My\_Template

1. /\*---------------Go Code GO---------------\*/
2. /\*------------Asaduzzaman Herok-----------\*/
4. #include<bits/stdc++.h>
5. **using** **namespace** std;
7. #define     PI              acos(-1.0)
8. #define     O\_O             ios\_base::sync\_with\_stdio(false); cin.tie(NULL)
9. #define     precision(a)    fixed<<setprecision(a)
10. #define     endl            '\n'
11. #define     Flush           cout<<flush
12. #define     LLMX            0x3fffffffffffffff
13. #define     PW2(a)          (long long)1<<a

16. **int** main()
17. {
18. **long** **long** i,j,k,l,m,n,o,p,q,tst,csin=0;

21. **return** 0;
22. }

Arithmatics

1. #include<bits/stdc++.h>
2. **using** **namespace** std;
4. //Power Exponentiation    (x^y)   O(log y);
5. **long** **long**  Power( unsigned **long** **long**  x, **long** **long** y)
6. {
7. **long** **long** res = 1;     // Initialize result
9. **while** (y > 0)
10. {
11. // If y is odd, multiply x with result
12. **if** (y & 1)
13. res = res\*x;
14. // y must be even now
15. y = y>>1; // y = y/2
16. x = x\*x;  // Change x to x^2
17. }
18. **return** res;
19. }
21. **long** **long** BigMod(unsigned **long** **long** x, unsigned **long** **long**  y, **long** **long**  MOD)
22. {
23. **long** **long**  res = 1;      // Initialize result
25. x = x % MOD;  // Update x if it is more than or
26. // equal to MOD
27. **while** (y > 0)
28. {
29. // If y is odd, multiply x with result
30. **if** (y & 1)
31. res = (res\*x) % MOD;
33. // y must be even now
34. y = y>>1; // y = y/2
35. x = (x\*x) % MOD;
36. }
37. **return** res;
38. }
40. **long** **long**  Inverse\_Modulo(**long** **long**  a,**long** **long**  MOD)
41. {
42. **return** BigMod(a,MOD-2,MOD);
43. }

46. //Modular multiplication without overflow;
47. **long** **long**  ModularMUL(**long** **long** a, **long** **long** b, **long** **long** mod)
48. {
49. **long** **long** res = 0; // Initialize result
50. a = a % mod;
51. **while** (b > 0)
52. {
53. // If b is odd, add 'a' to result
54. **if** (b % 2 == 1)
55. res = (res + a) % mod;
56. // Multiply 'a' with 2
57. a = (a \* 2) % mod;
58. // Divide b by 2
59. b /= 2;
60. }
61. // Return result
62. **return** res % mod;
63. }

Segment Tree

1. #include<bits/stdc++.h>
2. **using** **namespace** std;
3. #define LLMX 0x7fffffffffffffff
5. **long** **long**  arr[100005];
6. **long** **long**  SegTr[100005\*3];
7. **int** MAX = INT\_MAX;
9. **void** Build(**int** nod, **int** lf, **int** rt)              //call with nod = 1; lft=1; rgt=N
10. {
11. **if** (lf == rt)
12. {
13. SegTr[nod] = arr[lf];
14. **return**;
15. }
16. **int** mid = (lf + rt)/2;
18. Build(2\*nod, lf, mid);
19. Build(2\*nod+1, mid + 1, rt);
21. SegTr[nod] = min(SegTr[2\*nod],SegTr[2\*nod+1]);        //range min
22. //SegTr[nod] = SegTr[2\*nod] + SegTr[2\*nod+1];      //range sum
23. }
24. **int** query(**int** nod, **int** lf, **int** rt, **int** qlf, **int** qrt)
25. {
26. **if** (qlf > rt || qrt < lf)                       // Non overlap;
27. //return 0;                                       //range Sum
28. **return** MAX;                                     //range min;
30. **if** (qlf<=lf && rt<=qrt)               //   both left & right falls under range;
31. **return** SegTr[nod];
33. **int** mid = (lf + rt)/2;
34. **int** a = query(2\*nod,lf,mid,qlf,qrt);
35. **int** b = query(2\*nod+1,mid+1,rt,qlf,qrt);
36. **return** min(a,b);    //range min
37. //return query(a+b);         //range sum
38. }

