IIT Jammu ICPC Team Notebook (2017-18)

Contents

1 Data Structures and Libraries

1.1 Union Find Disjoint Set

```
// Union-Find Disjoint Sets Library written in OOP manner, using both path compression and union by rank heuristics
class UnionFind {                                   // OOP style
private:
  vi p, rank, setSize;                        // remember: vi is vector<int>
  int numSets;
public:
  UnionFind(int N) {
    setSize.assign(N, 1); numSets = N; rank.assign(N, 0);
    p.assign(N, 0); for (int i = 0; i < N; i++) p[i] = i; }
  int findSet(int i) { return (p[i] == i) ? i : (p[i] = findSet(p[i])); }
  bool isSameSet(int i, int j) { return findSet(i) == findSet(j); }
  void unionSet(int i, int j) {
    if (!isSameSet(i, j)) { numSets--;
    int x = findSet(i), y = findSet(j);
    // rank is used to keep the tree short
    if (rank[x] > rank[y]) { p[y] = x; setSize[x] += setSize[y]; }
    else                   { p[x] = y; setSize[y] += setSize[x];
                             if (rank[x] == rank[y]) rank[y]++; } } }
  int numDisjointSets() { return numSets; }
  int sizeOfSet(int i) { return setSize[findSet(i)]; }
};
```

1.2 Segment Tree

```
class SegmentTree {         // the segment tree is stored like a heap array
private: vi st, A;            // recall that vi is: typedef vector<int> vi;
  int n;
  int left (int p) { return p << 1; }     // same as binary heap operations
  int right(int p) { return (p << 1) + 1; }

  void build(int p, int L, int R) {                          // O(n log n)
    if (L == R)                            // as L == R, either one is fine
      st[p] = L;                                        // store the index
```

```
    else {                                          //
       recursively compute the values
      build(left(p) , L                 , (L + R
         ) / 2);
      build(right(p), (L + R) / 2 + 1, R
                    );
      int p1 = st[left(p)], p2 = st[right(p)
         ];
      st[p] = (A[p1] <= A[p2]) ? p1 : p2;
  } }
  int rmq(int p, int L, int R, int i, int j)
     {                     // O(log n)
    if (i >  R || j <  L) return -1; //
       current segment outside query range
    if (L >= i && R <= j) return st[p];
                        // inside query range

     // compute the min position in the left
        and right part of the interval
    int p1 = rmq(left(p) , L                , (
       L+R) / 2, i, j);
    int p2 = rmq(right(p), (L+R) / 2 + 1, R
                 , i, j);

    if (p1 == -1) return p2;    // if we try
       to access segment outside query
    if (p2 == -1) return p1;
                                      // same
       as above
    return (A[p1] <= A[p2]) ? p1 : p2; }
                 // as as in build routine
  int update_point(int p, int L, int R, int
     idx, int new_value) {
    // this update code is still preliminary,
        i == j
    // must be able to update range in the
       future!
    int i = idx, j = idx;

    // if the current interval does not
       intersect
    // the update interval, return this st
       node value!
    if (i > R || j < L)
      return st[p];

    // if the current interval is included in
        the update range,
    // update that st[node]
    if (L == i && R == j) {
      A[i] = new_value; // update the
         underlying array
      return st[p] = L; // this index
    }

    // compute the minimum pition in the
    // left and right part of the interval
    int p1, p2;
    p1 = update_point(left(p) , L
                        , (L + R) / 2, idx,
       new_value);
    p2 = update_point(right(p), (L + R) / 2 +
        1, R            , idx, new_value);

    // return the pition where the overall
       minimum is
    return st[p] = (A[p1] <= A[p2]) ? p1 : p2
       ;
  }
public:
  SegmentTree(const vi &_A) {
    A = _A; n = (int)A.size();
       // copy content for local usage
    st.assign(4 * n, 0);               // create
        large enough vector of zeroes
    build(1, 0, n - 1);
                                             //
       recursive build
  }
  int rmq(int i, int j) { return rmq(1, 0, n
     - 1, i, j); }   // overloading

  int update_point(int idx, int new_value) {
    return update_point(1, 0, n - 1, idx,
       new_value); }
};
```

1.3 Fenwick Tree

```
#define LSOne(S) (S & (-S))

class FenwickTree {
private:
  vi ft;

public:
  FenwickTree() {}
  // initialization: n + 1 zeroes, ignore
     index 0
```

```
  FenwickTree(int n) { ft.assign(n + 1, 0); }
  int rsq(int b) {                                    //
     returns RSQ(1, b)
    int sum = 0; for (; b; b -= LSOne(b)) sum
        += ft[b];
    return sum; }
  int rsq(int a, int b) {                      // returns
     RSQ(a, b)
    return rsq(b) - (a == 1 ? 0 : rsq(a - 1))
        ; }
  // adjusts value of the k-th element by v (
     v can be +ve/inc or -ve/dec)
  void adjust(int k, int v) {
                                  // note: n = ft.size
     () - 1
    for (; k < (int)ft.size(); k += LSOne(k))
        ft[k] += v; }
};
```

1.4 Bit Manipulation

```
#define isOn(S, j) (S & (1 << j))
#define setBit(S, j) (S |= (1 << j))
#define clearBit(S, j) (S &= ~(1 << j))
#define toggleBit(S, j) (S ^= (1 << j))
#define lowBit(S) (S & (-S))
#define setAll(S, n) (S = (1 << n) - 1)

#define modulo(S, N) ((S) & (N - 1))   //
   returns S % N, where N is a power of 2
#define isPowerOfTwo(S) (!(S & (S - 1)))
#define nearestPowerOfTwo(S) ((int)pow(2.0, (
   int)((log((double)S) / log(2.0)) + 0.5)))
#define turnOffLastBit(S) ((S) & (S - 1))
#define turnOnLastZero(S) ((S) | (S + 1))
#define turnOffLastConsecutiveBits(S) ((S) &
   (S + 1))
#define turnOnLastConsecutiveZeroes(S) ((S) |
    (S - 1))

void printSet(int vS) {
                                  // in binary
   representation
  printf("S = %2d = ", vS);
  stack<int> st;
  while (vS)
    st.push(vS % 2), vS /= 2;
  while (!st.empty())
                                     // to reverse
     the print order
    printf("%d", st.top()), st.pop();
  printf("\n");
}
```

