Зачет по алгоритмам и структурам данных в Python

1 курс

МФТИ, 2021

Содержание

1 Хеширование. Алгоритм Рабина-Карпа

Хэширование - процесс преобразования массива данных произвольного размера в массив данных установленного размера. Функция, преобразующая таким образом массив называется Хэш-функцией, а результат - хэшем.

Свойства хэш-функции:

1. одинаковые данные - одинаковый хэш
2. разные данные "почти всегда разный хэш
3. весь доступный диапазон хэшей используется по максимуму
4. даже небольшое изменение во входных данных должно давать абсолютно разный хэш

Почему почти всегда разный? Мощность множества образов хэш-преобразования сильно меньше мощности множества прообразов. По принципу Дирихле в какой-то момет разные данные будут иметь одинаковый хэш.

Алгоритм Рабина-Карпа

Поиск подстроки в строке с использованием хеширования.

Для работы алгоритма потребуется считать хеш подстроки s[i..j]. Делать это можно следующим образом:

1. Рассмотрим хеш s[0..j]:
   $hash(s[0..j]) = s[0] + ps[1] + ... + p^{i-1}s[i-1] + p^i s[i] + ... + p^{j-1}s[j-1] + p^j s[j]$
2. Разобьем это выражение на две части:
   $\text{hash}(s[0..j]) = (s[0] + ps[1] + ... + p^{i-1}s[i-1]) + (p^i s[i] + ... + p^{j-1}s[j-1] + p^j s[j])$
3. Вынесем из последней скобки множитель p^i:
   $\text{hash}(s[0..j]) = (s[0] + ps[1] + ... + p^{i-1}s[i-1]) + p^i(s[i] + ... + p^{j-i-1}s[j-1] + p^{j-i}s[j])$
4. Выражение в первой скобке есть не что иное, как хеш подстроки $s[0..i-1]$, $^-s[i..j]$. Итак, мы получили, что:
   $\text{hash}(s[0..j]) = \text{hash}(s[0..i-1]) + p^i\text{hash}(s[i..j])$
5. Отсюда получается следующая формула для $\text{hash}(s[i..j])$:

$\text{hash(s[i..j])} = (1/\text{p}^i)(\text{hash}(s[0..j]) - \text{hash}(s[0..i-1]))$

Реализация:

```
M = (1 << 61) - 1
B = 257
def poly_hash(s):
    h = [0]
    p = 1
```

```
    for c in s:
        h.append(((h[-1] * B) % M + ord(c)) % M)
        p = (p * B) % M
    return (h, p)

s = input() # искомаястрока
t = input() # строка, вкоторойищем
s_hash, p = poly_hash(s)
s_hash = s_hash[-1]
h, _ = poly_hash(t)
pos = []
for l in range(len(t) - len(s) + 1):
    r = l + len(s)
    if s_hash == (h[r] - (h[l] * p) % M) % M:
        pos.append(str(l))
for i in range(len(pos)):
    if s != pos[i]:
        pos.pop(i)
if not pos:
    print(-1)
else:
    print("␣".join(pos))
```

2 Открытая хеш и закрытая хеш-таблицы. Проблема удаления из закрытой хеш-таблицы. Перехеширование.

Хэш-таблица - структура данных, реализующая интерфейс ассоциативного массива, позволяет хранить пары (ключ, значение) и поддерживате три типа операции:

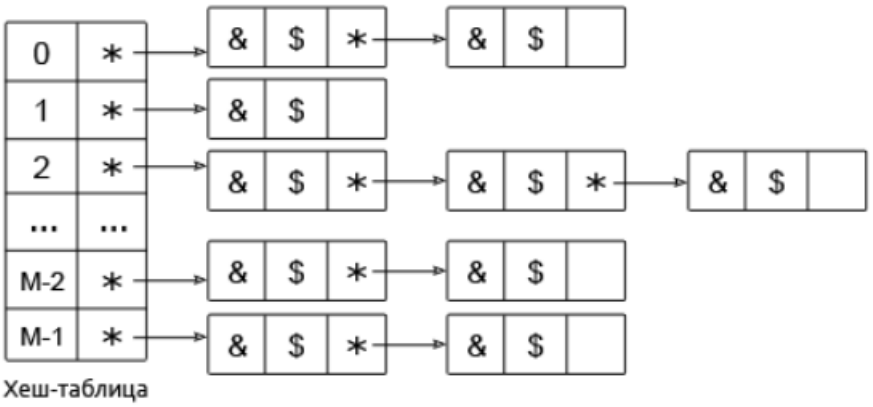
1. Добавление новой пары
2. Поиск по ключу
3. Удаление пары по ключу

2.1 Открытое хеширование.

Принцип организации хеш-таблицы методом открытого хеширования заключается в реализации логически связанных цепочек, начинающихся в ячейках хеш-таблицы. Под цепочками подразумеваются связанные списки, указатели на которые хранятся в ячейках хеш-таблицы. Каждый элемент связанного списка – блок данных, и если с некоторым указателем, хранящимся по адресу i, связаны n таких блоков (n>1), то это свидетельствует о том, что n ключей получили один и тот же хеш-код i, т. е. имеет место коллизия. Но метод открытого хеширования устраняет конфликт, поскольку

данные хранятся не в самой таблице, а в связных списках, которые увеличиваются при возникновении конфликта.

На рисунке изображены связанные списки со ссылающейся на них хеш-таблицей (ее размер = M). Первый столбец таблицы содержит хешированные значения ключей, второй – ссылки на списки. Количество последних ограничено лишь числом элементов исходного массива (он не показан, но предполагается). Состоят списки из трех (последний элемент подсписка – из двух) полей: — адрес элемента списка, $ — данные, * — указатель (ссылка).



Если в исходном массиве было всего N элементов (столько же будут содержать в совокупности и все списки), то средняя длина списков будет равна =N/M, где M – число элементов хеш-таблицы, – коэффициент заполнения хеш-таблицы. Предположив, например, что в списке на рисунке выше M=5 (заклеив 4-ую по счету строку), получим среднее число списков =2.

Чтобы увеличить скорость работы операций поиска, вставки и удаления нужно, зная N, подобрать Mпримерно равное ему, т. к. тогда будет равняться 1-ому или 1-ому, следовательно, можно рассчитывать на оптимальное время, в лучшем случае равное O(1). В худшем случае все N элементов окажутся в одном списке, и тогда, например, операция нахождения элемента (в худшем случае) потребует O(N) времени.

Закрытое хеширование.

