Solved Problem - Balanced Parentheses Sequence

In this lesson, we'll discuss a popular stack problem.

We'll cover the following

Problem statement

Given a string sequence consisting of length N only of opening or closing parentheses `( ) [ ] { }`, determine if the sequence is a correct bracket sequence.

For example: If $s1$ and $s2$ are correct bracket sequences, so are the following:

- $(s1)s2$
- $(s1s2)$
- $[(\{s1\}s2)]$

But these are not:

- $)s1s2)$
- $[s1)s2$

Sample

Input:

`[()]{}{[()()]()}`

Output:

`Yes`

Constraints:

$1 <= N <= 10^6$

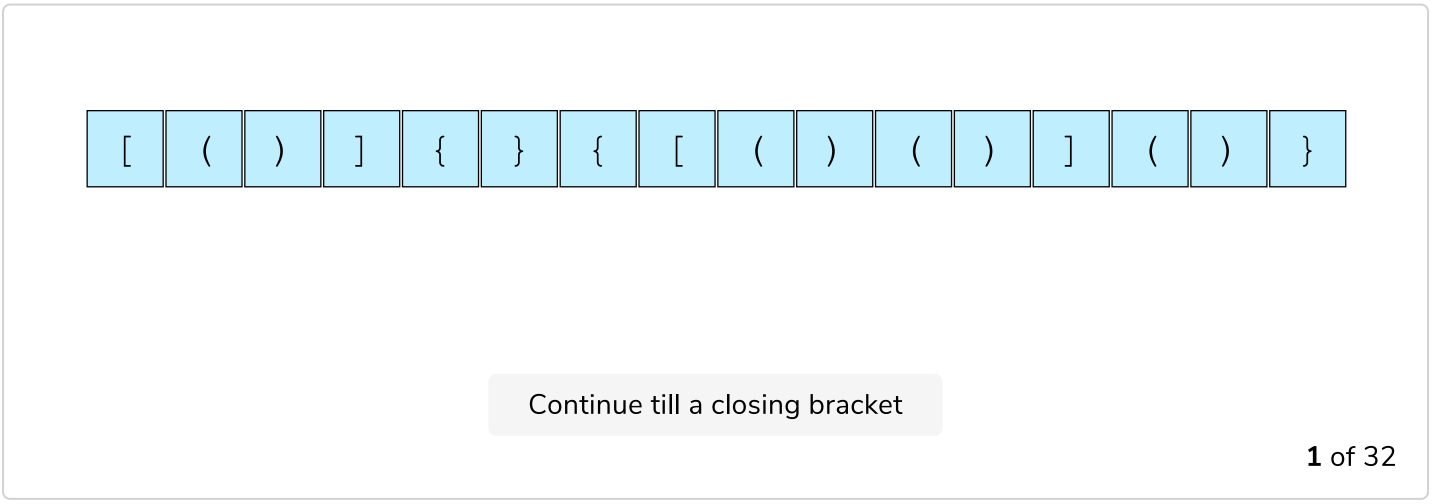
Brute force

Obviously, if it’s a balanced sequence, every opening bracket, `( [ {`, is matched to exactly one closing bracket, `) ] }`, of the same type.

We can parse to look for every closing bracket and try to find its corresponding opening bracket in the string parsed so far.

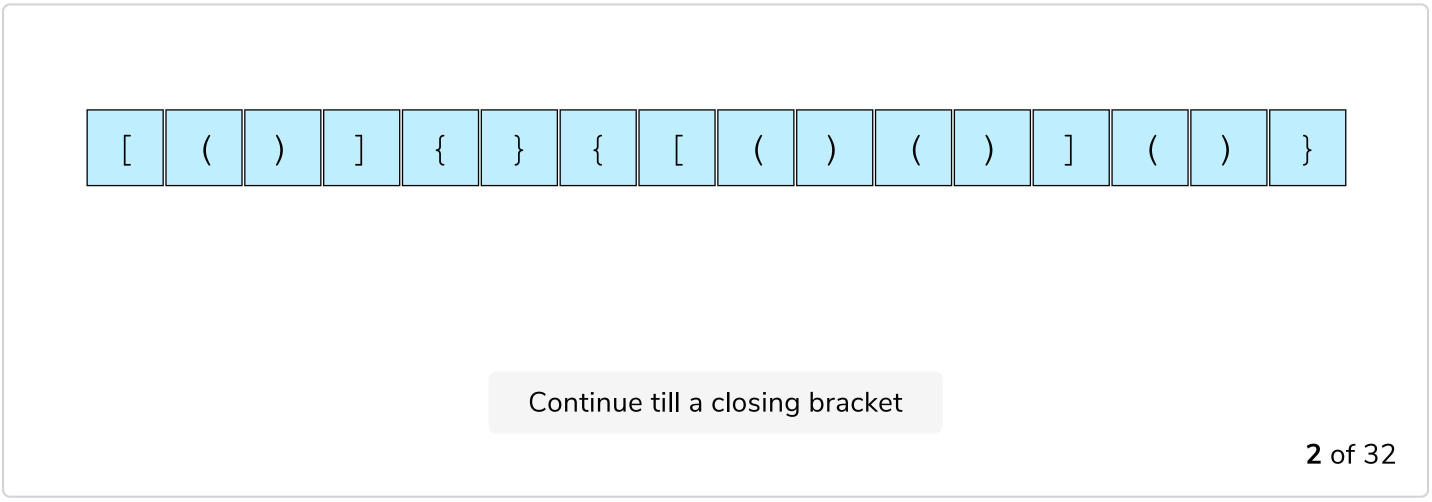
It would be the first opening bracket to its left that didn’t match up with any bracket yet. So, you keep track of matched brackets as well.

