Chapter 1

Contest

Template.py

```python
import sys
from collections import *
from itertools import permutations #No repeated elements
sys.setrecursionlimit(10**5)
itr = (line for line in sys.stdin.read().strip().split('\n'))
INP = lambda: next(itr)
def ni(): return int(INP())
def nl(): return [int(_) for _ in INP().split()]



def solve(n,a):
    pass


t = ni()
for case in range(t):
    n = ni()
    a = nl()
    solve(n,a)
```

Troubleshooting: Pre-submit: Write a few simple test cases if sample is not enough. Are time limits close? If so, generate max cases. Is the memory usage fine? Could anything overflow? Make sure to submit the right file.

Wrong answer: Print your solution! Print debug output, as well. Are you clearing all data structures between test cases? Can your algorithm handle the whole range of input? Read the full problem statement again. Do you handle all corner cases correctly? Have you understood the problem correctly? Any uninitialized variables? Any overflows? Confusing N and M, i and j, etc.? Are you sure your algorithm works? What special cases have you not thought of? Are you sure the STL functions you use work as you think? Add some assertions, maybe resubmit. Create some testcases to run your algorithm on. Go through the algorithm for a simple case. Go through this list again. Explain your algorithm to a teammate. Ask the teammate to look at your code. Go for a small walk, e.g. to the toilet. Is your output format correct? (including whitespace) Rewrite your solution from the start or let a teammate do it.

Runtime error: Have you tested all corner cases locally? Any uninitialized variables? Are you reading or writing outside the range of any vector? Any assertions that might fail? Any possible division by 0? (mod 0 for example) Any possible infinite recursion? Invalidated pointers or iterators? Are you using too much memory? Debug with resubmits (e.g. remapped signals, see Various).

Time limit exceeded: Do you have any possible infinite loops? What is the complexity of your algorithm? Are you copying a lot of unnecessary data? (References) How big is the input and output? (consider buffering output) What do your teammates think about your algorithm?

Memory limit exceeded: What is the max amount of memory your algorithm should need? Are you clearing all data structures between test cases?

Chapter 2

Mathematics

2.1 Equations

$$ax^2 + bx + c = 0 \Rightarrow x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

The extremum is given by $x = -b/2a$.

$$\begin{aligned} ax + by &= e \\ cx + dy &= f \end{aligned} \Rightarrow \begin{aligned} x &= \frac{ed - bf}{ad - bc} \\ y &= \frac{af - ec}{ad - bc} \end{aligned}$$

In general, given an equation $Ax = b$, the solution to a variable x_i is given by

$$x_i = \frac{\det A_i'}{\det A}$$

where A_i' is A with the i'th column replaced by b.

2.2 Recurrences

If $a_n = c_1 a_{n-1} + \cdots + c_k a_{n-k}$, and $r_1, \ldots, r_k$ are distinct roots of $x^k + c_1 x^{k-1} + \cdots + c_k$, there are $d_1, \ldots, d_k$ s.t.

$$a_n = d_1 r_1^n + \cdots + d_k r_k^n.$$

Non-distinct roots r become polynomial factors, e.g. $a_n = (d_1 n + d_2) r^n$.

2.3 Trigonometry

$$\sin(v + w) = \sin v \cos w + \cos v \sin w$$
$$\cos(v + w) = \cos v \cos w - \sin v \sin w$$

$$\tan(v + w) = \frac{\tan v + \tan w}{1 - \tan v \tan w}$$
$$\sin v + \sin w = 2 \sin \frac{v + w}{2} \cos \frac{v - w}{2}$$
$$\cos v + \cos w = 2 \cos \frac{v + w}{2} \cos \frac{v - w}{2}$$
$$(V + W) \tan(v - w)/2 = (V - W) \tan(v + w)/2$$

where V, W are lengths of sides opposite angles v, w.

$$a \cos x + b \sin x = r \cos(x - \phi)$$
$$a \sin x + b \cos x = r \sin(x + \phi)$$

where $r = \sqrt{a^2 + b^2}, \phi = \operatorname{atan2}(b, a)$.

2.4 Geometry

2.4.1 Triangles

Side lengths: a, b, c

Semiperimeter: $p = \dfrac{a + b + c}{2}$

Area: $A = \sqrt{p(p - a)(p - b)(p - c)}$

Circumradius: $R = \dfrac{abc}{4A}$

Inradius: $r = \dfrac{A}{p}$

Length of median (divides triangle into two equal-area triangles): $m_a = \frac{1}{2}\sqrt{2b^2 + 2c^2 - a^2}$

Length of bisector (divides angles in two): $s_a =$

$$\sqrt{bc\left[1-\left(\frac{a}{b+c}\right)^2\right]}$$

Law of sines: $\frac{\sin\alpha}{a}=\frac{\sin\beta}{b}=\frac{\sin\gamma}{c}=\frac{1}{2R}$
Law of cosines: $a^2=b^2+c^2-2bc\cos\alpha$
Law of tangents: $\frac{a+b}{a-b}=\frac{\tan\frac{\alpha+\beta}{2}}{\tan\frac{\alpha-\beta}{2}}$

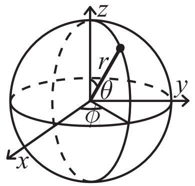
2.4.2 Quadrilaterals

With side lengths a, b, c, d, diagonals e, f, diagonals angle θ, area A and magic flux $F=b^2+d^2-a^2-c^2$:

$$4A=2ef\cdot\sin\theta=F\tan\theta=\sqrt{4e^2f^2-F^2}$$

For cyclic quadrilaterals the sum of opposite angles is $180°$, $ef=ac+bd$, and $A=\sqrt{(p-a)(p-b)(p-c)(p-d)}$.