Первый метод назывался открытым, потому что он позволял хранить сколь угодно много элементов, а при закрытом хешировании их количество ограниченно размером хеш-таблицы.

В отличие от открытого хеширования закрытое не требует каких-либо дополнительных структур данных. В ячейках таблицы хранятся не указатели, а элементы исходного массива, доступ к каждому из которых осуществляется по хеш-коду ключа, при этом одна ячейка может содержать только один элемент. Сам процесс заполнения хеш-таблицы с использованием алгоритма закрытого хеширования осуществляется следующим образом: имеется изначально пустая хеш-таблица T размера M, массив A размера N (MN) и хеш-функция h(), пригодная для обработки ключей массива A;

элемент xi, ключ которого keyi, помещается в одну из ячеек хеш-таблицы, руководствуясь следующим правилом: a) если h(keyi) – номер свободной ячейки таблицы T, то в последнюю записывается xi; b) если h(keyi) – номер уже занятой ячейки таблицы T, то на занятость проверяется другая ячейка, если она свободна то xi заноситься в нее, иначе вновь проверяется другая ячейка, и так до тех пор, пока не найдется свободная или окажется, что все M ячеек таблицы заполнены. Последовательность, в которой просматриваются ячейки хеш-таблицы, называется последовательностью проб. Последовательность проб задается специальной функцией, например интервал между просматриваемыми ячейками может вычисляться линейно, или увеличиваться на некоторое изменяющееся значение.

Одна из возможных реализаций:

```
def hesh(a, b, s):
  s = s[::-1]
  d = 0
  for i in range(len(s)-1, -1, -1):
    d += ord(s[i])*(a**(i))
  return d%b

ht=[[] for i in range(10)]
n = int(input())
for i in range(n):
  key, value = input().split()
  h = hesh(91, 100, key)
  m = h%10
  flag = 1
  if (ht[m]==[]):
    ht[m].append([h, key, value])
    flag = 0
  else:
    for i in ht[m]:
      if (i[1] == key):
        i[2] = value
        flag = 0
  if (flag == 1):
    ht[m].append([h, key, value])
for i in ht:
  if(i==[]):
    continue
  else:
    print(ht.index(i))
    for j in i:
      print(*j)
```

3 Словари и множества в Python.

Множества

Множество (set) - встроенная структура данных языка Python, имеющая следующие свойства:

1. множества - это коллекция(содержит элементы)
2. множество неупорядочено (Множество не записывает (не хранит) позиции или порядок добавления его элементов. Таким образом, множество не имеет свойств последовательности (например, массива): у элементов множества нет индексов, невозможно взять срез множества...
3. элементы множества уникальны
4. элементы множества - хешируемые объекты

Литералом множества являются фигурные скобки , в которых через запятую указаны элементы. Так, ещё один способ создать непустое множество - воспользоваться литералом.

При попытке добавления изменяемого объекта возникнет ошибка

При попытке удаления элемента, не входящего в множество, возникает ошибка KeyError.

Множества Python поддерживают привычные математические операции.

Используемые функции:

1. len(s) - число элементов в множестве (размер множества).
2. x in s - принадлежит ли x множеству s.
3. set.isdisjoint(other) - истина, если set и
4. other не имеют общих элементов.
5. set == other - все элементы set принадлежат
6. other, все элементы other принадлежат set.
7. set.issubset(other) или set <= other - все элементы set принадлежат other.
8. set.issuperset(other) или set >= other - аналогично.
9. set.union(other, ...) или set | other | ... - объединение нескольких множеств.
10. set.intersection(other, ...) или set other ... - пересечение.
11. set.difference(other, ...) или set - other - ... - множество из всех элементов set, не принадлежащие ни одному из other.
12. set.symmetric_difference(other) - множество из элементов, встречающихся в одном множестве, но не встречающиеся в обоих.

13. set.copy() - копия множества.

Операции, непосредственно изменяющие множество:

1. set.update(other, ...) - объединение.
2. set.intersection_update(other, ...) - пересечение.
3. set.difference_update(other, ...) - вычитание.
4. set.symmetric_difference_update(other) - множество из элементов, встречающихся в одном множестве, но не встречающиеся в обоих.
5. set.add(elem) - добавляет элемент в множество.
6. set.remove(elem) - удаляет элемент из множества. KeyError, если такого элемента не существует.
7. set.discard(elem) - удаляет элемент, если он находится в множестве.
8. set.pop() - удаляет первый элемент из множества. Так как множества не упорядочены, нельзя точно сказать, какой элемент будет первым.
9. set.clear() - очистка множества.

Неизменяемые объекты

В Python существует неизменяемая версия множества - frozenset. Этот тип объектов поддерживает все операции обычного множества set, за исключением тех, которые его меняют.

Неизменяемые множества являются хешируемыми объектами, поэтому они могут быть элементами множества set. Так можно реализовать, например, множество множеств, где множество set состоит из множеств типа frozenset.

Для создания frozenset используется функция frozenset(iterable), в качестве аргумента принимающая итерирумый объект.

```
>>> FS = frozenset({1, 2, 3})
>>> FS
frozenset({1, 2, 3})
>>> A = {1, 2, 4}
>>> FS & A
frozenset({1, 2})
>>> A & FS
{1, 2}
```

1, 2 В этом примере показано создание frozenset из обычного множества 1, 2, 3. Обратите внимание на тип возвращаемого объекта для операции пересечения . Возвращаемый объект имеет тип, соответствующий типу первого аргумента. Такое же поведение будет и с другими операциями над множествами.

Словари

Словарь (dictionary) в Python - это ассоциативный массив. Ассоциативный массив это структура данных, содержащая пары вида ключ:значение. Ключи в ассоциативном массиве уникальны.

В Python есть встроенный ассоциативный массив - dict. Его реализация основана на хеш-таблицах. Поэтому

ключом может быть только хешируемый объект значением может быть любой объект

Пустой словарь можно создать двумя способами:

```
>>> d1 = dict()
>>> d2 = {}
>>> d1
{}
>>> d2
{}
>>> type(d1)
<class 'dict'>
>>> type(d2)
<class 'dict'>
```

Добавить элемент в словарь можно с помощью квадратных скобок:

```
>>> domains = {}
>>> domains['ru'] = 'Russia'
>>> domains['com'] = 'commercial'
>>> domains['org'] = 'organizations'
>>> domains
{'ru': 'Russia', 'com': 'commercial', 'org': 'organizations'}
```

Из этого примера видно, что литералом словаря являются квадратные скобки, в которых через запятую перечислены пары в формате ключ:значение.

