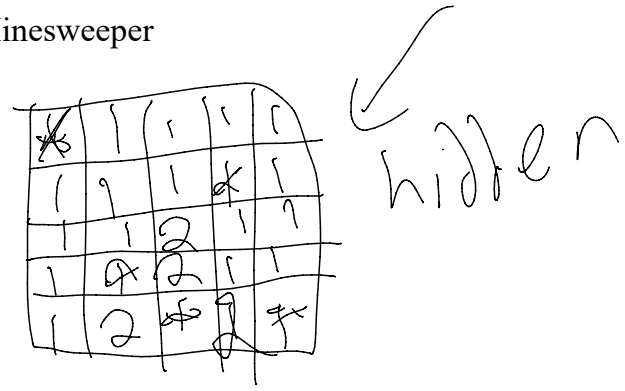
6/22/2020 Recursion #3 (Floodfill, Brute Force)

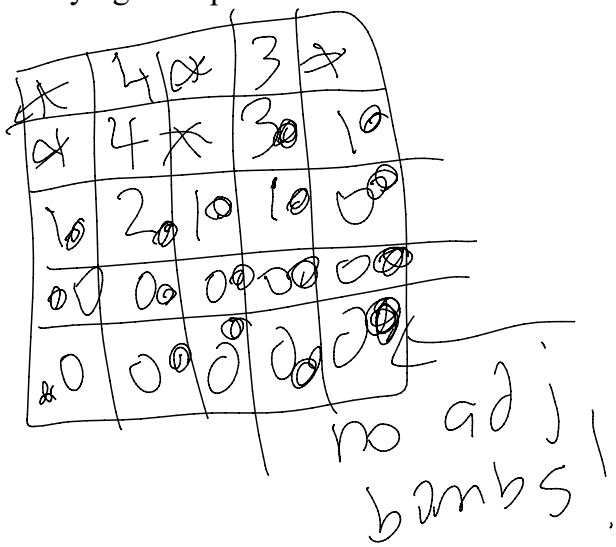
Monday, June 22, 2020 4:05 PM

1. Minesweeper
2. Brute Force
   a. Permutations
   b. Combinations
   c. Derangements
   d. Other Adjustments

Minesweeper



How to play
User is shown blanks…doesn't know where the bombs are.
Clicks on a square. If the square is a bomb, the user loses.
If it's not a bomb, a number is revealed. That number is the number of adjacent squares that have bombs.
Goal: click on every square that doesn't have a bomb, therefore identifying the squares that do.



If a square has no adjacent bombs, then it IS SAFE to click on all of the adjacent squares…so, instead of making the user do this one by one, in the real game, if you ever click on a square with no adjacent bombs, it

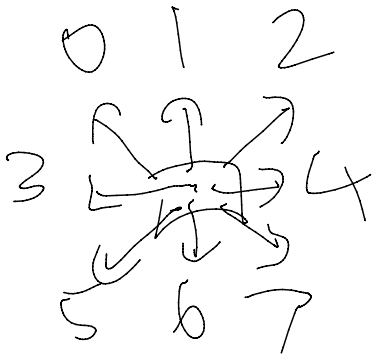
RECURSIVELY clears all of the adjacent squares!!!

When we are done, effectively, all SAFE squares that could have been cleared as a direct or indirect consequence of our one click are clicked for us. The "border" of squares not clicked will all have at least 1 adjacent bomb.

Basically, this is a floodfill!!! The squares/region to fill are all squares with 0 adjacent bombs, with a border of squares that don't have 0 adjacent bombs (no recursion is called on these squares).

DX/DY array
Stores directions of movement from a current square

(-1, -1) up and to the left
(-1, 0) up
(-1, 1) up and to the right
(0, -1) left
(0, 1) right
(1, -1) down to the left
(1, 0) down
(1, 1) down to the right

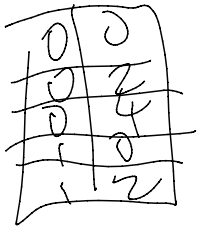



```
const int DX[] = {-1,-1,-1,0,0,1,1,1};
const int DY[] = {-1,0,1,-1,1,-1,0,1};
```

in minesweeper2

I get a list of bomb locations:

(0, 0)
(0, 2)
(0, 4)
(1, 0)
(1, 2)



Click on (2, 2)…

I decide how many adjacent bombs there are, but comparing (2, 2) to each ordered pair in the bomb location list.

(2,2) is NOT adjacent to (0, 0), since dx = 2, dy = 2

For a bomb to be adjacent to a square both |dx| <= 1 and |dy| <= 1.
If both are 0, then I clicked on a bomb. We have to filter out these cases first.