2.4.3 Spherical coordinates



$$\begin{aligned} x&=r\sin\theta\cos\phi & r&=\sqrt{x^2+y^2+z^2}\\ y&=r\sin\theta\sin\phi & \theta&=\operatorname{acos}(z/\sqrt{x^2+y^2+z^2})\\ z&=r\cos\theta & \phi&=\operatorname{atan2}(y,x)\end{aligned}$$

2.5 Derivatives/Integrals

$$\frac{d}{dx}\arcsin x=\frac{1}{\sqrt{1-x^2}} \qquad \frac{d}{dx}\arccos x=-\frac{1}{\sqrt{1-x^2}}$$
$$\frac{d}{dx}\tan x=1+\tan^2 x \qquad \frac{d}{dx}\arctan x=\frac{1}{1+x^2}$$
$$\int\tan ax=-\frac{\ln|\cos ax|}{a} \qquad \int x\sin ax=\frac{\sin ax-ax\cos ax}{a^2}$$
$$\int e^{-x^2}=\frac{\sqrt{\pi}}{2}\mathrm{erf}(x) \qquad \int xe^{ax}dx=\frac{e^{ax}}{a^2}(ax-1)$$

Integration by parts:

$$\int_a^b f(x)g(x)dx=[F(x)g(x)]_a^b-\int_a^b F(x)g'(x)dx$$

2.6 Sums

$$c^a+c^{a+1}+\cdots+c^b=\frac{c^{b+1}-c^a}{c-1}, c\neq 1$$

$$\begin{aligned} 1+2+3+\cdots+n&=\frac{n(n+1)}{2}\\ 1^2+2^2+3^2+\cdots+n^2&=\frac{n(2n+1)(n+1)}{6}\\ 1^3+2^3+3^3+\cdots+n^3&=\frac{n^2(n+1)^2}{4}\\ 1^4+2^4+3^4+\cdots+n^4&=\frac{n(n+1)(2n+1)(3n^2+3n-1)}{30}\end{aligned}$$

2.7 Series

$$e^x=1+x+\frac{x^2}{2!}+\frac{x^3}{3!}+\ldots,\ (-\infty<x<\infty)$$
$$\ln(1+x)=x-\frac{x^2}{2}+\frac{x^3}{3}-\frac{x^4}{4}+\ldots,\ (-1<x\leq 1)$$
$$\sqrt{1+x}=1+\frac{x}{2}-\frac{x^2}{8}+\frac{2x^3}{32}-\frac{5x^4}{128}+\ldots,\ (-1\leq x\leq 1)$$
$$\sin x=x-\frac{x^3}{3!}+\frac{x^5}{5!}-\frac{x^7}{7!}+\ldots,\ (-\infty<x<\infty)$$
$$\cos x=1-\frac{x^2}{2!}+\frac{x^4}{4!}-\frac{x^6}{6!}+\ldots,\ (-\infty<x<\infty)$$

2.8 Probability theory

Let X be a discrete random variable with probability $p_X(x)$ of assuming the value x. It will then have an expected value (mean) $\mu=\mathbb{E}(X)=\sum_x xp_X(x)$ and variance $\sigma^2=V(X)=\mathbb{E}(X^2)-(\mathbb{E}(X))^2=\sum_x(x-\mathbb{E}(X))^2p_X(x)$ where σ is the standard deviation. If X is instead continuous it will have a probability density function $f_X(x)$ and the sums above will instead be integrals with $p_X(x)$ replaced by $f_X(x)$.

Expectation is linear:

$$\mathbb{E}(aX+bY)=a\mathbb{E}(X)+b\mathbb{E}(Y)$$

For independent X and Y,

$$V(aX+bY)=a^2V(X)+b^2V(Y).$$

2.8.1 Discrete distributions

Binomial distribution

The number of successes in n independent yes/no experiments, each which yields success with probability p is $\mathrm{Bin}(n,p)$, $n=1,2,\ldots$, $0\leq p\leq 1$.

$$p(k)=\binom{n}{k}p^k(1-p)^{n-k}$$
$$\mu=np,\ \sigma^2=np(1-p)$$

$\mathrm{Bin}(n,p)$ is approximately $\mathrm{Po}(np)$ for small p.

First success distribution

The number of trials needed to get the first success in independent yes/no experiments, each wich yields success with probability p is $\mathrm{Fs}(p)$, $0\leq p\leq 1$.

$$p(k)=p(1-p)^{k-1},\ k=1,2,\ldots$$
$$\mu=\frac{1}{p},\ \sigma^2=\frac{1-p}{p^2}$$

Poisson distribution

The number of events occurring in a fixed period of time t if these events occur with a known average rate κ and independently of the time since the last event is $\mathrm{Po}(\lambda)$, $\lambda=t\kappa$.

$$p(k)=e^{-\lambda}\frac{\lambda^k}{k!},k=0,1,2,\ldots$$
$$\mu=\lambda,\ \sigma^2=\lambda$$

2.8.2 Continuous distributions

Uniform distribution

If the probability density function is constant between a and b and 0 elsewhere it is $\mathrm{U}(a,b)$, $a<b$.

$$f(x)=\begin{cases}\frac{1}{b-a} & a<x<b\\ 0 & \text{otherwise}\end{cases}$$
$$\mu=\frac{a+b}{2},\ \sigma^2=\frac{(b-a)^2}{12}$$

Exponential distribution

The time between events in a Poisson process is $\mathrm{Exp}(\lambda)$, $\lambda>0$.

$$f(x)=\begin{cases}\lambda e^{-\lambda x} & x\geq 0\\ 0 & x<0\end{cases}$$
$$\mu=\frac{1}{\lambda},\ \sigma^2=\frac{1}{\lambda^2}$$

Normal distribution

Most real random values with mean μ and variance σ^2 are well described by $\mathcal{N}(\mu, \sigma^2)$, $\sigma > 0$.

