Team Name: DU_nextTime

Mushfiqur Rahman Chowdhury
Md. Inzamam-Ul Haque Sobuz
Labib Muntasir

Articulation Bridge

```
int N; //Number of Nodes (0.....n-1)
vector <int> List[Nn];  // N = MAX of n
vector <bool> vis;
vector <pair <int, int> > articulationBridges;
int timer;
int inTime[Nn], lowTime[Nn];

void DFSAB(int node, int parent) {
    vis[node] = true;
    inTime[node] = lowTime[node] = timer++;
    for (int i = 0;i < List[node].size(); ++i) {
        int child = List[ node ][ i ];
        if (child == parent)   continue;
        if (vis[child]) lowTime[node] = min(lowTime[node], inTime[child]);
        else {
            DFSAB(child, node);
            lowTime[node] = min(lowTime[child], lowTime[node]);
            if ( lowTime[child] > inTime[node])
                articulationBridges.push_back({min(node, child) + 1, max(node,
child) + 1});
        }
    }
}

void findArticulationBridge() { /* timer -> 0  and vis -> false */
    for (int i = 0; i < N; ++i) {
        if ( !vis[i] ) DFSAB(i, -1);
    }
}
```

Articulation Point

```
int N; //Number of Nodes (0.....n-1)
vector <int> List[Nn];  // N = MAX of n
vector <bool> vis;
set <int> articulationPoints; // use a flag array if needed
int timer;
int inTime[Nn], lowTime[Nn];

void DFSAP(int node, int parent) {
    vis[node] = true;
    inTime[node] = lowTime[node] = timer++;
    int subTree = 0;
    for (int i = 0; i < List[node].size(); ++i) {
        int child = List[node][i];
        if (child == parent)   continue;
```

```
        if (vis[child]) lowTime[node] = min(lowTime[node], inTime[child]);
        else {
            DFSAP(child, node);
            lowTime[node] = min(lowTime[child], lowTime[node]);
            if (parent != -1 && lowTime[child] >= inTime[node])
                articulationPoints.insert(node);
            ++subTree;
        }
    }

    if (parent == -1 && subTree > 1)
        articulationPoints.insert(node);
}

void findArticulationPoint() {
    /* timer -> 0  and vis -> false and articulationPoints -> clear */
    for (int i = 0; i < N; ++i) {
        if (!vis[i])  DFSAP(i, -1);
    }
}
```

Binary Search Tree(BST)

```
struct Node {
    Node *left, *right;
    int val, cnt;
    Node (int val) {
        this->val = val; this->cnt = 1; this->left = this->right = NULL;
    }
};

class BST {
    public:
        Node *info = NULL;
        Node* insert(Node *root, int val) {
            if (root == NULL)           return new Node(val);
            else if (root->val == val)  root->cnt++;
            else if (root->val > val)   root->left = insert( root->left, val );
            else              root->cnt++, root->right = insert(root->right, val);
            return root;
        }

        int search(Node *root, long tar) {
            if (root == NULL)           return 0;
            else if (tar == root->val)  return root->cnt;
            else if (root->val > tar)   return root->cnt + search(root->left, tar);
            else                        return search(root->right, tar);
        }
```

```
        void print(Node *root) {
            if (root == NULL)  return;
            cout << root->val;
            print(root->left);
            print(root->right);
        }
};
```

0-1 Knapsack

```
int knapsack(int need) {    // DP[need] -> memo,  P -> profit,  W -> weight
    for (int i = 1; i <= W.size(); ++i) {
        for (int j = need; j >= 0; --j)
            if (j - W[i] >= 0)  DP[j] = max(DP[j], DP[j - W[i]] + P[i]);
    }
    return DP[need];
}
```

Matrix Chain Multiplication(MCM)

```
ll marge_cost( ll L, ll R, ll M ) {
    // return cost for merging, mats have int row and int col
    return mats[L].row * mats[M].col * mats[R].col;
}

ll make_MCM(ll L, ll R, ll cost = 1e18) {
    // DP[mats.size][mats.size] -> memo, Base case -> for interval size 1,2..
    if (L >= R)             return 0;
    if (DP[L][R] != -1)  return DP[L][R];
    for (ll i = L; i < R; ++i)
       cost = min(cost, make_MCM(L, i) + make_MCM(i + 1, R) + marge_cost(L, R, i));
    return DP[L][R] = cost;
}
```

