第六章

一、选择题

1．设无向图G 的顶点数为n（$n\geq2$），边数为e（$e\geq1$），下列关于该无向图的叙述中正确的是 D 。

A．至少有一个顶点的度为奇数　B．当$e\geq n-1$时，无向图G 是连通图

C．至少有一个顶点的度为偶数　D．所有顶点的度之和为偶数

2．设有5 个结点的无向图，该图至少应有 C 条边才能确保是一个连通图。

A．5　B．6　C．7　D．8

3．用有向无环图描述表达式$(a+b)*c+(a+b+c)*d$，至少需要 B 个顶点。

A．8　B．9　C．10　D．11

4．已知某无向图有20 条边，其中度为4 的顶点个数为2、度为3 的顶点个数为6、度为2 的顶点个数为4、其他顶点的度均不超过1，则该图所含的顶点个数至少是 B 。

A．17　B．18　C．19　D．20

5．若将n 个顶点 e 条弧的有向图采用邻接矩阵存储，则拓扑排序算法的时间复杂度是 D 。

A．$O(n\log e)$　B．$O(n\bullet e)$　C．$O(n+e)$　D．$O(n^2)$

6．若用邻接矩阵存储有向图，邻接矩阵为上三角矩阵（或下三角矩阵）则关于该图的结论是 D 。

A．该图可能有回路　B．该图一定没有回路，有唯一的拓扑排序

C．该图不存在拓扑排序　D．该图一定没有回路，拓扑排序不一定唯一

7．图6.29 所示为AOE 网，下列不是其拓扑排序的选项是 C 。

A．1,2,3,4,5,6　B．1,3,2,5,4,6　C．1,4,3,2,5,6　D．1,3,2,4,5,6

8．图6.29 中有9 个活动的AOE 网，活动a_5 的最早开始和最迟开始时间分别是 B 。

A．6、6　B．6、8　C．7、13　D．13 、 15

9. 使用迪杰斯特拉（Dijkstra）算法求图6.30 中从顶点1 到其他各顶点的最短路径，依次得到各最短路径的目标顶点是__C__。

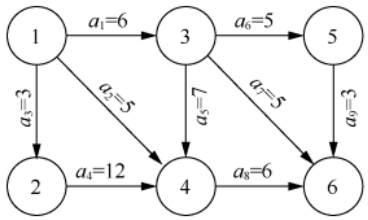


图 6.29　AOV 网（1）

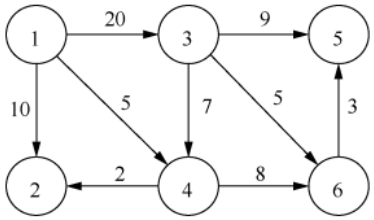


图 6.30　AOV 网（2）

A．1,2,3,4,5,6　　B．1,3,5,6,4,2　　C．1,4,2,6,5,3　　D．1,3,2,4,5,6

10. 从图6.30 所示顶点1 出发进行深度优先遍历，不能得到的顶点序列是__A__。

A．1,2,3,4,5,6　　B．1,3,4,2,6,5　　C．1,4,2,6,5,3　　D．1,3,5,4,2,6

11. 可借助__A__算法判断有向图中是否存在回路。

A．拓扑排序　　B．迪杰斯特拉　　C．Floyd　　D．Prim

12. 对图 6.31 所示的无向连通网，使用 Prim 算法求从顶点 *A* 出发，得到的最小生成树是__C__。

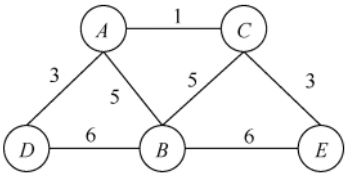
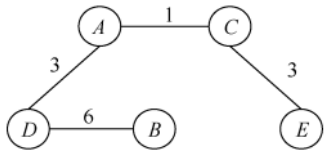
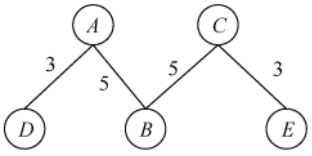
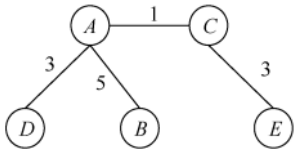
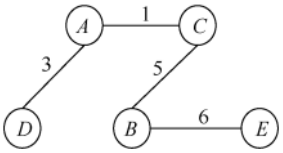


图 6.31　无向连通网（1）

A.


B.


C.


D.


二、填空题

1. 有 *n* 个顶点的强连通图至少有__n__条弧。

2. 有 *n* 个顶点的无向图最多__n(n-1)/2__条边。

3. 在有向图的邻接矩阵中，每一行包含的“1”的个数为对应顶点的__出度_____。

4. *n* 个顶点 *e* 条边的有向图采用邻接表存储，求某结点度的算法时间复杂度为__O(n)__。

5. 设有一个稀疏图，则采用邻接表 存储比较节省存储空间。

6. 十字链表适用于有向图（网），邻接多重表适用于无向图（网）。

7. 对 *n* 个顶点 *e* 条边的无向连通图进行深度优先搜索遍历的路径上，经过 n-1 条边。

8. 图的深度优先遍历类似于二叉树的先根遍历。

9. 按广度优先搜索遍历图的算法需要借助的辅助数据结构是队列。

10. 可以借助于图的遍历算法 算法求出无向图的所有连通分量。

11. Prim 算法适用于稠密 图，Kruskal 算法适用于稀疏 图。

12. 可以借助于拓扑排序 算法判断一个有向图是否为 DAG 图。

三、问答题

1. 求解图 6.32 所示有向图的全部强连通分量

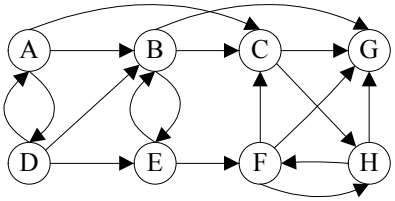
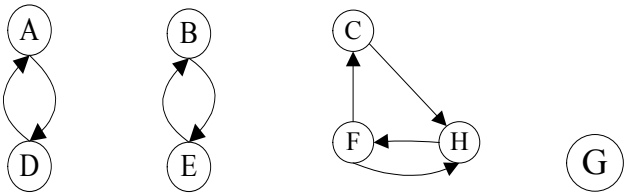


图6.32 有向图G

答案：利用遍历算法求出 4 个强连通分量



2. 已知无向图 G=（V，R），G 的邻接表如图 6.33 所示。① 画出该无向图；② 根据该存储结构保存的顶点和边的次序，从顶点 A 出发，进行深度优先搜索遍历，依次写出访问到的顶点序列；③ 进行广度优先搜索遍历，依次写出访问到的顶点序列。

