UPLB Eliens ICPC Notebook (Python 3)

Contents

1 Dynamic Programming

1.1 Max Sum Subarray (Kadane's Algorithm)

```python
def maxSubArraySum(a,size):
    max_so_far = 0
    max_ending_here = 0
    for i in range(0, size):
        max_ending_here = max_ending_here + a[i]
        if max_ending_here < 0:
            max_ending_here = 0
        elif (max_so_far < max_ending_here):
            max_so_far = max_ending_here
    return max_so_far
```

1.2 Longest Common Subsequence

```python
def lcs(X , Y):
    # find the length of the strings
    m = len(X)
    n = len(Y)

    # declaring the array for storing the dp values
    L = [[None]*(n+1) for i in xrange(m+1)]

    """Following steps build L[m+1][n+1] in bottom up
        fashion
    Note: L[i][j] contains length of LCS of X[0..i-1]
    and Y[0..j-1]"""
    for i in range(m+1):
        for j in range(n+1):
            if i == 0 or j == 0 :
                L[i][j] = 0
            elif X[i-1] == Y[j-1]:
                L[i][j] = L[i-1][j-1]+1
            else:
                L[i][j] = max(L[i-1][j] , L[i][j-1])

    # L[m][n] contains the length of LCS of X[0..n-1] &
        Y[0..m-1]
    return L[m][n]
```

1.3 Levenshtein Distance

```python
def levenshtein(s1, s2):
    if len(s1) < len(s2):
        return levenshtein(s2, s1)

    # len(s1) >= len(s2)
    if len(s2) == 0:
        return len(s1)

    previous_row = range(len(s2) + 1)
    for i, c1 in enumerate(s1):
        current_row = [i + 1]
        for j, c2 in enumerate(s2):
            insertions = previous_row[j + 1] + 1 # j+1
                instead of j since previous_row and
                current_row are one character longer
            deletions = current_row[j] + 1       # than
                s2
            substitutions = previous_row[j] + (c1 != c2)
            current_row.append(min(insertions, deletions
                , substitutions))
        previous_row = current_row
    return previous_row[-1]
```

1.4 Longest Increasing Subsequence

```
def lis(arr):
    n = len(arr)

    # Declare the list (array) for LIS and initialize
        LIS
    # values for all indexes
    lis = [1]*n

    # Compute optimized LIS values in bottom up manner
    for i in range (1 , n):
        for j in range(0 , i):
            if arr[i] > arr[j] and lis[i]< lis[j] + 1 :
                lis[i] = lis[j]+1

    # Initialize maximum to 0 to get the maximum of all
    # LIS
    maximum = 0

    # Pick maximum of all LIS values
    for i in range(n):
        maximum = max(maximum , lis[i])
    return maximum
```

2 Geometry

2.1 Convex Hull

```
def convex_hull(points):
    """Computes the convex hull of a set of 2D points.

    Input: an iterable sequence of (x, y) pairs
        representing the points.
    Output: a list of vertices of the convex hull in
        counter-clockwise order,
      starting from the vertex with the
        lexicographically smallest coordinates.
    Implements Andrew's monotone chain algorithm. O(n
        log n) complexity.
    """

    # Sort the points lexicographically (tuples are
        compared lexicographically).
    # Remove duplicates to detect the case we have just
        one unique point.
    points = sorted(set(points))

    # Boring case: no points or a single point, possibly
        repeated multiple times.
    if len(points) <= 1:
        return points

    # 2D cross product of OA and OB vectors, i.e. z-
        component of their 3D cross product.
    # Returns a positive value, if OAB makes a counter-
        clockwise turn,
    # negative for clockwise turn, and zero if the
        points are collinear.
    def cross(o, a, b):
        return (a[0] - o[0]) * (b[1] - o[1]) - (a[1] - o
            [1]) * (b[0] - o[0])

    # Build lower hull
    lower = []
    for p in points:
        while len(lower) >= 2 and cross(lower[-2], lower
            [-1], p) <= 0:
            lower.pop()
        lower.append(p)

    # Build upper hull
    upper = []
    for p in reversed(points):
        while len(upper) >= 2 and cross(upper[-2], upper
            [-1], p) <= 0:
            upper.pop()
        upper.append(p)

    # Concatenation of the lower and upper hulls gives
        the convex hull.
    # Last point of each list is omitted because it is
        repeated at the beginning of the other list.
    return lower[:-1] + upper[:-1]

# Example: convex hull of a 10-by-10 grid.
assert convex_hull([(i//10, i%10) for i in range(100)])
    == [(0, 0), (9, 0), (9, 9), (0, 9)]
```

2.2 Misc Geometry Functions (C++)

```
// C++ routines for computational geometry.

#include <iostream>
#include <vector>
#include <cmath>
#include <cassert>
```

```cpp
using namespace std;

double INF = 1e100;
double EPS = 1e-12;

struct PT {
  double x, y;
  PT() {}
  PT(double x, double y) : x(x), y(y) {}
  PT(const PT &p) : x(p.x), y(p.y)    {}
  PT operator + (const PT &p)  const { return PT(x+p.x,
      y+p.y); }
  PT operator - (const PT &p)  const { return PT(x-p.x,
      y-p.y); }
  PT operator * (double c)     const { return PT(x*c,
      y*c  ); }
  PT operator / (double c)     const { return PT(x/c,
      y/c  ); }
};

double dot(PT p, PT q)     { return p.x*q.x+p.y*q.y; }
double dist2(PT p, PT q)   { return dot(p-q,p-q); }
double cross(PT p, PT q)   { return p.x*q.y-p.y*q.x; }
ostream &operator<<(ostream &os, const PT &p) {
  os << "(" << p.x << "," << p.y << ")";
}

// rotate a point CCW or CW around the origin
PT RotateCCW90(PT p)   { return PT(-p.y,p.x); }
PT RotateCW90(PT p)    { return PT(p.y,-p.x); }
PT RotateCCW(PT p, double t) {
  return PT(p.x*cos(t)-p.y*sin(t), p.x*sin(t)+p.y*cos(t)
      );
}

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

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

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

// compute distance between point (x,y,z) and plane ax+
    by+cz=d
double DistancePointPlane(double x, double y, double z,
                          double a, double b, double c,
                             double d)
{
  return fabs(a*x+b*y+c*z-d)/sqrt(a*a+b*b+c*c);
}

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

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

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

// compute intersection of line passing through a and b
// with line passing through c and d, assuming that
    unique
// intersection exists; for segment intersection, check
```

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

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

// determine if point is in a possibly non-convex
    polygon (by William
// Randolph Franklin); returns 1 for strictly interior
    points, 0 for
// strictly exterior points, and 0 or 1 for the
    remaining points.
// Note that it is possible to convert this into an *
    exact* test using
// integer arithmetic by taking care of the division
    appropriately
// (making sure to deal with signs properly) and then by
     writing exact
// tests for checking point on polygon boundary
bool PointInPolygon(const vector<PT> &p, PT q) {
  bool c = 0;
  for (int i = 0; i < p.size(); i++){
    int j = (i+1)%p.size();
    if ((p[i].y <= q.y && q.y < p[j].y ||
      p[j].y <= q.y && q.y < p[i].y) &&
      q.x < p[i].x + (p[j].x - p[i].x) * (q.y - p[i].y)
          / (p[j].y - p[i].y))
      c = !c;
  }
  return c;
}

// determine if point is on the boundary of a polygon
bool PointOnPolygon(const vector<PT> &p, PT q) {
  for (int i = 0; i < p.size(); i++)
    if (dist2(ProjectPointSegment(p[i], p[(i+1)%p.size()
        ], q), q) < EPS)
      return true;
    return false;
}

