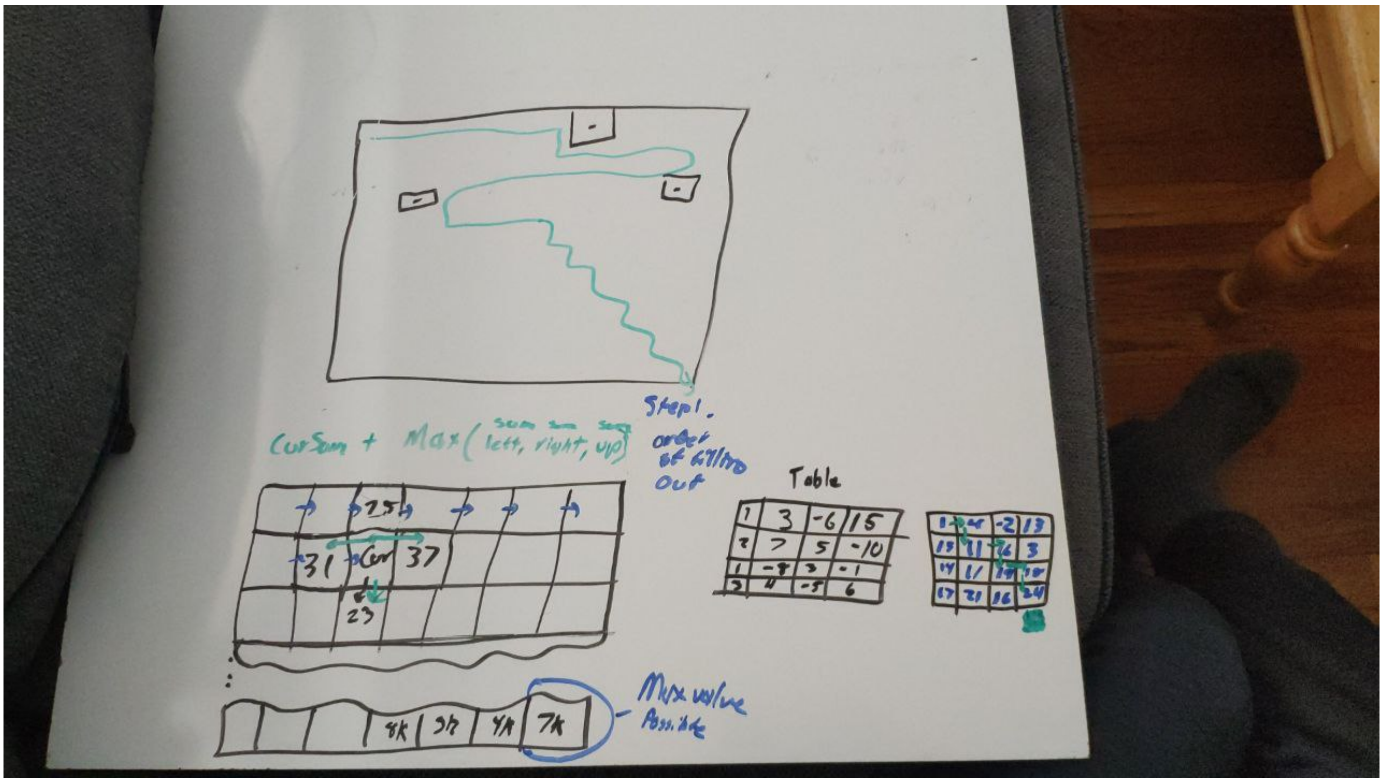
Pset3

Problem 0 - Dynamic Programming (25%)

Zoe starts at the top left corner of an n x n square grid of numbers, and must end at the bottom right corner. The numbers can be positie or negative. At each step, Zoe can go left, right, or down (but not up), and she can never revisit a square that she has already visited. Find an O(n^2) algorithm to find the maximum sum of numberes Zoe can visit.