41. **void** update(**int** nod, **int** lf, **int** rt, **int** idx, **int** val)
42. {
43. **if** (idx > rt || idx < lf)
44. **return**;                                 //out of range
45. **if** (lf == idx && rt == idx)                //found the node
46. {
47. SegTr[nod] = val;
48. **return**;
49. }
50. **int** mid = (lf + rt)/2;
52. update(2\*nod, lf, mid, idx, val);
53. update(2\*nod+1, mid + 1, rt, idx, val);
55. //SegTr[nod] = SegTr[2\*nod] + SegTr[2\*nod+1];          //range sum
56. SegTr[nod] = min(SegTr[2\*nod],SegTr[2\*nod+1]);         //range min
57. }
58. **int** main()
59. {
61. **int**  tst,csn=1,i,j,k,l,m,n,o,p;
62. scanf("%d",&tst);
63. **while**(tst--)
64. {
65. scanf("%d %d",&n,&p);
66. **for**(i=1; i<=n; i++) scanf("%d",&arr[i]);
67. Build(1,1,n);
68. printf("Case %d:\n",csn++);
69. **while**(p--)
70. {
71. scanf("%d %d",&k,&l);
72. m=query(1,1,n,k,l);
73. printf("%d\n",m);
74. }
75. }
76. **return** 0;
77. }

KMP

1. **int** \_next[1000000+1] ;
3. **void** failure( string T ) {
4. **int** \_left = 0 , \_right ;
5. \_next[0] = 0 ;
6. **for**( \_right = 1 ; \_right < T.size() ; \_right ++ ) {
7. **while**( \_left > 0  && T[\_left] != T[\_right] ) {
8. \_left = \_next[\_left-1]  ;
9. }
10. **if**( T[\_left] == T[\_right] )\_left++ ;
11. \_next[\_right] = \_left ;
12. }
13. }
15. **int** KMP(string T,string P){
16. **int** \_left = 0 , \_right , cnt = 0 ;
17. **for**( \_right = 0 ; \_right < T.size() ; \_right ++ ){
18. **while**( \_left > 0 && P[\_left] != T[\_right] ) \_left = \_next[\_left-1] ;
19. **if**( P[\_left] == T[\_right] )\_left ++ ;
20. **if**( \_left == P.size() ) {
21. cnt ++ ;
22. \_left = \_next[\_left-1] ;
23. }
24. }
25. **return** cnt ;
26. }

Z-Algo

1. **int** M[1000000] ;
2. **int** z[1000000] ;
3. string s ;
5. **void** calc() {
6. memset(z,0,**sizeof** z) ;
7. **int** L = 0, R = 0;
8. **int** n = s.size() ;
9. **for** (**int** i = 1; i < n ; i++) {
10. **if** (i > R) {
11. L = R = i;
12. **while** (R < n && s[R-L] == s[R]) R++;
13. z[i] = R-L;
14. R--;
15. } **else** {
16. **int** k = i-L;
17. **if** (z[k] < R-i+1) z[i] = z[k];
18. **else** {
19. L = i;
20. **while** (R < n && s[R-L] == s[R]) R++;
21. z[i] = R-L;
22. R--;
23. }
24. }
25. }
26. }