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

If $X_1 \sim \mathcal{N}(\mu_1, \sigma_1^2)$ and $X_2 \sim \mathcal{N}(\mu_2, \sigma_2^2)$ then

$$aX_1 + bX_2 + c \sim \mathcal{N}(\mu_1 + \mu_2 + c, a^2\sigma_1^2 + b^2\sigma_2^2)$$

2.9 Markov chains

A *Markov chain* is a discrete random process with the property that the next state depends only on the current state. Let $X_1, X_2, \ldots$ be a sequence of random variables generated by the Markov process. Then there is a transition matrix $\mathbf{P} = (p_{ij})$, with $p_{ij} = \Pr(X_n = i | X_{n-1} = j)$, and $\mathbf{p}^{(n)} = \mathbf{P}^n \mathbf{p}^{(0)}$ is the probability distribution for X_n (i.e., $p_i^{(n)} = \Pr(X_n = i)$), where $\mathbf{p}^{(0)}$ is the initial distribution.

π is a stationary distribution if $\pi = \pi\mathbf{P}$. If the Markov chain is *irreducible* (it is possible to get to any state from any state), then $\pi_i = \frac{1}{\mathbb{E}(T_i)}$ where $\mathbb{E}(T_i)$ is the expected time between two visits in state i. π_j/π_i is the expected number of visits in state j between two visits in state i.

For a connected, undirected and non-bipartite graph, where the transition probability is uniform among all neighbors, π_i is proportional to node i's degree.

A Markov chain is *ergodic* if the asymptotic distribution is independent of the initial distribution. A finite Markov chain is ergodic iff it is irreducible and *aperiodic* (i.e., the gcd of cycle lengths is 1). $\lim_{k\to\infty} \mathbf{P}^k = \mathbf{1}\pi$.

A Markov chain is an A-chain if the states can be partitioned into two sets $\mathbf{A}$ and $\mathbf{G}$, such that all states in $\mathbf{A}$ are absorbing ($p_{ii} = 1$), and all states in $\mathbf{G}$ leads to an absorbing state in $\mathbf{A}$. The probability for absorption in state $i \in \mathbf{A}$, when the initial state is j, is $a_{ij} = p_{ij} + \sum_{k\in\mathbf{G}} a_{ik} p_{kj}$. The expected time until absorption, when the initial state is i, is $t_i = 1 + \sum_{k\in\mathbf{G}} p_{ki} t_k$.

The formula for elements in pascals triangel is

$(n+1)C(r) = (n)C(r-1) + (n)C(r)$

Chapter 3

Graph Algorithms

bfs:

```
# S is index, G is adjacancy list
# finds distance from S to all verticies in G
def bfs(S, G):
    q = [S]
    INF = 10**18
    dist = [INF]*len(G)
    dist[S] = 0
    while q:
        q2 = []
        for u in q:
            for v in G[u]:
                # early break here if only interesed in length of S -> T path.
                if dist[u] + 1 < dist[v]:
                    dist[v] = dist[u] + 1
                    q2.append(v)
        q = q2
    return dist
```

djikstra:

```
from heapq import heappop as pop, heappush as push
# adj: adj-list where edges are tuples (node_id, weight):
# (1) --2-- (0) --3-- (2) has the adj-list:
# adj = [[(1, 2), (2, 3)], [(0, 2)], [0, 3]]
def dijk(adj, S, T):
    N = len(adj)
    INF = 10**18
    dist = [INF]*N
    pq = []
    def add(i, dst):
        if dst < dist[i]:
            dist[i] = dst
            push(pq, (dst, i))
    add(S, 0)

    while pq:
        D, i = pop(pq)
        if i == T: return D
        if D != dist[i]: continue
        for j, w in adj[i]:
            add(j, D + w)

    return dist[T]
```

twoSat:

```
# used in sevenkingdoms, illumination
import sys
sys.setrecursionlimit(10**5)
class Sat:
    def __init__(self, no_vars):
        self.size = no_vars*2
        self.no_vars = no_vars
        self.adj = [[] for _ in range(self.size)]
        self.back = [[] for _ in range(self.size)]
    def add_imply(self, i, j):
        self.adj[i].append(j)
        self.back[j].append(i)
    def add_or(self, i, j):
        self.add_imply(i^1, j)
        self.add_imply(j^1, i)
    def add_xor(self, i, j):
        self.add_or(i, j)
        self.add_or(i^1, j^1)
    def add_eq(self, i, j):
        self.add_xor(i, j^1)

    def dfs1(self, i):
        if i in self.marked: return
        self.marked.add(i)
        for j in self.adj[i]:
            self.dfs1(j)
        self.stack.append(i)

    def dfs2(self, i):
        if i in self.marked: return
        self.marked.add(i)
        for j in self.back[i]:
            self.dfs2(j)
        self.comp[i] = self.no_c

    def is_sat(self):
        self.marked = set()
        self.stack = []
        for i in range(self.size):
            self.dfs1(i)
        self.marked = set()
        self.no_c = 0
        self.comp = [0]*self.size
        while self.stack:
            i = self.stack.pop()
            if i not in self.marked:
                self.no_c += 1
                self.dfs2(i)
        for i in range(self.no_vars):
            if self.comp[i*2] == self.comp[i*2+1]:
                return False
        return True

    # assumes is_sat.
    # If not xi is after xi in topological sort,
    # xi should be FALSE. It should be TRUE otherwise.
    # https://codeforces.com/blog/entry/16205
    def solution(self):
        V = []
        for i in range(self.no_vars):
            V.append(self.comp[i*2] > self.comp[i*2^1])
        return V

if __name__ == '__main__':
    S = Sat(1)
```

```
    S.add_or(0, 0)
    print(S.is_sat())
    print(S.solution())
```