Удалить элемент можно с помощью оператора del. Если ключа в словаре нет, произойдет ошибка KeyError

```
>>> domains
{'ru': 'Russia', 'com': 'commercial', 'org': 'organizations'}
>>> del domains['de']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'de'
>>> del domains['ru']
>>> domains
{'com': 'commercial', 'org': 'organizations'}
```

Кроме того, для добавления, получения и удаления элементов есть методы dict.setdefault, dict.get, dict.pop, которые задействует дополнительный аргумент на случай, если ключа в словаре нет

Ключом может являться и число: int или float. Однако при работе со словарями в Python помните, что два ключа разные, если для них верно k1 != k2 True.

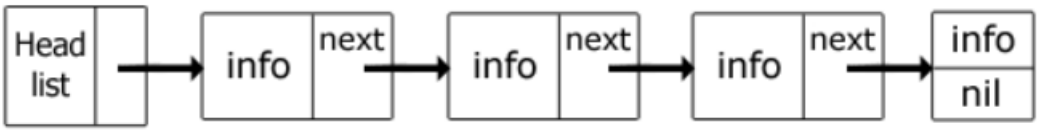
4 Списки: односвязный, двусвязный, кольцо. Время работы основных операций

Списки в Python - упорядоченные изменяемые коллекции объектов произвольных типов. Структура данных, представляющая собой конечное множество упорядоченных элементов (узлов), связанных друг с другом посредством указателей, называется связным списком.

Каждый элемент связного списка содержит поле с данными, а также указатель (ссылку) на следующий и/или предыдущий элемент. Эта структура позволяет эффективно выполнять операции добавления и удаления элементов для любой позиции в последовательности.

По типу связности выделяют односвязные, двусвязные, XOR-связные, кольцевые и некоторые другие списки. Каждый узел односвязного (однонаправленного связного) списка содержит указатель на следующий узел. Из одной точки можно попасть лишь в следующую точку, двигаясь тем самым в конец. Так получается своеобразный поток, текущий в одном направлении.

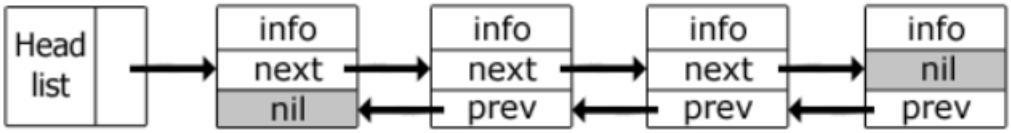
Односвязный список



На изображении каждый из блоков представляет элемент (узел) списка. Здесь и далее Head list – заголовочный элемент списка (для него предполагается поле next). Он не содержит данные, а только ссылку на следующий элемент. На наличие данных указывает поле info, а ссылки – поле next (далее за ссылки будет отвечать и поле prev). Признаком отсутствия указателя является поле nil. Односвязный список не самый удобный тип связного списка, т. к. из одной точки можно попасть лишь в следующую точку, двигаясь тем самым в конец

Двусвязный список

Когда кроме указателя на следующий элемент есть указатель на предыдущий, то такой список называется двусвязным.



Та особенность двусвязного списка, что каждый элемент имеет две ссылки: на следующий и на предыдущий элемент, позволяет двигаться как в его конец, так и в начало. Операции добавления и удаления здесь наиболее эффективны, чем в односвязном списке, поскольку всегда известны адреса тех элементов списка, указатели

которых направлены на изменяемый элемент. Но добавление и удаление элемента в двусвязном списке, требует изменения большого количества ссылок, чем этого потребовал бы односвязный список.

Еще один вид связного списка – кольцевой список. В кольцевом односвязном списке последний элемент ссылается на первый. В случае двусвязного кольцевого списка – плюс к этому первый ссылается на последний. Таким образом, получается зацикленная структура.

5 Двоичное дерево поиска

Двоичный, или бинарный, поиск значения в списке или массиве используется только для упорядоченных последовательностей, то есть отсортированных по возрастанию или убыванию.

Описание алгоритма.

1. Находится средний элемент последовательности. Для этого первый и последний индексы связываются с переменными, а индекс среднего элемента вычисляется.

2. Значение среднего элемента сравнивается с искомым значением. В зависимости от того, больше оно или меньше значения среднего элемента, дальнейший поиск будет происходить только в левой или только в правой половинах массива. Если значение среднего элемента оказывается равным искомому, поиск завершается.

3. Иначе одна из границ исследуемой последовательности сдвигается. Если искомое значение больше значения среднего элемента, то нижняя граница сдвигается за средний элемент на один элемент справа. Если искомое значение меньше значения среднего элемента, то верхняя граница сдвигается на элемент перед средним.

4. Снова находится средний элемент теперь уже в выбранной половине. Описанный выше алгоритм повторяется для данного среза.

```
from random import randint

# Созданиесписка ,
# егосортировкаповозрастанию
# ивыводнаэкран
a = []
for i in range(15):
    a.append(randint(1, 50))
a.sort()
print(a)
# искомоечисло
value = int(input())
mid = len(a) // 2
low = 0
```

```
high = len(a) - 1
while a[mid] != value and low <= high:
    if value > a[mid]:
        low = mid + 1
    else:
        high = mid - 1
    mid = (low + high) // 2
if low > high:
    print("No value")
else:
    print("ID =", mid)
```

6 Куча. Сортировка кучей

Куча - структура данных, которую можно описать еще как очередь с приоритетами. Она имеет вид двоичного дерева, со следующими свойствами:

1. Значение в любой вершине не меньше(не больше) значения ее потомков
2. На i-том(нумерация с 0) слое 2^i вершин, кроме последнего.
3. Последний слой заполняется слева направо без пропусков. Для элементов ячейки с индексом i потомки будут храниться в ячейках с индексами 2i+1 и 2i+2 (Предок - (i-1)//2)

Для элементов ячейки с индексом i потомки будут храниться в ячейках с индексами 2i+1 и 2i+2 (Предок - (i-1)//2)

Пусть минимальный элемент вверху

Реализация кучи

```
from math import log2
def sift_up(heap, i):
    while i > 0 and heap[(i - 1) // 2] > heap[i]:
        heap[i], heap[(i - 1) // 2] = heap[(i - 1) // 2], heap[i]
        i = (i - 1) // 2

def sift_down(heap, i):
    n = len(heap)
    while i * 2 + 1 < n:
        j = i
        if heap[i] > heap[i * 2 + 1]:
            j = i * 2 + 1
        if i * 2 + 2 < n and heap[j] > heap[i * 2 + 2]:
            j = i * 2 + 2
        if i == j:
```

```
            break
        heap[i], heap[j] = heap[j], heap[i]
        i = j

def add(heap, x):
    heap.append(x)
    sift_up(heap, len(heap) - 1)

def extract_min(heap):
    x = heap[0]
    heap[0] = heap.pop()
    sift_down(heap, 0)
    return x
```

Пирамидальная сортировка

Общая идея пирамидальной сортировки заключается в том, что сначала строится пирамида из элементов исходного массива, а затем осуществляется сортировка элементов.

