① 程序是如何实现倒位序的过程?

```
    /********** following code invert sequence ************/
    for ( i=0;i<SAMPLENUMBER;i++ )
    {
        x0=x1=x2=x3=x4=x5=x6=0;
        x0=i&0x01;
        x1=(i/2)&0x01;
        x2=(i/4)&0x01;
        x3=(i/8)&0x01;
        x4=(i/16)&0x01;
        x5=(i/32)&0x01;
        x6=(i/64)&0x01;
        xx=x0*64+x1*32+x2*16+x3*8+x4*4+x5*2+x6;
        dataI[xx]=dataR[i];
    }
```

这个程序的目的是实现倒位序（bit-reversed）操作，它对输入的索引进行反转，然后将数据从一个数组 `dataR` 复制到另一个数组 `dataI`。我们逐步分析这个代码，解释倒位序的过程。

1. 理解倒位序操作

倒位序（bit-reversed order）是指将一个数字的二进制位顺序反转。例如，对于一个 6 位二进制数字 `101110`，它的倒位序是 `011101`。

在程序中，倒位序的目标是将数组的索引按照其二进制表示的倒位顺序重新排列，从而将 `dataR[i]` 存储到 `dataI[xx]` 中，其中 `xx` 是 `i` 的倒位序。

2. 程序的核心步骤

```
for (i = 0; i < SAMPLENUMBER; i++) {
    x0 = x1 = x2 = x3 = x4 = x5 = x6 = 0;
    x0 = i & 0x01;           // 提取 i 的最低有效位
    x1 = (i / 2) & 0x01;     // 提取 i 的第 2 位
    x2 = (i / 4) & 0x01;     // 提取 i 的第 3 位
    x3 = (i / 8) & 0x01;     // 提取 i 的第 4 位
    x4 = (i / 16) & 0x01;    // 提取 i 的第 5 位
    x5 = (i / 32) & 0x01;    // 提取 i 的第 6 位
    x6 = (i / 64) & 0x01;    // 提取 i 的第 7 位
    xx = x0 * 64 + x1 * 32 + x2 * 16 + x3 * 8 + x4 * 4 + x5 * 2 + x6;
    dataI[xx] = dataR[i];    // 将倒位序索引的值从 dataR 移到 dataI
}
```

3. 逐步解析

- **位提取**:
  - `x0 = i & 0x01`：这个表达式通过 `&` 运算提取 `i` 的最低有效位。`i & 0x01` 结果为 `i` 的二进制表示的最后一位（`i % 2`）。
  - `x1 = (i / 2) & 0x01`：这个表达式通过右移 1 位（相当于除以 2）提取 `i` 的第二位。
  - 其他 `x2, x3, ..., x6` 以此类推，依次提取 `i` 的第三至第七位。
- **倒位序索引计算**:
  - `xx = x0 * 64 + x1 * 32 + x2 * 16 + x3 * 8 + x4 * 4 + x5 * 2 + x6`

    :
    - 通过将每一位按倒位序的位置赋值到 `x0, x1, ..., x6` 中，重新组合出 `xx`，这个值就是倒位序的索引。

    - 比如对于一个 7 位的 `i`，`xx` 是通过将 `i` 的各位从低到高反转后重新组合成新的数字。
- **数据存储**：
    - `dataI[xx] = dataR[i]`：把原数组 `dataR` 中索引为 `i` 的值，存储到倒位序的索引 `xx` 对应的位置。这样，`dataI` 中的元素就是 `dataR` 按照倒位序排列的结果。

4. 示例

假设 `i` 为 3：

- `i = 3` 的二进制表示是 `00000011`。
- 按照程序中的位提取：
    - `x0 = 3 & 0x01 = 1`，
    - `x1 = (3 / 2) & 0x01 = 1`，
    - `x2 = (3 / 4) & 0x01 = 0`，
    - `x3 = (3 / 8) & 0x01 = 0`，
    - `x4 = (3 / 16) & 0x01 = 0`，
    - `x5 = (3 / 32) & 0x01 = 0`，
    - `x6 = (3 / 64) & 0x01 = 0`。
- 重新组合得到倒位序索引：
    - `xx = x0 * 64 + x1 * 32 + x2 * 16 + x3 * 8 + x4 * 4 + x5 * 2 + x6 = 1 * 64 + 1 * 32 + 0 * 16 + 0 * 8 + 0 * 4 + 0 * 2 + 0 = 96`。

所以，`i = 3` 时，对应的倒位序索引 `xx` 为 96，将 `dataR[3]` 存入 `dataI[96]` 中。

② 程序中的 `L`、`b`、`p`、`j` 分别代表什么含义?

```
    /************** following code FFT *******************/
    for ( L=1;L<=7;L++ )
    { /* for(1) */
        b=1; i=L-1;
        while ( i>0 )
        {
            b=b*2; i--;
        } /* b= 2^(L-1) */
        for ( j=0;j<=b-1;j++ ) /* for (2) */
        {
            p=1; i=7-L;
            while ( i>0 ) /* p=pow(2,7-L)*j; */
            {
                p=p*2; i--;
            }
            p=p*j;
            for ( k=j;k<128;k=k+2*b ) /* for (3) */
            {
                TR=dataR[k]; TI=dataI[k]; temp=dataR[k+b];
                dataR[k]=dataR[k]+dataR[k+b]*cos_tab[p]+dataI[k+b]*sin_tab[p];
                dataI[k]=dataI[k]-dataR[k+b]*sin_tab[p]+dataI[k+b]*cos_tab[p];
                dataR[k+b]=TR-dataR[k+b]*cos_tab[p]-dataI[k+b]*sin_tab[p];
                dataI[k+b]=TI+temp*sin_tab[p]-dataI[k+b]*cos_tab[p];
            } /* END for (3) */
        } /* END for (2) */
    } /* END for (1) */
```

