Java数据结构和算法

内容提要：线性表、二叉树、图、散列、排序搜索、跳跃表、字典树、倒排索引、哈夫曼树、串操作、五大算法

数据结构：是相互之间存在一种或者多种特定关系的数据元素的集合。在逻辑上可以分为线性结构，散列结构、树形结构，图形结构等等。

算法：算法是求解具体问题的步骤的描述，代码上表现出来是解决特定问题的一组有限的指令序列。

算法复杂度：时间和空间复杂度，衡量算法效率，算法在执行过程中，随着数据规模n的增长，算法执行所花费的时间和空间的增长速度。

常见的时间复杂度：

大O计法	应用示例
O(1)	数组随机访问、哈希表
O(logn)	二分搜索、BST、AVL、RB查找
O(n)	线性搜索
O(nlogn)	堆排序、快速排序、归并排序
O(n^2)	冒泡排序、选择排序
O(2^n)	子集树
O(n!)	排列树

常见算法的时间复杂度关系：O(1) < O(logn) < O(n) < O(nlogn) < O(n^2) < O(2^n) < O(n!) < O(n^n)

线性表

线性表的顺序存储 - 数组 ArrayList

数组增加元素

末尾插入：O(1)　非末尾元素插入：O(n)

数组删除元素

末尾删除：O(1)　非末尾元素删除：O(n)

数组查找元素

数组的随机访问：arr[index] O(1)　数组元素的线性查找：O(n)

有序数组的搜索 - 二分查找算法

二分查找的时间复杂度：O(logn)　Arrays类库里面提供了相应的搜索算法 Arrays.binarySearch

线性表的链式存储 - 链表 LinkedList

链表区别于数组最大的特点：链表不需要大片连续的空间，所以内存使用效率要高；数组的每一个元素其内存空间都是连续的，因此在内存碎片化严重的时候，开辟大容量的数组，往往会出现内存分配失败（OOM）。

在链表中，每一个节点都包含了一个数据域和一个地址域（存储的是下一个节点的地址）。

单向链表

头插法时间复杂度：O(1)

尾插法时间复杂度：O(n) 先遍历一遍链表，找末尾节点O(n)，尾插法本身的操作是O(1)，合起来就是O(n)

删除操作： 从链表的头节点开始搜索元素O(n)，删除节点本身的操作O(1)，合起来就是O(n)

插入和删除操作本身：O(1)

链表的搜索(cur = cur.next)：O(n)　同数组搜索操作的(cur++)

数组和链表的应用场景对比：

数组的优势：随机访问 arr[index] 访问指定下标index号元素的值 O(1)，而链表中访问指定index号元素，需要从头节点开始一个个遍历，时间复杂度是O(n)。

单链表的优势：增加和删除操作本身是O(1)的，而数组的增加和删除需要涉及大量元素的移动，因此时间复杂度是O(n)。

扩容的优势：链表不存在满了之说，添加元素直接new新的节点进行存储即可；但是数组满了以后进行扩容，涉及了新内存的开辟，原内存的释放，以及大量的数据拷贝，扩容的效率较低。

结论：如果随机访问元素多，建议使用数组；如果增加删除操作多，建议使用链表。

单向循环链表

特点：末尾节点的地址域，保存头节点的地址。

作用：给定链表的任意一个节点，都可以遍历访问链表的所有节点。

应用场景：约瑟夫环（Josephus）

约瑟夫环是一个数学的应用问题：已知n个人（以编号1，2，3...n分别表示）围坐在一张圆桌周围，从编号为k的人开始报数，数到m的那个人出列，它的下一个人又从1开始报数，数到m的那个人又出列，依此规律重复下去，直到圆桌周围的人全部出列，输出人的出列顺序。

双向链表

特点：每一个节点包含数据域，前驱节点地址域和后继节点地址域。

作用：在链表的任意一个节点，都可以向前或者后方向遍历链表的其它节点。

链表应用-合并两个有序的单链表

链表应用-单链表逆置

链表应用-判断一个单链表是否有环，如果有环求出环的入口节点

链表应用-判断两个单链表是否相交

链表应用-求单链表倒数第K个节点

栈 LinkedList（链式栈）

特点：先进后出，后进先出。

顺序栈和链式栈

顺序栈底层是数组存储，扩容时需要开辟更大的空间，拷贝数据，释放原来的内存空间，所以顺序栈的扩容操作消耗是比较大的，但是链式栈是通过链表来存储栈元素的，直接开辟新的节点存储栈的元素即可，不需要专门进行扩容操作，效率比较好。

队列 LinkedList（链式队列）

特点：先进先出，后进后出。

循环队列和链式队列

直接用数组存储队列的元素，随着入队和出队操作，数组的空间使用效率是比较低的，所以一般实现为循环队列，但是循环队列的扩容操作也涉及内存开辟和释放，以及大量的数据拷贝，因此实现成链式队列效率更好。

栈和队列应用

两个队列实现一个栈

两个栈实现一个队列

四则运算表达式求解

深度优先遍历求解迷宫路径

广度优先遍历求解迷宫路径-求迷宫最短路径

散列表

散列表/哈希表定义：

使关键字和其存储位置满足关系：存储位置 = f（关键字），这就是一种新的存储技术-散列技术。

散列技术是在记录的存储位置和它的关键字之间建立一个确定的对应关系f，使得每个关键字key对应一个存储位置f（key），在查找时，根据这个确定的对应关系找到给定key的映射f(key)，如果待查找集合中存在这个记录，则必定在f（key）的位置上。

我们把这种对应关系f称为散列函数，又称为哈希函数。采用散列技术将记录存储在一块连续的存储空间中，这块连续的存储空间称为**散列表或者哈希表**（Hash Table）。

