图

图的存储方式：

1. 邻接表法

   由两部分组成：表头结点表和边表。

   邻接表中每个单链表的第一个结点存放有关顶点的信息，把这一结点看成链表的表头，其余结点存放有关边的信息

   （1）表头结点表：包括数据域和链域，数据域存储顶点的名称，链域用于指向链表中第一个结点（与顶点邻接的第一个顶点）

   （2）边表：包括邻接点域（指示与顶点邻接的点在图中的位置，即数组下标）、数据域（存储和边相关的信息，如权值）、链域（指示与顶点邻接的下一条边的结点）。

2. 邻接矩阵

模板：经典的图结构：点 边。点：值（编号）、入点、出、邻点和边；边：权值、from、to

```
function Graph() {
    this.nodes = new Map();
    this.edges = new Set();
}

function Node(val) {
    this.val = val;
    this.in = 0;
    this.out = 0;
    this.nexts = [];
    this.edges = [];
}

function Edge(wight, from, to) {
    this.wight = wight;
    this.from = from;
    this.to = to;
}
```

图的宽度优先遍历

1. 利用队列实现
2. 从源节点开始依次按照宽度进队列，然后弹出
3. 每弹出一个点，把该节点所有没有进过队列的邻接点放入队列
4. 直到队列变空

```
function bfs(node) {
    if (node === null) return null;

    let queen = [];
    let set = new Set();
    queen.push(node);
    set.add(node);
    let curr = null;
```

```
    while (queen.length !== 0) {
        curr = queen.shift();
        console.log(curr.val, curr.nexts);

        for (let next of curr.nexts) {
            if (!set.has(next)) {
                queen.push(next);
                set.add(next);
            }
        }
    }
}
```

广度优先遍历

1. 利用栈实现
2. 从源节点开始把节点按照深度放入栈，然后弹出
3. 每弹出一个点，把该节点下一个没有进过栈的邻接点放入栈，并同时将该节点压入栈
4. 直到栈变空

```
function dfs(node) {
    if (node === null) return null;

    let stack = [];
    let set = new Set();
    stack.push(node);
    set.add(node);
    let curr = node;

    console.log("dfs", curr.val, curr.nexts);
    while (stack.length !== 0) {
        curr = stack.pop();

        for (let next of curr.nexts) {
            if (!set.has(next)) {
                // 将当前节点和下个节点都压回栈中
                stack.push(curr);
                stack.push(next);
                set.add(next);
                console.log(next.val);
                break;
            }
        }
    }
}
```

拓扑排序算法

适用范围：要求有向图，且有入度为 0 的节点，且没有环（有向无环图）
拓扑序（例子）：模块中各依赖的加载顺序
解法:

1. 将全部节点加入 hashMap 并记录节点入度，如果节点入度为 0 加入队列中
2. 依次将队列中的节点弹出加入 result，并将该节点的所有下个节点的入度减 1，如果下个节点的入度减完后为 0 则加入队列

```
function sortedTopology(G) {
    let hashMap = new Map();
    let queen = [];
    // 将全部节点加入hashMap并记录节点入度，如果节点入度为0加入队列中
    for (let node of G.nodes.values()) {
        hashMap.set(node, node.in);
        if (node.in === 0) {
            queen.push(node);
        }
    }

    let result = [];
    let curr;
    while (queen.length !== 0) {
        curr = queen.shift();
        result.push(curr);

        for (let next of curr.nexts) {
            hashMap.set(next, hashMap.get(next) - 1);
            console.log(next, hashMap.get(next));
            if (hashMap.get(next) === 0) {
                queen.push(next);
            }
        }
    }
    return result;
}
```

最小生成树

** 最小生成树 (minimum weight spanning tree)：** 在连通网的所有生成树中，所有边的代价和最小的生成树，称为最小生成树。

kruskal 算法(K 算法)

适用范围：要求无向图
此算法可以称为 “加边法”，初始最小生成树边数为 0，每迭代一次就选择一条满足条件的最小代价边，加入到最小生成树的边集合里。
思路： 从最小的边开始，依次加入边，如果形成环则不加入

1. 为所有节点的生成一个集合
2. 从最小的边开始，依次考察边
3. 看边的两个节点是否在同一集合中，如果是则说明形成环，不进行操作；如果不是，则将两个节点的集合合并

