Anup Barman's

CP-Arsenal

CODEFORCES

AnupBarman

AtCoder

AnupBarman

anup_barman

Updated: January 5, 2026

Contents

1 Setup 1
1.1 Linux Build 1
1.2 Windows Build 1

2 DataStructures 1
2.1 Anti Hash Unordered Map 1
2.2 Co-Ordinate Compression 1
2.3 Lazy Propagation 1
2.4 PBDS 1
2.5 Segment Tree :: Binary search on Unsorted array 1
2.6 Segment Tree 2
2.7 Sparse Table 2

3 Geometry 2
3.1 Angular Sort 2
3.2 CircleCircleIntersection 2
3.3 CircleLineIntersection 2
3.4 Closest Pair of Points 2
3.5 ComputeCentroid 2
3.6 ComputeCircleCenter 2
3.7 ComputeLineIntersection 2
3.8 ComputeSignedArea 3
3.9 Convex Hull 3
3.10 DistancePointPlane 3
3.11 DistancePointSegment 3
3.12 Half Plane Intersection 3
3.13 IsSimple 3
3.14 LinesCollinear 3
3.15 LinesParallel 3
3.16 Point 3
3.17 PointInPolygon 4
3.18 ProjectPointLine 4
3.19 ProjectPointSegment 4
3.20 SegmentsIntersect 4

4 Graphs 4
4.1 Articulation Point 4
4.2 Bridge Finding Algorithm 4
4.3 Cycle Detection in DAG 4
4.4 DSU on Trees 4
4.5 DSU 4
4.6 Euler Tour 5
4.7 Floyd Warshall 5
4.8 LCA using Binary Lifting 5
4.9 MST 5
4.10 Max Bipartite Matching [Hopcroft-Karp] 5
4.11 Max Bipartite Matching [Kuhn's] 5
4.12 Topological Sorting 6
4.13 Weighted Union Find 6

5 Number Theory 6
5.1 Unique Prime Factorization using Sieve 6
5.2 nCr 6

1 Setup

1.1 Linux Build

```
{
  "cmd" : ["ulimit -s 268435456; g++ -std=c++20
      $file_name -o $file_base_name && timeout 4s
  ↪   ./$file_base_name < input.txt > output.txt"],
  "selector" : "source.c++",
  "shell" : true,
  "working_dir" : "$file_path"
}
```

1.2 Windows Build

```
{
  "cmd" : [ "g++.exe", "-std=c++20", "${file}", "-o",
      "${file_base_name}.exe", "&&",
  ↪   "${file_base_name}.exe<input.txt>output.txt" ],
  "selector" : "source.cpp",
  "shell" : true,
  "working_dir" : "$file_path"
}
```

2 DataStructures

2.1 Anti Hash Unordered Map

```
unordered_map<int, int> mp;
mp.reserve(1 << 20);       // about 1M buckets
mp.max_load_factor(0.7);
```

2.2 Co-Ordinate Compression

```
vector<int> pos;
sort(pos.begin(), pos.end());
pos.erase(unique(pos.begin(), pos.end()), pos.end());
// then lower_bound on this pos array to find the
↪  compressed co-ordinate
```

2.3 Lazy Propagation

```
class stree {
  vector<int> seg, lazy;
public:
  segtree(int n) {
    seg.resize(4 * n + 5);
    lazy.resize(4 * n + 5);
  }
  void propagate(int i, int low, int high) {
    if (lazy[i] != 0) {
      seg[i] += (high - low + 1) * lazy[i];
      if (low != high) {
        lazy[2 * i + 1] += lazy[i];
        lazy[2 * i + 2] += lazy[i];
      }
      lazy[i] = 0;
    }
  }
  void build(int i, int low, int high, int arr[]) {
    if (low == high) {
      seg[i] = arr[low];
      return;
    }
    int mid = (low + high) >> 1;
    build(2 * i + 1, low, mid, arr);
    build(2 * i + 2, mid + 1, high, arr);
    seg[i] = seg[2 * i + 1] + seg[2 * i + 2];
  }
  void update(int i, int low, int high, int l, int r,
  ↪  int val) {
    propagate(i, low, high);
    if (high < l or r < low) return;
    if (low >= l and high <= r) {
      seg[i] += (high - low + 1) * val;
      if (low != high) {
        // has children
        lazy[2 * i + 1] += val;
        lazy[2 * i + 2] += val;
      }
      return;
    }
    int mid = (low + high) >> 1;
    update(2 * i + 1, low, mid, l, r, val);
    update(2 * i + 2, mid + 1, high, l, r, val);
    seg[i] = seg[2 * i + 1] + seg[2 * i + 2];
  }
  int query(int i, int low, int high, int l, int r) {
    propagate(i, low, high);
    if (high < l or r < low) return 0;
    if (low >= l and high <= r) return seg[i];
    int mid = (low + high) >> 1;
    int left = query(2 * i + 1, low, mid, l, r);
    int right = query(2 * i + 2, mid + 1, high, l, r);
    return left + right;
  }
};
```