If we are able to match all the brackets, then it’s a balanced sequence, otherwise it’s not. See the illustration below for a better understanding:



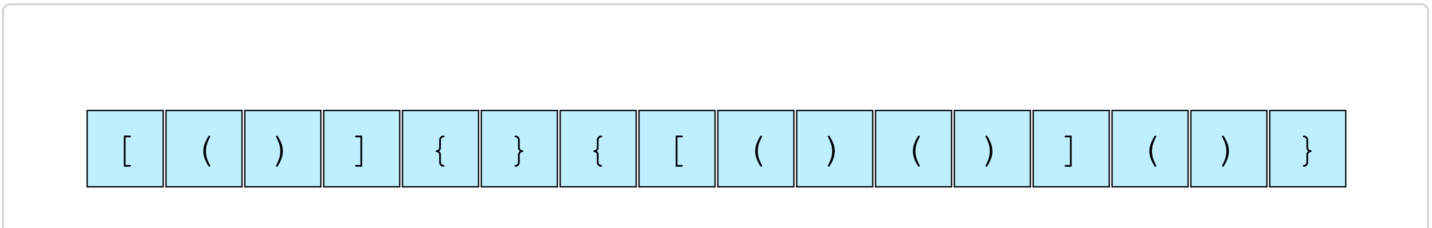
Continue till a closing bracket

1 of 32



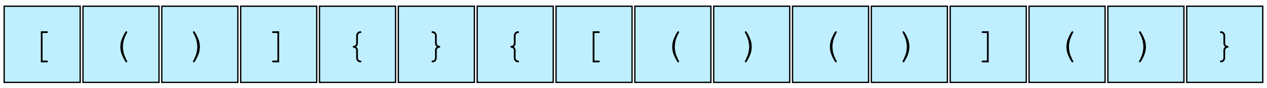
Continue till a closing bracket

2 of 32



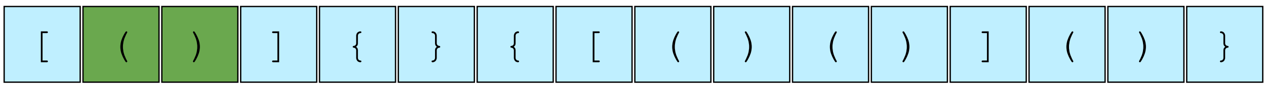
Now for this closing bracket. Find the first opening bracket of the same type

3 of 32



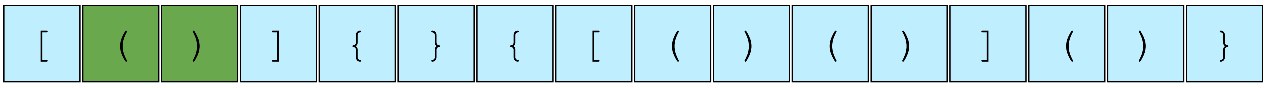
This satisfies the condition. So match these 2.

4 of 32



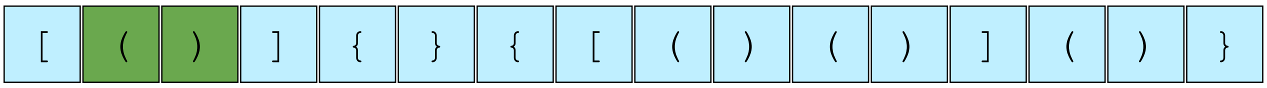
Not blue = matched. Continue down the string

5 of 32



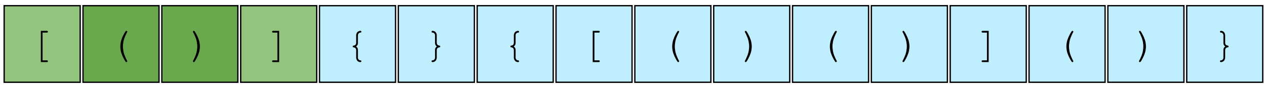
Search for corresponding 'unmatched' opening bracket

