PROBLEM

(Erickson, 6.42b + LeetCode#32): A string W of parentheses and brackets is balanced if:

- W is the empty string
- W = (X), where X is balanced
- W = [Y], where Y is balanced
- W = XY, where X and Y are balanced

Find an algorithm that computes the length of the longest balanced substring of W.

ANALYSIS

It is not exactly clear how to construct the longest balanced substring of W from the longest balanced subsequence of a substring of W. Indeed, consider:

- X = ([])[(), which has longest balanced subsequence ([])
- Y = [][]), which has longest balanced subsequence [][]
- XY = ([])[()[][]), which has longest balanced subsequence ()[][], which is not constructible from the longest balanced subsequences of its constituent parts

The problem is that there is no guarantee that the longest balanced subsequence of a particular substring can be extended by appending to the string.

To guarantee extensibility, we don't find *the* longest balanced subsequence of a substring, but instead the longest balanced substring that ends at the end of the substring. Later on in a helper/wrapper function, we take the maximum length over all longest balanced subsequences ending at each index to get the length of the overall longest balanced subsequence.

Some Notation and terminology:

- Let W{0..i} be the substring from index 0 to i (inclusive) of W.
- Let W{i} be the character at index i of W.
- I will refer to "()" as parentheses, "[]" as square brackets, and both of them by the general term brackets. I will call '(' and '[' opening brackets, and ')' and ']' as closing brackets.

Let the function f{i} return the longest balanced subsequence which ends at index i, inclusive. A naive way to calculate f{i}, is to start

at f{i} and scan leftwards with a stack, pushing closing brackets, popping opening brackets that match the most recent closing bracket, stopping if you fail a pop, and backtracking if you hit the beginning with closing brackets still on the stack. Each scan is O(n), so the solution as a whole is O(n^2). However, we can also calculate f{i}, recursively:

1. f{0} = “”
    a. W{0..0} has only 1 character, which cannot ever be balanced
2. f{1} =
    a. W{0..1} if W{0..1} is balanced (i.e. “()” or “[]”),
    b. “” otherwise
3. f{i} =
    a. f{i - 2} prepended to W{i-1..i} if W{i-1..i} is balanced
        i. There is no point in not selecting the maximum balanced substring ending at i - 2, meaning this recursive call preserves correctness
    b. The longest balanced substring immediately before [f{i-1}] prepended to [f{i - 1}]
        i. calculate the prefix with a recursive call to f{i - len{f{i - 1}} - 2}
        ii. There is no point in not using the longest prefix you can. (again preservation of correctness)
    c. The longest balanced substring immediately before (f{i-1}) prepended to (f{i - 1})
        i. Same analysis as case b
    d. “” otherwise, including (non-exhaustively)
        i. Whenever W{i} is an opening bracket
        ii. When W{i} is a closing bracket, when W{i - 1} is also a closing bracket, and when f{i - 1} is not preceded by the corresponding opening bracket to W{i}

For actual code, instead of returning substrings and doing expensive string copy operations, we return the length of the longest balanced substring (this works, because the caller of f already knows the index of the end of the substring).

The worst case for basic recursion is when we never get case 3a (branching factor 1), and instead consistently get cases 3b and 3c (branching factor 2), which happens for e.g. ((((())))). In this case

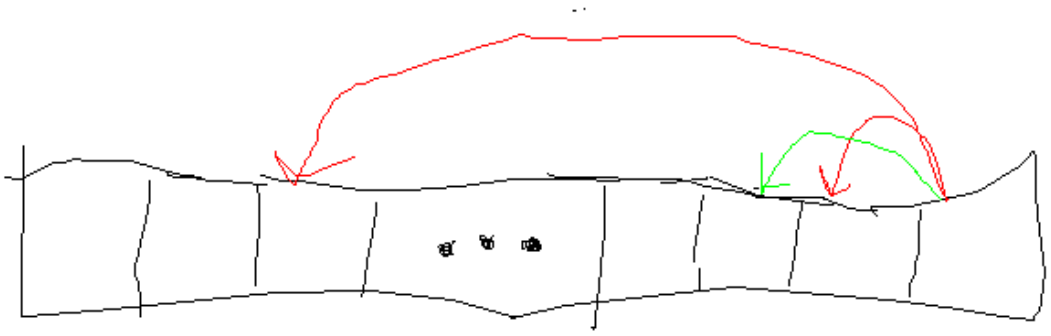
W{i} is at least 2 + W{i - 1}, meaning at most we recurse to a depth of n/2, this allows us to bound the runtime at O(2^(n/2)).

So far, it seems this recursive solution takes a lot more time. Indeed even brute force, i.e. enumerating all substrings and checking each for balance is faster, with a theoretical runtime of O(n^3).