LCA

1. //LCA using sparse table
2. //Complexity: O(NlgN,lgN)
3. #define EX 100002
4. **int** L[EX];
5. **int** P[EX][22];
6. **int** T[EX];
7. vector<**int**>g[EX];
8. **void** dfs(**int** from,**int** u,**int** dep)
9. {
10. T[u]=from;
11. L[u]=dep;
12. **for**(**int** i=0;i<(**int**)g[u].size();i++)
13. {
14. **int** v=g[u][i];
15. **if**(v==from) **continue**;
16. dfs(u,v,dep+1);
17. }
18. }
20. **int** lca\_query(**int** N, **int** p, **int** q)
21. {
22. **int** tmp, log, i;
24. **if** (L[p] < L[q])
25. tmp = p, p = q, q = tmp;
27. log=1;
28. **while**(1) {
29. **int** next=log+1;
30. **if**((1<<next)>L[p])**break**;
31. log++;
33. }
35. **for** (i = log; i >= 0; i--)
36. **if** (L[p] - (1 << i) >= L[q])
37. p = P[p][i];
39. **if** (p == q)
40. **return** p;
42. **for** (i = log; i >= 0; i--)
43. **if** (P[p][i] != -1 && P[p][i] != P[q][i])
44. p = P[p][i], q = P[q][i];
46. **return** T[p];
47. }
49. **void** lca\_init(**int** N)
50. {
51. memset (P,-1,**sizeof**(P));
52. **int** i, j;
53. **for** (i = 0; i < N; i++)
54. P[i][0] = T[i];
55. **for** (j = 1; 1 << j < N; j++)
56. **for** (i = 0; i < N; i++)
57. **if** (P[i][j - 1] != -1)
58. P[i][j] = P[P[i][j - 1]][j - 1];
59. }
61. **int** main(**void**) {
62. g[0].pb(1);
63. g[0].pb(2);
64. g[2].pb(3);
65. g[2].pb(4);
66. dfs(0, 0, 0);
67. lca\_init(5);
68. printf( "%d\n", lca\_query(5,3,4) );
69. **return** 0;
70. }

Matrix Exponentiation

1. **template**<**int** N> **class** Matrix {
3. **public**:
4. **long** **long**  arr[N][N];
6. Matrix() {
7. **for**( **int** i = 0 ; i < N ; i ++ ) {
8. **for**( **int** j = 0 ; j < N ; j++ ) {
9. arr[i][j] = 0 ;
10. }
11. }
12. }
14. Matrix<N> operator \*(**const** Matrix<N> &in) {
15. Matrix<N> ret ;
16. **for**( **int** i = 0 ; i < N ; i++ ) {
17. **for**( **int** j = 0 ; j < N ; j++ )
18. **for**( **int** k = 0 ; k < N ; k++ ) {
19. ret.arr[i][j] += (arr[i][k] ) \* ( in.arr[k][j] ) ;
20. ret.arr[i][j] %= 10000 ;
21. }
22. }
23. **return** ret ;
24. }
26. Matrix<N> operator ^( **int** POW ) {
28. Matrix<N> ret ;
29. **for**( **int** i = 0 ; i < N ; i++ ) {
30. ret.arr[i][i] = 1 ;
31. }
33. Matrix<N> ME = \***this** ;
35. **while**( POW ) {
37. **if**( POW&1 ) {
38. ret = ret \* ME ;
39. }
40. ME = ME \* ME ;
41. POW >>= 1 ;
43. }
44. **return** ret ;
45. }
46. };

Segmented Sieve

1. /\*
2. Generates primes within interval [a, b] when b - a <= 100000
3. and 1 <= a <= b <= 2147483647
4. \*/
5. **int** base[MAX>>6], segment[RNG>>6], primes[LEN], prlen;
7. #define chkC(x,n) (x[n>>6]&(1<<((n>>1)&31)))
8. #define setC(x,n) (x[n>>6]|=(1<<((n>>1)&31)))
10. **void** sieve() {
11. **int** i, j, k;
12. **for**(i=3; i<LMT; i+=2) **if**(!chkC(base, i)) **for**(j=i\*i, k=i<<1; j<MAX; j+=k) setC(base, j);
13. **for**(i=3, prlen=0; i<MAX; i+=2) **if**(!chkC(base, i)) primes[prlen++] = i;
14. }
16. **int** segmented\_sieve(**int** a, **int** b) {
17. **int** rt, i, k, cnt = (a<=2 && 2<=b)? 1 : 0;
18. **if**(b<2) **return** 0;
19. **if**(a<3) a = 3;
20. **if**(a%2==0) a++;
21. memset(segment, 0, **sizeof** segment);
22. **for**(i=0, rt=(**int**)sqrt((**double**)b); i < prlen && primes[i] <= rt; i++) {
23. unsigned j = primes[i] \* ( (a+primes[i]-1) / primes[i] );
24. **if**(j%2==0) j += primes[i];
25. **for**(k=primes[i]<<1; j<=b; j+=k) **if**(j!=primes[i]) setC(segment, (j-a));
26. }
27. **for**(i=0; i<=b-a; i+=2) **if**(!chkC(segment, i)) cnt++;
28. **return** cnt;
29. }