// compute intersection of line through points a and b
    with
// circle centered at c with radius r > 0
vector<PT> CircleLineIntersection(PT a, PT b, PT c,
    double r) {
  vector<PT> ret;
  b = b-a;
  a = a-c;
  double A = dot(b, b);
  double B = dot(a, b);
  double C = dot(a, a) - r*r;
  double D = B*B - A*C;
  if (D < -EPS) return ret;
  ret.push_back(c+a+b*(-B+sqrt(D+EPS))/A);
  if (D > EPS)
    ret.push_back(c+a+b*(-B-sqrt(D))/A);
  return ret;
}

// compute intersection of circle centered at a with
    radius r
// with circle centered at b with radius R
vector<PT> CircleCircleIntersection(PT a, PT b, double r
    , double R) {
  vector<PT> ret;
  double d = sqrt(dist2(a, b));
  if (d > r+R || d+min(r, R) < max(r, R)) return ret;
  double x = (d*d-R*R+r*r)/(2*d);
  double y = sqrt(r*r-x*x);
  PT v = (b-a)/d;
  ret.push_back(a+v*x + RotateCCW90(v)*y);
  if (y > 0)
    ret.push_back(a+v*x - RotateCCW90(v)*y);
  return ret;
}

// This code computes the area or centroid of a (
    possibly nonconvex)
// polygon, assuming that the coordinates are listed in
    a clockwise or
// counterclockwise fashion.  Note that the centroid is
    often known as
// the "center of gravity" or "center of mass".
double ComputeSignedArea(const vector<PT> &p) {
  double area = 0;
  for(int i = 0; i < p.size(); i++) {
    int j = (i+1) % p.size();
    area += p[i].x*p[j].y - p[j].x*p[i].y;
  }
```

```
  return area / 2.0;
}

double ComputeArea(const vector<PT> &p) {
  return fabs(ComputeSignedArea(p));
}

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

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

int main() {

  // expected: (-5,2)
  cerr << RotateCCW90(PT(2,5)) << endl;

  // expected: (5,-2)
  cerr << RotateCW90(PT(2,5)) << endl;

  // expected: (-5,2)
  cerr << RotateCCW(PT(2,5),M_PI/2) << endl;

  // expected: (5,2)
  cerr << ProjectPointLine(PT(-5,-2), PT(10,4), PT(3,7))
       << endl;

  // expected: (5,2) (7.5,3) (2.5,1)
  cerr << ProjectPointSegment(PT(-5,-2), PT(10,4), PT
     (3,7)) << " "
       << ProjectPointSegment(PT(7.5,3), PT(10,4), PT
          (3,7)) << " "
       << ProjectPointSegment(PT(-5,-2), PT(2.5,1), PT
          (3,7)) << endl;

  // expected: 6.78903
  cerr << DistancePointPlane(4,-4,3,2,-2,5,-8) << endl;

  // expected: 1 0 1
  cerr << LinesParallel(PT(1,1), PT(3,5), PT(2,1), PT
     (4,5)) << " "
       << LinesParallel(PT(1,1), PT(3,5), PT(2,0), PT
          (4,5)) << " "
       << LinesParallel(PT(1,1), PT(3,5), PT(5,9), PT
          (7,13)) << endl;

  // expected: 0 0 1
  cerr << LinesCollinear(PT(1,1), PT(3,5), PT(2,1), PT
     (4,5)) << " "
       << LinesCollinear(PT(1,1), PT(3,5), PT(2,0), PT
          (4,5)) << " "
       << LinesCollinear(PT(1,1), PT(3,5), PT(5,9), PT
          (7,13)) << endl;

  // expected: 1 1 1 0
  cerr << SegmentsIntersect(PT(0,0), PT(2,4), PT(3,1),
     PT(-1,3)) << " "
       << SegmentsIntersect(PT(0,0), PT(2,4), PT(4,3),
          PT(0,5)) << " "
       << SegmentsIntersect(PT(0,0), PT(2,4), PT(2,-1),
          PT(-2,1)) << " "
       << SegmentsIntersect(PT(0,0), PT(2,4), PT(5,5),
          PT(1,7)) << endl;

  // expected: (1,2)
  cerr << ComputeLineIntersection(PT(0,0), PT(2,4), PT
     (3,1), PT(-1,3)) << endl;

  // expected: (1,1)
  cerr << ComputeCircleCenter(PT(-3,4), PT(6,1), PT(4,5)
     ) << endl;

  vector<PT> v;
  v.push_back(PT(0,0));
  v.push_back(PT(5,0));
  v.push_back(PT(5,5));
  v.push_back(PT(0,5));

  // expected: 1 1 1 0 0
  cerr << PointInPolygon(v, PT(2,2)) << " "
```

```
    << PointInPolygon(v, PT(2,0)) << " "
    << PointInPolygon(v, PT(0,2)) << " "
    << PointInPolygon(v, PT(5,2)) << " "
    << PointInPolygon(v, PT(2,5)) << endl;

// expected: 0 1 1 1 1
cerr << PointOnPolygon(v, PT(2,2)) << " "
     << PointOnPolygon(v, PT(2,0)) << " "
     << PointOnPolygon(v, PT(0,2)) << " "
     << PointOnPolygon(v, PT(5,2)) << " "
     << PointOnPolygon(v, PT(2,5)) << endl;

// expected: (1,6)
//           (5,4) (4,5)
//           blank line
//           (4,5) (5,4)
//           blank line
//           (4,5) (5,4)
vector<PT> u = CircleLineIntersection(PT(0,6), PT(2,6)
    , PT(1,1), 5);
for (int i = 0; i < u.size(); i++) cerr << u[i] << " "
    ; cerr << endl;
u = CircleLineIntersection(PT(0,9), PT(9,0), PT(1,1),
    5);
for (int i = 0; i < u.size(); i++) cerr << u[i] << " "
    ; cerr << endl;
u = CircleCircleIntersection(PT(1,1), PT(10,10), 5, 5)
    ;
for (int i = 0; i < u.size(); i++) cerr << u[i] << " "
    ; cerr << endl;
u = CircleCircleIntersection(PT(1,1), PT(8,8), 5, 5);
for (int i = 0; i < u.size(); i++) cerr << u[i] << " "
    ; cerr << endl;
u = CircleCircleIntersection(PT(1,1), PT(4.5,4.5), 10,
     sqrt(2.0)/2.0);
for (int i = 0; i < u.size(); i++) cerr << u[i] << " "
    ; cerr << endl;
u = CircleCircleIntersection(PT(1,1), PT(4.5,4.5), 5,
    sqrt(2.0)/2.0);
for (int i = 0; i < u.size(); i++) cerr << u[i] << " "
    ; cerr << endl;

// area should be 5.0
// centroid should be (1.1666666, 1.166666)
PT pa[] = { PT(0,0), PT(5,0), PT(1,1), PT(0,5) };
vector<PT> p(pa, pa+4);
PT c = ComputeCentroid(p);
cerr << "Area: " << ComputeArea(p) << endl;
cerr << "Centroid: " << c << endl;
```

```
  return 0;
}
```

3 Graphs/Trees

3.1 Graph structure example for our DFS and BFS algorithms

```python
graph = {'A': set(['B', 'C']),
         'B': set(['A', 'D', 'E']),
         'C': set(['A', 'F']),
         'D': set(['B']),
         'E': set(['B', 'F']),
         'F': set(['C', 'E'])}
```

3.2 Breadth-First Search

```python
def bfs(graph, start):
    visited, queue = set(), [start]
    while queue:
        vertex = queue.pop(0)
        if vertex not in visited:
            visited.add(vertex)
            queue.extend(graph[vertex] - visited)
    return visited

bfs(graph, 'A') # {'B', 'C', 'A', 'F', 'D', 'E'}
```

3.3 Breadth-First Search Paths

