TIP102 | Intermediate Technical Interview Prep

Intermediate Technical Interview Prep Fall 2024 (@ Section 1 | Tuesdays and Thursdays 5PM - 7PM PT)
Personal Member ID#: **111572**

Unit 7 Cheatsheet

Overview

Here is a helpful cheatsheet outlining common syntax and concepts that will help you in your problem solving journey! Use this as a reference while you solve the breakout problems for Unit 7. This is not an exhaustive list of all data structures, algorithmic techniques, and syntax you may encounter; it only covers the most critical concepts needed to ace Unit 7. In addition to the material below, you are expected to know any required concepts from previous units. You may also find advanced concepts and bonus concepts included at the bottom of this page helpful for solving problem set questions, but you are not required to know them for the unit assessment.

Standard Concepts

Recursion

Put simply, recursion is the process of a function calling itself.

Example Usage:

```
def recursive_crash():
    print("I will run forever")
    recursive_crash()
```

If we call the function `recursive_crash()` above, it will print `"I will run forever"` and then call itself, causing the function to execute again. `"I will run forever"` will repeat again, and the function will call itself again. This will happen over and over until your program crashes.

Iteration vs Recursion

Like a for loop or while loop (also referred to as *iteration*), recursion is a way to repeatedly execute a chunk of code. In fact, recursion and iteration both achieve the same goal (repetition), but with diverse approaches.

Iteration uses a *bottom-up approach*. It begins by solving the smallest subproblem and then works it way up to solving larger and larger subproblems, working our way up to a solution to the overall problem.

In contrast, recursion uses a *top-down approach*. It takes the overall problem and breaks it apart into smaller and smaller subproblems until it finally finds one that can be solved readily. Then, if necessary, recursion follows the same pattern as iteration of using the subproblem solutions to build back up to the overall solution.

Example Usage:

	Iterative Approach	**Recursive Approach**

Function	`def count_iterative(num):`
	`  i = 1`
	`  while i <= num:`
	`  print(f"Count {i}!")`
	`  i += 1`
		`def count_recursive(num):`
		`  print(f"Count {num}!")`
		`  if num == 1:`
		`  return`
		`  else:`
		`  count_recursive(num - 1)`

Input	`5`	`5`

Output	`'Count 1!'`
	`'Count 2!'`
	`'Count 3!'`
	`'Count 4!'`
	`'Count 5!'`
		`'Count 5!'`
		`'Count 4!'`
		`'Count 3!'`
		`'Count 2!'`
		`'Count 1!'`

In the example above, the iterative approach uses a loop to repeat code. A loop variable `i` is initialized, and `Count {i}` is printed once. and then the function works forward, repeating the *loop body*. We move towards ending the repetition by *incrementing* `i` to a larger or larger value until the overall goal is achieved (printing `Count {i}` `i` number of times).

In contrast, the recursive function starts by printing the input value, `num`, then creates repetition by calling itself so that the *function body* repeats. It moves towards terminating the repetition by *decrementing* the argument `num` with each new function call, finally terminating the cycle of function calls with a `return` statement when we reach the smallest possible value for `num`, `1`.

Building a Recursive Function

Every recursive function has at least two main components:

- The **base case** End condition. Describes the condition and code that should run when we want our function to stop making recursive calls. Often the base case is the smallest subproblem of the overall problem we are working to solve.
- The **recursive case** Code to execute in all other cases. The recursive case calls the function again, but usually passes in a smaller or simpler input to move us closer to reaching the base

case.

```
def count_recursive(num):
    # Action to repeat
    print(f"Count {num}!")

    # Base Case: If num is 1 we want to stop counting down
    if num == 1:
        # Terminate the function by returning
        return

    # Recursive Case: If num is larger than 1
    else:
       # Call count_recursive() again, but decrement the input value by 1
       count_recursive(num - 1)
```

A recursive function may have multiple base cases. This is useful when we have multiple conditions under which we want to stop repeating our function body and want to specify different behavior for each condition.

