

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ**

**«ДОНСКОЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»**

**(ДГТУ)**

Факультет: Информатика и вычислительная техника

Кафедра: КБИС

**Отчёт по лабораторным работам**

**по дисциплине “Методы программирования”**

Выполнил – ст. гр. ВКБ31 Котелевец К.А.

Ростов-на-Дону

2025

**Лабораторная работа 1**

**Задача 1: Гоблины и шаманы**

Гоблины Мглистых гор очень любях ходить к своим шаманам. Так как гоблинов много, к шаманам часто образуются очень длинные очереди. А поскольку много гоблинов в одном месте быстро образуют шумную толку, которая мешает шаманам проводить сложные медицинские манипуляции, последние решили установить некоторые правила касательно порядка в очереди.

Обычные гоблины при посещении шаманов должны вставать в конец очереди. Привилегированные же гоблины, знающие особый пароль, встают ровно в ее середину, причем при нечетной длине очереди они встают сразу за центром.

Так как гоблины также широко известны своим непочтительным отношением ко всяческим правилам и законам, шаманы попросили вас написать программу, которая бы отслеживала порядок гоблинов в очереди.

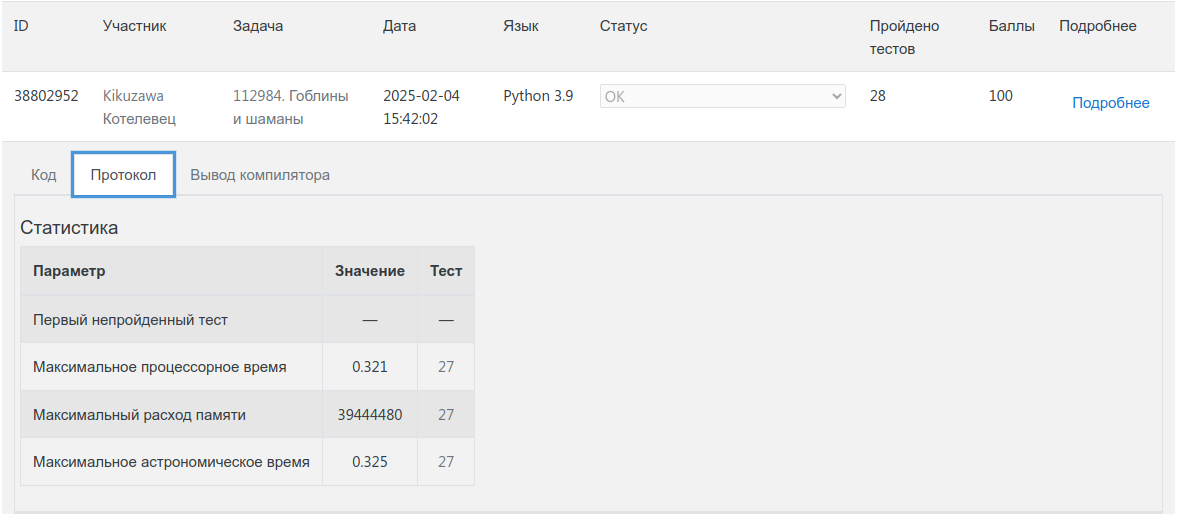
**Входные данные**

В первой строке входных данный записано число N (1≤N≤105) - количество запросов к программе. Следующие N строк содержат описание запросов в формате:

* "+ i" - гоблин с номером i (1≤i≤N) встает в конец очереди.
* "\* i" - привилегированный гоблин с номером i встает в середину очереди.
* "-" - первый гоблин из очереди уходит к шаманам. Гарантируется, что на момент такого запроса очередь не пуста.

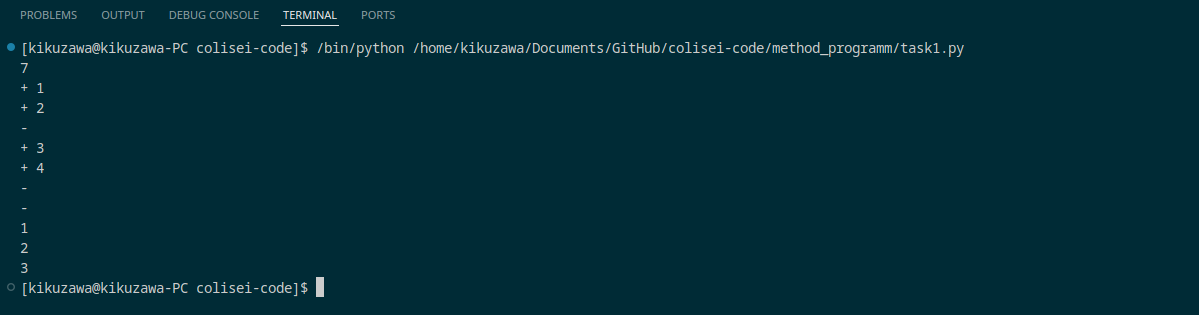
#### Выходные данные

Для каждого запроса типа "-" программа должна вывести номер гоблина, который должен зайти к шаманам.



Листинг программы:

|  |
| --- |
| from collections import deque  from typing import Sequence, List, Iterable  **def** process\_queue(actions: Sequence[Sequence[str]]) -> Iterable[str]:  q1 = deque()  q2 = deque()  results: List[str] = []  for action in actions:  if action[0] == '+':  q2.append(action[1])  elif action[0] == '\*':  q2.appendleft(action[1])  else:  results.append(q1.popleft())  if len(q1) < len(q2):  q1.append(q2.popleft())  return results  **def** main() -> None:  n: int = int(input())  actions: List[List[str]] = []  for \_ in range(n):  action: List[str] = input().split()  actions.append(action)  results: Iterable[str] = process\_queue(actions)  for result in results:  print(result)  if \_\_name\_\_ == "\_\_main\_\_":  main() |



**Задача 2: Баржа**

На барже располагается K грузовых отсеков. В каждый отсек можно поместить некоторое количество бочек с одним из 10 000 видов топлива. Причём извлечь бочку из отсека можно лишь в случае, если все бочки, помещённые в этот отсек после неё, уже были извлечены. Таким образом в каждый момент времени в каждом непустом отсеке имеется ровно одна бочка, которую можно извлечь не трогая остальных. Будем называть такие бочки крайними.

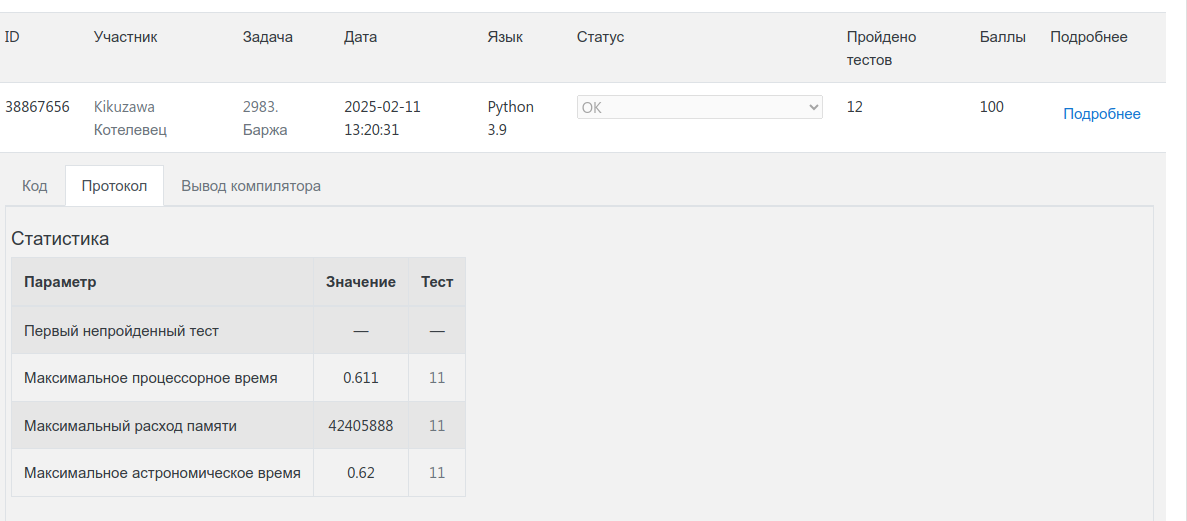
Изначально баржа пуста. Затем она последовательно проплывает через N доков, причём в каждом доке на баржу либо погружается бочка с некоторым видом топлива в некоторый отсек, либо выгружается крайняя бочка из некоторого отсека. Однако, если указанный отсек пуст, либо если выгруженная бочка содержит не тот вид топлива, который ожидалось, следует зафиксировать ошибку. Если на баржу оказывается погружено более P бочек или если после прохождения всех доков она не стала пуста, следует также зафиксировать ошибку. От вас требуется либо указать максимальное количество бочек, которые одновременно пребывали на барже либо зафиксировать ошибку.

**Входные данные**

В первой строке три целых числа N, K и P (1 ≤ N, K, P ≤ 100 000). Далее следует N строк с описанием действия, выполняемого в очередном доке. Если в нём происходит погрузка, то строка имеет вид «+ A B», где A — номер отсека, в который помещается бочка, а B — номер вида топлива в ней. Если же док занимается разгрузкой, то строка имеет вид «- A B», где A — номер отсека, из которого извлекается бочка, а B — номер ожидаемого вида топлива.

**Выходные данные**

Вывести либо одно число, равное искомому максимуму в случае безошибочного прохождения баржой маршрута, либо вывести слово «Error» в противном случае.



Листинг программы:

|  |
| --- |
| from collections import deque  from typing import List, Deque, Iterable  from dataclasses import dataclass  @dataclass(frozen=True)  class Action:  operation: str # Операция: "+" для погрузки, "-" для разгрузки  cell\_index: int # Номер отсека (1-based индекс)  fuel\_type: str # Тип топлива в бочке  def process\_docks(cells: int, max\_len: int, actions: Iterable[Action]):  max\_tanks: int = 0 # Максимальное количество бочек на барже  current\_tanks: int = 0 # Текущее количество бочек на барже  error: bool = False # Флаг ошибки  # Инициализация отсеков: каждый отсек представлен деком (очередью)  cell: List[Deque[str]] = [deque() for \_ in range(cells)]  # Обработка каждого действия  for action in actions:  if action.operation == "+": # Погрузка бочки  cell\_index: int = action.cell\_index - 1 # Преобразуем в 0-based индекс  cell[cell\_index].append(action.fuel\_type) # Добавляем бочку в отсек  current\_tanks += 1 # Увеличиваем счетчик бочек  # Проверка на превышение максимального количества бочек  if current\_tanks > max\_len:  error = True  break  # Обновляем максимальное количество бочек  max\_tanks = max(max\_tanks, current\_tanks)  else: # Разгрузка бочки  cell\_index: int = action.cell\_index - 1 # Преобразуем в 0-based индекс  # Проверка на пустой отсек или несоответствие типа топлива  if not cell[cell\_index] or cell[cell\_index].pop() != action.fuel\_type:  error = True  break  current\_tanks -= 1 # Уменьшаем счетчик бочек  # Возвращаем -1, если баржа не пуста или произошла ошибка, иначе возвращаем максимальное количество бочек  return -1 if current\_tanks > 0 or error else max\_tanks  def main():  # Чтение входных данных  request: List[str] = input().split()  docks: int = int(request[0]) # Количество доков  cells: int = int(request[1]) # Количество отсеков  max\_len: int = int(request[2]) # Максимальное количество бочек на барже  # Создание списка действий  actions: List[Action] = []  for \_ in range(docks):  action\_data: List[str] = input().split()  action = Action(operation=action\_data[0], cell\_index=int(action\_data[1]), fuel\_type=action\_data[2])  actions.append(action)  # Обработка действий и получение результата  result: int = process\_docks(cells, max\_len, actions)  # Вывод результата  print("Error" if result == -1 else result)  main() |

**Задание 3. Реклама**

Фирма NNN решила транслировать свой рекламный ролик в супермаркете XXX. Однако денег, запланированных на рекламную кампанию, хватило лишь на две трансляции ролика в течение одного рабочего дня. И при этом обязательно транслировать ровно два рекламных ролика в день.

Фирма собрала информацию о количестве покупателей в каждый момент некоторого дня. Менеджер по рекламе предположил, что и на следующий день покупатели будут приходить и уходить ровно в те же моменты времени.

Помогите ему определить моменты времени, когда нужно включить трансляцию рекламных роликов, чтобы как можно большее количество покупателей прослушало ролик.

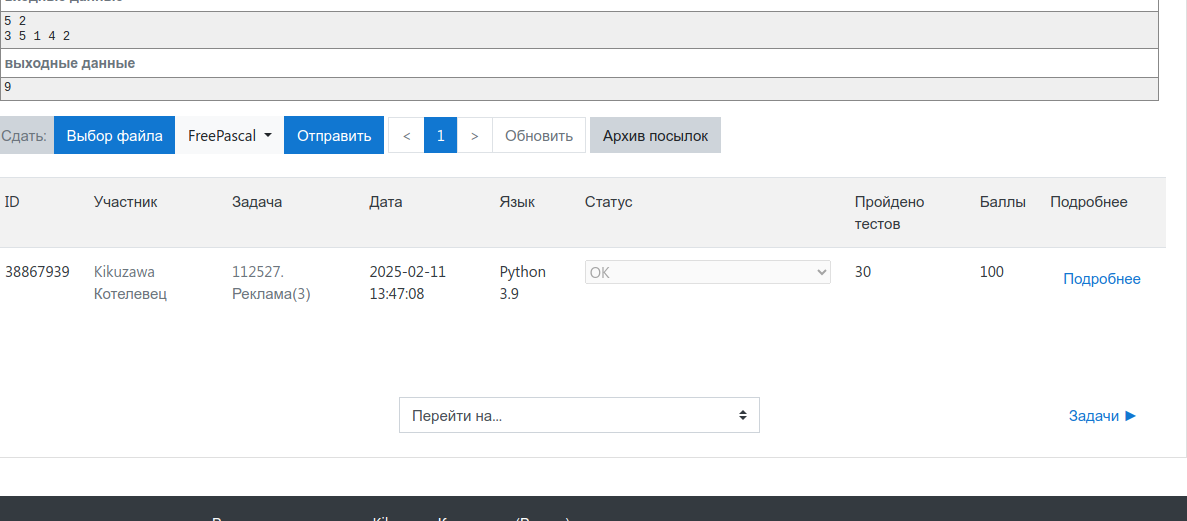
Ролик длится ровно одну единицу времени. Для каждого момента времени известно количество покупателей, находящихся в магазине в этот момент. Между концом первой рекламы и началом следующей должна пройти как минимум *К*- 1 единица времени.

**Входные данные**

Первая строка содержит два числа *N*(моментов времени) и *K*(время совершения покупки одним покупателем). Гарантируется, что *N*> *K.*( *N, K*≤ 200 000 ). Во второй строке содержится *N*чисел *a i*— количество покупателей в момент времени *i*. ( 0 ≤ *a i*≤ 2 000 000 000 ).

**Выходные данные**

Выведите единственное число — количество просмотревших рекламу покупателей.



Листинг программы

|  |
| --- |
| from itertools import combinations, product  from typing import List, Set, Iterable, Tuple  def find\_best\_broadcast(n: int, k: int, customer\_counts: List[int]) -> int:  """  Находит максимальное количество покупателей, которые увидят два рекламных ролика,  транслируемых с учетом ограничений на время между показами.  :param n: Количество моментов времени.  :param k: Минимальное время между окончанием первого ролика и началом второго.  :param customer\_counts: Список количества покупателей в каждый момент времени.  :return: Максимальное количество покупателей, которые увидят оба ролика.  """  # Находим максимальное количество покупателей в любой момент времени  max\_customers: int = max(customer\_counts)  # Находим все моменты времени, когда количество покупателей максимально  max\_customers\_times: Set[int] = {index for index, value in enumerate(customer\_counts)  if value == max\_customers}  # Проверяем, есть ли два момента с максимальным количеством покупателей,  # между которыми прошло достаточно времени  for time1, time2 in combinations(max\_customers\_times, 2):  if abs(time1 - time2) > k - 1:  return max\_customers \* 2 # Если да, возвращаем удвоенное максимальное значение  # Если таких моментов нет, ищем пару (максимальный момент + другой момент),  # где между ними прошло достаточно времени  all\_possible\_pairs: Iterable[Tuple[int, int]] = product(  max\_customers\_times,  range(len(customer\_counts))  )  # Находим второе по величине количество покупателей, которое можно использовать  # в паре с максимальным, соблюдая ограничение по времени  second\_max\_customers: int = max(  map(  lambda pair: customer\_counts[pair[1]],  filter(  lambda pair: pair[0] != pair[1] and abs(pair[0] - pair[1]) > k - 1,  all\_possible\_pairs  )  ),  default=0  )  # Если нашли подходящую пару, возвращаем сумму максимального и второго по величине  if second\_max\_customers > 0:  return max\_customers + second\_max\_customers  # Если не нашли подходящих пар, перебираем все возможные пары моментов времени  # с учетом ограничения на минимальное время между показами  max\_viewers\_sum: int = 0  for first\_ad\_time in range(n - k):  second\_ad\_time: int = first\_ad\_time + k  if second\_ad\_time < n:  viewers\_sum: int = customer\_counts[first\_ad\_time] + max(customer\_counts[second\_ad\_time:n])  max\_viewers\_sum = max(max\_viewers\_sum, viewers\_sum)  return max\_viewers\_sum  def main():  """  Основная функция для ввода данных и вывода результата.  """  # Ввод данных  n, k = map(int, input().split())  customer\_counts: List[int] = list(map(int, input().split()))    # Вычисление результата  result: int = find\_best\_broadcast(n, k, customer\_counts)    # Вывод результата  print(result)  if \_\_name\_\_ == "\_\_main\_\_":  main() |

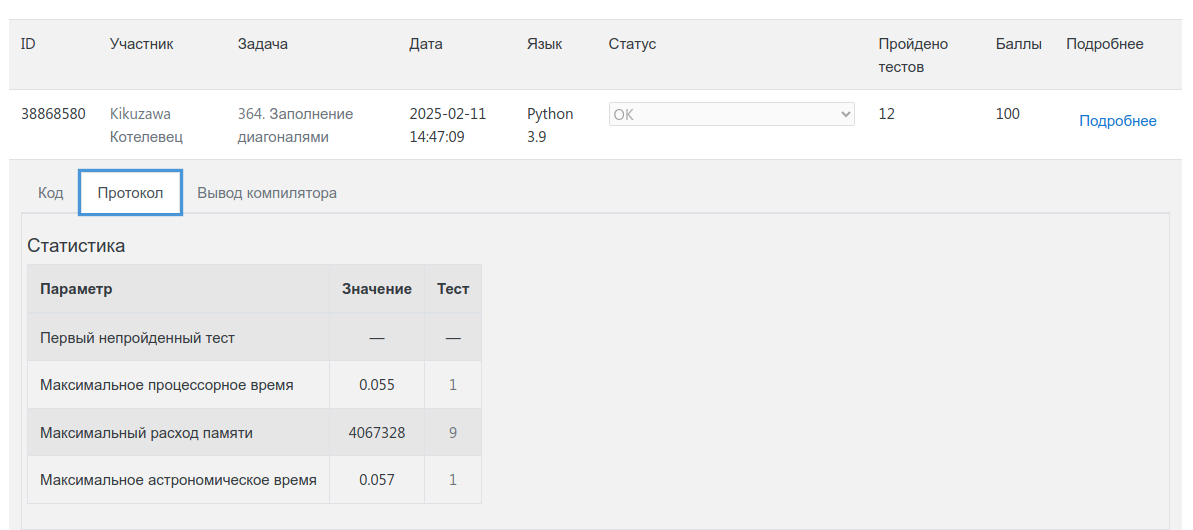
**Задание 4.**  **Заполнение диагоналями**  
 Даны числа n и m. Создайте массив A[n][m] и заполните его, как показано на примере.

**Входные данные**

Программа получает на вход два числа n и m.

**Выходные данные**

Программа должна вывести  полученный массив.



Листинг программы

|  |
| --- |
| from typing import List  def fill\_matrix(n: int, m: int) -> List[List[int]]:  """  Создает и заполняет двумерный массив (матрицу) размером n x m по диагоналям.  :param n: Количество строк в матрице.  :param m: Количество столбцов в матрице.  :return: Заполненная матрица.  """  # Создаем пустую матрицу размером n x m, заполненную нулями  matrix: List[List[int]] = [[0 for \_ in range(m)] for \_ in range(n)]  # Счетчик для заполнения матрицы значениями  count: int = 0  # Перебираем все диагонали матрицы  for index\_of\_diagonal in range(n + m - 1):  # Определяем начальную точку для заполнения текущей диагонали  start\_col: int  start\_row: int  if index\_of\_diagonal < m:  # Если диагональ начинается в верхней строке  start\_col = index\_of\_diagonal  start\_row = 0  else:  # Если диагональ начинается в последнем столбце  start\_col = m - 1  start\_row = index\_of\_diagonal - m + 1  # Заполняем текущую диагональ  while start\_col >= 0 and start\_row < n:  matrix[start\_row][start\_col] = count  count += 1  start\_col -= 1 # Двигаемся влево по столбцам  start\_row += 1 # Двигаемся вниз по строкам  return matrix  def main():  """  Основная функция для ввода данных и вывода результата.  """  # Ввод данных: количество строк (n) и столбцов (m)  n, m = map(int, input().strip().split())    # Заполнение матрицы  matrix = fill\_matrix(n, m)    # Вывод матрицы  for row in matrix:  # Форматированный вывод каждой строки матрицы  print(' '.join(f'{num:3d}' for num in row))  if \_\_name\_\_ == "\_\_main\_\_":  main() |

**Задание 5. Правильная скобочная последовательность**

Рассмотрим последовательность, состоящую из круглых, квадратных и фигурных скобок. Программа дожна определить, является ли данная скобочная последовательность правильной.

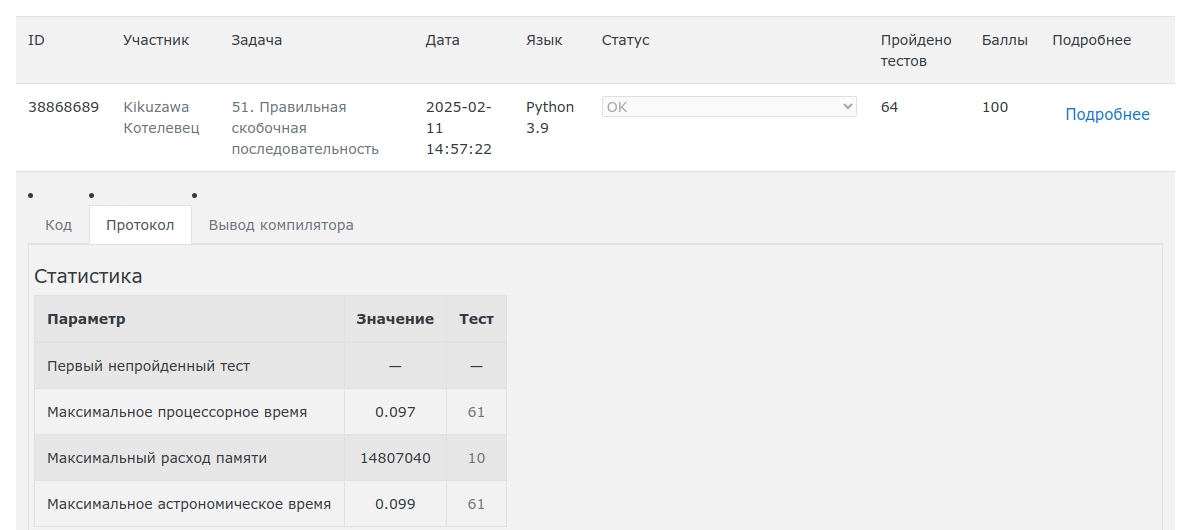
Пустая последовательность явлется правильной. Если A – правильная, то последовательности (A), [A], {A} – правильные. Если A и B – правильные последовательности, то последовательность AB – правильная.

**Входные данные**

В единственной строке записана скобочная последовательность, содержащая не более 100000 скобок.

**Выходные данные**

Если данная последовательность правильная, то программа должна вывести строку yes, иначе строку no.



Листинг программы:

|  |
| --- |
| # Импортируем необходимые модули для работы со стеком и типами данных  from collections import deque  from typing import Dict  def is\_balanced(sequence):  """  Проверяет, является ли скобочная последовательность правильной согласно условиям задачи:  - пустая последовательность считается правильной  - если A правильная, то (A), [A], {A} тоже правильные  - если A и B правильные, то AB тоже правильное    Args:  sequence (str): строка из скобок длиной не более 100000 символов    Returns:  bool: True если последовательность правильная, False иначе  """  # Создаем стек для хранения открывающих скобок  stack: deque[str] = deque()    # Словарь соответствия закрывающих и открывающих скобок  matching\_brackets: Dict[str, str] = {')': '(', '}': '{', ']': '['}  # Проходим по каждому символу в последовательности  for char in sequence:  # Если встретили открывающую скобку - добавляем её в стек  if char in matching\_brackets.values():  stack.append(char)  # Если встретили закрывающую скобку  elif char in matching\_brackets:  # Проверяем, что стек не пустой и верхняя скобка совпадает с текущей  if not (stack and stack[-1] == matching\_brackets[char]):  return False # Последовательность неправильная  stack.pop() # Убираем соответствующую открывающую скобку из стека  # Последовательность правильная, если стек пуст (все скобки закрыты)  return not bool(stack)  def main():  """  Основная функция программы для обработки входных данных.  Читает строку из скобок и выводит результат проверки.  """  # Читаем последовательность скобок из входного потока  sequence = input()    # Выводим результат в требуемом формате  print("yes" if is\_balanced(sequence) else "no")  # Запускаем программу  main() |

**Задание 6. Сортировка вагонов**

К тупику со стороны пути 1 (см. рисунок) подъехал поезд. Разрешается отцепить от поезда один или сразу несколько первых вагонов и завезти их в тупик (при желании, можно даже завезти в тупик сразу весь поезд). После этого часть из этих вагонов вывезти в сторону пути 2. После этого можно завезти в тупик еще несколько вагонов и снова часть оказавшихся вагонов вывезти в сторону пути 2. И так далее (так, что каждый вагон может лишь один раз заехать с пути 1 в тупик, а затем один раз выехать из тупика на путь 2). Заезжать в тупик с пути 2 или выезжать из тупика на путь 1 запрещается. Нельзя с пути 1 попасть на путь 2, не заезжая в тупик.

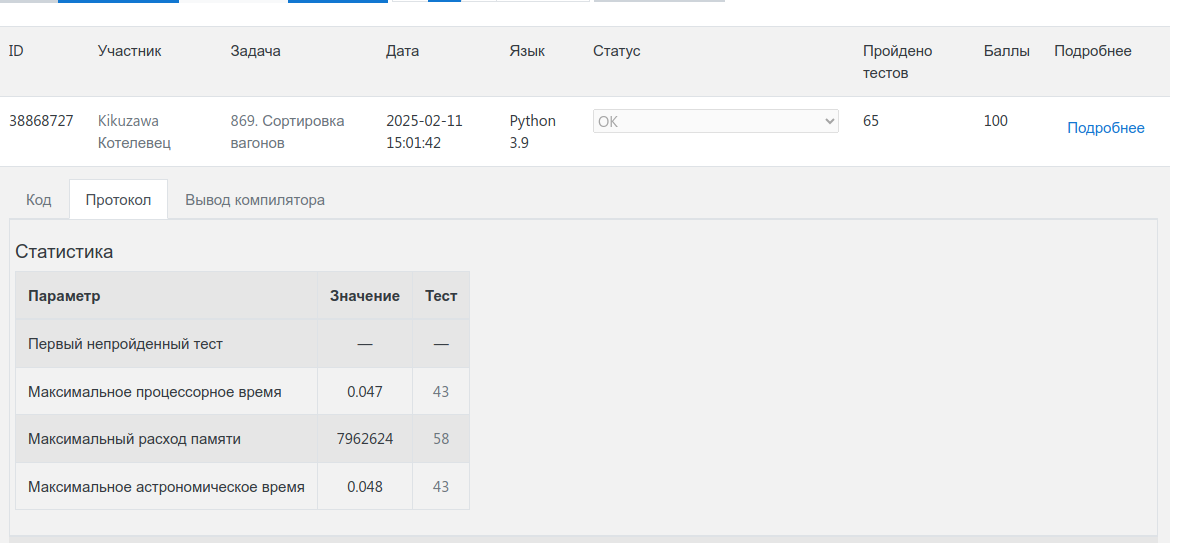
Известно, в каком порядке изначально идут вагоны поезда. Требуется с помощью указанных операций сделать так, чтобы вагоны поезда шли по порядку (сначала первый, потом второй и т.д., считая от головы поезда, едущего по пути 2 в сторону от тупика). Напишите программу, определяющую, можно ли это сделать.

