# University of Calgary Team Resource Document

May 22, 2023

## Contents



# 1 General Tips

The constant $\pi$ is built-in as `math.pi`.

Beware Python's recursion limit of 1000.

## 1.1 Optimized I/O

This code speeds up processing for large amounts of input and output, but will break solutions to interactive problems.

```
from sys import stdin, stdout
inputs = stdin.read().splitlines()
def input():
  input.line += 1
  return inputs[input.line-1]
input.line = 0
outputs = []
def print(*args):
  outputs.append(' '.join(str(arg) for arg in args))
# Write your solution here
# Use input() and print() normally
stdout.write('\n'.join(outputs))
```

# 2 Geometry

## 2.1 Basic 2D Geometry

### Basic definitions

```
EP = 1e-9 # do not use for angles
BAD = complex(1e100,1e100)
```

### Cross/dot product, same slope test

$$\left(\vec{a} \times \vec{b}\right)_z = a_x b_y - a_y b_x$$

```
def cp(a, b): return (a.conjugate()*b).imag
def dp(a, b): return (a.conjugate()*b).real
def ss(a, b): return abs(cp(a,b)) < EP
```

### Orientation: −1=CW, 1=CCW, 0=collinear

```
# Can be used to check if a point is on a line (0)
def ccw(a, b, c):
  r = cp(b-a, c-a)
  if abs(r) < EP: return 0
  return 1 if r > 0 else -1
```

### Check if $x$ is on line segment from $p_1$ to $p_2$

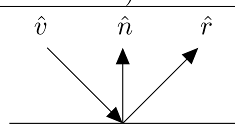
```
def onSeg(p1, p2, x):
  # Sometimes 1e-14 may be a better choice here
  return abs(abs(p2-p1)-abs(x-p1)-abs(x-p2))<EP
```

### Angle between vectors (0 to $\pi$)

```
abs(cmath.phase(x/y))
```

### Reflect vector using line normal (unit vectors)

Orientation of $\hat{n}$ does not matter.
Use `r = n * n / (-v)`




### Intersection point of two lines

```
def lineIntersect(p1, v1, p2, v2):
  # If exact same line, pick random point (p1)
  if ss(v1, v2):
    return p1 if ss(v1, p2-p1) else BAD
  return p1 + (cp(p2-p1,v2)/cp(v1,v2))*v1
```

### Point to line distance ($x$ to $p + \vec{v}t$)

```
def ptToLine(p, v, x):
  # Closest point on line: p + v*dp(v, x-p)
  return abs(cp(v, x-p) / abs(v))
```

### Line segment $(a, b)$ to point $p$ distance

```
def lsp_dist(a, b, p):
  if dp(b-a, p-a) > 0 and dp(a-b, p-b) > 0:
    return abs(cp(b-a, p-a) / abs(b-a))
  else:
    return min(abs(a-p), abs(b-p))
```

### Check if two line segments intersect ($p_1p_2$ and $q_1q_2$)

```
# WARNING UNTESTED - intersection point is tested
def segIntersect(p1, p2, q1, q2):
  o1 = ccw(p1, p2, q1)
  o2 = ccw(p1, p2, q2)
  o3 = ccw(q1, q2, p1)
  o4 = ccw(q1, q2, p2)
  if o1 != o2 and o3 != o4: return True

  # p1, p2 and q1 are collinear and q1 on p1p2
  if o1 == 0 and onSeg(p1, p2, q1): return True

  # p1, p2 and q1 are collinear and q2 on p1p2
  if o2 == 0 and onSeg(p1, p2, q2): return True

  # q1, q2 and p1 are collinear and p1 on q1q2
  if o3 == 0 and onSeg(q1, q2, p1): return True

  # q1, q2 and p2 are collinear and p2 on q1q2
  if o4 == 0 and onSeg(q1, q2, p2): return True

  return False # Doesn't fall in above cases
```

### Intersection point of two line segments ($p_1p_2$ and $q_1q_2$)

```
def segIntersect(p1, p2, q1, q2):
  # Handle special cases for collinear
  if onSeg(p1, p2, q1): return q1
  if onSeg(p1, p2, q2): return q2
  if onSeg(q1, q2, p1): return p1
  if onSeg(q1, q2, p2): return p2
  ip = lineIntersect(p1, p2-p1, q1, q2-q1)
  if onSeg(p1, p2, ip) and onSeg(q1, q2, ip):
    return ip
  else:
    return BAD
```

### Area of polygon (including concave)

```
# points must be in CW or CCW order
def area(P):
  a = 0.0
  for i in range(len(P)):
    a += cp(P[i], P[(i+1)%len(P)])
  return 0.5 * abs(a)
```

### Centroid of polygon (including concave)

```
# points must be in CW or CCW order
def centroid(P):
  c = complex()
  scale = 0.0
  for i in range(len(P)):
    j = (i+1) % len(P)
    x = cp(P[i], P[j])
    c += (P[i] + P[j]) * x
    scale += 3.0 * x
  return c / scale
```

Cut convex polygon with straight line

Returns polygon on left side of $p+\vec{v}t$. Points can be in CW or CCW order. Do not use with duplicate points.

```
pts = []
def cutPolygon(p, v):
  global pts
  P = pts[:] # make a copy
  pts = []
  for i in range(len(P)):
    if cp(v, P[i]-p) > EP: pts.append(P[i])
    pr = P[(i+1)%len(P)]
    ip = lineIntersect(P[i], pr - P[i], p, v)
    if ip != BAD and onSeg(P[i], pr, ip):
      pts.append(ip)
    # remove duplicate points
    while (
      len(pts) >= 2 and
      abs(pts[len(pts)-1] - pts[len(pts)-2]) < EP
    ):
      pts.pop()
```

## 2.2 Convex Hull (Floating Point)

Graham's scan. Complexity: $O(n \log n)$

```
pts = []
def convexHull():
  global pts
  if not pts: return
  fi = 0
  for i in range(len(pts)):
    if (pts[i].imag + EP < pts[fi].imag or
        (abs(pts[i].imag - pts[fi].imag)<EP
        and pts[i].real + EP < pts[fi].real)
    ):
      fi = i
  pts[0], pts[fi] = pts[fi], pts[0]
  def compare(a, b):
    v1, v2 = a - pts[0], b - pts[0]
    a1, a2 = cmath.phase(v1), cmath.phase(v2)
    # Use smaller epsilon for angles
    if abs(a1 - a2) > 1e-14: return a1 - a2
    return abs(v1) - abs(v2)
  pts[1:] = sorted(pts[1:],
            key=functools.cmp_to_key(compare))
  M = 2
  for i in range(2,len(pts)):
    while (M > 1 and
      ccw(pts[M-2], pts[M-1], pts[i]) <= 0
    ):
      M -= 1
    pts[i], pts[M] = pts[M], pts[i]
    M += 1
  if M == 2 and abs(pts[0]-pts[1])<EP: M = 1
  if M < len(pts): pts = pts[:M]
```

Notes:

- All intermediate collinear points and duplicate points are discarded
- If all points are collinear, the algorithm will output the two endpoints of the line
- Works with any number of points including 0, 1, 2
- Works with line segments collinear to the starting point

Example usage

```
pts = [complex(1,1),complex(0,0)] # put all the points in
convexHull()
# pts now contains the convex hull in CCW order
# starting from lowest y point
```

Check if point is within polygon

```
# P must be a convex/concave polygon sorted CCW/CW
def inPolygon(P, p):
  sum = 0.0
  for i in range(len(P)):
    a, b = P[i], P[(i+1)%len(P)]
    # to exclude edges, MUST return False
    if (onSeg(a, b, p)): return True
    sum += cmath.phase((a-p) / (b-p))
  # Use 1e-14 for angle
  return abs(abs(sum) - 2.0*math.pi) < 1e-14
```

## 2.3 Convex Hull (Integer)

Returns convex hull in CCW order starting from lowest $x$ point instead of $y$ point. This method encodes points as pair of integers rather than complex numbers as in the preceding code. Complexity: $O(n \log n)$

Note: Even for floating point, it might be better to use this line sweep method as it is more numerically stable.

```
pts = []

def ccw(a, b, c):
  return ((b[0]-a[0])*(c[1]-a[1]) -
          (b[1]-a[1])*(c[0]-a[0]))

def convexHull():
  global pts
  b, t = [], []
  pts.sort()
  for pt in pts:
    # If you want to keep intermediate collinear
    # points, use < and >
    while (len(b) >= 2 and ccw(b[len(b)-2],
           b[len(b)-1], pt) <= 0): b.pop()
    while (len(t) >= 2 and ccw(t[len(t)-2],
           t[len(t)-1], pt) >= 0): t.pop()
    b.append(pt)
    t.append(pt)
  pts = b
  for i in range(len(t)-2,0,-1):
    pts.append(t[i])
  if len(pts) == 2 and pts[0] == pts[1]: pts.pop()
```