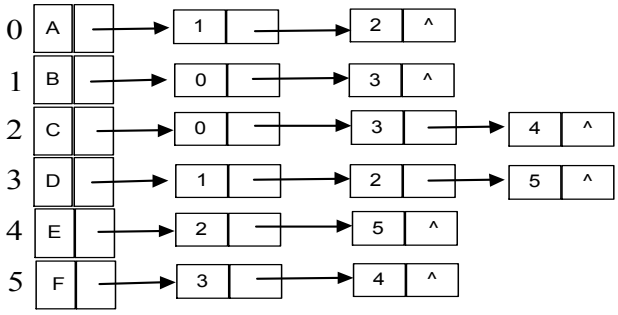
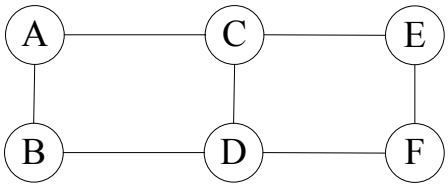


图 6.33 无向图的邻接表

解答：

①



② A B D C E F

③ A B C D E F

3. 简述图的广度优先搜索遍历与二叉树的按层遍历的异同点。

相同点： 都需要借助队列完成遍历

相异点： 图中可能存在回路，遍历过程中可能再次到达已经访问过的顶点，所以需要 visited 数组，而二叉树是不需要的。

4. 如图 6.34 所示无向连通网，使用 Prim 算法，从顶点 A 出发，其最小生成树。

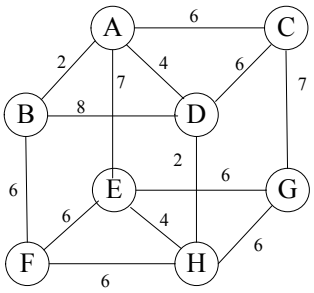
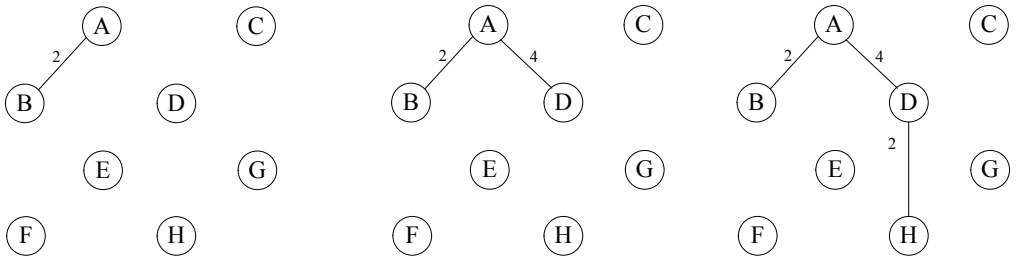
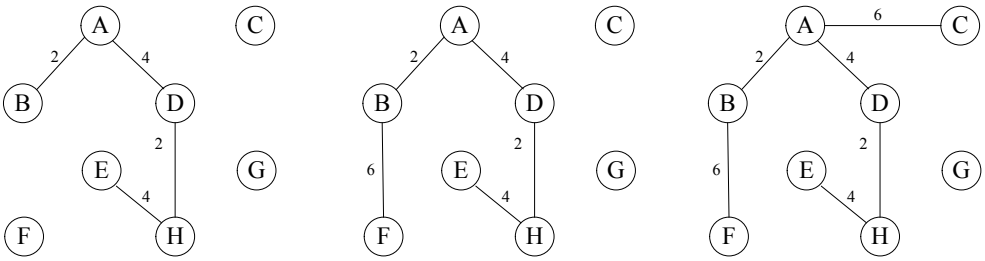


图 6.34 无向连通网

答案：





5. 如图 6.35 所示有向网，试完成：① 使用 Dijkstra 算法求顶点 A 到其它顶点的最短路径和长度；② 使用 Floyed 算法求每对顶点的最短路径和长度；③列出所有的拓扑排序。

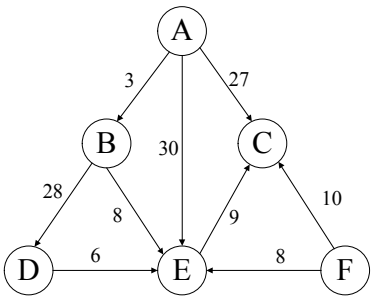


图 6.35 有向网

答案：

①

源点	终点	最短路径	长度
A	B	A->B	3
A	C	A->B->E	20
A	D	A->B->D	31
A	E	A->B->E	11
A	F	无	无穷大

②

	A	B	C	D	E	F

A		A->B （3）	A->B->E （20）	A->B->D （31）	A->B->E （11）	
B			B->E->C (17)	B->D (28)	B->E (8)	
C						
D		D->E->C (15)			D->E (6)	
E		E->C (9)				
F		F->C (10)			F->E (8)	

③ F A B D E C

A F B D E C

A B F D E C

A B D F E C

6. 试证明：①如果一个有向图的邻角矩阵是下三角形或上三角形，则一定无回路；②如果一个有向图无回路，则通过调整顶点在顶点数组中的次序，一定能使其邻接矩阵为下三角形或上三角形。

① 假定邻接矩阵为上三角形，则任意一条弧<i,j>,必满足 i<j；如果有回路，则存在顶点对序列<i1,i2><i2,i3>,….<in,i1>,满足 i<i2,i2<i3,….in<i1,产生矛盾，所以没有汇通，同理下三角。

② 如果没有回路，则可进行拓扑排序，按拓扑排序的次序在顶点数组中存放顶点，对于序号为 i 的顶点 v_i 和序号为 j 的顶点 v_j（i<j），按拓扑排序的定义，不可能出现 v_i 到 v_j 的弧，所以可能存在弧<i,j>，但不可能存在弧<j，i>，所以得到的邻接矩阵是上三角形。如果按逆拓扑排序的次序在顶点数组中存放顶点，则所以得到的邻接矩阵是下三角形。

7. 试证明 Floyed 算法的正确性。

证明：以任意两顶点路径上的最大序号顶点为依据，进行数学归纳法证明，按算法步骤，任意两个顶点间的最短路径都是可以计算出来的：

对于无中间顶点的最短路径，在由邻接矩阵初始化时就已得到了 M^{-1}，按算法流程，这类最短路径都是可以求出的；

对于某两个顶点的最短路径，如果是中间顶点最大序号为 0 的顶点 v_0 的最短路径，则按算法第一步即可求出 M^0，即完成了这类最短路径的计算，但对于中间顶点序号大于 0 的最短路径在 M^0 中尚未确定；

假定中间顶点最大序号为 k 的顶点 v_k 这类最短路径，按算法流程，在 M^k 中已经求出，但对于中间顶点序号由大于 k 的最短路径在 M^k 中尚未确定；

对于中间顶点最大序号为 k+1 的顶点 v_{k+1} 这类最短路径，不妨假定为 vi 和 vj 的最短路径属于此类，按算法步骤，在第 k 步后，需要考察边<v_i，v_{k+1}>和< v_{k+1}，v_j>的和，按归纳假设，<v_i，v_{k+1}>和< v_{k+1}，v_j>两对顶点间的中间顶点序号都不大于 k，所以两对顶点间的最短路径已经计算出来，将这两段最短路径相加，这样在第 k+1 步就能把中间顶点序号最大值为 k+1 的最短路径就计算出来。

