Notes & Homework: Jul 3, 2021

How long stuff takes

We can get a *feel* for things taking forever viscerally, but there's a way of quantifying how long stuff takes.

Let's say I have a for loop,

```
for i in range(3):
   print(i)
```

How many `i`'s will I see?

What if I replace that with n , how many `i`'s will I see?

Yes, I will see `i` print out n times. The *time complexity* of this algorithm, then, is $O(n)$. This is linear. The time of the algorithm is dependent on the size of n .

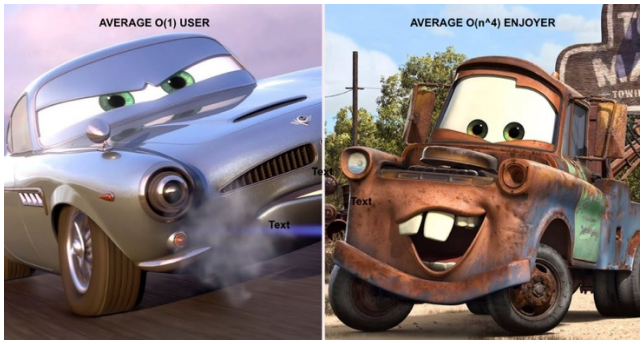
What about a nested for loop?

```
for i in range(3):
   for j in range(3):
          print(i)
```

Yes, I will see `i` print out n^2 times. The *time complexity* of this algorithm, then, is $O(n^2)$. This means our algorithm's time will increase by a power of 2 for every 1 increase in n .

See how big that O is, O – as if calling out into space, the "infinite", right? That's why this is called big-oh notation. It also just makes you go *oh* (in pain) *oh* -- playing in my head, is: "Oh my god so in love found you finally …" does anyone know what song that is?

Often times, people will ask you, yes perhaps a Gentle Lad on the street will glance at you: what's the time complexity of this algorithm, of that? he might say. And what would you say? And even if nobody's in the room, if you're writing a web scraping algorithm as part of your ETL pipeline, you *want* speed. It's Cars 5 <<early release>> up in this shit, for real man.



What time complexity is, though, is just how long does something take, and it's quantified via big-oh notation.

Let's walk through a classic problem, Two-Sum. Try to solve it on your own first, and we will walk through it step by step.

Active solve: you are given an array of n integers 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 1 (index 0) and 7 (index 2) add up to 8.

Oh yes, I've given you this big of a space to THINK

You may start off by thinking that, well (recommended Soundtrack to this next part)

assignment statement `n = [1, 3, 7] k = 8`

(1) loop through that darn damn god Rabbit array `for i in n:`
(2) Monsieur, j'ai besoin de *compliments*, i.e. 1+3 1+7 3+1 3+7 7+1 7+3, but *how* ? 1 + [`for i in n]` → just a nested loop 🐿
(3) if statement to check if == (equality operator) *target* HERE HERE Monsieur, << en route!! >>
(4) return `i, j`

hold fast, dearest Lab members, your Captain, your Capitán, the Tour de France, I have forgotten this is about *indices* not just `i, j` AH, .. stupid me (this sadness, we must blow through it)

(5) index-based looping, double rainbow, all the way, i.e. `for i in range(len(n)):`

OK, now we can get *crackin' codin'*

<<Our initial approach is using two for loops and then iterate through the array, checking whether the sum of the current number and each of the numbers in front of it adds up to the target.>>


lassos at the ready, saddle up


we gettin' crackin'codin'

```
def twosum(n=[1,3,7],k=8):
    for i in range(len(n)):
        for j in range(len(n)):
            if n[i] + n[j] == k:
                print([i, j])
```

nice work!

This is called a **brute force** solution because it's like just forcing your way through, you're not really like being *careful* about how long it will take, or concerned about space/time complexity, you're just going

This is also an $O(n^2)$ solution, it's quadratic in time, since we have a nested for loop here

So, as is customary, I must further beseech you, what is a faster way to do it?

How about this:

We can sort the array first and use a left and a right pointer and move those towards the middle of the array accordingly based on whether the sum is less than or greater than our target.

