四、并查集

用于解决动态连通性问题，能动态连接两个点，并且判断两个点是否连通。

方法	描述
UF(int N)	构造一个大小为 N 的并查集
void union(int p, int q)	连接 p 和 q 节点
int find(int p)	查找 p 所在的连通分量编号
boolean connected(int p, int q)	判断 p 和 q 节点是否连通

```
public abstract class UF {

    protected int[] id;

    public UF(int N) {
        id = new int[N];
        for (int i = 0; i < N; i++) {
            id[i] = i;
        }
    }

    public boolean connected(int p, int q) {
        return find(p) == find(q);
    }

    public abstract int find(int p);

    public abstract void union(int p, int q);
}
```

Quick Find

可以快速进行 find 操作，也就是可以快速判断两个节点是否连通。

需要保证同一连通分量的所有节点的 id 值相等。**id 数组用来表示节点所在的连通分量**

但是 union 操作代价却很高，需要将其中一个连通分量中的所有节点 id 值都修改为另一个节点的 id 值。

访问数组次数：判断是否联通需要 2 次读操作，union 获取联通分量需要2次读操作，遍历需要size 次读操作，修改需要 x 次写操作

size + 4 + x

```
public class QuickFindUF extends UF {

    public QuickFindUF(int N) {
        super(N);
    }
```

```
    @Override
    public int find(int p) {
        return id[p];
    }


    @Override
    public void union(int p, int q) {
        int pID = find(p);
        int qID = find(q);

        if (pID == qID) {
            return;
        }

        for (int i = 0; i < id.length; i++) {
            if (id[i] == pID) {
                id[i] = qID;
            }
        }
    }
}
```

Quick Union

可以快速进行 union 操作，只需要修改一个节点的 id 值即可。

但是 find 操作开销很大，因为同一个连通分量的节点 id 值不同，id 值只是用来指向另一个节点。因此需要一直向上查找操作，直到找到最上层的节点。 **id数组中记录同一个分量中的另一个节点名称，根节点连接指向自己**

2*两个待合并子树的深度和 + 1

```
public class QuickUnionUF extends UF {

    public QuickUnionUF(int N) {
        super(N);
    }


    @Override
    public int find(int p) {
        while (p != id[p]) {
            p = id[p];
        }
        return p;
    }


    @Override
    public void union(int p, int q) {
        int pRoot = find(p);
        int qRoot = find(q);
```

```
        if (pRoot != qRoot) {
            id[pRoot] = qRoot;
        }
    }
}
```

这种方法可以快速进行 union 操作，但是 find 操作和树高成正比，最坏的情况下树的高度为节点的数目。

加权 Quick Union

为了解决 quick-union 的树通常会很高的问题，加权 quick-union 在 union 操作时会让较小的树连接较大的树上面。

理论研究证明，加权 quick-union 算法构造的树深度最多不超过 logN。

2*两个待合并子树的深度和 + 4

```
public class WeightedQuickUnionUF extends UF {

    // 保存节点的数量信息
    private int[] sz;


    public WeightedQuickUnionUF(int N) {
        super(N);
        this.sz = new int[N];
        for (int i = 0; i < N; i++) {
            this.sz[i] = 1;
        }
    }


    @Override
    public int find(int p) {
        while (p != id[p]) {
            p = id[p];
        }
        return p;
    }


    @Override
    public void union(int p, int q) {

        int i = find(p);
        int j = find(q);

        if (i == j) return;

        if (sz[i] < sz[j]) {
            id[i] = j;
            sz[j] += sz[i];
        } else {
            id[j] = i;
```

```
            sz[i] += sz[j];
        }
    }
}
```

路径压缩的加权 Quick Union

在检查节点的同时将它们直接链接到根节点，只需要在 find 中添加一个循环即可。

```
while (id[p] != p){
    id[p] = id[id[p]];
    p = id[p];
}
```

比较

算法	union	find
Quick Find	N	1
Quick Union	树高	树高
加权 Quick Union	logN	logN
路径压缩的加权 Quick Union	非常接近 1	非常接近 1

算法的复杂度分析

对数图像：　(log-log plot)

对于一个算法，如果其运行时间和数据规模有如下关系：

$$T(N) = aN^b$$

对运行时间和数据规模都取对数，可以得到如下关系：

$$\lg(T(N)) = b \lg N + c$$
$$a = 2^c$$

图像的斜率为 b，

倍率实验：因此，如果 T(n) ~ an^b ，可以采用数据量加倍，测量运行时间，求对数斜率的方法对 ab 进行估算，但注意这种方法无法用来估计存在对数关系的计算复杂度。

