(Code for selection sort.)

What is the complexity of the algorithm?

- The outer loop runs through all ~~a~~ n entries.
- The inner loop goes through ~~...~~ $n-1, n-2, \ldots, 2, 1, 0$ entries. Since

$$n - i = O(n)$$

for $i = 1, 2, \ldots, n$, the inner loop runs through $O(n)$ entries.

Because each entry in the outer loop does an $O(n)$-operation, and the outer loop has n entries, the algorithm is $O(n^2)$.

This is not ~~a~~ the fastest sorting algorithm.

(Bubble sort code.)

What's happening? How is the algorithm sorting the list? (Is it actually sorting?!)

It is sorting, and it is making swaps of adjacent entries. The algorithm performs a number of passes through the list. In the worst-case, the first pass puts the last element in place. The second pass puts the second-to-last element in place, and so on.

Bubblesort is also, therefore, $O(n^2)$.

Two better algorithms involve ideas of recursion.

Recursion

This is a powerful tool, and when used correctly makes code so easy to understand. It has the benefit of simplifying code. However, when done poorly it can break things. It can be confusing at first, so like with all challenging topics, stick with it!

Stated simply, recursion is when a function calls itself. To do recursion well, you need to think about two instances:

1. the base case,

2. the recursive case.

Just about every bad instance of recursive programming arises when one does not ~~cart~~ consider carefully 1 the base case.

The "hello world" example for recursion is factorial.

I'll give two styles of definition. Assume n is a positive integer.

Def 1. $n! = n \cdot (n-1) \cdot (n-2) \cdot \cdots \cdot 3 \cdot 2 \cdot 1$.

Def 2. $n! = n \cdot (n-1)!$ and $1! = 1$.

Both definitions are correct. The 2nd def. might seem less clear, but they are both precise and yield the same (actual) definition of factorial.

The second def is recursive, and it has 2 parts

① $1! = 1$ (base)

② $n! = n \cdot (n-1)!$ (recursive)

Here's one way to "uncover" the definition via a recursive def.

$1! = 1$ as defined.

$2! = 2 \cdot 1! = 2 \cdot 1 = 2$ using base case

$3! = 3 \cdot 2! = 3 \cdot 2 \cdot 1! = 3 \cdot 2 \cdot 1 = 6$

$4! = 4 \cdot 3! = 4 \cdot 3 \cdot 2! = 4 \cdot 3 \cdot 2 \cdot 1! = 4 \cdot 3 \cdot 2 \cdot 1 = 24.$

Let's look at coding up factorial using both definitions.

(factorial code)

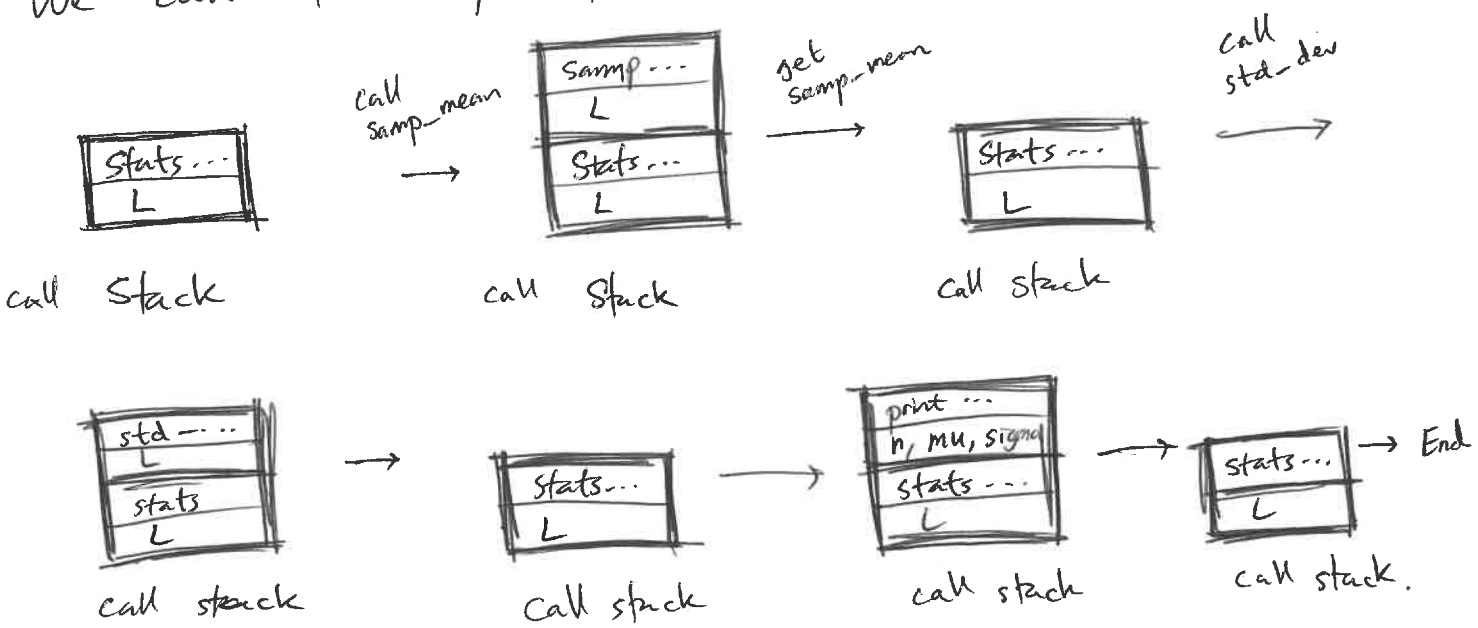
What would happen if we did not code up the base case?