优势：适用于快速的查找 O(1)　**缺点**：占用内存空间比较大

散列函数：

设计特点：1.计算简单（复杂会降低查找的时间）　　2.散列地址分布均匀（减少哈希冲突）

1.直接定址法　f（key）= a x key + b（a、b为常数）

2.数字分析法　使用key的一部分来计算散列码（包括为了减少哈希冲突而进行位移操作、数字叠加操作等）

3.平方取中法　通过取key求平方以后的中间部分数字

4.折叠法、随机数法

5.除留余数法　f（key）= key mod p

6.md5,sha加密hash算法等

散列冲突的处理：

1.线性探测（开放定址法）f（key）=（key + di）mod m （di = 1，2，3，4，...，m-1）

2.二次探测 f（key）=（key + di）mod m （di = 1^2，-1^2，2^2，-2^2，...，q^2，-q^2） q<=m/2

　二次探测是对线性探测法的改进，在散列冲突位置的前后都可以查找空位置，di取平方是为了让数据更加散列的存储，不会聚集。

3.链地址法 用链表存储组织产生哈希冲突的key（参考HashMap的实现）

素数表：

{3，7，23，47，97，251，443， 911，1471， 42773}

快速的查找：

1.**哈希表**　查重或者统计重复的次数　查询的效率高但是占用内存空间较大

2.**位图法**

位图法，就是用一个比特位（0或者1）来存储数据的状态，比较适合状态简单，数据量比较大，要求内存使用率低的问题场景。

位图法解决问题，首先需要知道待处理数据中的最大值，然后按照size =（maxNumber / 8）(byte)+1的大小来开辟一个char类型的数组，当需要在位图中查找某个元素是否存在的时候，首先需要计算该数字对应的数组中的比特位，然后读取值，**0表示不存在，1表示已存在。**

位图法有一个很大的缺点，就是数据没有多少，但是最大值却很大，比如有10个整数，最大值是10亿，那么就得按10亿这个数字计算开辟位图数组的大小，太浪费内存空间。

3.**Bloom Filter布隆过滤器**

校招面试过程中，搜索查找问题经常会考察到，除了考察二分查找，哈希表，还有BST，AVL，红黑树，跳跃表，前缀树（字典树），倒排索引等等。

在内存有所限制的情况下（如上面的面试问题），快速判断一个元素是否在一个集合（容器）当中，还可以使用布隆过滤器。

布隆过滤器到底是个什么东西呢？通俗来讲，在使用哈希表比较占内存的情况下，它是一种更高级的“位图法”解决方案，之所以说它更高级，是因为它没有上面位图法所说的缺陷。

这里总结一下Bloom Filter的注意事项：
1.Bloom Filter是通过一个位数组+k个哈希函数构成的。
2.Bloom Filter的空间和时间利用率都很高，但是它有一定的错误率，虽然错误率很低，Bloom Filter判断某个元素不在一个集合中，那该元素肯定不在集合里面；Bloom Filter判断某个元素在一个集合中，那该元素有可能在，有可能不在集合当中。
3.Bloom Filter的查找错误率，当然和位数组的大小，以及哈希函数的个数有关系，具体的错误率计算

有相应的公式（错误率公式的掌握看个人理解，不做要求）。
4.Bloom Filter默认只支持add增加和query查询操作，不支持delete删除操作（因为存储的状态位有可能也是其它数据的状态位，删除后导致其它元素查找判断出错）。

Bloom Filter增加元素的过程：把元素的值通过k个哈希函数进行计算，得到k个值，然后把k当作位数组的下标，在位数组中把相应k个值修改成1。

Bloom Filter查询元素的过程：把元素的值通过k个哈希函数进行计算，得到k个值，然后把k当作位数组的下标，看看相应位数组下标标识的值是否全部是1，如果有一个为0，表示元素不存在（判断不存在绝对正确）；如果都为1，表示元素存在（判断存在有错误率）。

很显然，过小的布隆过滤器很快所有的bit位均为1，那么查询任何值都会返回“可能存在”，起不到过滤的目的。布隆过滤器的长度会直接影响误报率，布隆过滤器越长其误报率越小。另外，哈希函数的个数也需要权衡，个数越多则布隆过滤器bit位置为1的速度越快，且布隆过滤器的效率越低；但是如果太少的话，那误报率就会变高。

二叉树（查找）

BST

特点：每一个节点都满足　左孩子的值 < 父节点的值 < 右孩子的值

实现：
BST树插入，删除，查询操作，递归和非递归实现
BST树前序，中序，后序，层序遍历的递归和非递归实现
BST求树的高度，节点元素个数
BST树区间元素查找
判断二叉树是否是一颗BST树
BST求子树问题
BST的LCA问题：求寻找最近公共祖先节点
BST树的镜像反转问题
已知BST树的前序遍历和中序遍历，重建BST树
判断一颗BST树是否是平衡树
求BST树中序遍历倒数第K个节点

AVL

特点：在BST树的基础上，引入了节点“平衡”的概念，任意一个节点的左右子树高度差不超过1，为了维持节点的平衡，引入了四种旋转操作，如下：

1.左孩子左子树太高，做右旋转操作

2.右孩子的右子树太高，做左旋转操作

3.左孩子的右子树太高，做左-右旋转操作（也叫左平衡）

4.右孩子的左子树太高，做右-左旋转操作（也叫右平衡）

红黑树 (TreeSet / TreeMap)

特点：

1.树的每一个节点都有颜色

2.null是黑色

3.root是黑色

4.不能出现连续的红色节点

5.从root根节点到每一个叶子节点的路径上，黑色节点的数量是相同的

操作	AVL	红黑树
平衡树	是	否
增删查时间复杂度	O(logn)	O(logn)
insert最多旋转次数	2	2
remove最多旋转次数	O(logn)	3

