero degree.
55. ret = 2;
56. **for**(**int** i = 1 ; i <= n ; i++) {
57. **if**( deg[i] ) {
58. found\_src = **true**;
59. src = i;
60. **break**;
61. }
62. }
63. }
65. **if**(!found\_src) **return** ret; /// every node has degree 0
67. CLR(vis);
68. sltn.clear();
69. dfs(src);
70. **for**(**int** i = 1; i <= n ; i++) {
71. /// the underlying graph is not even weakly connected.
72. **if**(deg[i] && !vis[i]) **return** 0;
73. }
75. /// printing path
76. **for**(**int** i = (**int**)sltn.size()-1; i>=0; i--) printf("%d ",sltn[i]);
77. puts("");
79. **return** ret;
80. }

Centroid Decomposition

1. /\*
2. Problem : Two kinds of operations in a tree(Initially node 1 is white, others are black)
3. Update v : Change the color of node v to white
4. Query v : Distance of closest white node from node v(can be node v)
5. \*/
7. /\* Using dis[i][j] array to calculate distance between two node in original tree \*/
9. #include <bits/stdc++.h>
11. **using** **namespace** std;
13. **const** **int** MAX = ?;
14. **const** **int** INF = ?;
16. vector <**int**> ed[MAX]; // adjacency list of the input tree
17. **bool** isCentroid[MAX]; // if the node is already a centroid of some part
18. **int** sub[MAX]; // subtree size of a node
19. **int** cpar[MAX]; // parent of a node in the centroid tree
20. **int** clevel[MAX]; // level of a node in centroid tree
21. **int** dis[20][MAX]; // dis[i][j] = distance of node j from the root of the i'th level of decomposition
23. **void** calcSubTree(**int** node,**int** par) {
24. **int** i,v;
25. sub[node] = 1;
26. **for**(i=0; i<ed[node].size(); i++) {
27. v = ed[node][i];
28. **if**(v!=par && !isCentroid[v]) {
29. calcSubTree(v,node);
30. sub[node] += sub[v];
31. }
32. }
33. }
35. **int** nn;// number of nodes in the part
37. **int** getCentroid(**int** node,**int** par) {
38. **int** i,v;
39. **for**(i=0; i<ed[node].size(); i++) {
40. v = ed[node][i];
41. **if**(!isCentroid[v] && v!=par && sub[v]>(nn/2)) **return** getCentroid(v,node);
42. }
43. **return** node;
44. }

47. **void** setDis(**int** node, **int** from, **int** par, **int** l) {
48. dis[from][node] = l;
49. **int** i,v;
50. **for**(i=0; i<ed[node].size(); i++) {
51. v = ed[node][i];
52. **if**(v!=par && !isCentroid[v]) {
53. setDis(v, from, node, l+1);
54. }
55. }
56. }
58. //complexity --> O(nlog(n))
59. **void** decompose(**int** node,**int** par,**int** l) {
60. calcSubTree(node,par);
61. nn = sub[node];
62. **int** c = getCentroid(node,par);
63. setDis(c,l,par,0);
65. isCentroid[c] = **true**;
66. cpar[c] = par;
67. clevel[c] = l;
69. **int** i,v;
70. **for**(i=0; i<ed[c].size(); i++) {
71. v = ed[c][i];
72. **if**(!isCentroid[v]) decompose(v,c,l+1);
73. }
74. }
76. **int** ans[MAX];
78. **inline** **void** update(**int** v)
79. {
80. **int** u = v;
81. **while**(u!=-1) {
82. ans[u] = min(ans[u], dis[clevel[u]][v]);
83. u = cpar[u];
84. }
85. }
87. **inline** **int** query(**int** v) {
88. **int** ret = INF;
89. **int** u = v;
90. **while**(u != -1) {
91. ret = min(ret, dis[clevel[u]][v]+ans[u]);
92. u = cpar[u];
93. }
94. **return** ret;
95. }
96. **int** main() {
97. decompose(root,-1,0);
98. **for**(**int** i=1; i<=n; i++) ans[i] = INF;
99. update(v);
100. query(v));
101. **return** 0;
102. }

Flow and Matching

## Maximum Flow

### Edmonds Karp

1. //Edmonds Carp Algorithm
2. //Finds Max Flow using ford fulkerson method
3. //Finds path from source to sink using bfs
4. //Complexity V\*E\*E
6. vector <**int**> ed[MAX];
7. **int** cap[MAX][MAX];
8. **int** par[MAX]; //keeps track of the parent in a path from s to d
9. **int** mCap[MAX]; //mCap[i] keeps track edge that have minimum cost on the shortest path from s to i
11. **bool** getPath(**int** s,**int** d,**int** n){
12. **for**(**int** i=0; i<=n; i++) mCap[i] = INF;
13. SET(par);
14. queue <**int**> q;
15. q.push(s);
16. **while**(!q.empty()){
17. **int** u = q.front();
18. q.pop();
19. **for**(**int** i=0; i<ed[u].size(); i++){
20. **if**(cap[u][ed[u][i]]!=0 && par[ed[u][i]]==-1){
21. par[ed[u][i]] = u;
22. mCap[ed[u][i]] = min(mCap[u],cap[u][ed[u][i]]);
23. **if**(ed[u][i]==d) **return** **true**;
24. q.push(ed[u][i]);
25. }
26. }
27. }
28. **return** **false**;
29. }
31. **int** getFlow(**int** s,**int** d,**int** n){
32. **int** F = 0;
33. **while**(getPath(s,d,n)){
34. **int** f = mCap[d];
35. F += f;
36. **int** u = d;
37. **while**(u!=s){
38. **int** v = par[u];
39. cap[u][v] += f;
40. cap[v][u] -= f;
41. u = v;
42. }
43. }
44. **return** F;
45. }
47. **int** main(){
48. **int** maxFlow = getFlow(s,d,n);
49. **return** 0;
50. }

### Dinic

1. /// Complexity V\*V\*E
3. /// For every testcase
4. /// Clear ed
5. /// SET M if the second addEdge function is used
7. #define MAXN    5010
8. **int** src,snk;
9. **int** dist[MAXN] , work[MAXN];
11. **struct** Edge{
12. **int** to, rev\_pos, c, f;
13. };
15. vector <Edge> ed[MAXN];
17. **void** addEdge(**int** u,**int** v,**int** c)  {
18. Edge a = {v,ed[v].size(),c,0};
19. Edge b = {u,ed[u].size(),c,0}; /// cap c should be replaced by 0 for directed graphs
20. ed[u].pb(a);
21. ed[v].pb(b);
22. }
24. /\*\* if there are multiple edges between same pair of nodes
25. v will not appear in the edge list of u multiple times \*\*/
26. **int** M[MAXN][MAXN];
27. **void** addEdge(**int** u,**int** v,**int** c){
28. **if**(M[u][v]==-1){
29. M[u][v] = ed[u].size();
30. M[v][u] = ed[v].size();
31. Edge a = {v,M[u][v],c,0};
32. Edge b = {u,M[v][u],c,0};/// c should be replaced by 0 for directed graphs
33. ed[u].pb(a);
34. ed[v].pb(b);
35. }
36. **else**{
37. **int** x = M[u][v];
38. **int** y = M[v][u];
39. ed[u][x].c += c;
40. ed[v][y].c += c; /// If the graph is directed, comment it out
41. }
42. }
44. **bool** dinic\_bfs(){
45. SET(dist);
46. dist[src] = 0;
47. queue <**int**> q;
48. q.push(src);
49. **while**(!q.empty()){
50. **int** u = q.front();
51. q.pop();
52. **for**(**int** i=0; i<ed[u].size(); i++){
53. Edge &e = ed[u][i];
54. **int** v = e.to;
55. **if**(dist[v]==-1 && e.f<e.c){
56. dist[v] = dist[u]+1;
57. q.push(v);
58. }
59. }
60. }
61. **return** (dist[snk]>=0);
62. }
64. **int** dinic\_dfs(**int** u, **int** fl){
65. **if** (u == snk) **return** fl;
66. **for** (; work[u] < ed[u].size(); work[u]++){
67. Edge &e = ed[u][work[u]];
68. **if** (e.c <= e.f) **continue**;
69. **int** v = e.to;
70. **if** (dist[v] == dist[u] + 1){
71. **int** df = dinic\_dfs(v, min(fl, e.c - e.f));
72. **if** (df > 0){
73. e.f += df;
74. ed[v][e.rev\_pos].f -= df;
75. **return** df;
76. }
77. }
78. }
79. **return** 0;
80. }
82. **int** maxFlow(**int** \_src,**int** \_snk) {
83. src = \_src , snk = \_snk ;
84. **int** result = 0;
85. **while** (dinic\_bfs()){
86. CLR(work);
87. **while** (**int** delta = dinic\_dfs(src, INF)) result += delta;
88. }
89. **return** result;
90. }

### Notes

#### Finding the Edges of Minimum Cut

After running max flow, do a DFS from source on the residual graph. Then an edge from u to v will be in the minimum cut if and only if

* u is visited and v is unvisited
* u is unvisited and v is visited

#### Finding Minimum Cut with minimum Number of Edges

Multiply the capacity of each edge by some constant T. Increase all the capacities by one. The minimum cut int the new graph is the minimum cut of the original graph with least number of edges.  
Keeping T greater than the number of edges int the graph is safe enough.

#### Maximum Flow With Demands (Fixing lower bound on edges)

* Create a super source(s’), a super sink(t’). If edge u🡪v has a lower bound of LB,

1. Give an edge from s’ to v with capacity LB.
2. Give an edge from u to t’ with capacity LB.
3. Change the capacity of edge u🡪v to (C – LB)

* Give an edge from normal sink to normal source with capacity infinity (If source and sink are specified in the original network)

If maxflow is equal to sum of LBs , then the lower bound can be satisfied.

Actual flow trough edge u🡪v is = LB + Flow through u🡪v in the modified network

#### Project Selection Problem

* Maximize total profit.
* Doing i'th project profits you by P[i].
* Doing i'th project requires you to buy a list of instruments each of which has different cost.
* Different projects may require the same instrument in which case, buying one instrument is ok.
* Make a flow graph with projects on the left and instruments on the right.
* cap[source][i'th project] = P[i]
* cap[j'th instrument][sink] = Cost[j]
* cap[i'th project][j'th instrument] = inf ( if i'th project requires j'th instrument )

here mincut will minimize this function ( sacrifice profits of projects + cost of instruments to be bought )  
So ans is = Total profit of all projects - mincut.

#### Image Segmentation Problem

There are n pixels. Each pixel i can be assigned a foreground value fi or a background value bi. There is a penalty of pij if pixels i, j are adjacent and have different assignments. The problem is to assign pixels to foreground or background such that the sum of their values minus the penalties is maximum.

Let P be the set of pixels assigned to foreground and Q be the set of points assigned to background, then the problem can be formulated as

* maximize ( totalF + totalB - sacrificeFore for Q - sacrificeBack for P - penalty pij)
* or, minimize( sacrificeFore for Q + sacrificeBack for P + penalty pij)

The above minimization problem can be formulated as a minimum-cut problem by constructing a network where the source is connected to all the pixels with capacity fi, and the sink is connected by all the pixels with capacity bi. Two edges (i, j) and (j, i) with pij capacity are added between two adjacent pixels.  
The s-t cut-set then represents the pixels assigned to the foreground in P and pixels assigned to background in Q.

## Minimum Cost Maximum Flow

1. /// Complexity --> E\*Flow (A lot less actually, not sure)
2. /\*\* Maximizes the flow first, then minimizes the cost
4. The algorithm finds a path with minimum cost to send one unit of flow
5. and sends flow over the path as much as possible. Then tries to find
6. another path in the residual graph.
8. SPFA Technique :
9. The basic idea of SPFA is the same as Bellman Ford algorithm in that each
10. vertex is used as a candidate to relax its adjacent vertices. The improvement
11. over the latter is that instead of trying all vertices blindly, SPFA maintains
12. a queue of candidate vertices and adds a vertex to the queue only if that vertex
13. is relaxed. This process repeats until no more vertex can be relaxed.
14. This doesn't work if there is a negative cycle in the graph \*/
16. **const** **int** MAX = ?; /// maximum number of nodes
17. **const** **int** INF = ?;
19. **int**  N , src , snk , P;
20. **int** dis[MAX] , par[MAX] , mCap[MAX] , pos[MAX];
21. **bool** vis[MAX];
23. **struct** Edge{
24. **int** to, rev\_pos, cap, cost, flow;
25. };
27. vector <Edge> ed[MAX];
29. **void** init(**int** \_N,**int** \_src,**int** \_snk) {
30. N = \_N , src = \_src , snk = \_snk;
31. **for**(**int** i=1;i<=N;i++) ed[i].clear();
32. }
34. **void** addEdge(**int** u,**int** v,**int** cap,**int** cost){
35. Edge a = {v,ed[v].size(),cap,cost,0};
36. Edge b = {u,ed[u].size(),0,-cost,0};
37. ed[u].pb(a);
38. ed[v].pb(b);
39. }
41. **inline** **bool** SPFA(){
42. **int** i,u,v;
43. CLR(vis);
44. **for**(i=1; i<=N; i++) mCap[i] = dis[i] = INF;
45. queue <**int**> q;
46. dis[src] = 0;
47. vis[src] = **true**; // src is in the queue now
48. q.push(src);
50. **while**(!q.empty()){
51. u = q.front();
52. q.pop();
53. vis[u] = **false**; // u is not in the queue now
54. **for**(i=0; i<ed[u].size(); i++){
55. Edge &e = ed[u][i];
56. v = e.to;
57. **if**(e.cap>e.flow && dis[v]>dis[u]+e.cost){
58. dis[v] = dis[u] + e.cost;
59. par[v] = u;
60. pos[v] = i;
61. mCap[v] = min(mCap[u],e.cap-e.flow);
62. **if**(!vis[v]) {
63. vis[v] = **true**;
64. q.push(v);
65. }
66. }
67. }
68. }
69. **return** (dis[snk]!=INF);
70. }
72. **inline** pair <**int**,**int**> MCMF() {
73. **int** F = 0, C = 0, f , u , v;
74. **while**(SPFA()){
75. u = snk;
76. f = mCap[u];
77. F += f;
78. **while**(u!=src){
79. v = par[u];
80. ed[v][pos[u]].flow += f; // edge of v-->u increases
81. ed[u][ed[v][pos[u]].rev\_pos].flow -= f;
82. u = v;
83. }
84. C += dis[snk] \* f;
85. }
86. **return** mp(F,C);
87. }

## Maximum Bipartite Matching

1. /// worst case complexity V\*E
2. #define L   105
3. #define R   105
5. vector <**int**> G[L]; ///The adjacency list for the nodes in the left side
6. **int** matchL[L],matchR[R];
7. **bool** vis[L];
9. **bool** dfs(**int** s) {
10. **int** i,x;
11. vis[s] = **true**;
12. **for**(i = 0;i<G[s].size();i++){
13. x = G[s][i];
14. **if**(matchR[x]==-1 || (!vis[matchR[x]] && dfs(matchR[x]))){
15. matchL[s] = x;
16. matchR[x] = s;
17. **return** 1;
18. }
19. }
20. **return** 0;
21. }
23. /// n = number of nodes in the left side
24. **int** match(**int** n) {
25. **int** i;
26. **bool** done;
27. SET(matchL); SET(matchR);
28. **while**(**true**){
29. CLR(vis);
30. done = **true**;
32. **for**(i=1;i<=n;i++)
33. **if**(matchL[i]==-1 && !vis[i] && dfs(i)) done = **false**;
35. **if**(done) **break**;
36. }
38. **int** cnt = 0; /// number of matches
39. **for**(i=1; i<=n; i++)
40. **if**(matchL[i]!=-1) cnt++;
42. **return** cnt;
43. }

## Posets (Partially Ordered Sets)

A DAG basically. If there is an edge from node u to node v, then node u and node v are related (For example u <= v )

### **Antichain**

An **antichain** is a set of nodes of a graph such that there is no path from any node to another node using the edges of the graph. (Independent set)

### Chain

Any path in the DAG.

### **Mirsky’s Theorem**

For the nodes of A DAG,  
Minimum number of partitions into antichains = Maximum Chain (Longest path in the DAG)

#### Construction

Let’s suppose , L = number of nodes in the longest path.  
Let lp[u] denote the length of the longest path that ends at node u. For a pair of nodes u and v, if lp[u] = lp[v] then we can easily prove that there is no edge between them. There are atmost L distinct values of lp, so we need at most L partitions.

### Dilworth’s Theorem

For the nodes of A DAG,  
Minimum partition into chain = Maximum length anti chain(Maximum Independent Set of a DAG)  
or, Minimum General Path Cover = Maximum length anti chain(Maximum Independent Set of a DAG)

### Konig’s Theorem

A minimum vertex cover of a graph is a minimum set of nodes **S** such that each edge og the graph has at least one end point in **S**.  
Minimum Vertex Cover = Maximum Matching

**Maximum independent set** of a graph is a maximum set of nodes such that there is no path from one node to another of that set (Maximum Length Antichain)

Maximum Independent Set = Total Nodes – Minimum Vertex Cover

## Finding Minimum Vertex Cover

See the min cut solution of the weighted version or do the following.  
Run BPM. After that do a dfs from every **unmatched** node in the left side. Go from u to v if and only if :

* u is on the left side and matchL[u] != v or
* u is on the right side and matchR[u] == v

After this, all unvisited nodes in the left side and the visited nodes in the right side will form a MVC.

See the proof of Konig’s theorem.

The rest of the nodes will form Maximum Independent Set then.

## Node Disjoint Path Cover

Each node belongs to exactly one path  
We can find a minimum node-disjoint path cover by constructing a matching graph where each node of the original graph is represented by two nodes: a left node and a right node. There is an edge from a left node to a right node if there is a such an edge in the original graph.

Minimum node disjoint path cover = Total nodes(N) – maximum matching

Let’s assume we need N paths initially. Each match reduces the number of needed paths by one.

## General Path Covers

A **general path cover** is a path cover where a node can belong to more than one path.

A minimum general path cover can be found almost like a minimum nodedisjoint path cover. It suffices to add some new edges to the matching graph so that there is an edge a! b always when there is a path from a to b in the original graph (possibly through several edges).