# 3 Graphs

## 3.1 Articulation Points and Bridges

Graph does not need to be connected. Complexity: $O(V + E)$

```
val = [0 for _ in range(n)]
ART = [0 for _ in range(n)]
id = 0

def visit(x, root):
  global id
  stack = [(0, x, root)]
  while stack:
    params = stack.pop()
    if params[0] == 1:
      _, x, root, i, res, child = params
      res = min(res, m)
      if m >= val[x] and not root:
        ART[x] = 1
      # if m > val[x]: (x,adj[x][i]) is a bridge
    else:
      _, x, root = params
      i, res, child = 0, 0, 0
      id += 1
      res, val[x] = id, id
    ok = 1
    for i in range(i, len(adj[x])):
      y = adj[x][i]
      if not val[y]:
        if root:
          child += 1
          if child > 1:
            ART[x] = 1
        stack.append((1, x, root, i+1, res, child))
        stack.append((0, y, 0))
        ok = 0
        break
      else:
        res = min(val[y], res)
    if ok:
      m = res

def articulate():
  global id
  for i in range(n):
    if not val[i]:
      id = i
      visit(i, 1)
```

## 3.2 Bellman-Ford

Consider terminating the loop if no weight was modified in the loop. Complexity: $O(VE)$

```
let weight[V] = all infinity except weight[source] = 0
let parent[V] = all null

loop V-1 times
  for each edge (u,v) with weight w
    if weight[u] + w < weight[v]
      weight[v] = weight[u] + w
      parent[v] = u

# detecting negative weight cycles
for each edge (u,v) with weight w
  if weight[u] + w < weight[v]
  then graph has negative weight cycle
```

## 3.3 Cycle Detection in Directed Graph

We can keep track of vertices currently in recursion stack of function for DFS traversal. If we reach a vertex that is already in the recursion stack, then there is a cycle.

## 3.4 Dijkstra's

Remember to consider using Floyd-Warshall or $O\left(V^2\right)$ version of Dijkstra's. Complexity: $O\left((E + V)\log V\right)$

```
# N = number of nodes
dist = [float('inf') for _ in range(N)]
dja = [(0, START_NODE)]
dist[START_NODE] = 0
while (dja):
  pt = heapq.heappop(dja)
  if pt[0] != dist[pt[1]]: continue
  for ps in adj[pt[1]]:
    if pt[0] + ps[1] < dist[ps[0]]:
      dist[ps[0]] = pt[0] + ps[1]
      heapq.heappush(dja,(dist[ps[0]], ps[0]))
```

## 3.5 Eulerian Cycle

This non-recursive version works with large graphs. Complexity: $O(E)$ for directed graphs or undirected simple graphs using sets, $O(E^2)$ for undirected multigraphs using lists (modify to use dictionary-based multisets to get $O(E)$ for undirected multigraphs)

```
def euler(n):
  global adj, cyc
  stk = [n]
  while stk:
    n = stk[-1]
    if not adj[n]:
      cyc.append(n)
      stk.pop()
    else:
      m = adj[n].pop()
      adj[m].remove(n) # for undirected graphs only
      stk.append(m)
```

Example usage, undirected multigraph:

```
adj = [[], []]
adj[0].append(1); adj[1].append(0) # 0 to 1
adj[0].append(1); adj[1].append(0) # another one
adj[0].append(0); adj[0].append(0) # loop on 0
cyc = []
euler(0) # find eulerian cycle starting from 0
# cyc contains complete cycle including endpoints
# e.g. 0, 0, 1, 0
```

## 3.6 Floyd-Warshall

Negative on diagonal means a vertex is in a negative cycle. Complexity: $O\left(V^3\right)$

```
# let dist[V][V] be initialized to
#   dist[v][v] = 0
#   dist[u][v] = weight of edge else infinity
for k in range(V):
  for i in range(V):
    for j in range(V):
      if dist[i][j] > dist[i][k] + dist[k][j]:
        dist[i][j] = dist[i][k] + dist[k][j]
```

## 3.7 Bipartite Graphs

### 3.7.1 Maximum Bipartite Matching

Complexity: $O(VE)$

```
def bpm(l):
  if visL[l]: return 0
  visL[l] = 1
  for r in adj[l]:
    if matchR[r] == l: continue
    visR[r] = 1
    if matchR[r]<0 or bpm(matchR[r]):
      matchR[r] = l
      return 1
  return 0
```

Example usage:

```
Vleft, Vright = 3, 3
# adj stores the right-side neighbours of left-side vertices
adj = [[1,2],[0,2],[1]]
ans = 0 # cardinality
U = [] # unmatched left vertices
matchR = [-1] * Vright
for l in range(Vleft):
  visL = [0] * Vleft
  visR = [0] * Vright
  if bpm(l): ans += 1
  else: U.append(l)
```

Minimum vertex cover is $(L - Z) \cup (R \cap Z)$

```
visL = [0] * Vleft
visR = [0] * Vright
[bpm(l) for l in U]
left_cover = []
for l in range(Vleft):
  if not visL[l]: left_cover.append(l)
right_cover = []
for r in range(Vright):
  if visR[r]: right_cover.append(r)
```

### 3.7.2 Stable Marriage/Matching

Only tested with equal numbers of men and women.
Complexity: $O(MW)$

```
def stableMarriage():
  M = len(mPref)
  pr = [0] * M
  fm = list(range(M))
  wPartner = [-1] * len(wPref)
  while fm:
    m = fm[-1]
    w = mPref[m][pr[m]]
    pr[m] += 1
    if (wPartner[w] == -1 or wPref[w][m]
    < wPref[w][wPartner[w]]):
      fm.pop()
      if wPartner[w] != -1:
        fm.append(wPartner[w])
      wPartner[w] = m
  return wPartner
```

Example usage:

```
mPref, wPref = [], []

# Man 0 ranks women 2, 0, 1 (best to worst)
mPref.append([2,0,1])

# Woman 0 ranks men 1, 2, 0 (best to worst)
wPref.append([2,0,1])

stableMarriage() # matching is in wPartner
```

### 3.7.3 Notes

- Konig's theorem: In any bipartite graph, the number of edges in a maximum matching equals the number of vertices in a minimum vertex cover.
- The complement of a minimum vertex cover is a maximum independent set.
- For minimum/maximum weight bipartite matching, use min cost max flow or Hungarian algorithm.
- For minimum/maximum weight vertex cover, reduce to minimum cut (max flow).

## 3.8 Max Flow (with Min Cut)

Dinic's algorithm.

Complexity: $O\left(\min\left(V^2E, fE\right)\right)$ where $f$ is the maximum flow

For unit capacities: $O\left(\min\left(V^{2/3}, E^{1/2}\right)E\right)$

For bipartite matching: $O\left(\sqrt{V}E\right)$

```
SOURCE, SINK = 0, 1 # change if necessary

class edge:
  def __init__(self, to, idx, cap):
    self.to, self.idx, self.cap = to, idx, cap
def dfs(n, f):
  global totalflow
  if n == SINK:
    totalflow += f
    return f
  if lvl[n] == lvl[SINK]:
    return 0
  while ptr[n] < len(adj[n]):
    e = adj[n][ptr[n]]
    ptr[n] += 1
    if lvl[e.to] == lvl[n]+1 and e.cap > 0:
      nf = dfs(e.to, min(f, e.cap))
      if nf:
        e.cap -= nf
        adj[e.to][e.idx].cap += nf
        return nf
  return 0
def runMaxFlow():
  global lvl, ptr
  lvl, ptr = [-1] * len(adj), [0] * len(adj)
  lvl[SOURCE] = 0
  bfs = deque() # from collections import deque
  bfs.append(SOURCE)
  while bfs:
    t = bfs[0]
    bfs.popleft()
    for e in adj[t]:
      if lvl[e.to] != -1 or e.cap <= 0: continue
      lvl[e.to] = lvl[t]+1
      bfs.append(e.to)
  if lvl[SINK] == -1: return False
  while dfs(SOURCE, 1<<60): pass
  return True
def initMaxFlow(nodes):
  global totalflow, adj
  totalflow = 0
  adj = [[] for _ in range(nodes)]
def addEdge(a, b, w):
  adj[a].append(edge(b, len(adj[b]), w))
  adj[b].append(edge(a, len(adj[a]) - 1, 0))
```

Example usage

```
initMaxFlow(N) # nodes 0 to N-1
addEdge(0, 3, 123) # 0 to 3 with capacity 123
while runMaxFlow(): pass
# The max flow is now in totalflow
# The min cut: Nodes where lvl[i] == -1 belong to the T
# component, otherwise S
```