2 Mathematics

2.1 Prime Numbers

```
ll _sieve_size;
bitset<10000010> bs;
vi primes;

void sieve(ll upperbound) {
  _sieve_size = upperbound + 1;
  bs.set();
  bs[0] = bs[1] = 0;
  for (ll i = 2; i <= _sieve_size; i++) if (
     bs[i]) {
    for (ll j = i * i; j <= _sieve_size; j +=
        i) bs[j] = 0;
    primes.push_back((int)i);
} }

bool isPrime(ll N) {
  if (N <= _sieve_size) return bs[N];
  for (int i = 0; i < (int)primes.size(); i
     ++)
    if (N % primes[i] == 0) return false;
  return true;
}

vi primeFactors(ll N) {
  vi factors;
  ll PF_idx = 0, PF = primes[PF_idx];
  while (N != 1 && (PF * PF <= N)) {
    while (N % PF == 0) { N /= PF; factors.
        push_back(PF); }
    PF = primes[++PF_idx];
  }
  if (N != 1) factors.push_back(N);
  return factors;
}

ll numDiv(ll N) {
```

```
  ll PF_idx = 0, PF = primes[PF_idx], ans = 1;
  while (N != 1 && (PF * PF <= N)) {
    ll power = 0;
    while (N % PF == 0) { N /= PF; power++; }
    ans *= (power + 1);
    PF = primes[++PF_idx];
  }
  if (N != 1) ans *= 2;
  return ans;
}

ll sumDiv(ll N) {
  ll PF_idx = 0, PF = primes[PF_idx], ans = 1;
  while (N != 1 && (PF * PF <= N)) {
    ll power = 0;
    while (N % PF == 0) { N /= PF; power++; }
    ans *= ((ll)pow((double)PF, power + 1.0) - 1) / (PF - 1);
    PF = primes[++PF_idx];
  }
  if (N != 1) ans *= ((ll)pow((double)N, 2.0) - 1) / (N - 1);
  return ans;
}

ll EulerPhi(ll N) {
  ll PF_idx = 0, PF = primes[PF_idx], ans = N;
  while (N != 1 && (PF * PF <= N)) {
    if (N % PF == 0) ans -= ans / PF;
    while (N % PF == 0) N /= PF;
    PF = primes[++PF_idx];
  }
  if (N != 1) ans -= ans / N;
  return ans;
}
```

2.2 Gaussian Elimination

```
#define MAX_N 3
   // adjust this value as needed
struct AugmentedMatrix { double mat[MAX_N][MAX_N + 1]; };
struct ColumnVector { double vec[MAX_N]; };

ColumnVector GaussianElimination(int N, AugmentedMatrix Aug) {
  // input: N, Augmented Matrix Aug, output: Column vector X, the answer
  int i, j, k, l; double t;

  for (i = 0; i < N - 1; i++) {          // the forward elimination phase
    l = i;
    for (j = i + 1; j < N; j++)       // which row has largest column value
      if (fabs(Aug.mat[j][i]) > fabs(Aug.mat[l][i]))
        l = j;

          // remember this row l
    // swap this pivot row, reason: minimize floating point error
    for (k = i; k <= N; k++)            // t is a temporary double variable
      t = Aug.mat[i][k], Aug.mat[i][k] = Aug.mat[l][k], Aug.mat[l][k] = t;
    for (j = i + 1; j < N; j++)     // the actual forward elimination phase
      for (k = N; k >= i; k--)
        Aug.mat[j][k] -= Aug.mat[i][k] * Aug.mat[j][i] / Aug.mat[i][i];
  }

  ColumnVector Ans;
    // the back substitution phase
  for (j = N - 1; j >= 0; j--) {
                          // start from back
    for (t = 0.0, k = j + 1; k < N; k++) t += Aug.mat[j][k] * Ans.vec[k];
    Ans.vec[j] = (Aug.mat[j][N] - t) / Aug.mat[j][j]; // the answer is here
  }
  return Ans;
}

int main() {
  AugmentedMatrix Aug;
  Aug.mat[0][0] = 1; Aug.mat[0][1] = 1; Aug.mat[0][2] = 2; Aug.mat[0][3] = 9;
  Aug.mat[1][0] = 2; Aug.mat[1][1] = 4; Aug.mat[1][2] = -3; Aug.mat[1][3] = 1;
  Aug.mat[2][0] = 3; Aug.mat[2][1] = 6; Aug.mat[2][2] = -5; Aug.mat[2][3] = 0;

  ColumnVector X = GaussianElimination(3, Aug);
  printf("X = %.1lf, Y = %.1lf, Z = %.1lf\n",
```

```
        X.vec[0], X.vec[1], X.vec[2]);
    return 0;
}
```

2.3 Linear Diophantine Equation

```
ll x,y,d;

void solve(ll a,ll b){
        if(b == 0){
                x = 1;
                y = 0;
                d = a;
                return ;
        }
        solve(b, a%b);
        ll x1 = y;
        ll y1 = x -(a/b)*y;
        x = x1;
        y = y1;
}

int main()
{

        #ifndef ONLINE_JUDGE
        freopen("input.txt","r",stdin);
        freopen("output.txt","w",stdout);
        #endif

        ll a,b,n;
        cin >> n >> a >> b;

        solve(a,b);

        if(n % d!=0){
                cout << "NO";
        }
        else{
                ll zmax,zmin;
                ll p,q;
                //cout << x << " " << y << "
                    " << d << "\n";
                ll xres1,yres1,xres2,yres2;
                p = (n*x)/d;
                q = (n*y)/d;
                //cout << a*p + b*q << "\n";
                //cout << p << " " << q << "\
                    n";
                zmax = (q*d)/a;
                zmin = -1*(p*d)/b;
                //cout << zmin << " " << zmax
                    << "\n";
                xres1 = p+((b/d)*zmin);
                yres1 = q-((a/d)*zmin);
                xres2 = p+((b/d)*zmax);
                yres2 = q-((a/d)*zmax);
                if(xres1 >= 0 && yres1 >= 0){
                        cout << "YES\n" <<
                            xres1 << " " <<
                            yres1;
                }
                else if(xres2 >=0 && yres2 >=
                    0){
                        cout << "YES\n" <<
                            xres2 << " " <<
                            yres2;
                }
                else{
                        cout << "NO\n";
                }
        }

        return 0;
}
```

3 String Algorithms

3.1 KMP

```
#define MAX_N 100010

char T[MAX_N], P[MAX_N]; // T = text, P =
    pattern
int b[MAX_N], n, m; // b = back table, n =
    length of T, m = length of P

void kmpPreprocess() { // call this before
    calling kmpSearch()
  int i = 0, j = -1; b[0] = -1; // starting
      values
  while (i < m) { // pre-process the pattern
      string P
    while (j >= 0 && P[i] != P[j]) j = b[j];
        // if different, reset j using b
    i++; j++; // if same, advance both
        pointers
    b[i] = j; // observe i = 8, 9, 10, 11, 12
        with j = 0, 1, 2, 3, 4
```

```
} }            // in the example of P = "SEVENTY SEVEN" above

void kmpSearch() { // this is similar as kmpPreprocess(), but on string T
  int i = 0, j = 0; // starting values
  while (i < n) { // search through string T
    while (j >= 0 && T[i] != P[j]) j = b[j]; // if different, reset j using b
    i++; j++; // if same, advance both pointers
    if (j == m) { // a match found when j == m
      printf("P is found at index %d in T\n", i - j);
      j = b[j]; // prepare j for the next possible match
} } }
```

