CSDN:111辄。祝学业顺利！

2022.5.28（计算机面试专业课5问）

2022.5.29（计算机面试专业课8问）

2022.5.30（剩下的几个问题太硬核了，乏了。今天写了5道代码了，排序题目。今天解决1 问，剩下的7个问题弄点别的。排序剩下的6个明天整完）

2022.5.31（排序这几个题属实给我整破防了，一直在敲代码）

2022.6.1（终于完成排序了，后三种一个太简单了没敲，另外两个抄答案）

2022.6.2（计算机面试专业课1问）

2022.6.3（计算机面试专业课1问）

2022.6.4（最短路径floyd和dijkstra整理完，还有bellman和bellman优化队列，明天搞）

2022.6.6（昨天摆烂，今天整理了bellman，优化队列感觉没必要；图的最小生成树。计算 机面试专业课完成，一轮完成）

1. **O(n)?时间复杂度？**

O(n)是指再最坏情况下的时间复杂度。

时间复杂度：对排序数据的总的操作次数。反映当n变化时，操作次数呈现什么规律。

空间复杂度：是指算法在计算机内执行时所需存储空间的度量，它也是数据规模n的函数。

1. **线性存储结构和链式存储结构的优缺点**

线性存储结构：地址空间相连，可以随机访问。但是删除、插入等操作需要花费很多时间移 动元素

链式存储结构：逻辑相邻的元素在物理位置上不一定相邻。删除、插入较方便，且不需要预先分配存储空间。但是不能随机存取了

1. **顺序存储与链式存储**

顺序存储：将逻辑上相邻的元素存放到物理位置上相邻的存储单元，数据元素之间的逻辑关 系由存储单元的邻接位置关系体现

链式存储：逻辑上相邻的元素在物理位置上不一定相邻，数据元素之间的逻辑关系由附加的 指针表示

1. **头指针和头结点的区别**

头指针指向链表的第一个结点，是第一个结点的存储位置。如果该链表有头结点，则头指针指向头结点。如果该链表没有头结点，则头指针指向链表的第一个结点。

头结点是链表的附加结点，是为了方便操作设立的。除了循环链表，所有链表都有头结点。头结点数据域一般无意义（有时也可存储链表长度等），其设立是为了优化单链表的增删操作，使得对链表第一个节点前的插入删除不必作为特殊情况处理，而是使用通用方法。

[参考链接](https://blog.csdn.net/qq_42552533/article/details/86105924)