综上所述，对于任意两个顶点，vi 和 vj，如果最短路径上最大的顶点序号为 k（0<=k<n），则在第 k 步就能计算该最短路径长度，其值在矩阵 M^k 中。所以 Floyed 算法是正确的，能求出所有的最短路径。

8. 如图 6.36 所示 AOE 网，① 计算各顶点事件的最早和最迟发生时间；② 计算各条弧表示的活动的最早和最迟开始时间。

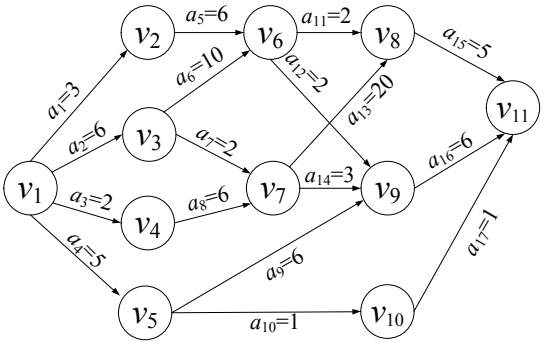


图 6.36 AOE 网

①

顶点	最早发生时间	最迟发生时间
V1	0	0
V2	3	19
V3	6	6
V4	2	2
V5	5	18

V6	16	25
V7	8	8
V8	28	28
V9	18	27
V10	6	32
V11	33	33

②

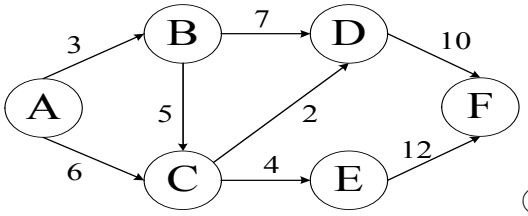
顶点	最早发生时间	最迟发生时间
a1	0	16
a2	**0**	**0**
a3	**0**	**0**
a4	0	13
a5	3	19
a6	6	15
a7	**6**	**6**
a8	**2**	**2**
a9	5	21
a10	5	31
a11	16	26
a12	16	25
a13	**8**	**8**
a14	8	24
a15	**28**	**28**
a16	18	27
a17	6	32

9. 已知一个有 6 个顶点的有向网 *N*，其顶点依次为 *A,B,C,D,E,F*，其邻接矩阵是一个上三角， 将其上三角（不包含对角线）按行序优先压缩存储到以下一维数组中，试完成：①画出该有向网；②画出邻接表；③写出所有的拓扑排序顶点序列；④求有向网 *N* 的关键路径。

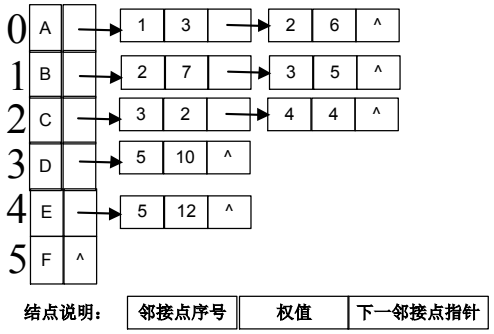
3	6	∞	∞	∞	5	7	∞	∞	2	4	∞	∞	10	12

答案：

①



②



③ A,B,C,D,E,F 和 A,B,C,E,D,F

④ <A,B> <B,C> <C,E> <E,F>

10. 试证明当无向连通网的任意一个环中包含边的权值均不相同时，其最小生成树是唯一的。

证明：假定无向连通网 N 存在一个生成树 T，则任意加上一条不属于 T，但属于 N 的边后，会形成一条回路。如果在此回路中，有权值相同的两条边，则去掉任意一条得到一棵最小生成树，这样就会出现多棵生成树，所以只要给定条件：所有回路中的各边互不相同，则一定存在唯一的最小生成树。这个条件是充分条件，但不是必要条件。

四、算法设计题

1. 以邻接矩阵作为无向图的存储结构，输入图的所有顶点和边的信息。试完成：① 创建图；② 增加、删除一个顶点；③ 增加、删除一条边。

```
#include "stdio.h"

#include "stdlib.h"

#include "string.h"

#define TRUE 1

#define FALSE 0

#define OK 1

#define ERROR 0
```

```
#define INFEASIBLE -1
#define OVERFLOW -2
#define MAX_VERTEX 20
typedef int status;
typedef int KeyType;
typedef enum {DG,DN,UDG,UDN} GraphKind;
typedef struct {
        KeyType    key;
        char others[20];
} VertexType; //顶点类型定义
typedef    struct {
            VertexType vexs[MAX_VERTEX];    // VertexType 为顶点类型，类似 ElemType
            int arcs[MAX_VERTEX][MAX_VERTEX];
            int vexnum,arcnum;
            GraphKind    kind;
} MGraph;
status GraphCreate(MGraph &G,VertexType V[],KeyType VR[][3],int vexnum,int
arcnum,GraphKind kind)
{
int i,j;
G.vexnum=vexnum; G.arcnum=arcnum; G.kind=kind;
for(i=0;i<G.vexnum; i++)        //初始化各顶点未访问状态
            G.vexs[i]=V[i];
if (kind==DN || kind==UDN)
      for(i=0;i<G.vexnum;i++)
            for(j=0;j<G.vexnum;j++)
                  G.arcs[i][j]=INFINITY;
else memset(&G.arcs[0][0],0,sizeof(G.arcs));
```

```
for(int k=0;k<G.arcnum; k++)
{
    for(i=0;i<G.vexnum;i++)     //查找顶点序号
        if (G.vexs[i].key==VR[k][0]) break;
    for(j=0;j<G.vexnum;j++)
        if (G.vexs[j].key==VR[k][1]) break;
    if (i>G.vexnum || j>G.vexnum )return ERROR;
    G.arcs[i][j]=VR[k][2];
    if (kind==UDG || kind==UDN)
        G.arcs[j][i]=VR[k][2];
}
return OK;
}
int InsertVex(MGraph &G,VertexType v)                    //插入顶点 v
{
    if (G.vexnum==MAX_VERTEX) return OVERFLOW;
    for(int i=0;i<G.vexnum;i++)                    //判断是否顶点重复
        if (G.vexs[i].key==v.key)
            return ERROR;
    G.vexs[G.vexnum++]=v;           //顶点加入顶点数组
    for(int i=0;i<G.vexnum;i++)       //对应邻接矩阵行列设置为 0
        G.arcs[G.vexnum-1][i]=G.arcs[i][G.vexnum-1]=0;
    return OK;
}
int DeleteVex(MGraph &G,KeyType u)
{
    int i,j,k,n=0;
    for(i=0;i<G.vexnum;i++)     //判断顶点是否存在
```

```
        if (G.vexs[i].key==u) break;
    if (i>=G.vexnum) return ERROR;
    for(j=i+1;j<G.vexnum;j++)
    {
        G.vexs[j-1]=G.vexs[j];   //将删除顶点后续顶点前移
        for(k=0;k<G.vexnum;k++)
        {   //邻接矩阵中，将删除顶点对应后续行列向前移动
            if (G.arcs[j-1][k]) n++;
            G.arcs[j-1][k]=G.arcs[j][k];
            G.arcs[k][j-1]=G.arcs[k][j];
        }
    }
    G.vexnum--;
    G.arcnum-=n;
    return OK;
}
InsertArc(MGraph &G,KeyType u,KeyType w)                //插入顶点 u 到顶点 w 的弧
{
    int i,j,k,n=0;
    for(i=0;i<G.vexnum;i++)   //判断顶点 u 是否存在，存在获取 u 的位序号 i
        if (G.vexs[i].key==u) break;
    for(j=0;j<G.vexnum;j++)   //判断顶点 w 是否存在，存在获取 w 的位序号 j
        if (G.vexs[j].key==w) break;
    if (i>=G.vexnum || i>=G.vexnum) return ERROR;
    if (!G.arcs[i][j])
    {
        G.arcs[i][j]=G.arcs[j][i]=1;
        G.arcnum++;
```

```
    }
    return OK;

}
DeleteArc(MGraph &G,KeyType u,KeyType w)
{
    int i,j,k,n=0;
    for(i=0;i<G.vexnum;i++)   //判断顶点 u 是否存在，存在获取 u 的位序号 i
        if (G.vexs[i].key==u) break;
    for(j=0;j<G.vexnum;j++)   //判断顶点 w 是否存在，存在获取 w 的位序号 j
        if (G.vexs[j].key==w) break;
    if (i>=G.vexnum || i>=G.vexnum) return ERROR;
    if (G.arcs[i][j])
    {
        G.arcs[i][j]=G.arcs[j][i]=0;
        G.arcnum--;
    }
    return OK;
}
```

