# 6/17/2020 - Recursion #2

Wednesday, June 17, 2020 4:04 PM

1. Recurrence Relations - Towers of Hanoi # of Steps
2. Merge Sort
3. Floodfill

Towers of Hanoi - Proof of Number of Steps

Towers(n):

1. We are FORCED to solve the problem for n-1 disks, since without doing so, we CAN'T move the bottom.
2. We must move the bottom disk
3. We are FORCED to solve the problem AGAIN for n-1 disks.

Let T(n) = the minimum # of steps to solve a towers of hanoi puzzle for n disks.

$T(n) = T(n-1) + 1 + T(n-1)$

$T(n) = 2T(n-1) + 1$, recursively defined function.
$T(1) = 1$

We will learn the iteration technique to solve recursively defined functions (recurrence relations).

- 1

$T(n) = 2T(n-1) + 1$, this is a formula for all n. Imagine plugging in n-1 for n…

$T(n-1) = 2T((n-1)-1) + 1 = 2T(n-2) + 1$
$T(n-2) = 2T((n-2)-1) + 1 = 2T(n-3) + 1$

$T(n) = 2T(n-1) + 1$ … given (1 iteration)
$= 2( 2T(n-2) + 1 ) + 1$
$= 4T(n-2) + 2 + 1$
$= 4T(n-2) + 3$ … (2 iterations)
$= 4 ( 2T(n-3) + 1 ) + 3$
$= 8T(n-3) + 4 + 3$

$= 8T(n-3) + 7$ … (3 iterations)

We can see after k iterations, we will have:

$T(n) = 2^kT(n - k) + (2^k - 1)$

Now, plug in k = n -1 to this formula:

$T(n) = 2^{n-1}T(n - (n-1)) + (2^{n-1} - 1)$
$= 2^{n-1}T(1) + (2^{n-1} - 1)$
$= 2^{n-1}(1) + (2^{n-1} - 1) = 2^{n-1} + 2^{n-1} - 1 = 2^{n-1}(1 + 1) - 1$
$= 2^{n-1}(2^1) - 1 = 2^{(n-1)+1} - 1 = 2^n - 1$
$= 2^n - 1$

Merge Sort - (out of order so you can get started on the assignment)

You are given a bunch of stuff out of order, as well as a definition of how to compare two items (which one comes first…) - use this to sort the list.

8 , 3 , 2 , 1 , 9, 4, 7, 6

Imagine sorting the blue stuff in order, recursively.
Then, doing the same for the purple:

1 , 2 , 3 , 8 , 4, 6, 7, 9

How can we put together (merge) these two sorted lists into one?

We already know that the smallest item must be either the first item in the first list or the first item in the second list. So, in one comparison, we can copy that value over to our new list:

1,

The work that remains is to merge these two lists:

2 , 3 , 8 , 4, 6, 7, 9

Now, I'll show the rest of the process on the video:

The work that remains is to merge these two lists:

1, 2 , 3 ,4, 6, 7, 8 , 9

If we had two input arrays, of size n and size m, the run time of this merge is O(n+m), because I do order 1 work for each item in each list…one comparison for each copy…If both lists are of size n, this takes O(n) time.

```
// Sort array[startIndex…endIndex]
void mergeSort(int* array, int startIndex, int endIndex) {
    if (startIndex < endIndex) {
        int mid = (startIndex+endIndex)/2;
        mergeSort(array, startIndex, mid);
        mergeSort(array, mid+1, endIndex);
        merge(array, start, mid+1, endIndex);
    }
}
```

What about the run time of Merge Sort?
Let T(n) be the run time for a merge sort of an array of size n.

T(n) = T(n/2) + T(n/2) + O(n)
(1st rec) (2nd rec) (merge)

T(n) = 2T(n/2) + O(n), recurrence relation for the run time of merge sort.