## Weighted Minimum Vertex Cover

Sum of the weights of the vertices in the cover should be minimum.  
In the bipartite graph, make all the edges directed from left to right and set their capacity to INF. Define a source node src and a sink node snk.  
Give an edge from src every node in the left side where the capacity of that edge is the weight of that node.  
Give an edge from every node in the right side to snk in the same way.  
Find Min cut(Max Flow) of the graph. This will be the minimum vertex cover weight.

Sum of weights of all the nodes – Min vertex cover = Maximum Independent set with maximum weight

**If all the nodes have weight 1, then this finds the MVC for unweighted version**

**For any graph,**  
**∑Weights of the nodes in MVC + ∑Weights of the nodes in Maximum Independent Set  
= ∑Weights of all the nodes**

## Hungarian Algorithm(Assignment Problem)

1. /\*\* Given a n by m matrix, a call to hungarian() returns
2. minimum/maximum cost of matching
3. Complexity n^3 \*\*/
5. **const** **int** MAX = ?;
6. **const** **int** INF = ?;
8. #define MAXIMIZE +1
9. #define MINIMIZE -1
11. **namespace** wm{ /// hash = 581023
12. **bool** visited[MAX];
13. **int** U[MAX], V[MAX], P[MAX], way[MAX], minv[MAX], match[MAX], ar[MAX][MAX];
15. /// n = number of row and m = number of columns in 1 based, flag = MAXIMIZE or MINIMIZE
16. /// match[i] contains the column to which row i is matched
17. **int** hungarian(**int** n, **int** m, **int** mat[MAX][MAX], **int** flag){
18. CLR(U), CLR(V), CLR(P), CLR(ar), CLR(way);
20. **for** (**int** i = 1; i <= n; i++){
21. **for** (**int** j = 1; j <= m; j++){
22. ar[i][j] = mat[i][j];
23. **if** (flag == MAXIMIZE) ar[i][j] = -ar[i][j];
24. }
25. }
26. **if** (n > m) m = n;
28. **int** i, j, a, b, c, d, r, w;
29. **for** (i = 1; i <= n; i++){
30. P[0] = i, b = 0;
31. **for** (j = 0; j <= m; j++) minv[j] = INF, visited[j] = **false**;
33. **do**{
34. visited[b] = **true**;
35. a = P[b], d = 0, w = INF;
37. **for** (j = 1; j <= m; j++){
38. **if** (!visited[j]){
39. r = ar[a][j] - U[a] - V[j];
40. **if** (r < minv[j]) minv[j] = r, way[j] = b;
41. **if** (minv[j] < w) w = minv[j], d = j;
42. }
43. }
45. **for** (j = 0; j <= m; j++){
46. **if** (visited[j]) U[P[j]] += w, V[j] -= w;
47. **else** minv[j] -= w;
48. }
49. b = d;
50. } **while** (P[b] != 0);
52. **do**{
53. d = way[b];
54. P[b] = P[d], b = d;
55. } **while** (b != 0);
56. }
57. **for** (j = 1; j <= m; j++) match[P[j]] = j;
59. **return** (flag == MINIMIZE) ? -V[0] : V[0];
60. }
61. }

Dominator Tree

### Notes

* A node u will be ancestor of node v in the dominator tree if all the the paths from source to node v contain node u
* If a problem asks for edge disjoint paths, for everu edge, take a new node w and turn the edge (u→v) to (u→w→v) and find node disjoint path now.

1. /\*
2. \* 11 based directed graph input
3. \* g is the edge list of the graph you want to build dominator tree of
4. \* tree is the edge list of the dominator tree
5. to get that we have to call the build() function
6. \* init() function must be called before the start of every test case
8. \* Only the nodes which are reachable from source will be in the dominator tree
9. KEEP THAT IN MIND
10. \*/

13. **const** **int** MAX = 200010;
15. vector<**int**> g[MAX+5],tree[MAX+5],rg[MAX+5],bucket[MAX+5];
17. **int** sdom[MAX+5],par[MAX+5],dom[MAX+5],dsu[MAX+5],label[MAX+5];
18. **int** arr[MAX+5],rev[MAX+5], Time ,n, source;
20. **void** init(**int** \_n, **int** \_source){
21. Time = 0;
22. n = \_n;
23. source = \_source;
24. **for**(**int** i = 1; i<=n; i++) {
25. g[i].clear(), rg[i].clear(), tree[i].clear(), bucket[i].clear();
26. arr[i] = sdom[i] = par[i] = dom[i] = dsu[i] = label[i] = rev[i] = 0;
27. }
28. }
30. **void** dfs(**int** u) {
31. Time++ ;
32. arr[u] = Time;
33. rev[Time] = u;
34. label[Time] = Time;
35. sdom[Time] = Time;
36. dsu[Time] = Time;
37. **int** i,w;
38. **for**(i=0; i<g[u].size(); i++) {
39. w = g[u][i];
40. **if**(!arr[w]) {
41. dfs(w);
42. par[arr[w]] = arr[u];
43. }
44. rg[arr[w]].push\_back(arr[u]);
45. }
46. }
48. **inline** **int** Find(**int** u,**int** x = 0) {
49. **if**(u == dsu[u]) **return** x ? -1 : u;
50. **int** v = Find(dsu[u],x+1);
51. **if**(v<0) **return** u;
52. **if**(sdom[label[dsu[u]]] < sdom[label[u]])
53. label[u] = label[dsu[u]];
54. dsu[u] = v;
55. **return** x ? v : label[u];
56. }
58. //Add an edge u-->v
59. **inline** **void** Union(**int** u,**int** v){
60. dsu[v]=u;
61. }
63. **void** build(){
64. dfs(source);
65. **for**(**int** i=n; i>=1; i--) {
66. **for**(**int** j=0; j<rg[i].size(); j++)
67. sdom[i] = min(sdom[i],sdom[Find(rg[i][j])]);
68. **if**(i>1)bucket[sdom[i]].push\_back(i);
69. **for**(**int** j=0; j<bucket[i].size(); j++) {
70. **int** w = bucket[i][j],v = Find(w);
71. **if**(sdom[v]==sdom[w]) dom[w]=sdom[w];
72. **else** dom[w] = v;
73. }
74. **if**(i>1) Union(par[i],i);
75. }
77. **for**(**int** i=2; i<=n; i++) {
78. **if**(dom[i]!=sdom[i])dom[i]=dom[dom[i]];
79. // comment the following line out if you don't want bidirectional edges in dominator tree
80. tree[rev[i]].push\_back(rev[dom[i]]);
81. tree[rev[dom[i]]].push\_back(rev[i]);
82. }
83. }

DSU on Tree

1. /\*
2. Problem :   Given a tree, every vertex has color. Query is how many
3. vertices in subtree of vertex v are colored with color c
5. A call to dfs(root,-1,0) will process the answer for every query offline
6. Only G needs to be cleared per case
7. \*/

## Using Map

1. // n log n log n
2. map<**int**, **int**> \*cnt[maxn];
3. **void** dfs(**int** v, **int** p){
4. **int** mx = -1, bigChild = -1;
5. **for**(auto u : g[v])
6. **if**(u != p){
7. dfs(u, v);
8. **if**(sz[u] > mx)
9. mx = sz[u], bigChild = u;
10. }
11. **if**(bigChild != -1)
12. cnt[v] = cnt[bigChild];
13. **else**
14. cnt[v] = **new** map<**int**, **int**> ();
15. (\*cnt[v])[ col[v] ] ++;
16. **for**(auto u : g[v])
17. **if**(u != p && u != bigChild){
18. **for**(auto x : \*cnt[u])
19. (\*cnt[v])[x.first] += x.second;
20. }
21. //now (\*cnt[v])[c] is the number of vertices in subtree of vertex v that has color c. You can answer the queries easily.
22. }

## Using Vector

1. // n log n
2. vector<**int**> \*vec[maxn];
3. **int** cnt[maxn];
4. **void** dfs(**int** v, **int** p, **bool** keep){
5. **int** mx = -1, bigChild = -1;
6. **for**(auto u : g[v])
7. **if**(u != p && sz[u] > mx)
8. mx = sz[u], bigChild = u;
9. **for**(auto u : g[v])
10. **if**(u != p && u != bigChild)
11. dfs(u, v, 0);
12. **if**(bigChild != -1)
13. dfs(bigChild, v, 1), vec[v] = vec[bigChild];
14. **else**
15. vec[v] = **new** vector<**int**> ();
16. vec[v]->push\_back(v);
17. cnt[ col[v] ]++;
18. **for**(auto u : g[v])
19. **if**(u != p && u != bigChild)
20. **for**(auto x : \*vec[u]){
21. cnt[ col[x] ]++;
22. vec[v] -> push\_back(x);
23. }
24. //now (\*cnt[v])[c] is the number of vertices in subtree of vertex v that has color c. You can answer the queries easily.
25. // note that in this step \*vec[v] contains all of the subtree of vertex v.
26. **if**(keep == 0)
27. **for**(auto u : \*vec[v])
28. cnt[ col[u] ]--;
29. }

## Heavy Light Trick

1. **const** **int** MAX = 1e5 + 10; // maximum number of nodes
3. vector <**int**> G[MAX]; // adjacency list of the tree
4. **int** sub[MAX]; // subtree size of a node
5. **int** color[MAX]; // color of a node
6. **int** freq[MAX];
7. **int** n;
9. **void** calcSubSize(**int** node,**int** par) {
10. sub[node] = 1;
11. **for**(auto v : G[node]) {
12. **if**(v==par) **continue**;
13. calcSubSize(v,node);
14. sub[node] += sub[v];
15. }
16. }
18. **void** add(**int** node,**int** par,**int** x,**int** bigchild = -1) {
19. freq[color[node]] += x;
20. **for**(auto v : G[node]) {
21. **if**(v==par || v==bigchild) **continue**;
22. add(v,node,x);
23. }
24. }
26. **void** dfs(**int** node,**int** par,**bool** keep) {
27. **int** bigChild = -1;
28. **for**(auto v : G[node]) {
29. **if**(v==par) **continue**;
30. **if**(bigChild==-1 || sub[bigChild] < sub[v] ) bigChild = v;
31. }
33. **for**(auto v : G[node]) {
34. **if**(v==par || v==bigChild) **continue**;
35. dfs(v,node,0);
36. }
38. **if**(bigChild!=-1) dfs(bigChild,node,1);
40. add(node,par,1,bigChild);
42. /// freq[c] now contains the number of nodes in
43. /// the subtree of 'node' that have color  c
44. /// Save the answer for the queries here
46. **if**(keep==0)
47. add(node,par,-1);
48. }
50. **int** main() {
51. input color
52. construct G
54. calcSubSize(root,-1);
55. dfs(root,-1,0);
56. **return** 0;
57. }

Segment Tree

## Point Update, Range Query

1. **int** ara[MAX];
3. **struct** node{
4. **int** sum;
5. }tree[4\*MAX];
7. node Merge(node a,node b){
8. node ret;
9. ret.sum = a.sum+b.sum;
10. **return** ret;
11. }
13. **void** build(**int** n,**int** st,**int** ed){
14. **if**(st==ed){
15. tree[n].sum = ara[st];
16. **return**;
17. }
18. **int** mid = (st+ed)/2;
19. build(2\*n,st,mid);
20. build(2\*n+1,mid+1,ed);
21. tree[n] = Merge(tree[2\*n],tree[2\*n+1]);
22. }
24. **void** update(**int** n,**int** st,**int** ed,**int** id,**int** v){
25. **if**(id>ed || id<st) **return**;
26. **if**(st==ed && ed==id){
27. tree[n].sum = v;
28. **return**;
29. }
30. **int** mid = (st+ed)/2;
31. update(2\*n,st,mid,id,v);
32. update(2\*n+1,mid+1,ed,id,v);
33. tree[n] = Merge(tree[2\*n],tree[2\*n+1]);
34. }
36. node query(**int** n,**int** st,**int** ed,**int** i,**int** j){
37. **if**(st>=i && ed<=j) **return** tree[n];
38. **int** mid = (st+ed)/2;
39. **if**(mid<i) **return** query(2\*n+1,mid+1,ed,i,j);
40. **else** **if**(mid>=j) **return** query(2\*n,st,mid,i,j);
41. **else** **return** Merge(query(2\*n,st,mid,i,j),query(2\*n+1,mid+1,ed,i,j));
42. }

## Range Update, Range Query

1. **int** ara[MAX];
3. **struct** node{
4. **int** sum;
5. }tree[4\*MAX];
7. **int** lazy[4\*MAX];
9. node Merge(node a,node b){
10. node ret;
11. ret.sum = a.sum+b.sum;
12. **return** ret;
13. }
15. **void** lazyUpdate(**int** n,**int** st,**int** ed){
16. **if**(lazy[n]!=0){
17. tree[n].sum += ((ed-st+1)\*lazy[n]);
18. **if**(st!=ed){
19. lazy[2\*n] += lazy[n];
20. lazy[2\*n+1] += lazy[n];
21. }
22. lazy[n] = 0;
23. }
24. }
26. **void** build(**int** n,**int** st,**int** ed){
27. lazy[n] = 0;
28. **if**(st==ed){
29. tree[n].sum = ara[st];
30. **return**;
31. }
32. **int** mid = (st+ed)/2;
33. build(2\*n,st,mid);
34. build(2\*n+1,mid+1,ed);
35. tree[n] = Merge(tree[2\*n],tree[2\*n+1]);
36. }
37. **void** update(**int** n,**int** st,**int** ed,**int** i,**int** j,**int** v){
38. lazyUpdate(n,st,ed);
39. **if**(st>j || ed<i) **return**;
40. **if**(st>=i && ed<=j){
41. lazy[n] += v;
42. lazyUpdate(n,st,ed);
43. **return**;
44. }
45. **int** mid = (st+ed)/2;
46. update(2\*n,st,mid,i,j,v);
47. update(2\*n+1,mid+1,ed,i,j,v);
48. tree[n] = Merge(tree[2\*n],tree[2\*n+1]);
49. }
51. node query(**int** n,**int** st,**int** ed,**int** i,**int** j){
52. lazyUpdate(n,st,ed);
53. **if**(st>=i && ed<=j) **return** tree[n];
54. **int** mid = (st+ed)/2;
55. **if**(mid<i) **return** query(2\*n+1,mid+1,ed,i,j);
56. **else** **if**(mid>=j) **return** query(2\*n,st,mid,i,j);
57. **else** **return** Merge(query(2\*n,st,mid,i,j),query(2\*n+1,mid+1,ed,i,j));
58. }

## How Many Non Zero Elements in the Tree

1. /\*
2. Given an array consisting of all zeroes
3. Update -> Add some value to all the elements of a range
4. (no element ever gets negative value)
5. Query -> How many non zero element in a range
6. \*/
8. **int** tree[MAX];  /// how many non zero elements in this segment
9. **int** lazy[MAX];  /// how many times a node is fully updated
11. **void** lazyUpdate(**int** n,**int** st,**int** ed){
12. **if**(st!=ed){
13. **if**(lazy[n]) tree[n] = ed-st+1;
14. **else** tree[n] = tree[2\*n]+tree[2\*n+1];
15. }
16. **else**{
17. **if**(lazy[n]) tree[n] = ed-st+1;
18. **else** tree[n] = 0;
19. }
20. }
22. **void** build(**int** n,**int** st,**int** ed){
23. lazy[n] = tree[n] = 0;
24. **if**(st==ed) **return**;
25. **int** mid = (st+ed)/2;
26. build(2\*n,st,mid);
27. build(2\*n+1,mid+1,ed);
28. }
30. **void** update(**int** n,**int** st,**int** ed,**int** i,**int** j,**int** v){
31. **if**(st>j || ed<i) **return**;
32. **if**(st>=i && ed<=j){
33. lazy[n] += v;
34. lazyUpdate(n,st,ed);
35. **return**;
36. }
37. **int** mid = (st+ed)/2;
38. update(2\*n,st,mid,i,j,v);
39. update(2\*n+1,mid+1,ed,i,j,v);
40. lazyUpdate(n,st,ed);
41. }
43. **int** query(**int** n,**int** st,**int** ed,**int** i,**int** j){
44. **if**(st>=i && ed<=j) **return** tree[n];
45. **int** mid = (st+ed)/2;
46. **if**(mid<i) **return** query(2\*n+1,mid+1,ed,i,j);
47. **else** **if**(mid>=j) **return** query(2\*n,st,mid,i,j);
48. **else** **return** query(2\*n,st,mid,i,j) + query(2\*n+1,mid+1,ed,i,j);
49. }

Implicit Segment Tree

## Point Update, Range Query

1. **struct** node{
2. **int** sum;
3. node \*left,\*right;
4. node(){}
5. node(**int** value){
6. sum = value;
7. left = right = NULL;
8. }
9. };
11. **void** update(node \*cur,**int** st,**int** ed,**int** id,**int** v)
12. {
13. **if**(id<st || id>ed) **return**;
14. **if**(id==st && id==ed){
15. cur->sum = v;
16. **return**;
17. }
18. **int** mid = (st+ed)/2;
19. **if**(cur->left==NULL) cur->left = **new** node(0);
20. **if**(cur->right==NULL) cur->right = **new** node(0);
21. update(cur->left,st,mid,id,v);
22. update(cur->right,mid+1,ed,id,v);
23. cur->sum = cur->left->sum + cur->right->sum;
24. }
26. **int** query(node \*cur,**int** st,**int** ed,**int** i,**int** j)
27. {
28. **if**(st>=i && ed<=j) **return** cur->sum;
29. **int** mid = (st+ed)/2;
30. **if**(cur->left==NULL) cur->left = **new** node(0);
31. **if**(cur->right==NULL) cur->right = **new** node(0);
32. **if**(mid<i) **return** query(cur->right,mid+1,ed,i,j);
33. **else** **if**(mid>=j) **return** query(cur->left,st,mid,i,j);
34. **else** **return** query(cur->right,mid+1,ed,i,j)+query(cur->left,st,mid,i,j);;
35. }
37. **int** main()
38. {
39. **int** n = 1000000000;
40. node \*root = **new** node(0);
41. update(root,1,n,5,1);
42. update(root,1,n,3,1);
43. cout << query(root,1,n,1,5) << endl;
44. **return** 0;
45. }