Solution:

We know this dynamic programming solution has three parts, first the building of the table, secondarily, the identification of the invariant problem and maintenence, and third identifing the solution from the DP table. We know that we have three options for the filling out of the table, **Go left, Go right, and Go Down, but not up.** This means what we're actually solving for is the sum of the past nodes + Max(func.left, func.right, func.down). First, we fill out the dp table using this algorithm (taking the max of the left, right and down functions), to get a filled out DP table. Now, the number at the bottom right will now be the largest possible sum of the numbers of the original n x n table. If we now trace back up (similar to how we used memoization previously to store these values, or by going to tne largest value to the above, right, or to the left until we end up at the top [storing those values along the way]). Here, at max we visit each node 2 times, once when building the table, and once when traversing back up the list if we were to return our path. Elsewise, we return the sum from the bottom right corner of the dp table and have solved our problem.



Step 1. order of filling out
CurSum + Max(sum left, sum right, sum up)
Table
Max value Possible

Problem 1 - Divide and Conquer (25%)

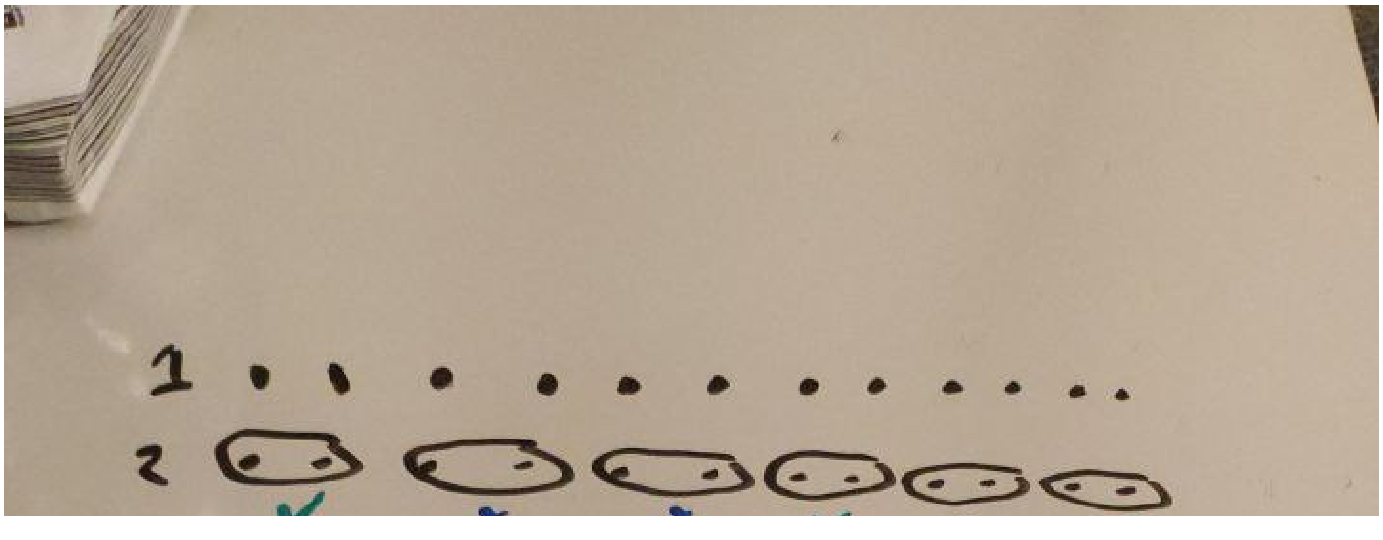
A mad scientist accidentally left her cloning machine on and open while vacationing in Storrs, Connecticut. While she was away, a murder of crows managed to get inside her office and use the cloning machine. The mad scientist returned from her vacation surprised to have her office full fo crows, but exicted to test a new hypothesis. As is well known, crows are very intelligent birds, and the scientist suspects that one or more crows may have taken advantage of the cloning machien to gain a selective genetic advantage.