**Входные данные**

Вводится число *N* — количество вагонов в поезде (1≤*N*≤100). Дальше идут номера вагонов в порядке от головы поезда, едущего по пути 1 в сторону тупика. Вагоны пронумерованы натуральными числами от 1 до *N*, каждое из которых встречается ровно один раз.

**Выходные данные**

Если сделать так, чтобы вагоны шли в порядке от 1 до *N*, считая от головы поезда, когда поезд поедет по пути 2 из тупика, можно, выведите сообщение YES, если это сделать нельзя, выведите NO.



Листинг программы:

|  |
| --- |
| from collections import deque  from typing import List, Iterable  def can\_sort\_train(n, wagons):  # Используем deque для стека тупика и стека пути 2  dead\_end\_stack: deque[int] = deque() # Стек для тупика  output\_track\_stack: deque[int] = deque() # Стек для пути 2  for wagon in wagons:  # Перемещаем вагоны из тупика на путь 2, если следующий вагон в порядке  while dead\_end\_stack and dead\_end\_stack[-1] == len(output\_track\_stack) + 1:  output\_track\_stack.append(dead\_end\_stack.pop())  # Проверяем, можно ли сразу отправить вагон на путь 2  if wagon == len(output\_track\_stack) + 1:  output\_track\_stack.append(wagon)  else:  # Если нельзя, отправляем вагон в тупик  dead\_end\_stack.append(wagon)  # Проверяем оставшиеся вагоны в тупике  while dead\_end\_stack and dead\_end\_stack[-1] == len(output\_track\_stack) + 1:  output\_track\_stack.append(dead\_end\_stack.pop())  # Если все вагоны успешно перемещены на путь 2, возвращаем True  return len(output\_track\_stack) == n  4  def main():  # Вводим количество вагонов  n: int = int(input())  # Вводим порядок вагонов на пути 1  wagons: List[int] = list(map(int, input().split()))  # Проверяем, можно ли отсортировать вагоны и выводим результат  print("YES" if can\_sort\_train(n, wagons) else "NO")  main() |

**Задание 7. Парикмахерская**

В парикмахерской работают три мастера. Каждый тратит на одного клиента ровно полчаса, а затем сразу переходит к следующему, если в очереди кто-то есть, либо ожидает, когда придет следующий клиент.

Даны времена прихода клиентов в парикмахерскую (в том порядке, в котором они приходили). Требуется для каждого клиента указать время, когда он выйдет из парикмахерской.

**Входные данные**

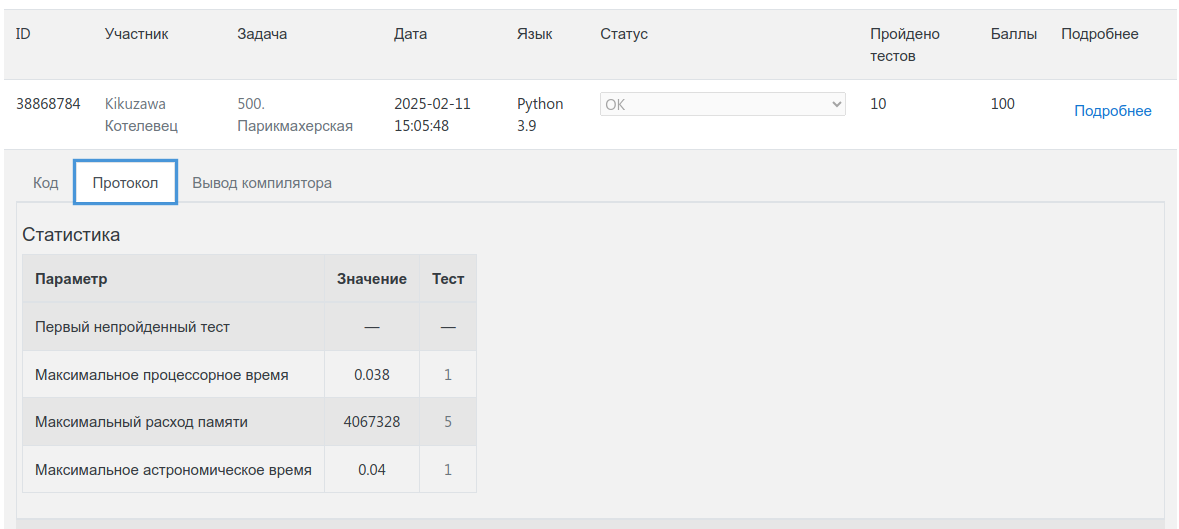
В первой строке вводится натуральное число *N*, не превышающее 100 – количество клиентов.

*N строках вводятся времена прихода клиентов – по два числа, обозначающие часы и минуты (часы – от 0 до 23, минуты – от 0 до 59). Времена указаны в порядке возрастания (все времена различны).*

*Гарантируется, что всех клиентов успеют обслужить до полуночи.*

***Выходные данные***

*Требуется вывести N пар чисел: времена выхода из парикмахерской 1-го, 2-го, …, N-го клиента (часы и минуты).*



Листинг программы

|  |
| --- |
| from typing import List, Tuple, cast, Iterable  from collections import deque  def calculate\_exit\_times(arrival\_times):  # Время окончания работы каждого мастера (в минутах с начала суток)  workers\_time: List[int] = [0, 0, 0]  # Очередь для хранения времени выхода клиентов (часы, минуты)  exit\_times: deque[Tuple[int, int]] = deque()  # Обрабатываем каждого клиента  for hours, minutes in arrival\_times:  # Переводим время прихода клиента в минуты с начала суток  arrival\_minutes = hours \* 60 + minutes  # Находим мастера, который освободится раньше всех  next\_worker\_index = workers\_time.index(min(workers\_time))  # Если мастер свободен к моменту прихода клиента, обслуживание начинается сразу  if workers\_time[next\_worker\_index] <= arrival\_minutes:  workers\_time[next\_worker\_index] = arrival\_minutes + 30  else:  # Если мастер занят, клиент ждет, пока мастер освободится  workers\_time[next\_worker\_index] += 30  # Вычисляем время выхода клиента и добавляем его в очередь  exit\_time = workers\_time[next\_worker\_index]  exit\_times.append((exit\_time // 60, exit\_time % 60)) # Переводим минуты обратно в часы и минуты  return exit\_times  def main():  # Вводим количество клиентов  n: int = int(input())  # Вводим времена прихода клиентов (часы, минуты)  arrival\_times: List[Tuple[int, int]] = cast(  List[Tuple[int, int]],  [tuple(map(int, input().split())) for \_ in range(n)]  )  # Вычисляем времена выхода клиентов  exit\_times: Iterable[Tuple[int, int]] = calculate\_exit\_times(arrival\_times)  # Выводим времена выхода для каждого клиента  for hours, minutes in exit\_times:  print(hours, minutes)  main() |

**Задание 8. Шифровка**

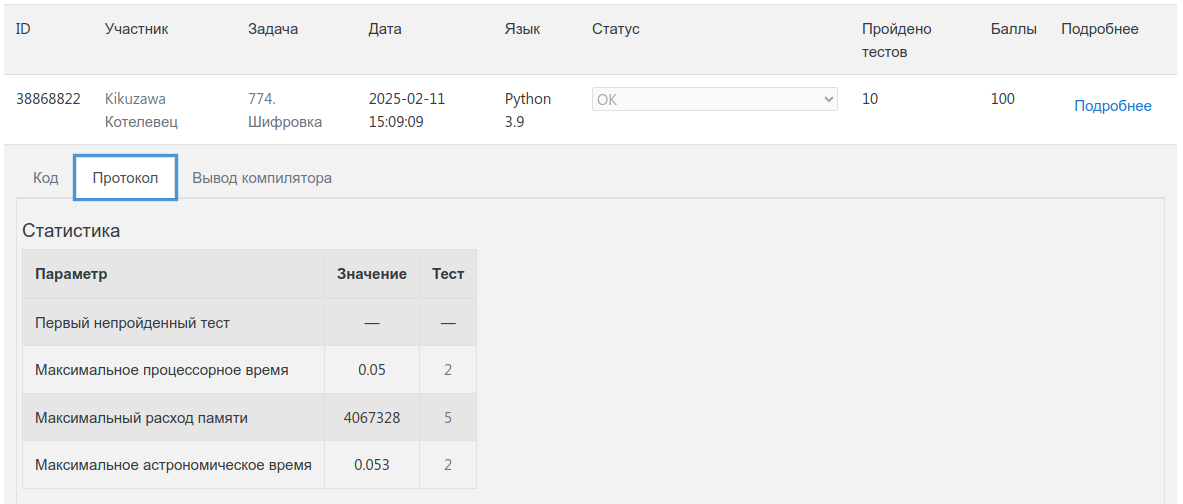
Шпион Коля зашифровал и послал в центр радиограмму. Он использовал такой способ шифровки: сначала выписал все символы своего сообщения (включая знаки препинания и т.п.), стоявшие на четных местах, в том же порядке, а затем ­– все символы, стоящие на нечетных местах. Напишите программу, которая расшифровывает сообщение.

**Входные данные**

Вводится одна непустая строка длиной не более 250 символов – зашифрованное сообщение. Строка может состоять из любых символов, кроме пробельных.

**Выходные данные**

Выведите одну строку – расшифрованное сообщение.



Листинг программы

|  |
| --- |
| from typing import List, AnyStr  def decrypt\_message(encrypted\_message):  """  Расшифровывает сообщение, которое было зашифровано методом Шпиона Коли.  Метод шифрования: сначала идут символы с чётными индексами,  затем символы с нечётными индексами.    Например: "Привет!" -> "Пиветр!"    Args:  encrypted\_message (AnyStr): Зашифрованное сообщение    Returns:  str: Расшифрованное исходное сообщение  """  # Преобразуем строку в список символов для возможности изменения элементов  symbols: List[AnyStr] = list(encrypted\_message)    # Находим середину строки - это длина первой части зашифрованного сообщения  n: int = len(encrypted\_message) // 2    # Переставляем символы на свои места  # Для каждой пары символов:  # - первый символ пары берём из второй половины (encrypted\_message[i + n])  # - второй символ пары берём из первой половины (encrypted\_message[i])  for i in range(n):  symbols[2 \* i: 2 \* i + 2] = encrypted\_message[i + n], encrypted\_message[i]    # Объединяем список символов обратно в строку  return ''.join(symbols)  def main():  """  Главная функция программы.  Читает зашифрованное сообщение и выводит его расшифрованную версию.  """  # Получаем зашифрованное сообщение от пользователя  encrypted\_message = input()    # Расшифровываем сообщение  decrypted\_message: str = decrypt\_message(encrypted\_message)    # Выводим результат  print(decrypted\_message)  # Запуск программы  main() |

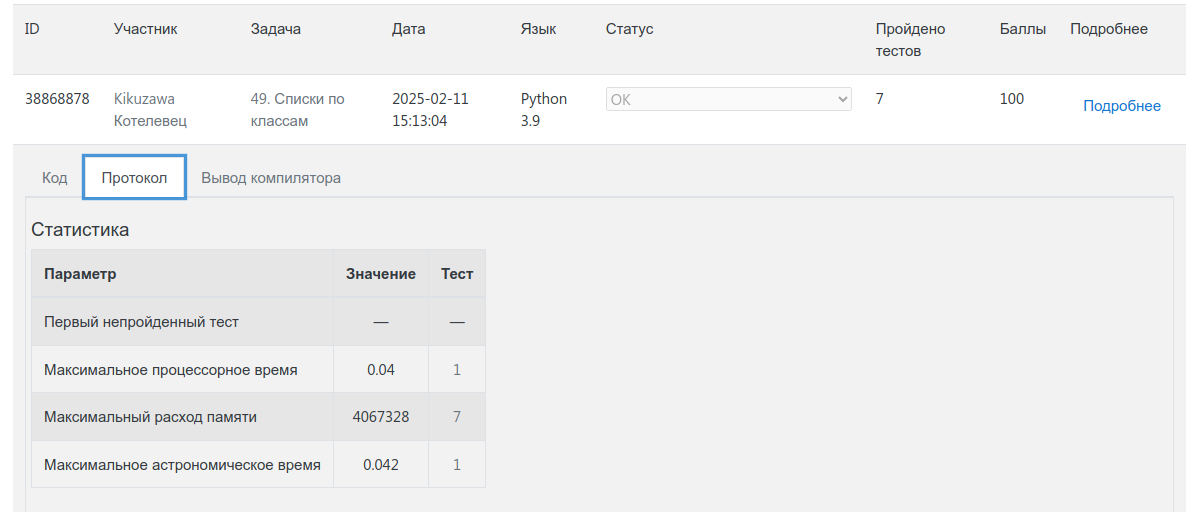
**Задание 9. Списки по классам**

***Формат входных данных***

В каждой строке сначала записан номер класса (число, равное 9, 10 или 11), затем (через пробел) – фамилия ученика. Общее число строк в файле не превосходит 100000. Длина каждой фамилии не превосходит 50 символов.

***Формат выходных данных***

Необходимо вывести список школьников по классам: сначала всех учеников 9 класса, затем – 10, затем – 11. Внутри одного класса порядок вывода фамилий должен быть таким же, как на входе.



Листинг программы

|  |
| --- |
| from typing import Dict, List, Iterable  def process\_students(students):  # Словарь для хранения учеников по классам  classes: Dict[int, List[str]] = {9: [], 10: [], 11: []}  # Обрабатываем каждого ученика  for student in students:  # Разделяем строку на номер класса и фамилию  class\_number, surname = student.split()  class\_number = int(class\_number) # Преобразуем номер класса в целое число  # Добавляем ученика в соответствующий класс  classes[class\_number].append(f"{class\_number} {surname}")  # Формируем итоговую строку для вывода  result = ""  # Проходим по классам в порядке 9, 10, 11  for class\_number in classes:  # Добавляем учеников текущего класса в результат  result += "\n".join(student for student in classes[class\_number]) + "\n"  return result  def main():  # Пути к входному и выходному файлам  input\_file\_path = "input.txt"  output\_file\_path = "output.txt"  # Чтение данных из входного файла  with open(input\_file\_path, 'r', encoding='KOI8-r') as file:  # Обрабатываем учеников и сортируем их по классам  students\_by\_class = process\_students(map(str.strip, file.readlines()))  # Запись результата в выходной файл  with open(output\_file\_path, 'w', encoding='KOI8-r') as file:  file.write(students\_by\_class)  main() |

**Задание 10. Игра в пьяницу**

В игре в пьяницу карточная колода раздается поровну двум игрокам. Далее они вскрывают по одной верхней карте, и тот, чья карта старше, забирает себе обе вскрытые карты, которые кладутся под низ его колоды. Тот, кто остается без карт – проигрывает.

Для простоты будем считать, что все карты различны по номиналу, а также, что самая младшая карта побеждает самую старшую карту ("шестерка берет туза").

Игрок, который забирает себе карты, сначала кладет под низ своей колоды карту первого игрока, затем карту второго игрока (то есть карта второго игрока оказывается внизу колоды).

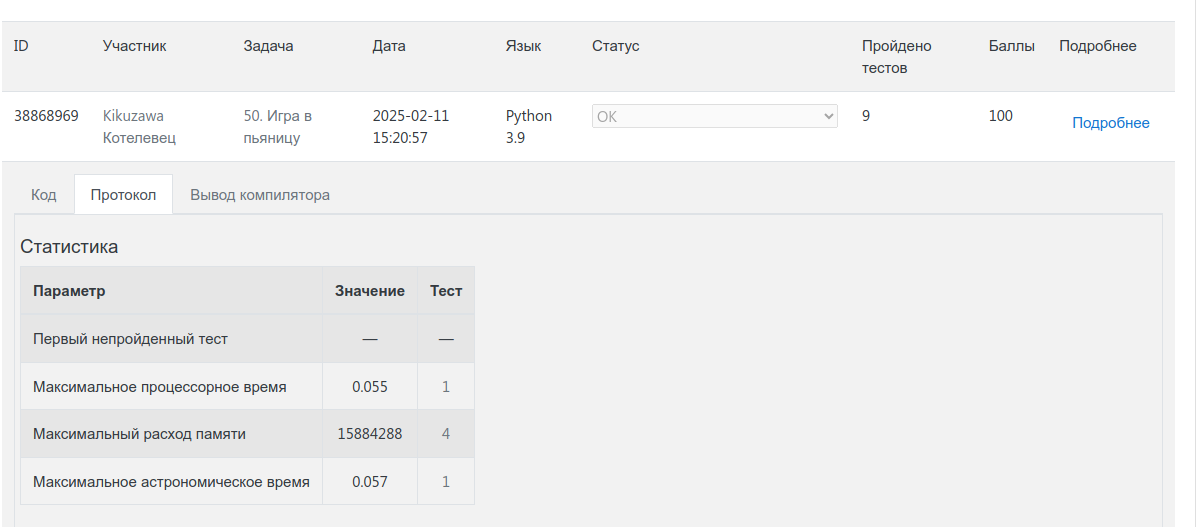
Напишите программу, которая моделирует игру в пьяницу и определяет, кто выигрывает. В игре участвует 10 карт, имеющих значения от 0 до 9, большая карта побеждает меньшую, карта со значением 0 побеждает карту 9.

**Входные данные**

Программа получает на вход две строки: первая строка содержит 5 чисел, разделенных пробелами — номера карт первого игрока, вторая – аналогично 5 карт второго игрока. Карты перечислены сверху вниз, то есть каждая строка начинается с той карты, которая будет открыта первой.

**Выходные данные**

Программа должна определить, кто выигрывает при данной раздаче, и вывести слово first или second, после чего вывести количество ходов, сделанных до выигрыша. Если на протяжении 106 ходов игра не заканчивается, программа должна вывести слово botva.



Листинг программы

|  |
| --- |
| from collections import deque  from typing import Tuple, Iterable, Dict  def is\_first\_player\_winner(card1, card2):  return (card1 > card2 and (card2, card1) != (0, 9)) or (card1 == 0 and card2 == 9)  def play\_game(  first\_deck: Iterable[int],  second\_deck: Iterable[int]  ) :    first\_deck: deque[int] = deque(first\_deck)  second\_deck: deque[int] = deque(second\_deck)  rounds: int = 0  max\_rounds: int = 1\_000\_000  while first\_deck and second\_deck:  rounds += 1  first\_card = first\_deck.popleft()  second\_card = second\_deck.popleft()  if is\_first\_player\_winner(first\_card, second\_card):  first\_deck.extend([first\_card, second\_card])  else:  second\_deck.extend([first\_card, second\_card])  if rounds >= max\_rounds:  return -1, -1  return not bool(first\_deck), rounds  def main():  cases: Dict[int, str] = {-1: "botva", 0: "first", 1: "second"}  result: Tuple[int, int] = play\_game(  map(int, input().split()),  map(int, input().split())  )  print(cases[result[0]], result[1] if result[1] != -1 else "")  main() |

Лабораторная работа 2

Задание 1.

При расследовании дорожно-транспортных происшествий часто возникают проблемы с розыском автомобилей, водители которых покинули место происшествия.

Получение свидетельских показаний – непростая работа. Ситуация осложняется тем, что очень часто свидетели могут только приблизительно вспомнить номер автомобиля. При этом с большой вероятностью опрашиваемый может перепутать порядок цифр или букв в номере.

По полученному от свидетеля происшествия номеру, подсчитайте, сколько различных номеров может получиться из него перестановкой букв и/или цифр, а также выведите все такие номера.

Напомним, что автомобильные номера в России состоят из трех букв и трех цифр, упорядоченных следующим образом: буква, три цифры, затем две буквы. Фрагмент номера, который идентифицирует регион, в котором зарегистрирован автомобиль, мы будем игнорировать.

В номере могут использоваться следующие буквы: «A», «B», «C», «E», «H», «K», «M», «O», «P», «T», «X», «Y» (эти буквы имеют схожие по написанию аналоги как в русском, так и в латинском алфавите). В этой задаче во входных данных будут использоваться буквы латинского алфавита.

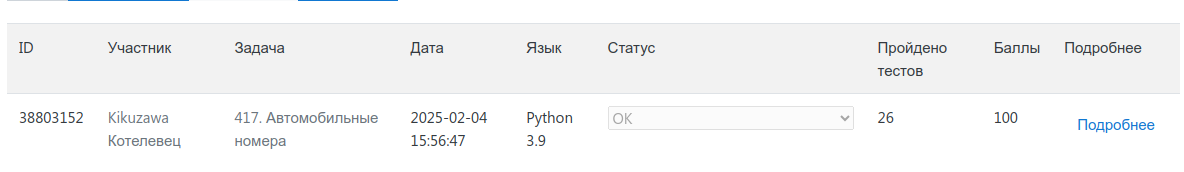
**Входные данные**

На вход программы поступает  одна строка, которая представляет собой корректный автомобильный номер.

**Выходные данные**

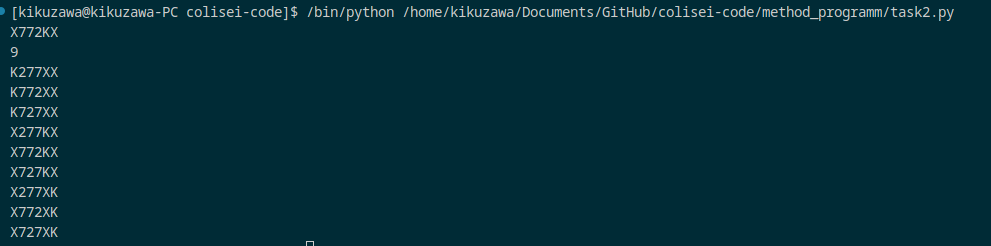
В первой строке  выведите число k – количество номеров, которые могут получиться из заданного перестановкой букв и/или цифр.

В последующих k строках выведите все такие номера в произвольном порядке.



Листинг программы:

|  |
| --- |
| from collections import deque  from itertools import permutations  from typing import Set, AnyStr, Sequence  **def** generate\_swaps(number: AnyStr) -> Sequence[AnyStr]:  letters: AnyStr = number[0] + number[4] + number[5]  digits: AnyStr = number[1] + number[2] + number[3]  unique\_digit\_swaps: Set[AnyStr] = {''.join(p) for p in permutations(digits)}  unique\_letter\_swaps: Set[AnyStr] = {''.join(p) for p in permutations(letters)}  all\_combinations: deque[AnyStr] = deque()  for letter in unique\_letter\_swaps:  for digit in unique\_digit\_swaps:  combined\_number = **f**"{letter[0]}{digit[0]}{digit[1]}{digit[2]}{letter[1]}{letter[2]}"  all\_combinations.append(combined\_number)  return all\_combinations  number\_by\_car: str = input()  swaps\_list: Sequence[AnyStr] = generate\_swaps(number\_by\_car)  print(len(swaps\_list), \*swaps\_list, sep='\n') |



Задача 2.

В классе учатся N человек. Классный руководитель получил указание направить на субботник R бригад по С человек в каждой.

Все бригады на субботнике будут заниматься переноской бревен. Каждое бревно одновременно несут все члены одной бригады. При этом бревно нести тем удобнее, чем менее различается рост членов этой бригады.

Числом неудобства бригады будем называть разность между ростом самого высокого и ростом самого низкого членов этой бригады (если в бригаде только один человек, то эта разница равна 0). Классный руководитель решил сформировать бригады так, чтобы максимальное из чисел неудобства сформированных бригад было минимально. Помогите ему в этом!

Рассмотрим следующий пример. Пусть в классе 8 человек, рост которых в сантиметрах равен 170, 205, 225, 190, 260, 130, 225, 160, и необходимо сформировать две бригады по три человека в каждой. Тогда одним из вариантов является такой:

1 бригада: люди с ростом 225, 205, 225

2 бригада: люди с ростом 160, 190, 170

При этом число неудобства первой бригады будет равно 20, а число неудобства второй — 30. Максимальное из чисел неудобств будет 30, и это будет наилучший возможный результат.

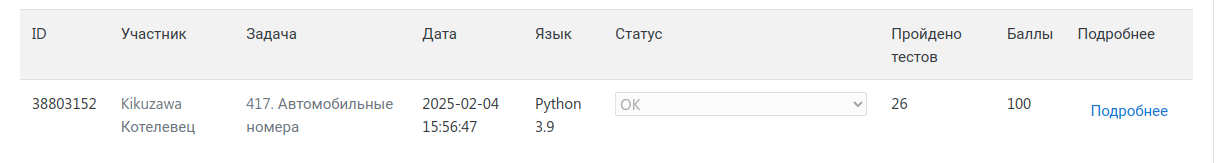
Формат входных данных

Сначала вводятся натуральные числа N, R и C — количество человек в классе, количество бригад и количество человек в каждой бригаде (1 ≤ R∙C ≤ N ≤ 100 000). Далее вводятся N целых чисел — рост каждого из N учеников. Рост ученика — натуральное число, не превышающее 1 000 000 000.

Формат выходных данных

Выведите одно число — наименьше возможное значение максимального числа неудобства сформированных бригад.

|  |
| --- |
| from typing import List, Iterable  def can\_form\_branches(  max\_uncomfortable: int,  discomforts: Iterable[int],  r: int,  c: int  ) -> bool:  count = 0  members\_in\_current\_branch = 0  for discomfort in discomforts:  members\_in\_current\_branch -= 1  if members\_in\_current\_branch < 1 and discomfort <= max\_uncomfortable:  count += 1  members\_in\_current\_branch = c  return count >= r  def find\_minimum\_max\_uncomfortable(n: int, r: int, c: int, heights: Iterable[int]) -> int:  heights: List[int] = sorted(heights)  discomforts: List[int] = [heights[i + c - 1] - heights[i] for i in range(n - c + 1)]  left = -1  right = heights[-1] - heights[0]  while left + 1 < right:  mid = (left + right) // 2  if can\_form\_branches(mid, discomforts, r, c):  right = mid  else:  left = mid  return right  def main() -> None:  n, r, c = map(int, input().split())  heights: List[int] = [int(input()) for \_ in range(n)]  result: int = find\_minimum\_max\_uncomfortable(n, r, c, heights)  print(result)  if \_\_name\_\_ == "\_\_main\_\_":  main() |



Задание 3.

Перед началом тараканьих бегов всем болельщикам было предложено сделать по две ставки на результаты бегов. Каждая ставка имеет вид "Таракан №A придет раньше, чем таракан №B".

Организаторы бегов решили выяснить, могут ли тараканы прийти в таком порядке, чтобы у каждого болельщика сыграла ровно одна ставка из двух (то есть чтобы ровно одно из двух утверждений каждого болельщика оказалось верным). Считается, что никакие два таракана не могут прийти к финишу одновременно.

Входные данные

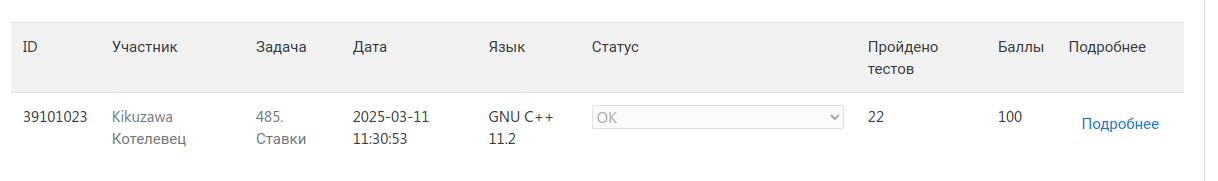
В первой строке входных данных содержатся два разделенных пробелом натуральных числа: число K, не превосходящее 10, - количество тараканов и число N, не превосходящее 100, - количество болельщиков. Все тараканы пронумерованы числами от 1 до K. Каждая из следующих N строк содержит 4 натуральных числа A, B, C, D, не превосходящих K, разделенных пробелами. Они соответствуют ставкам болельщика "Таракан №A придет раньше, чем таракан №B" и "Таракан №C придет раньше, чем таракан №D".

Выходные данные

Если завершить бега так, чтобы у каждого из болельщиков сыграла ровно одна из двух ставок, можно, то следует вывести номера тараканов в том порядке, в котором они окажутся в итоговой таблице результатов (сначала номер таракана, пришедшего первым, затем номер таракана, пришедшего вторым и т. д.) в одну строку через пробел. Если таких вариантов несколько, выведите любой из них.

