

Recursion

Cal Teach Student Name(s):	Morgan Rae Reschenberg		
Mentor Teacher Name:	TBA	School/Room#:	TBA
Grade Level and Subject:	Computer Science, High School (9th-12th grade)		
Date & Time to Be Taught:	TBA		

Lesson Source(s):	<p>Guo, Philip. "Visualize Python, Java, JavaScript, TypeScript, and Ruby Code Execution." Python Tutor. N.p., n.d. Web. 04 Apr. 2017. http://pythontutor.com/visualize.html#mode=edit.</p> <p>Katz, Randy. Boser, Bernhard. "Course Introduction". UC Berkeley. CS61C: Great Ideas in Computer Architecture, 30 August 2016. Web. 06 April 2017. http://inst.eecs.berkeley.edu/~cs61c/fa16/lec/01/L01%20Introduction.pdf.</p> <p>"Pair Programming in the Classroom." Khan Academy. Khan Academy, n.d. Web. 04 Apr. 2017. https://www.khanacademy.org/resources/out-of-school-time-programs/teaching-computing/a/pair-programming-in-the-classroom. - Paired programming slides (Reference)</p>
Focus/Essential Question:	How are iteration and recursion related? How do we use recursion? Where did the concept of recursion originate and why is it so fundamental in programming?
Student Learning Objectives:	Students will know what recursive functions are, how to write them, and why they are important in the field of Computer Science. Students will also understand how elements like recursion and frame diagramming have developed overtime and persisted into Computer Science as we know it today.
Content Standards:	[Currently, there are no state standards or national standards for Computer Science because it is not a mandated public school subject]
Student Prior Knowledge:	<ul style="list-style-type: none"> - Iteration (for, while loops) - Translation of basic algebraic functions into Python - Functions (how to define functions in Python, function execution behaviour, how to call a function) - Frames (new frames are generated when functions are executed/called)
Lesson Agenda for Your Students:	<ul style="list-style-type: none"> - Warm up: Math to Python - Iteration, Recursion Activity - Recursion (Lecture) - Building Fibonacci

	- Exit Ticket: Rehashing the Warm up Problems
Lesson Rationale:	Recursion is a fundamental aspect of Computer Science; it makes writing code easier and faster while also making code simpler and easier to read. The logic exercised in recursion is also one of mathematical significance when considering the computation of series and sequences. As important as recursion is today, it is not a new concept; Computer Science as a field has developed over many centuries. Constant change and technical development drive the field and valuable contributions are made from all kinds of Computer Scientists--including women and minorities.

Materials and Technology List:	Computers/Student lab stations equipped with Python 3+, Sublime Text or equivalent, and GitBash or terminal equivalent
Preparation Tasks:	Distribute student files via classroom dropbox
Safety Concerns:	No safety concerns foreseen

Recursion, Iteration, and Repeating History

	Evaluate: <i>Observe and adjust your lesson as you teach.</i>
Engage: <i>Activities that engage students' interest and build connections to their lives and prior knowledge.</i>	<i>Previous Experience and Baseline Learning</i>
<p>After completing their Math-to-Python warm up in the first five minutes of class (see attached files), students will begin a review of iteration and be introduced to recursion through the following interactive activity.</p> <p>Time: <u>10 minutes</u></p>	<ul style="list-style-type: none"> - Students who are unfamiliar with, or struggling with python syntax, should be directed to the “defining a function” visual and/or their notes from last lesson - Advanced students may use the last problem to explore functions-within-functions; all students should be encouraged to attempt this approach!
Explore: <i>Hands-on tasks designed to explore ideas and to develop skills together.</i>	<i>Focus, Involvement, Collaboration, Results, and Recording</i>
<p>Select two sets of three student volunteers for six student volunteers total. Give each group of volunteers one of the attached function sheets (one group should have the recursive function and the other should have the iterative version) and a set of variable cards. Students should be instructed to work as a group to perform the action on the card in front of the class one group at a time. Confer</p>	<ul style="list-style-type: none"> - Verify each demonstration group understands what their function does and what their roles are. - If group members argue over who has which role, pre-assign

with students if confusion arises over who is to perform which function and when. While the demonstration groups are preparing, the rest of the class can open a new Sublime document.

After each group demonstrates their function, encourage the rest of the class to discuss with their table partners what they observed. Students should consider whether or not the function fits with what they know about iteration.

Thank you [group members]! [To class] With your table partners, talk about what we just saw. Do you think the function demonstrated modeled iteration? Why or why not? Take a few minutes to discuss and write up a few lines of pseudocode. Remember, pseudocode isn't Python, it's a short and to-the-point explanation of how the function ran, what arguments it took in, and what argument it produced at the end.

If you finish writing your pseudocode early, go ahead and convert it to Python and try out some test cases to make sure your implementation matches what we have in our example. Remember, there are lots of ways to code, so if your solution looks different from your partners, don't worry! Write your test cases and make sure your code has the same behaviour.

After allowing students to discuss and generate pseudocode, take a volunteer to read out their pseudocode or share it on the projector. Confirm all students have the same understanding of the program. Students should recognise that the program:

- was iterative
- added three to a running total each cycle
- employed either a **for** or **while** loop
- stopped when it reached the **count** given as an argument
- returned the final total

Show the Python Tutor [demonstration](#) to assist students in visualising the frames.

How many frames do we see in this visualisation? How can we tell how many frames there are? Does the loop generate additional frames? How can we relate our table to the amount of frames we have?

Students should understand this function call sequence has two frames--the global frame and the local frame of the function itself. Students should recall from prior lessons on functions that every function call generates a new frame and that every execution has a global frame. There is only one function call, so there is one frame generated in addition to the global frame. In relation to the table, the row can be thought of as the local function frame and the table itself can be thought of as the global frame.