maxflow:

```
from collections import defaultdict
class Dinitz:
    def __init__(self, sz, INF=10**10):
        self.G = [defaultdict(int) for _ in range(sz)]
        self.sz = sz
        self.INF = INF

    def add_edge(self, i, j, w):
        self.G[i][j] += w

    def bfs(self, s, t):
        level = [0]*self.sz
        q = [s]
        level[s] = 1
        while q:
            q2 = []
            for u in q:
                for v, w in self.G[u].items():
                    if w and level[v] == 0:
                        level[v] = level[u] + 1
                        q2.append(v)
            q = q2
        self.level = level
        return level[t] != 0

    def dfs(self, s, t, FLOW):
        if s in self.dead: return 0
        if s == t: return FLOW

        for idx in range(self.pos[s], len(self.adj[s])):
            u = self.adj[s][idx]
            w = self.G[s][u]
            F = self.dfs(u, t, min(FLOW, w))
            if F:
                self.G[s][u] -= F
                self.G[u][s] += F
                if self.G[s][u] == 0:
                    self.pos[s] = idx+1
                    if idx + 1 == len(self.adj[s]):
                        self.dead.add(s)
                return F
            self.pos[s] = idx+1
        self.dead.add(s)
        return 0

    def setup_after_bfs(self):
        self.adj = [[v for v, w in self.G[u].items() if w and self.level[u] + 1 == self.level[v]] for u in range(self.sz)]
        self.pos = [0]*self.sz
        self.dead = set()
    def max_flow(self, s, t):
        flow = 0
        while self.bfs(s, t):
            self.setup_after_bfs()
            while True:
                pushed = self.dfs(s, t, self.INF)
                if not pushed: break
                flow += pushed
        return flow
```

hopcroftCarp:

```
# Hopcroft-Karp bipartite max-cardinality matching and max independent set
# David Eppstein, UC Irvine, 27 Apr 2002
# Used in https://open.kattis.com/problems/cuckoo
def bipartiteMatch(graph):
    '''Find maximum cardinality matching of a bipartite graph (U,V,E).
    The input format is a dictionary mapping members of U to a list
    of their neighbors in V.  The output is a triple (M,A,B) where M is a
    dictionary mapping members of V to their matches in U, A is the part
    of the maximum independent set in U, and B is the part of the MIS in V.
    The same object may occur in both U and V, and is treated as two
    distinct vertices if this happens.'''

    # initialize greedy matching (redundant, but faster than full search)
    matching = {}
    for u in graph:
        for v in graph[u]:
            if v not in matching:
                matching[v] = u
                break

    while 1:
        # structure residual graph into layers
        # pred[u] gives the neighbor in the previous layer for u in U
        # preds[v] gives a list of neighbors in the previous layer for v in V
        # unmatched gives a list of unmatched vertices in final layer of V,
        # and is also used as a flag value for pred[u] when u is in the first layer
        preds = {}
        unmatched = []
        pred = dict([(u,unmatched) for u in graph])
        for v in matching:
            del pred[matching[v]]
        layer = list(pred)

        # repeatedly extend layering structure by another pair of layers
        while layer and not unmatched:
            newLayer = {}
            for u in layer:
                for v in graph[u]:
                    if v not in preds:
                        newLayer.setdefault(v,[]).append(u)
            layer = []
            for v in newLayer:
                preds[v] = newLayer[v]
                if v in matching:
                    layer.append(matching[v])
                    pred[matching[v]] = v
                else:
                    unmatched.append(v)

        # did we finish layering without finding any alternating paths?
        if not unmatched:
            unlayered = {}
            for u in graph:
                for v in graph[u]:
                    if v not in preds:
                        unlayered[v] = None
            return (matching,list(pred),list(unlayered))

        # recursively search backward through layers to find alternating paths
        # recursion returns true if found path, false otherwise
        def recurse(v):
            if v in preds:
                L = preds[v]
                del preds[v]
                for u in L:
                    if u in pred:
                        pu = pred[u]
                        del pred[u]
                        if pu is unmatched or recurse(pu):
                            matching[v] = u
                            return 1
            return 0

        for v in unmatched: recurse(v)
```