Example Usage:

```
# Check if a given value is odd
def is_odd(n):

  # Base Case 1: n is 0, which is not odd
  if n == 0:
    # Return False
    return False
  # Base Case 2: n is 1, which is odd
  if n == 1:
    # Return True
    return True

  # Recursive case: n is greater than 1
  else:
    # Check if the input subtracted by 2 is odd
    # If n - 2 is odd, n must also be odd
    return is_odd(n - 2)

test_odd_value = is_odd(5)
test_even_value = is_odd(6)

print(test_odd_value) # Prints True
print(test_even_value) # Prints False
```

A recursive function may also have multiple recursive cases. This is useful when we want to specify different behavior depending on some condition(s).

Example Usage:

```
# Count the number of even values in a list
def count_evens(lst):
  # Base case: The list is empty
  if not lst:
    # There are 0 even values in the list
    return 0

  # Recursive Case 1: The first value in the list is even
  if lst[0] % 2 == 0:
    # Count of even values is 1 + the count of evens in the rest of the list
    return 1 + count_evens(lst[1:])
  # Recursive Case 2: The first value in the list is odd
  else:
    # Count of even values is the count of evens in the rest of the list
    return count_evens(lst[1:])

output = count_evens([1, 2, 3, 4])
print(output) # Prints 2
```

When we create iterative algorithms, we often use an accumulator variable to collect our final result.

Example Usage:

```
def count_evens_iterative(lst):
    # Accumulator variable
    count = 0
    for num in lst:
        if num % 2 == 0:
            count += 1
    return count
```

In recursive functions, we instead use the return statement as our 'accumulator variable'. The return statements combines a specified return value for the current function call with the results of recursive calls to generate the final output value.

Example Usage:

```
  if lst[0] % 2 == 0:
    # Count of even values is 1 + the count of evens in the rest of the list
    return 1 + count_evens(lst[1:])
```

In the snippet of the recursive implementation of `count_evens()` above, the output value is 1 plus the return value of `count_evens(lst[1:])`.

Recursive Driver and Helper Functions

It is common to want to pass in extra data to our recursive calls. In this case, we can create a recursive helper function that takes in additional parameters so that we can pass along this extra data.

Example Usage:

```
def partition_evens_odds(lst):
  return recurse_partition(lst, [], [])

def recurse_partition(lst, evens, odds):
  if not lst:
      return evens, odds
  if lst[0] % 2 == 0:
      evens.append(lst[0])
  else:
      odds.append(lst[0])
  return recurse_partition(lst[1:], evens, odds)


lst = [1, 2, 3, 4, 5, 6, 7, 8, 9]
evens, odds = partition_evens_odds(lst)
print(evens) # Prints: [2, 4, 6, 8]
print(odds)  # Prints: [1, 3, 5, 7, 9]
```

`partition_even_odds()` takes in a given list and returns two new lists: one with all the odd values in the input list, and one with all the even values in the input list. It only accepts one parameter: the input list.

To solve the problem recursively, we need to pass in our result lists `evens` and `odds` to each recursive call so that we can append new values to the lists. But those lists don't exist yet! So we create a helper function that takes in two additional parameters, `evens` and `odds`. Then the main 'driver' function creates initial values (empty lists) to pass in as arguments and calls the helper function which executes the recursive logic.

If we were to initialize the empty lists and put any recursive function calls inside the main function, any additions to the list, would be overwritten by the recursive call

```
def partition_evens_odds(lst):
    evens = [] # evens becomes an empty list every time we call the function
    odds = [] # odds becomes an empty list every time we call the function
    if not lst:
        return evens, odds
    if lst[0] % 2 == 0:
        evens.append(lst[0])
    else:
        odds.append(lst[0])
    return partition_evens_odds(lst[1:], evens, odds)
```

When we use recursive helper functions, the driver function usually does very little. It may handle a base case, but it's primary job is to make the first call to the recursive helper function, passing in initial values for any parameters the helper function needs that were not passed to the driver function. The helper function then does all the work!