## Range Update, Range Query

1. **struct** node{
2. **int** sum,lazy;
3. node \*left,\*right;
4. node(){}
5. node(**int** value){
6. sum = value;
7. lazy = 0;
8. left = right = NULL;
9. }
10. };
12. **void** lazyUpdate(node \*cur,**int** st,**int** ed)
13. {
14. **if**(cur->lazy!=0){
15. cur->sum += ((ed-st+1)\*cur->lazy);
16. **if**(st!=ed){
17. **if**(cur->left==NULL) cur->left = **new** node(0);
18. **if**(cur->right==NULL) cur->right = **new** node(0);
19. cur->left->lazy += cur->lazy;
20. cur->right->lazy += cur->lazy;
21. }
22. cur->lazy = 0;
23. }
24. }
26. **void** update(node \*cur,**int** st,**int** ed,**int** i,**int** j,**int** v){
27. lazyUpdate(cur,st,ed);
28. **if**(st>j || ed<i) **return**;
29. **if**(st>=i && ed<=j){
30. cur->lazy += v;
31. lazyUpdate(cur,st,ed);
32. **return**;
33. }
34. **int** mid = (st+ed)/2;
35. **if**(cur->left==NULL) cur->left = **new** node(0);
36. **if**(cur->right==NULL) cur->right = **new** node(0);
37. update(cur->left,st,mid,i,j,v);
38. update(cur->right,mid+1,ed,i,j,v);
39. cur->sum = cur->left->sum + cur->right->sum;
40. }
42. **int** query(node \*cur,**int** st,**int** ed,**int** i,**int** j){
43. lazyUpdate(cur,st,ed);
44. **if**(st>=i && ed<=j) **return** cur->sum;
45. **int** mid = (st+ed)/2;
46. **if**(cur->left==NULL) cur->left = **new** node(0);
47. **if**(cur->right==NULL) cur->right = **new** node(0);
48. **if**(mid<i) **return** query(cur->right,mid+1,ed,i,j);
49. **else** **if**(mid>=j) **return** query(cur->left,st,mid,i,j);
50. **else** **return** query(cur->right,mid+1,ed,i,j)+query(cur->left,st,mid,i,j);;
51. }
53. **int** main()
54. {
55. **int** n = 1000000000;
56. node \*root = **new** node(0);
57. update(root,1,n,1,5,1);
58. update(root,1,n,4,10,1);
59. update(root,1,n,9,14,1);
60. cout << query(root,1,n,1,20) << endl;
61. **return** 0;
62. }

Persistent Segment Tree

## Point Update Range Query( Static Array )

1. /\*\* Persistent Segment Tree using static Array
2. Point Update , Range Sum
3. Initialize ncnt to 0 in every test case \*\*/
5. **const** **int** MAX = 100010;
7. **int** ncnt = 0;
9. **struct** node {
10. **int** sum;
11. **int** left,right;
12. node() {}
13. node(**int** val) {
14. sum = val;
15. left = right = -1;
16. }
17. } tree[ ? ];
19. /// input araay
20. **int** ara[MAX];
21. /// root nodes for all versions
22. **int** version[MAX];
24. **void** build(**int** n,**int** st,**int** ed) {
25. **if** (st==ed) {
26. tree[n] = node(ara[st]);
27. **return**;
28. }
30. **int** mid = (st+ed) / 2;
32. tree[n].left = ++ncnt;
33. tree[n].right = ++ncnt;
35. build(tree[n].left, st, mid);
36. build(tree[n].right, mid+1, ed);
38. tree[n].sum = tree[tree[n].left].sum + tree[tree[n].right].sum;
39. }
41. **void** update(**int** prev,**int** cur,**int** st,**int** ed,**int** id, **int** val)
42. {
43. **if** (id > ed or id < st) **return**;
44. **if** (st == ed) {
45. tree[cur] = node(val);
46. **return**;
47. }
48. **int** mid = (st+ed) / 2;
49. **if** (id <= mid) {
50. tree[cur].right = tree[prev].right;
51. tree[cur].left = ++ncnt;
52. update(tree[prev].left,tree[cur].left, st, mid, id, val);
53. }
54. **else** {
55. tree[cur].left = tree[prev].left;
56. tree[cur].right = ++ncnt;
57. update(tree[prev].right, tree[cur].right, mid+1, ed, id, val);
58. }
59. tree[cur].sum = tree[tree[cur].left].sum + tree[tree[cur].right].sum;
60. }
62. **int** query(**int** n,**int** st,**int** ed,**int** i,**int** j){
63. **if**(st>=i && ed<=j) **return** tree[n].sum;
64. **int** mid = (st+ed)/2;
65. **if**(mid<i) **return** query(tree[n].right,mid+1,ed,i,j);
66. **else** **if**(mid>=j) **return** query(tree[n].left,st,mid,i,j);
67. **else** **return** query(tree[n].left,st,mid,i,j) + query(tree[n].right,mid+1,ed,i,j);
68. }
70. **int** main() {
71. **int** n,q,l,r,k;
73. sii(n,q);
75. version[0] = ++ncnt;
76. build(version[0],1,n);
78. version[1] = ++ncnt;
79. update(version[0],version[1],1,n,id,val);
81. query(version[0],1,n,id,id);
82. query(version[1],1,n,id,id);
84. **return** 0;
85. }

Binary Indexed Tree

1. /\*
2. Initially the tree array is set to zero
3. Point Update(Adding v to index p)
4. Query returns sum of the range [1,p]
5. \*/
7. **int** tree[10];
9. //n --> size of the array
10. //v --> value to be added to index idx
11. **void** update(**int** n,**int** p,**int** v){
12. **while**( p<=n ) tree[p] += v , p += p & (-p);
13. }
15. //returns sum of range[1,p]
16. **int** query(**int** p){
17. **int** sum = 0;
18. **while**(p > 0) sum += tree[p], p -= p & (-p);
19. **return** sum;
20. }
22. //returns sum of range[l,r]
23. **int** range\_query(**int** l,**int** r){
24. if(l>r) **return 0;**
25. **return** query(r)-query(l-1);
26. }

Mo’s Algorithm

1. // Complexity = nb \* N + bs \* Q
2. // Better to keep input array 0 based
4. **int** bs;//block size
5. **int** ara[MAXN] , cnt[MAXV] , res[MAXQ];
6. **int** ans;
8. **struct** data{
9. **int** l,r,id,bn;
10. **inline** data() {}
11. **inline** data(**int** \_l, **int** \_r, **int** \_id){
12. l = \_l , r = \_r , id = \_id;
13. bn = l / bs;
14. }
16. **inline** **bool** operator < (**const** data& other) **const**{
17. **if** (bn != other.bn) **return** (bn < other.bn);
18. **return** ((bn & 1) ? (r < other.r) : (r > other.r));
19. }
21. } query[MAXQ];
23. **void** Add(**int** id){
24. cnt[ara[id]]++;
25. ///update ans
26. }
28. **void** Remove(**int** id){
29. cnt[ara[id]]--;
30. ///update ans
31. }
33. **void** Mo(**int** q){
34. sort( query , query + q );
35. **int** L = 0, R = 0,l,r;
36. ans = 0;
37. Add(0);
38. **for**(**int** i=0; i<q; i++) {
39. l = query[i].l;
40. r = query[i].r;
42. **while**(L>l) Add(--L);
43. **while**(R<r) Add(++R);
45. **while**(L<l) Remove(L++);
46. **while**(R>r) Remove(R--);
48. res[query[i].id] = ans;
49. }
50. }

Query on Path of A Tree

For some query on the path from node u to node v, let’s suppose p = lca(u,v). We need keep the nodes in an array in dfs order.

## Way 1

The ranges [ st[p] , st[u] ] and [ st[p] , st[u] ] will give the answer. Here st[p] occurs twice. Needs to be handled.

## Way 2

If u == p : [ st[u] , st[v] ]

Else [ en[u] , st[v] ]. Here st[p] is not counted, needs to be handled.

Lowest Common Ancestor

1. // 1 based indexing , n = number of nodes
2. **const** **int** MAX = 100010;
4. **int** lg;
5. **int** L[MAX]; // Depth of a node
6. **int** P[MAX][20]; // P[i][j] denotes (2^j)th parent of node i
8. vector <**int**> ed[MAX];
10. **void** dfs(**int** s,**int** par,**int** l){
11. **int** i,x;
12. L[s] = l;
13. **for**(i=0; i<ed[s].size(); i++){
14. x = ed[s][i];
15. **if**(x!=par){
16. P[x][0] = s;
17. dfs(x,s,l+1);
18. }
19. }
20. }
22. **void** lca\_build(**int** n,**int** root){
23. SET(P);
25. dfs(root,-1,0);
27. lg = (log(n)/log(2.0))+2;
29. **int** i,j;
30. **for**(j=1; (1<<j)<=n; j++)
31. **for**(i=1; i<=n; i++)
32. **if**(P[i][j-1]!=-1) P[i][j] = P[P[i][j-1]][j-1];
33. }
35. **inline** **int** lca\_query(**int** x,**int** y){
36. **if**(L[x]<L[y]) swap(x,y);
37. **int** i,j;
38. **for**(i=lg; i>=0; i--)
39. **if**(L[x] - (1<<i) >= L[y]) x = P[x][i];
41. **if**(x==y) **return** x;
42. **for**(i=lg; i>=0; i--) {
43. **if**(P[x][i]!=-1 && P[x][i]!=P[y][i]) {
44. x = P[x][i];
45. y = P[y][i];
46. }
47. }
48. **return** P[x][0];
49. }

Trie

## Static Array Implementation

1. #define N       200000 // total number of characters given as input
2. #define S       26
4. **int** root,now;
5. **int** nxt[N][S], cnt[N];
7. // will be called from main
8. **void** init(){
9. root = now = 1;
10. CLR(nxt),CLR(cnt);
11. }
13. **inline** **int** scale(**char** ch) { **return** (ch - 'a'); }
15. **inline** **void** Insert(**char** s[],**int** sz){
16. **int** cur = root, to;
17. **for**(**int** i=0 ; i< sz ; i++){
18. to = scale(s[i]) ;
19. **if**( !nxt[cur][to] ) nxt[cur][to] = ++now;
20. cur = nxt[cur][to];
21. }
22. cnt[cur]++;
23. }
25. **inline** **bool** Find(**char** s[],**int** sz){
26. **int** cur = root, to;
27. **for**(**int** i=0 ; i<sz ; i++){
28. to = scale(s[i]) ;
29. **if**( !nxt[cur][to] ) **return** **false**;
30. cur = nxt[cur][to];
31. }
32. **return** (cnt[cur]!=0);
33. }
35. /// It's better to call the Delete() after checking if the
36. /// string we wanna delete actually exists in the trie
37. **inline** **void** Delete(**char** s[],**int** sz){
38. **int** cur = root, to;
39. **for**(**int** i=0 ; i<sz ; i++){
40. to = scale(s[i]) ;
41. cur = nxt[cur][to];
42. }
43. cnt[cur]--;
44. }

## Finding Maximum and Minimum Xor Match

1. // Find an integer x in the trie such than n^x is minimized
2. **int** FindMinMatch(**int** n,**int** nob){
3. **int** ret = 0;
4. **int** cur = root , to;
5. **for**(**int** i=nob ; i>=0 ; i--){
6. ret <<= 1;
7. to = checkBit(n,i);
8. **if**(nxt[cur][to]){
9. ret += to;
10. cur = nxt[cur][to];
11. }
12. **else**{
13. ret += (!to);
14. cur = nxt[cur][!to];
15. }
16. }
17. **return** ret;
18. }
19. // Find an integer x in the trie such than n^x is maximized
20. **int** FindMaxMatch(**int** n,**int** nob){
21. **int** ret = 0;
22. **int** cur = root , to;
23. **for**(**int** i=nob ; i>=0 ; i--){
24. ret <<= 1;
25. to = checkBit(n,i);
26. **if**(nxt[cur][!to]){
27. ret += (!to);
28. cur = nxt[cur][!to];
29. }
30. **else**{
31. ret += (to);
32. cur = nxt[cur][to];
33. }
34. }
35. **return** ret;
36. }

Heavy Light Decomposition

1. /\*Code of Lightoj-1348 : Aladdin and the Return Journey\*/
2. **int** n;
3. **int** chainNo , chainHead[MAX] , chainId[MAX] ;
4. **int** it , baseArray[MAX] , posInBaseArray[MAX] ;
5. **int** subtreeSz[MAX] ;
6. **int** ara[MAX] ;
7. vector <**int**> ed[MAX] ;
9. **int** level[MAX]; // Depth of a node
10. **int** parent[MAX];
12. **void** dfs(**int** s,**int** par,**int** l) {
13. **int** i,x;
14. level[s] = l;
15. subtreeSz[s] = 1;
16. **for**(i=0;i<ed[s].size();i++){
17. x = ed[s][i];
18. **if**(x!=par){
19. parent[x] = s;
20. dfs(x,s,l+1);
21. subtreeSz[s] += subtreeSz[x];
22. }
23. }
24. }
26. /\* Segment tree here \*/
28. **void** HLD(**int** s,**int** p){
29. it++;
30. posInBaseArray[s] = it;
31. baseArray[it] = ara[s];
32. **if**(chainHead[chainNo]==-1) chainHead[chainNo] = s;
33. chainId[s] = chainNo;
35. **int** heavyChild = -1, heavyChildSz = 0;
36. **int** i,x;
37. **for**(i=0;i<ed[s].size();i++){
38. x = ed[s][i];
39. **if**(x!=p && subtreeSz[x]>heavyChildSz){
40. heavyChildSz = subtreeSz[x];
41. heavyChild = x;
42. }
43. }
44. **if**(heavyChild!=-1) HLD(heavyChild,s);
45. **for**(i=0;i<ed[s].size();i++){
46. x = ed[s][i];
47. **if**(x!=p && x!=heavyChild){
48. chainNo++; HLD(x,s);
49. }
50. }
51. }
52. **void** HLDConstruct(){
53. it = 0;
54. chainNo = 1;
55. SET(chainHead);
56. parent[1] = -1;
57. dfs(1,-1,0);
58. HLD(1,-1);
59. build(1,1,n);
60. }
62. **int** LCA(**int** u,**int** v) {
63. **while**(chainId[u] != chainId[v]){
64. **if**(level[chainHead[chainId[u]]] < level[chainHead[chainId[v]]]) v = parent[chainHead[chainId[v]]];
65. **else** u = parent[chainHead[chainId[u]]];
66. }
67. **if**(level[u]<level[v]) **return** u;
68. **else** **return** v;
69. }
71. // path from u to v ( level[u] > level[v] )
72. **int** call(**int** u,**int** v) {
73. **int** ret = 0,l,r,head;
74. **while**(**true**){
75. l = posInBaseArray[v];
76. **if**(chainId[u]!=chainId[v]){
77. head = chainHead[chainId[u]];
78. l = posInBaseArray[head];
79. }
80. r = posInBaseArray[u];
81. ret += query(1,1,n,l,r).sum;
82. **if**(chainId[u]==chainId[v]) **return** ret;
83. u = parent[head];
84. }
85. }
86. **int** getResult(**int** u,**int** v){
87. **int** lca = LCA(u,v); **int** ret = 0;
88. **return** ret = call(u,lca) + call(v,lca) - baseArray[posInBaseArray[lca]];
89. }
90. **void** updateNode(**int** id, **int** v){
91. id = posInBaseArray[id];
92. update(1,1,n,id,v);
93. }
94. **int** main() {
95. // edge list input
96. // ara input
97. HLDConstruct();
98. **return** 0;
99. }

Treap

## Normal Treap

1. **struct** node{
2. **int** prior; // Heap value
3. **int** val; // BST value
4. **int** sz; // Subtree Size
5. **int** sum; // This bst maintains the sum of it's child nodes
6. **struct** node \*l,\*r,\*p;
7. };
9. **typedef** node\* pnode;
11. pnode Treap;
13. **int** get\_sz(pnode t) { **return** t?t->sz:0; }
14. **int** get\_sum(pnode t) { **return** t?t->sum:0; }
16. **void** update(pnode t){
17. **if**(!t) **return**;
18. **if**(t->l) t->l->p = t;
19. **if**(t->r) t->r->p = t;
20. t->sz = get\_sz(t->l) + 1 + get\_sz(t->r);
21. t->sum = get\_sum(t->l) + 1 + get\_sum(t->r);
22. }
24. pnode init(**int** val){
25. pnode ret = (pnode)malloc(**sizeof**(node));
26. ret->val = val;
27. ret->sz = 1;
28. ret->prior=rand();
29. ret->p = ret->l = ret->r = NULL;
30. **return** ret;
31. }
33. // l contains the nodes having BST value less than val, r contains the rest
34. **void** split(pnode t,pnode &l,pnode &r,**int** val){
35. **if**(!t) l = r = NULL;
36. **else** **if**(t->val<=val) split(t->r,t->r,r,val) , l = t ;
37. **else** split(t->l,l,t->l,val) , r = t ;
38. update(t);
39. }
41. **void** Merge(pnode &t,pnode l,pnode r){
42. **if**(!l || !r) t = l ? l : r;
43. **else** **if**(l->prior > r->prior) Merge(l->r,l->r,r), t = l ;
44. **else** Merge(r->l,l,r->l), t = r ;
45. update(t);
46. }
48. // Inserting a new node into BST
49. **void** Insert(pnode &t,pnode it){
50. **if**(!t) t = it ;
51. **else** **if**(it->prior>t->prior) split(t,it->l,it->r,it->val) , t = it ;
52. **else** **if**(t->val<=it->val) Insert(t->r,it);
53. **else** Insert(t->l,it);
54. update(t);
55. }
56. //or
57. **void** Insert(pnode &t,pnode it){
58. **if**(!t) t = it;
59. pnode l,r;
60. split(t,l,r,it->val);
61. Merge(t,l,it);
62. Merge(t,t,r);
63. }
65. // Removing a node having BST value = val
66. **void** Remove(pnode &t,**int** val){
67. **if**(!t)**return**;
68. **else** **if**(t->val==val){
69. pnode temp=t;
70. Merge(t,t->l,t->r);
71. free(temp);
72. }
73. **else** **if**(t->val<val) Remove(t->r,val);
74. **else** Remove(t->l,val);
75. update(t);
76. }
77. **void** Delete(pnode &t){
78. **if**(!t) **return**;
79. **if**(t->l) Delete(t->l);
80. **if**(t->r) Delete(t->r);
81. **delete**(t);
82. t = NULL;
83. }

