Recursion

Salaar Liaqat

Data Sciences Institute, UofT

Outline

- Call Stack
- Recursion
- Time and Space complexity of Recursion
- Mergesort
- Multiple recursive

Section 1

Call Stack

How a Call Stack Works

- Your computer internally uses a call stack (stack ADT) to execute functions
- When you run your Python file, the `main` functions is called. `main` is pushed onto the stack
  - Sounds familiar? `if __name__ == "__main__":`
- As the main function executes, it may call other functions, each functions is pushed to the top of the stack
  - The currently executing function is at the top of the stack
- When each function is executed, it is popped from the stack
- The function may return a value, which is passes to the calling function (the function below in the stack).
- The calling function can use the return value and continue execution until the stack is empty

Basic Example

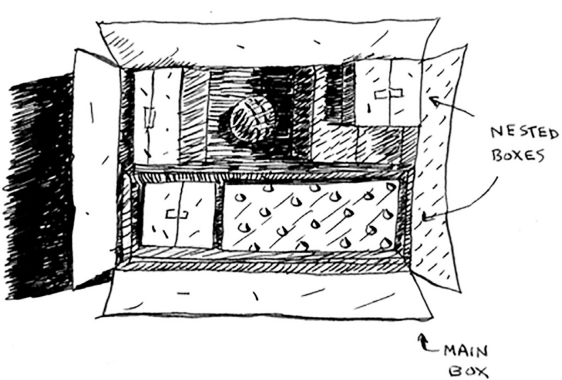
- If I run `round(float("20.24"))`, I expect 20
  - The `round` function is first to be called, it is pushed on the call stack
  - Then, `float("20.24")` is called and pushed on the call stack
- Now, we pop each function off the call stack.
  - `float("20.24")` returns `20.24`
  - `round` uses the return value of the previous function, `20.24`. It executes `round(20.24)`, which returns 20
  - The stack is empty, so the program finishes

Section 2

Recursion

Motivating Example

- Suppose you are looking for a key in a box, but the box contains more boxes!



- 2 minutes: write down the steps of the algorithm you would take to search for the key

Algorithm 1: Loop

1. Make a pile of all the boxes
2. Grab a box and open it
3. If it contains a box, append it to your pile of boxes
4. If it contains the key, you're done!
5. Repeat

Algorithm 2: Recursion

1. Grab a box and open it
2. If it contains a box, repeat step 1
3. If it contains the key, you're done!

Which algorithm do you like more?

- Notice the function is recursive because it calls itself
- Both algorithms achieve the same thing, but recursion is clearer (to me)

Formula to write a recursive function

- Since recursive functions call themselves, its easy to write an infinite loop
- Let's write a function that does a countdown

```
def countdown(i):
  print(i)
  countdown(i - 1)
```

- This runs forever, so we need a *base case* to tell the code when to stop

```
def countdown(i):
  print(i)
  if i <= 0:
    return
  else:
    countdown(i - 1)
```

Factorial

- The *factorial* is the product of all positive integers less than or equal to the given integer
  - $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$
  - We define $1! = 1$
- Let's use recursion to calculate factorials

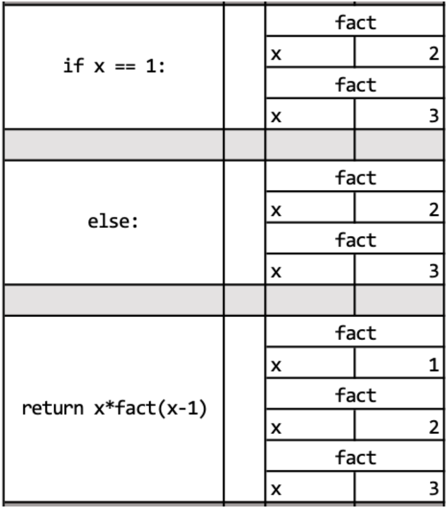
```
def factorial(n):
  if x == 1:
    return 1
  else:
    return x * factorial(x - 1)
```

- Let's examine the call stack when we call `factorial(3)`

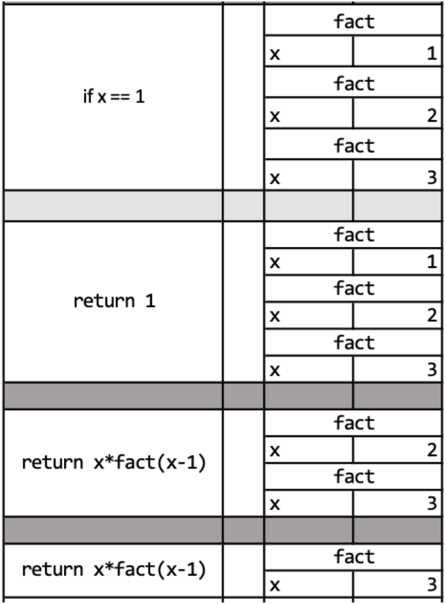
Recursion and the Call Stack

Code		Call Stack	
fact(3)		fact	
		x	3
if x == 1:		fact	
		x	3
else:		fact	
		x	3
return x*fact(x-1)		fact	
		x	2
		fact	
		x	3

Recursion and the Call Stack



Recursion and the Call Stack



Multiple Recursive Calls: Fibonacci Sequence

- In calculating the factorial, each recursion only calls itself once. This doesn't have to be the case
- The Fibonacci Sequence is a sequence of numbers where the first two numbers are 0 and 1, with each subsequent number being being the sum of the previous two numbers in the sequence.
  - Notice how the problem is defined recursively

```
def fib(n):
  if n <= 1:
    return n
  else:
    return fib(n - 1) + fib(n - 2)
```

2 minutes: what is its time and space complexity?

