Jul 11, 2021
PS 3 Solutions

Two-Sum: you are given an array of integers n and a number k . Return the indices of the two numbers in n that add up to k. For example, given the array $[1, 3, 7]$ and $k = 8$, the answer will be $[0, 2]$ since the values at 0^{th} index of n (=1) and the value at the 2^{nd} index of n (=7) add up to 8.

Brute Force

We use two for loops (nested) to iterate through the array n, checking whether the sum of the current number and each of the numbers in front of it adds up to the target k.

```
array, k = [1, 3, 7], 14

def two_sum(n, k):
    for i in range(len(n)):
        for j in range(len(n)):
            if n[i] + n[j] ==k:
                return [i, j]
```

The time complexity of this algorithm is $O(n^2)$, as we have a nested for loop. We can see this via having to, in the worst case, test n^2 pairs of i and j to see if the values at those indices in the array n sum to k. We can ask ourselves: what *is* the worst case? When we find k last!

```
array, k = [1, 3, 7], 14

def two_sum(n, k):
    for i in range(len(n)):
        #i will be 0, 1, 2

        for j in range(len(n)):
            #j will be 0, 1, 2

            print(f'Testing the values of {n} at indices {i}, {j} \n to see if {n[i]}
                    and {n[j]} add up to {k}')

            #We test the i-j pairs: 0 0 (1+1), 0 1 (1+3), 0 2 (1+7),
            #1 0 (3+1), 1 1 (3+3), 1 2 (3+7), 2 0 (7+1), 2 1 (7+3), and
            #2 2 (7+7).

            if n[i] + n[j] ==k:
                return [i, j]

print(two_sum(array, k))

#Output:

Testing the values of [1, 3, 7] at indices 0, 0 to see if 1 and 1 add up to 14
Testing the values of [1, 3, 7] at indices 0, 1 to see if 1 and 3 add up to 14
Testing the values of [1, 3, 7] at indices 0, 2 to see if 1 and 7 add up to 14
Testing the values of [1, 3, 7] at indices 1, 0 to see if 3 and 1 add up to 14
Testing the values of [1, 3, 7] at indices 1, 1 to see if 3 and 3 add up to 14
Testing the values of [1, 3, 7] at indices 1, 2 to see if 3 and 7 add up to 14
Testing the values of [1, 3, 7] at indices 2, 0 to see if 7 and 1 add up to 14
Testing the values of [1, 3, 7] at indices 2, 1 to see if 7 and 3 add up to 14
Testing the values of [1, 3, 7] at indices 2, 2 to see if 7 and 7 add up to 14
[2, 2]
```

While Loops

We van also think to *sort* the array first, then use a left & right pointer to move towards the middle of the array accordingly, based on if our sum is greater than or less to our target.

```
array, k = [1, 3, 7], 8

def two_sum(n, k):

    n = sorted(n)

    lhs, rhs = 0, len(n)-1

    while lhs < rhs:

        sum_indices = n[lhs] + n[rhs]
        print(lhs, rhs, sum_indices)

        if sum_indices == k:
            return [lhs, rhs]
        elif sum_indices < k:
            #left to right
            lhs += 1
        else:
            #right to left
            rhs -= 1
```

The time complexity of this algorithm is O(n log n) due to the *cost* of sorting.

Dictionaries

We can iterate through the index-loop of the array *n* and find each number's *compliment* (defined as *k – val*). If that compliment exists as a key in the dictionary, we return the current index and the value of the dictionary. If not, we store the value of the current index as its index.

```
array, k = [1, 3, 7], 8

def two_sum(n, k):

    d = {}

    for i, v in enumerate(n):
        if k-v in d:
            return [d[k-v], i]
        d[v] = i

print(two_sum(array, k))
```

The time complexity of this algorithm is O(n) – a dictionary is constant time, O(1), and a for loop is O(n). We take the "bigger" O as the algorithm's complexity.