Если требуемого результата добиться нельзя, выведите одно число 0.

|  |
| --- |
| #include <iostream>  #include <algorithm>  #include <vector>  using namespace std;  int main() {  int k, n;  cin >> k >> n;  vector<int> cockroach(k); // Перестановка тараканов  vector<vector<int>> Rates(n, vector<int>(4)); // Ставки болельщиков  // Считываем ставки  for (int i = 0; i < n; i++) {  cin >> Rates[i][0] >> Rates[i][1] >> Rates[i][2] >> Rates[i][3];  }  // Инициализация первой перестановки  for (int i = 0; i < k; i++) {  cockroach[i] = i + 1;  }  // Перебор всех перестановок  do {  bool valid = true; // Флаг для проверки условий ставок  // Создаем массив позиций для быстрого доступа  vector<int> positions(k + 1);  for (int i = 0; i < k; i++) {  positions[cockroach[i]] = i;  }  // Проверяем каждую ставку  for (int i = 0; i < n; i++) {  int a = Rates[i][0], b = Rates[i][1], c = Rates[i][2], d = Rates[i][3];  // Индексы тараканов  int index\_a = positions[a];  int index\_b = positions[b];  int index\_c = positions[c];  int index\_d = positions[d];  // Условия ставок  bool condition1 = (index\_a < index\_b); // A до B  bool condition2 = (index\_c < index\_d); // C до D  // Проверяем, чтобы ровно одно из условий было выполнено  if ((condition1 && condition2) || (!condition1 && !condition2)) {  valid = false;  break;  }  }  // Если все ставки выполнены, выводим результат  if (valid) {  for (int i = 0; i < k; i++) {  cout << cockroach[i] << " ";  }  cout << endl;  return 0;  }  } while (next\_permutation(cockroach.begin(), cockroach.end()));  // Если не нашли подходящей перестановки  cout << 0 << endl;  return 0;  } |



Задание 4.

В турнире по хоккею участвовало K команд, каждая сыграла с каждой по одному матчу. За победу команда получала 2 очка, за ничью – 1, за поражение – 0 очков.

Известно, сколько очков в итоге получила каждая команда, однако результаты конкретных матчей были утеряны. Требуется восстановить одну из возможных турнирных таблиц.

Входные данные

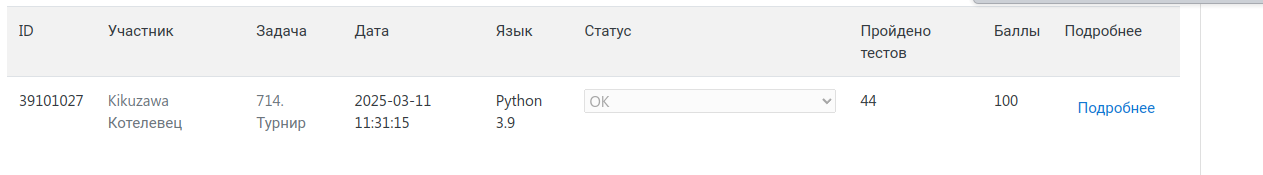
В первой строке входных данных содержится одно натурально число K, не превосходящее 100 – количество команд. Во второй строке  задаются  через пробел K целых неотрицательных чисел, не превосходящих 2(K–1), – количество очков, набранных командами, занявшими первое, второе, …, K-е места соответственно (то есть каждое следующее число не больше предыдущего).

Выходные данные

Выведите турнирную таблицу в следующем формате. Таблица должна состоять из K строк с результатами игр команд, занявших первое, второе, …, последнее место (команды, набравшие одинаковое число очков, могут быть расположены в таблице в любом порядке). В каждой строке должно быть записано K чисел через пробел – количество очков, набранных в игре данной команды с первой, второй, … командами соответственно. Количество очков – это число 0, 1 или 2. В клетках на главной диагонали (соответствующих не существующей игре команды "самой с собой") нужно записать нули.

Гарантируется, что входные данные соответствуют реальному турниру, то есть хотя бы одна таблица, соответствующая входным данным, может быть построена. Если таких таблиц несколько, выведите любую из них.

|  |
| --- |
| from typing import List, Iterable  def restore\_tournament\_table(num\_teams: int, points: List[int]) -> Iterable[Iterable[int]]:  # Инициализация таблицы результатов  tournament\_table: List[List[int]] = [[0] \* num\_teams for \_ in range(num\_teams)]  # Индексы команд  team\_indices = range(num\_teams)  for i in range(num\_teams - 1):  # Сортируем команды по очкам  sorted\_points, sorted\_indices = zip(\*sorted(zip(points, team\_indices)))  sorted\_points: List[int] = list(sorted\_points)  sorted\_indices: List[int] = list(sorted\_indices)  for j in range(i + 1, num\_teams):  if tournament\_table[sorted\_indices[i]][sorted\_indices[j]] > 0:  continue  elif sorted\_points[i] > 0:  # Ничья  tournament\_table[sorted\_indices[i]][sorted\_indices[j]] = 1  tournament\_table[sorted\_indices[j]][sorted\_indices[i]] = 1  sorted\_points[i] -= 1  points[sorted\_indices[j]] -= 1  elif sorted\_points[i] == 0:  # Поражение  tournament\_table[sorted\_indices[j]][sorted\_indices[i]] = 2  points[sorted\_indices[j]] -= 2  points[sorted\_indices[i]] = -i - 1  return tournament\_table  def main() -> None:  num\_teams: int = int(input())  points: List[int] = list(map(int, input().split()))  tournament\_table: Iterable[Iterable[int]] = restore\_tournament\_table(num\_teams, points)  for row in tournament\_table:  print(\*row)  if \_\_name\_\_ == "\_\_main\_\_":  main() |



Задание 5.

Дано два списка чисел, числа в первом списке упорядочены по неубыванию. Для каждого числа из второго списка определите номер первого и последнего появления этого числа в первом списке.

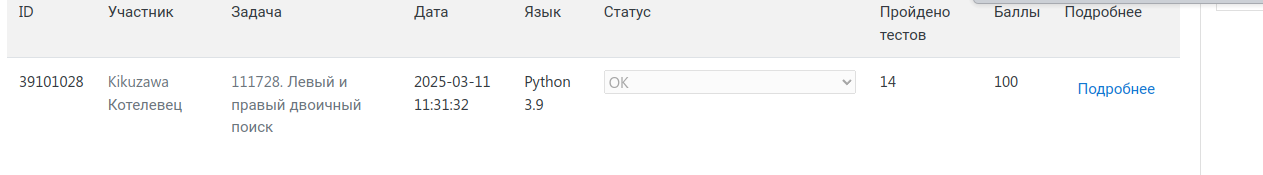
Входные данные

В первой строке входных данных записано два числа NN и MM (1≤N,M≤200001≤N,M≤20000). Во второй строке записано NN упорядоченных по неубыванию целых чисел — элементы первого списка. В третьей строке записаны MM целых неотрицательных чисел - элементы второго списка. Все числа в списках - целые 32-битные знаковые.

Выходные данные

Программа должна вывести MM строчек. Для каждого числа из второго списка нужно вывести номер его первого и последнего вхождения в первый список. Нумерация начинается с единицы. Если число не входит в первый список, нужно вывести одно число 00.

|  |
| --- |
| import bisect  from typing import List, Tuple, Sequence, Union  from collections import deque  def find\_first\_and\_last\_occurrences(  sorted\_list: Sequence[int],  search\_list: Sequence[int]  ) -> deque[Union[Tuple[int, int], Tuple[int]]]:  results: deque[Union[Tuple[int, int], Tuple[int]]] = deque()  for number in search\_list:  first\_index = bisect.bisect\_left(sorted\_list, number)  if first\_index >= len(sorted\_list) or sorted\_list[first\_index] != number:  results.append((0,))  else:  last\_index = bisect.bisect\_right(sorted\_list, number) - 1  results.append((first\_index + 1, last\_index + 1)) # +1 для единичной нумерации  return results  def main() -> None:  \_, \_ = map(int, input().split())  sorted\_list: List[int] = list(map(int, input().split()))  search\_list: List[int] = list(map(int, input().split()))  occurrences = find\_first\_and\_last\_occurrences(sorted\_list, search\_list)  for result in occurrences:  print(\*result)  if \_\_name\_\_ == "\_\_main\_\_":  main() |



Задание 6.

Дано N

натуральных чисел. Требуется для каждого числа найти количество вариантов разбиения его на сумму двух других чисел из данного набора.

Входные данные

В первой строке дано число N

( 1 ≤ N

≤ 10000). Далее заданы N

натуральных чисел, не превосходящих 109

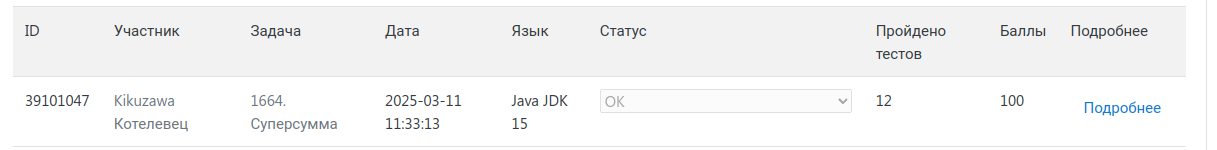
. Для каждого числа количество разбиений меньше 231.

Выходные данные

Вывести N

чисел – количество разбиений, в порядке, соответствующем исходному.

|  |
| --- |
| import java.util.Arrays;  import java.util.HashMap;  import java.util.Map;  import java.util.Scanner;  import java.util.stream.Collectors;  public class SuperSum {  static int[] countPartitions(int[] numbers) {  Map<Integer, Long> countMap = Arrays.stream(numbers)  .boxed()  .collect(Collectors.groupingBy(n -> n, Collectors.counting()));  int[] results = new int[numbers.length];  for (var i = 0; i < numbers.length; i++) {  int number = numbers[i];  int count = 0;  for (Map.Entry<Integer, Long> entry : countMap.entrySet()) {  var x = entry.getKey();  var y = number - x;  // Избегаем дублирования  if (y < x) {  continue;  }  var xCount = entry.getValue();  if (x == y) {  // Если x и y одинаковые, то выбираем 2 из countMap[x]  count += xCount \* (xCount - 1) / 2;  } else if (countMap.containsKey(y)) {  // Если x и y разные, то просто перемножаем их количества  count += xCount \* countMap.get(y);  }  }  results[i] = count;  }  return results;  }  public static void main(String[] args) {  Scanner scanner = new Scanner(System.in);  int n = scanner.nextInt();  int[] numbers = new int[n];  for (int i = 0; i < n; i++) {  numbers[i] = scanner.nextInt();  }  int[] partitions = countPartitions(numbers);  for (int result : partitions) {  System.out.println(result);  }  scanner.close();  }  } |



Задание 7.

На прямой расположены стойла, в которые необходимо расставить коров так, чтобы минимальное расcтояние между коровами было как можно больше.

Входные данные

В первой строке вводятся числа N

(2<N<10001)

– количество стойл и K

(1<K<N)

– количество коров. Во второй строке задаются N

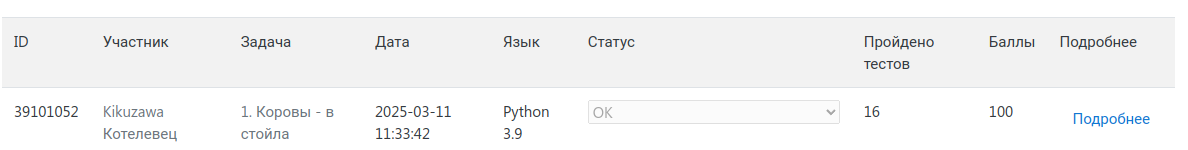
натуральных чисел в порядке возрастания – координаты стойл (координаты не превосходят 109

)

Выходные данные

Выведите одно число – наибольшее возможное допустимое расстояние.

|  |
| --- |
| from typing import List, Sequence  def max\_distance\_between\_cows(stalls: Sequence[int], num\_cows: int) -> int:  left: int = 0  right: int = stalls[-1] - stalls[0] + 1  while left < right:  mid: int = (left + right) // 2  cows\_placed: int = 1  last\_placed\_stall: int = stalls[0] # Последнее стойло, в которое была поставлена корова  for current\_stall in stalls[1:]:  if current\_stall - last\_placed\_stall >= mid:  cows\_placed += 1  last\_placed\_stall = current\_stall  if cows\_placed >= num\_cows:  left = mid + 1  else:  right = mid  return left - 1  def main() -> None:  n, k = map(int, input().split())  stalls: List[int] = list(map(int, input().split()))  result: int = max\_distance\_between\_cows(stalls, k)  print(result)  if \_\_name\_\_ == "\_\_main\_\_":  main() |



Задание 8.

Дано N отрезков провода длиной L1, L2, ..., LN сантиметров. Требуется с помощью разрезания получить из них K равных отрезков как можно большей длины, выражающейся целым числом сантиметров. Если нельзя получить K отрезков длиной даже 1 см, вывести 0.

Ограничения: 1 <= N <= 10 000, 1 <= K <= 10 000, 100 <= Li <= 10 000 000, все числа целые.

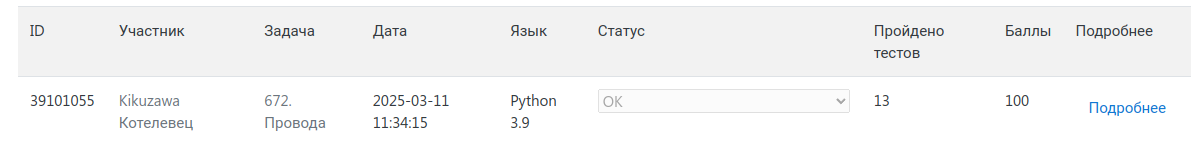
Входные данные

В первой строке находятся числа N и К. В следующих N строках - L1, L2, ..., LN, по одному числу в строке.

Выходные данные

Вывести одно число - полученную длину отрезков.

|  |
| --- |
| from typing import List, Sequence  def max\_segment\_length(segments: Sequence[int], required\_segments: int) -> int:  left: int = 1  right: int = 10\_000\_001  while left < right:  mid: int = (left + right) // 2  total\_segments: int = sum(length // mid for length in segments)  if total\_segments >= required\_segments:  left = mid + 1  else:  right = mid  return left - 1  def main() -> None:  n, k = map(int, input().split())  segments: List[int] = [int(input()) for \_ in range(n)]  result: int = max\_segment\_length(segments, k)  print(result)  if \_\_name\_\_ == "\_\_main\_\_":  main() |



Задание 9.

Банк «Кисловодск» переходит на новый вид банковских карт. Для этого производятся одинаковые заготовки, на которых есть специальное место для идентификации клиента. Изначально на этом месте записывается кодовое число X. В банке с помощью специального прибора можно стирать некоторые цифры числа X. Оставшиеся цифры, будучи записанными подряд, должны образовывать номер счета клиента. Например, при X = 12013456789 номера счетов 5, 12, 17 или 12013456789 получить можно, а номера 22 или 71 получить нельзя.

Способ распределения номеров счетов в банке очень прост. Счетам присваиваются последовательно номера 1, 2, … Очевидно, что при таком способе в какой-то момент впервые найдется номер счета N, который нельзя будет получить из цифр X указанным выше способом. Руководство банка хочет знать значение N.

Напишите программу, которая находила бы N по заданному X.

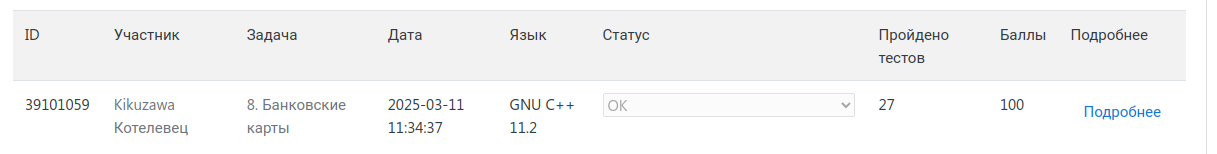
Входные данные

Вводится натуральное число X без ведущих нулей (1 ≤ X < 101000)

Выходные данные

Выведите искомое N без ведущих нулей.

|  |
| --- |
| #include <bits/stdc++.h>  using namespace std;  //#define int int64\_t  void solve() {  string s;  cin >> s;  int n = int(s.size());  vector<int> x(n);  for (int i = 0; i < n; i++) {  x[i] = int(s[i] - '0');  }  vector<vector<int>> blocks;  int cur\_block = 0;  blocks.push\_back({});  vector<bool> cnt(10, false);  for (int i = n - 1; i >= 0; i--) {  cnt[x[i]] = true;  blocks[cur\_block].push\_back(x[i]);  bool check\_all = true;  for (int j = 0; j < 10; j++) {  if (cnt[j] == false) {  check\_all = false;  break;  }  }  if (check\_all) {  cur\_block++;  blocks.push\_back({});  for (int j = 0; j < 10; j++) {  cnt[j] = false;  }  }  }  for (int i = 0; i <= cur\_block; i++) {  reverse(blocks[i].begin(), blocks[i].end());  }  /\*for (auto to : blocks) {  for (auto to2 : to) {  cout << to2;  }  cout << '\n';  }\*/  for (int j = 0; j < 10; j++) {  cnt[j] = false;  }  for (auto num : blocks[cur\_block]) {  cnt[num] = true;  }  int first\_digit = 0;  for (int j = 1; j < 10; j++) {  if (cnt[j] == false) {  first\_digit = j;  break;  }  }  if (first\_digit == 0) {  cout << 10;  }  else {  cout << first\_digit;  }  int cur\_digit = first\_digit;  for (int i = cur\_block - 1; i >= 0; i--) {  for (int j = 0; j < 10; j++) {  cnt[j] = false;  }  bool ff = true;  for (auto digit : blocks[i]) {  if (ff == false) {  cnt[digit] = true;  }  if (digit == cur\_digit && ff == true) {  ff = false;  }  }  for (int j = 0; j < 10; j++) {  if (cnt[j] == false) {  cout << j;  cur\_digit = j;  break;  }  }  }  }  int32\_t main() {  ios\_base::sync\_with\_stdio(0);  cin.tie(0);  cout.tie(0);  //freopen("inputik.txt", "r", stdin);  //freopen("outputik.txt", "w", stdout);  int t = 1;  //cin >> t;  while (t--) {  solve();  }  } |



Задание 10

Глеб обожает шоппинг. Как-то раз он загорелся идеей подобрать себе кепку, майку, штаны и ботинки так, чтобы выглядеть в них максимально стильно. В понимании Глеба стильность одежды тем больше, чем меньше разница в цвете элементов его одежды.

В наличии имеется N1 кепок, N2 маек, N3 штанов и N4 пар ботинок (1 ≤ Ni ≤ 100 000). Про каждый элемент одежды известен его цвет (целое число от 1 до 100 000). Комплект одежды — это одна кепка, майка, штаны и одна пара ботинок. Каждый комплект характеризуется максимальной разницей между любыми двумя его элементами. Помогите Глебу выбрать максимально стильный комплект, то есть комплект с минимальной разницей цветов.

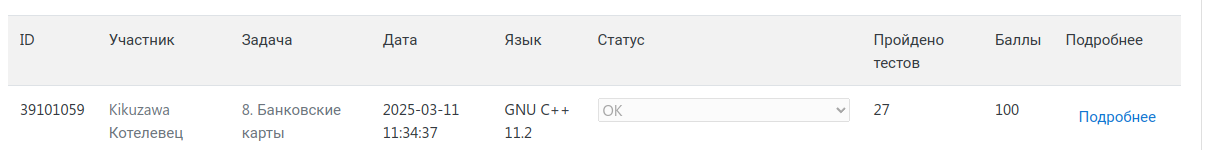
Входные данные

Для каждого типа одежды i (i = 1, 2, 3, 4) сначала вводится количество Ni элементов одежды этого типа, далее в следующей строке — последовательность из Ni целых чисел, описывающих цвета элементов. Все четыре типа подаются на вход последовательно, начиная с кепок и заканчивая ботинками. Все вводимые числа целые, положительные и не превосходят 100 000.

Выходные данные

Выведите четыре целых числа — цвета соответственно для кепки, майки, штанов и ботинок, которые должен выбрать Глеб из имеющихся для того, чтобы выглядеть наиболее стильно. Если ответов несколько, выведите любой.

|  |
| --- |
| from array import array, ArrayType  from copy import copy  from typing import List, Tuple, Sequence  def find\_most\_stylish\_outfit(  hats: Sequence[int],  shirts: Sequence[int],  pants: Sequence[int],  shoes: Sequence[int]  ) -> Tuple[int, int, int, int]:  indices: ArrayType[int] = array('i', [0, 0, 0, 0])  min\_diff: float = float('inf')  best\_indices: ArrayType[int] = array('i', [0, 0, 0, 0])  while all(indices[i] < len(lst) for i, lst in enumerate((hats, shirts, pants, shoes))):  current\_colors = (hats[indices[0]], shirts[indices[1]], pants[indices[2]], shoes[indices[3]])  min\_color = min(current\_colors)  max\_color = max(current\_colors)  current\_diff = max\_color - min\_color  # Обновляем минимальную разницу и сохраняем текущие индексы  if current\_diff < min\_diff:  min\_diff = current\_diff  best\_indices = copy(indices)  # Если разница минимальна (0), выходим из цикла  if current\_diff == 0:  break  # Сдвигаем указатель у минимального элемента  for i, color in enumerate(current\_colors):  if color == min\_color:  indices[i] += 1  break  return (  hats[best\_indices[0]],  shirts[best\_indices[1]],  pants[best\_indices[2]],  shoes[best\_indices[3]]  )  def main() -> None:  colors\_for\_clothes: List[List[int]] = []  for \_ in range(4):  \_ = int(input())  colors = sorted(map(int, input().split()))  colors\_for\_clothes.append(colors)  stylish\_outfit: Tuple[int, int, int, int] = find\_most\_stylish\_outfit(\*colors\_for\_clothes)  print(\*stylish\_outfit)  if \_\_name\_\_ == "\_\_main\_\_":  main() |



Задание 11.

Дано кубическое уравнение ax3+bx2+cx+d=0(a≠0)ax3+bx2+cx+d=0(a≠0). Известно, что у этого уравнения ровно один корень. Требуется его найти.

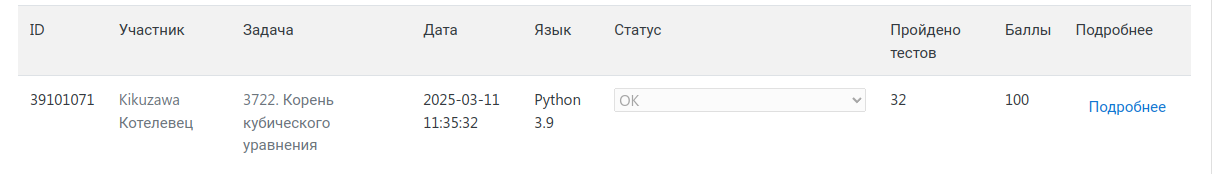
Входные данные

Во входных данных через пробел записаны четыре целых числа: −1000≤a,b,c,d≤1000−1000≤a,b,c,d≤1000.

Выходные данные

Выведите единственный корень уравнения с точностью не менее 4 знаков после десятичной точки.

|  |
| --- |
| def evaluate\_polynomial(a: int, b: int, c: int, d: int, x: float) -> float:  """Вычисляет значение кубического уравнения ax^3 + bx^2 + cx + d."""  return a \* x \*\* 3 + b \* x \*\* 2 + c \* x + d  def evaluate\_derivative(a: int, b: int, c: int, x: float) -> float:  """Вычисляет значение производной кубического уравнения 3ax^2 + 2bx + c."""  return 3 \* a \* x \*\* 2 + 2 \* b \* x + c  def find\_root\_newton(a: int, b: int, c: int, d: int, initial\_guess: float = 0.0, tolerance: float = 1e-5) -> float:  """Находит корень кубического уравнения методом Ньютона."""  x0 = initial\_guess  while True:  fx = evaluate\_polynomial(a, b, c, d, x0)  f\_prime\_x = evaluate\_derivative(a, b, c, x0)  # Новое приближение корня  x1 = x0 - fx / f\_prime\_x  # Если разница между приближениями достаточно мала, завершаем итерацию  if abs(x1 - x0) < tolerance:  return x1 # Возвращаем найденный корень  # Обновляем приближение  x0 = x1  def bisection(a: int, b: int, c: int, d: int, left: float, right: float, tol: float = 1e-5) -> float:  """Находит корень кубического уравнения методом бисекции."""  if evaluate\_polynomial(a, b, c, d, left) \* evaluate\_polynomial(a, b, c, d, right) >= 0:  return find\_root\_newton(a, b, c, d) # Если нет корня, используем метод Ньютона  while (right - left) / 2 > tol:  midpoint = (left + right) / 2  f\_mid = evaluate\_polynomial(a, b, c, d, midpoint)  if f\_mid == 0:  return midpoint # Найден точный корень  elif evaluate\_polynomial(a, b, c, d, left) \* f\_mid < 0:  right = midpoint  else:  left = midpoint  return (left + right) / 2 # Возвращаем приближенный корень  def main() -> None:  """Основная функция для чтения входных данных и вывода корня уравнения."""  a, b, c, d = map(int, input().split())  root = bisection(a, b, c, d, -1000, 1000)  print(root)  if \_\_name\_\_ == "\_\_main\_\_":  main() |



Задание 12.

Студенты одного из вузов спроектировали робота для частичной автоматизации процесса сборки авиационного двигателя.

В процессе сборки двигателя могут встречаться операции 26 типов, которые обозначаются строчными буквами латинского алфавита. Процесс сборки состоит из N операций.

Предполагается использовать робота один раз для выполнения части подряд идущих операций из процесса сборки.

Память робота состоит из K ячеек, каждая из которых содержит одну операцию. Операции выполняются последовательно, начиная с первой, в том порядке, в котором они расположены в памяти. Выполнив последнюю из них, робот продолжает работу с первой. Робота можно остановить после любой операции. Использование робота экономически целесообразно, если он выполнит хотя бы K + 1 операцию.

Требуется написать программу, которая по заданному процессу сборки определит количество экономически целесообразных способов использования робота.

Входные данные

В первой строке входного файла записано число K > 0 "— количество операций, которые можно записать в память робота.

Вторая строка состоит из N > K строчных латинских букв, обозначающих операции "— процесс сборки двигателя. Операции одного и того же типа обозначаются одной и той же буквой.

Выходные данные

Выходной файл должен содержать единственное целое число "— количество экономически целесообразных способов использования робота.

Примечание

Данная задача содержит четыре подзадачи. Для оценки каждой подзадачи используется своя группа тестов. Баллы за подзадачу начисляются только в том случае, если все тесты из этой группы пройдены.

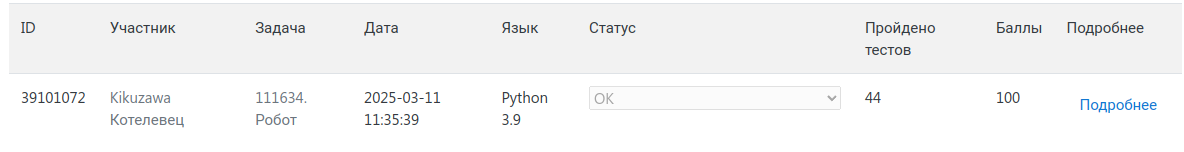
Тесты из условия. Подзадача оценивается в 0 баллов.

N ≤ 100. Подзадача оценивается в 30 баллов.

N ≤ 2000. Подзадача оценивается в 30 баллов.

N ≤ 200 000. Подзадача оценивается в 40 баллов.

|  |
| --- |
| def count\_economically\_viable\_ways(k: int, operations: str) -> int:  n: int = len(operations)  viable\_ways: int = 0  consecutive\_matches: int = 0  # Проходим по строке операций с конца к началу  for i in range(n - k - 1, -1, -1):  if operations[i] == operations[i + k]:  consecutive\_matches += 1  else:  consecutive\_matches = 0  viable\_ways += consecutive\_matches  return viable\_ways  def main() -> None:  k: int = int(input())  operations: str = input()  result: int = count\_economically\_viable\_ways(k, operations)  print(result)  if \_\_name\_\_ == "\_\_main\_\_":  main() |



Лабораторная работа 3

Задание 1.

Даны две последовательности, требуется найти и вывести их наибольшую общую подпоследовательность.

Входные данные

В первой строке входных данных содержится число N

– длина первой последовательности (1 ≤ N

≤ 1000). Во второй строке заданы члены первой последовательности (через пробел) – целые числа, не превосходящие 10000 по модулю.

В третьей строке записано число M

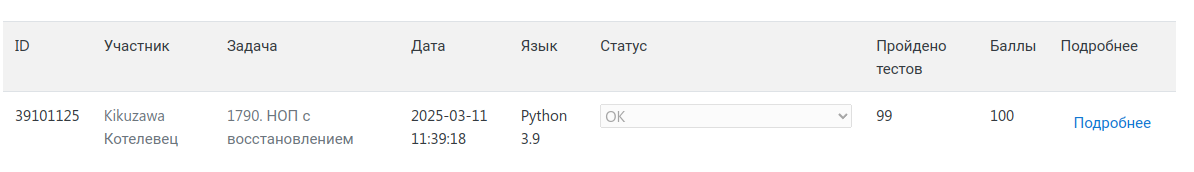
– длина второй последовательности (1 ≤ M

≤ 1000). В четвертой строке задаются члены второй последовательности (через пробел) – целые числа, не превосходящие 10000 по модулю.