B-树、B+树、B*树

m阶-平衡树(Balance)，应用场景：文件索引系统的实现

AVL树 2阶的平衡树 一个节点有两个地址域，一个数据域 一个节点有m个地址域，有m-1个数据域

平衡二叉搜索树和二分查找的时间复杂度效率

B-树的增加和删除过程

B-树和B+树的区别

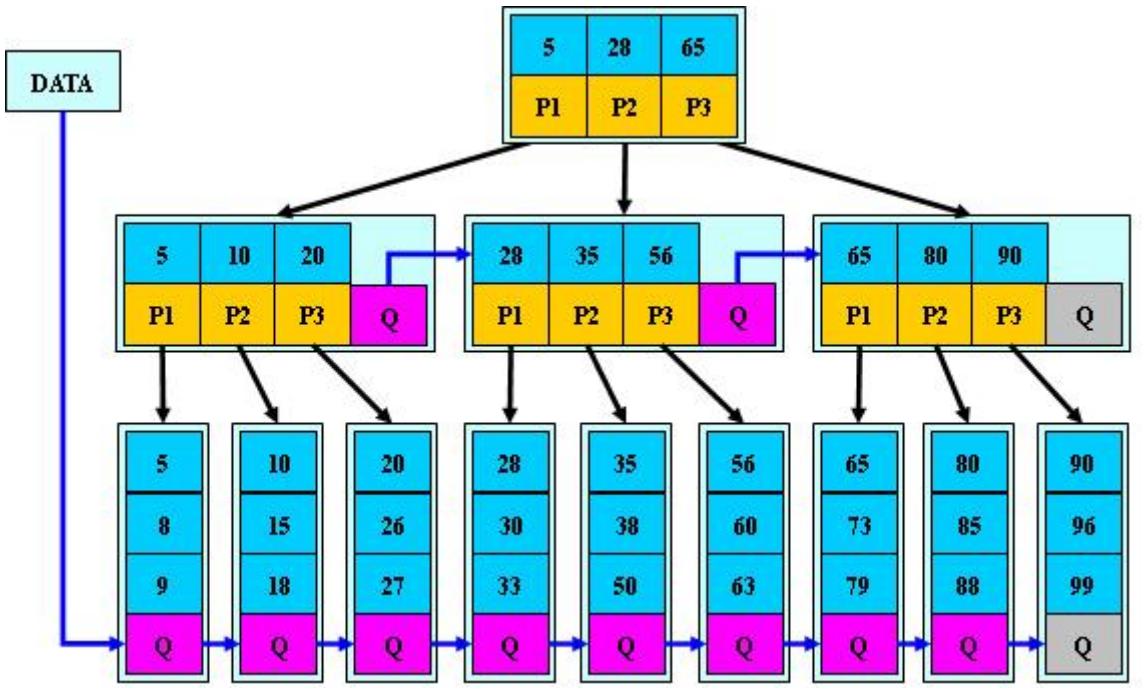
1.B-树的每一个节点，存了关键字和对应的数据地址，而B+树的非叶子节点只存关键字，不存数据地址。因此B+树的每一个非叶子节点存储的关键字是远远多于B-树的，B+树的叶子节点存放关键字和数据，因此，从树的高度上来说，**B+树**的高度要小于**B-树**，使用的**磁盘I/O次数少，因此查询会更快一些。**

2. B-树由于每个节点都存储关键字和数据，因此离根节点近的数据，查询的就快，离根节点远的数据，查询的就慢；B+树所有的数据都存在叶子节点上，因此**在B+**树上搜索关键字，找到对应数据的时间是比较平均的，没有快慢之分。

3. 在B-树上如果做区间查找，遍历的节点是非常多的；B+树所有叶子节点被连接成了有序链表结构，**因此做整表遍历和区间查找是非常容易的。**

B*树

是B+树的变体，在B+树的非根和非叶子结点再增加指向兄弟节点的指针。



B*树定义了非叶子结点关键字个数至少为(2/3)*M，即块的最低使用率为2/3（代替B+树的1/2）。

B+树的分裂：当一个结点满时，分配一个新的结点，并将原结点中1/2的数据复制到新结点，最后在父结点中增加新结点的指针；B+树的分裂只影响原结点和父结点，而不会影响兄弟结点，所以它不需要指向兄弟的指针；

B*树的分裂：当一个结点满时，如果它的下一个兄弟结点未满，那么将一部分数据移到兄弟结点中，再在原结点插入关键字，最后修改父结点中兄弟结点的关键字（因为兄弟结点的关键字范围改变了）；如果兄弟也满了，则在原结点与兄弟结点之间增加新结点，并各复制1/3的数据到新结点，最后在父结点增加新结点的指针；

所以，**B*树分配新结点的概率比B+树要低，空间使用率更高，在B+树基础上，为非叶子结点也增加链表指针，将结点的最低利用率从1/2提高到2/3。**

跳跃表（查找）

跳跃表具有如下性质：

1.由很多层链表组成

2.每一层都是一个有序的链表

3.最底层（level 1）的链表包含所有元素

4.如果一个元素出现在level i层的链表中，则它在level i之下的链表也都会出现

5.每个节点都包含两个指针，一个指向同一链表中的下一个元素，一个指向下面一层的元素

跳跃表的增加、删除、查询操作时间复杂度和红黑树一样，也是O(logn)，相比于红黑树，它的优势是：

1.实现起来更加简单

2.跳跃表的增加、删除操作只会改动局部，不像红黑树的增加、删除操作，因为需要节点重新着色和旋转，可能整棵树都要进行调整，因此在并发环境下，跳跃表加锁的粒度会更小一些，并发能力更强

3.因为跳跃表的每一层都是一个有序的链表，因此范围查找非常方便，优于红黑树的范围搜索的

跳跃表相比于红黑树，是用空间换时间，因此占用的内存空间比红黑树大。

倒排索引（查找）

倒排索引常使用在搜索引擎当中，是搜索引擎为文档内容建立索引，实现内容快速检索必不可少的数据结构。

倒排索引是由单词的集合“词典”和倒排列表的集合“倒排文件”组成的。

倒排索引的存储：内存索引和B+树索引。

理解**正排索引**结构和**倒排索引**结构；掌握**词典、倒排项，倒排列表**的具体实现。

哈夫曼树（Huffman）

哈夫曼树又称为最佳判定树、最优二叉树，是一种**带权路径长度最短的二叉树，常用于数据压缩。**
所谓树的带权路径长度，就是树中所有的**叶子节点的权值** 乘以 **其到根节点的路径长度**，因此树的带权路径长度记为WPL = (W1 * L1 + W2 * L2 + W3 * L3 + ... + Wn * Ln)，N个权值Wi(i=1,2,...,n)构成一颗有N个叶子节点的二叉树，相应的叶子节点的路径的长度是Li(i=1,2,...,n)。