Фаза 1 сортировки: построение пирамиды:

Начать построение пирамиды можно с a[k]...a[n], k = [size/2]. Эта часть массива удовлетворяет свойству пирамиды, так как не существует индексов i,j: i = 2i+1 (или j = 2i+2)... Просто потому, что такие i,j находятся за границей массива.

Следует заметить, что неправильно говорить о том, что a[k]..a[n] является пирамидой как самостоятельный массив. Это, вообще говоря, не верно: его элементы могут быть любыми. Свойство пирамиды сохраняется лишь в рамках исходного, основного массива a[0]...a[n].

Далее будем расширять часть массива, обладающую столь полезным свойством, добавляя по одному элементу за шаг. Следующий элемент на каждом шаге добавления - тот, который стоит перед уже готовой частью.

Чтобы при добавлении элемента сохранялась пирамидальность, будем использовать следующую процедуру расширения пирамиды a[i+1]..a[n] на элемент a[i] влево:

Смотрим на сыновей слева и справа - в массиве это a[2i+1] и a[2i+2] и выбираем наибольшего из них. Если этот элемент больше a[i] - меняем его с a[i] местами и идем к шагу 2, имея в виду новое положение a[i] в массиве. Иначе конец процедуры.

Фаза 2: собственно сортировка:

Как видно из свойств пирамиды, в корне всегда находится максимальный элемент. Отсюда вытекает алгоритм фазы 2:

Берем верхний элемент пирамиды a[0]...a[n] (первый в массиве) и меняем с последним местами. Теперь "забываем"об этом элементе и далее рассматриваем массив a[0]...a[n-1]. Для превращения его в пирамиду достаточно просеять лишь новый первый элемент.

Повторяем шаг 1, пока обрабатываемая часть массива не уменьшится до одного элемента.

```
def heap_sort (sequence):
    length = len(sequence)
```

```
    # Формированиепервичнойпирамиды
    for index in range((length >> 1) - 1, -1, -1):
        sift_down(index, length)
    # Окончательное \ упорядочение
    for index in range(length - 1, 0, -1):
        sequence[0], sequence[index] = sequence[index], sequence[0]
        sift_down(0, index)
```

7 Определение графа. Степень вершины, петли, кратные рёбра. Цепи, пути и циклы.

Граф G - пара множеств <V, E>, где V - множество вершин, а E - множество ребер (неупорядоченных пар v)

Петля - это дуга, начало и конец которой совпадают.

Простой граф - граф без кратных ребер и петель.

Степень вершины - количество приходящих в нее ребер и удвоенное количество петель.

Маршрут - последовательность вершин, в которой каждая вершина соединена со следующей ребром.

Длина маршрута - количество ребер.

Цепь - маршрут без повторяющихся ребер.

Простая цепь - цепь без повторяющихся вершин.

Путь - ориентированный маршрут.

Простой путь - ребра не повторяются.

Элементарный путь - вершины тоже не повторяются.

Цикл - цепь, где первая и последняя вершины совпадают.

8 Сильная и слабая связность графа. Компоненты связности.

Компонента связности - подграф данного графа, вершины которого связаны.

Граф называется **связным**, если любая пара его вершин связна.

Для ориентированного графа:

Компонента связности называется **слабой**, если мы не учитываем ориентацию ребер.

Если учитываем - связность **сильная**.

9 Способы представления графа в памяти.

Есть три способа представления графа в памяти.

1. Список вершин + список ребер. Самая неудачная реализация. Тяжело выполнять поиск поэлементно O(N) на перебор соседей.

2. Матрица смежности. Более наглядный и быстрый способ
3. Списки смежности. Возможно, самый быстрый, т.к. реализуется через dict, но ест много памяти

Реализация матрицы смежности

```
M, N = [int(x) for x in input().split()]
V = [None]*N
index = {}
A = [[0]*N for i in range(N)]
for i in range(N):
    v1, v2 = input().split()
    for v in v1, v2:
        if v not in index:
            V.append(v)
            index[v] = len(V)-1
    v1_i = index[v1]
    v2_i = index[v2]
    A[v1_i][v2_i] = 1
    A[v2_i][v1_i] = 1
```

Реализация списка смежности

```
M, N = [int(x) for x in input().split()]
G = {}
for i in range(N):
    v1, v2 = input().split()
    for v, u in range (v1, v2), (v2, v1):
        if v not in G:
            G[v]  = {u}
        else:
            G[v].add(u)
```

10 Выделение компоненты связности обходом в глубину.

Обход графа в глубину

Обход в глубину (англ. Depth-First Search, DFS) — один из основных методов обхода графа, часто используемый для проверки связности, поиска цикла и компонент сильной связности и для топологической сортировки.

Общая идея алгоритма состоит в следующем: для каждой не пройденной вершины необходимо найти все не пройденные смежные вершины и повторить поиск для них.

Пошаговое представление:

1. Выбираем любую вершину из еще не пройденных, обозначим ее как u.
2. Запускаем процедуру dfs(u).
3. Помечаем вершину u как пройденную.
4. Для каждой не пройденной смежной с u вершиной (назовем ее v) запускаем dfs(v).
5. Повторяем шаги 1 и 2, пока все вершины не окажутся пройденными.

Зачем?