2.4 PBDS

```
#include "ext/pb_ds/assoc_container.hpp"
#include "ext/pb_ds/tree_policy.hpp"
using namespace __gnu_pbds;
template <class T>
using oset = tree<T, null_type, less<T>, rb_tree_tag,
tree_order_statistics_node_update>;
Note: Use less_equal for multiset like behaviour
Usage:
st.find_by_order(k) :: returns iterator of the k-th
 ↪  smallest element
st.order_of_key(x) :: returns index of x (number of
 ↪  elements less than x)
```

2.5 Segment Tree :: Binary search on Unsorted array

```
// 1 based indexing
struct SegTree {
  int n;
  vector<int> seg;
  SegTree(int _n) {
    n = _n;
    seg.assign(4 * n + 5, 0);
  }
  void build(int node, int l, int r) {
    if (l == r) {
      seg[node] = 1;
      return;
    }  // each position initially present
    int mid = (l + r) >> 1;
    build(node * 2, l, mid);
    build(node * 2 + 1, mid + 1, r);
    seg[node] = seg[node * 2] + seg[node * 2 + 1];
  }
  // set position pos to value val (0 or 1)
  void update(int node, int l, int r, int pos, int
  ↪  val) {
    if (l == r) {
      seg[node] = val;
      return;
    }
    int mid = (l + r) >> 1;
    if (pos <= mid)
      update(node * 2, l, mid, pos, val);
    else
      update(node * 2 + 1, mid + 1, r, pos, val);
    seg[node] = seg[node * 2] + seg[node * 2 + 1];
  }
  // find index of k-th "present" element (1-based k)
  int kth(int node, int l, int r, int k) {
    if (l == r) return l;
    int leftCnt = seg[node * 2];
    int mid = (l + r) >> 1;
    if (k <= leftCnt)
      return kth(node * 2, l, mid, k);
    else
      return kth(node * 2 + 1, mid + 1, r, k -
       ↪  leftCnt);
  }
};
```

```cpp
void solve() {
  int n;
  cin >> n;
  vector<ll> a(n + 1), p(n + 1);
  for (int i = 1; i <= n; ++i) {
    cin >> a[i];
  }
  for (int i = 1; i <= n; ++i) {
    cin >> p[i];
  }
  SegTree st(n);
  st.build(1, 1, n);
  // For each removal request p[i], find the p[i]-th
  ↪  present element,
  // print it and mark that position as removed (set to
  ↪  0).
  for (int i = 1; i <= n; ++i) {
    int k = p[i];
    int idx = st.kth(1, 1, n, k);  // index in original
    ↪  array
    cout << a[idx] << (i == n ? '\n' : ' ');
    st.update(1, 1, n, idx, 0);
  }
}
```

2.6 Segment Tree

```cpp
class stree {
  vector<int> seg;
public:
  segtree(int n) {
    seg.assign(4 * n + 5, 0);
  }
  void build(int ind, int low, int high, int arr[]) {
    if (low == high) {
      seg[ind] = arr[low];
      return;
    }
    int mid = (low + high) >> 1;
    build(2 * ind + 1, low, mid, arr);
    build(2 * ind + 2, mid + 1, high, arr);
    seg[ind] = min(seg[2 * ind + 1], seg[2 * ind + 2]);
  }
  int query(int ind, int low, int high, int l, int r) {
    if (r < low or high < l) return INT_MAX;
    if (low >= l and high <= r) return seg[ind];
    int mid = (low + high) / 2;
    int left = query(2 * ind + 1, low, mid, l, r);
    int right = query(2 * ind + 2, mid + 1, high, l,
    ↪  r);
    return min(left, right);
  }
  void update(int ind, int low, int high, int i, int
  ↪  val) {
    if (low == high) {
      seg[ind] = val;
      return;
    }
    int mid = (low + high) / 2;
    if (i <= mid) update(2 * ind + 1, low, mid, i,
    ↪  val);
    else update(2 * ind + 2, mid + 1, high, i, val);
    seg[ind] = min(seg[2 * ind + 1], seg[2 * ind + 2]);
  }
};
```