```python
def bfs_paths(graph, start, goal):
    queue = [(start, [start])]
    while queue:
        (vertex, path) = queue.pop(0)
        for next in graph[vertex] - set(path):
            if next == goal:
                yield path + [next]
            else:
                queue.append((next, path + [next]))

list(bfs_paths(graph, 'A', 'F')) # [['A', 'C', 'F'], ['A
    ', 'B', 'E', 'F']]
```

3.4 Breadth-First Search Shortest Path

```
def shortest_path(graph, start, goal):
    try:
        return next(bfs_paths(graph, start, goal))
    except StopIteration:
        return None

shortest_path(graph, 'A', 'F') # ['A', 'C', 'F']
```

3.5 Depth-First Search

```
def dfs(graph, start):
    visited, stack = set(), [start]
    while stack:
        vertex = stack.pop()
        if vertex not in visited:
            visited.add(vertex)
            stack.extend(graph[vertex] - visited)
    return visited

dfs(graph, 'A') # {'E', 'D', 'F', 'A', 'C', 'B'}
```

3.6 Depth-First Search Paths

```
#Returns all paths from start to goal
def dfs_paths(graph, start, goal):
    stack = [(start, [start])]
    while stack:
        (vertex, path) = stack.pop()
        for next in graph[vertex] - set(path):
            if next == goal:
                yield path + [next]
            else:
                stack.append((next, path + [next]))

list(dfs_paths(graph, 'A', 'F')) # [['A', 'C', 'F'], ['A
    ', 'B', 'E', 'F']]
```

3.7 Dijkstra's Algorithm

```
from collections import defaultdict
from heapq import *

def dijkstra(edges, f, t):
   g = defaultdict(list)
   for l,r,c in edges:
       g[l].append((c,r))

   q, seen = [(0,f,())], set()
   while q:
       (cost,v1,path) = heappop(q)
       if v1 not in seen:
           seen.add(v1)
           path = (v1, path)
           if v1 == t: return (cost, path)
           for c, v2 in g.get(v1, ()):
               if v2 not in seen:
                   heappush(q, (cost+c, v2, path))
   return float("inf")

#Code example
edges = [("A", "B", 7), ("A", "D", 5), ("B", "C", 8),
         ("B", "D", 9), ("B", "E", 7), ("C", "E", 5)]
print "A -> E:"
print dijkstra(edges, "A", "E")   #(14, ('E', ('B', ('A
    ', ()))))
```

3.8 Kruskal's Algorithm (including Merge-Find set)

```
parent = dict()
rank = dict()

def make_set(vertice):
    parent[vertice] = vertice
    rank[vertice] = 0

def find(vertice):
    if parent[vertice] != vertice:
        parent[vertice] = find(parent[vertice])
    return parent[vertice]

def union(vertice1, vertice2):
    root1 = find(vertice1)
    root2 = find(vertice2)
    if root1 != root2:
        if rank[root1] > rank[root2]:
            parent[root2] = root1
        else:
            parent[root1] = root2
        if rank[root1] == rank[root2]: rank[root2] += 1

def kruskal(graph):
    for vertice in graph['vertices']:
        make_set(vertice)
        minimum_spanning_tree = set()
        edges = list(graph['edges'])
        edges.sort()
        #print edges
```

```
    for edge in edges:
        weight, vertice1, vertice2 = edge
        if find(vertice1) != find(vertice2):
            union(vertice1, vertice2)
            minimum_spanning_tree.add(edge)

    return sorted(minimum_spanning_tree)
```

3.9 Bellman-Ford Algorithm

```
# Step 1: For each node prepare the destination and
    predecessor
def initialize(graph, source):
    d = {} # Stands for destination
    p = {} # Stands for predecessor
    for node in graph:
        d[node] = float('Inf') # We start admiting that
            the rest of nodes are very very far
        p[node] = None
    d[source] = 0 # For the source we know how to reach
    return d, p

def relax(node, neighbour, graph, d, p):
    # If the distance between the node and the neighbour
        is lower than the one I have now
    if d[neighbour] > d[node] + graph[node][neighbour]:
        # Record this lower distance
        d[neighbour]  = d[node] + graph[node][neighbour]
        p[neighbour] = node

def bellman_ford(graph, source):
    d, p = initialize(graph, source)
    for i in range(len(graph)-1): #Run this until is
        converges
        for u in graph:
            for v in graph[u]: #For each neighbour of u
                relax(u, v, graph, d, p) #Lets relax it

    # Step 3: check for negative-weight cycles
    for u in graph:
        for v in graph[u]:
            assert d[v] <= d[u] + graph[u][v]

    return d, p

def test():
    graph = {
        'a': {'b': -1, 'c':  4},
        'b': {'c':  3, 'd':  2, 'e':  2},
        'c': {},
        'd': {'b':  1, 'c':  5},
        'e': {'d': -3}
        }
    d, p = bellman_ford(graph, 'a')
        # d = {'a':0, 'b':-1, 'c':2, 'd':-2, 'e':1},
        # p = {'a':None, 'b':'a', 'c':'b', 'd':'e', 'e
            ':'b'}
```

3.10 Floyd-Warshall Algorithm

```
# Number of vertices in the graph
V = 4

# Define infinity as the large enough value. This value
    will be
# used for vertices not connected to each other
INF  = 99999

# Solves all pair shortest path via Floyd Warshall
    Algorithm
def floydWarshall(graph):

    """ dist[][] will be the output matrix that will
        finally
        have the shortest distances between every pair
            of vertices """
    """ initializing the solution matrix same as input
        graph matrix
    OR we can say that the initial values of shortest
        distances
    are based on shortest paths considering no
    intermediate vertices """
    dist = map(lambda i : map(lambda j : j , i) , graph)

    """ Add all vertices one by one to the set of
        intermediate
     vertices.
     ---> Before start of a iteration, we have shortest
        distances
     between all pairs of vertices such that the
        shortest
     distances consider only the vertices in set
    {0, 1, 2, .. k-1} as intermediate vertices.
      ----> After the end of a iteration, vertex no. k
         is
     added to the set of intermediate vertices and the
    set becomes {0, 1, 2, .. k}
    """
    for k in range(V):
```

```
        # pick all vertices as source one by one
        for i in range(V):

            # Pick all vertices as destination for the
            # above picked source
            for j in range(V):

                # If vertex k is on the shortest path
                   from
                # i to j, then update the value of dist[
                   i][j]
                dist[i][j] = min(dist[i][j] ,
                                 dist[i][k]+ dist[k][j]
                                )
    printSolution(dist)

"""
        10
   (0)------->(3)
    |         /|\
  5 |          |
    |          | 1
   \|/         |
   (1)------->(2)
        3           """
graph = [[0,5,INF,10],
         [INF,0,3,INF],
         [INF, INF, 0,   1],
         [INF, INF, INF, 0]
    ]

floydWarshall(graph) # [[0,5,8,9],[INF,0,3,4],[INF,INF
   ,0,1],[INF,INF,INF,0]]
```

3.11 Max Flow (Ford-Fulkerson Algorithm)

```
from collections import defaultdict

#This class represents a directed graph using adjacency
   matrix representation
class Graph:

    def __init__(self,graph):
        self.graph = graph # residual graph
        self.ROW = len(graph)
        #self.COL = len(gr[0])


    '''Returns true if there is a path from source 's'
       to sink 't' in
    residual graph. Also fills parent[] to store the
       path '''
    def BFS(self,s, t, parent):

        # Mark all the vertices as not visited
        visited =[False]*(self.ROW)

        # Create a queue for BFS
        queue=[]

        # Mark the source node as visited and enqueue it
        queue.append(s)
        visited[s] = True

        # Standard BFS Loop
        while queue:

            #Dequeue a vertex from queue and print it
            u = queue.pop(0)

