from queue import Queue

import math

import random

random.seed(55)

class Tree:

    root = None

    all\_nodes = []

    minimax\_counter = 0

    alphabeta\_counter = 0

    def \_\_init\_\_(self, n, notes):

        index = 0

        self.root = Node(index, -1)

        index += 1

        q = Queue()

        q.put(self.root)

        self.all\_nodes.append(self.root)

        for i in range(2\*n):

            q\_temp = Queue()

            while not q.empty():

                node = q.get()

                if node.child == None:

                    node.child = []

                for i in range(notes):

                    child = Node(index, node.index)

                    q\_temp.put(child)

                    self.all\_nodes.append(child)

                    node.child.append(child)

                    index += 1

            q = q\_temp

    def traverse(self):

        q = Queue()

        q.put(self.root)

        self.root.visited = True

        while not q.empty():

            u = q.get()

            print('parent:', u.index, 'Child:', end = ' ')

            if u.child != None:

                for child in u.child:

                    if not child.visited:

                        child.visited = True

                        q.put(child)

                        print(child.index, end = ' ')

            print()

        reset()

    def randomize\_leaf(self, min\_range, max\_range):

        q = Queue()

        q.put(self.root)

        self.root.visited = True

        while not q.empty():

            u = q.get()

            if u.child != None:

                for child in u.child:

                    if not child.visited:

                        child.visited = True

                        q.put(child)

            else:

                u.val = random.randint(min\_range, max\_range)

        self.reset()

    def minimax\_util(self, position, depth, maxP):

        if depth == 0:

            self.minimax\_counter += 1

            return position.val

        if maxP:

            maxEval = -math.inf

            for child in position.child:

                val = self.minimax\_util(child, depth - 1, False)

                maxEval = max(maxEval, val)

            return maxEval

        else:

            minEval = math.inf

            for child in position.child:

                val = self.minimax\_util(child, depth - 1, True)

                minEval = min(minEval, val)

            return minEval

    def minimax(self, position, depth, maxP):

        self.minimax\_counter = 0

        return (self.minimax\_util(position, depth, maxP), self.minimax\_counter)

    def alphabeta\_util(self, position, depth, alpha, beta, maxP):

        if depth == 0:

            self.alphabeta\_counter += 1

            return position.val

        if maxP:

            maxEval = -math.inf

            for child in position.child:

                val = self.alphabeta\_util(child, depth - 1, alpha, beta, False)

                maxEval = max(maxEval, val)

                alpha = max(alpha, val)

                if beta <= alpha:

                    break

            return maxEval

        else:

            minEval = math.inf

            for child in position.child:

                val = self.alphabeta\_util(child, depth - 1, alpha, beta, True)

                minEval = min(minEval, val)

                beta = min(beta, val)

                if beta <= alpha:

                    break

            return minEval

    def alphabeta(self, position, depth, alpha, beta, maxP):

        self.alphabeta\_counter = 0

        return (self.alphabeta\_util(position, depth, alpha, beta, maxP), self.alphabeta\_counter)

    def reset(self):

        for node in self.all\_nodes:

            node.visited = False

class Node:

    index = -1

    parent = -1

    visited = False

    child = None

    val = 1

    def \_\_init\_\_(self, index, parent):

        self.index = index

        self.parent = parent

n = int(input())

branch = int(input())

inp = input().split()

min\_range = int(inp[0])

max\_range = int(inp[1])

depth = 2 \* n

terminal\_states = branch \*\* depth

tree = Tree(n, branch)

tree.randomize\_leaf(min\_range, max\_range)

minimax = tree.minimax(tree.root, depth, True)

alphabeta = tree.alphabeta(tree.root, depth, -math.inf, math.inf, True)

print('Depth:', depth)

print('Brach:', branch)

print('Terminal States:', terminal\_states)

print('Maximum amount:', minimax[0])

print('Comparisons:', minimax[1])

print('Comparisons:', alphabeta[1])