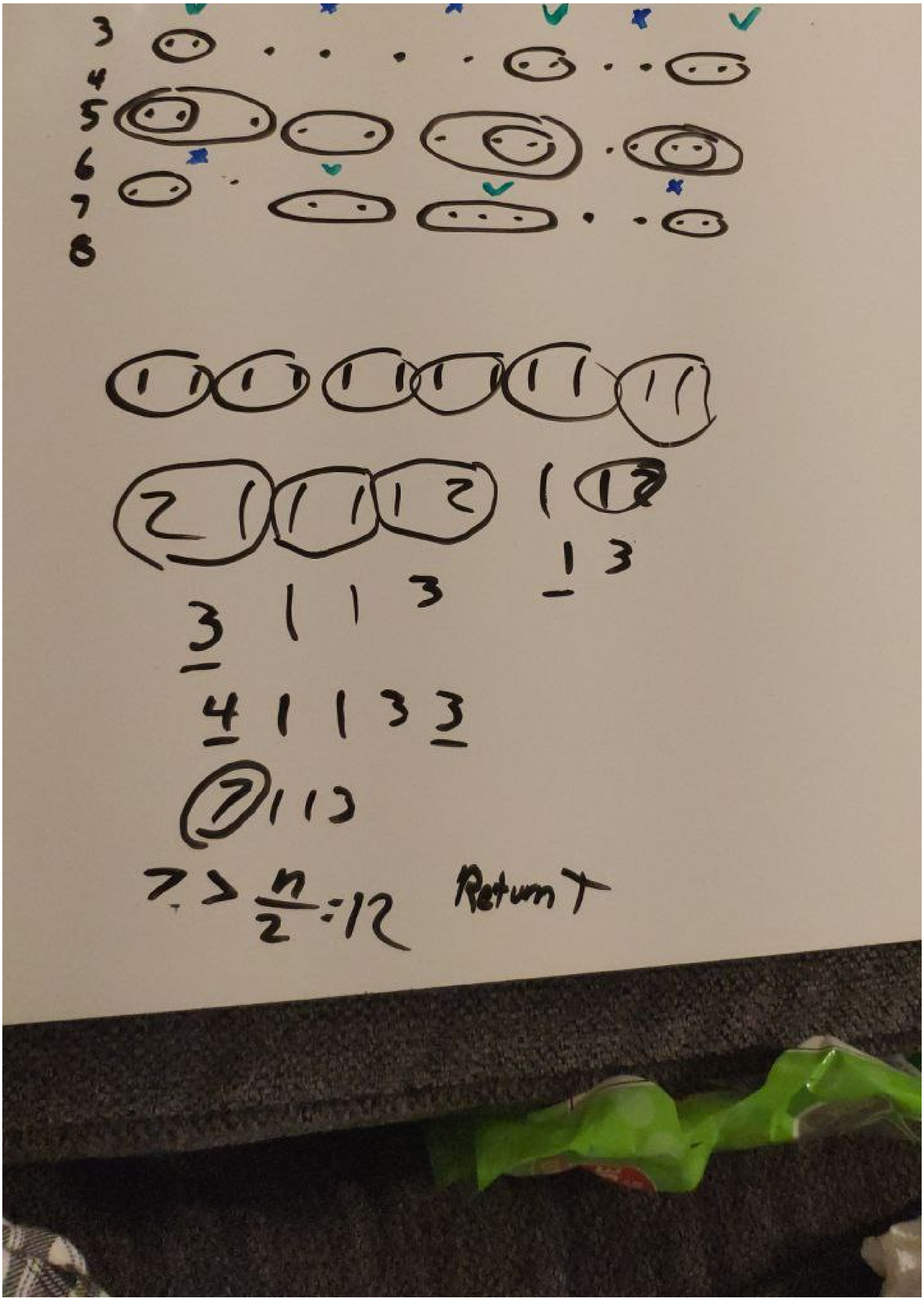
We say that two crows are *identical* if we genetically sequence the crows and their DNA is 99.99% identical. The mad scientist has a machine that will determine whether or not two crows are genetically identical, but it is costly to operate so she wants to minimize its use.