1. Выделение компонент связности
2. Подсчет к-ва компонент
3. Поиск простого цикла
4. Проверка на двудольность
5. Топологическая сортировка
6. Выделение сильных компонент связности

Реализация с цветами

```
def doDfs(G[n]: Graph):
    color = array[n, white]
    def dfs(u: int):
        color[u] = gray
        for v: (u, v) in G:
            if color[v] == white:
                dfs(v)
        color[u] = black
    for i in range(1, n):
        if color[i] == white:
            dfs(i)
```

Подсчет компонент связности

#Реализация обходавглубину

```
def dfs(vertex, G, used):
    if used is None:
        used = set()
    used.add(vertex)
    for neighbour in G[vertex]:
        if neighbour not in used:
```

```
            dfs(neighbour, G, used)

#Подсчет компонентсвязности
#start - вершина, скоторойначинаемпроверку
used = {start}
N = 0
for vertex in G:
    if vertex not in used:
        dfs(vertex, G, used)
        N+=1
```

Выделение компонент связности. Алгоритм Касарайю

```
def read_graph(filename):  # функциячтенияграфаизфайла
    N = M = None
    G = {}
    G_inv = {}
    for line in open(filename, 'r'):
        if N is None:
            N, M = map(int, line.split())
            continue
        v1, v2 = line.split()
        for v in v1, v2:
            if v not in G:
                G[v] = []
                G_inv[v] = []
        G[v1].append(v2)
        G_inv[v2].append(v1)
    return G, G_inv

def dfs(G, vertex, used, stack):    # used - списокпройденныхвершин
    used.add(vertex)
    for v in G[vertex]:
        if v not in used:
            dfs(G, v, used, stack)
    stack.append(vertex)

G, G_inv = read_graph('Косараю.txt')
used = set()
stack = []
for vertex in G.keys():
    if vertex not in used:
        dfs(G, vertex, used, stack)
print(stack)

N = 0
```

```
used = set()
while len(stack) > 0:
    v = stack.pop()
    if v not in used:
        componenta = []
        dfs(G_inv, v, used, componenta)
        print(componenta)
        N += 1
print(N)
```

11 Проверка двудольности графа.

```
def read\_graph():
    N, M = map(int, input().split())
    G = {}
    for i in range(M):
        v1, v2 = input().split()
        for v in v1, v2:
            if v not in G:
                G[v] = []
        G[v1].append(v2)
        G[v2].append(v1)
    return G, N, M

def dvydolnost(G, vertex, black, white, used, last = 0):
    used.add(vertex)
    if last == 0:
        white.append(vertex)
    else:
        black.append(vertex)
    for neig in G[vertex]:
        if neig not in used:
            flag = dvydolnost(G, neig, black, white,
            used, (last + 1) % 2)
            if flag == 0:
                return 0
        elif last == 0 and neig in white:
            return 0
        elif last == 1 and neig in black:
            return 0
    return (black, white)

G, N, M = read\_graph()
black = []
white = []
```

```
used = set()
for v in G.keys():
    if v not in used:
        flag = dvydolnost(G, v, black, white, used)
if flag == 0:
    print('Это не двудольный граф')
else:
    print('Первая доля: ', *black)
    print('Вторая доля: ', *white)
```

12 Проверка графа на ацикличность или нахождение цикла обходом в глубину.

Проверка цикла в ориентированном графе

Цикл в ориентированном графе можно обнаружить по наличию ребра, ведущего из текущей вершины в вершину, которая в настоящий момент находится в стадии обработки, то есть алгоритм DFS зашел в такую вершину, но еще не вышел из нее.

```
def read_graph():
    N, M = map(int, input().split())
    G = {}
    for i in range(M):
        v1, v2 = input().split()
        for v in v1, v2:
            if v not in G:
                G[v] = []
        G[v1].append(v2)
    return G, N, M

def dfs(G, vertex, used, stack):
    zikl = []
    used.add(vertex)
    for v in G[vertex]:
        if v not in used:
            stack.append(v)
            componenta = dfs(G, v, used, stack)
            if componenta != []:
                return componenta
            stack.pop()
        else:
            if v in stack:
                return stack[stack.index(v)::]
    return []

def ifcomponenta(G):
    used = set()
```

```
    for v in G.keys():
        if v not in used:
            stack = [v]
            componenta = dfs(G, v, used, stack)
            if componenta != []:
                return componenta
    return componenta

G, N, M = read_graph()
componenta = ifcomponenta(G)
if componenta != []:
    print(*componenta)
else:
    print('NO')
```

13 Выделение компонент связности обходом в ширину

Поиск в ширину (обход в ширину, breadth-first search) — это один из основных алгоритмов на графах. В результате поиска в ширину находится путь кратчайшей длины в невзвешенном графе, т.е. путь, содержащий наименьшее число рёбер. Алгоритм работает за O(M+N) , где N — число вершин, M — число рёбер.

Описание алгоритма На вход алгоритма подаётся заданный граф (невзвешенный), и номер стартовой вершины . Граф может быть как ориентированным, так и неориентированным, для алгоритма это не важно.

Сам алгоритм можно понимать как процесс "поджигания"графа: на нулевом шаге поджигаем только вершину . На каждом следующем шаге огонь с каждой уже горящей вершины перекидывается на всех её соседей; т.е. за одну итерацию алгоритма происходит расширение "кольца огня"в ширину на единицу (отсюда и название алгоритма).

Более строго это можно представить следующим образом. Создадим очередь q, в которую будут помещаться горящие вершины, а также заведём булевский массив u = [], в котором для каждой вершины будем отмечать, горит она уже или нет (или иными словами, была ли она посещена).

Изначально в очередь помещается только вершина s , и used[s] = true, а для всех остальных вершин used = [false]. Затем алгоритм представляет собой цикл: пока очередь не пуста, достать из её головы одну вершину, просмотреть все рёбра, исходящие из этой вершины, и если какие-то из просмотренных вершин ещё не горят, то поджечь их и поместить в конец очереди.

В итоге, когда очередь опустеет, обход в ширину обойдёт все достижимые из s вершины, причём до каждой дойдёт кратчайшим путём. Также можно посчитать длины кратчайших путей (для чего просто надо завести массив длин путей d=[]), и компактно сохранить информацию, достаточную для восстановления всех этих кратчайших путей (для этого надо завести массив "предков"p[] , в котором для каждой вершины хранить номер вершины, по которой мы попали в эту вершину).