3.2 Suffix Array

```
typedef pair<int, int> ii;

#define MAX_N 100010 // second approach: O(n log n)
char T[MAX_N];                     // the input string, up to 100K characters
int n;                                               // the length of input string
int RA[MAX_N], tempRA[MAX_N];          // rank array and temporary rank array
int SA[MAX_N], tempSA[MAX_N];      // suffix array and temporary suffix array
int c[MAX_N];                                       // for counting/radix sort

char P[MAX_N];                   // the pattern string (for string matching)
int m; // the length of pattern string

int Phi[MAX_N];                          // for computing longest common prefix
int PLCP[MAX_N];
int LCP[MAX_N];  // LCP[i] stores the LCP between previous suffix T+SA[i-1]   //

bool cmp(int a, int b) { return strcmp(T + a, T + b) < 0; }     // compare

void constructSA_slow() {                 // cannot go beyond 1000 characters
  for (int i = 0; i < n; i++) SA[i] = i; // initial SA: {0, 1, 2, ..., n-1}
  sort(SA, SA + n, cmp); // sort: O(n log n) * compare: O(n) = O(n^2 log n)
}

void countingSort(int k) {

  // O(n)
  int i, sum, maxi = max(300, n);   // up to 255 ASCII chars or length of n
  memset(c, 0, sizeof c);                          // clear frequency table
  for (i = 0; i < n; i++)        // count the frequency of each integer rank
    c[i + k < n ? RA[i + k] : 0]++;
  for (i = sum = 0; i < maxi; i++) {
    int t = c[i]; c[i] = sum; sum += t;
  }
  for (i = 0; i < n; i++)          // shuffle the suffix array if necessary
    tempSA[c[SA[i]+k < n ? RA[SA[i]+k] : 0]++] = SA[i];
  for (i = 0; i < n; i++)                         // update the suffix array SA
    SA[i] = tempSA[i];
}

void constructSA() {         // this version can go up to 100000 characters
  int i, k, r;
  for (i = 0; i < n; i++) RA[i] = T[i];                  // initial rankings
```

```
  for (i = 0; i < n; i++) SA[i] = i;      // initial SA: {0, 1, 2, ..., n-1}
  for (k = 1; k < n; k <<= 1) {        // repeat sorting process log n times
    countingSort(k);  // actually radix sort: sort based on the second item
    countingSort(0);          // then (stable) sort based on the first item
    tempRA[SA[0]] = r = 0;             // re-ranking; start from rank r = 0
    for (i = 1; i < n; i++)                          // compare adjacent suffixes
      tempRA[SA[i]] = // if same pair => same rank r; otherwise, increase r
      (RA[SA[i]] == RA[SA[i-1]] && RA[SA[i]+k] == RA[SA[i-1]+k]) ? r : ++r;
    for (i = 0; i < n; i++)                          // update the rank array RA
      RA[i] = tempRA[i];
    if (RA[SA[n-1]] == n-1) break;                 // nice optimization trick
} }
void computeLCP_slow() {
  LCP[0] = 0;                                              // default value
  for (int i = 1; i < n; i++) {                     // compute LCP by definition
    int L = 0;                                       // always reset L to 0
    while (T[SA[i] + L] == T[SA[i-1] + L]) L++;      // same L-th char, L++
    LCP[i] = L;
} }
void computeLCP() {
  int i, L;
  Phi[SA[0]] = -1;                                         // default value
  for (i = 1; i < n; i++)                                  // compute Phi in O(n)
    Phi[SA[i]] = SA[i-1];    // remember which suffix is behind this suffix
  for (i = L = 0; i < n; i++) {             // compute Permuted LCP in O(n)
    if (Phi[i] == -1) { PLCP[i] = 0; continue; }        // special case
    while (T[i + L] == T[Phi[i] + L]) L++;        // L increased max n times
    PLCP[i] = L;
    L = max(L-1, 0);                                  // L decreased max n times
  }
  for (i = 0; i < n; i++)                                  // compute LCP in O(n)
    LCP[i] = PLCP[SA[i]];    // put the permuted LCP to the correct position
}
ii stringMatching() {                         // string matching in O(m log n)
  int lo = 0, hi = n-1, mid = lo;               // valid matching = [0..n-1]
  while (lo < hi) {                                        // find lower bound
    mid = (lo + hi) / 2;                                  // this is round down
    int res = strncmp(T + SA[mid], P, m);  // try to find P in suffix 'mid'
    if (res >= 0) hi = mid;        // prune upper half (notice the >= sign)
    else          lo = mid + 1;           // prune lower half including mid
  }                                          // observe '=' in "res >= 0" above
  if (strncmp(T + SA[lo], P, m) != 0) return ii(-1, -1);    // if not found
  ii ans; ans.first = lo;
  lo = 0; hi = n - 1; mid = lo;
  while (lo < hi) {            // if lower bound is found, find upper bound
    mid = (lo + hi) / 2;
    int res = strncmp(T + SA[mid], P, m);
    if (res > 0) hi = mid;                                 // prune upper half
    else          lo = mid + 1;           // prune lower half including mid
  }                                                   // (notice the
```

```
    selected branch when res == 0)
  if (strncmp(T + SA[hi], P, m) != 0) hi--;
                         // special case
  ans.second = hi;
  return ans;
} // return lower/upperbound as first/second
  item of the pair, respectively

ii LRS() {                   // returns a pair
   (the LRS length and its index)
  int i, idx = 0, maxLCP = -1;
  for (i = 1; i < n; i++)
                              // O(n), start
    from i = 1
    if (LCP[i] > maxLCP)
      maxLCP = LCP[i], idx = i;
  return ii(maxLCP, idx);
}

int owner(int idx) { return (idx < n-m-1) ? 1
   : 2; }

ii LCS() {                   // returns a pair
   (the LCS length and its index)
  int i, idx = 0, maxLCP = -1;
  for (i = 1; i < n; i++)
                              // O(n), start
    from i = 1
    if (owner(SA[i]) != owner(SA[i-1]) && LCP
       [i] > maxLCP)
      maxLCP = LCP[i], idx = i;
  return ii(maxLCP, idx);
}
```

3.3 String Alignment

```
  char A[20] = "ACAATCC", B[20] = "AGCATGC";
  int n = (int)strlen(A), m = (int)strlen(B);
  int i, j, table[20][20]; // Needleman
    Wunsnch's algorithm

  memset(table, 0, sizeof table);
  // insert/delete = -1 point
  for (i = 1; i <= n; i++)
    table[i][0] = i * -1;
  for (j = 1; j <= m; j++)
    table[0][j] = j * -1;

  for (i = 1; i <= n; i++)
    for (j = 1; j <= m; j++) {
      // match = 2 points, mismatch = -1
         point
      table[i][j] = table[i - 1][j - 1] + (A[
         i - 1] == B[j - 1] ? 2 : -1); //
         cost for match or mismatches
      // insert/delete = -1 point
      table[i][j] = max(table[i][j], table[i
         - 1][j] - 1); // delete
      table[i][j] = max(table[i][j], table[i
         ][j - 1] - 1); // insert
    }

  printf("DP table:\n");
  for (i = 0; i <= n; i++) {
    for (j = 0; j <= m; j++)
      printf("%3d", table[i][j]);
    printf("\n");
  }
  printf("Maximum Alignment Score: %d\n",
     table[n][m]);
```

