SBU Janus Code Sheet

Contents

1 Math

```
import sys
from functools import lru_cache

sys.setrecursionlimit(100000)

#############################
# SOME COMMON MATH FORMULAS #
#############################

@lru_cache(maxsize=None)
def C(n, k):
    if k == 0 or k == n:
        return 1
    else:
        return C(n-1, k-1) + C(n-1,k)

# Number of distinct binary trees with n vertices
# Number of expressions of n pairs of correctly matched paren
# Number of ways n+1 factors can be parenthesized
# Number of ways a convex polygon of n+2 sides can be triangulated
# Number of monotonic paths along nxn grid which do not pass above diagonal
@lru_cache(maxsize=None)
def cat(n):
    if n == 0:
        return 1
    else:
        return (4*n-2)*Cat(n-1)//(n+1)

# number of permutations such that none appear in original position
@lru_cache(maxsize=None)
def der(n):
    if n == 0:
        return 1
    elif n == 1:
        return 0
    else:
        return (n-1) * (der(n-1) + der(n-2))

# number of permutations of n elements with k disjoint cycles
@lru_cache(maxsize=None)
def stirl1(n, k):
    if n == 0 and k == 0:
        return 1
    elif n == 0 or k == 0:
        return 0
    else:
        return (n-1)*stirl1(n-1, k) + stirl1(n-1, k-1)

# number of ways to partition a set of n objects into k non-empty subsets
@lru_cache(maxsize=None)
def stirl2(n, k):
    if n == 0 and k == 0:
        return 1
    elif n == 0 or k == 0:
        return 0
    else:
        return k*stirl2(n-1, k) + stirl2(n-1, k-1)
```

```python
# number of permutations of numbers 1 to n in which exactly m elements are greater
# than the previous element (permutations with m "ascents")
@lru_cache(maxsize=None)
def euler1(n, m):
    if m == 0 or m == n-1:
        return 1
    else:
        return (n-m)*euler1(n-1,m-1) + (m+1)*euler1(n-1,m)

# The permutations of the multiset {1, 1, 2, 2, ..., n, n} which have the
# property that for each k, all the numbers appearing between the two
# occurrences of k in the permutation are greater than k are counted by the
# double factorial number (2n-1)!! The Eulerian number of the second kind
# counts the number of all such permutations that have exactly m ascents.
@lru_cache(maxsize=None)
def euler2(n, m):
    if n == 0 and m == 0:
        return 1
    elif n == 0:
        return 0
    else:
        return (2*n-m-1)*euler2(n-1, m-1) + (m+1)*euler2(n-1, m)

# Fibonacci Facts
# [[1,1],[1,0]] mat mul for O(log n)
# Zeckendorf's theorem - greedily choose largest fibonacci to represent
# Pisano Period - last one/two/three/four digits repeat
# with period 60/300/1500/15000 respectively

# Burnside's lemma
# X = set of all possible ways to arrange
# G = set of rotations
# X/G = set of all possible ways to arrange rotationally invariant
# X/G = (1/|G|) sum_{g in G} |X^g|

# 3 colors, 4 sided table
# G = {0, 90, 180, 270}
# (1/4) * (3^4 + 3 + 9 + 3) = 92/4 = 24

# number of rotationally distinct colorings of faces of cube in n colors
# (1/24) * (n**6 + 3*n**4 + 12*n**3 + 8*n**2)

# Cayley's formula
# There are n^{n-2} spanning trees of a complete graph with n labeled vertices

# Degree sequence of graph d1 >= d2 >= ... >= dn
# sum of d_i is even
# sum_{i=1}^k d_i <= k*(k-1) + sum_{i=k+1}^n min(d_i, k)

# Euler's formula for planar graphs
# V - E + F = 2
# F is number of faces in graph

# Moser's circle - number of pieces a circle is divided if n points on
# circumference are joined by chords with no three internally concurrent
# g(n) = nC4 + nC2 + 1

# Pick's theorem
# I: number of integer points in the polygon
# A: area of polygon
```

```python
# b: number of integer points on boundary
# A = i + (b//2) - 1

# Number of spanning trees on complete bipartite graph
# K_{n, m} = m^{n-1} * n^{m-1}

# Josephus
# n people and k-th person gets killed
# when k = 2
# n = 1 b_1 b_2 b_3..b_n, answer is b_1 b_2 b_3...b_n 1
# move most significant bit to the back
# people labeled from 0 to n-1
# otherwise, F(n, k) = (F(n-1, k) + k) % n
# F(1, k) = 0

#################