            # Get all adjacent vertices of the dequeued
               vertex u
            # If a adjacent has not been visited, then
               mark it
            # visited and enqueue it
            for ind, val in enumerate(self.graph[u]):
                if visited[ind] == False and val > 0 :
                    queue.append(ind)
                    visited[ind] = True
                    parent[ind] = u

        # If we reached sink in BFS starting from source
           , then return
        # true, else false
        return True if visited[t] else False


    # Returns tne maximum flow from s to t in the given
       graph
    def FordFulkerson(self, source, sink):

        # This array is filled by BFS and to store path
        parent = [-1]*(self.ROW)

        max_flow = 0 # There is no flow initially

        # Augment the flow while there is path from
           source to sink
        while self.BFS(source, sink, parent) :
```

```python
            # Find minimum residual capacity of the
               edges along the
            # path filled by BFS. Or we can say find the
                maximum flow
            # through the path found.
            path_flow = float("Inf")
            s = sink
            while(s !=  source):
                path_flow = min (path_flow, self.graph[
                    parent[s]][s])
                s = parent[s]

            # Add path flow to overall flow
            max_flow +=  path_flow

            # update residual capacities of the edges
               and reverse edges
            # along the path
            v = sink
            while(v !=  source):
                u = parent[v]
                self.graph[u][v] -= path_flow
                self.graph[v][u] += path_flow
                v = parent[v]

        return max_flow


# Create a graph given in the above diagram

graph = [[0, 16, 13, 0, 0, 0],
        [0, 0, 10, 12, 0, 0],
        [0, 4, 0, 0, 14, 0],
        [0, 0, 9, 0, 0, 20],
        [0, 0, 0, 7, 0, 4],
        [0, 0, 0, 0, 0, 0]]

g = Graph(graph)

source = 0; sink = 5

print ("The maximum possible flow is %d " % g.
   FordFulkerson(source, sink))
```

4 Mathematics

4.1 Gauss-Jordan Elimination (Matrix inversion and linear system solving)

```python
def gauss_jordan(m, eps = 1.0/(10**10)):
  """Puts given matrix (2D array) into the Reduced Row
     Echelon Form.
     Returns True if successful, False if 'm' is
        singular.
     NOTE: make sure all the matrix items support
        fractions! Int matrix will NOT work!
     Written by Jarno Elonen in April 2005, released
        into Public Domain"""
  (h, w) = (len(m), len(m[0]))
  for y in range(0,h):
    maxrow = y
    for y2 in range(y+1, h):    # Find max pivot
      if abs(m[y2][y]) > abs(m[maxrow][y]):
        maxrow = y2
    (m[y], m[maxrow]) = (m[maxrow], m[y])
    if abs(m[y][y]) <= eps:     # Singular?
      return False
    for y2 in range(y+1, h):    # Eliminate column y
      c = m[y2][y] / m[y][y]
      for x in range(y, w):
        m[y2][x] -= m[y][x] * c
  for y in range(h-1, 0-1, -1): # Backsubstitute
    c  = m[y][y]
    for y2 in range(0,y):
      for x in range(w-1, y-1, -1):
        m[y2][x] -=  m[y][x] * m[y2][y] / c
    m[y][y] /= c
    for x in range(h, w):       # Normalize row y
      m[y][x] /= c
  return True

def solve(M, b):
  """
  solves M*x = b
  return vector x so that M*x = b
  :param M: a matrix in the form of a list of list
  :param b: a vector in the form of a simple list of
     scalars
  """
  m2 = [row[:]+[right] for row,right in zip(M,b) ]
  return [row[-1] for row in m2] if gauss_jordan(m2)
     else None

def inv(M):
  """
  return the inv of the matrix M
  """
  #clone the matrix and append the identity matrix
  # [int(i==j) for j in range_M] is nothing but the i(th
```

```
        row of the identity matrix
    m2 = [row[:]+[int(i==j) for j in range(len(M) )] for i
        ,row in enumerate(M) ]
    # extract the appended matrix (kind of m2[m:,...]
    return [row[len(M[0]):] for row in m2] if gauss_jordan
        (m2) else None

def zeros( s , zero=0):
    """
    return a matrix of size 'size'
    :param size: a tuple containing dimensions of the
        matrix
    :param zero: the value to use to fill the matrix (by
        default it's zero )
    """
    return [zeros(s[1:] ) for i in range(s[0] ) ] if not
        len(s) else zero
```

4.2 Miller-Rabin Primality Test

```
def miller_rabin(n, k):
    # The optimal number of rounds (k) for this test is
        40
    # for justification

    if n == 2:
        return True
    if n % 2 == 0:
        return False
    r, s = 0, n - 1
    while s % 2 == 0:
        r += 1
        s //= 2
    for _ in xrange(k):
        a = random.randrange(2, n - 1)
        x = pow(a, s, n)
        if x == 1 or x == n - 1:
            continue
        for _ in xrange(r - 1):
            x = pow(x, 2, n)
            if x == n - 1:
                break
        else:
            return False
    return True
```

4.3 Segment Tree

```
#encoding:utf-8
class SegmentTree(object):
    def __init__(self, start, end):
        self.start = start
        self.end = end
        self.max_value = {}
        self.sum_value = {}
        self.len_value = {}
        self._init(start, end)

    def add(self, start, end, weight=1):
        start = max(start, self.start)
        end = min(end, self.end)
        self._add(start, end, weight, self.start, self.
            end)
        return True

    def query_max(self, start, end):
        return self._query_max(start, end, self.start,
            self.end)

    def query_sum(self, start, end):
        return self._query_sum(start, end, self.start,
            self.end)

    def query_len(self, start, end):
        return self._query_len(start, end, self.start,
            self.end)

    """"""
    def _init(self, start, end):
        self.max_value[(start, end)] = 0
        self.sum_value[(start, end)] = 0
        self.len_value[(start, end)] = 0
        if start < end:
            mid = start + int((end - start) / 2)
            self._init(start, mid)
            self._init(mid+1, end)

    def _add(self, start, end, weight, in_start, in_end)
        :
        key = (in_start, in_end)
        if in_start == in_end:
            self.max_value[key] += weight
            self.sum_value[key] += weight
            self.len_value[key] = 1 if self.sum_value[
                key] > 0 else 0
            return

        mid = in_start + int((in_end - in_start) / 2)
```

```
        if mid >= end:
            self._add(start, end, weight, in_start, mid)
        elif mid+1 <= start:
            self._add(start, end, weight, mid+1, in_end)
        else:
            self._add(start, mid, weight, in_start, mid)
            self._add(mid+1, end, weight, mid+1, in_end)
        self.max_value[key] = max(self.max_value[(
            in_start, mid)], self.max_value[(mid+1,
            in_end)])
        self.sum_value[key] = self.sum_value[(in_start,
            mid)] + self.sum_value[(mid+1, in_end)]
        self.len_value[key] = self.len_value[(in_start,
            mid)] + self.len_value[(mid+1, in_end)]

    def _query_max(self, start, end, in_start, in_end):
        if start == in_start and end == in_end:
            ans = self.max_value[(start, end)]
        else:
            mid = in_start + int((in_end - in_start) /
                2)
            if mid >= end:
                ans = self._query_max(start, end,
                    in_start, mid)
            elif mid+1 <= start:
                ans = self._query_max(start, end, mid+1,
                     in_end)
            else:
                ans = max(self._query_max(start, mid,
                    in_start, mid),
                        self._query_max(mid+1, end, mid
                        +1, in_end))
        #print start, end, in_start, in_end, ans
        return ans

    def _query_sum(self, start, end, in_start, in_end):
        if start == in_start and end == in_end:
            ans = self.sum_value[(start, end)]
        else:
            mid = in_start + int((in_end - in_start) /
                2)
            if mid >= end:
                ans = self._query_sum(start, end,
                    in_start, mid)
            elif mid+1 <= start:
                ans = self._query_sum(start, end, mid+1,
                     in_end)
            else:
                ans = self._query_sum(start, mid,
                    in_start, mid) + self._query_sum(mid
                    +1, end, mid+1, in_end)
        return ans

    def _query_len(self, start, end, in_start, in_end):
        if start == in_start and end == in_end:
            ans = self.len_value[(start, end)]
        else:
            mid = in_start + int((in_end - in_start) /
                2)
            if mid >= end:
                ans = self._query_len(start, end,
                    in_start, mid)
            elif mid+1 <= start:
                ans = self._query_len(start, end, mid+1,
                     in_end)
            else:
                ans = self._query_len(start, mid,
                    in_start, mid) + self._query_len(mid
                    +1, end, mid+1, in_end)