4 Computational Geometry

4.1 Points and Lines

```
#define INF 1e9
#define EPS 1e-9
#define PI acos(-1.0) // important constant;
   alternative #define PI (2.0 * acos(0.0))

double DEG_to_RAD(double d) { return d * PI /
   180.0; }

double RAD_to_DEG(double r) { return r *
   180.0 / PI; }

// struct point_i { int x, y; };    // basic
   raw form, minimalist mode
struct point_i { int x, y;      // whenever
   possible, work with point_i
  point_i() { x = y = 0; }
                               // default
     constructor
  point_i(int _x, int _y) : x(_x), y(_y) {}
     };           // user-defined

struct point { double x, y;    // only used if
    more precision is needed
```

```
  point() { x = y = 0.0; }                      // default constructor
  point(double _x, double _y) : x(_x), y(_y) {}      // user-defined
  bool operator < (point other) const { // override less than operator
    if (fabs(x - other.x) > EPS)                   // useful for sorting
      return x < other.x;            // first criteria , by x-coordinate
    return y < other.y; }            // second criteria, by y-coordinate
  // use EPS (1e-9) when testing equality of two floating points
  bool operator == (point other) const {
   return (fabs(x - other.x) < EPS && (fabs(y - other.y) < EPS)); } };

double dist(point p1, point p2) {
                  // Euclidean distance
                      // hypot(dx, dy)
                         returns sqrt(dx * dx + dy * dy)
  return hypot(p1.x - p2.x, p1.y - p2.y); }
               // return double

// rotate p by theta degrees CCW w.r.t origin (0, 0)
point rotate(point p, double theta) {
  double rad = DEG_to_RAD(theta);     // multiply theta with PI / 180.0
  return point(p.x * cos(rad) - p.y * sin(rad),
               p.x * sin(rad) + p.y * cos(rad)); }

struct line { double a, b, c; };          // a way to represent a line

// the answer is stored in the third parameter (pass by reference)
void pointsToLine(point p1, point p2, line &l) {
  if (fabs(p1.x - p2.x) < EPS) {
                  // vertical line is fine
    l.a = 1.0;   l.b = 0.0;   l.c = -p1.x;
                 // default values
  } else {
    l.a = -(double)(p1.y - p2.y) / (p1.x - p2.x);
    l.b = 1.0;              // IMPORTANT: we fix the value of b to 1.0
    l.c = -(double)(l.a * p1.x) - p1.y;
} }

// not needed since we will use the more robust form: ax + by + c = 0 (see above)
struct line2 { double m, c; };       // another way to represent a line

int pointsToLine2(point p1, point p2, line2 &l) {
 if (abs(p1.x - p2.x) < EPS) {           // special case: vertical line
   l.m = INF;                   // l contains m = INF and c = x_value
   l.c = p1.x;                  // to denote vertical line x = x_value
   return 0;   // we need this return variable to differentiate result
 }
 else {
   l.m = (double)(p1.y - p2.y) / (p1.x - p2.x);
   l.c = p1.y - l.m * p1.x;
   return 1;     // l contains m and c of the line equation y = mx + c
} }

bool areParallel(line l1, line l2) {         // check coefficients a & b
  return (fabs(l1.a-l2.a) < EPS) && (fabs(l1.b-l2.b) < EPS); }

bool areSame(line l1, line l2) {             // also check coefficient c
  return areParallel(l1 ,l2) && (fabs(l1.c - l2.c) < EPS); }

// returns true (+ intersection point) if two lines are intersect
bool areIntersect(line l1, line l2, point &p) {
  if (areParallel(l1, l2)) return false;
                   // no intersection
  // solve system of 2 linear algebraic equations with 2 unknowns
  p.x = (l2.b * l1.c - l1.b * l2.c) / (l2.a * l1.b - l1.a * l2.b);
  // special case: test for vertical line to avoid division by zero
```

```
  if (fabs(l1.b) > EPS) p.y = -(l1.a * p.x +
     l1.c);
  else                  p.y = -(l2.a * p.x +
     l2.c);
  return true; }

struct vec { double x, y;  // name: 'vec' is
   different from STL vector
  vec(double _x, double _y) : x(_x), y(_y) {}
      };

vec toVec(point a, point b) {       //
   convert 2 points to vector a->b
  return vec(b.x - a.x, b.y - a.y); }

vec scale(vec v, double s) {        //
   nonnegative s = [<1 .. 1 .. >1]
  return vec(v.x * s, v.y * s); }
                   // shorter.same.longer

point translate(point p, vec v) {        //
   translate p according to v
  return point(p.x + v.x , p.y + v.y); }

// convert point and gradient/slope to line
void pointSlopeToLine(point p, double m, line
    &l) {
  l.a = -m;
     // always -m
  l.b = 1;
     // always 1
  l.c = -((l.a * p.x) + (l.b * p.y)); }
                    // compute this

void closestPoint(line l, point p, point &ans
  ) {
  line perpendicular;         //
     perpendicular to l and pass through p
  if (fabs(l.b) < EPS) {             //
     special case 1: vertical line
    ans.x = -(l.c);   ans.y = p.y;
       return; }

  if (fabs(l.a) < EPS) {            //
     special case 2: horizontal line
    ans.x = p.x;      ans.y = -(l.c);
       return; }

  pointSlopeToLine(p, 1 / l.a, perpendicular)
     ;          // normal line
  // intersect line l with this perpendicular
      line
  // the intersection point is the closest
     point
  areIntersect(l, perpendicular, ans); }

// returns the reflection of point on a line
void reflectionPoint(line l, point p, point &
   ans) {
  point b;
  closestPoint(l, p, b);
     // similar to distToLine
  vec v = toVec(p, b);
                              // create a
     vector
  ans = translate(translate(p, v), v); }
           // translate p twice

double dot(vec a, vec b) { return (a.x * b.x
   + a.y * b.y); }

double norm_sq(vec v) { return v.x * v.x + v.
   y * v.y; }

// returns the distance from p to the line
   defined by
// two points a and b (a and b must be
   different)
// the closest point is stored in the 4th
   parameter (byref)
double distToLine(point p, point a, point b,
   point &c) {
  // formula: c = a + u * ab
  vec ap = toVec(a, p), ab = toVec(a, b);
  double u = dot(ap, ab) / norm_sq(ab);
  c = translate(a, scale(ab, u));
                          // translate a to c
  return dist(p, c); }          // Euclidean
      distance between p and c

// returns the distance from p to the line
   segment ab defined by
// two points a and b (still OK if a == b)
// the closest point is stored in the 4th
   parameter (byref)
double distToLineSegment(point p, point a,
   point b, point &c) {
  vec ap = toVec(a, p), ab = toVec(a, b);
  double u = dot(ap, ab) / norm_sq(ab);
  if (u < 0.0) { c = point(a.x, a.y);
                          // closer to a
```

```
    return dist(p, a); }          // Euclidean
        distance between p and a
  if (u > 1.0) { c = point(b.x, b.y);
                          // closer to b
    return dist(p, b); }          // Euclidean
        distance between p and b
  return distToLine(p, a, b, c); }
     // run distToLine as above

double angle(point a, point o, point b) {  //
    returns angle aob in rad
  vec oa = toVec(o, a), ob = toVec(o, b);
  return acos(dot(oa, ob) / sqrt(norm_sq(oa)
     * norm_sq(ob))); }

double cross(vec a, vec b) { return a.x * b.y
    - a.y * b.x; }

//// another variant
//int area2(point p, point q, point r) { //
   returns 'twice' the area of this triangle
   A-B-c
//  return p.x * q.y - p.y * q.x +
//         q.x * r.y - q.y * r.x +
//         r.x * p.y - r.y * p.x;
//}

// note: to accept collinear points, we have
   to change the '> 0'
// returns true if point r is on the left
   side of line pq
bool ccw(point p, point q, point r) {
  return cross(toVec(p, q), toVec(p, r)) > 0;
     }

// returns true if point r is on the same
   line as the line pq
bool collinear(point p, point q, point r) {
  return fabs(cross(toVec(p, q), toVec(p, r))
     ) < EPS; }
```

