competitive programming notes

```
#include <iomanip>        #include <sstream>      #include<unordered_map>
#include <iostream>       #include <algorithm>    #include <cmath>
#include <fstream>        #include <stdio.h>      #include <queue>
#include <math.h>         #include <bitset>       #include <map>
#include <string>         #include <cstring>      #include <bits/stdc++.h>
```

Bit manipulations

```
turn on jth item -> S |= (1<<j)
check if jth item is on -> S & (1<<j)
turn off the jth item -> S &= ~ (1 << j)
flip status of jth item -> S ^= (1 << j)
least significant bit -> S&-S
turn on all bits in a set of size n -> (1<<n)-1
```

Iterative Complete Search (bit manipulation) (all possible sums in an array)

```
for(int i=0;i<(1<<n);i++){
      sum=0;
      for(int j=0;j<n;j++)
         if( i&(1<<j))
           sum += array[j]
      if(sum==TARGET)break;}
```

Union Find

```
class UnionFind {// OOP style
private: vi p, rank; // remember: vi is vector<int>
public:
      UnionFind(int 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)) { // if from different set
         int x = findSet(i), y = findSet(j);
          if (rank[x] > rank[y]) p[y] = x; // rank keeps the tree short
          else { p[x] = y;
          if (rank[x] == rank[y]) rank[y]++; }
} } };
```

Minimum Spanning Tree (Kruskal's) pseudocode

```
vector< pair< weight, pair<v1,v2>>> EdgeList;
sort them
Declare an unions disjoint set
loop through all edges
    if not is the same set
      union them
      add the cost
```

Fenwick Tree (Range query) O(logn)

```
class FenwickTree {
private: vi ft; // recall that vi is: typedef vector<int> vi;
public: FenwickTree(int n) { ft.assign(n + 1, 0); } // init n + 1 zeroes
      int rsq(int b) { // returns RSQ(1, b)
         int sum = 0; for (; b; b -= LSOne(b)) sum += ft[b];
         return sum; } // note: LSOne(S) (S & (-S))
      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; }
};
```

Geometry

```
pi = acos(-1)
```

Overload the struct of points to sort

```
bool operator < (point other) const{
 if(fabs(x-other.x)>EPS)       // test equality
   return x < other.x;
 else return y<other.y
}
```

Rotating a point by angle theta counter clockwise around (0,0) by rotation matrix:

```
point rotate(point p, double theta){
        double rad = DEG_to_RAD(theta) // multiply theta with PI/180
        return point( p.x * cos(rad) - p.y * sin(rad)
                        p.x * sin(rad) + p.y * sin(cos));
}
```

Finding Convex Hull of a set of points:

```
vector<Point> convex_hull(vector<Point> A)
{   int n = A.size(), k = 0;
    if (n <= 3) return A;
    vector<Point> ans(2 * n);
    sort(A.begin(), A.end());  //sort points
   // Build lower hull
    for (int i = 0; i < n; ++i) {s
        // If the point at K-1 position is not a parta
        // of hull as vector from ans[k-2] to ans[k-1]
        // and ans[k-2] to A[i] has a clockwise turn
        while (k >= 2 && cross_product(ans[k - 2],ans[k - 1], A[i]) <= 0)
            k--;
        ans[k++] = A[i];
    }
    // Build upper hull
    for (size_t i = n - 1, t = k + 1; i > 0; --i) {
        // If the point at K-1 position is not a part
        // of hull as vector from ans[k-2] to ans[k-1
        // and ans[k-2] to A[i] has a clockwise turn
        while (k >= t && cross_product(ans[k - 2],
                            ans[k - 1], A[i - 1]) <= 0)
            k--;
        ans[k++] = A[i - 1];
    }
    // Resize the array to desired size
    ans.resize(k - 1);
    return ans;}
```

Lines

1. *Representing a line using structs.*

```
struct line {double a, b, c};    // way to represent a line (ax+by+c=0)
```

2. *We can compute the line if we are given at least 2 points.*

```
// 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;
} }
```

3. *Check whether the lines are parallel by checking their coefficients a and b are the same.*

```
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); }
```

4. If two lines are not parallel they will intersect at a point (x,y). Solution of two equations of lines.

```
// 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; }
```

5. **Line Segment** is a line with two end points with finite length.

6. Compute the angle aob between three points a, o and b using **dot product**.

Since $oa \cdot ob = |oa| \times |ob| \times \cos(\theta)$, we have $theta = \arccos(oa \cdot ob / (|oa| \times |ob|))$.

```
// dot product
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 = toVector(o, a), ob = toVector(o, b);
  return acos(dot(oa, ob) / sqrt(norm_sq(oa) * norm_sq(ob))); }
```

7. **Vector** is a line segment with direction.

```
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); }
```

8. *Minimum distance between a point **p** and a line **l***. Line **l** is given by points **a** and **b**.

