```python
from typing import TypeVar, List, Tuple, Dict
Numeric = TypeVar('Numeric', int, float, complex)
Reals = TypeVar('Reals', int, float, complex)
Rationals = TypeVar('Rationals', int, float)
Integers = int
Num = int | float
IntList = List[int]
FloatList = List[float]
BoolList = List[float]
NumList = List[Num]
TupleListMST = List[Tuple[Num, int, int]]
EdgeTupleList = List[Tuple[int, int]]
EdgeTypeList = List[Tuple[int, int, int]]


def pairwise_func(seq):
  it = iter(seq)
  next(it)
  return zip(iter(seq), it)


def sort_3_elements(a, b, c):
  if b < a:
    a, b = b, a
  if c < b:
    b, c = c, b
    if b < a:
      a, b = b, a
  return a, b, c


################################
from sys import setrecursionlimit


setrecursionlimit(10000000)  # 10 million should be good enough for most problems


class UnionFindDisjointSets:
  """This Data structure is for non-directional disjoint sets."""

  def __init__(self, n):
    """Attributes declared here must be passed in or global if not used in class format."""
    self.num_sets = n      # optional information
    self.rank = [0] * n    # optional optimization
    self.set_sizes = [1] * n   # optional information
    self.parent = [i for i in range(n)]

  def find_set_recursive(self, u):
    """Recursively find which set u belongs to. Memoize on the way back up.

    Complexity: Time: O(\alpha(n)) -> O(1), inverse ackerman practically constant
          Space: Amortized O(1) stack space
    """
    root_parent = u if self.parent[u] == u else self.find_set_recursive(self.parent[u])
    self.parent[u] = root_parent
    return root_parent

  def find_set(self, x):
    """Iteratively find which set u belongs to. uses stack to memoize.

    Complexity: Time: O(\alpha(n)) -> O(1), inverse ackerman practically constant
          Space: O(\alpha(n)) stack space
    """
    xp, local_stack = x, []
    while xp != self.parent[xp]:
      local_stack.append(xp)
      xp = self.parent[xp]
    for c in local_stack:
      self.parent[c] = xp
    return xp

  def is_same_set(self, u, v):
    """Checks if u and v in same set. TIME and SPACE Complexity is the same as find_set"""
    return self.find_set(u) == self.find_set(v)

  def union_set(self, u, v):
    """Join the set that contains u with the set that contains v.

    Complexity: Time: O(\alpha(n)) -> O(1), inverse ackerman practically constant
          Space: Amortized O(1) stack space
    """
    if not self.is_same_set(u, v):
      u_parent, v_parent = self.find_set(u), self.find_set(v)
      if self.rank[u_parent] > self.rank[v_parent]:   # keep u_parent shorter than v_parent
        u_parent, v_parent = v_parent, u_parent

      if self.rank[u_parent] == self.rank[v_parent]:     # an optional speedup
        self.rank[v_parent] = self.rank[v_parent] + 1
      self.parent[u_parent] = v_parent            # this line joins u with v
      self.num_sets -= 1
      self.set_sizes[v_parent] += self.set_sizes[u_parent]  # u -> v so add size_u to size_v

  def size_of_u(self, u):  # optional information
    """Gives you the size of set u. TIME and SPACE Complexity is the same as find_set"""
    return self.set_sizes[self.find_set(u)]


################################################

"""The following sections are here to support range query operations.
CODE USES ONLY MIN OR SUM QUERIES EXAMPLES
Other functions will most of the time just be replacing this format one of the following
  1.) name_of_variable = f(name_of_variable, data_structure[i])
  2.) name_of_variable = f( data_structure[i], data_structure[k])
  Where f(a, b) = a + b, a * b, gcd(a, b), max(a, b) to list a few examples

Requirements:
  function f(x) must commutative
  able to return the answer i to j
  when listed will support element or range updates
"""


from math import isqrt, gcd


# min use big value, max use smallest val, sum use 0, product use 1
# for gcd and others you will need to change the code so that you set it to the starting value
# this might mean longer implementations but that is the cost of the way I set it up
RQ_DEFAULT = 2 ** 32


class SquareRootDecomposition:
  """Block based query data structure based around square root sizes.

  Operations Supported:
    Update Element at position n.
    Range Query Operation: min, max, gcd, sum, product and commutative functions.
  """

  __slot__ = ("decomposed_blocks", 'size_of_blocks', 'num_of_blocks', 'range_of_data', 'data')

  def __init__(self):
    """Attributes declared here must be passed in or global if not used in class format."""
    self.size_of_blocks = self.num_of_blocks = self.range_of_data = 0
    self.decomposed_blocks = []
    self.data_copy = []

  def prepare_square_root_decomposition(self, new_data):
    """Prepare a new set of data for queries. O(f(x)) is complexity of calling f(x)

    Complexity per call: Time: O(n * O(f(x))), Space: O(n + sqrt(n)).
    """
    sqrt_len = isqrt(len(new_data)) + 1  # + 1 here to for to cover int(x) == floor(x)
    pad_val = RQ_DEFAULT  # DEFAULT INFO AT TOP OF CLASS
    self.size_of_blocks = self.num_of_blocks = sqrt_len
    self.decomposed_blocks, self.range_of_data = [pad_val] * sqrt_len, sqrt_len ** 2
    self.data_copy = [el for el in new_data] + [pad_val] * (self.range_of_data - len(new_data))
    for i in range(sqrt_len):
      self.update_block_n(i)

  def update_block_n(self, block_n):
    """For a given block n, compute the ranged operation on the block.

    Complexity per call: Time: O(sqrt(n) * O(f(x))), Space: (1)
    """
    start, end = block_n * self.size_of_blocks, (block_n + 1) * self.size_of_blocks
```

```
    result = RQ_DEFAULT  # DEFAULT INFO AT TOP OF CLASS
    for i in range(start, end):
      result = min(result, self.data_copy[i])
    self.decomposed_blocks[block_n] = result

  def update_position_i_with_value(self, position_i, value):
    """Update position then follow it up with a block update.

    Complexity per call: Time: O(sqrt(n) * O(f(x))), Space: (1)
    """
    self.data_copy[position_i] = value            # O(1)
    self.update_block_n(position_i // self.num_of_blocks)   # but this costs O(sqrt(n))

  def range_query_i_to_j(self, left, right):
    """Compute a range query from left to right. Note right is inclusive not exclusive here.

    Complexity per call: Time: O(sqrt(n) * O(f(x))), Space: (1)
    """
    block_size, result = self.size_of_blocks, RQ_DEFAULT  # DEFAULT INFO AT TOP OF CLASS
    if (right + 1) - left <= block_size:  # special case where right-left ~= block_size
      for i in range(left, right + 1):  # right + 1 since right is inclusive in this function
        result = min(result, self.data_copy[i])
    else:
      left_block, right_block = left // block_size, right // block_size
      end1, start2 = (left_block + 1) * block_size, right_block * block_size
      for i in range(left, end1):
        result = min(result, self.data_copy[i])
      for i in range(start2, right + 1):  # right + 1 since right is inclusive in this func
        result = min(result, self.data_copy[i])
      for i in range(left_block + 1, right_block):
        result = min(result, self.decomposed_blocks[i])
    return result


################################################


from math import gcd


class SparseTable:
  """Logarithmic Datastructure based around levels of precomputed data.

  Operations Supported:
    DOES NOT SUPPORT DYNAMIC UPDATES much recompute structure after an update or before queries
    Range Query Operation: min, max, gcd, sum, product and commutative functions.
  """
  __slots__ = ("sparse_table", "k_value", "max_n")

  def __init__(self, max_n):
    """Attributes declared here must be passed in or global if not used in class format."""
    self.k_value = max_n.bit_length()
    self.max_n = max_n
    self.sparse_table = [[0] * max_n for _ in range(self.k_value)]

  def prepare_sparse_table(self, array):
    """Prepare a new set of data for queries. O(f(x)) is complexity of calling f(x)

    Complexity per call: Time: O(O(f(x)) * n ln n), Space: O(n ln n), S(n log2 n).
    Variants shown:   f(x) might be O(ln n) like for gcd and lcm
      range min query: O(f(x)) = O(1) like most
      range sum query: O(f(x)) = O(1) like others [is commented out]
    """
    spare_table, i_end, local_max_n = self.sparse_table, self.k_value + 1, self.max_n + 1
    #  spare_table[0] = [el for el in array]  # USE IF max_n == len(array)
    for i, el in enumerate(array):
      spare_table[0][i] = el
    for i in range(1, i_end):  # start from 1 and compute sub problems lvl by lvl
      prev_i = i - 1  # compute once here for better readability and avoid n ln n computations
      j_end, prev_pow_2 = local_max_n - (1 << i), 1 << prev_i  # read comment above
      for j in range(j_end):
        spare_table[i][j] = min(spare_table[prev_i][j], spare_table[prev_i][j + prev_pow_2])
        # spare_table[i][j] = spare_table[prev_i][j] + spare_table[prev_i][j + prev_pow_2]

  def range_query_from_i_to_j(self, left, right):
    """Compute the range query over precomputed data.

    Complexity per call: Time: V1 = O(O(f(x)) * 1), V2 = O(O(f(x)) * ln n), Space: O(1)
    Variants shown:  f(x) might be O(ln n) like for gcd
      range min query: O(f(x)) = O(1) like most  V1
      range sum query: O(f(x)) = O(1) like others  V2
    Notes: V1 is min and max (there might be others) but for these you need only compute
    two ranges so long as lower_bound == left and upper_bound == right, overlap won't matter
    """
    result = RQ_DEFAULT  # see SquareRootDecomposition class for more info
    ind = (right - left + 1).bit_length()-1  # I think this is same as log2(r - l + 1)
    result = min(self.sparse_table[ind][left], self.sparse_table[ind][right - 2**ind + 1])
    for i in range(self.k_value-1, -1, -1):
      if (1 << i) <= right - left + 1:
        result = result + self.sparse_table[i][left]
        left = left + (1 << i)
    return result


################################################


class FenwickTree:
  """Logarithmic Data structure based dynamic use of binary index tree. 1-based indexing.

  Operations Supported:
    Supports Dynamic updates on ranges and elements
    Range Query Operation: main use sum query, other functions exist tho like above
  """
  __slots__ = ("fenwick_tree", "fenwick_tree_size")

  def __init__(self, frequency_array=None):
    """Attributes declared here must be passed in or global if not used in class format."""
    self.fenwick_tree = []
    self.fenwick_tree_size = 0
    if frequency_array:  # None is False, so we can have this optional check :)
      self.build_tree_from_frequency_array(frequency_array)
    # self.num_rows = self.num_cols = 0         # for 2d
    # self.fenwick_tree_2d = [[0] for _ in range(1)]  # for 2d

  def last_set_bit(self, a):
    """Will get the last set-bit of a used in 1-indexing. for 0-indexing use i & (i+1)"""
    return a & (-a)

  def build_tree_from_frequency_array(self, frequency_array):
    """Prepare a new set of data for queries. O(f(x)) is complexity of calling f(x)

    Complexity per call: Time: O(n * O(f(x)) ), T(n) usually, Space: O(n)

    Variant: just call updated on each element Not shown but is like O(n ln n)
    Variants shown:   f(x) might be O(ln n) like for gcd and lcm
      range sum query: O(f(x)) = O(1) like others [is commented out]
    """
    n = len(frequency_array)
    self.fenwick_tree_size = n
    self.fenwick_tree = [0] * (n + 1)
    for i, frequency in enumerate(frequency_array, 1):  # start from 1 in 1 based indexing
      self.fenwick_tree[i] += frequency
      pos = i + self.last_set_bit(i)
      if pos <= n:
        self.fenwick_tree[pos] += self.fenwick_tree[i]

  def build_tree_from_s(self, max_n, s):
    """Builds Fenwick tree given data s."""
    frequency_array = [0] * (max_n + 1)
    for i in s:
      frequency_array[i] += 1
    self.build_tree_from_frequency_array(frequency_array)

  def range_sum_from_i_to_j(self, left, right):
    """Returns the inclusive-exclusive range sum [i...j). version is 1-index based.
    rsq(i, j) = sum(1, right) - sum(1, left - 1)

    Complexity per call: Time: O(log n), Space: O(1).
    """
    if left > 1:  # when left isn't 1 we compute the formula above.
      return self.range_sum_from_i_to_j(1, right) - self.range_sum_from_i_to_j(1, left - 1)
    sum_up_to_right = RQ_DEFAULT  # is 0 but see SquareRootDecomposition class for more info
    while right:
      sum_up_to_right += self.fenwick_tree[right]
      right -= self.last_set_bit(right)
    return sum_up_to_right
```

```
  def update_index_by_delta(self, index, delta):
    """Updates the branch that follows index by delta. version is 1-index based.
    Branch defined as "for i in range(index, tree_size, f(i)=i & (-i))", for(start, end, step)

    Complexity per call: Time: O(log n), Space: O(1).
    """
    while index <= self.fenwick_tree_size:
      self.fenwick_tree[index] += delta
      index += self.last_set_bit(index)

  # def update_index_by_delta_2d(self, row, col, delta):
  #   i, j, fenwick_2d = row, col, self.fenwick_tree_2d
  #   while i <= self.num_rows:
  #     while j <= self.num_cols:
  #       fenwick_2d[i][j], j = fenwick_2d[i][j] + delta, j + self.last_set_bit(j)
  #     j, i = col, i + self.last_set_bit(i)
  #
  # def sum_query_2d(self, row, col):
  #   i, j, result = row, col, 0
  #   while i:
  #     while j:
  #       result, j = result + self.fenwick_tree_2d[i][j], j - self.last_set_bit(j)
  #     j, i = col, i - self.last_set_bit(i)
  #   return result

  def select_k(self, k):  # TODO semi tested
    p = 2**(self.fenwick_tree_size.bit_length()-1)
    i = 0
    while p:
      fen_val = self.fenwick_tree[i + p]
      if k > fen_val:
        k -= fen_val
        i = i + p
      p //= 2
    return i + 1


class RangeUpdatePointQuery:
  def __init__(self, m):
    self.point_update_range_query = FenwickTree([0] * m)

  def update_range_i_to_j_by_delta(self, left, right, delta):  # TODO semi tested
    """For a range update we need only update all indices in [i,i+1...n] to have +delta and
    then all indices in [j+1,j+2...n] with -delta to negate range(j+1, n) that was affected

    Complexity per call: Time: O(log n), Space: O(1).
    """
    self.point_update_range_query.update_index_by_delta(left, delta)
    self.point_update_range_query.update_index_by_delta(right + 1, -delta)

  def point_sum_query_of_index(self, index):  # TODO semi tested
    """With restriction of only point sum query, will return range_sum(1, index).

    Complexity per call: Time: O(log n), Space: O(1).
    """
    return self.point_update_range_query.range_sum_from_i_to_j(1, index)


class RangeUpdateRangeQuery:
  def __init__(self, m):
    self.range_update_point_query = RangeUpdatePointQuery(m)
    self.point_update_range_query = FenwickTree([0] * m)

  def update_range_i_to_j_by_delta(self, left, right, delta):  # TODO semi tested
    """Updates range by delta like above but then also updates the augmented tree which is
    needed for dynamic range updates and queries for the same structure.

    Complexity per call: Time: O(log n), Space: O(1).
    """
    self.range_update_point_query.update_range_i_to_j_by_delta(left, right, delta)
    self.point_update_range_query.update_index_by_delta(left, delta * (left - 1))
    self.point_update_range_query.update_index_by_delta(right + 1, -delta * right)

  def range_sum_from_i_to_j(self, left, right):  # TODO semi tested
    """Similar to the Fenwick tree we use inclusion exclusion, but we need to use our augmented
    tree to handle 3 cases to compute formula sum[0, i] = sum(rupq, i) * i - sum(purq, i)
          | original            | simplified      |   conditions
    sum[0,i] =  | 0*i - 0             | 0           |   i < left
          | x*i - x(left - 1)       | x(i - (left - 1))   |   left <= i <= right
          | 0*i - (x(left - 1) - x*right) | x(right - left + 1)   |   i > right

    Complexity per call: Time: O(log n), Space: O(1).
    """
    if left > 1:
      return self.range_sum_from_i_to_j(1, right) - self.range_sum_from_i_to_j(1, left - 1)
    return (self.range_update_point_query.point_sum_query_of_index(right) * right
            - self.point_update_range_query.range_sum_from_i_to_j(1, right))


################################################


class SegmentTree:
  def __init__(self, new_array):
    self.is_lazy = []
    self.lazy = []
    self.segment_tree = []
    self.array_a = []
    self.tree_size = 0
    self._version_1(new_array)
    # self._version_2(new_array)

  def _version_1(self, new_array):
    n = len(new_array)
    next_2_pow_n = n if 2**(n.bit_length()-1) == n else 2**n.bit_length()
    self.lazy = [-1] * (4 * next_2_pow_n)     # 4 * n used to avoid index out of bounds
    self.segment_tree = [0] * (4 * next_2_pow_n)  # 4 * n used to avoid index out of bounds
    self.array_a = [el for el in new_array]
    if next_2_pow_n != n:                     # we add extra padding to the end to
      self.array_a.extend([RQ_DEFAULT]*(next_2_pow_n-n))  # make the len a power of 2
    self.tree_size = len(self.array_a)
    self.build_segment_tree_1()

  def _version_2(self, new_array):
    n = len(new_array)
    self.is_lazy = [False] * (4 * n)
    self.lazy = [0] * (4 * n)
    self.segment_tree = [0] * (4 * n)
    self.array_a = [el for el in new_array]
    self.tree_size = n
    self.build_segment_tree_2()

  def left_child(self, parent):
    """Macro function, gets left child in array based binary tree, paste in code for speedup."""
    return parent << 1

  def right_child(self, parent):
    """Macro function, gets right child in array based binary tree, paste in code to speedup."""
    return (parent << 1) + 1

  def conqur(self, a, b):
    """Macro function, handles to conqur conditional, paste in code for speedup."""