Allow the second group to do their demonstration and, again, encourage student discussion.

roles while picking students ("I need a volunteer to play the OS in this group", "The next volunteer will be the function!", etc.)

- Students may be confused by what constitutes a "row" in the table since in the first example, the Function Body calculates multiple values in one frame
- To help students visualise this, draw thicker lines around the program frames
- Students who finish discussing early or who write their pseudocode quickly should convert their pseudocode to real python code and write test cases to confirm their code is correct.
- Emphasise variation in programming; students should be encouraged to find *their* solution not just the one their partner has.
- If students cannot yet discern how the table relates to the frame count, allow the second group to do their demonstration and come back to the question after students have a chance to compare the two Python Tutor simulations and the two tables.

<p>Thank you [group members]! Discuss with your table partners what we saw in this demonstration. What is different about the tables generated by each group? Which person/part of the program played the biggest role in the first group? What about the second? What was our final result in the first case? What about the second?</p> <p>If you finish discussing early, go ahead and try to write out the pseudocode for this program. This is a new kind of function, so don't worry if you struggle with writing out the pseudocode--we're going to go over this in more depth today!</p> <p>Students should understand the first table has one row while the second has a new row for each “adding” sequence. In the first group, the function body did most of the work whereas in the second group, the function body and OS had fairly even roles. Both functions have the same final result (12) and do not generate any side effects (they are both <i>pure functions</i>).</p> <p>Students should be familiar with the concept of frames from their work with functions. Note that each “row” of the table creates a new frame/function call. In the second example, the table has a N rows (where N is the index we'd like to return from the series). Show the Python Tutor demonstration to help with visualising frames.</p> <p>Students who finish discussing early (or students who have prior experience with recursion) should write both the pseudocode and Python code for this function. They should also write test cases to confirm their solution matches the example given on the board.</p> <p>If time permits, give a few students the opportunity to present their pseudocode on the board or on the projector to the class. Have the class discuss the pseudocode and offer suggestions to help out struggling students. <u>Students should recognise that this format is similar to iteration but takes more frames where there is less work done per frame.</u></p> <p>[Transition to Elaborate/Lecture section] Let's take a closer look at how the structure of each program differs. Pull up your lecture slides if you haven't already. Keep your Sublime document open and, as we go through the lecture, see if you can convert your pseudocode for this function into Python code</p> <p>Time: <u>40 minutes</u></p>	<ul style="list-style-type: none"> - If students have trouble distinguishing between the two programs, run the Python Tutor simulation first. Frames in this context may be more familiar/easier to understand than the table visualisation depending on the prior experience of the students.
<p>Explain: <i>Students explain the phenomena they explored and discuss their different ideas and perspectives.</i></p>	<p><i>Participation, Reporting, Debating, and Evidence-Based Reasoning</i></p>
<p><u>[This section should be taught <i>after</i> the “Elaborate” section]</u></p> <p>After being armed with more definitional, contextual knowledge about recursion, students should be ready to apply the knowledge themselves.</p>	<ul style="list-style-type: none"> - Students with higher-level mathematical knowledge should be familiar with the Fibonacci numbers sequence. Students without this

<p>Students should be given the following definition of a Fibonacci number where $F_0 = F_1 = 1$:</p> $F_n = F_{n-1} + F_{n-2}$ <p>In their table groups, students should implement first an iterative function to calculate the Nth Fibonacci number and, after successfully doing so, modify their existing program to become recursive.</p> <p>Each group should operate according to paired programming conventions. Review these with the class before starting the activity:</p> <ul style="list-style-type: none"> - “<u>Driver</u>”: has control of the mouse and keyboard; physically writes the code and raises concerns to the navigator about code behaviour. The driver should use correct syntax and explain what is being written as it is being written. The driver makes stylistic programming choices (ie. multiple/single line declaration, for vs. while loops, etc.) - “<u>Navigator</u>”: thinks about the “big picture” and communicates how the code should run/what behaviour the code should have. The navigator talks through the problem presented in the code and discusses a way to solve it. The navigator should <i>not</i> code the solution themselves. <p>Students should switch roles every 10-20 minutes depending on the level of communication in the classroom and the comfort level the students demonstrate with the material. Students who are more unfamiliar may benefit from longer intervals to become comfortable with the algorithm or programming environment. Students who demonstrate a high level of understanding should be challenged with shorter intervals to develop technical communication skills.</p> <p>At each interval switch, the Navigator should debrief the new Navigator on problems they were watching, ideas they had, and problems they weren’t quite sure how to solve. Likewise, the Driver should inform the new Driver of syntax they were confused on and coding comments that need to be resolved. This roleshift will help students understand all aspects of the programming process while working with their partner to expand their knowledge of recursive functions specifically.</p> <p>Students who finish the problem early should write test cases for their program before submitting it to the autograder. Students may also be challenged to develop their own sequence/series on paper. Then, students should take their invented series and devise a recursive function to generate the numbers they came up with.</p> <p>Time: <u>40 minutes</u></p>	<p>background should be given an introduction to this topic within the scope of this course--that is, that Fibonacci numbers are significant mathematically and they are naturally occurring.</p> <ul style="list-style-type: none"> - In this lesson plan, it is assumed that students have worked with fibonacci numbers in their math classes before and need only a definitional reminder. - Paired programming is implemented here to foster a less competitive CS environment and to assist students in developing technical communication skills - Students should be paired with partners close to or just above their skill levels to prevent a “just let me do it” attitude of frustration. - See resources for more information on paired programming and how to implement it in the classroom if not standard already
<p>Elaborate: <i>Teacher-stimulated application and clarification of concepts, skills, attitudes, processes or terminology.</i></p>	<p><i>Demonstrated Understanding, Use of Skills, and Other Applications</i></p>
<p>Begin powerpoint lecture, see attached slides below. Each pink slide with white text is an opportunity for student discussion and for questions on the previous example/topic.</p> <p>Review the group activity students have just completed. Using the tables</p>	<ul style="list-style-type: none"> - Give students plenty of time to discuss with their table partners during open-ended, question slides - Each of these slides should