        #print start, end, in_start, in_end, ans
        return ans
```

4.4 Prime Number Sieve (generator)

```
from itertools import count

def postponed_sieve():                    # postponed
    sieve, by Will Ness
    yield 2; yield 3; yield 5; yield 7;   # original code
         David Eppstein,
    sieve = {}                            #   Alex
        Martelli, ActiveState Recipe 2002
    ps = postponed_sieve()                # a separate
        base Primes Supply:
    p = next(ps) and next(ps)             # (3) a Prime
        to add to dict
    q = p*p                               # (9) its
        sQuare
    for c in count(9,2):                  # the Candidate
        if c in sieve:                # c's a multiple of
             some base prime
            s = sieve.pop(c)          #     i.e. a
                composite ; or
        elif c < q:
             yield c                  # a prime
             continue
        else:   # (c==q):             # or the next base
            prime's square:
```

```
            s=count(q+2*p,2*p)      #    (9+6, by 6 :
                15,21,27,33,...)
            p=next(ps)              #    (5)
            q=p*p                   #    (25)
        for m in s:                 # the next multiple
            if m not in sieve:      # no duplicates
                break
        sieve[m] = s                # original test
           entry: ideone.com/WFv4f
```

4.5 GCD and Euler's Totient Function

```
# Function to return gcd of a and b
def gcd(a, b):
    if a == 0:
        return b
    return gcd(b%a, a)

# A simple method to evaluate Euler Totient Function
def phi(n):
    result = 1
    for i in range(2, n):
        if gcd(i, n) == 1:
            result = result + 1
    return result
```

5 Strings

5.1 Knuth-Morris-Pratt Algorithm (fast pattern matching)

```
def KnuthMorrisPratt(text, pattern):

    '''Yields all starting positions of copies of the
        pattern in the text.
Calling conventions are similar to string.find, but its
    arguments can be
lists or iterators, not just strings, it returns all
    matches, not just
the first one, and it does not need the whole text in
    memory at once.
Whenever it yields, it will have read the text exactly
    up to and including
the match that caused the yield.'''

    # allow indexing into pattern and protect against
        change during yield
    pattern = list(pattern)

    # build table of shift amounts
    shifts = [1] * (len(pattern) + 1)
    shift = 1
    for pos in range(len(pattern)):
        while shift <= pos and pattern[pos] != pattern[
            pos-shift]:
            shift += shifts[pos-shift]
        shifts[pos+1] = shift

    # do the actual search
    startPos = 0
    matchLen = 0
    for c in text:
        while matchLen == len(pattern) or \
              matchLen >= 0 and pattern[matchLen] != c:
            startPos += shifts[matchLen]
            matchLen -= shifts[matchLen]
        matchLen += 1
        if matchLen == len(pattern):
            yield startPos
```

5.2 Rabin-Karp Algorithm (multiple pattern matching)

```
# d is the number of characters in input alphabet
d = 256

# pat  -> pattern
# txt  -> text
# q    -> A prime number

def search(pat, txt, q):
    M = len(pat)
    N = len(txt)
    i = 0
    j = 0
    p = 0    # hash value for pattern
    t = 0    # hash value for txt
    h = 1

    # The value of h would be "pow(d, M-1)%q"
    for i in xrange(M-1):
        h = (h*d)%q

    # Calculate the hash value of pattern and first
        window
    # of text
    for i in xrange(M):
        p = (d*p + ord(pat[i]))%q
        t = (d*t + ord(txt[i]))%q
```

```
    # Slide the pattern over text one by one
    for i in xrange(N-M+1):
        # Check the hash values of current window of text and
        # pattern if the hash values match then only check
        # for characters on by one
        if p==t:
            # Check for characters one by one
            for j in xrange(M):
                if txt[i+j] != pat[j]:
                    break

            j+=1
            # if p == t and pat[0...M-1] = txt[i, i+1, ...i+M-1]
            if j==M:
                print "Pattern found at index " + str(i)

        # Calculate hash value for next window of text: Remove
        # leading digit, add trailing digit
        if i < N-M:
            t = (d*(t-ord(txt[i])*h) + ord(txt[i+M]))%q

            # We might get negative values of t, converting it to
            # positive
            if t < 0:
                t = t+q

# Driver program to test the above function
txt = "GEEKS FOR GEEKS"
pat = "GEEK"
q = 101 # A prime number
search(pat,txt,q)
```

6 Techniques

6.1 Various algorithm techniques

