1)**Find Common Elements**

**How it could work:** Take all elements in Array1 and compare the entire array to the first element in Array2 then if there is a match print it else move on to the next element and try again and keep repeating the process



**Algorithm:**

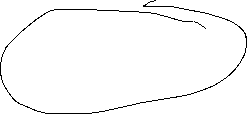
//Implements 2 arrays that contain integers and checks to see if there is a matching number between them then prints that number out



//input: integers > 0

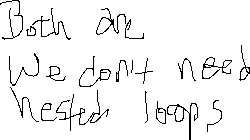
//Output: All integers that have a match in the other list

For (Array1[i]) {



For (Array2[i+1] {

If(Array1[i] == Array2[i+1] {



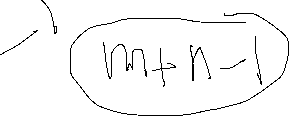
Print Array[i]

} Else {

continue

}

}

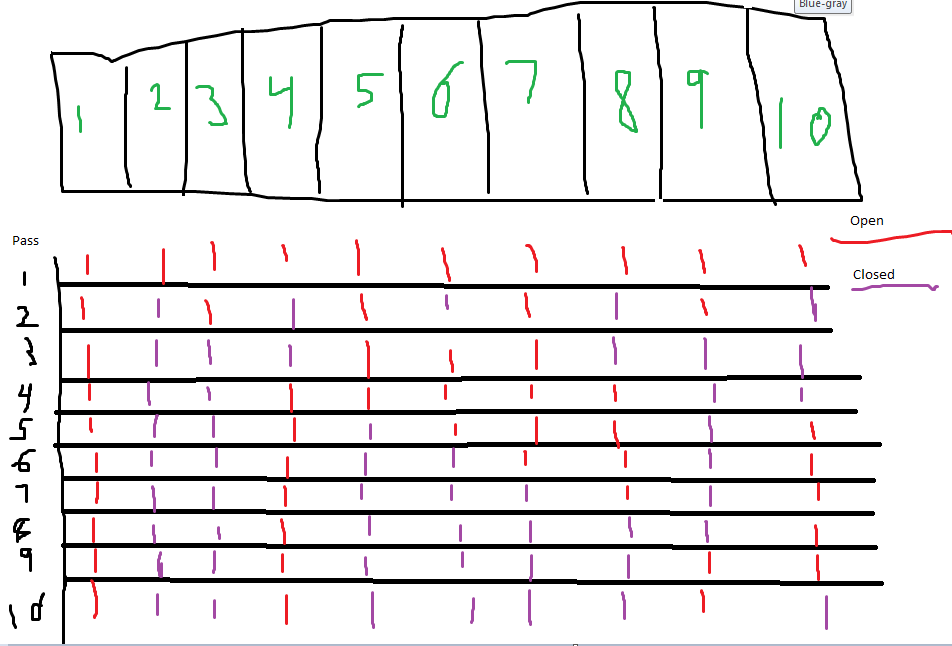


}

**Results:** With this implementation the efficiency would be O(n^2) as there are 2 for loops that iterate through every number to try and find matches as each for loop is an 0(n) performance.

2)**Locker Doors: How it could work:** Start with an Array of all (n) doors in the room, on each pass use a loop to open the doors related to that path, so on pass 1 open every door starting at 1, then on the second pass start back at door 1 and open/close every other door, keep repeating this until you have iterated through all doors (n) times

**Drawing:**



**Interesting things:** I notice due to the drawing only 1 and perfect squares are remaining open so according to this data I collected I’m going to change my thinking and instead of use a For loop I think just find the perfect squares and first locker and use that as my answer

**Algorithm:**

//Implements finding the square root of the number and adding 1 to find total lockers open and prints which doors are open

//Inputs: (n) number of positive integers



//Outputs: number of doors that are open after the (ith) pass

doorsIn = input(“How many doors”)