# Cycle finding #
#################

def floyd_cycle(f, x0):
    tortoise = f(x0)
    hare = f(tortoise)
    while tortoise != hare:
        tortoise = f(tortoise)
        hare = f(f(hare))
    # start of cycle
    mu = 0
    hare = x0
    while tortoise != hare:
        tortoise = f(tortoise)
        hare = f(hare)
        mu += 1
    # len of cycle
    l = 1
    hare = f(tortoise)
    while tortoise != hare:
        hare = f(hare)
        l += 1
    return (mu, l)


##################################
# COMMON NUMBER THEORY FUNCTIONS #
##################################
sieve_size = 10000001
bs = [True] * 10000010
primes = []


def sieve():
    bs[0] = bs[1] = False
    for i in range(2, sieve_size+1):
        if bs[i]:
            for j in range(i*i, sieve_size+1, i):
                bs[j] = False
        primes.append(i)

def is_prime(N):
    if N <= sieve_size:
        return bs[N]
```

```
    for i in range(len(primes)):
        if N % primes[i] == 0:
            return False
    return True

# Takes a few seconds
def test_prime():
    sieve()
    print(is_prime(2147483647))
    print(is_prime(136117223861))

def gcd(a, b):
    if b == 0:
        return a
    else:
        return gcd(b, a % b)

def lcm(a, b):
    return a * (b // gcd(a, b))

# Make sure sieve is called before using
def prime_factors(N):
    factors = []
    PF_idx = 0
    PF = primes[PF_idx]
    while PF*PF <= N:
        while N % PF == 0:
            N //= PF
            factors.append(PF)
        PF_idx += 1
        PF = primes[PF_idx]
    # special case when N is prime
    if N != 1:
        factors.append(N)
    return factors

def test_prime_factors():
    sieve()
    r = prime_factors(2147483647)
    print(r)
    r = prime_factors(136117223861)
    print(r)
    r = prime_factors(142391208960)
    print(r)

def num_pf(N):
    PF_idx = 0
    PF = primes[PF_idx]
    ans = 0
    while PF * PF <= N:
        # check if divisible and incr (for num_diff_pf)
        # also, don't increment inside loop
        while N % PF == 0:
            N //= PF
            ans += 1
            # ans += PF (for sum_pf)
        PF_idx += 1
        PF = primes[PF_idx]
    # special case when N is prime
    if N != 1:
```

```
        ans += 1
    return ans

# num_diff_pf and num_sum_pf is trivial change to above
# num_diff_pf for many values can be done through sieve
def num_diff_pf_sieve(N):
    numDiffPF = [0] * N
    for i in range(2, N):
        if numDiffPF[i] == 0:
            for j in range(i, N, i):
                numDiffPF[j] += 1
    return numDiffPF

def num_div(N):
    PF_idx = 0
    PF = primes[PF_idx]
    ans = 1
    while PF * PF <= N:
        power = 0
        while N % PF == 0:
            N //= PF
            power += 1
        ans *= (power + 1)
        PF_idx += 1
        PF = primes[PF_idx]
    if N != 1:
        ans *= 2
    return ans

def sum_div(N):
    PF_idx = 0
    PF = primes[PF_idx]
    ans = 1
    while PF * PF <= N:
        power = 0
        while N % PF == 0:
            N //= PF
            power += 1
        ans *= (PF**(power+1) - 1) // (PF - 1)
        PF_idx += 1
        PF = primes[PF_idx]
    if N != 1:
        ans *= (N*N-1) // (N - 1)
    return ans

def euler_phi(N):
    PF_idx = 0
    PF = primes[PF_idx]
    ans = N
    while PF * PF <= N:
        if N % PF == 0:
            ans -= ans // PF
        while N % PF == 0:
            N /= PF
        PF_idx += 1
        PF = primes[PF_idx]
    if N != 1:
        ans -= ans // N
    return ans
```

```python
def test_prime_factors():
    sieve()
    print(num_pf(60))
    print(num_div(60))
    print(num_diff_pf_sieve(100))
    print(sum_div(60))
    print(euler_phi(36))

# ax + by = c
# d = gcd(a, b)
# if d | c is not true, no solution

# return (x0, y0, d)
# a*x0 + b*y0 = d
def extended_euclid(a, b):
    if b == 0:
        return (1, 0, a)
    else:
        x, y, d = extended_euclid(b, a % b)
        return (y, x - (a // b) * y, d)

