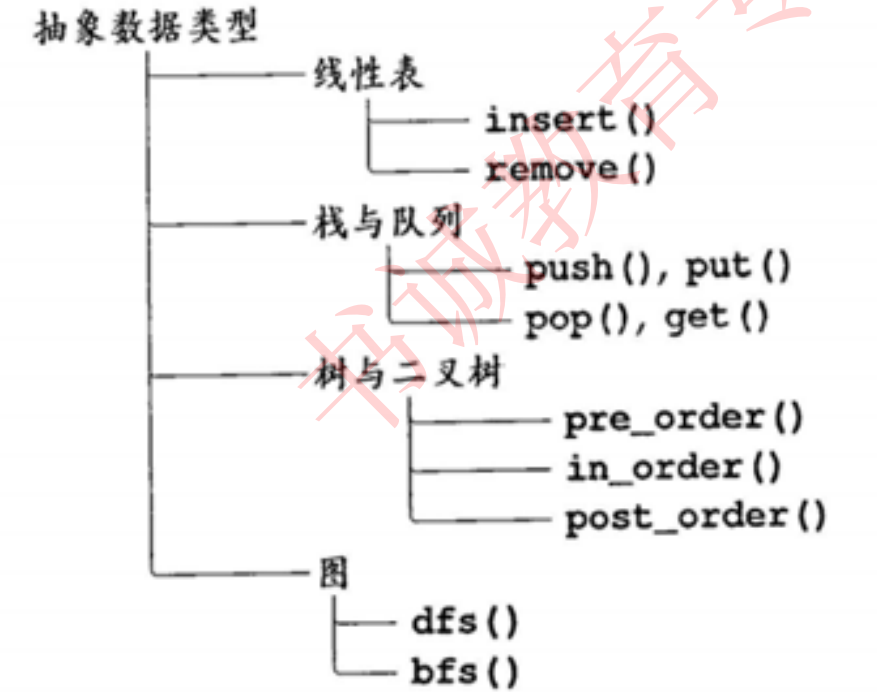
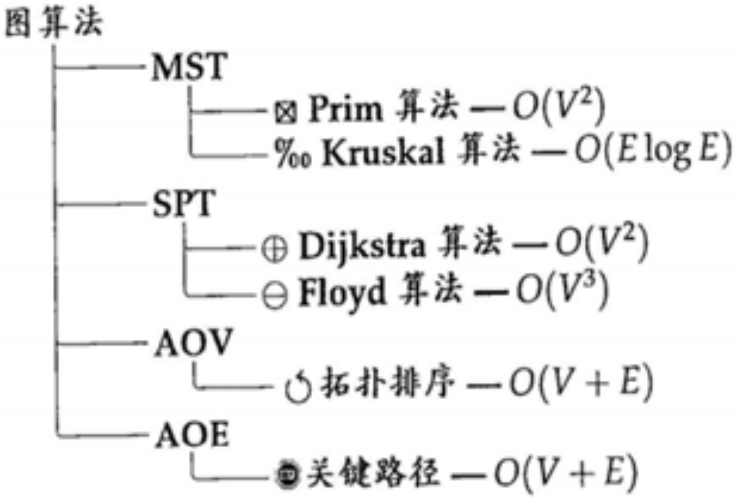
数据结构知识点总结

内容概要：

基本概念——线性表——栈与队列——树与二叉树——图——查找算法——排序算法

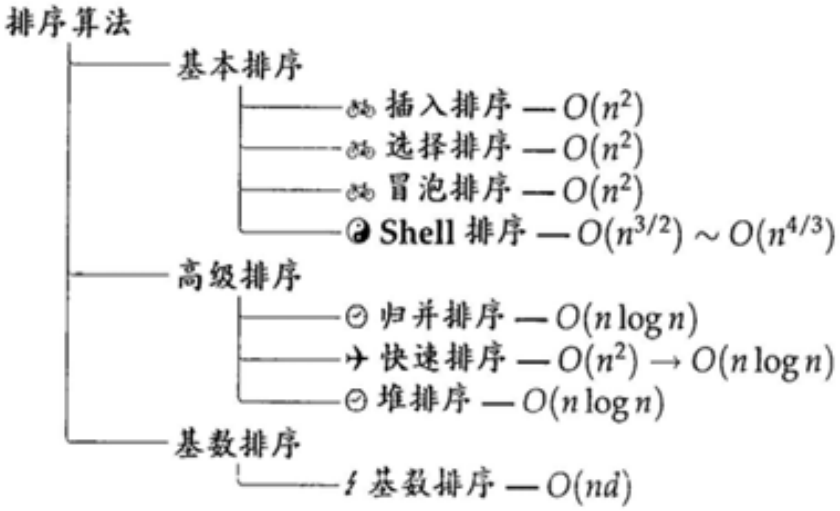


图算法



查找算法

排序算法



一、 基本概念

1、数据元素是数据的基本单位。

2、数据项是数据不可分割的最小单位。

3、数据结构的

逻辑结构（抽象的，与实现无关）

物理结构（存储结构） 顺序映像（顺序存储结构）位置“相邻”

非顺序映像（链式存储结构）指针表示关系

4、算法特性：算法具有正确性、有穷性，确定性，（可行性）、输入，输出

正确性：能按设计要求解决具体问题，并得到正确的结果。

有穷性：任何一条指令都只能执行有限次，即算法必须在执行有限步后结束。

确定性：算法中每条指令的含义必须明确，不允许由二义性

可行性：算法中待执行的操作都十分基本，算法应该在有限时间内执行完毕。

输入：一个算法的输入可以包含零个或多个数据。

输出：算法有一个或多个输出

5、算法设计的要求：

（1）正 确 性：算法应能满足设定的功能和要求 。

（2）可 读 性：思路清晰、层次分明、易读易懂 。

（3）健 壮 性：输入非法数据时应能作适当的反应和处理。

（4）高 效 性（时间复杂度）：解决问题时间越短，算法的效率就越高。

（5）低存储量（空间复杂度）：完成同一功能，占用存储空间应尽可能少。

二、 线性表

1、线性表 List：最常用且最简单的数据结构。

含有大量记录的线性表称为文件。

2、线性表是 n 个数据元素的有限序列。

线性结构的特点： ①“第一个” ②“最后一个” ③前驱 ④后继。

3、顺序表——线性表的顺序存储结构

特点

a) 逻辑上相邻的元素在物理位置上相邻。

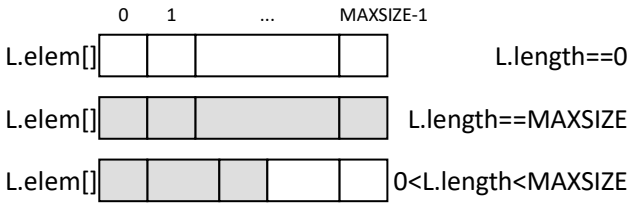
b) 随机访问。

1) **typedef struct**{

DataType elem[MAXSIZE];

int length;

} SqList;



2) 表长为 n 时，线性表进行插入和删除操作的时间复杂度为 O（n）‘

插入一个元素时大约移动表中的一半元素。

删除一个元素时大约移动表中的（n-1）\2

4、线性表的链式存储结构

1) 类型定义

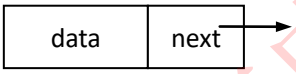
简而言之，“数据 + 指针”。

typedef struct LNode {

DataType data;

struct LNode *next;