Point **c** is the point in the line closest to point **p.** We can view point **c** as point **a** translated by a scaled magnitude **u** of vector ab, or c = a+u x ab. To get **u** we do scalar projection of vector ap onto vector ab by using the dot product.

```
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); }
```

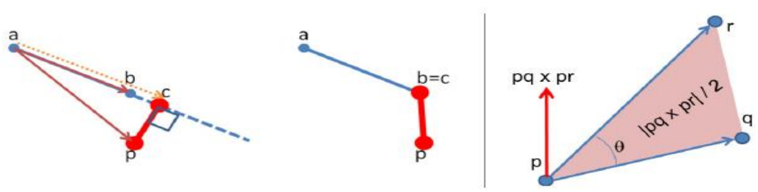

Figure 1 Distance form line(left), segment(middle) and the cross product(right).

9. Minimum distance between a segment and a point **p**.

```
// the closest point is stored in the 4th parameter (by reference)
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
```

10. If we have 3 points p,r and q, and form 2 vectors. Let CP be their cross product. The magnitude of these vectors is the are of parallelogram. Magnitude +/0/- → right turn/same line/ left turn

Triangles Formulas

- Heron's Formula: (a,b,c)-sides and s =(a+b+c)/2 then area = `A=sqrt(s*(s-a)*(s-b)*(s-c))`
- A triangle with area A and semi-perimeter s has an **inscribed circle** with radius r=A/s
- A triangle with area A and sides a,b and c has an **circumscribed circle** with radius R=a*b*c/(4*A)
- The center of incircle is the meeting point between the triangle's angle bisectors.

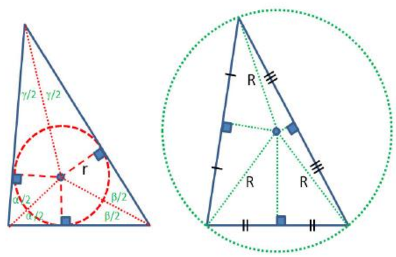

Figure 3 Inscribed and circumscribed circle

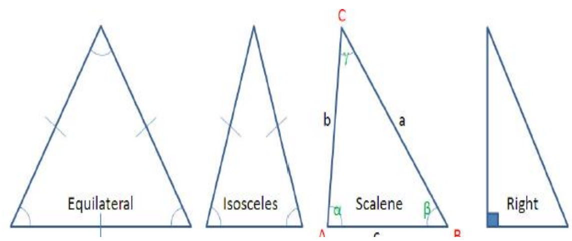

Figure 2 Triangles

- Law of cosines relates the lengths of sides of a triangle with the cosine of one if its angles. See the scalene in Figure 3. We have <u>$c^2 = a^2 + b^2 - 2ab * \cos(y)$</u>
- Law of sines relates the sides of a triangle with the sines of its angles $\frac{a}{\sin \alpha} = \frac{b}{\sin \beta} = \frac{c}{\sin y} = 2R$

Area of Polygon

$$A = \frac{1}{2} \times \begin{bmatrix} x_0 & y_0 \\ x_1 & y_1 \\ x_2 & y_2 \\ \dots & \dots \\ x_{n-1} & y_{n-1} \end{bmatrix} = \frac{1}{2} \times (x_0 \times y_1 + x_1 \times y_2 + \dots + x_{n-1} \times y_0 - x_1 \times y_0 - x_2 \times y_1 - \dots - x_0 \times y_{n-1})$$

```
// 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; }
```

Shortest paths

All pairs shortest paths (Floyd Marshall)

```
// precondition: AdjMat[i][j] contains the weight of edge (i, j)
for (int k = 0; k < V; k++) // remember that loop order is k->i->j
  for (int i = 0; i < V; i++)
    for (int j = 0; j < V; j++)
      AdjMat[i][j] = min(AdjMat[i][j], AdjMat[i][k] + AdjMat[k][j]);
```

Printing the shortest path using four liner Floyd Marshall:

```
// let p be a 2D parent matrix, where p[i][j] is the last vertex before j
// on a shortest path from i to j, i.e. i -> ... -> p[i][j] -> j
for (int i = 0; i < V; i++)
  for (int j = 0; j < V; j++)
    p[i][j] = i; // initialize the parent matrix
for (int k = 0; k < V; k++)
  for (int i = 0; i < V; i++)
    for (int j = 0; j < V; j++) // this time, we need to use if statement
      if (AdjMat[i][k] + AdjMat[k][j] < AdjMat[i][j]) {
        AdjMat[i][j] = AdjMat[i][k] + AdjMat[k][j];
              p[i][j] = p[k][j]; // update the parent matrix
      }
// when we need to print the shortest paths, we can call the method below:
void printPath(int i, int j) {
if (i != j) printPath(i, p[i][j]);
cout<<j;
```