Выходные данные

Требуется вывести наибольшую общую подпоследовательность данных последовательностей, через пробел.

|  |
| --- |
| from typing import List, Sequence  from itertools import product  def find\_longest\_common\_subsequence(  len\_seq\_a: int,  sequence\_a: Sequence[int],  len\_seq\_b: int,  sequence\_b: Sequence[int]  ) -> List[int]:  # Создаем таблицу для хранения длины LCS  lcs\_length\_table: List[List[int]] = [[0] \* (len\_seq\_b + 1) for \_ in range(len\_seq\_a + 1)]  # Заполняем таблицу  for i, j in product(range(1, len\_seq\_a + 1), range(1, len\_seq\_b + 1)):  if sequence\_a[i - 1] == sequence\_b[j - 1]:  lcs\_length\_table[i][j] = lcs\_length\_table[i - 1][j - 1] + 1  else:  lcs\_length\_table[i][j] = max(lcs\_length\_table[i - 1][j], lcs\_length\_table[i][j - 1])  # Восстанавливаем LCS  longest\_common\_subseq: List[int] = []  i: int = len\_seq\_a  j: int = len\_seq\_b  while i > 0 and j > 0:  if sequence\_a[i - 1] == sequence\_b[j - 1]:  longest\_common\_subseq.append(sequence\_a[i - 1])  i -= 1  j -= 1  elif lcs\_length\_table[i - 1][j] == lcs\_length\_table[i][j]:  i -= 1  else:  j -= 1  return longest\_common\_subseq[::-1]  def main() -> None:  len\_seq\_a: int = int(input())  sequence\_a: List[int] = list(map(int, input().split()))  len\_seq\_b: int = int(input())  sequence\_b: List[int] = list(map(int, input().split()))  longest\_common\_subseq: List[int] = find\_longest\_common\_subsequence(len\_seq\_a, sequence\_a, len\_seq\_b, sequence\_b)  print(' '.join(map(str, longest\_common\_subseq)))  if \_\_name\_\_ == "\_\_main\_\_":  main() |



Задание 2.

Дана текстовая строка. С ней можно выполнять следующие операции:

1. Заменить один символ строки на другой символ.

2. Удалить один произвольный символ.

3. Вставить произвольный символ в произвольное место строки.

Например, при помощи первой операции из строки "СОК" можно получить строку "СУК", при помощи второй операции - строку "ОК", при помощи третьей операции - строку "СТОК.

Минимальное количество таких операций, при помощи которых можно из одной строки получить другую, называется стоимостью редактирования или расстоянием Левенштейна.

Определите расстояние Левенштейна для двух данных строк.

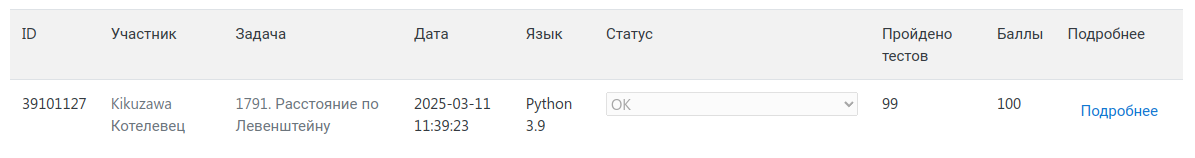
Входные данные

Программа получает на вход две строки, длина каждой из которых не превосходит 1000 символов, строки состоят только из заглавных латинских букв.

Выходные данные

Требуется вывести одно число – расстояние Левенштейна для данных строк.

|  |
| --- |
| from typing import List, AnyStr  from itertools import product  def levenshtein\_distance(source: AnyStr, target: AnyStr) -> int:  len\_source: int = len(source)  len\_target: int = len(target)  # Создаем таблицу для хранения расстояний  distance\_table: List[List[int]] = [[0] \* (len\_target + 1) for \_ in range(len\_source + 1)]  # Инициализация первой строки и первого столбца  for i, j in zip(range(len\_source + 1), range(len\_target + 1)):  distance\_table[i][0] = i # Расстояние от source до пустой строки  distance\_table[0][j] = j # Расстояние от пустой строки до target  # Заполняем таблицу  for i, j in product(range(1, len\_source + 1), range(1, len\_target + 1)):  if source[i - 1] == target[j - 1]:  distance\_table[i][j] = distance\_table[i - 1][j - 1] # Символы совпадают  else:  distance\_table[i][j] = 1 + min(distance\_table[i - 1][j],  distance\_table[i][j - 1],  distance\_table[i - 1][j - 1]) # Удаление, вставка, замена  return distance\_table[len\_source][len\_target]  def main() -> None:  a: str = input()  b: str = input()  distance = levenshtein\_distance(a, b)  print(distance)  if \_\_name\_\_ == "\_\_main\_\_":  main() |



Задание 3.

Числовая последовательность задана рекуррентной формулой: ai+1ai+1=(k∗aik∗ai+bb)mod mm. Найдите её наибольшую возрастающую подпоследовательность.

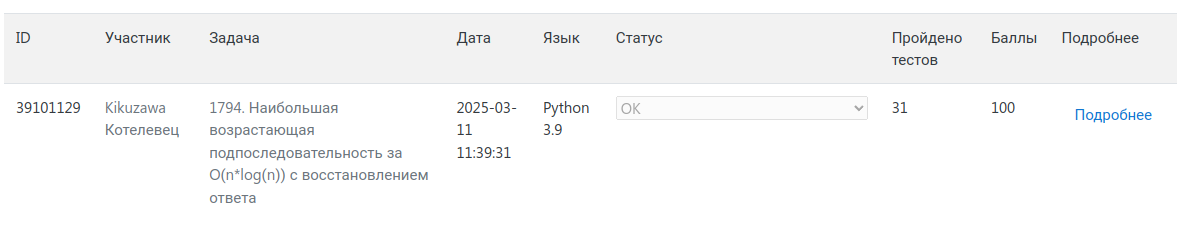
Входные данные

Программа получает на вход пять целых чисел: длину последовательности nn (1≤nn≤105105), начальный элемент последовательности a1a1, параметры kk, bb, mm для вычисления последующих членов последовательности (1≤mm≤104104, 0≤kk<mm, 0≤bb<mm, 0≤a1a1<mm).

Выходные данные

Требуется вывести наибольшую возрастающую подпоследовательность данной последовательности, разделяя числа пробелами. Если таких последовательностей несколько, необходимо вывести одну (любую) из них.

|  |
| --- |
| from bisect import bisect\_left  from typing import List, Sequence, Iterable  from collections import deque  def longest\_increasing\_subsequence(sequence: Sequence[int]) -> Iterable[int]:  largest\_increasing\_subsequence: List[int] = []  prev\_index: List[int] = [-1] \* len(sequence)  indices: List[int] = []  for i, value in enumerate(sequence):  pos = bisect\_left(largest\_increasing\_subsequence, value)  # Если значение больше всех элементов в lis, добавляем его  if pos == len(largest\_increasing\_subsequence):  largest\_increasing\_subsequence.append(value)  indices.append(i)  else:  largest\_increasing\_subsequence[pos] = value  indices[pos] = i  # Восстанавливаем индексы  if pos > 0:  prev\_index[i] = indices[pos - 1]  # Восстанавливаем саму последовательность  result: deque[int] = deque()  k = indices[-1]  while k != -1:  result.append(sequence[k])  k = prev\_index[k]  result.reverse()  return result  def main() -> None:  n, a1, k, b, m = map(int, input().split())  # Генерируем последовательность по формуле, которая дана в условии  sequence = [0] \* n  sequence[0] = a1  for i in range(1, n):  sequence[i] = (k \* sequence[i - 1] + b) % m  lis = longest\_increasing\_subsequence(sequence)  print(" ".join(map(str, lis)))  if \_\_name\_\_ == "\_\_main\_\_":  main() |



Задание 4.

Дана последовательность, требуется найти её наибольшую возрастающую подпоследовательность.

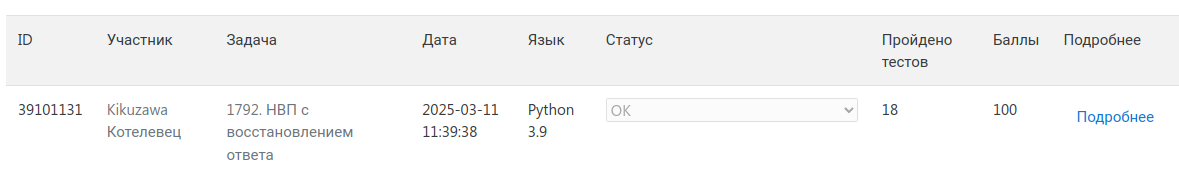
Входные данные

В первой строке входных данных задано число NN - длина последовательности (1 ≤ NN ≤ 1000). Во второй строке задается сама последовательность (разделитель - пробел). Элементы последовательности - целые числа, не превосходящие 10000 по модулю.

Выходные данные

Требуется вывести наибольшую возрастающую подпоследовательность данной последовательности. Если таких подпоследовательностей несколько, необходимо вывести одну (любую) из них.

|  |
| --- |
| from collections import deque  from typing import List, Iterable, Sequence  def longest\_increasing\_subsequence(sequence: Sequence[int]) -> Iterable[int]:  n = len(sequence)  distance\_table: List[int] = [1] \* n # Массив для хранения длины LIS до каждого элемента  prev\_index: List[int] = [-1] \* n # Массив для восстановления последовательности  # Заполняем массив dp  for i in range(n):  for j in range(i):  if sequence[j] < sequence[i] and distance\_table[j] + 1 > distance\_table[i]:  distance\_table[i] = distance\_table[j] + 1  prev\_index[i] = j  # Находим максимальную длину и индекс последнего элемента LCS  max\_length: int = max(distance\_table)  max\_index: int = distance\_table.index(max\_length)  # Восстанавливаем саму последовательность  result = deque()  while max\_index != -1:  result.append(sequence[max\_index])  max\_index = prev\_index[max\_index]  result.reverse()  return result  def main() -> None:  \_ = int(input())  sequence = list(map(int, input().split()))  lis = longest\_increasing\_subsequence(sequence)  # Выводим результат  print(' '.join(map(str, lis)))  if \_\_name\_\_ == "\_\_main\_\_":  main() |



Задание 5.

Дано N предметов массой m1, …, mN и стоимостью c1, …, cN соответственно.

Ими наполняют рюкзак, который выдерживает вес не более M. Определите набор предметов, который можно унести в рюкзаке, имеющий наибольшую стоимость.

Входные данные

В первой строке вводится натуральное число N, не превышающее 100 и натуральное число M, не превышающее 10000.

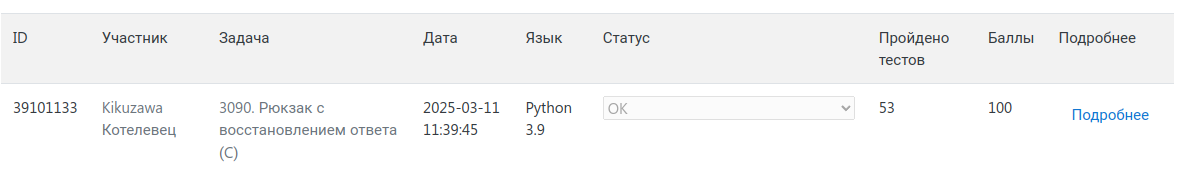
Во второй строке вводятся N натуральных чисел mi, не превышающих 100.

В третьей строке вводятся N натуральных чисел сi, не превышающих 100.

Выходные данные

Выведите номера предметов (числа от 1 до N), которые войдут в рюкзак наибольшей стоимости.

|  |
| --- |
| from itertools import product  from collections import deque  from typing import Iterable, Sequence, List  def knapsack(n: int, m: int, weights: Sequence[int], costs: Sequence[int]) -> Iterable[int]:  # Инициализация таблицы для динамического программирования  distance\_table: List[List[int]] = [[0] \* (m + 1) for \_ in range(n + 1)]  # Заполнение таблицы  for i, j in product(range(1, n + 1), range(m + 1)):  distance\_table[i][j] = distance\_table[i - 1][j] # Не берем текущий предмет  if j >= weights[i] and distance\_table[i - 1][j - weights[i]] + costs[i] > distance\_table[i][j]:  distance\_table[i][j] = distance\_table[i - 1][j - weights[i]] + costs[i] # Берем текущий предмет  # Восстановление набора предметов  selected\_items: deque[int] = deque()  remaining\_weight: int = m  for i in range(n, 0, -1):  if distance\_table[i][remaining\_weight] != distance\_table[i - 1][remaining\_weight]: # Если предмет n был выбран  selected\_items.append(i) # Добавляем номер предмета  remaining\_weight -= weights[i] # Уменьшаем оставшийся вес  selected\_items.reverse()  return selected\_items  def main() -> None:  n, m = map(int, input().split())  weights: List[int] = [0] + list(map(int, input().split()))  costs: List[int] = [0] + list(map(int, input().split()))  selected\_items = knapsack(n, m, weights, costs)  # Выводим результат  print(' '.join(map(str, selected\_items)))  if \_\_name\_\_ == "\_\_main\_\_":  main() |



Задание 6.

Дана строка, составленная из круглых, квадратных и фигурных скобок. Определите, какое наименьшее количество символов необходимо удалить из этой строки, чтобы оставшиеся символы образовывали правильную скобочную последовательность.

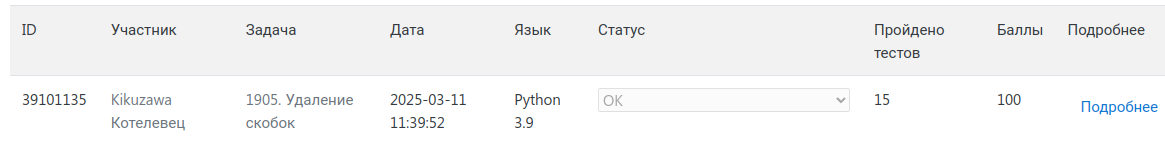
Входные данные

Строка из круглых, квадратных и фигурных скобок. Длина строки не превосходит 100 символов.

Выходные данные

Выведите строку максимальной длины, являющуюся правильной скобочной последовательностью, которую можно получить из исходной строки удалением некоторых символов.Если возможных ответов несколько, выведите любой из них.

|  |
| --- |
| from typing import Final, Dict, List, Sequence, AnyStr  BRACKET\_PAIRS: Final[Dict[str, str]] = {  '(': ')',  '[': ']',  '{': '}'  }  def reconstruct(  s: AnyStr,  distance\_table: Sequence[Sequence[int]],  table\_for\_restoring\_sequence: Sequence[Sequence[int]],  l: int,  r: int  ) -> AnyStr:  if distance\_table[l][r] == r - l + 1: # Все символы уже в правильной последовательности  return ""  if distance\_table[l][r] == 0: # Неправильная последовательность  return s[l:r + 1]  if table\_for\_restoring\_sequence[l][r] == -1: # Если нет разделения  return s[l] + reconstruct(s, distance\_table, table\_for\_restoring\_sequence, l + 1, r - 1) + s[r]  return (reconstruct(s, distance\_table, table\_for\_restoring\_sequence, l, table\_for\_restoring\_sequence[l][r]) +  reconstruct(s, distance\_table, table\_for\_restoring\_sequence, table\_for\_restoring\_sequence[l][r] + 1, r))  def fill\_tables(  s: AnyStr,  n: int,  distance\_table: List[List[int]],  table\_for\_restoring\_sequence: List[List[int]]  ) -> None:  for right in range(n):  for left in range(right, -1, -1):  if left == right:  distance\_table[left][right] = 1  else:  min\_removals = float('inf')  split\_index = -1  # Проверка на соответствие скобок  if s[left] in BRACKET\_PAIRS and s[right] == BRACKET\_PAIRS[s[left]]:  min\_removals = distance\_table[left + 1][right - 1]  # Разделение на подзадачи  for k in range(left, right):  current\_removals = distance\_table[left][k] + distance\_table[k + 1][right]  if min\_removals > current\_removals:  min\_removals = current\_removals  split\_index = k  distance\_table[left][right] = min\_removals  table\_for\_restoring\_sequence[left][right] = split\_index  def min\_removals\_to\_valid\_parentheses(s: AnyStr) -> AnyStr:  n: int = len(s)  # Таблица для хранения минимального количества удалений  # Один символ всегда является правильной последовательностью  distance\_table: List[List[int]] = [[1 if i == j else 0 for j in range(n)] for i in range(n)]  # Таблица для восстановления последовательности  table\_for\_restoring\_sequence: List[List[int]] = [[0] \* n for \_ in range(n)]  fill\_tables(s, n, distance\_table, table\_for\_restoring\_sequence)  return reconstruct(s, distance\_table, table\_for\_restoring\_sequence, 0, n - 1)  def main() -> None:  s: str = input()  result = min\_removals\_to\_valid\_parentheses(s)  print(result)  if \_\_name\_\_ == "\_\_main\_\_":  main() |



Задание 7.

Палиндромом называется строка, которая одинаково читается как слева направо, так и справа налево. Подпалиндромом данной строки называется последовательность символов из данной строки, не обязательно идущих подряд, являющаяся палиндромом. Например, HELOLEH является подпалиндромом строки HTEOLFEOLEH. Напишите программу, находящую в данной строке подпалиндром максимальной длины.

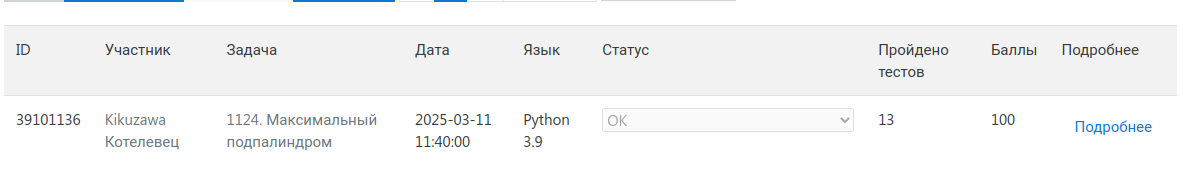
Входные данные

Во входном файле находится строка длиной не более 100 символов, состоящая из заглавных букв латинского алфавита.

Выходные данные

Выведите на первой строке выходного файла длину максимального подпалиндрома, а на второй строке сам максимальный подпалиндром. Если таких подпалиндромов несколько, то ваша программа должна вывести любой из них.

|  |
| --- |
| from typing import Tuple, List  from collections import deque  def fill\_table(n: int, s: str, distance\_table: List[List[int]]) -> None:  for length in range(2, n + 1): # Длина подстроки  for i in range(n - length + 1):  j = i + length - 1  if s[i] == s[j]:  distance\_table[i][j] = distance\_table[i + 1][j - 1] + 2  else:  distance\_table[i][j] = max(distance\_table[i + 1][j], distance\_table[i][j - 1])  def longest\_palindromic\_subsequence(s: str) -> Tuple[int, str]:  n = len(s)  # Таблица для хранения длины максимального подпалиндрома  # Все одиночные символы являются палиндромами длины 1  distance\_table: List[List[int]] = [[1 if i == j else 0 for j in range(n)] for i in range(n)]  fill\_table(n, s, distance\_table)  # Длина максимального подпалиндрома  max\_length: int = distance\_table[0][n - 1]  # Восстановление самого подпалиндрома  left\_index: int = 0  right\_index: int = n - 1  subsequence: deque[str] = deque()  while left\_index <= right\_index:  if s[left\_index] == s[right\_index]:  subsequence.append(s[left\_index])  left\_index += 1  right\_index -= 1  elif distance\_table[left\_index + 1][right\_index] >= distance\_table[left\_index][right\_index - 1]:  left\_index += 1  else:  right\_index -= 1  # Если длина подпалиндрома четная, то мы добавляем его в обратном порядке  # Если нечетная, то добавляем последний символ  palindromic\_subsequence = ''.join(subsequence)  if len(palindromic\_subsequence) \* 2 == max\_length:  result = palindromic\_subsequence + palindromic\_subsequence[::-1]  else:  result = palindromic\_subsequence + palindromic\_subsequence[-2::-1]  return max\_length, result  def main() -> None:  s = input()  max\_length, palindromic\_subsequence = longest\_palindromic\_subsequence(s)  print(max\_length)  print(palindromic\_subsequence)  if \_\_name\_\_ == "\_\_main\_\_":  main() |



Задание 8.

Как-то раз гному Кварку попала в руки карта сокровищ. На карте отмечено NN точек, в которых может находиться клад. Все точки пронумерованы числами от 11 до NN. Для каждой пары точек Кварк знает длину дороги, их соединяющей. Свои поиски Кварк начинает от точки с номером 11. Прежде чем начать свой долгий путь, хитрый гном вычеркивает точки, в которых, по его мнению, клада быть не может. Гарантируется, что точка с номером 11 никогда не бывает вычеркнута. После этого Кварк выбирает некоторый маршрут, проходящий через все оставшиеся на карте точки. Маршрут не проходит через одну и ту же точку более одного раза. Кварк может ходить только по дорогам, соединяющим невычеркнутые точки.

Кварк хочет выбрать маршрут минимальной длины. Необходимо найти такой маршрут для Кварка.

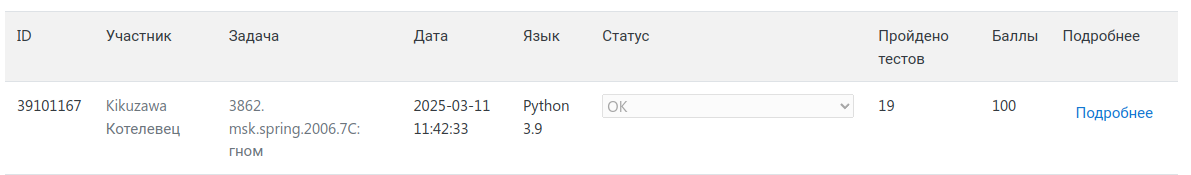
Входные данные

В первой строке находится одно целое число NN (1<N≤15)(1<N≤15) — количество точек, отмеченных на карте. В последующих NN строках находятся расстояния между точками. В (i+1)(i+1)-й строке находятся NN целых чисел di1,di2,…,diNdi1,di2,…,diN — длины дорог от ii-й точки до всех остальных. Гарантируется, что dij=djidij=dji, dii=0dii=0 и 0<dij<1000<dij<100. В (N+2)(N+2)-й строке находится одно целое число QQ (1<Q≤1000)(1<Q≤1000) — количество вариантов вычеркивания точек для данной карты. В последующих QQ строках содержится описание вариантов вычеркивания. Описание начинается с числа CC (0≤C<N)(0≤C<N) — количества точек, в которых, по мнению Кварка, клада быть не может. Следующие CC чисел задают номера этих точек.

Выходные данные

Выведите QQ строк. В каждой строке выведите одно целое число — длину минимального маршрута при соответствующем варианте вычеркивания точек.

|  |
| --- |
| def min\_path(mask, pos, n, dist):  # Базовый случай: если все точки посещены, возвращаемся в точку 11  if mask == ((1 << n) - 1):  return dist[pos][11]  ans = float('inf')  # Проверяем все непосещенные точки  for i in range(n):  if (mask & (1 << i)) == 0: # Если точка не посещена  # Вычисляем расстояние через текущую точку  ans = min(ans, dist[pos][i] + min\_path(mask | (1 << i), i, n, dist))  return ans  # Чтение входных данных  N = int(input())  dist = []  for \_ in range(N):  dist.append(list(map(int, input().split())))  Q = int(input())  results = []  # Обработка каждого варианта удаления точек  for \_ in range(Q):  # Читаем количество удаляемых точек и сами точки  C = list(map(int, input().split()))  count = C[0] # Количество точек для удаления  points = C[1:] # Список точек для удаления  # Создаём маску активных точек  active\_mask = (1 << N) - 1  for point in points:  active\_mask &= ~(1 << (point - 1)) # Приводим к нулевому индексу  # Если точка 11 не удалена и есть хотя бы одна активная точка  if active\_mask & (1 << 11):  result = min\_path(active\_mask, 11, N, dist)  results.append(str(result))  print('\n'.join(results)) |



Задание 9

Даны N золотых слитков известных масс. Определите, какую наибольшую массу золота можно унести, если вместимость рюкзака не превышает S.

Входные данные

Программа получает на вход целое число S — вместимость рюкзака, не превосходящее 10000 и количество слитков N, не превосходящее 300. Далее следует N целых неотрицательных чисел, не превосходящих 100000 — веса слитков.

Выходные данные

Программа должна вывести единственное целое число — максимально возможных вес золота, который поместится в данный рюкзак.

|  |
| --- |
| from typing import List  def max\_gold\_weight(capacity: int, weights: list) -> int:  achievable\_weights: List[int] = [0] \* (capacity + 1)  achievable\_weights[0] = 1 # Нулевой вес всегда достижим  # Проходим по каждому слитку  for weight in weights:  # Обновляем массив достижимых весов в обратном порядке  for current\_capacity in range(capacity, weight - 1, -1):  if achievable\_weights[current\_capacity - weight] == 1:  achievable\_weights[current\_capacity] = 1  # Находим максимальный вес, который можно унести  for i in range(capacity, -1, -1):  if achievable\_weights[i] == 1:  return i # Возвращаем максимальный вес  return 0 # Если ничего не удалось унести  def main() -> None:  capacity, n = map(int, input().split())  weights: List[int] = list(map(int, input().split()))  max\_weight: int = max\_gold\_weight(capacity, weights)  print(max\_weight)  if \_\_name\_\_ == "\_\_main\_\_":  main() |

Лабораторная работа 4

Задание 1

На стандартной шахматной доске (8×88×8) живут 2 шахматных коня: Красный и Зелёный. Обычно они беззаботно скачут по просторам доски, пощипывая шахматную травку, но сегодня особенный день: у Зелёного коня день рождения. Зелёный конь решил отпраздновать это событие вместе с Красным. Но для осуществления этого прекрасного плана им нужно оказаться на одной клетке. Заметим, что Красный и Зёленый шахматные кони сильно отличаются от черного с белым: они ходят не по очереди, а одновременно, и, если оказываются на одной клетке, никто никого не съедает. Сколько ходов им потребуется, чтобы насладиться праздником?

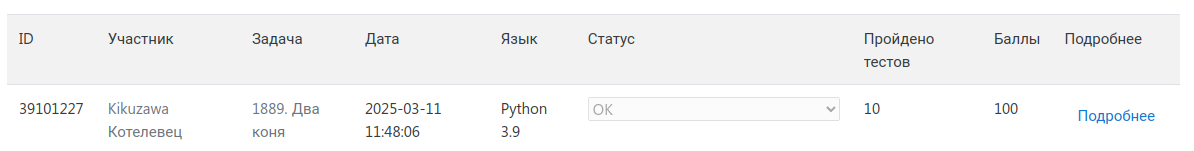
**Входные данные**

Во входном файле содержатся координаты коней, записанные по стандартным шахматным правилам (т. е. двумя символами — маленькая латинская буква (от a до h) и цифра (от 1 до 8), задающие столбец и строку соответственно).

**Выходные данные**

Выходной файл должен содержать наименьшее необходимое количество ходов, либо −1−1, если кони не могут встретиться.

|  |
| --- |
| from collections import deque  from itertools import product  from typing import Tuple, Generator, Any, Set  def get\_moves(x: int, y: int) -> Generator[Tuple[int, int], Any, None]:  moves: Tuple[Tuple[int, int], ...] = (  (x + 1, y + 2), (x + 2, y + 1), (x + 2, y - 1), (x + 1, y - 2),  (x - 1, y - 2), (x - 2, y - 1), (x - 2, y + 1), (x - 1, y + 2)  )  yield from ((mx, my) for mx, my in moves if 0 <= mx <= 7 and 0 <= my <= 7)  def knights\_meet(knight1: Tuple[int, int], knight2: Tuple[int, int]) -> int:  visited: Set[Tuple[Tuple[int, int], Tuple[int, int]]] = set()  queue: deque[Tuple[Tuple[int, int], Tuple[int, int], int]] = deque([(knight1, knight2, 0)])  while queue:  k1, k2, moves = queue.popleft()  if k1 == k2:  return moves  visited.add((k1, k2))  for move1, move2 in product(get\_moves(k1[0], k1[1]), get\_moves(k2[0], k2[1])):  if (move1, move2) not in visited:  visited.add((move1, move2)) # Добавляем в visited сразу  queue.append((move1, move2, moves + 1))  return -1  def main() -> None:  knight1, knight2 = map(lambda k: (ord(k[0]) - ord('a'), int(k[1]) - 1), input().split())  result: int = knights\_meet(knight1, knight2)  print(result)  if \_\_name\_\_ == "\_\_main\_\_":  main() |



Задание 2

На банкет были приглашены N Очень Важных Персон (ОВП). Были поставлены 2 стола. Столы достаточно большие, чтобы все посетители банкета могли сесть за любой из них. Проблема заключается в том, что некоторые ОВП не ладят друг с другом и не могут сидеть за одним столом. Вас попросили определить, возможно ли всех ОВП рассадить за двумя столами.