2.7 Sparse Table

```cpp
const int MX = 2e5 + 10;
int n, arr[MX], st[25][MX];
int log2Floor(int i) {
  return 31 - __builtin_clz(i);
}
void build() {
  int k = log2Floor(n);
  copy(arr, arr + n, st[0]);
  for (int i = 1; i <= k; ++i) {
    for (int j = 0; j + (1 << i) <= n; j++) {
      st[i][j] = min(st[i - 1][j], st[i - 1][j + (1 <<
      ↪  (i - 1))]);
    }
  }
}
int query(int l, int r) {
  int i = log2Floor(r - l + 1);
  return min(st[i][l], st[i][r - (1 << i) + 1]);
}
```

3 Geometry

3.1 Angular Sort

```cpp
inline bool up (point p) {
  return p.y > 0 or (p.y == 0 and p.x >= 0);
}
sort(v.begin(), v.end(), [] (point a, point b) {
  return up(a) == up(b) ? a.x * b.y > a.y * b.x :
  ↪  up(a) < up(b);
});
inline int quad (point p) {
  if (p.y >= 0)  return p.x < 0;
  return 2 + (p.x >= 0);
}
sort(pt.begin(), pt.end(), [] (point a, point b) {
  return quad(a) == quad(b) ? a.x * b.y > a.y * b.x :
  ↪  quad(a) < quad(b);
});
```

3.2 CircleCircleIntersection

Description: compute intersection of circle centered at a with radius r with circle centered at b with radius R.

```cpp
vector<PT> CircleCircleIntersection(PT a, PT b, double
↪  r, double R) {
  vector<PT> ret;
  double d = sqrt(dist2(a, b));
  if (d > r+R || d+min(r, R) < max(r, R)) return ret;
  double x = (d*d-R*R+r*r)/(2*d);
  double y = sqrt(r*r-x*x);
  PT v = (b-a)/d;
  ret.push_back(a+v*x + RotateCCW90(v)*y);
  if (y > 0)
    ret.push_back(a+v*x - RotateCCW90(v)*y);
  return ret;
}
```

3.3 CircleLineIntersection

Description: Compute intersection of line through points a and b with circle centered at c with radius $r > 0$.

```cpp
vector<PT> CircleLineIntersection(PT a, PT b, PT c,
↪  double r) {
  vector<PT> ret;
  b = b-a; a = a-c;
  double A = dot(b, b); double B = dot(a, b);
  double C = dot(a, a) - r*r;
  double D = B*B - A*C;
  if (D < -EPS) return ret;
  ret.push_back(c+a+b*(-B+sqrt(D+EPS))/A);
  if (D > EPS)
    ret.push_back(c+a+b*(-B-sqrt(D))/A);
  return ret;
}
```

3.4 Closest Pair of Points

```cpp
ll min_dis(vector<array<int, 2>> &pts, int l, int r) {
  if (l + 1 >= r)  return LLONG_MAX;
  int m = (l + r) / 2;
  ll my = pts[m-1][1];
  ll d = min(min_dis(pts, l, m), min_dis(pts, m, r));
  inplace_merge(pts.begin()+l, pts.begin()+m,
  ↪  pts.begin()+r);
  for (int i = l; i < r; ++i) {
    if ((pts[i][1] - my) * (pts[i][1] - my) < d) {
      for (int j = i + 1; j < r and (pts[i][0] -
      ↪  pts[j][0]) * (pts[i][0] - pts[j][0]) < d;
      ↪  ++j) {
        ll dx = pts[i][0] - pts[j][0], dy = pts[i][1]
        ↪  - pts[j][1];
        d = min(d, dx * dx + dy * dy);
      }
    }
  }
  return d;
}
vector<array<int, 2>> pts(n);
sort(pts.begin(), pts.end(), [&] (array<int, 2> a,
↪  array<int, 2> b){
  return make_pair(a[1], a[0]) < make_pair(b[1], b[0]);
});
```

3.5 ComputeCentroid

```cpp
// centroid of a (possibly nonconvex) polygon.
PT ComputeCentroid(const vector<PT> &p) {
  PT c(0,0);
  double scale = 6.0 * ComputeSignedArea(p);
  for (int i = 0; i < p.size(); i++){
    int j = (i+1) % p.size();
    c = c + (p[i]+p[j])*(p[i].x*p[j].y -
    ↪  p[j].x*p[i].y);
  }
  return c / scale;
}
```

3.6 ComputeCircleCenter

```cpp
// compute center of circle passing through three
↪  points
PT ComputeCircleCenter(PT a, PT b, PT c) {
  b=(a+b)/2;
  c=(a+c)/2;
  return ComputeLineIntersection(b, b+RotateCW90(a-b),
  ↪  c, c+RotateCW90(a-c));
}
```