We could solve this problem without the helper function:

```
def partition_evens_odds(lst, evens, odds):
  if not lst:
      return evens, odds
  if lst[0] % 2 == 0:
      evens.append(lst[0])
  else:
      odds.append(lst[0])
  return partition_evens_odds(lst[1:], evens, odds)


lst = [1, 2, 3, 4, 5, 6, 7, 8, 9]
evens, odds = partition_evens_odds(lst, [], []) # User has to pass in empty lists to hold re
print(evens) # Prints: [2, 4, 6, 8]
print(odds)  # Prints: [1, 3, 5, 7, 9]
```

However, in this case the user needs to pass in an initial list for the `evens` and `odds` parameter, which doesn't provide a good user experience.

Divide & Conquer Algorithms

Divide and Conquer is an algorithmic technique that solves problems by breaking the overall problem down into subproblems of the same type, solving each subproblem, and then combining the results to find the final answer.

The divide and conquer approach can be split into three basic steps:

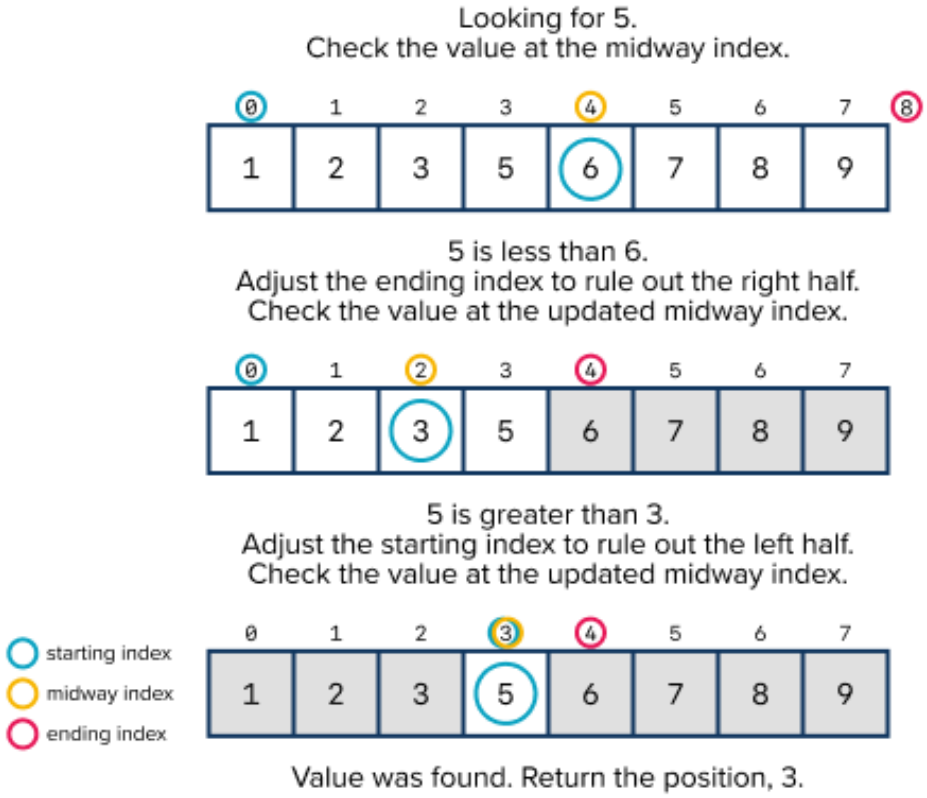
1. **Divide** the overall problem into smaller subproblems.
    - Each subproblem should be solvable using the same technique.
    - The goal is to divide the problem into the smallest possible subproblems.
2. **Conquer** by solving each subproblem.
    - Solve each subproblem (often using recursion).
    - The smallest subproblems (base cases) can be solved directly without using recursion.
3. **Combine** the solutions to each subproblem to find the final result.
    - Combine the results of each subproblem to solve larger subproblems.
    - The goal is to find the final answer by merging the results of larger and larger subproblems.

Binary Search

Binary search is one of the most classic examples of a divide and conquer algorithm. Binary search finds the location of a given target value within a sorted list by dividing the list in half at each step.Regardless of the size of the list, we reduce the area we need to search by half by using logic to determine which half of the list contains the target value at each step. This creates an incredibly efficient searching algorithm with just `O(log n)` time complexity!

nary search can be split into the following steps:

1. Check whether the middle value of the list is the target value.
2. If the middle value is the target value, return the index of the middle/target value.
3. Otherwise determine whether the target value should be in the left or right half of the sorted list by determining if it is smaller or larger than the target value.
4. Perform a binary search on the half of the list that the target value must be in.
5. If at any point, we have a list that is empty or of size 1 and still have not found our target value, we can return `None` or `-1` or some other value to indicate the target value is not in the list.

