0. 标准读取

BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));//需要抛出 **IOException**

1. 基础算法

1.1 二分查找

```java
public int binarySearch(int[] array, int target) {
  int left = 0;
  int right = array.length - 1;
  while(left + 1< right){
    int mid = left + (right - left) / 2;
    // 这里的代码针对的问题是：找到最后一次出现的target的位置
    if(array[mid] >= target){ // 因为是最后一次出现，要选择右边的区间
      left = mid;
    }else{
      right = mid;
    }
  }
  // 循环结束后，再对left和right单独判断
  if(array[right] == target){
    return right;
  }else if(array[left] == target){
    return left;
  }
  return -1;
}
```

1.2 快速排序 （快速选择就去掉一次递归）

```java
public void quickSort(int[] array, int l, int r) {
    if (l >= r) {return; }
    int i = l - 1, j = r + 1, x = array[l + (r - l) / 2];
    while(i < j){
      do{
        i++;
      }while(array[i] < x);
      do{
        j--;
      }while(array[j])
      if(i < j){
        swap(array, i, j);
      }
    }
    quickSort(array, l, j);
    quickSort(array, j + 1, r)
}
```

1.3 归并排序

```java
public static void mergeSort(int[] array, int[] helper, int l, int r) {
  if (l >= r) {return; }
  int mid = l + (r - l) / 2;
  mergeSort(array, helper, l, mid);
  mergeSort(array, helper, mid + 1, r);
  int k = 0,  i = l, j = mid + 1;
  while (i <= mid && j <= r) {
    helper[k++] = array[i] <= array[j] ? array[i++] : array[j++];
  }
  while (i <= mid) {
    helper[k++] = array[i++];
  }
  while (j <= r) {
    helper[k++] = array[j++];
  }
  for (i = l, j = 0; i <= r; ) {
    array[i++] = helper[j++];
  }
}
```

1.4 二维快排

```java
public void sort2D(int[][] array, int l, int r){
  if(l >= r){
    return;
  }
  int i = l - 1;
  int j = r + 1;
  int x = array[l + (r - l) / 2][0];
  while(i < j){
    do{
      i++;
    }while(array[i][0] < x);
    do{
      j--;
    }while(array[j][0] > x);

    if(i < j){
      int[] temp = array[i];
      array[i] = array[j];
      array[j] = temp;
    }
  }
  sort2D(array, l, j);
  sort2D(array, j+1, r);
}
```

1.5 宽度优先搜索

```java
public static void bfs(GraphNode node) {
  if(node == null) return;
  Queue<GraphNode> queue = new LinkedList<>();
  queue.offer(node);
  while(!q.isEmpty()){
    GraphNode cur = q.poll();
    "对cur的邻居进行处理"
    "例如  1.检查节点合法性，"
    "      2.获取同权重的所有节点(q.size())"
    "      3. 修改权重(修改数组的值)"
    q.offer(q valid adjusted neighbor node);
  }
}
```

1.6 深度优先搜索

深度优先搜索的本质是递归树的与树同高的叶子节点的集合。

```java
public void dfs("1. 输出容器 2. 局部解 3. 原始内容 4. index"){
    if("退出条件，  一般是判断局部解的size或者是index的大小"){
        "输出或保存一个局部解"
        return;
    }
    // 这里的index可以是0 也可以是层数，要根据具体题目来确定i的初始值
    for(int i = index/0; i < N; i++){ //N为当前层的分叉数量
      // 可以通过一个used数组或者一些其他的技巧去重
        "更新局部解/走到下一个分支"
        dfs("一样的参数，除了index需要更新：index + 1或者i+1");
        "如果局部解有添加新元素，删掉这个元素"
      // 如果用了额外数组去重，这里要归位
    }
}
```

1.7 双指针

没有模板 自己理解

1.8 位运算

1.8.1 lowbit 和第 k 位

```
1. N的二进制表示的从右往左第K位
  int kth = N>>k & 1 (从0开始)
2. 最低位的1所对应的值
  int lowbit(x){
    return x&(-x);
  }
   6 = 110  -> 10  = 2
  12 = 1100 -> 100 = 8

整数N的二进制表示：
  for(int i = 31; i >= 0; i--){
    sb.append(N>>i & 1);
  }
```

1.8.2 快速幂

```
//使用logb的复杂度计算a^b % p
//如果不取模，则不需要%p这个操作，也不需要传入p
public long fastPower(long a, long b, long p){
    long ret = 1 % p;
    while(b != 0){
        ret = (b & 1) == 1 ? ret * a % p : ret;
        a = a * a % p;
        b >>= 1;
    }
    return ret;
}
```