Sieve

1. #define MAX 10000000
2. unsigned flag[MAX/64];
3. vector<**long** **long**>prime ;
5. #define chkC(n) (flag[n>>6]&(1<<((n>>1)&31)))
6. #define setC(n) (flag[n>>6]|=(1<<((n>>1)&31)))
7. **int** lim;
8. **void** sieve() {
9. unsigned i, j, k;
10. flag[0]|=0;
11. **int** sqrtN = sqrt(MAX) ;
12. **for**(i=3; i<= sqrtN ; i+=2)
13. **if**(!chkC(i))
14. **for**(j=i\*i,k=i<<1; j<MAX; j+=k)
15. setC(j);
16. prime.push\_back(2);
18. **for**(i=3; i<MAX; i+=2)
19. **if**(!chkC(i))
20. prime.push\_back(i) ;
21. lim = prime.size() ;
22. }

nCr

1. **void** precalc() {
2. **for**(**int** i = 1 ; i <= 1000 ; i++) {
3. nCr[i][0]=1;
4. nCr[i][1]=i;
5. **for**(**int** j = 2 ; j <= i ; j++ ) {
6. nCr[i][j] = ( nCr[i-1][j-1] + nCr[i-1][j] ) % MOD;
7. }
8. }
9. }

1 to N number of divisors

1. **int** D[1000010];
2. **void** DivisorGenerate()
3. {
4. **int** i,j,val,N,M,count;
6. D[1]=1;
7. **for**(i=2;i<=1000000;i++)
8. {
9. N=M=i;
10. val=sqrt(N)+1;
11. **for**(j=0;primes[j]<val;j++)
12. {
13. **if**(M%primes[j]==0)
14. {
15. count=0;
16. **while**(M%primes[j]==0)
17. {
18. M/=primes[j];
19. count++;
20. }
21. D[N]=(count+1)\*D[M];
22. **break**;
23. }
24. }
25. **if**(M==N)        //Special Case if N equal prime
26. {
27. D[N]=2;
28. }
29. }
30. }

Euler phi

1. #define MAXN 3000000
2. **int** phi[MAXN + 1], prime[MAXN/10], sz;
3. bitset <MAXN + 1> mark;
5. **for** (**int** i = 2; i <= MAXN; i++ ){
6. **if**(!mark[i]){
7. phi[i] = i-1;
8. prime[sz++]= i;
9. }
10. **for** (**int** j=0; j<sz && prime[j]\*i <= MAXN; j++ ){
11. mark[prime[j]\*i]=1;
12. **if**(i%prime[j]==0){
13. phi[i\*prime[j]] = phi[i]\*prime[j];
14. **break**;
15. }
16. **else** phi[i\*prime[j]] = phi[i]\*(prime[j]-1 );
17. }
18. }