2. 以邻接多重表作为无向图的存储结构，重做第（1）题。

```
typedef int KeyType;
typedef enum {DG,DN,UDG,UDN} GraphKind;
typedef struct {
    KeyType   key;
    char others[20];
} VertexType; //顶点类型定义
typedef struct EBox {              //表结点类型定义
    int ivex,jvex;                 //边两端顶点位置编号
```

```
        bool mark;
        struct EBox    *ilink,*jlink;  //下一个表结点指针
} EBox;
typedef struct VexNode{                    //头结点及其数组类型定义
        VertexType data;                //顶点信息
        EBox *firstarc;                    //指向第一条边结点
      } VexNode,AdjMList[MAX_VERTEX];
typedef    struct {    //邻接表的类型定义
      AdjMList vertices;                 //头结点数组
      int vexnum,arcnum;                //顶点数、弧数
    } AMLGraph;
status GraphCreate (AMLGraph &G,VertexType V[],KeyType VR[][3],    \
                                                    int vexnum,int arcnum)
{
      G.vexnum=vexnum; G.arcnum=arcnum;
      for(int i=0;i<G.vexnum; i++)                        //初始化各头结点数组
      {
            for(int j=0;j<i;j++)
                  if (G.vertices[j].data.key==V[i].key)
                        return ERROR;                              //关键字有重复
            G.vertices[i].data=V[i];
            G.vertices[i].firstarc=NULL;
      }
      for(int i,j,k=0;k<G.arcnum; k++)
      {
            for(i=0;i<G.vexnum;i++)                        //查找顶点序号
                  if (G.vertices[i].data.key==VR[k][0]) break;
            for(j=0;j<G.vexnum;j++)
```

```
            if (G.vertices[j].data.key==VR[k][1]) break;
        if (i>G.vexnum || j>G.vexnum) return ERROR;
        EBox *p=(EBox *) malloc(sizeof(EBox));
        p->ivex=i; p->jvex=j;
        p->ilink=G.vertices[i].firstarc;
        p->jlink=G.vertices[j].firstarc;
        G.vertices[i].firstarc=G.vertices[j].firstarc=p;
    }
    return OK;
}
status InsertVex(AMLGraph &G,VertexType v)                    //插入顶点 v
{
    G.vertices[G.vexnum].data=v;
    G.vertices[G.vexnum++].firstarc=NULL;
    return OK;
}
status DeleteVex(AMLGraph &G,KeyType u)
{
    int i,j;
    for(i=0;i<G.vexnum;i++)
        if (G.vertices[i].data.key==u)
            break;
    if (i>=G.vexnum) return ERROR;
    for(EBox *q,*p=G.vertices[i].firstarc; p; p=q)
    {
        j=p->ivex==i?p->jvex:p->ivex;
        q=p->ivex==i?p->ilink:p->jlink;   //q 指向下一条关联 i 的边
        EBox *p1=NULL,*p2=G.vertices[j].firstarc,*p3=p2->ivex==j?p2->ilink:p2->jlink;
```

```
        while (p2!=p)
        {
            p1=p2;
            p2=p3;
            p3=p3->ivex==j?p3->ilink:p3->jlink;
        }
        if (p1==NULL)
            G.vertices[j].firstarc=p3;
        else
            if (p1->ivex==j)
                p1->ilink=p3;
            else p1->jlink=p3;
        free(p);
        G.arcnum--;
    }
    for(j=i+1;j<G.vexnum;j++)
    {
        EBox *q, *p=G.vertices[j].firstarc;
        while (p)
        {
            q=p->ivex==j?p->ilink:p->jlink;
            if (p->ivex==j) p->ivex--;
            else p->jvex--;
            p=q;
        }
    }
    for(j=i+1;j<G.vexnum;j++)
        G.vertices[j-1]=G.vertices[j];
```

```
    G.vexnum--;
    return OK;
}
status    InsertArc(AMLGraph &G,KeyType u,KeyType w)                    //插入顶点 u 到顶点 w 的弧
{
    int i,j;
    for(i=0;i<G.vexnum;i++)
        if (G.vertices[i].data.key==u)
            break;
    for(j=0;j<G.vexnum;j++)
        if (G.vertices[j].data.key==w)
            break;
    if (i>=G.vexnum || j>=G.vexnum) return ERROR;
    EBox *p=(EBox *) malloc(sizeof(EBox));
    p->ivex=i; p->jvex=j;
    p->ilink=G.vertices[i].firstarc;
    p->jlink=G.vertices[j].firstarc;
    G.vertices[i].firstarc=G.vertices[j].firstarc=p;
    G.arcnum++;
    return OK;
}
status DeleteArc(AMLGraph &G,KeyType u,KeyType w)
{
    int i,j;
    for(i=0;i<G.vexnum;i++)
        if (G.vertices[i].data.key==u)
            break;
```

```
for(j=0;j<G.vexnum;j++)
    if (G.vertices[j].data.key==w)
        break;
if (i>=G.vexnum || j>=G.vexnum) return ERROR;
EBox *p1=NULL,*p2=G.vertices[i].firstarc,*p3=p2->ivex==i?p2->ilink:p2->jlink;
while (p2->ivex!=j && p2->jvex!=j)
{
    p1=p2;
    p2=p3;
    p3=p3->ivex==i?p3->ilink:p3->jlink;
}
if (p1==NULL)
    G.vertices[i].firstarc=p3;
else
    if (p1->ivex==i)
        p1->ilink=p3;
    else p1->jlink=p3;
p1=NULL,p2=G.vertices[j].firstarc,p3=p2->ivex==j?p2->ilink:p2->jlink;
while (p2->ivex!=i && p2->jvex!=i)
{
    p1=p2;
    p2=p3;
    p3=p3->ivex==j?p3->ilink:p3->jlink;
}
if (p1==NULL)
    G.vertices[j].firstarc=p3;
else
    if (p1->ivex==j)
```

```
            p1->ilink=p3;
        else p1->jlink=p3;
    G.arcnum--;
    return OK;
}
```