集合查询和合并：

```
class MySets {
    constructor(graph) {
        this.graph = graph;
        this.nodes = graph.nodes;
        this.setMap = new Map();
        this.mySets(this.nodes);
    }
    mySets(nodes) {
        for (let node of nodes.values()) {
            let set = new Set();
```

```
            set.add(node);
            this.setMap.set(node, set);
        }
    }
    isSameSet(from, to) {
        let fromSet = this.setMap.get(from);
        let toSet = this.setMap.get(to);
        return fromSet === toSet;
    }
    union(from, to) {
        let fromSet = this.setMap.get(from);
        let toSet = this.setMap.get(to);
        for (let node of toSet) {
            fromSet.add(node);

            this.setMap.set(node, fromSet);
        }
    }
}
```

```
function kruskalMST(graph) {
    let unionFind = new MySets(graph);

    let queue = [];
    for (let edge of graph.edges) {
        queue.push(edge);
    }
    queue.sort((a, b) => a.weight - b.weight);
    let result = new Set();
    for (let edge of queue) {
        let fromNode = edge.from;
        let toNode = edge.to;
        if (!unionFind.isSameSet(fromNode, toNode)) {
            result.add(edge);
            unionFind.union(fromNode, toNode);
        }
    }

    return result;
}
```

Prim 算法

适用范围：要求无向图
此算法可以称为 “加点法”，每次迭代选择代价最小的边对应的点，加入到最小生成树中。算法从某一个顶点 s 开始，逐渐长大覆盖整个连通网的所有顶点。
思路：

1. 从任意节点开始，搜索其所有 edge，找到权重最小的 edge，看这个 edge 的节点是否已经访问过（加入 set）
    - 已经访问过，则在剩下的 edge 中选择最小的重复判断
    - 没有访问过，将对应的节点加入 set 中并在 result 中记录这条 edge。然后将新记录到 set 中的节点的所有边加入小根堆中，在寻找权重最小的 edge 重复上述操作。

(这期间可能会把同一个 edge 多次存入到小根堆，这个没事，因为之后我们取出来都会看那个 toNode 是不是已经是在 set 了，如已经在了那个 edge 也不会被处理)
结果就是找到最小生成树，result 存的都是一个一个最小生成树的 edges

Dijkstra(迪杰斯特拉算法)算法

求解单元点的最短路径问题：给定带权 (weight) 有向图 G 和源点 v，求 v 到 G 中其他顶点的最短路径

限制条件：图 G 中不可以存在 整体累加和权值为负数的环 negative cycle (肯定不行), 也不可以有 negative weight (貌似)

1. 不断运行广度优先算法找可见点，计算可见点到源点的距离长度
2. 从当前已知的路径中选择长度最短的将其顶点加入 S 作为确定找到的最短路径的顶点。

- 一开始存答案的 hashmap, 每个 key 对应一个图中的节点然后 value 就是 source 到那个节点的最小 weight, 一开始 source 到自己是 0 其他都是无限
  先从 source 出发，然后找到他所有的 edges
- 然后把当前自己的 hashmap 中的值加上每个 edge 的 weight 值然后跟那个 edge 连向的节点当前数组中 hashmap 中的值比较，然后把 hashmap 值改成最小的那个值
- 处理完这个所有 edges, 我们就相当于结束了由 source 开始的所有 edges (我们也可以锁定当前节点在 hashmap 中的值，因为是第一次所以我们这次锁定就是 source 自己再 hashmap 中的值也就是 0, 他的距离到他自己就是 0)
- 然后我们从 hashmap 中选出除了以及当过当前节点的比如说 source 的所拥有的最小值的那个节点，然后让那个节点进行一样的操作 (看 edges 然后对于所有的 edges 的 toNodes 让他们的更新), 然后之后我们就把当前的这个节点在 hashmap 中的值锁死，然后让下一个...so on...
- 直到所有的都锁死了就结束，此时 hashmap 中存的值就是从 source 开始到图中每一个节点所需要的最小 weight (如果没有路就是无限)
- 初始化 hashMap，key 为图中节点，value 为 source 到对应节点的最短路径。初始化时 source 到自己的最小路径为 0
- 初始化 hashSet 记录访问过的节点，锁死这个节点
- 在 hashMap 中找为被锁定最小距离的节点（此时只有 source）
- 获取当前节点的 distance，遍历其所有的边，判断 toNode 是否存在于 hashMap 中，如果不存在此时 source 到节点的路径为无穷大，此时更新距离时直接将权重和路径求和；否则更新为之前的 distance 和 distance+weight 中较小值。将当前节点加入 hashSet 中锁死。

```
function dijkstra(head) {
    let hashMap = new Map();
    let hashSet = new Set();

    hashMap.set(head, 0);
    let minNode = getMinDistanceAndUmSelectNode(hashMap, hashSet);
    while (minNode !== null) {
        let currDistance = hashMap.get(minNode);

        let edges = minNode.edges;
        for (let edge of edges) {
            if (!hashMap.has(edge.to)) {
                hashMap.set(edge.to, currDistance + edge.weight);
            }

            let distance = hashMap.get(edge.to);
            hashMap.set(edge.to, Math.min(distance, currDistance +
edge.weight));
        }
        hashSet.add(minNode)
```

```
        minNode = getMinDistanceAndUmSelectNode(hashMap, hashSet)
    }
    return hashMap;
}

function getMinDistanceAndUmSelectNode(hashMap, hashSet) {
    let minDistance = Infinity;
    let minNode = null;

    for (let node of hashMap.keys()) {
        if (hashMap.get(node) < minDistance && !hashSet.has(node)) {
            minNode = node;
        }
    }

    return minNode;
}
```