} LNode, *LinkList;



2) 不带头结点的空表判定为 L= =null

带头结点的空表判定为 L->next= =null

循环单链表为空的判定条件为 L.next= =L

线性链表的最后一个结点的指针为 NULL

头结点的数据域为空，指针域指向第一个元素的指针。

5、顺序表和单链表的比较

顺序表	单链表
以地址相邻表示关系	用指针表示关系
随机访问，取元素 O(1)	顺序访问，取元素 O(n)
插入、删除需要移动元素 O(n)	插入、删除不用移动元素 O(n)(用于查找位置)

6、顺序存储：优点：存储密度大，可随机存储

缺点：大小固定；不利于增减节点;存储空间不能充分利用；容量难扩充

链式存储：优点：易于插入删除；可动态申请空间；表容量仅受内存空间限制

缺点：增加了存储空间的开销；不可以随机存储元素

三、 栈与队列

1、栈

栈：限定仅在表尾进行插入或删除操作的线性表。

栈顶：表尾端

栈底：表头

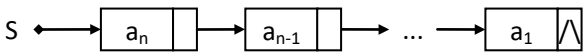
栈是先进后出的线性表。

插入栈顶元素称为入栈，删除栈顶元素称为出栈。

2、栈分为链栈和顺序栈

• 链栈

用不带头结点的单链表实现。



• 顺序栈

类似于顺序表，插入和删除操作固定于表尾。

进栈: 进栈运算是在栈顶位置插入一个新元素x。

算法思想:

(a) 判栈是否为满，若栈满，作溢出处理，并返回0;

(b) 若栈未满，栈顶指针top加1;

(c) 将新元素x送入栈顶，并返回1。

算法实现:

```
int Push (SeqStack *s, datatype x)
    { if （s->top= =MAXLEN–1）
          return 0;              // 栈满不能入栈，且返回 0
      else { s->top++;
             s->data[s->top]=x;  // 栈不满，则压入元素x
             return 1; }         // 进栈成功，返回1
    }
```

出栈: 出栈运算是指取出栈顶元素，赋给某一个指定变量x。

算法步骤:

(a) 判栈是否为空，若栈空，作下溢处理，并返回0;

(b) 若栈非空，将栈顶元素赋给变量x;

(c) 指针top减1，并返回1。

算法实现:

```
datatype Pop (SeqStack *s)
      {  datatype x;
         if (SEmpty ( s ) )
           return 0;  // 若栈空不能出栈 ，且返回0
         else { x=s->data[s->top];
// 栈不空则栈顶元素存入*x
           s->top - -;
           return x; }     // 出栈成功，返回1
           }
```

3、队列

先进先出的线性表。

队尾入队 对头出队

允许插入的一端叫做队尾

允许删除的一端叫做对头

4、链队列

```
typedef struct queuenode
 {datatype  data;
  struct  queuenode  *next;
  }queuenode;      // 链队结点的类型datatype
typedef  struct
 {queuenode  *front,*rear;
 }linkqueue;     // 将头指针、尾指针封装在一起的链队
```

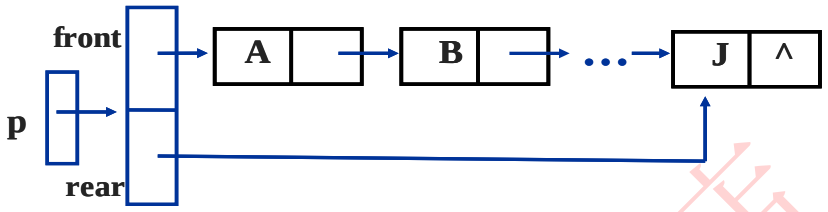


图4-6　链队列示意图

-

3．链队的基本运算：

（1）入队（进队）

```
void InQueue (linkqueue *q,queuenode *p)
    // 进队函数
{int x;                  // 定义入队元素类型
 queuenode *p=malloc (sizeof ( queuenode) );
 p->data=x;              // 输入元素
 p->next=NULL;           // 修改指针
 if (q->front= =NULL)
       q->front=p;
 else
       q->rear->next=p;
 q->rear=p;
}
```

（2）出队

```
int OutQueue(linkqueue *q, datatype *V)  //
   出队函数
{queuenode *p;
 if (q->front= =NULL)          // 判队空
      return 0;
 else
  {p=q->front;          // 若队不空，则作出
 队处理
   v=p->data;
   q->front=p->next;
   if(q->front==NULL)
      q->rear=NULL;
   free ( p );          // 回收结点
   return 1;            // 返回1
  }
}
```

5、 循环队列

```
typedef struct {
    DataType elem[MAXSIZE];
    int front, rear;                    //  队头、队尾位置
} SqQueue;
```

• 循环队列判断队空的条件为 front=rear

循环队列判断队满的条件为 （rear+1）%m=front

在一个循环队列中删除元素时，首先需要后移队首指针。

6、栈与队列比较：都是线形结构，栈的操作 LIFO（后进先出），队列操作 FIFO（先进先出）。

四、 树和二叉树

1. 树的定义

树（Tree）：是 n（n≥0）个有限数据元素的集合。

在任意一棵非空树 T 中：

（1）有且仅有一个特定的称为树根（Root）的结点，根结点无前趋结点；

（2）当 n>1 时，除根结点之外的其余结点被分成 m(m>0)个互不相交的集合 T1，T2，…，Tm，其中每一个集合 Ti（1≤ i ≤m）本身又是一棵树，并且称为根的子树。

2. 基本术语：

结点的度数：结点的非空子树（即后缀）个数叫作结点的度数

树叶、分支结点：左（右）子树均为空二叉树的结点称作树叶否则称作分支结点。

结点的层数：规定根的层数是 0，其余结点的层数等于其父结点的层数加 1

孩子和双亲：

树的深度：

树的度数：树中度数最大的结点度数叫作树的度数

树林：是由零个或多个不相交的树所组成的集合。

3. 二叉树性质：

1) 二叉树的第 i 层上至多有 2^{i-1} 个结点。

2) 深度为 k 的二叉树至多有 2^k-1 个结点。

满二叉树：深度为 k，有 2^k-1 个结点。

完全二叉树：给满二叉树的结点编号，从上至下，从左至右，n 个结点的完全二叉树中结点在对应满二叉树中的编号正好是从 1 到 n。

3) 叶子结点 n_0，度为 2 的结点为 n_2，则 $n_0 = n_2+1$。

考虑结点个数：$n = n_0 + n_1 + n_2$

考虑分支个数：$n-1 = 2n_2 + n_1$

可得 $n_0 = n_2+1$

4) n 个结点的完全二叉树深度为 $\lfloor \log n \rfloor + 1$ **。**

5) n 个结点的完全二叉树，结点按层次编号

有： i 的双亲是$\lfloor n/2 \rfloor$，如果 i = 1 时为根（无双亲）；

i 的左孩子是 2i，如果 2i>n，则无左孩子；

i 的右孩子是 2i + 1，如果 2i + 1>n 则无右孩子。

4. 二叉树的存储结构

• 顺序存储结构

用数组、编号 i 的结点存放在[i-1]处。适合于存储完全二叉树。

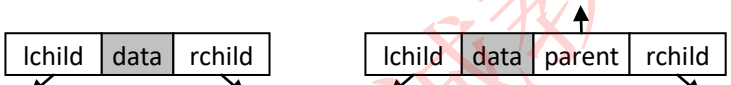
• 链式存储结构

二叉链表：

```
typedef struct BTNode {
   DataType data;
   struct BTNode *lchild, *rchild;
} BTNode, *BinTree;
```