3. 以十字链表作为有向网的存储结构，输入图的所有顶点和弧（包含权值）的信息。试完成：① 创建图；② 增加、删除一个顶点；③ 增加、删除一条弧。

```
status GraphCreate (OLGraph &G,VertexType V[],KeyType VR[][3],  \
                                                int vexnum,int arcnum)
{
    G.vexnum=vexnum; G.arcnum=arcnum;
    for(int i=0;i<G.vexnum; i++)                    //初始化各头结点数组
    {
        for(int j=0;j<i;j++)
            if (G.xlist[j].data.key==V[i].key)
                return ERROR;                       //关键字有重复
        G.xlist[i].data=V[i];
        G.xlist[i].firstin=G.xlist[i].firstout=NULL;
    }
    for(int i,j,k=0;k<G.arcnum; k++)
    {
        for(i=0;i<G.vexnum;i++)                     //查找顶点序号
            if (G.xlist[i].data.key==VR[k][0]) break;
        for(j=0;j<G.vexnum;j++)
            if (G.xlist[j].data.key==VR[k][1]) break;
        if (i>G.vexnum || j>G.vexnum) return ERROR;
        ArcBox *p=(ArcBox *) malloc(sizeof(ArcBox));
        p->tailvex=i; p->headvex=j;
```

```
            ArcBox *p1=NULL,*p2=G.xlist[i].firstout;
            while (p2 && j>p2->headvex)
                p1=p2,p2=p2->tlink;
            if (p1==NULL)
                G.xlist[i].firstout=p;
            else p1->tlink=p;
            p->tlink=p2;
            p1=NULL,p2=G.xlist[j].firstin;
            while (p2 && i>p2->tailvex)
                p1=p2,p2=p2->hlink;
            if (p1==NULL)
                G.xlist[j].firstin=p;
            else p1->hlink=p;
            p->hlink=p2;
        }
        return OK;
}
status InsertVex(OLGraph &G,VertexType v)                    //插入顶点 v
{
        G.xlist[G.vexnum].data=v;
        G.xlist[G.vexnum].firstin=G.xlist[G.vexnum].firstout=NULL;
        G.vexnum++;
        return OK;
}
status DeleteVex(OLGraph &G,KeyType u)
{
        int i,j;
        for(i=0;i<G.vexnum;i++)
```

```
        if (G.xlist[i].data.key==u)
            break;
    if (i>=G.vexnum) return ERROR;
    for(ArcBox *q,*p=G.xlist[i].firstout; p; p=q)
    {   //删除以 u 为狐尾的弧
        q=p->tlink;                 //q 指向下一条以 u 为狐尾的弧
        ArcBox *p1=NULL,*p2=G.xlist[p->headvex].firstin;
        while (p2->tailvex!=i)
        {
            p1=p2;
            p2=p2->hlink;
        }
        if (p1==NULL)
             G.xlist[p->headvex].firstin=p2->hlink;
        else p1->hlink=p2->hlink;
        free(p);
        G.arcnum--;
    }
    for(ArcBox *q,*p=G.xlist[i].firstin; p; p=q)
    {   //删除以 u 为狐头的弧
        q=p->hlink;                 //q 指向下一条以 u 为狐尾的弧
        ArcBox *p1=NULL,*p2=G.xlist[p->tailvex].firstout;
        while (p2->headvex!=i)
        {
            p1=p2;
            p2=p2->tlink;
        }
        if (p1==NULL)
```

```
                G.xlist[p->tailvex].firstout=p2->tlink;
            else p1->tlink=p2->tlink;
            free(p);
            G.arcnum--;
        }
        for(j=i+1;j<G.vexnum;j++)
            G.xlist[j-1]=G.xlist[j];
        G.vexnum--;
        for(j=0;j<G.vexnum;j++)
        {
            ArcBox *p=G.xlist[j].firstout;
            while (p)
            {    //序号大于 i 的减一
                if (p->headvex>i) p->headvex--;
                if (p->tailvex>i) p->tailvex--;
                p=p->tlink;
            }
        }
        return OK;
}
status InsertArc(OLGraph &G,KeyType u,KeyType w)                //插入顶点 u 到顶点 w 的弧
{
        int i,j;
        for(i=0;i<G.vexnum;i++)
            if (G.xlist[i].data.key==u)
                break;
        for(j=0;j<G.vexnum;j++)
            if (G.xlist[j].data.key==w)
```

```
            break;
    if (i>=G.vexnum || j>=G.vexnum) return ERROR;
    ArcBox *p=(ArcBox *) malloc(sizeof(ArcBox));
    p->tailvex=i; p->headvex=j;
    ArcBox *p1=NULL,*p2=G.xlist[i].firstout;
    while (p2 && j>p2->headvex)
        p1=p2,p2=p2->tlink;
    if (p1==NULL)
        G.xlist[i].firstout=p;
    else p1->tlink=p;
    p->tlink=p2;
    p1=NULL,p2=G.xlist[j].firstin;
    while (p2 && i>p2->tailvex)
        p1=p2,p2=p2->hlink;
    if (p1==NULL)
        G.xlist[j].firstin=p;
    else p1->hlink=p;
    p->hlink=p2;
    G.arcnum++;
    return OK;

}
status DeleteArc(OLGraph &G,KeyType u,KeyType w)
{
    int i,j;
    for(i=0;i<G.vexnum;i++)
        if (G.xlist[i].data.key==u)
            break;
```

```
    for(j=0;j<G.vexnum;j++)
        if (G.xlist[j].data.key==w)
            break;
    if (i>=G.vexnum || j>=G.vexnum) return ERROR;
    ArcBox *p1=NULL,*p2=G.xlist[i].firstout;
    while (p2->headvex!=j)
    {
        p1=p2;
        p2=p2->tlink;
    }
    if (p1==NULL)
          G.xlist[i].firstout=p2->tlink;
    else p1->tlink=p2->tlink;
    p1=NULL,p2=G.xlist[j].firstin;
    while (p2->tailvex!=i)
    {
        p1=p2;
        p2=p2->hlink;
    }
    if (p1==NULL)
          G.xlist[j].firstin=p2->hlink;
    else p1->hlink=p2->hlink;
    free(p2);
    G.arcnum--;
    return OK;
}
```