Поиск компонент связности в графе за O(M+N Для этого мы просто запускаем обход в ширину от каждой вершины, за исключением вершин, оставшихся посещёнными (used = true) после предыдущих запусков. Таким образом, мы выполняем обычный запуск в ширину от каждой вершины, но не обнуляем каждый раз массив used[], за счёт чего мы каждый раз будем обходить новую компоненту связности, а суммарное время работы алгоритма составит по-прежнему O(M+N) (такие несколько запусков обхода на графе без обнуления массива used называются серией обходов в ширину

Реализация Считаем, что G, N, M уже считаны

```
from queue import Queue
N = 0
def wfs(G, vertex, used = None):
    N = 0
    if used is None:
        used = set()
    q = Queue() # созданиеочереди
    q.put(vertex)
    while q.empty(): # циклпокаочередьНЕпустая
        v = q.get() # беремвершину
        if v not in used:
            N += 1
            used.add(v)
            print('{} is {}-th vertex.format(v, N)')
            for neig in G[v]:
                if neig not in used:
                    q.put(neig)

G = read_graph('graph.txt')
used = set()
for vertex in G.keys():
    if vertex not in used:
        wfs(G, vertex, used)
        N += 1
print(N)
```

14 Нахождение кратчайшего цикла в невзвешенном графе

Нахождение кратчайшего цикла в ориентированном невзвешенном графе: производим поиск в ширину из каждой вершины; как только в процессе обхода мы пытаемся пойти из текущей вершины по какому-то ребру в уже посещённую вершину, то это означает, что мы нашли кратчайший цикл, и останавливаем обход в ширину; среди всех таких найденных циклов (по одному от каждого запуска обхода) выбираем кратчайший.

Реализация

```
from queue import Queue

def read_graph():
    N, M = map(int, input().split())
    G = {}
    for i in range(M):
        v1, v2 = input().split()
        vertex = v1
        for v in v1, v2:
            if v not in G:
                G[v] = set()
        G[v1].add(v2)
    return G, vertex

def bfs_path(G, start, loops):
    path = {v: None for v in G}
    q = Queue()
    q.put(start)
    path[start] = [start]
    while not q.empty():
        v = q.get()
        used.add(v)
        for neig in G[v]:
            if path[neig] is None:
                q.put(neig)
                path[neig] = path[v] + [neig]
            elif neig in path[v]:
                loops.append(path[v][path[v].index(neig)::])
    return []

G, vertex = read_graph()
loops = []
used = set()
for v in G.keys():
    if v not in used:
        bfs_path(G, v, loops)
min = float('+inf')
for el in loops:
    if len(el) < min and el != []:
        min = len(el)
        min_loop = el
if min == float('+inf'):
    print('NO_CYCLES')
else:
    print(*min_loop)
```

15 Алгоритм Дейкстры

Снова вернемся к задаче поиска кратчайшего растояния от одной вершины до всех остальных, но теперь во взвешенном графе. Для ее решения будем применять алгоритм Дейкстры, который работает следующим образом:

1. На каждой итерации алгоритм среди непомеченных вершин вибирает с наименьшим до нее расстоянием;
2. Помечает вершину как посещенную.
3. Пытается улучшить расстояние до смежных с ней вершин;
4. Пытается улучшить расстояние до смежных с ней вершин;

На каждой итерации поддерживается инвариант, что расстояния до помеченных вершин являются кратчайшими и более меняться не будут. Однако, чтобы это условие не нарушалось, граф не должен содержать ребер отрицательного веса. Иначе, алгоритм в такой задаче не применим.

Наивная реализация Все нужные данные считаем считанными

```
def deikstra_1(G, start):
    distance = {v: float('+inf') for v in G}
    q = Queue()
    q.put(start)
    distance[start] = 0
    while not q.empty():
        v = q.get()
        for neighbor in G[v]:
            tmp = distance[v] + G[v][neighbor]
            if distance[neighbor] > tmp:
                q.put(neighbor)
                distance[neighbor] = tmp
    return distance
```

Реализация Кучей

```
def deikstra_2(G, start):
    heap = []
    distance = {v: float('+inf') for v in G}
    distance[start] = 0
    heapq.heappush(heap, (0, start))

    while len(heap) > 0:
        v = heapq.heappop(heap)[1]
        for neighbor in G[v]:
            tmp = distance[v] + G[v][neighbor]
            if distance[neighbor] > tmp:
                heapq.heappush(heap, (tmp, neighbor))
                distance[neighbor] = tmp
```

```
    return distance

G = create(M, N)
print(G)
dist1 = deikstra_1(G, 'A')
dist2 = deikstra_2(G, 'A')
for i in G:
    print(dist1[i], dist2[i], dist1[i] - dist2[i], sep='\t')
```

16 Алгоритм Флойда-Уоршелла.

Алгоритм Флойда — Уоршелла — алгоритм для нахождения кратчайших расстояний между всеми вершинами взвешенного графа без циклов с отрицательными весами с использованием метода динамического программирования.

Сам по себе алгоритм достаточно простой. Он перебирает все возможные тройки вершин i, j, k. Если существуют пути из i в k и из k в j, то алгоритм пытается улучшить путь из i в j, проложив его через k, т.е. d(i, j) = min(d(i, j), d(i, k) + d(k, j). Уже из описания алгоритма понятно, что его асимптотика O(N3).

Перебор k происходит во внешнем цикле. После каждой итерации этого цикла известны кратчашие расстояния между всеми парами вершин, проходящие только через вершины из множества 1 . . . k (не считая начальную и конечную вершины).

Тогда после выполнения всех итераций внешнего цикла будут звестны кратчашие расстояния между всеми парами вершин, проходящие только через вершины из множества V = 1 . . . N.

Реализация Изначально массив d выглядит следующим образом: • на главной диагонали стоят нули; • d[i][j] = w[i][j] если есть ребро (если есть кратные ребра, выбирайте наименьшее); • d[i][j] = +, иначе.

```
if __name__ == "__main__":
# d initialized from input
for k in range(n):
for i in range(n):
for j in range(n):
d[i][j] = min(d[i][j], d[i][k] + d[k][j])
print(d)
```

Если в графе есть циклы отрицательного веса, то формально алгоритм Флойда-Уоршелла неприменим к такому графу.

На самом же деле, для тех пар вершин i и j, между которыми нельзя зайти в цикл отрицательного вес, алгоритм отработает корректно. Для тех же пар вершин, ответа для которых не существует (по причине наличия отрицательного цикла на пути между ними), алгоритм Флойда найдёт в качестве ответа какое-то число (возможно, сильно отрицательное, но не обязательно).