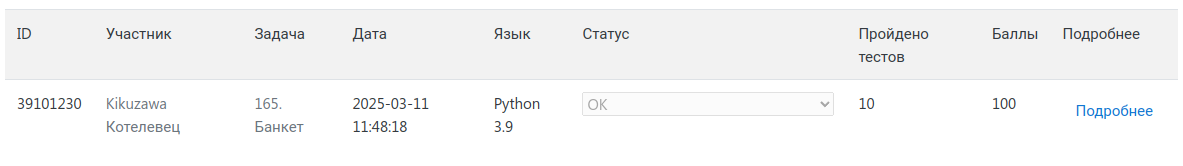
Входные данные

В первой строке входных данных содержатся два числа: N и M (1 <= N,M <= 100), где N – количество ОВП, а M – количество пар ОВП, которые не могут сидеть за одним столом. В следующих M строках записано по 2 числа – пары ОВП, которые не могут сидеть за одним столом.

Выходные данные

Если способ рассадить ОВП существует, то выведите YES в первой строке и номера ОВП, которых необходимо посадить за первый стол, во второй строке. В противном случае в первой и единственной строке выведите NO.

|  |
| --- |
| from collections import deque  from typing import List, cast, Tuple  class Graph:  def \_\_init\_\_(self, n: int) -> None:  """Инициализируем граф с n вершинами."""  self.\_n = n  self.\_adj\_list: List[List[int]] = [[] for \_ in range(n)] # Список смежности для графа  self.\_color: List[int] = []  def add\_edge(self, u: int, v: int) -> None:  """Добавляем ребро между вершинами u и v."""  self.\_adj\_list[u - 1].append(v - 1)  self.\_adj\_list[v - 1].append(u - 1)  def is\_bipartite(self) -> bool:  self.\_color = [-1] \* self.\_n # -1: не окрашена, 0: первая доля, 1: вторая доля  def bfs(start\_inner: int) -> bool:  queue: deque[int] = deque([start\_inner])  self.\_color[start\_inner] = 0  while queue:  v = queue.popleft()  for u in self.\_adj\_list[v]:  if self.\_color[u] == -1:  self.\_color[u] = 1 - self.\_color[v]  queue.append(u)  elif self.\_color[u] == self.\_color[v]:  return False  return True  for start in range(self.\_n):  if self.\_color[start] == -1 and not bfs(start): # Если вершина не посещена  return False  return True  @property  def bipartite\_partition(self) -> List[int]:  if self.is\_bipartite():  return [i + 1 for i in range(self.\_n) if self.\_color[i] == 0]  return []  def main() -> None:  n, m = map(int, input().split())  pairs = cast(List[Tuple[int, int]], [tuple(map(int, input().split())) for \_ in range(m)])  # Создаем граф и добавляем ребра  graph = Graph(n)  for u, v in pairs:  graph.add\_edge(u, v)  # Проверяем, возможно ли рассадить  result = graph.bipartite\_partition  if result:  print("YES")  print(\*result)  else:  print("NO")  if \_\_name\_\_ == "\_\_main\_\_":  main() |



Задание 3

один из нескольких возможных рельефов, для простоты ограничимся тремя видами рельефов - поле, лес и вода. Поселенец перемещается по карте, при этом на перемещение в клетку, занятую полем, необходима одна единица времени, на перемещение в лес - две единицы времени, а перемещаться в клетку с водой нельзя.

У вас есть один поселенец, вы определили место, где нужно построить город, чтобы как можно скорее завладеть всем миром. Найдите маршрут переселенца, приводящий его в место строительства города, требующий минимального времени. На каждом ходе переселенец может перемещаться в клетку, имеющую общую сторону с той клеткой, где он сейчас находится.

Входные данные

Во входном файле записаны два натуральных числа N и M, не превосходящих 1000 - размеры карты мира (N - число строк в карте, M - число столбцов). Затем заданы координаты начального положения поселенца x и y, где x - номер строки, y - номер стролбца на карте (1 ≤ x ≤ N, 1 ≤ y ≤ M), строки нумеруются сверху вниз, столбцы - слева направо. Затем аналогично задаются координаты клетки, куда необходимо привести поселенца.

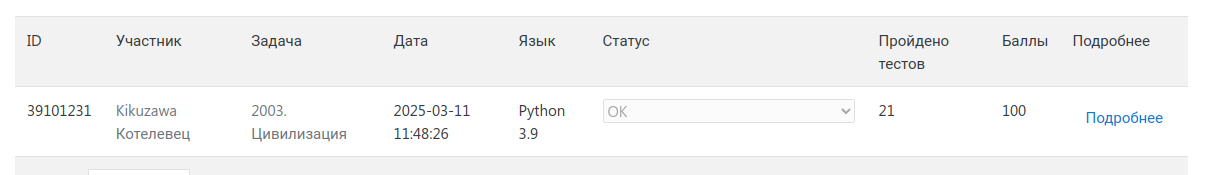
Далее идет описание карты мира в виде N строк, каждая из которых содержит M символов. Каждый символ может быть либо “.” (точка), обозначающим поле, либо “W”, обозначающим лес, либо “#”, обозначающим воду. Гарантируется, что начальная и конечная клетки пути переселенца не являются водой.

Выходные данные

В первой строке выходного файла выведите количество единиц времени, необходимое для перемещения поселенца (перемещение в клетку с полем занимает 1 единицу времени, перемещение в клетку с лесом - 2 единицы времени). Во второй строке выходного файла выведите последовательность символов, задающих маршрут переселенца. Каждый символ должен быть одним из четырех следующих: “N” (движение вверх), “E” (движение вправо), “S” (движение вниз), “W” (движение влево). Если таких маршрутов несколько, выведите любой из них.

Если дойти из начальной клетки в конечную невозможно, выведите число -1.

|  |
| --- |
| from collections import deque  from typing import List, cast, Tuple  class Graph:  def \_\_init\_\_(self, n: int) -> None:  """Инициализируем граф с n вершинами."""  self.\_n = n  self.\_adj\_list: List[List[int]] = [[] for \_ in range(n)] # Список смежности для графа  self.\_color: List[int] = []  def add\_edge(self, u: int, v: int) -> None:  """Добавляем ребро между вершинами u и v."""  self.\_adj\_list[u - 1].append(v - 1)  self.\_adj\_list[v - 1].append(u - 1)  def is\_bipartite(self) -> bool:  self.\_color = [-1] \* self.\_n # -1: не окрашена, 0: первая доля, 1: вторая доля  def bfs(start\_inner: int) -> bool:  queue: deque[int] = deque([start\_inner])  self.\_color[start\_inner] = 0  while queue:  v = queue.popleft()  for u in self.\_adj\_list[v]:  if self.\_color[u] == -1:  self.\_color[u] = 1 - self.\_color[v]  queue.append(u)  elif self.\_color[u] == self.\_color[v]:  return False  return True  for start in range(self.\_n):  if self.\_color[start] == -1 and not bfs(start): # Если вершина не посещена  return False  return True  @property  def bipartite\_partition(self) -> List[int]:  if self.is\_bipartite():  return [i + 1 for i in range(self.\_n) if self.\_color[i] == 0]  return []  def main() -> None:  n, m = map(int, input().split())  pairs = cast(List[Tuple[int, int]], [tuple(map(int, input().split())) for \_ in range(m)])  # Создаем граф и добавляем ребра  graph = Graph(n)  for u, v in pairs:  graph.add\_edge(u, v)  # Проверяем, возможно ли рассадить  result = graph.bipartite\_partition  if result:  print("YES")  print(\*result)  else:  print("NO")  if \_\_name\_\_ == "\_\_main\_\_":  main() |



Задание 4.

Предприятие «Авто-2010» выпускает двигатели для известных во всём мире автомобилей. Двигатель состоит ровно из n

деталей, пронумерованных от 1 до n

, при этом деталь с номером i

изготавливается за pi

секунд. Специфика предприятия «Авто-2010» заключается в том, что там одновременно может изготавливаться лишь одна деталь двигателя. Для производства некоторых деталей необходимо иметь предварительно изготовленный набор других деталей.

Генеральный директор «Авто-2010» поставил перед предприятием амбициозную задачу — за наименьшее время изготовить деталь с номером 1, чтобы представить её на выставке.

Требуется написать программу, которая по заданным зависимостям порядка производства между деталями найдёт наименьшее время, за которое можно произвести деталь с номером 1.

Входные данные

Первая строка входного файла содержит число n

(1≤n≤100000

) — количество деталей двигателя. Вторая строка содержит n

натуральных чисел p1,p2,…,pn

, определяющих время изготовления каждой детали в секундах. Время для изготовления каждой детали не превосходит 109

секунд.

Каждая из последующих n

строк входного файла описывает характеристики производства деталей. Здесь i

-я строка содержит число деталей ki

, которые требуются для производства детали с номером i

, а также их номера. В i

-й строке нет повторяющихся номеров деталей. Сумма всех чисел ki

не превосходит 200000.

Известно, что не существует циклических зависимостей в производстве деталей.

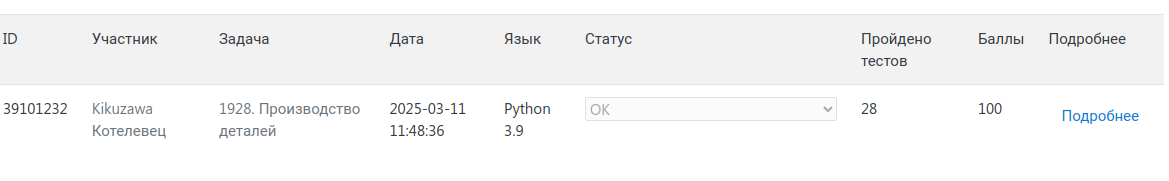
Выходные данные

В первой строке выходного файла должны содержаться два числа: минимальное время (в секундах), необходимое для скорейшего производства детали с номером 1 и число k

деталей, которые необходимо для этого произвести. Во второй строке требуется вывести через пробел k

чисел — номера деталей в том порядке, в котором следует их производить для скорейшего производства детали с номером 1.

|  |
| --- |
| from typing import List, Tuple, Sequence  from collections import deque  def produce\_part(  part\_id: int,  production\_times: Sequence[int],  dependencies: Sequence[Sequence[int]],  produced: List[bool],  production\_order: deque[int]  ) -> int:  if produced[part\_id]:  return 0  produced[part\_id] = True  total\_time = production\_times[part\_id] # Время на изготовление текущей детали  for dependency in dependencies[part\_id - 1]: # Индексация зависит от 0  total\_time += produce\_part(dependency, production\_times, dependencies, produced, production\_order)  production\_order.append(part\_id)  return total\_time  def calculate\_minimum\_production\_time(  n: int,  production\_times: Sequence[int],  dependencies: Sequence[Sequence[int]]  ) -> Tuple[int, Sequence[int]]:  produced: List[bool] = [False] \* (n + 1)  production\_order: deque[int] = deque()  total\_time: int = produce\_part(1, production\_times, dependencies, produced, production\_order)  return total\_time, production\_order  def main() -> None:  n: int = int(input())  production\_times: List[int] = [0] + list(map(int, input().split()))  dependencies: List[List[int]] = [list(map(int, input().split()[1:])) for \_ in range(n)]  total\_time, order = calculate\_minimum\_production\_time(n, production\_times, dependencies)  print(total\_time, len(order))  print(\*order)  if \_\_name\_\_ == "\_\_main\_\_":  main() |



Задание 5.

В стране N

городов, некоторые из которых соединены между собой дорогами. Для того, чтобы проехать по одной дороге, требуется один бак бензина. Помимо этого у вас есть канистра для бензина, куда входит столько же топлива, сколько входит в бензобак.

В каждом городе бак бензина имеет разную стоимость. Вам требуется добраться из первого города в N

-й, потратив как можно меньшее денег.

В каждом городе можно заправить бак, заправить бак и канистру или же перелить бензин из канистры в бак. Это позволяет экономить деньги, покупая бензин в тех городах, где он стоит дешевле, но канистры хватает только на одну заправку бака!

Входные данные

В первой строке вводится число N

(1≤N≤100

), в следующей строке идет N

чисел, i

-е из которых задает стоимость бензина в i

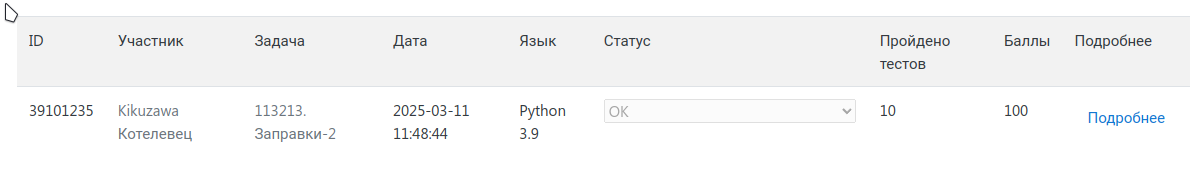
-м городе (всё это целые числа из диапазона от 0 до 100). Затем идет число M

– количество дорог в стране, далее идет описание самих дорог. Каждая дорога задается двумя числами – номерами городов, которые она соединяет. Все дороги двухсторонние (то есть по ним можно ездить как в одну, так и в другую сторону), между двумя городами всегда существует не более одной дороги, не существует дорог, ведущих из города в себя.

Выходные данные

Требуется вывести одно число – суммарную стоимость маршрута или -1, если добраться невозможно.

|  |
| --- |
| from collections import defaultdict  from dataclasses import dataclass  from functools import total\_ordering  from heapq import heappop, heappush  from typing import List, Tuple, Dict, Set, cast  @dataclass(frozen=True)  @total\_ordering  class State:  cost: int  city: int  tank: int  canister: int  def \_\_lt\_\_(self, other: 'State') -> bool:  return self.cost < other.cost  def \_\_eq\_\_(self, other: object) -> bool:  if not isinstance(other, State):  return NotImplemented  return (self.cost, self.city, self.tank, self.canister) == (other.cost, other.city, other.tank, other.canister)  class Graph:  def \_\_init\_\_(self) -> None:  self.\_adjacency\_list: Dict[int, List[int]] = defaultdict(list)  def add\_edge(self, u: int, v: int) -> None:  self.\_adjacency\_list[u].append(v)  self.\_adjacency\_list[v].append(u)  def get\_neighbors(self, city: int) -> List[int]:  return self.\_adjacency\_list[city]  def min\_fuel\_cost(n: int, fuel\_costs: List[int], graph: Graph) -> int:  # Используем очередь с приоритетом  pq: List[State] = [State(0, 1, 0, 0)] # начинаем с города 1, пустой бак и канистра  visited: Set[Tuple[int, int, int]] = set()  while pq:  state = heappop(pq)  # Если добрались до города N, возвращаем стоимость  if state.city == n:  return state.cost  # Проверка на повторное посещение состояния  if (state.city, state.tank, state.canister) in visited:  continue  visited.add((state.city, state.tank, state.canister))  # 1. Заправка только бака  if state.tank == 0:  heappush(pq, State(state.cost + fuel\_costs[state.city - 1], state.city, 1, state.canister))  # 2. Заправка бака и канистры  if state.tank == 0 and state.canister == 0:  heappush(pq, State(state.cost + 2 \* fuel\_costs[state.city - 1], state.city, 1, 1))  # 3. Переливание бензина из канистры в бак  if state.tank == 0 and state.canister > 0:  heappush(pq, State(state.cost, state.city, 1, 0))  # 4. Переход в соседний город (тратим 1 единицу топлива)  if state.tank > 0:  for neighbor in graph.get\_neighbors(state.city):  heappush(pq, State(state.cost, neighbor, state.tank - 1, state.canister))  return -1 # Если не удалось добраться до города N  def main() -> None:  n: int = int(input())  fuel\_costs: List[int] = list(map(int, input().split()))  m: int = int(input())  roads: List[Tuple[int, int]] = cast(List[Tuple[int, int]], [tuple(map(int, input().split())) for \_ in range(m)])  graph = Graph()  for u, v in roads:  graph.add\_edge(u, v)  print(min\_fuel\_cost(n, fuel\_costs, graph))  if \_\_name\_\_ == '\_\_main\_\_':  main() |



Задание 6

Группа Pink Floyd собирается отправиться в новый концертный тур по всему миру. По предыдущему опыту группа знает, что солист Роджер Уотерс постоянно нервничает при перелетах. На некоторых маршрутах он теряет вес от волнения, а на других — много ест и набирает вес.

Известно, что чем больше весит Роджер, тем лучше выступает группа, поэтому требуется спланировать перелеты так, чтобы вес Роджера на каждом концерте был максимально возможным.

Группа должна посещать города в том же порядке, в котором она дает концерты. При этом между концертами группа может посещать промежуточные города.

Входные данные

Первая строка входного файла содержит три натуральных числа n

, m

и k

— количество городов в мире, количество рейсов и количество концертов, которые должна дать группа соответственно (n≤100

, m≤10000

, 2≤k≤10000

). Города пронумерованы числами от 1

до n

.

Следующие m

строк содержат описание рейсов, по одному на строке. Рейс номер i

описывается тремя числами bi

, ei

и wi

— номер начального и конечного города рейса и предполагаемое изменение веса Роджера в миллиграммах (1≤bi,ei≤n

, −100000≤wi≤100000

).

Последняя строка содержит числа a1,a2,...,ak

— номера городов, в которых проводятся концерты (ai≠ai+1

). В начале концертного тура группа находится в городе a1

.

Гарантируется, что группа может дать все концерты.

Выходные данные

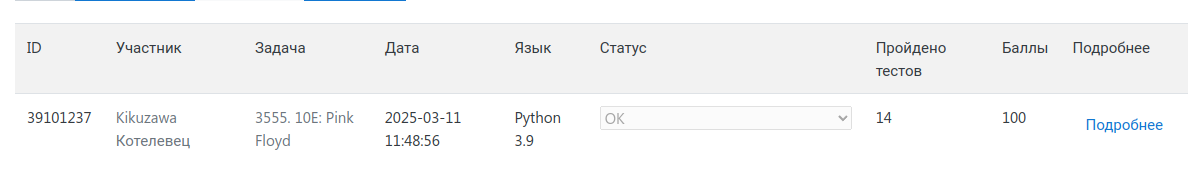
Первая строка выходного файла должна содержать число l

— количество рейсов, которые должна сделать группа. Вторая строка должна содержать l

чисел — номера используемых рейсов.

Если существует такая последовательность маршрутов между концертами, что Роджер будет набирать вес неограниченно, то первая строка выходного файла должна содержать строку „infinitely kind“.

|  |
| --- |
| import sys  from dataclasses import dataclass  from typing import List, Tuple  INF = sys.maxsize // 2  @dataclass(frozen=True)  class Flight:  start: int  end: int  weight\_change: int  index: int  def find\_path\_between\_concerts(  n: int,  c: int,  flights: List[Flight],  concerts: List[int]  ) -> Tuple[int, List[int]]:  matrix = [[0 if i == j else INF for j in range(n)] for i in range(n)]  parents = [[0 for \_ in range(n)] for \_ in range(n)]  for flight in flights:  matrix[flight.start][flight.end] = -flight.weight\_change  parents[flight.start][flight.end] = flight.index  # Алгоритм Флойда-Уоршелла для нахождения кратчайших путей  for k in range(n):  for i in range(n):  for j in range(n):  if matrix[i][j] > matrix[i][k] + matrix[k][j]:  matrix[i][j] = matrix[i][k] + matrix[k][j]  parents[i][j] = parents[i][k]  # Проверка на отрицательные циклы  for i in range(c):  if matrix[concerts[i]][concerts[i]] < 0:  return -1, []  # Строим маршрут между концертами  path = []  for i in range(c - 1):  v = concerts[i]  while v != concerts[i + 1]:  path.append(parents[v][concerts[i + 1]])  v = flights[parents[v][concerts[i + 1]]].end  if len(path) > 10000000:  return -1, []  return len(path), [p + 1 for p in path]  def main() -> None:  n, m, c = map(int, input().split())  flights: List[Flight] = []  for i in range(m):  vertex1, vertex2, w = map(int, input().split())  vertex1 -= 1 # Приводим города к индексации с нуля  vertex2 -= 1  flights.append(Flight(vertex1, vertex2, w, i))  concerts = [i - 1 for i in map(int, input().split())]  # Получаем путь между концертами  result, path = find\_path\_between\_concerts(n, c, flights, concerts)  if result == -1:  print("infinitely kind")  else:  print(result)  print(" ".join(map(str, path)))  if \_\_name\_\_ == "\_\_main\_\_":  main() |



Задание 7

Дан ориентированный граф. Определить, есть ли в нем цикл отрицательного веса, и если да, то вывести его.

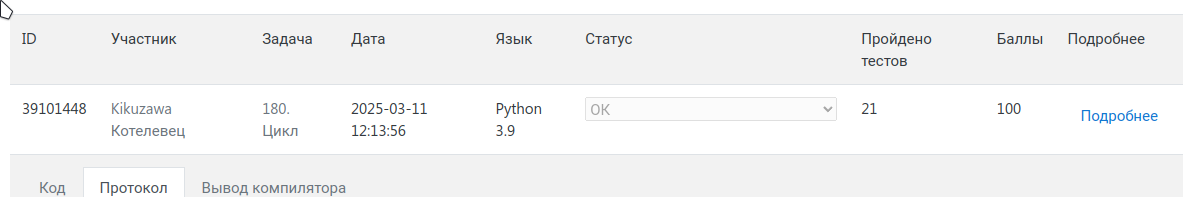
Входные данные

В первой строке содержится число N (1 <= N <= 100) – количество вершин графа. В следующих N строках находится по N чисел – матрица смежности графа. Веса ребер по модулю меньше 100000. Если ребра нет, соответствующее значение равно 100000.

Выходные данные

В первой строке выведите "YES", если цикл существует, или "NO", в противном случае. При наличии цикла выведите во второй строке количество вершин в нем (считая одинаковые – первую и последнюю), а в третьей строке – вершины, входящие в этот цикл, в порядке обхода. Если циклов несколько, то выведите любой из них.

|  |
| --- |
| from array import array, ArrayType  from collections import deque  from itertools import chain  from typing import Final, Sequence, Dict, List, Tuple  INF: Final[float] = float('inf')  def restore\_cycle(path: Sequence[int], start\_index: int) -> Sequence[int]:  if start\_index == -1:  return []  cycle\_start = path[path.index(start\_index)]  p = deque([cycle\_start])  cycle = path[cycle\_start]  while cycle != cycle\_start:  p.appendleft(cycle)  cycle = path[cycle]  p.appendleft(cycle\_start)  return p  def main() -> None:  n: int = int(input())  graph: Dict[int, List[Tuple[int, int]]] = {  i: [(j, w) for j, w in enumerate(map(int, input().split())) if w < 100000]  for i in range(n)  }  distance: ArrayType[float] = array("d", [INF] \* n)  path: ArrayType[int] = array("i", [-1] \* n)  cycle: int = -1  # Алгоритм Беллмана-Форда  # Тут слишком сильные временные рамки, поэтому если выносить в отдельную функцию, то вместо 20 тестов будет 16 пройдено.  for k in filter(lambda x: distance[x] == INF, range(n)):  distance[k] = 0  for \_ in range(n):  cycle = -1  for u, (v, weight) in chain.from\_iterable(((u, edge) for edge in edges) for u, edges in graph.items()):  if distance[u] < INF and distance[v] > distance[u] + weight:  distance[v] = distance[u] + weight  path[v] = u  cycle = v  result = restore\_cycle(path, cycle)  if result:  print("YES", len(result), " ".join(str(node + 1) for node in result), sep="\n")  else:  print("NO")  if \_\_name\_\_ == '\_\_main\_\_':  main() |



Задание 8

Между N

населенными пунктами совершаются пассажирские рейсы на машинах времени.

В момент времени 0 вы находитесь в пункте A

. Вам дано расписание рейсов. Требуется оказаться в пункте B как можно раньше (то есть в наименьший возможный момент времени).

При этом разрешается делать пересадки с одного рейса на другой. Если вы прибываете в некоторый пункт в момент времени T

, то вы можете уехать из него любым рейсом, который отправляется из этого пункта в момент времени T

или позднее (но не раньше).

Входные данные

В первой строке вводится число N

– количество населенных пунктов ( 1≤

N≤

1000). Вторая строка содержит два числа A

и B

– номера начального и конечного пунктов. В третьей строке задается K

– количество рейсов ( 0≤

K≤

1000). Следующие K

строк содержат описания рейсов, по одному на строке. Каждое описание представляет собой четверку целых чисел. Первое число каждой четверки задает номер пункта отправления, второе – время отправления, третье – пункт назначения, четвертое – время прибытия. Номера пунктов – натуральные числа из диапазона от 1 до N

. Пункт назначения и пункт отправления могут совпадать. Время измеряется в некоторых абсолютных единицах и задается целым числом, по модулю не превышающим 109

. Поскольку рейсы совершаются на машинах времени, то время прибытия может быть как больше времени отправления, так и меньше, или равным ему.

Гарантируется, что входные данные таковы, что добраться из пункта A

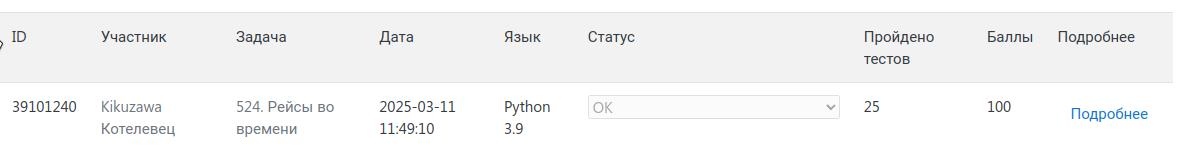
в пункт B

всегда можно.

Выходные данные

Выведите минимальное время, когда вы сможете оказаться в пункте B

|  |
| --- |
| import heapq  from collections import defaultdict  from dataclasses import dataclass  from typing import List, Dict, Tuple  @dataclass  class Flight:  src: int  dep\_time: int  dest: int  arr\_time: int  class Graph:  def \_\_init\_\_(self) -> None:  # Словарь, где ключ - номер пункта, а значение - список кортежей, представляющих соседние пункты  # с временем отправления и прибытия.  self.\_adjacency\_list: Dict[int, List[Tuple[int, int, int]]] = defaultdict(list)  def add\_flight(self, flight: Flight) -> None:  self.\_adjacency\_list[flight.src].append((flight.dest, flight.dep\_time, flight.arr\_time))  def get\_neighbors(self, node: int) -> List[Tuple[int, int, int]]:  return self.\_adjacency\_list.get(node, [])  def min\_time\_to\_destination(n: int, a: int, b: int, graph: Graph) -> int:  min\_time = [float('inf')] \* (n + 1)  min\_time[a] = 0  heap = [(0, a)]  while heap:  time, node = heapq.heappop(heap)  if time > min\_time[node]:  continue  for dest, dep\_time, arr\_time in graph.get\_neighbors(node):  if dep\_time >= time and min\_time[dest] > arr\_time:  min\_time[dest] = arr\_time  heapq.heappush(heap, (arr\_time, dest))  return round(min\_time[b])  def main() -> None:  n: int = int(input())  a, b = map(int, input().split())  k: int = int(input())  flights: List[Flight] = [Flight(\*map(int, input().split())) for \_ in range(k)]  graph = Graph()  for flight in flights:  graph.add\_flight(flight)  result: int = min\_time\_to\_destination(n, a, b, graph)  print(result)  if \_\_name\_\_ == "\_\_main\_\_":  main() |



Лабораторная работа 5

Задание 1

На плоскости заданы координаты n (4 ≤ n ≤ 20) разных вершин.

Найти кратчайший замкнутый маршрут, начинающийся и заканчивающийся в 1-й вершине и посещающий все остальные вершины по одному разу. Разрешается (если так оказывается выгодно) «проезжать через вершину, не останавливаясь» (см. пример 1).

Длина маршрута считается как сумма длин составляющих его рёбер, длины отдельных рёбер считаются согласно обычной евклидовой метрике, как .

Входные данные

Первая строка содержит количество вершин n (4 ≤ n ≤ 20). Каждая из следующих n строк содержит по два разделённых пробелом числа с плавающей точкой — x- и y-координаты соответствующей вершины.

Выходные данные

Первая строка должна содержать единственное число (с плавающей точкой) — найденную минимальную длину замкнутого тура. Вторая строка должна содержать перестановку чисел 2, 3, ..., N — порядок, в котором надо посещать эти вершины. Числа внутри второй строки должны разделяться одинарными пробелами.