6 of 32



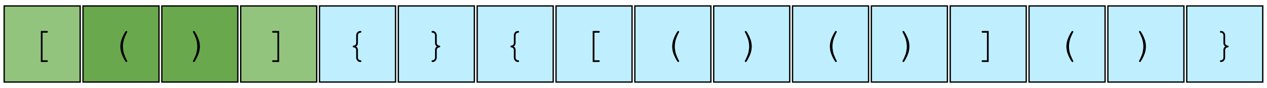
Green brackets don't qualify. Match with the found bracket

7 of 32



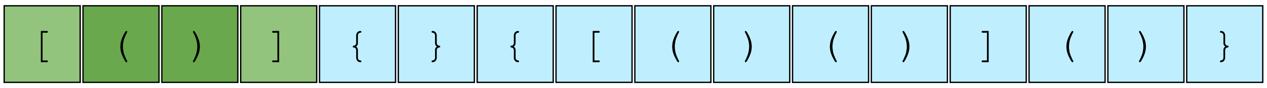
Green brackets don't qualify. Match with the found bracket

8 of 32



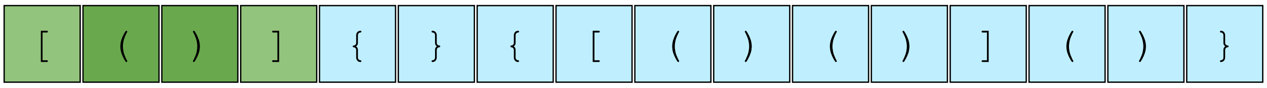
Go to next closing bracket

9 of 32

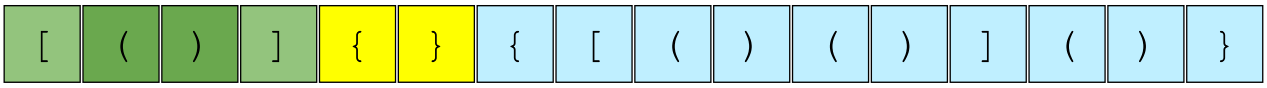


Repeat

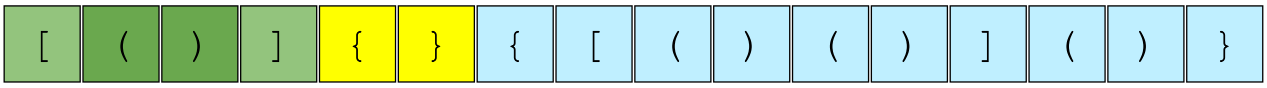
10 of 32



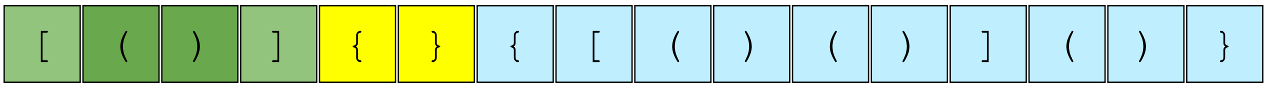
11 of 32



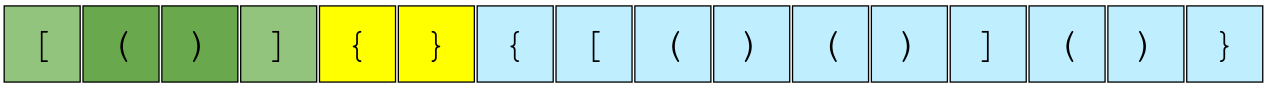
12 of 32



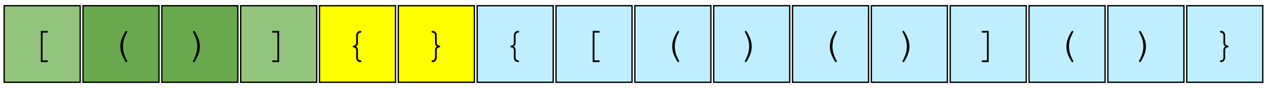
13 of 32



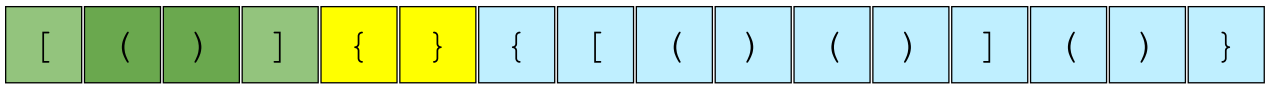
14 of 32

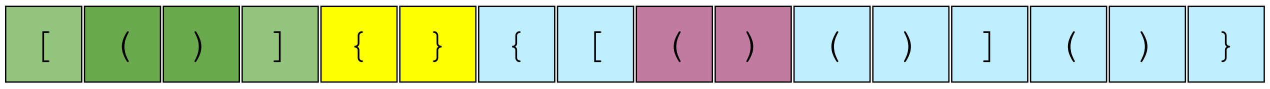


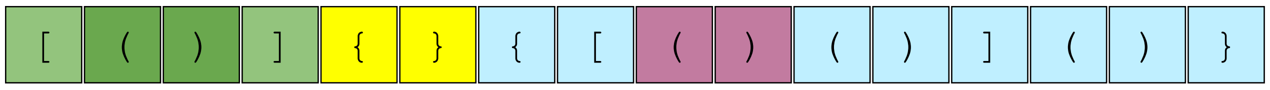
15 of 32

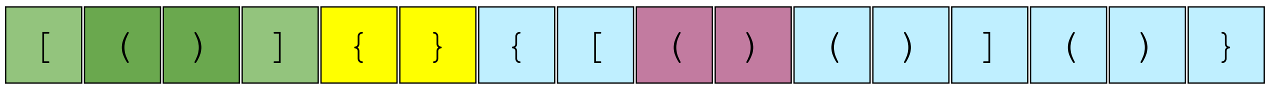


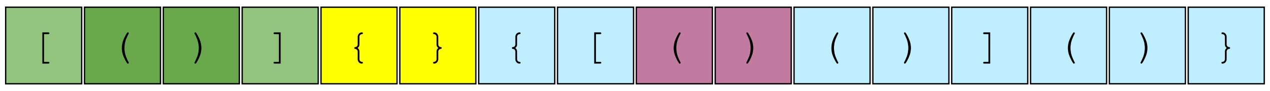
16 of 32

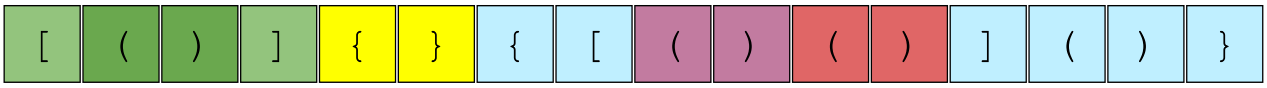




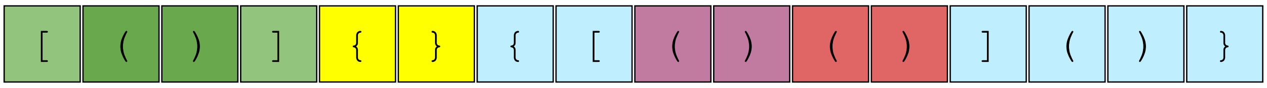




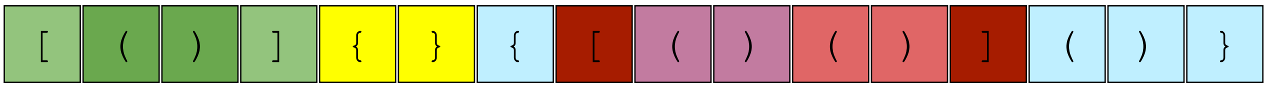




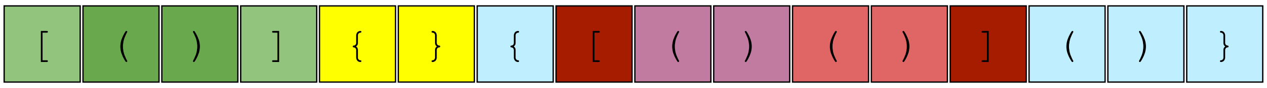
22 of 32



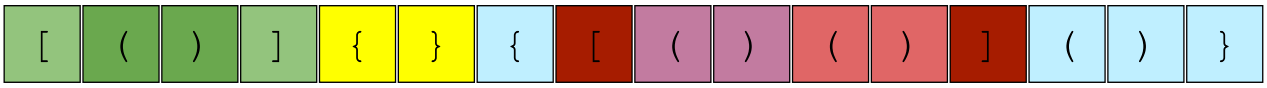
23 of 32



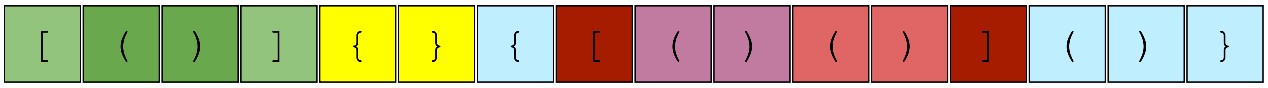
24 of 32



