```cpp
// ALGORITHM-1

// Shared variable
int i = 0, j = 1;
int turn = i; // initial turn of Pi
```

```cpp
// Process i
void Pi()
{
    do
    {
        // Entry Section
        while (turn != i);

        // Critical Section
        cout << "Critical Section of
Pi" << endl;

        // Exit Section
        turn = j;

        // Remainder Section
        cout << "Remainder Section of
Pi" << endl;

    } while (true);
}
```

```cpp
// Process j
void Pj()
{
    do
    {
        // Entry Section
        while (turn != j);

        // Critical Section
        cout << "Critical Section of
Pj" << endl;

        // Exit Section
        turn = i;

        // Remainder Section
        cout << "Remainder Section of
Pj" << endl;

    } while (true);
}
```

```
// ALGORITHM-2

// Shared variable
int i = 0, j = 1;
bool flag[2];
flag[i] = flag[j] = false; // not ready to enter
```

```
// Process i
void Pi()
{
    do
    {
        // Entry Section
        flag[i] = true;
        while (flag[j]);

        // Critical Section
        cout << "Critical Section of
Pi" << endl;

        // Exit Section
        flag[i] = false;

        // Remainder Section
        cout << "Remainder Section of
Pi" << endl;

    } while (true);
}
```

```
// Process j
void Pj()
{
    do
    {
        // Entry Section
        flag[j] = true;
        while (flag[i]);

        // Critical Section
        cout << "Critical Section of
Pj" << endl;

        // Exit Section
        flag[j] = false;

        // Remainder Section
        cout << "Remainder Section of
Pj" << endl;

    } while (true);
}
```

```cpp
// PETERSON'S SOLUTION

// Shared variable
int i = 0, j = 1;
int turn = i;
bool flag[2];
flag[i] = flag[j] = false; // not ready to enter
```

```cpp
// Process i
void Pi()
{
    do
    {
        // Entry Section
        flag[i] = true;
        turn = j;
        while (flag[j] && turn == j);

        // Critical Section
        cout << "Critical Section of Pi" << endl;

        // Exit Section
        flag[i] = false;

        // Remainder Section
        cout << "Remainder Section of Pi" << endl;

    } while (true);
}
```

```cpp
// Process j
void Pj()
{
    do
    {
        // Entry Section
        flag[j] = true;
        turn = i;
        while (flag[i] && turn == i);

        // Critical Section
        cout << "Critical Section of Pj" << endl;

        // Exit Section
        flag[j] = false;

        // Remainder Section
        cout << "Remainder Section of Pj" << endl;

    } while (true);
}
```

```
// SIMPLIFIED BAKERY ALGORITHM

int n;
int num[n];
memset(num, 0, sizeof(num));
```

```
// Lock Function
lock(i)
{
    num[i] = max(num[0], num[1], ...,
num[n-1]) + 1;
    for (p = 0; p < n; p++)
        while (num[p] != 0 && num[p]
< num[i]);
}

// Unlock Function
unlock(i)
{
    num[i] = 0;
}
```

```
// Process Pi
void P(int i)
{
    // Entry Section
    lock(i);

    // Critical Section
    cout << "Critical Section of P"
<< i << endl;

    // Exit Section
    unlock(i);

    // Remaining Section
    cout << "Remaining Section of P"
<< i << endl;
}
```

```
// ORIGINAL BAKERY ALGORITHM

int n;
int num[n];
memset(num, 0, sizeof(num));
bool choosing[n];
memset(choosing, false, sizeof(choosing));
```

```
// Lock Function
lock(i)
{
    choosing[i] = true;
    num[i] = max(num[0], num[1], ...,
num[n-1]) + 1;
    choosing[i] = false;
    for (p = 0; p < n; p++)
    {
        while (choosing[p]);
        while (num[p] != 0 &&
(num[p], p) < (num[i], i));
    }
}

// Unlock Function
unlock(i)
{
    num[i] = 0;
}
```

```
// Process Pi
void P(int i)
{
    // Entry Section
    lock(i);

    // Critical Section
    cout << "Critical Section of P"
<< i << endl;

    // Exit Section
    unlock(i);

    // Remaining Section
    cout << "Remaining Section of P"
<< i << endl;
}
```

```cpp
// MUTUAL EXCLUSION WITH TEST-AND-SET
bool lock = false;

bool TestAndSet(boolean * target)
{
    bool rv = *target;
    *target = true; // lock it
    return rv;
}

// Process P
void P(int i)
{
    do
    {
        // Acquire Lock
        while (TestAndSet(&lock));

        // Critical Section
        cout << "Critical Section of P" << i << endl;

        // Release lock
        lock = false;

        // Remainder section
        cout << "Remainder Section of P" << i << endl;

    } while (true);

}
```

```
// MUTUAL EXCLUSION WITH SWAP
bool lock = false;

void swap(bool * a, bool * b)
{
    bool tmp = *a;
    *a = *b;
    *b = tmp;
}

// Process P
void P(int i)
{
    do
    {
        // Acquire lock
        key = true; // unique to each process
        while (key == true)
            swap(&lock, &key);

        // Critical Section
        cout << "Critical Section of Process " << i << endl;

        // Release lock
        lock = false;

        // Remaining Section
        cout << "Remaining Section of Process " << i << endl;

    } while (true);


}
```