Подзадачи и система оценки

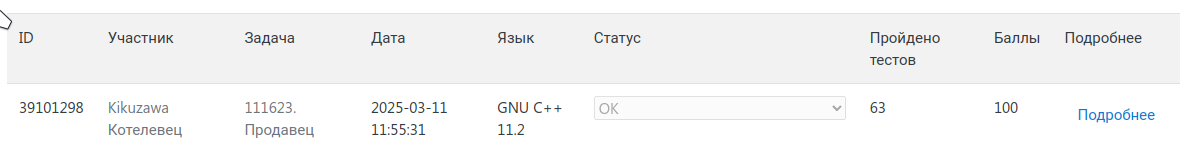
В этой задаче есть 3 подзадачи

Тест 1 (0 баллов). Это тест из условия

Тесты 2-25 (40 баллов). В этой подзадаче N≤12

Тесты 26-63 (60 баллов). В этой подзадаче дополнительные ограничения отсутствуют

|  |
| --- |
| #include <iostream>  #include <vector>  #include <cmath>  #include <limits>  #include <deque>  #include <tuple>  #include <algorithm>  using namespace std;  class TravelingSalesmanGraph {  public:  TravelingSalesmanGraph(const vector<pair<double, double>>& coordinates)  : num\_vertices(coordinates.size()), coordinates(coordinates) {  distance\_matrix.resize(1 << num\_vertices, vector<double>(num\_vertices, numeric\_limits<double>::infinity()));  distance\_matrix[1][0] = 0.0; // Начальная вершина посещена, длина пути 0  initialize\_distance\_matrix();  }  int get\_num\_vertices() const {  return num\_vertices;  }  pair<double, int> find\_shortest\_path\_length() {  int final\_mask = (1 << num\_vertices) - 1;  double min\_distance = numeric\_limits<double>::infinity();  int end\_vertex = -1;  for (int u = 1; u < num\_vertices; ++u) {  double distance = distance\_matrix[final\_mask][u] + calculate\_distance(u, 0);  if (distance < min\_distance) {  min\_distance = distance;  end\_vertex = u;  }  }  return {min\_distance, end\_vertex};  }  vector<int> reconstruct\_path(int end\_vertex, int final\_mask) {  deque<int> path;  int mask = final\_mask;  int current\_vertex = end\_vertex;  while (current\_vertex != 0) {  path.push\_front(current\_vertex);  double current\_distance = distance\_matrix[mask][current\_vertex];  for (int prev\_vertex = 0; prev\_vertex < num\_vertices; ++prev\_vertex) {  if ((mask & (1 << prev\_vertex)) &&  abs(current\_distance - (distance\_matrix[mask ^ (1 << current\_vertex)][prev\_vertex] + calculate\_distance(prev\_vertex, current\_vertex))) < 1e-9) {  current\_vertex = prev\_vertex;  mask ^= (1 << path.front());  break;  }  }  }  vector<int> result(path.begin(), path.end());  return result;  }  private:  int num\_vertices;  vector<pair<double, double>> coordinates;  vector<vector<double>> distance\_matrix;  void initialize\_distance\_matrix() {  for (int mask = 0; mask < (1 << num\_vertices); ++mask) {  for (int u = 0; u < num\_vertices; ++u) {  if (!(mask & (1 << u))) continue;  double current\_distance = distance\_matrix[mask][u];  for (int v = 0; v < num\_vertices; ++v) {  if (!(mask & (1 << v))) {  int next\_mask = mask | (1 << v);  double new\_distance = current\_distance + calculate\_distance(u, v);  distance\_matrix[next\_mask][v] = min(distance\_matrix[next\_mask][v], new\_distance);  }  }  }  }  }  double calculate\_distance(int u, int v) {  double dx = coordinates[u].first - coordinates[v].first;  double dy = coordinates[u].second - coordinates[v].second;  return sqrt(dx \* dx + dy \* dy);  }  };  pair<double, vector<int>> find\_optimal\_tour(const vector<pair<double, double>>& coordinates) {  TravelingSalesmanGraph graph(coordinates);  auto [shortest\_path\_length, end\_vertex] = graph.find\_shortest\_path\_length();  auto path = graph.reconstruct\_path(end\_vertex, (1 << graph.get\_num\_vertices()) - 1);  for (int& vertex : path) {  vertex += 1; // Преобразование индекса в 1-based  }  return {shortest\_path\_length, path};  }  int main() {  int num\_vertices;  cin >> num\_vertices;  vector<pair<double, double>> coordinates(num\_vertices);  for (int i = 0; i < num\_vertices; ++i) {  cin >> coordinates[i].first >> coordinates[i].second;  }  auto [shortest\_path\_length, optimal\_path] = find\_optimal\_tour(coordinates);  cout.precision(14);  cout << fixed << shortest\_path\_length << endl;  for (size\_t i = 0; i < optimal\_path.size(); ++i) {  if (i > 0) cout << " ";  cout << optimal\_path[i];  }  cout << endl;  return 0;  } |



Задание 2

В королевстве N

; городов, пронумерованных от 1 до N

. Столица имеет номер 1. Каждый город окружен городской стеной с 4 воротами. Ворота пронумерованы следующим образом: ворота i

-го города (1 ≤

i

≤

N

) имеют номера 4i

−3, 4i

−2, 4i

−1, 4i

. Через каждые ворота проходит ровно 1 дорога, которая ведет до некоторых ворот другого города (заметьте, что может существовать несколько дорог между двумя городами). По всем дорогам можно двигаться в обоих направлениях. Благодаря системе туннелей и мостов дороги не пересекаются вне городов.

Королевский гонец должен развесить копии Очень важного Королевского Указа на внешней стороне всех ворот каждого города. Гонец может свободно передвигаться от одних ворот к другим в пределах города, но вне города он может двигаться только по дорогам. Гонец выезжает из столицы и должен туда вернуться после выполнения задания.

Может ли гонец выполнить поручение, проходя через каждые ворота только один раз? Выход из города через ворота только для того, чтобы вывесить на их внешней стороне указ, а затем немедленное возвращение в город, считается за один проход через ворота.

Входные данные

Первая строка входного файла содержит целое число N

(2 ≤

N

≤

1000). Каждая из последующих 2N

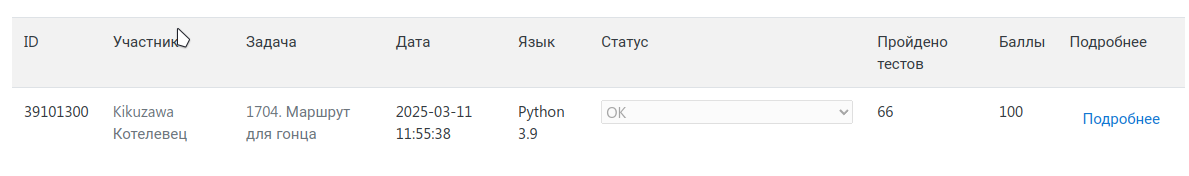
строк описывает одну дорогу и содержит 2 целых числа, разделенных пробелом: номера ворот, соединенных дорогой.

Выходные данные

В первую строку выходного файла выведите Yes или No в зависимости от того, может ли поручение гонца быть выполнено, если проходить через ворота только один раз. В случае если это возможно, вторая строка выходного файла должна содержать 4N

целых чисел, разделенных пробелом: номера ворот в порядке прохождения через них гонцом. Если существует несколько решений, выведите любое из них.

|  |
| --- |
| from collections import deque  from dataclasses import dataclass  from typing import List, Tuple, cast  @dataclass  class Road:  from\_city: int  to\_city: int  from\_gate: int  to\_gate: int  visited: bool = False  class CityGraph:  def \_\_init\_\_(self, n: int, roads\_data: List[Tuple[int, int]]) -> None:  self.n = n  self.roads: List[Road] = []  self.graph: List[List[int]] = [[] for \_ in range(n + 1)]  # Заполняем граф и список дорог  for gate1, gate2 in roads\_data:  from\_city = (gate1 + 3) // 4  to\_city = (gate2 + 3) // 4  road\_index = len(self.roads)  self.graph[from\_city].append(road\_index)  self.graph[to\_city].append(road\_index)  self.roads.append(Road(from\_city, to\_city, gate1, gate2))  def mark\_road\_visited(self, road\_index: int) -> None:  self.roads[road\_index].visited = True  def \_\_getitem\_\_(self, city: int) -> List[int]:  return self.graph[city]  def can\_complete\_task(n: int, graph: CityGraph) -> List[int]:  stack = deque() # Используем deque как стек  ans = [] # Список ворот  cur\_city = 1  while True:  if not graph[cur\_city]:  if not stack:  break  # Возвращаемся к предыдущему городу  to\_gate, from\_gate, prev\_city = (stack.pop() for \_ in range(3))  ans.extend((to\_gate, from\_gate))  cur\_city = prev\_city  continue  road\_index = graph[cur\_city][-1]  # Если дорога уже посещена  if graph.roads[road\_index].visited:  graph[cur\_city].pop()  continue  # Обновляем состояние дороги и перемещаемся в соседний город  if cur\_city == graph.roads[road\_index].from\_city:  stack.extend([cur\_city, graph.roads[road\_index].from\_gate, graph.roads[road\_index].to\_gate])  graph.mark\_road\_visited(road\_index)  cur\_city = graph.roads[road\_index].to\_city  else:  stack.extend([cur\_city, graph.roads[road\_index].to\_gate, graph.roads[road\_index].from\_gate])  graph.mark\_road\_visited(road\_index)  cur\_city = graph.roads[road\_index].from\_city  # Проверяем, удалось ли пройти через все ворота  return ans if len(ans) == 4 \* n else []  def main() -> None:  n: int = int(input())  roads\_data: List[Tuple[int, int]] = cast(  List[Tuple[int, int]],  [tuple(map(int, input().split())) for \_ in range(2 \* n)]  )  graph = CityGraph(n, roads\_data)  result = can\_complete\_task(n, graph)  if result:  print("Yes", " ".join(map(str, result)), sep="\n")  else:  print("No")  if \_\_name\_\_ == "\_\_main\_\_":  main() |



Задание 3

Зал Большого галактического театра состоит из S

рядов, по S

мест в каждом ряду.Продажа билетов на каждый спектакль происходит по следующему принципу: первые S2−N

ценителей прекрасного приобретают билеты на любые места по их вкусу, а оставшиеся N

кресел администрация бесплатно выделяет студентам, отдавая дань сложившимся традициям.

Во избежание обвинений в дискриминации по половому признаку, рассаживать студентов по этим N

местам необходимо таким образом, чтобы:

в каждом ряду количество девушек-студенток и количество юношей-студентов различалось бы не более чем на 1;

на каждой "вертикали мест" (т. е. местах, имеющих один и тот же номер, но расположенных в разных рядах) количество девушек-студенток и количество юношей-студентов также различалось бы не более чем на 1.

Таким образом, после продажи билетов ценителям прекрасного билетёры должны распределить оставшиеся N

кресел на женские и мужские с соблюдением этих правил.

Каждое место в зале определяется двумя числами от 1 до S

- номером ряда и номером самого места в этом ряду. Студенческое кресло номер i

расположено в ai

-м ряду и имеет в нём номер bi

. Поскольку ценители прекрасного могли занять совершенно любые места, числа ai

и bi

могут принимать любые значения от 1 до S

. В частности, может оказаться так, что в каком-нибудь ряду не будет ни одного студенческого места.

Ради упрощения работы билетёров администрация обращается к вам с заданием написать программу, которая автоматизирует процесс распределения студенческих мест на мужские и женские.

Входные данные

Сначала вводятся два целых числа S

и N

(1≤S≤100000

, 1≤N≤min{100000,S2}

). Далее расположены N

пар натуральных чисел (ai,bi)

, не превосходящих S

. Гарантируется, что все места различные.

Выходные данные

Если искомого способа не существует, выведите Impossible.Иначе выведите единственную строку из N

символов ‘M’ (мужское) и ‘W’ (женское). Символ на i

-й позиции соответствует статусу i

-го места в той же нумерации, в которой они были перечислены во входных данных.

Примечания

Тесты состоят из четырёх групп.

Тесты 1 и 2. Тесты из условия, оцениваются в 0 баллов.

Тесты 3--19. В них S≤1000

, N≤30

. Группа оценивается в 30 баллов, баллы начисляются только при прохождении всех тестов группы.

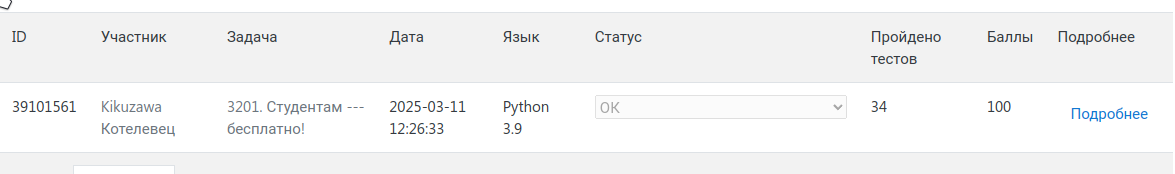
Тесты 20--30. В них S≤1000

, N≤1000

. Группа оценивается в 30 баллов, баллы начисляются только при прохождении всех тестов группы.

Тесты 31--34. Off-line группа, полные ограничения. Каждый тест оценивается в 10 баллов (тесты оцениваются независимо друг от друга). При этом баллы за тесты этой группы ставятся только тогда, когда программа проходит все тесты групп 1 и 2. Если программа не проходит хотя бы один из тестов групп 1 и 2, то баллы за тесты группы 3 не ставятся.

|  |
| --- |
| def solve(S, N, positions):  # Создаем матрицу театра  theater = [[None] \* S for \_ in range(S)]  # Заполняем позиции студентов  student\_positions = []  for i, (ai, bi) in enumerate(positions):  theater[ai - 1][bi - 1] = i  # Проверяем возможность решения  def check\_solution():  # Проверяем строки  for row in theater:  males = females = 0  for cell in row:  if cell is not None:  if solution[cell] == 'M':  males += 1  else:  females += 1  if abs(males - females) > 1:  return False  # Проверяем столбцы  for col in range(S):  males = females = 0  for row in range(S):  if theater[row][col] is not None:  if solution[theater[row][col]] == 'M':  males += 1  else:  females += 1  if abs(males - females) > 1:  return False  return True  # Ищем решение методом перебора  def dfs(pos=0):  if pos >= len(positions):  return check\_solution()  ai, bi = positions[pos]  ai -= 1  bi -= 1  for gender in ['M', 'W']:  solution[pos] = gender  if dfs(pos + 1):  return True  solution[pos] = None  return False  global solution  solution = [None] \* N  if dfs():  return ''.join(solution)  return "Impossible"  # Пример использования  S, N = map(int, input().split())  positions = []  for \_ in range(N):  ai, bi = map(int, input().split())  positions.append((ai, bi))  result = solve(S, N, positions)  print(result) |



Задание 4

Даны несколько точек на плоскости, некоторые из которых соединены отрезками. Множество точек называется связанным, если из любой его точки можно перейти в любую точку, перемещаясь только по отрезкам (переходить с отрезка на отрезок возможно только в точках исходного множества). Можно за определенную плату добавлять новые отрезки (стоимость добавления равна длине добавляемого отрезка). Требуется за минимальную стоимость сделать данное множество связанным.

Входные данные

В первой строке входных данных содержится одно целое число N

(1 ≤ N

≤ 50) – количество точек. Далее в N

строках записано по 2 натуральных числа – координаты точек (координаты не превышают 100). Все точки различны. Далее дано число M

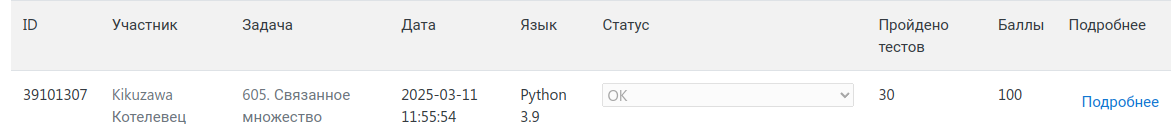
– количество уже существующих отрезков. В следующих M

строках записаны по 2 числа – номера начала и конца соответствующего отрезка.

Выходные данные

Вывести единственное число – минимально возможную стоимость дополнения с точностью 5 знаков после запятой.

|  |
| --- |
| import math  from itertools import combinations  from typing import List, Tuple, cast  def calculate\_distance(point1: Tuple[int, int], point2: Tuple[int, int]) -> float:  """Вычисляет евклидово расстояние между двумя точками."""  return math.sqrt((point1[0] - point2[0]) \*\* 2 + (point1[1] - point2[1]) \*\* 2)  def prim\_algorithm(points: List[Tuple[int, int]], connections: List[Tuple[int, int]]) -> float:  """Находит минимальную стоимость соединения всех точек с помощью алгоритма Прима."""  num\_points = len(points)  adj\_matrix = [[float('inf')] \* num\_points for \_ in range(num\_points)]  # Заполнение матрицы расстояний  for i, j in combinations(range(num\_points), 2):  adj\_matrix[i][j] = adj\_matrix[j][i] = calculate\_distance(points[i], points[j])  # Учитываем существующие соединения  for start, end in connections:  adj\_matrix[start - 1][end - 1] = adj\_matrix[end - 1][start - 1] = 0.0  # Алгоритм Прима  in\_mst = [False] \* num\_points  key\_values = [float('inf')] \* num\_points  key\_values[0] = 0.0  total\_cost = 0.0  for \_ in range(num\_points):  # Находим минимальный ключ  u = min((val, idx) for idx, val in enumerate(key\_values) if not in\_mst[idx])[1]  in\_mst[u] = True  total\_cost += key\_values[u]  # Обновляем ключи соседей  for v in range(num\_points):  if adj\_matrix[u][v] < key\_values[v] and not in\_mst[v]:  key\_values[v] = adj\_matrix[u][v]  return total\_cost  def main() -> None:  num\_points = int(input())  points = cast(List[Tuple[int, int]], [tuple(map(int, input().split())) for \_ in range(num\_points)])  num\_connections = int(input())  connections = cast(List[Tuple[int, int]], [tuple(map(int, input().split())) for \_ in range(num\_connections)])  min\_cost = prim\_algorithm(points, connections)  print(f"{min\_cost:.5f}")  if \_\_name\_\_ == "\_\_main\_\_":  main() |



Задание 5

Вам задан ориентированный граф G

. Каждое ребро имеет некоторую пропускную способность. Найдите максимальный поток между вершинами 1

и n

.

Входные данные

Первая строка входного файла содержит n

и m

— число вершин и рёбер в графе (2≤n≤500

, 1≤m≤10000

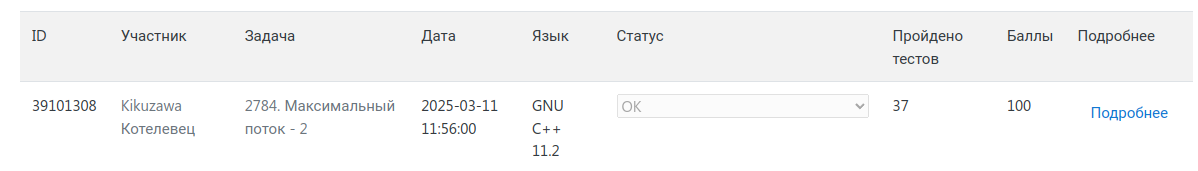
). Последующие строки описывают рёбра. Каждое ребро задается тремя числами: начальная вершина ребра, конечная вершина ребра и пропускная способность ребра. Пропускные способности — целые числа, не превосходящие 109

.

Выходные данные

Выведите величину максимального потока между вершинами 1и n.

|  |
| --- |
| #include <iostream>  #include <vector>  #include <queue>  #include <climits>  size\_t n, m;  const long long INF = LONG\_LONG\_MAX;  struct Edge {  int from, to;  long long flow, cap;  int reversed = -1;  Edge(int from, int to, long long f, long long cap) : from(from), to(to), flow(f), cap(cap) {}  };  std::vector<std::vector<Edge>> edges;  std::vector<long long> d;  std::vector<int> id;  std::vector<Edge\*> p;  void addEdge(int from, int to, long long flow, long long cap) {  Edge edge = Edge(from, to, flow, cap);  Edge rev = Edge(to, from, flow, 0);  edges[from].push\_back(edge);  edges[to].push\_back(rev);  edges[from].back().reversed = edges[to].size() - 1;  edges[to].back().reversed = edges[from].size() - 1;  }  long long dinicAlgorithm() {  long long maxFlow = 0;  while (true) {  // Очереди для BFS  std::queue<int> q1, q2;  id.assign(n + 1, 0);  d.assign(n + 1, INF);  p.resize(n + 1);  // Инициализация BFS  d[1] = 0;  q1.push(1);  id[1] = 1;  while (!q1.empty() || !q2.empty()) {  // Обрабатываем первую очередь  if (!q1.empty()) {  int u = q1.front();  q1.pop();  id[u] = 2;  for (Edge& edge : edges[u]) {  if (edge.flow < edge.cap && d[edge.to] > d[edge.from] + 1) {  d[edge.to] = d[edge.from] + 1;  if (id[edge.to] == 0) {  q2.push(edge.to); // Переводим в очередь q2 для дальнейшей обработки  id[edge.to] = 1;  p[edge.to] = &edge;  }  }  }  }  // Обрабатываем вторую очередь  if (!q2.empty()) {  int u = q2.front();  q2.pop();  id[u] = 2;  for (Edge& edge : edges[u]) {  if (edge.flow < edge.cap && d[edge.to] > d[edge.from] + 1) {  d[edge.to] = d[edge.from] + 1;  if (id[edge.to] == 0) {  q1.push(edge.to); // Переводим обратно в очередь q1  id[edge.to] = 1;  p[edge.to] = &edge;  }  }  }  }  }  long long del = INF;  if (d[n] == INF) {  break;  } else {  for (int u = n; u != 1; u = p[u]->from) {  Edge\* edge = p[u];  del = std::min(del, edge->cap - edge->flow);  }  // Обновление потоков  for (int u = n; u != 1; u = p[u]->from) {  Edge\* edge = p[u];  Edge\* reversed = &edges[edge->to][edge->reversed];  edge->flow += del;  reversed->flow -= del;  }  maxFlow += del;  }  }  return maxFlow;  }  int main() {  std::cin >> n >> m;  edges.assign(n + 1, std::vector<Edge>());  // Чтение рёбер  for (int i = 0; i < m; ++i) {  int from, to;  long long cap;  std::cin >> from >> to >> cap;  addEdge(from, to, 0, cap);  }  // Вывод максимального потока  std::cout << dinicAlgorithm() << std::endl;  return 0;  } |



Задание 6

*K* участникам сборов для решения было предложено *K* задач. Участники решили разделить задачи между собой, решить каждому по одной задаче, а затем обменяться решениями (они не учли, что система ejudge способна отследить данный факт J). Известно ориентировочное время, за которое каждый из участников сборов может решить каждую из предложенных задач.

Помогите участникам сборов распределить задачи так (по одной каждому участнику), чтобы суммарное время, потраченное на их решение было минимальным.

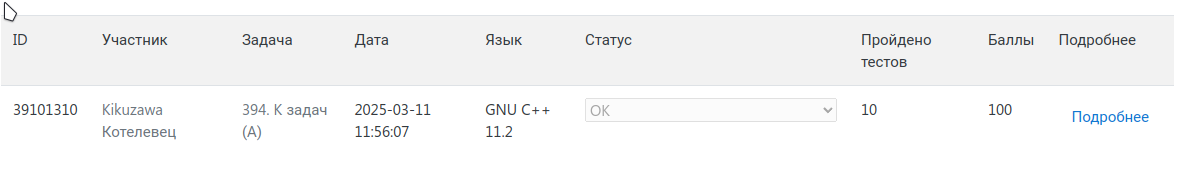
**Входные данные**

Во входном файле сначала записано число *K* (0 < *K* < 101) и далее *K*2 неотрицательных целых чисел, не превосходящие 20000, описывающих матрицу *K*x *K*, времен решения каждым из участников каждой из задач.

**Выходные данные**

В файл выведите суммарное минимальное время решения всех задач, при условии, что каждый участник решит ровно одну задачу.

|  |
| --- |
| /\* Венгерский алгоритм.  \* Реализация навеяна псевдокодом А.С.Лопатина из книги  \* "Оптимизация на графах (алгоритмы и реализация)".  \*/  #include <vector>  #include <limits>  #include <iostream>  using namespace std;  typedef pair<int, int> PInt;  typedef vector<int> VInt;  typedef vector<VInt> VVInt;  typedef vector<PInt> VPInt;  const int inf = numeric\_limits<int>::max();  /\*  \* Решает задачу о назначениях Венгерским методом.  \* matrix: прямоугольная матрица из целых чисел (не обязательно положительных).  \* Высота матрицы должна быть не больше ширины.  \* Возвращает: Список выбранных элементов, по одному из каждой строки матрицы.  \*/  VPInt hungarian(const VVInt &matrix) {  // Размеры матрицы  int height = matrix.size(), width = matrix[0].size();  // Значения, вычитаемые из строк (u) и столбцов (v)  VInt u(height, 0), v(width, 0);  // Индекс помеченной клетки в каждом столбце  VInt markIndices(width, -1);  // Будем добавлять строки матрицы одну за другой  for(int i = 0; i < height; i++) {  VInt links(width, -1);  VInt mins(width, inf);  VInt visited(width, 0);  // Разрешение коллизий (создание "чередующейся цепочки" из нулевых элементов)  int markedI = i, markedJ = -1, j;  while(markedI != -1) {  // Обновим информацию о минимумах в посещенных строках непосещенных столбцов  // Заодно поместим в j индекс непосещенного столбца с самым маленьким из них  j = -1;  for(int j1 = 0; j1 < width; j1++)  if(!visited[j1]) {  if(matrix[markedI][j1] - u[markedI] - v[j1] < mins[j1]) {  mins[j1] = matrix[markedI][j1] - u[markedI] - v[j1];  links[j1] = markedJ;  }  if(j==-1 || mins[j1] < mins[j])  j = j1;  }  // Теперь нас интересует элемент с индексами (markIndices[links[j]], j)  // Произведем манипуляции со строками и столбцами так, чтобы он обнулился  int delta = mins[j];  for(int j1 = 0; j1 < width; j1++)  if(visited[j1]) {  u[markIndices[j1]] += delta;  v[j1] -= delta;  } else {  mins[j1] -= delta;  }  u[i] += delta;  // Если коллизия не разрешена - перейдем к следующей итерации  visited[j] = 1;  markedJ = j;  markedI = markIndices[j];  }  // Пройдем по найденной чередующейся цепочке клеток, снимем отметки с  // отмеченных клеток и поставим отметки на неотмеченные  for(; links[j] != -1; j = links[j])  markIndices[j] = markIndices[links[j]];  markIndices[j] = i;  }  // Вернем результат в естественной форме  VPInt result;  for(int j = 0; j < width; j++)  if(markIndices[j] != -1)  result.push\_back(PInt(markIndices[j], j));  return result;  }  int main() {  int k;  VVInt a;  int s=0;  cin >> k;  for(int i=0; i<k; ++i) {  a.push\_back(vector<int>(k));  }  for(int i=0; i<k; ++i) {  for(int j=0; j<k; ++j) {  cin >> a[i][j];  }  }  VPInt r=hungarian(a);  for(int i=0; i<k; ++i) {  s+=a[r[i].first][r[i].second];  }  cout << s;  return 0;  } |



Задание 7

На плоскости заданы координаты *n* (4 ≤ *n* ≤ 15) разных вершин.

Найти кратчайший замкнутый маршрут, начинающийся и заканчивающийся в 1-й вершине и посещающий все остальные вершины по одному разу. Разрешается (если так оказывается выгодно) «проезжать через вершину, не останавливаясь» (см. пример 1).

Длина маршрута считается как сумма длин составляющих его рёбер, длины отдельных рёбер считаются согласно обычной евклидовой метрике, как https://informatics.msk.ru/moodle_probpics/3785/3785.png.

**Входные данные**

Первая строка содержит количество вершин *n* (4 ≤ *n* ≤ 15). Каждая из следующих *n* строк содержит по два разделённых пробелом числа с плавающей точкой — *x*- и *y*-координаты соответствующей вершины.

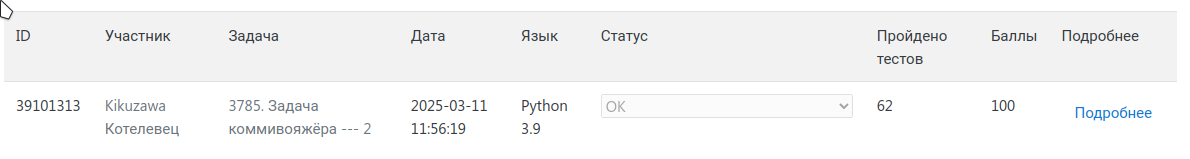
**Выходные данные**

Первая строка должна содержать единственное число (с плавающей точкой) — найденную минимальную длину замкнутого тура. Вторая строка должна содержать перестановку чисел 2, 3, ..., *N* — порядок, в котором надо посещать эти вершины. Числа внутри второй строки должны разделяться одинарными пробелами.

