习 题 答 案

第 1 章参考答案

一、填空题

1. 数据元素的有限集，D 中数据元素之间客观存在的关系的有限集
2. 顺序存储结构，链式存储结构，散列存储结构
3. $O(m*n)$
4. $O(n)$
5. $O(\log_3 n)$

二、选择题

1. C　　2. D　　3. B　　4. C，A　　5. C

三、简答题

1. 数据结构就是带有结构的数据的集合，包括数据元素、元素之间的关系和对数据元素的操作 3 个部分；逻辑结构包括线性表、栈、队列、树、二叉树、无向图、有向图等，结构特征即元素之间的关系。
2. 5 个特点：1）有穷性；2）确定性；3）可行性；4）输入；5）输出。
   2 个区别：1）有穷性；2）描述方法。
3. 设 $fact(n)$ 的运行时间函数是 $T(n)$。该函数中语句①的运行时间是 $O(1)$，语句②的运行时间是 $T(n-1)+O(1)$，其中 $O(1)$ 为运行时间。因此：

$$T(n)=\begin{cases}O(1) & n\leqslant 1\\ T(n-1)+O(1) & n>1\end{cases}$$

   则：

$$\begin{aligned}T(n)&=O(1)+T(n-1)\\&=2*O(1)+T(n-2)\\&\cdots\\&=(n-1)*O(1)+T(1)\\&=n*O(1)\\&=O(n)\end{aligned}$$

   即 $fact(n)$ 的时间复杂度为 $O(n)$。

四、算法设计题

1. 算法描述 1(自然语言描述)：

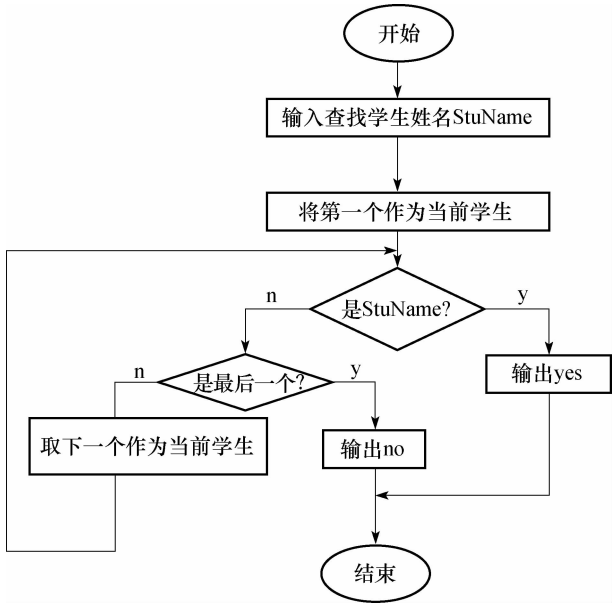
   步骤 1：输入要查找的学生姓名 StuName。

   步骤 2：将该班级名册中的第一个学生作为当前学生。

   步骤 3：将当前学生的姓名与 StuName 进行比较，如果相符，则输出“yes”，表示找到，算法结束；否则，执行步骤 4。

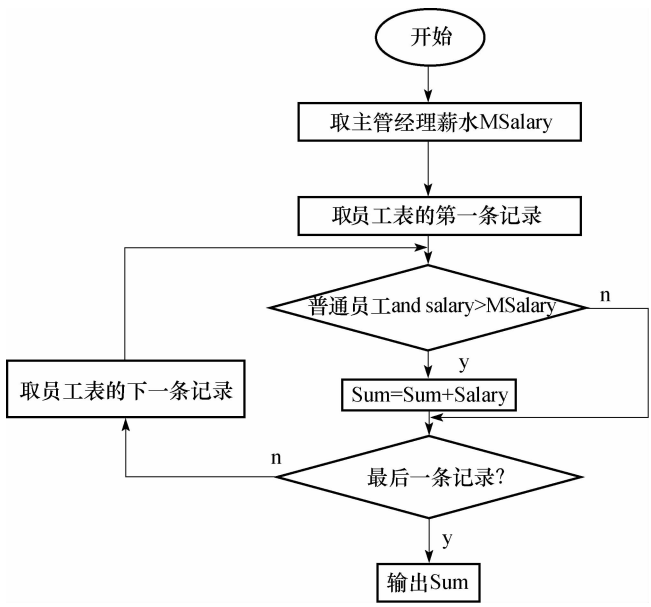
   步骤 4：如果当前学生不是该班最后一个学生，则取下一个学生作为当前学生，执行步骤 2；否则，输出“no”，表示没有找到，算法结束。

算法描述 2(框图描述)：



2.
```
int sum(A){
 int i,start,end,s = 0;
 scanf("%d",start);
 scanf("%d",end);
 for( i = start + 1; i < end; i++ )
        s = s + A[i];
return s;}
```

3.



第 2 章参考答案

一、选择题

1. C　2. D　3. C　4. C　5. B　6. A　7. A　8. B　9. C　10. D

二、填空题

1. $n/2$，$(n-1)/2$
2. 顺序，链式
3. 将 *list* 改为指向第二个结点，然后释放第一个结点的空间
4. 线性链表（单链表），循环链表，双向链表，线性链表（单链表）
5. 8
6. 489
7. `front == rear`
8. $n-1$
9. 逻辑顺序，物理顺序
10. `HS == NULL`

三、简答题

1. 开始结点是指链表中的第一个结点，也就是没有直接前趋的那个结点。

   链表的头指针是一指向链表开始结点的指针（没有头结点时），单链表由头指针唯一确定，因此单链表可以用头指针的名字来命名。

   头结点是我们人为地在链表的开始结点之前附加的一个结点。有了头结点之后，头指针指向头结点，不论链表否为空，头指针总是非空。而且头指针的设置使得对链表的第一个位置上的操作与对表的其他位置上的操作一致（都是在某一结点之后）。
2. 若频繁地对线性表进行插入和删除操作，该线性表应采用链式存储的方式。因为若采用顺序存储，在插入或删除元素时，要频繁地移动元素的位置，从而使得时间效率变低，而链式存储由于只需修改指针，能很好地克服这一缺陷。
3. 对于上三角矩阵，存储思想是常量存放在第一个位置，其他元素以列为主序顺序存储上三角部分。对于 $n\times n$ 的上三角矩阵，由于 a_{ij} 的前面有 $j-1$ 列，其中第 1 列有 1 个元素，第 2 列有 2 个元素，……，第 $j-1$ 列有（$j-1$）个元素，又第 j 列前面有（$i-1$）个数据元素，再加上第一个单元的常数，其前面共有 $1+2+\cdots+j-1+i-1+1=j(j-1)/2+i$ 个元素，则 $sa[k]$ 与 a_{ij} 的对应关系为：$k=j(j-1)/2+i(i\leqslant j)$，$k=0(i>j)$。
4. 循环队列的优点是：它可以克服顺序队列的“假上溢”现象，能够使存储队列的向量空间得到充分的利用。判别循环队列的“空”或“满”不能以头尾指针是否相等来确定，一般是通过以下几种方法：一是另设一布尔变量来区别队列的空和满；二是少用一个元素的空间，每次入队前测试入队后头尾指针是否会重合，如果会重合就认为队列已满；三是设置一计数器记录队列中的元素总数，这不仅可判别空或满，还可以得到队列中元素的个数。
5. 1）IIIOOOIOIO