1. Chop by letter

Can you write a function that takes in a given string *s* and recursively prints out all but the last letter? Clearly articulate your base/terminal case.

```
chop_by_letter('Banana')
Banana
Banan
Bana
Ban
Ba
B
```

This problem has three parts: (i) slicing a string to chop off the last letter; (ii) doing that recursively; and (iii) knowing when we want to stop, a.k.a defining our base case.

(i) To chop off the last letter of a string *s* we can use python's indexing syntax to return a ***slice object*** of *s* representing the set of indices specified by range(start, stop, step). This operation is, if *s* is most sequence types (both mutable and immutable), *s*[i:j:k], which gives a slice of *s* from *i* to *j* with step *k*. If *x* is "banana", x[::1] returns "banana" (*b* -> step 1 -> *a* -> step 1 -> *n* …), x[2:3] returns "n", where our starting index is 2 (the letter "n") and we go *to* index 3 (the letter "a"); remember we do not *get* all the way to index 3, we just go to it. This syntax offers negative indexing, meaning the index -1 is our last entry in a given sequence type. In the context of chopping a letter off a string, the last letter of *s* would then be *s*[-1], and everything up to the last letter would be *s*[:-1].

(ii) In any problem of recursion, we will define a function and inside the function have it call itself. If we look at what's going on, we start with printing *s*, "Banana". So, we can just write *print(s)*. What next? Well, we want to print "Banan", then "Bana", "Ban", "Ba", "B". If we wanted to do this iteratively, that's just an index-based for loop that loops over the reversed range of len(s),

```
string = 'Banana'

def chop_by_letter(s):
       '''Iterative solution to chop_by_letter'''
       print(s)
       for i in range(1, len(s))[::-1]:
              print(s[:i])
```

To do that "Banan", "Bana", etc. through *recursion*, we need to just call the function on itself. What will the input be? Everything up until the last letter! Now, when do we want to stop?

(iii) The base/terminal case is when we want to *stop* recursing – when we want to stop chopping off letters. What if our *s* reaches a len(1), just one letter? There's no point in recursing down to an empty string '', so our base case is defined as when we run out of letters, a.k.a the length of our string ==1.

```
string = 'Banana'

def chop_by_letter(s):
    print(s)

    #Base case
    if len(s) ==1:
        return;

    return chop_by_letter(s[:-1])

chop_by_letter(string)
```

2. Recursive Fibonacci

Problem Statement:

In mathematics, the Fibonacci numbers, commonly denoted F_n, form a **sequence**, called the Fibonacci sequence, such that each number is the sum of the two preceding ones, starting from 0 and 1. That is,

$$F_0 = 0, F_1 = 1$$

and

$$F_n = F_{n-1} + F_{n-2}$$

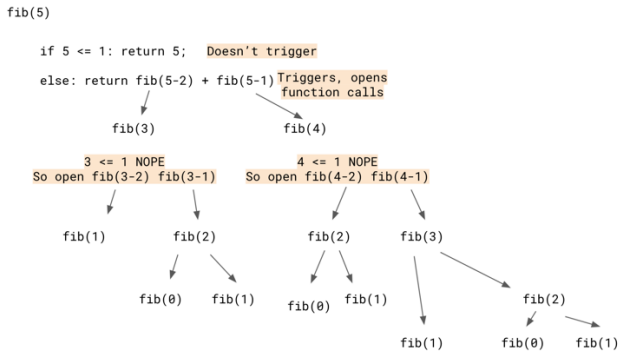
for $n > 1$.

Recursively, we may think to write

```
def recursive_fib(n):
    '''Returns nth Fibonacci number'''
    if n <= 1: return n
    else: return recursive_fib(n-2) + recursive_fib(n-1)
```

(i) Pt. 1: What is the time complexity of this algorithm?
(ii) Pt. 2: Can you write a recursive solution to returning the *nth* number of the Fibonacci sequence in $O(n)$ time?

Pt. 1: To find out the time complexity of this recursive algorithm, we can think to just take *n* as 5, and see what our "recursive trace" looks like. We can diagram what the code will look like.