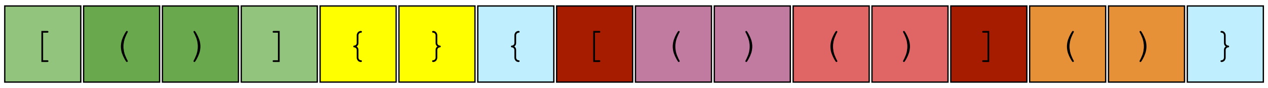
25 of 32



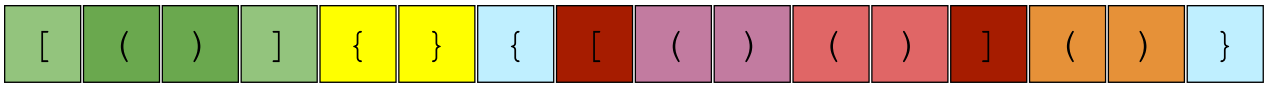
26 of 32



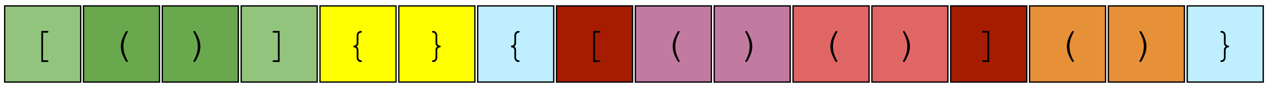
27 of 32



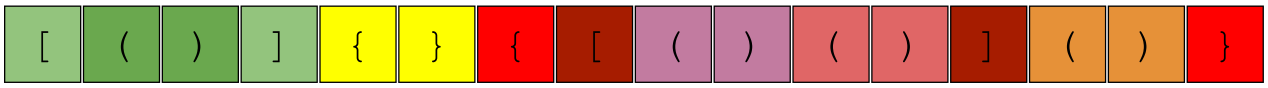
28 of 32



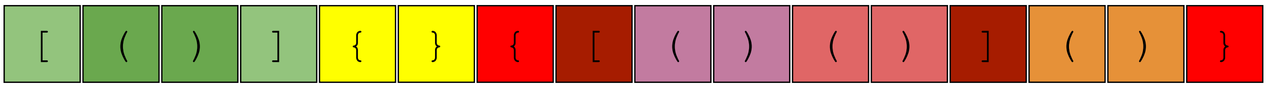
29 of 32



30 of 32



31 of 32



32 of 32

The code for the above solution is pretty straightforward. We keep a Boolean array to check if we have already matched that bracket or not.

```
#include <iostream>
#include <vector>
using namespace std;

char opening(char c) { // get opening bracket of same type
  if (c == ')') return '(';
  if (c == ']') return '[';
  if (c == '}') return '{';
}

bool is_opening(char c){
  return c == '(' || c == '[' || c == '{';
}

bool is_closing(char c) {
  return !is_opening(c);
}

string is_balanced(string s) {
  int N = s.size();
  vector<bool> matched(N, false);

  for (int i = 0; i < N ;i ++) {
    if (is_closing(s[i])) {
      int found_pos = -1; // position of mathcing opening bracket
      for (int j = i - 1; j >= 0; j--){
        if ( !matched[j] && is_opening(s[j]) && s[j] == opening(s[i]) ) {