4. 以邻接矩阵作为无向图的存储结构，试设计算法实现，输入起始和终点，求起始到终点的最短路径和长度。

算法思想：利用图的广度优先遍历算法，从起始点开始做深度优先遍历，当遇到终点时结束，输出对应最短路径，一次广度优先遍历结束，没访问到终点，表示无路径。为了能输出对应最短路径的顶点序列，在遍历中维护了一个栈，记录每次访问时，对应边的起始起始、终止的 2 个顶点序号，一旦遇到终点，分析该栈即可得到最短路径顶点序列。

```
void BFSTraverse(MGraph G,int begin,int end)
{
    int Q[100],f,r,flag=1,u,length=0;
    struct {int v[30][2],top;}s;//栈 s 存放访问时经过的边顶点序号对
    bool visited[MAX_VERTEX];
    for(u=0; u<G.vexnum; u++)  visited[u]=false;
    f=r=0;          //队列 Q 初始化
    s.top=0;
    visited[begin]=true;
    Q[r++]=begin;      //v 入队列 Q;
    while(f!=r && flag)
    {
        u=Q[f++];   //DeQueue(Q,u);
        for(int w=FirstAdjVex(G,u);w>=0; w=NextAdjVex(G,u,w))
            if (!visited[w])
            {
                visited[w]=true;
                Q[r++]=w;                    //EnQueue(Q,w);}
                s.v[s.top][0]=u;          //边的起点
                s.v[s.top++][1]=w;        //边的终点
                if (w==end)
                {
                    flag=0;
                    break;
                }
```

```
            }
        }
        if (flag)
        {
            printf("两个顶点不可达\n");
            return;
        }
        u=s.v[--s.top][0];              //u 为最短路径终点的前驱顶点序号
        printf("最短路径:(%d,%s)",G.vexs[end].key,G.vexs[end].others);
        length=1;
        while (s.top)
        {
            if (u==s.v[s.top-1][1])
            {
                printf("<--(%d,%s)",G.vexs[u].key,G.vexs[u].others);
                u=s.v[s.top-1][0];   //修改 u 为最短路径为前驱顶点序号
                length++;
            }
            s.top--;
        }
        printf("<--(%d,%s)\n 路径长度=%d\n",G.vexs[begin].key,G.vexs[begin].others,length);
}
```

5. 假定以邻接矩阵作为无向图的存储结构，试求出该无向图的全部连通分量并输出各连通分量的顶点集合与边集合。

算法思想：借助于遍历算法（深度和广度优先遍历都可以），每次遍历访问到的顶点集合与其该集合中的各顶点间关系构成一个连通分量。

```
void BFSTraverse(MGraph G)
{
    int Q[100],f,r,nums=0;
```

```
bool visited[MAX_VERTEX];
struct {int v[30],length;}SList;//顺序表存放连通分量的顶点集合
for(int v=0; v<G.vexnum; v++)     visited[v]=false;
f=r=0;
for(int v=0;v<G.vexnum; v++)        //按顶点位置序号依次选择顶点
      if (!visited[v])                    //遇到未访问过的顶点开始遍历
      {
            visited[v]=true; Q[r++]=v;//EnQueue(Q,v);
            SList.length=0;
            SList.v[SList.length++]=v;
            while(f!=r)
            {
                  int u=Q[f++];    //DeQueue(Q,u);
                  for(int w=FirstAdjVex(G,u);w>=0; w=NextAdjVex(G,u,w))
                        if (!visited[w])
                        {
                              visited[w]=true;
                              SList.v[SList.length++]=w;
                              printf("%s ",G.vexs[w].others);
                              Q[r++]=w;//EnQueue(Q,w);}
                        }
             }
             nums++;
             MGraph G1;
             G1.kind=G.kind;G1.vexnum=SList.length;G1.arcnum=0;
             for(int i=0;i<SList.length;i++)
             {
                   G1.vexs[i]=G.vexs[SList.v[i]];
```

```
                for(int j=i;j<SList.length;j++)
                {
                    int i1=SList.v[i],j1=SList.v[j];
                    G1.arcs[i][j]=G.arcs[i1][j1];
                    G1.arcs[i][j]=G.arcs[i1][j1];
                    G1.arcnum++;
                }
            }
            printf("\n\n 第%d 个连通分量：\n 顶点序列：",nums);
            for(int i=0;i<G1.vexnum;i++)
                printf("(%d %s) ",G1.vexs[i].key,G1.vexs[i].others);
            printf("\n 关系序列：");
            for(int i=0;i<G1.vexnum;i++)
                for(int j=i+1;j<G1.vexnum;j++)
                    if (G1.arcs[i][j])
                        printf("    (%d %s,%d %s)",G1.vexs[i].key,G1.vexs[i].others,\
                                                    G1.vexs[j].key,G1.vexs[j].others);
        }
}
```

6. 假定以邻接表作为无向图的存储结构，试求出顶点 v 出发，经过一个长度为 k 的简单路径到达的顶点集合。

算法思想：利用图的深度优先遍历算法，增加一个长度的参数，以开始顶点出发，调用 DFS 进行一次深度优先遍历，每次递归调用时长度减一，当长度为 0 时，，并且该顶点不在结果集合中时，将当前访问的顶点显示出来。

```
void DFSTraverse(ALGraph G,int begin,int length)
{
    bool visited[MAX_VERTEX];
    bool displayed[MAX_VERTEX];
    for(int v=0;v<G.vexnum; v++)        //初始化各顶点未访问状态
```

```
        visited[v]=displayed[v]=false;
    DFS(G, begin, visited,displayed, length);
}
void DFS(ALGraph G,int v, bool visited[],bool displayed[],int length)
{
    visited[v]=true;                    //标注访问过的标记
    if (!length && !displayed[v])
    {
        printf("%d %s\n",G.vertices[v].data.key,G.vertices[v].data.others);
        displayed[v]=true;
        return;
    }
    for(ArcNode *p=G.vertices[v].firstarc;p; p=p->nextarc)
         if (!visited[p->adjvex])          //处理所有未访问的邻接顶点
         {
             DFS(G,p->adjvex, visited,displayed,length-1);
             visited[p->adjvex]=false;
         }
}
```

7. 假定以邻接表作为有向图的存储结构，给定一个顶点 *v*，判断是否有含顶点 *v* 的简单回路，是则输出一条包含该顶点的回路。

算法思想：从顶点 v 开始进行一次深度优先遍历，调用递归函数 DFS，每次访问到一个顶点，将顶点进栈，当遍历中，如果当前访问到的结点邻接顶点有 v，则找到回路，根据栈进行输出回路结点。

```
typedef struct STACK{int v[30],top;}STACK;
status DFS(ALGraph G,int v, bool visited[],STACK &s);
void DFSTraverse(ALGraph G,int v)
{
    bool visited[MAX_VERTEX];
```

```
    STACK s; s.v[0]=v, s. top=1;
    for(int v=0;v<G.vexnum; v++)
        visited[v]=false;
    if (DFS(G, v,visited,s))
        for(int i=0;i<s.top;i++)
            printf("-(%d,%s)",G.vertices[s.v[i]].data.key,G.vertices[s.v[i]].data.others);
    else     printf("无回路");
}
status DFS( ALGraph G, int v, bool visited[], STACK &s)
{
    visited[v]=true;
    for(ArcNode *p=G.vertices[v].firstarc;    p;    p=p->nextarc)
    {
        s.v[s.top++]=p->adjvex;
        if (p->adjvex==s.v[0]) return true;
          if (!visited[p->adjvex])
          {
            if (DFS(G,p->adjvex, visited,s)) return true;
            visited[p->adjvex]=false;
          }
        s.top--;
    }
    return false;
}
```