1.9 区间合并

```java
public int[][] merge(int[][] intervals) {
  // intervals 需是一个排好序的二维数组
  sort2D(intervals, 0, intervals.length - 1);
  List<int[]> ret = new ArrayList<>();
  int start = Integer.MIN_VALUE;
  int end = Integer.MIN_VALUE;
  for(int[] temp : intervals){
    if(end < temp[0]){
      if(start!= Integer.MIN_VALUE){
        ret.add(new int[]{start, end});
      }
      start = temp[0];
      end = temp[1];
    }else{
      end = Math.max(end, temp[1]);
    }
  }
  if(start != Integer.MIN_VALUE){
    ret.add(new int[]{start, end});
  }
  int[][] res = new int[ret.size()][];
  for(int i = 0; i < ret.size(); i++){
    res[i] = ret.get(i);
  }
  return res;
}
```

1.10 单调栈

```
单调栈 可能会用到两个单调栈 一个存下标一个存数值
int[] stack = new int[input.length];
int[] ret = new int[input.length];
int top = -1;
for(int i = 0; i < input.length; i++){
  while(top > -1 && input[i] >/< stack[top]){
    {
      logic
      可能更新RET
      可能更新哈希表
    }
    top --;
  }
  top++;
  stack[top] = T[i];
  //second stack
}
```

1.11 KMP

```
KMP
// 建next数组 核心思想 一步一步往后走 发现不够长就再往回走
  // 至少比之前要多匹配一个才算足够长啊
next[0] = -1;
for(int i = 1, j = -1; i < p.length(); i++){
  //不匹配就回跳 一直到匹配多一个长度为止
  while(j != -1 && p[j + 1] != p[i]){
    j = next[j];
  }
  if(p[j + 1] == p[i]){
    j++;
  }
  next[i] = j;
}

//匹配过程  核心思想 兽人永不为奴 位置永不回头 弹弹弹往回弹
  for (int i = 0, j = -1; i < s.length(); i++ ) {
    while(j != -1 && str[i] != p[j + 1]){
      j = next[j];
    }
    if(s[i] == p[j + 1]){
      j++; //这里i就不回头了
    }
    if( j == p.length() - 1){
      return i - j ;
      // 如果要继续匹配 那就不要return
      // 而是把这个值加入到答案中
      // 继续执行 j = next[j]
      j = next[j]
    }
  }
```

2. 初级数据结构

2.1 链表

2.1.1 翻转链表

```java
public ListNode reverse(ListNode head){
  if(head == null || head.next == null){
    return head;
  }
  ListNode newHead = reverseNode(head.next);
  head.next.next = head;
  head.next = null;
  return newHead;
}
```

```java
public ListNode reverse(ListNode head){
  ListNode prev = null;
  while(head != null){
    ListNode temp = head.next;
    head.next = prev;
    prev = head;
    head = temp;
  }
  return prev;
}
```

2.1.2 两两翻转

```java
public ListNode reverseInPairs(ListNode head) {
  if(head == null || head.next == null){
    return head;
  }

  ListNode temp = head.next;
  head.next = reverseInPairs(head.next.next);
  temp.next = head;
  return temp;
}
```

2.1.3 寻找中点

```java
public ListNode middleNode(ListNode head) {
  if(head == null){
    return head;
  }
  ListNode slow = head;
  ListNode fast = head;
  while(fast.next != null && fast.next.next != null){
    slow = slow.next;
    fast = fast.next.next;
  }
  return slow;
}
```

2.1.4 判断环及环入口

```java
public class Solution {
    public ListNode detectCycle(ListNode head) {
        if(head == null){
            return head;
        }
        boolean hasCircle = false;
        ListNode fast = head;
        ListNode slow = head;
        while(fast != null && fast.next != null){

            fast = fast.next.next;
            slow = slow.next;
            if(fast == slow){
                hasCircle = true;
                break;
            }
        }
        if(!hasCircle){
            return null;
        }
        fast = head;
        while(fast != slow){
            fast = fast.next;
            slow = slow.next;
        }
        return fast;
    }
}
```

2.2 栈和队列

最经典的题就是那个最小栈。只是一个数据结构，通常用于搜索问题的辅助。

2.3 堆

通过数组脑补出来的一个树形结构

2.3.1 KeyPoints

```
leftChildIndex = parentIndex * 2 + 1
rightChildIndex = parentIndex * 2 + 2
parentIndex = (anyChildIndex - 1) / 2
"A root's parentIndex is it self (0)"

常用复杂度
heapify建堆:     O(n)
insert/offer:    O(logN)
update:
    指定位置     O(logN)
    指定元素     O(n)
delete/poll:     O(logN)