   2）IOIIOOIIOO

四、算法设计题

1. 因已知顺序表 L 是非递减有序表，所以只要从头开始找到第一个比它大（或相等）的结点数据，把 x 插入到这个数据所在的位置就可以了。算法如下：

```
void InsertIncreaseList( Sqlist * L ,Datatype x ){
int i;
for( i = 0 ; i < L -> length && L -> data[ i ] < x ; i ++ ); // 查找并比较,分号不能少
```

```
InsertList( L ,x ,i ); // 调用顺序表插入函数
}
```

2. 所谓逆转一个线性链表，是指在不增加新的链结点空间的前提下，依次改变链表中链结点的链接方向，即链表第一个链结点成为链表的最末端的那个结点，最末端那个结点成为链表的第一个结点，第二个结点成为倒数第二个结点，依次类推。算法如下：

```
void Invert(List L){
    p = L. list;        //变量 p 首先指向链表的第一个结点
    q = null;
    while(p! = null)  //交互对应结点位置,直至列表末端
    do{
      r = q;
      q = p;
      p = p. next;
      q. next = r;
    }
    L. list = q
   }
```

3.

```
SPMatrix * TransM2(SPMatrix * A){
    SPMatrix * B;
     int i,j,k;
     int num[A -> nu + 1],cpot[A -> nu + 1];
     B = new SPMatrix;
     B -> mu = A -> nu;
     B -> nu = A -> mu ;
     B -> tu = A -> tu;
     if(B -> tu > 0)
    {  for(i = 1;i <= A -> nu;i ++ )     num[i] = 0;
       for(i = 1;i < A -> nu;i ++ )
       {   j = A -> data[i]. j;
           num[j] ++ ;
       }
       cpot[1] = 1;
       for(i = 2;i < = A -> nu;i ++ )
           cpot[i] = cpot[i - 1] + num[i - 1];
       for(i = 1;i <= A -> tu;i ++ )
       {   j = A -> data[i]. j;
           k = cpot[j];
           B -> data[k]. i = A -> data[i]. j;
           B -> data[k]. j = A -> data[i]. i;
           B -> data[k]. v = A -> data[i]. v;
           cpot[j] ++ ;
       }}
    return B;}
```

4. 知道了尾指针和元素个数，当然就能知道队头元素了。算法如下：

```
int FullQueue( CirQueue * Q)
 {
```

```
        // 判队满,队中元素个数等于空间大小
        return Q->quelen==QueueSize;
    }
    void EnQueue( CirQueue *Q,Datatype x)
    {
        // 入队
        if (FullQueue( Q)){
          Error("队已满,无法入队");
          exit(0);
        }
        Q->Data[Q->rear]=x;
        Q->rear=(Q->rear+1)%QueueSize;//在循环意义上的加 1
        Q->quelen++;
    }
    Datatype DeQueue( CirQueue *Q)
    {
        //出队
        if (Q->quelen==0){
          Error("队已空,无元素可出队");
          exit(0);
        }
        int tmpfront; //设一个临时队头指针
        if (Q->rear>Q->quelen)//计算头指针位置
          tmpfront=Q->rear - Q->quelen;
        else
          tmpfront=Q->rear + QueueSize - Q->quelen;
        quelen--;
        return Q->Data[tmpfront];
    }
```

5.

```
#include "iostream.h"
const int n0=30;
int s1[n0+1]; //操作数栈
char s2[n0+1]; //运算符栈
int t1,t2;
int num[4]; //提取表达式中的整数

void calcu(){
    int x1,x2,x;
    char p;
    //弹出一个运算符
    p=s2[t2--];
    //弹出两个操作数
    x2=s1[t1--];
    x1=s1[t1--];
    //进行一次运算
    switch(p){
```

```
        case '+':x = x1 + x2;break;
        case '-':x = x1 - x2;break;
        case '*':x = x1 * x2;break;
        case '/':x = x1/x2;
    }
       //结果压入操作数栈
    s1[++t1] = x; }

int calculator(char *f){
    int v,i = 0;
    char *p = f;
    t1 = t2 = 0; //设置空栈
    while(*p! ='\0'){
        switch(*p){
           case '+': case '-':
             while(t2&&(s2[t2]! ='('))
                  //执行先遇到的加、减、乘、除运算
                  calcu();
                  //当前运算符入栈
             s2[++t2] = *p;
             //读下一个字符
             p++;
             break;
           case '*': case '/':
             if(t2&&(s2[t2] == '*')||(s2[t2] == '/'))
                 //执行先遇到的乘、除运算
                 calcu();
                 //当前运算符入栈
             s2[++t2] = *p;
             //读下一个字符
                 p++;
                 break;
           case'(':
             //左括号入栈
             s2[++t2] = *p;
             //读下一个字符
             p++;
             break;
           case ')':
               while(s2[t2]! ='(')
                    //执行括号内的加、减、乘、除运算
                    calcu();
               //弹出左括号
               t2--;
               //读下一个字符
               p++;
                    break;
```

```
            default:
            //把字符串转换成整数值
            v = 0;
                do {
                    v = 10 * v + * p - '0';
                    p ++ ;
            } while(( * p >= '0')&&( * p <= '9'));
            //操作数入栈
            s1[ ++ t1] = v;
            num[i ++ ] = v;
        }
        }
        //执行先遇到的加、减、乘、除运算
        while(t2)calcu();
        //返回结果
        return s1[t1];
    }

    void main()
    {
        char a[] = "5 * (40 + 6) - 39";
        cout << calculator(a)<< endl;
            cout <<"其中的数字为:\n";
        for(int i = 0;i < 4;i ++ )
        {
                    cout << num[i]<<";
        }
            cout << endl;
    }
```