三叉链表：

```
typedef struct BTNode {
   DataType data;
   struct BTNode *lchild, *rchild, *parent;
} BTNode, *BinTree;
```



在链式存储结构中，含有 n 个结点的二叉链表有 n+1 个空链域。

5. 遍历二叉树（先序 DLR、中序 LDR、后序 LRD）方法与 C 语言描述

由二叉树的递归定义可知，一棵二叉树由根结点（D）、根结点的左子树（L）和根结点的右子树（R）三部分组成。因此，只要依次遍历这三部分，就可以遍历整个二叉树。一般有三种方法：先序(前序)遍历 DLR（根左右）、中序遍历 LDR（左根右）、 后序遍历 LRD（左右根）。

1. 先序遍历（DLR）的递归过程

```
void Preorder(BT *T)              // 先序遍历二叉树BT
 {  if (T= =NULL) return;         // 递归调用的结束条件
      { printf(T->data);          // 输出结点的数据域
        Preorder(T->lchild);      // 先序递归遍历左子树
        Preorder(T->rchild);      // 先序递归遍历右子树
      }
 }
```

2. 中序遍历（LDR）的递归过程

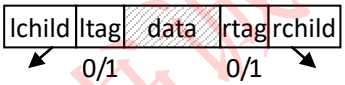
```
void Inorder(BT *T)               // 中序遍历二叉树BT
 { if (T= =NULL) return;          // 递归调用的结束条件
      { Inorder(T->lchild);       // 中序递归遍历左子树
        printf(T->data);          // 输出结点的数据域
        Inorder(T->rchild);       // 中序递归遍历右子树
      }
 }
```

3. 后序遍历（LRD）的递归过程

```
void Postorder(BT *T)             // 后序遍历二叉树BT
 { if (T= =NULL) return;          // 递归调用的结束条件
     { Postorder(T->lchild);      // 后序递归遍历左子树
       Postorder(T->rchild);      // 后序递归遍历右子树
       printf(T->data);           // 输出结点的数据域
     }
 }
```

6. 线索二叉树

n 个结点的二叉链表中有 n+1 个空指针，可以利用其指向前驱或后继结点，叫**线索**，同时需附加一个标志，区分是子树还是线索。



lchild	有左子树，则指向左子树，标志 ltag == 0； 没有左子树，可作为前驱线索，标志 ltag == 1。
rchild	有右子树，则指向右子树，标志 rtag == 0； 没有右子树，可作为后继线索，标志 rtag == 1。

7. 树和森林

树的存储结构

双亲表示法，孩子表示法，孩子兄弟表示法。

特点：双亲表示法容易求得双亲，但不容易求得孩子；孩子表示法容易求得孩子，但求双亲麻烦；两者可以结合起来使用。孩子兄弟表示法，容易求得孩子和兄弟，求双亲麻烦，也可以增加指向双亲的指针来解决。

树与二叉树的转换

表 5.1 树和二叉树的对应关系

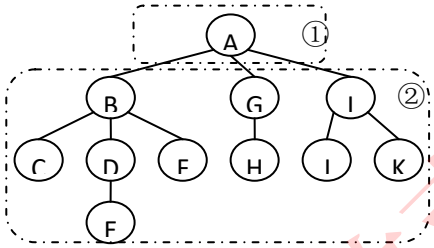
树	对应的二叉树
根	根
第一个孩子	左孩子
下一个兄弟	右孩子

树的遍历

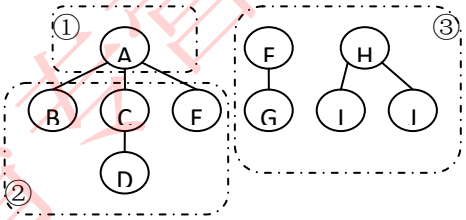
树的结构：①根，②根的子树。
先根遍历：①②。例：ABCDEFGHIJK。
后根遍历：②①。例：CEDFBHGJKIA。

遍历森林

森林的结构：①第一棵树的根，②第一棵树的根的子树森林，③ 其余树(除第一棵外)组成的森林。
先序遍历：①②③。例：ABCDEFGHIJ。
中序遍历：②①③。例：BDCEAGFIJH。
注：先序遍历森林，相当于依次先根遍历每一棵树；中根遍历森林相当于后根遍历每一棵树。



树的结构划



森林的结构划分

遍历树、森林与遍历二叉树的关系

遍历树、森林和二叉树的关系

树	森林	二叉树
先根遍历	先序遍历	先序遍历
后根遍历	中序遍历	中序遍历

8. 哈夫曼树：叶子结点带有权值的最小带权路径长度的最优二叉树

$$WPL=\sum_{k=1}^{n} W_k \cdot L_k$$

构造赫夫曼树

每次取两个最小的树组成二叉树

赫夫曼编码(前缀码)

向左分支为 0，向右分支为 1，从根到叶子的路径构成叶子的前缀编码。

五、 图

1.

完全图：有 1\2 n(n-1)条变得无向图
有向完全图：具有 n（n-1）条弧的有向图。
权：与图的边或弧相关的数。
顶点 v 的度：和 v 相关联的边的数目。
入度：以顶点 v 为头的弧的数目
出度：以顶点 v 为尾的弧的数目
回路或环：第一个顶点和最后一个顶点相同的路径。
简单路径：序列中顶点不重复出现的路径。

2. 图的存储结构

- 邻接矩阵：

A	0	1	1	0	0
B	0	0	1	1	0
C	0	0	0	1	0
D	1	0	0	0	0
E	1	0	0	1	0

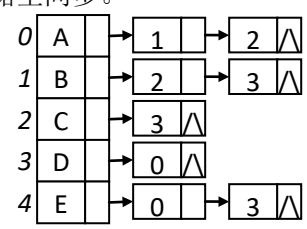
- 邻接表：

```
typedef struct ArcNode {   // 弧结点
   int                adjvex;                  // 邻接点
   struct ArcNode *nextarc;                    // 下一个邻接点
} ArcNode;

typedef struct VexNode {   // 顶点结点
   VertexType         data;                    // 顶点信息
   ArcNode *firstarc;   // 第一个邻接点
} VexNode;

const int MAX_VERTEX =  最大顶点个数;
typedef struct Graph {   // 图
   VexNode            vexs[MAX_VERTEX];        // 顶点向量
   int                vexnum, arcnum;          // 顶点和弧的个数
} Graph;
```

边(弧)多则需要存储空间多。



• 十字链表：

十字链表是有向图的另一种链式存储结构。可以看成是将有向图的邻接表和逆邻接表结合起来的一种链表。在十字链表中，对应于有向图中每一条弧有一个结点，对应于每个顶点有一个结点。