重要概念：路径和长度　　节点的权以及带权路径长度　　树的带权路径长度

应用场景：哈夫曼编码

目的：**为给定的字符集合构建二进制编码，使得编码的期望长度达到最短。**

哈夫曼编码不同于ASCII和Unicode这些字符编码，这些字符集中的码长都采用的是长度相同的编码方案，而哈夫曼编码使用的是**变长编码**，而且哈夫曼编码满足**立刻可解码性**（就是说任一字符的编码都不会是另一个更长字符编码的前缀），这样当一个字符的编码中的位被接收时，可以立即进行解码而无须等待之后的位来决定是否存在另一个合法的更长的编码。

第一张表不满足立刻可解码性，第二张表满足。

字符	编码
A	01
B	1000
C	1010
D	100
E	0

字符	编码
A	01
B	1000
C	0001
D	001
E	1

示例：用如下的字符权值构建哈夫曼树，得到字符序列的哈夫曼编码。

字符	A	B	C	D	E
权值	0.2	0.1	0.1	0.15	0.45

图

有向图是由一个有限的称为**顶点**（vertices）的元素集合以及一个有限的连接每对顶点的**有向边**的集合组成的，简写为digraph。和树的区别是，有向图不需要指定一个根节点，并且一个节点到另一个节点之间可能存在好几条（或者没有）的路径。

在建模通信网络，设备节点，城市节点的时候，可以用图来表示从一个节点通过不同的路径流向另一个节点的过程。在另外一些应用场景当中，节点之间的连接是没有方向的，可以使用图来进行建模，也称作**无向图**。

节点的入度：终点是当前节点的边的个数

节点的出度：起点是当前节点的边的个数

图的深度遍历和广度遍历

求不带权值有向图最短路径 - 广度遍历的应用

最短路径-Dijkstra算法

迪杰斯特拉算法是典型的单源最短路径算法，用于计算一个节点到其它所有节点的最短路径。主要特点是以起始点为中心向外层层扩展，直到扩展到终点为止。

迪杰斯特拉算法实际上是计算了起始节点到其它所有节点的最短路径。

最小生成树-Prim算法和Kruskal算法

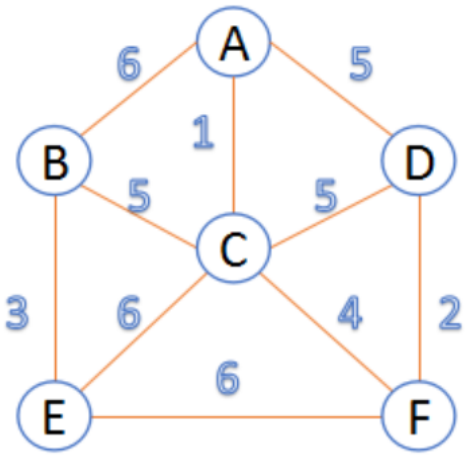
连通图：在无向图中，若任意两个顶点Vi和Vj都有路径相通，则称该无向图为连通图。

强连通图：在有向图中，若任意两个顶点Vi和Vj都有路径相通，则称该有向图为强连通图。

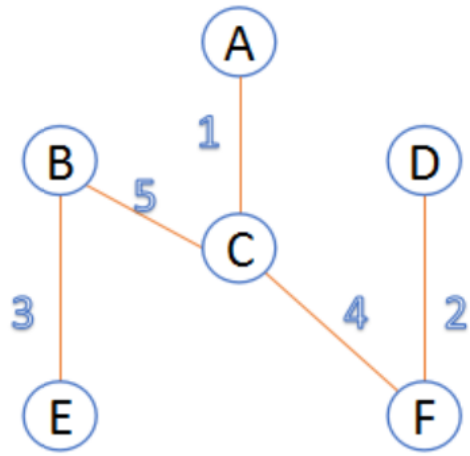
连通网：在连通图中，若图的边具有一定的意义，每一条边都对应着一个数，称为权值，权值代表着连接各个顶点的代价，称这种连通图叫做连通网。

生成树：一个连通子图，它包含连通图中全部n个顶点，有n-1条边。如果生成树中再添加一条边，则必定成环。

最小生成树：在连通网的所有生成树中，所有边的代价之和最小的生成树，称为最小生成树。



连通网G



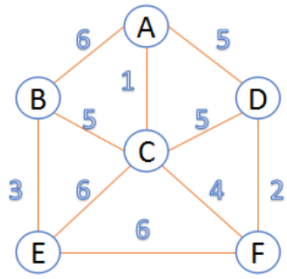
最小生成树

Prim算法：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。算法从某一顶点s开始，逐渐增长覆盖整个连通网的所有顶点。

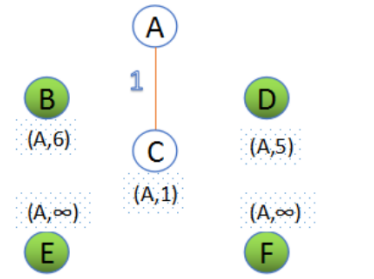
1.图的所有顶点集合是V，初始令集合u={s}，v=V-u。

2.在两个集合u，v能够组成的边中，选择一条代价最小的边（ui, vi）加入到最小生成树当中，并把vi加入到集合u中，更新v集合中剩余顶点的权值为最小代价。

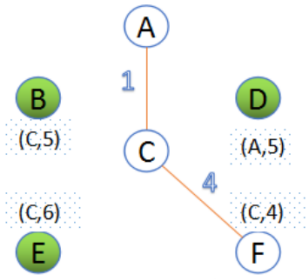
3.重复步骤2，直到最小生成树有n-1条边或者n个顶点为止。



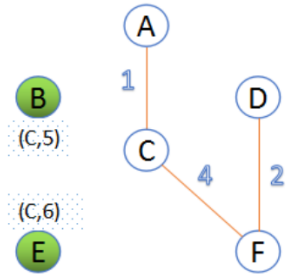
连通网G



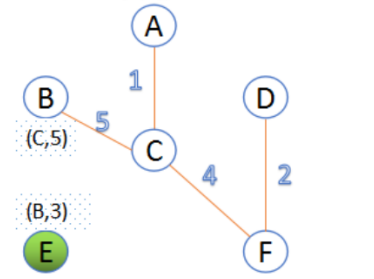
1.初始u={A},v={B,C,D,E,F};顶点B下方(A,6),表示与集合u中A的代价为6作为最小代价边。选择最小的代价边(A,C),把C并入到集合u中。



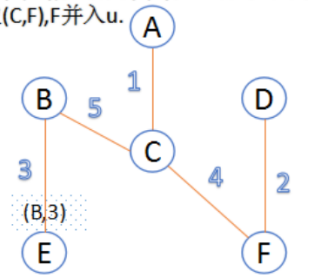
2.u={A,C},v={B,D,E,F};更新v中顶点与集合u的最小的代价边;例如:顶点E之前为(A,∞),更新为(C,6)；选择最小代价边(C,F),F并入u.