优化:
一个优化 -> 在选择我们没有处理过且值最小的节点的时候是遍历的方式，可以使用堆结构来存储，然后每次就是存还没处理的，最小值的在堆顶，然后处理过的让他别参加堆结构

但是有一个问题，因为我们那些 values 是存从 source 到每个节点最小的 weight, 所以可能哪次遍历中我们会把一个堆结构中一个节点的值改了改成了一个更小的值，我们知道系统提供的堆结构无法接收让我们改变里面已经在的节点 (你要是硬改，系统里面自己做出的操作其实就跟我们直接遍历的复杂度都差不多了 —> 他需要全局扫描), 所以要是想实现，必须我们自己写一个堆

```
class NodeRecord {
    constructor(node, distance) {
        this.node = node;
        this.distance = distance;
    }
}
// 小根堆
class Heap {
    constructor() {
        this.heap = []
        // 记录节点在heap中的index
        this.heapIndexMap = new Map();
        // 记录节点的distance
        this.distanceMap =  new Map();
    }
    addOrUpdateOrIgore(node, distance) {
        if (this.inHeap(node)) {
            this.distanceMap.set(node, Math.min(this.distanceMap.get(node),
distance))
            this.insertHeapify(node, this.heapIndexMap.get(node))
        }
        if (!this.isEntered(node)) {
            this.heap.push(node)
            this.heapIndexMap.set(node, this.size()-1)
            this.distanceMap.set(node, distance)
            this.insertHeapify(node, this.size()-1)
        }
    }
    pop() {
```

```
        this.swap(0, this.size() - 1)
        let node = this.heap.pop()
        let nodeRecord = new NodeRecord(node, this.distanceMap.get(node))
        this.heapIndexMap.set(node, -1)
        this.heapify(0, this.size())
        return nodeRecord
    }
    // 在指定位置插入时向上调整
    insertHeapify(node, index) {
        while (this.distanceMap.get(node) <
this.distanceMap.get(this.heap[(index-1)>>1])) {
            this.swap(index, (index - 1) >> 1)
            index = (index - 1) >> 1
        }
    }
    // 向下调整
    heapify(index, size) {
        let left = index * 2 + 1
        while (left < size) {
            let minIndex = left + 1 < size &&
this.distanceMap.get(this.heap[left+1]) < this.distanceMap.get(this.heap[left])
? left + 1 : left
            if (this.distanceMap.get(this.heap[index]) >
this.distanceMap.get(this.heap[minIndex])) {
                break
            }
            this.swap(index, minIndex)
            index = minIndex
            left = index * 2 + 1
        }
    }
    // heap的长度
    size() {
        return this.heap.length
    }
    // heap是否为空
    isEmpty() {
        return this.heap.length === 0
    }
    // 节点是否进入过heap
    isEntered(node) {
        return this.heapIndexMap.has(node)
    }
    // 节点是否在heap上
    inHeap(node) {
        return this.heapIndexMap.has(node) && this.heapIndexMap.get(node) !== -1
    }
    swap(i, j) {
        // 交换heapIndexMap中记录的节点位置
        this.heapIndexMap.set(this.heap[i], j)
        this.heapIndexMap.set(this.heap[j], i)
        // 交换heap中的记录
        let temp = this.heap[i]
        this.heap[i] = this.heap[j]
        this.heap[j] = temp
        // [this.heap[i], this.heap[j]] = [this.heap[j], this.heap[i]]

    }
```

```
}

function dijkstra2(node) {
    let heap = new Heap();
    heap.addOrUpdateOrIgore(node, 0)

    let result = new Map()
    while (!heap.isEmpty()) {
        let curr = heap.pop()
        console.log(curr, 'curr')
        let node = curr.node
        let distance = curr.distance
        for (let edge of node.edges) {
            heap.addOrUpdateOrIgore(edge.to, edge.weight + distance)
        }
        result.set(node, distance)
    }
    return result
}
```