3.7 ComputeLineIntersection

Description: compute intersection of line passing through a and b with line passing through c and d, assuming that unique intersection exists; for segment intersection, check if segments intersect first.

```cpp
PT ComputeLineIntersection(PT a, PT b, PT c, PT d) {
  b=b-a; d=c-d; c=c-a;
  assert(dot(b, b) > EPS && dot(d, d) > EPS);
  return a + b*cross(c, d)/cross(b, d);
}
```

3.8 ComputeSignedArea

Description: Computes the area of a (possibly nonconvex) polygon, assuming that the coordinates are listed in a clockwise or counterclockwise fashion.

```
double ComputeSignedArea(const vector<PT> &p) {
  double area = 0;
  for(int i = 0; i < p.size(); i++) {
    int j = (i+1) % p.size();
    area += p[i].x*p[j].y - p[j].x*p[i].y;
  }
  return area / 2.0;
}
double ComputeArea(const vector<PT> &p) {
  return fabs(ComputeSignedArea(p));
}
// integer area
void computeIntArea() {
  int n; cin >> n;
  point arr[n];
  for (int i = 0; i < n; i++) {
    arr[i].read();
  }
  point a = {0, 0};
  ll ans = 0;
  for (int i = 0; i + 1 < n; i++) {
    ans += a.triangle(arr[i], arr[i + 1]);
  }
  ans += a.triangle(arr[n - 1],  arr[0]);
  cout << abs(ans) << "\n";
}
```

3.9 Convex Hull

```
vector <PT> convexHull (vector <PT> p) {
  int n = p.size(), m = 0;
  if (n < 3) return p;
  vector <PT> hull(n + n);
  sort(p.begin(), p.end(), [&] (PT a, PT b) {
    return (a.x==b.x? a.y<b.y: a.x<b.x);
  });
  for (int i = 0; i < n; ++i) {
    while (m > 1 and cross(hull[m - 2] - p[i], hull[m
    ↪ - 1] - p[i]) <= 0) --m;
    hull[m++] = p[i];
  }
  for (int i = n - 2, j = m + 1; i >= 0; --i) {
    while (m >= j and cross(hull[m - 2] - p[i], hull[m
    ↪ - 1] - p[i]) <= 0) --m;
    hull[m++] = p[i];
  }
  hull.resize(m - 1); return hull;
}
```

3.10 DistancePointPlane

Description: compute distance between point (x,y,z) and plane $ax+by+cz=d$

```
double DistancePointPlane(double x, double y, double
 ↪ z, double a, double b, double c, double d) {
  return fabs(a*x+b*y+c*z-d)/sqrt(a*a+b*b+c*c);
}
```

3.11 DistancePointSegment

```
// compute distance from c to segment between a and b
double DistancePointSegment(PT a, PT b, PT c) {
  return sqrt(dist2(c, ProjectPointSegment(a, b, c)));
}
```

3.12 Half Plane Intersection

Description: Calculates the intersection of halfplanes, assuming every half-plane allows the region to the left of its line.

```
struct Halfplane {
  PT p, pq; ld angle;
  Halfplane() {}
  // Two points on line
  Halfplane(const PT& a, const PT& b) : p(a), pq(b -
  ↪ a) {
    angle = atan2l(pq.y, pq.x);
  }
  bool out(const PT& r) {
    return cross(pq, r - p) < -EPS;
  }
  bool operator < (const Halfplane& e) const {
    return angle < e.angle;
  }
  friend PT inter(const Halfplane& s, const Halfplane&
  ↪ t) {
    ld alpha = cross((t.p - s.p), t.pq) / cross(s.pq,
    ↪ t.pq);
    return s.p + (s.pq * alpha);
  }
};
vector<PT> hp_intersect(vector<Halfplane>& H) {
  PT box[4] = {  // Bounding box in CCW order
    PT(INF, INF),  PT(-INF, INF),
    PT(-INF, -INF), PT(INF, -INF)
  };
  for(int i = 0; i<4; i++) { // Add bounding box
  ↪ half-planes.
      Halfplane aux(box[i], box[(i+1) % 4]);
      H.push_back(aux);
  }
  sort(H.begin(), H.end());
  deque<Halfplane> dq; int len = 0;
  for(int i = 0; i < int(H.size()); i++) {
    while (len > 1 && H[i].out(inter(dq[len-1],
    ↪ dq[len-2]))) {
      dq.pop_back(); --len;
    }
    while (len > 1 && H[i].out(inter(dq[0], dq[1]))) {
      dq.pop_front(); --len;
    }
    if (len > 0 && fabsl(cross(H[i].pq, dq[len-1].pq))
    ↪ < EPS) {
      if (dot(H[i].pq, dq[len-1].pq) < 0.0)
        return vector<PT>();
      if (H[i].out(dq[len-1].p)) {
        dq.pop_back(); --len;
      }
      else continue;
    }
    dq.push_back(H[i]); ++len;
  }
  while (len > 2 && dq[0].out(inter(dq[len-1],
  ↪ dq[len-2]))) {
    dq.pop_back(); --len;
  }
  while (len > 2 && dq[len-1].out(inter(dq[0],
  ↪ dq[1]))) {
    dq.pop_front(); --len;
  }
  // Report empty intersection if necessary
  if (len < 3) return vector<PT>();
  // Reconstruct the convex polygon from the remaining
  ↪ half-planes.
  vector<PT> ret(len);
  for(int i = 0; i+1 < len; i++) {
    ret[i] = inter(dq[i], dq[i+1]);
  }
  ret.back() = inter(dq[len-1], dq[0]);
  return ret;
}
```