## 3.9 Min Cost Max Flow

Edmonds-Karp with Bellman-Ford algorithm. Complexity: $O\left(\min\left(V^2E^2, fVE\right)\right)$ where $f$ is the maximum flow

```
NODES = 101 # maximum number of nodes

def runMCMF(source, sink):
  global totalcost, totalflow
  mf, parent = [0] * NODES, [-1] * NODES
  weight = [1 << 60] * NODES # must be larger than longest path
  weight[source] = 0
  mf[source] = 1 << 60 # value must be larger than max flow
  i, lm = 0, 0
  while i < NODES-1 and lm == i:
    for u in range(NODES):
      for v in adj[u]:
        if not cap[u][v] and not flow[v][u]: continue
        w = -cost[v][u] if flow[v][u] else cost[u][v]
        if weight[u] + w < weight[v]:
          weight[v] = weight[u] + w
          parent[v] = u
          mf[v] = min(mf[u], flow[v][u] if flow[v][u] else cap[u][v])
          lm = i+1
    i += 1
  f = mf[sink]
  if not f: return False
  j = sink
  while j != source:
    p = parent[j]
    if flow[j][p]:
      cap[j][p] += f
      flow[j][p] -= f
    else:
      cap[p][j] -= f
      flow[p][j] += f
    totalcost += f * (weight[j] - weight[p])
    j = p
  totalflow += f
  return True
def initMCMF():
  global totalflow, totalcost, adj, cap, cost, flow
  totalflow, totalcost = 0, 0
  adj = [[] for _ in range(NODES)]
  cap = [[0] * NODES for _ in range(NODES)]
  cost = [[0] * NODES for _ in range(NODES)]
  flow = [[0] * NODES for _ in range(NODES)]
def addEdge(a, b, w, c):
  adj[a].append(b)
  adj[b].append(a) # this line is necessary even without bidirectional edges
  cap[a][b] = w # set cap[b][a] and cost[b][a] to the same to get bidirectional edges
  cost[a][b] = c
```

Example usage

```
initMCMF()
addEdge(0, 3, 123, 5) # adds edge from 0 to 3 with capacity 123 and cost 5
while runMCMF(source, sink): pass
# The max flow is now in totalflow and total cost in totalcost
```

## 3.10 Strongly Connected Components

Tarjan's algorithm. Note that Tarjan's algorithm generates SCCs in reverse topological order, while Kosaraju's algorithm generates in topological order. Complexity: $O(V+E)$

```
def dfs(i):
  global cnt, numScc
  if dfsNum[i] != -1: return
  dfsLow[i], dfsNum[i] = cnt, cnt
  cnt += 1
  sccStack.append(i)
  vis[i] = True
  for j in adj[i]:
    dfs(j)
    if vis[j]:
      dfsLow[i] = min(dfsLow[i], dfsLow[j])
  if dfsLow[i] == dfsNum[i]:
    j = -1
    while i != j:
      j = sccStack.pop()
      vis[j] = False
      sccNum[j] = numScc
    numScc += 1

def scc():
  global cnt, numScc, dfsLow, dfsNum
  global sccNum, vis, sccStack
  N = len(adj)
  cnt, numScc = 0, 0
  dfsLow, dfsNum = [0] * N, [-1] * N
  sccNum, vis, sccStack = [0] * N, [False] * N, []
  for n in range(N): dfs(n)
```

### 3.10.1 2-SAT

```
def VAR(i): return 2*i
def NOT(i): return i^1
def NVAR(i): return NOT(VAR(i))

def addCond(c1, c2):
  adj[NOT(c1)].append(c2)
  adj[NOT(c2)].append(c1)
def init2SAT(numVars):
  global adj, truthValues
  adj = [[] for _ in range(2*numVars)]
  truthValues = [False] * numVars
def run2SAT():
  scc()
  for i in range(0,len(adj),2):
    if sccNum[i] == sccNum[i+1]: return False
    # If SCC is computed with Kosaraju's, use > instead
    truthValues[i//2] = sccNum[i] < sccNum[i+1]
  return True
```

Example usage

```
init2SAT(N) # variables from 0 to N-1
addCond(VAR(4), NVAR(0)) # v4 or not v0
if (run2SAT()):
  # there is a solution
  # truth values are in truthValues[0 to N-1]
```

## 3.11 Tree Algorithms

### 3.11.1 Lowest Common Ancestor

LCA can be solved using RMQ of the dfs tree (using sparse table or segment tree), or the following code using this binary-search like method. Complexity: $O(n\log n)$ preprocessing and $O(\log n)$ per query. Note that the `x.bit_length()` method returns the index of the highest set bit of $x$, which equals $\lfloor \log_2 x \rfloor + 1$.

```
N = 100000                                  # number of vertices
T = [i for i in range(N)]                   # parent of each vertex (parent of root should be itself)
L = [0 for _ in range(N)]                   # depth of each node from root (calculate with dfs or something)
P = [[i] * N.bit_length() for i in range(N)] # P[i][j] is the 2^j parent of i, or root if nonexistent

def lcaBuild():
  for n in range(N.bit_length()):
    for i in range(N):
      P[i][n] = P[P[i][n-1]][n-1] if n else T[i]

def lcaQuery(p, q):
  if L[p] < L[q]: p, q = q, p # ensure p is deeper in tree
  while L[p] > L[q]: p = P[p][(L[p] - L[q]).bit_length()-1] # get p on same level as q
  if (p == q): return p # special case if p/q is the LCA
  for j in range(N.bit_length()-1,-1,-1):
    if P[p][j] != P[q][j]: p, q = P[p][j], P[q][j]
  return T[p]
```

### 3.11.2 Eccentricity

The maximum distances from every vertex are stored in maxd. Complexity: $O(V)$

```
def dfs1(n, p):
  edged[n] = 0
  for m in adj[n]:
    if m != p:
      edged[n] = max(edged[n], 1+dfs1(m, n))
  return edged[n]
def dfs2(n, p, pd):
  maxEdged, nwmg = [pd, 0], 1
  for m in adj[n]:
    if m != p:
      if edged[m] + 1 > maxEdged[0]:
        maxEdged[1] = maxEdged[0]
        maxEdged[0] = edged[m] + 1
        nwmg = 1
      elif edged[m] + 1 == maxEdged[0]:
        nwmg += 1
      elif edged[m] + 1 > maxEdged[1]:
        maxEdged[1] = edged[m] + 1
  for m in adj[n]:
    if m != p:
      npd = maxEdged[0]
      if npd == edged[m] + 1 and nwmg == 1:
        npd = maxEdged[1]
      dfs2(m, n, npd+1)
  maxd[n] = max(pd, maxEdged[0])
```

Example usage (goes through each connected component):

```
edged, maxd = [-1] * N, [0] * N
for n in range(N):
  if (edged[n] == -1):
    dfs1(n, -1)
    dfs2(n, -1, 0)
```

### 3.11.3 Number of times an edge is used in all paths between two vertices

Complexity: $O(V)$

```
# adj stores pairs of (vertex, edge ID)
usage = [0] * E # usage of each edge ID is here
visited = [False] * N
def dfs(n):
  visited[n] = True
  nn = 1
  for pt in adj[n]:
    if visited[pt[0]]: continue
    x = dfs(pt[0])
    usage[pt[1]] += (N - x) * x
    nn += x
  return nn
```

### 3.11.4 Heavy-Light Decomposition

This code provides a basic framework for HLD.

```
nChain, nChainIndex, parent, parentLen, tSize = \
  [[0] * N for _ in range(5)]

def dfs(n, p):
  tSize[n] = 1
  for m in adj[n]:
    if m[0] != p:
      parent[m[0]] = n
      parentLen[m[0]] = m[1]
      dfs(m[0], n)
      tSize[n] += tSize[m[0]]
def hld(n, p):
  nChainIndex[n] = cLength[-1]
  cLength[-1] += 1
  nChain[n] = len(cLength)-1

  # Find largest child
  h = -1
  for m in adj[n]:
    if (m[0] != p and
        (h == -1 or tSize[m[0]] > tSize[h])):
      h = m[0]

  if (h == -1): return
  hld(h, n)

  # Process other children
  for m in adj[n]:
    if (m[0] != p and m[0] != h):
      cLength.append(0)
      cParent.append(n)
      cDepth.append(1+cDepth[nChain[n]])
      hld(m[0], n)
```

Example usage:

```
cDepth = [0]
cLength = [0]
cParent = [-1]
dfs(0, -1)
hld(0, -1)
```

### 3.11.5 Notes

- The diameter of a tree (longest distance between two vertices) can be found by choosing any vertex, then finding a furthest vertex $v_1$, then finding a furthest vertex $v_2$. The distance between $v_1$ and $v_2$ is the diameter, and the centers (1 or 2) are median elements of that path.
- The radius of a tree (longest distance from the best root) is $\left\lfloor \frac{diameter}{2} \right\rfloor$