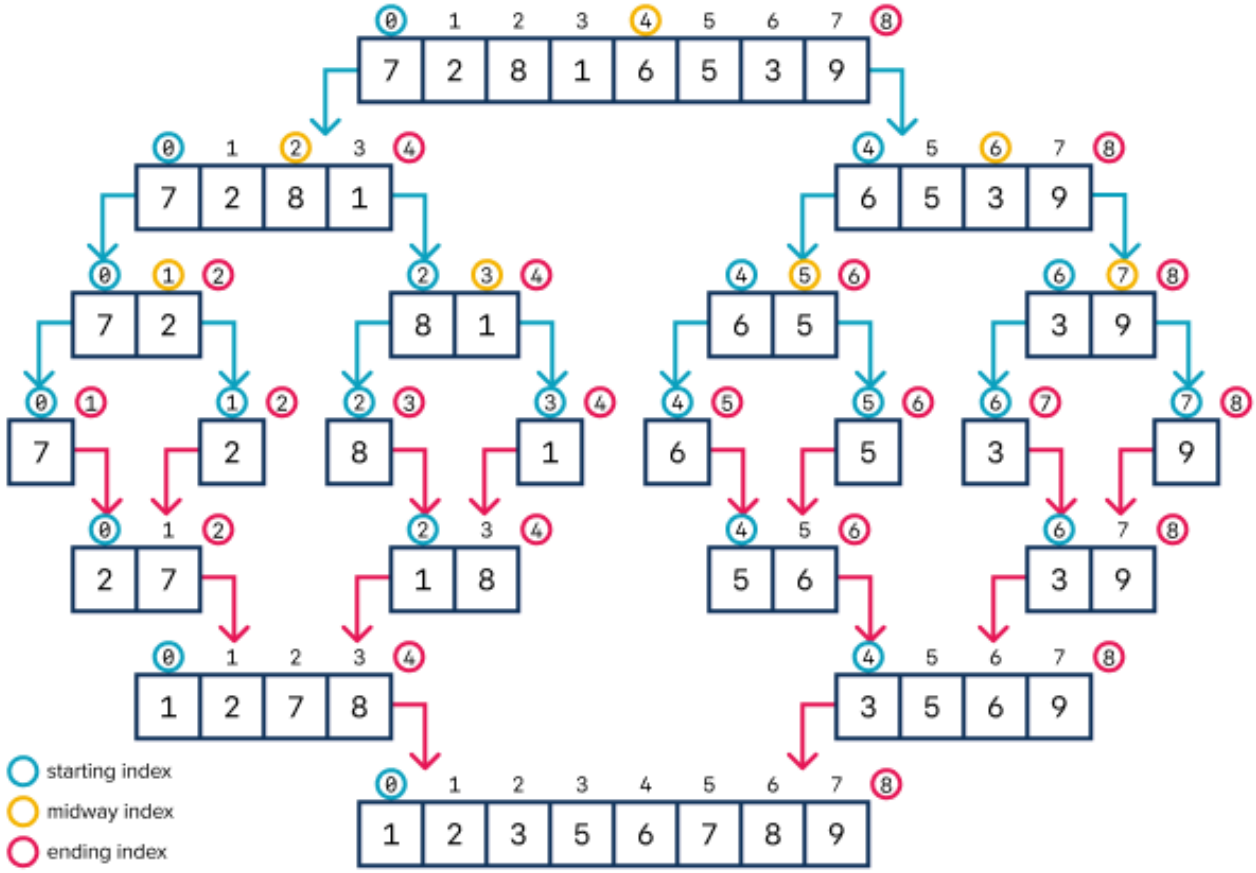


Merge Sort

Merge sort is another classic example of a divide and conquer algorithm. Merge sort is one of the most efficient sorting algorithms, sorting a given unsorted list in `O(n log n)` time.