```

除非让你手写 heap，否则直接用 PriorityQueue 就好

下面给出一个 maxHeap 的实现

2.3.2 大顶堆的简单实现

```java
/*
  @Author Jinwei
  @Version 2019/12/1
 */
public class MaxHeap {
  int[] heap; // the length can be the max volumn
  int size; // how many elements in the heap so far

  public static MaxHeap heapify(int[] array){
    return new MaxHeap(array);
  }

  public MaxHeap(int[] array){
    heap = new int[array.length];
    size = 0;
    for(int i = 0; i < array.length; i++){
      insert(array[i]);
    }
  }

  public boolean update(int index, int newValue){
    if(index > size){
      System.out.println("index too large");
      return false;
    }
    if(index == size){
      insert(newValue);
    }else{
      if(newValue < heap[index]){
        heap[index] = newValue;
        swimDown(index);
      }else{
        heap[index] = newValue;
        swimUp(index);
      }
    }
    return true;
  }
```

```java

  public void insert(int value){
    if(size == heap.length){ // too full
      int[] temp = new int[size * 2];
      for(int i = 0; i < heap.length; i++){
        temp[i] = heap[i];
      }
      heap = temp;
    }
    heap[size] = value;
    swimUp(size);
    size++;
  }
```

```
52
53    private void swimDown(int index){
54      int curIdx= index;
55      while(curIdx < size && curIdx * 2 + 1 < size){
56        int leftChildIdx =curIdx * 2 + 1;
57        int rightChildIdx = leftChildIdx + 1;
58        int larger = rightChildIdx < size
59                  && heap[rightChildIdx] > heap[leftChildIdx]
60                  ? rightChildIdx
61                  : leftChildIdx;
62        if(heap[curIdx] < heap[larger]){
63          swap(curIdx, larger);
64          curIdx = larger;
65        }else{
66          break;
67        }
68      }
69    }
70
71    private void swimUp(int index){
72      int curIdx = index;
73      while(curIdx > 0){
74        int parIdx = (curIdx - 1)/2;
75        if(heap[curIdx] > heap[parIdx]){
76          swap(curIdx, parIdx);
77          curIdx = parIdx;
78        }else{
79          break;
80        }
81      }
82    }
83
84    private void swap(int i, int j){
85      int temp = heap[i];
86      heap[i] = heap[j];
87      heap[j] = temp;
88    }
```

```
89
90    @Override
91    public String toString(){
92      StringBuilder sb = new StringBuilder();
93      sb.append('[');
94      for(int i = 0; i < size; i++){
95        sb.append(heap[i]);
96        sb.append(",");
97      }
98      sb.setCharAt(sb.length() - 1, ']');
99      String s = " with size of " + size;
100     sb.append(s);
101     return sb.toString();
102   }
103 }
```

2.4 图论-基础

2.4.1 树的节点的数量

```java
public static Node[] nodes;
public static int [] nodeNum;
public static boolean[] visited;

public static int getNodeTotalNum(Node root){
    if(root.neigh.size() == 1){
        nodeNum[root.val] = 1;
        return 1;
    }
    visited[root.val] = true;
    int sum = 1;
    for(Node n : root.neigh){
        if(nodeNum[n.val] == 0 && !visited[n.val]){
            sum += getNodeTotalNum(n);
        }
    }
    nodeNum[root.val] = sum;
    return sum;
}

// 给ab之间加一条双向边
public static void add(int a, int b){
    nodes[a].neigh.add(nodes[b]);
    nodes[b].neigh.add(nodes[a]);
}
add(1, 2);
add(1, 3);
add(4, 2);
add(4, 5);
add(3, 6);
add(7, 6);
add(8, 6);
add(9, 6);
add(7, 10);
// 调用getNodeTotalNum后, nodeNum将变成
nodeNum:
[0, 10, 3, 6, 2, 1, 5, 2, 1, 1, 1]
```

2.4.2 树的重心

```java
int ret = Integer.MAX_VALUE;
List<Node> centroids = new ArrayList<>();
for(int i = 1; i < n; i++){
    //删掉nodes[i]后 所有子连通块的值的最大值
    int temp = 0;
    for(Node nn : nodes[i].neigh){
        int adder = nodeNum[nn.val] > nodeNum[nodes[i].val]
                    ? nodeNum[nn.val] - nodeNum[nodes[i].val]
                    : nodeNum[nn.val];
        temp = Math.max(temp, adder);
    }
    temp = Math.max(temp, n - nodeNum[nodes[i].val]);
    if(temp == ret){
        centroids.add(nodes[i])
    }else if(temp < ret){
        ret = temp;
        centroids.clear();
        centroids.add(nodes[i])
    }
}
```

2.4.3 拓扑排序

```java
// 一个数组 存每个节点入度
// BFS, 入度 == 0 入队
// 队列中每个Node出队, 该Node的neigh的入度全部-1;
// 如果入度-1后==0, 继续入队
// NOTE1: 不存在入度=0则不存在拓扑排序
// NOTE2: 出现自环不存在拓扑排序
```