- 时间
- 空间
- telde 表示 `~`，数组访问，比较次数

Tilde表示法

抹掉低阶项但系数保留

排序

- 插入排序
- 归并排序、分治法
- 快速排序、随机快送排序
- 堆排序
- 比较性能，以及特征
    - 稳定性不同
    - 原地（使用的额外空间）
- 元素比较的顺序细节问题，是否会产生这样的中间结果
- n^2 排序的其他算法：冒泡，选择，控制稳定性

选择排序

选择出数组中的最小元素，将它与数组的第一个元素交换位置。再从剩下的元素中选择出最小的元素，将它与数组的第二个元素交换位置。不断进行这样的操作，直到将整个数组排序。

选择排序需要 ~N^2^/2 次比较和 ~N 次交换，它的运行时间与输入无关，这个特点使得它**对一个已经排序的数组也需要这么多的比较和交换操作。**

```
public class Selection<T extends Comparable<T>> extends Sort<T> {

    @Override
    public void sort(T[] nums) {
        int N = nums.length;
        for (int i = 0; i < N - 1; i++) {
            int min = i;
            for (int j = i + 1; j < N; j++) {
                if (less(nums[j], nums[min])) {
                    min = j;
                }
            }
            swap(nums, i, min);
        }
    }
}
```

冒泡排序

从左到右不断交换相邻逆序的元素，在一轮的循环之后，可以让未排序的最大元素上浮到右侧。

在一轮循环中，如果没有发生交换，就说明数组已经是有序的，此时可以直接退出。

以下演示了在一轮循环中，将最大的元素 5 上浮到最右侧。

```
public class Bubble<T extends Comparable<T>> extends Sort<T> {
```

```
    @Override
    public void sort(T[] nums) {
        int N = nums.length;
        boolean hasSorted = false;
        for (int i = N - 1; i > 0 && !hasSorted; i--) {
            hasSorted = true;
            for (int j = 0; j < i; j++) {
                if (less(nums[j + 1], nums[j])) {
                    hasSorted = false;
                    swap(nums, j, j + 1);
                }
            }
        }
    }
}
```

插入排序

每次都将当前元素插入到左侧已经排序的数组中，使得插入之后左侧数组依然有序。

对于数组 {3, 5, 2, 4, 1}，它具有以下逆序：(3, 2), (3, 1), (5, 2), (5, 4), (5, 1), (2, 1), (4, 1)，插入排序每次只能交换相邻元素，令逆序数量减少 1，因此插入排序需要交换的次数为逆序数量。

插入排序的复杂度取决于数组的初始顺序，如果数组已经部分有序了，逆序较少，那么插入排序会很快。

- 平均情况下插入排序需要 ~N^2^/4 比较以及 ~N^2^/4 次交换；
- 最坏的情况下需要 ~N^2^/2 比较以及 ~N^2^/2 次交换，最坏的情况是数组是倒序的；
- 最好的情况下需要 N-1 次比较和 0 次交换，最好的情况就是数组已经有序了。

以下演示了在一轮循环中，将元素 2 插入到左侧已经排序的数组中。

```
public class Insertion<T extends Comparable<T>> extends Sort<T> {

    @Override
    public void sort(T[] nums) {
        int N = nums.length;
        for (int i = 1; i < N; i++) {
            for (int j = i; j > 0 && less(nums[j], nums[j - 1]); j--) {
                swap(nums, j, j - 1);
            }
        }
    }
}
```

希尔排序

对于大规模的数组，插入排序很慢，因为它只能交换相邻的元素，每次只能将逆序数量减少 1。

希尔排序的出现就是为了解决插入排序的这种局限性，它通过交换不相邻的元素，每次可以将逆序数量减少大于 1。

希尔排序使用插入排序对间隔 h 的序列进行排序。通过不断减小 h，最后令 h=1，就可以使得整个数组是有序的。

```
public class Shell<T extends Comparable<T>> extends Sort<T> {

    @Override
    public void sort(T[] nums) {

        int N = nums.length;
        int h = 1;

        while (h < N / 3) {
            h = 3 * h + 1; // 1, 4, 13, 40, ...
        }

        while (h >= 1) {
            for (int i = h; i < N; i++) {
                for (int j = i; j >= h && less(nums[j], nums[j - h]); j -= h) {
                    swap(nums, j, j - h);
                }
            }
            h = h / 3;
        }
    }
}
```

希尔排序的运行时间达不到平方级别，使用递增序列 1, 4, 13, 40, ... 的希尔排序所需要的比较次数不会超过 N 的若干倍乘于递增序列的长度。后面介绍的高级排序算法只会比希尔排序快两倍左右。