O(n) = O(2n) but T(n) != T(n/2)

T(n) = 2T(n/2) + cn, since O(n) <= cn for some c. (1 iteration)

= 2 ( 2T(n/4) + c(n/2) ) + cn
= 4T(n/4) + cn + cn
= 4T(n/4) + 2cn (2 iteration)

$= 4 ( 2T(n/8) + c(n/4) ) + 2cn$
$= 8T(n/8) + cn + 2cn$
$= 8T(n/8) + 3cn$ (3 iterations)

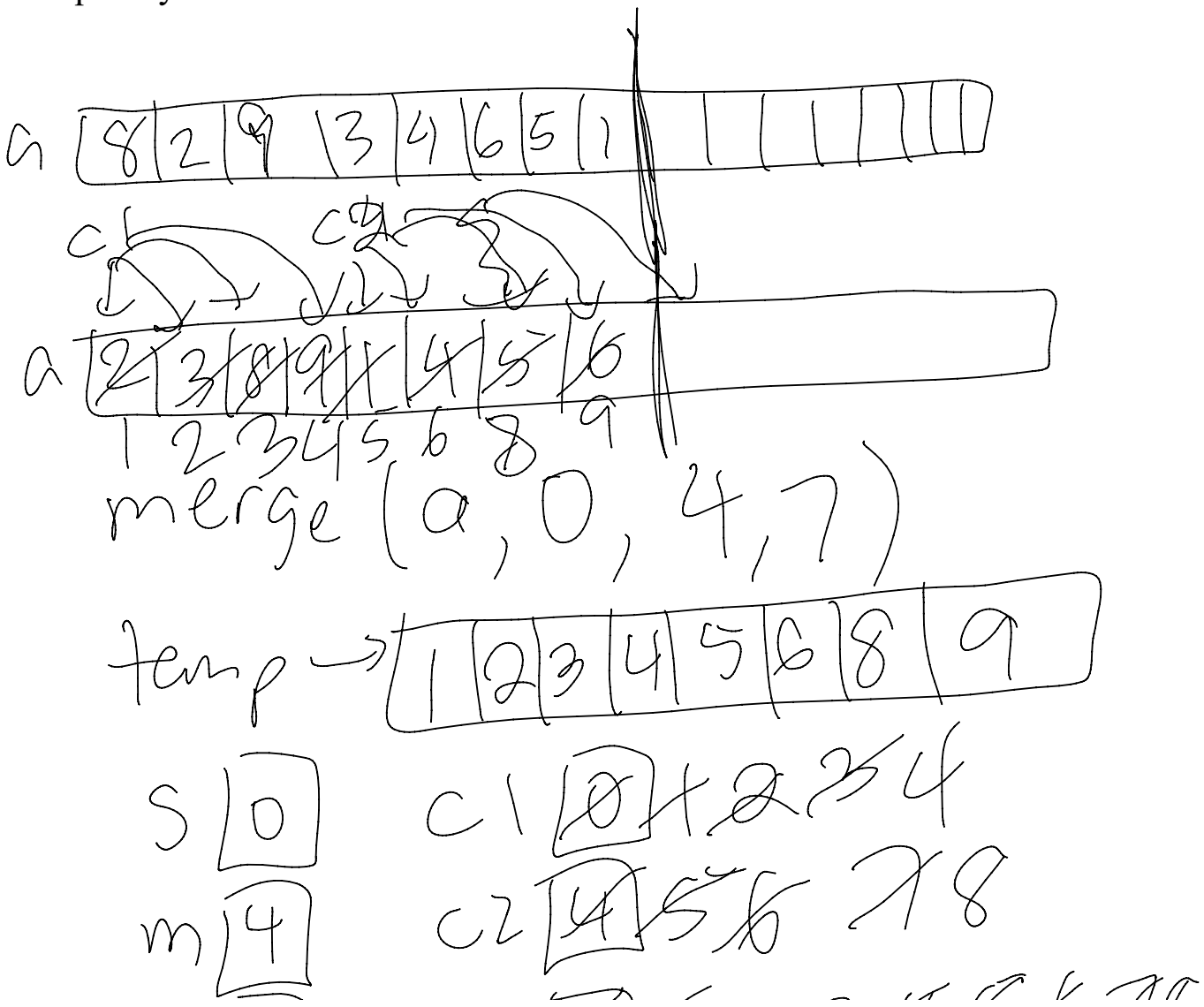
After k iterations we find:

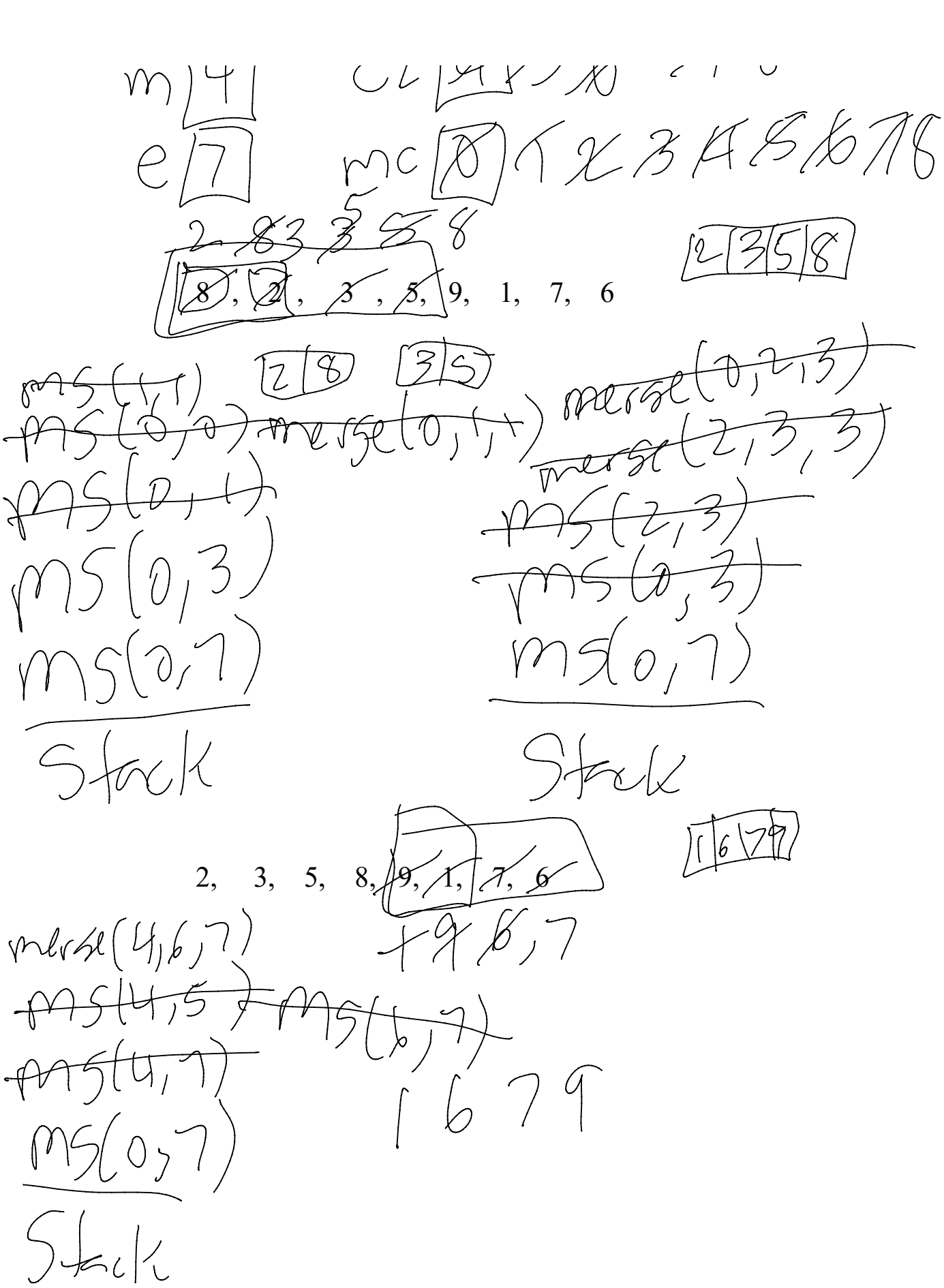
$T(n) = 2^kT(n/2^k) + kcn$

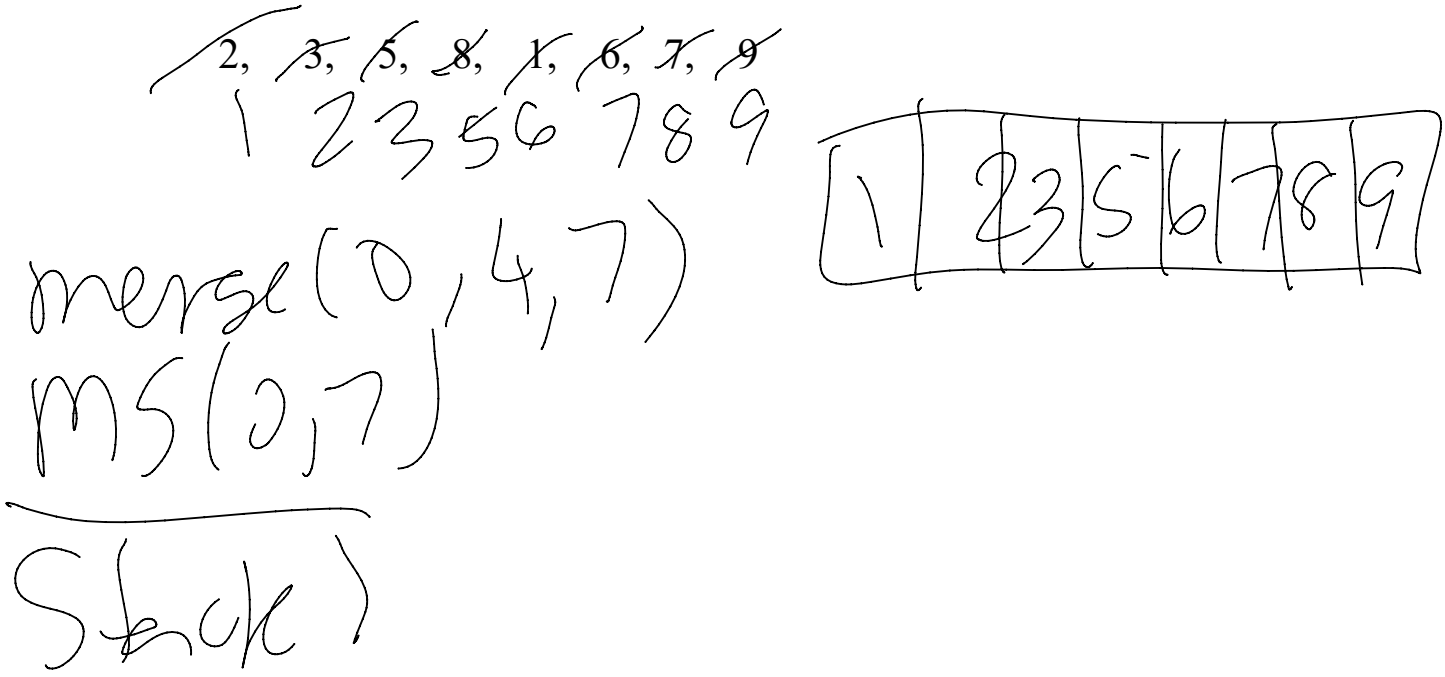
We want to plug in k such that $n/2^k = 1$ , since $T(1) = 1$ (const time to sort an array of size 1). $n = 2^k$, $k = \log_2 n$.

$T(n) = nT(1) + cn(\log_2 n)$
$= n + cn(\lg n)$
$= O(n \lg n)$

So, if $n = 10^6$, nlgn ~ 20 million, which a computer can do pretty quickly.




8, 2, 3, 5, 9, 1, 7, 6
2 8
3 5
2 3 5 8
ms(1,1)
ms(0,0) merge(0,1,1)
ms(0,1)
ms(0,3)
ms(0,7)
Stack
merge(0,2,3)
merge(2,3,3)
ms(2,3)
ms(0,3)
ms(0,7)
Stack
2, 3, 5, 8, 9, 1, 7, 6
1 6 7 9
merge(4,6,7)
ms(4,5) ms(6,7)
ms(4,7)
ms(0,7)
Stack
1 6 7 9




Insertion Sort

3, 8, 2, 9, 1, 5, 4

We will insert each new item into the sorted list.

We start with a sorted list of size 1: 3

Now, insert 8…compare 8 to 3, it's bigger, so we are good.

No we have a sorted list of size 2: 3, 8, now we must insert 2:

3, 8, 2
3, 2, 8
2, 3, 8

Now we have a new sorted list of size 3: 2, 3, 8, now insert 9

2, 3, 8, 9

Now sorted list size 4: 2, 3, 8, 9, insert 1

2, 3, 8, 9, 1
2, 3, 8, 1, 9
2, 3, 1, 8, 9
2, 1, 3, 8, 9
1, 2, 3, 8, 9

Insert 5 into this list:

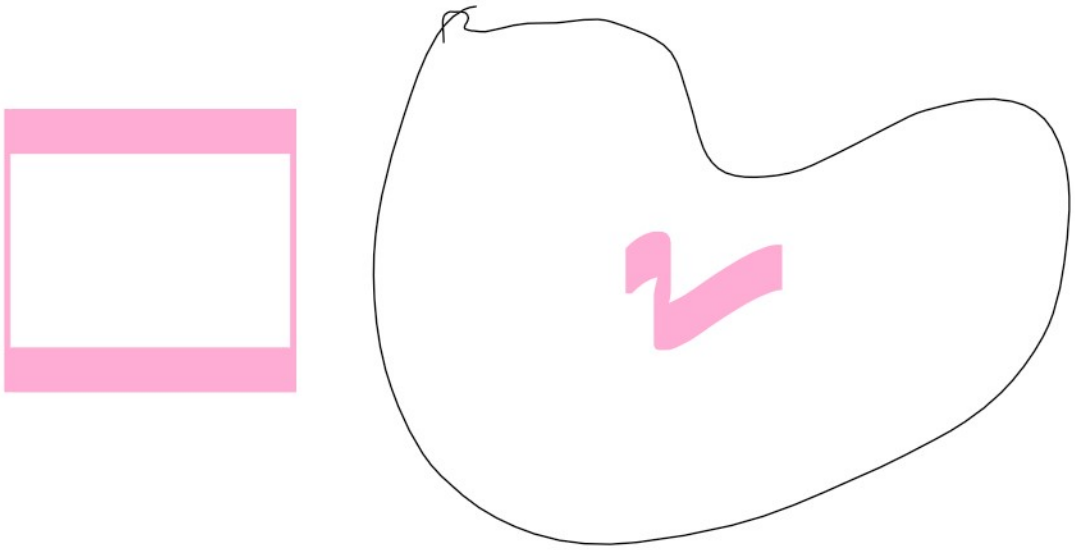
1, 2, 3, 8, 9, 5
1, 2, 3, 8, 5, 9
1, 2, 3, 5, 8, 9 done

1, 2, 3, 5, 8, 9, 4
1, 2, 3, 5, 8, 4, 9
1, 2, 3, 5, 4, 8, 9
1, 2, 3, 4, 5, 8, 9 done

Won't code today…will do later…

Floodfill:

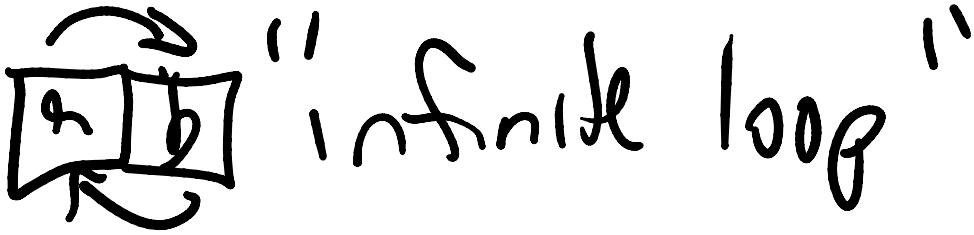
Motivation--- there are a bunch of problems where you start some process on a grid at a location, and it spreads, until you contain it.



How do I code a floodfill?

We call the function on some coordinate (x, y).
But, we don't want to call a floodfill on the same square twice. It would

We call the function on some coordinate (x, y).
But, we don't want to call a floodfill on the same square twice. It would do it forever.




We need a separate array to keep track of which squares have already been filled, so we don't fill them again. I usually call mine the "used" array.

The other way to do it, is make an actual change in your input array, and don't recall the recursive funciton on any square that's been changed.

Also, make sure you don't go out of bounds.

```
const int NUMDIR = 4;
const int DX[] = {-1,0,0,1};
const int DY[] = {0,-1,1,0};

void floodfill(int x, int y, int, int board[][NUMCOLS], int fillcolor, int
used[][NUMCOLS]) {

    if (!inbounds(x, y)) return;
    if (used[x][y]) return;
    if (boundary(x, y)) return;

    board[x][y] = fillcolor;
    used[x][y] = 1;

    for (int k=0; k<NUMDIR; k++) {
        int newx = x + DX[k];
        int newy = y + DY[k];
        floodfill(newx, newy);
    }
}


void floodfill(int x, int y, int, int board[][NUMCOLS], int fillcolor, int
```

```
used[][NUMCOLS]) {

    board[x][y] = fillcolor;
    used[x][y] = 1;

    for (int k=0; k<NUMDIR; k++) {
        int newx = x + DX[k];
        int newy = y + DY[k];
        if (!inbounds(newx, newy)) continue;
        if (used[newx][newy]) continue;
        if (boundary(newx, newy) continue;
        floodfill(newx, newy);
    }
}
```