The only operation you are allowed to perform is to take two crows and determine if they are genetically identical. Develop an O($nlogn$) algorithm that, given *n* crows, determines if there is a set of more than n/2 crows that are genetically identical.

Solution:

The way this problem is solved is by using a derivative of the pigeonhole principle, where if you split the list into pairs, and test each of those pairs, if n/2 crows are genetically identical atleast one of the pairs of crows we've added will contain a matching pair. If that's not the case, we know that there are not n/2 crows which are genetically identical. While this case not happening proves that it does not exist, its existence does not prove that we have a dominant species of crow (greater than n/2). The way we would then do this problem is first divide the set of crows into pairs and compare them for similarity. Those which are of the same type genetically will be kept together in pairs, and those which do not match will be mixed and tested against other singles in the next round. You then can treat all the pairs as if they are individual units, and then organize into pairs and test for similarity (see picture below). As the pairs get larger and bunch into larger units, the number of test's you're "saving" grows exponentially. Another potential way to solve this would be to treat individual crows like "1's" and then sum the individuals if they are equal. Using the same technique as before, you can combine the pairs of individuals, making sure not to test any two individuals which have already been tested against one another. Finally, you will have a largest number that has tested against every other possible option and if that number is greater than half of the list, return true, else, return false. This is the termination sequence, when the largest possible option has been tested against every other option.





Problem 2 - SillySort (25%)

Prove that SillySort(L) terminates and correctly sorts its input list of numbers *L*.

```
def sillysort(L):
    while true:
        if L is sorted:
            return L
        else:
            compute an arbitrary index i, such that L[i] > L[i+1]
            swap L[i] and L[i+1]
```

Solution:

SillySort(L) will correctly solve this problem. The **initialization** starts with the while loop (while true:) with two conditions, if L issorted (a python check for if a list is sorted), return L, Else move to the maintenence. The maintenence is the else, compute an arbitrary index i, such that L[i] > L[i+1], then swap L[i] and L[i+1]. This **maintenence** step will work, as it is similar to a randomized version of bubble sort. If the algorithm only computes indexes where L[i] > L[i+1] and swaps them, it will decrease the level of "unsortedness" within a list with each swap. After a certain number of swaps, which is uperbound with len(L)! as a max and and a minimum of 1 swaps. If all these swaps are done a max of len(L)! times, then the list will be sorted and satisfy the **temination sequence**, returning the list. I'll note that this algorithm has similar problems to a 1 way bubble sort in that small values at the end of the list are going to likely be troublesome for this algorithm to move as it only moves things in one direction, and in general this is likely a slower algorithm than many other options.

Problem 3 - Data Structures (25%)

1. We saw in class the algorithm for building a max-heap

```
import math
def buildmaxheap(A):
    A.heapsize = A.length
    for i in range(int(math.floor(A.length/2)),0,-1)
        maxheapify(A,i)
```