# 4 Sequences and Strings

## 4.1 AVL Tree

Creating your own binary search tree can be useful in certain situations; e.g. to find the kth element in a set in $O(\log n)$. Note that Python does not have a built-in BST.

```
class node:
  def __init__(self, v):
    self.l = None
    self.r = None
    self.nodes = 1
    self.height = 1
    self.val = v

def height(n): return n.height if n else 0
def nodes(n): return n.nodes if n else 0
def gb(n): return height(n.l)-height(n.r) if n else 0

def updHeight(n):
  n.height = max(height(n.l), height(n.r)) + 1
  n.nodes = nodes(n.l) + nodes(n.r) + 1

def leftRotate(n):
  n.l, n.r = n.r, n.l
  n.l.l, n.l.r, n.r = n.r, n.l.l, n.l.r
  n.val, n.l.val = n.l.val, n.val
  updHeight(n.l)
  updHeight(n)

def rightRotate(n):
  n.r, n.l = n.l, n.r
  n.r.r, n.r.l, n.l = n.l, n.r.r, n.r.l
  n.val, n.r.val = n.r.val, n.val
  updHeight(n.r)
  updHeight(n)

def fix(n):
  if not n: return
  gbn = gb(n)
  if gbn > 1:
    if gb(n.l) < 0: leftRotate(n.l)
    rightRotate(n)
  elif gbn < -1:
    if gb(n.r) > 0: rightRotate(n.r)
    leftRotate(n)
  else:
    updHeight(n)

def insert(n, val):
  if val < n.val:
    if n.l:
      insert(n.l, val)
    else:
      n.l = node(val)
  elif val > n.val:
    if n.r:
      insert(n.r, val)
    else:
      n.r = node(val)
  fix(n)

def predec(n):
  if n.r:
    ret = predec(n.r)
    if n.r.val == None: n.r = None
  else:
    ret = n.val
    if n.l: n.__dict__ = n.l.__dict__
    else: n.val = None
  fix(n)
  return ret

def remove(n, val):
  if not n: return
  if val < n.val: remove(n.l, val)
  elif val > n.val: remove(n.r, val)
  elif n.l: n.val = predec(n.l)
  elif n.r: n.__dict__ = n.r.__dict__
  else: n.val = None
  if n.l and n.l.val == None: n.l = None
  if n.r and n.r.val == None: n.r = None
  fix(n)
```

Example: in-order traversal

```
def inorder(n):
  if not n: return
  inorder(n.l)
  print(n.val)
  inorder(n.r)
```

Example: get kth element in set (zero-based)

```
def kth(n, k):
  if not n: return float('inf')
  if k < nodes(n.l): return kth(n.l, k)
  elif k > nodes(n.l):
    return kth(n.r, k - nodes(n.l) - 1)
  return n.val
```

Example: count number of elements strictly less than $x$

```
def count(n, x):
  if not n: return 0
  if x <= n.val: return count(n.l, x)
  return 1 + nodes(n.l) + count(n.r, x)
```

## 4.2 Aho-Corasick and Trie

The Aho-Corasick string matching algorithm locates elements of a finite set of strings (the “dictionary”) within an input text. Complexity: Linear in length of patterns plus searched text

```
AS = 256 # alphabet size
class node:
  def __init__(self):
    self.match, self.child = -1, [0] * AS
    self.suffix, self.dct = 0, 0

def acMatch(s):
  n = 0
  for i in range(len(s)):
    c = ord(s[i])
    while (n and not nodes[n].child[c]):
      n = nodes[n].suffix
    n = nodes[n].child[c]
    m = n
    while(m):
      if (nodes[m].match >= 0):
        # Replace with whatever you want
        print(f"Matched {nodes[m].match} at {i}")
      m = nodes[m].dct
```

```
def acBuild():
  global nodes
  # Build trie
  nodes = [node()] # create root
  for i in range(len(dct)):
    n = 0
    for c in dct[i]:
      c = ord(c)
      if not nodes[n].child[c]:
        nodes[n].child[c] = len(nodes)
        nodes.append(node())
      n = nodes[n].child[c]
    nodes[n].match = i

  # Build pointers to longest proper suffix and dct
  bfs = deque() # from collections import deque
  bfs.append(0)
  while bfs:
    n = bfs.popleft()
    for i in range(AS):
      if nodes[n].child[i]:
        m, v = nodes[n].child[i], nodes[n].suffix
        while v and not nodes[v].child[i]:
          v = nodes[v].suffix
        s = nodes[v].child[i]
        if s != m:
          nodes[m].suffix = s
          nodes[m].dct = s if nodes[s].match >= 0 \
                         else nodes[s].dct
        bfs.append(m)
```

Example usage:

```
dct = ["bc", "abc"] # do not add duplicates
acBuild()
acMatch("abcabc")
```

Example output:

```
Matched 1 at 2
Matched 0 at 2
Matched 1 at 5
Matched 0 at 5
```

## 4.3 KMP and Z-function

Knuth-Morris-Pratt algorithm. Complexity: $O(m+n)$

This function returns a list containing the zero-based index of the start of each match of K in S. It works with strings, lists, and pretty much any array-indexed data structure that has a length method. Matches may overlap.

In Python 3.6, the `str.find(sub,start,end)` method has complexity $O(m \times n)$ in the worst case. (They improved this to $O(m+n)$ in Python 3.10, but sadly Kattis currently uses PyPy version Python 3.6.9.)

Z-function complexity: $O(n)$. z[i] is the length of the longest common prefix between s and the suffix of s starting at i.

```
def KMP(S, K):
  b = [-1] * (len(K) + 1)
  matches = []

  # Preprocess
  for i in range(1,len(K)+1):
    pos = b[i - 1]
    while pos != -1 and K[pos] != K[i - 1]:
      pos = b[pos]
    b[i] = pos + 1

  # Search
  sp, kp = 0, 0
  while sp < len(S):
    while kp != -1 and (kp == len(K) or K[kp] != S[sp]): kp = b[kp]
    kp += 1
    sp += 1
    if kp == len(K): matches.append(sp - len(K))

  return matches
```

```
def calcZ(s):
  n, l, r = len(s), -1, -1
  z = [0] * n
  for i in range(1,n):
    if i <= r: z[i] = min(z[i-l], r-i+1)
    while i+z[i] < n and s[i+z[i]] == s[z[i]]: z[i] += 1
    if (i+z[i]-1 > r):
      l = i
      r = i+z[i]-1
  return z
```

Example of KMP preprocessing array b[i] and Z-function z[i]:

| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| K[i] | f | i | x | p | f | i | x | f | i | x | p | s | u | f | i | x | \0 |
| b[i] | -1 | 0 | 0 | 0 | 0 | 1 | 2 | 3 | 1 | 2 | 3 | 4 | 0 | 0 | 1 | 2 | 3 |
| z[i] | 0 | 0 | 0 | 0 | 3 | 0 | 0 | 4 | 0 | 0 | 0 | 0 | 0 | 3 | 0 | 0 | |

## 4.4 Longest Common Subsequence

Note that if characters are never repeated in at least one string, LCS can be reduced to LIS. Complexity: $O(nm)$

```
def LCS(A, B):
  dp = [[0] * len(B) for _ in range(len(A))]
  for a in range(len(A)):
    for b in range(len(B)):
      if a: dp[a][b] = max(dp[a][b], dp[a-1][b])
      if b: dp[a][b] = max(dp[a][b], dp[a][b-1])
      if A[a] == B[b]:
        dp[a][b] = max(dp[a][b], (dp[a-1][b-1] if a and b else 0) + 1)
  return dp[-1][-1]
```

## 4.5 Longest Increasing Subsequence

Complexity: $O(n \log k)$ where $k$ is the length of the LIS

```
L = [] # L[x] = smallest end of length x LIS
for x in seq:
  i = bisect.bisect_left(L,x)
  if i == len(L): L.append(x)
  else: L[i] = x
# Length of LIS is len(L)
```

## 4.6 Fenwick Tree / Binary Indexed Tree

This implements a $D$-dimensional Fenwick tree with indexes $[1, N-1]$. Complexity: $O\left(\log^D N\right)$ per operation

```
class FenwickTree:
  def __init__(self, N, D=1):
    self.tree = [0]*(N**D)
    self.N, self.D = N, D
  def __isum(self, ps, n=0, *tail):
    if not n: return self.tree[ps]
    a = 0
    while n:
      a += self.__isum(ps*self.N+n, *tail)
      n -= (n & -n)
    return a
  def __iupd(self, u, ps, n=0, *tail):
    if not n: self.tree[ps] += u; return
    while n < self.N: # TODO: check cond
      self.__iupd(u, ps*self.N+n, *tail)
      n += (n & -n)
  def sum(self, *v):
    return self.__isum(0, *v)
  def upd(self, u, *v):
    return self.__iupd(u, 0, *v)
```

Example usage

```
# create 1D fenwick tree with indexes [1,129]
t = FenwickTree(130)
t.upd(5, 7) # adds 5 to index 7
t.sum(14) # gets sum of all points [1, 14]

# create 3D fenwick tree with indexes [1,129]
t = FenwickTree(130, 3)
t.upd(5, 7, 8, 9) # adds 5 to the point (7, 8, 9)
# get sum of all points [(1, 1, 1), (14, 15, 16)]
t.sum(14, 15, 16)
```

Simple 1D tree (remember, first index is 1)

```
N = 100002
f1, f2 = [0] * N, [0] * N

def sum(f, n):
  a = 0
  while (n):
    a += f[n]
    n -= (n & -n)
  return a
```

```
def upd(f, n, v):
  while n < N:
    f[n] += v
    n += (n & -n)

# only required for range queries
# with range updates
def rsum(n):
  return sum(f1, n) * n - sum(f2, n)
```