```
Recursion
Divide and conquer
        Finding interesting points in N log N
Greedy algorithm
        Scheduling
        Max contigous subvector sum
        Invariants
        Huffman encoding
Graph theory
        Dynamic graphs (extra book-keeping)
        Breadth first search
        Depth first search
        * Normal trees / DFS trees
        Dijkstra's algoritm
        MST: Prim's algoritm
        Bellman-Ford
        Konig's theorem and vertex cover
        Min-cost max flow
        Lovasz toggle
        Matrix tree theorem
        Maximal matching, general graphs
        Hopcroft-Karp
        Hall's marriage theorem
        Graphical sequences
        Floyd-Warshall
        Eulercykler
        Flow networks
        * Augumenting paths
        * Edmonds-Karp
        Bipartite matching
        Min. path cover
        Topological sorting
        Strongly connected components
        2-SAT
        Cutvertices, cutedges och biconnected components
        Edge coloring
        * Trees
        Vertex coloring
        * Bipartite graphs (=> trees)
        * 3^n (special case of set cover)
        Diameter and centroid
        K'th shortest path
        Shortest cycle
Dynamic programmering
        Knapsack
        Coin change
        Longest common subsequence
        Longest increasing subsequence
        Number of paths in a dag
        Shortest path in a dag
        Dynprog over intervals
        Dynprog over subsets
        Dynprog over probabilities
        Dynprog over trees
        3^n set cover
        Divide and conquer
        Knuth optimization
        Convex hull optimizations
```

- RMQ (sparse table a.k.a 2^k-jumps)
- Bitonic cycle
- Log partitioning (loop over most restricted)

Combinatorics
- Computation of binomial coefficients
- Pigeon-hole principle
- Inclusion/exclusion
- Catalan number
- Pick's theorem

Number theory
- Integer parts
- Divisibility
- Euklidean algorithm
- Modular arithmetic
- ⋆ Modular multiplication
- ⋆ Modular inverses
- ⋆ Modular exponentiation by squaring
- Chinese remainder theorem
- Fermat's small theorem
- Euler's theorem
- Phi function
- Frobenius number
- Quadratic reciprocity
- Pollard-Rho
- Miller-Rabin
- Hensel lifting
- Vieta root jumping

Game theory
- Combinatorial games
- Game trees
- Mini-max
- Nim
- Games on graphs
- Games on graphs with loops
- Grundy numbers
- Bipartite games without repetition
- General games without repetition
- Alpha-beta pruning

Probability theory

Optimization
- Binary search
- Ternary search
- Unimodality and convex functions
- Binary search on derivative

Numerical methods
- Numeric integration
- Newton's method
- Root-finding with binary/ternary search
- Golden section search

Matrices
- Gaussian elimination
- Exponentiation by squaring

Sorting
- Radix sort

Geometry
- Coordinates and vectors
- ⋆ Cross product
- ⋆ Scalar product
- Convex hull
- Polygon cut
- Closest pair
- Coordinate-compression
- Quadtrees
- KD-trees
- All segment-segment intersection

Sweeping
- Discretization (convert to events and sweep)
- Angle sweeping
- Line sweeping
- Discrete second derivatives

Strings
- Longest common substring
- Palindrome subsequences
- Knuth-Morris-Pratt
- Tries
- Rolling polynom hashes
- Suffix array
- Suffix tree
- Aho-Corasick
- Manacher's algorithm
- Letter position lists

Combinatorial search
- Meet in the middle
- Brute-force with pruning
- Best-first (A⋆)
- Bidirectional search
- Iterative deepening DFS / A⋆

Data structures
- LCA (2^k-jumps in trees in general)
- Pull/push-technique on trees
- Heavy-light decomposition
- Centroid decomposition
- Lazy propagation
- Self-balancing trees
- Convex hull trick (wcipeg.com/wiki/Convex_hull_trick)
- Monotone queues / monotone stacks / sliding queues
- Sliding queue using 2 stacks
- Persistent segment tree

$f(n) = O(g(n))$	iff $\exists$ positive c, n_0 such that $0 \leq f(n) \leq cg(n)\ \forall n \geq n_0$.
$f(n) = \Omega(g(n))$	iff $\exists$ positive c, n_0 such that $f(n) \geq cg(n) \geq 0\ \forall n \geq n_0$.
$f(n) = \Theta(g(n))$	iff $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.
$f(n) = o(g(n))$	iff $\lim_{n\to\infty} f(n)/g(n) = 0$.
$\lim\limits_{n\to\infty} a_n = a$	iff $\forall \epsilon > 0$, $\exists n_0$ such that $\lvert a_n - a\rvert < \epsilon$, $\forall n \geq n_0$.
$\sup S$	least $b \in \mathbb{R}$ such that $b \geq s$, $\forall s \in S$.
$\inf S$	greatest $b \in \mathbb{R}$ such that $b \leq s$, $\forall s \in S$.
$\liminf\limits_{n\to\infty} a_n$	$\lim\limits_{n\to\infty} \inf\{a_i \mid i \geq n, i \in \mathbb{N}\}$.
$\limsup\limits_{n\to\infty} a_n$	$\lim\limits_{n\to\infty} \sup\{a_i \mid i \geq n, i \in \mathbb{N}\}$.
$\binom{n}{k}$	Combinations: Size k subsets of a size n set.
$\left[{n \atop k}\right]$	Stirling numbers (1st kind): Arrangements of an n element set into k cycles.
$\left\{{n \atop k}\right\}$	Stirling numbers (2nd kind): Partitions of an n element set into k non-empty sets.
$\left\langle{n \atop k}\right\rangle$	1st order Eulerian numbers: Permutations $\pi_1\pi_2\dots\pi_n$ on $\{1, 2, \dots, n\}$ with k ascents.
$\left\langle\!\left\langle{n \atop k}\right\rangle\!\right\rangle$	2nd order Eulerian numbers.
C_n	Catalan Numbers: Binary trees with $n+1$ vertices.

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}, \quad \sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}, \quad \sum_{i=1}^{n} i^3 = \frac{n^2(n+1)^2}{4}.$$

In general:

$$\sum_{i=1}^{n} i^m = \frac{1}{m+1}\left[(n+1)^{m+1} - 1 - \sum_{i=1}^{n}\left((i+1)^{m+1} - i^{m+1} - (m+1)i^m\right)\right]$$

$$\sum_{i=1}^{n-1} i^m = \frac{1}{m+1}\sum_{k=0}^{m}\binom{m+1}{k} B_k n^{m+1-k}.$$

Geometric series:

$$\sum_{i=0}^{n} c^i = \frac{c^{n+1}-1}{c-1}, \quad c \neq 1, \quad \sum_{i=0}^{\infty} c^i = \frac{1}{1-c}, \quad \sum_{i=1}^{\infty} c^i = \frac{c}{1-c}, \quad |c| < 1,$$