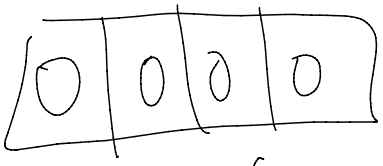
squaresCleared

pSquares

Brute Force

1. Odometer
2. Permutation
3. Combinations
4. Derangements
5. One Other Problem

Odometer



printOdom(int pdom[], int len, int k)

0000
0001
0002
..
0009
0010
0011
0012
…
0099
0100
…
9999

k represents the number of digits that are fixed in the odometer.

printOdom([2, 3, x, x], 4, 2) should print every odometer setting, in order, that starts with 23, so

2300
2301
2302
…
2309

printOdom([2, 3, 0, x], 4, 3)

2310
…
2319

printOdom([2, 3, 1, x], 4, 3)

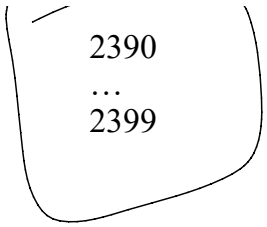
2320
…
2329

printOdom([2, 3, 2, x], 4, 3)

…

2390
…

printOdom([2, 3, 9, x], 4, 3)



printOdom([2, 3, 9, x], 4, 3)

At the end of the day, any printOdometer task can be broken down into 10 recursive printOdometer tasks, EXCEPT for when all the digits are FIXED!!!

printOdometer([2, 3, 2, 2], 4, 4) doesn't need recursion, it just needs to print 2322.

```
void printOdometer(int odometer[], int length, int k) {

    // If all digits are fixed, we have one thing to print, so print it and stop.
    if (k == length) {
        printArray(odometer, length);
        return;
    }

    // Otherwise, just try each possible digit in slot k, the first free slot and recurse.
    for (int digit=0; digit<10; digit++) {
         odometer[k] = digit;
         printOdometer(odometer, length, k+1);
    }
}
```

Permutations

0, 1, 2
0, 2, 1
1, 0, 2
1, 2, 0
2, 0, 1
2, 1, 0

In general, there are n! permutations of n distinct objects.

So, there are lots of problems, where we may want to know the best "ordering" of some items, and if we only have about 10 items at most, what we can do is try every ordering, and pick the best one.

We usually use this code to solve problems as follows:

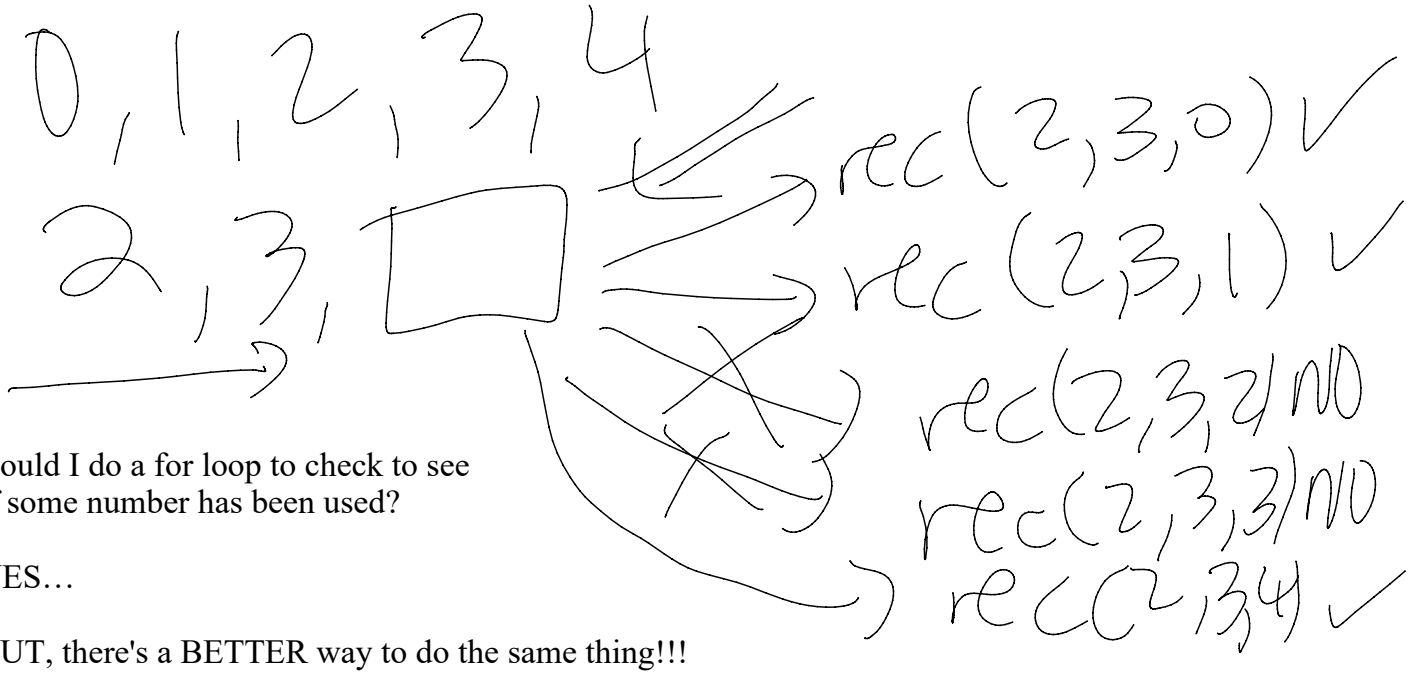
1. Store a current permutation always as an ordering of 0, 1, 2…, n-1 in one array, I'll call it perm.
2. Keep your objects you are reordering in a separate array, maybe call it objects or something else. NEVER REORDER THIS ARRAY.
3. When you are considering a particular permutation, use the perm array to INDEX into the object array. object[perm[0]], object[perm[1]], object[perm[2]], etc.