Тем не менее, можно улучшить алгоритм Флойда, чтобы он аккуратно обрабатывал такие пары вершин и выводил для них, например, . Для этого можно сделать, например, следующий критерий "не существования пути". Итак, пусть на данном

графе отработал обычный алгоритм Флойда. Тогда между вершинами i и j не существует кратчайшего пути тогда и только тогда, когда найдётся такая вершина k, достижимая из i и из которой достижима j, для которой выполняется d[k][k] < 0.

```
if __name__ == "__main__":
# d initialized from input
for k in range(n):
for i in range(n):
for j in range(n):
d[i][j] = min(d[i][j], d[i][k] + d[k][j])
for k in range(n):
for i in range(n):
for j in range(n):
if d[i][k] < float("inf") and d[k][k] < 0 and \
d[k][j] < float("inf"):
d[i][j] = float("-inf");
print(d)
```

17 Алгоритм Беллмана-Форда

Состояния описываются двумя параметрами (i, j) и означают "длину кратчайшего пути, проходящего не более, чем по i ребрам, и заканчивающегося в вершине j". Алгоритм реализуется двумя вложенными циклами. Первый цикл перебирает значения первого параметра, т.е. количество ребер в кратчайших расстояниях. Одну итерацию этого цикла назовем фазой алгоритма. i-ая фаза алгоритма выглядит следующим образом:

1. перебираем все ребра (u, v) в графе (считаем, что ребра ориентированы);
2. пытаемся улучшить расстояние d[i][v] до вершины v, сравнив с d[i 1][u] + w(u, v).

Неизвестным остается количество фаз алгоритма Сам алгоритм имеет несколько особенностей:

1. алгоритм работает корректно даже при наличии ребер отрицательного веса, 1 – валидное значение для расстояний, поэтому массив d инициализировать 1 нельзя;
2. алгоритм работает корректно при наличии циклов отрицательного веса (сумма весов ребер цикла отрицательна) в и позволяет искать их.

   Оценка количества фаз

   Теорема 2. Любой кратчайший путь в графе с ребрами положительного веса не содержит циклов.

   Следствие 2.1. Кратчайший путь между двумя вершинами содержит не более N 1 ребер, где N – количество вершин в графе. **Теорема 3**. После выполнения i фаз алгоритм Форда-Беллмана корректно находит все кратчайшие пути, длина которых (по числу рёбер) не превосходит i.

```
from queue import Queue
from random import randint
import heapq

def read_graph(m):
    G = {}
    edges = []
    for i in range(m):
        v1, v2, ves = map(int, input().split())
        for v in v1, v2:
            if v not in G:
                G[v] = {}
        G[v1][v2] = ves
        edges.append([v1, v2])
    return G, edges

def Belman_Ford(N, G, edges):
    d = [float('+inf') for _ in range(N)]
    d[start] = 0
    for i in range(N):
        for x, y in edges:
            d[y] = min(d[y], d[x] + G[x][y])
    return d

n, m, start = map(int, input().split())
G, edges = read_graph(m)
dist = Belman_Ford(n, G, edges)
d = dist[:]
for x, y in edges:
    d[y] = min(d[y], d[x] + G[x][y])
for x, y in edges:
    d[y] = min(d[y], d[x] + G[x][y])
for i in range(n):
    if d[i] != dist[i]:
        dist[i] = float('+inf')
for el in dist:
    if el == float('+inf'):
        print('UDF', end='␣')
    else:
        print(el, end='␣')
```

18 Остовные деревья. Алгоритм Прима (наивная реализация)

Остовное дерево — ациклический связный подграф данного связного неориентированного графа, в который входят все его вершины.

Алгоритм Прима — алгоритм построения минимального остовного дерева взвешенного связного неориентированного графа.

На вход алгоритма подаётся связный неориентированный граф. Для каждого ребра задаётся его стоимость.

Сначала берётся произвольная вершина и находится ребро, инцидентное данной вершине и обладающее наименьшей стоимостью. Найденное ребро и соединяемые им две вершины образуют дерево. Затем, рассматриваются рёбра графа, один конец которых — уже принадлежащая дереву вершина, а другой — нет; из этих рёбер выбирается ребро наименьшей стоимости. Выбираемое на каждом шаге ребро присоединяется к дереву. Рост дерева происходит до тех пор, пока не будут исчерпаны все вершины исходного графа.

Результатом работы алгоритма является остовное дерево минимальной стоимости.

```
def search_min(tr, vizited):#1 местодляоптимизации
    min=max(tr)
    for ind in vizited:
        for index, elem in enumerate(tr[ind]):
            if elem>0 and elem<min and index not in vizited:
                min=elem #веса путей
                index2=index # индексвершины
    return [min, index2]

def prim(matr):
    toVisit=[i for i in range(1,len(matr))] #кроме начального(0)
    vizited=[0]
    result=[0] # начинаемслюбой
    for index in toVisit:
        weight, ind=search_min(matr, vizited)
        result.append(weight) #в результатбудутзаноситьсявеса
        vizited.append(ind) # содержиткартупути
    return result
```

19 Остовные деревья. Алгоритм Краскала. Система непересекающихся множеств для оптимизации алгоритма.

Алгоритм Краскала является типичным жадным алгоритмом. Он состоит из следующих шагов:

1. Сначала отсортируем ребра в порядке возрастания их весов
2. Каждая вершина изначально находится в своем множестве.
3. На каждом шаге мы выбираем ребро наименьшего веса, концы которого находятся в разных множествах.
4. Объединяем эти множества.

В конце мы получим N 1 ребро, которые объединят все вершины в одно множество. Данные ребра будут образовывать минимальный остов. Доказательство этого алгоритма тут не приводится, т.к. это потребует введение ряда определений и теорем.

Особенность этого алгоритма в том, что он не требует модификации для построения минимального остовного леса. Так как на каждой итерации алгоритм просто объединяет два дерева, то в случае несвязного графа просто останется несколько необъединенных деревьев. Каждое дерево будет минимальным остовом в своей компоненте связности.

Наивная реализация

```
def get_key(x):
    return x[2]

if __name__ == "__main__":
    # a - edge list
    a.sort(key=get_key)
    tree_id = [i for i in range(n)]
    edges = []
    weight = 0
    for u, v, w in a:
        if tree_id[u] != tree_id[v]:
            weight += w
            edges.append((u, v, w))
            j = tree_id[v]
            for i in range(n):
                if tree_id[i] == j:
                tree_id[i] = tree_id[u]
print(weight)
print(*edges, sep="\n")
```

Реализация с помощью СМН

Как вы могли заметить, операция объединения достаточно долгая. Ключ к оптимизации алгоритма лежит в оптимизации объединения. Для этого можно использовать систему непересекающихся множеств (СНМ) с эвристиками. Здесь не будет приводиться описание данной структуры.