    return b if a == -1 else (a if b == -1 else min(a, b))

  def propagate(self, parent, left, right):
    lazy_p = self.lazy[parent]
    if lazy_p != -1:
      self.segment_tree[parent] = lazy_p
      if left != right:
        self.lazy[self.left_child(parent)] = self.lazy[self.right_child(parent)] = lazy_p
      else:
        self.array_a[left] = lazy_p
      self.lazy[parent] = -1

  def _build_segment_tree_1(self, parent, left, right):
    if left == right:
      self.segment_tree[parent] = self.array_a[left]
    else:
      mid = (left + right) // 2
      left_path, right_path = self.left_child(parent), self.right_child(parent)
      self._build_segment_tree_1(left_path, left, mid)
      self._build_segment_tree_1(right_path, mid + 1, right)
      left_val, right_val = self.segment_tree[left_path], self.segment_tree[right_path]
      self.segment_tree[parent] = self.conqur(left_val, right_val)
      # seg_tree = self.segment_tree  # uncomment if self.segment_tree gets too costly
      # seg_tree[parent] = self.conqur(seg_tree[left_path], seg_tree[right_path])

  def _build_segment_tree_2(self, parent, left, right):
```

```
    if left == right:
      self.segment_tree[parent] = left
    else:
      mid = (left + right) // 2
      left_path, right_path = self.left_child(parent), self.right_child(parent)
      self._build_segment_tree_2(left_path, left, mid)
      self._build_segment_tree_2(right_path, mid + 1, right)
      left_ind = self.segment_tree[left_path]
      right_ind = self.segment_tree[right_path]
      self.segment_tree[parent] = (left_ind if (self.array_a[left_ind]
                                                <= self.array_a[right_ind]) else right_ind)

  def _range_min_query_1(self, parent, left, right, i, j):
    self.propagate(parent, left, right)
    if i > j:
      return -1
    if i <= left and j >= right:
      return self.segment_tree[parent]
    mid = (left + right) // 2
    left_min = self._range_min_query_1(self.left_child(parent), left, mid,
                                       i, min(mid, j))
    right_min = self._range_min_query_1(self.right_child(parent), mid + 1, right,
                                        max(i, mid + 1), j)
    return self.conqur(left_min, right_min)

  def _range_min_query_2(self, parent, left, right, i, j):
    if right < i or j < left:
      return -1, -1
    if self.is_lazy[parent]:
      return i, self.lazy[parent]
    if i <= left and j >= right:
      return self.segment_tree[parent], self.array_a[self.segment_tree[parent]]
    mid = (left + right) // 2
    left_ind, left_val = self._range_min_query_2(self.left_child(parent), left, mid, i, j)
    right_ind, right_val = self._range_min_query_2(self.right_child(parent), mid+1, right, i, j)
    if left_ind == -1:
      return right_ind, right_val
    elif right_ind == -1:
      return left_ind, left_val
    else:
      return (left_ind, left_val) if left_val <= right_val else (right_ind, right_val)

  def _update_segment_tree_1(self, parent, left, right, i, j, value):
    self.propagate(parent, left, right)
    if i > j:
      return
    elif i <= left and j >= right:
      self.lazy[parent] = value
      self.propagate(parent, left, right)
    else:
      mid = (left + right) // 2
      left_path, right_path = self.left_child(parent), self.right_child(parent)
      self._update_segment_tree_1(left_path, left, mid, i, min(mid, j), value)
      self._update_segment_tree_1(right_path, mid + 1, right, max(i, mid + 1), j, value)
      left_subtree = (self.lazy[left_path] if self.lazy[left_path] != -1
                      else self.segment_tree[left_path])
      right_subtree = (self.lazy[right_path] if self.lazy[right_path] != -1
                       else self.segment_tree[right_path])
      self.segment_tree[parent] = (self.segment_tree[left_path]
                                   if left_subtree <= right_subtree
                                   else self.segment_tree[right_path])

  def _update_segment_tree_2(self, parent, left, right, i, j, new_value):
    if right < i or j < left:
      return self.segment_tree[parent]
    if i == left and j == right:
      if i == j:
        self.array_a[left] = new_value
        self.is_lazy[parent] = False
      else:
        self.lazy[parent] = new_value
        self.is_lazy[parent] = True
      self.segment_tree[parent] = i
      return i
    mid = (left + right) // 2
    left_path, right_path = self.left_child(parent), self.right_child(parent)
    if self.is_lazy[parent]:
      self.is_lazy[parent] = False
      self.is_lazy[left_path] = self.is_lazy[right_path] = True
      self.lazy[left_path] = self.lazy[right_path] = self.lazy[parent]
      self.segment_tree[left_path] = left
      self.segment_tree[right_path] = mid
    left_ind = self._update_segment_tree_2(left_path, left, mid,
                                           max(i, left), min(j, mid), new_value)
    right_ind = self._update_segment_tree_2(right_path, mid+1, right,
                                            max(i, mid+1), min(j, right), new_value)
    self.segment_tree[parent] = (left_ind if (self.array_a[left_ind] <= self.array_a[right_ind])
                                 else right_ind)
    return self.segment_tree[parent]

  def build_segment_tree_1(self):
    self._build_segment_tree_1(1, 0, self.tree_size - 1)

  def build_segment_tree_2(self):
    self._build_segment_tree_2(1, 0, self.tree_size - 1)

  def update_segment_tree_1(self, i, j, value):
    self._update_segment_tree_1(1, 0, self.tree_size - 1, i, j, value)

  def update_segment_tree_2(self, i, j, value):
    self._update_segment_tree_2(1, 0, self.tree_size - 1, i, j, value)

  def range_min_query_1(self, i, j):
    return self._range_min_query_1(1, 0, self.tree_size - 1, i, j)

  def range_min_query_2(self, i, j):
    return self._range_min_query_2(1, 0, self.tree_size - 1, i, j)[0]


################################################

# from math import log2
from collections import deque
from heapq import heappush, heappop, heapify
from sys import setrecursionlimit

setrecursionlimit(10000000)  # 10 million should be good enough for most contest problems


INF: int = 2**31
# turn these into enums later
UNVISITED: int = -1
EXPLORED: int = -2
VISITED: int = -3
# turn these into enums later
TREE: int = 0
BIDIRECTIONAL: int = 1
BACK: int = 2
FORWARD: int = 3


class Graph:
  def __init__(self, v: int, e: int, r=None, c=None):
    self.num_edges: int = e
    self.num_nodes: int = v
    self.num_rows: int = r
    self.num_cols: int = c

    self.adj_list = []
    self.adj_list_trans = []  # for topological sort
    self.adj_matrix = []
    self.edge_list = []
    self.grid = []

    self.data_to_code = {}  # used for converting input into integer vertices
    self.code_to_data = []  # used for integer vertices back into original input

  def convert_data_to_code(self, data: object) -> int:
    """Converts data to the form: int u | 0 <= u < |V|, stores (data, u) pair, then return u."""
    if data not in self.data_to_code:
      self.data_to_code[data] = len(self.code_to_data)
      self.code_to_data.append(data)  # can be replaced with a count variable if space needed
    return self.data_to_code[data]

  def add_edge_u_v_wt_into_directed_graph(self, u: object, v: object, wt: Num, data: Num):
    """A pick and choose function will convert u, v into index form then add it to the structure
    you choose.
    """
```

```
    u: int = self.convert_data_to_code(u)  # omit if u,v is in the form: int u | 0 <= u < |V|
    v: int = self.convert_data_to_code(v)  # omit if u,v is in the form: int u | 0 <= u < |V|

    self.adj_list[u].append(v)
    # self.adj_list[u].append((v, wt))  # Adjacency list usage with weights
    self.adj_matrix[u][v] = wt      # Adjacency matrix usage
    self.edge_list.append((wt, u*self.num_nodes + v))   # Edge list usage space optimized
    # the following lines come as a pair-set used in max flow algorithm and are used in tandem.
    self.edge_list.append([v, wt, data])
    self.adj_list[u].append(len(self.edge_list) - 1)

  def add_edge_u_v_wt_into_undirected_graph(self, u: object, v: object, wt: Num, data: Num):
    """undirected graph version of the previous function. wt can be omitted if not used."""
    self.add_edge_u_v_wt_into_directed_graph(u, v, wt, data)
    self.add_edge_u_v_wt_into_directed_graph(v, u, wt, data)

  def fill_grid_graph(self, new_grid: list[list[object]]):
    self.num_rows = len(new_grid)
    self.num_cols = len(new_grid[0])
    self.grid = [[self.convert_data_to_code(el) for el in row] for row in new_grid]


class GraphAlgorithms:
  def __init__(self, new_graph):
    self.path_i_j = []
    self.graph: Graph = new_graph
    self.min_spanning_cost: int = 0
    self.dfs_counter: int = 0
    self.dfs_root: int = 0
    self.root_children: int = 0
    self.region_num: int = 0

    self.dir_rc = [(1, 0), (0, 1), (-1, 0), (0, -1)]
    self.visited = []
    self.mst_node_list = []
    self.dist = []
    self.topo_sort_node_list = []
    self.parent = []
    self.low_values = []
    self.articulation_nodes = []
    self.bridge_edges = []
    self.directed_edge_type = []  # change last int to enum sometime
    self.component_region = []
    self.decrease_finish_order = []
    self.nodes_on_stack = []
    self.node_state = []  # change to enum type sometime
    self.bipartite_colouring = []
    self.last = []

  def flood_fill_via_dfs(self, row: int, col: int, old_val: object, new_val: object):
    """Computes flood fill graph traversal via recursive depth first search. Use on grid graphs.

    Complexity: Time: O(|V| + |E|), Space: O(|V|): for grids usually |V|=row*col and |E|=4*|V|
    More uses: Region Colouring, Connectivity, Area/island Size, misc
    Input
      row, col: integer pair representing current grid position
      old_val, new_val: unexplored state, the value of an explored state
    """
    self.graph.grid[row][col] = new_val
    for row_mod, col_mod in self.dir_rc:
      new_row, new_col = row + row_mod, col + col_mod
      if (0 <= new_row < self.graph.num_rows
          and 0 <= new_col < self.graph.num_cols
          and self.graph.grid[new_row][new_col] == old_val):
        self.flood_fill_via_dfs(new_row, new_col, old_val, new_val)

  def flood_fill_via_bfs(self, start_row: int, start_col: int, old_val: object, new_val: object)
      :
    """Computes flood fill graph traversal via breadth first search. Use on grid graphs.

    Complexity: Time: O(|V| + |E|), Space: O(|V|): for grids usually |V|=row*col and |E|=4*|V|
    More uses: previous uses tplus shortest connected pah
    """
    queue = deque([(start_row, start_col)])
    while queue:
      row, col = queue.popleft()
      for row_mod, col_mod in self.dir_rc:
        new_row, new_col = row + row_mod, col + col_mod
        if (0 <= new_row < self.graph.num_rows
            and 0 <= new_col < self.graph.num_cols
            and self.graph.grid[new_row][new_col] == old_val):
          self.graph.grid[new_row][new_col] = new_val
          queue.append((new_row, new_col))

  def min_spanning_tree_via_kruskals_and_heaps(self):  # tested
    """Computes mst of graph G stored in edge_list, space optimized via heap.

    Complexity per call: Time: O(|E|log |V|), Space: O(|E|) + Union_Find
    More uses: finding min spanning tree
    Variants: min spanning subgraph and forrest, max spanning tree, 2nd min best spanning tree
    Optimization: We use a heap to make space comp. O(|E|). instead of O(|E|log |E|)
    when using sort, however edge_list is CONSUMED. Also uses space optimization u*V + u = u, v
    """
    vertices = self.graph.num_nodes
    heapify(self.graph.edge_list)
    ufds = UnionFindDisjointSets(vertices)
    min_spanning_tree = []
    while self.graph.edge_list and ufds.num_sets > 1:  # num_sets == 1 means graph connected
      wt, uv = heappop(self.graph.edge_list)   # uv is u*num_nodes + v
      v, u = divmod(uv, vertices)         # u, v = uv//n, uv%n
      if not ufds.is_same_set(u, v):
        min_spanning_tree.append((wt, u, v))
        ufds.union_set(u, v)
    self.mst_node_list = min_spanning_tree

  def prims_visit_adj_matrix(self, source: int):
    """Find min weight edge in adjacency matrix implementation of prims.

    Complexity per call: Time: O(|V|^2), T(|V| * 4|V|), Space: O(|V|), S(~5|V|)
    """
    vertices = self.graph.num_nodes
    not_seen_and_min_dist, mst_parent = {v: INF for v in range(vertices)}, [-1] * vertices
    min_spanning_tree, min_weight, not_seen_and_min_dist[source] = [], 0, 0
    for _ in range(vertices):
      u, best_dist = min(not_seen_and_min_dist.items(), key=lambda x: x[1])
      # if best_dist == INF:  # graph is disconnected case
      #   return      # uncomment when graph might be disconnected
      min_spanning_tree.append((min(u, mst_parent[u]), max(u, mst_parent[u])))
      min_weight += best_dist
      del not_seen_and_min_dist[u]  # remove u since it has been seen now
      for v, min_dist in not_seen_and_min_dist.items():
        if self.graph.adj_matrix[u][v] < min_dist:
          not_seen_and_min_dist[v], mst_parent[v] = self.graph.adj_matrix[u][v], u
    # min_spanning_tree.sort()  # optional for when edges need to be ordered: nlog n cost
    self.mst_node_list = min_spanning_tree[1:]  # cut off the root node

  def min_spanning_tree_via_prims_adj_list(self, source: int):
    """Computes mst of graph G stored in adj_list. Uses edge compression u, v = u*|V| + v.

    Complexity: Time: O(|E|log |V|), Space: O(|E|)
    More uses: Gets a different min spamming tree than kruskal's
    """
    vertices = self.graph.num_nodes
    not_visited, mst_best_dist = [True] * vertices, [INF] * vertices
    heap = [(wt, v * vertices + source) for v, wt in self.graph.adj_list[source]]
    heapify(heap)
    for v, wt in self.graph.adj_list[source]:
      mst_best_dist[v] = wt
    min_spanning_tree, nodes_taken, msc, not_visited[source] = [], 0, 0, False
    while heap and nodes_taken < vertices - 1:  # max edges in mst == |V| - 1
      edge_wt, u_parent = heappop(heap)
      u, parent = divmod(u_parent, vertices)  # edge uncompress u, v = uv//|V|, uv%|V|
      if not_visited[u]:
        min_spanning_tree.append((min(u, parent), max(u, parent)))
        msc += edge_wt
        nodes_taken += 1
        not_visited[u] = False
        for v, wt in self.graph.adj_list[u]:
          if wt < mst_best_dist[v] and not_visited[v]:
            mst_best_dist[v] = wt
            heappush(heap, (wt, v * vertices + u))  # edge compression v*|V| + u.
    # min_spanning_tree.sort()  # optional for when edges need to be ordered: nlog n cost
    self.mst_node_list = min_spanning_tree
    self.min_spanning_cost = msc if nodes_taken == vertices - 1 else -1

  def breadth_first_search_vanilla_template(self, source: int):
    """Template for distance based bfs traversal from node source.
```

```
  Complexity per call: Time: O(|V| + |E|), Space: O(|V|)
  More uses: connectivity, shortest path on monotone weighted graphs
  """
  distance = [UNVISITED] * self.graph.num_nodes
  queue, distance[source] = deque([source]), 0
  while queue:
    u = queue.popleft()
    for v in self.graph.adj_list[u]:
      if distance[v] == UNVISITED:
        distance[v] = distance[u] + 1
        queue.append(v)
  self.dist = distance

def topology_sort_via_tarjan_helper(self, u: int):
  """Recursively explore unvisited graph via dfs.

  Complexity per call: Time: O(|V|), Space: O(|V|) at deepest point
  """
  self.visited[u] = VISITED
  for v in self.graph.adj_list[u]:
    if self.visited[v] == UNVISITED:
      self.topology_sort_via_tarjan_helper(v)
  self.topo_sort_node_list.append(u)

def topology_sort_via_tarjan(self, source=-1):
  """Compute a topology sort via tarjan method, on adj_list.

  Complexity per call: Time: O(|V| + |E|), Space: O(|V|)
  More Uses: produces a DAG, topology sorted graph, build dependencies
  """
  self.visited = [UNVISITED] * self.graph.num_nodes
  self.topo_sort_node_list = []
  if source != -1:
    self.topology_sort_via_tarjan_helper(source)
  else:
    for u in range(self.graph.num_nodes):
      if self.visited[u] == UNVISITED:
        self.topology_sort_via_tarjan_helper(u)
  self.topo_sort_node_list = self.topo_sort_node_list[::-1]

def topology_sort_via_kahns(self):
  """Compute a topology sort via kahn's method, on adj_list.

  Complexity per call: Time: O(|E|log|V|), Space: O(|V|)
  More uses: different ordering as tarjan's method
  bonus: heaps allow for custom ordering (i.e. use lowest indices first)
  """
  in_degree = [0] * self.graph.num_nodes
  for list_of_u in self.graph.adj_list:
    for v in list_of_u:
      in_degree[v] += 1
  topo_sort = []
  heap = [u for u, el in enumerate(in_degree) if el == 0]
  heapify(heap)
  while heap:
    u = heappop(heap)
    topo_sort.append(u)
    for v in self.graph.adj_list[u]:
      in_degree[v] -= 1
      if in_degree[v] <= 0:
        heappush(heap, v)
  self.topo_sort_node_list = topo_sort

def amortized_heap_fix(self, heap):
  """Should we need |V| space this will ensure that while still being O(log|V|)"""
  tmp = [-1] * self.graph.num_nodes
  for wt, v in heap:
    if tmp[v] == -1:
      tmp[v] = wt
  heap = [(wt, v) for v, wt in enumerate(tmp) if wt != -1]
  heapify(heap)

def single_source_shortest_path_dijkstras(self, source: int, sink: int):
  """It is Dijkstra's pathfinder using heaps.

  Complexity per call: Time: O(|E|log |V|), Space: O(|V|)
  More uses: shortest path on state based graphs
  Input:
    source: can be a single nodes or list of nodes
    sink: the goal node
  """
  distance, parents = [INF] * self.graph.num_nodes, [UNVISITED] * self.graph.num_nodes
  distance[source], parents[source] = 0, source
  heap = [(0, source)]
  while heap:
    cur_dist, u = heappop(heap)
    if distance[u] < cur_dist:
      continue
    # if u == sink: return cur_dist # uncomment this line for fast return
    for v, wt in self.graph.adj_list[u]:
      if distance[v] > cur_dist + wt:
        distance[v] = cur_dist + wt
        parents[v] = u
        heappush(heap, (distance[v], v))
  self.dist = distance
  self.parent = parents

def all_pairs_shortest_path_floyd_warshall_print(self, i, j):  # TODO RETEST
  """Generate the shortest path from i to j, used with Floyd Warshall."""
  path = []
  while i != j:
    path.append(j)
    j = self.parent[i][j]
  self.path_i_j = path[::-1]  # we need to reverse it ???

def all_pairs_shortest_path_floyd_warshall(self):
  """Computes essentially a matrix operation on a graph.

  Complexity per call: Time: O(|V|^3), Space: O(|V|^2)
  More uses: Shortest path, Connectivity.
  Variants: Transitive closure, Maximin and Minimax path, Cheapest negative cycle,
        Finding diameter of a graph, Finding SCC of a directed graph.
  """
  matrix, matrix_size = self.graph.adj_matrix, self.graph.num_nodes
  parents = [[i] * matrix_size for i in range(matrix_size)]  # optional for path i to j
  for k in range(matrix_size):
    for i in range(matrix_size):
      for j in range(matrix_size):
        matrix[i][j] = min(matrix[i][j], matrix[i][k] + matrix[k][j])
        # if matrix[i][k] + matrix[k][j] < matrix[i][j]:  # for printing path i to j
        #   matrix[i][j], parents[i][j] = matrix[i][k] + matrix[k][j], parents[k][j]
  self.parent = parents

def apsp_floyd_warshall_variants(self):
  """Compressed Compilation of 5 variants of APSP. PICK AND CHOOSE IMPLEMENTATION.
