12 Longest valid parentheses substring

Given a string `s` that contains only the '{' and '}' characters, return the length of the longest substring of correctly closed parentheses.

Example: for `s = "{}{{}}{"`, the longest valid parentheses substring is `"{}{{}}"` of length 6.

Clarification questions

Q: How large can the string be?
A: Up to 10,000 characters.

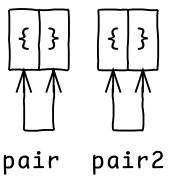
Q: What answer should be returned when the input is the empty string?
A: The result should be 0.

Solution 1: dynamic programming, bottom-up, $O(n)$

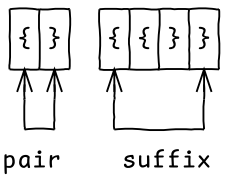
Let's first think about the structure of valid parentheses strings, to see if there are any properties we can use in our advantage. Specifically, we are interested in anything that can help us reduce the problem to smaller subproblems—or the other way around: building valid parentheses strings incrementally starting from smaller ones.

The smallest valid parentheses string is the pair:

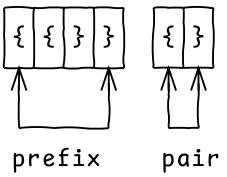
We can build a longer valid pararantheses string by appending another pair:



In general, we can build a longer valid pararantheses string by appending another valid pararantheses string:

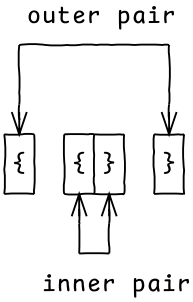


We can do the same by prepending a valid pararantheses string:

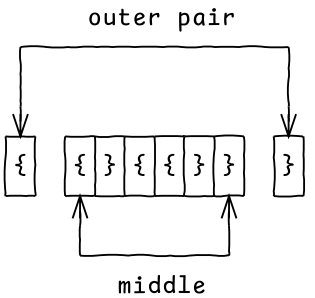


Prepending or appending are equivalent. We may use one or the other depending on the direction we use to build the string (left to right or right to left).

We can also build a longer valid pararantheses string by inserting a pair inside the outer pair:



In general, we can insert any valid parentheses substring inside a pair:

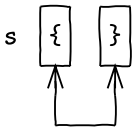


These operations are sufficient to build any valid parentheses substring.

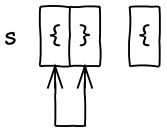
For example, let's see how we can apply them incrementally to `s = "{}{{}}{"`.

We look at each character from `s`, starting from the left:

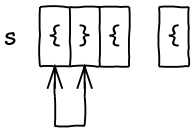
No valid substring can be formed with the first character. Let's add the second character:



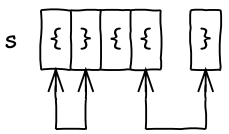
Since the second character is '}', we look for a match to its left to form a pair. We obtain a valid substring of length 2. We continue by adding the third character:



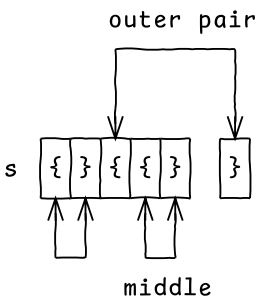
No new valid substring can be formed, since '{' cannot match anything on its left. We add the fourth character:



Once again, no new valid substring can be formed, since '{' cannot match anything on its left. We add the fifth character:



Since the new character is '}', we look for a match to the left to form a pair. We form another valid substring of length 2. We continue by adding the sixth character:

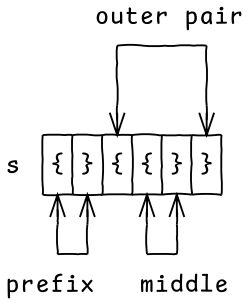


Since the new character is '}', we look for a match to the left to form a pair. There is no '{' behind it, so we cannot form a simple pair.

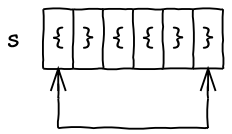
Nevertheless, we can try applying the outer pair strategy. We see that there is a valid substring of length 2 behind it `"{}"`. We try to use it as the middle substring, and embed it into an outer

pair. We succeed, since the character before the middle substring is '{', which allows us to form an outer pair.

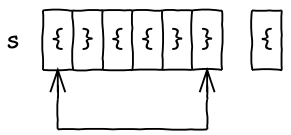
Naively, one would think that we formed a valid substring of length 4. However, before this substring, there is a valid substring prefix:



Thus we can apply the appending rule to form a longer valid substring:



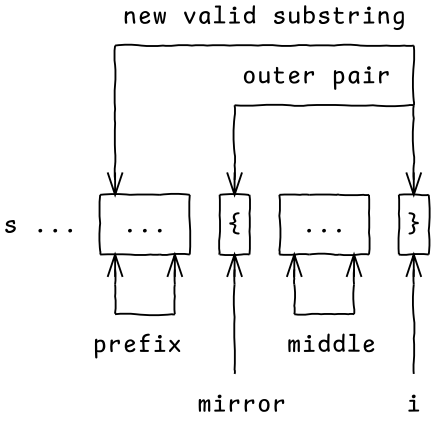
This is the maximal valid substring that can be obtained so far. We can now add the last character:



This character cannot be matched with anything, so the longest valid substring overall is 6.

Based on this example, we can design the algorithm that solves the problem. We process the input string s from left to right, character by character.