Does the list of permutations look a lot like the odometer printing?
Yes…BUT what's different?
all permutations are valid odometer settings, but…not all odometer settings are valid permutations…

Odometers allow repeats, a permutation doesn't…that's the ONLY difference.

Here is the odometer recursion (with changed names, and using a permutation of n items instead of 10) again:

```
for (int i=0; i<n; i++) {
    // In some cases, we want to SKIP a recursive call, because we already used
    // that digit…I want to skip i, if I previously placed i before.
    if (/* some condition */) {
        perm[k] = i;
        printPerms(perm, length, k+1);
    }
}
```



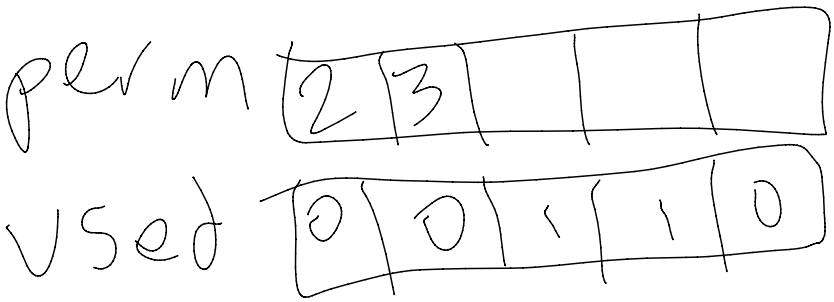
Could I do a for loop to check to see if some number has been used?

YES…

BUT, there's a BETTER way to do the same thing!!!

This is a huge principle in problem solving - a lot of time, you can save time by storing (potentially) redundant information. (We call this a time-memory trade-off, I use more memory to save time.)

I will store an array that marks which items have already been used.

