I like parentheticals (a lot).

"Sometimes (when I nest them (my parentheticals) too much (like this (and this))) they get confusing."

Write a function that, given a sentence like the one above, along with the position of an opening parenthesis, finds the corresponding closing parenthesis.

Example: if the example string above is input with the number 10 (position of the first parenthesis), the output should be 79 (position of the last parenthesis).

Gotchas

We can do this in $O(n)$ time.

We can do this in $O(1)$ additional space.

Breakdown

How would you solve this problem by hand with an example input?

Try looping through the string, keeping a count of how many open parentheses we have.

Solution

We simply walk through the string, starting at our input opening parenthesis position. As we iterate, we keep a count of how many additional "(" we find as `open_nested_parens`. When we find a ")" we decrement `open_nested_parens`. If we find a ")" and `open_nested_parens` is 0, we know that ")" closes our initial "(", so we return its position.

Python 3.6 ▾

```
def get_closing_paren(sentence, opening_paren_index):
    open_nested_parens = 0

    for position in range(opening_paren_index + 1, len(sentence)):
        char = sentence[position]

        if char == '(':
            open_nested_parens += 1
        elif char == ')':
            if open_nested_parens == 0:
                return position
            else:
                open_nested_parens -= 1

    raise Exception("No closing parenthesis :(")
```

Complexity

$O(n)$ time, where n is the number of chars in the string. $O(1)$ space.

The for loop with `range` keeps our space cost at $O(1)$. It might be more Pythonic to use:

Python 2.7

```
for char in sentence[position:]:
```

but then our space cost would be $O(n)$, because in the worst case `position` would be 0 and we'd take a slice↴

Array slicing involves taking a subset from an array and **allocating a new array with those elements**.

In Python 3.6 you can create a new list of the elements in `my_list`, from `start_index` to `end_index` (exclusive), like this:

Python 2.7

```
my_list[start_index:end_index]
```

You can also get everything from `start_index` onwards by just omitting `end_index`:

Python 2.7

```
my_list[start_index:]
```

Careful: there's a hidden time and space cost here! It's tempting to think of slicing as just "getting elements," but in reality you are:

1. allocating a new list
2. *copying* the elements from the original list to the new list

This takes $O(n)$ time and $O(n)$ space, where n is the number of elements in the *resulting* list.

That's a bit easier to see when you save the result of the slice to a variable:

Python 2.7

```
tail_of_list = my_list[1:]
```

But a bit harder to see when you don't save the result of the slice to a variable:

Python 2.7

```
return my_list[1:]
# Whoops, I just spent O(n) time and space!
```

Python 2.7

```
for item in my_list[1:]:
    # Whoops, I just spent O(n) time and space!
    pass
```

So keep an eye out. Slice wisely.

of the entire input.

What We Learned

The trick to many "parsing" questions like this is *using a stack* to track which brackets/phrases/etc are "open" as you go.

So next time you get a parsing question, one of your first thoughts should be "use a stack!"