HungarianAlgorithm

```
# used on https://open.kattis.com/problems/arboriculture
# G is Bipartite graph N x M (N <= M) where [i][j] is cost to match L[i] and R[j]
# Ported from: https://raw.githubusercontent.com/kth-competitive-programming/kactl/main/content/graph/WeightedMatching.h
# Description: Given a weighted bipartite graph, matches every node on
# the left with a node on the right such that no
# nodes are in two matchings and the sum of the edge weights is minimal. Takes
# cost[N][M], where cost[i][j] = cost for L[i] to be matched with R[j] and
# Returns: (min cost, match), where L[i] is matched with R[match[i]].
# Negate costs for max cost.
# Time: O(N^2M)
#
def hungarian(G):
    INF = 10**18
    if len(G) == 0:
        return 0, []

    n, m = len(G) + 1, len(G[0]) + 1
    u, v, p = [0]*n, [0]*m, [0]*m
    ans = [0]*(n-1)
    for i in range(1, n):
        p[0], j0 = i, 0
        dist, pre = [INF]*m, [-1]*m
        done = [False]*(m+1)
        while True:
            done[j0] = True
            i0, j1, delta = p[j0], 0, INF
            for j in range(1, m):
                if done[j]: continue
                cur = G[i0 - 1][j-1] - u[i0] - v[j]
                if cur < dist[j]:
                    dist[j], pre[j] = cur, j0
                if dist[j] < delta:
                    delta, j1 = dist[j], j
```

```
            for j in range(0, m):
                if done[j]:
                    u[p[j]] += delta
                    v[j] -= delta
                else:
                    dist[j] -= delta
            j0 = j1
            if p[j0] == 0: break
        while j0:
            j1 = pre[j0]
            p[j0] = p[j1]
            j0 = j1
    return -v[0], ans
```

ShortestCycle

```
from collections import *
def shortest_cycle(G):
    ''' Returns the length of shortest cycle even in an undirected graph.
    Floyd Warshall only handles directed graphs,
    but considers an undirected edge to be a cycle of length 2.
    G is adjacency list. '''
    n = len(G)
    ans = 10**18
    INF = 10**9
    for i in range(n):
        dist = [INF] * n
        par = [-1] * n
        dist[i] = 0
        q = deque()
        q.append(i)
        while q:
            x = q[0]
            q.popleft()

            for child in G[x]:
                if dist[child] == INF:
                    dist[child] = 1 + dist[x]

                    par[child] = x
                    q.append(child)

                elif par[x] != child and par[child] != x:
                    ans = min(ans, dist[x] +
                                   dist[child] + 1)
    return ans
```

ConvexHull:

```
def convex_hull(pts):
    pts = sorted(set(pts))

    if len(pts) <= 2:
        return pts

    def cross(o, a, b):
        return (a[0] - o[0]) * (b[1] - o[1]) - (a[1] - o[1]) * (b[0] - o[0])

    lo = []
    for p in pts:
        while len(lo) >= 2 and cross(lo[-2], lo[-1], p) <= 0:
            lo.pop()
        lo.append(p)

    hi = []
    for p in reversed(pts):
        while len(hi) >= 2 and cross(hi[-2], hi[-1], p) <= 0:
            hi.pop()
        hi.append(p)

    return lo[:-1] + hi[:-1]
```

Chapter 4

Data Structures

Segment tree:

```
# Tested on: https://open.kattis.com/problems/supercomputer
class SegmentTree:
    def __init__(self, arr, func=min):
        self.sz = len(arr)
        assert self.sz > 0
        self.func = func
        sz4 = self.sz*4
        self.L, self.R = [None]*sz4, [None]*sz4
        self.value = [None]*sz4
        def setup(i, lo, hi):
            self.L[i], self.R[i] = lo, hi
            if lo == hi:
                self.value[i] = arr[lo]
                return
            mid = (lo + hi)//2
            setup(2*i, lo, mid)
            setup(2*i + 1, mid+1, hi)
            self._fix(i)
        setup(1, 0, self.sz-1)
    def _fix(self, i):
        self.value[i] = self.func(self.value[2*i], self.value[2*i+1])

    def _combine(self, a, b):
        if a is None: return b
        if b is None: return a
        return self.func(a, b)

    def query(self, lo, hi):
        assert 0 <= lo <= hi < self.sz
        return self.__query(1, lo, hi)

    def __query(self, i, lo, hi):
        l, r = self.L[i], self.R[i]
        if r < lo or hi < l:
            return None
        if lo <= l <= r <= hi:
            return self.value[i]
        return self._combine(
            self.__query(i*2, lo, hi),
            self.__query(i*2 + 1, lo, hi)
            )

    def assign(self, pos, value):
        assert 0 <= pos < self.sz
        return self.__assign(1, pos, value)

    def __assign(self, i, pos, value):
        l, r = self.L[i], self.R[i]
        if pos < l or r < pos: return
        if pos == l == r:
            self.value[i] = value
            return
        self.__assign(i*2, pos, value)
        self.__assign(i*2 + 1, pos, value)
        self._fix(i)

    def inc(self, pos, delta):
        assert 0 <= pos < self.sz
        self.__inc(1, pos, delta)

    def __inc(self, i, pos, delta):
        l, r = self.L[i], self.R[i]
        if pos < l or r < pos: return
        if pos == l == r:
            self.value[i] += delta
            return
        self.__inc(i*2, pos, delta)
        self.__inc(i*2 + 1, pos, delta)
        self._fix(i)

    # for indexing - nice to have but not required
    def __setitem__(self, i, v):
        self.assign(i, v)
    def __fixslice__(self, k):
        return slice(k.start or 0, self.sz if k.stop == None else k.stop)
    def __getitem__(self, k):
        if type(k) == slice:
            k = self.__fixslice__(k)
            return self.query(k.start, k.stop - 1)
        elif type(k) == int:
            return self.query(k, k)
```