归并排序

归并排序的思想是将数组分成两部分，分别进行排序，然后归并起来。

1. 归并方法

归并方法将数组中两个已经排序的部分归并成一个。

```
public abstract class MergeSort<T extends Comparable<T>> extends Sort<T> {

    protected T[] aux;


    protected void merge(T[] nums, int l, int m, int h) {
        //将aux[lo,mid] 和 aux[m+1,h]归并
        int i = l, j = m + 1;

        for (int k = l; k <= h; k++) {
            aux[k] = nums[k]; // 将数据复制到辅助数组
        }

        for (int k = l; k <= h; k++) {
            if (i > m) { //左边用尽
                nums[k] = aux[j++];

            } else if (j > h) { // 右边用尽
                nums[k] = aux[i++];

            } else if (aux[i].compareTo(nums[j]) <= 0) { //左比右小
```

```
                nums[k] = aux[i++]; // 先进行这一步，保证稳定性

            } else { //左比右大
                nums[k] = aux[j++];
            }
        }
    }
}
```

2. 自顶向下归并排序

将一个大数组分成两个小数组去求解。

因为每次都将问题对半分成两个子问题，这种对半分的算法复杂度一般为 O(NlogN)。

```
public class Up2DownMergeSort<T extends Comparable<T>> extends MergeSort<T> {

    @Override
    public void sort(T[] nums) {
        aux = (T[]) new Comparable[nums.length];
        sort(nums, 0, nums.length - 1);
    }

    private void sort(T[] nums, int l, int h) {
        if (h <= l) {
            return;
        }
        int mid = l + (h - l) / 2;
        sort(nums, l, mid);
        sort(nums, mid + 1, h);
        merge(nums, l, mid, h);
    }
}
```

3. 自底向上归并排序

先归并那些微型数组，然后成对归并得到的微型数组。

```
public class Down2UpMergeSort<T extends Comparable<T>> extends MergeSort<T> {

    @Override
    public void sort(T[] nums) {

        int N = nums.length;
        aux = (T[]) new Comparable[N];

        for (int sz = 1; sz < N; sz += sz) { //每一趟归并的数组大小
            for (int lo = 0; lo < N - sz; lo += sz + sz) {
                merge(nums, lo, lo + sz - 1, Math.min(lo + sz + sz - 1, N - 1));
            }
        }
    }
}
```

快速排序

1. 基本算法

- 归并排序将数组分为两个子数组分别排序，并将有序的子数组归并使得整个数组排序；
- 快速排序通过一个切分元素将数组分为两个子数组，左子数组小于等于切分元素，右子数组大于等于切分元素，将这两个子数组排序也就将整个数组排序了。

```
public class QuickSort<T extends Comparable<T>> extends Sort<T> {

    @Override
    public void sort(T[] nums) {
        shuffle(nums);
        sort(nums, 0, nums.length - 1);
    }

    private void sort(T[] nums, int l, int h) {
        if (h <= l)
            return;
        int j = partition(nums, l, h);
        sort(nums, l, j - 1);
        sort(nums, j + 1, h);
    }

    private void shuffle(T[] nums) {
        List<Comparable> list = Arrays.asList(nums);
        Collections.shuffle(list);
        list.toArray(nums);
    }
}
```

2. 切分

取 a[l] 作为切分元素，然后从数组的左端向右扫描直到找到第一个大于等于它的元素，再从数组的右端向左扫描找到第一个小于它的元素，交换这两个元素。不断进行这个过程，就可以保证左指针 i 的左侧元素都不大于切分元素，右指针 j 的右侧元素都不小于切分元素。当两个指针相遇时，将切分元素 a[l] 和 a[j] 交换位置。

```
private int partition(T[] nums, int l, int h) {
    int i = l, j = h + 1;
    T v = nums[l];
    while (true) {
        while (less(nums[++i], v) && i != h) ;
        while (less(v, nums[--j]) && j != l) ;
        if (i >= j)
            break;
        swap(nums, i, j);
    }
    swap(nums, l, j);
    return j;
}
```

3. 性能分析

快速排序是原地排序，不需要辅助数组，但是递归调用需要辅助栈。

快速排序最好的情况下是每次都正好将数组对半分，这样递归调用次数才是最少的。这种情况下比较次数为 CN=2CN/2+N，复杂度为 O(NlogN)。

最坏的情况下，第一次从最小的元素切分，第二次从第二小的元素切分，如此这般。因此最坏的情况下需要比较 N2/2。为了防止数组最开始就是有序的，在进行快速排序时需要随机打乱数组。

4. 算法改进

4.1 切换到插入排序

因为快速排序在小数组中也会递归调用自己，对于小数组，插入排序比快速排序的性能更好，因此在小数组中可以切换到插入排序。

4.2 三数取中

最好的情况下是每次都能取数组的中位数作为切分元素，但是计算中位数的代价很高。一种折中方法是取 3 个元素，并将大小居中的元素作为切分元素。

4.3 三向切分

对于有大量重复元素的数组，可以将数组切分为三部分，分别对应小于、等于和大于切分元素。

三向切分快速排序对于有大量重复元素的随机数组可以在线性时间内完成排序。

```
public class ThreewayQuickSort<T extends Comparable<T>> extends QuickSort<T> {

    @Override
    protected void sort(T[] nums, int l, int h) {
        if (h <= l) {
            return;
        }
        int lt = l, i = l + 1, gt = h;
        T v = nums[l];
        while (i <= gt) {
            int cmp = nums[i].compareTo(v);
            if (cmp < 0) {
                swap(nums, lt++, i++);
            } else if (cmp > 0) {
                swap(nums, i, gt--);
            } else {
                i++;
            }
        }
        sort(nums, l, lt - 1);
        sort(nums, gt + 1, h);
    }
}
```