## Implicit Treap

1. /\*\*Treap as Interval Tree(1 based) With Insert and Remove Operation\*\*/
3. **typedef** **struct** node{
4. **int** prior,sz;
5. **int** val; //value stored in the array
6. **int** sum; //whatever info you want to maintain in segtree for each node
7. **int** lazy; //whatever lazy update you want to do
8. **struct** node \*l,\*r,\*p;
9. } node;
11. **typedef** node\* pnode;
12. pnode Treap;
13. **int** get\_sz(pnode t){ **return** t?t->sz:0; }
14. **int** get\_sum(pnode t){ **return** t?t->sum:0; }
16. **void** lazyUpdate(pnode t){
17. **if**(!t || !t->lazy)**return**;
18. t->val += t->lazy;
19. t->sum += t->lazy\*get\_sz(t);
20. **if**(t->l) t->l->lazy += t->lazy;
21. **if**(t->r) t->r->lazy += t->lazy;
22. t->lazy=0;
23. }
25. //operation of segtree and size,parent update
26. **void** operation(pnode t) {
27. **if**(!t)**return**;
28. lazyUpdate(t->l); lazyUpdate(t->r); //imp:propagate lazy before combining t->l,t->r;
29. t->sz=get\_sz(t->l)+1+get\_sz(t->r);
30. t->sum = get\_sum(t->l) + t->val + get\_sum(t->r); // updateing sum
31. **if**(t->l) t->l->p = t;
32. **if**(t->r) t->r->p = t;
33. }
35. // The subarray[1:pos] is saved in node l, the rest in r
36. **void** split(pnode t,pnode &l,pnode &r,**int** pos,**int** add=0){
37. **if**(!t) **return** **void**( l = r = NULL) ;
38. lazyUpdate(t);
39. **int** curr\_pos = add + get\_sz(t->l)+1;
40. **if**(curr\_pos<=pos) split(t->r,t->r,r,pos,curr\_pos),l=t;
41. **else** split(t->l,l,t->l,pos,add),r=t;
42. operation(t);
43. }
45. **void** Merge(pnode &t,pnode l,pnode r){
46. lazyUpdate(l); lazyUpdate(r);
47. **if**(!l || !r) t = l?l:r;
48. **else** **if**(l->prior>r->prior) Merge(l->r,l->r,r) , t = l ;
49. **else**    Merge(r->l,l,r->l) , t = r ;
50. operation(t);
51. }
53. pnode init(**int** val){
54. pnode ret = (pnode)malloc(**sizeof**(node));
55. ret->prior = rand();
56. ret->sz = 1;
57. ret->val = ret->sum = val;
58. ret->lazy = 0;
59. ret->p = ret->l = ret->r = NULL;
60. **return** ret;
61. }
62. //changes the value of the node at position id to val
63. **void** point\_update(pnode &t,**int** id,**int** val){
64. pnode L,mid,R;
65. split(t,L,mid,id-1);
66. split(mid,t,R,1);
67. t->val = val;
68. Merge(mid,L,t);
69. Merge(t,mid,R);
70. }
72. //deletes the node at position id
73. **void** Remove(pnode &t,**int** id){
74. pnode L,mid,R,X;
75. split(t,L,mid,id-1);
76. split(mid,X,R,1);
77. **delete** X;
78. Merge(t,L,R);
79. }
81. //inserts a node at position id having array value = val
82. **void** Insert(pnode &t,**int** id,**int** val){
83. pnode L,R,mid;
84. pnode it = init(val);
85. split(t,L,R,id-1);
86. Merge(mid,L,it);
87. Merge(t,mid,R);
88. }
90. **int** range\_query(pnode t,**int** l,**int** r){
91. pnode L,mid,R;
92. split(t,L,mid,l-1);
93. split(mid,t,R,r-l+1);
94. **int** ans = t->sum;
95. Merge(mid,L,t);
96. Merge(t,mid,R);
97. **return** ans;
98. }
100. **void** range\_update(pnode t,**int** l,**int** r,**int** val){
101. pnode L,mid,R;
102. split(t,L,mid,l-1);
103. split(mid,t,R,r-l+1);
104. t->lazy += val;
105. Merge(mid,L,t);
106. Merge(t,mid,R);
107. }
109. **void** Delete(pnode &t){
110. **if**(!t) **return**;
111. **if**(t->l) Delete(t->l);
112. **if**(t->r) Delete(t->r);
113. **delete**(t);
114. t = NULL;
115. }
117. **int** ara[10];
119. **int** main(){
120. //creating a treap to use it as an interval tree of ara (1 based)
121. **int** n = 10;
122. **for**(**int** i=1; i<=n; i++){
123. **if**(i==1) Treap = init(ara[i]);
124. **else** Merge(Treap,Treap,init(ara[i]));
125. }
126. Delete(Treap);  //Deleting when work done
127. **return** 0;
128. }

## Maximum Contiguous Sum Merging

1. // Maximum contiguous sum merging
2. **void** operation(pnode t){
3. **if**(!t)**return**;
4. t->sum = get\_sum(t->l) + t->val + get\_sum(t->r);
5. t->res = max( max(get\_res(t->l), get\_res(t->r)), max(0, get\_rsum(t->l)) + t->val + max(0, get\_lsum(t->r)));
6. t->lsum = max(max(0,get\_lsum(t->r)) + t->val + get\_sum(t->l),get\_lsum(t->l));
7. t->rsum = max(get\_sum(t->r) + t->val + max(0,get\_rsum(t->l)),get\_rsum(t->r));
8. }

Matrix Exponentiation

1. **struct** matrix{
2. **ll** mat[100][100];
3. **int** dim;
4. matrix(){};
5. matrix(**int** d){
6. dim = d;
7. **for**(**int** i=0;i<dim;i++)
8. **for**(**int** j=0;j<dim;j++)
9. mat[i][j] = 0;
10. }
11. // mat = mat \* mul
12. matrix operator \*(**const** matrix &mul){
13. matrix ret = matrix(dim);
14. **for**(**int** i=0;i<dim;i++){
15. **for**(**int** j=0;j<dim;j++){
16. **for**(**int** k=0;k<dim;k++){
17. ret.mat[i][j] += (mat[i][k])\*(mul.mat[k][j]) ;
18. ret.mat[i][j] %= MOD ;
19. }
20. }
21. }
22. **return** ret ;
23. }
24. matrix operator + (**const** matrix &add){
25. matrix ret = matrix(dim);
26. **for**(**int** i=0;i<dim;i++){
27. **for**(**int** j=0;j<dim;j++){
28. ret.mat[i][j] = mat[i][j] + add.mat[i][j] ;
29. ret.mat[i][j] %= MOD ;
30. }
31. }
32. **return** ret ;
33. }
34. matrix operator ^(**int** p){
35. matrix ret = matrix(dim);
36. matrix m = \***this** ;
37. **for**(**int** i=0;i<dim;i++) ret.mat[i][i] = 1 ; //identity matrix
38. **while**(p){
39. **if**( p&1 ) ret = ret \* m ;
40. m = m \* m ;
41. p >>= 1 ;
42. }
43. **return** ret ;
44. }
45. };

Extended Euclid (Solving Linear Diophantine Equation)

1. /\*
2. c = gcd(a,b);
4. ax + by = c;
5. (bq + r)x + by = c;
6. bqx + rx + by = c;
7. b(qx + y) + rx = c;
8. bx' + ry' = c;      [r = a % b]
10. We get,
11. x' = qx + y;
12. y' = x
14. So,
15. y = x' - qx;
16. y = x' - qy';       [y' = x]
17. and
18. x = y'
20. If c is not the gcd then,
22. actual x = x \* (c/gcd)
23. actual y = y \* (c/gcd)
24. But if gcd doesn't divide c, there is no solution.
25. \*/

28. // returns (x,y) for ax + by = gcd(a,b)
29. // keep in mind that if a or b or both are negative,  gcd(a,b) will be negative
31. PLL extEuclid(ll a,ll b)
32. {
33. **if**(b==0LL) **return** mp(1LL,0LL);
34. PLL ret,got;
35. got = extEuclid(b,a%b);
36. ret = mp(got.yy,got.xx-(a/b)\*got.yy);
37. **return** ret;
38. }
40. /\*
41. From one solution (x0,y0), we can obtain all the solutions of the given equation.
42. Let g = gcd(a,b) and let x0,y0 be integers which satisfy the following:
43. a\*x0+b\*y0 = c
44. Now, we should see that adding b/g to x0 and at the same time subtracting a/g
45. from y0 will not break the equality:
47. a\*(x0 + b/g) + b\*(y0 - a/g)
48. = a\*x0 + b\*y0 + (a\*b)/g - (b\*a)/g
49. = c
50. Obviously, this process can be repeated again, so all the numbers of the form:
52. x = x0 + k \* (b/g)
53. y = y0 - k \* (a/g)
54. are solutions of the given Diophantine equation.
56. In the solution returned by extEuclid :
57. |x| and |y| is minimized
58. |x| <= b/2g
59. |y| <= a/2g
60. Because we get a new x after every b/g amount of jump
61. and we get a new y after every a/g amount of jump
63. Solution with minimum (x+y):
64. x + y = x0 + y0 + k\*(b/g - a/g)
65. x + y = x0 + y0 + k\*((b-a)/g)
67. If b>a, we need to find the k with the minimum value
68. else we need to find the k with the maximum value
69. \*/
71. // Iterative Implementation
72. PLL extEuclid(ll a,ll b){
73. ll s = 1,t = 0,st = 0,tt = 1;
74. **while**(b) {
75. s = s - (a/b)\*st;
76. swap(s,st);
77. t = t - (a/b)\*tt;
78. swap(t,tt);
79. a = a % b;
80. swap(a,b);
81. }
82. **return** mp(s,t);
83. }
84. // returns number of solutions for the equation ax + by = c
85. // where minx <= x <= maxx and miny <= y <= maxy
86. ll numberOfSolutions(ll a,ll b,ll c,ll minx,ll maxx,ll miny,ll maxy)
87. {
88. **if**(a==0 && b==0){
89. **if**(c!=0) **return** 0;
90. **else** **return** (maxx-minx+1)\*(maxy-miny+1); // all possible (x,y) within the ranges can be a solution
91. }
93. ll gcd = \_\_gcd(a,b);
94. **if**(c%gcd!=0) **return** 0;// no solution , gcd(a,b) doesn't divide c
96. // If b==0, x will be fixed, any y in the range can form a pair with that x
97. **if**(b==0){
98. c /= a;
99. **if**(c>=minx && c<=maxx) **return** maxy-miny+1;
100. **else** **return** 0;
101. }
103. // If a==0, x will be fixed, any x in the range can form a pair with that y
104. **if**(a==0){
105. c /= b;
106. **if**(c>=miny && c<=maxy) **return** maxx-minx+1;
107. **else** **return** 0;
108. }
110. // gives a particular solution to the equation ax + by = gcd(a,b) {gcd(a,b) can be negative also}
111. PLL sol = extEuclid(a,b);
113. a /= gcd;
114. b /= gcd;
115. c /= gcd;
117. ll x,y;
118. x = sol.xx\*c;
119. y = sol.yy\*c;
121. ll lx,ly,rx,ry;
123. // lx -> minimum value of k such that sol.xx + k \* (b/g) is in range[minx,maxx]
124. // rx -> maximum value of k such that sol.xx + k \* (b/g) is in range[minx,maxx]
125. **if**(x<minx) lx = ceil( (minx-x) / (**double**)abs(b) );
126. **else** lx = -floor( (x-minx) / (**double**)abs(b) );
128. **if**(x<maxx) rx = floor((maxx-x) / (**double**)abs(b) );
129. **else** rx = -ceil((x-maxx) / (**double**)abs(b) );
131. // Doing this I because I ignored sign of b before passing to getCeil/getFloor
132. **if**(b<0){
133. lx \*= -1;
134. rx \*= -1;
135. swap(lx,rx);
136. }
137. **if**(lx>rx) **return** 0;
139. // ly -> minimum value of k such that sol.yy - k \* (a/g) is in range[miny,maxy]
140. // ry -> maximum value of k such that sol.yy - k \* (a/g) is in range[miny,maxy]
141. **if**(y<miny) ly = ceil( (miny-y) / (**double**)abs(a) );
142. **else** ly = -floor( (y-miny) / (**double**)abs(a) );
144. **if**(y<maxy) ry = floor( (maxy-y) / (**double**)abs(a) );
145. **else** ry = -ceil( (y-maxy) / (**double**)abs(a) );
147. // Doing this because I ignored sign of a before passing to getCeil/getFloor
148. **if**(a<0){
149. ly \*= -1;
150. ry \*= -1;
151. swap(ly,ry);
152. }
153. **if**(ly>ry) **return** 0;
155. ly \*= -1;
156. ry \*= -1;
157. swap(ly,ry);
159. // getting the intersection between (x range) and (y range) of k
160. ll li = max(lx,ly);
161. ll ri = min(rx,ry);
163. **return** max( ri - li + 1 , 0LL );
164. }

Discrete Logarithm

1. /// returns (a^b) % m
2. ll bigMod(ll a,ll b,ll m){
3. ll ret = 1LL;
4. a %= m;
5. **while** (b){
6. **if** (b & 1LL) ret = (ret \* a) % m;
7. a = (a \* a) % m;
8. b >>= 1LL;
9. }
10. **return** ret;
11. }
13. PLL extEuclid(ll a,ll b) {
14. **if**(b==0LL) **return** make\_pair(1LL,0LL);
15. PLL ret,got;
16. got = extEuclid(b,a%b);
17. ret = make\_pair(got.yy,got.xx-(a/b)\*got.yy);
18. **return** ret;
19. }

22. /// returns modular invers of a with respect to m
23. /// inverse exists if and only if a and m are co-prime
24. ll modularInverse(ll a, ll m){
25. ll x, y, inv;
26. PLL sol = extEuclid(a,m);
27. inv = (sol.xx + m) % m;
28. **return** inv;
29. }
31. /\*\*\*
32. \* returns smallest x such that (g^x) % p = h, -1 if none exists
33. \* p must be a PRIME ( gcd(g,p)=1 should be enough :/ )
34. \* function returns x, the discrete log of h with respect to g modulo p

37. \* g^x = h (mod p)
38. \* g^(mq+r) = h (mod p)
39. \* g^mq \* g^r = h mod(p)
40. \* g^r = h \* ((g^-1)^m)^q (mod p)
42. \* we will precompute all possible (g^r % p) and store the values in a map (value-->r)
43. \* for every q from 0 to m, we will find corresponding r in a map
45. \*\*\*/
46. ll discrete\_log(ll g, ll h, ll p){
47. **if** (h >= p) **return** -1LL;
48. **if** ( ( g % p) == 0LL ){
49. /// return -1 if strictly positive integer solution is required
50. **if** ( h == 1LL ) **return** 0;
51. **else** **return** -1;
52. }
54. unordered\_map <ll, ll> mp;
55. ll i, q, r, m = ceil(sqrt(p));
56. ll d = 1LL, inv = bigMod(modularInverse(g, p), m, p);
58. **for** (r = 0; r <= m; r++){
59. **if** (mp.find(d)!=mp.end()) mp[d] = r ;
60. d \*= g;
61. **if** (d >= p) d %= p;
62. }
64. d = h;
65. **for** (q = 0; q <= m; q++){
66. **if**(mp.find(d)!=mp.end()) {
67. r = mp[d];
68. **return** (m \* q) + r;
69. }
70. d \*= inv;
71. **if** (d >= p) d %= p;
72. }
73. **return** -1LL;
74. }

Fraction Class

1. //a --> numerator
2. //b --> denominator
4. /\*
5. Every fraction must be reduced before passing
6. Fraction f2(-3,7); // declaration process
7. \*/
9. **struct** Fraction{
10. ll a,b;
11. Fraction(){}
12. Fraction(ll \_a,ll \_b){
13. a = \_a;
14. b = \_b;
15. }
16. };
18. ll GCD(ll a,ll b){
19. **if**(b==0) **return** a;
20. **else** **return** GCD(b,a%b);
21. }
23. ll LCM(ll a,ll b){
24. ll ret;
25. ll gcd = GCD(a,b);
26. ret = a/gcd;
27. ret \*= b;
28. **return** ret;
29. }
31. Fraction Reduce(Fraction f){
32. ll gcd = GCD(f.a,f.b);
33. f.a /= gcd;
34. f.b /= gcd;
35. **return** f;
36. }
37. Fraction Add(Fraction x,Fraction y){
38. Fraction sum;
40. sum.b = LCM(x.b,y.b);
41. sum.a = 0;
42. sum.a += (sum.b/x.b)\*x.a;
43. sum.a += (sum.b/y.b)\*y.a;
45. **return** Reduce(sum);
46. }
48. Fraction Sub(Fraction x,Fraction y){
49. y.a \*= -1;
50. **return** Add(x,y);
51. }


55. Fraction Mul(Fraction x,Fraction y){
56. Fraction prod;
58. ll gcd = GCD(x.a,y.b);
59. x.a /= gcd;
60. y.b /= gcd;
62. gcd = GCD(y.a,x.b);
63. y.a /= gcd;
64. x.b /= gcd;
66. prod.a = x.a\*y.a;
67. prod.b = x.b\*y.b;
69. **return** Reduce(prod);
70. }
72. Fraction Div(Fraction x,Fraction y){
73. Fraction ret;
74. swap(y.a,y.b);
75. **return** Mul(x,y);
76. }

Gaussian Elimination

1. #define SZ 105
2. #define EPS 1e-8
3. **double** mat[SZ][SZ]; // Augmented Matrix
4. **int** where[SZ]; // where[i] denotes the row index of the pivot element of column i
5. **double** ans[SZ];
7. /// n for row, m for column
8. **int** Gauss(**int** n,**int** m)
9. {
10. SET(where);
11. **for**(**int** row=0,col=0;col<m && row<n;col++){
12. **int** max\_row = row;
13. **for**(**int** i=row;i<n;i++)
14. **if**( abs(mat[i][col]) > abs(mat[max\_row][col]) ) max\_row = i;
16. **if**( abs(mat[max\_row][col]) < EPS ) **continue**;
18. **for**(**int** i=col;i<=m;i++) swap(mat[row][i],mat[max\_row][i]);
20. where[col] = row;
22. **double** mul;
23. **for**(**int** i=row+1;i<n;i++){
24. **if**(abs(mat[i][col])>EPS){
25. mul = mat[i][col]/mat[row][col];
26. **for**(**int** j=col;j<=m;j++) mat[i][j] -= mul\*mat[row][j];
27. }
28. }
29. row++;
30. }
32. // checking 0 row
33. **double** sum;
34. **for**(**int** i=0;i<n;i++){
35. sum = 0;
36. **for**(**int** j=0;j<m;j++) sum += abs(mat[i][j]);
37. **if**( abs(sum) < EPS && abs(mat[i][m]) > EPS ) **return** 0; //no solution
38. }
40. // back substitution
41. **double** sltn;
42. **int** cur;
43. **for**(**int** i=m-1;i>=0;i--){
44. //if(where[i] == -1)  return INF; // infinitely many solutions
45. sltn = mat[where[i]][m];
46. cur = where[i];
47. **for**(**int** j = i+1; j<m; j++)
48. sltn -= mat[cur][j]\*ans[j];
49. ans[i] = sltn/mat[cur][i];
50. }
52. **return** 1; // unique solution
53. }