<p>diagrammed by students, walk through the process of constructing the Python code for the recursive function. Give students the opportunity to copy down the “Anatomy of a Recursive Function” into their notebooks for reference during the next activity.</p> <p>Compare the tables generated by students to the table Ada Lovelace used in her Bernoulli numbers program. Using the same vocabulary, discuss the concept of each line of the table as a frame. Reiterate the recursive nature of the machine itself (programs were run on cards, cards containing processes to be repeated had to be re-fed into the machine) and the recursive nature of the function (each time the process is called, new variables ($V_{10} - 1$) are passed in as markers for how many <i>more</i> calls the program must make).</p> <p>Introduce the Babbage Machine (the Analytical Engine) within the context of “early computing machines”. Emphasise how the concept students are engaging with is a fundamental aspect of computer science; complex functions allow programmers today to write shorter, cleaner code and use developments made by programmers before them. We don’t have to reinvent the wheel every time we want to repeat an action we’ve already done. Similar to Lovelace’s approach, we don’t have to re-program how to calculate the Nth Bernoulli number because we have a function which does that.</p> <p>Technological advancements and human ingenuity drive the field of Computer Science. Within students’ lifetimes, they’re likely to come across bigger changes and maybe invent some new, Earth-shattering technologies themselves!</p> <p>Time: <u>40 minutes</u></p>	<p>serve as a mini-break and opportunity to redirect the class; lectures can make it hard for students to maintain focus. If students need more breaks toward the end, interject questions about what they think technology will look like in the near future.</p> <ul style="list-style-type: none"> - Watch for students struggling with making the “recursive leap of faith”--trusting that a function is allowed to call itself. It may be helpful to pull up the Python Tutor demonstration for each “partial function” as students work toward a complete solution - For more practise (similar to the exit ticket) encourage students to rewrite their warm up problems recursively
<p>Evaluate:</p>	
<p>Students will complete an exit ticket to demonstrate their understanding of recursive functions.</p> <p>Let’s think back to the warm up problems we did, specifically the bonus problem. We wanted to create a function which gave us Nth number in a sequence of numbers increasing by three. If you had time to attempt the bonus problem during the warm up, you probably used iteration. Now, I want you to write the recursive version.</p> <p>Use your cheat sheet from lecture if you need help remembering what the structure of a recursive function includes. Turn in your files to my dropbox before you leave.</p>	<ul style="list-style-type: none"> - Students who did not have time to attempt the bonus problem during warm up time may benefit from first writing out the iterative version. - Students may need clarification on the wording of the problem; if necessary, select a student volunteer to help write out the sequence you’re using. Run through an example with $N = 3$.

Math to Python Warm-up:

Convert the following algebra problems into python functions. Define each function individually. After you finish, write two tests for each function. Before you run your program, write what you expect your results to be for each case. Verify your solution is accurate.

- 1.) Create an adder. This function should take in one argument and return the sum of that argument and the number 7
- 2.) Create a multiplier. This function should take in one argument and return the product of that number and 10
- 3.) Create a function which takes one argument and returns the product of 10 and that number plus 7.

BONUS: Create a function which gives us the Nth number in the sequence of numbers increasing by three. This function should take in one argument N and return the Nth number in the sequence.

Possible test cases:

- 1.) `adder(3) → 10`
- 2.) `adder(20) → 27`
- 3.) `adder(-100) → -93`
- 4.) `adder(adder(12)) → 26`
- 5.) `mult(100) → 1000`
- 6.) `mult(-7) → -70`
- 7.) `mult(3 * mult(-40)) → -12000`
- 8.) `comp(13) → 200`
- 9.) `comp(-27) → -200`
- 10.) `comp(comp(10)) → 1770`

Function Sheet (1)

- In your group of three, select one person to act as the Operating System, another as the Function Body, and the last as the Return Statement. Create a table on the board with three columns (OS, Function Body, Return Value). Each person should perform the following actions:
 - Operating System: You've been given a variable card. Write the name and value of this variable in your column on the board. Pass your variable card to the Function Body at the beginning of the demonstration.
 - Function Body: Initialise a variable `<i>` with value 0. `<i>` will be your running total. On the board, compute the `<count>`th number in the sequence of numbers increasing by 3. Show your work for `i = 0, 1, ..., <count>`. Once you've finished, write your final value of `<i>` on the back of the function card next to "return" and pass the card to Return.
 - Return: After Function Body has finished computing the `<count>`th value in the sequence, take the variable card handed to you and write the value in the appropriate column.