1. **栈和队列的区别**

栈：后进先出。栈是只能在表末端进行插入和删除的线性表。允许插入和删除的一端叫栈顶，另一端叫栈底。

队列：先进先出。队列是只允许在表的一端插入，在另一端删除的线性表。允许插入元素的一端为队尾，另一端为队首

1. **栈和队列的存储结构**

栈：基于数组的顺序存储结构（栈顶指针）、基于链表的链式存储结构（栈顶指针）

队列：基于数组的顺序存储结构（队首指针、队尾指针）、基于链表的链式存储结构（队首指针、队尾指针）

1. **循环队列Q，编号0至n-1，头尾指针分别为f、p，求Q中元素的个数**

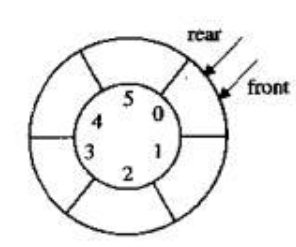
(p-f+n)%n

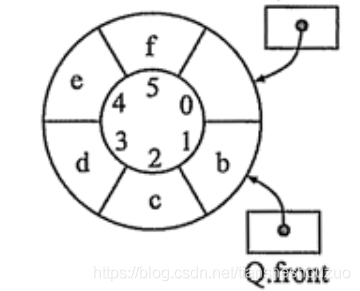
1. **如何区分循环队列是队空还是队满？**

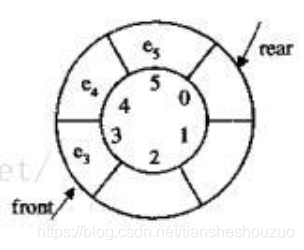
队头指针（front）：指向队列内首元素

队尾指针（rear）：指向队列内尾元素的下一个位置

队列容量（m）：包含留空的位置

队空：front=rear****

队满：front=（rear+1）%m

队内元素个数：（rear-front+m）%m

1. **树、二叉树、满二叉树、完全二叉树、二叉搜索树（二叉排序树）、平衡二叉树、最优二叉树（哈夫曼树）**

**树**：没有回路的连通无向图

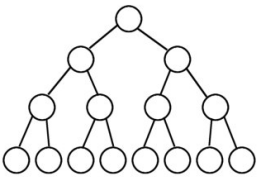
树的特性：一棵树中的任意两个结点有且只有唯一的路径相连

如果一棵树有n个结点，那么它一定恰好有n-1条边

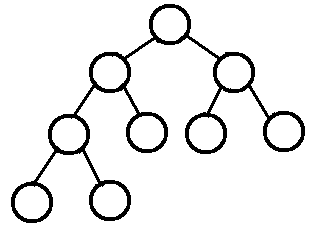
在一棵树中加一条边会构成一个回路

**二叉树**：树的每个父结点最多有两个子结点

**满二叉树**：树的非叶子结点都有两个子结点，即所有页子结点的深度都一致

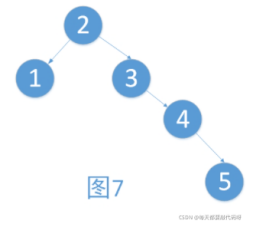
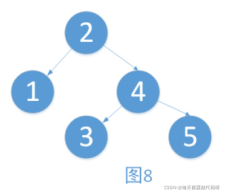


**完全二叉树**：假设二叉树的高度为h，则除了第h层外，其它所有层都达到最大结点数，第 h层从右向左缺少若干结点。即某结点若含有右子结点，则一定含有左子结点



**二叉排序树**（**二叉搜索树**）：左子树的值小于根节点，右子树的值大于根节点，左右子树都 是平衡二叉树。便于查找。且对二叉排序树进行中序遍历，即 可得到从小到大的排序

**平衡二叉树（ALV树）**：一种特殊的二叉搜索树，左右子树的高度差不超过1，且左右子树都是平衡二叉树

（非平衡二叉树）（是平衡二叉树）

**最优二叉树**（**哈夫曼树**）：给定N个带权的叶子节点，构造的带权路径最小的二叉树。可 以发现，权值较大的节点离根较近

1. **堆、大顶堆、小顶堆的实现与应用**

堆是一种特殊的完全二叉树

小顶堆：父结点的值小于其子结点

大顶堆：父结点的值大于其子结点

堆排序：时间复杂度O(NlogN)。例如从小到大排序，建立小顶堆，每次输出第一个结点的值 并删除该结点，剩下的结点重新排序。

1. **哈希表的概念、构造方法。哈希冲突的解决方法？**

哈希表：哈希表（Hash Table）又称散列表，是一种数据结构。它含有key-value键值对。它 通过散列函数将key映射到内存空间的某个位置，将键值对储存在这里。所以哈希 表的本质是数组。可以由数组+链表/数组+二叉树实现。为了解决冲突（即不同的 key映射到了同一个存储位置），可以使用开放寻址法（即该位置已经被前个key 占用了，则这个key排到紧邻的下一个存储单元中），也可以使用拉链法，即映射 到相同存储位置的key形成一个链表，上一个key的next指针指向下一个key

哈希表便于**查找**，复杂度是O(1)

[参考链接](https://blog.csdn.net/sinat_33921105/article/details/103344078?ops_request_misc=%257B%2522request%255Fid%2522%253A%2522165378899016782395385211%2522%252C%2522scm%2522%253A%252220140713.130102334..%2522%257D&request_id=165378899016782395385211&biz_id=0&utm_)

1. **先序和后序能否确定唯一的二叉树？**

不能。先序+后序不可以。但是其它两种（先+中，中+后）可以

[leetcode(根据前+中确定二叉树)](https://leetcode.cn/problems/construct-binary-tree-from-preorder-and-inorder-traversal/)：中序知根节点，然后在找到前序中的根节点，知道左孩子和右孩子，再依次递归即可。