4.2 Circles

```
double DEG_to_RAD(double d) { return d * PI /
    180.0; }

double RAD_to_DEG(double r) { return r *
   180.0 / PI; }

struct point_i { int x, y;     // whenever
   possible, work with point_i
  point_i() { x = y = 0; }
                            // default
     constructor
  point_i(int _x, int _y) : x(_x), y(_y) {}
     };            // constructor

struct point { double x, y;    // only used if
    more precision is needed
  point() { x = y = 0.0; }
                            // default
     constructor
  point(double _x, double _y) : x(_x), y(_y)
     {} };        // constructor

int insideCircle(point_i p, point_i c, int r)
    { // all integer version
  int dx = p.x - c.x, dy = p.y - c.y;
  int Euc = dx * dx + dy * dy, rSq = r * r;
               // all integer
  return Euc < rSq ? 0 : Euc == rSq ? 1 : 2;
     } //inside/border/outside

bool circle2PtsRad(point p1, point p2, double
    r, point &c) {
  double d2 = (p1.x - p2.x) * (p1.x - p2.x) +
              (p1.y - p2.y) * (p1.y - p2.y);
  double det = r * r / d2 - 0.25;
  if (det < 0.0) return false;
  double h = sqrt(det);
  c.x = (p1.x + p2.x) * 0.5 + (p1.y - p2.y) *
     h;
  c.y = (p1.y + p2.y) * 0.5 + (p2.x - p1.x) *
     h;
  return true; }          // to get the other
     center, reverse p1 and p2
```

4.3 Triangles

```
double DEG_to_RAD(double d) { return d * PI /
    180.0; }

double RAD_to_DEG(double r) { return r *
   180.0 / PI; }

struct point_i { int x, y;     // whenever
   possible, work with point_i
  point_i() { x = y = 0; }
                            // default
     constructor
```

```
  point_i(int _x, int _y) : x(_x), y(_y) {}
  };            // constructor

struct point { double x, y;   // only used if
    more precision is needed
  point() { x = y = 0.0; }
                          // default
    constructor
  point(double _x, double _y) : x(_x), y(_y)
    {} };       // constructor

double dist(point p1, point p2) {
  return hypot(p1.x - p2.x, p1.y - p2.y); }

double perimeter(double ab, double bc, double
    ca) {
  return ab + bc + ca; }

double perimeter(point a, point b, point c) {
  return dist(a, b) + dist(b, c) + dist(c, a)
    ; }

double area(double ab, double bc, double ca)
  {
  // Heron's formula, split sqrt(a * b) into
    sqrt(a) * sqrt(b); in implementation
  double s = 0.5 * perimeter(ab, bc, ca);
  return sqrt(s) * sqrt(s - ab) * sqrt(s - bc
    ) * sqrt(s - ca); }

double area(point a, point b, point c) {
  return area(dist(a, b), dist(b, c), dist(c,
     a)); }

//
   ==========================================================================

// from ch7_01_points_lines
struct line { double a, b, c; }; // a way to
   represent a line

// the answer is stored in the third
   parameter (pass by reference)
void pointsToLine(point p1, point p2, line &l
  ) {
  if (fabs(p1.x - p2.x) < EPS) {
                 // vertical line is fine
    l.a = 1.0;   l.b = 0.0;   l.c = -p1.x;
                // default values
  } else {
    l.a = -(double)(p1.y - p2.y) / (p1.x - p2
      .x);
    l.b = 1.0;                // IMPORTANT: we
      fix the value of b to 1.0
    l.c = -(double)(l.a * p1.x) - p1.y;
} }

bool areParallel(line l1, line l2) {
   // check coefficient a + b
  return (fabs(l1.a-l2.a) < EPS) && (fabs(l1.
    b-l2.b) < EPS); }

// returns true (+ intersection point) if two
    lines are intersect
bool areIntersect(line l1, line l2, point &p)
   {
  if (areParallel(l1, l2)) return false;
             // no intersection
  // solve system of 2 linear algebraic
     equations with 2 unknowns
  p.x = (l2.b * l1.c - l1.b * l2.c) / (l2.a *
      l1.b - l1.a * l2.b);
  // special case: test for vertical line to
     avoid division by zero
  if (fabs(l1.b) > EPS) p.y = -(l1.a * p.x +
     l1.c);
  else                  p.y = -(l2.a * p.x +
     l2.c);
  return true; }

struct vec { double x, y;  // name: 'vec' is
   different from STL vector
  vec(double _x, double _y) : x(_x), y(_y) {}
      };

vec toVec(point a, point b) {       //
   convert 2 points to vector a->b
  return vec(b.x - a.x, b.y - a.y); }

vec scale(vec v, double s) {        //
   nonnegative s = [<1 .. 1 .. >1]
  return vec(v.x * s, v.y * s); }
                      // shorter.same.longer

point translate(point p, vec v) {        //
   translate p according to v
  return point(p.x + v.x , p.y + v.y); }
//
   ==========================================================================

double rInCircle(double ab, double bc, double
     ca) {
  return area(ab, bc, ca) / (0.5 * perimeter(
     ab, bc, ca)); }
```

```
double rInCircle(point a, point b, point c) {
  return rInCircle(dist(a, b), dist(b, c), dist(c, a)); }

// assumption: the required points/lines functions have been written
// returns 1 if there is an inCircle center, returns 0 otherwise
// if this function returns 1, ctr will be the inCircle center
// and r is the same as rInCircle
int inCircle(point p1, point p2, point p3, point &ctr, double &r) {
  r = rInCircle(p1, p2, p3);
  if (fabs(r) < EPS) return 0;                  // no inCircle center

  line l1, l2;                    // compute these two angle bisectors
  double ratio = dist(p1, p2) / dist(p1, p3);
  point p = translate(p2, scale(toVec(p2, p3), ratio / (1 + ratio)));
  pointsToLine(p1, p, l1);

  ratio = dist(p2, p1) / dist(p2, p3);
  p = translate(p1, scale(toVec(p1, p3), ratio / (1 + ratio)));
  pointsToLine(p2, p, l2);

  areIntersect(l1, l2, ctr);           // get their intersection point
  return 1; }

double rCircumCircle(double ab, double bc, double ca) {
  return ab * bc * ca / (4.0 * area(ab, bc, ca)); }

double rCircumCircle(point a, point b, point c) {
  return rCircumCircle(dist(a, b), dist(b, c), dist(c, a)); }

// assumption: the required points/lines functions have been written
// returns 1 if there is a circumCenter center, returns 0 otherwise
// if this function returns 1, ctr will be the circumCircle center
// and r is the same as rCircumCircle
int circumCircle(point p1, point p2, point p3, point &ctr, double &r){
  double a = p2.x - p1.x, b = p2.y - p1.y;
  double c = p3.x - p1.x, d = p3.y - p1.y;
  double e = a * (p1.x + p2.x) + b * (p1.y + p2.y);
  double f = c * (p1.x + p3.x) + d * (p1.y + p3.y);
  double g = 2.0 * (a * (p3.y - p2.y) - b * (p3.x - p2.x));
  if (fabs(g) < EPS) return 0;

  ctr.x = (d*e - b*f) / g;
  ctr.y = (a*f - c*e) / g;
  r = dist(p1, ctr);  // r = distance from center to 1 of the 3 points
  return 1; }

// returns true if point d is inside the circumCircle defined by a,b,c
int inCircumCircle(point a, point b, point c, point d) {
  return (a.x - d.x) * (b.y - d.y) * ((c.x - d.x) * (c.x - d.x) + (c.y - d.y) * (c.y - d.y)) +
         (a.y - d.y) * ((b.x - d.x) * (b.x - d.x) + (b.y - d.y) * (b.y - d.y)) * (c.x - d.x) +
         ((a.x - d.x) * (a.x - d.x) + (a.y - d.y) * (a.y - d.y)) * (b.x - d.x) * (c.y - d.y) -
         ((a.x - d.x) * (a.x - d.x) + (a.y - d.y) * (a.y - d.y)) * (b.y - d.y) * (c.x - d.x) -
         (a.y - d.y) * (b.x - d.x) * ((c.x - d.x) * (c.x - d.x) + (c.y - d.y) * (c.y - d.y)) -
         (a.x - d.x) * ((b.x - d.x) * (b.x - d.x) + (b.y - d.y) * (b.y - d.y)) * (c.y - d.y) > 0 ? 1 : 0;
}

bool canFormTriangle(double a, double b, double c) {
  return (a + b > c) && (a + c > b) && (b + c > a); }
```

