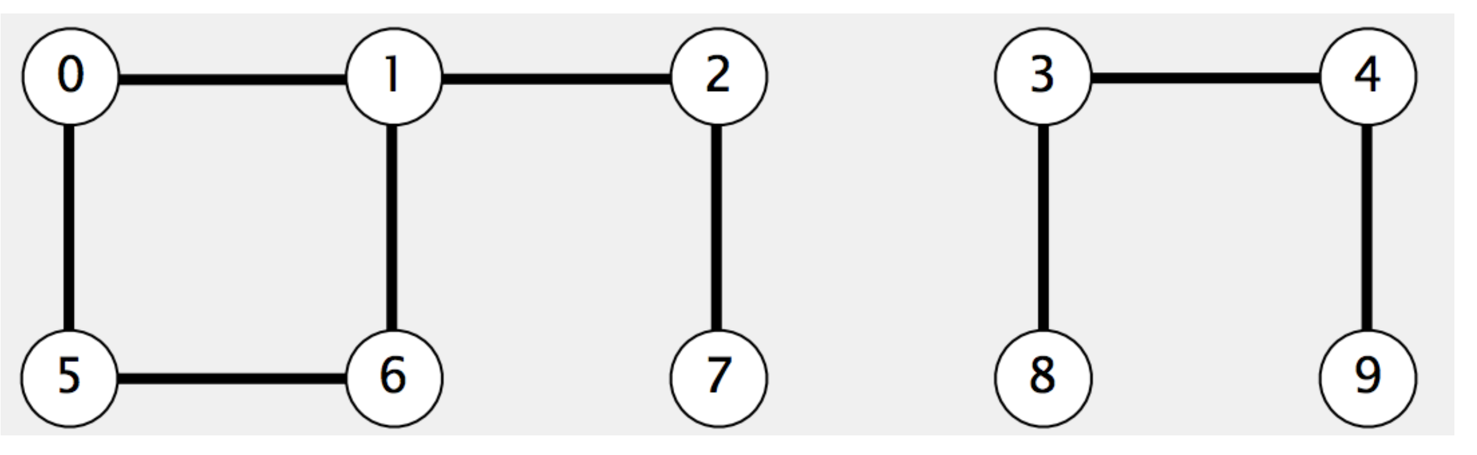
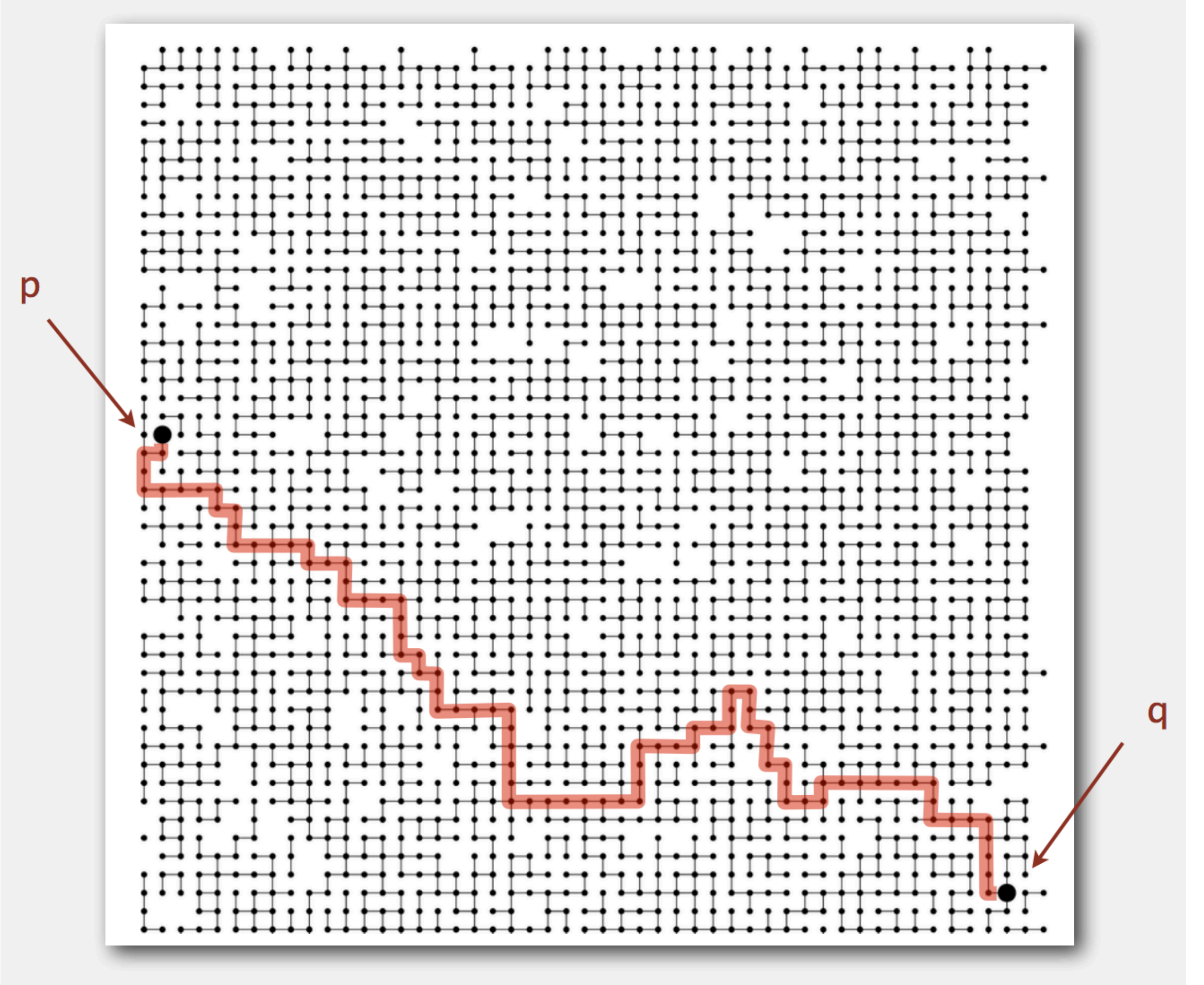
算法思考的步骤

