Daily Coding Problem #46

Problem

This problem was asked by Amazon.

Given a string, find the longest palindromic contiguous substring. If there are more than one with the maximum length, return any one.

For example, the longest palindromic substring of "aabcdcb" is "bcdcb". The longest palindromic substring of "bananas" is "anana".

Solution

We can compute the longest palindromic contiguous substring in O(N^3) using brute force. We can just iterate over each substring of the array and check if it's a palindrome.

```
def is_palindrome(s):
    return s[::-1] == s

def longest_palindrome(s):
    longest = ''
    for i in range(len(s) - 1):
        for j in range(1, len(s)):
            substring = s[i:j]
            if is_palindrome(substring) and len(substring) > len(longest):
                longest = substring
    return longest
```

We can improve the running time of this algorithm by using dynamic programming to store any repeated computations. Let's keep an N by N table `A`, where N is the length of the input string. Then, at each index `A[i][j]` we'll keep whether or not the substring made from `s[i:j]` is a palindrome. We'll make use of the following relationships:

made from `s[i:j]` is a palindrome. We'll make use of the following relationships:

- All strings of length 1 are palindromes
- `s` is a palindrome if `s[1:-1]` is a palindrome and the first and last character of `s` are the same

So, when we fill up our table, we can do the following:

- First, set each item along the diagonal `A[i:i] to true, since strings of length 1 are always palindromes
- Then, check `A[i:i+1]` and set it to true if `A[i] == A[i + 1]` and false otherwise (check all strings of length 2)
- Finally, iterate over the matrix from top to bottom, left to right, only examining the upper diagonal. Note that it doesn't make sense for `j` to be smaller than `i`, so that's why we only need to deal with half of the matrix. Set `A[i][j]` to true only if `A[i + 1][j - 1]` is true and `A[i] == A[j]`.
- Keep track of the longest palindromic substring we've seen so far.

Let's go through an example with the word "bananas".

	b	**a**	**n**	**a**	**n**	**a**	**s**
b	t	f	f	f	f	f	f
a		t	f	t	f	t	f
n			t	f	t	f	f
a				t	f	f	f
n					t	f	f
a						t	f
s							t

We can see from this table that the longest palindromic substring here is "ananas", since `A[1:5]` is the longest substring that's true in the table.

```
def longest_palindrome(s):
    if not s:
        return ''

    longest = s[0]
    A = [[None for _ in range(len(s))] for _ in range(len(s))]

    # Set all substrings of length 1 to be true
```

```
    for i in range(len(s)):
        A[i][i] = True

    # Try all substrings of length 2
    for i in range(len(s) - 1):
        A[i][i + 1] = s[i] == s[i + 1]

    i, k = 0, 3
    while k <= len(s):
        while i < (len(s) - k + 1) :

            j = i + k - 1
            A[i][j] = A[i + 1][j - 1] and s[i] == s[j]
            # Update longest if necessary
            if A[i][j] and len(s[i:j + 1]) > len(longest):
                longest = s[i:j + 1]
            i += 1
        k += 1
        i = 0
    return longest
```

This runs in O(N^2) time and space.