2.4.4 二分图

2.4.4.1 判断二分图

可以用 BFS 或者 DFS 染色来做，等价于任意连通块没有奇数边的环

需要一个数组或者 Map 来记录每一个节点的颜色，一旦冲突说明不是二分图

这里给出 DFS 和 BFS 两种解法

```java
public boolean dfs(GraphNode node, int c){
  map.put(node, c);
  for(GraphNode temp: node.neighbors){
    int cc = map.getOrDefault(temp, -1); // -1表示未被染色
    if(cc == -1){
      if(!dfs(temp, 1 - c )) return false;
    }else{
      if(cc == c){
        return false;
      }
    }
  }
  return true;
}

public boolean bfs(Map<GraphNode, Integer> map, GraphNode node){
  if(map.containsKey(node)){
    return true;
  }
  Queue<GraphNode> q = new LinkedList<>();
  map.put(node, 0); //initialize an unsean node to be tag 0;
  q.offer(node);
  while(!q.isEmpty()){
    GraphNode temp = q.poll();
    int neiTag = map.get(temp) == 0 ? 1 : 0;
    for(GraphNode neiNode : temp.neighbors){
      Integer i = map.get(neiNode);
      if(i == null){
        q.offer(neiNode);
        map.put(neiNode, neiTag);
      }else{
        if(i != neiTag){
          return false;
        }
      }
    }
  }
  return true;
}
```

2.4.4.2 二分图最大匹配

匈牙利算法

```java
Node[] left = new Node[n1 +1];
Node[] right = new Node[n2 + 1];
int[] match; // match[x] 表示右边配对的在左边的编号
boolean[] visited;

int ret = 0;
for(int i = 1; i < n1 + 1; i++){
    visited = new boolean[n2 + 1];
    if(find(i)){
        ret++;
    }
}
// 匈牙利算法
// 1. 如果自己的邻居中有没有配对的，或者配对了但是可以让另一半移情别恋的 返回True
// 2. 注意这里是一个递归，如果男方A的目标B配对了，那就让B的男人去找新的女人，
//      如果B要找的新的女人又有对象了，让这个新女人的男对象再去找更新的女人。
//      这里需要记录一个已经在排队等待另一半的额外信息，防止脚踏多条船
public boolean find(int x){
    for(Node candidate : left[x].neigh){
        if(!visited[candidate.val]){
            visited[candidate.val] = true;
            if(match[candidate.val] == 0 || find(match[candidate.val])){
                match[candidate.val] = x;
                return true;
            }
        }
    }
    return false;
}
```

2.4.5 最近公共祖先 LCA

待更新~ ~

3. 中级数据结构

3.1 前缀和（有效 index 推荐从 1 开始）

3.1.1 一维前缀和

```
定义：
  S[i] = a[0] + a[1] + ... + a[i]

构造：
  1. S[i] = S[i - 1] + a[i]
  2.  a[i] += a[i - 1]  inplace
性质：
  Sum([a[l], a[r]]) = S[r] - S[l - 1]
```

3.1.2 二维前缀和

```
定义：
S[i][j] = m[0][0]到 m[i][j] 矩阵的元素和
构造：
  1. S[i][j] = S[i][j - 1] + S[i - 1][j] + m[i][j] - S[i - 1][j - 1]
  2. m[i][j] += m[i][j - 1] + m[i - 1][j] - m[i - 1][j - 1]  inplace
性质：
  Sum([(x1, y1), (x2, y2)])  从m[x1][x2]到m[x2][y2]的子矩阵的元素和
    = S[x2][y2] - S[x2][y1 - 1] - S[x1 - 1][y2] + S[x1 - 1][y1 - 1]


```

3.2 差分 - 如果 b 的前缀和为 S　那么 b 就是 S 的差分

3.2.1 一维差分

```
构造方法1
  b[i] = S[i] - S[i - 1]
构造方法2 // recommended,
  通过n次循环insert(b, i, i, S[i])  i = [1, n] - 见模板总结
  // 给区间[l, r]中的每个数加上c：
  // 通过这个办法 每次传S[i], 传n次 可以直接初始化b
  insert(int[] b, int l, int r, int c){
    b[l] += c;
    b[r + 1] -= c;
  }

模板总结
  1. 定义insert()
  2. 执行insert(b, i, i, S[i]) i = [1, n]
  3. 通过insert(b, l, r, c)来执行询问更新b
```

3.2.2 二维差分

```
构造方法1
  b[i][j] = S[i][j] - S[i][j - 1] - S[i - 1][j] + S[i - 1][j - 1]
构造方法2
  通过n次循环insert(b, i, j, i, j, S[i][j]) - 见模板总结
  // 给矩形[x1, y1] - [x2, y2]中的每个数加上c：
  // 通过这个办法 每次传S[i], 传n次 可以直接初始化b
  insert(int[] b, int x1, int y1, int x2, int y2, int c){
    b[x1][y1] += c;
    b[x2 + 1][y1] -= c;
    b[x1][]y2 + 1] -= c;
    b[x2 + 1][y2 + 1] += c;
  }
模板总结
  1. 定义insert()
  2. 执行insert(b, i, j, i, j, S[i][j])
  3. 通过insert(b, x1, y1, x2, y2, c)来执行询问更新b
```