第 3 章参考答案

一、选择题

1. B　2. C　3. A　4. D　5. D　6. B　7. A　8. A　9. C　10. D

二、填空题

1. p -> firstchild == NULL

2. 3^{h-1}

3. 6，261

4. 5

5. 3，6

6. 10，1023

7. 叶子结点的相对次序不发生改变

8. ++a * b3 * 4 − cd，18

9. stack[tp] = t;,p = stack[tp --];,p;, ++ tp

10. 0,hl > hr,hr = hl

三、简答题

1. 1）K^{h-1}（h 为层数）。

   2）因为该树每层上均有 K^{h-1} 个结点，从根开始编号为 1，则结点 i 的从右向左数第 2 个孩子的结点编号为 $K\times i$。设 n 为结点 i 的子女，则关系式 $(i-1)\times K+2\leqslant n\leqslant i\times K+1$ 成立，因 i 是整数，故结点 n 的双亲 i 的编号为 $\lfloor (n-2)/K \rfloor+1$。

   3）结点 $n(n>1)$ 的前一结点编号为 $n-1$（其最右边子女编号是 $(n-1)*K+1$），故结点 n 的第 i 个孩子的编号是 $(n-1)*K+1+i$。

   4）根据以上分析，结点 n 有右兄弟的条件是：它不是双亲的从右数的第一个子女，即 $(n-1)\%K!=0$，其右兄弟编号是 $n+1$。

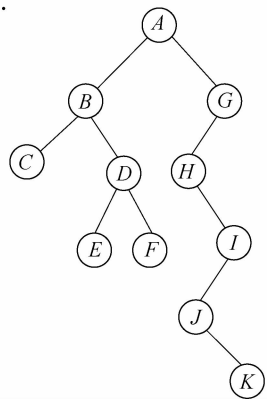
2. 设树的结点数为 n，分支数为 B，则下面两式成立：

$$n = n_0 + n_1 + n_2 + \cdots + n_m \tag{1}$$

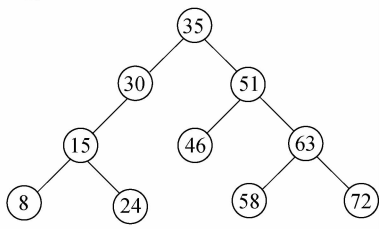
$$n = B + 1 = n_1 + 2n_2 + \cdots + mn_m \tag{2}$$

   由（1）和（2），可得 $n_0 = n_2 + 2n_3 + \cdots + (m-1)n_m$。

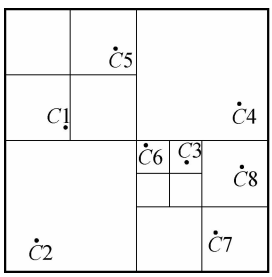
3.

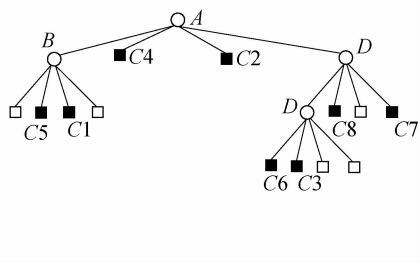


4.



5.





四、算法设计题

1. 由指示结点 i 左儿子和右儿子的两个一维数组 $L[i]$ 和 $R[i]$，很容易建立指示结点 i 的双亲的一维数组 $T[i]$，根据 T 数组，判断结点 U 是否是结点 V 后代的算法，转为判断结点 V 是否是结点 U 的祖先的问题。

```
int Generation(int U,V,N,L[],R[],T[]){
    //L[]和 R[]是含有 N 个元素且指示二叉树结点 i 左儿子和右儿子的一维数组，
    //本算法据此建立结点 i 的双亲数组 T,并判断结点 U 是否是结点 V 的后代。
    for(i = 1;i<= N;i++) T[i] = 0;  //T 数组初始化
```

```
    for(i=1;i<=N;i++)          //根据 L 和 R 填写 T
    if(L[i]!=0)T[L[i]]=i;     //若结点 i 的左子女是 L,则结点 L 的双亲是结点 i
    for(i=1;i<=N;i++)
    if(R[i]!=0)T[R[i]]=i;     //i 的右子女是 R,则 R 的双亲是 i
    int parent=U;              //判断 U 是否是 V 的后代
    while(parent!=V && parent!=0)parent=T[parent];
    if(parent==V){printf("结点 U 是结点 V 的后代");return(1);}
    else{ printf("结点 U 不是结点 V 的后代");return(0);}
}结束 Generation
```

2. 由于以双亲表示法作为树的存储结构，所以找结点的双亲容易。因此我们可求出每一结点的层次，取其最大层次就是树的深度。对每一结点，找其双亲，再找双亲的双亲，直至（根）结点双亲为 0 为止。

```
int Depth(Ptree t){
    //求以双亲表示法作为存储结构的树的深度,Ptree 的定义参看教材
    int maxdepth=0;
    for(i=1;i<=t.n;i++){
      temp=0; f=i;
      while(f>0)
      {temp++; f=t.nodes[f].parent;}      // 深度加 1,并取新的双亲
      if(temp>maxdepth) maxdepth=temp;   //最大深度更新
    }
    return(maxdepth);  //返回树的深度
} //结束 Depth
```

3.

```
typedef struct DataType{
     int key;
    …
} DataType;
void shift(DataType R[],int n){
       //假设 R[1..n-1]是大堆 ,本算法把 R[1..n]调成大堆
       j=n;  R[0]=R[j];
       for(i=n/2;i>=1;i=i/2)
         if(R[0].key>R[i].key){ R[j]=R[i];j=i;} else break;
       R[j]=R[0];
}//shift
```

4.

```
void Forest2BT(Forest F,BiTree &B){
   if(F==NULL)B=NULL;
   else{
     B->data=F->data;
     Forest2BT(F->firstchild,B->lchild);
     Forest2BT(F->nextsibling,B->rchild);
   }
}
```

5.

```
void preorder(BiTree T,int i){
   //先序遍历,并打印结点数据及层号 i
   if(T==NULL)return;
```

```
    printf( T->data,i);  //打印信息
    preorder(T->lchild,i+1); //先序遍历左子树
    preorder(T->rchild,i+1); //先序遍历右子树
}
```

第 4 章参考答案

一、填空题

1. 入度　　2. n　　3. $n(n-1)$　　4. v_1，v_2，v_3，v_6，v_5，v_4，v_1，v_2，v_5，v_4，v_3，v_6

5. n　　6. $O(e\log_2 e)$，稀疏　　7. $O(n^3)$　　8. $((e),a,(a,f),(()))$

9. $head(tail(head(tail(A))))$　　10. (a,b)

二、选择题

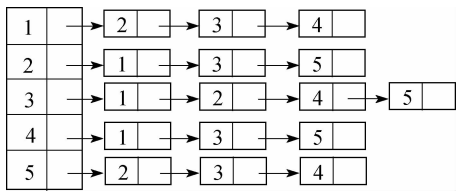
1. B　　2. B，B，D　　3. D　　4. D　　5. C，C　　6. B　　7. B　　8. A　　9. B　　10. D