3.13 IsSimple

```
// tests whether or not a given polygon (in CW or CCW
 ↪ order) is simple
bool IsSimple(const vector<PT> &p) {
  for (int i = 0; i < p.size(); i++) {
    for (int k = i+1; k < p.size(); k++) {
      int j = (i+1) % p.size();
      int l = (k+1) % p.size();
      if (i == l || j == k) continue;
      if (SegmentsIntersect(p[i], p[j], p[k], p[l]))
        return false;
    }
  }
  return true;
}
```

3.14 LinesCollinear

```
bool LinesCollinear(PT a, PT b, PT c, PT d) {
  return LinesParallel(a, b, c, d)
    && fabs(cross(a-b, a-c)) < EPS
    && fabs(cross(c-d, c-a)) < EPS;
}
```

3.15 LinesParallel

```
// determine if lines from a to b and c to d are
 ↪ parallel or collinear
bool LinesParallel(PT a, PT b, PT c, PT d) {
  return fabs(cross(b-a, c-d)) < EPS;
}
```

3.16 Point

```
double INF = 1e100;
double EPS = 1e-12;
struct PT {
  double x, y;
  PT() {}
  PT(double x, double y) : x(x), y(y) {}
  PT(const PT &p) : x(p.x), y(p.y)    {}
  PT operator + (const PT &p)  const { return
  ↪ PT(x+p.x, y+p.y); }
  PT operator - (const PT &p)  const { return
  ↪ PT(x-p.x, y-p.y); }
  PT operator * (double c)     const { return PT(x*c,
  ↪ y*c  ); }
  PT operator / (double c)     const { return PT(x/c,
  ↪ y/c  ); }
};
double dot(PT p, PT q)     { return p.x*q.x+p.y*q.y; }
double dist2(PT p, PT q)   { return dot(p-q,p-q); }
double abs(PT p) { return sqrt(p.x*p.x + p.y*p.y); }
double cross(PT p, PT q)   { return p.x*q.y-p.y*q.x; }
ostream &operator<<(ostream &os, const PT &p) {
  return os << "(" << p.x << "," << p.y << ")";
}
// rotate a point CCW or CW around the origin
PT RotateCCW90(PT p)   { return PT(-p.y,p.x); }
PT RotateCW90(PT p)    { return PT(p.y,-p.x); }
PT RotateCCW(PT p, double t) {
  return PT(p.x*cos(t)-p.y*sin(t),
  ↪ p.x*sin(t)+p.y*cos(t));
}
// angle (range [0, pi]) between two vectors
double angle(PT v, PT w) {
  return acos(clamp(dot(v,w) / abs(v) / abs(w), -1.0,
  ↪  1.0));
}
```

3.17 PointInPolygon

Description: −1 = strictly inside, 0 = on, 1 = strictly outside.

```
int PointInPolygon(vector<PT> &P, PT a) {
  int cnt = 0, n = P.size();
  for(int i = 0; i < n; ++i) {
    PT q = P[(i + 1) % n];
    if (onSegment(P[i], q, a)) return 0;
    cnt ^= ((a.y < P[i].y) - (a.y < q.y)) * cross(P[i]
    ↪  - a, q - a) > 0;
  } return cnt > 0 ? -1 : 1;
}
int PointInConvexPolygon(vector<PT> &P, const PT& q) {
↪  // O(log n)
  int n = P.size();
  ll a = cross(P[0] - q, P[1] - q), b = cross(P[0] -
  ↪  q, P[n - 1] - q);
  if (a < 0 or b > 0) return 1;
  int l = 1, r = n - 1;
  while (l + 1 < r) {
    int mid = l + r >> 1;
    if (cross(P[0] - q, P[mid] - q) >= 0) l = mid;
    else r = mid;
  }
  ll k = cross(P[l] - q, P[r] - q);
  if (k <= 0) return k < 0 ? 1 : 0;
  if (l == 1 and a == 0) return 0;
  if (r == n - 1 and b == 0) return 0;
  return -1;
}
```