Fenwick Tree:

```
# Tested on: https://open.kattis.com/problems/froshweek
class FenwickTree: # zero indexed calls!
    # Give array or size!
    def __init__(self, blob):
        if type(blob) == int:
            self.sz = blob
            self.data = [0]*(blob+1)
        elif type(blob) == list:
            A = blob
            self.sz = len(A)
            self.data = [0]*(self.sz + 1)
            for i, a in enumerate(A):
                self.inc(i, a)
    # A[i] = v
    def assign(self, i, v):
        currV = self.query(i, i)
        self.inc(i, v - currV)
    # A[i] += delta
```

```python
    # this method is ~3x faster than doing A[i] += delta
    def inc(self, i, delta):
        i += 1 # (to 1 indexing)
        while i <= self.sz:
            self.data[i] += delta
            i += i&-i # lowest oneBit
    # sum(A[:i+1])
    def sum(self, i):
        i += 1 # (to 1 indexing)
        S = 0
        while i > 0:
            S += self.data[i]
            i -= i&-i
        return S
    # return sum(A[lo:hi+1])
    def query(self, lo, hi):
        return self.sum(hi) - self.sum(lo-1)

    # for indexing - nice to have but not required
    def __fixslice__(self, k):
        return slice(k.start or 0, self.sz if k.stop == None else k.stop)
    def __setitem__(self, i, v):
        self.assign(i, v)
    def __getitem__(self, k):
        if type(k) == slice:
            k = self.__fixslice__(k)
            return self.query(k.start, k.stop - 1)
        elif type(k) == int:
            return self.query(k, k)
```

RMQ:

```python
import math
class RMQ:
    def __init__(self, arr, func=min):
        self.sz = len(arr)
        self.func = func
        MAXN = self.sz
        LOGMAXN = int(math.ceil(math.log(MAXN + 1, 2)))
        self.data = [[0]*LOGMAXN for _ in range(MAXN)]
        for i in range(MAXN):
            self.data[i][0] = arr[i]
        for j in range(1, LOGMAXN):
            for i in range(MAXN - (1<<j)+1):
                self.data[i][j] = func(self.data[i][j-1],
                        self.data[i + (1<<(j-1))][j-1])

    def query(self, a, b):
        if a > b:
            # some default value when query is empty
            return 1
        d = b - a + 1
        k = int(math.log(d, 2))
        return self.func(self.data[a][k], self.data[b-(1<<k)+1][k])
```

Uniion Find:

```python
class UnionFind:
    def __init__(self, N):
        self.parent = [i for i in range(N)]
        self.sz = [1]*N
    def find(self, i):
        path = []
        while i != self.parent[i]:
            path.append(i)
            i = self.parent[i]
        for u in path: self.parent[u] = i
        return i
    def union(self, u, v):
        uR, vR = map(self.find, (u, v))
        if uR == vR: return False
        if self.sz[uR] < self.sz[vR]:
            self.parent[uR] = vR
            self.sz[vR] += self.sz[uR]
        else:
            self.parent[vR] = uR
            self.sz[uR] += self.sz[vR]
        return True
```

Chapter 5

Div

Hungarian algorithm:

```python
# G is Bipartite graph N x M (N <= M) where [i][j] is cost to match L[i] and R[j]
# Description: Given a weighted bipartite graph, matches every node on
# the left with a node on the right such that no
# nodes are in two matchings and the sum of the edge weights is minimal. Takes
# cost[N][M], where cost[i][j] = cost for L[i] to be matched with R[j] and
# Returns: (min cost, match), where L[i] is matched with R[match[i]].
# Negate costs for max cost.
# Time: O(N^2M)
#
def hungarian(G):
    INF = 10**18
    if len(G) == 0:
        return 0, []

    n, m = len(G) + 1, len(G[0]) + 1
    u, v, p = [0]*n, [0]*m, [0]*m
    ans = [0]*(n-1)
    for i in range(1, n):
        p[0], j0 = i, 0
        dist, pre = [INF]*m, [-1]*m
        done = [False]*(m+1)
        while True:
            done[j0] = True
            i0, j1, delta = p[j0], 0, INF
            for j in range(1, m):
                if done[j]: continue
                cur = G[i0 - 1][j-1] - u[i0] - v[j]
                if cur < dist[j]:
                    dist[j], pre[j] = cur, j0
                if dist[j] < delta:
                    delta, j1 = dist[j], j
            for j in range(0, m):
                if done[j]:
                    u[p[j]] += delta
                    v[j] -= delta
                else:
                    dist[j] -= delta
            j0 = j1
            if p[j0] == 0: break
        while j0:
            j1 = pre[j0]
            p[j0] = p[j1]
            j0 = j1
    return -v[0], ans
```

Gauss:

```python
# monoid needs to implement
# __add__, __mul__, __sub__, __div__ and isZ
def gauss(A, b, monoid=None):
    def Z(v): return abs(v) < 1e-6 if not monoid else v.isZ()

    N = len(A[0])
    for i in range(N):
        try:
            m = next(j for j in range(i, N) if Z(A[j][i]) == False)
        except:
            return None #A is not independent!
        if i != m:
            A[i], A[m] = A[m], A[i]
            b[i], b[m] = b[m], b[i]
        for j in range(i+1, N):
            sub = A[j][i]/A[i][i]
            b[j] -= sub*b[i]
            for k in range(N):
                A[j][k] -= sub*A[i][k]

    for i in range(N-1, -1, -1):
        for j in range(N-1, i, -1):
            sub = A[i][j]/A[j][j]
            b[i] -= sub*b[j]
        b[i], A[i][i] = b[i]/A[i][i], A[i][i]/A[i][i]
    return b
```