第七章

一、选择题

1．【2019 统考真题】选择一个排序算法时，除考虑算法的时空效率外，还需要考虑的是____。

Ⅰ．数据的规模　　Ⅱ．数据的存储方式

Ⅲ．算法的稳定性　　Ⅳ．数据的初始状态

A．仅Ⅲ　　B．仅Ⅰ、Ⅱ　　C．仅Ⅱ、Ⅲ、Ⅳ　　D．Ⅰ、Ⅱ、Ⅲ、Ⅳ

答案：D

2．下列排序算法中平均复杂度为$O(n\log n)$且稳定的是_____。

A．插入排序　　B．归并排序　　C．堆排序　　D．快速排序

答案：B

解释：插入排序平均复杂度不为 $O(nlogn)$，堆排序和快速排序不稳定。

3．【2019 统考真题】排序过程中，对尚未确定最终位置的所有元素进行一遍处理称为“趟”。下列序列中可能是快速排序算法第二趟结果的是_____。

A．5,2,16,12,28,60,32,72　　B．2,16,5,28,12,60,32,72

C．2,12,16,5,28,32,72,60　　D．5,2,12,28,16,32,72,60

答案：ABC

解释：每一趟快排，至少确认一个元素的最终位置。序列最终位置是：2,5,12,16,28,32,60,72，而选项中正确的位置有：

A．5,2,16,12,**28**,60,32,**72**　　B．**2**,16,5,28,12,60,32,**72**

C．**2**,12,16,5,**28**,**32**,72,60　　D．5,2,**12**,28,16,**32**,72,60

第一趟排序，确定一个元素位置

第二趟排序，有确定一个或者两个元素位置

1. 当第一趟元素确认的位置为最左或最右时，第二趟排序只能确认一个位置（A、B选项情况）
2. 当第一趟元素确认不是最左或最右时，第二趟能确认2个位置（C选项情况）

D选项：第一趟确认的元素不管是12还是32，第二次都应该能确认2个位置，两趟下来共能确认3个元素，而D选项不符，所以错误。

4. 【2009 统考真题】已知关键字序列5,8,12,19,28,20,15,22 是小根堆（最小堆），插入关键字3，调整后得到的小根堆是_____。

A. 3,5,12,8,28,20,15,22,19　　B. 3,5,12,19,20,15,22,8,28

C. 3,8,12,5,20,15,22,28,19　　D. 3,12,5,8,28,20,15,22,19

答案：A

解释：插入3之后（5,8,12,19,28,20,15,22,3）的置换顺序3--19，3--8，3--5。

5.【2009 统考真题】若数据元素序列11,12,13,7,8,9,23,4,5 是采用下列排序算法之一得到第二趟排序后的结果，则该排序算法只能是_____。

A. 冒泡排序　　B. 插入排序　　C. 选择排序　　D. 2 路归并排序

答案：B

解释：对于冒泡排序和选择排序，每一趟都能确定一个元素的最终位置，而题目中，前 2 个元素和后 2 个元素均不是最小或最大的 2 个元素并按序排列。选项 D 中的 2 路归并排序，第一趟排序结束都可以得到若干个有序子序列，而此时的序列中并没有两两元素有序排列。插入排序在每趟排序后能确定前面的若干元素是有序的，而此时第二趟排序后，序列的前三个元素是有序的，符合其特征。

6.【2010 统考真题】对一组数据(2,12,16,88,5,10)进行排序，若前3 趟排序结果如下：第一趟排序结果为2,12,16,5,10,88，第二趟排序结果为2,12,5,10,16,88，第三趟排序结果为2,5,10,12,16,88，则采用的排序算法可能是_____。

A. 冒泡排序　　B. 希尔排序　　C. 归并排序　　D. 基数排序

答案：A

解释：冒泡排序每一趟都确定一个元素的最终位置，前三趟依次确定了序列倒数三位的元素。

7.【2012 统考真题】下列排序算法中，每一趟排序结束都至少能够确定一个元素最终位置的算法是_____。

Ⅰ. 简单选择排序　Ⅱ. 希尔排序　Ⅲ. 快速排序　Ⅳ. 堆排序　Ⅴ. 2 路归并排序

A. 仅Ⅰ、Ⅲ、Ⅳ　　B. 仅Ⅰ、Ⅲ、Ⅴ　　C. 仅Ⅱ、Ⅲ、Ⅳ　　D. 仅Ⅲ、Ⅳ、Ⅴ

答案：A

解释：每趟排序至少能确定一个元素最终位置的排序算法主要是两类：选择排序算法（简单选择、堆排序）和交换排序算法（冒泡排序、快速排序）。

8.【2012 统考真题】对一个待排序序列分别进行折半插入排序和直接插入排序，两者之间可能的不同之处是_____。

A．排序的总趟数　　B．元素的移动次数

C．使用辅助空间的数量　　D．元素之间的比较次数

答案：D

解释：折半插入排序基本思想和直接插入排序一样，区别在于寻找插入位置的方法不同，折半插入排序采用折半查找法来寻找插入位置。

9.【2013 统考真题】对给定的关键字序列110,119,007,911,114,120,122 进行基数排序，则第二趟分配、收集后得到的关键字序列是_____。

A．007,110,119,114,911,120,122　　B．007,110,119,114,911,122,120

C．007,110,911,114,119,120,122　　D．110,120,911,122,114,007,119

答案：C

解释：第一趟按照个位排，得到：110,120,911,122,114,007,119。第二趟按照十位排，得到：

007,110,911,114,119,120,122。

10.【2013 统考真题】用希尔排序算法对一个数据序列进行排序时，若第一趟排序结果为9,1,4,13,7,8,20,23,15，则该趟排序采用的增量（间隔）可能是_____。

A．2　　B．3　　C．4　　D．5

答案：B

解释：假如增量为2，整个序列分为2组，[9,4,7,20,15]和[1,13,8,23]，可以看出这两组元素不是有序的，因此增量不为2。同理代入其他选项，得到答案为B。

11．在下列排序算法中，时间复杂度与待排序序列初始状态无关的算法是_____

A．插入排序　　B．堆排序　　C．冒泡排序　　D．归并排序

答案：BD

解释：堆排序和归并排序在最好情况、最坏情况以及平均情况下的时间复杂度均为$O(nlogn)$。

12.【2013 统考真题】下列选项中，不可能是快速排序算法第二趟排序结果的是_____。

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

答案：C

解释：每一趟快排，至少确认一个元素的最终位置，两趟快排至少可以确认2个元素的最终位置。序列最终位置是：2,3,4,5,6,7,9，而选项中正确的位置有：

A．**2**,**3**,5,4,**6**,**7**,**9**　　B．**2**,7,5,6,4,3,**9**　　C．3,2,5,4,7,6,**9**　　D．4,2,3,**5**,7,6,**9**

因此，C选项不可能是第二趟快排结果。

13.【2015 统考真题】下列排序算法中，元素的移动次数与关键字的初始排列次序无关的是_____。

A．直接插入排序　B．冒泡排序　C．基数排序　D．快速排序

答案：C

14.【2015 统考真题】已知小根堆为8,15,10,21,34,16,12，删除关键字 8 之后需重建堆。在此过程中，关键字之间的比较次数是_____。

A．1　B．2　C．3　D．4

答案：C

解释：删除关键字8，需要将末尾的叶子结点12置换到根结点。（1）比较根结点12的两个孩子15和10的大小；（2）比较根结点12和它更小的孩子10的大小（将10和12进行交换）；（3）比较交换后结点12和它的孩子16的大小（不发生交换，完成堆重建）。