Minimum Vertex Cover

```
void DFSMVC(ll u) {
    //DP[number of vertex][2]  -> memo,  vis[number of vertex] -> is it visited?
    vis[u] = true;
    ll cost0 = 0; // if v is not taken
    ll cost1 = 1; // if v is taken

    for (int i = 0; i < adj[u].size(); ++i) {
        ll v = adj[u][i];
        if (!vis[v]) {
            DFSMVC(v);
            cost0 += DP[v][1];
            cost1 += min(DP[v][0], DP[v][1]);
        }
    }
```

```
    DP[u][0] = cost0;
    DP[u][1] = cost1;
}

ll MVC() {
 //suppose root is 1; DP[v][1] mane if v is taken, DP[v][0] mane if v is not taken
    DFSMVC(1); // vertex are[ 1....n ]
    return min(DP[1][0], DP[1][1]);
}
```

Fenwick Tree(BIT)

```
class FenwickTree { // from cp-algorithm
    public:
        vector <int> bit;  // 0-based binary indexed tree
        int sz;

        FenwickTree(int sz) {
            this->sz = sz;
            bit.assign(sz, 0);
        }

        FenwickTree(vector <int> a) : FenwickTree(a.size()) {
            for (size_t i = 0; i < a.size(); ++i) add(i, a[i]);
        }

        int sum(int r) {
            int ret = 0;
            for (; r >= 0; r = (r & (r + 1)) - 1) ret += bit[r];
            return ret;
        }

        int sum(int l, int r) {
            return sum(r) - sum(l - 1);
        }

        void add( int idx, int delta ) {
            for (; idx < n; idx = idx | (idx + 1))  bit[idx] += delta;
        }
};
```

Segment Tree

```
class SegmentTree {
    #define Lc(idx)      idx * 2
    #define Rc(idx)      idx * 2 + 1
    public:
        struct node {
            int value, lazy;
```

```
            node() {
                this->value = ??;
                this->lazy = ??;
            }
        };
        vector <node> segT;
        vector <int> A;

        SegmentTree(int sz) { // need to clear!
            segT.resize(4 * sz + 10);
            A.resize(sz + 1); /* 1-base index */
        }

        node Merge(node L, node R) {
            node F;
            F = ??
            return F;
        }

        void Relax(int L, int R, int idx) {
            //Do something
            segT[idx].lazy = ??; //after Relaxing
        }

        void MakeSegmentTree(int L, int R, int idx) {
            if (L == R) {
                segT[idx].value = ??;
                return;
            }
            int M = (L + R) / 2;
            MakeSegmentTree(L, M, Lc(idx));
            MakeSegmentTree(M + 1, R, Rc(idx));
            segT[idx] = Merge(segT[Lc(idx)], segT[Rc(idx)]);
        }

        node RangeQuery(int L, int R, int idx, int l, int r) {
            Relax(L, R, idx);
            node F;
            if (L > r || R < l)     return F;
            if (L >= l && R <= r)   return segT[idx];
            int M = (L + R) / 2;
            F = Merge(RangeQuery(L, M, Lc(idx), l, r), RangeQuery(M + 1, R,
Rc(idx), l, r));
            segT[idx] = Merge(segT[Lc(idx)], segT[Rc(idx)]); //is it useful?
            return F;
        }
```

```
        void RangeUpdate(int L, int R, int idx, int l, int r, int lz) {
            Relax(L, R, idx);
            if (L > r || R < l)     return;
            if (L >= l && R <= r) {
                // Do something
                segT[idx].lazy = ??;
                Relax(L, R, idx);
                return;
            }
            int M = (L + R) / 2;
            RangeUpdate(L, M, Lc(idx), l, r, lz);
            RangeUpdate(M + 1, R, Rc(idx ), l, r, lz);
            segT[idx] = Merge( segT[Lc(idx)], segT[Rc(idx)]);
        }
};
```

Sparse Table

```
class SparseTable {  /* Min / Max -> OK, if Sum -> Use SegmentTree  */
    #define MX          200010
    #define LOG         22
    public:
        int LOGBaseTwo[MX];
        int SpT[MX][LOG];
        int MinOrMax = -1; // if Min -> 0, Max -> 1;

        SparseTable(int OP) {
            this->MinOrMax = OP;
        }

        void MakeSparseTable(vector <int> &A) {
            for (int i = 0; i < A.size(); ++i)
                SpT[i][0] = A[i];
            for (int i = 1; i < LOG; ++i) {
                for (int j = 0; j < A.size(); ++j)
                    SpT[j][i] = Merge(SpT[j][i - 1], SpT[min((int)A.size() - 1, j +
(1 << (i - 1)))][i - 1]);
            }
            MakeLog();
        }

        int Merge(int A, int B) {
            if (MinOrMax)  return max(A, B);
            else           return min(A, B);
        }

        void MakeLog() {
            LOGBaseTwo[1] = 0;
```

```
            for (int i = 2; i < MX; ++i)
                LOGBaseTwo[i] = LOGBaseTwo[i / 2] + 1;
        }

        int GetNeed(int L, int R) {
            assert(R >= L);
            int Log = LOGBaseTwo[R - L + 1];
            return Merge(SpT[L][Log], SpT[R - (1 << Log) + 1][Log]);
        }
};
```