1. 建立问题模型
2. 找到合适的算法解决问题
3. 是否运行快速？是否节省空间？
4. 如果不，找出问题的原因
5. 找出处理该问题的方法
6. 不断迭代直到自己满意

并查集(union find): 该数据结构所解决的问题是连接问题，而不是路径问题。只能够判断出两个点之间是否连接。



如图： 如果我们有一个union方法，该方法是将两个顶点进行连接
那么： 如果要实现上图的连接状况，应该分别执行了union(0, 1) 、union(1, 2)、union(2, 7)、union(0, 5)、union(5, 6)、union(1, 6)、union(2, 7)、union(3, 8)、union(3, 4)、union(4, 9)
试问：如果我们有一个connect方法，该方法是判断两个定点是否连接。connect(6, 7)是否连接，connect(7, 3)是否连接。
其实：由图中可以看出connect(6, 7)应返回true、connect(7, 3)应返回false。
所以：并查集就是解决该类问题的数据结构



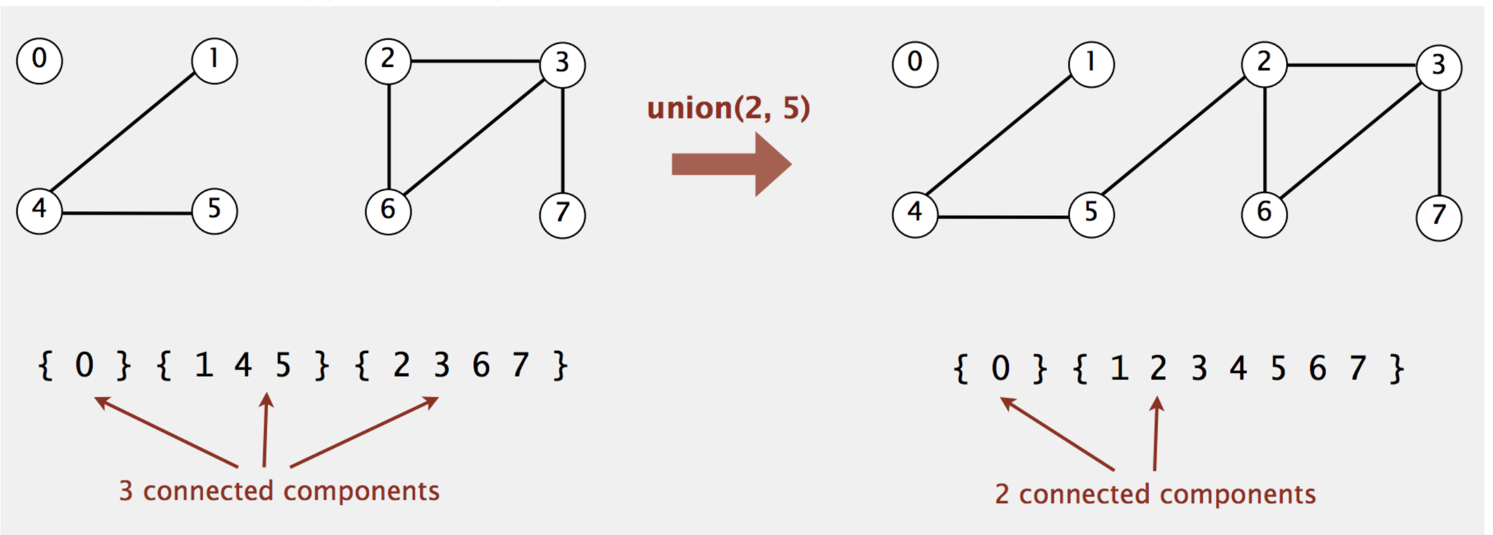
解决问题(实际)：

1. 数字图片中像素
2. 互联计算机
3. 社交
4. 电脑芯片晶管体
5. 数学集合

一、确保连接的等价性

假如：q 连接 p
那么：p 也连接 q
假如：q 连接 p，p 连接 f
那么：q 也连接 f

二、问题的根本性(quick Find)



我们使用集合的方式来代表不同的联通分量，那么总共有三个集合表示该图，那么当我们用union(2, 5)来连接参数中两个顶点，则整个的集合变成了两个

三、使用合适的数据类型以及api

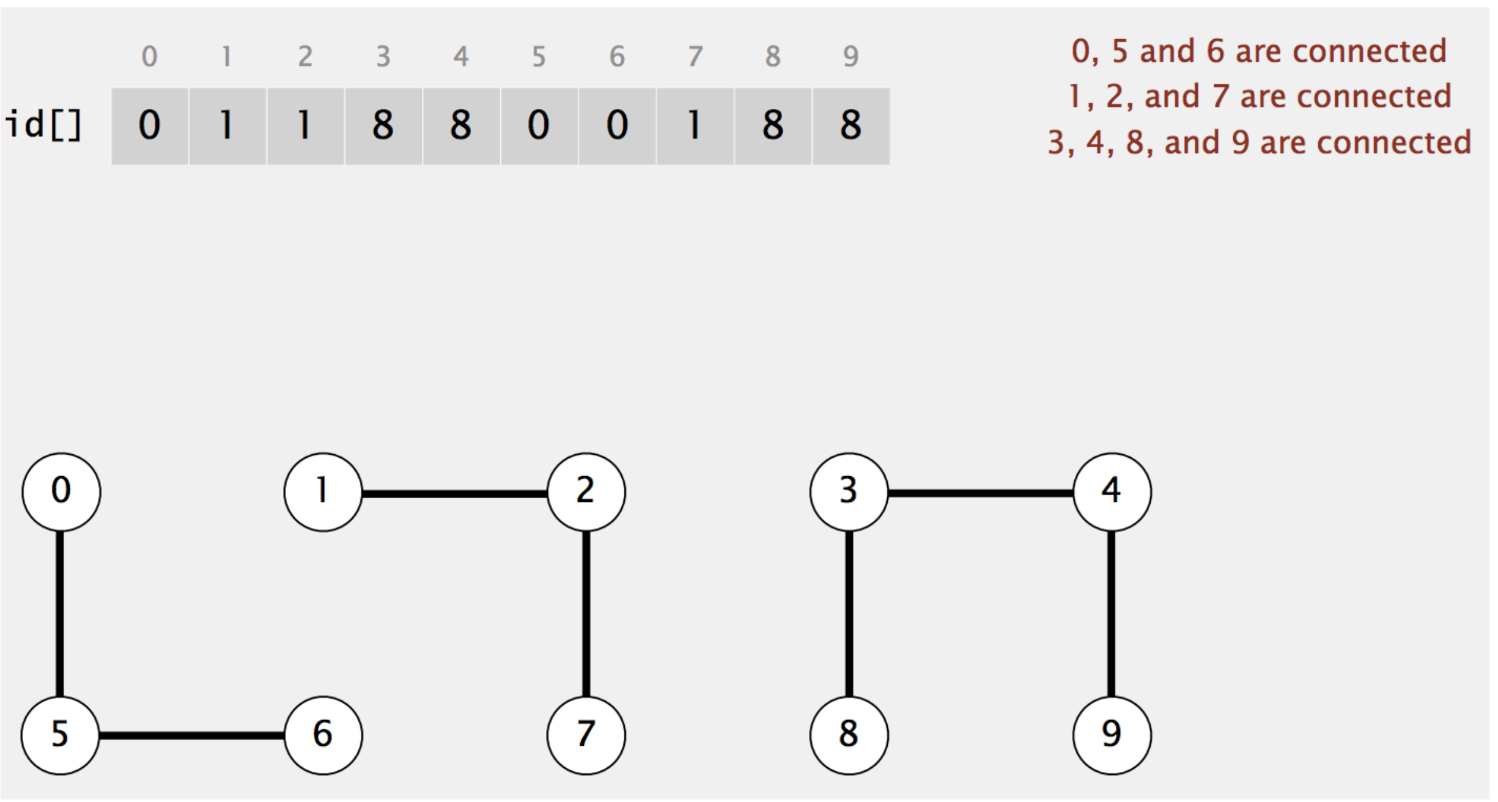
存储数据的类型为数组
api：{
  union: 用于连接两个顶点, union(参数1， 参数2) ,
  connent: 用于判断两个定点是否连接, connect(参数1， 参数2); 返回值为boolean
}