Shortest path shortest algorithm SPFA (remove Bellman Ford's redundant operations)

```
// initially, only S has dist = 0 and in the queue
vi dist(n, INF); dist[S] = 0;
queue<int> q; q.push(S);
vi in_queue(n, 0); in_queue[S] = 1;
while (!q.empty()) {
   int u = q.front(); q.pop(); in_queue[u] = 0;
   for (j = 0; j < (int)AdjList[u].size(); j++) { // all neighbors of u
     int v = AdjList[u][j].first, weight_u_v = AdjList[u][j].second;
     if (dist[u] + weight_u_v < dist[v]) { // if can relax
       dist[v] = dist[u] + weight_u_v; // relax
       if (!in_queue[v]) { // add to the queue
          q.push(v); in_queue[v]=1;// only if it is not already in the queue
```

```
}}}}
```

Dijkstra’s Algorithm (doesn’t work for negative cycle)

```
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));
while (!pq.empty()) { // main loop
  ii front = pq.top(); pq.pop(); // greedy: get shortest unvisited vertex
  int d = front.first, u = front.second;
  if (d > dist[u]) continue; // this is a very important check
  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));
} } } // this variant can cause duplicate items in the priority queue
```

Bipartite Graph Check (using BFS)

```
queue<int> q; q.push(s);
vi color(V, INF); color[s] = 0;
bool isBipartite = true; // addition of one boolean flag, initially true
while (!q.empty() & isBipartite) { // similar to the original BFS routine
  int u = q.front(); q.pop();
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j];
    if (color[v.first] == INF) { // but, instead of recording distance,
     color[v.first] = 1 - color[u]; // we just record two colors {0, 1}
     q.push(v.first); }
    else if (color[v.first] == color[u]) { // u & v.first has same color
     isBipartite = false; break; } } } // we have a coloring
```

Binary Search (lower_bound and upper_bound)

```
vector<int> a = {1,1,1,1,2,3,4,5};
vector<int>::iterator low,up;
low = lower_bound(a.begin(),a.end(),2);
up = upper_bound(a.begin(),a.end(),2);
cout<< up-a.begin(); // output: 5
cout<< low-a.begin(); // output: 4
binary_search(a.begin(),a.end(),3)) // return a boolean TRUE value
```

Prime factors

```
void primeFactors(int n)
{   while (n%2 == 0) { cout<<2<<" ";n = n/2;}
    // n must be odd at this point.  So we can skip by 2
    for (int i = 3; i <= sqrt(n); i = i+2)
    {   // While i divides n, print i and divide n
         while (n%i == 0) { printf("%d ", i); n = n/i;}
    }
    if (n > 2) printf ("%d ", n); // if n is prime and bigger than 2

}
```

Sieve of Eratosthenes

```
int n;
vector<char> is_prime(n+1, true);
is_prime[0] = is_prime[1] = false;
for (int i = 2; i * i <= n; i++) {
    if (is_prime[i]) {
        for (int j = i * i; j <= n; j += i)
            is_prime[j] = false;
    }
} // check the index in is_prime whether is prime or not
```

Extended Euclid (Solving Diophantine Linear Equations)

```
int gcdExtended(int a, int b)   // x,y and d(gcd(a,b)) are global variables
// finds coefficients x and y such that ax+by=gcd(a,b)
{
    If (b==0) { x=1;y=0;d=a;return;} // base case
    gcdExtended(b,a%b);
    int x1=y; int y1 = x - (a/b) * y;
    x=x1; y = y1;
}
```

EXAMPLE

Find the Linear Diophantine Equation 25x + 18y = 839.

gcdExtended(25,18) produces x=-5, y=7, d=1, or **25 x (-5) + 18 x 7 = 1**

Multiply the left and right hand side of the equation above with 839/gcd(25,18) = 839

25 x (-4195) + 18 x 5873 = 839

Thus, **x= -4195 +(18/1)n** and **y=5873-(25/1)n**

If we need positive values for x and y then: $4195/18 \le n \le 5873/25$ or $233.05 \le n \le 234.92$.

Only integer for n is 234, so *x = -4195 + 18 x 234=17* and *y = 5873-25 x 234=23*

Solution: x=17, y=23