**Примечание**

Задача с такими ограничениями, по идее, должна решаться хоть методом ветвей и границ, хоть динамическим программированием по подмножествам. Но она, по идее, не должна решаться одними лишь отсечениями поиска с возвратом (backtracking), не пытающегося оценивать возможный диапазон длины ещё не построенной части пути.

|  |
| --- |
| import math  from itertools import combinations  from typing import List, Tuple, Iterable, Sequence, cast, Dict  from collections import deque  def reconstruct\_path(memoization\_table, bits: int, parent: int, n: int) -> Iterable[int]:  path: deque[int] = deque()  for \_ in range(n - 1):  path.appendleft(parent)  new\_bits: int = bits & ~(1 << parent)  \_, parent = memoization\_table[(bits, parent)]  bits: int = new\_bits  return path  def tsp\_dynamic\_programming(points: Sequence[Tuple[float, float]]) -> Tuple[float, Iterable[int]]:  n: int = len(points)  dist\_matrix: List[List[float]] = [[math.dist(points[i], points[j]) for j in range(n)] for i in range(n)]  # Таблица мемоизации, где ключами являются пары (набор посещенных вершин, текущая вершина)  # Устанавливаем начальные значения, когда первая вершина уже посещена  memoization\_table: Dict[Tuple[int, int], Tuple[float, int]] = {(1 << k, k): (dist\_matrix[0][k], 0) for k in  range(1, n)}  # Проходимся по вершинам  for subset\_size in range(2, n):  for subset in combinations(range(1, n), subset\_size):  # Устанавливаем биты для всех вершин в подмножестве  bits: int = sum(1 << bit for bit in subset)  # Находим кратчайший путь к этому подмножеству, заканчивающийся в вершине k  for k in subset:  prev: int = bits & ~(1 << k)  memoization\_table[(bits, k)] = min(  (memoization\_table[(prev, m)][0] + dist\_matrix[m][k], m)  for m in subset if m != 0 and m != k  )  # Мы возвращаемся к первой вершине, завершаем тур  bits: int = (2 \*\* n - 1) - 1  opt, parent = min((memoization\_table[(bits, k)][0] + dist\_matrix[k][0], k) for k in range(1, n))  path = reconstruct\_path(memoization\_table, bits, parent, n)  return opt, path  def main() -> None:  n: int = int(input())  points: List[Tuple[int, int]] = cast(List[Tuple[int, int]], [tuple(map(float, input().split())) for \_ in range(n)])  min\_length, min\_path = tsp\_dynamic\_programming(points)  print('{:.15E}'.format(min\_length))  print(' '.join(map(lambda x: str(x + 1), min\_path)))  if \_\_name\_\_ == '\_\_main\_\_':  main() |



Задание 8

Дана шахматная доска n×nn×n. Пусть конь стоит на клетке (1,1). Необходимо найти такую последовательность ходов коня, при которой он побывает на каждой клетке доски ровно по одному разу.

**Входные данные**

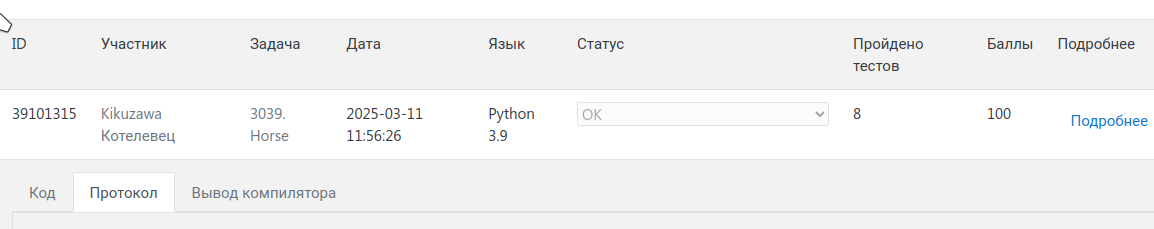
На вход программе подается натуральное число nn (n≤8n≤8).

**Выходные данные**

Если обход невозможен, то выведите в выходной файл 0, если возможен, то 1, а на следующих строчках выведите матрицу n×nn×n, иллюстрирующую порядок обхода. Выравнивать числа по столбцам не обязательно.

*Примечание.* Скорость работы рекурсивной программы в этой задаче существенно зависит от порядка, в каком будут рассматриваться варианты хода коня из очередной клетки. Одним из удачных порядков является размещение всех восьми вариантов хода "по кругу".

|  |
| --- |
| def is\_valid\_move(board, x, y):  return 0 <= x < len(board) and 0 <= y < len(board) and board[x][y] == -1  def get\_possible\_moves(x, y):  moves = [(2, 1), (1, 2), (-1, 2), (-2, 1), (-2, -1), (-1, -2), (1, -2), (2, -1)]  valid\_moves = []  for dx, dy in moves:  new\_x, new\_y = x + dx, y + dy  if is\_valid\_move(board, new\_x, new\_y):  valid\_moves.append((new\_x, new\_y))  return valid\_moves  def solve\_knight\_tour(n):  # Проверка невозможных случаев  if n <= 1 or n == 3 or n == 4:  return None  # Инициализация доски  board = [[-1] \* n for \_ in range(n)]  board[0][0] = 0 # Начальная позиция  def solve\_util(x, y, move\_num):  if move\_num == n \* n:  return True  # Получаем возможные ходы в определенном порядке  moves = get\_possible\_moves(x, y)  for next\_x, next\_y in moves:  board[next\_x][next\_y] = move\_num  if solve\_util(next\_x, next\_y, move\_num + 1):  return True  board[next\_x][next\_y] = -1  return False  if solve\_util(0, 0, 1):  return board  return None  def main():  n = int(input())  result = solve\_knight\_tour(n)  if result is None:  print(0)  else:  print(1)  for row in result:  print(' '.join(str(x + 1) for x in row))  if \_\_name\_\_ == "\_\_main\_\_":  main() |



Задание 9

Требуется найти в связном графе остовное дерево минимально веса.

**Входные данные**

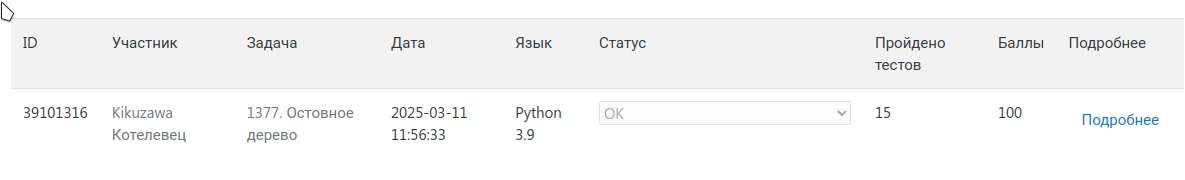
Первая строка входного файла содержит два натуральных числа n и m - количество вершин и ребер графа соответственно (1≤n≤20000, 0≤m≤100000). Следующие m строк содержат описание ребер по одному на строке. Ребро номер i описывается тремя натуральными числами bi, ei и wi - номера концов ребра и его вес соответственно (1≤bi,ei≤n, 0≤wi≤100000).

Граф является связным.

**Выходные данные**

Выведите единственное целое число - вес минимального остовного дерева.

|  |
| --- |
| from array import array  from collections import namedtuple  from typing import List, Sequence, TypeVar, Generic  Edge = namedtuple('Edge', ['weight', 'vertex1', 'vertex2'])  T = TypeVar('T')  class DisjointSetUnion(Generic[T]):  def \_\_init\_\_(self, size: int):  self.parent = array("i", range(size))  self.rank = array("i", [0] \* size)  def find(self, node: T) -> int:  if node != self.parent[node]:Требуется найти в связном графе остовное дерево минимально веса.  Входные данные  Первая строка входного файла содержит два натуральных числа n и m - количество вершин и ребер графа соответственно (1≤n≤20000, 0≤m≤100000). Следующие m строк содержат описание ребер по одному на строке. Ребро номер i описывается тремя натуральными числами bi, ei и wi - номера концов ребра и его вес соответственно (1≤bi,ei≤n, 0≤wi≤100000).  Граф является связным.  Выходные данные  Выведите единственное целое число - вес минимального остовного дерева.  # Сжатие пути (оптимизация поиска)  self.parent[node] = self.find(self.parent[node])  return self.parent[node]  def union(self, node1: T, node2: T) -> None:  root1 = self.find(node1)  root2 = self.find(node2)  if root1 != root2:  # Union by rank  if self.rank[root1] < self.rank[root2]:  root1, root2 = root2, root1  self.parent[root2] = root1  if self.rank[root1] == self.rank[root2]:  self.rank[root1] += 1  def kruskal(num\_vertices: int, edges: Sequence[Edge]) -> int:  dsu: DisjointSetUnion[int] = DisjointSetUnion(num\_vertices)  minimum\_spanning\_tree\_weight: int = 0  # Сортируем рёбра по весу в порядке возрастания  for edge in sorted(edges, key=lambda e: e.weight):  # Если вершины ещё не соединены, то объединяем их и добавляем вес ребра в итоговый результат  if dsu.find(edge.vertex1) != dsu.find(edge.vertex2):  dsu.union(edge.vertex1, edge.vertex2)  minimum\_spanning\_tree\_weight += edge.weight  return minimum\_spanning\_tree\_weight  def main() -> None:  num\_vertices, num\_edges = map(int, input().split())  edges: List[Edge] = []  for \_ in range(num\_edges):  vertex1, vertex2, weight = map(int, input().split())  edges.append(Edge(weight, vertex1 - 1, vertex2 - 1))  print(kruskal(num\_vertices, edges))  if \_\_name\_\_ == "\_\_main\_\_":  main() |



Лабораторная работа 6

Задание 1

В новых элитных электричках каждому пассажиру положено сидячее место. Естественно, количество сидячих мест ограничено и на всех их может не хватить. Маршрут электрички проходит через N+1 станция, занумерованные от 0 до N. Когда человек хочет купить билет, он называет два числа x и y – номера станций, откуда и куда он хочет ехать. При наличии хотя бы одного сидячего места на этом участке на момент покупки ему продается билет, иначе выдается сообщение «билетов нет» и билет не продается. Ваша задача – написать программу, обслуживающую такого рода запросы в порядке их прихода.

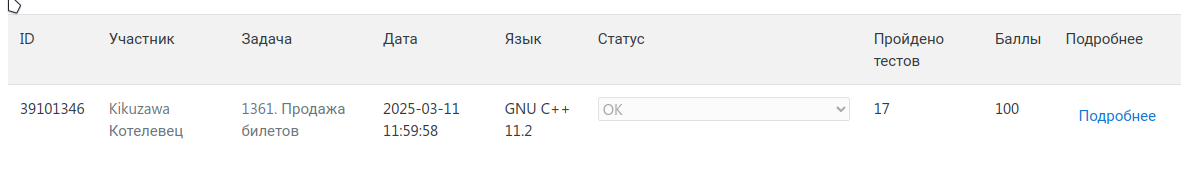
**Входные данные**

В первой строке содержаться три числа N – количество станций (1 ≤ N ≤ 200 000), K – количество мест в электричке (1 ≤ K ≤ 1000) и M – количество запросов (1 ≤ M ≤ 100 000). В следующих M строках описаны запросы, каждый из которых состоит из двух чисел x и y (0 ≤ x < y <= N).

**Выходные данные**

На каждый запрос ваша программа должна выдавать результат в виде числа 0 если билет не продается и 1 если билет был продан. Каждый результат должен быть на отдельной строке

|  |
| --- |
| #include <iostream>  #include <vector>  using namespace std;  bool isAvailable(const vector<int>& segments, int x, int y, int k) {  // Проверяет, есть ли хотя бы одно свободное место на участке [x, y).  for (int i = x; i < y; ++i) {  if (segments[i] >= k) {  return false;  }  }  return true;  }  void processRequests(int n, int k, const vector<pair<int, int>>& requests) {  vector<int> segments(n + 1, 0); // Инициализируем массив мест  for (const auto& req : requests) {  int x = req.first;  int y = req.second;  if (isAvailable(segments, x, y, k)) {  // Если есть свободное место, продаем билет и увеличиваем занятые места  for (int i = x; i < y; ++i) {  segments[i]++;  }  cout << 1 << endl;  } else {  cout << 0 << endl;  }  }  }  int main() {  int n, k, m;  cin >> n >> k >> m; // Ввод данных  vector<pair<int, int>> requests(m);  for (int i = 0; i < m; ++i) {  cin >> requests[i].first >> requests[i].second; // Ввод запросов  }  processRequests(n, k, requests); // Обработка запросов  return 0;  } |



Задание 2

Реализуйте структуру данных для эффективного вычисления номера максимального из нескольких подряд идущих элементов массива.

**Входные данные**

В первой строке вводится одно натуральное число *N* (1 ≤ *N* ≤ 100000) — количество чисел в массиве.

Во второй строке вводятся N чисел от 1 до 100000 — элементы массива.

В третьей строке вводится одно натуральное число *K* (1 ≤ *K* ≤ 30000) — количество запросов на вычисление максимума.

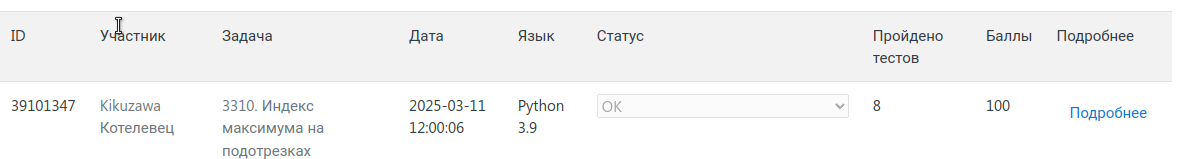
В следующих *K* строках вводится по два числа — номера левого и правого элементов отрезка массива (считается, что элементы массива нумеруются с единицы).

**Выходные данные**

Для каждого запроса выведите индекс максимального элемента на указанном отрезке массива. Если максимальных элементов несколько, выведите любой их них.

Числа выводите в одну строку через пробел.

|  |
| --- |
| from dataclasses import dataclass  from typing import List, Tuple  @dataclass  class ArrayElement:  value: int  index: int  def build\_sparse\_table(array: List[int]) -> Tuple[List[List[ArrayElement]], List[int]]:  n: int = len(array)  log\_values: List[int] = [0] \* (n + 1)  # Предварительное вычисление логарифмов  for i in range(2, n + 1):  log\_values[i] = log\_values[i // 2] + 1  max\_power\_of\_two: int = log\_values[n] + 1  sparse\_table: List[List[ArrayElement]] = [[ArrayElement(0, 0)] \* max\_power\_of\_two for \_ in range(n)]  # Инициализация разреженной таблицы  for i in range(n):  sparse\_table[i][0] = ArrayElement(array[i], i)  # Строим таблицу для отрезков длиной 2^j  for j in range(1, max\_power\_of\_two):  for i in range(n - (1 << j) + 1): # (1 << j) = 2^j  if sparse\_table[i][j - 1].value >= sparse\_table[i + (1 << (j - 1))][j - 1].value:  sparse\_table[i][j] = sparse\_table[i][j - 1]  else:  sparse\_table[i][j] = sparse\_table[i + (1 << (j - 1))][j - 1]  return sparse\_table, log\_values  def query\_max(sparse\_table: List[List[ArrayElement]], log\_values: List[int], left: int, right: int) -> int:  # Определяем наибольшую степень двойки, которая укладывается в отрезок  max\_power\_of\_two\_in\_range: int = log\_values[right - left + 1]  # Определяем максимальные элементы для двух подотрезков  left\_max: ArrayElement = sparse\_table[left][max\_power\_of\_two\_in\_range]  right\_max: ArrayElement = sparse\_table[right - (1 << max\_power\_of\_two\_in\_range) + 1][max\_power\_of\_two\_in\_range]  # Возвращаем индекс максимального элемента  max\_element: ArrayElement = left\_max if left\_max.value >= right\_max.value else right\_max  return max\_element.index + 1  def main() -> None:  \_: int = int(input())  array: List[int] = list(map(int, input().split()))  query\_count: int = int(input())  sparse\_table, log\_values = build\_sparse\_table(array)  results: List[int] = []  for \_ in range(query\_count):  left, right = map(lambda x: int(x) - 1, input().split())  max\_index = query\_max(sparse\_table, log\_values, left, right)  results.append(max\_index)  print(' '.join(map(str, results)))  if \_\_name\_\_ == "\_\_main\_\_":  main() |



Задание 3

Ценные бумаги на фондовом рынке характеризуются множеством параметров. У них есть цена и ликвидность, также оценивать динамичность изменения цены, среднюю прибыльность, потенциал роста прибыльности и др. показатели. Аналитики трейдовой компании "WebMarket" ввели специальный показатель надежности ценной бумаги и научились эффективно его оценивать. Большое значение этого показателя соответствует малому риску покупки ценной бумаги. Но с ростом надежности обычно падает среднее оцениваемое значение прибыльности.

Для своих клиентов, играющих на рынке ценных бумаг, компания "WebMarket" решила открыть значения этого показателя и более того, автоматизировать покупку ценных бумаг с заданным порядковым номером по значению надежности. Аналитики проанализировали идею, и решили, что наличие такого функционала будет способствовать привлечению новых клиентов на рынок "WebMarket", повышению объемов сделок, а значит, и повышению прибылей "WebMarket". Важно также отметить, что торговля на базе этого показателя может позитивно сказаться на российском фондовом рынке и cделать его более здоровым. Алгоритмы оценки надежности уже написаны, средства выделены, необходимая реклама проведена. Осталось только написать сам код.

Ценные бумаги в базе данных имеют три атрибута:

* code — непустая строка латинских символов и цифр длины 30 или меньше
* id — целочисленный идентификатор (начиная с 0)
* reliability — целое число из диапазона [−231,231)[−231,231).

Каждой новой ЦБ выдается следующий по порядку id и значение её надёжности устанавливается в 0. Если ценная бумага отзывается с рынка, ее id для новых бумаг не используется.

База данных получает запросы, которые позволяют вводить новые ЦБ на рынок, получать текущую информацию о ЦБ, отзывать ЦБ с рынка, менять значение надежности у ЦБ и находить ЦБ, которая стоит на n-м месте, если упорядочить ЦБ по убыванию надежности, а при одинаковых значениях по возрастанию идентификатора.

При добавлении ЦБ с кодом, который раньше встречался, но соответствующая ЦБ была отозвана с рынка, ей назначается новый идентификатор.

**Входные данные**

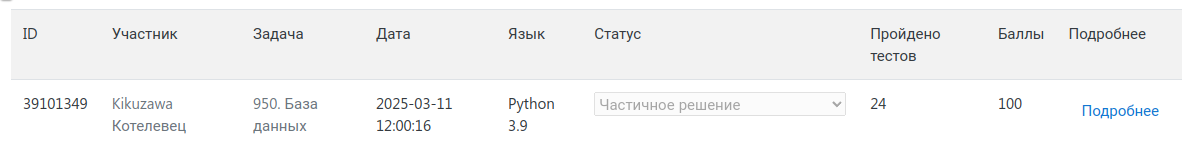
Первая строка входа содержит число запросов NN. (1≤N≤1000001≤N≤100000). Затем идут NN строк, каждая из которых содержит один запрос. Запросы бывают 4 типов:

* ISSUE code: добавить новую ЦБ code в базу данных, или вывести информацию о ЦБ, если она уже есть в базе
  + если ЦБ code существует, то вывести EXISTS id reliability
  + если ЦБ code не существует, то вывести CREATED id 0
* DELETE code: удалить ЦБ code из базы
  + если ЦБ code существует, то вывести OK id reliability
  + если ЦБ code не существует, то вывести BAD REQUEST
* RELIABILITY code reliability: увеличить количество очков ЦБ, гарантируется, что значение останется в диапазоне [−231,231)[−231,231).
  + если ЦБ code существует, то вывести OK id new\_reliability
  + если ЦБ code не существует, то вывести BAD REQUEST
* FIND n: найти n-ю ЦБ (начиная с 0)
  + если база не пуста, найти ЦБ, которая имела бы порядковый номер n (нумерация начинается с 0), если все ЦБ упорядочить сначала по убыванию баллов, а группы ЦБ с одинаковыми баллами упорядочить по времени их добавления (то есть по возрастанию id); а если n больше, чем количество ЦБ в базе, то найти последнюю ЦБ в этом порядке; для обоих случаев вывести OK code id reliability
  + если база пуста, то вывести EMPTY

**Выходные данные**

Таким образом, на каждый запрос на входе нужно вывести одну строку с результатом.

|  |
| --- |
| #include <bits/stdc++.h>  using namespace std;  #define int int64\_t  mt19937 rnd(int(3131));  struct Node {  int val;  int prior;  int l, r;  int sz;  };  vector<Node> a(0);  int new\_(int u) {  int gg=a.size();  a.push\_back({});  a[gg].val=u;  a[gg].sz=1;  a[gg].prior=rnd();  a[gg].l=-1, a[gg].r=-1;  return gg;  }  int size\_(int u) {  if(u==-1) {  return 0;  }  return a[u].sz;  }  void upd(int u) {  if(u==-1) {  return;  }  a[u].sz=size\_(a[u].l)+size\_(a[u].r)+1;  }  int root=-1;  int merge\_(int l, int r) {  upd(l);  upd(r);  if(l==-1) {  return r;  }  if(r==-1) {  return l;  }  if(a[l].prior>a[r].prior) {  a[l].r=merge\_(a[l].r, r);  upd(l);  return l;  }  else {  a[r].l=merge\_(l, a[r].l);  upd(r);  return r;  }  }  pair<int, int> split(int u, int k) {  if(u==-1) {  return {-1, -1};  }  upd(u);  if(size\_(a[u].l)+1<=k) {  pair<int, int> w=split(a[u].r, k-size\_(a[u].l)-1);  a[u].r=w.first;  upd(u);  return {u, w.second};  }  else {  pair<int, int> w=split(a[u].l, k);  a[u].l=w.second;  upd(u);  return {w.first, u};  }  }  void get(int u) {  if(u==-1) {  return;  }  get(a[u].l);  cout << a[u].val << ' ';  get(a[u].r);  }  void sol() {  int n,m;  cin >> n >> m;  for(int i=1;i<=n;i++) {  root=merge\_(root, new\_(i));  }  for(int q=1;q<=m;q++) {  int l,r;  cin >> l >> r;  pair<int, int> w=split(root, r);  pair<int, int> g=split(w.first, l-1);  root=merge\_(merge\_(g.second, g.first), w.second);  }  get(root);  }  int32\_t main() {  /\*ios\_base::sync\_with\_stdio(false);  cin.tie(nullptr);  cout.tie(nullptr);\*/  //freopen("movetofront.in", "r", stdin);  //freopen("movetofront.out", "w", stdout);  int t=1;  //cin >> t;  while (t--) {  sol();  }  } |



Задание 4

Реализуйте структуру данных, которая поддерживает множество SS целых чисел, с котором разрешается производить следующие операции:

* add(i)add(i) — добавить в множество SS число ii (если он там уже есть, то множество не меняется);
* next(i)next(i) — вывести минимальный элемент множества, не меньший ii. Если искомый элемент в структуре отсутствует, необходимо вывести -1.

**Входные данные**

Исходно множество SS пусто. Первая строка входного файла содержит nn — количество операций (1≤n≤3000001≤n≤300000). Следующие nn строк содержат операции. Каждая операция имеет вид либо «+ ii», либо «? ii». Операция «? ii» задает запрос next(i)next(i).

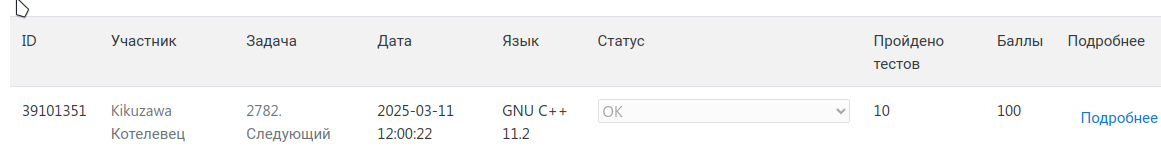
Если операция «+ ii» идет во входном файле в начале или после другой операции «+», то она задает операцию add(i)add(i). Если же она идет после запроса «?», и результат этого запроса был yy, то выполняется операция add((i+y)mod109)add((i+y)mod109).

Во всех запросах и операциях добавления параметры лежат в интервале от 00 до 109109.

**Выходные данные**

Для каждого запроса выведите одно число — ответ на запрос.

|  |
| --- |
| #include <iostream>  #include <set>  class SortedList {  private:  std::set<int> s;  int lastQuery;  public:  SortedList() : lastQuery(-1) {}  void add(int x) {  if (lastQuery != -1) {  x = (x + lastQuery) % 1000000000;  }  s.insert(x);  lastQuery = -1;  }  int next(int x) {  auto it = s.lower\_bound(x); // Находим первый элемент, не меньший x  if (it != s.end()) {  lastQuery = \*it;  return lastQuery;  } else {  lastQuery = -1;  return -1;  }  }  };  int main() {  int n;  std::cin >> n;  SortedList sortedList;  for (int i = 0; i < n; ++i) {  char operation;  int x;  std::cin >> operation >> x;  if (operation == '+') {  sortedList.add(x);  } else if (operation == '?') {  std::cout << sortedList.next(x) << std::endl;  }  }  return 0;  } |



Задание 5

Вам даны пары чисел (ai,bi)(ai,bi), Вам необходимо построить декартово дерево, такое что ii-ая вершина имеет ключи (ai,bi)(ai,bi), вершины с ключом aiai образуют бинарное дерево поиска, а вершины с ключом bibi образуют кучу на минимум.

**Входные данные**

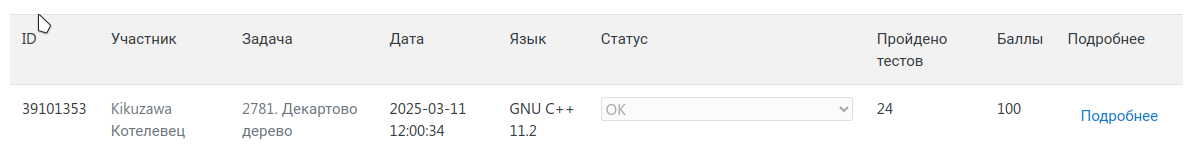
В первой строке записано число NN — количество пар. Далее следует NN (1≤N≤500001≤N≤50000) пар (ai,bi)(ai,bi). Для всех пар |ai|,|bi|≤30000|ai|,|bi|≤30000. ai≠ajai≠aj и bi≠bjbi≠bj для всех i≠ji≠j.

**Выходные данные**

Если декартово дерево с таким набором ключей построить возможно, выведите в первой строке YES, в противном случае выведите NO. В случае ответа YES, выведите NN строк, каждая из которых должна описывать вершину. Описание вершины состоит из трёх чисел: номер предка, номер левого сына и номер правого сына. Если у вершины отсутствует предок или какой-либо из сыновей, то выводите на его месте число 0.