1. **先序、中序、后序遍历**

先序：左孩子->根节点->右孩子

中序：根节点->左孩子->右孩子

后序：左孩子->右孩子->根节点

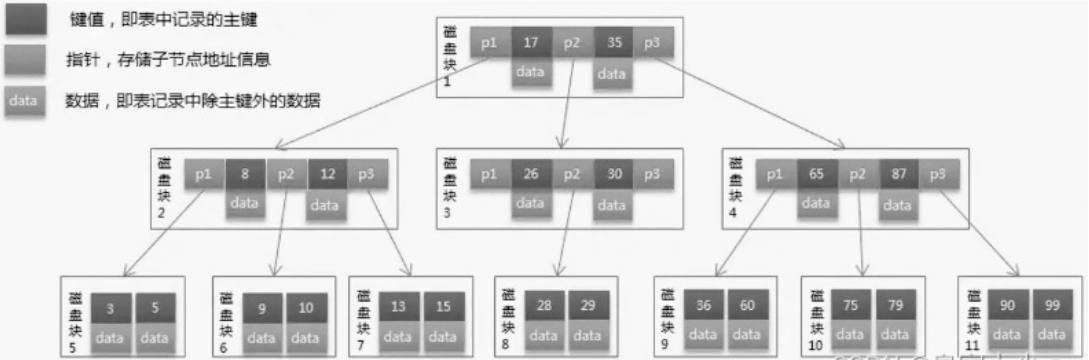
1. **B树和B+树**

B树：多路平衡查找树。它满足平衡二叉树的规则，但它可以有多个子树，子树的数量为该 节点关键字数量+1.

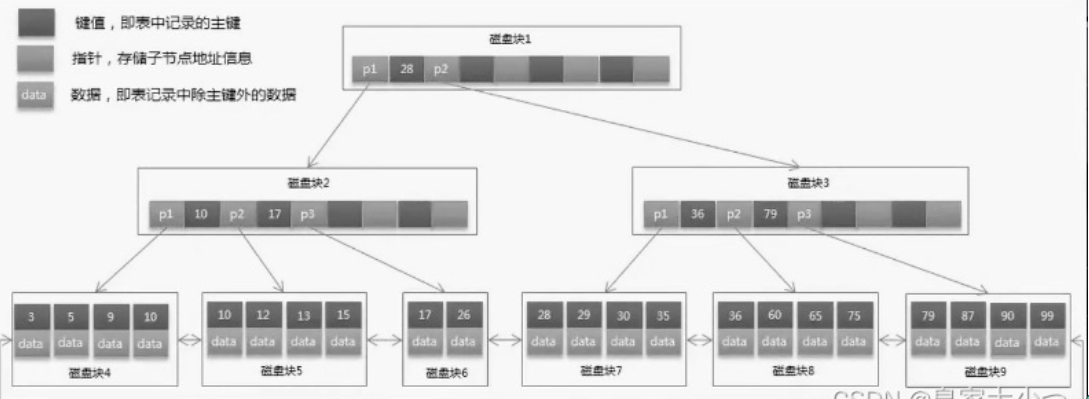
特点：在存储同等数据量的情况下，平衡二叉树的高度一定大于B树高度

B树和B+树有两个区别。其一，B树的数据存储在每个节点上；B+树的数据存储在叶子节点上，且通过链表链接叶子节点的所有数据。其二，B树的子树数量为关键字数量+1，B+树的子树数量为关键字数量。

B树：



B+树：



用途：数据库查询中，以数存储数据。树有多少层，意味着要读多少次磁盘I/0。所以树的高度越矮，意味着插叙数据时，需要读I/0的次数越少。B树的一个节点可以装多个值，读取时，将整个节点读到内存，在内存中，对节点的值进行处理（在内存中处理速度比磁盘快）。