MOs Algorithm

```
int block_size;
class MOsALGO {   /* 0-base index */
    public:
        struct query {
            int L, R, idx;
            bool operator <(query other)const {
                return (make_pair(L / block_size, R) < make_pair(other.L /
block_size, other.R));
            }
        };

        vector <int> Cnt, Arr;
        vector <query> Qry;

        MOsALGO(int sz, int nq) {
            block_size = (int) sqrt(sz + 0.0) + 1;
            Qry.resize(nq);
            Arr.resize(sz);
            Cnt.resize(1000009);
        }

        ll add(int idx) {
            ll res = (ll) Cnt[Arr[idx]] * Cnt[Arr[idx]];
            Cnt[Arr[idx ]]++;
            res = (ll) Cnt[Arr[idx]] * Cnt[Arr[idx]] - res;
            return res * Arr[idx];
        }

        ll del(int idx) {
            ll res = (ll) Cnt[Arr[idx]] * Cnt[Arr[idx]];
            Cnt[Arr[idx]]--;
            res = (ll) Cnt[Arr[idx]] * Cnt[Arr[idx]] - res;
            return res * Arr[idx];
        }
```

```
        void getANS(vector <ll> &ANS) {
            ll POWER = 0;
            int CL = -1, CR = -1;
            sort(Qry.begin(), Qry.end());
            for (int i = 0; i < ANS.size(); ++i) {
                while (CR < Qry[i].R)        POWER += add(++CR);
                while (CR > Qry[i].R)        POWER += del(CR--);

                while (CL + 1 < Qry[i].L)    POWER += del(++CL);
                while (CL >= Qry[i].L)       POWER += add(CL--);

                ANS[Qry[i].idx] = POWER;
            }
        }
};
```

Square Root Decomposition

```
int block_size = ??;
int Block[block_size + 5];

int getBlock(int idx) {
    return (idx + block_size - 1) / block_size;    //for 1-base index
    return idx / block_size;                       //for 0-base index
}

int getQueryAns(int L, int R) { //0-base index
    int ANS = 0, CL = L / block_size, CR = R / block_size;
    if (CL == CR) {
        for (int i = L; i <= R; ++i)  ANS += ArrName[i];
    }
    else {
        for (int i = L, LM = (CL + 1) * block_size - 1; i <= LM; ++i)
            ANS += ArrName[i];
        for (int i = CL + 1; i <= CR - 1; ++i)      ANS += Block[i];
        for (int i = CR * block_size; i <= R; ++i) ANS += ArrName[i];
    }

    return ANS;
}
//Update :   Block[ idx / block_size ] += ??
```

Z Value

```
class ZFunction {
    public:
        string S;
        vector <int> Z;
        ZFunction(string S) {
```

```
            this->S = S;
            Z.resize(S.size());
            Z[0] = 0;
        }

        void calZvalue() {
            int L = 0, R = 0, len = S.size();
            for (int i = 1; i < len; ++i) {
                Z[i] = 0;
                if (i <= R)     Z[i] = min(Z[i - L], R - i + 1);
                while (i + Z[i] < len && S[i + Z[i]] == S[Z[i]]) Z[i]++;
                if (i + Z[i] - 1 > R)      L = i, R = i + Z[i] - 1;
            }
        }
};
```