Articulation

1. #define mx 500005
3. vector<**long** **long**>g[mx];
4. vector<pair<**long** **long**,**long** **long**> >ed;
5. **long** **long** visited[mx];
6. **bool** articulation[mx];
7. **long** **long** low[mx],d[mx],parent[mx],x[500005],vis[500005],root[500005],maxx[500005],sz[500005];
8. **long** **long** t;
10. **void** articulating(**long** **long** s,**bool** root)
11. {
12. t++;
13. low[s]=d[s]=t;
14. visited[s]=1;
15. **long** **long** child=0;
17. **for**(**long** **long** i=0; i<g[s].size(); i++)
18. {
19. **long** **long** v=g[s][i];
21. **if**(visited[v])
22. {
23. low[s]=min(low[s],d[v]);
24. }
25. **if**(!visited[v])
26. {
27. parent[s]=v;
28. articulating(v,**false**);
30. **if**(d[s]<=low[v]&&!root)
31. {
32. articulation[s]=**true**;
33. }
34. low[s]=min(low[s],low[v]);
35. child++;
36. }
37. **if**(child>1&&root)
38. {
39. articulation[s]=**true**;
40. }
42. }
44. }
45. **void** setting()
46. {
47. **for**(**long** **long** i=0; i<mx; i++)
48. {
49. g[i].clear();
50. visited[i]=articulation[i]=low[i]=d[i]=parent[i]=0;
51. t=0;
52. }
53. }
55. **int** main()
56. {
57. fastio;
58. setting();
59. **long** **long** n,m,f=0,k,p,i,j,q;
60. cin>>n>>m;
61. **long** **long** u,v;
62. **for**(i=0; i<m; i++)
63. {
64. cin>>u>>v;
65. ed.push\_back({u,v});
66. g[u].push\_back(v);
67. g[v].push\_back(u);
68. }
69. **for**(**long** **long** i=1; i<=n; i++)
70. {
71. **if**(visited[i]==0)
72. articulating(i,**true**);
74. }
75. //IF artculation[i] is true then its a articulation point.
76. **return** 0;
78. }

Sparse Table

1. **int** st[MAXN][K + 1];      //K=log2(MAXN)+1;
3. **for** (**int** i = 0; i < N; i++)
4. st[i][0] = f(array[i]);
6. **for** (**int** j = 1; j <= K; j++)
7. **for** (**int** i = 0; i + (1 << j) <= N; i++)
8. st[i][j] = f(st[i][j-1], st[i + (1 << (j - 1))][j - 1]);
10. //Range sum query
11. **long** **long** sum = 0;
12. **for** (**int** j = K; j >= 0; j--) {
13. **if** ((1 << j) <= R - L + 1) {
14. sum += st[L][j];
15. L += 1 << j;
16. }
17. }
19. //Range minimum query
20. **int** j = log[R - L + 1];     //precalculated log(R-L+1)
21. **int** minimum = min(st[L][j], st[R - (1 << j) + 1][j]);

LIS

1. **int** longestIncreasingSubsequence(vector<**int**> arr) {
3. **long** **long** Temp[arr.size()+5], Re[arr.size()+5]={-1};
4. **long** **long** nowLis=0;
5. Temp[0]=0;
7. **for**(**int** i =1; i<arr.size(); i++)
8. {
9. **if**(arr[i]>=arr[Temp[nowLis]])
10. {
11. Re[i]=Temp[nowLis];
12. nowLis++;
13. Temp[nowLis]=i;
14. }
15. **else** **if**(arr[Temp[0]]>=arr[i])
16. {
17. Temp[0]=i;
18. }
19. **else**
20. {
21. **int** lw=0, hi=nowLis, md, pre=0;
22. **while**(lw<=hi)
23. {
24. md=(lw+hi)/2;
25. **if**(arr[Temp[md]]<arr[i])
26. {
27. lw=md+1;
28. }
29. **else**
30. {
31. pre=md;
32. hi=md-1;
33. }
34. }
35. Re[i]=Temp[pre-1];
36. Temp[pre]=i;
37. }
38. }
39. **return** nowLis+1;
40. }
41. **void** printLis(**long** **long** Re[], vector<**int**> &arr, **int** idx)
42. {
43. **if**(idx==-1) **return**;
44. printLis(Re, arr, Re[idx]);
45. cout<<arr[idx]<<' ';
47. }