三、简答题

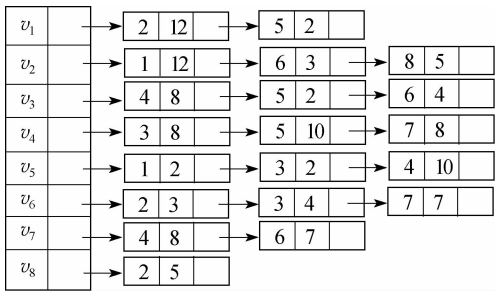
1. 证明：在 n 个顶点的无向完全图中，每个顶点与其余顶点均有一条边，第一个顶点到其他各顶点的边数为 $n-1$，第二个顶点到其余顶点的边数为 $n-1$，但它与第一个顶点之间的边已在第一个顶点的边中，故第二个顶点到其余 $n-2$ 个顶点的边数为 $n-2$，……，而第 $n-1$ 个顶点到其余剩下的第 n 个顶点的边数为 1，则总的边数为：

$$(n-1)+(n-2)+\cdots+2+1=n(n-1)/2$$

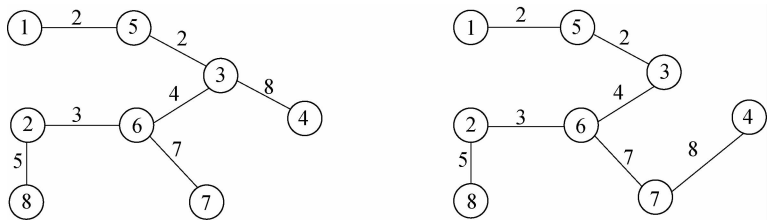
2.
$$A=\begin{bmatrix}1&1&1&1&0\\1&1&1&0&1\\1&1&1&1&1\\1&0&1&1&1\\0&1&1&1&1\end{bmatrix}$$



3. 1）邻接表如下图所示。



2）最小代价生成树如下图所示。



3）深度优先遍历顶点序列：1，2，6，3，4，5，7，8

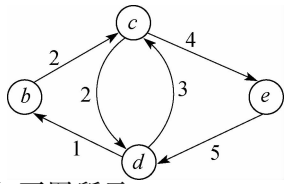
广度优先遍历顶点序列：1，2，5，6，8，3，4，7

4）v_1 到 v_2 的最短路径是：

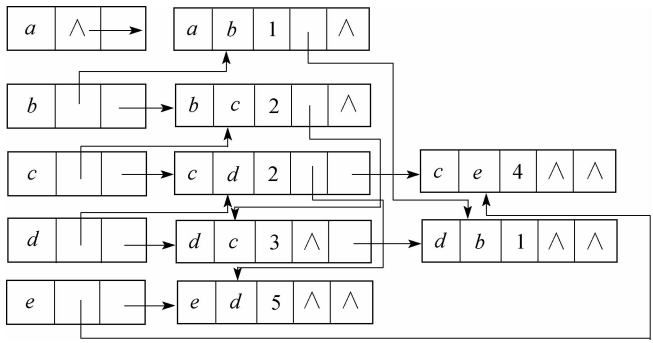
v_1, v_5, v_3, v_6, v_2

长度为 11。

4. 1）有强连通分量，如下图所示。



2）有向图的十字链表存储结构如下图所示：

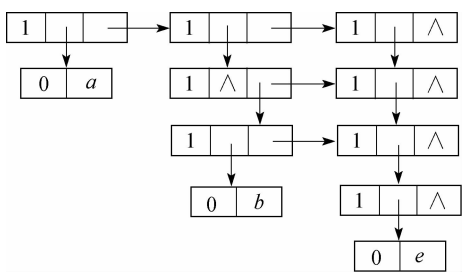


3）各村庄之间的最近距离：

	a	*b*	*c*	*d*	*e*
a	—	1	3	5	7
b	—	—	2	4	6
c	—	3	—	2	4
d	—	1	3	—	7
e	—	6	8	5	—

医院设在 *b* 村庄较好，因为这样各村庄到医院距离和最小。

5. 广义表的存储结构如下图所示：



设 $A=(a,((),b),(((e))))$，$Head(Head(Head(Head(Tail(Tail(A))))))=e$。

四、算法设计题

1.

```
Struct jzGra{
    Char vex[n];
    Int edge[n][n];
}
Void creategraph(jzGra g)
  {
```

```
        For(i = 0;i < n;i ++ )
          Scanf(" % c",g. vex[i]);
        For(i = 1;i < n;i ++ )
          For(j = 1;j < n;j ++ )
          g. edge[i][j] = 0;
        for(k = 0;k < e;k ++ )
          { scanf(" % d, % d",i,j);
            g. edge[i][j] = 1;
            g. edge[j][i] = 1;
          }
}
```

2. 事实上，图的广度优先遍历算法正好符合本题要求，因此算法如下：

```
void bfs(Datagraph g;int v0)
{ n = nodes(g);  setnull(Q);
  for(i = 1;i <= nodes(g);i ++ )
      visited[i] = 0;
  printf(" % d",v0);visited[v0] = 1;addqueue(Q,v0);
  while(! empty(Q)){
      v = delqueue(Q);
      w = firstadj(g,v);
     while(w! = 0){
       if(! visited[w]){
          printf(" % d",w);visited[w] = 1;addqueue(Q,w);
       }
     w = nextadj(g,v,w);
  }
}
```

3.

```
int connect(Datagraph g)
{
   for(i = 1;i <= nodes(g);i ++ )
   visited[i] = 0;
k = 0;
for(i = 1;i <= nodes(g);i ++ )
   if(! visited[i]){
         k ++ ;dfs(g,i);
   }
if(k == 1)return 1;
else return 0;
}
void dfs(Datagraph g;int v)
{
   visited[i] = 1;
   w = firstadj(G,v);
   while(w! = 0){
    if(! visited[w]){
      dfs(g,w);
```

```
        w = nextadj(G,v,w);
    }
}
```

4. 本题中，为满足问题 1）和 2）的要求，最好是设计一个算法以求出给定顶点的出度。可将该算法定义为函数形式，*outdegree*(G，v）返回图 G 中顶点 v 的出度。在此基础上，实现问题 1）和 2）就比较容易。问题 3）较容易实现，只需检查表头指针是否为空即可。问题 4）的实现即是在第 i 个表中查找是否有值为 j 的元素。各算法如下：

```
int outdegree(adjlist G;int v)
{
    p = G[v]. firste;
    outd = 0;
    while(p! = NULL){
    outd = outd + 1;
    p = p->next;
}
return outd;
}
```