Union Find

```
class UnionFind {
      public:
            vector <int> Par, Siz;
            int StartingGroupCount, MaxGroupSize = ??;  /* 1 or 0 */
            UnionFind(int sz) {
                  this->StartingGroupCount = sz;
                  for (int i = 0; i < sz; ++i) {
                        Par.push_back(i);
                        Siz.push_back(??);  /* 1 or 0 */
                  }
            }

            int FindRoot(int u) {
                  if (Par[u] != u) Par[u] = FindRoot(Par[u] );
                  return Par[u];
            }

            void Merge(int u, int v) {
                  if (FindRoot(u) != FindRoot(v)) {
                        if (Siz[Par[u]] <= Siz[Par[v]])   swap(u, v);
                        Siz[Par[u]] += Siz[Par[v]];
                        MaxGroupSize = max(MaxGroupSize, Siz[Par[u]]);
                        Par[Par[v]] = Par[u];
                  }
            }

            int GetMaxGroupSize() {
                  return MaxGroupSize;
            }
```

```
            int GetNumberOfGroup() {
                  vector <bool> Yes(StartingGroupCount, false);
                  for (int i = 0; i < StartingGroupCount; ++i) { // ??
                      if (Siz[i] == ??)  Yes[FindRoot(i)] = true;
                  }
                  return count(Yes.begin(), Yes.end(), true);
            }
};
```

Trie

```
class TrieNode {    //from http://www.shafaetsplanet.com/?p=1679
      public:
            struct node {  // only lower-case letter
                  bool endmark;
                  node* next[26 + 1];
                  node() {
                        endmark = false;
                        for (int i = 0; i < 26; ++i)
                        next[i] = NULL;
                  }
            } *root;

            void insert(string str) {
                  int len = str.size();
                  node* curr = root;
                  for (int i = 0; i < len; ++i) {
                        int id = str[i] - 'a';
                        if ( curr->next[id] == NULL )
                        curr->next[id] = new node();
                        curr = curr->next[id];
                  }
                  curr->endmark = true;
            }

            bool search(string str) {
                  int len = str.size();
                  node* curr = root;
                  for (int i = 0; i < len; ++i) {
                        int id = str[i] - 'a';
                        if (curr->next[id] == NULL )
                        return false;
                        curr = curr->next[id];
                  }
                  return curr->endmark;
            }

            void del(node* cur) {
```

```
                  for (int i = 0; i < 26; i++)
                        if (cur->next[i])
                        del(cur->next[i]);

                  delete(cur);
            }

            TrieNode() {
                  root = new node();
            }
};
```

Histogram

```
ll forOneRowHistogram()
{// for ar[ sz ] array
      ll sz = sizeof ar;
      stack <pair <ll, ll>> sk, sr;
      ll left[sz], right[sz];
      memset(left, 0, sizeof left);
      memset(right, 0, sizeof right);

      for (ll i = 0; i < sz; i++) {
            left[i] = -1;
            ll value = ar[i];
            while(sk.empty() == false && sk.top().second >= value)  sk.pop();
            if(sk.empty() == false) left[i] = sk.top().first;
            sk.push({i, value});
      }

      for (ll i = sz - 1; i >= 0; i--) {
            right[i] = sz;
            ll value = ar[i];
            while (sr.empty() == false && sr.top().second >= value)  sr.pop();
            if (sr.empty() == false) right[i] = sr.top().first;
            sr.push({i, value});
      }

      ll res = 0;
      for (ll i = 0; i < sz; i++) {
            if ((right[i] - left[i] - 1) * ar[i]) res = max(2 * ((right[i] -
left[i] - 1) + ar[i]), res);  // calculate part
      }

      return res;
}
```

inzamam_inz

```
/* Graph Move */
const int fx[] = {+1, -1, +0, +0};
const int fy[] = {+0, +0, +1, -1};
const int fx[] = {+0, +0, +1, -1, -1, +1, -1, +1};   // Kings Move
const int fy[] = {-1, +1, +0, +0, +1, +1, -1, -1};   // Kings Move
const int fx[] = {-2, -2, -1, -1,  1,  1,  2,  2};    // Knights Move
const int fy[] = {-1,  1, -2,  2, -2,  2, -1,  1};    // Knights Move

/* FastIO */
ios_base::sync_with_stdio(false);
cin.tie(NULL);
cout.tie(NULL);
cout << setprecision(12);
```

Lowest Common Ancestor

```
const int LOG = 20;

vector <int> List[N];  // Tree's Adj List
vector <int> Dist(N);
int Ancestor[N][LOG], inTime[N], outTime[N], Timer;

void DFS(int node, int parent) {
      Dist[node] = Dist[parent] + 1;
      inTime[node] = Timer;
      Ancestor[node][0] = parent;
      for (int i = 1; i < LOG; ++i)
            Ancestor[node][i] = Ancestor[Ancestor[node][i - 1]][i - 1];
      for (int i = 0; i < List[node].size(); ++i) {
            if (List[node][i] != parent)
                  DFS(List[node][i], node);
      }
      outTime[node] = ++Timer;
}

bool is_ancestor(int u, int v) {
      return inTime[u] <= inTime[v] && outTime[u] >= outTime[v];
}

int LCA(int u, int v) { // Lowest Common Ancestor(LCA)
      if (is_ancestor(u, v))    return u;
      if (is_ancestor(v, u))    return v;
      for (int i = 19; i >= 0; --i) {
            if (!is_ancestor(Ancestor[u][i], v))   u = Ancestor[u][i];
      }
      return Ancestor[ u ][ 0 ];
}
//rooted tree(1-index based) -> DFS( root, 0 ); After DFS -> outTime[ 0 ] = Timer;
```