Gauss-Jordan Elimination

1. #define SZ 105
2. **double** mat[SZ][SZ]; // Augmented Matrix
3. **int** where[SZ]; // where[i] denotes the row index of the pivot element of column i
4. **double** ans[SZ];
6. // n for row, m for column
7. **int** GaussJordan(**int** n,**int** m)
8. {
9. SET(where);
10. **for**(**int** row=0,col=0; col<m && row<n; col++){
11. **int** max\_row = row;
12. **for**(**int** i=row; i<n; i++)
13. **if**( abs(mat[i][col]) > abs(mat[max\_row][col]) ) max\_row = i;
15. **if**( abs(mat[max\_row][col]) < EPS ) **continue**;
17. **for**(**int** i=col; i<=m; i++) swap(mat[row][i],mat[max\_row][i]);
19. where[col] = row;
21. **double** mul;
22. **for**(**int** i=0; i<n; i++)
23. **if**( i!=row && abs(mat[i][col])>EPS){
24. mul = mat[i][col]/mat[row][col];
25. **for**(**int** j=col; j<=m; j++) mat[i][j] -= mul\*mat[row][j];
26. }
27. row++;
28. }
29. **for**(**int** i=0; i<m; i++)
30. **if**(where[i]!=-1)
31. ans[i] = mat[where[i]][m]/mat[where[i]][i];
33. **double** sum;
34. **for**(**int** i=0; i<n; i++){
35. sum = 0;
36. **for**(**int** j=0; j<m; j++) sum += ans[j] \* mat[i][j];
37. **if**( abs(sum - mat[i][m]) > EPS ) **return** 0;  // no solution
38. }
40. **for**(**int** i=0; i<m; i++)
41. **if** (where[i]==-1) **return** INF;  // Infinitely many solutions
43. **return** 1;  // unique solution
44. }

Gauss Related Problem

1. /\*
2. Problem :   Given a set of numbers, Find a subset such that the xor of the elements
3. of the subset is as large as possible
4. Idea    :   We will try to make the MSB of the result 1 first, then the next bit
5. There will be an equation for every bit
6. if the numbers are 1101,0010,1010 then the equation form MSB is
7. 1\*ans[0] + 0\*ans[1] + 1\*ans[2] = 1
8. \*/
10. #define MAX         105
11. ll ara[MAX];
12. bitset <MAX> mat[70];
14. **int** row,ans[MAX],where[MAX];
16. // bn = bit number
17. // n = number of columns
18. // val = the target value of the bn'th bit of the result
19. **void** add(**int** bn,**int** val,**int** n){
20. ++row;
21. mat[row][MAX-1] = val;
22. //Stores the bn'th bit of every number in the matrix
23. **for**(**int** col=0; col<n; col++) mat[row][col]=( (ara[col]>>bn) & 1 );
24. // If this column has a pivot entry, we will xor the row with the row containg pivot entry
25. **for**(**int** col=0; col<n; col++){
26. **if**(mat[row][col]){
27. **if**(where[col]) mat[row] ^= mat[where[col]];
28. **else** **break**;
29. }
30. }
31. // Setting the pivot
32. **for**(**int** col=0; col<n; col++){
33. **if**(mat[row][col]){
34. where[col]=row;
35. **return**;
36. }
37. }
38. // If no pivot element in the row, the equation is not added
39. --row;
40. }
42. **void** solve(**int** n,**int** m){
43. CLR(where);
44. row = 0;
45. **for**(**int** i=m;i>=0;i--){
46. add(i,1,n); // Trying to keep the i'th bit 1 of theresult
47. }
48. // Back Substitution
49. **for**(**int** i=n-1;i>=0;i--){
50. **if**(mat[where[i]][MAX-1]){
51. ans[i] = 1;
52. **for**(**int** j=1;j<=row;j++){
53. **if**(mat[j][i]) mat[j].flip(MAX-1);
54. }
55. }
56. **else** ans[i] = 0;
57. }
59. }
61. **int** main(){
62. **int** n;
63. ll tot;
65. scanf("%d",&n);
66. tot = 0;
67. **for**(**int** i=0;i<n;i++){
68. scanf("%lld",&ara[i]);
69. tot ^= ara[i];
70. }
71. solve(n,63);
72. ll res = 0;
73. **for**(**int** i=0; i<n; i++) res ^= (ara[i]\*ans[i]);
74. printf("%lld\n",res);
75. }

Multiplying Two Polynomials

## Notes

* Karatsuba, FFT, NTT they all do the same thing.
* Karatsuba, NTT uses LL where fft uses doubles.
* So FFT might face loss of precision but karatsuba/NTT won't face that ever.
* Karatsuba,NTT returns the coefficients%MOD. FFT returns the coefficients as they are.
* Karatsuba runs O(n^(log2(3)), FFT runs in O(nlog2(n)), NTT runs in O(nlog2(n)). But NTT is slower than FFT due to constant factors.
* You cannot use NTT for arbitrary mods. NTT works for special mods only. So for the general case, NTT is not an option.

**Rule of thumb:** Use karatsuba if you're sure time limit is okay for it. Otherwise go to FFT.

## FFT

1. /\*
2. multiply (7x^2 + 8x^1 + 9x^0) with (6x^1 + 5x^0)
3. ans = (42x^3 + 83x^2 + 94x^1 + 45x^0)
4. A = (9,8,7), B = (5,6)
5. multiply(A,B,res)
6. res will be (45 94 83 42)
7. \*/
8. **typedef** complex<**double**> base;
9. **void** fft (vector<base> & a, **bool** invert) {
10. **int** n = (**int**) a.size();
12. **for** (**int** i=1, j=0; i<n; ++i) {
13. **int** bit = n >> 1;
14. **for** (; j>=bit; bit>>=1)
15. j -= bit;
16. j += bit;
17. **if** (i < j)
18. swap (a[i], a[j]);
19. }
21. **for** (**int** len=2; len<=n; len<<=1) {
22. **double** ang = 2\*PI/len \* (invert ? -1 : 1);
23. base wlen (cos(ang), sin(ang));
24. **for** (**int** i=0; i<n; i+=len) {
25. base w (1);
26. **for** (**int** j=0; j<len/2; ++j) {
27. base u = a[i+j],  v = a[i+j+len/2] \* w;
28. a[i+j] = u + v;
29. a[i+j+len/2] = u - v;
30. w \*= wlen;
31. }
32. }
33. }
34. **if** (invert)
35. **for** (**int** i=0; i<n; ++i)
36. a[i] /= n;
37. }
38. // A and B doesnt change after passing, res can be any vector
39. // A==B || B==res || A==res should not create any problem
40. **void** multiply (**const** vector<**int**> & a, **const** vector<**int**> & b, vector<**int**> &res) {
41. vector<base> fa (a.begin(), a.end()),  fb (b.begin(), b.end());
42. **size\_t** n = 1;
43. **while** (n < max (a.size(), b.size()))  n <<= 1;
44. n <<= 1;
45. fa.resize (n),  fb.resize (n);
47. fft (fa, **false**),  fft (fb, **false**);
48. **for** (**size\_t** i=0; i<n; ++i)
49. fa[i] \*= fb[i];
50. fft (fa, **true**);
51. res.resize (n);
52. **for** (**size\_t** i=0; i<n; ++i)
53. res[i] = **int** (fa[i].real() + 0.5);
54. **while**(res.size() && res.back() == 0)
55. res.pop\_back();
56. **if**(res.empty())
57. res.push\_back(0);
58. }

## Karatsuba

1. /\*
2. multiply (7x^2 + 8x^1 + 9x^0) with (6x^1 + 5x^0)
3. ans = (42x^3 + 83x^2 + 94x^1 + 45x^0)
4. A = [7,8,9], B = [6,5], n = 3, m = 2
5. multiply(A,B,res)
6. res will be (42 83 94 45)
7. \*/
8. #define MAX 200000\*4 /// Must be a power of 2 (Not really actually, 4\*MAX suffices)
9. #define MOD ?
10. #define ran(a, b) ((((rand() << 15) ^ rand()) % ((b) - (a) + 1)) + (a))
12. **long** **long** ptr = 0;
13. **long** **long** temp[128];
14. **long** **long** buffer[MAX \* 6];
15. **const** **long** **long** INF = 8000000000000000000LL;
17. **void** karatsuba(**int** n, **long** **long** \*a, **long** **long** \*b, **long** **long** \*res){ /// n is a power of 2
18. **int** i, j, s;
19. **if** (n < 33){ /// Reduce recursive calls by setting a threshold
20. **for** (i = 0; i < (n + n); i++) temp[i] = 0;
21. **for** (i = 0; i < n; i++){
22. **if** (a[i]){
23. **for** (j = 0; j < n; j++){
24. temp[i + j] += (a[i] \* b[j]);
25. **if** (temp[i + j] > INF) temp[i + j] %= MOD;
26. }
27. }
28. }
29. **for** (i = 0; i < (n + n); i++) res[i] = temp[i] % MOD;
30. **return**;
31. }
33. s = n >> 1;
34. karatsuba(s, a, b, res);
35. karatsuba(s, a + s, b + s, res + n);
36. **long** **long** \*x = buffer + ptr, \*y = buffer + ptr + s, \*z = buffer + ptr + s + s;
38. ptr += (s + s + n);
39. **for** (i = 0; i < s; i++){
40. x[i] = a[i] + a[i + s], y[i] = b[i] + b[i + s];
41. **if** (x[i] >= MOD) x[i] -= MOD;
42. **if** (y[i] >= MOD) y[i] -= MOD;
43. }
45. karatsuba(s, x, y, z);
46. **for** (i = 0; i < n; i++) z[i] -= (res[i] + res[i + n] - MOD);
47. **for** (i = 0; i < n; i++) res[i + s] = (res[i + s] + z[i] + MOD) % MOD;
48. ptr -= (s + s + n);
49. }
51. /// multiplies two polynomial a( degree n-1 ) and b( degree m-1 ) and returns the result modulo MOD in a
52. /// returns (the degree of the multiplied polynomial + 1)
53. /// note that a and b are changed in the process
54. **int** mul(**int** n, **long** **long** \*a, **int** m, **long** **long** \*b){
55. **int** i, r, c = (n < m ? n : m), d = (n > m ? n : m);
56. **long** **long** \*res = buffer + ptr;
57. r = 1 << (32 - \_\_builtin\_clz(d) - (\_\_builtin\_popcount(d) == 1));
58. **for** (i = d; i < r; i++) a[i] = b[i] = 0;
59. **for** (i = c; i < d && n < m; i++) a[i] = 0;
60. **for** (i = c; i < d && m < n; i++) b[i] = 0;
62. ptr += (r << 1), karatsuba(r, a, b, res), ptr -= (r << 1);
63. **for** (i = 0; i < (r << 1); i++) a[i] = res[i];
64. **return** (n + m - 1);
65. }

68. /// For a polynomial of degree D ,
69. /// coeff[0] will contain the coefficient of x^D
70. /// coeff[D] will contain the coefficient of x^0
71. /// coeff[D-i] will contain the coefficient of x^i

Knuth Morris Pratt

1. #include <bits/stdc++.h>
2. **using** **namespace** std;
4. /\* Complexity = O(P+S) \*/
5. /\* Searches pat in str \*/
7. **const** **int** MAX = 100010;
9. **char** str[MAX],pat[MAX];
10. **int** pref[MAX];
11. **int** match[MAX];
13. // pref[i] =   length of the longest suffix which is also
14. // a proper prefix of the original string
15. **void** prefixFunction(**int** P) {
16. **int** j=0;
17. **for**(**int** i=1; i<P; i++) {
18. **while**(**true**) {
19. **if**(pat[i]==pat[j]) {
20. j = pref[i] = j+1; **break**;
21. }
22. **else** {
23. **if**(j==0) {
24. pref[i] = 0; **break**;
25. }
26. **else** j = pref[j-1];
27. }
28. }
29. }
30. }
32. **void** KMPMatcher(**int** S) {
33. **int** j = 0;
34. **for**(**int** i=0; i<S; i++) {
35. **while**(**true**) {
36. **if**(str[i]==pat[j]) {
37. j = match[i] = j+1; **break**;
38. }
39. **else** {
40. **if**(j==0) {
41. match[i] = 0; **break**;
42. }
43. **else** j = pref[j-1];
44. }
45. }
46. }
47. }
49. **int** main() {
50. scanf("%s",str);
51. scanf("%s",pat);
52. **int** S = strlen(str);
53. **int** P = strlen(pat);
55. prefixFunction(P);
56. KMPMatcher(S);
57. **return** 0;
58. }

## Notes

* We can avoid the matcher by calculating the prefix function for the string = P + “#” + S
* To find the smallest period of a string , If ( len % ( len – pref[len-1] ) ) == 0 then the string hasa period of length ( len – pref[len-1] ) and it occurs len / ( len – pref[len-1] ) times

Suffix Array

## Notes

* Think of suffix as a trie of suffixes.
* lcp (x, y) = minimum { lcp(x, x + 1), lcp(x + 1, x + 2), ... lcp(y – 1, y) }.

## n \* lg2n \* lg2n

1. #define MAXL 10100
2. #define MAXLG 15
4. **char** str[MAXL], base = 'a';
5. **int** stp;
6. **int** S[MAXL]; //will contain the starting idx of the i'th suffix in suffix array
7. **int** P[MAXLG+5][MAXL+5];
8. // P[i][j] = position of the suffix starting at character j after sorting
9. // on the basis of 2^i characters

12. **struct** entry{
13. **int** pr[2]; // parameters for sorting
14. **int** id; // starting index of the suffix
15. }suf[MAXL];
17. **inline bool** cmp(entry a,entry b){
18. **if**(a.pr[0]==b.pr[0]) **return** a.pr[1]<b.pr[1];
19. **else** **return** a.pr[0]<b.pr[0];
20. }
21. // n \* lg n \* lg n
22. **void** generateSA(**int** L){
23. **int** now,i;
25. **for**(i=0;i<L;i++) P[0][i] = str[i]-base;
27. **for**(now=1,stp=1 ; now <L ; stp++,now \*= 2){
28. **for**(i=0;i<L;i++){
29. suf[i].pr[0] = P[stp-1][i];
30. **if**(i+now<L) suf[i].pr[1] = P[stp-1][i+now];
31. **else** suf[i].pr[1] = -1;
32. suf[i].id = i;
33. }
34. sort(suf,suf+L,cmp);
35. **for**(i=0;i<L;i++)
36. **if**(i>0 && suf[i].pr[0]==suf[i-1].pr[0] && suf[i].pr[1]==suf[i-1].pr[1])
37. P[stp][suf[i].id] = P[stp][suf[i-1].id];
38. **else**
39. P[stp][suf[i].id] = i;
40. }
41. **for**(i=0; i<L ; i++) S[P[stp-1][i]] = i;
42. }
43. // n \* lg n
44. **inline int** getLCP(**int** x,**int** y,**int** L){
45. **int** ret = 0,add,i;
46. **if**(x==y) **return** L-x;
47. **for**(i = stp-1 ; i >= 0 && x<L && y<L; i--){
48. **if**(P[i][x]==P[i][y]){
49. add = (1<<i);
50. ret += add;
51. x += add , y += add;
52. }
53. }
54. **return** ret;
55. }

## n \* lg n

1. /\* Reducing lg n using counting sort \*/
3. **const** **int** MAXL = ?;// 1<< 20
4. **const** **int** MAXLG = ?;// 20
6. #define MAXL 300100
7. #define MAXLG 22
9. **char** str[MAXL], base = 'a';
10. **int** L , stp;
11. **int** S[MAXL]; //will contain the starting idx of the i'th suffix in suffix sufay
12. **int** P[MAXLG+5][MAXL+5];
13. // P[i][j] = position of the suffix starting at character j after sorting
14. // on the basis of 2^i characters
16. **struct** entry {
17. **int** pr[2]; // parameters for sorting
18. **int** id; // starting index of the suffix
19. } suf[MAXL] , out[MAXL];
21. **int** cnt[MAXL] , taken[MAXL] , cum[MAXL];
22. **int** special , specialTaken , rdLim;
24. **inline** **void** countingSort(**int** type) {
25. **int** i;
26. CLR(cnt);
27. CLR(taken);
28. special = specialTaken = 0;
29. **for**(i = 0; i<L ; i++) {
30. **if**(suf[i].pr[type] == -1) special++;
31. **else** cnt[ suf[i].pr[type] ]++;
32. }
33. cum[0] = special;
34. **for**(i = 1; i <= rdLim ; i++) cum[i] = cum[i-1] + cnt[i-1];
35. **for**(i = 0; i<L ; i++) {
36. **if**(suf[i].pr[type] == -1) out[ specialTaken++ ] = suf[i];
37. **else** out[ cum[ suf[i].pr[type] ] + taken[ suf[i].pr[type] ]++ ] = suf[i];
38. }
39. **for**(i = 0; i<L ; i++) suf[i] = out[i];
40. }
42. // n \* lg n
43. **void** generateSA(){
44. **int** now,i;
46. rdLim = 0;
48. **for**(i=0; i<L; i++){
49. P[0][i] = str[i]-base;
50. rdLim = max(rdLim, P[0][i]);
51. }
53. **for**(now=1,stp=1 ; now <L ; stp++,now \*= 2){
54. **for**(i=0; i<L; i++){
55. suf[i].pr[0] = P[stp-1][i];
56. **if**(i+now<L) suf[i].pr[1] = P[stp-1][i+now];
57. **else** suf[i].pr[1] = -1;
58. suf[i].id = i;
59. }
60. countingSort(1);
61. countingSort(0);
62. rdLim = 0;
63. **for**(i=0; i<L; i++){
64. **if**(i>0 && suf[i].pr[0]==suf[i-1].pr[0] && suf[i].pr[1]==suf[i-1].pr[1])
65. P[stp][suf[i].id] = P[stp][suf[i-1].id];
66. **else**
67. P[stp][suf[i].id] = i;
69. rdLim = max(rdLim, P[stp][ suf[i].id ]);
70. }
71. }
72. **for**(i=0; i<L ; i++) S[P[stp-1][i]] = i;
73. }
74. // n \* lg n
75. **inline** **int** getLCP(**int** x,**int** y){
76. **int** ret = 0,add,i;
77. **if**(x==y) **return** L-x;
78. **for**(i = stp-1 ; i >= 0 && x<L && y<L; i--){
79. **if**(P[i][x]==P[i][y]){
80. add = (1<<i);
81. ret += add;
82. x += add, y += add;
83. }
84. }
85. **return** ret;
86. }