4.4 Polygons

```cpp
double DEG_to_RAD(double d) { return d * PI / 180.0; }

double RAD_to_DEG(double r) { return r * 180.0 / PI; }

struct point { double x, y;   // only used if more precision is needed
  point() { x = y = 0.0; }                      // default constructor
  point(double _x, double _y) : x(_x), y(_y) {}        // user-defined
  bool operator == (point other) const {
   return (fabs(x - other.x) < EPS && (fabs(y - other.y) < EPS)); } };

struct vec { double x, y;  // name: `vec' is different from STL vector
  vec(double _x, double _y) : x(_x), y(_y) {} };

vec toVec(point a, point b) {       // convert 2 points to vector a->b
  return vec(b.x - a.x, b.y - a.y); }

double dist(point p1, point p2) {
                  // Euclidean distance
  return hypot(p1.x - p2.x, p1.y - p2.y); }
               // return double

// returns the perimeter, which is the sum of Euclidian distances
// of consecutive line segments (polygon edges)
double perimeter(const vector<point> &P) {
  double result = 0.0;
  for (int i = 0; i < (int)P.size()-1; i++)  // remember that P[0] = P[n-1]
    result += dist(P[i], P[i+1]);
  return result; }

// returns the area, which is half the determinant
double area(const vector<point> &P) {
  double result = 0.0, x1, y1, x2, y2;
  for (int i = 0; i < (int)P.size()-1; i++) {
    x1 = P[i].x; x2 = P[i+1].x;
    y1 = P[i].y; y2 = P[i+1].y;
    result += (x1 * y2 - x2 * y1);
  }
  return fabs(result) / 2.0; }

double dot(vec a, vec b) { return (a.x * b.x + a.y * b.y); }

double norm_sq(vec v) { return v.x * v.x + v.y * v.y; }

double angle(point a, point o, point b) {  // returns angle aob in rad
  vec oa = toVec(o, a), ob = toVec(o, b);
  return acos(dot(oa, ob) / sqrt(norm_sq(oa) * norm_sq(ob))); }

double cross(vec a, vec b) { return a.x * b.y - a.y * b.x; }

// note: to accept collinear points, we have to change the `> 0'
// returns true if point r is on the left side of line pq
bool ccw(point p, point q, point r) {
  return cross(toVec(p, q), toVec(p, r)) > 0; }

// returns true if point r is on the same line as the line pq
bool collinear(point p, point q, point r) {
  return fabs(cross(toVec(p, q), toVec(p, r))) < EPS; }

// returns true if we always make the same turn while examining
// all the edges of the polygon one by one
bool isConvex(const vector<point> &P) {
  int sz = (int)P.size();
  if (sz <= 3) return false;   // a point/sz=2 or a line/sz=3 is not convex
  bool isLeft = ccw(P[0], P[1], P[2]);
                      // remember one result
  for (int i = 1; i < sz-1; i++)
    // then compare with the others
    if (ccw(P[i], P[i+1], P[(i+2) == sz ? 1 : i+2]) != isLeft)
      return false;            // different sign -> this polygon is concave
  return true; }
                                              // this polygon is convex

// returns true if point p is in either convex/concave polygon P
```

```
bool inPolygon(point pt, const vector<point> &P) {
  if ((int)P.size() == 0) return false;
  double sum = 0;    // assume the first vertex is equal to the last vertex
  for (int i = 0; i < (int)P.size()-1; i++) {
    if (ccw(pt, P[i], P[i+1]))
         sum += angle(P[i], pt, P[i+1]);                   // left turn/ccw
    else sum -= angle(P[i], pt, P[i+1]); }                 // right turn/cw
  return fabs(fabs(sum) - 2*PI) < EPS; }

// line segment p-q intersect with line A-B.
point lineIntersectSeg(point p, point q, point A, point B) {
  double a = B.y - A.y;
  double b = A.x - B.x;
  double c = B.x * A.y - A.x * B.y;
  double u = fabs(a * p.x + b * p.y + c);
  double v = fabs(a * q.x + b * q.y + c);
  return point((p.x * v + q.x * u) / (u+v), (p.y * v + q.y * u) / (u+v)); }

// cuts polygon Q along the line formed by point a -> point b
// (note: the last point must be the same as the first point)
vector<point> cutPolygon(point a, point b, const vector<point> &Q) {
  vector<point> P;
  for (int i = 0; i < (int)Q.size(); i++) {
    double left1 = cross(toVec(a, b), toVec(a, Q[i])), left2 = 0;
    if (i != (int)Q.size()-1) left2 = cross(toVec(a, b), toVec(a, Q[i+1]));
    if (left1 > -EPS) P.push_back(Q[i]);      // Q[i] is on the left of ab
    if (left1 * left2 < -EPS)        // edge (Q[i], Q[i+1]) crosses line ab
      P.push_back(lineIntersectSeg(Q[i], Q[i+1], a, b));
  }
  if (!P.empty() && !(P.back() == P.front()))
    P.push_back(P.front());          // make P's first point = P's last point
  return P; }

point pivot;

bool angleCmp(point a, point b) {                   // angle-sorting function
  if (collinear(pivot, a, b))                                   // special case
    return dist(pivot, a) < dist(pivot, b);      // check which one is closer
  double d1x = a.x - pivot.x, d1y = a.y - pivot.y;
  double d2x = b.x - pivot.x, d2y = b.y - pivot.y;
  return (atan2(d1y, d1x) - atan2(d2y, d2x)) < 0; }   // compare two angles

vector<point> CH(vector<point> P) {   // the content of P may be reshuffled
  int i, j, n = (int)P.size();
  if (n <= 3) {
    if (!(P[0] == P[n-1])) P.push_back(P[0]); // safeguard from corner case
    return P;                           // special case, the CH is P itself
  }

  // first, find P0 = point with lowest Y and if tie: rightmost X
  int P0 = 0;
  for (i = 1; i < n; i++)
    if (P[i].y < P[P0].y || (P[i].y == P[P0].y && P[i].x > P[P0].x))
      P0 = i;

  point temp = P[0]; P[0] = P[P0]; P[P0] = temp;    // swap P[P0] with P[0]

  // second, sort points by angle w.r.t. pivot P0
  pivot = P[0];                    // use this global variable as reference
  sort(++P.begin(), P.end(), angleCmp);              // we do not sort P[0]

  // third, the ccw tests
  vector<point> S;
  S.push_back(P[n-1]); S.push_back(P[0]); S.push_back(P[1]);   // initial S
  i = 2;

    // then, we check the rest
  while (i < n) {           // note: N must be >= 3 for this method to work
    j = (int)S.size()-1;
```

```
    if (ccw(S[j-1], S[j], P[i])) S.push_back(
       P[i++]);  // left turn, accept
    else S.pop_back(); }   // or pop the top
       of S until we have a left turn
  return S; }
     // return the result
```

