练习题 3.38　考虑下面的源代码，其中 M 和 N 是用# define 声明的常数：

```
long P[M][N];
long Q[N][M];

long sum_element(long i, long j) {
    return P[i][j] + Q[j][i];
}
```

在编译这个程序中，GCC 产生如下汇编代码：

```
    long sum_element(long i, long j)
    i in %rdi, j in %rsi
1   sum_element:
2     leaq    0(,%rdi,8), %rdx
3     subq    %rdi, %rdx
4     addq    %rsi, %rdx
5     leaq    (%rsi,%rsi,4), %rax
6     addq    %rax, %rdi
7     movq    Q(,%rdi,8), %rax
8     addq    P(,%rdx,8), %rax
9     ret
```

运用逆向工程技能，根据这段汇编代码，确定 M 和 N 的值。

3.8.4　定长数组

C 语言编译器能够优化定长多维数组上的操作代码。这里我们展示优化等级设置为-O1时 GCC 采用的一些优化。假设我们用如下方式将数据类型 fix_matrix 声明为 16×16 的整型数组：

```
#define N 16
typedef int fix_matrix[N][N];
```

(这个例子说明了一个很好的编码习惯。当程序要用一个常数作为数组的维度或者缓冲区的大小时，最好通过# define 声明将这个常数与一个名字联系起来，然后在后面一直使用这个名字代替常数的数值。这样一来，如果需要修改这个值，只用简单地修改这个# define 声明就可以了。)图 3-37a 中的代码计算矩阵 A 和 B 乘积的元素 i，k，即 A 的行 i 和 B 的列 k 的内积。GCC 产生的代码(我们再反汇编成 C)，如图 3-37b 中函数 fix_prod_ele_opt 所示。这段代码包含很多聪明的优化。它去掉了整数索引 j，并把所有的数组引用都转换成了指针间接引用，其中包括(1)生成一个指针，命名为 Aptr，指向 A 的行 i 中连续的元素；(2)生成一个指针，命名为 Bptr，指向 B 的列 k 中连续的元素；(3)生成一个指针，命名为 Bend，当需要终止该循环时，它会等于 Bptr 的值。Aptr 的初始值是 A 的行 i 的第一个元素的地址，由 C 表达式 &A[i][0]给出。Bptr 的初始值是 B 的列 k 的第一个元素的地址，由 C 表达式 &B[0][k]给出。Bend 的值是假想中 B 的列 j 的第(n+1)个元素的地址，由 C 表达式 &B[N][k]给出。

下面给出的是 GCC 为函数 fix_prod_ele 生成的这个循环的实际汇编代码。我们看到 4 个寄存器的使用如下：%eax 保存 result，%rdi 保存 Aptr，%rcx 保存 Bptr，而%rsi 保存 Bend。