1）

```
void OutDs(adjlist G)
{
printf("outdegree:");
for(i = 0;i<n;i++)
   {
        x = outdegree(G,i);
        printf("<%d ,%d>",i,x);
   }
}
```

2）

```
maxoutDGs(adjlist G;int maxv)
{
   maxdg = 0; maxv = 0;
   for(i = 0;i<n;i++)
 {
       x = outdegree(G,i);
       if(x>maxdg){
         maxdg = x;   maxv = i;
       }
     }
 printf("Vertex of Max Outdegree: %d, %d",maxv,G[maxv]. vinfo);
}
```

3）

```
int OutDG0(adjlist G)
{
   num0 = 0;
   for(i = 0;i<n;i++)
     if(outdegree(G,i) == 0) num0++;
   return num0
```

```
    }
```

4）
```
int have_arc(adjlist G;i,j;integer)
{  p = G[i].firste;
   while(p! = NULL && p -> next! = j) p = p -> next;
   if(p! = NULL) return 1;
   else return 0;
}
```

5.
```
struct node{
   int tag;
   union{
    datatype data;
    node * hp,tp;
   }u
 }
 createlist(node * s1)
{
   scanf("%c",x);
   switch(x){
    case":s1 = NULL;
    case'(':new(ls);ls -> tag = 1;createlist(ls -> hp);
    case'a' - 'z':ls -> tag = 0;ls -> data = x;
   }
scanf("%c",x);
if(ls! = NULL)
     if(x = ',') createlist(ls -> tp);
else
   ls -> next = NULL;
}
```

第 5 章参考答案

一、填空题

1. 可行性
2. 时间复杂度、空间复杂度
3. 循环、递归
4. 贪心
5. $D_k(i,\ j)=\min\{D_{k-1}(i,\ j),\ D_{k-1}(i,\ k)\ +D_{k-1}(k,\ j)\}$

二、选择题

1. A　　2. B　　3. D，B　　4. B　　5. C，B

三、简答题

1. 复杂度　　C_1 可解规模　　C_2 可解规模　　可解规模的关系

$O(n)$	N_{11}	N_{21}	$N_{21}=10N_{11}$
$O(n\log_2 n)$	N_{12}	N_{22}	$N_{22}=10N_{12}$
$O(n!)$	N_{13}	N_{23}	$N_{23}=N_{13}+$小常数

不必追求高效算法、低效算法可由高速计算机来弥补的看法是错误的。

2. 循环控制变量 i 从 1 增加到 n，最外层循环体执行 n 次，循环控制变量 j 从 1 增加到 n，中间层循环体执行 n 次，循环控制变量 k 从 1 增加到 n，最内层循环体执行 n 次，所以该程序片段总的时间代价为 $O(n^3)$。

3. 1) $k=0$

   2) $j<=N$

   3) $k=k+1$

   4) $d[i]+6$

   5) $O(N)$

四、算法设计题

1. 设有 n 位选手被顺序编号为 1，2，…，n，比赛的日程表是一个 n 行 $n-1$ 列的表，第 i 行 j 列的内容是第 i 号选手第 j 天的比赛对手。用分治法设计日程表，就是从其中一半选手（2^{m-1}位）的比赛日程导出全体（2^m 位）选手的比赛日程。从众所周知的只有两位选手的比赛日程出发，反复这个过程，直至为 n 位选手安排好比赛日程为止。

   为了从 2^{m-1} 位选手的比赛日程表中导出 2^m 位选手的比赛日程表，假定只有 8 位选手，如下图，若 1 至 4 号选手之间的比赛日程填在表的左上角（4 行 3 列），5 至 8 号选手之间的比赛日程填在日程表的左下角（4 行 3 列），则左下角的内容可以由左上角对应项加上数 4 得到。至此，剩下的右上角（4 行 4 列）是为编号小的 1 至 4 号选手与编号大的 5 至 8 号选手之间的比赛安排日程。最后，参照右上角得到右下角的比赛日程。

	1
1	2
2	1

a）2位选手比赛日程表

	1	2	3
1	2	3	4
2	1	4	3
3	4	1	2
4	3	2	1

b）4位选手比赛日程表

	1	2	3	4	5	6	7
1	2	3	4	5	6	7	8
2	1	4	3	6	7	8	5
3	4	1	2	7	8	5	6
4	3	2	1	8	5	6	7
5	6	7	8	1	4	3	2
6	5	8	7	2	1	4	3
7	8	5	6	3	2	1	4
8	7	6	5	4	3	2	1

c）8位选手比赛日程表

为 2^k 位选手安排比赛日程的程序如下所示：

```
int a[MAXN+1][MAXN]
void main( ){
    int twoml,twom,i,j,m,k;
    printf("请输入 n(=2 的 k 次幂)位选手,请输入 k\n");
    scanf("%d",&k);
    a[1][1]=2; a[2][1]=1;                          /* 设置两位选手的比赛日程 */
    m=1;twoml=1;
    while(m<k){
    m++;
    twoml+=twoml;   twom=2*twoml;                  /* 为 2^m 位选手安排比赛日程 */
    /* 填日程表的左下角   */
    for(i=twoml+1;i<=twom;i++)
       for(j=1;j<=twoml-1;j++)
          a[i][j]=a[i-twoml][j]+twoml;
```

```
    /*  填日程表的右上角  */
     a[1][twoml] = twoml + 1;                                    /* 填日程表右上角第 1 列 */
     for(i = 2;i <= twoml;i ++ )  a[i][twoml] = a[i - 1][twoml] + 1;
  /* 填日程表右上角的其他列,参照前一列填当前列  */
     for(j = twoml + 1;j < twom;j ++ ){
          for(i = 1;i < twoml;i ++ ) a[i][j] = a[i + 1][j - 1];
          a[twoml][j] = a[1][j - 1];
       }
       /* 填日程表的右下角  */
       for(j = twoml;j < twom;j ++ )
          for(i = 1;i <= twoml;i ++ )
            a[a[i][j]][j] = i;
       for(i = 1;i <= twom;i ++ ){
           for(j = 1;j < twom;j ++ ) printf(" % 4d",a[i][j]);
           printf("\n");
        }
}
```

2. 可用试探法找到问题的解，即从第一个方格开始，为当前方格寻找一个合理的整数填入，并在当前位置正确填入后，为下一方格寻找可填入的合理整数。如不能为当前方格找到一个合理的可填整数，就要回退到前一方格，调整前一方格的填入数。当为第 9 格也填入合理的整数后，就找到了一个解，将该解输出，并调整第 9 格的填数去找下一个解。