• 邻接多重表

3. 图的遍历

1) 深度优先（DFS）搜索

访问方式：从图中某顶点 v 出发：

a）访问顶点 v；

b）从 v 的未被访问的邻接点出发，继续对图进行深度优先遍历，若从某点出发所有邻接点都已访问过，退回前一个点继续上述过程，若退回开始点，结束。

2) 广度（宽度，BFS）优先搜索

a）访问顶点 v ；

b）访问同 v 相邻的所有未被访问的邻接点 $w_1, w_2, \ldots w_k$；

c）依次从这些邻接点出发，访问它们的所有未被访问的邻接点；依此类推，直到图中所有访问过的顶点的邻接点都被访问；

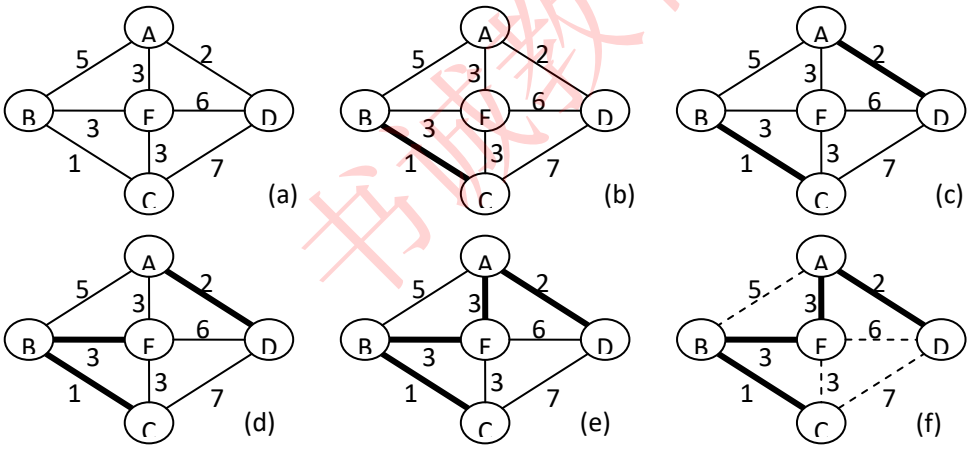
4. 生成树和最小生成树

每次遍历一个连通图将图的边分成遍历所经过的边和没有经过的边两部分，将遍历经过的边同图的顶点构成一个子图，该子图称为生成树。因此有 DFS 生成树和 BFS 生成树。

1) 最小生成树

• Kruskal 算法

一句话，“不构成环的情况下，每次选取最小边”。



• 普里姆算法

记 V 是连通网的顶点集，U 是求得生成树的顶点集，TE 是求得生成树的边集。

普里姆算法：

(a) 开始时，U={v_0}，TE=Φ；

(b) 计算 U 到其余顶点 V-U 的最小代价，将该顶点纳入 U，边纳入 TE；

(c) 重复(b)直到 U=V。

普里姆算法和克鲁斯卡尔算法的比较

算法	普里姆算法	克鲁斯卡尔算法
时间复杂	$O(n^2)$	O(e loge)

度		
特点	只与顶点个数 n 有关 与边的数目 e 无关 适用于稠密图	只与边的数目 e 有关 与顶点个数 n 无关 适用于稀疏图

5. AOV-网

用顶点表示活动，边表示活动的优先关系的有向图称为 AOV 网。

拓扑排序：对 AOV 网络中顶点构造拓扑有序序列的过程。
拓扑排序的方法
(1)在有向图中选一个没有前驱的顶点且输出之
(2)从图中删除该顶点和所有以它为尾的弧

6. 关键路径

AOE 网，关键路径

AOE 网(Activity On Edge)——带权的有向无环图，其中顶点表示事件，弧表示活动，权表示活动持续时间。在工程上常用来表示工程进度计划。
关键路径：从源点到汇点的最长的一条路径，或者全部由关键活动构成的路径。

7. 最短路径

(1) 迪杰斯特拉算法

求一个顶点到其他各顶点的最短路径。
算法：(a) 初始化：用起点 v 到该顶点 w 的直接边(弧)初始化最短路径，否则设为∞；
(b) 从未求得最短路径的终点中选择路径长度最小的终点 u：即求得 v 到 u 的最短路径；
(c) 修改最短路径：计算 u 的邻接点的最短路径，若(v,…,u)+(u,w)<(v,…,w)，则以(v,…,u,w)代替。
(d) 重复(b)-(c)，直到求得 v 到其余所有顶点的最短路径。
特点：总是按照从小到大的顺序求得最短路径。

(2) 弗洛伊德算法

求每对顶点之间的最短路径。
依次计算 $A^{(0)}$，$A^{(1)}$，…，$A^{(n)}$。$A^{(0)}$为邻接矩阵，计算 $A^{(k)}$时，$A^{(k)}(i,j)=\min\{A^{(k-1)}(i,j), A^{(k-1)}(i,k)+A^{(k-1)}(k,j)\}$。
技巧：计算 $A^{(k)}$的技巧。第 k 行、第 k 列、对角线的元素保持不变，对其余元素，考查 A(i,j)与 A(i,k)+A(k,j) （“行+列”），如果后者更小则替换 A(i,j)，同时修改路径。

六、 查找

1. 查找分为：静态查找表、动态查找表、哈希查找表

2. 静态查找表：对查找表只作查找操作的查找表。
动态查找表：查找过程中同时插入表中不含的元素，或者删除查找表中已有的元素的操作的查找表。
3. 顺序查找：顺序查找又称线性查找，是最基本的查找方法之一。顺序查找既适用于顺序表，也适用于链表。
4. 二分法（折半）查找：是一种效率较高的查找方法，但前提是表中元素必须按关键字有序（按关键字递增或递减）排列。
5. 索引顺序表查找又称分块查找。分块查找：块内无序、块间有序、如何分块效率最高
6. 动态查找表：二叉排序树查找：二叉排序树的生成
二叉排序树(二叉查找树)：或者是一颗空树。或者如下 1 若它的左子树不空，则左子树上所有的结点的值均小于他的根结点的值。2 若右子树不空，则右子树所有结点的值均大于她得根结点的值。3 左右子树也分别为二叉排序树。
7. 哈希表：哈希函数构造：直接定址法、除留余数法、平方取中法，随机数法，　　数字分析法
冲突解决方法：开放定址法、拉链法、公共溢出区法

七、　排序

1.插入类排序：

• 直接插入排序
每次将一个待排序的数据元素，插入到前面已经排好序的数列中的适当位置，使数列依然有序；直到待排序数据元素全部插入完为止。
• 折半插入排序
• 希尔排序
基本思想：先将整个待排序记录序列分成为若干个子序列分别进行直接插入排序，待整个序列中的记录 基本有序 时在对全体序列进行一次直接插入排序，子序列的构成不是简单的逐段分割，而是将像个某个增量的记录组成一个子序列。

2.交换类排序

• 起泡排序
也称冒泡法，每相邻两个记录关键字比大小, 大的记录往下沉（也可以小的往上浮）。每一遍把最后一个下沉的位置记下，下一遍只需检查比较到此为止；到所有记录都不发生下沉时，整个过程结束（每交换一次，记录减少一个反序数）。
• 快速排序
在当前无序区 R[1..H]中任取一个数据元素作为比较的"基准"(不妨记为 X)，用此基准将当前无序区划分为左右两个较小的无序区：R[1..I-1]和 R[I+1..H]，且左边的无序子区中数据元素均小于等于基准元素，右边的无序子区中数据元素均大于等于基准元素，而基准 X 则位于最终排序的位置上，即 R[1..I-1]≤X.Key≤R[I+1..H](1≤I≤H)，当 R[1..I-1]和 R[I+1..H]均非空时，分别对它们进行上述的划分过程，直至所有无序子区中的数据元素均已排序为止。