3.3 离散化

```
离散化：
  set.add(allInput);
  int k = 0;
  for(E e : set){
    mapping[k++] = e;
  }

  二分找映射关系，找到下标
```

3.4 字典树 Trie

```java
public class Trie { // 以只包含小写字母的trie为例
  public Node root;
  public Trie(){
    root = new Node();
  }
  private class Node{
    public int val = 0;        // 经过这个节点的单词数量
    public int endCount = 0;  //以该节点为终点的单词数量
    public Node[] branches = new Node[26];
  }
  public void insert(String str){
    Node cur = this.root;
    for(int i = 0; i < str.length(); i++){
      int idx = str.charAt(i) - 'a';
      if(cur.branches[idx] == null){
        cur.branches[idx] = new Node();
      }
      cur = cur.branches[idx];
      cur.val += 1;
    }
    cur.endCount +=1;
  }
  public int count(String str){
    Node cur = this.root;
    for(int i = 0; i < str.length(); i++){
      int idx = str.charAt(i) - 'a';
      if(cur.branches[idx] == null){
        return 0;
      }
      cur = cur.branches[idx];
    }
    return cur.endCount;
  }
  public int startsWith(String prefix){
    Node cur = this.root;
    for (int i = 0; i < prefix.length() ; i++) {
      int idx = prefix.charAt(i) - 'a';
      if(cur.branches[idx] == null){
        return 0;
      }
      cur = cur.branches[idx];
    }
    return cur.val;
  }
}
```

3.5 并查集 Union Find

作用：

合并两个集合

询问两个元素是否在一个集合中

3.5.1 朴素并查集

```
class UnionFind{
  public int[] parent;

  public int find(x){
    if(parent[x] != x){
      parent[x] = find(parent[x]);
    }
    return parent[x];
  }

  public int union(int x, int y){
    int xp = find(x);
    int yp = find(y);
    if(xp != yp){
      parentp[xp] = yp;
    }
  }
}
```

3.5.2 维护大小距离的并查集

```
class UnionFind{
  public int[] parent;
  public int[] size;
  public int[] dist;
  public int find(x){
    if(parent[x] != x){
      int temp = find(parent[x]);
      dist[x] += dist[parent[x]];
      parent[x] = temp;
    }
    return parent[x];
  }

  public int union(int x, int y, int offset){
    int xp = find(x);
    int yp = find(y);
    if(xp != yp){
      // 把x的祖先变成y的祖先
      // x挂在y下面
      parentp[xp] = yp;
      // dist[x] + dist[xp] = dist[y] + offset
      dist[x] = dist[y] + offset - dist[xp];
      size[yp] += size[xp];
    }
  }
}
```