15．堆是一种有用的数据结构，在以下排序序列中小根堆是_____。

A．16,72,31,23,94,53　　B．94,53,31,72,16,53

C．16,53,23,94,31,72　　D．16,31,23,94,53,72

答案：D

16.【2015 统考真题】希尔排序算法的组内排序采用的是_____。

A．直接插入排序　B．折半插入排序　C．快速排序　D．归并排序

答案：A

17．对数组存储线性表(16,15,32,11,6,30)用快速排序算法进行由小到大排序，若排序下标范围为0～5，选择元素16 作为支点，调用一趟快速排序算法后，元素16 在数组中的下标位置为_____。

A．1　B．2　C．3　D．4

答案：C

解释：一趟快速排序算法后，元素16左边有3个小于它的元素（分别对应数组下标0,1,2），所以元素16在数组中的下标位置为3。

18.【2016 统考真题】对10 TB 的数据文件进行排序，应使用的方法是_____。

A．希尔排序　B．堆排序　C．快速排序　D．归并排序

答案：D

解释：对于10TB的海量数据，数据不可能一次全部载入内存，传统的排序方法就不适用了，需要用到外排序的方法。

19.【2017 统考真题】在内部排序时，若选择归并排序而没有选择插入排序，则可能的理由是_____。

Ⅰ．归并排序的程序代码更短　Ⅱ．归并排序的占用空间更少　Ⅲ．归并排序的运行效率更高

A．仅Ⅱ　　B．仅Ⅲ　　C．仅Ⅰ、Ⅱ　　D．仅Ⅰ、Ⅲ

答案：B

解释：对并排序代码量大。归并排序的最坏情况和平均情况时间复杂度优于插入排序。空间复杂度劣于插入排序。

20．用某种排序算法对关键字序列(25,84,21,47,15,27,68,35,20)进行排序时，序列的变化情况如下：

15,20,21,25,47,27,68,35,84

15,20,21,25,35,27,47,68,84

15,20,21,25,27,35,47,68,84

则采用的排序算法是哪种？_____

A．希尔排序　　B．选择排序　　C．快速排序　　D．归并排序

答案：C

解释：（1）第一步以元素25为基准，小于25的放在25的左边，大于25的放在25的右边，得到20,15,21,25,47,27,68,35,84；

（2）第二步在25的两边分别进行快速排序，左边以20为基准，右边以47为基准，得到15,20,21,25,35,27,47,68,84；

（3）第三步得到15,20,21,25,27,35,47,68,84。

21.【2017 统考真题】下列排序算法中，若将顺序存储更换为链式存储，则算法的时间效率会降低的是_____。

Ⅰ．插入排序　　Ⅱ．选择排序　　Ⅲ．冒泡排序　　Ⅳ．希尔排序
Ⅴ．堆排序

A．仅Ⅰ、Ⅱ　　B．仅Ⅱ、Ⅲ　　C．仅Ⅲ、Ⅳ　　D．仅Ⅳ、Ⅴ

答案：D

解释：插入排序、选择排序、起泡排序原本时间复杂度是$O(n^2)$，更换为链式存储后的时间复杂度还是$O(n^2)$。希尔排序和堆排序都利用了顺序存储的随机访问特性，而链式存储不支持这种性质，所以时间复杂度会增加，因此选D。

22.【2018 统考真题】对初始数据序列(8, 3, 9, 11, 2, 1, 4, 7, 5, 10, 6)进行希尔排序，若第一趟排序结果为(1, 3, 7, 5, 2, 6, 4, 9, 11, 10, 8)，第二趟排序结果为(1, 2, 6, 4, 3, 7, 5, 8, 11, 10, 9)，则两趟排序采用的增量（间隔）依次是_____。

A．3, 1　　B．3,2　　C．5,2　　D．5,3

答案：D

解释：先看第一趟，1从最开始的5号位移动到0号位，说明第一趟增量是5，直接排除A,B。再看第二趟，2从第一趟排序结果的第4位移动到1号位，说明第二趟增量是3。

23．一个元素序列为{46,79,56,38,40,84}，采用快速排序（以第一个元素为轴点）得到的结果为_____。

A．38,46,79,56,40,84　　B．38,79,56,46,40,84

C．40,38,46,79,56,84　　D．38,46,56,79,40,84

答案：C

24.【2018 统考真题】在将数据序列(6, 1, 5, 9, 8, 4, 7) 建成大根堆时，正确的序列变化过程是_____。

A．6,1,7,9,8,4,5 → 6,9,7,1,8,4,5 → 9,6,7,1,8,4,5 → 9,8,7,1,6,4,5

B．6,9,5,1,8,4,7 → 6,9,7,1,8,4,5 → 9,6,7,1,8,4,5 → 9,8,7,1,6,4,5

C．6,9,5,1,8,4,7 → 9,6,5,1,8,4,7 → 9,6,7,1,8,4,5 → 9,8,7,1,6,4,5

D．6,1,7,9,8,4,5 → 7,1,6,9,8,4,5 → 7,9,6,1,8,4,5 → 9,7,6,1,8,4,5 → 9,8,6,1,7,4,5

答案：A

25．下列排序算法中，平均时间复杂度为$O(n^2)$的排序算法有哪些？_____

A．归并排序　　B．插入排序　　C．冒泡排序　　D．快速排序

答案：BC

解释：归并排序和快速排序的平均时间复杂度均为*O(nlogn)*。

二、问答题

1．试证明 2 路归并排序算法是稳定的排序算法。

解：在归并排序中，如果两个元素值相等，那么归并的时候优先选择左边的元素，这就保证了相同元素的相对位置不变，即使在不同的子数组中出现了相同元素也不会打乱它们的顺序。因此，2路归并排序算法是一种稳定的排序算法。

2．有如下 12 个整数：23,37,7,79,29,43,73,19,31,61,23,47。堆排序中，通过以下语句：

```
    for(int i=H->length/2; i>=1; i--)

        HeapAdjust(H, i, H->length);
```

调用HeapAdjust()建立初始的堆。

（1）画出每次调用HeapAdjust()之后形成的堆结构图。

（2）试分析完成建立初始堆所需的时间。

（3）在调用HeapAdjust()的 for 循环中，循环变量为什么是由 length/2 到1 递减，而不是由1到 length/2 递增呢？

解：

（1）假设此堆为最小堆，编号从1开始，则每次调用HeapAdjust()后的结构图如下所示：