          found_pos = j;
          break;
        }
      }
      if (found_pos == -1) // didn't find matching bracket
        return "No\n";

      matched[i] = matched[found_pos] = true;
    }
  }
  return "Yes\n"; //Nothing went wrong, all brackets matched
}

int main() {
  cout << is_balanced("[()]{}{[()()]()}");
  cout << is_balanced("]]");
  cout << is_balanced("[(){[]({})}]");
  cout << is_balanced("[(){[)({})}]");
  return 0;
}
```

The time complexity of the above solution is $O(N^2)$, because for each closing bracket, we might end up iterating the entire string again.

The solution is good enough for N up to 10^3.

Optimization

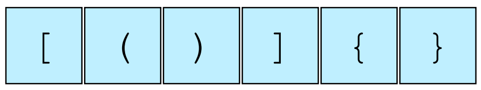
Instead of keeping which bracket has matched already in a Boolean array, what if we could just delete the character from the string? That way, we won't have to iterate over all the characters to the left.

But deleting a character in a string is an $O(N)$ operation. We can optimize this step using a stack.

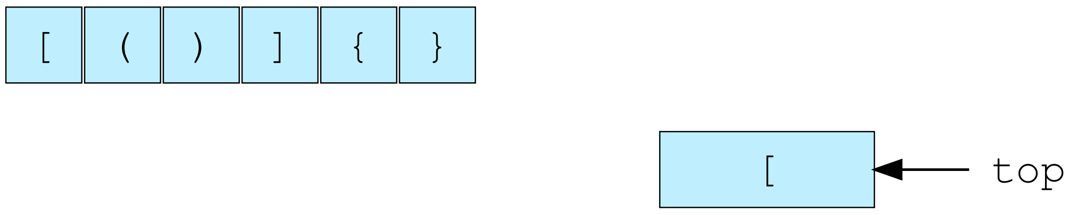
Let's start with an empty stack. Traverse the string from left to right.

