

## I like parentheses (a lot).

"Sometimes (when I nest them (my parentheses) 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 the string, starting at our input opening parenthesis position. As we iterate, we keep a count of how many additional "(" we find as `openNestedParens`. When we find a ")" we decrement `openNestedParens`. If we find a ")" and `openNestedParens` is 0, we know that ")" closes our initial "(", so we return its position.

```
size_t getClosingParen(const string& sentence, size_t openingParenIndex)
{
    size_t openNestedParens = 0;

    for (size_t position = openingParenIndex + 1; position < sentence.length(); ++position) {
        char c = sentence[position];

        if (c == '(') {
            ++openNestedParens;
        }
        else if (c == ')') {
            if (openNestedParens == 0) {
                return position;
            }
            else {
                --openNestedParens;
            }
        }
    }

    throw invalid_argument("No closing parenthesis :(");
}
```

## Complexity

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

## 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!"**

In *this* problem we can realize our stack would only hold '(' characters. So instead of storing each of those characters in a stack, we can store the *number* of items our stack *would be holding*.

That gets us from  $O(n)$  space to  $O(1)$  space.

It's pretty cool when you can replace a whole data structure with a single integer :)

---

Want more coding interview help?

Check out **interviewcake.com** for more advice, guides, and practice questions.