Task 1

0 H H H T T T

1 _ _ H T T T **H** **H**

2 _ _ H T T _ _ H **T** **H**

3 _ _ _ _ T H **T** **H** T H

Yellow shows the pair that is about to get moved. The Bold is what was just moved. It is solvable in 3 moves.

Firstly, we initialise an array that has 4 additional elements on the end.
1. We then jump the first two heads to the first two new “blank spots”.
2. After that we jump the last “TH” we see till the last two blank spots.
3. Then jump forward the first HT in to the spots left behind by the previous step.

For any n that is above 3, the algorithm has one more step. 1. 2. Remain the same. In the case of n = 4 or anything above we do this:

0 H H H H T T T T

1 _ _ H H T T T T **H** **H**

2 _ _ H H T T T _ _ H **T** **H**

For the next step, we then jump forward the H T into the next blank spots we can find:

3 _ _ H _ _ T T H **T** **H** T H

Then we move back the further away TT into the holes left from the moved TH

4 _ _ H **T** **T** _ _ H T H T H

Then we jump this H T Forward into the holes left by the TT that was moved back.

Step 3 and 4 are replicated back and forth until we've run out of double TT's to move back, which means we and all that needs to occur is the final H T jumping into the blank spots.

Java Code:

```
coins = new char[(heads * 2)  + 4];
for (int i = 0; i < coins.length; i++) {
    if(heads > 0){
        coins[i] = 'H';
        heads--;
    } else if(tails > 0){
        coins[i] = 'T';
        tails--;
    } else{
        coins[i] = '_';
    }
}
int setup = 2 * heads;
int to = 2 * heads - 1;
int from = heads - 1;

coins[setup] = coins[0];
coins[0] = '_';
coins[setup + 1] = coins[1];
coins[1] = '_';
coins[setup + 2] = coins[setup - 1];
coins[setup - 1] = '_';
coins[setup + 3] = coins[setup];
coins[setup] = '_';
moves += 2;
while(from >= 2){
    coins[to] = coins[from];
    coins[to + 1] = coins[from + 1];
    coins[from] = '_';
    coins[from + 1] = '_';
    moves++;
    to -= 2;
    if(from > 2){
        moves++;
        coins[from] = coins[to];
        coins[to] = '_';
        coins[from + 1] = coins[to + 1];
        coins[to + 1] = '_';
    }
    from--;
}
```

Task 2

0 _ _ H H H T T

1 **H T** H H _ _ T

2 H T _ _ **H H** T

3 H T **H T** H _ _

Yellow shows the pair that is about to get moved. The Bold is what was just moved. It is solvable in 3 moves. This algorithim is very similar to the n, n version but we have to increment 2 negative indexes for the array

Firstly, we initialise the array with 2 additional elements at the start.
1. We then jump the first H T to the beginning of the array into the blank spots
2. We then move across the first pair of HH into the blank spots left by the previous spot.
3. We then jump the last HT into the spot left behind from the previous step.
Steps 2 and 3 are repeated until algorthim is complete. 4, 3 for illustration:

0 _ _ H H H H T T T

1 **H T** H H H _ _ T T

2 H T _ _ H **H H** T T

3 H T **H T** H H _ _ T

4 H T H T _ _ **H H** T

5 H T H T **H T** H _ _

Java Code for Task 2:

```
coins = new char[(heads + tails)  + 2];
coins[0] = '_';
coins[1] = '_';
for (int i = 2; i < coins.length; i++) {
    if(heads > 0){
        coins[i] = 'H';
        heads--;
    } else if(tails > 0){
        coins[i] = 'T';
        tails--;
    }
}

int from = heads + 1;
int to = 0;
while(from < coins.length - 1){
    coins[to] = coins[from];
    coins[to + 1] = coins[from + 1];
    coins[from] = '_';
    coins[from + 1] = '_';
    to += 2;

    if(from == coins.length - 2){
        break;
    }
    coins[from] = coins[to];
    coins[from + 1] = coins [to + 1];
    coins[to] = '_';
    coins[to + 1] = '_';
    from++;
}
```

Because of the repeated nature of both algorithms we initialise the code with the indexes hard coded as functions of the number of heads, and increment or decrement the moving indexes as the above steps are taken. We know the exact index to pull from, and where to place, for both the TH forward steps and TT back steps. This prevents us from having to look at the entire array to determine “is this a pair of TT”, which greatly increases efficiency.

Efficiency (this was done with the n, n – 1 test).
Because we're not looping through the iteration, we weren't too sure how to count the actions/iterations. We included a num_actions counter for all actions that were updating coin positions, deleting coin positions and making changes to the various indexes for our algorithm. Our n is based on the number of coins.

The n, n algorithim also does not loop through the array. It knows the exact indexes to move every time.