四、分析问题

1. 数据类型为数组(id[])，每个数组在初始化的时候要有不同的id表示，每一个值都是一个独立的连通分量

	0	1	2	3	4	5	6	7	8	9
id[]	0	1	2	3	4	5	6	7	8	9

2.当调用union方式时，将其参数中所在的连通分量集合中所有的顶点的id都替换成一个相同id，使其表示同一个连通分量

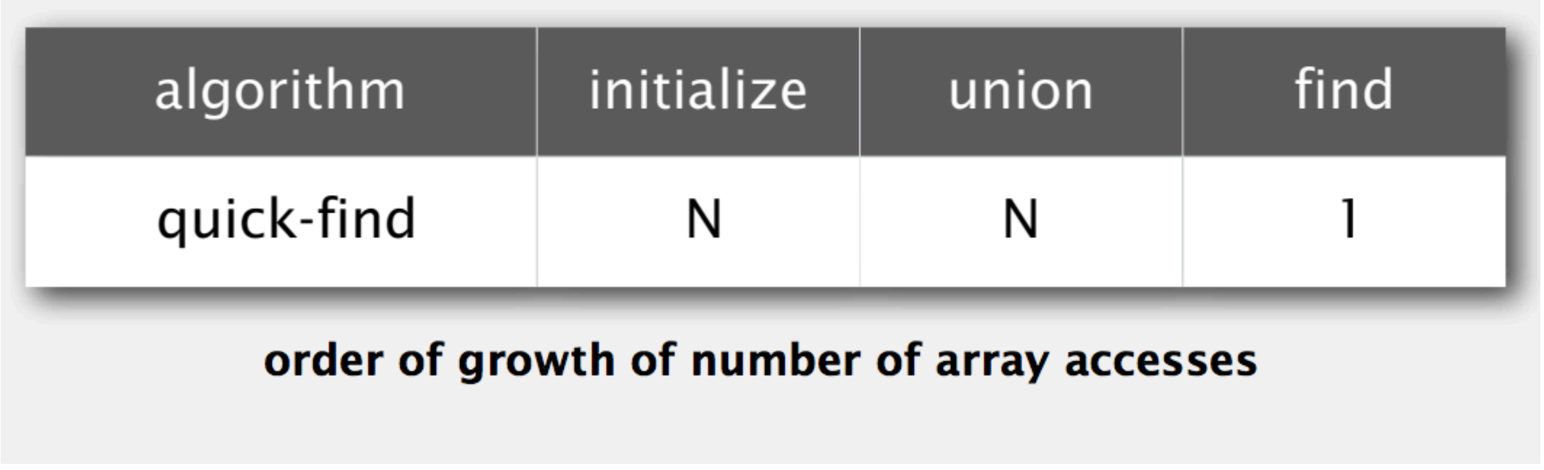


	0	1	2	3	4	5	6	7	8	9
id[]	0	1	1	8	8	0	0	1	8	8

五、代码实现

```
// 并查集主要是解决两点之间是否存在连接关系，而并非解决路径问题 (quick-find)
class UnionFind {
  constructor(n){
    this.ids = n; // 当前用户输入的顶点个数
    this.id = []; // 存储id数组
    this._uniqueId();
  };

  // 初始化顶点的id标识
  _uniqueId () {
    let ids = this.ids;
    let id = this.id;
    for(var i = 0; i < ids; i++) {
      id[i] = i;
    };
  };

  // 判断越界问题
  _assert ($id1, $id2) {
    let maxinum = this.ids;
    return ($id1 >= 0 && $id1 < maxinum) || ($id2 >= 0 && $id2 < maxinum);
  }