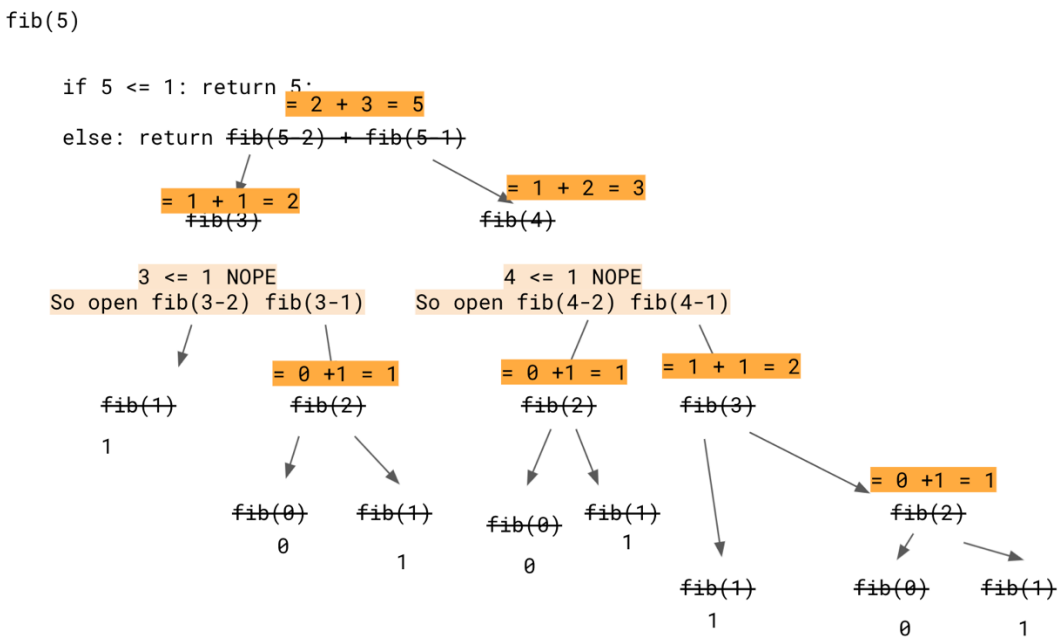


We've traced out fib(5) to all its open function calls until our *base* case has been reached for each one. We've dug ourselves into a hole, and now we *compute* our way out of it. We can start crossing off what fib(0), fib(1) are, and use that to replace fib(2) as the sum of fib(0) and fib(1).

Mathematically, we can envision the process as:

$$
\begin{aligned}
fib(5) &= fib(4) + fib(3) \\
&= \big(fib(4-2) + fib(4-1)\big) + \big(fib(3-2) + fib(3-1)\big) \\
&= \big(fib(2) + fib(3)\big) + (fib(1) + fib(2))) \\
&= ((fib(2-1) + fib(2-2) + fib(3-1) + fib(3-2)) + (1 + fib(2-2) + fib(2-1)) \\
&= ((fib(1) + fib(0) + fib(2) + fib(1)) + (1 + fib(0) + fib(1)) \\
&= (1 + 0 + fib(2-2) + fib(2-1) + 1) + (1 + 0 + 1) \\
&= (1 + 0 + fib(0) + fib(1) + 1) + (1 + 0 + 1) \\
&= (1 + 0 + 0 + 1 + 1) + (1 + 0 + 1) \\
&= 3 + 2 \\
&= 5
\end{aligned}
$$