Merge sort can can be split into the following steps:

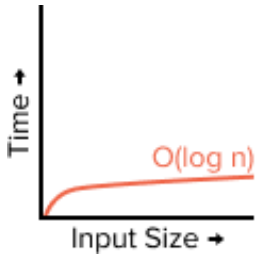
1. **Divide** the list into two halves at each step until we have sublists of length 1.
2. **Conquer** each subproblem, by sorting each sublist (A sublist of length 1 is already sorted).
3. **Combine** the sorted sublists by merging them together. Compare matching indices of two sublists to place them in sorted order.



Logarithmic Time Complexity

⚠️ The following section contains an implementation of the Binary Search Algorithm. For your own learning, we highly recommend you attempt Problem 1 of the Session 2 problem sets (any version) before reading this portion of the cheatsheet.

Divide and conquer algorithms often have a logarithmic time complexity. As a reminder logarithmic algorithms are considered efficient because the time complexity increases incredibly slowly as the input size increases. The algorithm grows in complexity proportional to the **log** of the input size.



What is a log?

A logarithm can be thought of as the inverse of an exponent. It is the power to which a fixed number (the base) must be raised to produce a given number. For example we say the log with base 2 of 8 is 3 because $2^3 = 8$. The log of base 10 of 10000 is 4 because $10^4 = 10000$.

$$\log_2 16 = 4$$

Equivalent to

$$2^4 = 16$$

Practically speaking, **we encounter logarithmic time complexity whenever we reduce the input size to an algorithm with each iteration by dividing the remaining input at a constant rate.**

Take this real world example from Stack Overflow:

```
Given a person's name, find the phone number by picking a random point
about halfway through the part of the book you haven't searched yet,
then checking to see whether the person's name is at that point. Then
repeat the process about halfway through the part of the book where
the person's name lies. (This is a binary search for a person's name.)
```

Binary Search Time Complexity

Binary Search is a classic example of a problem with `O(log n)` time complexity.

```
def binary_search(numbers, value):
    low = 0
    high = len(numbers) - 1
    while low <= high:
        mid = (low + high) // 2
        if numbers[mid] > value:
            high = mid - 1
        elif numbers[mid] < value:
            low = mid + 1
        else:
            return mid

    return None
```

If the input is eight elements [1, 2, 3, 4, 5, 6, 7, 8] and `value` we are searching for is 2:

- In the first iteration `low` will be 0 and `high` would be 7, and `mid` would be 3 ((0+7) / 2).
- On the second iteration `low` remains 0 and `high` becomes 2 and `mid` becomes 1.
- Then value is found in the list and the function returns 1.

Notice with each iteration the size of the input involved in the search (`low` to `high`) is halved. So worst-case a list of 8 items would take 3 iterations to find the value. We could double the size of `numbers` to 16 items and it would only take 4 iteration to find the value and for 32 items it would only take 5 iterations. Thus while the function *does* take longer as the input size increases it does not increase very rapidly.

In the worst case scenario, the algorithm will take the following number of iterations given the input of size `n`:

n	Number of iterations
8	3
16	4
32	5
64	6

In general:

- If the size of the input is *divided* by some value with each iteration, the time complexity involves a logarithm with a base equal to the divisor.
- By far, the most common logarithmic base is 2 because our algorithms often, like binary search, divide the input size by two with each iteration.
    - However, we could have a different base such as 4 which would mean we are dividing the input size by 4 with each iteration.
    - When we use Big O notation, we do not indicate the base. We say the time complexity is `O(log n)` whether we are dividing the input by two, ten, or some other number each time.

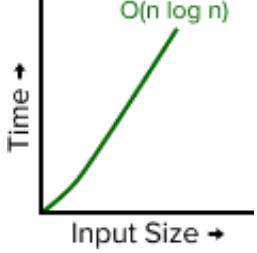
Advanced Concepts

Log-Linear Time Complexity

> ⚠️ The following section contains an implementation of the Merge Sort Algorithm. For your own learning, we highly recommend you attempt Problem 6 of the Session 2 Advanced problem sets (any version) before reading this portion of the cheatsheet.