3.选择类排序

• 简单选择排序
每一趟从待排序的数据元素中选出最小（或最大）的一个元素，顺序放在已排好序的数列的最后，直到全部待排序的数据元素排完。
• 堆排序
堆排序是一树形选择排序，在排序过程中，将 R[1..N]看成是一颗完全二叉树的顺序存储结构，利用完全

二叉树中双亲结点和孩子结点之间的内在关系来选择最小的元素。

4.归并类排序

- 二路归并排序

5.基数类排序

- 基数排序

主要特点不需要进行关键字间的比较。

多关键字排序：

最高为优先（MSD 法）从主关键字开始排序，再从次高位排序，一次类推，最后将所有子序列依次连接在一起成为一个有序序列。

最低位优先（LSD 法）从最次位关键字开始排序，在对高一位的进行排序，直到成为一个有序序列。

排序方法	稳定性	平均时间	最坏情况	辅助存储
直接插入排序	稳定	O（n*n）	O（n*n）	O（1）
快速排序	不稳定	O（nlbn）	O（n*n）	O（lbn）
归并排序	稳定	O(nlbn)	O（nlbn）	O（n）
简单选择排序	稳定	O（n*n）	O（n*n）	O（1）
堆排序	不稳定	O（nlbn）	O（nlbn）	O(1)
基数排序	稳定	O（d（n+rd））	O（d（n+rd））	O（rd）

- 选取排序方法需要考虑的因素：

(1) 待排序的元素数目 n；

(2) 元素本身信息量的大小；

(3) 关键字的结构及其分布情况；

(4) 语言工具的条件，辅助空间的大小等。

- 小结：

(1) 若 n 较小(n <= 50)，则可以采用直接插入排序或直接选择排序。由于直接插入排序所需的记录移动操作较直接选择排序多，因而当记录本身信息量较大时，用直接选择排序较好。

(2) 若文件的初始状态已按关键字基本有序，则选用直接插入或冒泡排序为宜。

(3) 若 n 较大，则应采用时间复杂度为 O(nlog2n)的排序方法：快速排序、堆排序或归并排序。

快速排序是目前基于比较的内部排序法中被认为是最好的方法。

(4) 在基于比较排序方法中，每次比较两个关键字的大小之后，仅仅出现两种可能的转移，因此可以用一棵二叉树来描述比较判定过程，由此可以证明：当文件的 n 个关键字随机分布时，任何借助于"比较"的排序算法，至少需要 O(nlog2n)的时间。

算法分析知识点总结

算法概述

1、**算法的五个性质：**有穷性、确定性、能行性、输入、输出。

2、**算法的复杂性取决于：**（1）求解问题的规模（N），（2）具体的输入数据（I），（3）算法本身的设计（A），C=F(N,I,A)。

3、算法的时间复杂度的上界记号 O，

 下界记号 Ω (记为 f(N) = Ω(g(N))。　　即算法的实际运行时间至少需要 g(n)的某个常数倍时间)，

同阶记号 Θ：f(N)= Θ(g(N))表示 f(N)和 g(N)同阶 。

即算法的实际运行时间大约为 g(n)的某个常数倍时间。

低阶记号 o：f(N)=o(g(N))表示 f(N)比 g(N)低阶。

多项式算法时间：

O(1)<O(logn)<O(n)<O(nlogn)<O(n2)<O(n3)

约定 logn 表示以 2 为底的对数。指数时间算法时间：

O(2n)<O(n!)<O(nn)

4、**常用算法的设计技术：**分治法、动态规划法、贪心法、回溯法和分支界限法。

5、**常用的几种数据结构：**线性表、树、图。

6、**算法:**是指解决某一问题的运算序列

递归与分治

1、**递归算法的思想：**将对较大规模的对象的操作归结为对较小规模的对象实施同样的操作。

递归的时间复杂性可归结为递归方程：

$$T(n) \leq \begin{cases} 1 & n = 1 \\ aT(n\sim b) + D(n) & n > 1 \end{cases}$$

其中，a 是子问题的个数，b 是递减的步长， ~表示递减方式，D(n)是合成子问题的开销。

递归元的递减方式~有两种：1、减法，即 n – b，的形式。2、除法，即 n / b，的形式。

2、**递减方式为减法** 设 D(n)为常数，则有 T(n) = O(an)。

(这里都是针对递减方式为除法的哦)D(n)为常数 c：这时，T(n) = **O(n^p)**。

D(n)为线形函数 cn：

$$T(n) = \begin{cases} O(n) & 当a < b时 \\ O(n\log_b n) & 当a = b时 \\ O(n^p) & 当a > b时 \end{cases}$$

其中，$p = \log_b a$。

D(n)为幂函数 n^x：

$$T(n) = \begin{cases} O(n^x) & 当a < D(b)时 \\ O(n^p\log_b n) & 当a = D(b)时 \\ O(n^p) & 当a > D(b)时 \end{cases}$$

其中，$p = \log_b a$。

考虑下列递归方程：T(1) = 1

(1)T(n) = 4T(n/2) +n

(2) T(n) = 4T(n/2) +n^2

(3) $T(n) = 4T(n/2) + n^3$

解：方程中均为 a = 4，b = 2，其齐次解为 n^2。

对(1)，∵ $a > b^1$ $(D(n) = n)$ ∴ $T(n) = O(n^2)$；

对(2)，∵ $a = b^2$ $(D(n) = n^2)$ ∴ $T(n) = O(n^2 \log n)$；

对(3)，∵ $a < b^3$ $(D(n) = n^3)$ ∴ $T(n) = O(n^3)$；

证明一个算法的正确性需要证明两点：1、算法的部分正确性。2、算法的终止性。

3、汉诺塔问题：

```
void Hanoi(int n, int Fr, int To, int As)
        {
            if   (n > 0) {
                Hanoi(n–1, A, C, B);
                Move(A, B);
                Hanoi(n–1, C, B, A)}
         }
```

4、二分查找代码

二分搜索有序表的递归式算法

```
int recursive_binary_search ( int a[] , int left , int right , int T )
{ int k , mid;
  if ( left > right )  k = -1 ;  //*数组中不存在T，返回-1
   else
    {mid = ( left + right ) / 2 ;//*取中点下标
     if ( a[mid] == T )      k = mid ;
     else if (a[mid] < T )
             k = recursive_binary_search ( a , mid+1 , right , T ) ;
           else
             k = recursive_binary_search ( a , left , mid-1 ,T ) ;
    }
  return ( k ) ;//*返回T在数组a中位置的下标值
}
```