3.u={A,C,F},v={B,D,E};更新v中顶点与集合u的最小的代价边;选择最小代价边(F,D),D并入u.



4.u={A,C,F,D},v={B,E};更新v中顶点与集合u的最小的代价边;选择最小代价边(C,B),B并入u.



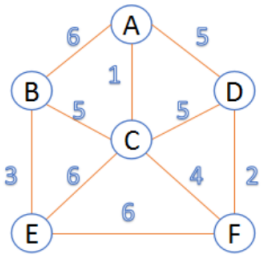
5.u={A,C,F,D,B},v={E};更新v中顶点与集合u的最小的代价边;选择最小代价边(B,E),E并入u.

Kruskal算法：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，步骤如下：

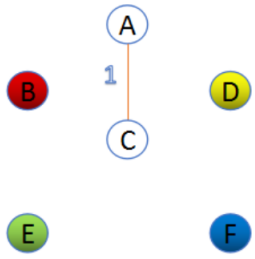
1.把图中的所有边按权值从小到大排序，把图中的n个顶点看成独立的n颗树组成的森林

2.按权值从小到大选择边，所选的边连接的两个顶点Vi和Vj应属于两颗不同的树，则称为最小生成树中的一条边，并合并成一棵树

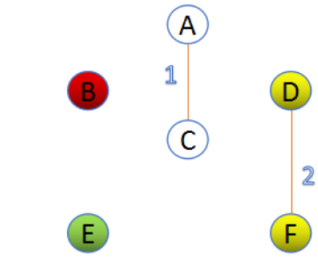
3.重复步骤2，直到所有顶点都在一棵树内或者有n-1条边为止。



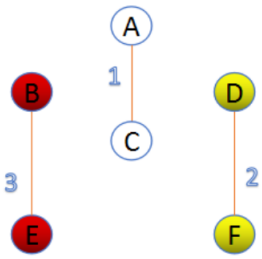
连通网G



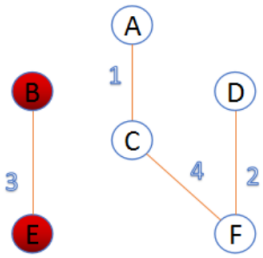
1.选择代价最小的边(A,C);并保证A,C不在同一颗树上,然后合并A,C



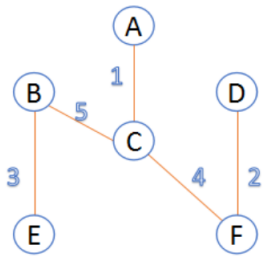
2.选择代价最小的边(D,F);并保证D,F不在同一颗树上,然后合并D,F



3.选择代价最小的边(B,E);并保证B,E不在同一颗树上,然后合并B,E



4.选择代价最小的边(C,F);并保证C,F不在同一颗树上,然后合并C,F所在的树



5.选择代价最小的边(A,D),顶点A,D在同一颗树上,丢弃;
选择最小的边(C,D),顶点C,D在同一颗树上,丢弃;
选择最小的边(B,C),顶点B,C不在同一颗树上,加入此边,然后合并B,C所在的树,此时所有顶点在同一颗树上，返回;

排序算法

关注算法的**时间复杂度（平均，最优，最差）**和**空间复杂度**和**稳定性。**

冒泡、选择、插入排序、希尔排序

排序性能对比:

排序算法	**10000组（单位：ms）**
冒泡排序（效率最低）	160
选择排序（效率次之）	50
插入排序（效率最高）	30
希尔排序（效率更高）	6

排序算法	**平均时间复杂度**	**最好时间复杂度**	**最坏时间复杂度**	**空间复杂度**	**稳定性**
冒泡排序	O(n^2)	O(n)	O(n^2)	O(1)	稳定
选择排序	O(n^2)	O(n^2)	O(n^2)	O(1)	不稳定
插入排序	O(n^2)	O(n)	O(n^2)	O(1)	稳定

希尔排序	依赖不同的增量序列设置	O(n)	O(n^2)	O(1)	不稳定
排序算法	**平均时间复杂度**	**最好时间复杂度**	**最坏时间复杂度**	**空间复杂度**	**稳定性**
	O(n^1.3)				

插入排序的效率最好，尤其是在数据已经趋于有序的情况下，采用**插入排序效率最高。**

一般中等数据量的排序都用希尔排序，选择合适的增量序列，效率就已经不错了，如果数据量比较大，可以选择高级的排序算法，如快速排序。

堆排序

二叉堆：就是一颗**完全二叉树**，分为两种典型的堆，分别是**大根堆**和**小根堆**

满足 **0 <= i <= (n-1)/2**，n代表最后一个元素的下标。

如果arr[i] <= arr[2 * i+1] && arr[i] <= arr[2 * i+2]，就是小根堆

如果arr[i] >= arr[2 * i+1] && arr[i] >= arr[2 * i+2]，就是大根堆

【掌握】：**大根堆，小根堆，优先级队列，堆排序**

快速排序

冒泡排序的升级算法。

每次选择基准数，把小于基准数的放到基准数的左边，把大于基准数的放到基准数的右边，采用"分治思想"处理剩余的序列元素，直到整个序列变为有序序列。

算法效率提升:
1、对于小段趋于有序的序列采用插入排序
2、三数取中法。旨在挑选合适的基准数，防止快排退化成冒泡排序
3、随机数法

归并排序 - 外部排序

也采用"分治思想"，先进行序列划分，再进行元素的有序合并。

排序算法	平均时间复杂度	最好时间复杂度	最坏时间复杂度	空间复杂度	稳定性
堆排序	O（n*logn）	O（n*logn）	O（n*logn）	O（1）	不稳定
快速排序	O（n*logn）	O（n*logn）	O（n^2）	O（logn）-O（n）	不稳定
归并排序	O（n*logn）	O（n*logn）	O（n*logn）	O（n）	稳定