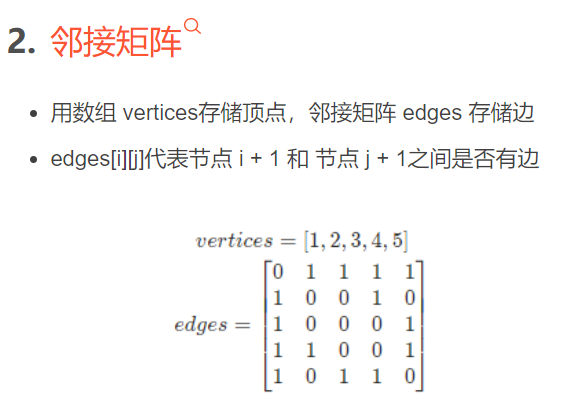
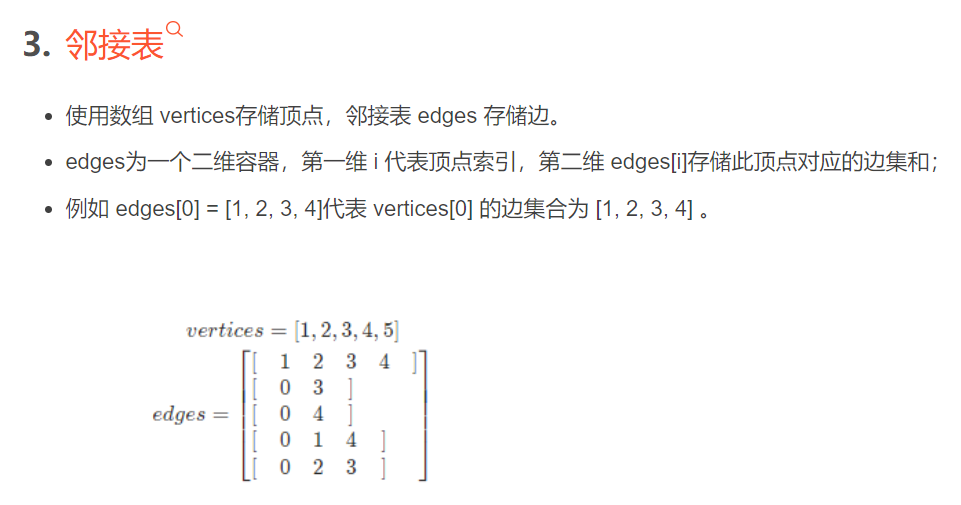
所以B树可以提升查找效率。但是它不利于范围查找，例如，要找0-100的索引值，则B树需要多次从根节点逐个查找

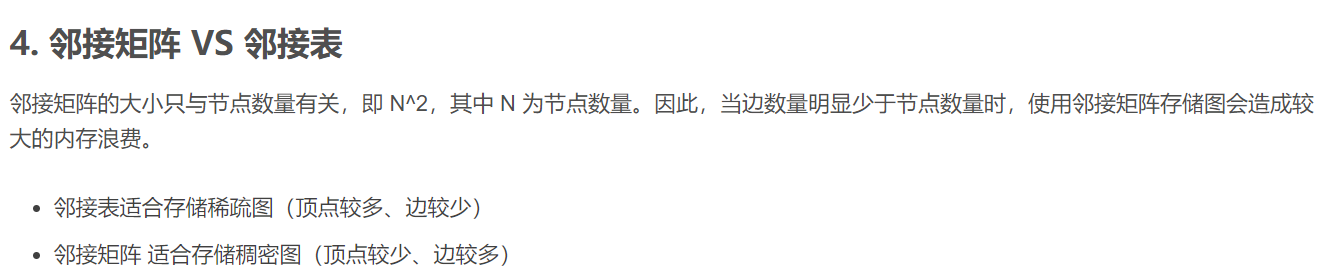
[参考链接](https://blog.csdn.net/ahuangqingfeng/article/details/124027058)

1. **红黑树**

红黑树：一种特殊的二叉搜索树。在每个节点上增加一个存储位置表示节点的颜色，red/black。 对节点的着色方式有5条限制。红黑树确保没有任何一条路径会比其它路径长处两 倍，是接近平衡的，他的节点插入、删除效率较高

1. **邻接表和邻接矩阵**



1. **二分搜索和线性搜索的区别**

线性搜索：O(n) 遍历范围内的所有数据，找到与key相同的数据。

二分搜索：O(logn) 需要待搜索数据有序排列，每次将搜索区间一分为二，与中间的数字进 行比较，并依据比较结果选择左区间或者右区间

1. **冒泡排序、选择排序、插入排序、希尔排序、归并排序、快速排序、堆排序、计数排序、桶排序、基数排序**

**冒泡排序**：每趟比较相邻两个元素大小，每趟会将最大的数放到未排序数的尾端，会进行 n-1趟。时间复杂度为O(n2)，稳定

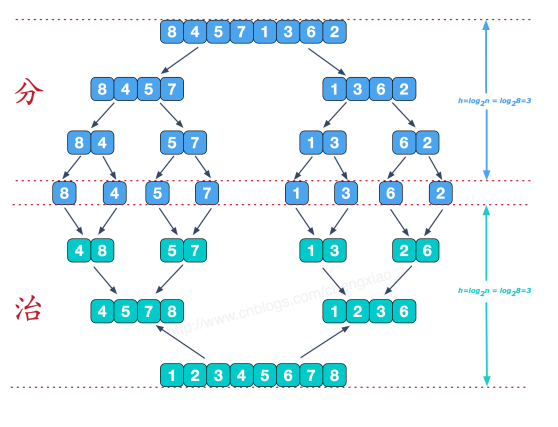
**快速排序**：对冒泡排序的改进。每次遍历会设置一个基准值，遍历后，左边的数均比该数小，右边的数均比此数大。递归。时间复杂度O(nlogn)，不稳定