  Contents:  # var 2 and 4 are untested right now
    Variant 1: Transitive Closure to check if i is directly or indirectly connected to j.
    Variant 2: MiniMax and MaxiMin path problem. A[i][j] = INF if no edge exists
    Variant 3: Cheapest/Negative cycle, From APSP
    Variant 4: Diameter of a Graph, biggest shortest path in the graph, From APSP
    Variant 5: SCC on small graph, labeling and existence path from i->j and j->i exists

  Complexity per call: Time: O(V^3), T(V^3 + V^2), Space: O(V^2)
  """
  matrix, matrix_size = self.graph.adj_matrix, self.graph.num_nodes
  graph_diameter = 0
  for k in range(matrix_size):
    for i in range(matrix_size):
      for j in range(matrix_size):
        matrix[i][j] |= matrix[i][k] and matrix[k][j]             # Variant 1
        matrix[i][j] = min(matrix[i][j], max(matrix[i][k], matrix[k][j]))   # Variant 2
        if matrix[j][j] < 0 and matrix[k][j] < INF and matrix[j][i] < INF:  # Variant 3
          matrix[k][i] = -INF                       # Variant 3
        # following only requires loops i and j, Variant 4 IS LINE BELOW THIS COMMENT
        graph_diameter = max(graph_diameter, matrix[i][j] if matrix[i][j] < INF else 0)
        is_j_strongly_connected_to_i = matrix[i][j] and matrix[j][i]  # Variant 5

def articulation_point_and_bridge_helper_via_dfs(self, u: int):
  """Recursion part of the dfs. It kind of reminds me of how Union find works.

  Complexity per call: Time: O(|E|), Space: O(|V|) or O(|E|) depending on points vs edges.
  """
  self.visited[u] = self.dfs_counter
  self.low_values[u] = self.visited[u]
  self.dfs_counter += 1
  for v in self.graph.adj_list[u]:
    if self.visited[v] == UNVISITED:
      self.parent[v] = u
```

```
      if u == self.dfs_root:
        self.root_children += 1
      self.articulation_point_and_bridge_helper_via_dfs(v)
      if self.low_values[v] >= self.visited[u]:
        self.articulation_nodes[u] = True
        if self.low_values[v] > self.visited[u]:
          self.bridge_edges.append((u, v))
      self.low_values[u] = min(self.low_values[u], self.low_values[v])
    elif v != self.parent[u]:
      self.low_values[u] = min(self.low_values[u], self.visited[v])

def articulation_points_and_bridges_via_dfs(self):
  """Generates the name on an adj_list based graph.

  Complexity per call: Time: O(|E| + |V|), Space: O(|V|)
  More uses: finding the sets of single edge and vertex removals that disconnect the graph.
  Bridges stored as edges and points are True values in articulation_nodes.
  """
  self.dfs_counter = 0
  for u in range(self.graph.num_nodes):
    if self.visited[u] == UNVISITED:
      self.dfs_root = u
      self.root_children = 0
      self.articulation_point_and_bridge_helper_via_dfs(u)
      self.articulation_nodes[self.dfs_root] = (self.root_children > 1)

def cycle_check_on_directed_graph_helper(self, u: int):
  """Recursion part of the dfs. It is modified to list various types of edges.

  Complexity per call: Time: O(|E|), Space: O(|V|) at deepest call
  More uses: listing edge types: Tree, Bidirectional, Back, Forward/Cross edge. On top of
  listing Explored, Visited, and Unvisited.
  """
  self.visited[u] = EXPLORED
  for v in self.graph.adj_list[u]:
    edge_type: int = TREE
    if self.visited[v] == UNVISITED:
      edge_type = TREE
      self.parent[v] = u
      self.cycle_check_on_directed_graph_helper(v)
    elif self.visited[v] == EXPLORED:
      edge_type = BIDIRECTIONAL if v == self.parent[u] else BACK  # case graph is not DAG
    elif self.visited[v] == VISITED:
      edge_type = FORWARD
    self.directed_edge_type.append((u, v, edge_type))
  self.visited[u] = VISITED

def cycle_check_on_directed_graph(self):
  """Determines if a graph is cyclic or acyclic via dfs.

  Complexity per call: Time: O(|E| + |V|),
            Space: O(|E|) if you label each edge O(|V|) otherwise.
  More uses: Checks if graph is acyclic(DAG) which can open potential for efficient algorithms
  """
  self.visited = [UNVISITED] * self.graph.num_nodes
  self.directed_edge_type = []  # can be swapped out for a marked variable if checking for DAG
  for u in range(self.graph.num_nodes):
    if self.visited[u] == UNVISITED:
      self.cycle_check_on_directed_graph_helper(u)

def strongly_connected_components_of_graph_kosaraju_helper(self, u: int, pass_one: bool):
  """Pass one explore G and build stack, Pass two mark the SCC regions on transposition of G.
  # TODO RETEST
  Complexity per call: Time: O(|E| + |V|), Space: O(|V|)
  """
  self.visited[u] = VISITED
  self.component_region[u] = self.region_num
  neighbours: IntList = self.graph.adj_list[u] if pass_one else self.graph.adj_list_trans[u]
  for v in neighbours:
    if self.visited[v] == UNVISITED:
      self.strongly_connected_components_of_graph_kosaraju_helper(v, pass_one)
  if pass_one:
    self.decrease_finish_order.append(u)

def strongly_connected_components_of_graph_kosaraju(self):  # TODO RETEST
  """Marks the SCC of a directed graph using Kosaraju's method.

  Complexity per call: Time: O(|E| + |V|), Space: O(|V|)
  More Uses: Labeling and Identifying SCC regions(marks regions by numbers).
  """
  self.visited = [UNVISITED] * self.graph.num_nodes
  self.component_region = [0] * self.graph.num_nodes
  for u in range(self.graph.num_nodes):
    if self.visited[u] == UNVISITED:
      self.strongly_connected_components_of_graph_kosaraju_helper(u, True)
  self.visited = [UNVISITED] * self.graph.num_nodes
  self.region_num = 1
  for u in reversed(self.decrease_finish_order):
    if self.visited[u] == UNVISITED:
      self.strongly_connected_components_of_graph_kosaraju_helper(u, False)
      self.region_num += 1

def strongly_connected_components_of_graph_tarjans_helper(self, u: int):  # TODO RETEST
  """Recursive part of tarjan's, pre-order finds the SCC regions, marks regions post-order.

  Complexity per call: Time: O(|E| + |V|), Space: O(|V|)
  """
  self.low_values[u] = self.node_state[u] = self.dfs_counter
  self.dfs_counter += 1
  self.nodes_on_stack.append(u)
  self.visited[u] = VISITED
  for v in self.graph.adj_list[u]:
    if self.node_state[v] == UNVISITED:
      self.strongly_connected_components_of_graph_tarjans_helper(v)
    if self.visited[v] == VISITED:
      self.low_values[u] = min(self.low_values[u], self.low_values[v])
  if self.low_values[u] == self.node_state[u]:
    self.region_num += 1
    while True:
      v: int = self.nodes_on_stack.pop()
      self.visited[v], self.component_region[v] = UNVISITED, self.region_num
      if u == v:
        break

def strongly_connected_components_of_graph_tarjans(self):  # TODO RETEST
  """Marks the SCC regions of a directed graph using tarjan's method.

  Complexity per call: Time: O(|E| + |V|), Space: O(|V|)
  More uses: Labeling and Identifying SCC regions(marks regions by numbers).
  """
  max_v = self.graph.num_nodes
  self.visited, self.node_state = [UNVISITED] * max_v, [UNVISITED] * max_v
  self.low_values, self.component_region = [0] * max_v, [0] * max_v
  self.nodes_on_stack, self.region_num, self.dfs_counter = [], 0, 0
  for u in range(self.graph.num_nodes):
    if self.node_state[u] == UNVISITED:
      self.strongly_connected_components_of_graph_tarjans_helper(u)

def bipartite_check_on_graph_helper(self, source: int, color: IntList):  # TODO RETEST
  """Uses bfs to check if the graph region connected to source is bipartite.

  Complexity per call: Time: O(|E| + |V|), Space: O(|V|)
  """
  queue = deque([source])
  color[source] = 0
  is_bipartite = True
  while queue and is_bipartite:
    u = queue.popleft()
    for v in self.graph.adj_list[u]:
      if color[v] == UNVISITED:
        color[v] = not color[u]
        queue.append(v)
      elif color[v] == color[u]:
        is_bipartite = False
        break
  return is_bipartite

def bipartite_check_on_graph(self):  # TODO RETEST
  """Checks if a graph has the bipartite property.

  Complexity per call: Time: O(|E| + |V|), Space: O(|V|)
  More Uses: check bipartite property, labeling a graph for 2 coloring if it is bipartite.
  """
  is_bipartite, color = True, [UNVISITED] * self.graph.num_nodes
  for u in range(self.graph.num_nodes):
    if color[u] == UNVISITED:
      is_bipartite = is_bipartite and self.bipartite_check_on_graph_helper(u, color)
      if not is_bipartite:
```

```
        break
    self.bipartite_colouring = color if is_bipartite else None

  def max_flow_find_augmenting_path_helper(self, source: int, sink: int):  # TODO RETEST NEW
    """Will check if augmenting path in the graph from source to sink exists via bfs.

    Complexity per call: Time: O(|E| + |V|), Space O(|V|)
    Input
      source: the node which we are starting from.
      sink: the node which we are ending on.
    """
    distance, parents = [-1] * self.graph.num_nodes, [(-1, -1)] * self.graph.num_nodes
    queue, distance[source] = deque([source]), 0
    while queue:
      u = queue.popleft()
      if u == sink:
        self.dist, self.parent = distance, [el for el in parents]
        return True
      for idx in self.graph.adj_list[u]:
        v, cap, flow = self.graph.edge_list[idx]
        if cap - flow > 0 and distance[v] == -1:
          distance[v] = distance[u] + 1
          parents[v] = (u, idx)
          queue.append(v)
    self.dist, self.parent = [], []
    return False

  def send_flow_via_augmenting_path(self, source: int, sink: int, flow_in: Num):  # TODO RETEST
    """Function to recursively emulate sending a flow. returns min pushed flow.

    Complexity per call: Time: O(|V|), Space O(|V|)
    Uses: preorder finds the min pushed_flow post order mutates edge_list based on that flow.
    Input:
      source: in this function it's technically the goal node
      sink: the current node we are observing
      flow_in: the smallest flow found on the way down
    """
    if source == sink:
      return flow_in
    u, edge_ind = self.parent[sink]
    _, edge_cap, edge_flow = self.graph.edge_list[edge_ind]
    pushed_flow = self.send_flow_via_augmenting_path(
      source, u, min(flow_in, edge_cap - edge_flow))
    self.graph.edge_list[edge_ind][2] = edge_flow + pushed_flow
    self.graph.edge_list[edge_ind ^ 1][2] -= pushed_flow
    return pushed_flow

  def send_max_flow_via_dfs(self, u: int, sink: int, flow_in: Num):  # TODO RETEST
    """Function to recursively emulate sending a flow via dfs. Returns min pushed flow.

    Complexity per call: Time: O(|E| * |V|), Space O(|V|)
    More uses: a more efficient way of sending a flow
    Input:
      u: is the current node to be observed.
      sink: is the goal node (we might be able to just put it as instance var?).
      flow_in: the smallest flow found on the way down.
    """
    if u == sink or flow_in == 0:
      return flow_in
    start, end = self.last[u], len(self.graph.adj_list[u])
    for i in range(start, end):
      self.last[u], edge_ind = i, self.graph.adj_list[u][i]
      v, edge_cap, edge_flow = self.graph.edge_list[edge_ind]
      if self.dist[v] != self.dist[u] + 1:
        continue
      pushed_flow = self.send_max_flow_via_dfs(v, sink, min(flow_in, edge_cap - edge_flow))
      if pushed_flow != 0:
        self.graph.edge_list[edge_ind][2] = edge_flow + pushed_flow
        self.graph.edge_list[edge_ind ^ 1][2] -= pushed_flow
        return pushed_flow
    return 0

  def max_flow_via_edmonds_karp(self, source: int, sink: int):
    """Compute max flow using edmonds karp's method.

    Complexity per call: Time: O(|V| * |E|^2), Space O(|V|)
    More Uses: max flow of the graph, min cut of the graph.
    """
    max_flow = 0
    while self.max_flow_find_augmenting_path_helper(source, sink):
      flow = self.send_flow_via_augmenting_path(source, sink, INF)
      if flow == 0:
        break
      max_flow += flow
    return max_flow

  def max_flow_via_dinic(self, source: int, sink: int):
    """Compute max flow using Dinic's method.

    Complexity per call: Time: O(|E| * |V|^2), Space O(|V|)
    More Uses: faster than the one above for most cases.
    """
    max_flow = 0
    while self.max_flow_find_augmenting_path_helper(source, sink):
      self.last = [0] * self.graph.num_nodes
      flow = self.send_max_flow_via_dfs(source, sink, INF)
      while flow != 0:
        max_flow += flow
        flow = self.send_max_flow_via_dfs(source, sink, INF)
    return max_flow


################################################


from bisect import bisect_left
from collections import Counter
from functools import lru_cache
from itertools import takewhile, accumulate
from math import isqrt, log, gcd, prod, cos, sin, tau
from operator import mul as operator_mul


class MathAlgorithms:
  def __init__(self):
    """Only take what you need. This list needs to be global or instance level or passed in.
    Attributes declared here must be passed in or global if not used in class format.
    """
    self.mod_p = 0

    self.sum_prime_factors = []
    self.num_divisors = []
    self.sum_divisors = []
    self.euler_phi = []
    self.sum_diff_prime_factors = []
    self.num_diff_prime_factors = []
    self.num_prime_factors = []

    self.binomial = {}
    self.fact = []
    self.inv_fact = []
    self.catalan_numbers = []

    self.primes_list = []
    self.primes_set = set()
    self.prime_factors = []
    self.factor_list = {}
    self.min_primes_list = []

    self.mrpt_known_bounds = []
    self.mrpt_known_tests = []

    self.fibonacci_list = []
    self.fibonacci_dict = {0: 0, 1: 1, 2: 1}

    self.fft_lengths = []
    self.fft_swap_indices = []
    self.fft_roots_of_unity = []

  def is_prime_trivial(self, n: int) -> bool:
    """Tests if n is prime via divisors up to sqrt(n).

    Complexity per call: Time: O(sqrt(n)), T(sqrt(n)/3), Space: O(1)
    Optimizations: 6k + i method, since we checked 2 and 3 only need test form 6k + 1 and 6k + 5
    """
    if n < 4:  # base case of n in 0, 1, 2, 3
      return n > 1
    if n % 2 == 0 or n % 3 == 0:  # this check is what allows us to use 6k + i
```

```
    return False
  limit = isqrt(n) + 1
  for p in range(5, limit, 6):
    if n % p == 0 or n % (p+2) == 0:
      return False
  return True

def sieve_of_eratosthenes(self, n_inclusive: int) -> None:
  """Generates list of primes up to n via eratosthenes method.

  Complexity: Time: O(n lnln(n)), Space: post call O(n/ln(n)), mid-call O(n)
  Variants: number and sum of prime factors, of diff prime factors, of divisors, and euler phi
  """
  limit, prime_sieve = isqrt(n_inclusive) + 1, [True] * (n_inclusive + 1)
  prime_sieve[0] = prime_sieve[1] = False
  for prime in range(2, limit):
    if prime_sieve[prime]:
      for composite in range(prime * prime, n_inclusive + 1, prime):
        prime_sieve[composite] = False
  self.primes_list = [i for i, is_prime in enumerate(prime_sieve) if is_prime]

def sieve_of_eratosthenes_optimized(self, n_inclusive: int) -> None:
  """Odds only optimized version of the previous method. Optimized to start at 3.

  Complexity: Time: O(max(n lnln(sqrt(n)), n)), Space: post call O(n/ln(n)), mid-call O(n/2)
  """
  sqrt_n, limit = ((isqrt(n_inclusive) - 3) // 2) + 1, ((n_inclusive - 3) // 2) + 1
  primes_sieve = [True] * limit
  for i in range(sqrt_n):
    if primes_sieve[i]:
      prime = 2*i + 3
      start = (prime*prime - 3)//2
      for j in range(start, limit, prime):
        primes_sieve[j] = False
  self.primes_list = [2] + [2*i + 3 for i, el in enumerate(primes_sieve) if el]

def sieve_of_min_primes(self, n_inclusive: int) -> None:
  """Stores the min or max prime divisor for each number up to n.

  Complexity: Time: O(max(n lnln(sqrt(n)), n)), Space: post call O(n)
  """
  min_primes = [0] * (n_inclusive + 1)
  min_primes[1] = 1
  for prime in self.primes_list:
    min_primes[prime] = prime
    start, end, step = prime * prime, n_inclusive + 1, prime if prime == 2 else 2 * prime
    for j in range(start, end, step):
      min_primes[j] = prime
  self.min_primes_list = min_primes

def sieve_of_eratosthenes_variants(self, n_inclusive: int) -> None:
  """Seven variants of prime sieve listed above.