# the prime means we have to multiply by c//d to get a solution
# x = x0' + (b//d)*n
# y = y0' - (a//d)*n

########################
# GAUSSIAN ELIMINATION #
########################

# aug is the augmented matrix with size n x n+1
def gauss_elim(aug):
    N = len(aug)
    X = [0]*N
    for j in range(N-1):
        l = j
        for i in range(j+1, N):
            if abs(aug[i][j]) > abs(aug[l][j]):
                l = i
        for k in range(j, N+1):
            t = aug[j][k]
            aug[j][k] = aug[l][k]
            aug[l][k] = t
        for i in range(j+1, N):
            for k in range(N, j-1, -1):
                aug[i][k] -= aug[j][k] * aug[i][j] / aug[j][j]
    for j in range(N-1, -1, -1):
        t = 0
        for k in range(j+1, N):
            t += aug[j][k] * X[k]
        X[j] = (aug[j][N] - t) / aug[j][j]
    return X

# aug = [[1,1,2,9],[2,4,-3,1],[3,6,-5,0]]
# X = [1,2,3]
# print(gauss_elim(aug))

##################
# ROMAN NUMERALS #
##################
```

```python
def AtoR(A):
    cvt = {}
    cvt[1000] = "M"
    cvt[900] = "CM"
    cvt[500] = "D"
    cvt[400] = "CD"
    cvt[100] = "C"
    cvt[90] = "XC"
    cvt[50] = "L"
    cvt[40] = "XL"
    cvt[10] = "X"
    cvt[9] = "IX"
    cvt[5] = "V"
    cvt[4] = "IV"
    cvt[1] = "I"
    s = ""
    keys = sorted(cvt.keys(),reverse=True)
    for i in keys:
        while A >= i:
            s += cvt[i]
            A -= i
    return s

def RtoA(R):
    cvt = {}
    cvt['I'] = 1
    cvt['V'] = 5
    cvt['X'] = 10
    cvt['L'] = 50
    cvt['C'] = 100
    cvt['D'] = 500
    cvt['M'] = 1000

    value = 0
    i = 0
    while i < len(R):
        if i+1 < len(R) and cvt[R[i]] < cvt[R[i+1]]:
            value += cvt[R[i+1]] - cvt[R[i]]
            i += 1
        else:
            value += cvt[R[i]]
        i += 1
    return value
```

2 Geometry

```python
import math
import numbers

def cmp_to_key(mycmp):
    'Convert a cmp= function into a key= function'
    class K:
        def __init__(self, obj, *args):
            self.obj = obj
        def __lt__(self, other):
            return mycmp(self.obj, other.obj) < 0
        def __gt__(self, other):
            return mycmp(self.obj, other.obj) > 0
        def __eq__(self, other):
            return mycmp(self.obj, other.obj) == 0
        def __le__(self, other):
            return mycmp(self.obj, other.obj) <= 0
        def __ge__(self, other):
            return mycmp(self.obj, other.obj) >= 0
        def __ne__(self, other):
            return mycmp(self.obj, other.obj) != 0
    return K

EPS=1e-6
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __lt__(self, other):
        if abs(self.x - other.x) > EPS:
            return self.x < other.x

        return self.y < other.y

    def __eq__(self, other):
        return abs(self.x - other.x) < EPS and abs(self.y - other.y) < EPS

    def set(self, other):
        self.x = other.x
        self.y = other.y @staticmethod
    def distance(p1, p2):
        return math.sqrt((p1.x - p2.x)**2 + (p1.y - p2.y)**2)

    @staticmethod
    def rotate(p, theta): # theta is in radians, rotate w.r.t origin
        return Point(p.x * math.cos(theta)  - p.y * math.sin(theta),
                p.x * math.sin(theta) + p.y * math.cos(theta))

    def __str__(self):
        return "Point({}, {})".format(self.x, self.y)

    def __repr__(self):
        return str(self)

class Line:
    def __init__(self, a, b, c):
        self.a = a
        self.b = b
```

```python
        self.c = c

    def __str__(self):
        return "Line({}, {}, {})".format(self.a, self.b, self.c)

    def __repr__(self):
        return str(self)

    @staticmethod
    def from_points(p1, p2):
        if abs(p1.x - p2.x) < EPS:
            return Line(1, 0, -p1.x)
        else:
            a = -(p1.y - p2.y)/(p1.x - p2.x)
            b = 1
            c = -(a * p1.x) - p1.y
            return Line(a, b, c)

    @staticmethod
    def are_parallel(l1, l2):
        return abs(l1.a - l2.a) < EPS and abs(l1.b - l2.b) < EPS