Hashing

```
const ll MOD1 = 1e9 + 7;
const ll MOD2 = 1e7 + 9;
const ll POW1 = 313;
const ll POW2 = 373;

ll HashST[N][2];
ll POWER1[N], POWER2[N];

void HASH(string &str) {
	HashST[0][0] = HashST[0][1] = 0;
	for (int i = 0; i < str.size(); ++i) {
		HashST[i + 1][0] = (HashST[i][0] * POW1 + str[i]) % MOD1;
		HashST[i + 1][1] = (HashST[i][1] * POW2 + str[i]) % MOD2;
	}
}

void init() {
	POWER1[0] = POWER2[0] = 1;
	for (int i = 1; i < N; ++i ) {
		POWER1[i] = (POWER1[i - 1] * POW1) % MOD1;
		POWER2[i] = ( POWER2[i - 1] * POW2) % MOD2;
	}
}

#include <vector>
#include <string>

using namespace std;

class HashedString {
	private:
		// change M and P if you want
		static const long long M = 1e9 + 9;
		static const long long P = 9973;

		// pow[i] contains P^i % M
		static vector<long long> pow;

		// p_hash[i] is the hash of the first i characters of the given string
		vector<long long> p_hash;
	public:
		HashedString(const string& s) : p_hash(s.size() + 1) {
			while (pow.size() < s.size()) {
				pow.push_back((pow.back() * P) % M);
			}
```

```
                p_hash[0] = 0;
                for (int i = 0; i < s.size(); i++) {
                        p_hash[i + 1] = ((p_hash[i] * P) % M + s[i]) % M;
                }
        }

        long long getHash(int start, int end) {
                long long raw_val = (
                        p_hash[end + 1] - (p_hash[start] * pow[end - start + 1])
                );
                return (raw_val % M + M) % M;
        }
};
vector<long long> HashedString::pow = {1};
```

nCr

```
ll F[N], FI[N];
void fact() {       /* O( N ) */
       F[0] = 1;
       FI[0] = powerMod(F[0], MOD - 2, MOD);
       for (int i = 1; i < N; ++i) {
              F[i] = (F[i - 1] * i) % MOD;
              FI[i] = powerMod(F[i], MOD - 2, MOD);
       }
}

ll nCr( ll x, ll y ) {  /* O( 1 ) */
       if (x < y)    return 0;
       return  ((F[x] * ((FI[y] * FI[x - y]) % MOD)) % MOD);
}
```

GCD-LCM

```
ll gcd(ll a, ll b) {/* faster version( Maybe ) */
       if (!a || !b)     return a | b;
       unsigned shift = __builtin_ctz(a | b);
       a >>= __builtin_ctz(a);
       do {  b >>= __builtin_ctz(b);
              if (a > b) swap(a, b);
              b -= a;
       } while (b);

       return a << shift;
}
ll lcm(ll a, ll b) { return a / gcd(a, b) * b;}
ll gcd(ll par1, ll par2) { return par2 ? gcd(par2, par1 % par2) : par1; }
ll lcm(ll par1, ll par2) { return par1 * par2 / gcd(par1, par2); }
```

Power Mod

```
Long PowerMod( Long par1, Long par2, Long par3 ) {  /* ( par1 ^ par2 ) % par3 */
      Long res = 1; par1 %= par3;
      assert( par2 >= 0 );
      for(; par2; par2 >>= 1) {
            if (par2 & 1) res = res * par1 % par3;
            par1 = par1 * par1 % par3;
      }
      return res;
}
```

Phi Function

```
ll phi[MAX];
void phi_phi( ){
    for (ll i = 0; i < MAX; ++i)  phi[i] = i;
    for (ll i = 2; i < MAX; ++i) if (phi[i] == i)
        for(ll j = i; j <= MAX; j += i) phi[j] -= phi[j] / i;
}



ll giveMePhi( ll x ) { # For single number:
    ll res = x;
    for (ll i = 2; i * i <= x; ++i) {
        if (x % i == 0)  while ( x % i == 0 ) x /= i;
        res -= res / i;
    }
    if ( x > 1 )  res -= res / x;
    ret( res );
}
```