Expected finished table:

Operating System	Function Body	Return Value
<code>count = 4</code>	<code>i = 0</code> <code>i = 0 + 3 → 3</code> <code>i = 3 + 3 → 6</code> <code>i = 6 + 3 → 9</code> <code>i = 9 + 3 → 12</code>	<code>return 12</code>

Code for [demonstration](#): (use linked python tutor for students who need more help visualising the execution of the code)

```
def countByThree(count):
```



```
total = 0
for i in range(0, count):
    total += 3
return total
```

Function Sheet (2):

- In your group of three, select one person to act as the Operating System, another as the Function Body, and the last as the Return Statement. Create a table on the board with three columns (OS, Function Body, Return Value). Each person should perform the following actions:
 - Operating System: You've been given two variable cards. Write the names and values of these variables in your column on the board. Pass your variable cards to the Function Body at the beginning of the demonstration.
 - Function Body: You've been passed two variable cards by OS (<count> and <current>).
 - If <count> is not equal to zero, add three to <current> and subtract 1 from <count>. Hand your <current> and <count> values to OS.
 - If <count> is passed in as zero, perform no computation. Hand your <current> and <count> values to Return.
 - Return: After Function Body has finished computing <current>, return <current>.
 - Operating System: take the new values of <current> and <count> from Function Body. Hand them back to Function Body simulating a new call with these arguments.

Expected finished table:

Operating System	Function Body	Return Value
count = 4 current = 0	count = count - 1 → count = 3 current = current + 3 → current = 3	
count = 3 current = 3	count = count - 1 → count = 2 current = current + 3 → current = 6	
count = 2 current = 6	count = count - 1 → count = 1 current = current + 3 → current = 9	
count = 1 current = 9	count = count - 1 → count = 0 current = current + 3 → current = 12	
count = 0 current = 12		return 12

Code for [demonstration](#): (use linked python tutor for students who need more help visualising the execution of the code)

```
def countByThree(current, count):
    if count == 0:
        return current
    else:
        return countByThree(current + 3, count - 1)
```


Exit Ticket

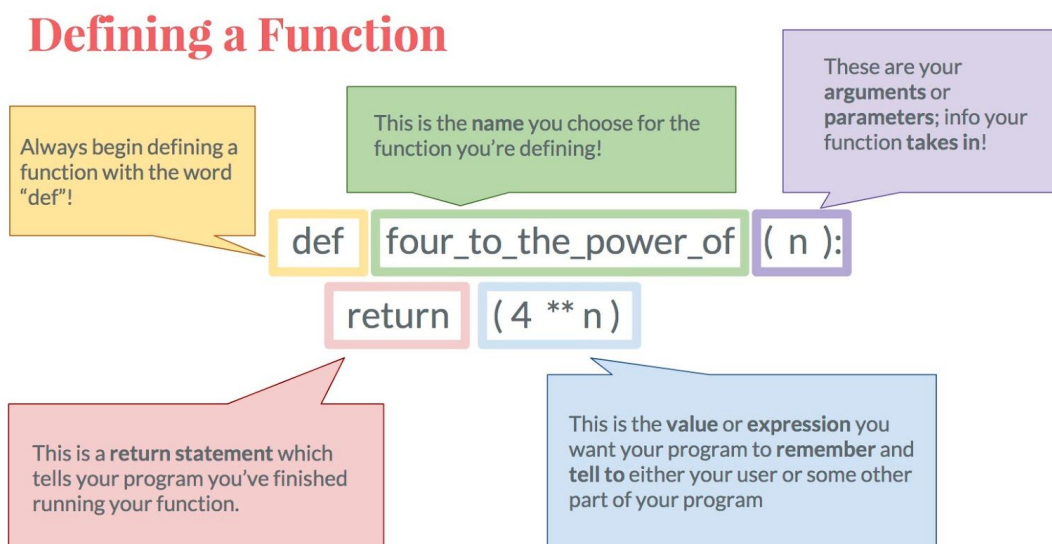
Create a function which returns the Nth number in a sequence of numbers starting at some number which is a multiple of three and increasing by three each time. Your function should take in two arguments <start> and <N>.

EXTRA CREDIT: Write an error-catch to your function such that if a user inputs a number for <start> that is NOT a multiple of three, your function returns “Error: <start> is ____ but should be a multiple of three”

Solution

```
def threes(start, N):  
    //EC:  
    // if start % 3 != 0:  
    //     print("Error! <start> is " + start + " but should be a multiple of three!")  
    //     return None  
    if N == 0:  
        return start  
    else:  
        start += 3  
        N -= 1  
        return threes(start, N)
```

“Defining a Function” reference sheet:



Recursion and Repeating History

Computer Science

Review: Group 1

- 2 frames (1 local and 1 global frame)
- Returned 12 to the user/callee
- Computed everything within one function
- Used a looping structure

```
def countByThree(count):  
    total = 0  
    for i in range(0, count):  
        total += 3  
    return total
```

Self Check: What is the name of this function? How many arguments does it require?

Operating System	Function Body	Return Value
count = 4	i = 0 i = 0 + 3 → 3 i = 3 + 3 → 6 i = 6 + 3 → 9 i = 9 + 3 → 12	return 12

Review: Group 2

- 6 frames (5 local and 1 global frame)
- Returned 12 to the user/callee
- Used several function calls to compute the final value
- Each function call does a very small amount of work

Self Check: How many arguments does this function require?

Operating System	Function Body	Return Value
count = 4 current = 0	count = count - 1 → count = 3 current = current + 3 → current = 3	
count = 3 current = 3	count = count - 1 → count = 2 current = current + 3 → current = 6	
count = 2 current = 6	count = count - 1 → count = 1 current = current + 3 → current = 9	
count = 1 current = 9	count = count - 1 → count = 0 current = current + 3 → current = 12	
count = 0 current = 12		return 12

What might this code look like?

Let's build this function!

- For the function demonstrated by Group 1, we were able to build the function by looking at just one row of the table. Let's try that strategy here:

Operating System	Function Body	Return Value
count = 4 current = 0	count = count - 1 → count = 3 current = current + 3 → current = 3	

- What is the operation being performed on **count** by the function body?
- What is the operation being performed on **current**?
- Is anything returned?