棋盘覆盖算法的复杂性

- 设T(k)是棋盘覆盖算法覆盖$2^k \times 2^k$的棋盘所需要的时间，当k= 0时，size等于1，覆盖它将花费常数时间d。当k>0时，将进行4次递归的函数调用，这些调用需花费的时间为 4*T(k-1)。除了这些时间外，if条件测试和覆盖3个非特殊方格也需要时间，假设用常数O(1)表示这些额外时间，则 T(k)满足如下递归方程：

$$T(k) = \begin{cases} O(1) & k = 0 \\ 4T(k-1) + O(1) & k > 0 \end{cases}$$

- 递归元递减方式是减法k – 1，a = 4，因此

$$T(k) = O(4^k)。$$

- 由于覆盖$2^k \times 2^k$的棋盘要用 $(4^k-1)/3$个L型骨牌，故此算法是一个在渐进意义下最优的算法。

棋盘覆盖算法的正确性

- 算法的终止性：
  递归的终止条件为递归元size等于1时递归终止。
  递归元size的初值为2^k。每次递归时递归元减半，即size=size/2；因此，必然在有穷步内递减为1。
  所以此算法必定终止。

- 由部分正确性和终止性可知，此算法是完全正确的。

贪心算法（旅行商问题、单源最短路径问题）

以下两种算法都是为了查找最小生成树问题的算法：

1、**Prim 算法的基本思想：**在保证连通的前提下依次选出权重较小的 n－1 条边。
（在实现中体现为 n 个顶点的选择）。
G=(V, E)为无向连通带权图，令 V={1, 2, …, n}。
设置一个集合 S ，初始化 S = {1}，T = Φ。
贪心策略：如果 V－S 中的顶点 j 与 S 中的某个点 i 连接且(i, j) 的权重最小，于是就选择 j(将 j 加入 S)，并将(i, j)加入 T 中 。
重复执行贪心策略，直至 V－S 为空。
==========证明最小生成树必然包含最小权值边==========
若 G 的任何最小生成树都不包含 e_1。设 T 为 G 的最小生成树，$e_1 \notin T$。于是 $T+e_1$ 是一个有回路的图且该回路中包含 e_1。该回路中必有条不是 e 的边 e_i。令 $T'=\{T+e_1\}-e_i$。T'也是 G 的生成树。又 $c(T') = c(T) + c(e_1) - c(e_i)$，$c(e_1) \le c(e_i)$，从而 $c(T') \le c(T)$，T'是 G 的最小生成树且含有边 e_1。矛盾。故必定有图 G 的最小生成树包含了 e_1。
==========

2、**Kruskal 算法的基本思想：**基本思想：在保证无回路的前提下依次选出权重较小的 n－1 条边。如果(i, j)是 E 中尚未被选中的边中权重最小的，并且(i, j)不会与已经选择的边构成回路，于是就选择 (i, j)。具体做法:先把所有 n 个点画出来。不画边。然后把权值最小的那条边画上去。然后再把当前权值最小的边(不算画了的边)画上去。如果构成回路则舍弃这条边。然后一直直到画了 n-1 条边

3、**Prim 与 Kruskal 两算法的复杂性：**
Prim 算法为两重循环，外层循环为 n 次，内层循环为 O(n)，因此其复杂性为 $O(n^2)$。
Kruskal 算法中，设边数为 e，则边排序的时间为 O(eloge)，最多对 e 条边各扫描一次，每次确定边的时间为 O(loge)，所以整个时间复杂性为 O(eloge)。
当 $e = \Omega(n^2)$时，Kruskal 算法要比 Prim 算法差；
当 $e = o(n^2)$时，Kruskal 算法比 Prim 算法好得多。

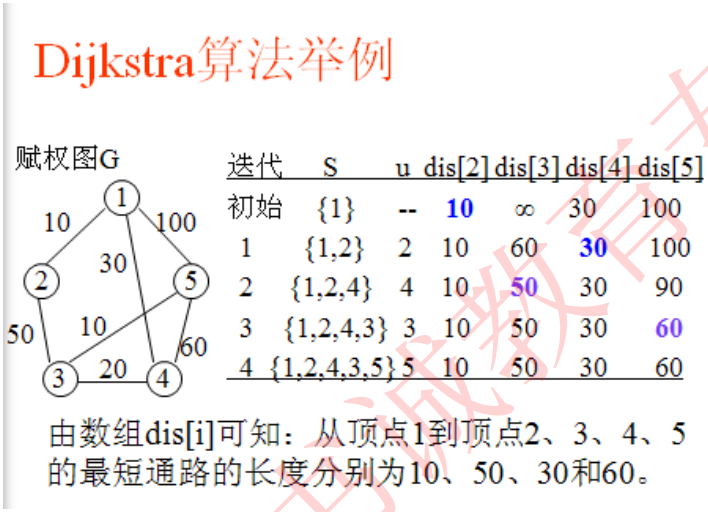
4、**贪心算法的基本要素是：**贪心选择性质。

5、最小生成树问题、单源最短路径问题、旅行商问题、0—1 背包问题，贪心算法不能够找到最优解。

6、活动安排问题、最优装载问题，贪心算法可以找到最优解。

7、**Dijkstra 算法**

```
Procedure Dijkstra {
( 1) S:={1}; //初始化 S
(2) for i:= 2    to n do //初始化 dis[]
(3) dis[i] =C[1, i] ; //初始时为源到顶点 i 一步的距离。不能一步到达就是无穷
(4) for i :=1    to    n do {
(5)       从 V-S 中选取一个顶点 u 使得 dis[u]最小；
(6)       将 u 加入到 S 中；//将新的最近者加入 S
(7)       for □ w∈V-S do    //依据最近者 u 修订 dis[w]
(8)       dis[w] := min(dis[w] , dis[u]+C[u ,w])
        }
 }
```



8、**活动安排问题**:先把活动按结束时间早晚排序。然后选取最早结束的。然后选取第一个开始时间比上一个结束时间大的再用这个原则选取。总的时间复杂度为 O(nlogn)

==========代码==========

```
typedef struct {
  int    number; //活动的编号
  float start;   //活动开始的时间
  float end;     //活动结束的时间
  Bool selected; //标记活动是否被选择
}Active;
 int greedySelector(Active activity[], int n)
 {QuitckSort(activity,n);          //将活动按结束时间排序
    activity[1].selected=true;   j=1;count=1;
   for(i=2;i<=n;i++)
   if(activity[i].start>=activity[j].end)
```

```
            {activity[i].selected=true;
      j=i;        count++;     }
   else   activity[i].selected=false;
  return count;  }
```

9、**为什么贪心算法不能求得旅行商问题的最优解**：最临近法不保证求得旅行商问题的精确解，只能得到问题地近似解。一般地，贪心选择只依赖于前面选择步骤地最优性，因此是局部最优的，所以贪心法不能够确保求出问题的最优解。

10、**最优装载问题**：基本思想：这个题目比较简单，可以用贪心法求解。每次采用重量最轻者优先装入的贪心选择策略

11、**贪心算法的特点是什么？怎么样知道一个问题是否能用贪心算法呢？**

贪心算法每次选择目前最优的解，即通过一系列局部最优来获得整体最优。贪心算法只有在具有贪心选择性质时才能保证获得整体最优。

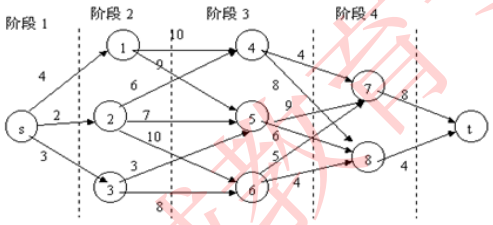
证明贪心选择性质：⑴第一个选择是对的；⑵在作出贪心选择后原问题转化为同样的子问题；⑶由归纳法知问题具有贪心选择性质。