A log-linear time complexity, denoted as `O(nlogn)`, describes an algorithm whose runtime grows proportional to the input size `n` *and* includes a logarithmic factor. This time complexity is most often seen with Merge Sort and Quick Sort algorithms.

The `n` in `O(nlogn)` represents a linear pass through all elements in the input. The `log(n)` portion represents the number of times the input is split or reduced in size, as is commonly seen in divide and conquer algorithms like Merge Sort where problems are repeatedly broken down into smaller subproblems.



Merge Sort Time Complexity

Merge sort is a classic example of an algorithm with `O(nlogn)` time complexity.

```
 def merge_sort(arr):
    if len(arr) <= 1:
        return arr  # Base case: arrays with 1 or no elements are already sorted

    # Divide the array into two halves
    mid = len(arr) // 2
    left_half = merge_sort(arr[:mid])
    right_half = merge_sort(arr[mid:])

    # Merge the sorted halves
    return merge(left_half, right_half)

def merge(left, right):
    result = []
    i = 0  # Pointer for left half
    j = 0  # Pointer for right half

    # Compare elements from both halves and merge them in sorted order
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1

    # Append any remaining elements in the left or right half
    result.extend(left[i:])
    result.extend(right[j:])

    return result
```

Breaking down the algorithm:

1. Dividing the Array (Logarithmic `O(log n)`)
    - Merge Sort uses a divide and conquer approach to recursively split the input array into two halves until each half contains only one element.
    - The number of times we can split the array `arr` of length `n` until we get to arrays of length `1` is proportional to $\log_2(n)$.
        - For example, an array of 8 elements can be split 3 time: `8 -> 4 -> 2 -> 1`. This is equivalent to $\log_2(8) = 3$.
2. Merging the Subarrays (Linear `O(n)`)

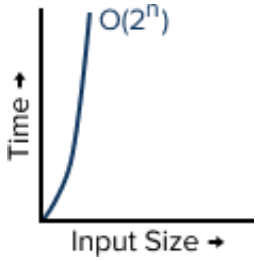
- After dividing the array into pieces, the algorithm merges the sorted subarrays back together.
- Merging each of the subarrays takes `O(n)` time where `n` is the combined number of elements in the two subarrays because we have to compare each element of each subarray in order to combine them into one sorted array.
- In the worst case (when we are combining two halves of the original array `arr`), there are `n` elements to combine so this part takes `O(n)` time.

3. Putting it All Together (Log-Linear `O(nlogn)`)
   - At each recursive iteration, the algorithm performs `O(n)` operations merging the subarrays.
   - There are `log(n)` recursive iterations, because that is the number of times we split the array.
   - Combining these time complexities we get `O(n) * O(log n) = O(nlogn)`.
   - We multiply rather than sum because this is a nested operation, not a sequential operation: for each division, we have to go through and merge the two subarrays.

Exponential Time Complexity

A function has exponential time complexity when its runtime doubles with each addition to the input size. Exponential algorithms have a complexity that increases very fast in proportion to the input size so we want to prevent exponential algorithms wherever possible.

Exponential algorithms are often seen when problems use a recursive approach or conduct an exhaustive search where every possible combination of elements must be considered to find a solution.