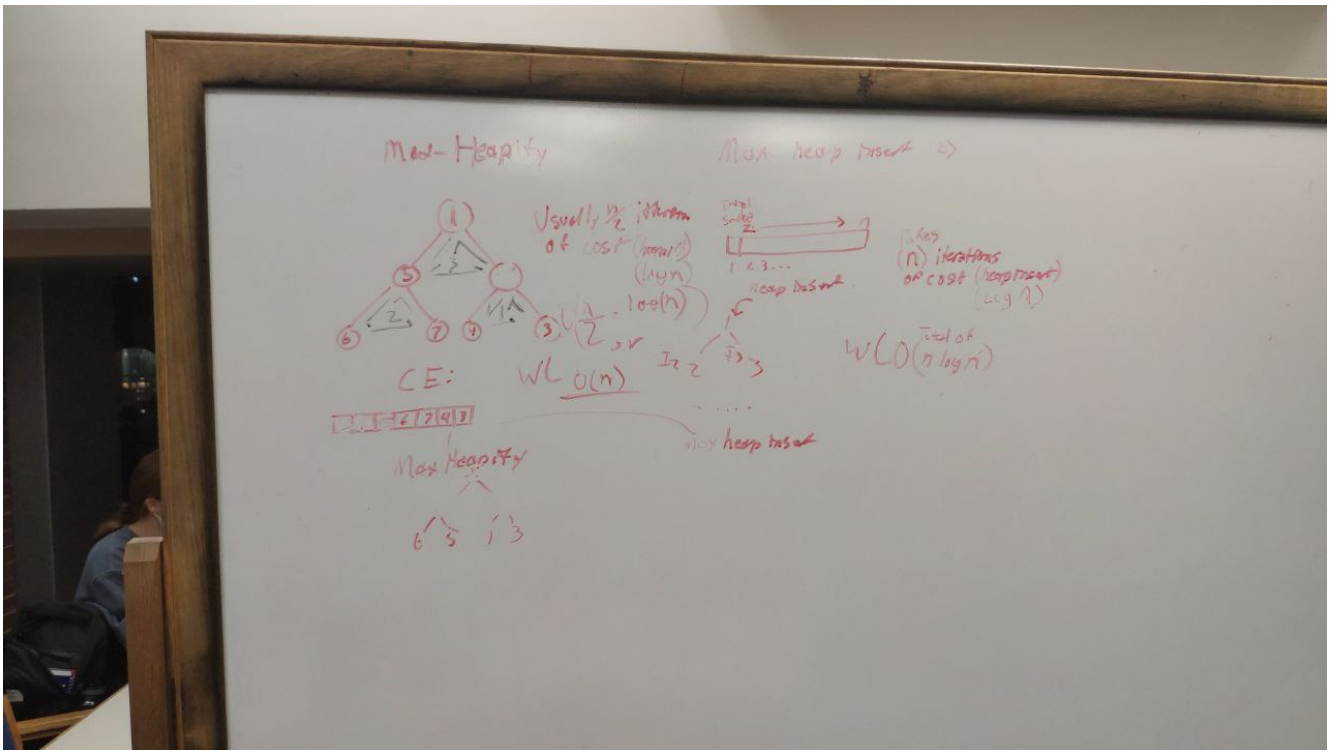
Consider the following alternative way to build a max-heap:

```
def buildmaxheap2(A):
    A.heapsize = 1
    for i in range(2,A.length+1):
        maxheapinsert(A,A[i])
```

Prove that buildmaxheap and buildmaxheap2 do not always create the same heap by providing a counterexample.

2. What is the worst case time complexity of buildmaxheap2? Explain your reasoning.

Soultions:

1. Both of these provide solutions that are maxheaps. Maxheapify, doing it using heapify on the n/2 number of nodes, whereas buildmaxheap2 takes the **Initialization** property being a list of size 1 is a max heap, and added each item from the list of size n into the max heap using maxheapinsert(logn) (**maintencence**). This algorithm then **terminates** when there are no more elements to add to the heap with a total time of n inserts costing logn each for O(nlogn). The other works differently, first making sure the "triangles" of nodes at the bottom first are following the maxheap property, then swapping up and down accordingly to make individual mini-triangles following the heap property, heaping up the max elements of the children, and down the minimum parent elements. Its possible to have them not match up if the element being inserted gets inserted on the opposite side of the tree in heapinsert (buildmaxheap2) compared to heapify which would have swapped that element in place. An example of an array that has this property is [10,14,19,21,23,31].


2. The worst case time complexity of buildmaxheap2 is nlogn, where each heapinsert takes log(n) time because it disrupts the heap property of the array, and this worst case scenario is performed on every item in the list (n times), making the worst case total runtime O(nlogn).