5. 基于切分的快速选择算法

快速排序的 partition() 方法，会返回一个整数 j 使得 a[l..j-1] 小于等于 a[j]，且 a[j+1..h] 大于等于 a[j]，此时 a[j] 就是数组的第 j 大元素。

可以利用这个特性找出数组的第 k 个元素。

该算法是线性级别的，假设每次能将数组二分，那么比较的总次数为 (N+N/2+N/4+..)，直到找到第 k 个元素，这个和显然小于 2N。

```
public T select(T[] nums, int k) {
    int l = 0, h = nums.length - 1;
    while (h > l) {
        int j = partition(nums, l, h);

        if (j == k) {
            return nums[k];

        } else if (j > k) {
            h = j - 1;

        } else {
            l = j + 1;
        }
    }
    return nums[k];
}
```

堆排序

1. 堆

堆中某个节点的值总是大于等于其子节点的值，并且堆是一颗完全二叉树。

堆可以用数组来表示，这是因为堆是完全二叉树，而完全二叉树很容易就存储在数组中。**位置 k 的节点的父节点位置为 k/2，而它的两个子节点的位置分别为 2k 和 2k+1。这里不使用数组索引为 0 的位置**，是为了更清晰地描述节点的位置关系。

```
public class Heap<T extends Comparable<T>> {

    private T[] heap;
    private int N = 0;

    public Heap(int maxN) {
        this.heap = (T[]) new Comparable[maxN + 1];
    }

    public boolean isEmpty() {
        return N == 0;
    }

    public int size() {
        return N;
    }
```

```
    private boolean less(int i, int j) {
        return heap[i].compareTo(heap[j]) < 0;
    }

    private void swap(int i, int j) {
        T t = heap[i];
        heap[i] = heap[j];
        heap[j] = t;
    }
}
```

2. 上浮和下沉

在堆中，**当一个节点比父节点大，那么需要交换这个两个节点。交换后还可能比它新的父节点大，因此需要不断地进行比较和交换操作，把这种操作称为上浮。**

```
private void swim(int k) {
    while (k > 1 && less(k / 2, k)) {
        swap(k / 2, k);
        k = k / 2;
    }
}
```

类似地，当一个节点比子节点来得小，也需要**不断地向下进行比较和交换操作**，把这种操作称为**下沉。一个节点如果有两个子节点，应当与两个子节点中最大那个节点进行交换。**

```
private void sink(int k) {
    while (2 * k <= N) {
        int j = 2 * k;
        if (j < N && less(j, j + 1))
            j++;
        if (!less(k, j))
            break;
        swap(k, j);
        k = j;
    }
}
```

3. 插入元素

将新元素放到数组末尾，然后上浮到合适的位置。

```
public void insert(Comparable v) {
    heap[++N] = v;
    swim(N);
}
```

4. 删除最大元素

从数组顶端删除最大的元素，并将数组的最后一个元素放到顶端，并让这个元素下沉到合适的位置。

```
public T delMax() {
    T max = heap[1];
    swap(1, N--);
    heap[N + 1] = null;
    sink(1);
    return max;
}
```

5. 堆排序

把**最大元素和当前堆中数组的最后一个元素交换位置，并且不删除它**，那么就可以得到一个从尾到头的递减序列，从正向来看就是一个递增序列，这就是堆排序。

5.1 构建堆

无序数组建立堆最直接的方法是从左到右遍历数组进行上浮操作。一个更高效的方法是从右至左进行下沉操作，如果一个节点的两个节点都已经是堆有序，那么进行下沉操作可以使得这个节点为根节点的堆有序。叶子节点不需要进行下沉操作，可以忽略叶子节点的元素，因此只需要遍历一半的元素即可。

5.2 交换堆顶元素与最后一个元素

交换之后需要进行下沉操作维持堆的有序状态。

```
public class HeapSort<T extends Comparable<T>> extends Sort<T> {
    /**
     * 数组第 0 个位置不能有元素
     */
    @Override
    public void sort(T[] nums) {
        int N = nums.length - 1;
        for (int k = N / 2; k >= 1; k--)
            sink(nums, k, N);

        while (N > 1) {
            swap(nums, 1, N--);
            sink(nums, 1, N);
        }
    }

    private void sink(T[] nums, int k, int N) {
        while (2 * k <= N) {
            int j = 2 * k;
            if (j < N && less(nums, j, j + 1))
                j++;
            if (!less(nums, k, j))
                break;
            swap(nums, k, j);
            k = j;
        }
    }
```

```
    private boolean less(T[] nums, int i, int j) {
        return nums[i].compareTo(nums[j]) < 0;
    }
}
```