## n \* lg n (Zobayer Vai)

1. //Suffix array implementation using bucket sorting + lcp.
3. **const** **int** MAXL = ?; // 50000
4. **const** **int** MAXLG = ?; // 20;
6. **int** L, stp, mv, tmp[MAXL];
7. **int** sum[MAXL], cnt[MAXL];
9. **char** str[MAXL];
10. **int** P[MAXLG][MAXL];
11. // P[i][j] = position of the suffix starting at character j after sorting
12. // on the basis of 2^i characters
13. **int** S[MAXL]; //will contain the starting idx of the i'th suffix in suffix array
15. **inline** **bool** eq(**const** **int** &u, **const** **int** &v){
16. **if**(!stp) **return** str[u] == str[v];
17. **if**(P[stp-1][u] != P[stp-1][v]) **return** **false**;
18. **int** a = u + mv < L ? P[stp-1][u+mv] : -1;
19. **int** b = v + mv < L ? P[stp-1][v+mv] : -1;
20. **return** a == b;
21. }
23. **void** update(){
24. **int** i, rnk;
25. **for**(i = 0; i < L; i++) sum[i] = 0;
26. **for**(i = rnk = 0; i < L; i++) {
27. S[i] = tmp[i];
28. **if**(i && !eq(S[i], S[i-1])) {
29. P[stp][S[i]] = ++rnk;
30. sum[rnk+1] = sum[rnk];
31. }
32. **else** P[stp][S[i]] = rnk;
33. sum[rnk+1]++;
34. }
35. }
37. **void** Sort() {
38. **int** i;
39. **for**(i = 0; i < L; i++) cnt[i] = 0;
40. memset(tmp, -1, **sizeof** tmp);
41. **for**(i = 0; i < mv; i++){
42. **int** idx = P[stp - 1][L - i - 1];
43. **int** x = sum[idx];
44. tmp[x + cnt[idx]] = L - i - 1;
45. cnt[idx]++;
46. }
47. **for**(i = 0; i < L; i++){
48. **int** idx = S[i] - mv;
49. **if**(idx < 0)**continue**;
50. idx = P[stp-1][idx];
51. **int** x = sum[idx];
52. tmp[x + cnt[idx]] = S[i] - mv;
53. cnt[idx]++;
54. }
55. update();
56. **return**;
57. }
59. **inline** **bool** cmp(**const** **int** &a, **const** **int** &b){
60. **if**(str[a]!=str[b]) **return** str[a]<str[b];
61. **return** **false**;
62. }
64. **void** SortSuffix() {
65. **int** i;
66. **for**(i = 0; i < L; i++) tmp[i] = i;
67. sort(tmp, tmp + L, cmp);
68. stp = 0;
69. update();
70. ++stp;
71. **for**(mv = 1; mv < L; mv <<= 1) {
72. Sort();
73. stp++;
74. }
75. stp--;
76. **for**(i = 0; i <= stp; i++) P[i][L] = -1;
77. }
79. **inline** **int** getLCP(**int** u, **int** v) {
80. **if**(u == v) **return** L - u;
81. **int** ret, i;
82. **for**(ret = 0, i = stp; i >= 0; i--) {
83. **if**(P[i][u] == P[i][v]) {
84. ret += 1<<i;
85. u += 1<<i;
86. v += 1<<i;
87. }
88. }
89. **return** ret;
90. }

Longest Increasing Subsequence (nlog(n))

1. /\*
2. The size of the vector after each iteration denotes the size of the LIS of the sub array starting at 1 and ending at i
3. \*/
5. **int** ara[MAX];
6. vector <**int**> v;
7. **int** max\_lis = 0;
8. **for**(i=1;i<=n;i++){
9. x = lower\_bound(all(v),ara[i])-v.begin();
10. **if**(x==0){
11. **if**(v.size()==0) v.pb(ara[i]);
12. **else** v[0] = ara[i];
13. }
14. **else** **if**(x==v.size()) v.pb(ara[i]);
15. **else** **if**(ara[i]<v[x]) v[x] = ara[i];
16. max\_lis = max(max\_lis,(**int**)v.size());
17. }
18. cout << "The size of the lis is : " << max\_lis << endl;

Sorting an Array with Minimum Moves

## Type 1

### Moves

Taking an element and inserting it in the **front** of the array.

### Solution

The idea is to traverse array from end. We expect n at the end, so we initialize expectedItem as n. All the items which are between actual position of expectedItem and current position must be moved to front. So, we calculate the number of items between current item and expected item. Once we find expectedItem, we look for next expectedItem by reducing expectedITem by one.

## Type 2

### Moves

Taking an element and inserting it in **any position** of the array.

### Solution

Number of elements in the array – LIS

## Type 3

### Move

Swapping two elements

### Solution

This can be easily done by visualizing the problem as a graph. We will have **n** nodes and an edge directed from node **i** to node **j** if the element at i’th index must be present at j’th index in the sorted array. The graph will now contain many non-intersecting cycles. Now a cycle with 2 nodes will only require 1 swap to reach the correct ordering, similarly a cycle with 3 nodes will only require 2 swaps to do so.

Hence,

ans = **Σ** cycle\_size – number of cycles

Fast IO

1. // For windows system use getchar() in place of getchar\_unlocked()
2. **inline** **int** RI() {
3. **int**  ret = 0, flag = 1,ip = getchar\_unlocked();
4. **for**(; ip < 48 || ip > 57; ip = getchar\_unlocked()) {
5. **if**(ip == 45) {
6. flag = -1;
7. ip = getchar\_unlocked();
8. **break**;
9. }
10. }
11. **for**(; ip > 47 && ip < 58; ip = getchar\_unlocked())
12. ret = ret \* 10 + ip - 48 ;
13. **return** flag \* ret;
14. }
16. // scanning syntax
17. **int** n = RI();

Game Theory

## Impartial Game

An [impartial game](https://en.wikipedia.org/wiki/Impartial_game) is one such as [nim](https://en.wikipedia.org/wiki/Nim), in which each player has exactly the same available moves as the other player in any position.

## Nim Game

N piles each containing some stones. He who removes the last stones wins.

Solution : If xor sum of the number of stones in the piles>0, the first player will win, otherwise the second player.

## Turning Turtles

Given a horizontal line of N coins with some coins showing heads and some tails. Each turn, a player has to flip one coin from head to tail, and in the same time (if he/she wants), flip one more coin to the left of it.

Solution : This game is equivalent to Nim Game with each coin showing head in kth position equals to a pile of k stones.

## Grundy Number

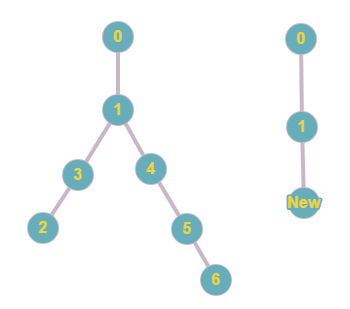
Grundy number of a losing state is 0.

Grundy number of a state A is the mex of the following set :

{ x : x is the grundy number of some state which is reachable from the state A by a single move }

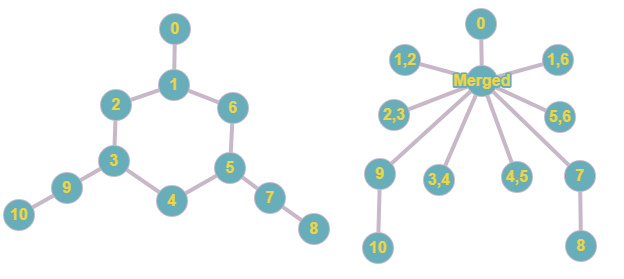
If we know the grundy number for every part of a composite game, the winner will be the first player if the xor sum of the grundy numbers is greater than 0.

## Colon principle

****

The trees in the image above are similar according to colon principle. Node 2 is 2 edges away from node 1 and node 6 is 3 edges away from node 1. So, we get a new node 2^3 = 1 edges away from node 1.

## Fusion Principle

****

Node 1,2,3,4,5,6 are in a cycle cycle, they are merged to a single node. For every edge in the cycle a new node arises. Thus, we can get a tree from any graph and apply colon principle there.

Sudoku Solver

1. /\*
2. Input:
3. .46...9..
4. .3.1.....
5. .2..6..85
6. ...87....
7. 6...3...4
8. ....14...
9. 79..5..3.
10. .....2.4.
11. ..2...61.
12. \*/
14. **bool** row[10][10],col[10][10],box[10][10];
15. **char** mat[10][10];
17. **int** availableNumbers(**int** x,**int** y) {
18. **int** cnt = 0;
19. **int** bn = (x/3)\*3 + y/3;
20. **for**(**int** i=1;i<=9;i++) {
21. **if**( !(row[x][i] || col[y][i] || box[bn][i]) ) cnt++;
22. }
23. **return** cnt;
24. }
26. **bool** solve() {
27. **int** r , c , mn = 100;
28. **for**(**int** i=0;i<9;i++) {
29. **for**(**int** j=0;j<9;j++) {
30. **if**(mat[i][j]!='.') **continue**;
31. **int** cnt = availableNumbers(i,j);
32. **if**(cnt<mn) {
33. mn = cnt;
34. r = i , c = j;
35. }
36. }
37. }
38. **if**(mn==100) **return** **true**;
39. **int** bn = (r/3)\*3 + c/3;
40. **for**(**int** i=1;i<=9;i++) {
41. **if**( !(row[r][i] || col[c][i] || box[bn][i]) ) {
43. mat[r][c] = '0' + i;
44. row[r][i] = col[c][i] = box[bn][i] = **true**;
46. **if**(solve()) **return** **true**;
48. mat[r][c] = '.';
49. row[r][i] = col[c][i] = box[bn][i] = **false**;
51. }
52. }
53. **return** **false**;
54. }
56. **int** main() {
57. **int** t,T,i,j;
58. si(T);
59. **for**(t=1;t<=T;t++) {
60. CLR(row); CLR(col); CLR(box);
61. **for**(i=0;i<9;i++) {
62. scanf("%s",mat[i]);
63. }
64. **for**(i=0;i<9;i++) {
65. **for**(j=0;j<9;j++) {
66. **if**(mat[i][j]=='.') **continue**;
67. row[i][mat[i][j]-'0'] = **true**;
68. col[j][mat[i][j]-'0'] = **true**;
69. **int** bn = (i/3)\*3 + j/3;
70. box[bn][mat[i][j]-'0'] = **true**;
71. }
72. }
73. printf("Case %d:\n",t);
74. solve();
75. **for**(i=0;i<9;i++) printf("%s\n",mat[i]);
76. }
77. **return** 0;
78. }

Knight Distance in Infinite Chessboard

1. /// Minimum number of knight moves to reach (x,y) from (0,0) in infinite chessboard
2. /// x or y can be negative
3. /// minimum move to reach from (x1,y1) to (x2,y2) = knight\_move( x1-x2 , y1-y2 )
4. **int** knight\_move(**int** x, **int** y){
5. **int** a, b, z, c, d;
6. x = abs(x), y = abs(y);
7. **if** (x < y) a = x, x = y, y = a;
8. **if** (x == 2 && y == 2) **return** 4;
9. **if** (x == 1 && y == 0) **return** 3;
11. **if** (y == 0 || (y << 1) < x){
12. c = y & 1;
13. a = x - (c << 1), b = a & 3;
14. **return** ((a - b) >> 1) + b + c;
15. }
16. **else**{
17. d = x - ((x - y) >> 1);
18. z = ((d % 3) != 0), c = (x - y) & 1;
19. **return** ((d / 3) \* 2) + c + (z \* 2 \* (1 - c));
20. }
21. }

STL

## Vector

1. /// idx contains the index of the leftmost element in the vector which is greater than val
2. **int** idx = upper\_bound( V.begin() , V.end() , val ) - V.begin();
4. /// idx contains the index of the leftmost element in the vector which is not less than val
5. **int** idx = lower\_bound( V.begin() , V.end() , val ) - V.begin();
7. /// idx = V.size() if no such element in both cases
9. /\*\*\*
10. V.begin() and V.end() are iterators
11. To get the value in a range [L,R) :
12. V.begin() should be replaced by iterator to L
13. V.end() should be replaced by iterator to R
15. Iterator to the elemnet at index i = ( v.begin() + i )
16. \*\*\*/
18. /// returns true if val is in the vector, false otherwise
19. binary\_search ( V.begin() , V.end() , val )
21. ///merging two sorted vectors V1 and V2 to vector V
22. V.resize( V1.size() + V2.size() );
23. merge(V1.begin(),V1.end(),V2.begin(),V2.end(),V.begin());

## Priority Queue

1. // to keep the elements in ascending order
2. priority\_queue < **int** , vector < **int** > , greater <**int**> > Q;

## Set

1. set <**int**> S;
2. S.lower\_bound(x) returns the iterator to the first element >= x
3. S.upper\_bound(x) returns the iterator to the first element > x
4. In both cases **if** there is no such element, the functions returns
5. the iterator to the end of the set

## Policy Based Data Structures

1. #include <bits/stdc++.h>
2. #include <ext/pb\_ds/assoc\_container.hpp> // Common file
3. #include <ext/pb\_ds/tree\_policy.hpp> // Including tree\_order\_statistics\_node\_update
4. **using** **namespace** std;
5. **using** **namespace** \_\_gnu\_pbds;
6. // we can replace int with other data types
7. // If the data type is user defined, we need to define less operator for that type
8. **typedef** tree<
9. **int** ,
10. null\_type ,
11. less < **int** > ,
12. rb\_tree\_tag,
13. tree\_order\_statistics\_node\_update > ordered\_set;
14. // ordered\_set has become a data type, OS is an ordered\_set
15. ordered\_set OS;
16. /\*
17. \* this ordered\_set is a set basically
18. \* ordered\_set declared as above can supports all the set operations
19. like insert() , erase() , find() , lower\_bound() , upper\_bound()
21. \* Ordered set supports two extra functions
22. OS.find\_by\_order(x)
23. returns the iterator to the k'th largest element starting count from 0
24. OS.order\_of\_key(x)
25. returns number of items in the set strictly smaller than x
26. \*/
28. **int** main(){
29. OS.insert(1);
30. OS.insert(2);
31. OS.insert(4);
32. OS.insert(8);
33. OS.insert(16);
34. cout << ( \*OS.find\_by\_order(0) ) << endl; // 1
35. cout << ( \*OS.find\_by\_order(2) ) << endl; // 4
36. cout << ( \*OS.find\_by\_order(4) ) << endl; // 16
37. cout << ( end(OS) == OS.find\_by\_order(5) ) <<endl; // true
38. cout << OS.order\_of\_key(-5) << endl;  // 0
39. cout << OS.order\_of\_key(3) << endl;   // 2
40. cout << OS.order\_of\_key(400) << endl; // 5
41. **return** 0;
42. }

## Bitset

1. // A bitset of size S
2. bitset < S > B;
3. // A bitset of size S initialized with bits of 10("1010")
4. bitset < S > B(10); //...00001010
5. // A bitset of size S initialized with bits of 10("1010")
6. bitset < S > B(string("1010")) //...00001010
8. bitset < S > B[MAX] // array of bitsets each having size S
9. // to access the j'th element of the i't bitset B[i][j] is to be used
11. B.set(); // makes all the bits 1 (if no parameter given)
12. B.reset(); // makes all the bits 0 (if no parameter given)
13. B.flip(i); // flips all the bits (if no parameter given)
14. B.any(); // returns true if some bits are set
15. B.count(); // how many ones
17. and, or, xor , right shift, left shift operations are also allowed in bitsets
19. bitset < S > B1,B2,B;
20. Example
21. B = B1 ^ B2;

Convex Hull

1. /\*
2. ConvexHull : Graham's Scan O( n \* lg(n) )
3. P[]: holds all the Points, C[]: holds Points on the hull(in anti clockwise order)
4. np: number of Points in P[], nc: number of Points in C[]
5. If there are duplicate Points in P, call makeUnique() before
6. calling convexHull() call convexHull() if you have np >= 3
7. to remove co-linear Points on hull, call compress() after convexHull()
8. Call getBakiPoints() to get all the points that lie in the perimeter of
9. the convex hull
10. \*/