Sieve

```
class Sieve {
      public:
            vector <int> Sie;
            vector <int> Spf;
            Sieve(int MSS) {
                  Spf.resize(MSS + 7, 0);
                  for (int i = 2; i < MSS; ++i) {
                        if (Spf[i] == 0) {
                              Spf[i] = i;
                              Sie.push_back(i);
                        }
                        for (int j = 0; j < Sie.size() && i * Sie[j] <= MSS &&
Sie[j] <= Spf[i]; ++j)
                              Spf[i * Sie[j]] = Sie[j];
                  }
            }
```

```
};
```

NOTE

1. Blog Link: https://codeforces.com/blog/entry/84150
   - [ ] Sum-Xor property: **a + b = a ^ b + 2 * (a & b)**. Extended Version with two equations:
     a + b = a | b + a & b.
     a ^ b = a | b - a & b.
   - [ ] Upto $\mathbf{10^{12}}$ there can be at most **300** non-prime numbers between any two consecutive prime numbers.
   - [ ] Any number greater than 1 can be split into prime number(minimum number of prime):

```
if (isPrime(n))          ans = 1;
else if (n % 2 == 0)     ans = 2;
else if (isPrime(n - 2)) ans = 2;
else                     ans = 3;
```

   - [ ] Sometimes it is better to write a brute force / linear search solution because its overall complexity can be less.
   - [ ] When **A≤B** then ⌊**B−1/A**⌋ **≤ N ≤** ⌈**B−1/A**⌉ where N is the number of multiples of A between any two multiples of B.
   - [ ] Coordinate Compression Technique when value of numbers doesn't matter. It can be done with the help of mapping the shortest number to 1, next greater to 2 and so on.
   - [ ] Event method: When there is a problem in which two kinds of events are there (say start and end events), then you can give -ve values to start events and +ve values to end events, put them in a vector of pairs, sort them and then use as required.
   - [ ] When applying binary search on doubles / floats just run a loop upto 100 times instead of comparing l and r. It will make things easier.
   - [ ] For binary search you can also do binary lifting sort of thing, see for more details. (I don't know how to add that code without messing up the list, that's why the link: https://codeforces.com/blog/entry/84150?#comment-716582).
   - [ ] Sometimes, it is useful to visualize an array into a number of blocks to move towards a solution.
   - [ ] **gcd(Fn,Fm)=Fgcd(n,m)**, where Fx is the xth fibonacci numbers and the first two terms are 0,1.

2. Random:
   - [ ] gcd(a, b, c, d, e) = gcd(a, a - b, b - c, c - d, d - e).
     So, gcd(a + x, b + x, c + x, d + x, e + x) = gcd(a + x, a - b, b - c, c - d, d - e).
   - [ ] If you mean the number of independent cycles, for undirected graphs it is just edges minus vertices plus connected components (use DFS or BFS) but for directed graphs it's NP-hard.
   - [ ] Bit count: __builtin_popcountll = long long
   - [ ] [1-N]^-1 % MOD

```
for ( int i = 2; i <= N; ++i )
        inv[ i ] = MOD - ( MOD / i ) * inv[ MOD % i ] % MOD;
```

   - [ ] Point A(x, y), B(x, y), C(x, y), ...
     connected those points like A-C-B... / A-B-C.../.... such every angle < 90
     Solution: Pick a point and find a point that has max distance from that point. then find a point

that has max distance from the last added point.
Problem link: https://codeforces.com/contest/1477/problem/C

- [ ] Nth Fibonacci number = ceil(pow(goldenRatio, N) / sqrt(5)).
goldenRatio = (1 + sqrt(5)) / 2;

- [ ] Ordered_set: https://codeforces.com/blog/entry/11080

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;

#set-
typedef tree <int, null_type, less <int>, rb_tree_tag,
tree_order_statistics_node_update> ordered_set;

ordered_set X;
X.insert(/*1, 2, 4, 8, 16*/);
cout<<*X.find_by_order(1)<<endl; // 2
cout<<*X.find_by_order(2)<<endl; // 4
cout<<*X.find_by_order(4)<<endl; // 16
cout<<(end(X)==X.find_by_order(6))<<endl; // true

cout<<X.order_of_key(-5)<<endl;  // 0
cout<<X.order_of_key(1)<<endl;   // 0
cout<<X.order_of_key(3)<<endl;   // 2
cout<<X.order_of_key(4)<<endl;   // 2
cout<<X.order_of_key(400)<<endl; // 5