$$\sum_{i=0}^{n} ic^i = \frac{nc^{n+2} - (n+1)c^{n+1} + c}{(c-1)^2}, \quad c \neq 1, \quad \sum_{i=0}^{\infty} ic^i = \frac{c}{(1-c)^2}, \quad |c| < 1.$$

Harmonic series:

$$H_n = \sum_{i=1}^{n}\frac{1}{i}, \qquad \sum_{i=1}^{n} iH_i = \frac{n(n+1)}{2}H_n - \frac{n(n-1)}{4}.$$

$$\sum_{i=1}^{n} H_i = (n+1)H_n - n, \quad \sum_{i=1}^{n}\binom{i}{m}H_i = \binom{n+1}{m+1}\left(H_{n+1} - \frac{1}{m+1}\right).$$

1. $\binom{n}{k} = \frac{n!}{(n-k)!k!}$, **2.** $\sum_{k=0}^{n}\binom{n}{k} = 2^n$, **3.** $\binom{n}{k} = \binom{n}{n-k}$,

4. $\binom{n}{k} = \frac{n}{k}\binom{n-1}{k-1}$, **5.** $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$,

6. $\binom{n}{m}\binom{m}{k} = \binom{n}{k}\binom{n-k}{m-k}$, **7.** $\sum_{k=0}^{n}\binom{r+k}{k} = \binom{r+n+1}{n}$,

8. $\sum_{k=0}^{n}\binom{k}{m} = \binom{n+1}{m+1}$, **9.** $\sum_{k=0}^{n}\binom{r}{k}\binom{s}{n-k} = \binom{r+s}{n}$,

10. $\binom{n}{k} = (-1)^k\binom{k-n-1}{k}$, **11.** $\left\{{n \atop 1}\right\} = \left\{{n \atop n}\right\} = 1$,

12. $\left\{{n \atop 2}\right\} = 2^{n-1} - 1$, **13.** $\left\{{n \atop k}\right\} = k\left\{{n-1 \atop k}\right\} + \left\{{n-1 \atop k-1}\right\}$,

14. $\left[{n \atop 1}\right] = (n-1)!$, **15.** $\left[{n \atop 2}\right] = (n-1)!H_{n-1}$, **16.** $\left[{n \atop n}\right] = 1$, **17.** $\left[{n \atop k}\right] \geq \left\{{n \atop k}\right\}$,

18. $\left[{n \atop k}\right] = (n-1)\left[{n-1 \atop k}\right] + \left[{n-1 \atop k-1}\right]$, **19.** $\left\{{n \atop n-1}\right\} = \left[{n \atop n-1}\right] = \binom{n}{2}$, **20.** $\sum_{k=0}^{n}\left[{n \atop k}\right] = n!$, **21.** $C_n = \frac{1}{n+1}\binom{2n}{n}$,

22. $\left\langle{n \atop 0}\right\rangle = \left\langle{n \atop n-1}\right\rangle = 1$, **23.** $\left\langle{n \atop k}\right\rangle = \left\langle{n \atop n-1-k}\right\rangle$, **24.** $\left\langle{n \atop k}\right\rangle = (k+1)\left\langle{n-1 \atop k}\right\rangle + (n-k)\left\langle{n-1 \atop k-1}\right\rangle$,

25. $\left\langle{0 \atop k}\right\rangle = \begin{cases} 1 & \text{if } k = 0, \\ 0 & \text{otherwise} \end{cases}$ **26.** $\left\langle{n \atop 1}\right\rangle = 2^n - n - 1$, **27.** $\left\langle{n \atop 2}\right\rangle = 3^n - (n+1)2^n + \binom{n+1}{2}$,

28. $x^n = \sum_{k=0}^{n}\left\langle{n \atop k}\right\rangle\binom{x+k}{n}$, **29.** $\left\langle{n \atop m}\right\rangle = \sum_{k=0}^{m}\binom{n+1}{k}(m+1-k)^n(-1)^k$, **30.** $m!\left\{{n \atop m}\right\} = \sum_{k=0}^{n}\left\langle{n \atop k}\right\rangle\binom{k}{n-m}$,

31. $\left\langle{n \atop m}\right\rangle = \sum_{k=0}^{n}\left\{{n \atop k}\right\}\binom{n-k}{m}(-1)^{n-k-m}k!$, **32.** $\left\langle\!\left\langle{n \atop 0}\right\rangle\!\right\rangle = 1$, **33.** $\left\langle\!\left\langle{n \atop n}\right\rangle\!\right\rangle = 0$ for $n \neq 0$,

34. $\left\langle\!\left\langle{n \atop k}\right\rangle\!\right\rangle = (k+1)\left\langle\!\left\langle{n-1 \atop k}\right\rangle\!\right\rangle + (2n-1-k)\left\langle\!\left\langle{n-1 \atop k-1}\right\rangle\!\right\rangle$, **35.** $\sum_{k=0}^{n}\left\langle\!\left\langle{n \atop k}\right\rangle\!\right\rangle = \frac{(2n)^{\underline{n}}}{2^n}$,

36. $\left\{{x \atop x-n}\right\} = \sum_{k=0}^{n}\left\langle\!\left\langle{n \atop k}\right\rangle\!\right\rangle\binom{x+n-1-k}{2n}$, **37.** $\left\{{n+1 \atop m+1}\right\} = \sum_{k}\binom{n}{k}\left\{{k \atop m}\right\} = \sum_{k=0}^{n}\left\{{k \atop m}\right\}(m+1)^{n-k}$, The Chinese remainder theorem: There exists a number C such that:

$$C \equiv r_1 \bmod m_1$$
$$\vdots \quad \vdots \quad \vdots$$
$$C \equiv r_n \bmod m_n$$

if m_i and m_j are relatively prime for $i \neq j$.

Euler's function: $\phi(x)$ is the number of positive integers less than x relatively prime to x. If $\prod_{i=1}^n p_i^{e_i}$ is the prime factorization of x then

$$\phi(x) = \prod_{i=1}^n p_i^{e_i - 1}(p_i - 1).$$

Euler's theorem: If a and b are relatively prime then

$$1 \equiv a^{\phi(b)} \bmod b.$$

Fermat's theorem:

$$1 \equiv a^{p-1} \bmod p.$$

The Euclidean algorithm: if $a > b$ are integers then

$$\gcd(a, b) = \gcd(a \bmod b, b).$$

If $\prod_{i=1}^n p_i^{e_i}$ is the prime factorization of x then

$$S(x) = \sum_{d|x} d = \prod_{i=1}^n \frac{p_i^{e_i+1} - 1}{p_i - 1}.$$

Perfect Numbers: x is an even perfect number iff $x = 2^{n-1}(2^n - 1)$ and $2^n - 1$ is prime.

Wilson's theorem: n is a prime iff

$$(n-1)! \equiv -1 \bmod n.$$

Möbius inversion:

$$\mu(i) = \begin{cases} 1 & \text{if } i = 1. \\ 0 & \text{if } i \text{ is not square-free.} \\ (-1)^r & \text{if } i \text{ is the product of} \\ & r \text{ distinct primes.} \end{cases}$$

If

$$G(a) = \sum_{d|a} F(d),$$

then

$$F(a) = \sum_{d|a} \mu(d) G\left(\frac{a}{d}\right).$$

Prime numbers:

$$p_n = n \ln n + n \ln \ln n - n + n \frac{\ln \ln n}{\ln n} + O\left(\frac{n}{\ln n}\right),$$

$$\pi(n) = \frac{n}{\ln n} + \frac{n}{(\ln n)^2} + \frac{2!n}{(\ln n)^3} + O\left(\frac{n}{(\ln n)^4}\right).$$

Definitions:	
Loop	An edge connecting a vertex to itself.
Directed	Each edge has a direction.
Simple	Graph with no loops or multi-edges.
Walk	A sequence $v_0 e_1 v_1 \ldots e_\ell v_\ell$.
Trail	A walk with distinct edges.
Path	A trail with distinct vertices.
Connected	A graph where there exists a path between any two vertices.
Component	A maximal connected subgraph.
Tree	A connected acyclic graph.
Free tree	A tree with no root.
DAG	Directed acyclic graph.
Eulerian	Graph with a trail visiting each edge exactly once.
Hamiltonian	Graph with a cycle visiting each vertex exactly once.
Cut	A set of edges whose removal increases the number of components.
Cut-set	A minimal cut.
Cut edge	A size 1 cut.
k-Connected	A graph connected with the removal of any $k-1$ vertices.
k-Tough	$\forall S \subseteq V, S \neq \emptyset$ we have $k \cdot c(G - S) \leq \|S\|$.