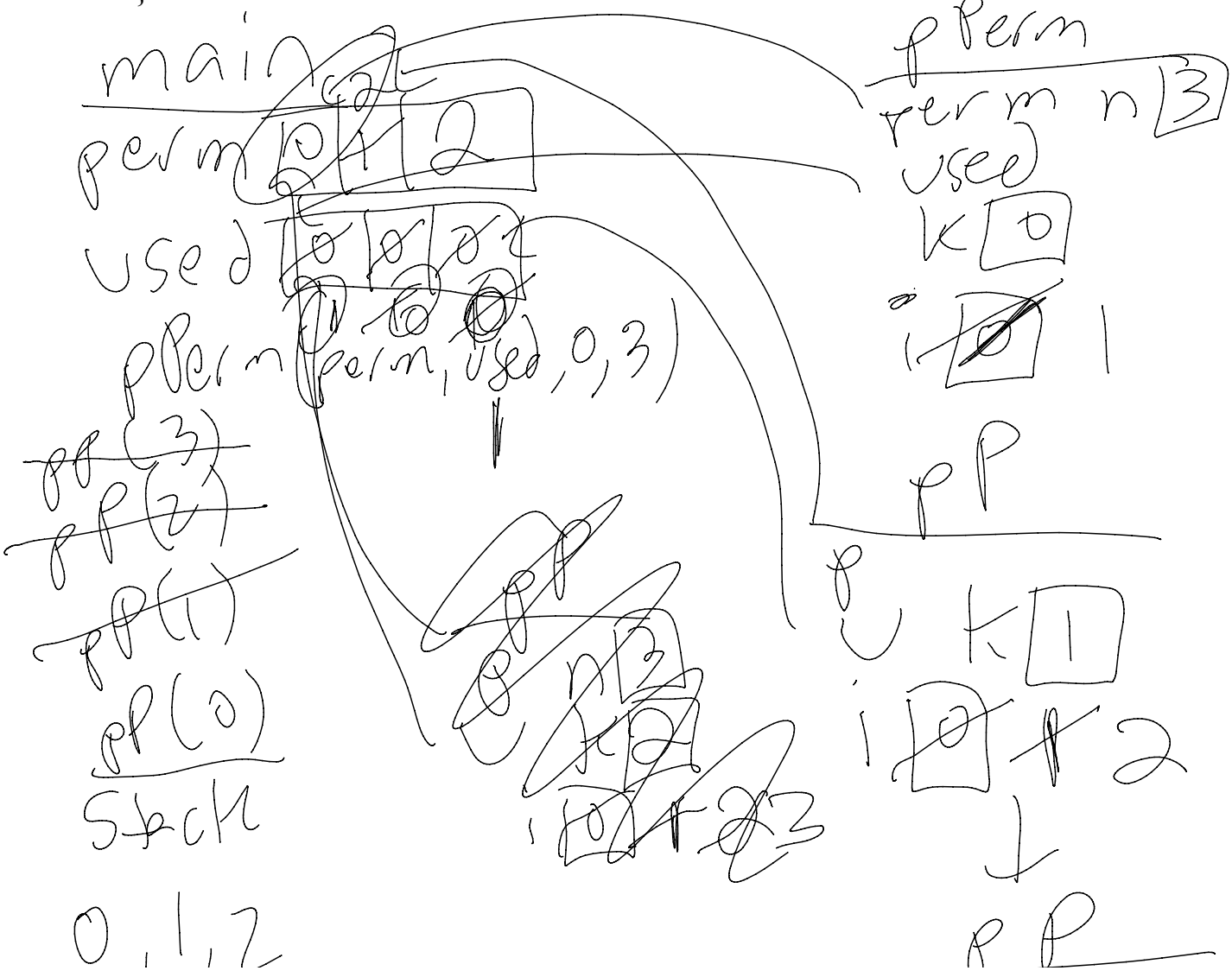


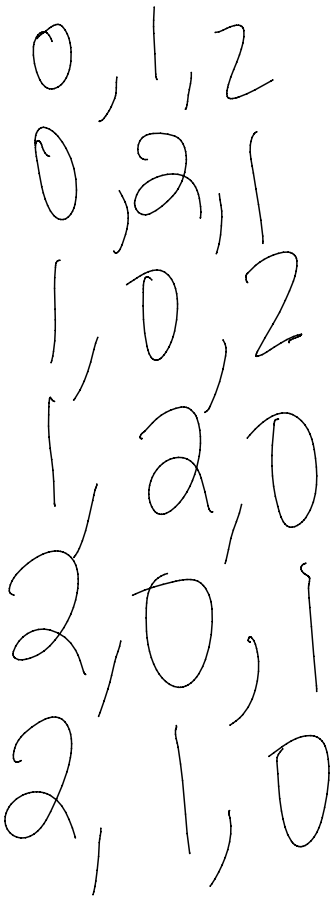
We will maintain BOTH arrays at the same time…

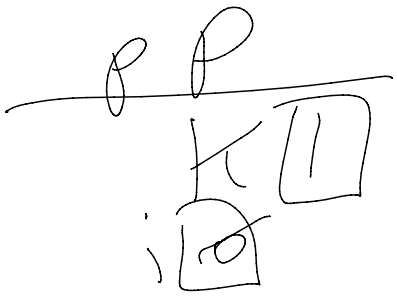
```
void printPerms(int perm[], int used[], int k, int n) {

    if (k == n) {
        print(perm, n);
        return;
    }

    // I represents the value we are trying in slot k.
    for (int i=0; i<n; i++) {
        if (!used[i]) {
            perm[k] = i;
            used[i] = 1; // mark i as used for the future.
            printPerms(perm, used, k+1, n);
            used[i] = 0; // HAVE TO DO THIS!!!
        }
    }
}
```







MAXJUMP

0,1,2,3,4,5

3,0,2,5,1,4

jump = 3 + 2 + 3+ 4 + 3 = 15
The amount of jumping for a permutation is the sum of the absolute values of the differences of each adjacent pair of values in the permutation.

Find the maximum sum of jumps of any permutation.

Method of solution:

Try all permutations.
For each, calculate the jump score.
Return the maximum of all of these.

Combinations…

A combination is all the possible combinations of items out of n items without regard to order

So all the combinations of

0, 1, 2 are

{} - nothing	000
{0}	001
{1}	010
{2}	100
{0,1}	011
{0,2}	101
{1, 2}	110
{0,1,2}	111

Another way of thinking about combinations is that for each item, it's either in the combination or NOT in the combination

This is EXACTLY an odometer with 2 digits, 0 and 1, so binary…