## Fill Grid

Given two one-dimensional arrays \( LL \) and \( UR \), fill in the program on the next page to insert the elements of \( LL \) into the lower-left triangle of a square two-dimensional array \( S \) and \( UR \) into the upper-right triangle of \( S \), without modifying elements along the main diagonal of \( S \). You can assume \( LL \) and \( UR \) both contain at least enough elements to fill their respective triangles. (Spring 2020 MT1)

For example, consider

```c
int[] LL = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 0, 0 };
int[] UR = { 11, 12, 13, 14, 15, 16, 17, 18, 19, 20 };
int[][] S = {
    { 0, 0, 0, 0, 0},
    { 0, 0, 0, 0, 0},
    { 0, 0, 0, 0, 0},
    { 0, 0, 0, 0, 0},
    { 0, 0, 0, 0, 0}
};
```

After calling `fillGrid(LL, UR, S)`, \( S \) should contain

```c
{
    { 0, 11, 12, 13, 14 },
    { 1, 0, 15, 16, 17 },
    { 2, 3, 0, 18, 19 },
    { 4, 5, 6, 0, 20 },
    { 7, 8, 9, 10, 0 }
}
```

(The last two elements of \( LL \) are excess and therefore ignored.)
/** Fill the lower-left triangle of S with elements of LL and the
  * upper-right triangle of S with elements of UR (from left-to
  * right, top-to-bottom in each case). Assumes that S is square and
  * LL and UR have at least sufficient elements. */
public static void fillGrid(int[] LL, int[] UR, int[][] S) {
    int N = S.length;
    int kL, kR;
    kL = kR = 0;
    for (int i = 0; i < N; i += 1) {
       什么都别做
    }
}
Solution:

```java
public static void fillGrid(int[] LL, int[] UR, int[][] S) {
    int N = S.length;
    int kL, kR;
    kL = kR = 0;
    for (int i = 0; i < N; i += 1) {
        for (int j = 0; j < N; j += 1) {
            if (i < j) {
                S[i][j] = UR[kR);
                kR += 1;
            } else if (i > j) {
                S[i][j] = LL[kL];
                kL += 1;
            }
        }
    }
}
```

Alternate Solutions:

```java
public static void fillGrid(int[] LL, int[] UR, int[][] S) {
    int N = S.length;
    int kL, kR;
    kL = kR = 0;
    for (int i = 0; i < N; i += 1) {
        for (int j = 0; j < i; j += 1) {
            S[i][j] = LL[kL];
            kL += 1;
        }
        for (int j = i + 1; j < N; j += 1) {
            S[i][j] = UR[kR];
            kR += 1;
        }
    }
}
```

```java
public static void fillGrid(int[] LL, int[] UR, int[][] S) {
    int N = S.length;
    int kL, kR;
    kL = kR = 0;
    for (int i = 0; i < N; i += 1) {
        System.arraycopy(LL, kL, S[i], 0, i);
        System.arraycopy(UR, kR, S[i], i + 1, N - i - 1);
        kL += i;
        kR += square.length - i - 1; /*
    }
}
```

2 Even Odd

Implement the method evenOdd by destructively changing the ordering of a given IntList so that even indexed links precede odd indexed links.

For instance, if lst is defined as IntList.list(0, 3, 1, 4, 2, 5), evenOdd(lst) would modify lst to be IntList.list(0, 1, 2, 3, 4, 5).

You may not need all the lines.

**Hint:** Make sure your solution works for lists of odd and even lengths.

```java
public class IntList {
    public int first;
    public IntList rest;
    public IntList(int f, IntList r) {
        this.first = f;
        this.rest = r;
    }

    public static void evenOdd(IntList lst) {
        if (__________________________________________) {
            return;
        }

        while (____________________________________________) {
            ________________________________________________
            ________________________________________________
            ________________________________________________
            ________________________________________________
            ________________________________________________
            ________________________________________________
        }
    }
}
```
Solution:

```java
public static void evenOdd(IntList lst) {
    if (lst == null || lst.rest == null) {
        return;
    }
    IntList oddList = lst.rest;
    IntList second = lst.rest;
    while (lst.rest != null && oddList.rest != null) {
        lst.rest = lst.rest.rest;
        oddList.rest = oddList.rest.rest;
        lst = lst.rest;
        oddList = oddList.rest;
    }
    lst.rest = second;
}
```

Alternate Solution:

```java
public static void evenOdd(IntList lst) {
    if (lst == null || lst.rest == null || lst.rest.rest == null) {
        return;
    }
    IntList second = lst.rest;
    int index = 0;
    while (!(index % 2 == 0 && (lst.rest == null || lst.rest.rest == null))) {
        IntList temp = lst.rest;
        lst.rest = lst.rest.rest;
        lst = temp;
        index++;
    }
    lst.rest = second;
}
```

Explanation: For any linked list, observe that we simply want to change the rest attribute of each IntList instance to skip an IntList instance. Looking at `lst`, we want to link 0 to 1, 3 to 4, and so on. This will constitute the work of the body of the while loop, so we just need to figure out how to link the last even indexed IntList instance to the first odd indexed IntList instance. To keep track of the first odd indexed IntList instance, we can use `second`. Now, we just need to exit the while loop when we are at the last even indexed IntList instance. This occurs when the index is even and we are either at the second to last element (`lst.rest.rest == null`) or the last element (`lst.rest == null`).
3 Partition

Implement partition, which takes in an IntList lst and an integer k, and destructively partitions lst into k IntLists such that each list has the following properties:

1. It is the same length as the other lists. If this is not possible, i.e. lst cannot be equally partitioned, then the later lists should be one element smaller.
   For example, partitioning an IntList of length 25 with k = 3 would result in partitioned lists of lengths 9, 8, and 8.

2. Its ordering is consistent with the ordering of lst, i.e. items in earlier in lst must precede items that are later.

These lists should be put in an array of length k, and this array should be returned.
For instance, if lst contains the elements 5, 4, 3, 2, 1, and k = 2, then a possible partition (note that there are many possible partitions), is putting elements 5, 3, 2 at index 0, and elements 4, 1 at index 1.

You may assume you have the access to the method reverse, which destructively reverses the ordering of a given IntList and returns a pointer to the reversed IntList. You may not create any IntList instances. You may not need all the lines.

**Hint:** You may find the % operator helpful.

```java
public static IntList[] partition(IntList lst, int k) {
    IntList[] array = new IntList[k];
    int index = 0;
    IntList L = ______________________________
    while (L != null) {
        ______________________________
        ______________________________
        ______________________________
        ______________________________
        ______________________________
        ______________________________
        ______________________________
        ______________________________
        ______________________________
    }
    return array;
}
```
Solution:

```java
public static IntList[] partition(IntList lst, int k) {
    IntList[] array = new IntList[k];
    int index = 0;
    IntList L = reverse(lst);
    while (L != null) {
        IntList prevAtIndex = array[index];
        IntList next = L.rest;
        array[index] = L;
        array[index].rest = prevAtIndex;
        L = next;
        index = (index + 1) % array.length;
    }
    return array;
}
```