#multiset-
Main idea is to keep pairs like {elem,?id}.

typedef tree <pair <int, int>, null_type, less <pair <int, int> >, rb_tree_tag,
tree_order_statistics_node_update> ordered_set;
int t = 0;
ordered_set me;
me.insert({x, t++});
me.erase(me.lower_bound({x, 0}));
cout << me.order_of_key({x, 0}) << "\n";
```

- [ ] lower_bound returns an iterator pointing to the first element in the range [first,last) which has a value not less than 'val'. And if the value is not present in the vector then it returns the end iterator.
- [ ] upper_bound returns an iterator pointing to the first element in the range [first,last) which has a value greater than 'val'.

Pick's Theorem:

S: Area of lattice polygon, I: the number of points with integer coordinates lying strictly inside the polygon, B: the number of points lying on polygon sides

S=I + (B/2) -1

Lucas Theorem:

```
int nCrModpDP(int n, int r, int p) {
    int C[r+1];
    memset(C, 0, sizeof(C));
    C[0]=1;
    for (int i = 1; i <= n; i++)
        for (int j = min(i, r); j > 0; j--) C[j] = (C[j] + C[j-1])%p;
    return C[r];
}

int nCrModpLucas(int n, int r, int p) {
   if (r==0) return 1;
   int ni = n%p, ri = r%p;
   return (nCrModpLucas(n/p, r/p, p)*nCrModpDP(ni, ri, p)) % p;
}
```

Base Template:

```
#include <bits/stdc++.h>
using namespace std;
typedef long long int ll;
typedef pair<ll, ll> pi;
#define mp make_pair
#define pb push_back
#define F first
#define S second
#define forn(i, n) for (int i = 1; i <= int(n); i++)
#define sz(v) (int)v.size()
int main()
{
freopen("input.txt", "r", stdin);
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);cout.tie(NULL);
    int T; T=1;
    cin >> T;
    while(T--)
    {
        solve();
    }
}
```

Bitwise Functions:

```
ll turnOn(ll x, int pos) {
    return x | (1LL<<pos);
}
```

```
bool isOn(ll x ,int pos) {
    return (bool)(x & (1LL<<pos));
}
```

Dijkstra:

```
const int INF = 1000000000;
vector<vector<pair<int, int>>> adj;

void dijkstra(int s, vector<int> & d, vector<int> & p) {
    int n = adj.size();
    d.assign(n, INF);
    p.assign(n, -1);

    d[s] = 0;
    set<pair<int, int>> q;
    q.insert({0, s});
    while (!q.empty()) {
        int v = q.begin()->second;
        q.erase(q.begin());

        for (auto edge : adj[v]) {
            int to = edge.first;
            int len = edge.second;

            if (d[v] + len < d[to]) {
                q.erase({d[to], to});
                d[to] = d[v] + len;
                p[to] = v;
                q.insert({d[to], to}); } } } }
```

ncr mod m:

```
#define M 1000000007
typedef long long int ll;
ll fr[1001]; ll fac[100005];
 ll p(ll a, ll n) {
    ll res=1;
    while(n) {
        if(n%2){
            res=((res%M)*(a%M))%M;   n--;
        }
        else {
            a=((a%M)*(a%M))%M; n=n/2; } }
    return res; }

void f(ll n) {
```

```
    fac[0]=1;
    ll res=1;
    for(ll i=1;i<=n;i++) {
        res=((res%M)*(i%M))%M;
        fac[i]=res;
    }
}

ll ncr(ll n, ll r) {
    ll ans=0;
    ans=(ans+fac[n])%M;
    ll invr=p(fac[r], M-2);
    ll invnmr=p(fac[n-r], M-2);
    ans=((ans%M)*(invr%M))%M;
    ans=((ans%M)*(invnmr%M))%M;
    return ans;
}
```

Convex Hull Trick:

```
struct line{
ll m, c;
ll value(ll x) { return m*x+c; }
};

vector<line> hull;
ll ptr=0;
bool isBad(line p, line q, line r) {
   return ((r.c-p.c)*(1.0))/((p.m-r.m)*(1.0))<=((q.c-p.c)*(1.0)/(p.m-q.m)*(1.0));
}

void addLine(line l) {
   while(!hull.empty() && hull.back().m==l.m) {
       if(l.c<hull.back().c) hull.pop_back();
       else return;
   }

   while(hull.size()>=2 && isBad(hull[hull.size()-2], hull.back(), l))         {
       hull.pop_back();
   }
   hull.push_back(l);
}

ll query(ll x)      {
   while(ptr+1<hull.size() && hull[ptr].value(x)>hull[ptr+1].value(x))  {
                    ptr++;
    }
```