To get sum from $[p, q]$:

```
rsum(q) - rsum(p-1)
```

To add $v$ to $[p, q]$:

```
upd(f1, p, v)
upd(f1, q+1, -v)
upd(f2, p, v*(p-1))
upd(f2, q+1, -v*q)
```

## 4.7 Sparse Table

Solves static range min/max query with $O(n \log n)$ preprocessing and $O(1)$ per query. This code does range minimum query.

```
def sptBuild():
  global spt
  spt = [[0]*(N-i).bit_length() for i in range(N)] # spt[N][floor(log2(N-i))+1]
  for n in range(N.bit_length()):
    for i in range(N + 1 - (1 << n)):
      spt[i][n] = min(spt[i][n-1], spt[i+(1<<(n-1))][n-1]) if n else A[i]

def sptQuery(i, j):
  n = (j-i+1).bit_length() - 1 # floor(log2(j-i+1))
  return min(spt[i][n], spt[j+1-(1<<n)][n])
```

Example usage

```
N = 10 # size of array
A = [1, 5, -3, 7, -2, 1, 6, -8, 4, -2]
sptBuild()
sptQuery(0, 9) # returns -8
sptQuery(1, 1) # return 5
sptQuery(1, 4) # returns -3
sptQuery(5, 8) # returns -8
```

## 4.8 Segment Tree

The size of the segment tree should be 4 times the data size. Building is $O(n)$. Querying and updating is $O(\log n)$.

### 4.8.1 Example 1 (no range updates)

This segment tree finds the maximum subsequence sum in an arbitrary range.

```
class node:
  def merge(self, ls, rs):
    self.bestPrefix \
      = max(ls.bestPrefix, ls.sum + rs.bestPrefix)
    self.bestSuffix \
      = max(rs.bestSuffix, rs.sum + ls.bestSuffix)
    self.bestSum \
      = max(ls.bestSuffix + rs.bestPrefix,
        max(ls.bestSum, rs.bestSum))
    self.sum = ls.sum + rs.sum

def segBuild(n, l, r):
  if l == r:
    seg[n].bestPrefix = seg[n].bestSuffix \
      = seg[n].bestSum = seg[n].sum = A[l]
    return
  m = (l+r)//2
  segBuild(2*n+1, l, m)
  segBuild(2*n+2, m+1, r)
  seg[n].merge(seg[2*n+1], seg[2*n+2])

def segQuery(n, l, r, i, j):
  if i <= l and r <= j: return seg[n]
  m = (l+r)//2
  if m < i: return segQuery(2*n+2, m+1, r, i, j)
  if m >= j: return segQuery(2*n+1, l, m, i, j)
  ls = segQuery(2*n+1, l, m, i, j)
  rs = segQuery(2*n+2, m+1, r, i, j)
  a = node()
  a.merge(ls, rs)
  return a

def segUpdate(n, l, r, i):
  if i < l or i > r: return
  if i == l and l == r:
    seg[n].bestPrefix = seg[n].bestSuffix \
      = seg[n].bestSum = seg[n].sum = A[l]
    return
  m = (l+r)//2
  segUpdate(2*n+1, l, m, i)
  segUpdate(2*n+2, m+1, r, i)
  seg[n].merge(seg[2*n+1], seg[2*n+2])
```

### 4.8.2 Example 2 (with range updates)

This segment tree stores a series of booleans and allows swapping all booleans in any range.

```
class node:
  def __init__(self):
    self.inv = False
  def apply(self, x):
    self.sum = x - self.sum
    self.inv = not self.inv
  def split(self, ls, rs, l, m, r):
    if self.inv:
      ls.apply(m-l+1)
      rs.apply(r-m)
      self.inv = False
  def merge(self, ls, rs):
    self.sum = ls.sum + rs.sum

def segQuery(n, l, r, i, j):
  if i <= l and r <= j: return seg[n]
  m = (l+r)//2
  seg[n].split(seg[2*n+1],seg[2*n+2],l,m,r)
  if m < i: return segQuery(2*n+2, m+1, r, i, j)
  if m >= j: return segQuery(2*n+1, l, m, i, j)
  ls = segQuery(2*n+1, l, m, i, j)
  rs = segQuery(2*n+2, m+1, r, i, j)
  a = node()
  a.merge(ls, rs)
  return a

def segUpdate(n, l, r, i, j):
  if i > r or j < l: return
  if i <= l and r <= j:
    seg[n].apply(r-l+1)
    return
  m = (l+r)//2
  seg[n].split(seg[2*n+1],seg[2*n+2],l,m,r)
  segUpdate(2*n+1, l, m, i, j)
  segUpdate(2*n+2, m+1, r, i, j)
  seg[n].merge(seg[2*n+1], seg[2*n+2])
```

Example usage:

```
N = 50000
A = [0]*N
seg = [node() for _ in range(4*N)]
segBuild(0, 0, N-1)
segQuery(0, 0, N-1, i, j) # queries range [i, j]
segUpdate(0, 0, N-1, i, j) # updates range [i, j] (you may need to add parameters)
```

## 4.9 Suffix Array

### 4.9.1 Notes

- Terminating character ($) is not required (unlike CP book), but it is useful to compute the longest common substring of multiple strings
- Use slow version if possible as it is shorter

### 4.9.2 Initialization

Complexity: $O\left(n \log^2 n\right)$

```
def saInit():
  global sa, ra
  l = len(s)
  sa = [i for i in range(l)]
  ra = [ord(i) for i in s]
  k = 1
  while k < l:
    # To use radix sort, replace sort() with:
    # csort(l, k); csort(l, 0)
    sa.sort(key=lambda a: \
      (ra[a], ra[a+k] if a+k < l else -1))
    ra2, x = [0]*l, 0
    for i in range(1,l):
      if (ra[sa[i]] != ra[sa[i-1]] or
        sa[i-1]+k >= l or
        ra[sa[i]+k] != ra[sa[i-1]+k]): x += 1
      ra2[sa[i]] = x
    ra = ra2
    k *= 2
```

### 4.9.3 Initialization (slow)

Complexity: $O\left(n^2 \log n\right)$

```
def saInit():
  global sa
  l = len(s)
  sa = [i for i in range(l)]
  sa.sort(key=lambda a: s[a:])
```

### 4.9.4 Example suffix array

| i | sa[i] | lcp[i] | Suffix |
|---|---|---|---|
| 0 | 0 | 0 | abacabacx |
| 1 | 4 | 4 | abacx |
| 2 | 2 | 1 | acabacx |
| 3 | 6 | 2 | acx |
| 4 | 1 | 0 | bacabacx |
| 5 | 5 | 3 | bacx |
| 6 | 3 | 0 | cabacx |
| 7 | 7 | 1 | cx |
| 8 | 8 | 0 | x |

### 4.9.5 Longest Common Prefix array

Complexity: $O\left(n\right)$

```
def saLCP():
  global lcp
  l = len(s)
  lcp = [0] * l
  p, rsa = [0] * l, [0] * l
  for i in range(l):
    p[sa[i]] = sa[i-1] if i else -1
    rsa[sa[i]] = i
  x = 0
  for i in range(l):
    # Note: The $ condition is optional and is
    # useful for finding longest common substring
    while (p[i] != -1 and p[i]+x < l and
      s[i+x] == s[p[i]+x] and s[i+x] != '$'): x += 1
    lcp[rsa[i]] = x
    if x: x -= 1
```

### 4.9.6 String matching

Returns a vector containing the zero-based index of the start of each match of `m` in `s`. Complexity: $O\left(m \log n\right)$

```
def saFind(m):
  l = len(m)
  lo, hi = 0, len(s)
  while lo < hi:
    mid = (lo + hi) // 2
    if s[sa[mid]:sa[mid]+l] < m:
      lo = mid + 1
    else:
      hi = mid
  occ = []
  for idx in range(lo,len(s)):
    if s[sa[idx]:sa[idx]+l] != m: break
    occ.append(sa[idx])
  occ.sort() # optional
  return occ
```

### 4.9.7 Optional counting sort

Improves `saInit()` performance to $O\left(n \log n\right)$
Usually not necessary, about 4x speed up on a 1M string (timing not tested in Python). However reduces performance in some cases. Not recommended.

```
def csort(l, k):
  global sa
  m = max(300, l+1)
  # VI c(m), sa2(l)
  c, sa2 = [0] * m, [0] * l
  for i in range(l):
    c[ra[i+k]+1 if i+k<l else 0] += 1
  s = 0
  for i in range(m):
    c[i], s = s, c[i]; s += c[i]
  for i in range(l):
    idx = ra[sa[i]+k]+1 if sa[i]+k<l else 0
    sa2[c[idx]] = sa[i]
    c[idx] += 1
  sa = sa2
```