FFT:

```python
import cmath
# A has to be of length a power of 2.

def FFT(A, inverse=False):
    N = len(A)
    if N <= 1:
        return A
    if inverse:
        D = FFT(A) # d_0/N, d_{N-1}/N, d_{N-2}/N, ...
        return map(lambda x: x/N, [D[0]] + D[:0:-1])
    evn = FFT(A[0::2])
    odd = FFT(A[1::2])
    Nh = N//2
    return [evn[k%Nh]+cmath.exp(2j*cmath.pi*k/N)*odd[k%Nh]
```

```python
            for k in range(N)]

# A has to be of length a power of 2.
def FFT2(a, inverse=False):
    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]

    L = 2
    MUL = -1 if inverse else 1
    while L <= N:
        ang = 2j*cmath.pi/L * MUL
        wlen = cmath.exp(ang)
        for i in range(0, N, L):
            w = 1
            for j in range(L//2):
                u = a[i+j]
                v = a[i+j+L//2] * w
                a[i+j] = u + v
                a[i+j+L//2] = u - v
                w *= wlen
        L *= 2
    if inverse:
        for i in range(N):
            a[i] /= N
    return a

def uP(n):
    while n != (n&-n):
        n += n&-n
    return n

# C[x] = sum_{i=0..N}(A[x-i]*B[i])
def polymul(A, B):
    sz = 2*max(uP(len(A)), uP(len(B)))
    A = A + [0]*(sz - len(A))
    B = B + [0]*(sz - len(B))
    fA = FFT(A)
    fB = FFT(B)
    fAB = [a*b for a, b in zip(fA, fB)]
    C = [x.real for x in FFT(fAB, True)]
    return C
```

Convex Hull:

```python
def convex_hull(pts):
    pts = sorted(set(pts))

    if len(pts) <= 2:
        return pts

    def cross(o, a, b):
        return (a[0] - o[0]) * (b[1] - o[1]) - (a[1] - o[1]) * (b[0] - o[0])

    lo = []
    for p in pts:
        while len(lo) >= 2 and cross(lo[-2], lo[-1], p) <= 0:
            lo.pop()
        lo.append(p)

    hi = []
    for p in reversed(pts):
        while len(hi) >= 2 and cross(hi[-2], hi[-1], p) <= 0:
            hi.pop()
        hi.append(p)

    return lo[:-1] + hi[:-1]
```

Chapter 6

Geometry

Diverse:

```python
import math

# Distance between two points
def dist(p, q):
    return math.hypot(p[0]-q[0], p[1] - q[1])

# Square distance between two points
def d2(p, q):
    return (p[0] - q[0])**2 + (p[1] - q[1])**2

# Converts two points to a line (a, b, c),
# ax + by + c = 0
# if p == q, a = b = c = 0
def pts2line(p, q):
    return (-q[1] + p[1],
           q[0] - p[0],
           p[0]*q[1] - p[1]*q[0])

# Distance from a point to a line,
# given that a != 0 or b != 0
def distl(l, p):
    return (abs(l[0]*p[0] + l[1]*p[1] + l[2])
      /math.hypot(l[0], l[1]))

# intersects two lines.
# if parallell, returnes False.
# lines on format (a, b, c) where ax + by + c == 0
def line_intersection(l1, l2):
    a1,b1,c1 = l1
    a2,b2,c2 = l2
    cp = a1*b2 - a2*b1
    if cp != 0:
        return float(b1*c2 - b2*c1)/cp, float(a2*c1 - a1*c2)/cp
    else:
        return False

# projects a point on a line
def project(l, p):
    a, b, c = l
    return ((b*(b*p[0] - a*p[1]) - a*c)/(a*a + b*b),
        (a*(a*p[1] - b*p[0]) - b*c)/(a*a + b*b))

# Intersections between circles
def circle_intersection(c1, c2):
    if c1[2] > c2[2]:
        c1, c2 = c2, c1
    x1, y1, r1 = c1
    x2, y2, r2 = c2
    if x1 == x2 and y1 == y2 and r1 == r2:
        return False

    dist2 = (x1 - x2)*(x1-x2) + (y1 - y2)*(y1 - y2)
    rsq = (r1 + r2)*(r1 + r2)
    if dist2 > rsq or dist2 < (r1-r2)*(r1-r2):
        return []
    elif dist2 == rsq:
        cx = x1 + (x2-x1)*r1/(r1+r2)
        cy = y1 + (y2-y1)*r1/(r1+r2)
        return [(cx, cy)]
    elif dist2 == (r1-r2)*(r1-r2):
        cx = x1 - (x2-x1)*r1/(r2-r1)
        cy = y1 - (y2-y1)*r1/(r2-r1)
        return [(cx, cy)]

    d = math.sqrt(dist2)
    f = (r1*r1 - r2*r2 + dist2)/(2*dist2)
    xf = x1 + f*(x2-x1)
    yf = y1 + f*(y2-y1)
    dx = xf-x1
    dy = yf-y1
    h = math.sqrt(r1*r1 - dx*dx - dy*dy)
    norm = abs(math.hypot(dx, dy))
    p1 = (xf + h*(-dy)/norm, yf + h*(dx)/norm)
    p2 = (xf + h*(dy)/norm, yf + h*(-dx)/norm)
    return sorted([p1, p2])

# Finds the bisector through origo
# between two points by normalizing.
def bisector(p1, p2):
    d1 = math.hypot(p1[0], p2[1])
    d2 = math.hypot(p2[0], p2[1])
    return ((p1[0]/d1 + p2[0]/d2),
          (p1[1]/d1 + p2[1]/d2))

# Distance from P to origo
def norm(P):
    return (P[0]**2 + P[1]**2 + P[2]**2)**(0.5)

# Finds ditance between point p
# and line A + t*u in 3D
def dist3D(A, u, p):
    AP = tuple(A[i] - p[i] for i in range(3))
    cross = tuple(AP[i]*u[(i+1)%3] - AP[(i+1)%3]*u[i]
        for i in range(3))
    return norm(cross)/norm(u)

def vec(p1, p2):
    return p2[0]-p1[0], p2[1] - p1[1]


def sign(x):
    if x < 0: return -1
    return 1 if x > 0 else 0
```

```python

def cross(u, v):
    return u[0] * v[1] - u[1] * v[0]

def on_segment(p, q, r):
    """Check if point q lies on line segment 'pr'"""
    if (q[0] <= max(p[0], r[0]) and q[0] >= min(p[0], r[0]) and
        q[1] <= max(p[1], r[1]) and q[1] >= min(p[1], r[1])):
        return True
    return False

def is_segment_intersection(s1, s2):
    u = vec(*s1)
    v = vec(*s2)
    p1, p2 = s1
    q1, q2 = s2

    # Calculate cross products
    d1 = cross(u, vec(p1, q1))
    d2 = cross(u, vec(p1, q2))
    d3 = cross(v, vec(q1, p1))
    d4 = cross(v, vec(q1, p2))

    # Check general case
    if d1 != 0 or d2 != 0 or d3 != 0 or d4 != 0:
        return sign(d1) != sign(d2) and sign(d3) != sign(d4)

    # Check collinear case
    return (on_segment(p1, q1, p2) or on_segment(p1, q2, p2) or
            on_segment(q1, p1, q2) or on_segment(q1, p2, q2))
```

