Thì thực ra bài này thuộc thể loại rất kinh điển ứng dụng của cấu trúc dữ liệu ngăn xếp (stack). Thế nên nếu bạn nào chưa biết về cấu trúc dữ liệu này thì các bạn phải Google search tìm hiểu về nó trước rồi thì mới xem tiếp được nha 😋. Còn nếu bạn nào đã biết rồi nhưng chưa biết hướng vận dụng nó để xử lý cho bài này thì mình sẽ nói ra tư tưởng giải thuật để các bạn được hiểu:

Đầu tiên khởi tạo ngăn xếp có kiểu dữ liệu ký tự tức là nó sẽ chứa dữ liệu ở dạng ký tự. Và ngăn xếp này sẽ chỉ chứa các ký tự mở ngoặc như ( [ { . Thì ta cứ duyệt chuỗi gặp ký tự nào là các ký tự mở ngoặc thì đưa hết nó vào ngăn xếp. Còn nếu gặp ký tự không phải là ký tự mở ngoặc (tức là ký tự đóng ngoặc) thì lúc này kiểm tra xem ngăn xếp có đang tồn tại phần tử không? Nếu không thì kết luận No ngay và dừng chương trình vì nó đang phạm lỗi không có mở ngoặc mà lại đóng ngoặc. Còn nếu có phần tử thì lấy ra phần tử ở đỉnh ngăn xếp (top) tức là phần tử gần nhất được đưa vào và xét xem nó có cùng loại ngoặc với ký tự hiện tại không? Ví dụ nếu ký tự hiện tại đang là ) thì đỉnh stack phải là ( mới đúng, nếu khác ( thì ngay lập tức kết luận No ngay và dừng chương trình. Tương tự như vậy với 2 loại còn lại, nếu ký tự hiện tại đang là } thì đỉnh stack phải là {, nếu ký tự hiện tại đang là ] thì đỉnh stack phải là [ . Nếu ngay chỗ nào phát hiện không đúng thì ngay lập tức kết luận No và dừng chương trình. Nếu mọi thứ vẫn đúng thì cứ chạy xét tiếp các ký tự của chuỗi cho đến hết vòng lặp. Lúc này phải nhớ kiểm tra nếu đến khi hết vòng lặp mà ngăn xếp vẫn còn phần tử bên trong thì nghĩa là lúc này đang bị dư các ký tự mở ngoặc, ta kết luận No. Còn nếu ngăn xếp rỗng tức là lúc này mọi cặp ngoặc đều đã triệt tiêu nhau hết thì nghĩa là hợp lệ, ta kết luận Yes.

Source code để các bạn tham khảo thấy rõ những gì mình phân tích ở trên:

[#include](https://www.facebook.com/hashtag/include?__eep__=6&__cft__%5b0%5d=AZXRRKWKVnzz1XhQ8ZCBnN5SmL9nebuzpLQvj2-ZExSrNPwf8YphmdAKAow9IHAWw2xr8lq9Hl-HT12fKi11K4iiL1Z9LiUwkVwt-_GPl_Ss2C8c4RcdsKmRyu3Amjly9RQcvcdABQ88QwXF_8k6Byur&__tn__=*NK-R) <bits/stdc++.h>

using namespace std;

int main(){

string s;

cin >> s;

stack<char> st;

for(int i = 0; i < (int)s.length(); ++i){

if(s[i] == '(' || s[i] == '{' || s[i] == '['){

st.push(s[i]);

}

else{

if(st.empty()){

cout << "No";

return 0;

}

char c = st.top();

st.pop();

if((s[i] == ')' && c != '(') || (s[i] == '}' && c != '{') || (s[i] == ']' && c != '[')){

cout << "No";

return 0;

}

}

}

if(!st.empty()){

cout << "No";

}

else{

cout << "Yes";

}

return 0;

}

Đánh giá độ phức tạp của cách làm này:

+ Độ phức tạp không gian (Space Complexity): O(N) với N là độ dài chuỗi, là tối đa các phần tử mà ngăn xếp có thể phải chứa.

+ Độ phức tạp thời gian (Time Complexity): O(N) với N là độ dài chuỗi

Thì ta thấy độ dài chuỗi tối đa 10^5 nên với độ phức tạp thời gian là O(N) tức là 10^5 thì nó thoả nhỏ hơn ngưỡng giới hạn 1 giây mà khúc đầu mình có nói là (3 đến 5)\*10^7 hay cao lắm là 10^8 nên yên tâm sẽ không sợ bị TLE nhé.

Nói thêm cho các bạn nào nếu không biết: Là các thao tác push, pop, top, empty của stack (ngăn xếp) mà mình dùng trong code ở trên, mỗi thao tác đó chỉ tốn độ phức tạp thời gian O(1) thôi do bản chất của nó được xây dựng theo cơ chế danh sách liên kết con trỏ nên việc bẻ mối liên kết giữa tụi nó để thêm vào hay xoá ra thì ta chỉ mất O(1) tức là 1 tác vụ ngay lập tức và do chỉ làm việc với phần tử đầu stack nên việc truy xuất đến phần tử đầu stack cũng chỉ O(1). Bạn nào đã học môn Cấu Trúc Dữ Liệu & Giải Thuật sẽ hiểu rõ cái này. Chính vì thế nên ta chỉ tốn O(N) để duyệt qua chuỗi thôi vì thế mới kết luận độ phức tạp thời gian là O(N) với N là độ dài chuỗi.

