

## 1 10.1

Условие: Дан пустой граф на  $n$  вершин. Надо уметь за обратную функцию Аккермана

- добавлять ребро в граф
- находить число ребер в компоненте связности, где лежит  $x$

Решение: будем использовать систему непересекающихся подмножеств (СМН), будем поддерживать 4 операции:

- `find` (н-ти представителя)
- `union` (объединить две вершины)
- `count_edges` (посчитать кол-во рёбер в множестве, где лежит текущая вершина)
- `add_edge` (создать ребро)

Описание структуры данных:

- массив `parents` (для каждого элемента ссылка на его родителя) Изначально каждый элемент является своим родителем. Используем сжатие путей.
- массив `ranks` (для каждого корня хранится глубина смн). Изначально все ранги = 0.
- массив `edges` (для каждого корня хранится кол-во ребер в смн) Изначально у всех элементов по 0 ребер в их деревьях.

Функции:

- `find(v)`

```
1 def find(v):
2     if parents[v] == v:
3         return v
4     parents[v] = find(parents[v])
5     return parents[v]
```

- `union(a,b)`

```
1 def union(a,b):
2     a = find(a)
3     b = find(b)
4     if ranks[a] > ranks[b]:
5         swap(a,b)
6     if ranks[a] == ranks[b]:
7         ranks[b]++
8     if a != b:
9         parents[a] = b
10        edges[b] = edges[a]
11    edges[b] += 1
```

- `count_edges(v)`

```
1 def count_edges(v):
2     v = find(v)
3     return edges[v]
```

#### 4. add\_edge

```
1 def add_edge(a,b):
2     union(a,b)
```

Доказательство сложности обратной функции Аккермана: в функциях union и count\_edges используется find и остальные операции за  $O(1)$ . Тогда достаточно доказать, что find работает за Аккермана. Это верно, так как используется сжатие путей.

Это работает, так как мы храним количество ребер не зависимо от того, как связаны элементы со своим корнем. То есть достигается и сложность обратной функции Аккермана, и возможность валидно хранить количество ребер в каждой компоненте.

## 2 10.2

Условие: необходимо поддерживать структуру с массивом из нулей и единиц, в которой все операции, описанные ниже будут работать за обратную от функции Аккермана.

- присвоить 1 i-му элементу
- найти ближайший ноль к i-му элементу массива

Структура данных:

1. массив a, хранит нули и единицы
2. массив rank - такой же как в номере 10.1
3. массив parents хранит подмассив для каждого элемента, где
  - родитель на 0 индексе
  - ближайший ноль слева на 1 индексе
  - ближайший ноль справа на 2 индексе

Изначально хранит, что каждый элемент сам себе родитель и ближайший ноль слева, справа это сам элемент. Затем будем объединять в одну компоненту непрерывные последовательности из 1. Функция find такая же, как в номере 10.1 (она будет работать за обратную функцию Аккермана)

1. init\_one(i) Данная функция будет присваивать единицу и при необходимости объединять эту единицу в соседние компоненты.

```
1 def init_one(i):
2     a[i] = 1
3     #union with left
4     if a[i - 1] == 1:
5         left = find(i - 1)
6         parents[i][0] = left
7         parents[left][2] = i
8         if ranks[left] == ranks[i]: ranks[left]++ #two lonely "1" merged
9     else:
10        parents[i][1] = i - 1
11    #union with right
12    cur = find(i) #not just i, because we could merge with left on step
        before
13    if a[i + 1] == 1:
14        right = find(i + 1)
15        if ranks[cur] > ranks[right]:
```

```

16         swap(cur, right)
17         parents[cur][0] = right
18         parents[right][1] = parents[cur][1] #give cur's left zero to
           right's left zero
19     else:
20         parents[cur][2] = i + 1

```

Очевидно как сделать обработку граничных положений - сделать нули слева, справа от массива `a` и в конце `union` заифать на выход за пределы длины `a`. Не стал писать, ибо код и так получился довольно громоздким.

## 2. `closest_zero(i)`

```

1 def closest_zero(i):
2     cur = find(i)
3     if abs(parents[cur][1] - i) < abs(parents[cur][2] - i):
4         return parents[cur][1]
5     else:
6         return parents[cur][2]

```

Это работает, потому что всегда сохраняется инвариант - для каждой компоненты знаем ближайший ноль слева, справа

## 3 10.3

Условие: на пустом графе надо уметь добавлять ребра, находить число компонент связности, являющихся деревьями.