若原问题是求带权拟阵的最优独立子集问题，则必满足贪心选择性质。

动态规划

1、**最短路径问题:**

一个多段图例子



阶段4: C(7,t)=w(7,t)=8， C(8,t)=w(8,t)=4

阶段3: C(4,t)=min{w(4,7)+C(7,t), w(4,8)+C(8,t)}=12

C(5,t)=min{w(5,7)+C(7,t), w(5,8)+C(8,t)}=10

C(6,t)=min{w(6,7)+C(7,t), w(6,8)+C(8,t)}=8

阶段2: C(1,t)=min{w(1,4)+C(4,t), w(1,5)+C(5,t)}=19

C(2,t)=min{w(2,4)+C(4,t), w(2,5)+C(5,t), w(2,6)+C(6,t)}=17

C(3,t)=min{w(3,5)+C(5,t), w(3,6)+C(6,t)}=13

阶段1: C(s,t)=min{w(s,1)+C(1,t), w(s,2)+C(2,t), w(s,3)+C(3,t)}=16

沿求解中带下划线的项回溯，得最短路径解：**D(s,t)= <s,3,5,8,t>**

如果是从源点往后推:

阶段 1:

C(s,1)=w(s,1)=4， C(s,2)=w(s,2)=2， C(s,3)=w(s,3)=3

阶段 2:

C(s,4)=min{w(1,4)+C(s,1), w(2,4)+C(s,2)}=min{14,8}= 8

C(s,5)=min{w(1,5)+C(s,1), w(2,5)+C(s,2), w(3,5)+C(s,3)} =min{13,9,6}= 6

C(s,6)=min{w(2,6)+C(s,2), w(3,6)+C(s,3)}=min{12,11}= 11

阶段 3:

C(s,7)=min{w(4,7)+C(s,4), w(5,7)+C(s,5), w(6,7)+C(s,6)}　　=min{12,15,16}= 12

C(s,8)=min{w(4,8)+C(s,4), w(5,8)+C(s,5), w(6,8)+C(s,6)}　　=min{16,12,15}= 12

阶段 4:

C(s,t)=min{w(7,t)+C(s,7), w(8,t)+C(s,8)}=min{20,16}= 16

2、**最有子结构的性质：**最优解的子结构也是最优的。

3、**动态规划的基本要素：**（1）最优子结构：问题的最优解是由其子问题的最优解所构成的。（2）重叠子问题：子问题之间并非相互独立的，而是彼此有重叠的。

4、**动态规划算法的步骤：**1.找出最优解的性质，并刻画其结构特性。2.递归地定义最优解。3.以自底向上的方式计算出各子结构的最优值，并流入表格中保存。4.根据计算最优值时得到的信息，构造最优解。

5、在有 n 个顶点的凸多边形的三角剖分中，恰有 n-3 条弦，n-2 个三角形。

回溯法

1、**三种搜索方法：**（1）广度优先搜索的优点是一定能找到解；缺点是空间复杂性和时间复杂性都大。（2）深度优先搜索的优点是空间复杂性和时间复杂性较小；缺点是不一定能找到解。（3）启发式搜索是最好优先的搜索，优点是一般来说能较快地找到解，即其时间复杂性更小；缺点是需要设计一个评价函数，并且评价函数的优劣决定了启发式搜索的优劣。

2、 回溯法是一种通用的算法，实为深度优先的搜索方法。

回溯法可以递归实现，也可以迭代实现。

回溯法通常求解三类问题：

(1)求排列：时间复杂性为 O(f(n)n!)；

(2)求子集：时间复杂性为 $O(f(n)2^n)$；

(3)求路径：时间复杂性为 $O(f(n)k^n)$。

这里 f(n)为处理一个结点的时间复杂性。

3、**分支限界法回溯法联系与区别：**

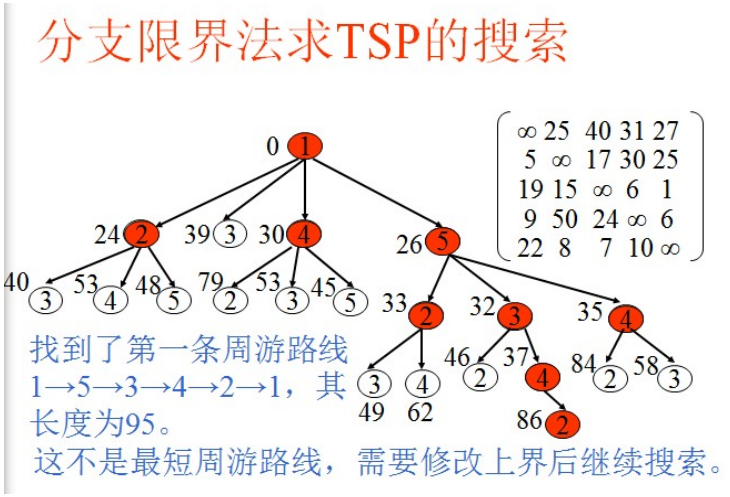
支限界法类似于回溯法，也是一种在问题的解空间树 T 上搜索问题解的算法。但在一般情况下，分支限界法与回溯法的求解目标不同。回溯法的求解目标是找出 T 中满足约束条件的所有解，而分支限界法的求解目标则是找出满足约束条件的一个解，或是在满足约束条件的解中找出使某一目标函数值达到极大或极小的解，即在某种意义下的最优解。

由于求解目标不同，导致分支限界法与回溯法在解空间树 T 上的搜索方式也不相同。回溯法以深度优先的方式搜索解空间树 T，而分支限界法则以广度优先或以最小耗费优先的方式搜索解空间树 T。分支限界法的搜索策略是：在扩展结点处，先生成其所有的儿子结点（分支），然后再从当前的活结点表中选择下一个扩展对点。为了有效地选择下一扩展结点，以加速搜索的进程，在每一活结点处，计算一个函数值（限界），并根据这些已计算出的函数值，从当前活结点表中选择一个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。

分支界限法

1、**分支界限法德两个要点：**（1）评价函数的构造。（2）搜索路径的构造。

2、所谓**分支限界法**就是通过评价函数及其上下界函数的计算，将状态空间中不可能产生最佳解的子树剪去，减少搜索的范围，提高效率。因而更准确的称呼应是“界限剪支法”



字符串

1、**几个名词的定义：**串的长度，子串，位置，串相等，模式匹配。简单模式匹配算法在正文设置一个指针指向第一个然后跟模式串匹配。不行就指针加一直到匹配成功或者正文结束。时间复杂度为 O(m+n)最坏为 O(mn)

2、**KMP 算法：**时间复杂度为 O(m+n)。next(j)函数的计算(计算到第 j 个就是从 1 到 j-1 这个子串首位分别截取尽可能长的相同子串。从尾巴那里截取的子串不用倒过来。得到的最长相同子串长+1 就是 next(j)的值啦。Next(1)固定等于 0。 恒有 next(2) = 1)，

```
int KMP_StrMatch(SString S, SString P) {
        int i = 1, j = 1, m = 0;
        while(i <= S[0] && j <= P[0])
                if (j = 0 || S[i] = P[j]) {i++;     j++;}
          else j = next[j]; //失配时从 next[j]重新比较
          if(j > P[0]) m = i – j + 1;
        return(m);
           }
```