Row 1 Function

- **count** is changing by a factor of -1
 $\text{count} = \text{count} - 1$
- **current** is changing by a factor of 3
 $\text{current} = \text{current} + 3$
- Given the passed in arguments are 4 and 0 as count and current respectively, how can we write this function?

```
def mystery(current, count):  
    count = count - 1  
    current = current + 3
```

What are we returning?

Operating System	Function Body	Return Value
count = 4 current = 0	count = count - 1 \rightarrow count = 3 current = current + 3 \rightarrow current = 3	

Row 2 Function

- Now that we've defined our function for row one, let's see if we can define a function for row two
- We're applying the same operations as in row one (they're just being applied to inputs of different values)
- Because our program works with *variables* though, our original function is still accurate.

```
def mystery(current, count):  
    count = count - 1  
    current = current + 3
```

We're still not returning anything...

count = 3 current = 3	count = count - 1 → count = 2 current = current + 3 → current = 6	
--------------------------	--	--

Does this pattern hold for the rest of the rows? Why? Why not?

Over and over and over and... wait!

- This pattern holds for all cases EXCEPT the last row!
- In the last row, we perform no computation but finally return a value!
- Think about it! What condition is met in the last row that allows us to finally return a value?

```
def finalRow(current, count):  
    return current
```

count = 0 current = 12		return 12
---------------------------	--	-----------

What does it *actually* do?

Now that we've simplified our table into functions, we can write our table as a series of function calls. Let's give our functions names that represent their behaviour:

```
def mystery(current, count):  
    count = count - 1  
    current = current + 3
```



```
def compute(current, count):  
    count = count - 1  
    current = current + 3
```

```
def finalRow(current, count):  
    return current
```



```
def end(current, count):  
    return current
```

Multiple Function Calls

Time to simplify! Remember, each row can be written as a call to **compute** while the final row is written as a call to **end**:

`compute(0, 4)`

`compute(3, 3)`

`compute(6, 2)`

`compute(9, 1)`

`compute(12, 0)`

`end(12)`

Operating System	Function Body	Return Value
count = 4 current = 0	count = count - 1 → count = 3 current = current + 3 → current = 3	
count = 3 current = 3	count = count - 1 → count = 2 current = current + 3 → current = 6	
count = 2 current = 6	count = count - 1 → count = 1 current = current + 3 → current = 9	
count = 1 current = 9	count = count - 1 → count = 0 current = current + 3 → current = 12	
count = 0 current = 12		return 12

What do you notice about our
list of function calls? What
about the arguments?
Is this a “good” program?

Repeat Repeat Repeat Stuff

Our function call list is very repetitive! Also, each time we call **compute** we've already found its arguments in the previous call! We're doing a lot of extra work.

Take the first two calls for example:

```
compute(0, 4)
```

```
// current = 3
```

```
// count = 3
```

```
// we find these values, but we don't return anything here, so we  
can't use them!
```

```
compute(3, 3)
```

```
// our next function call uses these variables!
```

```
def compute(current, count):  
    count = count - 1  
    current = current + 3
```

Sharing is Caring

We know that we're computing the arguments to the next call in the previous call; How can we pass this information along?

We can create a recursive function; a function which calls itself!

```
def compute(current, count):  
    current = current + 3  
    count = count - 1  
    compute(current, count)
```

Instead of having our OS handle multiple calls to compute, we'll do what we did in the activity and have our function body call itself!


```
def compute(current, count):  
    current = current + 3  
    count = count - 1  
    compute(current, count)
```

If we try to run this code as-is, what problem will we run into? How can we fix it?

HINT: Try drawing out a tabular representation of this function; does it match the table from the example? Why or why not?

Again and again and again and again and...

Right now, our code never returns anything and also never stops running! We have no conditional to tell it when to stop computing more values of **current** and **count**.

We have one more element of our table that we haven't incorporated into our function, however: our **end** case!

```
def end(current, count):  
    return current
```

When should we call this end case? What condition in our table makes our algorithm end? [If you were in the demonstration group, what did the pseudocode say?]

... and it's over!

Our termination case is when **count** is equal to zero:

```
if count == 0
```

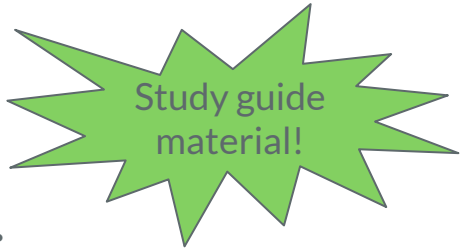
When this case is reached, we should call our **end** function. This is what our combined function looks like so far. Remember we *only* want to call our **end** function if our condition is reached, otherwise we want to continue computing.

```
def compute(current, count):  
    if count == 0:  
        return current  
    else:  
        count = count - 1  
        current = current + 3  
        return compute(current, count)
```

```
def compute(current, count):  
    if count == 0:  
        return current  
    else:  
        count = count - 1  
        current = current + 3  
        return compute(current, count)
```

Try it yourself! Does our function give the expected result when called with the arguments from our previous example? Write out your box diagram or table diagram and discuss your solution with your table partner.

Anatomy of a Recursive Function



Study guide
material!

```
def recursiveFunction(finalResult, timesToRecurse):
```

```
    if timesToRecurse == 0:  
        return finalResult
```

This is what's called the base case! It contains the conditional for when our program should stop recursing and returns our final result

```
    else:  
        finalResult = applyComputation(finalResult)  
        timesToRecurse = timesToRecurse - 1  
        return recursiveFunction(finalResult, timesToRecurse)
```

This is our recursive case! It applies some operation to our carried-through final result and also computes the arguments for our next function call. It returns the results of that function call so, in the end, our base-case result is returned to the user or callee.