数据规模	快速排序	归并排序	希尔排序	堆排序
1000万	0.75秒	1.22秒	1.77秒	3.57秒

数据规模	快速排序	归并排序	希尔排序	堆排序
5000万	3.79秒	6.29秒	9.49秒	26.54秒
1亿	7.65秒	13.06秒	18.79秒	61.31秒

【2015阿里巴巴研发工程师笔试题】个数约为50K的数列需要进行从小到大排序，数列特征是基本逆序(多数数字从大大小，个别乱序)，以下哪种排序算法在事先不了解数列特征的情况下性能最优。（）

A. 冒泡排序　　B. 改进冒泡排序　　C. 选择排序　　D. 快速排序　　E. 堆排序　　F.插入排序

【2016阿里巴巴校招笔试题】现有1GB数据进行排序，计算资源只有1GB内存可用，下列排序方法中最可能出现性能问题的是（）

A. 堆排序　　B. 插入排序　　C. 归并排序　　D. 快速排序　　E. 选择排序　　F. 冒泡排序

【京东】假设你只有100Mb的内存，需要对1Gb的数据进行排序，最合适的算法是（）

A. 归并排序　　B. 插入排序　　C. 快速排序　　D. 冒泡排序

Arrays.sort和Collections.sort，使用的都是称作Timsort排序算法，归并排序+插入排序的综合排序算法。

基数排序

排序算法	平均时间复杂度	最好时间复杂度	最坏时间复杂度	空间复杂度	稳定性
基数排序	O（nd）	O（nd）	O（nd）	O（n）	稳定

串-模式匹配算法

朴素模式匹配算法

BF算法：暴力匹配（BF）算法是普通的模式匹配算法，BF算法的思想就是将**目标串S**的第一个字符与**模式串T**的第一个字符进行匹配，若相等，则继续比较S的第二个字符和T的第二个字符；若不相等，则比较S的第二个字符和T的第一个字符，依次比较，直到得出最后的匹配结果。

BF算法的时间复杂度：　最好：O(1)　　差一点：O（n+m）　　最坏的：O（（n-m+1）*m）

KMP模式匹配算法

1.KMP算法的特点是主串的i值不回溯，只改变子串的j值。

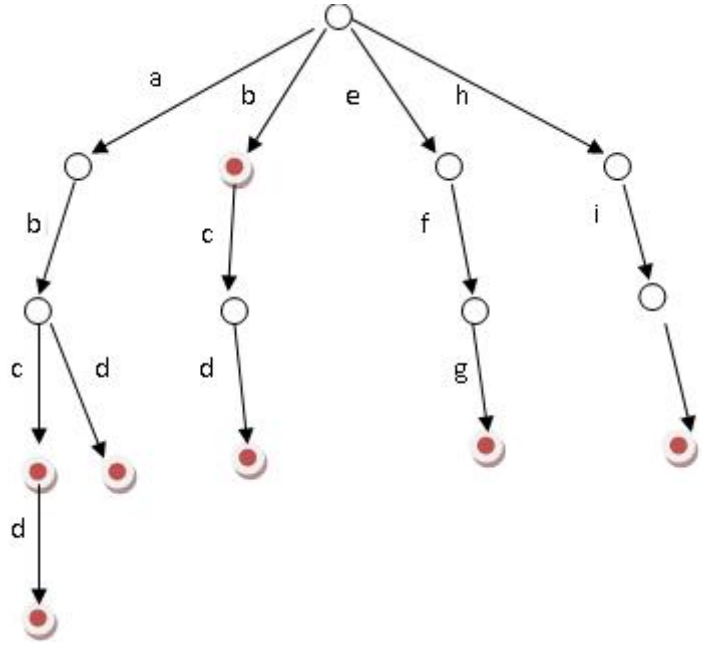
2.KMP算法对子串的每一个字符，都求出了该字符前面串的最长公共前后缀的长度（k值），放在一个称作next的数组里面。

3.时间复杂度 O（n+m）

Trie字典树（前缀树）

基本性质：　搜索的时间复杂度O（m） m表示串的长度

1.根节点不包含字符，除根节点外每一个节点都只包含一个字符
2.从根节点到某一结点，路径上经过的字符连接起来，为该节点对应的字符串
3.每个节点的所有子节点包含的字符都不相同