  Complexity:
    function 1: Time: O(n log(n)), Space: O(n)
    function 2: Time: O(n lnln(n)), Space: O(n)
    function 3: Time: O(n lnln(n) log(n)), Space: O(n)
    function 4: Time: O(n lnln(n) log(n)), Space: O(n)
  """
  self.euler_phi_plus_sum_and_number_of_diff_prime_factors(n_inclusive)
  self.num_and_sum_of_divisors(n_inclusive)
  self.num_and_sum_of_prime_factors(n_inclusive)

def num_and_sum_of_divisors(self, limit: int) -> None:
  """Does a basic sieve. Complexity function 1."""
  num_div = [1] * (limit + 1)
  sum_div = [1] * (limit + 1)
  for i in range(2, limit + 1):
    for j in range(i, limit + 1, i):
      num_div[j] += 1
      sum_div[j] += i
  self.num_divisors = num_div
  self.sum_divisors = sum_div

def euler_phi_plus_sum_and_number_of_diff_prime_factors(self, limit: int) -> None:
  """This is basically same as sieve just using different ops. Complexity function 2."""
  num_diff_pf = [0] * (limit + 1)
  sum_diff_pf = [0] * (limit + 1)
  phi = [i for i in range(limit + 1)]
  for i in range(2, limit + 1):
    if num_diff_pf[i] == 0:
      for j in range(i, limit + 1, i):
        num_diff_pf[j] += 1
        sum_diff_pf[j] += i
        phi[j] = (phi[j]//i) * (i-1)
  self.num_diff_prime_factors = num_diff_pf
  self.sum_diff_prime_factors = sum_diff_pf
  self.euler_phi = phi

def num_and_sum_of_prime_factors(self, limit: int) -> None:
  """This uses similar idea to sieve but avoids divisions. Complexity function 3."""
  num_pf = [0] * (limit + 1)
  sum_pf = [0] * (limit + 1)
  for prime in range(2, limit + 1):
    if num_pf[prime] == 0:  # or sum_pf if using that one
      exponent_limit = int(log(limit, prime)) + 1
      for exponent in range(1, exponent_limit):
        prime_to_exponent = prime**exponent
        for i in range(prime_to_exponent, limit + 1, prime_to_exponent):
          sum_pf[i] += prime
          num_pf[i] += 1
  self.num_prime_factors = num_pf
  self.sum_prime_factors = sum_pf

def num_and_sum_of_divisors_faster(self, limit: int) -> None:
  """runs in around x0.8-x0.5 the runtime of the slower one. Complexity function 4."""
  num_divs = [1] * (limit + 1)
  sum_divs = [1] * (limit + 1)
  cur_pows = [1] * (limit + 1)
  for prime in range(2, limit + 1):
    if num_divs[prime] == 1:
      exponent_limit = int(log(limit, prime)) + 1
      for exponent in range(1, exponent_limit):
        prime_to_exponent = prime ** exponent
        for i in range(prime_to_exponent, limit + 1, prime_to_exponent):
          cur_pows[i] += 1
      tmp = prime - 1  # this line and the line below used for sum_divs
      prime_powers = [prime ** exponent for exponent in range(0, exponent_limit+1)]
      for i in range(prime, limit + 1, prime):
        num_divs[i] *= cur_pows[i]
        sum_divs[i] *= ((prime_powers[cur_pows[i]] - 1) // tmp)
        cur_pows[i] = 1
  self.num_divisors = num_divs
  self.sum_divisors = sum_divs

def prime_factorize_n(self, n: int) -> Dict:
  """A basic prime factorization of n function. without primes its just O(sqrt(n))

  Complexity: Time: O(sqrt(n)/ln(sqrt(n))), Space: O(log n)
  Variants: number and sum of prime factors, of diff prime factors, of divisors, and euler phi
  """
  limit, prime_factors = isqrt(n) + 1, []
  for prime in takewhile(lambda x: x < limit, self.primes_list):
    if n % prime == 0:
      while n % prime == 0:
        n //= prime
        prime_factors.append(prime)
  if n > 1:  # n is prime or last factor of n is prime
    prime_factors.append(n)
  self.factor_list = Counter(prime_factors)
  return Counter(prime_factors)

def prime_factorize_n_log_n(self, n: int) -> Dict:
  """An optimized prime factorization of n function based on min primes already sieved.

  Complexity: Time: O(log n), Space: O(log n)
  Optimization: assign append to function and assign min_prime[n] to a value in the loop
  """
  prime_factors = []
  while n > 1:
    prime_factors.append(self.min_primes_list[n])
    n = n // self.min_primes_list[n]
  self.factor_list = Counter(prime_factors)
  return Counter(prime_factors)

def prime_factorize_n_variants(self, n: int) -> int:
  """Covers all the variants listed above, holds the same time complexity with O(1) space."""
  limit = isqrt(n) + 1
```

```
    sum_diff_prime_factors, num_diff_prime_factors = 0, 0
    sum_prime_factors, num_prime_factors = 0, 0
    sum_divisors, num_divisors, euler_phi = 1, 1, n
    for prime in takewhile(lambda x: x < limit, self.primes_list):
      if n % prime == 0:
        mul, total = prime, 1  # for sum of divisors
        power = 0        # for num of divisors
        while n % prime == 0:
          n //= prime
          power += 1     # for num prime factors, num divisors, and sum prime factors
          total += mul     # for sum divisors
          mul *= prime     # for sum divisors
        sum_diff_prime_factors += prime
        num_diff_prime_factors += 1
        sum_prime_factors += (prime * power)
        num_prime_factors += power
        num_divisors *= (power + 1)
        sum_divisors *= total
        euler_phi -= (euler_phi//prime)
    if n > 1:
      num_diff_prime_factors += 1
      sum_diff_prime_factors += n
      num_prime_factors += 1
      sum_prime_factors += n
      num_divisors *= 2
      sum_divisors *= (n + 1)
      euler_phi -= (euler_phi // n)
    return num_diff_prime_factors

  def polynomial_function_f(self, x: int, c: int, m: int) -> int:
    """Represents the function f(x) = (x^2 + c) in pollard rho and brent, cycle finding."""
    return (x * x + c) % m  # paste this in code for speed up. is here for clarity only

  def pollard_rho(self, n: int, x0=2, c=1) -> int:
    """Semi fast integer factorization. Based on the birthday paradox and floyd cycle finding.

    Complexity per call: Time: O(min(max(p), n^0.25) * ln n), Space: O(log2(n) bits)
    """
    x, y, g = x0, x0, 1
    while g == 1:  # when g != 1 then we found a divisor of n shared with x - y
      x = self.polynomial_function_f(x, c, n)
      y = self.polynomial_function_f(self.polynomial_function_f(y, c, n), c, n)
      g = gcd(abs(x - y), n)
    return g

  def brent_pollard_rho(self, n: int, x0=2, c=1) -> int:
    """Faster version of above. Similar time complexity. uses faster cycle finder."""
    x, m = x0, 128  # 128 here is used as a small power of 2 vs using 100 more below
    g = q = left = 1
    xs = y = 0
    while g == 1:
      y, k = x, 0
      for _ in range(1, left):
        x = (x * x + c) % n
      while k < left and g == 1:
        xs, end = x, min(m, left - k)   # here we are using a technique similar to cache
        for _ in range(end):      # and loop unrolling were we try for sets of cycles
          x = (x * x + c) % n     # if we over shoot we can just go back which is
          q = (q * abs(y - x)) % n  # technically what end computes
        k, g = k + m, gcd(q, n)
      left = left * 2
    if g == n:
      while True:
        xs = (xs * xs + c) % n
        g = gcd(abs(xs - y), n)
        if g != 1:
          break
    return g

  def is_composite(self, a: int, d: int, n: int, s: int) -> bool:
    """The witness test of miller rabin.

    Complexity per call: Time O(log^3(n)), Space: O(2**s, bits)
    """
    if 1 == pow(a, d, n):
      return False
    for i in range(s):
      if n-1 == pow(a, d * 2**i, n):
        return False
    return True

  def miller_rabin_primality_test(self, n: int, precision_for_huge_n=16) -> bool:
    """Probabilistic primality test with error rate of 4^(-k) past 341550071728321.

    Complexity per call: Time O(k log^3(n)), Space: O(2**s) bits
    Note: range(16) used to just do a small test to weed out lots of numbers.
    """
    if n < self.primes_list[-1]:
      return n in self.primes_set
    if any((n % self.primes_list[p] == 0) for p in range(16)) or n == 3215031751:
      return False  # 3215031751 is an edge case for this data set
    d, s = n-1, 0
    while d % 2 == 0:
      d, s = d//2, s+1
    if n < self.mrpt_known_bounds[-1]:
      for i, bound in enumerate(self.mrpt_known_bounds, 2):
        if n < bound:
          return not any(self.is_composite(self.mrpt_known_tests[j], d, n, s) for j in range(i))
    return not any(self.is_composite(self.primes_list[j], d, n, s)
                   for j in range(precision_for_huge_n))

  def miller_rabin_primality_test_prep(self):
    """This function needs to be called before miller rabin"""
    self.mrpt_known_bounds = [1373653, 25326001, 118670087467,
                              2152302898747, 3474749660383, 341550071728321]
    self.mrpt_known_tests = [2, 3, 5, 7, 11, 13, 17]
    self.sieve_of_eratosthenes(1000)  # comment out if different size needed
    self.primes_set = set(self.primes_list)  # comment out if already have bigger size

  def extended_euclid_recursive(self, a: int, b: int) -> Tuple[int, int, int]:
    """Solves coefficients of Bezout identity: ax + by = gcd(a, b), recursively

    Complexity per call: Time: O(log n), Space: O(log n) at the deepest call.
    """
    if 0 == b:
      return 1, 0, a
    x, y, d = self.extended_euclid_recursive(b, a % b)
    return y, x-y*(a//b), d

  def extended_euclid_iterative(self, a: int, b: int) -> Tuple[int, int, int]:
    """Solves coefficients of Bezout identity: ax + by = gcd(a, b), iteratively.

    Complexity per call: Time: O(log n) about twice as fast in python vs above, Space: O(1)
    Optimizations and notes:
      divmod and abs are used to help deal with big numbers, remove if < 2**64 for speedup.
    """
    last_remainder, remainder = abs(a), abs(b)
    x, y, last_x, last_y = 0, 1, 1, 0
    while remainder:
      last_remainder, (quotient, remainder) = remainder, divmod(last_remainder, remainder)
      x, last_x = last_x - quotient * x, x
      y, last_y = last_y - quotient * y, y
    return -last_x if a < 0 else last_x, -last_y if b < 0 else last_y, last_remainder

  def safe_modulo(self, a: int, n: int) -> int:
    """Existence is much for c++ which doesn't always handle % operator nicely.
    use ((a % n) + n) % n for getting proper mod of a potential negative value
    use (a + b) % --> ((a % n) + (b % n)) % n for operations sub out + for * and -
    """
    return ((a % n) + n) % n

  def modular_linear_equation_solver(self, a: int, b: int, n: int) -> List[int]:  # TODO RETEST
    """Solves gives the solution x in ax = b(mod n).

    Complexity per call: Time: O(log n), Space: O(d)
    """
    x, y, d = self.extended_euclid_iterative(a, n)
    if 0 == b % d:
      x = (x * (b//d)) % n
      return [(x + i*(n//d)) % n for i in range(d)]
    return []

  def linear_diophantine_1(self, a: int, b: int, c: int) -> Tuple[int, int]:  # TODO RETEST
    """Solves for x, y in ax + by = c. From stanford icpc 2013-14

    Complexity per call: Time: O(log n), Space: O(1).
    Notes: order matters? 25x + 18y = 839 != 18x + 25y = 839
    """
```

```
    d = gcd(a, b)
    if c % d == 0:
      x = c//d * self.mod_inverse(a//d, b//d)
      return x, (c - a * x) // b
    return -1, -1

  def linear_diophantine_2(self, a: int, b: int, c: int) -> Tuple[int, int]:  # TODO RETEST
    """Solves for x0, y0 in x = x0 + (b/d)n, y = y0 - (a/d)n.
    derived from ax + by = c, d = gcd(a, b), and d|c.
    Can further derive into: n = x0 (d/b), and n = y0 (d/a).

    Complexity per call: Time: O(log n), Space: O(1).
    Optimizations and notes:
      unlike above this function order doesn't matter if a != b
      for a speedup call math.gcd(a, b) at start and return accordingly on two lines
    """
    x, y, d = self.extended_euclid_iterative(a, b)
    return (-1, -1) if c % d != 0 else (x * (c // d), y * (c // d))

  def mod_inverse(self, b: int, m: int) -> None | int:
    """Solves b^(-1) (mod m).

    Complexity per call: Time: O(log n), Space: O(1)
    """
    x, y, d = self.extended_euclid_iterative(b, m)
    return None if d != 1 else x % m  # -1 instead of None if we intend to go on with the prog

  def chinese_remainder_theorem_1(self, remainders: List[int], modulos: List[int]) -> int:
    """Steven's CRT version to solve x in x = r[0] (mod m[0]) ... x = r[n-1] (mod m[n-1]).
    # TODO RETEST
    Complexity per call: Time: O(n log n), Space: O(1)? O(mt) bit size:
    Optimizations:
      prod is used from math since 3.8,
      we use mod mt in the forloop since x might get pretty big.
    """
    mt, x = prod(modulos), 0
    for i, modulo in enumerate(modulos):
      p = mt // modulo
      x = (x + (remainders[i] * self.mod_inverse(p, modulo) * p)) % mt
    return x

  def chinese_remainder_theorem_helper(self, mod1: int, rem1: int,
                                       mod2: int, rem2: int) -> Tuple[int, int]:
    """Chinese remainder theorem (special case): find z such that z % m1 = r1, z % m2 = r2.
    Here, z is unique modulo M = lcm(m1, m2). Return (z, M).  On failure, M = -1.
    from: stanford icpc 2016
    # TODO RETEST
    Complexity per call: Time: O(log n), Space: O(1)
    """
    s, t, d = self.extended_euclid_iterative(mod1, mod2)
    if rem1 % d != rem2 % d:
      mod3, s_rem_mod, t_rem_mod = mod1*mod2, s*rem2*mod1, t*rem1*mod2
      return ((s_rem_mod + t_rem_mod) % mod3) // d, mod3 // d
    return 0, -1

  def chinese_remainder_theorem_2(self, remainders: List[int],
                                  modulos: List[int]) -> Tuple[int, int]:
    """Chinese remainder theorem: find z such that z % m[i] = r[i] for all i.  Note that the
    solution is unique modulo M = lcm_i (m[i]).  Return (z, M). On failure, M = -1. Note that
    we do not require the r[i]'s to be relatively prime.
    from: stanford icpc 2016
    # TODO RETEST
    Complexity per call: Time: O(n log n), Space: O(1)? O(mt) bit size
    """
    z_m = remainders[0], modulos[0]
    for i, modulo in enumerate(modulos[1:], 1):
      z_m = self.chinese_remainder_theorem_helper(z_m[1], z_m[0], modulo, remainders[i])
      if -1 == z_m[1]:
        break
    return z_m

  def fibonacci_n_iterative(self, n: int) -> None:
    """Classic fibonacci solver. Generates answers from 0 to n inclusive.

    Complexity per call: Time: O(n), Space: O(n).
    """
    fib_list = [0] * (n + 1)
    fib_list[1] = 1
    for i in range(2, n+1):
      fib_list[i] = fib_list[i - 1] + fib_list[i - 2]
    self.fibonacci_list = fib_list

  @lru_cache(maxsize=None)
  def fibonacci_n_dp_cached(self, n: int) -> int:
    """Cached Dynamic programming to get the nth fibonacci. Derived from Cassini's identity.

    Complexity per call: Time: O(log n), Space: increase by O(log n).
    Optimization: can go back to normal memoization with same code but using dictionary.
    """
    if n < 3:
      return 1 if n else 0
    f1, f2 = self.fibonacci_n_dp_cached(n // 2 + 1), self.fibonacci_n_dp_cached((n - 1) // 2)
    return f1 * f1 + f2 * f2 if n & 1 else f1 * f1 - f2 * f2

  @lru_cache(maxsize=None)
  def fibonacci_n_dp_cached_faster(self, n: int) -> int:
    """Same as above but runs in ~Time*0.75 i.e. Above takes 20 seconds this takes 15."""
    if n < 3:
      return 1 if n else 0
    k = (n + 1)//2 if n & 1 else n//2
    k1, k2 = self.fibonacci_n_dp_cached_faster(k), self.fibonacci_n_dp_cached_faster(k-1)
    return k1*k1 + k2*k2 if n & 1 else (2*k2 + k1) * k1

  def generate_catalan_n(self, n: int) -> None:
    """Generate catalan up to n iteratively.

    Complexity per call: Time: O(n*O(multiplication)), Space: O(n * 2^(log n)).
    """
    catalan = [0] * (n+1)
    catalan[0] = 1
    for i in range(n-1):
      catalan[i + 1] = catalan[i] * (4*i + 2) // (i + 2)
    self.catalan_numbers = catalan

  def generate_catalan_n_mod_inverse(self, n: int, p: int) -> None:
    """Generate catalan up to n iteratively cat n % p.

    Complexity per call: Time: O(n log n), Space: O(n * (2^(log n)%p)).
    Variants: use prime factors of the factorial to cancel out the primes
    """
    catalan = [0] * (n+1)
    catalan[0] = 1
    for i in range(n-1):
      catalan[i+1] = (((4*i + 2) % p) * (catalan[i] % p) * pow(i+2, p-2, p)) % p
    self.catalan_numbers = catalan

  def catalan_via_prime_facts(self, n: int, k: int, mod_m: int) -> int:
    """Compute the nth Catalan number mod_n via prime factor reduction of C(2n, n)/(n+1).
    Notes: The function "num_and_sum_of_prime_factors" needs to be modified for computing number
    of each prime factor in all the numbers between 1-2n or 1 to n.

    Complexity per call: Time: O(max(n lnln(sqrt(n)), n)), Space: O(n/ln(n)).
    """
    top, bottom, ans = n, k, 1  # n = 2n and k = n in C(n, k) = (2n, n)
    self.num_and_sum_of_prime_factors(top)
    top_factors = [el for el in self.num_prime_factors]
    prime_array = [el for el in self.primes_list]  # saving primes to use in two lines later
    self.num_and_sum_of_prime_factors(bottom)    # will handle n!n! in one go stored in num
    for i, el in enumerate(self.num_prime_factors):  # num_prime_factors :)
      top_factors[i] -= (2*el)
    self.prime_factorize_n(k+1)  # factorizing here is faster than doing n! and (n+1)! separate
    for p, v in self.factor_list.items():
      top_factors[bisect_left(prime_array, p)] -= v
    for ind, exponent in enumerate(top_factors):  # remember use multiplication not addition
      if exponent > 0:
        ans = (ans * pow(prime_array[ind], exponent, mod_m)) % mod_m
    return ans

  def c_n_k(self, n: int, k: int) -> int:
    """Computes C(n, k) % p. From competitive programming 4.

    Complexity per call: Time: v1 = O(log n), v2 = O(1), Space: O(1).
    v1 is uncommented, v2 is the commented out line, and must be precomputed see below.
    """
    if n < k:  # base case: could flip them to be n, k = k, n but better to just return 0
      return 0
    n_fact, k_fact, n_k_fact, p = self.fact[n], self.fact[k], self.fact[n - k], self.mod_p
    return (n_fact * pow(k_fact, p - 2, p) * pow(n_k_fact, p - 2, p)) % p
```

```
  # return 0 if n < k else (self.fact[n] * self.inv_fact[k] * self.inv_fact[n-k]) % self.mod_p

def binomial_coefficient_n_mod_p_prep(self, max_n: int, mod_p: int):
  """Does preprocessing for binomial coefficients. From competitive programming 4.

  Complexity per call: Time: v1 O(n), v2 = O(n), Space: O(n).
  Optimization and notes: v2 -> uncomment lines for C(n, k) % p in O(1) time, see above.
  """
  factorial_mod_p = [1] * max_n
  for i in range(1, max_n):
    factorial_mod_p[i] = (factorial_mod_p[i - 1] * i) % mod_p
  self.mod_p, self.fact = mod_p, factorial_mod_p
  # inverse_factorial_mod_p = [0] * max_n
  # inverse_factorial_mod_p[-1] = pow(factorial_mod_p[-1], mod_p-2, mod_p)
  # for i in range(max_n-2, -1, -1):
  #   inverse_factorial_mod_p[i] = (inverse_factorial_mod_p[i+1] * (i+1)) % mod_p
  # self.inv_fact = inverse_factorial_mod_p

@lru_cache(maxsize=None)
def binomial_coefficient_dp_with_cache(self, n: int, k: int) -> int:
  """Uses the recurrence to calculate binomial coefficient. Cached for memoization.

  Complexity per call: Time: O(n*k), Space: O(n*k).