    def __eq__(self, other):
        return Line.are_parallel(self, other) and self.c == other.c

    @staticmethod
    def intersection(l1, l2):
        if (Line.are_parallel(l1, l2)):
            return None

        x = (l2.b * l1.c - l1.b * l2.c) / (l2.a * l1.b - l1.a * l2.b)
        y = -(l1.a * x + l1.c) if (abs(l1.b) > EPS) else (l2.a * x + l2.c)

        return Point(x, y)

class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return "Vector({}, {})".format(self.x, self.y)

    def __repr__(self):
        return str(self)

    @staticmethod
    def from_points(a, b): # vector from a-> b
        return Vector(b.x - a.x, b.y - a.y)

    def __mul__(self, k):
        if isinstance(k, numbers.Number):
            return Vector.scale(self, k)

        return Vector.dot_product(self, k)

    def __mod__(self, v): # cross product: % kind of looks like an 'x' if you squint hard enough
        return Vector.cross_product(self, v)
```

```
    def __add__(self, v):
        return Vector(self.x + v.x, self.y + v.y)

    def magnitude(self):
        return math.sqrt(self * self)

    @staticmethod
    def scale(v, k):
        return Vector(v.x * k, v.y * k)

    @staticmethod
    def dot_product(v1, v2):
        return v1.x * v2.x + v1.y * v2.y

    @staticmethod
    def cross_product(a, b):
        return a.x * b.y - a.y * b.x

    @staticmethod
    def translate_point(p, v):
        return Point(p.x + v.x, p.y + v.y)


def distance_to_line(p, a, b, c=Point(0, 0)): # distance. closest point returned by ref in c
    ap = Vector.from_points(a, p)
    ab = Vector.from_points(a, b)
    u = (ap * ab)/(ab * ab)
    c.set(Vector.translate_point(a, ab * u))
    return Point.distance(p, c), c

def distance_to_line_segment(p, a, b, c=Point(0, 0)):
    ap = Vector.from_points(a, p)
    ab = Vector.from_points(a, b)
    u = (ap * ab) / (ab * ab)
    if u < 0:
        c.set(a)
        return Point.distance(p, a)
    elif u > 1:
        c.set(b)
        return Point.distance(p, b)

    return distance_to_line(p, a, b, c)

def angle(a, o, b): # returns angle aob in radians
    oa = Vector.from_points(o, a)
    ob = Vector.from_points(o, b)

    if (oa * oa) == 0 or (ob * ob) == 0:
        raise ValueError("Duplicate point in ({}, {}, {})".format(a, o, b))

    return math.acos((oa * ob) / math.sqrt((oa * oa) * (ob * ob)))

def is_counter_clockwise(p, q, r): # returns true if r is on the left side of the line pq
    return Vector.from_points(p, q) % Vector.from_points(p, r) > 0

def is_collinear(p, q, r): # returns true if r is on the same line as the line pq
    return abs(Vector.from_points(p, q) % Vector.from_points(p, r)) < EPS

def inside_circle(p, c, r): # returns 0 if inside circle, 1 if on border, 2 if outside
    dx = p.x - c.x
```

```python
    dy = p.y - c.y
    euc = dx**2 + dy**2
    rs = r**2
    return 0 if euc < rs else (1 if euc == rs else 2)

# returns the centers of the circle that goes through p1 and p2 with radius r
def circle_from_points(p1, p2, r):
    def center_finder(p1, p2, r):
        d2 = (p1.x - p2.x)**2 + (p1.y - p2.y)**2
        det = r**2/d2 - 0.25
        if det < 0:
            return None

        h = math.sqrt(det)
        return Point((p1.x + p2.x) * 0.5 + (p1.y - p2.y) * h,
                     (p1.y + p2.y) * 0.5 + (p2.x - p1.x) * h)

    return center_finder(p1, p2, r), center_finder(p2, p1, r)

class Triangle:
    @staticmethod
    def perimeter(ab, bc, ca):
        return ab + bc + ca

    @staticmethod
    def area(ab, bc, ca):
        s = .5 * Triangle.perimeter(ab, bc, ca)
        return math.sqrt(s * (s - ab) * (s - bc) * (s - ca))

    @staticmethod
    def radius_inscribed_circle(ab, bc, ca):  # returns the rabdius of the inscaribced caircale
        return Triangle.area(ab, bc, ca) / (0.5 * Triangle.perimeter(ab, bc, ca))

    @staticmethod
    def get_inscribed_circle(p1, p2, p3):
        ab = Point.distance(p1, p2)
        bc = Point.distance(p2, p3)
        ca = Point.distance(p3, p1)