5 DP

5.1 Longest Increasing Subsequence

```
#define MAX_N 100000

void print_array(const char *s, int a[], int
   n) {
  for (int i = 0; i < n; ++i) {
    if (i) printf(", ");
    else printf("%s: [", s);
    printf("%d", a[i]);
  }
  printf("]\n");
}

void reconstruct_print(int end, int a[], int
   p[]) {
  int x = end;
  stack<int> s;
  for (; p[x] >= 0; x = p[x]) s.push(a[x]);
  printf("[%d", a[x]);
  for (; !s.empty(); s.pop()) printf(", %d",
     s.top());
  printf("]\n");
}

int main() {
  int n = 11, A[] = {-7, 10, 9, 2, 3, 8, 8,
     1, 2, 3, 4};
  int L[MAX_N], L_id[MAX_N], P[MAX_N];

  int lis = 0, lis_end = 0;
  for (int i = 0; i < n; ++i) {
    int pos = lower_bound(L, L + lis, A[i]) -
       L;
    L[pos] = A[i];
    L_id[pos] = i;
    P[i] = pos ? L_id[pos - 1] : -1;
    if (pos + 1 > lis) {
      lis = pos + 1;
      lis_end = i;
    }
    printf("Considering element A[%d] = %d\n"
       , i, A[i]);
    printf("LIS ending at A[%d] is of length
       %d: ", i, pos + 1);
    reconstruct_print(i, A, P);
    print_array("L is now", L, lis);
    printf("\n");
  }

  printf("Final LIS is of length %d: ", lis);
  reconstruct_print(lis_end, A, P);
  return 0;
}
```

5.2 Max 1D Range Sum

```
#include <algorithm>
#include <cstdio>
using namespace std;

int main() {
  int n = 9, A[] = { 4, -5, 4, -3, 4, 4, -4,
     4, -5 };   // a sample array A
  int running_sum = 0, ans = 0;
  for (int i = 0; i < n; i++)
                                          //
     O(n)
    if (running_sum + A[i] >= 0) {  // the
       overall running sum is still +ve
      running_sum += A[i];
      ans = max(ans, running_sum);
         // keep the largest RSQ overall
    }
    else         // the overall running sum is
       -ve, we greedily restart here
      running_sum = 0;       // because
         starting from 0 is better for future
                             // iterations than
                                starting from
                               -ve running sum
  printf("Max 1D Range Sum = %d\n", ans);
                          // should be 9
} // return 0;
```

6 Graphs

6.1 BFS

```
typedef pair<int, int> ii;        // In this chapter, we will frequently use these
typedef vector<ii> vii;         // three data type shortcuts. They may look cryptic
typedef vector<int> vi;     // but shortcuts are useful in competitive programming

int V, E, a, b, s;
vector<vii> AdjList;
vi p;                                          // addition: the predecessor/parent vector

void printPath(int u) {     // simple function to extract information from `vi p'
  if (u == s) { printf("%d", u); return; }
  printPath(p[u]);    // recursive call: to make the output format: s -> ... -> t
  printf(" %d", u); }

int main() {
  scanf("%d %d", &V, &E);

  AdjList.assign(V, vii()); // assign blank vectors of pair<int, int>s to AdjList
  for (int i = 0; i < E; i++) {
    scanf("%d %d", &a, &b);
    AdjList[a].push_back(ii(b, 0));
    AdjList[b].push_back(ii(a, 0));
  }

  // as an example, we start from this source, see Figure 4.3
  s = 5;

  // BFS routine
  // inside int main() -- we do not use recursion, thus we do not need to create separate function!
  vi dist(V, 1000000000); dist[s] = 0;   // distance to source is 0 (default)
  queue<int> q; q.push(s);                                   // start from source
  p.assign(V, -1); // to store parent information (p must be a global variable!)
  int layer = -1;                                // for our output printing purpose
  bool isBipartite = true;       // addition of one boolean flag, initially true

  while (!q.empty()) {
    int u = q.front(); q.pop();
                                              // queue: layer by layer!
    if (dist[u] != layer) printf("\nLayer %d: ", dist[u]);
    layer = dist[u];
    printf("visit %d, ", u);
    for (int j = 0; j < (int)AdjList[u].size(); j++) {
      ii v = AdjList[u][j];
                                              // for each neighbors of u
      if (dist[v.first] == 1000000000) {
        dist[v.first] = dist[u] + 1;
                                              // v unvisited + reachable
        p[v.first] = u;          // addition: the parent of vertex v->first is u
        q.push(v.first);
                                              // enqueue v for next step
      }
      else if ((dist[v.first] % 2) == (dist[u] % 2))       // same parity
        isBipartite = false;
  } }

  printf("\nShortest path: ");
  printPath(7), printf("\n");
  printf("isBipartite? %d\n", isBipartite);

  return 0;
}
```