  """
  if n == k or 0 == k:
    return 1
  take_case = self.binomial_coefficient_dp_with_cache(n-1, k)
  skip_case = self.binomial_coefficient_dp_with_cache(n-1, k-1)
  return take_case + skip_case

def fft_prepare_swap_indices(self, a_len: int):
  """Gives us all the swap pairs needed to for an FFT call stored in fft_swap_indices.

  Complexity per call: Time: O(n), Space: O(n), S(n/2) | n == 2^i -> minimax value of i.
  """
  a_bit_len, a_bit_half = a_len.bit_length(), (a_len.bit_length()-1)//2
  swap_size = (1 << a_bit_half) - 1
  swap_size = swap_size << (a_bit_half-1) if a_bit_len & 1 else swap_size << a_bit_half
  swaps, k, ind = [None] * swap_size, 0, -1
  for i in range(1, a_len):
    bit_mask = a_len >> 1
    while k & bit_mask:
      k = k ^ bit_mask
      bit_mask = bit_mask >> 1
    k = k ^ bit_mask
    if i < k:
      swaps[ind := ind + 1] = (i, k)
  self.fft_swap_indices = swaps

def fft_prepare_lengths_list(self, a_len: int):
  """Function for all powers 2 from 2 - a_len, inclusive. O(log n) complexity."""
  self.fft_lengths = [2**i for i in range(1, a_len.bit_length())]

def fft_prepare_roots_helper(self, length: int, angle: float) -> List[complex]:
  """Precomputes roots of unity for a given length and angle. accumulate used here :).

  Complexity per call: Time: O(n), Space: O(n) | n == 2^i.
  """
  initial_root_of_unity = complex(1)
  multiplier = complex(cos(angle), sin(angle))
  return list(accumulate([multiplier] * length, operator_mul, initial=initial_root_of_unity))

def fft_prepare_roots_of_unity(self, invert: bool):
  """Precomputes all roots of unity for all lengths. Stores the result for later use.

  Complexity per call: Time: O(2^((log n)+1)-1) = O(n),
            Space: O(n), S(2^((log n)+1)-1) | n == len of our data, which is 2^i.
  """
  signed_tau: float = -tau if invert else tau
  self.fft_roots_of_unity = [self.fft_prepare_roots_helper(length//2 - 1, signed_tau/length)
                             for length in self.fft_lengths]

def fft_in_place_fast_fourier_transform(self, a_vector: List[complex], invert: bool):
  """Optimized in-place Cooley-Tukey FFT algorithm. Modifies a_vector.

  Complexity per call: Time: O(n log n), Space: O(1) | technically O(n) but we precompute.
  Optimizations: swap indices, lengths, and roots of unity all have be calculated beforehand.
    This allows us to only do those once in when doing multiplication
  """
    a_len = len(a_vector)
    for i, j in self.fft_swap_indices:
      a_vector[i], a_vector[j] = a_vector[j], a_vector[i]
    for k, length in enumerate(self.fft_lengths):  # is [2, 4, 8..., 2^(i-1), 2^i] | n == 2^i
      j_end = length // 2  # j_end is to avoid repeated divisions in the innermost loop
      for i in range(0, a_len, length):
        for j, w in enumerate(self.fft_roots_of_unity[k]):
          i_j, i_j_j_end = i + j, i + j + j_end
          u, v = a_vector[i_j], w * a_vector[i_j_j_end]
          a_vector[i_j], a_vector[i_j_j_end] = u + v, u - v
    if invert:
      a_vector[:] = [complex_number/a_len for complex_number in a_vector]

  def fft_normalize(self, a_vector: List[int], base: int) -> List[int]:
    """Normalizes polynomial a for a given base. base 10 will result in a base 10 number

    Complexity per call: Time: O(n), Space: O(1) -> in fact we often reduce overall space.
    """
    carry, end = 0, len(a_vector) - 1
    for number in a_vector:
      number += carry
      carry, number = divmod(number, base)
    while 0 == a_vector[end]:
      end -= 1
    return a_vector[:end+1][::-1]

  def fft_multiply_in_place(self, polynomial_a: List[int], polynomial_b: List[int]) -> List[int
     ]:
    """Multiply two polynomials with the option to normalize then after.

    Complexity per call: Time: O(n log n), T(4(n log n)),
              Space: O(n),  S(4n) | [n == |a| + |b|]
    Optimizations: listed fft_in_place_fast_fourier_transform.
    """
    a_len, b_len = len(polynomial_a), len(polynomial_b)
    n = 2**((a_len + b_len).bit_length())     # computes 2^(log2(a+b) + 1)
    n = n if (a_len + b_len) != n//2 else n//2  # optimization that fixes n when (a+b) % 2 == 0
    a_vector = [complex(i) for i in polynomial_a] + [complex(0)] * (n - a_len)
    b_vector = [complex(i) for i in polynomial_b] + [complex(0)] * (n - b_len)
    self.fft_prepare_swap_indices(n)    # these three calls are for optimization with
    self.fft_prepare_lengths_list(n)    # multiplying, if calling fft outside multiply
    self.fft_prepare_roots_of_unity(False)  # you will need to call for each size of array.
    self.fft_in_place_fast_fourier_transform(a_vector, False)
    self.fft_in_place_fast_fourier_transform(b_vector, False)
    a_vector = [i * j for i, j in zip(a_vector, b_vector)]
    self.fft_prepare_roots_of_unity(True)
    self.fft_in_place_fast_fourier_transform(a_vector, True)
    a_vector = [int(round(el.real)) for el in a_vector]   # optional
    return self.fft_normalize(a_vector, 10)       # optional turns into base 10 num


################################################


from math import isclose, dist, sin, cos, acos, sqrt, fsum, pi
from itertools import combinations
# remember to sub stuff out for integer ops when you want only integers
# for ints need to change init, eq and
# hard code these in for performance speedup
CCW = 1  # counterclockwise
CW = -1  # clockwise
CL = 0   # collinear
EPS = 1e-12  # used in some spots


class Pt2d:  # TODO RETEST
  __slots__ = ("x", "y")
  # def __init__(self, x_val, y_val): self.x, self.y = map(float, (x_val, y_val))
  def __init__(self, x_val, y_val): self.x, self.y = x_val, y_val

  def __add__(self, other): return Pt2d(self.x + other.x, self.y + other.y)
  def __sub__(self, other): return Pt2d(self.x - other.x, self.y - other.y)
  def __mul__(self, scale): return Pt2d(self.x * scale, self.y * scale)
  def __truediv__(self, scale): return Pt2d(self.x / scale, self.y / scale)
  def __floordiv__(self, scale): return Pt2d(self.x // scale, self.y // scale)

  # def __eq__(self, other): return isclose(self.x, other.x) and isclose(self.y, other.y)
  def __eq__(self, other): return self.x == other.x and self.y == other.y
```

```
  # def __lt__(self, other):
  #   return self.x < other.x if not isclose(self.x, other.x) else self.y < other.y
  def __lt__(self, other): return self.x < other.x if self.x != other.x else self.y < other.y

  def __str__(self): return "{} {}".format(self.x, self.y)
  # def __str__(self): return "(x = {:20}, y = {:20})".format(self.x, self.y)
  def __round__(self, n): return Pt2d(round(self.x, n), round(self.y, n))
  def __hash__(self): return hash((self.x, self.y))

  def get_tup(self): return self.x, self.y


Triangle = Tuple[Pt2d, Pt2d, Pt2d] | None
ClosestPair = Tuple[Numeric, Pt2d, Pt2d]


class Pt3d:  # TODO RETEST
  def __init__(self, x_val, y_val, z_val):
    self.x, self.y, self.z = map(float, (x_val, y_val, z_val))

  def __add__(self, other): return Pt3d(self.x + other.x, self.y + other.y, self.z + other.z)
  def __sub__(self, other): return Pt3d(self.x - other.x, self.y - other.y, self.z - other.z)
  def __mul__(self, scale): return Pt3d(self.x * scale, self.y * scale, self.z * scale)
  def __truediv__(self, scale): return Pt3d(self.x / scale, self.y / scale, self.z / scale)
  def __floordiv__(self, scale): return Pt3d(self.x // scale, self.y // scale, self.z // scale)

  def __eq__(self, other):
    return isclose(self.x, other.x) and isclose(self.y, other.y) and isclose(self.z, other.z)

  def __lt__(self, other):
    return False if self == other else (self.x, self.y, self.z) < (other.x, other.y, other.y)

  def __str__(self): return "{} {} {}".format(self.x, self.y, self.z)
  # def __str__(self): return "(x = {:20}, y = {:20}), z = {:20})".format(self.x, self.y, self.z
    )
  def __round__(self, n): return Pt3d(round(self.x, n), round(self.y, n), round(self.z, n))
  def __hash__(self): return hash((self.x, self.y, self.z))


class QuadEdge:
  __slots__ = ("origin", "rot", "o_next", "used")

  def __init__(self):
    self.origin = None
    self.rot = None
    self.o_next = None
    self.used = False

  def rev(self): return self.rot.rot
  def l_next(self): return self.rot.rot.rot.o_next.rot
  def o_prev(self): return self.rot.o_next.rot
  def dest(self): return self.rot.rot.origin


QuadEdgePair = Tuple[QuadEdge, QuadEdge]


class QuadEdgeDataStructure:
  def __init__(self):
    pass

  def make_edge(self, in_pt, out_pt):
    e1 = QuadEdge()
    e2 = QuadEdge()
    e3 = QuadEdge()
    e4 = QuadEdge()
    e1.origin = in_pt
    e2.origin = out_pt
    e3.origin = None
    e4.origin = None
    e1.rot = e3
    e2.rot = e4
    e3.rot = e2
    e4.rot = e1
    e1.o_next = e1
    e2.o_next = e2
    e3.o_next = e4
    e4.o_next = e3
    return e1

  def splice(self, a, b):
    a.o_next.rot.o_next, b.o_next.rot.o_next = b.o_next.rot.o_next, a.o_next.rot.o_next
    a.o_next, b.o_next = b.o_next, a.o_next

  def delete_edge(self, edge):
    self.splice(edge, edge.o_prev())
    self.splice(edge.rev(), edge.rev().o_prev())
    # del edge.rot.rot.rot
    # del edge.rot.rot
    # del edge.rot
    # del edge

  def connect(self, a, b):
    e = self.make_edge(a.dest(), b.origin)
    self.splice(e, a.l_next())
    self.splice(e.rev(), b)
    return e


class GeometryAlgorithms:
  def __init__(self):
    self.quad_edges = QuadEdgeDataStructure()

  def compare_ab(self, a: Numeric, b: Numeric) -> int:
    """Compare a b, for floats and ints. It is useful when you want set values to observe.
    paste directly into code and drop isclose for runtime speedup."""
    return 0 if isclose(a, b) else -1 if a < b else 1

  def dot_product(self, left_vector: Pt2d, right_vector: Pt2d) -> Reals:
    """Compute the scalar product a.b of a,b equivalent to: a . b"""
    return left_vector.x*right_vector.x + left_vector.y*right_vector.y

  def cross_product(self, left_vector: Pt2d, right_vector: Pt2d) -> Reals:
    """Computes the scalar value perpendicular to a,b equivalent to: a x b"""
    return left_vector.x*right_vector.y - left_vector.y*right_vector.x

  def distance_normalized(self, left_point: Pt2d, right_point: Pt2d) -> float:
    """Normalized distance between two points a, b equivalent to: sqrt(a^2 + b^2) = distance."""
    return dist(left_point, right_point)

  def distance(self, left_point: Pt2d, right_point: Pt2d) -> Reals:
    """Squared distance between two points a, b equivalent to: a^2 + b^2 = distance."""
    return self.dot_product(left_point - right_point, left_point - right_point)

  def rotate_cw_90_wrt_origin(self, point: Pt2d) -> Pt2d:
    """Compute a point rotation on pt. Just swap x and y and negate x."""
    return Pt2d(point.y, -point.x)

  def rotate_ccw_90_wrt_origin(self, point: Pt2d) -> Pt2d:
    """Compute a point rotation on pt. Just swap x and y and negate y."""
    return Pt2d(-point.y, point.x)

  def rotate_ccw_rad_wrt_origin(self, point: Pt2d, degree_in_radians: float) -> Pt2d:
    """Compute a counterclockwise point rotation on pt. Accurate only for floating point cords.
    formula: x = (x cos(rad) - y sin(rad)), y = (x sin(rad) + y cos (rad)).

    Complexity per call: Time: O(1), Space: O(1).
    Optimizations: calculate cos and sin outside the return, so you don't double call each.
    """
    return Pt2d(point.x * cos(degree_in_radians) - point.y * sin(degree_in_radians),
                point.x * sin(degree_in_radians) + point.y * cos(degree_in_radians))

  def point_c_rotation_wrt_line_ab(self, a_point: Pt2d, b_point: Pt2d, c_point: Pt2d) -> int:
    """Determine orientation of c wrt line ab, in terms of collinear clockwise counterclockwise.
    Since 2d cross-product is the area of the parallelogram, we can use this to accomplish this.

    Complexity per call: Time: O(1), Space: O(1).
    Returns collinear(cl): 0, counterclockwise(ccw): 1, clockwise(cw): -1
    Optimizations: if x,y are ints, use 0 instead of 0.0 or just paste the code here directly.
    """
    return self.compare_ab(self.cross_product(b_point - a_point, c_point - a_point), 0.0)

  def angle_point_c_wrt_line_ab(self, a_point: Pt2d, b_point: Pt2d, c_point: Pt2d) -> float:
    """For a line ab and point c, determine the angle of a to b to c in radians.
    formula: arc-cos(dot(vec_ba, vec_bc) / sqrt(dist_sq(vec_ba) * dist_sq(vec_bc))) = angle

    Complexity per call: Time: O(1), Space: O(1).
    Optimizations: for accuracy we sqrt both distances can remove if distances are ints.
```

```
    """
    vector_ba, vector_bc = a_point - b_point, c_point - b_point
    dot_ba_bc = self.dot_product(vector_ba, vector_bc)
    dist_sq_ba = self.dot_product(vector_ba, vector_ba)
    dist_sq_bc = self.dot_product(vector_bc, vector_bc)
    return acos(dot_ba_bc / (sqrt(dist_sq_ba) * sqrt(dist_sq_bc)))
    # return acos(dot_ba_bc / sqrt(dist_sq_ba * dist_sq_bc))

def pt_on_line_segment_ab(self, point_a: Pt2d, point_b: Pt2d, point: Pt2d) -> bool:
    """Logic is cross == 0 mean parallel, and dot being <= 0 means different directions."""
    vec_pa, vec_pb = point_a - point, point_b - point
    return (self.compare_ab(self.cross_product(vec_pa, vec_pb), 0) == 0
            and self.compare_ab(self.dot_product(vec_pa, vec_pb), 0) <= 0)

def project_pt_c_to_line_ab(self, a_point: Pt2d, b_point: Pt2d, c_point: Pt2d) -> Pt2d:
    """Compute the point closest to c on the line ab.
    formula: pt = a + u x vector_ba, where u is the scalar projection of vector_ca onto
    vector_ba via dot-product
    # TODO RETEST
    Complexity per call: Time: O(1), Space: O(1).
    """
    vec_ab, vec_ac = b_point - a_point, c_point - a_point
    translation = vec_ab * (self.dot_product(vec_ac, vec_ab) / self.dot_product(vec_ab, vec_ab))
    return a_point + translation

def project_pt_c_to_line_seg_ab(self, a_point: Pt2d, b_point: Pt2d, c_point: Pt2d) -> Pt2d:
    """Compute the point closest to c on the line segment ab.
    Rule if a==b, then if c closer to a or b, otherwise we can just use the line version.
    # TODO RETEST
    Complexity per call: Time: O(1), Space: O(1).
    Optimizations: use compare_ab on the last line if needed better accuracy.
    """
    if a_point == b_point:  # base case, closest point is either, avoids division by 0 below
      return a_point
    vec_ab, vec_ac = b_point - a_point, c_point - a_point
    u = self.dot_product(vec_ac, vec_ab) / self.dot_product(vec_ab, vec_ab)
    return (a_point if u < 0.0 else b_point if u > 1.0           # closer to a or b
            else self.project_pt_c_to_line_ab(a_point, b_point, c_point))  # inbetween a and b

def distance_pt_c_to_line_ab(self, a_point: Pt2d, b_point: Pt2d, c_point: Pt2d) -> float:
    """Just return the distance between c and the projected point :)."""
    closest_point: Pt2d = self.project_pt_c_to_line_ab(a_point, b_point, c_point)
    return self.distance_normalized(c_point, closest_point)

def distance_pt_c_to_line_seg_ab(self, a_point: Pt2d, b_point: Pt2d, c_point: Pt2d) -> float:
    """Same as above, just return the distance between c and the projected point :)."""
    # TODO RETEST
    closest_point: Pt2d = self.project_pt_c_to_line_seg_ab(a_point, b_point, c_point)
    return self.distance_normalized(c_point, closest_point)

def is_parallel_lines_ab_and_cd(self, endpoint_a: Pt2d, endpoint_b: Pt2d,
                                endpoint_c: Pt2d, endpoint_d: Pt2d) -> bool:  # TODO RETEST
    """Two lines are parallel if the cross_product between vec_ab and vec_dc is 0."""
    vec_ab, vec_dc = endpoint_b - endpoint_a, endpoint_c - endpoint_d
    return self.compare_ab(self.cross_product(vec_ab, vec_dc), 0.0) == 0

def is_collinear_lines_ab_and_cd_1(self, end_pt_a: Pt2d, end_pt_b: Pt2d,
                                   end_pt_c: Pt2d, end_pt_d: Pt2d) -> bool:  # TODO RETEST
    """Old function. a!=b and c!=d and then returns correctly"""
    return (self.is_parallel_lines_ab_and_cd(end_pt_a, end_pt_b, end_pt_c, end_pt_d)
            and self.is_parallel_lines_ab_and_cd(end_pt_b, end_pt_a, end_pt_a, end_pt_c)
            and self.is_parallel_lines_ab_and_cd(end_pt_d, end_pt_c, end_pt_c, end_pt_a))

def is_collinear_lines_ab_and_cd_2(self, end_point_a: Pt2d, endpoint_b: Pt2d,
                                   endpoint_c: Pt2d, endpoint_d: Pt2d) -> bool:  # TODO RETEST
    """Two lines are collinear iff a!=b and c!=d, and both c and d are collinear to line ab."""
    return (self.point_c_rotation_wrt_line_ab(end_point_a, endpoint_b, endpoint_c) == 0
            and self.point_c_rotation_wrt_line_ab(end_point_a, endpoint_b, endpoint_d) == 0)

def is_segments_intersect_ab_to_cd(self, end_pt_a: Pt2d, end_pt_b: Pt2d,
                                   end_pt_c: Pt2d, end_pt_d: Pt2d) -> bool:
    """4 distinct points as two lines intersect if they are collinear and at least one of the
     end points c or d are in between a and b otherwise, need to compute cross products."""
    if self.is_collinear_lines_ab_and_cd_2(end_pt_a, end_pt_b, end_pt_c, end_pt_d):
      lo, hi = (end_pt_a, end_pt_b) if end_pt_a < end_pt_b else (end_pt_b, end_pt_a)
      return lo <= end_pt_c <= hi or lo <= end_pt_d <= hi
    vec_ad, vec_ab, vec_ac = end_pt_d - end_pt_a, end_pt_b - end_pt_a, end_pt_c - end_pt_a
    vec_ca, vec_cd, vec_cb = end_pt_a - end_pt_c, end_pt_d - end_pt_c, end_pt_b - end_pt_c
    point_a_value = self.cross_product(vec_ad, vec_ab) * self.cross_product(vec_ac, vec_ab)
    point_c_value = self.cross_product(vec_ca, vec_cd) * self.cross_product(vec_cb, vec_cd)
    return not (point_a_value > 0 or point_c_value > 0)

def is_lines_intersect_ab_to_cd(self, end_pt_a: Pt2d, end_pt_b: Pt2d,
                                end_pt_c: Pt2d, end_pt_d: Pt2d) -> bool:
    """Two lines intersect if they aren't parallel or if they collinear."""
    return (not self.is_parallel_lines_ab_and_cd(end_pt_a, end_pt_b, end_pt_c, end_pt_d)
            or self.is_collinear_lines_ab_and_cd_2(end_pt_a, end_pt_b, end_pt_c, end_pt_d))

def pt_lines_intersect_ab_to_cd(self, end_pt_a: Pt2d, end_pt_b: Pt2d,
                                end_pt_c: Pt2d, end_pt_d: Pt2d) -> Pt2d:  # TODO RETEST
    """Compute the intersection point between two lines via cross products of the vectors."""
    vec_ab, vec_ac, vec_dc = end_pt_b - end_pt_a, end_pt_c - end_pt_a, end_pt_c - end_pt_d
    vec_t = vec_ab * (self.cross_product(vec_ac, vec_dc) / self.cross_product(vec_ab, vec_dc))
    return end_pt_a + vec_t

def pt_line_seg_intersect_ab_to_cd(self, a: Pt2d, b: Pt2d, c: Pt2d, d: Pt2d) -> Pt2d:
    """Same as for line intersect but this time we need to use a specific formula.