### 4.9.8 Example usage

```
s = "abacabacx"
saInit() # Now sa[] is filled
saLCP() # Now lcp[] is filled
```

# 5 Math and Other Algorithms

## 5.1 Cycle-Finding (Floyd's)

Sequence starts at $x_0$, $x_\mu$ is start of cycle, $\lambda$ is cycle length. Complexity: $O(\mu + \lambda)$

```
t, h, mu, lam = f(X0), f(f(X0)), 0, 1
while t != h: t = f(t); h = f(f(h))
h = X0
while t != h: t = f(t); h = f(h); mu += 1
h = f(t)
while t != h: h = f(h); lam += 1
```

## 5.2 Exponentiation by Squaring

Computes $x^n$. Complexity: $O(\log n)$ assuming multiplication and division are constant time.

```
result = 1
while n:
  if n % 2:
    result *= x
    n-= 1
  x *= x
  n //= 2
```

## 5.3 Extended Euclidean and Modular Inverse

Complexity: $O(\log(\min(a, b)))$

```
def gcd(a, b):
  global x, y
  stack = []
  while b: stack.append((a, b)); a, b = b, a % b
  x = 1; y = 0; d = a
  while stack:
    a, b = stack.pop()
    x -= y * (a // b)
    x, y = y, x
  return d
```

Finds $d = \gcd(a, b)$ and solves the equation $ax + by = d$. The equation $ax + by = c$ has a solution iff $c$ is a multiple of $d = \gcd(a, b)$. If $(x, y)$ is a solution, all other solutions have the form $(x + k\frac{b}{d}, y - k\frac{a}{d}), k \in \mathbb{Z}$.
To get modular inverse of $a$ modulo $m$, do `gcd(a, m)` and the inverse is $x$ (assuming inverse exists).
It is guaranteed that $(x, y)$ is one of the two minimal pairs of Bézout coefficients. $|x| < \frac{b}{d}$ and $|y| < \frac{a}{d}$.

## 5.4 Fast Fourier Transform

Optimized Russian version. Complexity: $O(n \log n)$

```
def fft (a, invert):
  n = len(a)
  j = 0
  for i in range(1,n):
    bit = n >> 1
    while j >= bit:
      j -= bit; bit >>= 1
    j += bit
    if i < j:
      a[i], a[j] = a[j], a[i]
  length = 2
  while length <= n:
    ang = 2*math.pi/length * (-1 if invert else 1)
    wlen = complex(math.cos(ang), math.sin(ang))
    for i in range(0,n,length):
      w = complex(1)
      for j in range(length//2):
        u, v = a[i+j], a[i+j+length//2] * w
        a[i+j] = u + v
        a[i+j+length//2] = u - v
        w *= wlen
    length <<= 1
  if invert:
    for i in range(n):
      a[i] /= n
```

```
def multiply(a, b):
  fa = [complex(ai) for ai in a]
  fb = [complex(bi) for bi in b]
  n = max(len(a),len(b))
  n = 1 << ((n - 1).bit_length() + 1)
  fa += [complex(0) for _ in range(n-len(fa))]
  fb += [complex(0) for _ in range(n-len(fb))]

  fft (fa, False); fft (fb, False)
  for i in range(n):
    fa[i] *= fb[i]
  fft (fa, True)

  return [round(i.real) for i in fa]
```

### 5.4.1 Fast Polynomial Multiplication (Integer)

A bit faster than FFT (not timed in Python). Polynomials must be nonempty arrays in the range $[0, m)$ where $m = 2\,013\,265\,921 = 2^{31} - 2^{27} + 1$, a prime. Negative coefficients are not allowed. Complexity: $O(n \log n)$

```
def transform(a, tA, logN, primitiveRoot):
  tA += [0 for _ in range((1 << logN) - len(tA))]
  for j in range(len(a)):
    k = j << (32 - logN)
    k = ((k >> 1) & 0x55555555) | ((k & 0x55555555) << 1)
    k = ((k >> 2) & 0x33333333) | ((k & 0x33333333) << 2)
    k = ((k >> 4) & 0x0f0f0f0f) | ((k & 0x0f0f0f0f) << 4)
    k = ((k >> 8) & 0x00ff00ff) | ((k & 0x00ff00ff) << 8)
    tA[((k >> 16) | (k << 16)) & MASK] = a[j]
  root = [0] * LOG_MAX_LENGTH
  root[LOG_MAX_LENGTH - 1] = primitiveRoot
  for i in range(LOG_MAX_LENGTH-1, 0, -1):
    root[i - 1] = addMultiply(0, root[i], root[i])
  for i in range(logN):
    twiddle = 1
    for j in range(1 << i):
      for k in range(j, len(tA), 2 << i):
        x = tA[k]
        y = tA[k + (1 << i)]
        tA[k] = addMultiply(x, twiddle, y)
        tA[k + (1 << i)] = \
          addMultiply(x, MODULUS - twiddle, y)
      twiddle = addMultiply(0, root[i], twiddle)
```

```
LOG_MAX_LENGTH = 27
MODULUS = 2013265921
PRIMITIVE_ROOT = 137
PRIMITIVE_ROOT_INVERSE = 749463956
MASK = 0xffffffff

def addMultiply(x, y, z):
  return ((x + y * z) % MODULUS)

def multiply (a, b):
  minN = len(a) - 1 + len(b)
  logN = (minN-1).bit_length()
  tA, tB, nC = [], [], []
  transform(a, tA, logN, PRIMITIVE_ROOT)
  transform(b, tB, logN, PRIMITIVE_ROOT)
  for j in range(len(tA)):
    tA[j] = addMultiply(0, tA[j], tB[j])
  transform(tA, nC, logN, PRIMITIVE_ROOT_INVERSE)

  nInverse = MODULUS - ((MODULUS - 1) >> logN)
  return [addMultiply(0, nInverse, nC[j]) \
          for j in range(minN)]
```

## 5.5 Gauss-Jordan Elimination

This code tries to choose pivots to minimize error.
Complexity: $O\left(N^3\right)$

```
EP = 1e-9
def rref(mat):
  R, C = len(mat), len(mat[0])
  for i in range(min(R,C)):
    rr = i
    for r in range(i,R):
      if mat[r][i] > mat[rr][i]: rr = r
    if abs(mat[rr][i]) < EP: continue
    mat[rr], mat[i] = mat[i], mat[rr]
    for c in range(C-1,i-1,-1):
      mat[i][c] /= mat[i][i]
    for r in range(R):
      if r != i:
        for c in range(C-1,i-1,-1):
          mat[r][c] -= mat[i][c] * mat[r][i]
```

## 5.6 Union-Find Disjoint Sets

Complexity: $O(1)$ per operation. Note: $O(\log n)$ if one of union-by-rank or path compression is omitted

```
# ds[x] is parent of x, and dr[x] is rank
# rank is height of tree without path compression

def findSet(i):
  stack = []
  while ds[i] != i: stack.append(i); i = ds[i]
  while stack: ds[stack.pop()] = i
  return i
def unionSet(i, j):
  x, y = findSet(i), findSet(j)
  if x == y: return # Sometimes necessary if you are
                    # calculating additional info.
  if dr[x] < dr[y]: ds[x] = y
  elif dr[x] > dr[y]: ds[y] = x
  else: ds[x] = y; dr[y] += 1
def sameSet(i, j):
  return findSet(i) == findSet(j)
```

Example initialization:

```
dr = [0] * N
ds = [i for i in range(N)]
```

## 5.7 Sieve, Prime Factorization, Totient

This sieve stores the smallest prime divisor (sp). If the prime factorization of $n$ is $\prod_{i=1}^{k} p_i^{m_i}$, the factoring functions return a sorted list of $(p_i, m_i)$. The number of divisors in $n$ is $\prod_{i=1}^{k}(m_i+1)$, and the sum of all divisors is $\prod_{i=1}^{k} \frac{p_i^{m_i+1}-1}{p_i-1}$. To find Euler's totient function, use product over distinct prime numbers dividing $n$.

$$\varphi(n) = n \prod_{p|n} \left(1 - \frac{1}{p}\right)$$

Sieve: $O(n \log \log n)$

```
typedef vector<pair<int, int>> VP
typedef long long LL
MAX_P = 70000

primes = []
sp = [0] * MAX_P

def sieve():
  for i in range(2,MAX_P):
    if sp[i]: continue
    sp[i] = i
    primes.append(i)
    for j in range(i*i,MAX_P,i):
      if not sp[j]: sp[j] = i
```

Prime factorization: $O\left(\frac{\sqrt{n}}{\log n}\right)$
Works for $n <$ MAX_P$^2$

```
def primeFactorize(n):
  f = []
  for p in primes:
    if p*p > n: break
    a = 0
    while n % p == 0:
      n //= p; a += 1
    if a: f.append((p, a))
  if n != 1: f.append((n, 1))
  return f
```

Prime factorization: $O(\log n)$
Works for $n <$ MAX_P

```
def primeFactorize(n):
  f = []
  while n != 1:
    a, p = 0, sp[n]
    while (n % p == 0):
      n //= p; a += 1
    f.append((p, a))
  return f
```

Pollard's rho algorithm

Inputs: $n$, the integer to be factored and $f(x)$, a pseudo-random function modulo $n$ ($f(x) = x^2 + c$, $c \neq 0$, $c \neq -2$ works fine)

Output: a non-trivial factor of $n$, or failure.