3.6 树状数组

Log(n)完成更新和查询操作，是前缀和的折中版本。~ 待更新~

4. 数论和组合计数

4.1 质数

4.1.1 判断质数

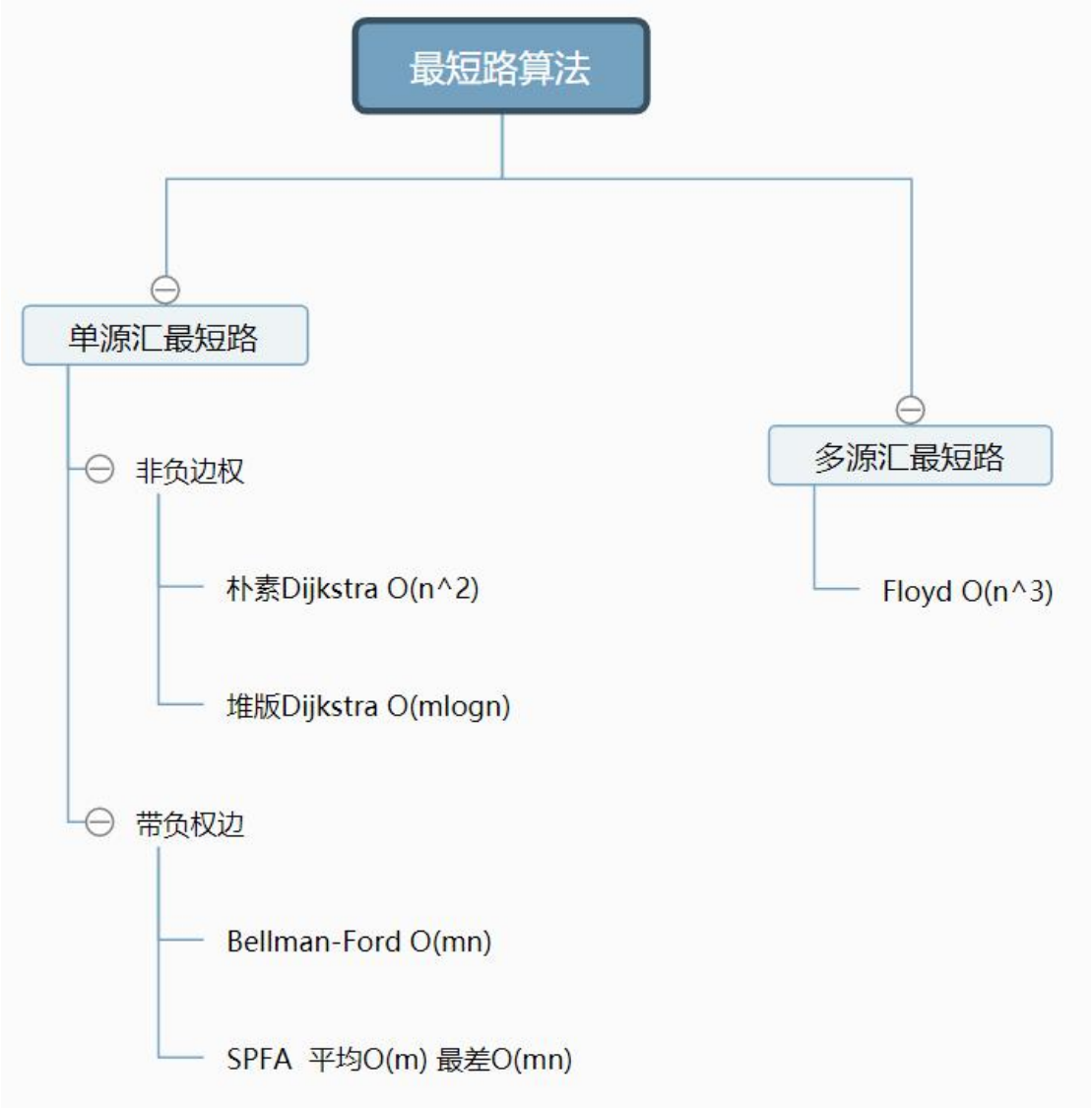
```java
// 试除法 O(Sqrt(n))  n = 输入的大小，这里为x
public List<Integer> checkPrime(int x){
    if(x == 2) {
        return true;
    }
    if(x % 2 == 0 || x <= 1){
        return false;
    }
    int upper = (int)Math.sqrt(x);

    for(int i = 3; i <= upper; i+=2){
        if(x % i == 0){
            return false;
        }
    }
    return true;
}
```

4.1.2 分解质因数

```java
// 试除法 O(logn) - O(Sqrt(n)之间
//      仅当n本身就是质数时候，取到复杂度上界
public List<Integer> decomposition (int x){
    List<Intger> ret = new ArrayList<>();
    int upper = (int) Math.sqrt(x);
    for(int i = 2; i <= upper, i++){
        if(x % i == 0){
            while(x % i == 0){
                ret.add(i);
                x /= i;
            }
        }
    }
    if(x > 1){ // 如果它本身就是一个质数
        ret.add(x);
    }
    return ret;
}
```

5. 图论-最短路

5.0 最短路总览



5.1 Dijkstra

5.1.1 朴素版本

```
找到A到B的最短路径
Arrays.fill(dist, 0x3f3f3f3f);
dist[A] = 0;
for(int i = 0; i < n; i++){
    int t = -1;
    //找到距离最短并且没有入档的点
    for(int j = 1; j < n + 1; j++){
        if(!st[j] && (t == -1 || dist[j] < dist[t])){
            t = j;
        }
    }
    // 更新这个点的邻居，并入档
    for(int j = 1; j < n + 1; j++){
        dist[j] = Math.min(dist[t] + nodes[t][j], dist[j]);
    }
    // 入档
    st[t] = true;

}
int ret = dist[n] == 0x3f3f3f3f ? -1 : dist[n];
```

5.1.2 堆优化版本

```java
//从A到B的稀疏图的最短路径
//堆优化版的Dijkstra算法
dist = new int[n + 1];
visited = new boolean[n + 1];
Arrays.fill(dist, 0x3f3f3f3f);
dist[a] = 0;
PriorityQueue<Node> q = new PriorityQueue<>(n + 1);
q.offer(nodes[A]);
while(!q.isEmpty()){
    Node temp = q.poll();
    visited[temp.val] = true;
    if(temp.val == B){
        break;
    }
    for(Map.Entry<Node, Integer> entry : temp.neigh.entrySet()){
        Node node = entry.getKey();
        int val = node.val;
        int wei = entry.getValue();
        if(visited[entry.getKey().val]){
            continue;
        }
        // 更新距离，入队
        if(dist[val] > dist[temp.val] + wei){
            dist[val] = dist[temp.val] + wei;
            q.offer(entry.getKey());  // 这句话一定要写在if里面
        }

    }
}
int ret = dist[b] == 0x3f3f3f3f ? -1 : dist[n];

class Node implements Comparable<Node>{
    public int val;
    public Map<Node, Integer> neigh;
    public Node(int i){
        this.val = i;
        neigh = new HashMap<>();
    }
    @Override
    public int compareTo(Node e){
        return dist[this.val] - dist[e.val]; //dist一定正数，不需要担心溢出
    }
}
```