  // 顶点的连接性，结果boolean
  connect ($id1, $id2) {
    if(this._assert($id1, $id2)) {
      return
    }
    return this.id[$id1] === this.id[$id2];
  }

  // 连接顶点
  union ($id1, $id2) {
    if(!this._assert($id1, $id2)) {
      return
    };
    let id1 = this.id[$id1];
    let id2 = this.id[$id2];
    let ids = this.ids;
    for(var i = 0; i < ids; i++) {
      if(id2 == this.id[i]) {
        this.id[i] = id1;
      };
    };
  }
};
```

六、性能问题



algorithm	initialize	union	find
quick-find	N	N	1

order of growth of number of array accesses

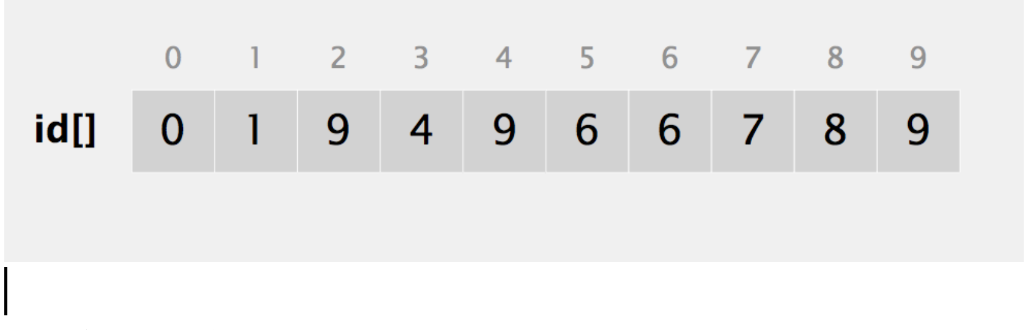
时间复杂度为o(n^2)

七、处理性能问题的原因

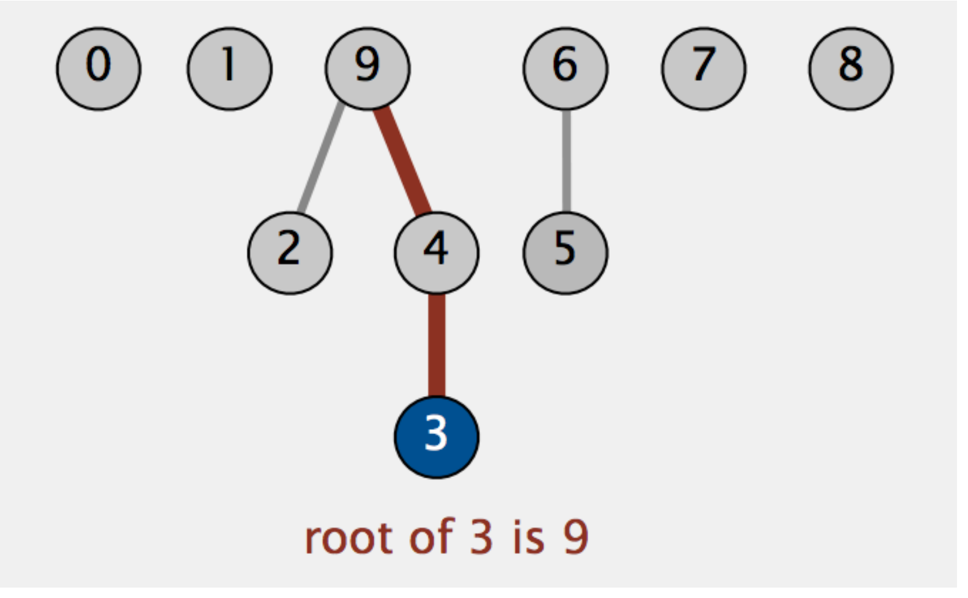
在每次调用union， 会重新遍历一次数组，并找到与之匹配的参数集合并赋值成另一个参数的id值

八、性能优化union方法 (quick union)