    Formula: # TODO RETEST"""
    x, y, cross_prod = c.x - d.x, d.y - c.y, self.cross_product(d, c)
    u = abs(y * a.x + x * a.y + cross_prod)
    v = abs(y * b.x + x * b.y + cross_prod)
    return Pt2d((a.x * v + b.x * u) / (v + u), (a.y * v + b.y * u) / (v + u))

def is_point_in_radius_of_circle(self, point: Pt2d, center_point: Pt2d, radius: float) -> bool
    :
    """True if point p in circle False otherwise. Use <= for circumference inclusion."""
    return self.compare_ab(self.distance_normalized(point, center_point), radius) < 0

def pt_circle_center_given_pt_abc(self, a_point: Pt2d, b_point: Pt2d, c_point: Pt2d) -> Pt2d:
    """Find the center of a circle based of 3 distinct points
    TODO add in the formula
    =# TODO RETEST
    """
    ab, ac = (a_point + b_point) / 2, (a_point + c_point) / 2
    ab_rotated: Pt2d = self.rotate_cw_90_wrt_origin(a_point - ab) + ab
    ac_rotated: Pt2d = self.rotate_cw_90_wrt_origin(a_point - ac) + ac
    return self.pt_lines_intersect_ab_to_cd(ab, ab_rotated, ac, ac_rotated)

def pts_line_ab_intersects_circle_cr(self, a, b, c, r):  # TODO RETEST
    """Compute the point(s) that line ab intersects circle c radius r. from stanford 2016
    TODO add in the formula
    """
    vec_ba, vec_ac = b-a, a-c
    dist_sq_ba = self.dot_product(vec_ba, vec_ba)
    dist_sq_ac_ba = self.dot_product(vec_ac, vec_ba)
    dist_sq_ac = self.dot_product(vec_ac, vec_ac) - r * r
    dist_sq = dist_sq_ac_ba*dist_sq_ac_ba - dist_sq_ba*dist_sq_ac
    result = self.compare_ab(dist_sq, 0.0)
    if result >= 0:
      first_intersect = c + vec_ac + vec_ba*(-dist_sq_ac_ba + sqrt(dist_sq + EPS))/dist_sq_ba
      second_intersect = c + vec_ac + vec_ba*(-dist_sq_ac_ba - sqrt(dist_sq))/dist_sq_ba
      return first_intersect if result == 0 else first_intersect, second_intersect
    return None  # no intersect

def pts_two_circles_intersect_cr1_cr2(self, c1: Pt2d, c2: Pt2d, r1, r2):  # TODO RETEST
    """I think this is the points on the circumference but not fully sure. from stanford 2016
    TODO add in teh formula
    """
    center_dist = self.distance_normalized(c1, c2)
    if (self.compare_ab(center_dist, r1 + r2) <= 0 <= self.compare_ab(center_dist + min(r1, r2),
                                                                      max(r1, r2))):
      x = (center_dist*center_dist - r2*r2 + r1*r1)/(2*center_dist)
      y = sqrt(r1*r1 - x*x)
      v = (c2-c1)/center_dist
      pt1, pt2 = c1 + v * x, self.rotate_ccw_90_wrt_origin(v) * y
      return (pt1 + pt2) if self.compare_ab(y, 0.0) <= 0 else (pt1+pt2, pt1-pt2)
    return None  # no overlap

def pt_tangent_to_circle_cr(self, center_point: Pt2d, radius: Numeric, pt: Pt2d) -> List[Pt2d
    ]:
    """Find the two points that create tangent lines from p to the circumference.
    TODO add in teh formula
    # TODO RETEST
    """
    vec_pc = pt - center_point
    x = self.dot_product(vec_pc, vec_pc)
    dist_sq = x - radius * radius
    result = self.compare_ab(dist_sq, 0.0)
```

```
    if result >= 0:
      dist_sq = dist_sq if result else 0
      q1 = vec_pc * (radius * radius / x)
      q2 = self.rotate_ccw_90_wrt_origin(vec_pc * (-radius * sqrt(dist_sq) / x))
      return [center_point + q1 - q2, center_point + q1 + q2]
    return []

  def tangents_between_2_circles(self, c1, r1, c2, r2):  # TODO RETEST
    """Between two circles there should be at least 4 points that make two tangent lines.
    TODO add in teh formula
    """
    if self.compare_ab(r1, r2) == 0:
      c2c1 = c2 - c1
      multiplier = r1/sqrt(self.dot_product(c2c1, c2c1))
      tangent = self.rotate_ccw_90_wrt_origin(c2c1 * multiplier)  # need better name
      r_tangents = [(c1+tangent, c2+tangent), (c1-tangent, c2-tangent)]
    else:
      ref_pt = ((c1 * -r2) + (c2 * r1)) / (r1 - r2)
      ps = self.pt_tangent_to_circle_cr(c1, r1, ref_pt)
      qs = self.pt_tangent_to_circle_cr(c2, r2, ref_pt)
      r_tangents = [(ps[i], qs[i]) for i in range(min(len(ps), len(qs)))]
    ref_pt = ((c1 * r2) + (c2 * r1)) / (r1 + r2)
    ps = self.pt_tangent_to_circle_cr(c1, r1, ref_pt)
    qs = self.pt_tangent_to_circle_cr(c2, r2, ref_pt)
    for i in range(min(len(ps), len(qs))):
      r_tangents.append((ps[i], qs[i]))
    return r_tangents

  def sides_of_triangle_abc(self, a, b, c):  # TODO RETEST
    """Compute the side lengths of a triangle."""
    dist_ab = self.distance_normalized(a, b)
    dist_bc = self.distance_normalized(b, c)
    dist_ca = self.distance_normalized(c, a)
    return dist_ab, dist_bc, dist_ca

  def pt_p_in_triangle_abc(self, a, b, c, p):  # TODO RETEST
    """Compute if a point is in or on a triangle. If all edges return the same orientation this
    should return true and the point should be in or on the triangle."""
    return (self.point_c_rotation_wrt_line_ab(a, b, p) >= 0
            and self.point_c_rotation_wrt_line_ab(b, c, p) >= 0
            and self.point_c_rotation_wrt_line_ab(c, a, p) >= 0)

  def perimeter_of_triangle_abc(self, side_ab, side_bc, side_ca):  # TODO RETEST
    """Computes the perimeter of triangle given the side lengths."""
    return side_ab + side_bc + side_ca

  def triangle_area_bh(self, base, height):  # TODO RETEST
    """Simple triangle area formula: area = b*h/2."""
    return base*height/2

  def triangle_area_from_heron_abc(self, side_ab, side_bc, side_ca):  # TODO RETEST
    """Compute heron's formula which gives us the area of a triangle given the side lengths."""
    s = self.perimeter_of_triangle_abc(side_ab, side_bc, side_ca) / 2
    return sqrt(s * (s-side_ab) * (s-side_bc) * (s-side_ca))

  def triangle_area_from_cross_product_abc(self, a, b, c):  # TODO RETEST
    """Compute triangle area, via cross-products of the pairwise sides ab, bc, ca."""
    return (self.cross_product(a, b) + self.cross_product(b, c) + self.cross_product(c, a))/2

  # def incircle_radis_of_triangle_abc_helper(self, ab, bc, ca):
  #   area = self.triangle_area_from_heron_abc(ab, bc, ca)
  #   perimeter = self.perimeter_of_triangle_abc(ab, bc, ca) / 2
  #   return area/perimeter

  def incircle_radius_of_triangle_abc(self, a, b, c):  # TODO RETEST
    """Computes the radius of the incircle, achieved by computing the side lengths then finding
    the area and perimeter to use in this Formula: r = area/(perimeter/2) Author: TODO Author
    """
    side_ab, side_bc, side_ca = self.sides_of_triangle_abc(a, b, c)
    area = self.triangle_area_from_heron_abc(side_ab, side_bc, side_ca)
    perimeter = self.perimeter_of_triangle_abc(side_ab, side_bc, side_ca) / 2
    return area / perimeter

  # def circumcircle_radis_of_triangle_abc_helper(self, ab, bc, ca):
  #   area = self.triangle_area_from_heron_abc(ab, bc, ca)
  #   return (ab*bc*ca) / (4*area)

  def circumcircle_radius_of_triangle_abc(self, a, b, c):  # TODO RETEST
    """Computes the radius of the circum-circle, achieved by computing the side lengths then
    gets the area for Formula: r = (ab * bc * ca) / (4 * area) Author: TODO Author
    """
    side_ab, side_bc, side_ca = self.sides_of_triangle_abc(a, b, c)
    area = self.triangle_area_from_heron_abc(side_ab, side_bc, side_ca)
    return (side_ab * side_bc * side_ca) / (4 * area)

  def incircle_pt_for_triangle_abc_1(self, a, b, c):  # TODO RETEST
    """Get the circle center of an incircle.

    Complexity per call: Time: lots of ops but still O(1), Space O(1)
    Formula: TODO add in the formula
    Optimization: get sides individually instead of through another call
    """
    radius = self.incircle_radius_of_triangle_abc(a, b, c)
    if self.compare_ab(radius, 0.0) == 0:  # if the radius was 0 we don't have a point
      return False, 0, 0
    side_ab, side_bc, side_ca = self.sides_of_triangle_abc(a, b, c)
    ratio_1 = side_ab/side_ca
    ratio_2 = side_ab/side_bc
    pt_1 = b + (c-b) * (ratio_1/(ratio_1 + 1.0))
    pt_2 = a + (c-a) * (ratio_2/(ratio_2 + 1.0))

    if self.is_lines_intersect_ab_to_cd(a, pt_1, b, pt_2):
      intersection_pt = self.pt_lines_intersect_ab_to_cd(a, pt_1, b, pt_2)
      return True, radius, round(intersection_pt, 12)  # can remove the round function
    return False, 0, 0

  def triangle_circle_center_pt_abcd(self, a, b, c, d):  # TODO RETEST
    """A 2 in one method that can get the middle point of both incircle circumcenter.
    Method: TODO add in the formula

    Complexity per call: Time: lots of ops but still O(1), Space O(1)
    Optimization: paste rotation code instead of function call
    """
    pt_1 = self.rotate_cw_90_wrt_origin(b-a)  # rotation on the vector b-a
    pt_2 = self.rotate_cw_90_wrt_origin(d-c)  # rotation on the vector d-c
    cross_product_1_2 = self.cross_product(pt_1, pt_2)
    # cross_product_2_1 = -cross_product_1_2  # self.cross_product(pt_2, pt_1)
    if self.compare_ab(cross_product_1_2, 0.0) == 0:
      return None
    pt_3 = Pt2d(self.dot_product(a, pt_1), self.dot_product(c, pt_2))
    x = ((pt_3.x * pt_2.y) - (pt_3.y * pt_1.y)) / cross_product_1_2
    y = ((pt_3.x * pt_2.x) - (pt_3.y * pt_1.x)) / -cross_product_1_2  # cross(pt_2, pt_1)
    return round(Pt2d(x, y), 12)

  def angle_bisector_for_triangle_abc(self, a, b, c):  # TODO RETEST
    """Compute the angle bisector point.
    Method: TODO add in the formula
    """
    dist_ba = self.distance_normalized(b, a)
    dist_ca = self.distance_normalized(c, a)
    ref_pt = (b-a) / dist_ba * dist_ca
    return ref_pt + (c-a) + a

  def perpendicular_bisector_for_triangle_ab(self, a, b):  # TODO RETEST
    """Compute the perpendicular bisector point.
    Method: TODO add in the formula
    """
    rotated_vector_ba = self.rotate_ccw_90_wrt_origin(b-a)  # code is a ccw turn. check formula
    return rotated_vector_ba + (a+b)/2

  def incircle_pt_for_triangle_abc_2(self, a, b, c):  # TODO RETEST
    """An alternative way to compute incircle. This one uses bisectors
    Method: TODO add in the formula
    """
    bisector_abc = self.angle_bisector_for_triangle_abc(a, b, c)
    bisector_bca = self.angle_bisector_for_triangle_abc(b, c, a)
    return self.triangle_circle_center_pt_abcd(a, bisector_abc, b, bisector_bca)

  def circumcenter_pt_of_triangle_abc_2(self, a, b, c):  # TODO RETEST
    """An alternative way to compute circumcenter. This one uses bisectors
    Method: TODO add in the formula
    """
    bisector_ab = self.perpendicular_bisector_for_triangle_ab(a, b)
    bisector_bc = self.perpendicular_bisector_for_triangle_ab(b, c)
    ab2, bc2 = (a+b)/2, (b+c)/2
    return self.triangle_circle_center_pt_abcd(ab2, bisector_ab, bc2, bisector_bc)

  def orthocenter_pt_of_triangle_abc_v2(self, a, b, c):  # TODO RETEST
```

```
    """Compute the orthogonal center of triangle abc.Z
    Method: TODO add in the formula
    """
    return a + b + c - self.circumcenter_pt_of_triangle_abc_2(a, b, c) * 2

  def perimeter_of_polygon_pts(self, pts: List[Pt2d]) -> float:
    """Compute summed pairwise perimeter of polygon in CCW ordering."""
    return fsum([self.distance_normalized(a, b) for a, b in pairwise_func(pts)])

  def signed_area_of_polygon_pts(self, pts: List[Pt2d]) -> float:
    """Compute sum of area of polygon, via shoelace method: half the sum of the pairwise
    cross-products."""  # see start of booklet for pairwise
    return fsum([self.cross_product(a, b) for a, b in pairwise_func(pts)]) / 2

  def area_of_polygon_pts(self, pts: List[Pt2d]) -> float:
    """Positive area of polygon using above method."""
    return abs(self.signed_area_of_polygon_pts(pts))

  def is_polygon_pts_convex(self, pts: List[Pt2d]) -> bool:
    """Determines if polygon is convex with options of allowing or disallowing collinearity.

    Complexity per call: Time: O(n), Space: O(1) ? maybe its O(n)
    Optimizations and Notes: pts[0] != pts[-1], Use iterators with zip instead of costly mod.
    """
    if len(pts) > 3:
      n, func = len(pts), self.point_c_rotation_wrt_line_ab
      rotations = {func(pts[i], pts[(i + 1) % n], pts[(i + 2) % n]) for i in range(n)}
      # return (len(rotations) == 1) and (CL not in rotations)  # use when CL not allowed
      return (CCW in rotations) != (CW in rotations)        # use when CL is allowed
    return False

  def pt_p_in_polygon_pts(self, pts: List[Pt2d], p: Pt2d) -> bool:
    """Determine if a point is in a polygon via, ray casting.

    Complexity per call: Time: O(n), Space: O(1)
    """
    ans, px, py = False, p.x, p.y
    for a, b in pairwise_func(pts):
      xi, yi = a.x, a.y
      xj, yj = b.x, b.y
      if min(yi, yj) <= py < max(yi, yj) and px < (xi + (xj - xi) * (py - yi) / (yj - yi)):
        ans = not ans
    return ans

  def pt_p_in_polygon_pts_alternative(self, pts: List[Pt2d], p: Pt2d) -> bool:
    """Determine if a point is in a polygon based on the sum of the angles.

    Complexity per call: Time: O(n), Space: O(1)
    """
    if len(pts) > 3:
      angle_sum = 0.0
      for a, b in pairwise_func(pts):
        angle = self.angle_point_c_wrt_line_ab(a, p, b)
        angle_sum += angle if self.point_c_rotation_wrt_line_ab(p, a, b) < CL else -angle
      return True if self.compare_ab(abs(angle_sum), pi) > 0 else False
    return False

  def pt_p_on_polygon_perimeter_pts(self, pts: List[Pt2d], p: Pt2d) -> bool:
    """Determine if a point is on the perimeter of a polygon simply via a distance check.

    Complexity per call: Time: O(n), Space: O(1)
    Optimizations: turn pts into a set and check that case first
    """
    return any(self.pt_on_line_segment_ab(a, b, p) for a, b in pairwise_func(pts)) or p in pts

  def pt_position_wrt_polygon_pts(self, pts: List[Pt2d], p: Pt2d) -> int:
    """Will determine if a point is in on or outside a polygon.

    Complexity per call: Time: O(n) Convex(log n), Space: O(1)
    Optimizations: use log n version if you need superfast, and it's a convex polygon
    Return: 0 for on, 1 for in, -1 for out
    """
    return (0 if self.pt_p_on_polygon_perimeter_pts(pts, p)
            else 1 if self.pt_p_in_polygon_pts(pts, p) else -1)

  def remove_collinear_points(self, pts: List[Pt2d]):
    """Removes all collinear points in O(n) time. MUTATES pts."""
    pts[:] = [pts[-1]] + [pt for pt in pts] + [pts[0]]
    pts[:] = [pts[i] for i in range(1, len(pts) - 1)
              if self.point_c_rotation_wrt_line_ab(pts[i - 1], pts[i], pts[i + 1]) != CL]

  def pt_p_in_convex_polygon_pts(self, pts: List[Pt2d], query_pt: Pt2d) -> bool:
    """For a convex Polygon we are able to search if point is in the polygon fast.
    Must be ordered in CounterClockWise ordering and no collinear points.