5.2 Bellman-ford

```java
// 寻找从A到B不超过k条边的最短路
// m = 边数，n = 点数
Edge[] edges = new Edge[m];
int[] dist = new int[n + 1];
int []backup = new int[n + 1];
Arrays.fill(dist, 0x3f3f3f3f);
dist[A] = 0;

//循环k次，每次枚举所有边(松弛操作)
//如果k == n，则不限定边的长度，也不需要backup数组
//如果n次迭代后，新的迭代中dist还有变化，表示有负权环
//如果某一次迭代时，dist毫无变化，说明迭代完成可以提前结束
for(int i = 0; i < k; i++){
    System.arraycopy(dist, 0, backup, 0, dist.length);
    for(Edge edge : edges){
        dist[edge.b] = Math.min(dist[edge.b], backup[edge.a] + edge.w );
    }
}
int ret = dist[B] > 0x3f3f3f3f / 2 ? -1 : dist[B];

class Edge{
    public int a;
    public int b;
    public int w;
    public Edge(int a, int b, int w){
        this.a = a;
        this.b = b;
        this.w = w;
    }
}
```

4.3 Spfa

4.3.1 朴素 spfa

```java
//spfa求A到B的最短路(不可以有负权回路)
Queue<Node> q = new LinkedList<>();
q.offer(nodes[A]);
visited[A] = true;
while(!q.isEmpty()){
    Node temp = q.poll();
    visited[temp.val] = false; // 出队之后重置
    for(Map.Entry<Node, Integer> entry : temp.neigh.entrySet()){
        //尝试更新dist 如果dist被更新过且不在队列中 入队
        int newDist = dist[temp.val] + entry.getValue();
        if(newDist < dist[entry.getKey().val]){
            dist[entry.getKey().val] = newDist;
            if(!visited[entry.getKey().val]){
                visited[entry.getKey().val] = true;
                q.offer(entry.getKey());
            }
        }
    }
}
int ret = dist[dest] > 0x3f3f3f3f / 2 ? -1 : dist[dest];
```

4.3.2 Spfa 判断负权回路(负环)

0. 定义一个 count 数组，用来存储到当前节点的路径上边的数量
1. 不初始化 dist，全部为 0 就行

2. 所有的 node 全部进队, inQueue(visited)数组全部 true
3. 如果 dist 需要更新，那么同时更新 count
4. 一旦 count > n 则代表存在负权回路
5. 正常判断是否在队里，没有则把需要更新的目标节点入队

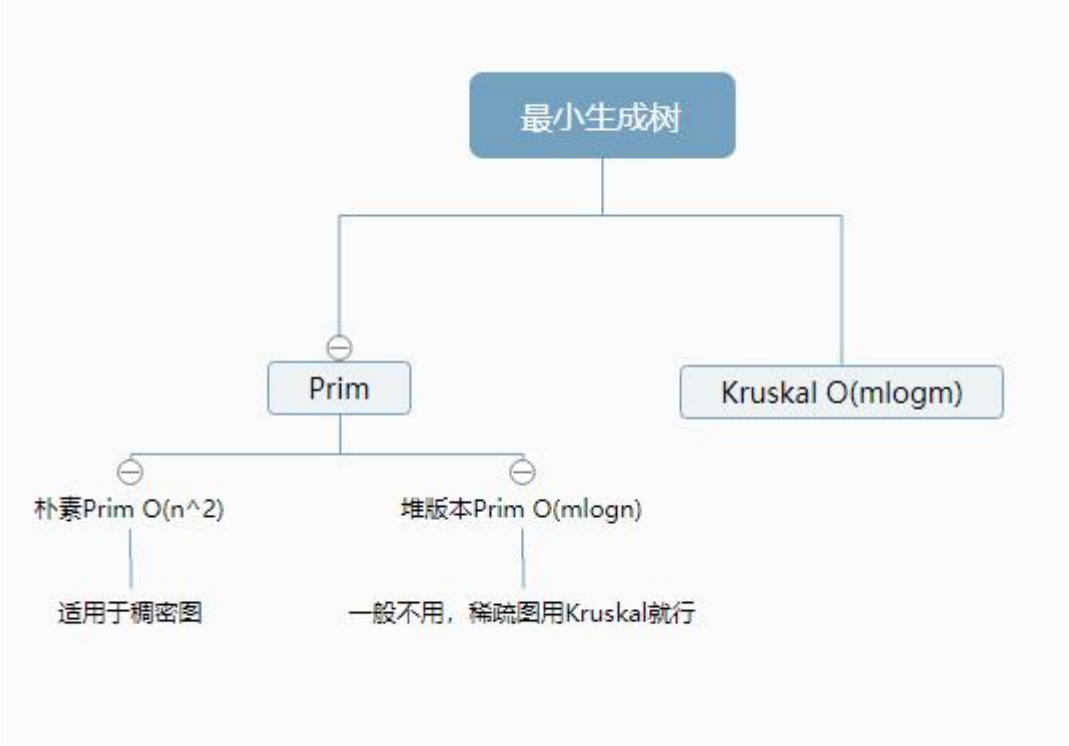
```java
public static boolean checkCycleSpfa(){
    // 0. 定义一个count数组, 用来存储到当前节点的路径上边的数量
    int[] count = new int[n +1];
    // 1. 不初始化dist, 全部为0就行
    Queue<Node> q = new LinkedList<>();
    // 2. 所有的node全部进队, inQueue(visited)数组全部true
    for(int i = 1; i < nodes.length; i++){
        q.offer(nodes[i]);
        inQueue[i] = true;
    }
    while(!q.isEmpty()){
        Node temp = q.poll();
        inQueue[temp.val] = false;
        for(Map.Entry<Node, Integer> entry : temp.neigh.entrySet()){
            int newDist = dist[temp.val] + entry.getValue();
            // 判断是否需要更新
            if(newDist < dist[entry.getKey().val]){
                dist[entry.getKey().val] = newDist;
                // 3. 如果dist需要更新, 那么同时更新count
                count[entry.getKey().val] = count[temp.val] + 1;
                // 4. 一旦count > n 则代表存在负权回路
                if(count[entry.getKey().val] >= nodes.length){
                    return true;
                }
                // 5. 正常判断是否在队里, 没有则把需要更新的目标节点入队
                if(!inQueue[entry.getKey().val]){
                    q.offer(entry.getKey());
                    inQueue[entry.getKey().val] = true;
                }
            }
        }
    }
    return false;
}
```