        r = Triangle.radius_inscribed_circle(ab, bc, ca)
        if abs(r) < EPS:
            return None

        ratio = Point.distance(p1, p2) / Point.distance(p1, p3)
        p = Vector.translate_point(p2, Vector.from_points(p2, p3) * (ratio / (1 + ratio)))
        l1 = Line.from_points(p1, p)

        ratio = Point.distance(p2, p1) / Point.distance(p2, p3)
        p = Vector.translate_point(p1, Vector.from_points(p1, p3) * (ratio / (1 + ratio)))
        l2 = Line.from_points(p2, p)

        c = Line.intersection(l1, l2)
        return c, r

    @staticmethod
    def radius_circumscribed_circle(ab, bc, ca):
        return (ab * bc * ca)/ (4 * Triangle.area(ab, bc, ca))

# center point of circumscribed circle is meeting point of triangle's perpendicular bisectors
# perpendicular bisectors cut the angle of a triangle in such a way that any line drawn from
# the bisector to the side is pi/2 radians
```

```python
class Polygon:
    # p is a list of points in clockwise or counterclockwise order
    # p[0] must be equal to p[-1]
    @staticmethod
    def perimeter(p):
        r = 0
        for i in range(len(p) - 1):
            r += Point.distance(p[i], p[i + 1])

        return r

    # area is half the determinant of the matrix:
    # [[x0, y0], [x1, y1], [x2, y2]...[xn, yn]]
    # p is a list of points in clockwise or counterclockwise order
    # p[0] must be equal to p[-1]
    @staticmethod
    def area(p):
        r = 0
        for i in range(len(p) - 1):
            x1, x2 = p[i].x, p[i + 1].x
            y1, y2 = p[i].y, p[i + 1].y
            r += (x1 * y2 - x2 * y1)

        return abs(r) / 2

    @staticmethod
    def is_convex(p):
        sz = len(p)
        if sz <= 3:
            return False

        is_left = is_counter_clockwise(p[0], p[1], p[2])
        for i in range(1, sz - 1):
            if is_counter_clockwise(p[i], p[i + 1], p[1 if i + 2 == sz else i + 2]) != is_left:
                return False

        return True

    # checks if point pt is in the polygon p
    @staticmethod
    def in_polygon(pt, p):
        if len(p) == 0:
            return False

        s = 0
        for i in range(len(p) - 1):
            if is_counter_clockwise(pt, p[i], p[i + 1]):
                s += angle(p[i], pt, p[i + 1])
            else:
                s -= angle(p[i], pt, p[i + 1])

        return abs(abs(s) - 2*math.pi) < EPS

    # cut polygon q using line segment a-b
    @staticmethod
    def cut_polygon(a, b, Q):
        # line segment p-q intersects with lien A-B
        def line_intersect_segment(p, q, A, B):
            a = B.y - A.y
```

```
        b = A.x - B.x
        c = B.x * A.y - A.x * B.y
        u = abs(a * p.x + b * p.y + c)
        v = abs(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))
    P = []
    for i in range(len(Q)):
        left1 = Vector.from_points(a, b) % Vector.from_points(a, Q[i])
        left2 = 0
        if i != len(Q) - 1:
            left2 = Vector.from_points(a, b) % Vector.from_points(a, Q[i + 1])
        if left1 > -EPS:
            P.append(Q[i]) # Q[i] is on the left of ab
        if left1 * left2 < -EPS:
            P.append(line_intersect_segment(Q[i], Q[i + 1], a, b))

    if len(P) != 0 and not (P[0] == P[-1]):
        P.append(P[0])

    return P

@staticmethod
def convex_hull(P):
    def angle_cmp_generator(pivot):
        def angle_cmp(a, b):
            if is_collinear(pivot, a, b):
                da = Point.distance(pivot, a)
                db = Point.distance(pivot, b)
                return -1 if da < db else (0 if da == db else 1)

            d1x, d1y = a.x - pivot.x, a.y - pivot.y
            d2x, d2y = b.x - pivot.x, b.y - pivot.y

            theta1 = math.atan2(d1y, d1x)
            theta2 = math.atan2(d2y, d2x)
            return -1 if theta1 < theta2 else (0 if theta1 == theta2 else 1)

        return angle_cmp

    n = len(P)
    if n <= 3:
        if n == 1:
            P.append(P[0])
        elif not (P[0] == P[-1]):