Table your frustrations

Today we've been using tables to help us visualise how our programs compute values and create frames during execution. This strategy is one utilised by many computer scientists--in fact, when we run our [Python Tutor simulation](#), it creates and stacks frames in almost the exact same way!

Take a look at the following table; what do you notice about it? Talk with your partner as you try to find two ways the table is similar to what we've been doing and two ways it is different.

Challenge: What kind of program does the table describe?

Number of Operation.	Nature of Operation.	Variables acted upon.	Variables receiving results.	Indication of change in the value on any Variable.	Statement of Results.	Data.			Working Variables.											
						$1V_1$	$1V_2$	$1V_3$	$0V_4$	$0V_5$	$0V_6$	$0V_7$	$0V_8$	$0V_9$	$0V_{10}$	$0V_{11}$	$0V_{12}$	$0V_{13}$		
						\bigcirc	\bigcirc	\bigcirc	\bigcirc	\bigcirc	\bigcirc	\bigcirc	\bigcirc	\bigcirc	\bigcirc	\bigcirc	\bigcirc	\bigcirc		
						0	0	0	0	0	0	0	0	0	0	0	0	0	0	
						1	2	4	0	0	0	0	0	0	0	0	0	0	0	
						1	2	n												
1	\times	$1V_2 \times 1V_3$	$1V_4, 1V_5, 1V_6$	$\left\{ \begin{array}{l} 1V_2 = 1V_2 \\ 1V_3 = 1V_3 \\ 1V_4 = 2V_3 \\ 1V_5 = 1V_1 \\ 1V_6 = 2V_5 \end{array} \right.$	$= 2n$	2	n	$2n$	$2n$	$2n$									
2	$-$	$1V_4 - 1V_1$	$2V_4$	$\left\{ \begin{array}{l} 1V_4 = 2V_3 \\ 1V_5 = 1V_1 \\ 1V_6 = 2V_5 \end{array} \right.$	$= 2n - 1$	1	$2n - 1$											
3	$+$	$1V_5 + 1V_1$	$2V_5$	$\left\{ \begin{array}{l} 1V_5 = 2V_5 \\ 1V_6 = 1V_1 \\ 1V_7 = 2V_6 \end{array} \right.$	$= 2n + 1$	1	$2n + 1$										
4	$+$	$2V_5 + 2V_4$	$1V_{11}$	$\left\{ \begin{array}{l} 2V_5 = 0V_5 \\ 2V_6 = 0V_4 \end{array} \right.$	$= \frac{2n - 1}{2n + 1}$	0	0							$\frac{2n - 1}{2n + 1}$			
5	$+$	$1V_{11} + 1V_2$	$2V_{11}$	$\left\{ \begin{array}{l} 1V_{11} = 2V_{11} \\ 1V_2 = 1V_2 \end{array} \right.$	$= \frac{1}{2} \cdot \frac{2n - 1}{2n + 1}$	2	...									$\frac{1}{2} \cdot \frac{2n - 1}{2n + 1}$			
6	$-$	$0V_{12} - 2V_{11}$	$1V_{12}$	$\left\{ \begin{array}{l} 2V_{11} = 0V_{11} \\ 0V_{12} = 1V_{12} \end{array} \right.$	$= -\frac{1}{2} \cdot \frac{2n - 1}{2n + 1} = A_0$									0			
7	$-$	$1V_2 - 1V_1$	$1V_{10}$	$\left\{ \begin{array}{l} 1V_2 = 1V_2 \\ 1V_1 = 1V_1 \end{array} \right.$	$= n - 1 (= 3)$	1	...	n								n - 1			$-\frac{1}{2} \cdot \frac{2n - 1}{2n + 1} = A_0$	

Talk with your partner as you try to find two ways the table is similar to what we've been doing and two ways it is different. Challenge: What kind of program does the table describe? (for a larger image, check your dropbox!)

Bernoulli Numbers & Tabular Programming

- This table describes a program to compute Bernoulli numbers (if you've done **Taylor and MacLaurin series expansions**, you've seen these before!)
- Just like the tables we've been using, each row in the table represents a frame. Operations performed in each frame are sorted by which variables they're applied on, and which variables receive the results of computation.

How does this program run? Is it iterative? Is it recursive? How do you know?

Hint: Take a look at the “Working Variables” section--what pattern do you find in variable V_{10} ?

Out with the old and in with the... old?

- This technique for writing and analysing programs has existed for hundreds of years!
- The Bernoulli numbers program you have in front of you was published in 1843 by Ada Lovelace--a female Mathematician-turned-Computer-Scientist considered by many to be one of the most influential historical figures in the field of Computer Science.

Computer Science is not a field created by the dot-com boom or the tech industries of today; ideas and applications of computational thinking have been around for centuries. YOU are a part of a longstanding line of creative thinkers finding technical solutions to modern problems!

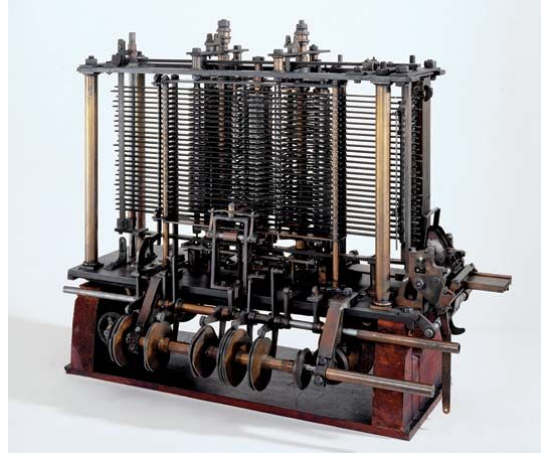
Computers: From Gears to GPU's

- But, wait! Did Ada Lovelace write this program to be run on machines like the one we have today?
- Computers have changed form and function as humanity has uncovered new technological advances.
 - Lovelace's program was built to be run on a mechanical computer (think levers, gears, rotors not transistors and CPU's!) conceptualised by Charles Babbage.
 - Babbage's Analytical Engine could perform algebraic operations routinely and accurately--programs were written to exploit this capability.
- Most math builds off of simple algebraic operations, hence the ability for more complex programs to be created from a small set of instructions (in Babbage's case: +, -, x, ÷)