3.18 ProjectPointLine

```
// project point c onto line through a and b, assuming
↪  a != b
PT ProjectPointLine(PT a, PT b, PT c) {
  return a + (b-a)*dot(c-a, b-a)/dot(b-a, b-a);
}
```

3.19 ProjectPointSegment

```
// project point c onto line segment through a and b
PT ProjectPointSegment(PT a, PT b, PT c) {
  double r = dot(b-a,b-a);
  if (fabs(r) < EPS) return a;
  r = dot(c-a, b-a)/r;
  if (r < 0) return a;
  if (r > 1) return b;
  return a + (b-a)*r;
}
```

3.20 SegmentsIntersect

```
// determine if line segment from a to b intersects
↪  with line segment from c to d
bool SegmentsIntersect(PT a, PT b, PT c, PT d) {
  if (LinesCollinear(a, b, c, d)) {
    if (dist2(a, c) < EPS || dist2(a, d) < EPS ||
      dist2(b, c) < EPS || dist2(b, d) < EPS) return
      ↪  true;
    if (dot(c-a, c-b) > 0 && dot(d-a, d-b) > 0 &&
    ↪  dot(c-b, d-b) > 0)
      return false;
    return true;
  }
  if (cross(d-a, b-a) * cross(c-a, b-a) > 0) return
  ↪  false;
  if (cross(a-c, d-c) * cross(b-c, d-c) > 0) return
  ↪  false;
  return true;
}
```

4 Graphs

4.1 Articulation Point

```
int n;                          // number of nodes
vector<vector<int>> lst;        // adjacency list of graph
vector<bool> vis;
vector<int> tin, low;
int timer;
void dfs(int u, int p = -1) {
  vis[u] = true;
  tin[u] = low[u] = timer++;
  int children = 0;
  for (int v : lst[u]) {
    if (v == p) continue;
    if (vis[v]) {
      low[u] = min(low[u], tin[v]);
    } else {
      dfs(v, u);
      low[u] = min(low[u], low[v]);
      if (low[v] >= tin[u] && p != -1)
        IS_CUTPOINT(u);
      ++children;
    }
  }
  // if no vertex below v can reach u or higher
  ↪  removing u disconnects that subtree
  if (p == -1 && children > 1)
    IS_CUTPOINT(u);
}
void find_cutpoints() {
  timer = 0;
  vis.assign(n, false);
  tin.assign(n, -1);
  low.assign(n, -1);
  for (int i = 0; i < n; ++i) {
    if (!vis[i])
      dfs(i);
  }
}
```

4.2 Bridge Finding Algorithm

```
const int MX = 1e5 + 10;
int n, m, timer = 0;
vector<int> adj[MX];
vector<int> tin(MX, -1), low(MX, -1);
vector<bool> vis(MX, false);
void is_bridge(int u, int v) {
  // do something with the edge
}
void dfs(int u, int p = -1) {
  vis[u] = true;
  tin[u] = low[u] = timer++;
  for (int v : adj[u]) {
    if (v == p) continue;
    if (vis[v]) {
      low[u] = min(low[u], tin[v]);
    } else {
      dfs(v, u);
      low[u] = min(low[u], low[v]);
      if (low[v] > tin[u]) {
        is_bridge(u, v);
      }
    }
  }
}
```

4.3 Cycle Detection in DAG

```
const int MX = 1e5 + 10;
bool vis[MX], pathVis[MX];
vector<int> lst[MX];
bool dfs(int u) {
  vis[u] = true;
  pathVis[u] = true;
  for (auto v : lst[u]) {
    if (!vis[v]) {
      if (dfs(v))
        return true;
    } else if (pathVis[v]) {
      return true;
    }
  }
  pathVis[u] = false;
  return false;
}
void solve() {
  // take graph input
  for (int i = 0; i < n; ++i) {
    if (!vis[i])
      dfs(i);
  }
}
```

4.4 DSU on Trees

```
int n, color[MX], ans[MX];
vector<int> g[MX];
set<int> bucket[MX];
int merge(int a, int b) {
  if (bucket[a].size() < bucket[b].size()) swap(a, b);
  bucket[a].insert(bucket[b].begin(), bucket[b].end());
  bucket[b].clear();
  return a;
}
int dfs(int u, int p = -1) {
  int cur = u;
  for (int v : g[u])
    if (v != p)
      cur = merge(cur, dfs(v, u));
  ans[u] = (int)bucket[cur].size();
  return cur;
}
void solve() {
  cin >> n;
  for (int i = 0; i < n; ++i) {
    cin >> color[i];
    bucket[i].insert(color[i]);
  }
  // graph input
  dfs(0);
  // print output
}
```