- For every opening bracket, push this on to the stack.
- For every closing bracket, match this with the top of the stack as the top of the stack is the first unmatched opening bracket to the left of this closing bracket. Pop the top bracket from the stack. If they don't match, then it's not a balanced parentheses sequence.
- At the end, the stack should be empty.

See the illustration below:

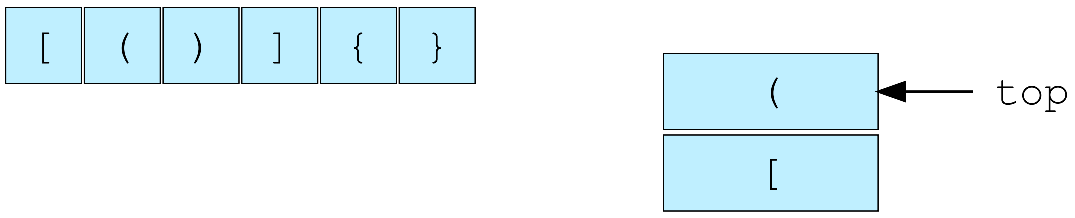


1 of 10



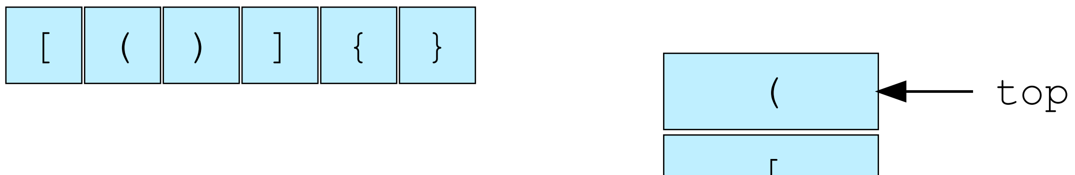
Opening bracket -> Push to stack

2 of 10



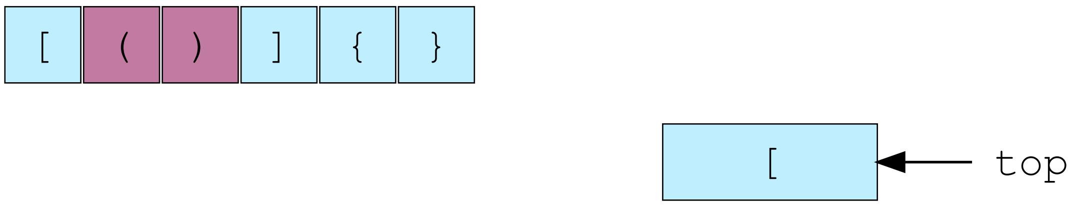
Opening bracket -> Push to stack

3 of 10



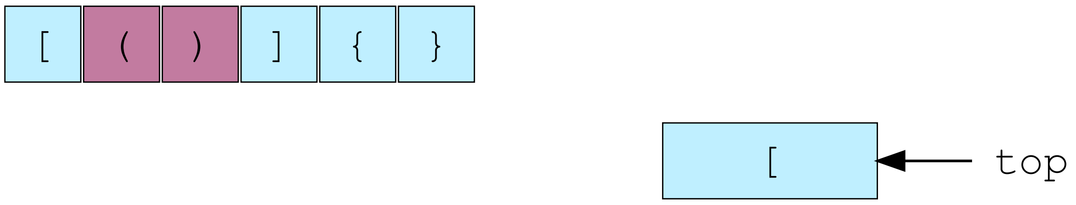
Try to match with 'top', if it matched, pop