   为找到一个满足要求的 9 个数的填法，从还未填一个数开始，按某种顺序（如从小到大的顺序）每次在当前位置填入一个整数，然后检查当前填入的整数是否满足要求，在满足要求的情况下，继续用同样的方法为下一方格填入整数。如果最近填入的整数不能满足要求，就改变填入的整数。如对当前方格试尽所有可能整数都不能满足要求，就得回退到前一格，并调整前一格填入的整数。如此重复执行扩展、检查或调整，直到找到一个满足问题要求的解。

   方格填数的程序如下：

```
/* 测试所求的值是否是质数的函数 */
int isPrime(int m)
{ int I;
 int primes[] = {2,3,5,7,11,13,17,19,23,29, - 1};
 if(m == 1 || m % 2 == 0) return 0     /* 不是质数  */
 for(i = 0;primes[i] > 0;i ++ )
    if(m == primes[i]) return 1;   /*  是质数  */
 for(i = 3;i * i <= m;){
    if(m %  i == 0) return 0;
    i + = 2;
 }
 return 1;
}
int checkMatrix[][3] = {{ - 1},{0, - 1},{1, - 1},          /* 每个元素的邻接元素位置 */
                  {0, - 1},{1,3, - 1},{2,4, - 1},
```

```
                {3,-1},{4,6,-1},{5,7,-1}};
int selectNum(int start)                          /* 选择下一个数,b[]是标志数组 */
{ int j;
 for(j=start;j<=N;j++)
   if(b[j]) return j;
   return 0;
}
int check(int pos)                                /*   检查相邻的元素和是否为质数   */
{ int I,j;
  if(pos<0) return 0;
  for(i=0;(j=checkMatrix[pos][i])>=0;i++)
     if(! isPrime(a[pos]+a[i])) return 0;
  return 1;
}
int extend(int pos)                               /* 扩展元素 */
{ a[++pos]=selectNum(1); b[a[pos]]=0;
  return pos;
}
int change(int pos)                               /* 改变所填的数字   */
{ int j;
 while(pos>=0 &&(j=selectNum(a[pos]+1))==0)
   b[a[pos--]]=1;
 if(pos<0) return -1;
 b[a[pos]]=1;
 a[pos]=j;
 b[j]=0;
 return pos;
}
void find()                                       /* 通过回溯法找出所有合理解 */
{ int ok=1,pos=0;
 a[pos]=1; b[a[pos]]=0;
 do{
   if(ok)
      if(pos==8){
        write(a);
        pos=change(pos);
      }
      else  pos=extend(pos);
   else pos=change(pos);
   Ok=check(pos);
   }while(pos>=0);
}
```

3. 线段类型定义如下：

```
struct xianduan{
    int l,r;
};            /* l 表示左端点,r 表示右端点     */
```

用一维数组 *seg* 存放线段信息：xianduan seg[100]。

采用贪心策略解题：

- 将所有的线段按照左端点从左到右排序，并按照这个顺序尝试覆盖整个区间。
- 定义一个“已覆盖区间”的概念，表示目前被选择了的线段可以完全覆盖的部分。因为按照左端点从左到右的顺序考虑每条线段，所以这个区间的左端点始终是 0，右端点初始化为 0，因为开始时没有任何覆盖。
- 设置循环变量 i，依次检查可选线段。
- 运用贪心策略，在所有左端点小于等于 $NowRight$ 的线段中选择右端点最大的一个，将它覆盖上去，使得已覆盖区间扩大。重复本步，直到覆盖完毕（$NowRight \geqslant m$）为止。

算法的主要步骤如下：

```
void greedy(){
      int num = 0                        /* 定义覆盖需要的线段数量,初始值为 0   */
      int LastRight,NowRight = 0;        /* 定义整型变量 */
      i = 0;
      while(NowRight < m){               /* 没有覆盖完   */
           LastRight = NowRight;         /* 暂存 NowRight 到 LastRight   */
           while(i < n && Seg[i].l <= LastRight){
               if(Seg[i].r > NowRight)NowRight = Seg[i].r;
               i++;
            }
      /* 以上 3 行代码在所有左端点在原先的 NowRight 左边的线段中选择一个右端点最靠右的,并把这个右端点
         赋给 NowRight   */
      num++;
      if(i == n && NowRight < m || LastRight == NowRight)break;
      /* 如果线段已用完但是还没有覆盖完,或者将选定的线段覆盖上之后没有产生任何效果,则跳出循环,无解 */
    }
      if(NowRight < m) printf("无解");
      else printf("the answer is %d",num);
}
```

4. 我们先对算法中涉及的变量说明如下：

n：总的食物项数。

v：营养价值数组，下标从 1 到 n，对应第 1 到第 n 项食物的营养价值。

p：价格数组，下标从 1 到 n，对应第 1 到第 n 项食物的价格。

M：总价格标准，即套餐的价格不超过 M。

x：解向量（数组），下标从 1 到 n，其元素值为 0 或 1，其中元素值为 0 表示对应的食物不出现在套餐中，元素值为 1 表示对应的食物出现在套餐中。nv：$n+1$ 行 $M+1$ 列的二维数组，其中行和列的下标均从 0 开始，$nv[i][j]$ 表示由前 i 项食物组合且价格不超过 j 的套餐的最大营养价值。问题最终要求的套餐的最大营养价值为 $nv[n][M]$。

本问题实质上是一个 0-1 背包问题，该最优化问题的目标函数是

$$\max \sum_{i=1}^{n} v_i x_i \qquad (x_i = 0,1)$$

约束函数是

$$\sum_{i=1}^{n} p_i x_i \leqslant M \qquad (x_i = 0,1)$$

0-1 背包问题可用动态规划策略求得最优解，求解的递归式为

$$nv[i][j] = \begin{cases} 0 & i = 0 \text{ 或 } j = 0 \\ nv[i-1][j] & j < p_i \\ \max(nv[i-1][j], nv[i-1][j-p_i] + v_i) & j \geqslant p_i \end{cases}$$

算法实现如下：

```
MaxNutrientValue( ){
      int i,j,nv[ n][M ];
      for(i = 0;i <= n;i ++ ) nv[i][0] = 0;
      for(j = 1;j <= n;j ++ ) nv[0][j] = 0;
      for(i = 1;i <= n;i ++ )
         for(j = 1;j <= M;j ++ )
            if(j < p[i])              /* 若食物 mi 不能加入到套餐中 */
               nv[i][j] = nv[i - 1][j];
            else if(nv[i - 1][j]>= nv[i - 1][j - p[i]] + v[i])
               nv[i][j] = nv[i - 1][j];
              else
               nv[i][j] = nv[i - 1][j - p[i]] + v[i];
      j = M;
      for(i = n;i => 1;i - - )
          if  (nv[i][j] = nv[i - 1][j])
               x[i] = 0;
           else  { x[i] = 1;
                j = j - p[i];
                }
}
```