4.5 DSU

```
const int MX = 1e5 + 10;
int par[MX], sz[MX];
void init() {
  for (int i = 1; i < MX; i++) {
    par[i] = i;
    sz[i] = 1;
  }
}
int findpar(int x) {
  if (par[x] == x) return x;
  return par[x] = findpar(par[x]);
}
```

```cpp
void unite(int u, int v) {
  u = findpar(u);
  v = findpar(v);
  if (u != v) {
    if (sz[u] < sz[v]) {
      swap(u, v);
    }
    sz[u] += sz[v];
    par[v] = u;
  }
}
```

4.6 Euler Tour

```cpp
const int MX = 2e5 + 10;
int timer = -1;
// s = start pos, e = end pos
int val[MX], s[MX], e[MX], flat[MX];
vector<int> lst[MX];
void dfs(int u, int p) {
  s[u] = ++timer;
  flat[timer] = val[u];
  for (auto v : lst[u]) {
    if (v != p)
      dfs(v, u);
  }
  e[u] = timer;
}
```

4.7 Floyd Warshall

```cpp
vector<vector<int>> d(n, vector<int> (n, INF));
// take graph input into d
for (int k = 0; k < n; ++k) {
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            if (d[i][k] < INF && d[k][j] < INF)
                d[i][j] = min(d[i][j], d[i][k] +
                 ↪  d[k][j]);
        }
    }
}
```

4.8 LCA using Binary Lifting

```cpp
int n, l;
vector<vector<int>> adj;

int timer;
vector<int> tin, tout;
vector<vector<int>> up;

void dfs(int v, int p) {
  tin[v] = ++timer;
  up[v][0] = p;
  for (int i = 1; i <= l; ++i)
    up[v][i] = up[up[v][i - 1]][i - 1];

  for (int u : adj[v]) {
    if (u != p)
      dfs(u, v);
  }

  tout[v] = ++timer;
}

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

int lca(int u, int v) {
  if (is_ancestor(u, v))
    return u;
  if (is_ancestor(v, u))
    return v;
  for (int i = l; i >= 0; --i) {
    if (!is_ancestor(up[u][i], v))
      u = up[u][i];
  }
  return up[u][0];
}

void preprocess(int root) {
  tin.resize(n);
  tout.resize(n);
  timer = 0;
  l = ceil(log2(n));
  up.assign(n, vector<int>(l + 1));
  dfs(root, root);
}
```

4.9 MST

```cpp
// DSU first
void solve() {
  int n, m;
  cin >> n >> m;
  vector<tuple<int, int, int>> edges;
  for (int i = 0; i < m; ++i) {
    int u, v, wt;
    cin >> u >> v >> wt;
    edges.push_back({wt, u, v});
  }
  sort(edges.begin(), edges.end());
  init(n);
  int cost = 0;
  for (tuple& [ wt, u, v ] : edges) {
    if (findpar(u) == findpar(v)) continue;
    unite(u, v);
    cost += wt;
  }
  cout << cost << endl;
}
```

4.10 Max Bipartite Matching [Hopcroft-Karp]

```cpp
const int INF = 1e9;
void hopcroftCarp() {
  int n, m, e;
  cin >> n >> m >> e;
  vector<int> adj[n];
  for (int i = 0; i < e; ++i) {
    int u, v;
    cin >> u >> v;
    --u;
    --v;
    adj[u].push_back(v);
  }
  vector<int> ml(m, -1), mr(n, -1), dist(n);
  auto bfs = [&]() -> bool {
    queue<int> q;
    for (int u = 0; u < n; ++u) {
      if (mr[u] == -1) {
        dist[u] = 0;
        q.push(u);
      } else {
        dist[u] = INF;
      }
    }
    bool foundAugmenting = false;
    while (!q.empty()) {
      int u = q.front();
      q.pop();
      for (int v : adj[u]) {
        int pairedLeft = ml[v];
        if (pairedLeft == -1) {
          foundAugmenting = true;
        } else if (dist[pairedLeft] == INF) {
          dist[pairedLeft] = dist[u] + 1;
          q.push(pairedLeft);
        }
      }
    }
    return foundAugmenting;
  };
  function<bool(int)> dfs = [&](int u) -> bool {
    for (int v : adj[u]) {
      int pairedLeft = ml[v];
      if (pairedLeft == -1 or (dist[pairedLeft] ==
       ↪  dist[u] + 1 and dfs(pairedLeft))) {
        mr[u] = v;
        ml[v] = u;
        return true;
      }
    }
    dist[u] = INF;
    return false;
  };
  int matching = 0;
  while (bfs()) {
    for (int u = 0; u < n; ++u) {
      if (mr[u] == -1) {
        if (dfs(u)) matching++;
      }
    }
  }
  cout << matching << el;
  for (int u = 0; u < n; ++u) {
    if (mr[u] != -1) {
      cout << u << " " << mr[u] << el;
    }
  }
}
```