**选择排序**：每趟在未排序的元素中直接选择最大的元素，放到未排序数的尾端，进行n-1次。 时间复杂度O(n2)，不稳定

**插入排序**：将元素逐个插入已排序元素列中（待插入元素依次与已排序元素比较，不是者后 移）。时间复杂度O(n2)，稳定

**希尔排序**：对插入排序的改进，相距一定元素排序。时间复杂度O(n1.3)，不稳定。

**归并排序**： 分治思想。将待排序元素分成若干个子序列，使其有序后合并。时间复杂度nlog(n)，稳定



**堆排序**： 每个非叶子节点的值都大于其子节点，大顶堆的顶是最大的元素。每次取出根节点元素，放到数组末尾，依次构建大顶堆即可。时间复杂度O(nlogn)，不稳定

**计数排序**：非比较类的排序算法。它首先找到最大和最小值，统计所有数的出现次数，存到另一个数组中，然后依次顺序填充回原数组。时间复杂度是O(n)，稳定。[code](https://blog.csdn.net/qq_41325698/article/details/88095563?ops_request_misc=%257B%2522request%255Fid%2522%253A%2522165405646016782425182538%2522%252C%2522scm%2522%253A%252220140713.130102334..%2522%257D&request_id=165405646016782425182538&biz_id=0&utm_medi)

**桶排序**： 将处于某区间的数放入各个桶中，然后对桶中的数字排序。时间复杂度O(n)，稳定

**基数排序**：按照个位、十位、百位排序。时间复杂度O(n)，稳定

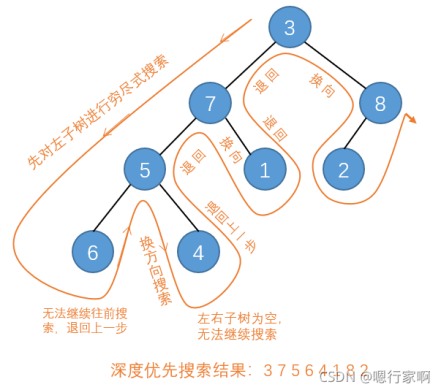
冒泡排序最坏情况：如4,3,2,1升序排序，需要比较 3+2+1=6=4\*3/2 次

快速排序最坏情况：已经排好顺序的（包括升序、降序、所有元素都一样）

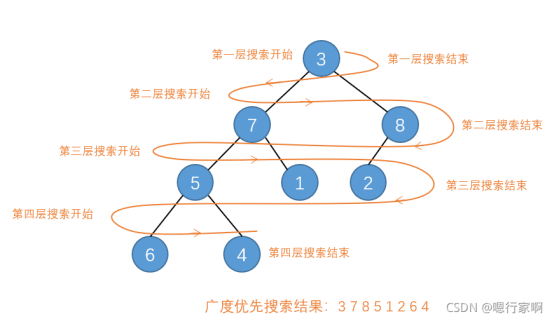
[参考链接](https://www.cnblogs.com/onepixel/articles/7674659.html)

1. **深度优先搜索算法和广度优先搜索算法**

**深度优先**：沿着一条路径不断向下搜索，直至不能继续再折返，搜索下一条路径



**广度优先**：根据离起点的位置，从近至远地对各节点搜索



1. **字符串匹配算法（朴素的和KMP算法）**

**朴素的**：暴力求解。对主串的每一个字符作为模式串的开头，依次于模式串进行匹配。每次匹配不成功，主串字符后移一位，直至匹配成功或者全部遍历为止。

**KMP**：每次匹配失败后模式串不必完全回退，找已经成功匹配成功的模式串的子串，找其相 同的最大前缀和最大后缀（小于该字串长度），然后将模式串直接后移到后缀的位置开 始新的匹配。这里的后移就是便于解释，真正实现的时候是通过数组的下标来进行操作。