第 6 章参考答案

一、选择题

1. B　2. C　3. D　4. D　5. C　6. D　7. A　8. D　9. D　10. B

二、填空题

1. 101

2. (12, 63, 36), (55, 40, 82), (23, 74)

3. 6, 9, 11, 12

4. 索引值，子表开始

5. 2

6. 4/3

7. $O(n)$，$O(\log_2 n)$

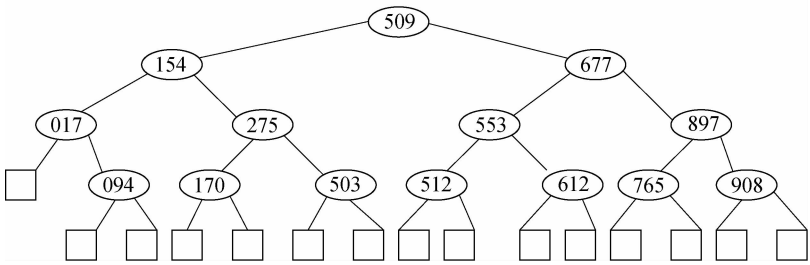
8. 1.5，1.5

9. 素数

10. 3，2

三、简答题

1.



$$ASL_{\text{succ}} = \frac{1}{14}\sum_{i=1}^{14} C_i = \frac{1}{14}(1+2*2+3*4+4*7) = \frac{45}{14}$$

$$ASL_{\text{unsucc}} = \frac{1}{15}\sum_{i=0}^{15} C'_i = \frac{1}{15}(3*1+4*14) = \frac{59}{15}$$

2. 1）不同。因为有序顺序表查找到其关键字比要查找值大的对象时就停止查找，报告失败信息，不必查找到表尾；而无序顺序表必须查找到表尾才能断定查找失败。
   2）相同。查找到表中对象的关键字等于给定值时就停止查找，报告成功信息。
   3）不同。有序顺序表中关键字相等的对象相继排列在一起，只要查找到第一个就可以连续查找到其他关键字相同的对象。而无序顺序表必须查找全部表中的对象才能确定相同关键字的对象都找了出来，所需时间就不相同了。
3. 散列表的定义见教材，散列表的优点是可以直接定址，可以节省查找时间；其缺点是容易产生冲突，且该冲突不能避免，只能优化。
4. 1）用顺序查找法：关键字可以按任意的次序排列，有序无序均可实现。
   2）采用二分查找法：关键字序列必须按从小到大或从大到小的顺序排列，而且只能顺序存储。
5. 当 α 非常接近 1 时，整个散列表几乎被装满。由于线性探查法在关键字同义时解决冲突的办法是线性地向后查找，当整个表几乎装满时，它就很类似于顺序查找了。
   当 α 比较小时，关键字冲突的几率比较小，一般情况下只要按照散列函数计算出的结果能够一次性就找到相应结点，因此它的平均查找时间接近于 1。

四、算法设计题

1.
```
int  BinSrch(DataType r[ ],KeyType  k,int low,int high{
  //在长为 n 的有序表中查找关键字 k,若查找成功,返回 k 所在位置,查找失败返回 0。
  if(low≤high) //low 和 high 分别是有序表的下界和上界
  {   mid = (low + high)/2;
      if(r[mid].key = = k)return mid;
      else if(r[mid].key>k)return (BinSrch(r,k,mid + 1,high));
      else return (BinSrch(r,k,low,mid - 1));
  }
  else  return  0; //查找失败。
}//算法结束
```

算法的时间复杂度为 $O(\log_2 n)$。

2. 本题实质上是排序题，因涉及顺序表和索引表而放在这里。顺序表无序，索引表有序。由顺序表中的关键字及其下标地址组成索引表中的一项。顺序表有 m 个记录，索引表应有 m

项。建立索引表宜采用“直接插入排序”，这样才能在顺序表有序时，算法的时间复杂度在最好情况下达到 $O(m)$。

```
#define m  顺序表中记录个数
typedef struct node
  {keytype key;                                  //关键字
   int  adr;                                     //该关键字在顺序表中的下标
  }idxnode;                                      //索引表的一项
typedef struct node
  {keytype key;                                  //关键字
   anytype other;                                //记录中的其他数据
  }datatype;
  void IndexT(idxnode index[m+1],datatyp seq[m+1])
    //给有 m 个记录的顺序表 seq 建立索引表 index
    {index[1].key = seq[1].key; index[1].adr = 1;
     for(i = 2,i<= m,i++)
     {j = i-1;
     index[0].key = seq[i].key;                  //监视哨
     while(index[j].key < index[0].key)
          {index[j+1] = index[j]; j--;}
     index[j+1].key = index[0].key;              //关键字放入正确位置
     index[j+1].adr = i;                         //第 i 个记录的下标
    }
  }//IndexT
```

3.
```
CNodeType ChainHashSearch(CHashTable T,KeyType K)
{  //查找关键字值为 K 的结点，若有返回该结点指针，否则返回 NULL
   CNodeType * p;
   int addr;
   addr = K % m;//求散列函数值
   p = T[addr];
   while(p)&&(p->key!  = K)
   p = p->next;
   return p;
}
```

4. 根据题意，查找满足条件的数据可以采用顺序查找法，即从数据序列的第一个数据开始进行关键字值的比较。算法如下：

```
DataType SeqStr(DataType FIR,KeyType def){
    p = FIR;                  //第一个数据
    while(p!  = nul){
        if(p.key = = def)
          return  p;          //查找成功
      p = p->next;            //下一个数据
     }
     return nul;
}
```

5. 构造散列表时，先把输入序列存入数组，然后顺序填入相应的链表。填入算法类似于单链

表的建立，只是链表头指针存放的位置由散列函数计算得到。

```
int Hash(int key)                                    /* 散列函数 */
{     return(int)(key % M);     }
hashcreat(HashNode * h[],int k)                      /* 构造散列表函数 */
{     int i;
      HashNode * p;
      i = Hash(k);                                    /* 确定链表头指针存放位置 */
      p = (struct node * )malloc(sizeof(HashNode));   /* 给结点分配空间 */
      p -> key = k;
      p -> next = NULL;
      h[i - 1] = p;
}
HashNode * hashfind(HashNode * h[],int k)            /* 查找函数 */
{  int i;
   HashNode * p;
   i = Hash(k);
   p = h[i - 1];                                      /* 链表头指针 */
   while(p! = NULL)
   {     if(p -> key = = k)return p;
         else p = p -> next;
         }
   return NULL;
}
main()
{     HashNode * p, * h[12];
      int k[12] = { 9,31,26,19,1,13,2,11,27,16,5,21};  /* 数据存入数组 */
      int m,n = 12;
      for(m = 0;m < n;m ++ )h[m] = NULL;               /* 初始化顺序表,链表头指针全为空 */
      for(m = 0;m < n;m ++ )hashcreat(h,k[m]);
      /* 从数组中取数,依次加入散列表 */
      for(m = 0;m < n;m ++ )                           /* 输出散列表中的元素 */
      {     p = h[m];
            while(p! = NULL)  {printf(" % 5d",p -> key);p = p -> next;}
            printf("\n")
      }
      printf("input k:");
      scanf(" % d",&m);                                /* 输入待查数据 */
      p = hashfind(h,m);                               /* 查找 */
      if(p)printf("找到元素 % d",p -> key);
      else  printf("未找到元素!");
}
```