            P = sorted(P, key=cmp_to_key(angle_cmp_generator(P[0])))
            P.append(P[0])
        else:
            P.pop()
            P = sorted(P, key=cmp_to_key(angle_cmp_generator(P[0])))
            P.append(P[0])

        return P

    P0 = 0
    for i in range(n):
        if P[i].y < P[P0].y or (P[i].y == P[P0].y and P[i].x > P[P0].x):
            P0 = i

    P[0], P[P0] = P[P0], P[0]
    P = sorted(P, key=cmp_to_key(angle_cmp_generator(P[0])))
```

```
        S = []
        S.append(P[-1])
        S.append(P[0])
        S.append(P[1])
        i = 2
        while i < n:
            j = len(S) - 1
            if is_counter_clockwise(S[j - 1], S[j], P[i]):
                S.append(P[i])
                i += 1
            else:
                S.pop()

        return S


# Art Gallery Problem
# Given:
# polygon P to describe the art gallery
# a set of pointsS to describe the guards where each guard is represented by a point in P
# a rule that a point A in S can guard another point B in P iff (A in S), (B in P), and
# a line segment AB is contained in P
# a question whether all points in P are guarded by S
# 4 Variants:
# 1) Determine the upper bound of the smallest size of set S
# 2) Determine if there exists a critical point C in polygon P and there exists another point D
# in P such that if the guard is not at position C, the guard cannot protect point D
# 3) Determine if the polygon P can be guarded with just one guard
# 4) Determine the smallest size of set S if the guards can only be placed at the vertices
# of polygon P and only the verticies need to be guarded
# Solutions:
# 1) floor(n/3) are always sufficient and sometimes necessary to guard a simple polygon with n vertices
# 2) The solution for variant 2 involves testing if polygon P is concave (and thus has a critical point).
# We can use the negation of isConvex
# The solution for variant 3 can be hard if one has not seen the solution before. We can use the
# cutPolygon function. We cut polygon P with all lines formed by the edges in P in counterclockwise
# fashion and retain the left side at all times. If we simply have a non-empty polygon at the end
# one guard can be placed in the non-empty polygon which can protect the entire polygon P
# The solution for variant 4 involves the computation of the minimum vertex cover of of the
# "visibility" graph of polygon P. In general this is another NP-hard problem

# Great Circle
# The Great Circle Distance between any two points is the shortest distance along a path on the
# surface of the sphere. This path is an arc of the Great-Circle of that sphere that pass
# through the two points. The Great circle cuts the sphere into two equal hemispheres
# to find the great-circle distance, we find the central angle AOB
# of the great-circle where O is the center of the great-circle. We can then determine
# the length of the arc A-B, which is the required Great-Circle distance
def great_circle_distance(plat, plong, qlat, qlong, radius):
    plat *= math.pi / 180
    plong *= math.pi/ 180
    qlat *= math.pi / 180
    qlong *= math.pi / 180
    acos = math.acos
    cos = math.cos
    sin = math.sin
    return radius * acos(cos(plat)*cos(plong)*cos(qlat)*cos(qlong) +
            cos(plat)*sin(plong)*cos(qlat)*sin(qlong)+
            sin(plat)*sin(qlat))
```

3 BFS

```
from queue import *

nvert, nedge, goal = [int(x) for x in input().split()]

nodes = {}
for i in range(nvert):
	nodes[i] = []

for _ in range(nedge):
	fr, to, w= [int(x) for x in input().split()]
	nodes[fr].append([to, w])
	nodes[to].append([fr, w])

visited = {}
for i in range(nvert):
	visited[i] = False


def bfs(start):
	q = Queue()
	q.put(start)
	while (not q.empty()):
		vertex = q.get()
		visited[vertex] = True
		if (visited[goal]):
			return
		nodes[vertex].sort(key = lambda x: x[1])
		for to in nodes[vertex]:
			if visited[to[0]]:
				continue
			q.put(to[0])

bfs(0)
print(visited[goal])
```

4 DFS

```
nvert, nedge, goal = [int(x) for x in input().split()]