对主串的每一个字符作为子串开头，与模式串进行匹配。对主串做大循环，每个字符开头做模式串长度的小循环，直到匹配成功或全部遍历完成为止。

[简单理解KMP（B站视频）](https://www.bilibili.com/video/BV1jb411V78H?spm_id_from=333.337.search-card.all.click)

[代码](https://blog.csdn.net/qq_36430106/article/details/90321872)

1. **最短路径算法**

**多源最短路径**：任意两点的最短路径

**单源最短路径**：一个源点到其余各个顶点的最短路径

Floyd是O(N3)，普通Dijkstra是O(N2)，优化的Dijkstra是O(NlogN)

① **Floyd**弗洛伊德算法（**多源最短路径**）（**动态规划**）

最开始只允许经过1号顶点中转，接下来只允许经过1号顶点和2号顶点中转……允许经过1-n号顶点中转。O(N3)

for(k=0;k<n;k++){

for(i=0;i<n;i++){

for(j=0;j<=n;j++){

if(e[i][j]>e[i][k]+e[k][j]){//e是邻接矩阵，e[i][j]为边i到j的权值

e[i][j]=e[i][k]+e[k][j];

}

}

}

}

② **Dijkstra**算法（**单源最短路径**）

核心思想是每次选择离源点最近的点，并以此点进行扩张，得到源点到其余点最短路径

具体为，它将所有顶点分为两部分：已知最短路径点集P和未知最短路径点集Q。每次在Q中选择一个离源点最近的点u，找到它的所有出边，做松弛操作。意思是，假如存在从u到v的边，那么比较从源点到u再到v的总路径长度，与s到v路径长度的大小，如果小则替代原路径长度。在对u的所有出边做松弛操作后，将u放入已知最短路径集P。O(N2)

详细步骤：

①将顶点分为已知最短路径的顶点集合P（book[i]=1）和未知最短路径的顶点集合Q（book[i]=0），最开始只有源点在P中

② 初始化最短路径数组dis[s]。源点为0，dis[i]为e[s][i]，其余为∞

③ 在Q中所有顶点选择距离s最近的顶点u（即dis[u]最小）加入P。对u的出边做松弛操作，比较dis[u]+e[u][v]和已知dis[v]大小。小则替代。直至Q为空，结束算法。dis数组是最短路径结果

for(i=0;i<n-1;i++){//遍历所有顶点，找其最短路径

//找离源点最近的点

min=inf;//inf为提前设定的无穷大(999999)

for(j=0;j<n;j++){

if(book[j]==0 &7 dis[j]<min){

min=dis[j];

u=j;

}

}

book[u]=1;//把u放到P里

for(v=0;v<n;v++){

if(e[u][v]<inf){

if(dis[v]>dis[u]+e[u][v]){

dis[v]=dis[u]+e[u][v];

}

}

}

}

对于稀疏图（边少），可用**邻接表**替代邻接矩阵，将O(N2)优化至O(M+N)logN

**数组实现邻接表**：

int n,m; //n为顶点数，m为边数

int u[6], v[6], w[6]; //第i条边从第u[i]号顶点到v[i]号顶点，权值为w[i]。（则u[]数组里可能有多个相同顶点号）

int first[5],next[6];//first和next为每个顶点设置了一个链表，保存从顶点i出发的所有边

scanf("%d %d",&n,&m);

for(int i=0;i<n;i++){

first[i]=-1;

}

for(int i=0;i<m;i++){

scanf("%d %d %d",&u[i],&v[i],&w[i]);//读入每条边

next[i]=first[u[i]];//next的下标表示边，数值表示该顶点的上一条边(然后first该点就覆盖了)

first[u[i]]=i;//first的下标表示顶点，数值表示这个顶点出发的边

//first[u[i]]保存顶点u[i]的第一条边的编号，next[i]保存"编号为i的边"的下一条边的编号

}

//遍历每个顶点的边

for(int i=0;i<n;i++){

k=first[i];

while(k!=-1){

prinf("%d %d %d\n",u[k],v[k],w[k]);

k=next[k];

}

}

③**Bellman-Ford**算法（**单源最短路径，解决带权负边**）

for(k=1;k<=n-1;k++){//共n-1轮，第k轮允许源点“经过k条边”到达其余各顶点的最短路径

Check=0;

for(i=1;i<=m;i++){

if(dis[v[i]]>dis[u[i]]+w[i])

dis[v[i]]=dis[u[i]]+w[i];