4.11 Max Bipartite Matching [Kuhn's]

```cpp
// left set size, right set size, edge count
int n, k, m, visToken = 1;
vector<int> lst[MX];
int mr[MX], ml[MX], vis[MX];
bool try_kuhn(int u) {
  if (vis[u] == visToken)
    return false;
  vis[u] = visToken;
  for (auto v : lst[u]) {
    if (ml[v] == -1 or try_kuhn(ml[v])) {
      ml[v] = u;
      mr[u] = v;
      return true;
    }
  }
  return false;
}
void solve() {
  cin >> n >> k >> m;
  for (int i = 0; i < m; ++i) {
    int u, v;
    cin >> u >> v;
    --u, --v;
    lst[u].push_back(v);
  }
  fill(mr, mr + n, -1);
  fill(ml, ml + k, -1);
  int ans = 0;
  for (int u = 0; u < n; ++u) {
    for (auto v : lst[u]) {
      if (ml[v] == -1) {
        ml[v] = u;
        mr[u] = v;
        ans++;
        break;
      }
    }
  }
  for (int u = 0; u < n; ++u) {
    if (mr[u] != -1) continue;
    visToken++;
    if (try_kuhn(u))
      ans++;
  }
  cout << ans << el;
  for (int v = 0; v < k; ++v) {
    if (ml[v] != -1) {
      cout << ml[v] + 1 << " " << v + 1 << el;
    }
  }
}
```

4.12 Topological Sorting

```
const int N = 1e5 + 10;
vector<int> g[N], indegree(N, 0);
vector<int> topSort(int n) {
  queue<int> q;
  vector<int> order;
  for (int i = 1; i <= n; i++) {
    if (indegree[i] == 0) {
      q.push(i);
    }
  }
  while (!q.empty()) {
    int u = q.front();
    q.pop();
    order.push_back(u);
    for (int v : g[u]) {
      indegree[v]--;
      if (indegree[v] == 0) {
        q.push(v);
      }
    }
  }
  return order;
}
```

4.13 Weighted Union Find

```
const int MX = 2e5 + 10;
int par[MX], sz[MX];
ll d[MX];
void init() {
  for (int i = 0; i < MX; ++i) {
    par[i] = i;
    sz[i] = 1;
    d[i] = 0;
  }
}
int findpar(int x) {
  if (par[x] == x) return x;
  int p = par[x];
  par[x] = findpar(p);
  d[x] += d[p];
  return par[x];
}
bool unite(int a, int b, ll w) {
  int ra = findpar(a);
  int rb = findpar(b);
  if (ra == rb) {
    return (d[b] - d[a] == w);
  }
  if (sz[ra] < sz[rb]) {
    swap(a, b);
    swap(ra, rb);
    w = -w;
  }
  par[rb] = ra;
  d[rb] = d[a] + w - d[b];
  sz[ra] += sz[rb];
  return true;
}
```

```
ll dist(int a, int b) {
  findpar(a), findpar(b);
  return d[b] - d[a];
}
```

5 Number Theory

5.1 Unique Prime Factorization using Sieve

```
const int MX = 2e5 + 10;
vector<int> pfac[MX];
void factorize() {
    for (int i = 2; i < MX; i++){
        if (!pfac[i].empty()) continue;
        for (int j = i; j < MX; j += i)
            pfac[j].push_back(i);
    }
}
```

5.2 nCr

```
const int MX = 1e6 + 10;
const int M = 1e9 + 7;
int fact[MX], inv_fact[MX];
int modPow(int a, int b) {
  int ans = 1;
  while (b) {
    if (b & 1) ans = (1LL * ans * a) % M;
    a = (1LL * a * a) % M;
    b >>= 1;
  }
  return ans;
}
void precalFact() {
  fact[0] = inv_fact[0] = 1;
  for (int i = 1; i < MX; i++) {
    fact[i] = (1LL * fact[i - 1] * i) % M;
  }
  inv_fact[MX - 1] = modPow(fact[MX - 1], M - 2);
  for (int i = MX - 2; i >= 1; i--) {
    inv_fact[i] = (1LL * inv_fact[i + 1] * (i + 1)) %
     ↪ M;
  }
}
int nCr(int n, int r) {
  if (r < 0 or r > n) return 0;
  return 1LL * fact[n] * inv_fact[r] % M * inv_fact[n
   ↪ - r] % M;
}
```