nodes = {}
for i in range(nvert):
        nodes[i] = []

for _ in range(nedge):
        fr, to, w= [int(x) for x in input().split()]
        nodes[fr].append([to, w])
        nodes[to].append([fr, w])

visited = {}
for i in range(nvert):
        visited[i] = False

def dfs(vertex):
        visited[vertex] = True
        if (visited[goal]):
                return
        nodes[vertex].sort(key = lambda x: x[1])
        for to in nodes[vertex]:
                if visited[to[0]]:
                        continue
                dfs(to[0])

dfs(0)
print(visited[goal])
```

5 Dijkstra

```python
from heapq import heappush, heappop, heapify

nvert, nedge, goal = [int(x) for x in input().split()]

nodes = {}
for i in range(nvert):
        nodes[i] = []

for _ in range(nedge):
        fr, to, w= [int(x) for x in input().split()]
        nodes[fr].append([to, w])
        nodes[to].append([fr, w])

INF = 10000000

distance = {}
prev = {}
unvisited = []

def dijk(start):

        distance[start] = 0

        for i in range(nvert):
                if (not i == start):
                         distance[i] = INF
                prev[i] = -1

                heappush(unvisited, (distance[i], i))

        while(unvisited):
                v = heappop(unvisited) #dist, node
                for n in nodes[v[1]]: #neighbor n (to, pathWeight)
                        alt = v[0] + n[1] #start to cur + cur to next
                        if (alt < distance[n[0]]): #shorter path, update
                                distance[n[0]] = alt
                                prev[n[0]] = v[1]
                                for i in range(len(unvisited)):
                                        if (unvisited[i][1] == n[0]):
                                                unvisited[i] = (alt, n[0])
                                                break
                                heapify(unvisited)
        return distance[goal]

print(dijk(0))
cur = goal
path = []
while (not prev[cur] == -1):
        path.append(cur)
        cur = prev[cur]
path.append(cur)
path.reverse()
print(path)
```

6 Network Flow

```python
from math import inf
from queue import Queue

MAX_V = 5
res = [[0]*MAX_V for _ in range(MAX_V)]
res[0][2] = 100
res[0][3] = 50
res[2][3] = 50
res[2][4] = 50
res[2][1] = 50
res[3][4] = 100
res[4][1] = 125

adj_list = [[2,3],[],[1,3,4],[4],[1]]
s = 0 # source
t = 1 # sink

mf = f = 0
p = []

def augment(v, min_edge):
    global res, f
    if v == s:
        f = min_edge
        return
    elif p[v] != -1:
        augment(p[v], min(min_edge, res[p[v]][v]))
        res[p[v]][v] -= f
        res[v][p[v]] += f

def edmond_karp():
    global res, mf, f, p
    mf = 0
    while True:
        f = 0
        vis = [False]*MAX_V
        vis[s] = True
        q = Queue()
        q.put(s)
        p = [-1]*MAX_V
        while not q.empty():
            u = q.get()
            if u == t:
                break
            for j in range(len(adj_list[u])):
                v = adj_list[u][j]
                if res[u][v] > 0 and not vis[v]:
                    vis[v] = True
                    q.put(v)
                    p[v] = u
        augment(t, inf)
        if f == 0:
            break
        mf += f
    return mf
```

7 Floyd Warshall

```
nvert, nedge, goal = [int(x) for x in input().split()]

INF = 10000000

nodes = {}
for i in range(nvert):
        nodes[i] = {}
        for j in range(nvert):
                nodes[i][j] = INF


for _ in range(nedge):
        fr, to, w = [int(x) for x in input().split()]
        nodes[fr][to] = w
        nodes[to][fr] = w

for k in range(nvert):
        for i in range(nvert):
                for j in range(nvert):
                        nodes[i][j] = min(nodes[i][j], nodes[i][k] + nodes[k][j])
print(nodes[0][goal])
```

8 Knapsack

```
n, cap = [int(x) for x in input().split()]

items = []
for i in range(n):
        items.append([int(x) for x in input().split()]) #weight, value