6. 分析

一个堆的高度为 logN，因此在堆中插入元素和删除最大元素的复杂度都为 logN。

对于堆排序，由于要**对 N 个节点进行下沉操作，因此复杂度为 NlogN。**

堆排序是一种原地排序，没有利用额外的空间。

现代操作系统很少使用堆排序，因为它无法利用局部性原理进行缓存，也就是数组元素很少和相邻的元素进行比较和交换。

小结

1. 排序算法的比较

算法	稳定性	时间复杂度	空间复杂度	备注
选择排序	×	N^2	1	
冒泡排序	√	N^2	1	
插入排序	√	N ~ N^2	1	时间复杂度和初始顺序有关
希尔排序	×	N 的若干倍乘于递增序列的长度	1	改进版插入排序
快速排序	×	NlogN	logN(递归)	
三向切分快速排序	×	N ~ NlogN	logN	适用于有大量重复主键
归并排序	√	NlogN	N	
堆排序	×	NlogN	1	无法利用局部性原理

快速排序是**最快的通用排序算法**，它的内循环的指令很少，而且它还能利用缓存，因为它总是顺序地访问数据。它的运行时间近似为 ~cNlogN，这里的 c 比其它线性对数级别的排序算法都要小。

使用三向切分快速排序，实际应用中可能出现的某些分布的输入能够达到线性级别，而其它排序算法仍然需要线性对数时间。

排序方法	平均时间	最好时间	最坏时间
桶排序(不稳定)	O(n)	O(n)	O(n)
基数排序(稳定)	O(n)	O(n)	O(n)
归并排序(稳定)	O(nlogn)	O(nlogn)	O(nlogn)

排序方法	平均时间	最好时间	最坏时间
快速排序(不稳定)	O(nlogn)	O(nlogn)	O(n^2)
堆排序(不稳定)	O(nlogn)	O(nlogn)	O(nlogn)
希尔排序(不稳定)	O(n^1.25)		
冒泡排序(稳定)	O(n^2)	O(n)	O(n^2)
选择排序(不稳定)	O(n^2)	O(n^2)	O(n^2)
直接插入排序(稳定)	O(n^2)	O(n)	O(n^2)

2. Java 的排序算法实现

Java 主要排序方法为 java.util.Arrays.sort()，对于原始数据类型使用三向切分的快速排序，对于引用类型使用归并排序。

最小生成树（Minimum Spanning Tree，MST）

> **最小生成树**：在一个有n个节点的**连通图**中，生成一棵**连通所有顶点**且**顶点间边的权重之和最小**的树。

切分定理： 在一幅加权图中，给定任意的切分，他的横切边中权重最小者必然属于图的最小生成树

在**连通图**中，基于边和顶点考虑有两个基本的算法可以生成**最小生成树**：

- **Prim算法**：以**顶点**为主要操作对象生成**最小生成树**
- **Kruskal算法**：以**边**为主要操作对象生成**最小生成树**

Prim算法

Prim算法思路

1. 从**连通图**中任意一个顶点开始，并将其放入**生成树**
2. 在**连通图**中找到**距离生成树最近**的**顶点**并将其放入**生成树**中
3. 重复**步骤2**直至**连通图**中所有顶点都添加进**生成树**

Prim算法的核心思想是：每次迭代都是在**连通图**中搜索离**生成树**最近的**顶点X**并将其添加进**生成树**。其中，**离生成树最近**表示**顶点X**相较于其它不在**生成树**中的顶点距离**生成树**最近。

Prim算法实现

Prim算法

`Prim算法`是用于解决`最小生成树`的算法之一,算法的每一步都会为一棵生长中的`树`添加一条边.一开始这棵树只有一个`顶点`,然后会一直添加到$V-1$条边,**每次总是将下一条连接树中的顶点与不在树中的顶点且权重最小的边加入到树中(也就是由树中顶点所定义的切分中的一条横切边).**

实现 Prim算法 还需要借助以下数据结构:

- 布尔值数组: 用于记录 顶点 是否已在 树 中.
- 队列: 使用一条队列来保存 最小生成树 中的边,也可以使用一个由 顶点 索引的 Edge 对象的数组.
- 优先队列: 优先队列用于保存 横切边 ,优先队列的性质可以每次取出 权值 最小的 横切边 .