```
    return hull[ptr].value(x);
}
```

Prefix Function: (KMP)

```
vector<int> prefix_function(string s) {
    int n = (int)s.length();
    vector<int> pi(n);
    for (int i = 1; i < n; i++) {
        int j = pi[i-1];
        while (j > 0 && s[i] != s[j])
            j = pi[j-1];
        if (s[i] == s[j])
            j++;
        pi[i] = j;
    }
    return pi;
}
```

Convex Hull Algorithm (Graham's Scan):

```
struct pt {
    double x, y;
};

int orientation(pt a, pt b, pt c) {
    double v = a.x*(b.y-c.y)+b.x*(c.y-a.y)+c.x*(a.y-b.y);
    if (v < 0) return -1; // clockwise
    if (v > 0) return +1; // counter-clockwise
    return 0;
}

bool cw(pt a, pt b, pt c, bool include_collinear) {
    int o = orientation(a, b, c);
    return o < 0 || (include_collinear && o == 0);
}
bool collinear(pt a, pt b, pt c) { return orientation(a, b, c) == 0; }

void convex_hull(vector<pt>& a, bool include_collinear = false) {
    pt p0 = *min_element(a.begin(), a.end(), [](pt a, pt b) {
        return make_pair(a.y, a.x) < make_pair(b.y, b.x);
    });
    sort(a.begin(), a.end(), [&p0](const pt& a, const pt& b) {
        int o = orientation(p0, a, b);
        if (o == 0)
            return (p0.x-a.x)*(p0.x-a.x) + (p0.y-a.y)*(p0.y-a.y)
                < (p0.x-b.x)*(p0.x-b.x) + (p0.y-b.y)*(p0.y-b.y);
        return o < 0;
```

```
    });
    if (include_collinear) {
        int i = (int)a.size()-1;
        while (i >= 0 && collinear(p0, a[i], a.back())) i--;
        reverse(a.begin()+i+1, a.end());
    }

    vector<pt> st;
    for (int i = 0; i < (int)a.size(); i++) {
        while (st.size() > 1 && !cw(st[st.size()-2], st.back(), a[i],
include_collinear))
            st.pop_back();
        st.push_back(a[i]);
    }

    a = st;
}
```

Code Template:

```
#include<bits/stdc++.h>
//#include<ext/pb_ds/assoc_container.hpp>
using namespace std;
//using namespace __gnu_pbds;

#define fastIO ios::sync_with_stdio(0);cin.tie(0);
#define endl "\n"
#define pb push_back
#define mp make_pair
#define ll long long
#define ld long double
#define vi vector<int>
#define vll vector<long long>
#define vs vector<string>
#define pi pair<int,int>
#define pll pair<long long>
#define pqll priority_queue<long long>
//typedef
tree<int,null_type,less<int>,rb_tree_tag,tree_order_statistics_node_update>
indexed_set;

void solve();

int main()
{
   fastIO;
   solve();
```

```
}
```

Longest Increasing Subsequence:

```
#define MAX_N 20
#define EMPTY_VALUE -1

int mem[MAX_N];
int next_index[MAX_N];

int f(int i, vector<int> &A) {
   if (mem[i] != EMPTY_VALUE) {
      return mem[i];
   }

   int ans = 0;
   for (int j = i + 1;j < A.size();j++) {
      if (A[j] > A[i]) {
         int subResult = f(j, A);
         if (subResult > ans) {
            ans = subResult;
            next_index[i] = j;
         }
      }
   }

   mem[i] = ans + 1;
   return mem[i];
}

vector<int> findLIS(vector<int> A){
  int ans = 0;

  for(int i = 0;i<A.size();i++) {
     mem[i] = EMPTY_VALUE;
     next_index[i] = EMPTY_VALUE;
  }

  int start_index = -1;

  for(int i = 0;i<A.size();i++) {
     int result = f(i, A);
     if (result > ans) {
        ans = result;
        start_index = i;
     }
  }

  vector<int> lis;
```

```
  while(start_index != -1) {
     lis.push_back(A[start_index]);
     start_index = next_index[start_index];
  }
  return lis;
}
```