dp = [[0 for x in range(cap+1)] for x in range(n+1)] #dp[n][capacity]

for i in range(n+1):
        for c in range(cap+1):
                if (i == 0 or c == 0): #fill in base cases
                        dp[i][c] = 0
                elif items[i-1][0] <= c: #can carry
                        # best here is the max of
                        # value of nth item + best value with n-1 items and c less capacity
                        # or n-1 items (rejecting the item)
                        dp[i][c] = max(items[i-1][1] + dp[i-1][c-items[i-1][0]], dp[i-1][c])
                else:
                        dp[i][c] = dp[i-1][c]
print(dp[i][c])
```

9 Kruskal

```
class UFDS:
    def __init__(self, N):
        self.rank = [0] * N
        self.p = list(range(N))

    def find(self, i):
        if not self.p[i] == i:
            self.p[i] = self.find(self.p[i])
        return self.p[i]

    def union(self, a, b):
        if not self.is_same_set(a, b):
            x = self.find(a)
            y = self.find(b)
            if self.rank[x] > self.rank[y]:
                self.p[y] = x
            else:
                self.p[x] = y
                if self.rank[x] == self.rank[y]:
                    self.rank[y] += 1

    def is_same_set(self, a, b):
        return self.find(a) == self.find(b)

# V: number of vertices
# edge_list: [(w,u,v),...] (weight, start, end)
def kruskal(V, edge_list):
    edge_list.sort()
    mst_cost = 0
    UF = UFDS(V)
    for w, u, v in edge_list:
        if not UF.is_same_set(u, v):
            mst_cost += w
            UF.union(u, v)
    return mst_cost
```

10 Strongly Connected Components

```python
dfs_num = []
dfs_low = []
S = []
visited = []
dfsNumberCounter = 0
numSCC = 0
V = 12
adj_list = [
    [1],
    [2],
    [3],
    [1],
    [5],
    [6],
    [4],
    [8,11],
    [9],
    [7,10],
    [7],
    [10]
]

def tarjanSCC(u):
    global dfs_num, dfs_low, S, visited, dfsNumberCounter, numSCC
    dfs_low[u] = dfs_num[u] = dfsNumberCounter
    dfsNumberCounter += 1
    S.append(u)
    visited[u] = 1
    for j in range(len(adj_list[u])):
        v = adj_list[u][j]
        if not dfs_num[v]:
            tarjanSCC(v)
        if visited[v]:
            dfs_low[u] = min(dfs_low[u], dfs_low[v])
    if dfs_low[u] == dfs_num[u]:
        numSCC += 1
        print("SCC {}".format(numSCC))
        while True:
            v = S.pop()
            visited[v] = 0
            print(" {}".format(v))
            if u == v:
                break
        print()

dfs_num = [False] * V
dfs_low = [0] * V
visited = [0] * V
dfsNumberCounter = numSCC = 0
for i in range(V):
    if not dfs_num[i]:
        tarjanSCC(i)
```

11 SPFA

```
from queue import *
nvert, nedge, goal = [int(x) for x in input().split()]

INF = 10000000

nodes = {}
dist = {}
for i in range(nvert):
        nodes[i] = []
        dist[i] = INF

for _ in range(nedge):
        fr, to, w= [int(x) for x in input().split()]
        nodes[fr].append([to, w])
        nodes[to].append([fr, w])


def spfa(start):
        dist[start] = 0
        q = Queue()
        q.put(start)
        while (not q.empty()):
                u = q.get()
                for v in nodes[u]:
                        if (dist[u] + v[1] < dist[v[0]]): #s -> u -> v shorted than s -> v
                                dist[v[0]] = dist[u] + v[1]
                                #if v[0] not in q:
                                q.put(v[0])
spfa(0)
print(dist[goal])
```

12 Topological

```
from queue import *
n = int(input())
nodes = [[]]*n
inc = [0]*n
total = 0

nedge = int(input())
for i in range(nedge):
        to, fr = [int(x) for x in input().split()]
        inc[fr] += 1
        nodes[to].append([fr])

ans = []
q = Queue()
for i in range(n):
        if inc[i] == 0:
                q.put(i)
while(not q.empty()):
        u = q.get()
        inc[u] -= 1
        ans.append(u)
        for n in nodes[u]:
                inc[n[0]] -= 1
                if inc[n[0]] == 0:
                        q.put(n[0])
print(ans)
```