Структура данных:

1. массив `parents` вместе с функцией `find` такие же как и в 1 задании
2. массив `size` (для каждого корня хранится кол-во элементов в его компоненте)  
изначально все = 1 (т.к. один элемент есть)  
массив `edges` (для каждого корня хранятся кол-во ребер в его компоненте)  
изначально все = 0 (т.к. пока что нет ребер)  
массив `isTree` (для каждого корня хранятся является ли он деревом)  
изначально все = 1 (т.к. один элемент - это дерево)

Функции:

### 1. `union(a,b)`

```

1 def union(a,b):
2     a = find(a)
3     b = find(b)
4     if ranks[a] > ranks[b]:
5         swap(a,b)
6     if ranks[a] == ranks[b]:
7         ranks[b]++
8     if a != b:
9         parents[a] = b
10        edges[b] = edges[a]
11        size[b] += size[a]
12    edges[b] += 1
13    if (isTree[a] and isTree[b] and size[b] - edges[b] == 1):

```

```

14         isTree[b] = 1
15     else:
16         isTree[b] = 0

```

## 2. count\_trees()

```

1 def count_trees():
2     cnt = 0
3     used = []
4     for i in range(len(parents)):
5         parent = find(i)
6         if i == parent and parent not in used:
7             if isTree[i] == 1:
8                 cnt++
9             used.add(parent)
10    return cnt

```

## 3. add\_edge(v), find(v) (аналогично как в пункте 10.1)

Модифицировали union, чтобы обновлять информацию о том, является ли компонента деревом. Это позволило сделать функцию count\_trees, подсчитывающую число компонент связности, являющихся деревьями

# 4 10.4

Условие: на пустом графе необходимо уметь выполнять запросы двух типов (вермя работы - обратная ф-я Аккермана)

- добавить ребро в граф
- найти кол-во компонент связности, являющихся циклами

Идея аналогичная той, что в номере 10.3. Только здесь нужно проверять не на деревья, а на циклы

Структура данных:

1. массив parents
2. массив size хранит размер каждой компоненты
3. массив edge хранит кол-во ребер в компонентах
4. массив isCycle хранит true/false
5. переменная cnt хранит ответ на вопрос о кол-ве циклов

Функции:

1. find(v) - со сжатием путей
2. union(a,b)

```

1 def union(a,b):
2     root_a = find(a)
3     root_b = find(b)
4     if size[root_a] > size[root_b]:
5         swap(a,b)
6     parents[root_a] = root_b

```

```

7     size[root_b] += size[root_a]
8     edge[root_b] += edge[root_a] + 1
9
10    # checking for creating cycle
11    if size[root_b] == edge[root_b]:
12        isCycle[root_b] = true
13        cnt++
14    else:
15        isCycle[root_b] = false
16        cnt -= 1

```

### 3. count\_cycles()

```

1 def count_cycles():
2     return cnt

```

## 5 10.5

Условие: требуется добавить в СНМ 2 операции:

- Увеличить веса всех элементов, находящихся в одном множестве с элементом  $x$ , на  $d$ .
- Найти вес элемента  $x$ .

Чтобы уметь добавлять к мн-ву число, не будем использовать сжатие путей (так как в условии не просят сделать сложность как у обратной функции Аккермана)

Структура данных:

1. Пусть  $w$  - массив с весами
2. массив  $parents$  - ссылки на родителей

Функции:

#### 1. find(v)

```

1 def find(v):
2     cur = parents[v]
3     while cur != v:
4         return find(cur)
5     return v

```

#### 2. union(a,b)

```

1 def union(a,b):
2     parents[a] = b
3     w[a] = w[a] - w[b]

```

#### 3. increase(v, d)

```

1 def increase(v, d):
2     root = find(v)
3     w[root] += d

```

#### 4. count\_weight(v) - пробегаемся от элемента до корня и аккумулируем искомую сумму

```
1 def count_weight(v):  
2     accum = w[v]  
3     root = parents[v]  
4     while root != v:  
5         accum += w[root]  
6         root = parents[root]  
7     return accum
```