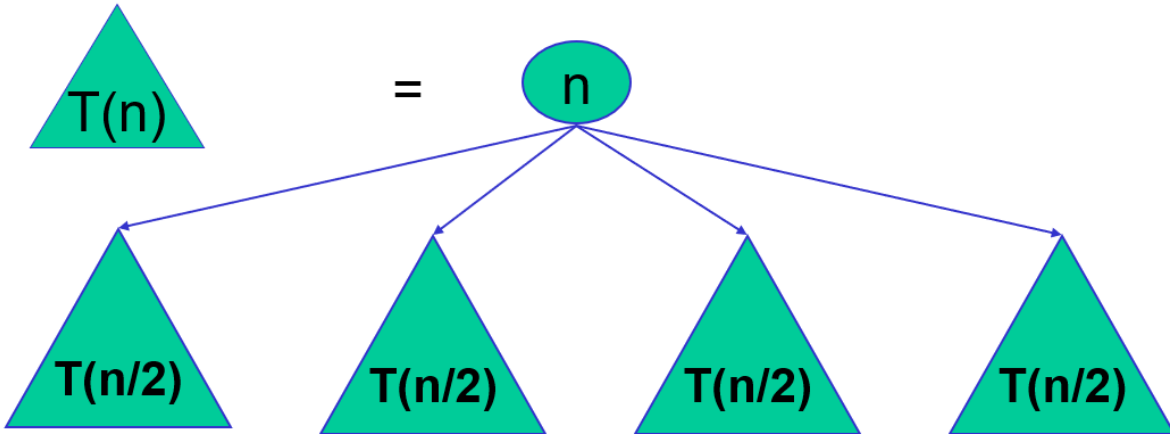


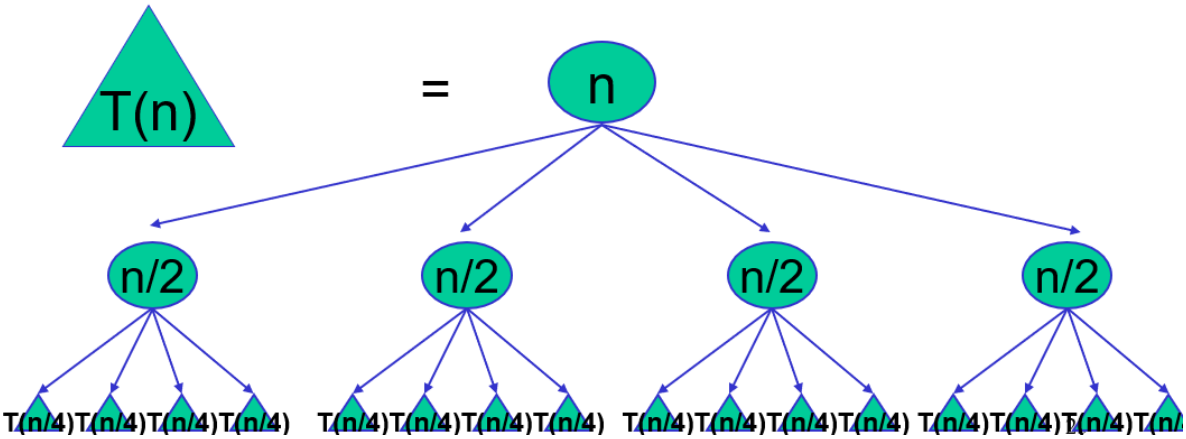
应用场景：串的快速检索、串排序、前缀搜索（单词自动完成）

经典五大算法

分治算法

分治算法思想：
规模为n的原问题的解无法直接求出，进行问题规模缩减，划分子问题（这里子问题相互独立而且和原问题解的性质是相同的，只是问题规模缩小了）。如果子问题的规模仍然不够小，再进行子问题划分，如此**递归**的进行下去，直到问题规模足够小，很容易求出其解为止，最后将求出的小规模的问题的解合并为一个更大规模的问题的解，自底向上逐步求出原问题的解。





分治算法适用条件：

分治法所能解决的问题一般具有以下几个特征：

1.原问题的规模缩小到一定的程度就可以容易地解决

2.原问题可以分解为若干个规模较小的相同问题，即原问题具有**最优子结构性质**

3.利用原问题分解出的子问题的解可以合并为原问题的解

4.原问题所分解出的各个子问题是相互独立的，即子问题之间**不包含公共的子问题**（这条特征涉及到分治法的效率，如果各个子问题不独立，也就是子问题划分有重合的部分，则分治法要重复的求解公共子问题的解，此时虽然也可用分治法，但采用**动态规划**更好）

分治算法问题案例：

二分搜索

快速排序

归并排序

快排分割函数求top K问题

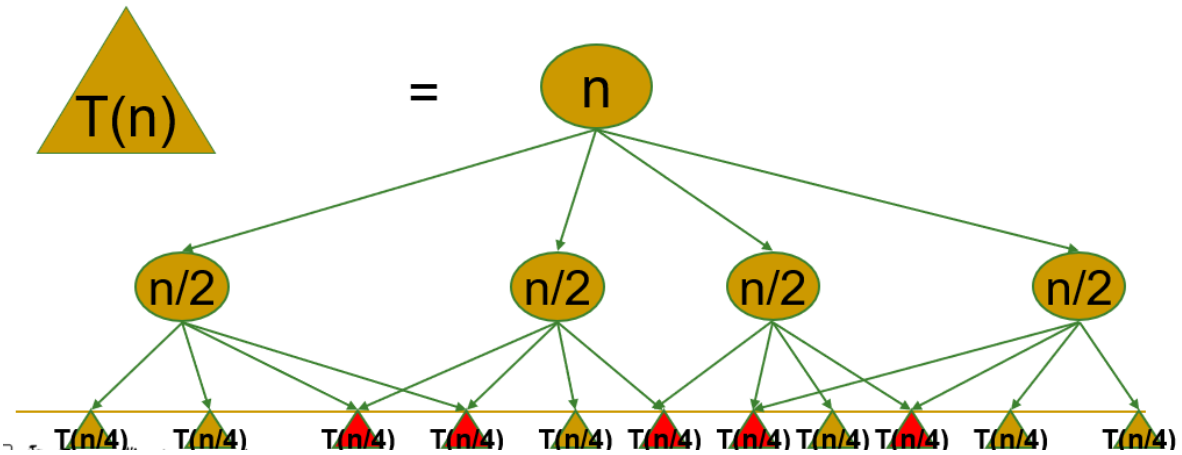
中位数（要求O(logn)对数时间）

动态规划算法

动态规划算法思想：

算法的基本思想与分治算法类似，也是将待求解的问题划分为若干子问题，按划分的顺序求解子阶段问题，前一个子问题的解，为后一子问题的求解提供了有用的信息(**最优子结构**)。在求解任一子问题时，列出各种可能的局部解，通过决策保留那些有可能达到最优的局部解，丢弃其它局部解。依次解决各个子问题，最后求出原问题的最优解。

与分治算法最大的区别是：**适合于用动态规划算法求解的问题，经分解后得到的子问题往往不是互相独立的。**



动态规划求解问题的基本步骤：

动态规划所处理的问题是一个多阶段决策问题，一般由初始状态开始，通过对中间阶段决策的选择，达到结束状态。动态规划算法的代码设计都有一定的模式，一般都要经过以下几个步骤：