k-Regular	A graph where all vertices have degree k.
k-Factor	A k-regular spanning subgraph.
Matching	A set of edges, no two of which are adjacent.
Clique	A set of vertices, all of which are adjacent.
Ind. set	A set of vertices, none of which are adjacent.
Vertex cover	A set of vertices which cover all edges.
Planar graph	A graph which can be embeded in the plane.
Plane graph	An embedding of a planar graph.

$$\sum_{v \in V} \deg(v) = 2m.$$

If G is planar then $n - m + f = 2$, so

$$f \leq 2n - 4, \quad m \leq 3n - 6.$$

Any planar graph has a vertex with degree ≤ 5.

Notation:	
$E(G)$	Edge set
$V(G)$	Vertex set
$c(G)$	Number of components
$G[S]$	Induced subgraph
$\deg(v)$	Degree of v
$\Delta(G)$	Maximum degree
$\delta(G)$	Minimum degree
$\chi(G)$	Chromatic number
$\chi_E(G)$	Edge chromatic number
G^c	Complement graph
K_n	Complete graph
K_{n_1,n_2}	Complete bipartite graph
$\mathrm{r}(k, \ell)$	Ramsey number

Geometry

Projective coordinates: triples (x, y, z), not all x, y and z zero.

$$(x, y, z) = (cx, cy, cz) \quad \forall c \neq 0.$$

Cartesian	Projective
(x, y)	$(x, y, 1)$
$y = mx + b$	$(m, -1, b)$
$x = c$	$(1, 0, -c)$

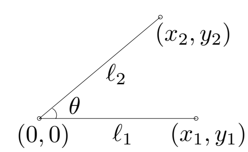
Distance formula, L_p and L_∞ metric:

$$\sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2},$$
$$\left[|x_1 - x_0|^p + |y_1 - y_0|^p\right]^{1/p},$$
$$\lim_{p \to \infty} \left[|x_1 - x_0|^p + |y_1 - y_0|^p\right]^{1/p}.$$

Area of triangle (x_0, y_0), (x_1, y_1) and (x_2, y_2):

$$\tfrac{1}{2} \operatorname{abs} \begin{vmatrix} x_1 - x_0 & y_1 - y_0 \\ x_2 - x_0 & y_2 - y_0 \end{vmatrix}.$$

Angle formed by three points:



$$\cos \theta = \frac{(x_1, y_1) \cdot (x_2, y_2)}{\ell_1 \ell_2}.$$

Line through two points (x_0, y_0) and (x_1, y_1):

$$\begin{vmatrix} x & y & 1 \\ x_0 & y_0 & 1 \\ x_1 & y_1 & 1 \end{vmatrix} = 0.$$

Area of circle, volume of sphere:

$$A = \pi r^2, \qquad V = \tfrac{4}{3} \pi r^3.$$

If I have seen farther than others, it is because I have stood on the shoulders of giants.
– Issac Newton

Taylor's series:

$$f(x) = f(a) + (x-a)f'(a) + \frac{(x-a)^2}{2}f''(a) + \cdots = \sum_{i=0}^{\infty} \frac{(x-a)^i}{i!} f^{(i)}(a).$$

Expansions:

$$\frac{1}{1-x} = 1 + x + x^2 + x^3 + x^4 + \cdots = \sum_{i=0}^{\infty} x^i,$$

$$\frac{1}{1-cx} = 1 + cx + c^2x^2 + c^3x^3 + \cdots = \sum_{i=0}^{\infty} c^i x^i,$$

$$\frac{1}{1-x^n} = 1 + x^n + x^{2n} + x^{3n} + \cdots = \sum_{i=0}^{\infty} x^{ni},$$

$$\frac{x}{(1-x)^2} = x + 2x^2 + 3x^3 + 4x^4 + \cdots = \sum_{i=0}^{\infty} ix^i,$$

$$x^k \frac{d^n}{dx^n}\left(\frac{1}{1-x}\right) = x + 2^n x^2 + 3^n x^3 + 4^n x^4 + \cdots = \sum_{i=0}^{\infty} i^n x^i,$$

$$e^x = 1 + x + \tfrac{1}{2}x^2 + \tfrac{1}{6}x^3 + \cdots = \sum_{i=0}^{\infty} \frac{x^i}{i!},$$

$$\ln(1+x) = x - \tfrac{1}{2}x^2 + \tfrac{1}{3}x^3 - \tfrac{1}{4}x^4 - \cdots = \sum_{i=1}^{\infty} (-1)^{i+1} \frac{x^i}{i},$$

$$\ln \frac{1}{1-x} = x + \tfrac{1}{2}x^2 + \tfrac{1}{3}x^3 + \tfrac{1}{4}x^4 + \cdots = \sum_{i=1}^{\infty} \frac{x^i}{i},$$

$$\sin x = x - \tfrac{1}{3!}x^3 + \tfrac{1}{5!}x^5 - \tfrac{1}{7!}x^7 + \cdots = \sum_{i=0}^{\infty} (-1)^i \frac{x^{2i+1}}{(2i+1)!},$$

$$\cos x = 1 - \tfrac{1}{2!}x^2 + \tfrac{1}{4!}x^4 - \tfrac{1}{6!}x^6 + \cdots = \sum_{i=0}^{\infty} (-1)^i \frac{x^{2i}}{(2i)!},$$

$$\tan^{-1} x = x - \tfrac{1}{3}x^3 + \tfrac{1}{5}x^5 - \tfrac{1}{7}x^7 + \cdots = \sum_{i=0}^{\infty} (-1)^i \frac{x^{2i+1}}{(2i+1)},$$

$$(1+x)^n = 1 + nx + \tfrac{n(n-1)}{2}x^2 + \cdots = \sum_{i=0}^{\infty} \binom{n}{i} x^i,$$

$$\frac{1}{(1-x)^{n+1}} = 1 + (n+1)x + \tbinom{n+2}{2}x^2 + \cdots = \sum_{i=0}^{\infty} \binom{i+n}{i} x^i,$$

$$\frac{x}{e^x - 1} = 1 - \tfrac{1}{2}x + \tfrac{1}{12}x^2 - \tfrac{1}{720}x^4 + \cdots = \sum_{i=0}^{\infty} \frac{B_i x^i}{i!},$$

$$\frac{1}{2x}(1 - \sqrt{1-4x}) = 1 + x + 2x^2 + 5x^3 + \cdots = \sum_{i=0}^{\infty} \frac{1}{i+1}\binom{2i}{i} x^i,$$

$$\frac{1}{\sqrt{1-4x}} = 1 + x + 2x^2 + 6x^3 + \cdots = \sum_{i=0}^{\infty} \binom{2i}{i} x^i,$$

$$\frac{1}{\sqrt{1-4x}}\left(\frac{1-\sqrt{1-4x}}{2x}\right)^n = 1 + (2+n)x + \tbinom{4+n}{2}x^2 + \cdots = \sum_{i=0}^{\infty} \binom{2i+n}{i} x^i,$$

$$\frac{1}{1-x}\ln\frac{1}{1-x} = x + \tfrac{3}{2}x^2 + \tfrac{11}{6}x^3 + \tfrac{25}{12}x^4 + \cdots = \sum_{i=1}^{\infty} H_i x^i,$$

$$\frac{1}{2}\left(\ln\frac{1}{1-x}\right)^2 = \tfrac{1}{2}x^2 + \tfrac{3}{4}x^3 + \tfrac{11}{24}x^4 + \cdots = \sum_{i=2}^{\infty} \frac{H_{i-1}x^i}{i},$$

$$\frac{x}{1-x-x^2} = x + x^2 + 2x^3 + 3x^4 + \cdots = \sum_{i=0}^{\infty} F_i x^i,$$

$$\frac{F_n x}{1 - (F_{n-1} + F_{n+1})x - (-1)^n x^2} = F_n x + F_{2n}x^2 + F_{3n}x^3 + \cdots = \sum_{i=0}^{\infty} F_{ni} x^i.$$

Ordinary power series:

$$A(x) = \sum_{i=0}^{\infty} a_i x^i.$$

Exponential power series:

$$A(x) = \sum_{i=0}^{\infty} a_i \frac{x^i}{i!}.$$

Dirichlet power series:

$$A(x) = \sum_{i=1}^{\infty} \frac{a_i}{i^x}.$$

Binomial theorem:

$$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^{n-k} y^k.$$

Difference of like powers:

$$x^n - y^n = (x-y) \sum_{k=0}^{n-1} x^{n-1-k} y^k.$$

For ordinary power series:

$$\alpha A(x) + \beta B(x) = \sum_{i=0}^{\infty} (\alpha a_i + \beta b_i) x^i,$$

$$x^k A(x) = \sum_{i=k}^{\infty} a_{i-k} x^i,$$

$$\frac{A(x) - \sum_{i=0}^{k-1} a_i x^i}{x^k} = \sum_{i=0}^{\infty} a_{i+k} x^i,$$

$$A(cx) = \sum_{i=0}^{\infty} c^i a_i x^i,$$

$$A'(x) = \sum_{i=0}^{\infty} (i+1) a_{i+1} x^i,$$

$$xA'(x) = \sum_{i=1}^{\infty} i a_i x^i,$$

$$\int A(x)\,dx = \sum_{i=1}^{\infty} \frac{a_{i-1}}{i} x^i,$$

$$\frac{A(x) + A(-x)}{2} = \sum_{i=0}^{\infty} a_{2i} x^{2i},$$

$$\frac{A(x) - A(-x)}{2} = \sum_{i=0}^{\infty} a_{2i+1} x^{2i+1}.$$

Summation: If $b_i = \sum_{j=0}^{i} a_i$ then

$$B(x) = \frac{1}{1-x} A(x).$$

Convolution:

$$A(x)B(x) = \sum_{i=0}^{\infty} \left(\sum_{j=0}^{i} a_j b_{i-j} \right) x^i.$$

God made the natural numbers;
all the rest is the work of man.
– Leopold Kronecker