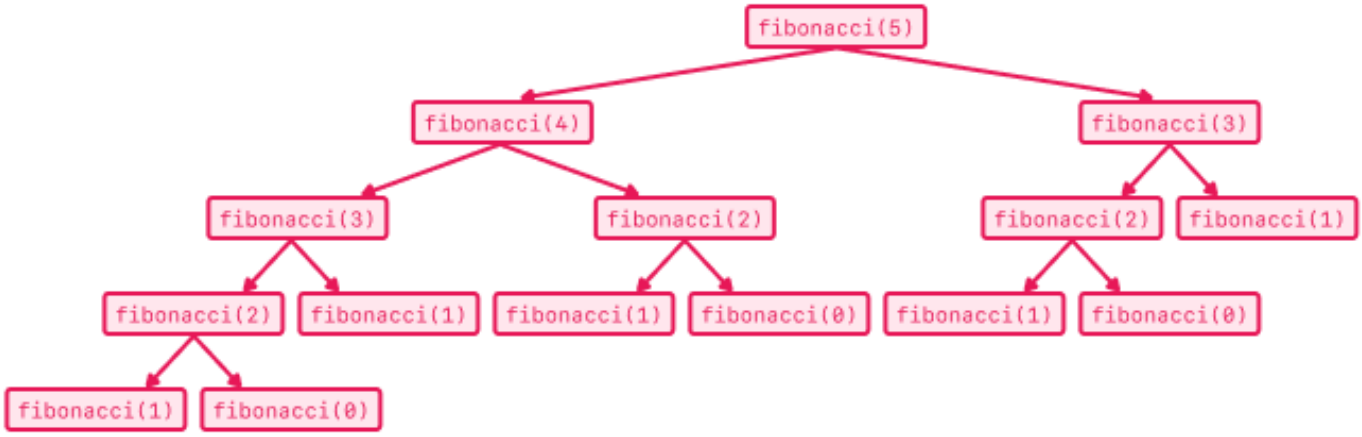
Fibonacci Time Complexity

The naive implementation (simple, non-optimized solution) of Fibonacci is a prominent example of a `O(2^n)` algorithm.

As a reminder, the Fibonacci sequence is a sequence of numbers where the `nth` number in the sequence is the sum of the previous two numbers in the sequence. The `0th` Fibonacci number is `0` and the `1st` Fibonacci number is `1` by definition.

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

We can observe how inefficient this is. For each recursive iteration, two more recursive calls are made. For example, if we call `fibonacci(5)` initially make 1 call. But then it calls `fibonacci(4)` and `fibonacci(3)`. Then those two subsequent calls each make 2 more calls (4 calls total), and so on. Therefore we get exponential or `O(2^n)` growth.



The Call Stack

When we make function calls, the body of the called function may call other functions, which may themselves call even more functions.

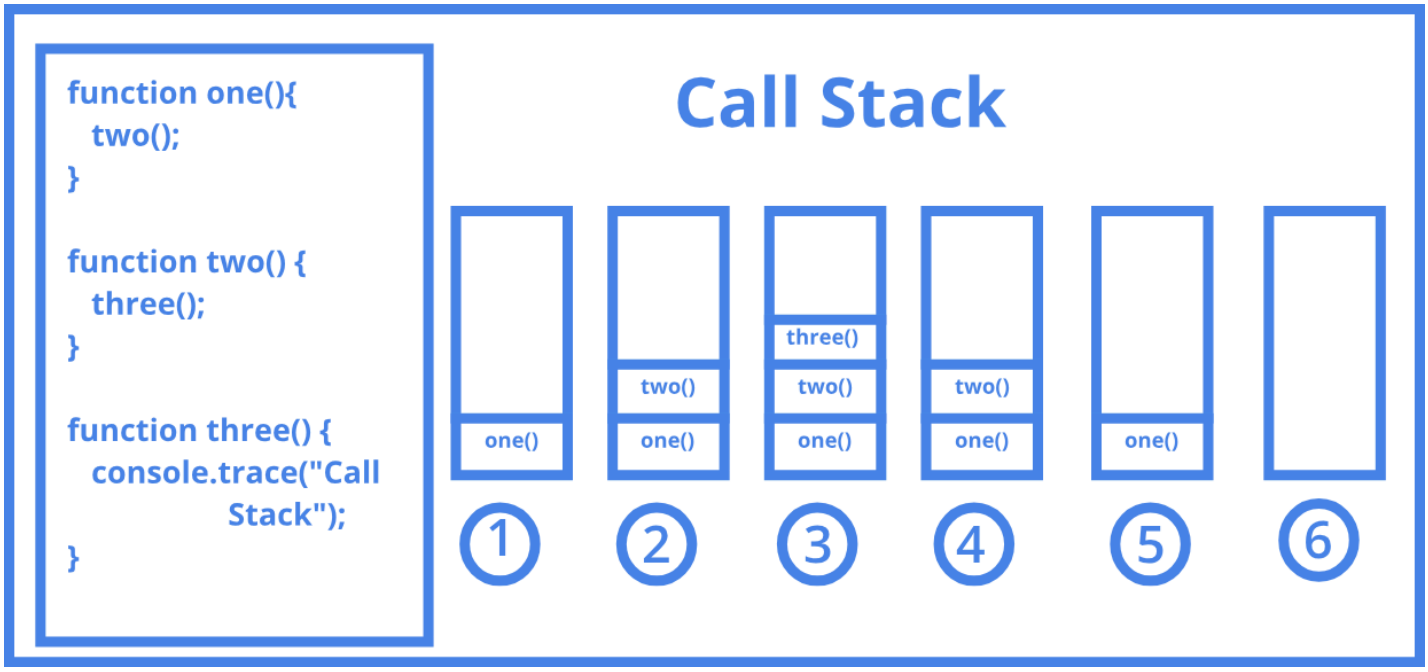
For example, in the following snippet, `function_a()` calls `function_b()` which then calls `function_c()`.

```
def function_c():
    print("I'm Function C!")
    print("Function C is done executing!")

def function_b():
    print("I'm Function B!")
    function_c()
    print("Function B is done executing!")

def function_a():
    print("I'm Function A!")
    function_b()
    print("Function A is done executing!")

function_a()
```