Основное, что надо помнить, это то, что время работы операций в такой структуре является почти константным и может быть опущено. Используя СНМ, мы сможем объединять два множество за почти константное время. Время, необходимое на проверку, что две вершины принадлежат разным множествам, увеличится с O(1) до почти константного, но как было сказано ранее, мы это не учитываем. Итого, время работы будет O(M log M + N) = O(M log M).

```
def get_key(x):
    return x[2]

def make_set(x):
    return {
       "parent": x,
        "rank": 0
    }

def find_set(x):
    if dsu[x]["parent"] == x:
        return x
    dsu[x]["parent"] = find_set(dsu[x]["parent"])
        return dsu[x]["parent"]

def union_sets(x, y):
    x = find_set(x)
    y = find_set(y)
    if x != y:
        if dsu[x]["rank"] < dsu[y]["rank"]:
            x, y = y, x
        dsu[y]["parent"] = x
        if dsu[x]["rank"] == dsu[y]["rank"]:
            dsu[x]["rank"] += 1

if __name__ == "__main__":
    # a - edge list
    a.sort(key=get_key)
    dsu = [make_set(i) for i in range(n)]
    edges = []
    weight = 0
    for u, v, w in a:
        if find_set(u) != find_set(v):
            weight += w
            edges.append((u, v, w))
            union_sets(u, v)
    print(weight)
    print(*edges, sep="\n")
```

20 Игра на ациклических графах

Игра на графе - пара A = <G, u>, где G - ориентированный граф, а u - начальная вершина этого графа. Вершины графа G - состояния в игре, дуги - ходы.

Свойства игры:

1. **Последовательная** - игроки ходят по очереди

2. **С полной информацией** - есть информация о текущем положении и возможных ходах соперника
3. **Равноправна** - набор возможных ходов зависит только от состояния игры, то есть:
   (a) Два игрока
   (b) Победитель определён, когда нет возможных ходов
   (c) Конечное число состояний и переходов
   (d) Ходы доступны обоим игрокам
   (e) Нет вероятностей

Считается, что игроки рациональны.

Состояния вершин:

1. Если из некоторой вершины есть ребро в проигрышную вершину, то эта вершина **выигрышная**
2. Если из некоторой вершины нет исходящих ребер или все ребра исходят в выигрышные вершины, то эта вершина **проигрышная**
3. Если в какой-то момент ещё остались неопределённые вершины, но они не относятся ни к первому, ни к второму, то эти вершины - **ничейные**

В играх на ациклических графах нет ничейных вершин. Решаются при помощи обхода в глубину.

21 Сумма игр. Функция Шпрага-Гранди

XOR сумма набора чисел представляет собой выполнение операции XOR над ими всеми.

Задача: Даны N игр, за один ход можно сходить только в одной из игр. Проигрыш наступает, когда игрок не может сделать ход ни в одной из игр.

Игра Ним: Есть несколько кучек, в каждой из которых несколько камней. За один ход игрок может взять из какой-нибудь кучки любое ненулевое число камней и выбросить их.

Решение: Текущий игрок имеет выигрышную стратегию тогда и только тогда, когда XOR-сумма размеров кучек отлична от нуля. В противном случае - проигрышное состояние. (Доказательство по индукции)

Следствие: Любое состояние игры Ним можно заменить эквивалентным состоянием, состоящим из единственной кучки размера, равного XOR-сумме размеров кучек в старом состоянии.

Функция Шпрага-Гранди - $y = mex\{x_1, ..., x_k\}$, где y - значение функции для текущего состояния u, x_i - значение функции из состояния, куда можно попасть из u за один ход, mex возвращает минимальное целое неотрицательное число, которого нет в указанном множестве.

Теорема Шпрага-Гранди об (эквивалентности игры ниму): Рассмотрим любое состояние u некоторой равноправной игры двух игроков. Пусть из него есть переходы в некоторые состояния v_i, $i = \overrightarrow{1,k}$, k >= 0. Утверждается, что состоянию u этой игры можно поставить в соответствие кучку Нима некоторого размера x, которая будет полностью описывать состояние нашей игры. Это число x называется **значением Шпрага-Гранди состояния u** и вычисляется при помощи функции Шпрага-Гранди.

```
def dfs(u):
    vals = set()
    for v in a[u]:
        if d[v] == -1:
            dfs(v)
        vals.add(d[v])
    vals = sorted(vals)
    if vals[0] != 0:
        d[u] = 0
    else:
        for i in range(1, len(vals)):
            if vals[i] != vals[i-1] + 1:
                d[u] = vals[i-1] + 1
                break
        else:
            d[u] = vals[-1] + 1


if __name__ == "__main__":
    n, m = map(int, input().split())
    a = {i: set() for i in range(n)} #считываниеграфавсписоксмежности
    for _ in range(m):
        u, v = map(int, input().split())
        a[u].add(v)
    d = [-1] * n #массивзначенийфункцииШГ  -
    d[0] = 0 #для самойпоследнейвершиныуже      0
    dfs(n-1)
    print(d[::-1])
```

22 Игры на произвольных графах

1. Возможны ничейные вершины
2. Обычный dfs и функция Шпрага-Гранди не работают
3. Решение: ретроспективный анализ

Алгоритм:

1. Помечаем все вершины как неизвестные, кроме конечных

2. Строим обратный граф G'
3. Запускаем модифицированный dfs из начальных вершин в G'
4. Игнорируем вершины с известным ответом
5. Если текущая вершина проигрышная, то dfs помечает следующую вершину как проигрышную и идёт в неё
6. Если текущая вершина проигрышная, то увеличиваем счётчик в следующей вершине
7. Если счётчик равен полустепени захода вершины в графе G', то помечаем эту вершину как проигрышную и идём в неё
8. В конце все неизвестные вершины становятся ничейными

```
def dfs(u):
    for v in a[u]:
        if d[v] != "D":
            continue
        if d[u] == "L":
            d[v] = "W"
            dfs(v)
        else:
            counter[v] += 1
            if counter[v] == deg_in[v]:
                d[v] = "L"
                dfs(v)


if __name__ == "__main__":
    n, m = map(int, input().split())
    a = {i: set() for i in range(n)}
    deg_in = [0] * n
    for _ in range(m):
        u, v = map(int, input().split())
        a[v].add(u)
        deg_in[u] += 1
    d = ["D"] * n
    for i in range(n):
        if not deg_in[i]:
            d[i] = "L"
    counter = [0] * n
    for i in range(n):
        if not deg_in[i]:
            dfs(i)
    print(d)
```