Check=1;

}

If(check==0) break;//如果dis数组没有更新，可结束循环

}

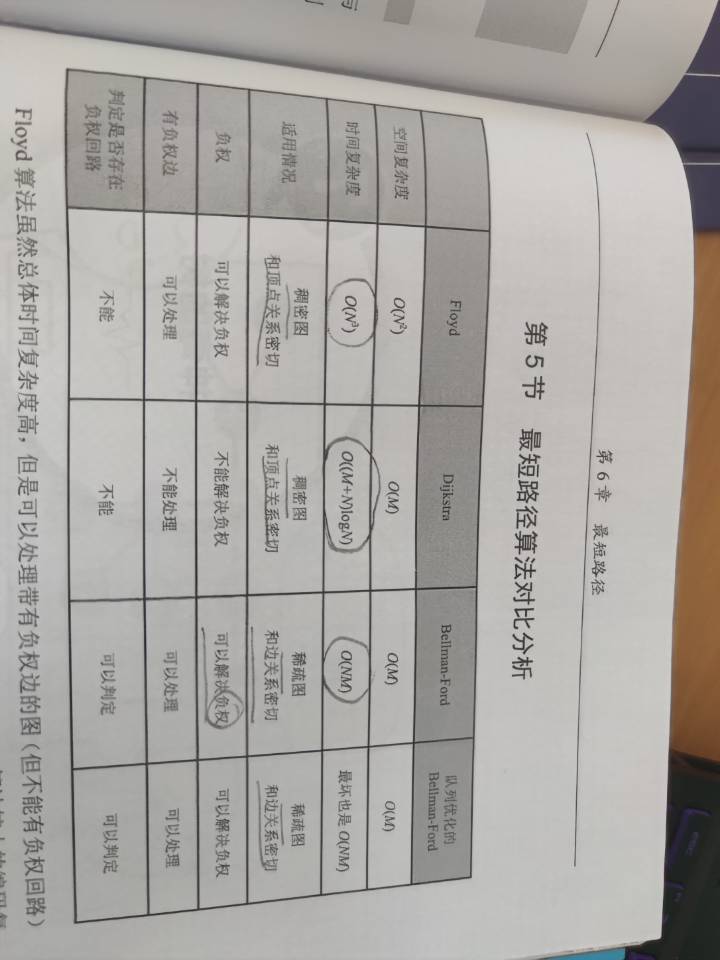
Flag=0;

For(int i=1;i<m;i++){

If(dis[v[i]]>dis[u[i]]+w[i]) flag=1;//如果经过n-1轮松弛后，还能继续松弛，则必然存在负 权路径

}

时间复杂度是O(MN)



1. **最小生成树**

图的最小生成树：一个不包含回路的连通无向图

Kruskal算法是一步步将森林中的树合并，而Prime算法是通过每次增加一条边来建立一棵树。Kruskal算法适用于稀疏图，Prime算法适用于稠密图（边多）

Kruskal和Prime均为O(NlogN)

1. **Kruskal算法（贪心）**

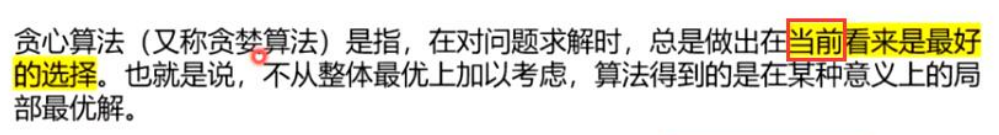
按边的权值进行从小到大排序，每次从剩余的边中选择权值较小且两个顶点不在一个集合的边（即不会产生回路），加入生成树中。时间复杂度为O(NlogN)

1. **Prime算法（贪心）**

将顶点分为两类：树顶点（已被加入生成树的顶点）和非树顶点（未被加入生成树的顶点）。首先任意选择一个顶点加入生成树，然后找一条边加入生成树（需枚举每一个树顶点到非树顶点所有边，找到最短边加入树）。时间复杂度为O(N2)，若使用邻接表储存图，时间复杂度降至O(NlogN)

1. **贪心和动态规划**

①**贪心（自上而下）**



②**动态规划（自下而上）**

动态规划是为了消除重叠子问题（一个子问题可能会被用到、计算多次）设计的