13. **struct** point { // Creates normal 2D point
14. **double** x, y;
15. point() {}
16. point( **double** xx, **double** yy ) { x = xx, y = yy; }
17. };
19. point P[MAX], C[MAX], P0;
21. **bool** nisi[MAX];
23. **inline** **double** triArea2(**const** point &a, **const** point &b, **const** point &c) {
24. **return** (a.x\*(b.y-c.y) + b.x\*(c.y-a.y) + c.x\*(a.y-b.y)); // returns area\*2
25. }
27. **inline** **double** sqDist(**const** point &a, **const** point &b) {
28. **return** ((a.x-b.x)\*(a.x-b.x) + (a.y-b.y)\*(a.y-b.y)); // returns dis\*dis
29. }
31. **inline** **bool** comp(**const** point &a, **const** point &b) {
32. **double** d = triArea2(P0, a, b);
33. **if**(d < 0) **return** **false**;
34. **if**(!d && sqDist(P0, a) > sqDist(P0, b)) **return** **false**;
35. **return** **true**;
36. }
38. **void** convexHull(**int** &np, **int** &nc) {
39. **int** i, j, pos = 0;
40. **for**(i = 1; i < np; i++)
41. **if**(P[i].y<P[pos].y || ( fabs(P[i].y - P[pos].y) < EPS && P[i].x<P[pos].x ) )
42. pos = i;
43. swap(P[0], P[pos]);
44. P0 = P[0];
45. sort(P+1, P+np, comp);
46. **for**(i = 0; i < 3; i++) C[i] = P[i];
47. **for**(i = j = 3; i < np; i++) {
48. **while**(triArea2(C[j-2], C[j-1], P[i]) < 0) j--;
49. C[j++] = P[i];
50. }
51. nc = j;
52. }
54. **inline** **bool** eq(**const** point &a,**const** point &b){
55. **return** ( fabs( a.x - b.x ) < EPS && fabs( a.y - b.y ) < EPS ); // if two points are equal
56. }
58. **inline** **bool** normal(**const** point &a, **const** point &b) {
59. **return** ( fabs( a.x - b.x ) < EPS ? a.y < b.y : a.x < b.x);
60. }
62. **inline** **void** makeUnique(**int** &np) {
63. sort(P , P+np , normal);
64. np = unique(P , P+np , eq) - P;
65. }
67. **void** compress(**int** &nc) {
68. **int** i, j;
69. **double** d;
70. C[nc] = C[0];
71. **for**(i=j=1; i < nc; i++) {
72. d = triArea2(C[j-1], C[i], C[i+1]);
73. **if**(d || (!d && eq(C[j-1], C[i+1]))) C[j++] = C[i];
74. }
75. nc = j;
76. }
78. **void** getBakiPoints(**int** &np, **int** &nc){
79. **int** j = 0;
80. **for**(**int** i=0;i<nc;i++){
81. **while**(!eq(C[i],P[j])) j++;
82. nisi[j] = **true**; // If the point is already taken
83. }
85. **int** last = nc;
86. **for**(**int** i = np-1; i >= 0 ; i--){
87. **if**(!nisi[i]){
88. **if**(!eq(P[i],C[0]) && !eq(P[i],C[last-1]))
89. **if**(triArea2(P[i],C[0],C[nc-1])==0)
90. C[nc++] = P[i];
91. }
92. }
93. }

Important Mathematical Formulas

* Number of ways of forming teams from members where each consists of members () is

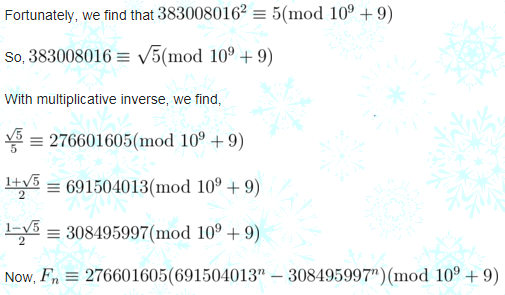
Ways of choosing members from and making one of them captain

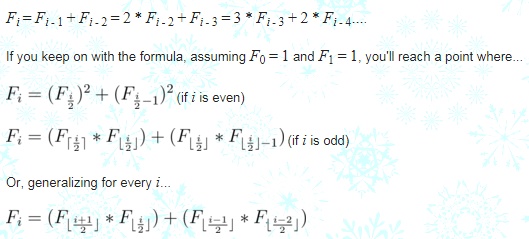
Ways of choosing members from and making one of them captain(k is not fixed here)

* Number of ways to tile a grid with two types of tiles() is

Hockey Stick Pattern

* is always a multiple of
* GCD() =



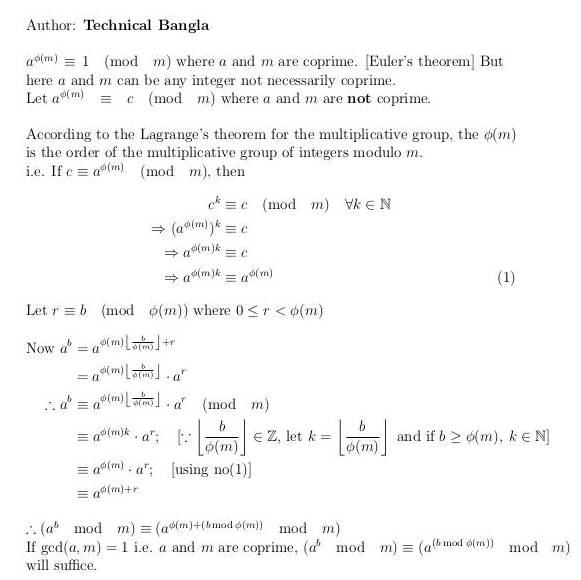


Modular Arithmatic

* because is a multiple of
* where and are pair wise co prime
* if is a prime
* Modular inverse of with respect to is
* If then has a square root modulo

Else and has no square root modulo

* where
* If then where is a divisor of
* Let and . If then
* If then
* implies that



Miscellaneous

* Given nodes numbered from to , number of different unrooted trees
* Counter-clockwise rotation matrix
* Clockwise rotation matrix

String Hash + Seg Tree

1. #include <bits/stdtr1c++.h>
3. #define clr(ar) memset(ar, 0, sizeof(ar))
4. #define read() freopen("lol.txt", "r", stdin)
5. #define dbg(x) cout << #x << " = " << x << endl
6. #define ran(a, b) ((((rand() << 15) ^ rand()) % ((b) - (a) + 1)) + (a))
8. **using** **namespace** std;
10. /// String Hash with Segment Tree
11. /// Allows setting range with a character, querying hash of a range in O(log n)
12. /// String functions uses 0-based index, however segment tree uses 1-based index
14. **namespace** strhash{
15. #define LET 10 /// Number of distinct letters in string, Set to digits [0-9] by default
16. #define ADD 10007 /// Just another prime added randomly to get better hash values :)
17. #define MAX 100010
19. **int** P[2][MAX], RD[2][LET][MAX];
20. **int** n, ar[MAX], lazy[MAX \* 4];
21. **const** **int** MOD[] = {2078526727, 2117566807};
22. **const** **int** BASE[] = {1572872831, 1971536491};
24. **void** build\_tree(**int** idx, **int** a, **int** b);
26. **inline** **int** getval(**char** ch){ /// Value of a character (without adding ADD)
27. **return** ch - 48; /// Change this for lowercase or uppercase letters
28. }
30. **void** precalc(){ /// Call precalc() just once in whole program
31. **int** i, j, k, d;
32. P[0][0] = P[1][0] = 1;
33. **for** (i = 1; i < MAX; i++){
34. P[0][i] = ((**long** **long**) P[0][i - 1] \* BASE[0]) % MOD[0];
35. P[1][i] = ((**long** **long**) P[1][i - 1] \* BASE[1]) % MOD[1];
36. }
38. **for** (i = 0; i < 2; i++){
39. **for** (d = 0; d < LET; d++){
40. k = ADD + d;
41. **long** **long** x = 0;
42. **for** (j = 1; j <= MAX; j++){
43. x = ((x \* BASE[i]) + k) % MOD[i];
44. RD[i][d][j] = x;
45. }
46. }
47. }
48. }
50. **void** init(**char**\* str){
51. n = strlen(str);
52. **for** (**int** i = 0; i < n; i++) ar[i + 1] = getval(str[i]) + ADD;
53. build\_tree(1, 1, n);
54. }
56. **struct** Node{
57. **int** H[2];
59. **inline** Node(){
60. }
62. **inline** Node(**int** h1, **int** h2){
63. H[0] = h1, H[1] = h2;
64. }
65. } tree[MAX << 2];
67. **inline** **void** propagate(**int** idx, **int** a, **int** b){
68. **if** (lazy[idx]){
69. **int** p = (idx << 1), q = p + 1;
70. **int** c = (a + b) >> 1, d = c + 1, val = lazy[idx];
72. **if** (a != b){
73. lazy[p] = lazy[q] = lazy[idx];
74. tree[p] = Node(RD[0][val - ADD][c - a + 1], RD[1][val - ADD][c - a + 1]);
75. tree[q] = Node(RD[0][val - ADD][b - d + 1], RD[1][val - ADD][b - d + 1]);
76. }
77. lazy[idx] = 0;
78. }
79. }
81. **inline** **void** build\_tree(**int** idx, **int** a, **int** b){
82. lazy[idx] = 0;
83. **if** (a == b){
84. tree[idx].H[0] = tree[idx].H[1] = ar[a];
85. **return**;
86. }
88. **int** p = (idx << 1), q = p + 1;
89. **int** c = (a + b) >> 1, d = c + 1;
91. build\_tree(p, a, c);
92. build\_tree(q, d, b);
94. tree[idx].H[0] = ((tree[p].H[0] \* (**long** **long**) P[0][b - d + 1]) + tree[q].H[0]) % MOD[0];
95. tree[idx].H[1] = ((tree[p].H[1] \* (**long** **long**) P[1][b - d + 1]) + tree[q].H[1]) % MOD[1];
96. }
98. **inline** **void** update(**int** idx, **int** a, **int** b, **int** l, **int** r, **int** val){
99. **if** (a == l && b == r){
100. lazy[idx] = val;
101. tree[idx] = Node(RD[0][val - ADD][b - a + 1], RD[1][val - ADD][b - a + 1]);
102. propagate(idx, a, b);
103. **return**;
104. }
106. propagate(idx, a, b);
107. **int** p = (idx << 1), q = p + 1;
108. **int** c = (a + b) >> 1, d = c + 1;
110. **if** (r <= c) update(p, a, c, l, r, val);
111. **else** **if** (l >= d) update(q, d, b, l, r, val);
112. **else**{
113. update(p, a, c, l, c, val);
114. update(q, d, b, d, r, val);
115. }
117. tree[idx].H[0] = ((tree[p].H[0] \* (**long** **long**) P[0][b - d + 1]) + tree[q].H[0]) % MOD[0];
118. tree[idx].H[1] = ((tree[p].H[1] \* (**long** **long**) P[1][b - d + 1]) + tree[q].H[1]) % MOD[1];
119. }
121. **inline** Node query(**int** idx, **int** a, **int** b, **int** l, **int** r){
122. propagate(idx, a, b);
123. **int** p = (idx << 1), q = p + 1;
124. **int** c = (a + b) >> 1, d = c + 1;
126. **if** (a == l && b == r) **return** tree[idx];
127. **if** (r <= c) **return** query(p, a, c, l, r);
128. **else** **if** (l >= d) **return** query(q, d, b, l, r);
129. **else**{
130. Node x = query(p, a, c, l, c);
131. Node y = query(q, d, b, d, r);
132. **for** (**int** i = 0; i < 2; i++){
133. x.H[i] = ((P[i][r - d + 1] \* (**long** **long**) x.H[i]) + y.H[i]) % MOD[i];
134. }
135. **return** x;
136. }
137. }
139. /// Note 0-based index used for string update and gethash
140. **inline** **void** update(**int** l, **int** r, **char** ch){ /// Sets the sub-string in str[l:r] to ch
141. update(1, 1, n, ++l, ++r, getval(ch) + ADD);
142. }
144. **inline** **long** **long** gethash(**int** l, **int** r){
145. Node h = query(1, 1, n, ++l, ++r);
146. **return** (h.H[0] << 32) | h.H[1];
147. }
148. }
150. **int** main(){
152. }

Discrete Log(General)

1. #include <bits/stdtr1c++.h>
3. #define clr(ar) memset(ar, 0, sizeof(ar))
4. #define read() freopen("lol.txt", "r", stdin)
5. #define dbg(x) cout << #x << " = " << x << endl
7. **using** **namespace** std;
9. **int** extended\_gcd(**int** a, **int** b, **int**& x, **int**& y){
10. /// Bezout's identity, ax + by = gcd(a,b)
12. **if** (!b){
13. y = 0, x = 1;
14. **return** a;
15. }
17. **int** g = extended\_gcd(b, a % b, y, x);
18. y -= ((a / b) \* x);
19. **return** g;
20. }
22. **int** discrete\_log(**int** g, **int** h, **int** p){ /// hash = 167626
23. /\*\*\*
24. \* returns smallest x such that (g^x) % p = h, -1 if none exists
25. \* function returns x, the discrete log of h with respect to g modulo p
26. \*\*\*/
28. **if** (h >= p) **return** -1;
29. **if** ((g % p) == 0){
30. **if** (h == 1) **return** 0; /// return -1 if strictly positive integer solution is required
31. **else** **return** -1;
32. }
34. **int** i, c, x, y, z, r, m, counter = 0;
35. **long** **long** v = 1, d = 1, mul = 1, temp = 1 % p;
37. **for** (**int** i = 0; i < 100; i++){
38. **if** (temp == h) **return** i;
39. temp = (temp \* g) % p;
40. }
42. **while** ((v = \_\_gcd(g, p)) > 1){
43. **if** (h % v) **return** -1;
44. h /= v, p /= v;
45. d = (d \* (g / v)) % p;
46. counter++;
47. }
49. m = ceil(sqrt(p)); /// may be change to sqrtl() ?
50. tr1::unordered\_map <**int**, **int**> mp;
52. **for** (i = 0; i < m; i++){
53. **if** (!mp[mul]) mp[mul] = i + 1;
54. mul = (mul \* g) % p;
55. }
57. **for** (i = 0; i < m; i++){
58. z = extended\_gcd(d, p, x, y);
59. c = p / z;
60. r = ((((**long** **long**)x \* h) / z) % p + p) % p;
61. **if** (mp[r]) **return** ((i \* m) + mp[r] + counter - 1);
62. d = (d \* mul) % p;
63. }
65. **return** -1;
66. }
68. **int** main(){
69. **int** g, h, p, res;
71. **for** (; ;){
72. scanf("%d %d %d", &g, &p, &h);
73. **if** (!g && !p && !h) **break**;
75. res = discrete\_log(g, h % p, p);
76. **if** (res == -1) puts("No Solution");
77. **else** printf("%d\n", res);
78. }
79. **return** 0;
80. }

Discrete Log

1. #include <bits/stdtr1c++.h>
3. #define clr(ar) memset(ar, 0, sizeof(ar))
4. #define read() freopen("lol.txt", "r", stdin)
5. #define dbg(x) cout << #x << " = " << x << endl
7. **using** **namespace** std;
9. **int** expo(**long** **long** x, **int** n, **int** m){
10. **long** **long** res = 1;
12. **while** (n){
13. **if** (n & 1) res = (res \* x) % m;
14. x = (x \* x) % m;
15. n >>= 1;
16. }
18. **return** (res % m);
19. }
21. **int** extended\_gcd(**int** a, **int** b, **int**& x, **int**& y){
22. /// Bezout's identity, ax + by = gcd(a,b)
24. **if** (!b){
25. y = 0, x = 1;
26. **return** a;
27. }
29. **int** g = extended\_gcd(b, a % b, y, x);
30. y -= ((a / b) \* x);
31. **return** g;
32. }
34. **int** inverse\_modulo(**int** a, **int** m){
35. /// inverse exists if and only if a and m are co-prime
37. **int** x, y, inv;
38. extended\_gcd(a, m, x, y);
39. inv = (x + m) % m;
40. **return** inv;
41. }
43. **int** discrete\_log(**int** g, **int** h, **int** p){
44. /\*\*\*
45. \* returns smallest x such that (g^x) % p = h, -1 if none exists
46. \* p must be a PRIME
47. \* function returns x, the discrete log of h with respect to g modulo p
48. \*\*\*/
50. **if** (h >= p) **return** -1;
51. **if** ((g % p) == 0){
52. **if** (h == 1) **return** 0; /// return -1 if strictly positive integer solution is required
53. **else** **return** -1;
54. }
56. unordered\_map <**int**, **int**> mp;
57. **int** i, q, r, m = ceil(sqrt(p)); /// may be change to sqrtl() ?
58. **long** **long** d = 1, inv = expo(inverse\_modulo(g, p), m, p);
60. **for** (i = 0; i <= m; i++){
61. **if** (!mp[d]) mp[d] = i + 1;
62. d \*= g;
63. **if** (d >= p) d %= p;
64. }
66. d = h;
67. **for** (q = 0; q <= m; q++){
68. r = mp[d];
69. **if** (r) **return** ((m \* q) + (--r));
70. d \*= inv;
71. **if** (d >= p) d %= p;
72. }
74. **return** -1;
75. }
77. **int** main(){
78. **int** T = 0, t, g, h, p;
80. scanf("%d", &t);
81. **while** (t--){
82. scanf("%d %d %d", &g, &h, &p);
83. **int** x = discrete\_log(g, h, p);
84. printf("Case %d: %d\n", ++T, x);
85. }
86. **return** 0;
87. }

Random Code

1. **struct** Point
2. {
3. **double** x, y;
4. Point(**double** px, **double** py)
5. {
6. x = px;
7. y = py;
8. }
9. Point sub(Point p2)
10. {
11. **return** Point(x - p2.x, y - p2.y);
12. }
13. Point add(Point p2)
14. {
15. **return** Point(x + p2.x, y + p2.y);
16. }
17. **double** distance(Point p2)
18. {
19. **return** sqrt((x - p2.x)\*(x - p2.x) + (y - p2.y)\*(y - p2.y));
20. }
21. Point normal()
22. {
23. **double** length = sqrt(x\*x + y\*y);
24. **return** Point(x/length, y/length);
25. }
26. Point scale(**double** s)
27. {
28. **return** Point(x\*s, y\*s);
29. }
30. };
32. **struct** line   // Creates a line with equation ax + by + c = 0
33. {
34. **double** a, b, c;
35. line() {}
36. line( Point p1,Point p2 )
37. {
38. a = p1.y - p2.y;
39. b = p2.x - p1.x;
40. c = p1.x \* p2.y - p2.x \* p1.y;
41. }
42. };
44. **inline** **bool** eq(**double** a, **double** b)
45. {
46. **return** fabs( a - b ) < eps;
47. }