    Complexity per call: Time: O(log n), Space: O(1)
    Optimizations: pre-compute pts[mid] - pts[0] for all values then use cross product.
    """
    left, right, min_point = 1, len(pts) - 1, pts[0]
    while left < right:
      mid = 1 + ((left + right) // 2)
      if self.point_c_rotation_wrt_line_ab(min_point, pts[mid], query_pt) == 1:
        left = mid
      else:
        right = mid - 1
    if (self.point_c_rotation_wrt_line_ab(min_point, pts[left], query_pt) == -1
        or left == len(pts) - 1):
      return False
    return self.point_c_rotation_wrt_line_ab(pts[left], pts[left + 1], query_pt) >= 0

  def centroid_pt_of_convex_polygon(self, pts: List[Pt2d]) -> Pt2d:  # TODO RETEST
    """Compute the centroid of a convex polygon.

    Complexity per call: Time: O(n), Space: O(1)
    Optimizations:
    """
    ans = Pt2d(0, 0)
    for a, b in pairwise_func(pts):  # defined at start of booklet
      ans = ans + (a + b) * self.cross_product(a, b)
    return ans / (6.0 * self.signed_area_of_polygon_pts(pts))

  def is_polygon_pts_simple_quadratic(self, pts: List[Pt2d]) -> bool:
    """Brute force method to check if a polygon is simple. check all line pairs

    Complexity per call: Time: O(n^2), Space: O(n)
    """
    lines = [(a, b) for a, b in pairwise_func(pts)]
    return not any(self.is_segments_intersect_ab_to_cd(pt_a, pt_b, pt_c, pt_d)
                   for (pt_a, pt_b), (pt_c, pt_d) in combinations(lines, 2)
                   if pt_a != pt_d and pt_b != pt_c)  # avoids lines that neighbor each other

  def polygon_cut_from_line_ab(self, pts: List[Pt2d], left, right) -> List[Pt2d]:  # TODO RETEST
    """Method computes the left side polygon resulting from a cut from the line a-b.
    Method: Walk around the polygon and only take points that return CCW to line ab

    Complexity per call: Time: O(n), Space: O(n)
    Optimizations:
    """
    left_partition = []
    for u, v in pairwise_func(pts):
      rot_1 = self.point_c_rotation_wrt_line_ab(left, right, u)
      rot_2 = self.point_c_rotation_wrt_line_ab(left, right, v)
      if 0 >= rot_1:
        left_partition.append(u)
        if 0 == rot_1:
          continue
      if rot_1 * rot_2 < 0:  # CCW -1, CW 1 so tests if they are opposite ie lines intersect.
        left_partition.append(self.pt_line_seg_intersect_ab_to_cd(u, v, left, right))
    if left_partition and left_partition[0] != left_partition[-1]:
      left_partition.append(left_partition[0])
    return left_partition

  def convex_hull_monotone_chain(self, pts: List[Pt2d]) -> List[Pt2d]:
    """Compute convex hull of a list of points via Monotone Chain method. CCW ordering returned.

    Complexity per call: Time: O(nlog n), Space: final O(n), aux O(nlog n)
    Optimizations: can use heapsort for Space: O(n)[for set + heap] or O(1) [if we consume pts],
    Can also optimize out the append and pop with using a stack like index.
    """
    def func(points, cur_hull, min_size):
      for p in points:
        while (len(cur_hull) > min_size
               and self.point_c_rotation_wrt_line_ab(cur_hull[-2], cur_hull[-1], p) == CW):
          cur_hull.pop()
        cur_hull.append(p)
      cur_hull.pop()
    unique_points, convex_hull = sorted(set(pts)), []
    if len(unique_points) > 1:
```

```
    func(unique_points, convex_hull, 1)
    func(unique_points[::-1], convex_hull, 1 + len(convex_hull))
    return convex_hull + [convex_hull[0]]  # add the first point because lots of our methods
  return unique_points              # require the first and last point be the same

def rotating_caliper_of_polygon_pts(self, pts: List[Pt2d]) -> float:
  """Computes the max distance of two points in the convex polygon?

  Complexity per call: Time: O(nlog n) unsorted O(n) sorted, Space: O(1)
  Optimizations:
  """
  convex_hull = self.convex_hull_monotone_chain(pts)
  n, t, ans = len(convex_hull) - 1, 1, 0.0
  for i, (a, b) in enumerate(pairwise_func(convex_hull)):
    p = b - a
    while ((t + 1) % n != i
           and self.compare_ab(self.cross_product(p, convex_hull[(t + 1) % n] - a),
                               self.cross_product(p, convex_hull[t] - a)) >= 0):
      t = (t + 1) % n
    ans = max(ans, self.distance(a, convex_hull[t]))
    ans = max(ans, self.distance(b, convex_hull[t]))
  return sqrt(ans)

def closest_pair_helper(self, lo: int, hi: int, x_ord: List[Pt2d]) -> ClosestPair:
  """brute force function, for small range will brute force find the closet pair. O(n^2)"""
  closest_pair = (self.distance(x_ord[lo], x_ord[lo + 1]), x_ord[lo], x_ord[lo + 1])
  for pt_a, pt_b in combinations(x_ord[lo:hi], 2):  # 2 for every pair of combinations
    distance_ij = self.distance(pt_a, pt_b)
    if distance_ij < closest_pair[0]:
      closest_pair = (distance_ij, pt_a, pt_b)
  return closest_pair

def closest_pair_recursive(self, lo: int, hi: int,
                           x_ord: List[Pt2d], y_ord: List[Pt2d]) -> ClosestPair:
  """Recursive part of computing the closest pair. Divide by y recurse then do a special check

  Complexity per call T(n/2) halves each time, T(n/2) halves each call, O(n) at max tho
  Optimizations and notes: If not working use y_part_left and y_part_right again. I did add in
  the optimization of using y_partition 3 times over rather than having 3 separate lists
  also can remove compare_ab for direct compare
  """
  n = hi - lo
  if n < 32:  # base case, just brute force: powers of 2 between with 32 working the fastest.
    return self.closest_pair_helper(lo, hi, x_ord)
  left_len, right_len = lo + n - n // 2, lo + n // 2
  mid = round((x_ord[left_len].x + x_ord[right_len].x) / 2)
  partition_left, partition_right = [], []
  append_left, append_right = partition_left.append, partition_right.append
  for pt in y_ord:
    append_right(pt) if pt.x > mid else append_left(pt)
  best_left = self.closest_pair_recursive(lo, left_len, x_ord, partition_left)
  best_right = self.closest_pair_recursive(left_len, hi, x_ord, partition_right)
  best_pair = best_left if best_left[0] <= best_right[0] else best_right
  if self.compare_ab(best_pair[0], 0) == 0:
    return best_pair
  y_partition = [pt for pt in y_ord if best_pair[0] > (pt.x - mid) ** 2]
  for i, pt_a in enumerate(y_partition):
    a_y = pt_a.y
    for pt_b in y_partition[i+1:]:  # slicing seemed to run the fastest, range was second.
      dist_ij = a_y - pt_b.y
      if dist_ij ** 2 >= best_pair[0]:
        break
      dist_ij = self.distance(pt_a, pt_b)
      if dist_ij < best_pair[0]:
        best_pair = (dist_ij, pt_a, pt_b)
  return best_pair

def compute_closest_pair(self, pts: List[Pt2d]) -> ClosestPair:
  """Compute the closest pair of points in a set of points. method is divide and conqur

  Complexity per call Time: O(nlog n), Space O(nlog n)
  Optimizations: use c++ if too much memory, haven't found the way to do it without nlog n
  """
  x_ord = sorted(pts, key=lambda pt: pt.x)
  y_ord = sorted(pts, key=lambda pt: pt.y)
  return self.closest_pair_recursive(0, len(pts), x_ord, y_ord)

def delaunay_triangulation_slow(self, pts: List[Pt2d]) -> List[Triangle]:
  """A very slow version of  Delaunay Triangulation. Can beat the faster version when n small.

  Complexity per call Time: O(n^4), Space O(n)
  Optimizations: use c++ if too much memory, haven't found the way to do it without nlog n
  """
  n, ans = len(pts), []
  z_arr = [el.x ** 2 + el.y ** 2 for el in pts]
  x_arr = [el.x for el in pts]
  y_arr = [el.y for el in pts]
  for i in range(n - 2):
    for j in range(i + 1, n):
      for k in range(i + 1, n):
        if j == k:
          continue
        xn = ((y_arr[j] - y_arr[i]) * (z_arr[k] - z_arr[i])
              - (y_arr[k] - y_arr[i]) * (z_arr[j] - z_arr[i]))
        yn = ((x_arr[k] - x_arr[i]) * (z_arr[j] - z_arr[i])
              - (x_arr[j] - x_arr[i]) * (z_arr[k] - z_arr[i]))
        zn = ((x_arr[j] - x_arr[i]) * (y_arr[k] - y_arr[i])
              - (x_arr[k] - x_arr[i]) * (y_arr[j] - y_arr[i]))
        flag = zn < 0.0
        for m in range(n):
          if flag:
            flag = flag and (self.compare_ab((x_arr[m] - x_arr[i]) * xn +
                                             (y_arr[m] - y_arr[i]) * yn +
                                             (z_arr[m] - z_arr[i]) * zn, 0.0) <= 0)
          else:
            break
        if flag:
          ans.append(sort_3_elements(pts[i], pts[j], pts[k]))
  return ans

def is_pt_left_of_edge(self, pt: Pt2d, edge: QuadEdge) -> bool:
  """A helper function with a name to describe the action. Remove for speedup."""
  return CCW == self.point_c_rotation_wrt_line_ab(pt, edge.origin, edge.dest())

def is_pt_right_of_edge(self, pt: Pt2d, edge: QuadEdge) -> bool:
  """A helper function with a name to describe the action. Remove for speedup."""
  return CW == self.point_c_rotation_wrt_line_ab(pt, edge.origin, edge.dest())

def det3_helper(self, a1, a2, a3, b1, b2, b3, c1, c2, c3) -> Numeric:
  """A helper function for determining the angle. Remove for speedup."""
  return (a1 * (b2 * c3 - c2 * b3) -
          a2 * (b1 * c3 - c1 * b3) +
          a3 * (b1 * c2 - c1 * b2))

def is_in_circle(self, a: Pt2d, b: Pt2d, c: Pt2d, d: Pt2d) -> bool:
  """Expensive calculation function that determines if """
  a_dot = self.dot_product(a, a)
  b_dot = self.dot_product(b, b)
  c_dot = self.dot_product(c, c)
  d_dot = self.dot_product(d, d)
  det = -self.det3_helper(b.x, b.y, b_dot, c.x, c.y, c_dot, d.x, d.y, d_dot)
  det += self.det3_helper(a.x, a.y, a_dot, c.x, c.y, c_dot, d.x, d.y, d_dot)
  det -= self.det3_helper(a.x, a.y, a_dot, b.x, b.y, b_dot, d.x, d.y, d_dot)
  det += self.det3_helper(a.x, a.y, a_dot, b.x, b.y, b_dot, c.x, c.y, c_dot)
  return det > 0
  # use this if above doesn't work for what ever reason
  # def angle(l, mid, r):
  #   x = self.dot_product(l-mid, r-mid)
  #   y = self.cross_product(l-mid, r-mid)
  #   return atan2(x, y)
  # kek = angle(a, b, c) + angle(c, d, a) - angle(b, c, d) - angle(d, a, b)
  # return self.compare_ab(kek, 0.0) > 0

def build_triangulation(self, left: int, right: int, pts: List[Pt2d]) -> QuadEdgePair:
  if right - left + 1 == 2:
    res = self.quad_edges.make_edge(pts[left], pts[right])
    return res, res.rev()
  if right - left + 1 == 3:
    edge_a = self.quad_edges.make_edge(pts[left], pts[left + 1])
    edge_b = self.quad_edges.make_edge(pts[left + 1], pts[right])
    self.quad_edges.splice(edge_a.rev(), edge_b)
    sg = self.point_c_rotation_wrt_line_ab(pts[left], pts[left + 1], pts[right])
    if sg == 0:
      return edge_a, edge_b.rev()
    edge_c = self.quad_edges.connect(edge_b, edge_a)
    return (edge_a, edge_b.rev()) if sg == 1 else (edge_c.rev(), edge_c)
  mid = (left + right) // 2
  ldo, ldi = self.build_triangulation(left, mid, pts)
```

```
    rdi, rdo = self.build_triangulation(mid + 1, right, pts)
    while True:
      if self.is_pt_left_of_edge(rdi.origin, ldi):
        ldi = ldi.l_next()
        continue
      if self.is_pt_right_of_edge(ldi.origin, rdi):
        rdi = rdi.rev().o_next
        continue
      break
    base_edge_l = self.quad_edges.connect(rdi.rev(), ldi)
    if ldi.origin == ldo.origin:
      ldo = base_edge_l.rev()
    if rdi.origin == rdo.origin:
      rdo = base_edge_l
    while True:
      l_cand_edge = base_edge_l.rev().o_next
      if self.is_pt_right_of_edge(l_cand_edge.dest(), base_edge_l):
        while self.is_in_circle(base_edge_l.dest(), base_edge_l.origin,
                                l_cand_edge.dest(), l_cand_edge.o_next.dest()):
          temp_edge = l_cand_edge.o_next
          self.quad_edges.delete_edge(l_cand_edge)
          l_cand_edge = temp_edge
      r_cand_edge = base_edge_l.o_prev()
      if self.is_pt_right_of_edge(r_cand_edge.dest(), base_edge_l):
        while self.is_in_circle(base_edge_l.dest(), base_edge_l.origin,
                                r_cand_edge.dest(), r_cand_edge.o_prev().dest()):
          temp_edge = r_cand_edge.o_prev()
          self.quad_edges.delete_edge(r_cand_edge)
          r_cand_edge = temp_edge
      l_check = self.is_pt_right_of_edge(l_cand_edge.dest(), base_edge_l)
      r_check = self.is_pt_right_of_edge(r_cand_edge.dest(), base_edge_l)
      if not (l_check or r_check):
        break
      if ((not l_check)
          or (r_check and self.is_in_circle(l_cand_edge.dest(), l_cand_edge.origin,
                                            r_cand_edge.origin, r_cand_edge.dest()))):
        base_edge_l = self.quad_edges.connect(r_cand_edge, base_edge_l.rev())
      else:
        base_edge_l = self.quad_edges.connect(base_edge_l.rev(), l_cand_edge.rev())
    return ldo, rdo

  def delaunay_triangulation_fast(self, pts: List[Pt2d]) -> List[Triangle]:
    def add_helper():
      cur = edge
      while True:
        cur.used = True
        pts.append(cur.origin)
        edges.append(cur.rev())
        cur = cur.l_next()
        if cur == edge:
          return
    pts.sort()
    result = self.build_triangulation(0, len(pts) - 1, pts)
    edge, edges = result[0], [result[0]]
    while self.point_c_rotation_wrt_line_ab(edge.o_next.dest(), edge.dest(), edge.origin) < CL:
      edge = edge.o_next
    add_helper()
    pts, kek = [], 0
    while kek < len(edges):
      edge = edges[kek]
      kek += 1
      if not edge.used:
        add_helper()
    return [sort_3_elements(pts[i], pts[i + 1], pts[i + 2]) for i in range(0, len(pts), 3)]


##################################################


from itertools import takewhile


# constants can paste into code for speedup
GREATER_EQUAL = -1
GREATER_THAN = 0


class StringAlgorithms:
  def __init__(self):
    """Attributes declared here must be passed in or global if not used in class format."""
    self.text_len = self.pattern_len = 0
    self.prime_p = self.mod_m = 0
    self.text, self.pattern = '', ''

    self.back_table = []

    self.suffix_array = []
    self.text_ord, self.pattern_ord = [], []
    self.longest_common_prefix = []
    self.owner = []
    self.seperator_list = []

    self.math_algos = MathAlgorithms()
    self.hash_powers = []
    self.hash_h_values = []
    self.left_mod_inverse = []

  def kmp_preprocess(self, new_pattern):
    """Preprocess the pattern for KMP. TODO add a bit more to this description ?

    Complexity per call: Time O(m + m), Space: O(m)
    """
    pattern = new_pattern
    pattern_len = len(pattern)  # m = length of pattern
    back_table = [0] * (pattern_len + 1)
    back_table[0], j = -1, -1
    for i, character in enumerate(pattern):
      while j >= 0 and character != pattern[j]:
        j = back_table[j]
      j += 1
      back_table[i + 1] = j
    self.pattern, self.pattern_len, self.back_table = pattern, pattern_len, back_table

  def kmp_search_find_indices(self, text_to_search):
    """Search the text for the pattern we preprocessed.

    Complexity per call: Time O(n + m), Space: O(n + m)
    """
    ans, j = [], 0
    for i, character in enumerate(text_to_search):
      while j >= 0 and character != self.pattern[j]:
        j = self.back_table[j]
      j += 1
      if j == self.pattern_len:
        ans.append(1 + i - j)
        j = self.back_table[j]
    return ans

  def suffix_array_counting_sort(self, k, s_array, r_array):
    """Basic count sort for the radix sorting part of suffix arrays.

    Complexity per call. Time: O(n), T(6n), Space: O(n), S(2n)
    Switching to non itertools versions and non enumerating version can speed it up.
    """
    n = self.text_len
    maxi, tmp = max(255, n), 0
    suffix_array_temp, frequency_array = [0] * n, [0] * maxi
    frequency_array[0] = n - (n - k)  # allows us to skip k values, handled in the second loop
    for i in range(k, n):       # here we skip those k iterations
      frequency_array[r_array[i]] += 1
    for i in range(maxi):
      frequency_array[i], tmp = tmp, tmp + frequency_array[i]
    for suffix_i in s_array:
      pos = 0 if suffix_i + k >= n else r_array[suffix_i + k]
      suffix_array_temp[frequency_array[pos]] = suffix_i
      frequency_array[pos] += 1
    for i, value in enumerate(suffix_array_temp):
      s_array[i] = value

  def suffix_array_build_array(self, new_texts):
    """Suffix array construction on a list of texts. n = sum lengths of all the texts.

    Complexity per call: Time: O(nlog n), T(3n log n), Space: O(n), S(6n)
    Optimizations: remove take while, don't use list(map(ord, text)), remove the pairwise
    """
    num_strings = len(new_texts)
    new_text = ''.join([txt + chr(num_strings - i) for i, txt in enumerate(new_texts)])
    self.text_len = new_len = len(new_text)
```

```
    suffix_arr, rank_arr = [i for i in range(new_len)], [ord(c) for c in new_text]
    for power in takewhile(lambda x: 2 ** x < new_len, range(32)):  # iterate powers of 2
      k = 2 ** power
      self.suffix_array_counting_sort(k, suffix_arr, rank_arr)
      self.suffix_array_counting_sort(0, suffix_arr, rank_arr)
      rank_array_temp = [0] * new_len
      rank = 0
      for last, curr in pairwise_func(suffix_arr):  # suffix[i] = curr, suffix[i - 1] last
        rank = rank if (rank_arr[curr] == rank_arr[last]
                        and rank_arr[curr + k] == rank_arr[last + k]) else rank + 1
        rank_array_temp[curr] = rank
      rank_arr = rank_array_temp
      if rank_arr[suffix_arr[-1]] == new_len - 1:  # exit loop early optimization
        break
    self.suffix_array, self.text = suffix_arr, new_text
    self.text_ord = [ord(c) for c in new_text]  # optional used in the binary search
    self.seperator_list = [num_strings - i for i in range(len(new_texts))]  # optional owners

  def compute_longest_common_prefix(self):
    """After generating a suffix array you can use that to find the longest common pattern.