6.2 DFS

```
#define DFS_WHITE -1 // normal DFS, do not change this with other values (other than 0), because we usually use memset with conjunction with DFS_WHITE
```

```
#define DFS_BLACK 1
vector<vii> AdjList;

void printThis(char* message) {
  printf("====================================\n");
  printf("%s\n", message);
  printf("====================================\n");
}

vi dfs_num;     // this variable has to be global, we cannot put it in recursion
int numCC;

void dfs(int u) {           // DFS for normal usage: as graph traversal algorithm
  printf(" %d", u);                                         // this vertex is visited
  dfs_num[u] = DFS_BLACK;      // important step: we mark this vertex as visited
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j];                                // v is a (neighbor, weight) pair
    if (dfs_num[v.first] == DFS_WHITE)            // important check to avoid cycle
      dfs(v.first);      // recursively visits unvisited neighbors v of vertex u
} }

// note: this is not the version on implicit graph
void floodfill(int u, int color) {
  dfs_num[u] = color;                                      // not just a generic DFS_BLACK
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j];
    if (dfs_num[v.first] == DFS_WHITE)
      floodfill(v.first, color);
} }

vi topoSort;              // global vector to store the toposort in reverse order

void dfs2(int u) {    // change function name to differentiate with original dfs
  dfs_num[u] = DFS_BLACK;
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j];
    if (dfs_num[v.first] == DFS_WHITE)
      dfs2(v.first);
  }
  topoSort.push_back(u); }
  // that is, this is the only change

#define DFS_GRAY 2                      // one more color for graph edges property check
vi dfs_parent;      // to differentiate real back edge versus bidirectional edge

void graphCheck(int u) {                       // DFS for checking graph edge properties
  dfs_num[u] = DFS_GRAY;   // color this as DFS_GRAY (temp) instead of DFS_BLACK
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j];
    if (dfs_num[v.first] == DFS_WHITE) {
      // Tree Edge, DFS_GRAY to DFS_WHITE
      dfs_parent[v.first] = u;
                                  // parent of this children is me
      graphCheck(v.first);
    }
    else if (dfs_num[v.first] == DFS_GRAY) {
                            // DFS_GRAY to DFS_GRAY
      if (v.first == dfs_parent[u])
        // to differentiate these two cases
        printf(" Bidirectional (%d, %d) - (%d, %d)\n", u, v.first, v.first, u);
      else  // the most frequent application: check if the given graph is cyclic
        printf(" Back Edge (%d, %d) (Cycle)\n", u, v.first);
    }
    else if (dfs_num[v.first] == DFS_BLACK)
                            // DFS_GRAY to DFS_BLACK
      printf(" Forward/Cross Edge (%d, %d)\n", u, v.first);
  }
  dfs_num[u] = DFS_BLACK;     // after recursion, color this as DFS_BLACK (DONE
```

```
      )
}
vi dfs_low;       // additional information
   for articulation points/bridges/SCCs
vi articulation_vertex;
int dfsNumberCounter, dfsRoot, rootChildren;

void articulationPointAndBridge(int u) {
  dfs_low[u] = dfs_num[u] = dfsNumberCounter
     ++;      // dfs_low[u] <= dfs_num[u]
  for (int j = 0; j < (int)AdjList[u].size();
      j++) {
    ii v = AdjList[u][j];
    if (dfs_num[v.first] == DFS_WHITE) {
                                    // a tree
       edge
      dfs_parent[v.first] = u;
      if (u == dfsRoot) rootChildren++;  //
         special case, count children of root

      articulationPointAndBridge(v.first);

      if (dfs_low[v.first] >= dfs_num[u])
                          // for articulation
         point
        articulation_vertex[u] = true;
                         // store this
           information first
      if (dfs_low[v.first] > dfs_num[u])
                                     // for
         bridge
        printf(" Edge (%d, %d) is a bridge\n"
           , u, v.first);
      dfs_low[u] = min(dfs_low[u], dfs_low[v.
         first]);       // update dfs_low[u]
    }
    else if (v.first != dfs_parent[u])
      // a back edge and not direct cycle
      dfs_low[u] = min(dfs_low[u], dfs_num[v.
         first]);       // update dfs_low[u]
} }
vi S, visited;
                                           //
   additional global variables
int numSCC;

void tarjanSCC(int u) {
  dfs_low[u] = dfs_num[u] = dfsNumberCounter
     ++;      // dfs_low[u] <= dfs_num[u]
  S.push_back(u);           // stores u in a
     vector based on order of visitation
  visited[u] = 1;
  for (int j = 0; j < (int)AdjList[u].size();
      j++) {
    ii v = AdjList[u][j];
    if (dfs_num[v.first] == DFS_WHITE)
      tarjanSCC(v.first);
    if (visited[v.first])
                                          //
       condition for update
      dfs_low[u] = min(dfs_low[u], dfs_low[v.
         first]);
  }

  if (dfs_low[u] == dfs_num[u]) {        //
     if this is a root (start) of an SCC
    printf("SCC %d:", ++numSCC);
       // this part is done after recursion
    while (1) {
      int v = S.back(); S.pop_back(); visited
         [v] = 0;
      printf(" %d", v);
      if (u == v) break;
    }
    printf("\n");
} }
```

6.3 Dijkstra's

```
#define INF 1000000000
  int V, E, s, u, v, w;
  vector<vii> AdjList;

  scanf("%d %d %d", &V, &E, &s);

  AdjList.assign(V, vii()); // assign blank
     vectors of pair<int, int>s to AdjList
  for (int i = 0; i < E; i++) {
    scanf("%d %d %d", &u, &v, &w);
    AdjList[u].push_back(ii(v, w));      //
       directed graph
  }

  // Dijkstra routine
  vi dist(V, INF); dist[s] = 0;
                          // INF = 1B to avoid
     overflow
  priority_queue< ii, vector<ii>, greater<ii>
```

```
    > pq; pq.push(ii(0, s));                        // ^to sort the pairs by increasing distance from s
  while (!pq.empty()) {                              // main loop
    ii front = pq.top(); pq.pop();     // greedy: pick shortest unvisited vertex
    int d = front.first, u = front.second;
    if (d > dist[u]) continue;    // this check is important, see the explanation
    for (int j = 0; j < (int)AdjList[u].size(); j++) {
      ii v = AdjList[u][j];                          // all outgoing edges from u
      if (dist[u] + v.second < dist[v.first]) {
        dist[v.first] = dist[u] + v.second;          // relax operation
        pq.push(ii(dist[v.first], v.first));
  } } }  // note: this variant can cause duplicate items in the priority queue

  for (int i = 0; i < V; i++) // index + 1 for final answer
    printf("SSSP(%d, %d) = %d\n", s, i, dist[i]);
```

7 Shortcuts

7.1 Template CPP

```
typedef vector <int> vi;
typedef pair <int,int> pii;
typedef long long ll;
typedef unsigned long long ull;
#define FOR(i,a,b) for(int i(a);i<(b);i++)
#define REP(i,n) FOR(i,0,n)
#define SORT(v) sort((v).begin(),(v).end())
#define pb push_back
#define MOD 1000000007
```