延时实现（LAZY）

当我们连接新加入 树 中的 顶点 与其他已经在 树 中 顶点 的所有边都失效了(由于两个 顶点 都已在 树 中,所以这是一条失效的 横切边).我们需要处理这种情况,**即时实现对无效边采取忽略(不加入到优先队列中),而延时实现会把无效边留在优先队列中,等到要删除优先队列中的数据时再进行有效性检查.**

上图为 Prim算法 延时实现的轨迹图,它的步骤如下:

- 将 顶点0 添加到 最小生成树 中,将它的 邻接表 中的所有边添加到优先队列中(将 横切边 添加到优先队列).
- 将 顶点7 和边 0-7 添加到 最小生成树 中,将 顶点 的 邻接表 中的所有边添加到优先队列中.
- 将 顶点1 和边 1-7 添加到 最小生成树 中,将 顶点 的 邻接表 中的所有边添加到优先队列中.
- 将 顶点2 和边 0-2 添加到 最小生成树 中,将边 2-3 和 6-2 添加到优先队列中,边 2-7 和 1-2 失效.
- 将 顶点3 和边 2-3 添加到 最小生成树 中,将边 3-6 添加到优先队列之中,边 1-3 失效.
- 将 顶点5 和边 5-7 添加到 最小生成树 中,将边 4-5 添加到优先队列中,边 1-5 失效.
- 从优先队列中删除失效边 1-3 , 1-5 , 2-7 .
- 将 顶点4 和边 4-5 添加到 最小生成树 中,将边 6-4 添加到优先队列中,边 4-7 , 0-4 失效.
- 从优先队列中删除失效边 1-2 , 4-7 , 0-4 .
- 将 顶点6 和边 6-2 添加到 最小生成树 中,和 顶点6 关联的其他边失效.
- 在添加 V 个顶点与 V - 1 条边之后, 最小生成树 就构造完成了,优先队列中剩余的边都为失效边.

```
public class LazyPrimMST {

    private final EdgeWeightedGraph graph;

    // 记录最小生成树的总权重
    private double weight;

    // 存储最小生成树的边
    private final Queue<Edge> mst;

    // 标记这个顶点在树中
    private final boolean[] marked;

    // 存储横切边的优先队列
    private final PriorityQueue<Edge> pq;

    public LazyPrimMST(EdgeWeightedGraph graph) {
        this.graph = graph;
        int vertex = graph.vertex();
        mst = new ArrayDeque<>();
```

```
        pq = new PriorityQueue<>();
        marked = new boolean[vertex];

        for (int v = 0; v < vertex; v++)
            if (!marked[v]) prim(v);
    }

    private void prim(int s) {
        scanAndPushPQ(s);
        while (!pq.isEmpty()) {
            Edge edge = pq.poll();  // 取出权重最小的横切边
            int v = edge.either(), w = edge.other(v);
            assert marked[v] || marked[w];

            if (marked[v] && marked[w])
                continue; // 忽略失效边

            mst.add(edge); // 添加边到最小生成树中
            weight += edge.weight(); // 更新总权重
            // 继续将非树顶点加入到树中并更新横切边
            if (!marked[v]) scanAndPushPQ(v);
            if (!marked[w]) scanAndPushPQ(w);
        }
    }

    // 标记顶点到树中,并且添加横切边到优先队列
    private void scanAndPushPQ(int v) {
        assert !marked[v];
        marked[v] = true;
        for (Edge e : graph.adj(v))
            if (!marked[e.other(v)]) pq.add(e);
    }

    public Iterable<Edge> edges() {
        return mst;
    }

    public double weight() {
        return weight;
    }

}
```

即时实现

在即时实现中,将`v`添加到树中时,对于每个`非树顶点w`,**不需要在优先队列中保存所有从w到树顶点的边,而只需要保存其中权重最小的边,所以在将v添加到树中后,要检查是否需要更新这条权重最小的边(如果v-w的权重更小的话).**

也可以认为只会在优先队列中保存每个`非树顶点w`的一条边(也是`权重`最小的那条边),将`w`和`树顶点`连接起来的其他`权重`较大的边迟早都会失效,所以没必要在优先队列中保存它们.

要实现即时版的`Prim算法`,需要使用两个顶点索引的数组`edgeTo[]`和`distTo[]`与一个索引优先队列,它们具有以下性质:

- 如果`顶点v`不在树中但至少含有一条边和树相连,那么`edgeTo[v]`是将`v`和树连接的最短边,`distTo[v]`为这条边的`权重`.
- 所有这类`顶点v`都保存在索引优先队列中,索引`v`关联的值是`edgeTo[v]`的边的`权重`.
- 索引优先队列中的最小键即是`权重`最小的`横切边`的`权重`,而和它相关联的顶点`v`就是下一个将要被添加到`树`中的`顶点`.