4 of 10



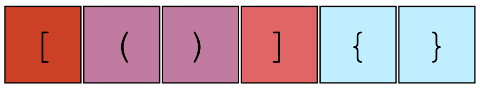
Essentially doing marked[i] = false

5 of 10



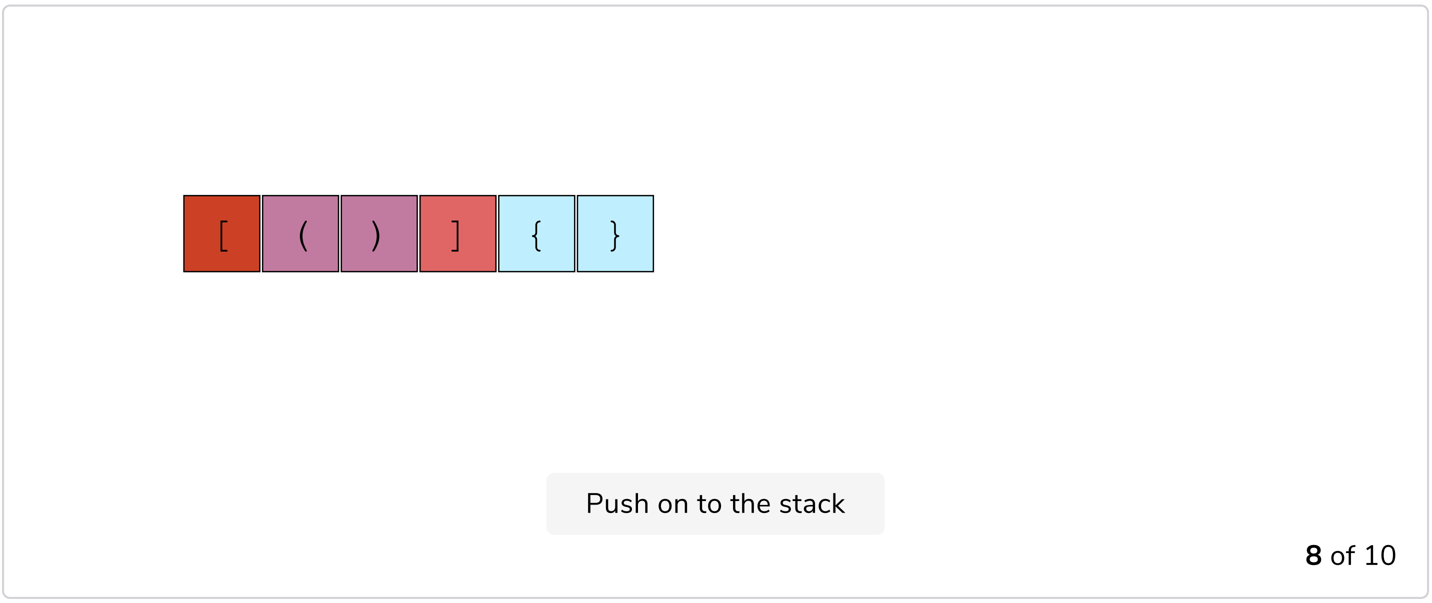
Match with top

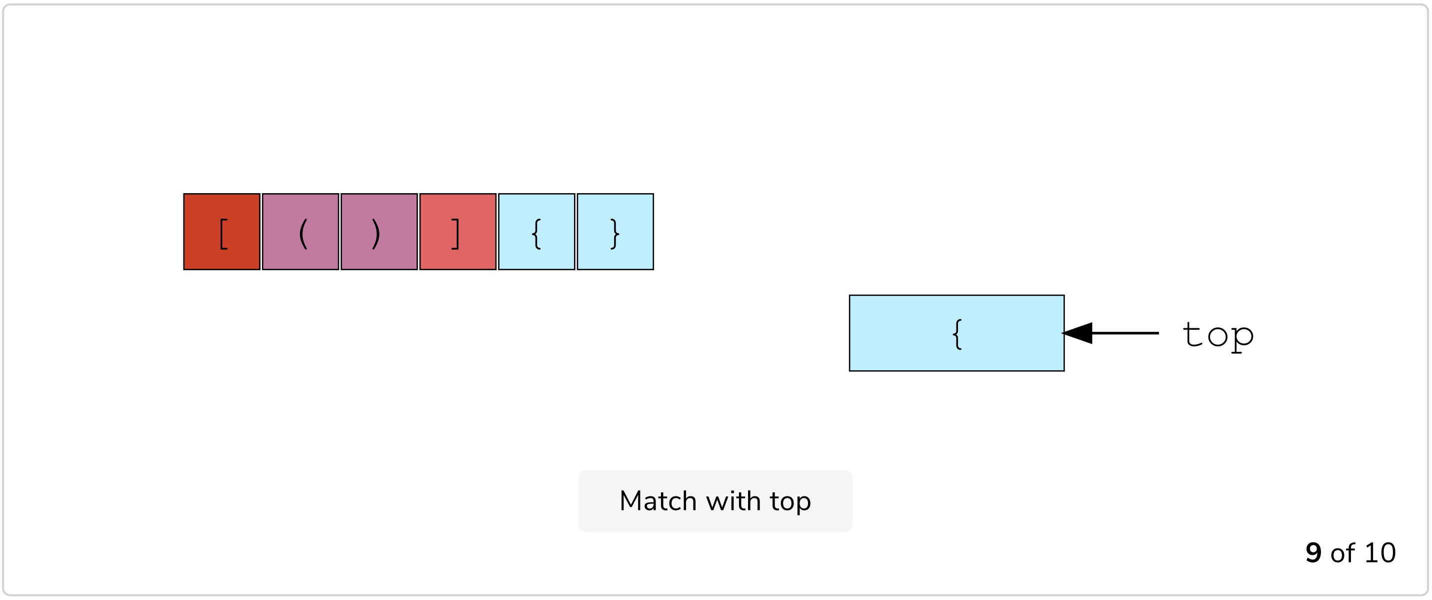
6 of 10

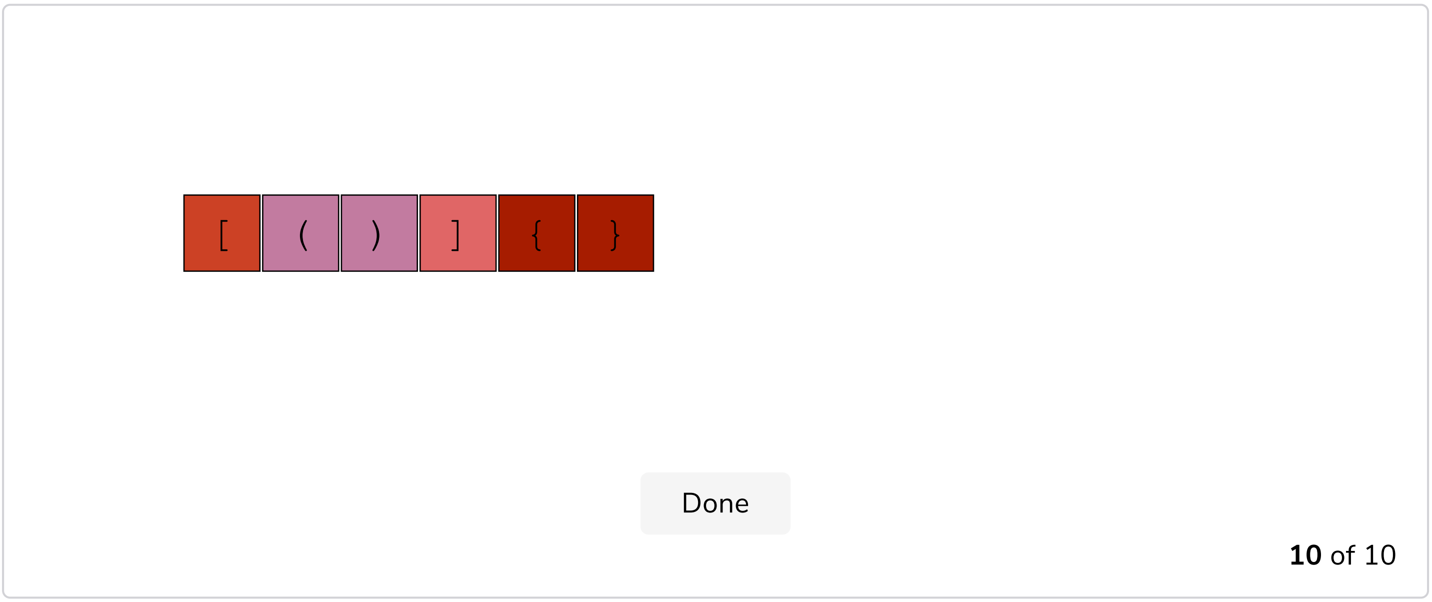


Match with top

7 of 10







Essentially, we are optimizing the second step in the brute force solution to find the matching bracket in $O(1)$ instead of $O(N)$.

Since each character is iterated over two times at most , once while pushing and once while popping, the complexity is $O(N)$ and works for the given constraints.

The code for the optimized solution is below.

```
#include <iostream>
#include <stack>
using namespace std;

char opening(char c) { // get opening bracket of same type
  if (c == ')') return '(';
  if (c == ']') return '[';
  if (c == '}') return '{';
}

bool is_opening(char c){
  return c == '(' || c == '[' || c == '{';
}

bool is_closing(char c) {
  return !is_opening(c);
}

string is_balanced(string s) {
  int N = s.size();
  stack<char> S;

  for (int i = 0; i < N ;i ++) {
    if (is_opening(s[i]))
      S.push(s[i]);
    else {
      if (S.empty())
        return "No\n"; // Looking for opening brakcet but stack it empty => Not balanced
      char c = S.top();

      if (c != opening(s[i]))  // top is not mathcing!
        return "No\n";

      S.pop(); // top matched, remove from stack
    }
  }

  return "Yes\n"; //Nothing went wrong, all brackets matched
}

int main() {
  cout<<is_balanced("[()]{}{[()()]()}");
  cout<<is_balanced("]]");
  cout<<is_balanced("[(){[]({})}]");
  cout<<is_balanced("[(){[)({})}]");
  return 0;
}
```

That’s it on stacks, for now, We’ll be using stacks in a wide range of topics in later levels. In the next chapter, we’ll discuss a similar data structure - *queue*.