Chapter 7

Number theory

Primes:

```python
large_primes = [
5915587277,
1500450271,
3267000013,
5754853343,
4093082899,
9576890767,
3628273133,
2860486313,
5463458053,
3367900313,
1000000000000061,
10**16 + 61,
10**17 + 3
]

def getPrimesBelow(N):
    primes = []
    soll = [1]*N
    for p in range(2, N):
        if soll[p]:
            primes.append(p)
            for k in range(p*p, N, p):
                soll[k] = 0
    return primes

def SieveOfEratosthenes(num):
    prime = [True for i in range(num+1)]
    # boolean array
    p = 2
    out = []
    while (p * p <= num):

        if (prime[p] == True):

            # Updating all multiples of p
            for i in range(p * p, num+1, p):
                prime[i] = False
        p += 1

    for p in range(2, num+1):
        if prime[p]:
            out.append(p)
    return out



def isPrime(N):
    if N < 2: return False
    if N%2 == 0: return N == 2
    mx = min(int(N**.5) + 2, N)
    for i in range(3, mx, 2):
        if N % i == 0: return False
    return True

def genPrimesFrom(N):
    while True:
        if isPrime(N):
            yield N
        N += 1

def getPrimesFrom(N, cnt):
    itr = genPrimesFrom(N)
    return [next(itr) for _ in range(cnt)]

def is_power_of_two(num):
    return num > 0 and (num & (num - 1)) == 0
```

Some useful functions:

```python
import math

# Evaluates to n! / (k! * (n - k)!) when k <= n and evaluates
    to zero when k > n.
# math.comb(n, k) #introduced in python3.8

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

# returns b where (a*b)%MOD == 1
def inv(a, MOD):
    return pow(a, -1, MOD)

# returns g = gcd(a, b), x0, y0,
# where g = x0*a + y0*b
def xgcd(a, b):
    x0, x1, y0, y1 = 1, 0, 0, 1
    while b != 0:
        q, a, b = (a // b, b, a % b)
        x0, x1 = (x1, x0 - q * x1)
        y0, y1 = (y1, y0 - q * y1)
    return (a, x0, y0)

def crt(la, ln):
    assert len(la) == len(ln)
    for i in range(len(la)):
        assert 0 <= la[i] < ln[i]
    prod = 1
    for n in ln:
        assert gcd(prod, n) == 1
        prod *= n
    lN = []
    for n in ln:
        lN.append(prod//n)
    x = 0
    for i, a in enumerate(la):
        print(lN[i], ln[i])
        _, Mi, mi = xgcd(lN[i], ln[i])
        x += a*Mi*lN[i]
    return x % prod

# finds x^e mod m
# Or just pow(x, e, m)
def modpow(x, m, e):
    res = 1
    while e:
        if e%2 == 1:
            res = (res*x) % m
        x = (x*x) % m
        e = e//2
    return res

# Divides a list of digits with an int.
# A lot faster than using bigint-division.
def div(L, d):
    r = [0]*(len(L) + 1)
    q = [0]*len(L)
    for i in range(len(L)):
        x = int(L[i]) + r[i]*10
        q[i] = x//d
        r[i+1] = x-q[i]*d
    s = []
    for i in range(len(L) - 1, 0, -1):
        s.append(q[i]%10)
        q[i-1] += q[i]//10

    while q[0]:
        s.append(q[0]%10)
        q[0] = q[0]//10
    s = s[::-1]
    i = 0
    while s[i] == 0:
        i += 1
    return s[i:]

# Multiplies a list of digits with an int.
# A lot faster than using bigint-multiplication.
```

```
def mul(L, d):
    r = [d*x for x in L]
    s = []
    for i in range(len(r) - 1, 0, -1):
        s.append(r[i]%10)
        r[i-1] += r[i]//10
    while r[0]:
        s.append(r[0]%10)
        r[0] = r[0]//10
    return s[::-1]
```