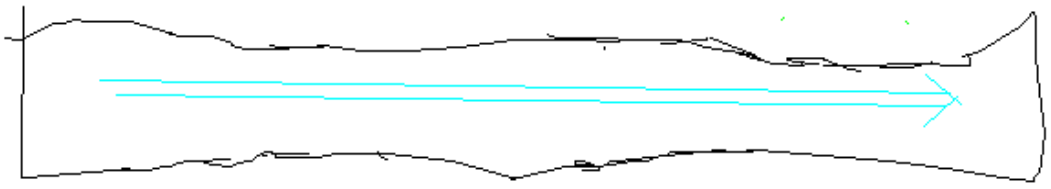
However, with dynamic programming, we can solve this problem in linear time. Since there are only n possible ending indices, we only need to calculate each f{i}, 0 <= i < n, at most once. Calculating the maximum of the lengths of all f{i}'s is also O(n).

DYNAMIC PROGRAMMING TABLE

Here is a table for all possible dependencies of a given call to f. The green arrow is for case 3a, and the red arrows are for cases 3b/3c.



This means we can use the following evaluation order:



I know that this is only one dimensional, but I think this algorithm is plenty complicated.

CODE

We need to write a function that takes in a string of parentheses and square brackets and returns the length of the longest balanced substring. I will write four functions, each with the same signature:

- unsigned brute_force(string s);
- unsigned scan_left(string s);
- unsigned basic_recursion(string s);
- unsigned dynamic_programming(string s);

LINK

RESULTS

Runtime of Longest Balanced Substring on **Random** String of n Brackets
(all times are in seconds)

	500	1000	2000	20000	2e5	2e6
brute	0.05361	0.2186	0.9256	Too Long	Too Long	Too Long
scan	1.957e-4	4.016e-4	7.448e-4	7.497e-3	7.566e-2	0.8225
basic	5.691e-5	1.122e-4	2.114e-4	2.152e-3	2.116e-2	0.2220
dynamic	4.034e-5	7.780e-5	1.415e-4	1.291e-3	1.314e-2	0.1434

Runtime of Longest Balanced Substring on **(((...)))** of length n
(all times are in seconds)

	500	1000	2000	20000	2e5	2e6
brute	0.6334	4.609	36.62	Too Long	Too Long	Too Long
scan	2.627e-3	1.064e-2	3.971e-2	3.912	Too Long	Too Long
basic	1.769e-3	6.835e-3	2.875e-2	2.793	Too Long	Too Long
dynamic	2.626e-5	5.214e-5	9.614e-5	9.906e-4	9.395e-3	0.08864

RUNTIME ANALYSIS

On random inputs, brute force is approximately quadratic time, and all non brute force solutions are approximately linear time. This is because long strings of balanced brackets become exponentially less likely to show up the longer they are. Thus, the worst case is extremely unlikely to be encountered.

Scan_left and basic_recursion take advantage of this by having mechanisms to return early the moment a candidate substring is no longer balanced. Since the expected length of a balanced substring is around log(n), we see that scan_left runs in around O(n log(n)) time. Even brute force takes advantage of this, because it needs to check if

substrings are balanced, and it can also return early on non-balanced substrings.

This also means the difference between basic_recursion and dyamic_programming is smaller, because the recursion depth is bounded. Additionally, since most of the input is not balanced, most recursive calls end up immediately returning. In summary:

- brute_force
    - Expected O(n^3), actual O(n^2log(n))
- scan_left
    - Expected O(n^2), actual O(nlog(n))
- basic_recursion
    - Expected O(2^(n/2)), actual O(n)
- dynamic_programming
    - Expected O(n), actual O(n)

On the input (((...))), the dynamic programming solution is still linear time, and in fact faster (my guess is this is due to more predictable inputs leading to better branch prediction). Brute force is also much slower, and is O(n^3), because there are longer balanced substrings for which it can take up to O(n) time to verify the balance of. Scan_left is O(n^2) for the same reasons.

basic_recursion, however, still does not take exponential time, because there's never a point where you have to concatenate two balanced substrings, meaning lot's of branching gets nipped in the bud. Instead, most of the repeated work happens near the center of (((...))), where basic_recursion essentially peels away the outer parentheses two at a time (this also explains why it is faster than scan_left, which goes one at a time), recursing into itself only once each time. Thus, it is O(n^2).

- brute_force
    - Expected O(n^3), actual O(n^3)
- scan_left
    - Expected O(n^2), actual O(n^2)
- basic_recursion
    - Expected O(2^(n/2)), actual O(n^2)
- dynamic_programming

- Expected O(n), actual O(n)