Если подходящих деревьев несколько, выведите любое.

|  |
| --- |
| #include <iostream>  #include <fstream>  #include <time.h>  #include <stdlib.h>  using namespace std;  ifstream in("input.txt");  ofstream out("output.txt");  class vertex{  public:  int x, y, label;  struct vertex \*parent, \*left, \*right;  vertex() {}  void set(int xi, int yi, int labeli){  x = xi;  y = -yi;  label = labeli;  parent = NULL;  left = NULL;  right = NULL;  }  vertex& operator=(const vertex& a){  x = a.x;  y = a.y;  label = a.label;  parent = parent;  left = left;  right = right;  return \*this;  }  friend void swap (vertex& a, vertex& b);  };  void swap (vertex& a, vertex& b){  vertex shift;  shift = a;  a = b;  b = shift;  }  typedef vertex \*pvertex;  class decTree{  private:  int treeSize;  vertex \*vertexs;  pvertex \*ordered;  pvertex root;  int vertexCount;  void \_sortIns(pvertex \*pvertexs , int left, int right){  pvertex shift;  for (int i = left + 1; i < right; i++){  int j = i - 1;  shift = pvertexs[i];  for (int k = i-1; ((k >= left) && (pvertexs[j]->x > shift->x)) ; k--, j--){  pvertexs[k + 1] = pvertexs[k];  }  pvertexs[j + 1] = shift;  }  }  void \_sortQuick(pvertex \*pvertexs, int left, int right){  srand(time(NULL));  int l = left;  int r = right-1;  pvertex p;  p = pvertexs[l + rand() % (right - left)];  do{  while (pvertexs[l]->x < p->x) {  l++;  }  while (pvertexs[r]->x > p->x) {  r--;  }  if (l <= r){  swap(pvertexs[l++], pvertexs[r--]);  }  } while (l <= r);  if (r - left > 16){  \_sortQuick(pvertexs, left, r + 1);  } else if (r > left){  \_sortIns(pvertexs, left, r + 1);  }  if (right - l > 16){  \_sortQuick(pvertexs, l, right);  } else if (l < right - 1){  \_sortIns(pvertexs, l , right);  }  ////  }  void \_makeTree(){  pvertex last = NULL;  root = ordered[0];  last = root;  for (int i = 1; i < treeSize; i++){  while (last->parent != NULL && ordered[i]->y > last->y){  last = last->parent;  }  if (ordered[i]->y <= last->y){  ordered[i]->left = last->right;  ordered[i]->parent = last;  if (last->right){  last->right->parent = ordered[i];  }  last->right = ordered[i];  } else {  ordered[i]->left = last;  last->parent = ordered[i];  root = ordered[i];  }  last = ordered[i];  }  }  public:  void vertexAdd(int x, int y){  vertexs[vertexCount].set(x, y, vertexCount + 1);  ordered[vertexCount] = vertexs + vertexCount;  vertexCount++;  \_sortQuick(ordered, 0, treeSize);  \_makeTree();  }  decTree(int size):treeSize(size){  vertexCount = 0;  vertexs = new vertex[treeSize];  ordered = new pvertex[treeSize];  }  decTree(){  int a , b;  vertexCount = 0;  in >> treeSize;  vertexs = new vertex[treeSize];  ordered = new pvertex[treeSize];  for (int i = 0; i < treeSize; i++ ){  in >> a >> b;  vertexs[vertexCount].set(a, b, vertexCount + 1);  ordered[vertexCount] = vertexs + vertexCount;  vertexCount++;  }  \_sortQuick(ordered, 0, treeSize);  \_makeTree();  }  void resultsDisplay(){  int parent, left, right;  out << "YES" << endl;  for (int i = 0; i < treeSize; i++){  if (vertexs[i].parent) {  parent = vertexs[i].parent->label;  } else {  parent = 0;  }  if (vertexs[i].left){  left = vertexs[i].left->label;  } else {  left = 0;  }  if (vertexs[i].right) {  right = vertexs[i].right->label;  } else {  right = 0;  }  out << parent << " " << left << " " << right << endl;  }  }  };  int main(void){  decTree \*mytree = new decTree();  mytree->resultsDisplay();  return 0;  } |



Задание 6

Дан массив. Надо научиться обрабатывать два типа запросов.

\* 1 L R - перевернуть отрезок [L,R][L,R]

\* 2 L R - найти минимум на отрезке [L,R][L,R]

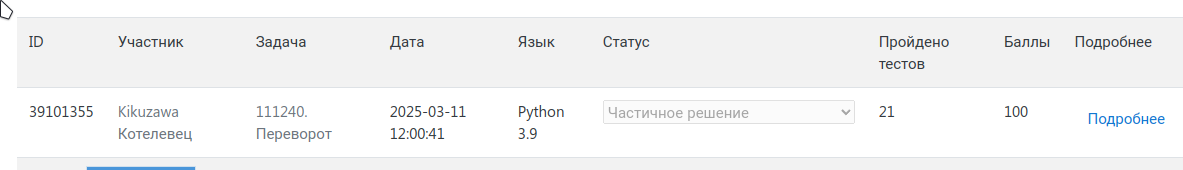
**Входные данные**

Первая строка файла содержит два числа nn, mm. (1≤n,m≤1051≤n,m≤105) Во второй строке находится nn чисел aiai (1≤ai≤1091≤ai≤109) — исходный массив. Остальные mm строк содержат запросы, в формате описанном в условии. Для чисел LL, RR выполняется ограничение (1≤L≤R≤n1≤L≤R≤n).

**Выходные данные**

На каждый запрос типа 2, во входной файл выведите ответ на него, в отдельной строке.

|  |
| --- |
| #include <iostream>  #include <vector>  using namespace std;  bool isAvailable(const vector<int>& segments, int x, int y, int k) {  // Проверяет, есть ли хотя бы одно свободное место на участке [x, y).  for (int i = x; i < y; ++i) {  if (segments[i] >= k) {  return false;  }  }  return true;  }  void processRequests(int n, int k, const vector<pair<int, int>>& requests) {  vector<int> segments(n + 1, 0); // Инициализируем массив мест  for (const auto& req : requests) {  int x = req.first;  int y = req.second;  if (isAvailable(segments, x, y, k)) {  // Если есть свободное место, продаем билет и увеличиваем занятые места  for (int i = x; i < y; ++i) {  segments[i]++;  }  cout << 1 << endl;  } else {  cout << 0 << endl;  }  }  }  int main() {  int n, k, m;  cin >> n >> k >> m; // Ввод данных  vector<pair<int, int>> requests(m);  for (int i = 0; i < m; ++i) {  cin >> requests[i].first >> requests[i].second; // Ввод запросов  }  processRequests(n, k, requests); // Обработка запросов  return 0;  } |



Задание 7

Физики проводят эксперимент для исследования частиц трёх типов: xx, yy и zz. Они запускают в коллайдер пронумерованный ряд из nn частиц. Во время эксперимента происходит воздействие на одну конкретную частицу, после чего частица исчезает с ii-ого места ряда и моментально появляется на месте jj. После её исчезновения номера частиц, стоящих правее, уменьшаются на 1, а после появления, номера частиц, стоящих правее, увеличиваются на 1. После определенного числа воздействий физики интересуются какая частица стоит на месте kk. Напишите программу, которая поможет физикам.

**Входные данные**

В первой строке файла два целых числа: nn – количество частиц и m — общее количество воздействий и вопросов (1 ≤≤ nn ≤≤ 1000000, 1 ≤≤ mm ≤≤ 15000). Во второй строке — последовательность из символов xx, yy и zz длиной nn. На каждой из следующих mm строк (1 ≤≤ mm ≤≤ 15000) описано воздействие или вопрос. Строка, в которой описано воздействие, начинается символом aa и после пробела дается два целых числа из интервала [1; nn]. Первое из них показывает начальное, а второе конечное местоположение частицы во время воздействия. Строка, в которой описан вопрос, начинается символом qq и после пробела дается одно целое число из интервала [1; nn]. Оно указывает позицию, которая интересует физиков.

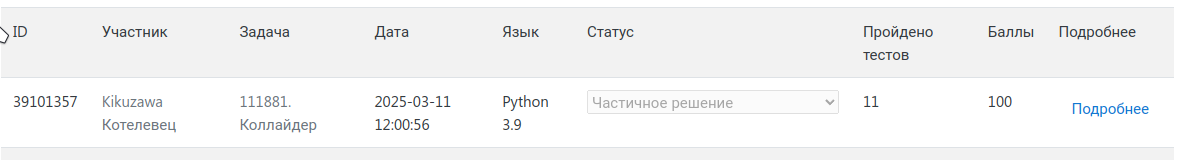
**Выходные данные**

Выведите столько строк, сколько вопросов во входном файле. В строке номер ii надо записать ответ на вопрос ii — название соответствующей частицы xx, yy или zz.

**Пояснения к примеру**

Последовательность после первого воздействия – xxyyzxxzxzyyzyx, последовательность после второго воздействия – xxyxyzxxzxzyyzy, последовательность после третьего воздействия – xyxyxyzxxzxzyzy,

|  |
| --- |
| #include <iostream>  #include <vector>  #include <string>  int main() {  int n, m;  std::cin >> n >> m;  std::string particles;  std::cin >> particles;  std::vector<int> indexes(n);  for (int i = 0; i < n; ++i) {  indexes[i] = i;  }  for (int i = 0; i < m; ++i) {  std::string action;  std::cin >> action;  if (action == "a") {  int x, y;  std::cin >> x >> y;  --x; // Приводим к 0-индексации  --y; // Приводим к 0-индексации  // Перемещаем элемент x в позицию y  if (x != y) {  int temp = indexes[x];  if (x < y) {  for (int j = x; j < y; ++j) {  indexes[j] = indexes[j + 1];  }  } else {  for (int j = x; j > y; --j) {  indexes[j] = indexes[j - 1];  }  }  indexes[y] = temp;  }  } else {  int k;  std::cin >> k;  --k; // Приводим к 0-индексации  std::cout << particles[indexes[k]] << std::endl;  }  }  return 0;  } |



Задание 8

Капрал Питуца любит командовать своим отрядом. Его любимый приказ «в начало строя». Он выстраивает свой отряд в шеренгу и оглашает последовательность приказов. Каждая приказ имеет вид «Солдаты с lili по riri — в начало строя!»

Пронумеруем солдат в начальном положении с 1 до nn, слева направо. Приказ «Солдаты с lili по riri — в начало строя!» означает, что солдаты, стоящие с lili по riri включительно, перемещаются в начало строя, сохраняя относительный порядок.

Например, если в некоторый момент солдаты стоят в порядке 2,3,6,1,5,42,3,6,1,5,4, после приказа: «Солдаты с 22 по 44 — в начало строя!» порядок будет 3,6,1,2,5,43,6,1,2,5,4.

По данной последовательности приказов найти конечный порядок солдат в строю.

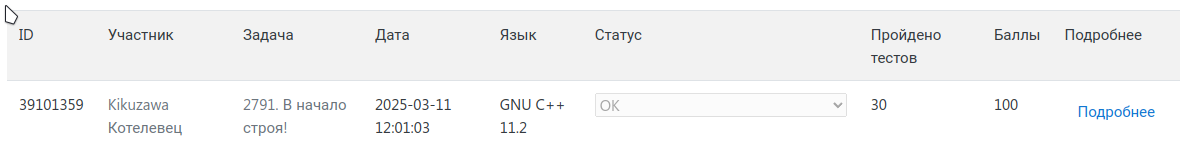
**Входные данные**

В первой строке два целых числа nn and mm (2≤n≤1000002≤n≤100000, 1≤m≤1000001≤m≤100000) — количество солдат и количество приказов. Следующие mm строк содержат по два целых числа lili и riri (1≤li≤ri≤n1≤li≤ri≤n).

**Выходные данные**

Выведите nn целых чисел — порядок солдат в конечном положении после выполнения всех приказов.

|  |
| --- |
| #include <bits/stdc++.h>  using namespace std;  #define int int64\_t  mt19937 rnd(int(3131));  struct Node {  int val;  int prior;  int l, r;  int sz;  };  vector<Node> a(0);  int new\_(int u) {  int gg=a.size();  a.push\_back({});  a[gg].val=u;  a[gg].sz=1;  a[gg].prior=rnd();  a[gg].l=-1, a[gg].r=-1;  return gg;  }  int size\_(int u) {  if(u==-1) {  return 0;  }  return a[u].sz;  }  void upd(int u) {  if(u==-1) {  return;  }  a[u].sz=size\_(a[u].l)+size\_(a[u].r)+1;  }  int root=-1;  int merge\_(int l, int r) {  upd(l);  upd(r);  if(l==-1) {  return r;  }  if(r==-1) {  return l;  }  if(a[l].prior>a[r].prior) {  a[l].r=merge\_(a[l].r, r);  upd(l);  return l;  }  else {  a[r].l=merge\_(l, a[r].l);  upd(r);  return r;  }  }  pair<int, int> split(int u, int k) {  if(u==-1) {  return {-1, -1};  }  upd(u);  if(size\_(a[u].l)+1<=k) {  pair<int, int> w=split(a[u].r, k-size\_(a[u].l)-1);  a[u].r=w.first;  upd(u);  return {u, w.second};  }  else {  pair<int, int> w=split(a[u].l, k);  a[u].l=w.second;  upd(u);  return {w.first, u};  }  }  void get(int u) {  if(u==-1) {  return;  }  get(a[u].l);  cout << a[u].val << ' ';  get(a[u].r);  }  void sol() {  int n,m;  cin >> n >> m;  for(int i=1;i<=n;i++) {  root=merge\_(root, new\_(i));  }  for(int q=1;q<=m;q++) {  int l,r;  cin >> l >> r;  pair<int, int> w=split(root, r);  pair<int, int> g=split(w.first, l-1);  root=merge\_(merge\_(g.second, g.first), w.second);  }  get(root);  }  int32\_t main() {  /\*ios\_base::sync\_with\_stdio(false);  cin.tie(nullptr);  cout.tie(nullptr);\*/  //freopen("movetofront.in", "r", stdin);  //freopen("movetofront.out", "w", stdout);  int t=1;  //cin >> t;  while (t--) {  sol();  }  } |



Лабораторная работа 7

Задание 1

Дана непустая строка *s*. Нужно найти такое наибольшее число *k* и строку *t*, что *s* совпадает со строкой *t*, выписанной *k* раз подряд.

Ограничение времени - 1 секунда.

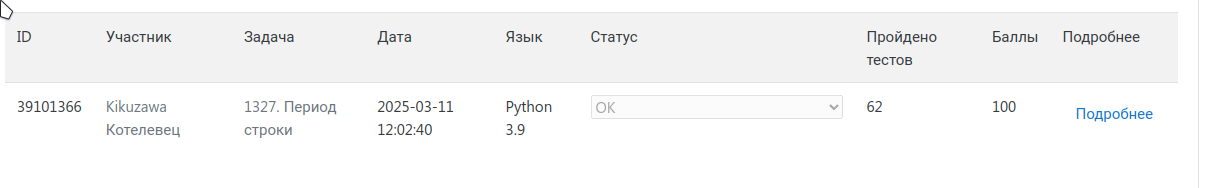
**Входные данные**

Одна строка длины *N*, 0 < N ≤ 106, состоящая только из маленьких латинских букв.

**Выходные данные**

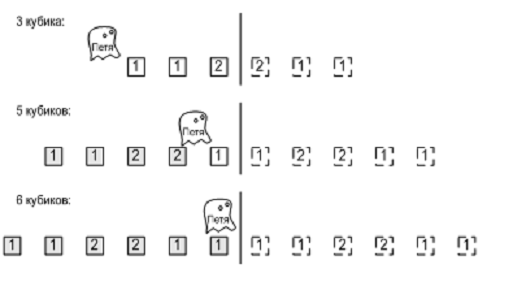
Одно число - наибольшее возможное *k*.

|  |
| --- |
| def find\_max\_repetition\_count(s: str) -> int:  length = len(s)  for period\_length in range(1, length // 2 + 1):  if length % period\_length == 0 and s[:period\_length] \* (length // period\_length) == s:  return length // period\_length  return 1  def main() -> None:  input\_string = input().strip()  print(find\_max\_repetition\_count(input\_string))  if \_\_name\_\_ == '\_\_main\_\_':  main() |



Задание 2

Привидение Петя любит играть со своими кубиками. Он любит выкладывать их в ряд и разглядывать свое творение. Однако недавно друзья решили подшутить над Петей и поставили в его игровой комнате зеркало. Ведь всем известно, что привидения не отражаются в зеркале! А кубики отражаются. Теперь Петя видит перед собой N цветных кубиков, но не знает, какие из этих кубиков настоящие, а какие — всего лишь отражение в зеркале. Помогите Пете! Выясните, сколько кубиков может быть у Пети. Петя видит отражение всех кубиков в зеркале и часть кубиков, которая находится перед ним. Часть кубиков может быть позади Пети, их он не видит.



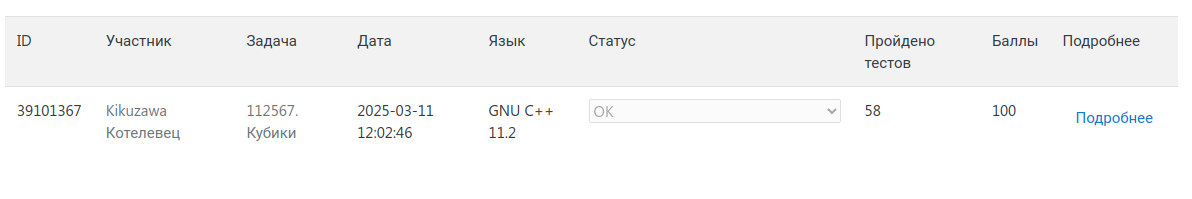
**Входные данные**

Первая строка входного файла содержит число *N*( 1 ≤ *N*≤ 1000000 ) и количество различных цветов, в которые могут быть раскрашены кубики — *M*( 1 ≤ *M*≤ 1000000 ). Следующая строка содержит *N*целых чисел от 1 до *M*— цвета кубиков.

**Выходные данные**

Выведите в выходной файл все такие *K*, что у Пети может быть *K*кубиков

|  |
| --- |
| #include <iostream>  #include <vector>  #include <cmath>  #include <sstream>  using namespace std;  class StringHash {  private:  const long X = 257;  const long P = 1000000007;  vector<long> hDir;  vector<long> hRev;  vector<long> x;  public:  StringHash(const vector<int>& digits) {  int len = digits.size();  hDir.resize(len + 1);  hRev.resize(len + 1);  x.resize(len + 1);  x[0] = 1;  // Вычисляем хеши и степени X  for (int i = 1; i <= len; i++) {  hDir[i] = (hDir[i - 1] \* X + digits[i - 1]) % P;  hRev[i] = (hRev[i - 1] \* X + digits[len - i]) % P;  x[i] = (x[i - 1] \* X) % P;  }  }  bool isEqualRev(int len, int from1, int from2) {  // Сравниваем хеши зеркальных префиксов  long one = (hDir[from1 + len] + hRev[hRev.size() - 1 - from2 - len] \* x[len]) % P;  long two = (hRev[hRev.size() - 1 - len] + hDir[from1] \* x[len]) % P;  return one == two;  }  };  int main() {  int n, m;  cin >> n >> m;  vector<int> digits(n);  for (int i = 0; i < n; i++) {  cin >> digits[i];  }  StringHash sH(digits);  stringstream result;  // Проверяем все возможные длины префиксов  for (int i = n / 2; i > 0; i--) {  if (sH.isEqualRev(i, 0, i)) {  result << n - i << " ";  }  }  result << n;  // Выводим результат  cout << result.str() << endl;  return 0;  } |



Задание 3

Строка называется палиндромом, если она читается одинаково как слева направо, так и справа налево. Например, строки abba, ata являются палиндромами.

Дана строчка. Ее подстрокой называется некоторая непустая последовательность подряд идущих символов. Напишите программу, которая определит, сколько подстрок данной строки является палиндромами.

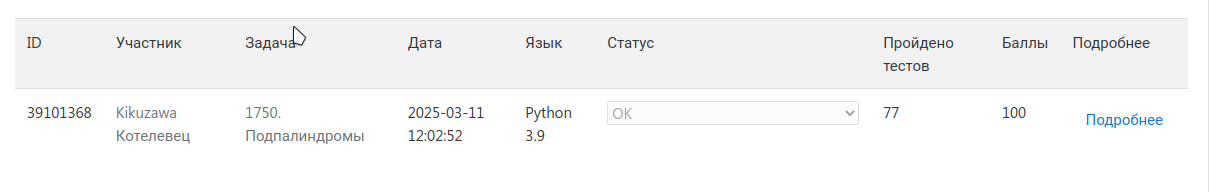
**Входные данные**

Вводится одна строка, состоящая из маленьких латинских букв. Длина строки не превышает 100000 символов.

**Выходные данные**

Выведите одно число — количество подстрок данной строки, являющихся палиндромами

|  |
| --- |
| from array import array, ArrayType  def find\_palindromes(s: str) -> int:  # Преобразуем строку для работы с палиндромами  transformed\_string: str = "@#" + "#".join(s) + "#$"  palindrome\_lengths: ArrayType[int] = array("i", [0] \* len(transformed\_string))  center: int = 0  right\_boundary: int = 0  for i in range(1, len(transformed\_string) - 1):  if i < right\_boundary:  palindrome\_lengths[i] = min(right\_boundary - i, palindrome\_lengths[2 \* center - i])  # Расширяем палиндром вокруг центра i  while transformed\_string[i + palindrome\_lengths[i] + 1] == transformed\_string[i - palindrome\_lengths[i] - 1]:  palindrome\_lengths[i] += 1  # Обновляем центр и правую границу, если нашли более длинный палиндром  if i + palindrome\_lengths[i] > right\_boundary:  center, right\_boundary = i, i + palindrome\_lengths[i]  # Считаем количество палиндромных подстрок  return sum((length + 1) // 2 for length in palindrome\_lengths)  def main() -> None:  input\_string: str = input()  print(find\_palindromes(input\_string))  if \_\_name\_\_ == "\_\_main\_\_":  main() |



Задание 4

Мальчик Кирилл написал однажды на листе бумаги строчку, состоящую из больших и маленьких латинских букв, а после этого ушел играть в футбол. Когда он вернулся, то обнаружил, что его друг Дима написал под его строкой еще одну строчку такой же длины. Дима утверждает, что свою строчку он получил циклическим сдвигом строки Кирилла на несколько шагов вправо (циклический сдвиг строки abcde на 2 позиции вправо даст строку deabc). Однако Дима известен тем, что может случайно ошибиться в большом количестве вычислений, поэтому Кирилл в растерянности – верить ли Диме? Помогите ему! По данным строкам выведите минимальный возможный размер сдвига или -1, если Дима ошибся.

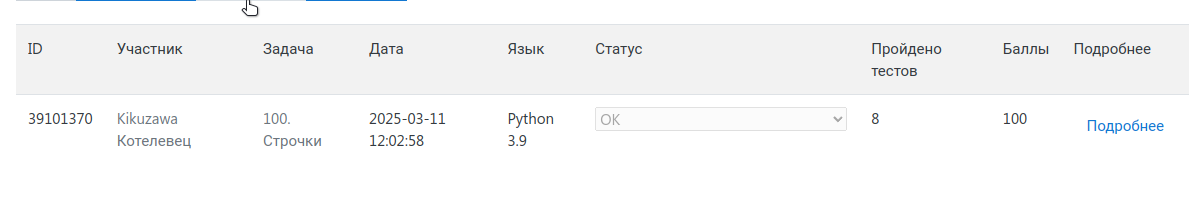
**Входные данные**

Первые две строки входных данных содержат строки Кирилла и Димы, соответственно. Длины строк одинаковы, не превышают 10000 и не равны 0.

**Выходные данные**

Выведите единственное число – ответ  на вопрос задачи.

|  |
| --- |
| def minimal\_cyclic\_shift(kirill: str, dima: str) -> int:  # Если строки совпадают, сдвиг равен 0  if kirill == dima:  return 0  # Удваиваем строку Димы для поиска циклического сдвига  doubled\_dima = dima \* 2  shift\_index = doubled\_dima.find(kirill)  # Если Кириллову строку не удалось найти, значит, Дима ошибся  return -1 if shift\_index == -1 else shift\_index  def main() -> None:  kirill\_string: str = input()  dima\_string: str = input()  result = minimal\_cyclic\_shift(kirill\_string, dima\_string)  print(result)  if \_\_name\_\_ == "\_\_main\_\_":  main() |



Задание 5

Для рекламы своей новой продукции в Китае одна компания решила разместить на небоскребе рекламный щит. Щит состоит из лампочек, организованных в форме прямоугольной сетки из nn строк и mm столбцов. В любой момент каждая из лампочек может быть либо включена, либо выключена.

Рекламное сообщение состоит из kk иероглифов, которые будут показываться один за другим. Для каждого иероглифа известно, какие лампочки должны быть включены при отображении этого иероглифа. Остальные лампочки должны быть выключены.

Для управления рекламным щитом разрабатывается специальная система. Система может включать и выключать лампочки целыми группами. Все лампочки разбиваются на несколько групп так, что в каждом иероглифе лампочки из одной группы должны быть либо все включены, либо все выключены.

Для оптимизации работы системы управления необходим разбить лампочки на минимальное возможное число таких групп. Помогите сотрудникам рекламного отдела компании решить эту задачу.

**Формат входного файла**

В первой строке входного файла заданы числа kk, nn и mm (1≤k,n,m≤1001≤k,n,m≤100) - количество иероглифов в рекламном сообщении, высота и ширина рекламного щита.

Далее, в knkn строках идет описание иероглифов. Каждый из kk иероглифов задается nn строками по mm символов в каждой. Все эти строки состоят только из символов «\*» и «.», «\*» соответствует включенной лампочке, «.» - выключенной.

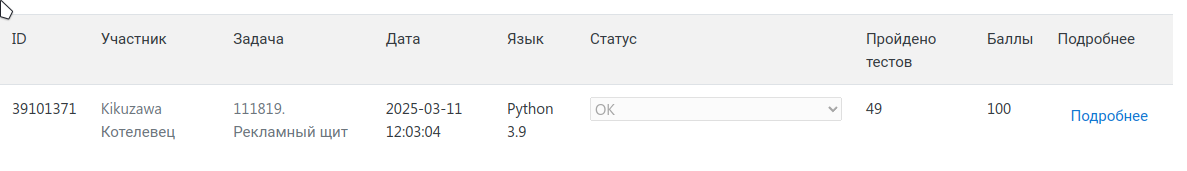
**Формат выходного файла**

Выведите минимальное число групп, на которое можно разбить лампочки.

**Пояснения к примеру**

В приведенном примере можно разбить лампочки на группы следующим образом: две лампочки из первого столбца образуют одну группу, две лампочки из последнего столбца - вторую, а каждая из двух оставшихся лампочек образует отдельную группу.

|  |
| --- |
| from collections import defaultdict  from typing import List, Sequence  from itertools import product  def find\_minimum\_groups(k: int, n: int, m: int, hieroglyphs: Sequence[str]) -> int:  lamp\_groups: defaultdict[str, int] = defaultdict(int)  for row, col in product(range(n), range(m)):  # Формируем ключ для группы лампочек на основе текущего столбца и всех иероглифов  lamp\_key = ''.join(hieroglyphs[glyph \* n + row][col] for glyph in range(k))  lamp\_groups[lamp\_key] += 1  # Количество уникальных групп лампочек  return len(lamp\_groups)  def main() -> None:  k, n, m = map(int, input().split())  hieroglyphs: List[str] = [input() for \_ in range(k \* n)]  print(find\_minimum\_groups(k, n, m, hieroglyphs))  if \_\_name\_\_ == '\_\_main\_\_':  main() |



Задание 6

Знаменитый художник Вася только что закончил работу над своим новым шедевром и хочет знать, сколько он сможет получить за свой труд.

Картина представляет собой прямоугольник *N* на *M* сантиметров, разделенный на маленькие квадратики 1 на 1 сантиметр со сторонами, параллельными сторонам картины. Для достижения гармонии каждый из этих квадратиков Вася покрасил одним из 26 особых цветов, обозначаемых маленькими латинскими буквами.

Стоимость картины в точности равна количеству «симпатичных» частей в ней. Частью картины называется любой прямоугольник, который может быть вырезан из нее по границам квадратиков. Часть называется «симпатичной», если при выполнении симметрии относительно ее центра получается прямоугольник, раскрашенный также, как и исходная часть. Например, в картине, раскрашенной так:

abc  
acb

симпатичными являются все части, состоящие из одного квадратика (их 6), а также части

bc и a

cb и a

Напишите программу, которая по информации о шедевре Васи определит его стоимость.

**Входные данные**

В первой строке содержатся два числа *N* и *M* (1 ≤ *N*, *M* ≤ 100). В следующих *N* строках идут строки, состоящие из *M* маленьких латинских символов. Символ в *i*-й строке *j*-м столбце определяет цвет соответствующего квадратика картины.

**Выходные данные**

Выведите стоимость шедевра — количество частей, симметричных относительно своего центра.

***Комментарии к примерам тестов***

Этот пример разобран в условии

Симпатичными являются шесть частей 1x1, одна часть 1x2 и сама картина.

***Частичные ограничения***

Первая группа состоит из тестов, в которых *N, M* ≤ 15. Данная группа оценивается в 30 баллов.

Вторая группа состоит из тестов, в которых *N, M* ≤ 50. Данная группа оценивается в 30 баллов.

|  |
| --- |
| #include <iostream>  #include <vector>  #include <string>  #include <algorithm>  using namespace std;  // Проверяет, является ли прямоугольник симметричным относительно его центра  bool is\_symmetric(const vector<string>& grid, int top, int bottom, int left, int right) {  int rows = bottom - top + 1;  int cols = right - left + 1;  // Проверяем симметрию сверху вниз  for (int i = 0; i <= rows / 2; ++i) {  string topRow = grid[top + i].substr(left, cols);  string bottomRow = grid[bottom - i].substr(left, cols);  reverse(bottomRow.begin(), bottomRow.end());  if (topRow != bottomRow) {  return false;  }  }  return true;  }  // Подсчитывает количество симпатичных прямоугольников  int count\_pretty\_rectangles(const vector<string>& grid, int n, int m) {  int count = 0;  // Перебор всех возможных прямоугольников  for (int top = 0; top < n; ++top) {  for (int bottom = top; bottom < n; ++bottom) {  for (int left = 0; left < m; ++left) {  for (int right = left; right < m; ++right) {  if (is\_symmetric(grid, top, bottom, left, right)) {  ++count;  }  }  }  }  }  return count;  }  int main() {  int n, m;  cin >> n >> m;  vector<string> grid(n);  for (int i = 0; i < n; ++i) {  cin >> grid[i];  }  int result = count\_pretty\_rectangles(grid, n, m);  cout << result << endl;  return 0;  } |