- 将`顶点0`添加到`最小生成树`之中,将它的`邻接表`中的所有边添加到优先队列中(这些边是目前唯一已知的横切边).
- 将`顶点7`和边`0-7`添加到`最小生成树`,将边`1-7`和`5-7`添加到优先队列中,将连接`顶点4`与树的最小边由`0-4`替换为`4-7`.
- 将`顶点1`和边`1-7`添加到`最小生成树`,将边`1-3`添加到优先队列.
- 将`顶点2`和边`0-2`添加到最小生成树,将连接`顶点6`与树的最小边由`0-6`替换为`6-2`,将连接`顶点3`与树的最小边由`1-3`替换为`2-3`.
- 将`顶点3`和边`2-3`添加到`最小生成树`.
- 将`顶点5`和边`5-7`添加到`最小生成树`,将连接`顶点4`与树的最小边`4-7`替换为`4-5`.
- 将`顶点4`和边`4-5`添加到`最小生成树`.
- 将`顶点6`和边`6-2`添加到`最小生成树`.
- 在添加了`V - 1`条边之后,`最小生成树`构造完成并且优先队列为空.

```
public class PrimMST {

    private final EdgeWeightedGraph graph;

    // 存放最小生成树中的边
    private final Edge[] edgeTo;

    // 每条边对应的权重
    private final double[] distTo;

    private final boolean[] marked;

    private final IndexMinPQ<Double> pq;

    public PrimMST(EdgeWeightedGraph graph) {
        this.graph = graph;
        int vertex = graph.vertex();
        this.edgeTo = new Edge[vertex];
        this.marked = new boolean[vertex];
        this.pq = new IndexMinPQ<>(vertex);
        this.distTo = new double[vertex];
        // 将权重数组初始化为无穷大
        for (int i = 0; i < vertex; i++)
            distTo[i] = Double.POSITIVE_INFINITY;

        for (int v = 0; v < vertex; v++)
            if (!marked[v]) prim(v);
    }
```

```
    private void prim(int s) {
        // 将起点设为0.0并加入到优先队列
        distTo[s] = 0.0;
        pq.insert(s, distTo[s]);
        while (!pq.isEmpty()) {
            // 取出权重最小的边,优先队列中存的顶点是与树相连的非树顶点,
            // 同时它也是下一次要加入到树中的顶点
            int v = pq.delMin();
            scan(v);
        }
    }

    private void scan(int v) {
        // 将顶点加入到树中
        marked[v] = true;

        for (Edge e : graph.adj(v)) {
            int w = e.other(v);
            // 忽略失效边
            if (marked[w]) continue;
            // 如果w与连接树顶点的边的权重小于其他w连接树顶点的边
            // 则进行替换更新
            if (e.weight() < distTo[w]) {
                distTo[w] = e.weight();
                edgeTo[w] = e;
                if (pq.contains(w))
                    pq.decreaseKey(w, distTo[w]);
                else
                    pq.insert(w, distTo[w]);
            }
        }
    }

    public Iterable<Edge> edges() {
        Queue<Edge> mst = new ArrayDeque<>();
        for (int v = 0; v < edgeTo.length; v++) {
            Edge e = edgeTo[v];
            if (e != null) {
                mst.add(e);
            }
        }
        return mst;
    }

    public double weight() {
        double weight = 0.0;
        for (Edge e : edges())
            weight += e.weight();
        return weight;
    }

}
```

不管是`延迟实现`还是`即时实现`,`Prim算法`的规律就是: **在树的生长过程中,都是通过连接一个和新加入的顶点相邻的顶点.当新加入的顶点周围没有非树顶点时,树的生长又会从另一部分开始.**

Kruskal算法

`Kruskal算法`的思想是**按照边的权重顺序由小到大处理它们**,将边添加到`最小生成树`,加入的边不会与已经在`树`中的边构成环,直到`树`中含有`V - 1`条边为止.**这些边会逐渐由一片森林合并为一棵树**,也就是我们需要的`最小生成树`.

与Prim算法的区别

- `Prim算法`是一条边一条边地来构造`最小生成树`,每一步都会为`树`中添加一条边.
- `Kruskal算法`构造`最小生成树`也是一条边一条边地添加,但不同的是它寻找的边会连接一片`森林`中的两棵`树`.从一片由`V`棵单`顶点`的树构成的`森林`开始并不断地将两棵`树`合并(可以找到的最短边)直到只剩下一棵`树`,它就是`最小生成树`.

实现

要实现`Kruskal算法`需要借助`Union-Find`数据结构,它是一种树型的数据结构,用于处理一些不相交集合的合并与查询问题.