OKOk, now that *last* sentence was quite a mouthful. I was just spitting kind of verbal fire towards you before that and you were probably like, *yeah* I get what Liam is saying *yeah yeah* (one of you, I know who, was probably physically nodding while reading)

Active solve: Can you try writing this in code: We can sort the array first and use a left and a right pointer and move those towards the middle of the array accordingly based on whether the sum is less than or greater than our target.

Some hints if you get stuck, use built-in function `sorted()` and left-hand and right-hand pointers basically just mean start at the left-hand side of the array (index 0) for lhs (left-hand side) and start at index -1 for rhs (right-hand side).

What do you think "moving towards the center" means for lhs, rhs pointers?? (hint: think -= and +=)

Yeah, it's basically just

```
def twosum(n, k):

      #Sort n
      n = sorted(n)

      #Instantiate pointers
      lhs = 0
      rhs = len(n) – 1

      #Don't go out of bounds you sneaky weasels
      while (lhs < rhs):

            #s is for sum ok?
            s = n[lhs] + n[rhs]

            if s == k:
                  #return that shit as a list bro POSITIVE masculinity
                  return [lhs, rhs]

            #not there yet, so move towards the center
            elif s < k:
                  #left to right
                  lhs += 1
            else:
                  #right to left
                  rhs –= 1
```

Hmm, so what's the time complexity of this thing?

Well, it's kind of a trick question. Depends on the sorting algo (algorithm) you use. In this case the `sorted()` function is time complexity `O(n log n)` [source].

Now … can we get a solution in *linear time?* O(n). Possibly switch to this soundtrack

Your first thought should be: just throw a dictionary at it. In python dictionaries are built on hash tables whose key-value pairs have constant look-up times, O(1) [good article on it by Jessica Yung]

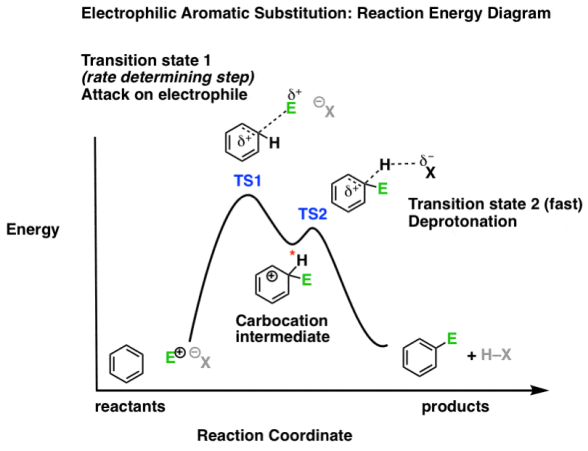
Active solve: Can you try writing this in code: Iterate through the array *n* and find the compliment of each number. If the compliment already exists as a key in the dictionary, return the corresponding number (which will be an index) along with our current index, and the sum of the two numbers at those indices will add up to *k*. If the compliment does not already exist as a key in our dictionary, we will just store our current number and index in our dictionary.

You might've gotten something like:

```
def twosum(n, k):

      #Instantiate empty dictionary
      d = {}

      #Loop over enumerate object of n
      for i, v in enumerate(n):

            #Find compliment , k-v , if in dictionary
            if (k - v) in d:

                  #Return current index and value of compliment (its index)
                  return [i, d[(k - v)]]

            #If not add to d
            d[v] = i
```

The time complexity is *O(1)* for a dictionary, yes, but it's like, you know the *rate limiting step* is that for loop which is *O(n)*, so we consider this algorithm to be *O(n)* . Bigger O wins out in Big-O notation.

It's like chemistry you know rate-limiting step, oh my god that takes me back. Did you know that ethylene is an important natural plant hormone and is used in agriculture to force the ripening of fruits?



Just out of respect to the past I paste this image.

Homework Problems
Due: Thursday, Jul 8 12:00 AM

Please turn in via email to liamnisaacs@gmail.com . Please complete all the stuff on `noteshomework.ipynb` and submit your answers via a Jupyter notebook file, .ipynb .

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
```

2. **Recursive Fibonacci**

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

3. **Subarray sums**

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!

Bonus

4. **Anagrams**

Two strings are said to be anagrams of one another if you can turn the first string into the second by rearranging its letters. For example, "table" and "bleat" are anagrams, as are "tear" and "rate". Your task, should you choose to accept it, is to write a function that takes in two strings as input and returns True/False depending on whether or not the two strings are anagrams of each other.