Section 3

Time and Space Complexity of Recursion

Time Complexity of Recursion

- Generally, recursion doesn't have performance benefits compared to loops (in problems like finding a key in nested boxes)
  - However, it is simpler to understand
- The time complexity of recursion depends on the number of time the function calls itself (branches)
  - Factorial: the `fact` is called n times before reaching the base case so its $O(1^n) = O(n)$
  - If a recursive function called itself twice, then its (2^n)
- When a recursive function makes multiple calls, the run time will often be $O(branches^{depth})$

Tricky Example

```
def recursive(n):
  for i in range(n):
    # Something happens
    i += 2
  if n <= 0:
    return 1
  else:
    return 1 + recursive(n - 3)
```

- Loop takes $n/2$ steps, because we increase `i` by 2
- Recursion takes $n/3$ steps **and** the loop is called recursively.
  - In other words, for each recursion, run the loop.
- The time complexity is $n/2 \times n/3 = \frac{n^2}{6} = O(n^2)$

Space complexity of recursion

- Notice the call stack takes up space in memory. How much depends on the depth of the recursion
- Think about the maximum amount of space the call stack will need
  - Factorial: $O(n)$, when recursion reaches the base case
- Even when you have multiple branches, it's possible only 1 branch at depth n is in memory at a time
- 2 minutes: to find the key in nested boxes, what is the memory complexity of the recursive approach versus the loop approach?

Section 4

Mergesort

Divide and Conquer Algorithms

- Divide and Conquer (D&C) is a general method to solve problems utilizing recursion.
  - Figure out the simplest case and use it as the base case
  - Figure out how to reduce your problem to the base case
- Let's start with a trivial example: how would you sum a list of integers?
  - Solution is obvious with a loop
  - Let's do it recursively

Divide and Conquer Algorithms

Step 1

- What is the simplest array to sum?
- Arrays with no elements or 1 element
    - sum of `[]` is 0, sum of `[8]` is 8

Step 2

- How can we reduce all arrays to empty array?
- Notice `sum[2, 4, 5]` $= 2 +$ `sum[4, 5]`, but the second version reduced the problem

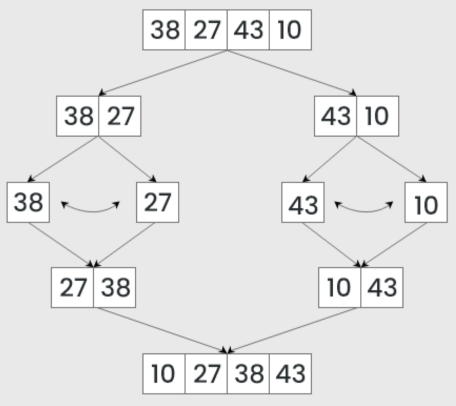
Divide and Conquer Algorithms

```
def rec_sum(lst):
  if not lst:
    return 0
  else:
    return lst[0] + rec_sum(lst[1:])
```

- Let's work on a real problem next!

Mergesort

- Some lists don't need to be sorted
  - Lists of size 1! This is our base case
- We can split lists in half until they contain 1 element, then merge all of the sub-lists
- Python's sort function uses a hybrid of merge and insertion sort, both of which you've learned!



Big-O of Merge Sort

First consider the non-recursive part of the code

- The "divide" step takes linear time, since slicing operations take roughly $n/2$ steps to make a left and right copy respectively.
- The merge operation also takes n steps approximately
- All other operations are constants
- Together, the non-recursive part of this algorithm is $O(n)$

Next consider the recursive calls

- Recall the big-O of recursion depends on the recursion depth and number of calls. $O(branches^{depth})$
- The depth in Merge Sort is the number of times you need to divide to get to a list of length 1.
- Mathematically, $2^{\text{depth}} = n$, then depth $= \log n$. So there are approximately $\log n$ levels

Big-O of Mergesort

- Since the $O(n)$ steps must be performed each recursion, the total run time is $O(n \log n)$. Our analysis only depended on the size of the list, so the best and worst case of mergesort is the same
- This is much faster than insertion sort!
- 2 minutes: does it have less space complexity than insertion sort?

Section 5

Recommended Problems and References

Recommended Problems

- Bhargava: Chapter 4 exercises
  - 4.1 to 4.8
- Write a recursive function that produces the `RecursionError: maximum recursion depth exceeded` error.
- Write a iterative function to calculate the *n*th Fibonacci number. What is its time and space complexity?
- Write a recursive function to determine if a string is a palindrome. What is its time and space complexity?
- Write a recursive function to check if a given positive integer is a prime number. What is its time and space complexity?

Recommended Problems

- Suppose you have a plot of land and want to divide the land into even square plots, while keeping the plots as big as possible. How would you do this using D&C? See Bhargava pg. 52.
- Explain why the "merge" step in mergesort is $O(n)$
- Implement mergesort. You might find using helper functions useful.
- Write a recursive function to perform binary search on a sorted list

Bonus Readings

- You may be interested in learning more about quicksort in Bhargava chapter 4 or here. Quicksort is another recursive sorting method

References

- Bhargava, A. Y. (2016). *Grokking algorithms: An illustrated guide for programmers and other curious people.* Manning. Chapter 3 and 4.
- Cormen, T. H. (Ed.). (2009). *Introduction to algorithms* (3rd ed). MIT Press. Chapter 4.