初始状态 -> 决策1 -> 决策2 -> ... -> 决策n -> 结束状态

1.找出最优解的性质，并刻划其结构特征。（**找问题状态**）
2.递归地定义最优值。（**找状态转移方程**）
3.自底向上的方式计算出最优值。
4.根据计算最优值时得到的信息，构造最优解。

动态规划算法问题案例：

硬币选择问题

最大子段和

最长非降子序列LIS

最长公共子序列LCS

0-1背包

三角数组求和

回溯算法

算法思想：在包含问题的所有解的解空间树中，按照深度优先搜索的策略，从根节点出发深度搜索解空间树。当搜索到某一节点时，要先判断该节点是否包含问题的解，如果包含就从该节点出发继续深度搜索下去，否则逐层向上回溯。一般在搜索的过程中都会添加相应的**剪枝函数，避免无效解的搜索，提高算法效率。**

解空间：解空间就是所有解的可能取值构成的空间，一个解往往包含了得到这个解的每一步，往往就是对应解空间树中一条从根节点到叶子节点的路径。**子集树和排列树**都是一种解空间，它们不是真实存在的数据结构，也就是说并不是真的有这样一颗树，只是抽象出的解空间树。

子集树和**排列树**

整数选择问题

2N整数选择问题

整数选择问题二

0-1背包

分支限界算法

分支限界法类似于回溯算法，是在问题的解空间树上搜索问题解的算法，主要体现在两点不同：
1.**求解目标不同。**回溯算法的求解目标是找出解空间树中满足约束条件的所有解，而分支限界法的求解目标是找出满足约束条件的一个解，或者是在满足约束条件的解中找出某种意义下的最优解。
2.**搜索解空间树的方式不同。**回溯算法以深度优先的方式搜索解空间树，而分支限界法则以广度优先或者以最小耗费优先的方式搜索解空间树。

分支限界算法基本思想：
分支限界法常以广度优先或以最小耗费（最大效益）优先的方式搜索问题的解空间树。在分支限界法中，每一个活结点只有一次机会称为扩展节点，活结点一旦成为扩展节点，就一次性产生其所有儿子节点（**分支**），在这些儿子节点中，导致不可行解或是导致非最优解的儿子节点会被舍弃掉，其余儿子节点会被加入活结点表中。

为了有效的选择下一个扩展节点加速搜索，在每一个活结点处计算一个函数值（**限界**），并根据计算的函数值结果从当前活结点表中取下一个最有利的节点成为当前的扩展节点，使搜索朝着解空间树上最优解的分支推进。重复上述节点扩展过程，直到找到所需的最优解或者活结点表为空。

扩展节点：一个正在产生儿子的节点称作扩展节点
活结点：一个自身已经生成，但其儿子还没有全部生成的节点
死结点：一个所有儿子已经产生的节点

深度优先搜索是对一个扩展节点R，一旦产生了它的一个儿子C，就把C当作新的扩展节点。在完成对子树C的深度搜索之后回溯到R时，将R重新变成扩展节点，继续生成R的下一个儿子。

广度优先搜索是在一个扩展节点R变成死节点之前，它一直是扩展节点。

从活结点表中选择下一个扩展节点时，不同的方式会导致不同的分支限界法，常见有：
1.队列式（FIFO）分支限界法
a.一开始，根结点是唯一的活结点，根结点加入活结点队列。
b.从活结点队列中取出队头结点后，作为当前扩展结点。
c.对当前扩展结点，先从左到右产生它的所有孩子节点，**用约束条件检查**，把所有满足约束函数的孩子节点加入活结点队列中。
d.再从活结点表中取出队首结点为当前扩展结点，重复上述过程，直到找到一个解或活结点队列为空为止。

2.优先级队列式分支限界法
a.对每一活结点计算一个优先级（某些信息的函数值）。
b.根据这些优先级从当前活结点表中优先选择一个优先级最高（最有利）的结点作为扩展结点，使搜索朝着解空间树上有最优解的分支推进，以便尽快地找出一个最优解。
c.对当前扩展结点，先从左到右产生它的所有孩子节点，**用约束条件检查**，对所有满足约束函数的孩子节点计算优先级并加入到活结点优先级队列中。
d.再从活结点表中取出下一个优先级最高的结点为当前扩展结点，重复上述过程，直到找到一个解或活结点队列为空为止。

0-1背包问题

装载问题（有一批共n个集装箱要装上2艘载重量分别为c1，c2的轮船，其中集装箱i的重量为wi，且要求确定是否有一个合理的装载方案可将这n个集装箱装上这2艘轮船）

贪心算法

当一个问题具有最优子结构性质时，可以使用动态规划法求解，但有时候使用贪心算法更简单，更直接而且解决问题的效率很高。

例如前面动态规划算法中的硬币问题就可以用贪心算法来解决，从算法名字上来看，贪心算法总是做出在当前看来最好的选择，也就是说贪心算法**并不从整体最优考虑**，它所做出的选择只是在某种意义上的**局部最优选择**，当然最终希望贪心算法得到的最终结果也是最优的。

虽然贪心算法不能对所有问题都得到整体最优解，但是对于很多问题它能够产生整体最优解，或者是趋近于最优解。

硬币问题

问题：最优装载问题，给出n个物体，第i个物体重量为wi。选择尽量多的物体，使得总重量不超过C。

问题：部分背包问题，有n个物体，第i个物体的重量为wi，价值为vi。在总重量不超过C的情况下让总价值尽量高。每一个物体都可以只取走一部分，价值和重量按比例计算。求最大总价值。

问题：乘船问题，有n个人，第i个人重量为wi。每艘船的最大载重量均为C，且最多只能乘两个人。用最少的船装载所有人。求需要最少的船的数量。

问题：m个柜台提供服务，每个柜台给一个用户提供服务的时间是t(用数组表示每一个柜台提供服务的时间)，问怎么排列，使得柜台给所有用户提供服务的时间最少。

本课程中学过的下面算法，都使用的是贪心算法思想：

哈夫曼编码

Dijkstra算法

最小生成树-Prim算法

最小生成树-Kruskal算法