To advance from one index to the next, we try to find the following pattern:



This is sufficiently generic to cover more particular cases:

- a simple pair "`{}`" can be formed with an outer pair having an empty `middle`;
- `prefix` may be empty if no valid prefix exists before the outer pair.

If no mirror can be found, or the current character is '{', no valid substring can be formed that ends at the current index.

We keep track of the longest valid substrings found so far in a vector `best_ending_at`: `best_ending_at[i]` stores the length of the longest valid substring ending at index `i`. This allows us to find quickly the `mirror` index: we just need to skip `len(middle) == best_ending_at[i-1]` characters.

This algorithm can be implemented as:

```
def longest_parentheses(s):
    if not s:
        return 0
    # best_ending_at[i] = length of longest valid parentheses
    #                     substring ending at index i.
    best_ending_at = [None] * len(s)
    for i, c in enumerate(s):
        if c == '{' or i == 0:
            # Impossible to form a valid substring ending at i.
            best_ending_at[i] = 0
            continue
        # prefix { middle }   where prefix and middle are valid
        #        ^        ^   and maximal parentheses substrings
        #      mirror     i
        middle_len = best_ending_at[i - 1]
        mirror = i - middle_len - 1
        if mirror < 0 or s[mirror] == '}':
            # Impossible to form a valid substring ending at i:
            # s = ' middle } ... ' or
            # s = ' } middle } ... '
```

```
            best_ending_at[i] = 0
            continue
        if mirror > 0:
            # s = ' prefix { middle } ... '
            prefix_len = best_ending_at[mirror - 1]
        else:
            # s = ' { middle } ... '
            prefix_len = 0
        best_ending_at[i] = prefix_len + middle_len + 2
    # From all valid, maximal substrings, take the longest:
    return max(best_ending_at)
```

The time complexity is $O(n)$, and the space complexity is the same. The space complexity cannot be reduced further, since we need the entire lookup table `best_ending_at` to find the length of the `prefix` substring.

Solution 3: dynamic programming, top-down, $O(n)$

We can also write the above algorithm as top-down. We define a helper function `get_best_ending_at` which computes the length of the longest valid substring ending at a given index. The result of the function needs to be cached to avoid exponential complexity due to redundant calls:

```
from functools import lru_cache

def longest_parentheses(s):
    if not s:
        return 0

    @lru_cache(maxsize=None)
    def get_best_ending_at(i):
        """
        Returns the length of the longest valid parentheses substring
        ending at index i.
        """
        if i == 0 or s[i] == '{':
            # Impossible to form a valid substring ending at i.
            return 0
        # prefix { middle }   where prefix and middle are valid
        #        ^       ^    and maximal parentheses substrings
        #      mirror    i
        middle_len = get_best_ending_at(i - 1)
        mirror = i - middle_len - 1
        if mirror >= 0 and s[mirror] == '{':
            if mirror > 0:
                # s = ' prefix { middle } ... '
```

```
                prefix_len = get_best_ending_at(mirror - 1)
            else:
                # s = ' { middle } ... '
                prefix_len = 0
            return prefix_len + middle_len + 2
        # Impossible to form a valid substring ending at i.
        return 0

    # From all valid, maximal substrings, take the longest:
    return max(get_best_ending_at(i) for i in range(len(s)))
```

The time and space complexities are both linear.

Unit tests

We first write test cases for very simple strings containing just one pair:

```
class Testparentheses(unittest.TestCase):
    def test_1_one_pair(self):
        self.assertEqual(longest_parentheses('{}'), 2)
        self.assertEqual(longest_parentheses('{}' + '{'), 2)
        self.assertEqual(longest_parentheses('}' + '{}'), 2)
        self.assertEqual(longest_parentheses('{}' + '}'), 2)
```

We write a test case to make sure we take into account the appending/prepending expansion:

```
    ...
    def test_2_neighbour_pairs(self):
        self.assertEqual(longest_parentheses('{}{}'), 4)
        self.assertEqual(longest_parentheses('{' + '{}{}'), 4)
        self.assertEqual(longest_parentheses('{}{}' + '{'), 4)
        self.assertEqual(longest_parentheses('{}{}' + '}'), 4)
```

Another test case to cover the nesting expansion:

```
    ...
    def test_3_nested_pairs(self):
        self.assertEqual(longest_parentheses('{{}}'), 4)
        self.assertEqual(longest_parentheses('{' + '{{}}'), 4)
        self.assertEqual(longest_parentheses('{{}}' + '}'), 4)
        self.assertEqual(longest_parentheses('{{}}' + '{'), 4)
```

Some simple tests to check that we can handle multiple substrings:

```
    ...
    def test_4_multiple_substrings(self):
        self.assertEqual(longest_parentheses('{}' + '}' + '{{}}' + '{'), 4)
        self.assertEqual(longest_parentheses('{{}}' + '}' + '{}' + '{'), 4)
```

We also cover the edge case where there are no valid substrings:

```
    ...
    def test_5_no_valid_substring(self):
        self.assertEqual(longest_parentheses(''), 0)
        self.assertEqual(longest_parentheses('}'), 0)
        self.assertEqual(longest_parentheses('{'), 0)
        self.assertEqual(longest_parentheses('}}}{{{'), 0)
```

Finally, a test with large input for performance benchmarking:

```
    ...
    def test_6_perf(self):
        n = 1000
        k = 10
        s = ')'.join(['{' * n + '}' * n] * k)
        self.assertEqual(longest_parentheses(s), n * 2)
        s = ')'.join(['{}' * n] * k)
        self.assertEqual(longest_parentheses(s), n * 2)
```