When one function calls a second function, the first function pauses executing any remaining steps until the second function is finished executing. Running the above example, results in the following output:

```
I'm Function A!
I'm Function B!
I'm Function C!
Function C is done executing!
Function B is done executing!
Function A is done executing!
```

To help keep track of what code it should run next, computers maintain a call stack.



Source: via Medium

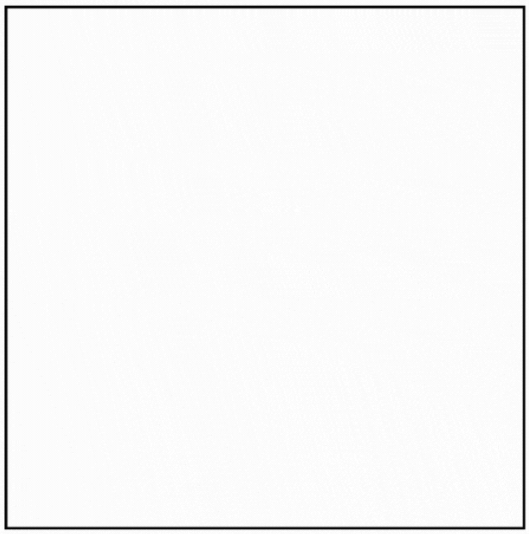
A call stack uses a stack data structure to track the order of function calls! Imagine functions as library books, and the call stack as a stack of library books we have checked out. Every time we make a new function call, the function gets added as a new book on the top of the stack. The computer always reads and executes any instructions inside the stack's topmost book first. When it finishes reading the book, it removes the book from the stack it returns it to the library.

When one function calls another function, it is akin to checking out a new book while the computer is busy reading another book. When this happens, the computer will put a bookmark in its current book and immediately start reading the new book. Once the computer finishes reading the new book, it is removed from the stack, and the computer goes back to reading the original book at its bookmarked place. When the computer has read all books in the stack, it means the program has finished executing!

Recursion and the Call Stack

Recursive functions repeatedly call themselves. The call stack works with recursive functions the same way as with non-recursive function calls, but instead of having a bunch of different books stacked on top of each other, it's like having a stack of multiple copies of the same book.

```python
def sum_zero_to_n(n):
    if n == 0:
        return 0
    else:
        return n + sum_zero_to_n(n-1)

sum_zero_to_n(5)
```



Call Stack

Recursion and Space Complexity

The call stack takes up memory! Stacks, including the call stack, are just a special type of list that insert and remove elements in a specific order. We can envision each function call as being an element in a list, which means the number of function calls our functions make affects our function's space complexity!

For example in the `sum_zero_to_n()` function pictured above, we can say `n` is the size of the input integer. Approximately `n` function calls get added to the call stack before we begin removing functions from the call stack, so the space complexity of `sum_zero_to_n()` is `O(n)`.

The call stack of non-recursive functions also takes up space, but it is almost always a constant amount of space.