第 7 章参考答案

一、选择题

1. D　2. D　3. C　4. A　5. B　6. C　7. A　8. C　9. A　10. B

二、填空题

1. 比较，交换
2. 1) i<n-i+1 2) j<=n-i+1 3) r[j].key<r[min].key 4) min!=i 5) max==i 6) r[max]<-->r[n-i+1]
3. {Q，A，C，S，Q，D，F，X，R，H，M，Y}，{F，H，C，D，Q，A，M，Q，R，S，Y，X}
4. {D，Q，F，X，A，P，B，N，M，Y，C，W}
5. {A，C，H，Q，Q，Y，M，S，R，D，F，X}
6. {972，203，564，624，135，416，347，798，078}
7. 1) 1 2) a[i]=t 3) (i=2;i<=n;i+=2) 4) 1 5) flag
8. {203，135，078，347，416，564，798，624，972}
9. 3，{10，7，－9，0，47，23，1，8，98，36}
10. 归并排序（或 2 路归并排序）

三、简答题

1. 可以做到。取 a 与 b 进行比较，c 与 d 进行比较。设 $a>b$，$c>d$（$a<b$ 和 $c<d$ 情况类似），此时需 2 次比较，取 b 和 d 比较，若 $b>d$，则有序 $a>b>d$；若 $b<d$ 则有序 $c>d>b$，此时已进行了 3 次比较。再把另外两个元素按二分插入排序方法，插入到上述某个序列中共需 4 次比较，从而共需 7 次比较。
2. 对冒泡算法而言，初始序列为反序时交换次数最多。若要求从大到小排序，则表现为初始是上升序。
3. 快速排序，以待排序列的第一个数据为支点的快速排序，其他排序方法排不出这种结果。
4. 初始序列：25，84，21，46，13，27，68，35，20，65，31，76，18

   25 21 46 13 27 68 35 20 65 31 76 18 **84**； **13** 25 21 46 18 27 68 35 20 65 31 76 **84**

   13 21 25 18 27 46 35 20 65 31 68 **76 84**； **13 18** 21 25 20 27 46 35 31 65 68 **76 84**

   13 18 21 20 25 27 35 31 46 65 **68 76 84**； **13 18 20** 21 25 27 31 35 46 65 **68 76 84**

   13 18 20 21 25 27 31 35 46 65 **68 76 84**；未交换，已有序
5. 初始序列：29，18，25，47，58，12，51，10

   1) 2 路归并

   第一趟：18，29，25，47，12，58，10，51

   第二趟：18，25，29，47，10，12，51，58

   第三趟：10，12，18，25，29，47，51，58

   2) 快速排序

   第一趟：10，18，25，12，29，58，51，47

   第二趟：10，18，25，12，29，47，51，88

   第三趟：10，12，18，25，29，47，51，88

四、算法设计题

1. 保存划分的第一个元素。以平均值作为界值，进行普通的快速排序，最后界值的位置存入已保存的第一个元素，若此关键字小于平均值，则它属于左半部，否则属于右半部。

```
int partition(RecType r[],int l,int h)  {
    int i=l,j=h,avg=0;
```

```
        for(;i<= h;i++)avg+ = R[i].key;
        i = l; avg = avg/(h - l + 1);
      while(i < j)
      {  while(i < j &&R[j].key >= avg)j--;
         if(i < j)R[i] = R[j];
         while(i < j &&R[i].key <= avg)i++;
         if(i < j)R[j] = R[i];
      }
      if(R[i].key <= avg)return  i; else return  i - 1;
    }
    void quicksort(RecType R[],int S,T){
         if(S < T) {
             k = partition(R,S,T);
         quicksort(R,S,k);  quicksort(R,k + 1,T);} //递归调用
    }
```

2. 以 Kn 为界值的一趟快速排序。将上题算法改为以最后一个为界值先从前向后再从后向前。

```
int  Partition(RecType K[],int l,int n){
   //交换记录子序列 K[l..n]中的记录,使界值记录到位,并返回其所在位置,
   //此时,在它之前(后)的记录均不大(小)于它
   int i = l; j = n; K[0] = K[j]; x = K[j].key;
   while(i < j){
     while(i < j && K[i].key <= x) i++;
     if(i < j)K[j] = K[i];
     while(i < j && K[j].key >= x) j--;
     if(i < j)K[i] = K[j];
   }//while
   K[i] = K[0]; return  i;
}//Partition
```

3.
```
typedef struct Node {
          int key; datatype info;
}Node;
void CountSort(RecType a[],b[],int n){ //计数排序算法,将 a 中的记录排序放入 b 中
          for(i = 0;i < n;i++) { //对每一个元素
            for(j = 0,cnt = 0;j < n;j++)
               if(a[j].key < a[i].key)cnt++; //统计关键字比它小的元素个数
            b[cnt] = a[i];
      }
}//Count_Sort
```

4.
```
Node * ChainSelectSort(Node * head){
   Node * headnew = null, * headtail = null, * p, * q, * pre;
   while(head){
    min = head -> key; q = p;
    for(p = head; p! = null; p = p -> next){
       if(p -> key < min){ q = p; min = p -> key; pre = p; }
    }
    if(q = = p)head = head -> next;
```

```
    else pre->next = q->next;
    if(headnew == null)headnew = headtail = q;
    else {
       headtail->next = q;  headtail = q;  }
    }
    return headnew;
}
```

5.
```
void BubbleSort2(int a[],int n){ //相邻两趟向相反方向冒泡的冒泡排序算法
    change = 1;low = 0;high = n - 1; //冒泡的上下界
    while(low < high && change){
    change = 0;                  //设不发生交换
    for(i = low;i < high;i++) //从左向右冒泡
       if(a[i] > a[i + 1]) {a[i]<--> a[i + 1];change = 1;} //有交换,修改标志 change
    high--; //修改上界
    for(i = high;i > low;i--)//从右向左冒泡
      if(a[i] < a[i - 1]) {a[i]<--> a[i - 1];change = 1;}
    low++; //修改下界
   }//while
}//BubbleSort2
```