Complexity: $O(n^{1/4})$ for a good choice of pseudo-random function

```
x = 2; y = 2; d = 1
while d == 1:
  x = f(x)
  y = f(f(y))
  d = gcd(abs(x - y), n)
if d == n: return -1 # failure
else: return d
```

Note that this algorithm will return failure for all prime $n$, but it can also fail for composite $n$. In that case, use a different $f(x)$ and try again.

## 5.8 Simplex

Working version (copied from Stanford notebook). Complexity: Exponential in worst case, quite good on average

Two-phase simplex algorithm for solving linear programs. Maximize $c^T x$ subject to $Ax \leq b$ and $x \geq 0$. $A$ should be an $m \times n$ matrix, $b$ should be an $m$-dimensional vector, and $c$ should be an $n$-dimensional vector. The optimal solution will be in vector $x$. It returns the value of the optimal solution (infinity if unbounded above, nan if infeasible).

```
EPS = 1e-9

class LPSolver:

  def __init__(self, A, b, c):
    self.m, self.n = len(b), len(c)
    self.B, self.N = [0] * self.m, [0] * (self.n+1)
    self.D = [[0]*(self.n+2) for _ in range(self.m+2)]
    print(A)
    for i in range(self.m): self.D[i] = A[i][:] + [0]*2
    print(self.D)
    for i in range(self.m):
      self.B[i] = self.n + i
      self.D[i][self.n] = -1; self.D[i][self.n + 1] = b[i]
    for j in range(self.n): self.N[j] = j; self.D[self.m][j] = -c[j]
    self.N[self.n] = -1; self.D[self.m + 1][self.n] = 1
```

```
  def Pivot(self, r, s):
    inv = 1.0 / self.D[r][s]
    for i in range(self.m+2):
      if (i != r):
        for j in range(self.n+2):
          if (j != s):
            self.D[i][j] -= self.D[r][j] * self.D[i][s] * inv
    for j in range(self.n+2):
      if (j != s):
        self.D[r][j] *= inv
    for i in range(self.m+2):
      if (i != r):
         self.D[i][s] *= -inv
    self.D[r][s] = inv
    self.B[r], self.N[s] = self.N[s], self.B[r]

  def Simplex(self, phase):
    x = self.m + 1 if phase == 1 else self.m
    while True:
      s = -1
      for j in range(self.n+1):
        if phase == 2 and self.N[j] == -1: continue
        if (s == -1 or self.D[x][j] < self.D[x][s] or
          self.D[x][j] == self.D[x][s] and self.N[j] < self.N[s]): s = j
      if self.D[x][s] > -EPS: return True
      r = -1
      for i in range(self.m):
        if self.D[i][s] < EPS: continue
        if (r == -1 or self.D[i][self.n + 1] / self.D[i][s] < self.D[r][self.n + 1] / self.D[r][s] or
          (self.D[i][self.n + 1] / self.D[i][s]) == (self.D[r][self.n + 1] / self.D[r][s]) and
          self.B[i] < self.B[r]): r = i
      if r == -1: return False
      self.Pivot(r, s)

  def Solve(self, x):
    r = 0
    for i in range(1, self.m):
      if self.D[i][self.n + 1] < self.D[r][self.n + 1]:
        r = i
    if self.D[r][self.n + 1] < -EPS:
      self.Pivot(r, self.n)
      if not self.Simplex(1) or self.D[self.m + 1][self.n + 1] < -EPS: return -float('inf')
      for i in range(self.m):
        if self.B[i] == -1:
          s = -1
          for j in range(self.n+1):
            if (s == -1 or self.D[i][j] < self.D[i][s] or
              self.D[i][j] == self.D[i][s] and self.N[j] < self.N[s]): s = j
          self.Pivot(i, s)
    if not self.Simplex(2): return float('inf')
    x.extend([0] * self.n)
    for i in range(self.m):
      if (self.B[i] < self.n): x[self.B[i]] = self.D[i][self.n + 1]
    return self.D[self.m][self.n + 1]


def main():

  m, n = 4, 3
  _A = [
    [ 6, -1, 0 ],
    [ -1, -5, 0 ],
    [ 1, 5, 1 ],
    [ -1, -5, -1 ]
  ]
  _b = [ 10, -4, 5, -5 ]
  _c = [ 1, -1, 0 ]

  A = [_A[i][:] for i in range(m)]
  b = _b[:]
  c = _c[:]

  solver = LPSolver(A, b, c)
  x = []
  value = solver.Solve(x)

  print(f"VALUE: {value}") # VALUE: 1.29032
  print("SOLUTION:", *x)
  # SOLUTION: 1.74194 0.451613 1
```

# 6 Tricks for Bit Manipulation

## 6.1 Bit Tricks

| | |
|---|---|
| `(x & -x).bit_length()` | Returns one plus the index of the least significant 1-bit of $x$. Returns 0 if $x = 0$. |
| `x.bit_length()` | Returns one plus the index of the most significant 1-bit of $x$. Returns 0 if $x = 0$. |
| `x.bit_length() - 1` | Returns the number of trailing 0-bits in $x \neq 0$, starting at the most significant bit position. |
| `bin(n).count("1")` | Returns the number of 1-bits in $x$. (Can run slowly for big integers) |
| `bin(n).count("1") & 1` | Returns the parity of $x$, i.e. the number of 1-bits in $x$ modulo 2. |
| `(x & (x - 1)) == 0` | Checks if $x$ is a power of 2 (only one bit set). Note: 0 is edge case. |
| `(x + y - 1) // y` | Finds $\left\lceil \frac{x}{y} \right\rceil$ (positive integers only) |

## 6.2 Lexicographically Next Bit Permutation

Suppose we have a pattern of N bits set to 1 in an integer and we want the next permutation of N 1 bits in a lexicographical sense. For example, if N is 3 and the bit pattern is 00010011, the next patterns would be 00010101, 00010110, 00011001, 00011010, 00011100, 00100011, and so forth. The following is a fast way to compute the next permutation.

```
bs = 0b11111 # whatever is first bit permutation
t = bs | (bs - 1) # t gets v's least significant 0 bits set to 1
# Next set to 1 the most significant bit to change,
# set to 0 the least significant ones, and add the necessary 1 bits.
bs = (t + 1) | (((~t & -~t) - 1) >> (bs & -bs).bit_length())
```

## 6.3 Loop Through All Subsets

For example, if **bs** $= 10110$, loop through **bt** $= 10100, 10010, 10000, 00110, 00100, 00010$

```
bt = (bs-1) & bs
while bt:
  bu = bt ^ bs # contains the opposite subset of bt (e.g. if bt = 10000, bu = 00110)
  bt = (bt-1) & bs
```

# 7 Math Formulas and Theorems

**Arithmetic series**

Arithmetic series: $a_n = a_1 + (n-1)d$

The sum of the first $n$ terms of an arithmetic series is: $S_n = \frac{n(a_1+a_n)}{2} = \frac{n}{2}\left[2a_1 + (n-1)d\right]$

**Catalan numbers (and Super Catalan)**

Starting from $n = 0$, $C_n$: $1, 1, 2, 5, 14, 42, 132, 429, 1430$, $S_n$: $1, 1, 3, 11, 45, 197, 903, 4279$

$$C_0 = 1, C_n = \frac{1}{n+1}\binom{2n}{n} = \sum_{i=0}^{n-1} C_i C_{n-1-i} = \frac{2(2n-1)}{n+1} C_{n-1}, S_n = \frac{1}{n}\left((6n-9)\,S_{n-1} - (n-3)\,S_{n-2}\right)$$

**Chinese remainder theorem**

Suppose $n_1 \cdots n_k$ are positive integers that are pairwise coprime. Then, for any series of integers $a_1 \cdots a_k$, there are an infinite number of solutions $x$ where

$$\begin{cases} x &= a_1 \pmod{n_1} \\ & \cdots \\ x &= a_k \pmod{n_k} \end{cases}$$

All solutions $x$ are congruent modulo $N = n_1 \cdots n_k$.

**Ellipse**

Equation is $\left(\frac{x}{a}\right)^2 + \left(\frac{y}{b}\right)^2 = 1$, area is $\pi ab$, distance from center to either focus is $f = \sqrt{a^2 - b^2}$.

**Euler/Fermat little's theorem**

For any prime $p$ and integer $a$, $a^p \equiv a \pmod{p}$. If $a$ is not divisible by $p$, then $a^{p-1} \equiv 1 \pmod{p}$ and $a^{p-2}$ is the modular inverse of $a$ modulo $p$. More generally, for any coprime $n$ and $a$, $a^{\varphi(n)} \equiv 1 \pmod{n}$. $\varphi(n)$ is Euler's totient function, the number positive integers up to a given integer $n$ that are relatively prime to $n$. If $\gcd(m, n) = 1$, then $\varphi(mn) = \varphi(m)\varphi(n)$ (multiplicative property). For all $n$ and $m$, and $e \geq \log_2(m)$, it holds that $n^e \pmod{m} \equiv n^{\varphi(m)+e \bmod \varphi(m)} \pmod{m}$. Starting from $n = 1$, $\varphi(n)$ values: 1, 1, 2, 2, 4, 2, 6, 4, 6.