    Complexity per call: Time: O(n), T(4n), Space: O(n), S(3n)
    """
    permuted_lcp, phi = [0] * self.text_len, [0] * self.text_len
    phi[0], left = -1, 0
    for last, curr in pairwise_func(self.suffix_array):
      phi[curr] = last
    for i, phi_i in enumerate(phi):
      if phi_i == -1:
        permuted_lcp[i] = 0
        continue
      while (i + left < self.text_len
             and phi_i + left < self.text_len
             and self.text_ord[i + left] == self.text_ord[phi_i + left]):
        left += 1
      permuted_lcp[i] = left
      left = 0 if left < 1 else left - 1  # this replaced max(left - 1, 0)
    self.longest_common_prefix = [permuted_lcp[suffix] for suffix in self.suffix_array]

  def suffix_array_compare_from_index(self, offset):  # TODO RETEST
    """C style string compare to compare 0 is equal 1 is greater than -1 is less than.

    Complexity per call: Time: O(k) len of pattern, Space: O(1)
    """
    for i, num_char in enumerate(self.pattern_ord):
      if num_char != self.text_ord[offset + i]:
        return -1 if self.text_ord[offset + i] < num_char else 1
    return 0

  def suffix_array_binary_search(self, lo, hi, comp_val):  # TODO RETEST
    """Standard binary search. comp_val allows us to select how strict we are, > vs >=

    Complexity per call: Time: O(k log n) len of pattern, Space: O(1)
    """
    while lo < hi:
      mid = (lo + hi) // 2
      if self.suffix_array_compare_from_index(self.suffix_array[mid]) > comp_val:
        hi = mid
      else:
        lo = mid + 1
    return lo, hi

  def suffix_array_string_matching(self, new_pattern):  # TODO RETEST
    """Utilizing the suffix array we can search efficiently for a pattern. gives first and last
    index found for patterns.

    Complexity per call: Time: O(k log n), T(2(k log n)), Space: O(k)
    """
    self.pattern_ord = [ord(c) for c in new_pattern]  # line helps avoid repeated ord calls
    lo, _ = self.suffix_array_binary_search(0, self.text_len - 1, GREATER_EQUAL)
    if self.suffix_array_compare_from_index(self.suffix_array[lo]) != 0:
      return -1, -1
    _, hi = self.suffix_array_binary_search(lo, self.text_len - 1, GREATER_THAN)
    if self.suffix_array_compare_from_index(self.suffix_array[hi]) != 0:
      hi -= 1
    return lo, hi

  def compute_longest_repeated_substring(self):
    """The longest repeated substring is just the longest common pattern. Require lcp to be
    computed already. Returns the first longest repeat pattern, so for other ones implement a
    forloop.

    Complexity per call: Time: O(n), T(2n), Space: O(1)
    for optimization implement the physical forloop itself, however it's still O(n).
    """
    max_lcp = max(self.longest_common_prefix)
    return max_lcp, self.longest_common_prefix.index(max_lcp)

  def compute_owners(self):  # TODO RETEST
    """Used to compute the owners of each position in the text. O(n) time and space."""
    tmp_owner = [0] * self.text_len
    it = iter(self.seperator_list)
    seperator = next(it)
    for i, ord_value in enumerate(self.text_ord):
      tmp_owner[i] = seperator
      if ord_value == seperator:
        seperator = next(it, None)
    self.owner = [tmp_owner[suffix_i] for suffix_i in self.suffix_array]

  def compute_longest_common_substring(self):  # TODO RETEST
    """Computes the longest common substring between two strings. returns index, value pair.

    Complexity per call: Time: O(n), Space: O(1)
    Pre-Requirements: owner, and longest_common_prefix must be built (also suffix array for lcp)
    Variants: LCS pair from k strings, LCS between all k strings.
    """
    it = iter(self.longest_common_prefix)
    max_lcp_index, max_lcp_value = 0, next(it) - 1  # - 1 here since next(it) should return 0
    for i, lcp_value in enumerate(it, 1):
      if lcp_value > max_lcp_value and self.owner[i] != self.owner[i - 1]:
        max_lcp_index, max_lcp_value = i, lcp_value
    return max_lcp_index, max_lcp_value

  def compute_rolling_hash(self, new_text):  # TODO RETEST
    """For a given text compute and store the rolling hash. we use the smallest prime lower than
    2^30 since python gets slower after 2^30, p = 131 is a small prime below 256.

    Complexity per call: Time: O(n), T(4n),  Space O(n), mid-call S(6n), post call S(4n)
    """
    len_text, p, m = len(new_text), 131, 2**30 - 35  # p is prime m is the smallest prime < 2^30
    h_vals, powers, ord_iter = [0] * len_text, [0] * len_text, map(ord, new_text)
    powers[0], h_vals[0] = 1, 0
    for i in range(1, len_text):
      powers[i] = (powers[i - 1] * p) % m
    for i, power in enumerate(powers):
      h_vals[i] = ((h_vals[i-1] if i != 0 else 0) + (next(ord_iter) * power) % m) % m
    self.text = new_text
    self.hash_powers, self.hash_h_values = powers, h_vals
    self.prime_p, self.mod_m, self.math_algos = p, m, MathAlgorithms()
    self.left_mod_inverse = [pow(power, m-2, m) for power in powers]  # optional

  def hash_fast_log_n(self, left, right):  # TODO RETEST
    """Log n time calculation of rolling hash formula: h[right]-h[left] * mod_inverse(left).

    Complexity per call: Time: O(log mod_m), Space: O(1)
    """
    ans = self.hash_h_values[right]
    if left != 0:
      ans = ((ans - self.hash_h_values[left - 1]) * pow(self.hash_powers[left],
                                                        self.mod_m - 2,
                                                        self.mod_m)) % self.mod_m
    return ans

  def hash_fast_constant(self, left, right):  # TODO RETEST
    """Constant time calculation of rolling hash. formula: h[right]-h[left] * mod_inverse[left]

    Complexity per call: Time: O(1), Space: O(1)
    more uses: string matching in n + m or constant if you know that it's a set size
    kattis typo: uses a variant of this were you will expand on the code further.
    """
    ans = self.hash_h_values[right]
    if left != 0:
      ans = ((ans - self.hash_h_values[left - 1]) * self.left_mod_inverse[left]) % self.mod_m
    return ans


################################################
```

```python
class Matrix:
  """Optimization notes: prefixed names of form local_NAME are used to avoid expensive load_attr
  calls they can be replaced with self.NAME or passed in as global values,
  REASON: even in pypy it seems N^3 operations can be speed up by doing this.
  """
  def __init__(self, n, m, init_value: Num):
    self.matrix = [[init_value] * m for _ in range(n)]
    self.num_rows = n
    self.num_cols = m
    self.matrix_left = []
    self.matrix_right = []

  def prep_matrix_multiply(self, matrix_a, matrix_b, size_n):  # TODO RETEST
    """loads up the left and right matrices for a matrix multiply. O(n^2) but optimized."""
    self.matrix_left = [[el for el in row] for row in matrix_a]
    self.matrix_right = [[el for el in row] for row in matrix_b]
    self.matrix = [[0] * size_n for _ in range(size_n)]
    self.num_rows = self.num_cols = size_n

  def fast_copy(self, other):  # TODO RETEST
    """Quickly copy one matrix to another. MIGHT CHANGE SIZE, O(n^2), is fast copy tho."""
    other_mat = other.matrix
    self.matrix = [[el for el in row] for row in other_mat]

  def fill_matrix_from_row_col(self, new_matrix, row_offset: int, col_offset: int):  # TODO
     RETEST
    """fill our matrix from row and col offset with new_matrix, O(n^2). Doesn't change size."""
    local_matrix = self.matrix  # prefix optimization see class docs for more info
    for i, row in enumerate(new_matrix):
      for j, new_value in enumerate(row):
        local_matrix[i + row_offset][j + col_offset] = new_value

  def matrix_multiply_mod_a_times_b(self, multiplier_1, multiplier_2, mod_m):  # TODO RETEST
    """Performs (A*B)%mod_m on matrix A,B and mod=mod_m, for normal multiply just remove mod_m.

    Complexity per call: Time: O(n^3), T(n^3 + n^2), Space: (1), does require 3n^2 in memory tho
    """
    local_num_rows = self.num_rows  # prefix optimization see class docs for more info.
    self.prep_matrix_multiply(multiplier_1.matrix, multiplier_2.matrix, local_num_rows)
    local_matrix = self.matrix  # prefix optimization see class docs for more info.
    mat_a = self.matrix_left   # prefix optimization see class docs for more info.
    mat_b = self.matrix_right  # prefix optimization see class docs for more info.
    for i in range(local_num_rows):
      for k in range(local_num_rows):
        mat_a_ik = mat_a[i][k]
        if 0 != mat_a_ik:  # skip because we will just be adding 0 to local
          for j in range(local_num_rows):
            local_matrix[i][j] = (local_matrix[i][j] + mat_a_ik * mat_b[k][j]) % mod_m
    self.matrix_left, self.matrix_right = [], []   # optional? keep space small

  def set_identity(self):  # TODO RETEST
    """Used for pow and pow mod on matrices. O(n^2) but memset(0) lvl speed for python."""
    local_matrix, local_num_rows = self.matrix, self.num_rows
    local_matrix = [[0] * local_num_rows for _ in range(local_num_rows)]  # memset(0) in python
    for i in range(local_num_rows):
      local_matrix[i][i] = 1

  def get_best_sawp_row(self, row: int, col: int, local_matrix, local_num_rows):  # TODO RETEST
    """Find the best pivot row defined as the row with the highest absolute value.

    Complexity per call: Time: O(n), Space: O(1)
    """
    best, pos = 0.0, -1
    for i in range(row, local_num_rows):
      column_value = abs(local_matrix[i][col])  # compute and lookup the value only once
      if column_value > best:
        best, pos = column_value, i
    return pos

  def swap_rows(self, row_a: int, row_b: int, local_matrix):  # TODO RETEST
    """Swaps two rows a and b, via reference swapping so should be constant time.

    Complexity per call: Time: O(1)[or very fast O(n) like memset], Space: O(1)
    """
    local_matrix[row_a], local_matrix[row_b] = local_matrix[row_b], local_matrix[row_a]

  def row_divide(self, row: int, div: float, local_matrix):  # TODO RETEST
    """Applies a vector divide operation on the whole row, via optimised list comprehension.

    Complexity per call: Time: O(n), Space: during O(n), post O(1)
    """
    local_matrix[row] = [el/div for el in local_matrix[row]]  # use int divide if possible

  def row_reduce_helper(self, i: int, row: int, val: Num, local_matrix):  # TODO RETEST
    """Applies a vector row reduce to a single row, via optimised list comprehension.

    Complexity per call: Time: O(n), Space: during O(n), post O(1)
    """
    local_matrix[i] = [el - val * local_matrix[row][col]
                       for col, el in enumerate(local_matrix[i])]

  def row_reduce(self, row: int, col: int, row_begin: int):  # TODO RETEST
    """Applies the whole row reduction step to the matrix.

    Complexity per call: Time: O(n^2), Space: during O(n), post O(1)
    """
    local_matrix = self.matrix  # prefix optimization see class docs for more info
    for i in range(row_begin, self.num_rows):
      if i != row:
        self.row_reduce_helper(i, row, local_matrix[i][col], local_matrix)

  def row_reduce_2(self, row: int, col: int, determinant):  # TODO RETEST
    """Applies the whole row reduction step to both the matrix and its determinant."""
    determinant_matrix = determinant.matrix  # prefix optimization see class docs for more info
    local_matrix = self.matrix          # it is optional
    for i in range(self.num_rows):
      if i != row:
        const_value, local_matrix[i][col] = local_matrix[i][col], 0.0
        self.row_reduce_helper(i, row, const_value, local_matrix)
        determinant.row_reduce_helper(i, row, const_value, determinant_matrix)

  def get_augmented_matrix(self, matrix_b):  # TODO RETEST
    """Given matrix A and B return augmented matrix = A | B.

    Complexity per call: Time: O(n^2), Space: O(n^2)
    """
    augmented = Matrix(self.num_rows + matrix_b.num_rows,
                       self.num_cols + matrix_b.num_cols,
                       self.matrix[0][0])
    augmented.fill_matrix_from_row_col(self.matrix, 0, 0)
    augmented.fill_matrix_from_row_col(matrix_b.matrix, 0, self.num_cols)
    return augmented

  def get_determinant_matrix(self):  # TODO RETEST
    """Compute the determinant of a matrix and return the result.

     Complexity per call: Time: O(n^3), T(n^3 + n^2), Space: (n^2)
    """
    det_num_rows = self.num_rows
    determinant = Matrix(det_num_rows, self.num_cols, self.matrix[0][0])
    determinant.fast_copy(self)
    det_matrix = determinant.matrix
    r = 1.0
    for i in range(det_num_rows):
      for j in range(det_num_rows):
        while det_matrix[j][i] != 0:
          ratio = det_matrix[i][i] / det_matrix[j][i]
          for k in range(i, det_num_rows):
            det_matrix[i][k] = (det_matrix[i][k] - ratio * det_matrix[j][k])
            det_matrix[i][k], det_matrix[j][k] = det_matrix[j][k], det_matrix[i][k]
          r = -r
      r = r * det_matrix[i][i]
    return r


################################################


class MatrixAlgorithms:
  def __init__(self):
    pass

  def matrix_pow_base_exponent_mod(self, base: Matrix, exponent: int, mod_m: int) -> Matrix:
    """Modular exponentiation applied to square matrices. For normal pow omit mod_m.
    [translated from Competitive Programming 4 part 2 c++ book, in the math section]
    # TODO RETEST
    Complexity per call: Time: [big n]O(n^3 log p), [small n] O(log p), Space: O(n^2)
```

```
    Input:
      base: the matrix represent the base, nxn matrix
    Mutations and return:
      base get  mutated to keep up with the calculation.
      returns the resulting matrix from (b^e)%m
    """
    result = Matrix(base.num_rows, base.num_rows, 0)
    result.set_identity()
    while exponent:
      if exponent % 2 == 1:
        result.matrix_multiply_mod_a_times_b(result, base, mod_m)
      base.matrix_multiply_mod_a_times_b(base, base, mod_m)
      exponent //= 2
    return result

  def get_rank_via_reduced_row_echelon(self, aug_ab: Matrix) -> int:  # TODO RETEST
    """Method used is Gauss-jordan elimination, only partial pivoting.
    [translated from standford 2016 c++ acm icpc booklet]

    Complexity per call: Time: O(n^3), Space: O(n)
    Input:
      aug_ab: a nxm augmented matrix of A | b
    Mutations and return:
      a -> rref: a mutations into nxm matrix of rref
      returns the rank of the matrix.
    """
    local_num_rows, local_num_cols = aug_ab.num_rows, aug_ab.num_cols
    augmented_matrix = aug_ab.matrix
    rank = 0
    for col in range(local_num_cols):
      if rank == local_num_rows:
        break
      swap_row = aug_ab.get_best_sawp_row(rank, col, augmented_matrix, local_num_rows)
      if swap_row != -1:
        aug_ab.swap_rows(swap_row, rank, augmented_matrix)
        aug_ab.row_divide(rank, augmented_matrix[rank][col], augmented_matrix)
        aug_ab.row_reduce(rank, col, 0)
        rank += 1
    return rank

  def gauss_elimination(self, aug_ab: Matrix) -> int:  # TODO RETEST
    """Computes gauss with constraints, Ax=b | A -> nxn matrix, b -> nx1 matrix aug_ab = A | b.
     [translated from foreverbell 2014 c++ acm icpc cheat sheet repo]

    Complexity per call: Time: O(n^3), Space: O(n)
    More uses: solving systems of linear equations (AX=B), getting the rank

    Input:
      aug_ab = nxm matrix | m = n + 1
    Mutation and return:
      aug_ab -> X
      returns rank
    """
    local_num_rows, local_num_cols = aug_ab.num_rows, aug_ab.num_cols
    n = local_num_rows
    augmented_matrix = aug_ab.matrix
    rank = 0
    for col in range(local_num_cols):
      if rank == local_num_rows:
        break
      swap_row = aug_ab.get_best_sawp_row(rank, col, augmented_matrix, local_num_rows)
      if swap_row != -1:
        aug_ab.swap_rows(swap_row, rank, augmented_matrix)
        aug_ab.row_divide(rank, augmented_matrix[rank][col], augmented_matrix)
        aug_ab.row_reduce(rank, col, rank + 1)
        rank += 1
    for i in range(n - 1, -1, -1):
      for j in range(i):
        augmented_matrix[j][n] -= (augmented_matrix[i][n] * augmented_matrix[j][i])
        augmented_matrix[j][i] = 0
    return rank

  def gauss_jordan_elimination(self, a: Matrix, b: Matrix) -> float:  # TODO RETEST
    """Full pivoting. Mutates a and b [translated from standford 2016 c++ acm icpc booklet]

    Complexity per call: Time: O(n^3), Space: O(n), S(4n)
    More uses: solving systems of linear equations (AX=B), Inverting matrices (AX=I), and
    Computing determinants of square matrices.
    Input:
      a = nxn matrix
      b = nxm matrix
    Mutation:
      b -> X | X = nxm matrix
      a -> A^{-1} | nxn matrix
      returns determinant of a
    """
    n = a.num_rows
    matrix_a = a.matrix
    matrix_b = b.matrix
    det = 1.0
    i_row, i_col, i_piv_j, i_piv_k = [0] * n, [0] * n, set(range(n)), set(range(n))
    for i in range(n):
      pj, pk = -1, -1
      for j in i_piv_j:
        for k in i_piv_k:
          if pj == -1 or abs(matrix_a[j][k]) > abs(matrix_a[pj][pk]):
            pj, pk = j, k
      i_piv_j.remove(pk)
      i_piv_k.remove(pk)
      a.swap_rows(pj, pk, matrix_a)
      b.swap_rows(pj, pk, matrix_b)
      if pj != pk:
        det = -det
      i_row[i], i_col[i] = pj, pk
      div = matrix_a[pk][pk]
      det /= div
      matrix_a[pk][pk] = 1.0
      a.row_divide(pk, div, matrix_a)
      b.row_divide(pk, div, matrix_b)
      a.row_reduce_2(pk, pk, b)
    for p in range(n - 1, -1, -1):
      if i_row[p] != i_col[p]:
        i_row_p, i_col_p = i_row[p], i_col[p]
        for k in range(n):
          matrix_a[k][i_row_p], matrix_a[k][i_col_p] = (matrix_a[k][i_col_p], matrix_a[k][
     i_row_p])
    return det
```