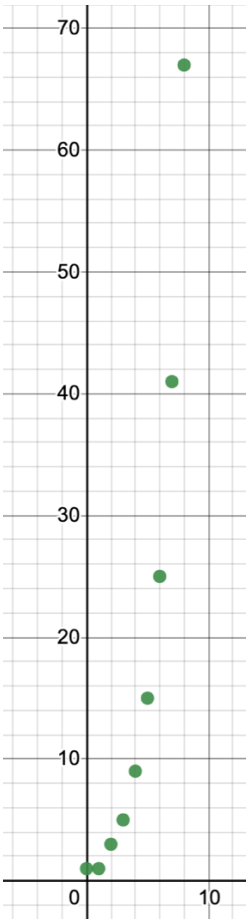
We start from the bottom, the base cases, and cross them out. We know `if n <= 1: return n` so fib(0) =0, fib(1) =1. Once we do that, we can say fib(2) = fib(0) + fib(1) = 0 + 1 = 1, so fib(2) can be crossed out as 1, then fib(3) = fib(1) + fib(2) = 1 + 1 = 2, so fib(3) can be crossed out as 2, so on and so forth.



How many calls do we make? For fib(0), fib(1) we make *one* call. For fib(2), *three calls* (fib(2), fib(2-2), fib(2-1). For fib(3), *five calls* (fib(3), fib(3-2), fib(3-1), fib(1), fib(0)). If we make a table,

fib(n)	# of calls
0	1
1	1
2	3
3	5
4	9
5	15

What's the relationship between *n* and the # of calls? Linear? No, because 5 != 5. n^2? No, because 5 != 25. If we graph this,



it's pretty clear that the time complexity is exponential, O(2^n), which means it scales terribly.

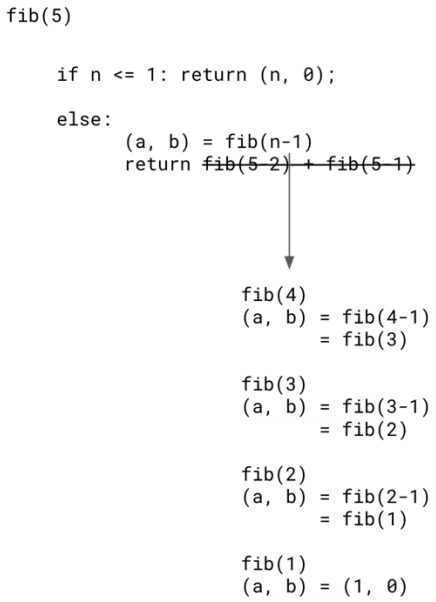
Pt. 2: Now we want to see if we can write a solution in *linear* time, O(n).

The code to do this is:

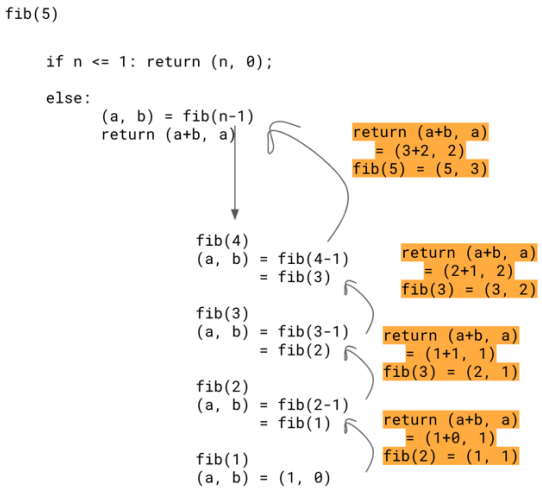
```
def fib(n):

    if n <=1:
        return (n, 0)
    else:
        (a, b) = fib(n-1)
        return (a+b, a)
```

How does this work? Let's make another recursive trace for fib(5). We start by digging ourselves into a hole until we reach our base case:



Then compute our way out using our new-found (a, b) and start to plug that into (a+b, a).



For this one, we can just count how many function calls we make: *5*. The time complexity of this algorithm is O(*n*) as fib(5) makes 5 calls, fib(4) makes 4 calls, etc.

1. **Subarray sums**

Problem statement:

A *subarray* is a consecutive sequence of zero or more values taken out of that array. For example, [1, 3, 7] has seven subarrays:

[] [1] [3] [7] [1, 3] [3, 7] [1, 3, 7]

Notice that [1, 7] is not a subarray of [1, 3, 7] because even though the values 1 and 7 appear in the array, they're not consecutive in the array. Similarly, the array [7, 3] isn't a subarray of the original array, because these values are in the wrong order.