**Factoring**

$a^2 - b^2 = (a+b)(a-b)$, $a^3 - b^3 = (a-b)(a^2 + ab + b^2)$, $a^3 + b^3 = (a+b)(a^2 - ab + b^2)$

**Fermat's last theorem**

No three positive integers $a$, $b$, and $c$ can satisfy the equation $a^n + b^n = c^n$ for any integer value of $n$ greater than 2.

**Geometric series**

The sum of the first $n$ terms of a geometric series is: $1 + r + r^2 + \cdots + r^{n-1} = \frac{1-r^n}{1-r}$

If the absolute value of $r$ is less than one, the series converges as $n$ goes to infinity: $\frac{1}{1-r}$

**Great-circle distance**

Let $\phi_1, \lambda_1$ and $\phi_2, \lambda_2$ be the geographical latitude and longitude of two points 1 and 2, and $R$ be sphere radius. This Haversine Formula provides higher numerical precision.

$$a = \sin^2\left(\frac{\phi_2 - \phi_1}{2}\right) + \cos(\phi_1)\cos(\phi_2)\sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right)$$

$$\text{Great-circle distance} = 2 \times R \times \operatorname{atan2}\left(\sqrt{a}, \sqrt{1-a}\right)$$

**Lagrange multiplier**

To maximize/minimize $f(x, y)$ subject to $g(x, y) = 0$, you may be able to set partial derivatives of $\mathcal{L}$ to zero, where $\mathcal{L}(x, y, \lambda) = f(x, y) - \lambda \cdot g(x, y)$

**Sum formulas**

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

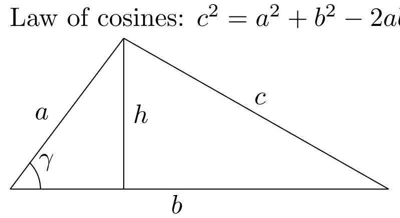
**Triangles**

Area $= \frac{1}{2}bh = \frac{1}{2}ab\sin\gamma = \sqrt{s(s-a)(s-b)(s-c)}$ where $s$ is the semiperimeter

Radius of incircle: $r = \frac{\text{Area}}{s}$

Law of sines: Diameter of circumcircle $= \frac{a}{\sin A} = \frac{b}{\sin B} = \frac{c}{\sin C}$

Law of cosines: $c^2 = a^2 + b^2 - 2ab\cos C$ / $\cos C = \frac{a^2+b^2-c^2}{2ab}$



## 7.1 Dynamic programming

### 7.1.1 Convex hull optimization

Suppose that you have a large set of linear functions $y = m_i x + b_i$. Each query consists of a value of $x$ and asks for the minimum value of $y$ that can be obtained if we select one of the linear functions and evaluate it at $x$. Sort the lines in descending order by slope, and add them one by one. Suppose $l_1$, $l_2$, and $l_3$ are the second line from the top, the line at the top, and the line to be added, respectively. Then, $l_2$ becomes irrelevant if and only if the intersection point of $l_1$ and $l_3$ is to the left of the intersection of $l_1$ and $l_2$. The overall shape of the lines will be concave downwards.

```
# Represent lines as (m, b) where y = mx + b
def ipl(l1, l2):
  return (l1[1] - l2[1]) / (l2[0] - l1[0])

lines = []
# To add a line to the data structure
line = (m, b) # The new line to add
while (len(lines) >= 2 and ipl(line, lines[len(lines)-2]) < ipl(lines[len(lines)-2], lines[len(lines)-1])):
  lines.pop()
lines.append(line)

# To find the lowest point at a certain x
lo, hi = 0, len(lines)
while lo < hi:
  mid = (lo + hi) // 2
  if mid < len(lines)-1 and ipl(lines[mid],lines[mid+1]) < x: lo = mid + 1
  else: hi = mid
```

### 7.1.2 Knuth-Yao optimization

This optimization converts certain $O(n^3)$ DP recurrences into $O(n^2)$.

Convex quadrangle inequality: $f(i,k) + f(j,l) \geq f(i,l) + f(j,k)$ for $i \leq j \leq k \leq l$

Concave quadrangle inequality: $f(i,k) + f(j,l) \leq f(i,l) + f(j,k)$ for $i \leq j \leq k \leq l$

If $A[i][j]$ is the smallest $k$ that gives the optimal answer (or largest $k$, doesn't matter), then $A[i][j-1] \leq A[i][j] \leq A[i+1][j]$. Let cost function $c(i,j) = w(i,j) + \min\left[c(i,k-1) + c(k,j)\right]$ (or max). For maximization, $w$ must be monotone and satisfy convex QI. For minimization, $w$ must be monotone and satisfy concave QI. $f(i,j) = a_i + a_{i+1} + \cdots + a_j$ satisfies both convex and concave QI. $f(i,j) = a_i a_{i+1} \cdots a_j$ satisfies concave QI if all $a_k \geq 1$.

## 7.2 Game theory

Applicable to impartial games under the normal play convention.

- Grundy number: Represents Nim pile size. $g(x) = \min\left(n \geq 0 : n \neq g(y)\,\forall y \in f(x)\right)$
- Remoteness: Moves left if winner forces a win as soon as possible and loser tries to lose as slowly as possible. $r(x) = 0$ if terminal, $1 +$ least even $r(k), k \in f(x)$ if such exists, otherwise $1 +$ greatest odd $r(k), k \in f(x)$.
- Suspense function: Moves left if winner tries to play as long as possible and loser tries to lose as soon as possible. $s(x) = 0$ if terminal, $1 +$ greatest even $r(k), k \in f(x)$ if such exists, otherwise $1 +$ least odd $r(k), k \in f(x)$.

Losing conditions:

- Ordinary sum of games: The player whose turn it is must choose one of the games and make a move in it. A player who is not able to move in all the games loses. $g_1 \oplus g_2 \oplus \cdots \oplus g_n = 0$.
- Union of games: The player whose turn it is must choose at least one of these games and make one move in every chosen one. A player who is not able to move loses. $\forall i\ g_i = 0$.
- Selective compound: The player whose turn it is must choose at least one of these subgames, but he cannot choose all of them and then make one move in every chosen one. A player who is not able to move loses. $g_1 = g_2 = ... = g_n$.
- Conjunctive compound: The player whose turn it is must make a move in every subgame. A player who is not able to move loses. $\min(r_1, r_2, \cdots, r_n)$ is even.
- Continued conjunctive compound: The player whose turn it is must make a move in every subgame he can and the game ends and a player loses only if he cannot move anywhere. $\max(s_1, s_2, \cdots, s_n)$ is even.

## 7.3 Prime numbers

(all primes up to 547, and selected ones thereafter) 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 101 103 107 109 113 127 131 137 139 149 151 157 163 167 173 179 181 191 193 197 199 211 223 227 229 233 239 241 251

257 263 269 271 277 281 283 293 307 311 313 317 331 337 347 349 353 359 367 373 379 383 389 397 401 409 419 421 431 433 439 443 449 457 461 463 467 479 487 491 499 503 509 521 523 541 547 577 607 641 661 701 739 769 811 839 877 911 947 983 1019 1049 1087 1109 1153 1193 1229 1277 1297 1321 1381 1429 1453 1487 1523 1559 1597 1619 1663 1699 1741 1783 1823 1871 1901 1949 1993 2017 2063 2089 2131 2161 2221 2267 2293 2339 2371 2393 2437 2473 2539 2579 2621 2663 2689 2713 2749 2791 2833 2861 2909 2957 3001 3041 3083 3137 3187 3221 3259 3313 3343 3373 3433 3467 3517 3541 3581 3617 3659 3697 3733 3779 3823 3863 3911 3931 4001 4021 4073 4111 4153 4211 4241 4271 4327 4363 4421 4457 4507 4547 4591 4639 4663 4721 4759 4799 4861 4909 4943 4973 5009 5051 5099 5147 5189 5233 5281 5333 5393 5419 5449 5501 5527 5573 5641 5659 5701 5743 5801 5839 5861 5897 5953 6029 6067 6101 6143 6199 6229 6271 6311 6343 6373 6427 6481 6551 6577 6637 6679 6709 6763 6803 6841 6883 6947 6971 7001 7043 7109 7159 7211 7243 7307 7349 7417 7477 7507 7541 7573 7603 7649 7691 7727 7789 7841 7879 13763 19213 59263 77339 117757 160997 287059 880247 2911561 4729819 9267707 9645917 11846141 23724047 39705719 48266341 473283821 654443183 793609931 997244057 8109530161 8556038527 8786201093 9349430521 70635580657 73695102059 79852211099 97982641721 219037536857 273750123551 356369453281 609592207993 2119196502847 3327101349167 4507255137769 7944521201081 39754306037983 54962747021723 60186673819997 98596209151961