由于知道调用union所导致的性能问题，所以可以采用一个链式数据结构的表达关联关系，某一个结点只关心与其parent的关联性，不用考虑所有的结点



	0	1	2	3	4	5	6	7	8	9
id[]	0	1	9	4	9	6	6	7	8	9



二、解题思路

如何找到是否同一个根节点，就需要不断去遍历该结点的id与父节点的id是否相同，且已知根节点的id就是自身序号所以可以作为循环退出的条件。

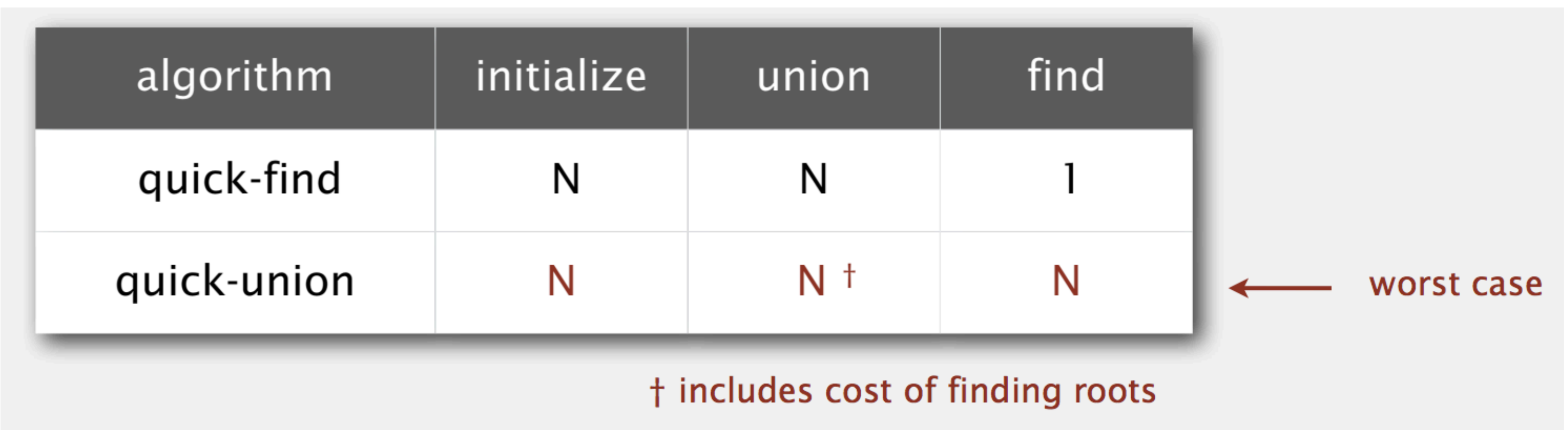
三、使用的数据类型以及api

数据类型: 使用数组的方式来表达这个结点关系的树形结构
api：{
  union: 用于连接两个顶点, union(参数1， 参数2) ,
  connent: 用于判断两个定点是否连接, connect(参数1， 参数2); 返回值为boolean
  _root: 用于找出某一结点的根节点 返回值为结点的id
}

四、代码实现

```
// 并查集主要是解决两点之间是否存在连接关系，而并非解决路径问题 quick-union
class UnionFind {
    constructor(n){
        this.ids = n; // 当前用户输入的顶点个数
        this.id = []; // 存储id数组
        this._uniqueId();
    };
    // 初始化顶点的id标识
    _uniqueId () {
        let ids = this.ids;
        let id = this.id;
        for(var i = 0; i < ids; i++) {
            id[i] = i;
        };
    };
    // 判断越界问题
    _assert ($id1, $id2) {
        let maxinum = this.ids;
        return ($id1 >= 0 && $id1 < maxinum) && ($id2 >= 0 && $id2 < maxinum);
    }
    // 找到根节点id
    _root ($id) {
        while(this.id[$id] != $id) {
            $id = this.id[$id]
         }
        return $id
    }
    // 顶点的连接性，结果boolean
    connect ($id1, $id2) {
        if(!this._assert($id1, $id2)) {
            return
        }
        return this._root($id1) == this._root($id2)
    }
    // 连接顶点
    union ($id1, $id2) {
    if(!this._assert($id1, $id2)) {
        return
    };
    let rid1 = this._root($id1)
    let rid2 = this._root($id2)
        if( rid1 != rid2) {
            this.id[$id2] = rid1
        }
    }
    // 打印关系日志
    print () {
        for(var i = 0; i < this.ids; i++) {
            console.log(i, this.id[i]);
        };
    };
};
```

五、性能分析



algorithm	initialize	union	find
quick-find	N	N	1
quick-union	N	N †	N

† includes cost of finding roots