Recursion

So the challenge at hand is to do better than selection sort and better than bubble sort, and ideally not just marginally better but fundamentally better.

— Can we do better than something on the order of n^2? —

If no doors
  Return false
If number behind middle door
  Return true
Else if number < middle door
  Search left half
Else if number > middle door
  Search right half

It's kind of cyclically defined.

I claim this is an algorithm for search, and yet it seems like unfair that I'm using the verb search inside of the algorithm for search.

It's like an English way of defining a word by using the word.

But the problem is that, by definition, half as large
→ this isn't going to be a cyclical argument in the same way.

This approach, by using search within search is going to whittle the problem down and down until hopefully, one door or no doors remains.

Recursion is a programming technique whereby a function calls itself.

↓

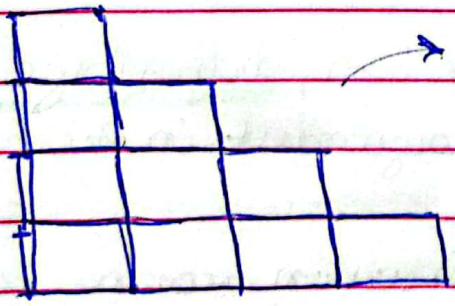
You can have a function call itself → Using the function's name inside of the function's implementation itself.

"Go back to line 3"
"
"
"Go back to line 3"

} In another pseudocode, or inducing a loop, a cycle. They are very mechanically,

Binary search algorithm for the phone book, it's just telling me to search the left half or search the right half.

→ It's an elegant way too of describing a problem by just having a function use itself to solve a smaller puzzle at hand.



→ How to implement this in C code?
~ left aligned pyramid ~

Example of iteration

$ vscode iteration.c

Print exactly this pyramid of height 4

```
#include <cs50.h>
#include <stdio.h>
void draw(int n);   (PROTOTYPE)
int main(void)
{
    int height = get_int("Height: ");
    draw(height);
}
```

How many bricks do I want to print?

C doesn't have draw functions...

```
void draw(int n)
{
    for(int i = 1; i <= n; i++)
    {
        for(int j = 0; j < i+1; j++)
        {
            printf("#");
        }
        printf("\n");
    }
}
```

or `(int i = 0; i < n; i++)` more conventional

$ make iteration
$ ./iteration

Height: 4

#
##
###
####

But I can think about implementing draw in a somewhat different way that's kind of clever.

How else could we think about this problem?

Well, this physical structure, these bricks, in some sense is a recursive structure, a structure that's defined in terms of itself. *

"What does a pyramid of height 4 look like?" you would point, of course, to this picture. "

But you could also kind of cleverly say to me, well, it's actually a pyramid of height 3 + 1 row. And there's that cyclical argument, right?

But we can kind of leverage this logic in code.

```
// include <cs50.h> *
// include <stdio.h> *
void draw(int n);
int main(void)
{
    int height = get-int("Height: ");

    draw(height);
}
```

→

```
void draw (int n)
{
    draw(n-1);
    .
    .
    .
```

(1º) if you ask me to draw a pyramid of height n, I'm going to be kind of a wise ass here and say, well, just draw a pyramid of n-1.

Ok. what happens after I print or draw a pyramid of height n-1 according to our structural definition a moment ago?

~ We need more row of bricks ~

```
    .
    {
    for (int i = 0; i < n; i++)
    {
        printf ("#");
    }
    printf ("\n");
    .
}
```

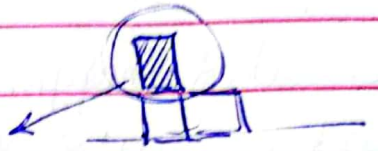
But this is kind of trippy now, because I've somehow boiled down the implementation of draw into printing a row after just drawing the thing above it.

↳ But this is problematic as is because in this case, my draw function, notice, is always going to call the draw function forever in some sense.

But ideally, when do I want this cyclical process

to stop?

"But ideally, when do I want this cyclical process to stop?" When do I want to NOT call draw anymore?



→ When n is 1, right? When I get to the top of the pyramid, when n is 1, or heck, when the pyramids all gone and n equals 0.

```
void draw(int n)
{
    if(n <= 0)
    {
        return;
    }

    draw(n-1);

    for(int i=0; i<n; i++)
    {
        ...

OR ...
```

→ super safe! <=

safety condition

```
void draw(int n)
{
    if (n != 0)
    {
        draw(n + 1); error.
        return;
    }

    draw(n - 1);

    for ...
```