在这段 FFT （快速傅里叶变换）代码中，`L`、`b`、`p` 和 `j` 是控制 FFT 计算过程中的重要变量。它们在不同的层次上决定了算法的结构和数据的操作方式。

1. `L` **（层数）**：

- `L` 是一个表示 FFT 算法中阶段或层的变量。FFT 是通过逐步分治的方式来实现的，在每一层，数据被分成不同的子部分来进行计算。`L` 从 1 到 7 依次迭代（因为 `SAMPLENUMBER = 128`，即 27=1282^7 = 128，所以 FFT 有 7 层）。
- 每一层都会进行“蝶形运算”，即将输入信号进行合并。

2. `b`**（蝶形的步长）**：
   - `b` 控制每一层的操作的步长，表示在每一层中，计算的 "大小"。`b` 是 `2^(L-1)`，每次 `L` 增加一，`b` 会变为上一层的两倍，表示每一层分组的大小。
   - 例如，`L = 1` 时 `b = 1`，`L = 2` 时 `b = 2`，`L = 3` 时 `b = 4`，依此类推。
3. `p`**（蝶形操作中的指数值）**：
   - `p` 是用来计算 FFT 中的旋转因子（即复数指数的幂次），通常用于计算复数的旋转因子。在 FFT 中，旋转因子通常为 e−2πik/Ne^{-2\pi i k / N}，其中 `N` 是总样本数，`k` 是当前计算的子问题的索引。
   - 在这里，`p` 用来从 `sin_tab` 和 `cos_tab` 数组中获取旋转因子的索引，计算出旋转因子。
4. `j`**（当前分组的索引）**：
   - `j` 是当前层中子问题的索引，决定了每次处理的两组数据之间的距离。例如，在每层的运算中，`j` 会遍历当前层中所有分组的开始位置。`j` 变化时，它决定了对哪些数据进行计算。

③ 程序是如何实现蝶形结的运算?

```
for ( j = 0; j <= b - 1; j++) /* for (2) */
{
    p = 1; i = 7 - L;
    while (i > 0) /* p = pow(2, 7 - L) * j; */
    {
        p = p * 2; i--;
    }
    p = p * j;
    for ( k = j; k < 128; k = k + 2 * b ) /* for (3) */
    {
        TR = dataR[k]; TI = dataI[k]; temp = dataR[k + b];
        dataR[k] = dataR[k] + dataR[k + b] * cos_tab[p] + dataI[k + b] * sin_tab[p];
        dataI[k] = dataI[k] - dataR[k + b] * sin_tab[p] + dataI[k + b] * cos_tab[p];
        dataR[k + b] = TR - dataR[k + b] * cos_tab[p] - dataI[k + b] * sin_tab[p];
        dataI[k + b] = TI + temp * sin_tab[p] - dataI[k + b] * cos_tab[p];
    } /* END for (3) */
} /* END for (2) */
```

在 FFT 算法中，蝶形结（Butterfly）运算是核心步骤，它通过合并和分解的方式计算傅里叶变换。蝶形结的运算公式通常是：

$$Output[k] = Input[k] + WNk \cdot Input[k + N/2] \text{Output}[k] = \text{Input}[k] + W_N^k \cdot \text{Input}[k + N/2]$$

$$Output[k + N/2] = Input[k] - WNk \cdot Input[k + N/2] \text{Output}[k + N/2] = \text{Input}[k] - W_N^k \cdot \text{Input}[k + N/2]$$

其中 $WNk = e - 2\pi ik/N W_N^k = e^{-2\pi ik/N}$ 是旋转因子。

在程序中，蝶形结的实现分为几个步骤：

1. **数据交换与合并**：
   - 每次运算将输入信号的不同部分合并（即“蝶形”），合并过程中涉及到数据的交换和加减运算。
   - `TR = dataR[k]` 和 `TI = dataI[k]` 用来保存当前索引 `k` 的数据，计算完成后，再将合并后的数据存回。
2. **旋转因子计算**：
   - `cos_tab[p]` 和 `sin_tab[p]` 存储了旋转因子的余弦和正弦值，这些值通常是预先计算好并存储在表中的。`p` 作为旋转因子的索引值，在每次蝶形运算中，根据当前的 `p` 查找相应的旋转因子。
   - 每个蝶形结的运算会涉及到对数据的旋转，即通过 `cos` 和 `sin` 来乘以旋转因子，进行复数乘法操作。
3. **蝶形运算**：

- 对于每一层 `L`，程序将数据按步长 `b` 划分成若干组（每组两个数据），然后通过旋转因子进行加法和减法运算。
- 具体来说，程序计算 `dataR[k]` 和 `dataI[k]`，然后与旋转因子相乘，更新 `dataR[k]` 和 `dataI[k]` 的值。接着，保存到 `dataR[k+b]` 和 `dataI[k+b]` 中。

这就是典型的蝶形运算，步骤如下：

- **加法部分**：`dataR[k] = dataR[k] + dataR[k+b] * cos_tab[p] + dataI[k+b] * sin_tab[p]`。
- **减法部分**：`dataI[k] = dataI[k] - dataR[k+b] * sin_tab[p] + dataI[k+b] * cos_tab[p]`。
- **交换部分**：`dataR[k+b] = TR - dataR[k+b] * cos_tab[p] - dataI[k+b] * sin_tab[p]` 和 `dataI[k+b] = TI + temp * sin_tab[p] - dataI[k+b] * cos_tab[p]`。

4. **逐步缩小问题规模**：
   - 在每一层中，算法逐步合并子问题，直到最终得到完整的傅里叶变换结果。在每层中，步长 `b` 逐渐增大，表示每次合并的块也变大，直到最终处理完整个信号。