4.4 Floyd 多源汇最短路

```java
// Floyd求最短路
// 初始化dist二维数组，dist[i][j] 表示从i到j的最短距离
// dist同时也是点和点之间的初始边权
dist = new int[n + 1][n + 1];
for(int i = 1; i < n + 1; i++){
    for(int j = 1; j < n + 1; j++){
        dist[i][j] = i == j ? 0 : 0x3f3f3f3f;
    }
}
// 三个for循环 动态规划的思想
for(int k = 1; k < n + 1; k++){
    for(int i = 1; i < n + 1; i++){
        for(int j = 1; j < n + 1; j++){
            dist[i][j] = Math.min(dist[i][j],
                                  dist[i][k] + dist[k][j]);
        }
    }
}
```

6. 图论-最小生成树

6.0 最小生成树总览



6.1 朴素 Prim

```
Prim算法求最小生成树
// 1. 一定是双向边/无向边
// 2. 可以有负权回路


// 先把1号点作为初始集合中的元素(但还不在集合中，只是为了让它先被取出来)
dist[1] = 0;
// n 次循环 每次找离集合最短的点，用这个点更新其他点到集合的距离，进入集合。
int ret = 0;
for(int i = 0; i < n; i++){
  // 找离集合最近的点
  int t = -1;
  for(int j = 1; j < n + 1; j++){
    if(!inSet[j] && (t == -1 || dist[j] < dist[t])){
      t = j;
    }
  }
  if(dist[t] == 0x3f3f3f3f){
    ret = 0x3f3f3f3f;
    break;
  }
  // 提前更新 防止负自环
  ret += dist[t];
  // 更新其他点到集合的距离
  for(int j = 1; j < n + 1; j++){
    dist[j] = Math.min(dist[j], dist[t] + graph[t][j]);
  }
  inSet[t] = true;
}

如果ret==0x3f3f3f3f 表示无法生成树
否则ret就是最小生成树的边长
```

6.2 Kruskal

1. 排序所有边
2. 枚举所有边 如果不连通 合并 （并查集）
3. 合并过程中维护合并次数和总权重
4. 如果合并次数小于 n - 1，无法连通，否则最小生成树长度为总权重

```java
// 0. 初始化并查集
ancestor = new int[n + 1];
for(int i = 1; i < n + 1; i++){
    ancestor[i] = i;
}
// 1. 排序所有边
Arrays.sort(edges);
int count = 0;
int ret = 0;
// 2. 枚举所有边 如果不连通 合并 （并查集）
for(Edge edge : edges){
    int a = edge.a;
    int b = edge.b;
    int w = edge.w;
    int pa = find(a);
    int pb = find(b);
    if(pa != pb){
        ancestor[pa] = pb;
        // 3. 合并过程中维护合并次数和总权重
        ret += w;
        count++;
    }
}
// 4. 如果合并次数小于n - 1，无法连通，否则最小生成树长度为总权重
ret = count < n - 1 ? 0x3f3f3f3f : ret;
```