1.
(1)

函数 g

Old ebp
%gs(14)
a[9]
a[8]
a[7]
a[6]
a[5]
a[4]
a[3]
a[2]
a[1]
a[0]

函数 f

Old ebp
%gs(14)
b[1]
b[0]

(2) 输入 161220049，运行程序后的输出：

```
Starting program: /home/hyc/workspace/lab04/161220049/array_init
input student id:
161220049
9 -48
```

用 gdb 调试后发现 b[0]与 b[1]分别对应这两个输出：

```
(gdb) b f
Breakpoint 2 at 0x8048627: file array_init.c, line 28.
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/hyc/workspace/lab04/161220049/array_init
input student id:
161220049

Breakpoint 2, f () at array_init.c:28
28      void f(){
(gdb) si
0x0804862d      28      void f(){
(gdb)
0x08048630      28      void f(){
(gdb)
30          print(b);
(gdb) display b[0]
1: b[0] = 9
(gdb) display b[1]
2: b[1] = -48
```

具体原因：

首先执行 g()函数，声明 a[]数组后执行 init(a)函数，输入九位学号后（不溢出），得到数组 a，储存着学号的每一位(int)，而后执行 f()函数，声明 b[]数组后执行 print(b)函数，由于没能初始化 b[]数组，它的值显示为输入区最后最后两位（’9’和’\0’）它们与’0’相减的结果分别是 9 和-48，因此输出这两个数。

未初始化局部变量的危害：未初始化的局部变量是随机而未定的，它的值会影响程序目的的达成。

2.

(1)A[i][j][k]地址的表达式为 $addr(A)+4(i*S*T+j*T+k)$

(2)

	%eax	%ecx	%edx
3	7	7	6
4	7	52	6
5	7	52	7
6	14	52	7
7	14	52	14
8	112	52	14
9	98	52	14
10	98	52	9694
11	98	52	9562

12	7	52	9562
13	7	52	9569
14	161220049	52	9569
15	161220049	52	9569
16	378560	52	9569

(3)0x5c6c0=378560，R*S*T=378560/4=94640；
通过追踪汇编代码，
又 182i+14j+k=S*T*i+T*J+k，因此 S*T=182，T=14，因此 S=13，T=14，R=520

3.
在此对题目给出的汇编代码进行注释，以此作为补全 C 代码的依据。

```
00000000 <recursion>:
        0:      55                      push   %ebp   //保存旧的帧指针
        1:      89 e5                   mov    %esp, %ebp    //设置栈指针
        3:      53                      push   %ebx   //讲寄存器%ebx 压入栈中
        4:      83 ec 04                sub    $0x4, %esp     //栈顶减小 4，腾出内存
        7:      83 7d 08 02             cmpl   $0x2, 0x8(%ebp)       //x 保存在 8(%ebp)中，作差 x-2
        b:      7f 07                   jg     14 <recursion+0x14>           //若 x>2，跳转至 14
        d:      b8 01 00 00 00          mov    $0x1, %eax     //给寄存器%eax 初始化为 1
       12:      eb 28                   jmp    3c <recursion+0x3c>           //直接跳转至 3c
       14:      8b 45 08                mov    0x8(%ebp), %eax       //讲 x 加载到寄存器%eax
       17:      83 e8 01                sub    $0x1, %eax     //%eax 的值减 1，为 x-1
       1a:      83 ec 0c                sub    $0xc, %esp     //栈顶减小 12，腾出内存
       1d:      50                      push   %eax   //将寄存器%eax 压入栈中
       1e:      e8 fc ff ff ff          call   1f <recursion+0x1f>           //启动递归，参数为 x-1
       23:      83 c4 10                add    $0x10, %esp   //%esp 增加 16
       26:      89 c3                   mov    %eax, %ebx    //将返回值保存到%ebx
       28:      8b 45 08                mov    0x8(%ebp), %eax       //将 x 加载到%eax
       2b:      83 e8 02                sub    $0x2, %eax     //%eax 变成 x-2
       2e:      83 ec 0c                sub    $0xc, %esp     //栈顶减小 12，腾出内存
       31:      50                      push   %eax   //将%eax 压入栈中
       32:      e8 fc ff ff ff          call   33 <recursion+0x33>           启动递归，参数为 x-2
       37:      83 c4 10                add    $0x10, %esp   //%esp 增加 16
       3a:      01 d8                   add    %ebx, %eax    //将 26 中的%ebx 加到返回值
       3c:      8b 5d fc                mov    -0x4(%ebp), %ebx
       3f:      c9                      leave
       40:      c3                      ret
```

因此 C 代码为：

```
int recursion(int x) {
        if (x > 2)
                return recursion(x-1)+recursion(x-2);
        else
                return 1;
}
```

发现是斐波那契数列。

4.

(1)(注意 union 即可)

0	4	0	4	8	12 16
e1.p	e1.x	y[0]	y[1]	y[2]	next

偏移量:

e1.p 0
e1.x 4
y 0
y[0] 0
y[1] 4
y[2] 8
next 12

(2)将汇编代码进行注释如下:

```
00000000 <proc>:
       0:   55              push   %ebp
       1:   89 e5           mov    %esp, %ebp
       3:   8b 45 08        mov    0x8(%ebp), %eax     //将 up 加载到%eax
       6:   8b 40 0c        mov    0xc(%eax), %eax     //将 up->y[2]加载到%eax
       9:   8b 55 08        mov    0x8(%ebp), %edx     //将 up 加载到%edx
       c:   8b 12           mov    (%edx), %edx        //将 up→e1.p 加载到%edx
       e:   8b 0a           mov    (%edx), %ecx        //将*(up→e1.p)加载到%ecx
      10:   8b 55 08        mov    0x8(%ebp), %edx     //将 up 加载到%edx
      13:   8b 52 08        mov    0x8(%edx), %edx     //将 up->y[2]加载到%edx
      16:   01 ca           add    %ecx, %edx          //%edx 的值变为 up→y[2]+*(up->e1.p)
      18:   89 10           mov    %edx, (%eax)        //p→next → y[0]=up→y[2]+*(up->e1.p)
      1a:   90              nop
      1b:   5d              pop    %ebp
      1c:   c3              ret
```

因此 C 代码为:

```
void proc(struct ele * up) {
        p → next → y[0] = * (up → e1.p) + up → y[2];
}
```

(3)

```
hyc@161220049:~/workspace/lab04/161220049$ vim address.c
hyc@161220049:~/workspace/lab04/161220049$ gcc -g address.c -o address
hyc@161220049:~/workspace/lab04/161220049$ ./address
array address:
bfb852bc        bfb852cc        bfb852dc

list address:
91940c8 9194080 9194048 9194020 9194008
hyc@161220049:~/workspace/lab04/161220049$
```

由数组和链表之间的比对可知，数组每个元素的地址连续，且为等差数列，相差 16，即为 sizeof(ele)；而链表之间的地址不存在连续性，无等差关系。数组利用静态内存，存放在栈里，其地址连续；而链表利用动态内存，存放在堆里，地址离散。