关于`Union-Find`的更多资料可以参考下面的链接:

- Union-Find简单实现
- Disjoint-set data structure - Wikipedia

```
public class KruskalMST {

    // 这条队列用于记录最小生成树中的边集
    private final Queue<Edge> mst;

    private double weight;

    public KruskalMST(EdgeWeightedGraph graph) {
        this.mst = new ArrayDeque<>();
        // 创建一个优先队列,并将图的所有边添加到优先队列中
        PriorityQueue<Edge> pq = new PriorityQueue<>();

        for (Edge e : graph.edges()) {
            pq.add(e);
        }

        int vertex = graph.vertex();
        // 创建一个Union-Find
        UF uf = new UF(vertex);
        // 一条一条地添加边到最小生成树,直到添加了 V - 1条边
        while (!pq.isEmpty() && mst.size() < vertex - 1) {
            // 取出权重最小的边
            Edge e = pq.poll();
            int v = e.either();
            int w = e.other(v);
            // 如果这条边的两个顶点不在一个分量中(对于union-find数据结构中而言)
            if (!uf.connected(v, w)) {
```

```
                // 将v和w归并(对于union-find数据结构中而言),然后将边添加进树中,并计算更新
权重
                uf.union(v, w);
                mst.add(e);
                weight += e.weight();
            }
        }
    }

    public Iterable<Edge> edges() {
        return mst;
    }

    public double weight() {
        return weight;
    }

}
```

上面代码实现的`Kruskal算法`使用了一条队列来保存`最小生成树`的边集,一条优先队列来保存还未检查的边,一个`Union-Find`来判断失效边.

性能比较

算法	空间复杂度	时间复杂度
Prim(延时)	E	ElogE
Prim(即时)	V	ElogV
Kruskal	E	ElogE

所需数据结构

算法	所需结构
Prime（Lazy）	优先队列、队列MST（记录已经加入MST里面的边）、bool 数组（记录顶点是否已经加入MST）
Prime (eager)	索引优先队列（K: 顶点 V: 顶点到MST的最小距离）、edgeto[] (记录到达 v的路径)、distTo[] (记录 v 到 MST 的距离，与索引优先队列里面的值相同)
Kruskal	队列 （记录已经加入到 MST 里面的边）、优先队列（从中取出权值最小的边），UF （判断失效边）

最短路

Dijkstra 算法

- 能解决边权非负数的加权有向图的单源最短路问题

算法：

1. 将 distTo[s] 初始化为 0，将distTo[] 中的其他元素初始化为正无穷
2. 初始化索引优先队列
3. 将（s,0）加入索引优先队列
4. 从优先队列中取出值最小的顶点 e
5. 遍历以该顶点为起点的每一条边 e,w ， 放松每一条边

   ```
   distTo[w] = distTo[v] + e.weight();
   ```

6. 同时维护优先队列

```
public class DijkstraSP {
   private DirectedEdge[] edgeTo;
   private double[] distTo;
   private IndexMinPQ<Double> pq;

   public DijkstraSP(EdgeWeightedDigraph G, int s) {
       edgeTo = new DirectedEdge[G.V()];
       distTo = new double[G.V()];
       pq = new IndexMinPQ<Double>(G.V());
       for (int v = 0; v < G.V(); v++)
           distTo[v] = Double.POSITIVE_INFINITY;
       distTo[s] = 0.0;

       pq.insert(s, 0.0);
       // relax vertices in order of distance from s
       while (!pq.isEmpty())
       {
           int v = pq.delMin();
           for (DirectedEdge e : G.adj(v))
               relax(e);
       }
   }
   private void relax(DirectedEdge e) {
       int v = e.from(), w = e.to();
       if (distTo[w] > distTo[v] + e.weight()) {
           distTo[w] = distTo[v] + e.weight();
           edgeTo[w] = e;
           // update PQ
           if (pq.contains(w)) pq.decreaseKey(w, distTo[w]);
           else                pq.insert     (w, distTo[w]);
       }
   }
}
```

时间复杂度：ElogV （V次插入操作，V次删除最小元素的操作、最坏情况下的 E 次改变优先级的操作，基于堆实现的优先队列这些操作的复杂度为 logV）

空间复杂度：V

Bellman Ford

将 distTo[] 数组元素初始化为无穷大，以任意顺序放松图的所有边，重复 V 轮

- 基于队列的 Bellman-Ford 算法

  使用 FIFO 队列记录 distTo[] 值发生变换的顶点，在队列中加入发生了松弛操作的边的终点，从队列中依次取出这些顶点，遍历以这些顶点为起点的边，进行松弛，直到队列为空

algorithm	**restriction**	**typical case**	**worst case**	**extra space**
topological sort	no directed cycles	E +V	E +V	V
Dijkstra (binary heap)	no negative weights	E log V	E log V	V
Bellman-Ford	no negative cycles	EV	EV	V
Bellman-Ford (queue-based)	no negative cycles	E+V	EV	V

MST

- gready
- 实现的数据结构所表现的性能

最短路径

- 不同约束条件下的计算
  - Dijkstra：
  - BellmanFord
  - 拓扑排序
  - spin Dag
- 性能、主题数据结构