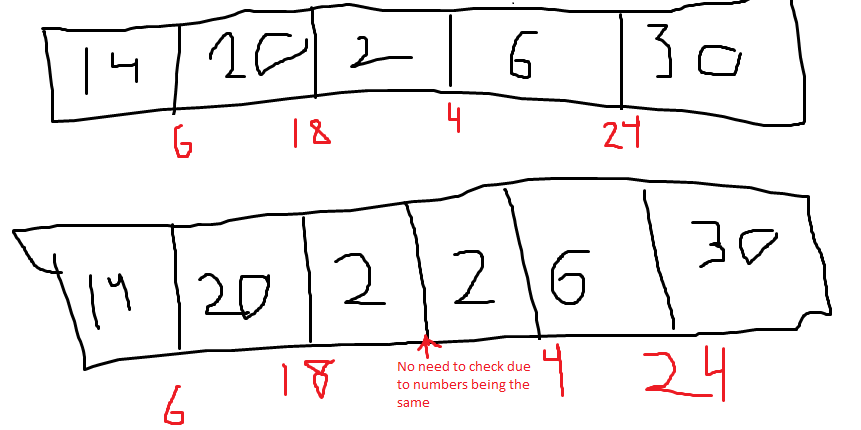
doorsopen = sqrt(doorsIn)

print(doorsopen)

3) **Min Distance:**

**How it could work:** To increase efficiency of the Algorithm you could implement a previous element tracker and current element tracker that remembers what or looks at what the last element you checked was to ensure you weren’t checking the same element again or one that is the same number

**Drawing:**



**Algorithm:**

//Implements 2 for loops that check the distance between elements in the array but also don’t check numbers that are the same

//Inputs: Array with (n) numbers

//Output: The smallest distance between two elements in the array

Prev = -1



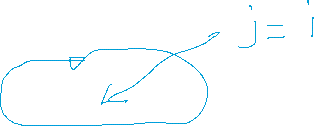
Current = 0

You only need the second loop to go from i to n-1. You don’t need to repeat covering elements you have covered previously.

You missed the efficiency… it’s

m + n -1

where m is number in first array and n is number in second array



For (I = 0 to n-1) do

For (j = 0 to n-1) do

If (Prev != -1 and ArrayD != ArrayD[Prev])

Dmin = ArrayD[i] – ArrayD[Prev]

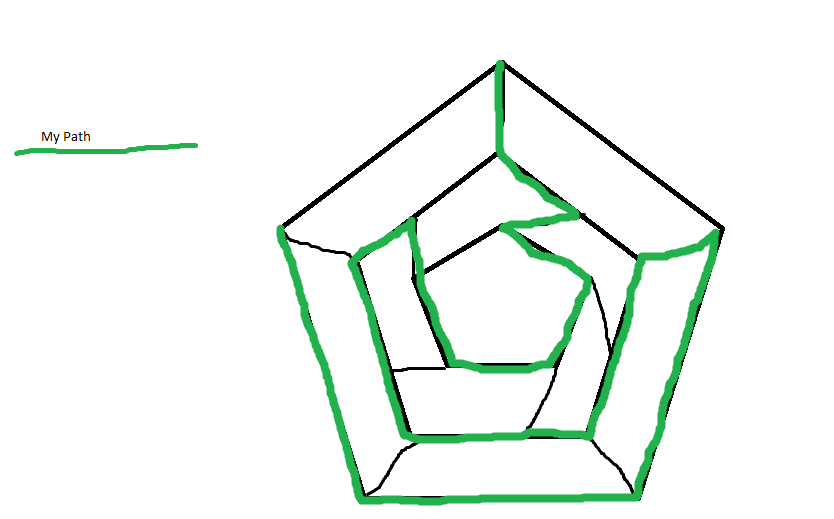
Prev = Current

Return Dmin

4) **Icosian Game:**

**How it might work:** Just at a first glance I notice all shapes inside are pentagons so that means I must hit 4 of the 5 vertices on the outside 2 pentagons and make it to the inner pentagon where I must then hit all 5 vertices and make it to a point I hadn’t hit on the second innermost pentagon and follow that out to a spot on the outermost pentagon that I haven’t hit yet.

**Drawing:**





**Thoughts:** Since I noticed right away the pattern that would have to be followed to ensure I touch all points it was relatively easy to touch almost all of the shape and leave myself a singular path that would allow me to make it to the end destination, The shape is treated like a map for the game snake in which you aren’t allowed to touch your own body or you lose so winding your way around the pentagons staying away from yourself is how you could solve it or just take the number of vertices on the shapes and only touch n-1 (where n is the number of vertices of the shape) of them, then move to the next level in and do the same thing. When you come to the Innermost layer touch all vertices and make your way out of the shape using the vertices you left untouched.