50. **struct** Circle
51. {
52. **double** x, y, r, left,right;
53. Circle () {}
54. Circle(**double** cx, **double** cy, **double** cr)
55. {
56. x = cx;
57. y = cy;
58. r = cr;
59. left = x - r;
60. right = x + r;
61. }
62. pair<Point, Point> intersections(Circle c)
63. {
64. Point P0(x, y);
65. Point P1(c.x, c.y);
66. **double** d, a, h;
67. d = P0.distance(P1);
68. a = (r\*r - c.r\*c.r + d\*d)/(2\*d);
69. h = sqrt(r\*r - a\*a);
70. Point P2 = P1.sub(P0).scale(a/d).add(P0);
71. **double** x3, y3, x4, y4;
72. x3 = P2.x + h\*(P1.y - P0.y)/d;
73. y3 = P2.y - h\*(P1.x - P0.x)/d;
74. x4 = P2.x - h\*(P1.y - P0.y)/d;
75. y4 = P2.y + h\*(P1.x - P0.x)/d;
77. **return** pair<Point, Point>(Point(x3, y3), Point(x4, y4));
78. }
80. };
82. **inline** **double** Distance( Point a, Point b )
83. {
84. **return** sqrt( ( a.x - b.x ) \* ( a.x - b.x ) + ( a.y - b.y ) \* ( a.y - b.y ) );
85. }
87. **inline** **double** Distance( Point P, line L )
88. {
89. **return** fabs( L.a \* P.x + L.b \* P.y + L.c ) / sqrt( L.a \* L.a + L.b \* L.b );
90. }

93. **bool** intersection(Circle C,line L,Point &p1,Point &p2)
94. {
95. **if**( Distance( {C.x,C.y}, L ) > C.r + eps ) **return** **false**;
96. **double** a, b, c, d, x = C.x, y = C.y;
97. d = C.r\*C.r - x\*x - y\*y;
98. **if**( eq( L.a, 0) )
99. {
100. p1.y = p2.y = -L.c / L.b;
101. a = 1;
102. b = 2 \* x;
103. c = p1.y \* p1.y - 2 \* p1.y \* y - d;
104. d = b \* b - 4 \* a \* c;
105. d = sqrt( fabs (d) );
106. p1.x = ( b + d ) / ( 2 \* a );
107. p2.x = ( b - d ) / ( 2 \* a );
108. }
109. **else**
110. {
111. a = L.a \*L.a + L.b \* L.b;
112. b = 2 \* ( L.a \* L.a \* y - L.b \* L.c - L.a \* L.b \* x);
113. c = L.c \* L.c + 2 \* L.a \* L.c \* x - L.a \* L.a \* d;
114. d = b \* b - 4 \* a \* c;
115. d = sqrt( fabs(d) );
116. p1.y = ( b + d ) / ( 2 \* a );
117. p2.y = ( b - d ) / ( 2 \* a );
118. p1.x = ( -L.b \* p1.y -L.c ) / L.a;
119. p2.x = ( -L.b \* p2.y -L.c ) / L.a;
120. }
121. **return** **true**;
122. }

Circle Operations

**import** java.util***.*** ***\**** ;

**public** **class** CircleOperations {

**static** **final** **double** EPS = 1e-10;

**public** **static** **double** fastHypot(**double** x, **double** y) {

**return** Math.**sqrt**(x \* x + y \* y);

}

**public** **static** **class** Point {

**public** **double** x,

y;

**public** Point(**double** x, **double** y) {

**this**.x = x;

**this**.y = y;

}

}

**public** **static** **class** Circle {

**public** **double** r,

x,

y;

**public** Circle(**double** x, **double** y, **double** r) {

**this**.x = x;

**this**.y = y;

**this**.r = r;

}

**public** boolean contains(Point p) {

**return** fastHypot(p.x - x, p.y - y) < r + EPS;

}

}

**public** **static** **class** Line {

**double** a,

b,

c;

**public** Line(**double** a, **double** b, **double** c) {

**this**.a = a;

**this**.b = b;

**this**.c = c;

}

**public** Line(Point p1, Point p2) {

a = +(p1.y - p2.y);

b = -(p1.x - p2.x);

c = p1.x \* p2.y - p2.x \* p1.y;

}

}

// geometric solution

**public** **static** Point[] circleLineIntersection(Circle circle, Line line) {

**double** a = line.a;

**double** b = line.b;

**double** c = line.c + circle.x \* a + circle.y \* b;

**double** r = circle.r;

**double** aabb = a \* a + b \* b;

**double** d = c \* c / aabb - r \* r;

**if** (d > EPS) **return** **new** Point[0];

**double** x0 = -a \* c / aabb;

**double** y0 = -b \* c / aabb;

**if** (d > -EPS) **return** **new** Point[] {

**new** Point(x0 + circle.x, y0 + circle.y)

};

d /= -aabb;

**double** k = Math.**sqrt**(d < 0 ? 0 : d);

**return** **new** Point[] {

**new** Point(x0 + k \* b + circle.x, y0 - k \* a + circle.y),

**new** Point(x0 - k \* b + circle.x, y0 + k \* a + circle.y)

};

}

// algebraic solution

**public** **static** Point[] circleLineIntersection2(Circle circle, Line line) {

**return** Math.**abs**(line.a) >= Math.**abs**(line.b) ? intersection(line.a, line.b, line.c, circle.x, circle.y, circle.r, **false**) : intersection(line.b, line.a, line.c, circle.y, circle.x, circle.r, **true**);

}

**static** Point[] intersection(**double** a, **double** b, **double** c, **double** CX, **double** CY, **double** R, boolean **swap**) {

// ax+by+c=0

// (by+c+aCX)^2+(ay-aCY)^2=(aR)^2

**double** A = a \* a + b \* b;

**double** B = 2.0 \* b \* (c + a \* CX) - 2.0 \* a \* a \* CY;

**double** C = (c + a \* CX) \* (c + a \* CX) + a \* a \* (CY \* CY - R \* R);

**double** d = B \* B - 4 \* A \* C;

**if** (d < -EPS) **return** **new** Point[0];

d = Math.**sqrt**(d < 0 ? 0 : d);

**double** y1 = ( - B + d) / (2 \* A);

**double** x1 = ( - c - b \* y1) / a;

**double** y2 = ( - B - d) / (2 \* A);

**double** x2 = ( - c - b \* y2) / a;

**return** **swap** ? d > EPS ? **new** Point[] {

**new** Point(y1, x1),

**new** Point(y2, x2)

}: **new** Point[] {

**new** Point(y1, x1)

}: d > EPS ? **new** Point[] {

**new** Point(x1, y1),

**new** Point(x2, y2)

}: **new** Point[] {

**new** Point(x1, y1)

};

}

**public** **static** Point[] circleCircleIntersection(Circle c1, Circle c2) {

**if** (fastHypot(c1.x - c2.x, c1.y - c2.y) < EPS) {

**if** (Math.**abs**(c1.r - c2.r) < EPS) **return** **null**; // infinity intersection points

**return** **new** Point[0];

}

**double** dx = c2.x - c1.x;

**double** dy = c2.y - c1.y;

**double** A = -2 \* dx;

**double** B = -2 \* dy;

**double** C = dx \* dx + dy \* dy + c1.r \* c1.r - c2.r \* c2.r;

Point[] res = circleLineIntersection(**new** Circle(0, 0, c1.r), **new** Line(A, B, C));

**for** (Point point: res) {

point.x += c1.x;

point.y += c1.y;

}

**return** res;

}

**public** **static** **double** circleCircleIntersectionArea(Circle c1, Circle c2) {

**double** r = Math.**min**(c1.r, c2.r);

**double** R = Math.**max**(c1.r, c2.r);

**double** d = fastHypot(c1.x - c2.x, c1.y - c2.y);

**if** (d < R - r + EPS) **return** Math.PI \* r \* r;

**if** (d > R + r - EPS) **return** 0;

**double** area = r \* r \* Math.acos((d \* d + r \* r - R \* R) / 2 / d / r) + R \* R \* Math.acos((d \* d + R \* R - r \* r) / 2 / d / R) - 0.5 \* Math.**sqrt**(( - d + r + R) \* (d + r - R) \* (d - r + R) \* (d + r + R));

**return** area;

}

**public** **static** Line[] tangents(Circle a, Circle b) {

List < Line > lines = **new** ArrayList < >();

**for** (**int** i = -1; i <= 1; i += 2)

**for** (**int** j = -1; j <= 1; j += 2)

tangents(**new** Point(b.x - a.x, b.y - a.y), a.r \* i, b.r \* j, lines);

**for** (Line line: lines)

line.c -= line.a \* a.x + line.b \* a.y;

**return** lines.toArray(**new** Line[lines.size()]);

}

**static** **void** tangents(Point center2, **double** r1, **double** r2, List < Line > lines) {

**double** r = r2 - r1;

**double** z = center2.x \* center2.x + center2.y \* center2.y;

**double** d = z - r \* r;

**if** (d < -EPS) **return**;

d = Math.**sqrt**(d < 0 ? 0 : d);

lines.add(**new** Line((center2.x \* r + center2.y \* d) / z, (center2.y \* r - center2.x \* d) / z, r1));

}

// min enclosing circle in O(n) on average

**public** **static** Circle minEnclosingCircle(Point[] pointsArray) {

**if** (pointsArray.length == 0) **return** **new** Circle(0, 0, 0);

**if** (pointsArray.length == 1) **return** **new** Circle(pointsArray[0].x, pointsArray[0].y, 0);

List < Point > points = Arrays.asList(pointsArray);

Collections.shuffle(points);

Circle circle = getCircumCircle(points.**get**(0), points.**get**(1));

**for** (**int** i = 2; i < points.size(); i++)

**if** (!circle.contains(points.**get**(i))) circle = minEnclosingCircleWith1Point(points.subList(0, i), points.**get**(i));

**return** circle;

}

**static** Circle minEnclosingCircleWith1Point(List < Point > points, Point q) {

Circle circle = getCircumCircle(points.**get**(0), q);

**for** (**int** i = 1; i < points.size(); i++)

**if** (!circle.contains(points.**get**(i))) circle = minEnclosingCircleWith2Points(points.subList(0, i), points.**get**(i), q);

**return** circle;

}

**static** Circle minEnclosingCircleWith2Points(List < Point > points, Point q1, Point q2) {

Circle circle = getCircumCircle(q1, q2);

**for** (Point point: points)

**if** (!circle.contains(point)) circle = getCircumCircle(q1, q2, point);

**return** circle;

}

**public** **static** Circle getCircumCircle(Point a, Point b) {

**double** x = (a.x + b.x) / 2.;

**double** y = (a.y + b.y) / 2.;

**double** r = fastHypot(a.x - x, a.y - y);

**return** **new** Circle(x, y, r);

}

**public** **static** Circle getCircumCircle(Point a, Point b, Point c) {

**double** Bx = b.x - a.x;

**double** By = b.y - a.y;

**double** Cx = c.x - a.x;

**double** Cy = c.y - a.y;

**double** d = 2 \* (Bx \* Cy - By \* Cx);

**if** (Math.**abs**(d) < EPS) **return** getCircumCircle(**new** Point(Math.**min**(a.x, Math.**min**(b.x, c.x)), Math.**min**(a.y, Math.**min**(b.y, c.y))), **new** Point(Math.**max**(a.x, Math.**max**(b.x, c.x)), Math.**max**(a.y, Math.**max**(b.y, c.y))));

**double** z1 = Bx \* Bx + By \* By;

**double** z2 = Cx \* Cx + Cy \* Cy;

**double** cx = Cy \* z1 - By \* z2;

**double** cy = Bx \* z2 - Cx \* z1;

**double** x = cx / d;

**double** y = cy / d;

**double** r = fastHypot(x, y);

**return** **new** Circle(x + a.x, y + a.y, r);

}

// Usage example

**public** **static** **void** main(String[] args) {

**Random** rnd = **new** **Random**(1);

**for** (**int** step = 0; step < 100\_000; step++) {

**int** range = 10;

**int** x = rnd.nextInt(range) - range / 2;

**int** y = rnd.nextInt(range) - range / 2;

**int** r = rnd.nextInt(range);

**int** x1 = rnd.nextInt(range) - range / 2;

**int** y1 = rnd.nextInt(range) - range / 2;

**int** x2 = rnd.nextInt(range) - range / 2;

**int** y2 = rnd.nextInt(range) - range / 2;

**if** (x1 == x2 && y1 == y2) **continue**;

Point[] p1 = circleLineIntersection(**new** Circle(x, y, r), **new** Line(**new** Point(x1, y1), **new** Point(x2, y2)));

Point[] p2 = circleLineIntersection2(**new** Circle(x, y, r), **new** Line(**new** Point(x1, y1), **new** Point(x2, y2)));

**if** (p1.length != p2.length || p1.length == 1 && !eq(p1[0], p2[0]) || p1.length == 2 && !(eq(p1[0], p2[0]) && eq(p1[1], p2[1]) || eq(p1[0], p2[1]) && eq(p1[1], p2[0]))) **throw** **new** RuntimeException();

}

}

**static** boolean eq(Point p1, Point p2) {

**return** ! (fastHypot(p1.x - p2.x, p1.y - p2.y) > 1e-9);

}

}

Line Operations

**import** java.util***.*** ***\**** ;

**public** **class** LineGeometry {

**static** **final** **double** EPS = 1e-10;

**public** **static** **int** sign(**double** a) {

**return** a < -EPS ? -1 : a > EPS ? 1 : 0;

}

**public** **static** **class** Point implements Comparable < Point > {

**public** **double** x,

y;

**public** Point(**double** x, **double** y) {

**this**.x = x;

**this**.y = y;

}

**public** Point minus(Point b) {

**return** **new** Point(x - b.x, y - b.y);

}

**public** **double** cross(Point b) {

**return** x \* b.y - y \* b.x;

}

**public** **double** dot(Point b) {

**return** x \* b.x + y \* b.y;

}

**public** Point rotateCCW(**double** angle) {

**return** **new** Point(x \* Math.**cos**(angle) - y \* Math.**sin**(angle), x \* Math.**sin**(angle) + y \* Math.**cos**(angle));

}

@Override

**public** **int** compareTo(Point o) {

// return Double.compare(Math.atan2(y, x), Math.atan2(o.y, o.x));

**return** Double.compare(x, o.x) != 0 ? Double.compare(x, o.x) : Double.compare(y, o.y);

}

}

**public** **static** **class** Line {

**public** **double** a,

b,

c;

**public** Line(**double** a, **double** b, **double** c) {

**this**.a = a;

**this**.b = b;

**this**.c = c;

}

**public** Line(Point p1, Point p2) {

a = +(p1.y - p2.y);

b = -(p1.x - p2.x);

c = p1.x \* p2.y - p2.x \* p1.y;

}

**public** Point intersect(Line line) {

**double** d = a \* line.b - line.a \* b;

**if** (sign(d) == 0) {

**return** **null**;

}

**double** x = -(c \* line.b - line.c \* b) / d;

**double** y = -(a \* line.c - line.a \* c) / d;

**return** **new** Point(x, y);

}

}

// Returns -1 for clockwise, 0 for straight line, 1 for counterclockwise order

**public** **static** **int** orientation(Point a, Point b, Point c) {

Point AB = b.minus(a);

Point AC = c.minus(a);

**return** sign(AB.cross(AC));

}

**public** **static** boolean cw(Point a, Point b, Point c) {

**return** orientation(a, b, c) < 0;

}

**public** **static** boolean ccw(Point a, Point b, Point c) {

**return** orientation(a, b, c) > 0;

}

**public** **static** boolean isCrossIntersect(Point a, Point b, Point c, Point d) {

**return** orientation(a, b, c) \* orientation(a, b, d) < 0 && orientation(c, d, a) \* orientation(c, d, b) < 0;

}

**public** **static** boolean isCrossOrTouchIntersect(Point a, Point b, Point c, Point d) {

**if** (Math.**max**(a.x, b.x) < Math.**min**(c.x, d.x) - EPS || Math.**max**(c.x, d.x) < Math.**min**(a.x, b.x) - EPS || Math.**max**(a.y, b.y) < Math.**min**(c.y, d.y) - EPS || Math.**max**(c.y, d.y) < Math.**min**(a.y, b.y) - EPS) {

**return** **false**;

}

**return** orientation(a, b, c) \* orientation(a, b, d) <= 0 && orientation(c, d, a) \* orientation(c, d, b) <= 0;

}

**public** **static** **double** pointToLineDistance(Point p, Line line) {

**return** Math.**abs**(line.a \* p.x + line.b \* p.y + line.c) / fastHypot(line.a, line.b);

}

**public** **static** **double** fastHypot(**double** x, **double** y) {

**return** Math.**sqrt**(x \* x + y \* y);

}

**public** **static** **double** sqr(**double** x) {

**return** x \* x;

}

**public** **static** **double** angleBetween(Point a, Point b) {

**return** Math.atan2(a.cross(b), a.dot(b));

}

**public** **static** **double** angle(Line line) {

**return** Math.atan2( - line.a, line.b);

}

**public** **static** **double** signedArea(Point[] points) {

**int** n = points.length;

**double** area = 0;

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

area += (points[i].x - points[j].x) \* (points[i].y + points[j].y);

// area += points[i].x \* points[j].y - points[j].x \* points[i].y;

}

**return** area / 2;

}

**public** **static** enum Position {

LEFT,

RIGHT,

BEHIND,

BEYOND,

ORIGIN,

DESTINATION,

BETWEEN

}

// Classifies position of point p against vector a

**public** **static** Position classify(Point p, Point a) {

**int** s = sign(a.cross(p));

**if** (s > 0) {

**return** Position.LEFT;

}

**if** (s < 0) {

**return** Position.RIGHT;

}

**if** (sign(p.x) == 0 && sign(p.y) == 0) {

**return** Position.ORIGIN;

}

**if** (sign(p.x - a.x) == 0 && sign(p.y - a.y) == 0) {

**return** Position.DESTINATION;

}

**if** (a.x \* p.x < 0 || a.y \* p.y < 0) {

**return** Position.BEYOND;

}

**if** (a.x \* a.x + a.y \* a.y < p.x \* p.x + p.y \* p.y) {

**return** Position.BEHIND;

}

**return** Position.BETWEEN;

}

// cuts right part of poly (returns left part)

**public** **static** Point[] convexCut(Point[] poly, Point p1, Point p2) {

**int** n = poly.length;

List < Point > res = **new** ArrayList < >();

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

**int** d1 = orientation(p1, p2, poly[j]);

**int** d2 = orientation(p1, p2, poly[i]);

**if** (d1 >= 0) res.add(poly[j]);

**if** (d1 \* d2 < 0) res.add(**new** Line(p1, p2).intersect(**new** Line(poly[j], poly[i])));

}

**return** res.toArray(**new** Point[res.size()]);

}

// Usage example

**public** **static** **void** main(String[] args) {}

***}***