==========**Newnext 函数的计算**==========

```
void get_newnext(){
int k,j;
newnext[1]=0;         j=2;
while(j<=P[0]){    k=next[j];
if(k==0||p[k]!=p[j])
newnext[j]=k;
else    newnext[j]=newnext[k];
j=j+1;}}
```

3、**BM 算法：**Boyer-Moore 串匹配算法(简称 BM 算法)。最坏时间复杂度 O(mn)

其思想是在匹配过程中，一旦发现在正文中出现模式中没有的字符时就可以将模式、正文大幅度地“滑过”一段距离。

同时考虑到多数不匹配的情形是发生在最后的若干个字符，采用从左到右的方式扫描将浪费很多时间，因此改自右到左的方式扫描模式和正文

==========**dist 函数的计算**==========

$$\text{dist}(c) = \begin{cases} m & c \neq P, \text{ 或 } c = p_m \text{ 且 } c \neq p_j (0<j<m) \\ m-j & \text{否则 } (j=\max\{c=p_j, 0<j<m\}) \end{cases}$$

4、**KR 算法：**指印函数的要求：（1）速度快。（2）冲突概率小。（3）相邻两个字符串的 HASH 值必须有相关性。

5、**KR 算法的基本思想：**首先计算模式串和正文串所有长度为 m 的子串的指印函数；然后匹配与模式串指印函数相等的正文串中的子串，找到匹配串。

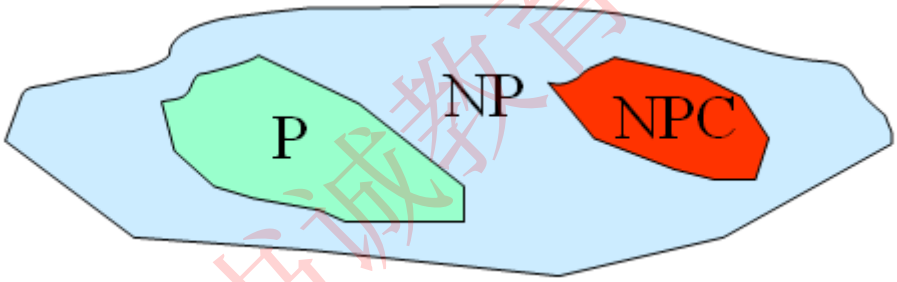
6、**KMP 算法，BM 算法，KR 算法的优缺点：**三者中 KMP 算法的最坏情形下的时间复杂度最低，而平均情形则三者差不多。KMP 算法最早提出，应用也最广泛，而且它的优势在于无需对正文回溯，当正文不能一次性读入内存时，这种优势是显而易见的。BM 算法的平均性能也很出色，但它需要对正文进行回溯，所以当正文不能一次性读入内存时，可能会出现“抖动”，需要用双缓冲区技术来处理这一问题。KR 算法的优势在于可以推广到二维模型，而且可以并行地处理，所以较多地运用于图形图像领域。

NP 完全问题

1、**NP 类问题定义：**由非确定性图灵机再多项式时间内可以计算的判定问题

2、**NP 困难问题与 NPC 问题是一类问题么？就旅行商问题和判定问题谈谈你的看法。**

不一定：对于问题 Q，若满足 Q∈NP 且 Q 是 NP 困难的，则称 Q 是 NP 完全的。旅行商问题不属于 NP 问题。因为我们无法在多项式时间内验证其解的正确性。当然也就不是 NPC 问题(NPC 问题就是 NP 完全问题)。但他们仍然是 NP 困难的。旅行商判定问题则完全可以在多项式时间内检验其正确性这就是一个 NP 问题更进一步说他是一个 NPC 问题。但是无论哪种问题。都没有改变这个问题的难度。二者难度是等价的一般来说。判定问题可能属于 NPC 问题相应的最优问题则是 NP 难题



3、

概率算法

1、**伪随机数的产生方法：**（1）线性同余法：实现简单，速度快，具有较好的统计性能，广泛应用于仿真，但不能用于加密。（2）线性反馈移位寄存器。

2、 **舍伍德算法：**（1）舍伍德算法总能求得问题的一个解，且所求得解的结果总是正确的。其主要目的是消除算法所需计算时间对输入实例的依赖。（2）当一个确定型算法最坏情况下的计算复杂性与其平均情形下的复杂度有较大差别时，可以在这个确定型算法中引入随机性将它改造成一个舍伍德算法，消除或减少问题的好坏实例间的这种差别。（3）舍伍德算法的精髓不是避免算法的最坏情况行为，而是设法消除这种最坏情形行为与特定实例之间的关联性。

3、**拉斯维加斯算法：**（1）显著地改进算法的有效性，甚至对某些迄今为止找不到有效算法的问题，也能找到满意的算法；但它所做的随机性决策有可能导致算法找不到需要的解。

（4） 由于后者的原因，通常用 boolean 型方法表示拉斯维加斯型算法。

4、**蒙特卡洛算法：**（1）蒙特卡罗算法用于求解问题的准确解。（2）蒙特卡罗算法能求得一个解，但未必是正确的。其求得正确解的概率依赖于算法所用的时间。所用时间越多，得到正确解的概率就越高，但它的缺点

也在于此。在一般情况下，无法有效地判定所得到的解是否肯定正确。

数据压缩算法

1、**被压缩的对象：**物理空间，即数据存储介质的尺寸。时间区间，传输消息集合所需要的时间。电磁频谱区域，如为传输消息的带宽等。

2、ASCII 码压缩方法是一种经过两级压缩之后可以减少 **56.25%**的存储空间的算法，它之适用于纯数字。

压缩方法：

把原始数据两两分组。然后把每个两位数化成十六进制。然后每 8 个一组。把第八个十六进制对应的二进制最高位(最左边)去掉。分别填在前面七个的最高位。就可以了

3、**哈夫曼编码：**哈夫曼编码的长度笔筒，它的基本理论基于下列定理：在变长编码中，若各码字长度严格按照所对应的符号出现概率的大小逆序排列，则其平均长度最小。

结点的权：在一些应用中，赋予树中结点的一个有某种意义的实数。

结点的带权路径长度：结点到树根之间的路径长度与该结点上权的乘积。

树的带权路径长度(Weighted Path Length of Tree)：定义为树中所有叶结点的带权路径长度之和。拥有最小带权路径长度的二叉树称为最有二叉树或者哈夫曼树。

编码方法：先构造哈弗曼树。就是每次每次选取最小的两个作为子节点生成一个父节点。父节点的值就是两个子节点的值的和然后从根节点开始往下走左节点为 0 右节点为 1 到达叶子节点经过的 01 就是他的编码

4、**字典法：**基本思想：构造一个字典，将信息中反复出现的字符串，登记为较短的字符串，解码时对这种字符串，通过查字典，转换为原字符串。最原始的字典法是模式置换压缩算法。

5、**字典法存在的几个困扰的难点：**（1）如何找到这些重复出现的字符串？（2）如何找到尽量长的字符串被替代？（3）怎样选用较短的字符串？如何区分原始信息中就已经存在的用于代替的字符串？

6、**LZW 算法：**先压入所有单字符。然后读入两个字符。把前缀替换成数字。然后依次往后读入字符。如果读入后已经在表中有了就再读入一个。再把前缀换成数字。最后如果不够了可以把整个都换成数字。解压缩算法对比下压缩时的过程就知道了