It can be useful to understand some basics of the call stack when it comes to understanding recursion. Suppose we have the following Python function

```
def samp_mean(L):
    <code>

def std_dev(L):
    <code>

def print_report(n, mu, sigma):
    <code>

def stats_report(L):
```

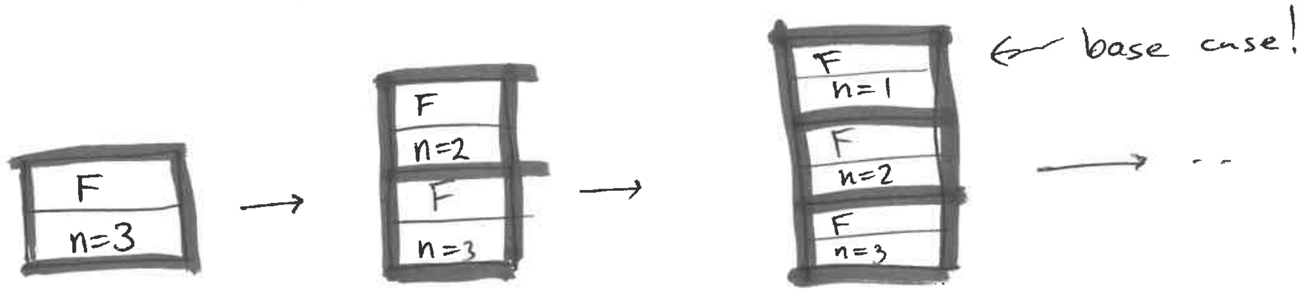
Call Stack code.

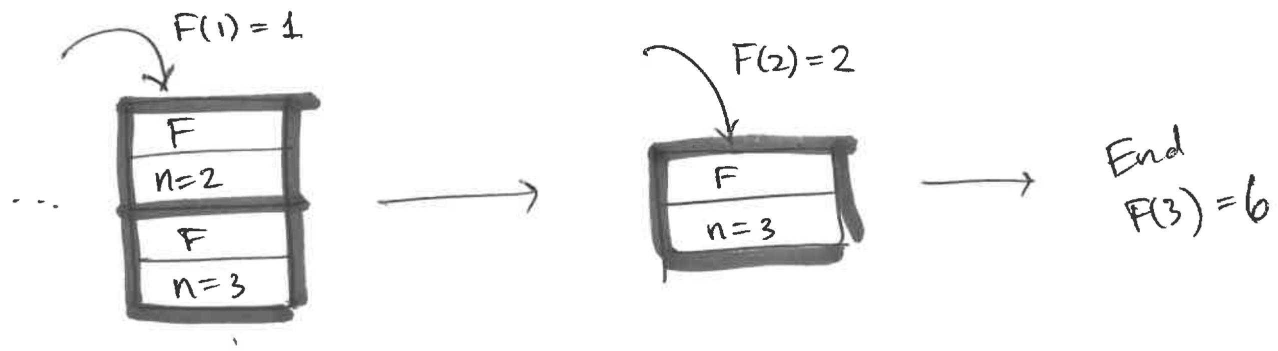
The top three functions are "subroutines" of stats_report. We call stats_report, then other functions are called:



Functions that recently enter the call stack, get off the stack sooner. (This is the Stack from before.) When a function pops off the stack, return to the previous just as we left it, except maybe with new results

Let's look at factorial2(3). Now called F(3).





Let's look at a different recursive function.
We'll look at a recursive function to determine if a string is a palindrome, that is, the string is the same if read from left to right or right to left.

(palindrome code)

Let's come back to binary search and rewrite that using a recursive function.

(binary search recursive)