Babbage's Analytical Engine

Because Babbage was constantly revising his design, the Analytical Engine was never completed; what you see here is one of the intermediate designs which was later revised and rebuilt

In theory the Analytical Engine was capable of reading in program commands on cards; if a recursive section was hit, the machine rewound the cards to find the function called and computed the variables needed!



The conceptualisation of recursive functions was extremely important for Computer Science as a field; these functions allow programmers today to write shorter, cleaner code and use developments made by programmers before them. Like Lovelace's approach, we don't have to re-program how to calculate the Nth Bernoulli number because we have a function which does that!

History is happening RIGHT NOW!

- It's hard to imagine using machines like the one Babbage invented when we're accustomed to the machines of today. Your phone isn't run by levers and gears and neither are simpler technical devices in your life (like microwaves or radios!)
- Though compared to today, these advancements seem huge, within your lifetime it's likely you'll see even bigger changes to the field of computer science!
- Let's explore some current research and opportunities for advancement in our current technology. You might find something interesting that you want to research for yourself!

New-School Machine Structures

- **Parallel Requests**

Assigned to computer
e.g., Search “cats”

- **Parallel Threads**

Assigned to core
e.g., Lookup, Ads

- **Parallel Instructions**

>1 instruction @ one time
e.g., 5 pipelined instructions

- **Parallel Data**

>1 data item @ one time
e.g., Add of 4 pairs of words

- **Hardware descriptions**

All gates functioning in
parallel at same time

Software

Hardware

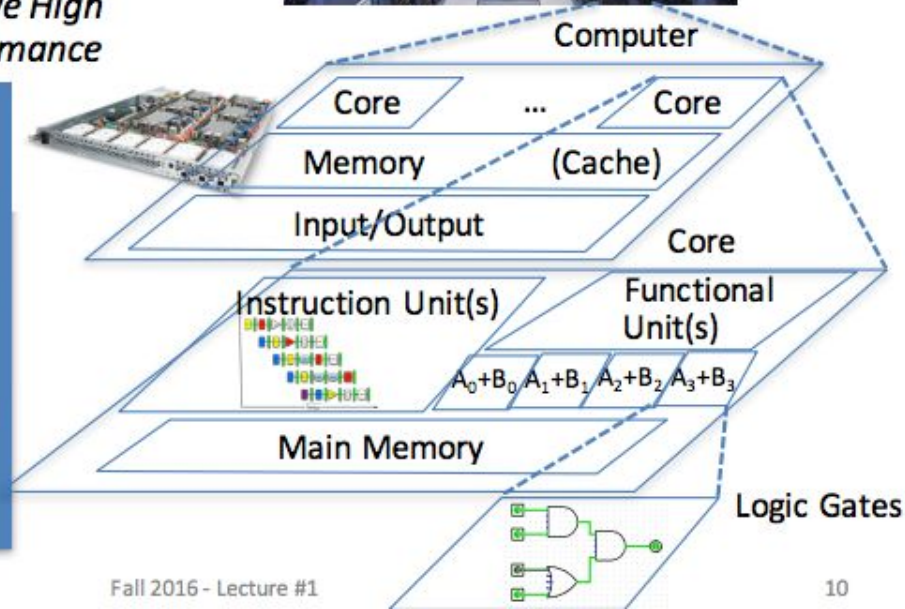
Warehouse
-Scale
Computer



Smart
Phone



*Harness
Parallelism &
Achieve High
Performance*



#2: Moore's Law

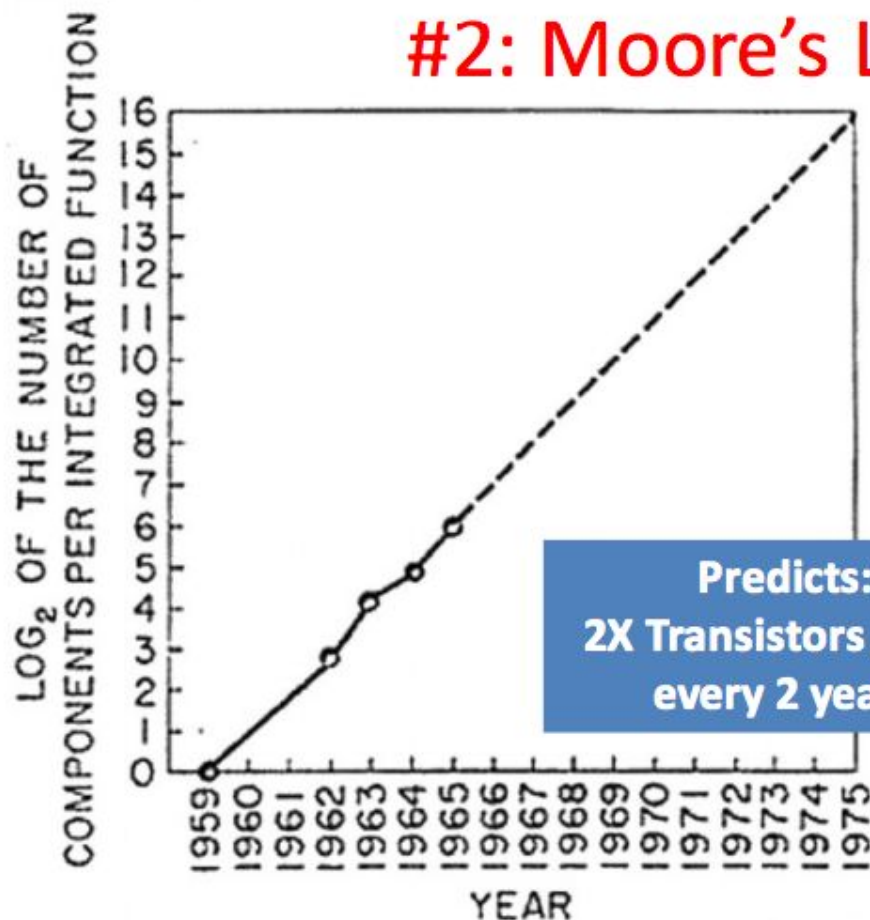


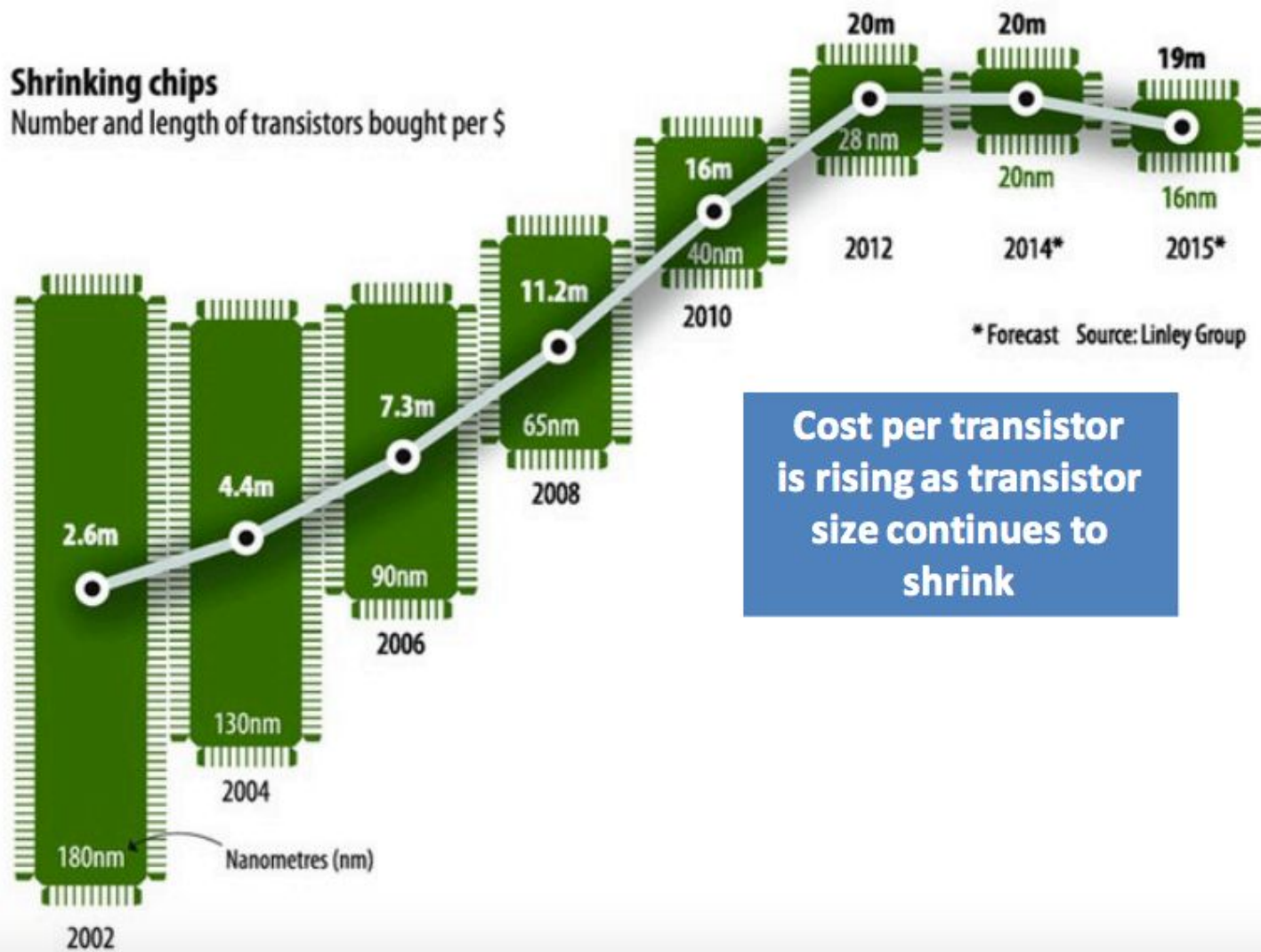
Fig. 2 Number of components per integrated function for minimum cost per component extrapolated vs time.



Gordon Moore
Intel Cofounder
B.S. Cal 1950!

Shrinking chips

Number and length of transistors bought per \$



**Cost per transistor
is rising as transistor
size continues to
shrink**

In Conclusion...

- Tabular diagrams can help you visualise program execution and work done per frame
- Recursive functions are functions which do relatively less work per frame when compared to an iterative function; these functions obtain their final values through repetitive self-calls. Each recursive call computes the arguments for the next function call!
- Ideas like recursion aren't new to the field of Computer Science, in fact, Computer Science has been evolving for centuries! As humanity develops new technology, components grow smaller and more efficient. Within your lifetime you're likely to see BIG developments in this field!