The *sum* of an array is the sum of all the values in that array. Your *task*, should you choose to accept it, is to write a function that takes as input an array and outputs the sum of all of its subarrays. For example, given [1, 3, 7], you'd output 36, because:

[] [1] [3] [7] [1, 3] [3, 7] [1, 3, 7] =

$$1 + 3 + 7 + 4 + 10 + 11 = 36$$

Please note the time complexity of your solution. If it's n^3 you can come up with something faster!

Solution:

Brute force: Our brute force solution has three parts: (i) use index-based looping to generate all start and stop indices of subarrays of a 2-D array in $O(n^2)$ time, (ii) use slicing to turn our indices, *i* and *j* respectively, into every subarray, and (iii) instantiate a counter *c* and increment *c* with all elements of our slice object.

(i) Index-based looping means `for i in range(len(some_sequence_type))` instead of `for i in some_sequence_type` . So, disregarding the start-stop index stuff, in $O(n^2)$ time will be a nested for loop, as in

```
n = [1, 3, 7]
for i in range(len(n)):
    for j in range(len(n)):
       #We know i and j are indicies of n, but what are they?
       print(i, j)
```

```
i, j
0 0
0 1
0 2
1 0
1 1
1 2
2 0
2 1
2 2
```

(ii) Using slicing to turn our indices *i* and *j* into every subarray of *n* means going from index values 0 and 0 to the subarray [1], since the 0th index to the 0th index is just the 0th index element's value in *n* : 1.

```
      i, j
   0, 0 --> [1]
  0, 1 --> [1, 3]
0, 2 --> [1, 3, 7]
   1, 0 --> []
  1, 1 --> [3]
 1, 2 --> [3, 7]
   2, 0 --> []
   2, 1 --> []
  2, 2 --> [7]
```

To get rid of this 1, 0 -> [] gunk, we can just say *j* cannot be less than *i*, so *j* can just start at *i*!

```
n = [1, 3, 7]
for i in range(len(n)):
    for j in range(i, len(n)):
       sub = n[i:j+1]
       print(f'{i}, {j} --> {sub}')
```

Note that slices' stop integer, *j* does not include *j* in the slice, so we add 1 to *j*.

(iii) To instantiate a counter *c* we can write

```
n = [1, 3, 7]
c = 0
for i in range(len(n)):
    for j in range(i, len(n)):
       sub = n[i:j+1]
       print(f'{i}, {j} --> {sub}')
```

To add the sum of our subarray *sum* to *c*, we can write yet another loop to go through everything in *sub* and add it to *c*, i.e.

```
n = [1, 3, 7]
c = 0
for i in range(len(n)):
    for j in range(i, len(n)):
       sub = n[i:j+1]
       print(f'{i}, {j} --> {sub}')
       for k in sub:
           c += k
```

$O(n^2)$: To solve this in quadratic time, we first think of writing just this:

```
n = [1, 3, 7]
c = 0
for i in range(len(n)):
    for j in range(i, len(n)):
       sub_val = n[j]
       print(f'{j} --> {sub_val}')
```

We see this output:

```
0 -> 1
1 -> 3
2 -> 7
1 -> 3
2 -> 7
2 -> 7
```

Isn't that just everything we need to sum right there? Instead of taking a *slice* object, we can just find the values that will be within that slice object immediately via n[j]. Now, if we just instantiate another sum of that in a variable called *sub_sum* , we can easily increment it to *c*!

```
n = [1, 3, 7]
c = 0
for i in range(len(n)):
    sub_sum = 0
    for j in range(i, len(n)):
       sub_val = n[j]
       sub_sum += sub_val
       c += sub_sum
```

There is an O(*n*) time, O(1) space solution, but it's a bit math heavy and I'm too lazy to write it up. Feel free to ask me about it if you're curious.